ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
Campus di Cesena
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

DEPLOYMENT E SCALING AUTOMATICI DI UN CLUSTER KUBERNETES LOW COST SU ARCHITETTURA ARM

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Lorenzo Mondani

Quarta sessione
Anno accademico 2019/2020

Ringraziamenti

A conclusione del mio percorso universitario desidero rinnovare la mia gratitudine agli insegnanti del corso triennale e formulare sentiti ringraziamenti agli insegnanti del corso di laurea magistrale, che hanno saputo offrire stimoli continui ed efficaci al mio desiderio di conoscere ed approfondire.

Un ringraziamento particolare va al mio relatore Prof. Vittorio Ghini, che durante il periodo di stesura di questo elaborato mi ha supportato, consigliato ed incoraggiato con competenza e disponibilità.

Infine, un ringraziamento speciale va alla mia famiglia che con il suo affetto e la sua pazienza mi ha dato il sostegno necessario perché potessi arricchire la mia formazione, dedicandomi allo studio e alle mie amate ricerche.

Un grazie di cuore anche agli amici, che non mi hanno mai dimenticato e mi hanno aiutato a sdrammatizzare le tensioni.

Lorenzo Mondani

Indice

Introduzione **6**

1 Cluster computing, infrastrutture cloud e microservizi **11**
1.1 I limiti dei processori single-core 11
1.2 Architetture parallele 13
1.2.1 Classificazione dei sistemi MIMD 13
1.2.2 Limiti delle architetture parallele 15
1.3 Scaling 15
1.3.1 Scale up 15
1.3.2 Scale out 16
1.4 Cluster computing 16
1.4.1 Le cause della diffusione 17
1.4.2 Cluster management software 17
1.5 Infrastrutture cloud 19
1.5.1 Risorse on demand 19
1.5.2 Modelli di servizio 21
1.5.3 Modelli di deployment 23
1.6 Architettura a microservizi 24
1.6.1 I problemi del software monolitico 25
1.6.2 Caratteristiche dell'architettura a microservizi 26
1.6.3 Deployment dei microservizi 28

2 Hardware in ambito server e cloud **30**
2.1 Instruction set architectures 30
2.1.1 CISC (Complex Instruction Set Computer) 31
2.1.2 RISC (Reduced Instruction Set Computer) 31
2.2 x86-64 e Arm 32
2.2.1 Diffusione di x86-64 32
2.2.2 Diffusione di Arm 33
2.3 Compatibilità tra hardware Arm e sistemi operativi 35
2.3.1 Storia 35

2.3.2 Arm SystemReady . . . 36
2.4 UEFI . . . 38
2.4.1 Storia . . . 38
2.4.2 Definizione . . . 38
2.4.3 Funzionalità . . . 39
2.4.4 Compatibilità . . . 41
2.5 ACPI . . . 41
2.5.1 Storia . . . 42
2.5.2 Definizione . . . 42
2.5.3 Funzionalità . . . 43
2.5.4 Compatibilità . . . 46

3 Bare-metal cloud 47
3.1 Definizione . . . 47
3.2 Principali alternative . . . 49
3.3 MAAS (Metal As A Service) . . . 51
3.3.1 Principali tecnologie . . . 51
3.3.2 Interazione con l'utente . . . 52
3.3.3 Controller . . . 53
3.3.4 Ciclo di vita delle macchine . . . 54
3.3.5 Modalità di gestione delle macchine . . . 57
3.3.6 Sistemi operativi . . . 59
3.3.7 Power Management . . . 60
3.3.8 Rete . . . 61
3.3.9 Pod KVM . . . 63

4 Infrastructure as Code 64
4.1 Definizione . . . 65
4.2 Principali alternative . . . 68
4.3 Juju . . . 71
4.3.1 Cloud provider e controller . . . 71
4.3.2 Interazione con l'utente . . . 72
4.3.3 Deployment di applicazioni . . . 73
4.4 Terraform . . . 77
4.4.1 Comandi e Workflow . . . 78
4.4.2 Providers . . . 79
4.4.3 HashiCorp Configuration Language . . . 79
4.4.4 Stato . . . 81

5 Container orchestration **82**
5.1 Definizione . . . 82
5.2 Principali alternative . . . 85
5.3 Kubernetes . . . 86
5.3.1 Architettura . . . 87
5.3.2 Principali oggetti Kubernetes . . . 89
5.3.3 Servizi e load balancing . . . 93
5.3.4 Volumi . . . 95
5.3.5 Autoscaling . . . 99
5.3.6 Controllo degli accessi . . . 100

6 Definizione degli obiettivi **102**
6.1 Realizzazione fisica del cluster . . . 102
6.2 Realizzazione di un bare-metal cloud . . . 103
6.3 Deployment automatico del cluster Kubernetes . . . 104
6.4 Scaling automatico del cluster Kubernetes . . . 104
6.5 Load balancer e volumi persistenti su Kubernetes . . . 105
6.6 Valutare la compatibilità del software con l'architettura Arm . . . 105

7 Analisi del problema e progettazione del cluster **107**
7.1 Hardware . . . 108
7.2 Possibili modalità di deployment . . . 111
7.3 MAAS . . . 113
7.3.1 Raspberry Pi 4 UEFI Firmware . . . 113
7.3.2 Progettazione del BMC . . . 124
7.4 Juju . . . 131
7.4.1 Kubernetes Core . . . 131
7.4.2 Ceph Base . . . 133
7.5 Autoscaler . . . 134
7.5.1 Possibili alternative . . . 135
7.5.2 Juju Charmscaler . . . 135
7.5.3 Estensione del Kubernetes cluster autoscaler . . . 136
7.6 Terraform . . . 140
7.7 Infrastruttura di rete e deployment dei controller . . . 141
7.8 Infrastruttura complessiva . . . 143

8 Realizzazione del cluster **145**
8.1 Installazione fisica . . . 145
8.2 Ricompilazione e configurazione del firmware UEFI . . . 147
8.2.1 Problemi riscontrati . . . 148
8.2.2 Aggiunta del power button in ACPI . . . 149

8.2.3 Ordinamento automatico delle opzioni di boot 156
8.2.4 Rimozione del limite relativo alla RAM 158
8.2.5 Ricompilazione del firmware UEFI 158
8.2.6 Riconfigurazione EEPROM 159
8.3 Realizzazione BMC 160
8.3.1 Programmazione microcontrollore tramite MicroPython 160
8.3.2 Interfacciamento con Raspberry 161
8.3.3 Realizzazione del MAAS Power Driver 163
8.4 Installazione MAAS 166
8.4.1 Deployment e configurazione del controller 166
8.4.2 Enlist e configurazione dei vari Raspberry 168
8.4.3 Configurazione dei Raspberry come pod KVM 168
8.5 Installazione Juju 169
8.5.1 Deployment e configurazione del controller 169
8.5.2 Deployment del modello relativo a Ceph e Kubernetes 171
8.6 Estensione del Kubernetes cluster autoscaler 178
8.6.1 Definizione e compilazione della specifica Protobuf 178
8.6.2 Estensione e compilazione del cluster autoscaler 184
8.6.3 Realizzazione del Juju Python Client 185
8.6.4 Definizione del providerID nei kubernetes-worker 187
8.6.5 Configurazione dei nodi Kubernetes 188
8.7 Terraform 188
8.7.1 Installazione e configurazione di Terraform 189
8.7.2 Deployment di MetalLB 189
8.7.3 Deployment del cluster autoscaler 190

9 Verifica e valutazione dei risultati 193
9.1 Realizzazione fisica del cluster 193
9.2 Bare-metal cloud tramite MAAS 194
9.3 Deployment e scaling tramite Juju 195
9.4 Load balancing tramite MetalLB 196
9.5 Volumi persistenti tramite Ceph 198
9.6 Kubernetes cluster autoscaler 200
9.7 Compatibilità del software con l'architettura Arm 202
9.8 Possibili sviluppi futuri 202

Conclusioni 205

Bibliografia 209

Sitografia 210

Introduzione

La crescente diffusione e l'incremento dei servizi digitali, in termini di affidabilità necessaria, ha determinato l'esigenza di progettare applicazioni sempre più performanti e tolleranti ai guasti. Inoltre, sebbene le capacità computazionali dell'hardware in commercio siano costantemente in aumento, si presenta, comunque, un limite fisico alle capacità del singolo elaboratore, che determina le prestazioni massime ottenibili da un'applicazione che sia in esecuzione su di esso. Per questo, è nata l'esigenza di applicazioni distribuibili su più nodi distinti, in modo da poter offrire le capacità computazionali richieste senza limitazioni. Tali necessità hanno portato ad un'evoluzione dello stile architetturale adottato nello sviluppo delle applicazioni, comunemente nota come architettura a microservizi.

L'architettura a microservizi prevede, in generale, di scomporre l'applicazione in più parti distinte, autonome e dotate di responsabilità ben precise. I singoli microservizi potranno essere distribuiti su nodi differenti, ma coopereranno tra loro per realizzare la logica dell'applicazione. Generalmente, il deployment dei singoli microservizi avviene sfruttando dei container, cioè dei pacchetti software contenenti l'applicazione, le sue dipendenze e tutto il necessario per poterla eseguire in maniera isolata ed indipendente dal sistema operativo sottostante. Il deployment di tali container, corrispondenti ai microservizi, potrebbe avvenire su un singolo nodo, utilizzando un container runtime engine, come, ad esempio, Docker, containerd o LXC, oppure su più nodi, raggruppati in un cluster. In quest'ultimo caso, è necessaria la presenza di un cluster management software, nello specifico di un cluster management software capace di orchestrare i diversi container sui vari nodi. Sebbene esistano vari orchestratori di questo tipo, in questa trattazione ci si concentrerà esclusivamente su Kubernetes, che, negli ultimi anni, è diventato il software leader in questo ambito.

Per realizzare un cluster Kubernetes vi sono diverse alternative, determinate dall'utilizzo o meno di un cloud provider e dalla modalità di deployment. Il cloud provider adottato potrebbe essere pubblico, come, ad esempio, Google Cloud Platform o Amazon Web Services, o privato, come, ad esempio, VMware vSphere o Canonical Metal As A Service (MAAS). Il deployment potrebbe poi avvenire tramite un servizio offerto dal cloud provider stesso, come, ad esempio, Amazon EKS o Google Kubernetes Engine, tramite tool di installazione, come, ad esempio, kubeadm o Juju,

che potrebbero comunque sfruttare l'eventuale infrastruttura cloud sottostante, o manualmente, configurando le singole macchine.

Uno dei principali obiettivi di questa trattazione sarà quello di realizzare un'infrastruttura cloud privata, che permetta di installare, configurare e gestire un insieme di macchine fisiche per poter effettuare il deployment automatico di un cluster Kubernetes attraverso un qualche tool di installazione. In particolare, ci si concentrerà su Canonical Metal As A Service (MAAS) per la gestione delle macchine fisiche, mentre, per il deployment del software Kubernetes, verrà utilizzato il tool di installazione Juju. La funzionalità principale di MAAS consiste nella capacità di installare il sistema operativo su una o più macchine fisiche, da remoto e in maniera completamente automatica, sfruttando il boot da rete. In questo modo, le macchine fisiche sono gestibili quasi come macchine virtuali, e possono essere installate, reinstallate o configurate sfruttando delle API messe a disposizione da MAAS. Juju, a sua volta, può sfruttare le macchine offerte da MAAS per installare una serie di software (charm), che, combinati fra loro, permettono di effettuare il deployment di vere e proprie infrastrutture cloud, fra cui, ad esempio, Kubernetes o OpenStack.

Un ulteriore obiettivo consisterà nel realizzare tutto ciò sfruttando dell'hardware a basso costo, in modo da rendere il progetto accessibile anche a chi volesse realizzare un proprio cluster Kubernetes su macchine fisiche. Attualmente, l'hardware presente sul mercato caratterizzato dai costi più bassi è costituito dai cosiddetti single-board computer (SBC), cioè computer completi implementati su una singola scheda. Di solito, gli SBC, per contenere i costi, sono costituiti da processori con architettura Arm, ma ne esistono anche altri con architetture diverse, come x86-64. Attualmente, il leader per eccellenza di questa categoria è il Raspberry Pi 4 B, costituito da un processore arm64 e da una memoria RAM che può raggiungere gli 8 GB. L'obiettivo sarà quindi quello di riuscire a realizzare un'infrastruttura cloud privata basata su tale SBC. Deriva da ciò il vincolo che la totalità del software e dei tool adottati dovrà essere compatibile con l'architettura Arm. La diffusione di tale architettura è in costante crescita, ed inizia ad essere utilizzata, oltre che in ambito mobile e IoT, anche in ambito desktop e server. Questo, anche grazie alla definizione, da parte di Arm, di un insieme di standard (Arm SystemReady), che, se rispettati dalla piattaforma hardware, ne abilitano la compatibilità nativa con un qualsiasi sistema operativo. L'utilizzo di tali SBC, permetterà quindi anche di determinare il grado di maturità degli strumenti di deployment automatico, di Kubernetes e, in generale, di tutto il software utilizzato, nei confronti dell'architettura Arm. Da ciò deriverà quindi la necessità di effettuare una serie di considerazioni e valutazioni per determinare se questo tipo di hardware sia adatto allo scopo o se siano presenti particolari carenze.

In questo documento verranno quindi affrontati una serie di problemi, fra cui, come adattare e rendere compatibile MAAS con i Raspberry Pi, in modo che possano essere gestiti come computer ordinari. Per fare ciò verrà utilizzata l'implementazione

open-source del firmware UEFI TianoCore EDKII per il Raspberry Pi 4 B. Come verrà descritto nei capitoli successivi, tale firmware verrà inoltre modificato e ricompilato, per aggiungere funzionalità e risolvere alcuni problemi. Verrà inoltre realizzato, attraverso un microcontrollore ESP32, un BMC (Baseboard Management Controller), per permettere a MAAS di controllare lo stato, l'accensione e lo spegnimento dei Raspberry Pi da remoto, evitando di intervenire o di comunicare col sistema operativo in esecuzione sul nodo. Rendendo MAAS capace di gestire i vari Raspberry Pi, sarà possibile configurare Juju in modo che utilizzi MAAS come cloud provider, permettendo di effettuare il deployment di un'infrastruttura, come Kubernetes, in maniera automatica su vari Raspberry.

Oltre a quello del deployment automatico del cluster Kubernetes, verrà affrontato anche il problema del ridimensionamento (scaling) automatico del cluster, cioè l'aggiunta e la rimozione automatica di nodi fisici per reagire alla variazione del carico di lavoro. Per realizzare ciò, verrà esteso il cluster autoscaler di Kubernetes in modo che possa interagire con l'infrastruttura cloud privata sottostante, composta da MAAS e Juju, per aggiungere o rimuovere nodi fisici in maniera automatica.

Il cluster Kubernetes sarà inoltre configurato in modo tale da poter allocare volumi persistenti. In particolare, attraverso Juju, verrà effettuato anche il deployment di Ceph, un servizio di storage distribuito, che potrà essere utilizzato dai container in esecuzione su Kubernetes per salvare i propri dati in maniera persistente e tollerante ai guasti.

Successivamente al deployment del cluster tramite Juju, verrà utilizzato anche un altro strumento, Terraform, per facilitare la definizione degli oggetti Kubernetes, come applicazioni o configurazioni, da applicare al cluster.

Infine, verranno effettuati alcuni test, per verificare il corretto funzionamento ed integrazione delle tecnologie sfruttate, mettendo in evidenza eventuali pregi o difetti del sistema realizzato; saranno quindi individuati gli sviluppi e i miglioramenti possibili, considerando anche le tematiche non affrontate, come la sicurezza, e tratte le relative conclusioni.

Questa trattazione non coprirà completamente ogni aspetto delle tecnologie e strumenti utilizzati, ma si limiterà a descrivere ed approfondire solamente le parti utili al raggiungimento degli obiettivi. Inoltre, poiché i vari software utilizzati sono in costante sviluppo, sicuramente, alcune delle funzionalità o problemi individuati non rimarranno tali. In ogni caso, si vuole comunque proporre una soluzione adottabile, per raggiungere gli obiettivi, nella condizione attuale, anche se questa muterà inevitabilmente nel prossimo futuro.

Il documento sarà strutturato in questo modo:

Cluster computing, infrastrutture cloud e microservizi Definizione di cluster computing e di cluster management software. Classificazione delle possibili infrastrutture cloud e delle modalità per realizzare cluster al di sopra di esse. Introduzione all'architettura a microservizi.

Hardware in ambito server e cloud Descrizione delle tipologie di hardware utilizzato in ambito server; tra queste, la crescente diffusione dell'architettura Arm. Descrizione delle caratteristiche dell'architettura Arm ed individuazione delle principali differenze rispetto all'architettura x86-64. Individuazione e descrizione dei principali fattori che stanno rendendo l'architettura Arm, in termini di compatibilità software, sempre più simile all'architettura x86-64, e quindi, sempre più diffusa sia in ambito server che desktop, con particolare attenzione agli standard Arm SystemReady, UEFI (Unified Extensible Firmware Interface) e ACPI (Advanced Configuration and Power Interface).

Bare-metal cloud Individuazione delle principali tecnologie per realizzare dei bare-metal cloud, cioè infrastrutture cloud che mettono a disposizione servizi idonei ad installare o reinstallare automaticamente macchine fisiche. Descrizione approfondita dell'alternativa scelta: Canonical Metal As A Service (MAAS).

Infrastructure as Code Definizione del termine Infrastructure as Code, definizione del legame tra Infrastructure as Code ed infrastruttura cloud, individuazione dei principali strumenti e tecnologie disponibili in questo ambito. Descrizione approfondita degli strumenti utilizzati: Canonical Juju e Terraform.

Container orchestration Definizione del termine Container orchestration ed individuazione delle principali alternative per l'orchestrazione di container. Introduzione e descrizione approfondita dell'orchestratore scelto per il deployment di applicazioni a microservizi sul cloud privato realizzato: Kubernetes. Descrizione dei principali oggetti Kubernetes, del load balancing, dei volumi persistenti e delle modalità di scaling delle applicazioni. Per ognuno di questi argomenti saranno approfondite alcune tecnologie specifiche adottabili. In particolare, MetalLB per quanto riguarda il load balancing, il servizio di storage distribuito Ceph per quanto riguarda i volumi persistenti ed il Kubernetes cluster autoscaler per quanto riguarda lo scaling automatico di nodi.

Definizione degli obiettivi Definizione degli obiettivi da raggiungere sfruttando ed integrando le tecnologie individuate nei capitoli precedenti. Fra questi, la realizzazione di un cluster fisico a basso costo composto da Raspberry, la realizzazione di un bare-metal cloud che permetta il deployment automatico di ogni singolo nodo e lo scaling autonomo del cluster per reagire a variazioni di carico.

Analisi del problema e progettazione del cluster Determinazione delle principali criticità da affrontare. Individuazione e descrizione dell'hardware e del software da utilizzare. Progettazione fisica del cluster, progettazione del BMC (Baseboard Management Controller), dell'infrastruttura di rete, collocazione dei controller di

MAAS e Juju, definizione delle modalità di deployment di Kubernetes, progettazione del cluster autoscaler. Definizione delle interazioni e responsabilità di ogni strumento utilizzato.

Realizzazione del cluster Descrizione delle strategie adottate, problemi affrontati e funzionalità aggiunte al software utilizzato per raggiungere gli obiettivi preposti. Fra questi, l'installazione fisica del cluster, la configurazione e ricompilazione del firmware UEFI per il Raspberry, la realizzazione del BMC, l'installazione e configurazione di MAAS, il deployment di Kubernetes e Ceph attraverso Juju, il set-up di Terraform e la realizzazione del cluster autoscaler.

Verifica e valutazione dei risultati Descrizione dei test effettuati sul sistema per verificare il raggiungimento degli obiettivi. Individuazione dei pregi e difetti del sistema realizzato, definizione dei possibili sviluppi futuri.

Conclusioni Riassunto e discussione del sistema realizzato e dei risultati ottenuti.

Capitolo 1

Cluster computing, infrastrutture cloud e microservizi

Come è stato accennato nel capitolo introduttivo e come viene specificato dal titolo di questo documento, l'obiettivo principale che si vuole raggiungere consiste nel riuscire a realizzare un cluster di elaboratori. In questo capitolo si andrà quindi a definire meglio cosa si intende per cluster, delineando quale sia il contesto attuale nell'ambito del calcolo parallelo.

Verranno quindi individuati i fattori principali che hanno favorito l'impiego sempre più diffuso, in diversi ambiti, di cluster di elaboratori e quali architetture applicative sia necessario adottare per poter realizzare applicazioni capaci di sfruttare appieno le potenzialità di tali sistemi.

Saranno inoltre esplorate le possibili alternative per realizzare questi sistemi, affidandosi o meno ad un particolare cloud provider pubblico o privato. Verranno individuate le principali alternative cloud pubbliche e private, classificandole in base al livello di astrazione delle risorse esposte all'utente.

1.1 I limiti dei processori single-core

Fino all'inizio degli anni 2000, i microprocessori erano composti da una singola unità di elaborazione, e le prestazioni miglioravano, di anno in anno, aumentando la frequenza di lavoro di tale unità, cioè il numero di istruzioni eseguite in un determinato intervallo di tempo. Le prestazioni delle applicazioni potevano quindi essere migliorate semplicemente eseguendole sul nuovo modello di processore, che poteva garantire velocità di esecuzione più elevate del precedente.

In generale, esistono due alternative possibili per migliorare la frequenza di lavoro di un microprocessore. La prima consiste nel migliorare il processo produttivo, cioè diminuire la dimensione dei componenti interni del microprocessore (transistor).

Riducendo le dimensioni, si riduce la distanza che gli impulsi elettrici devono percorrere e si riduce il tempo di transizione da uno stato all'altro dei singoli transistor [137], permettendo di aumentare la frequenza di lavoro del processore.

La seconda alternativa consiste invece nell'aumentare il voltaggio del processore: ogni transistor deve accumulare una carica per cambiare di stato e il tempo di accumulazione è direttamente proporzionale alla corrente, che, a sua volta è direttamente proporzionale al voltaggio [137]. Aumentando il voltaggio, quindi, è possibile aumentare la frequenza di lavoro, ma sorge un problema: la potenza dissipata. La formula della potenza dissipata da un processore è la seguente:

$$P \simeq C * V^2 * f$$

dove:

P è la potenza dissipata dal processore sotto forma di calore.

C è la capacità dinamica del circuito.

V è il voltaggio applicato al circuito del processore.

f è la frequenza del processore.

Da tale formula è possibile notare che il calore dissipato è proporzionale al quadrato del voltaggio. Dal ragionamento precedente, cioè che la frequenza del processore è proporzionale al voltaggio, deriva che il calore dissipato è proporzionale al cubo del voltaggio applicato al processore [137]. Per questo, non è possibile aumentare il voltaggio di un processore senza incorrere in problemi di dissipazione.

La prima strategia è stata quella utilizzata fino ai primi anni 2000, ma, attualmente, sono stati pressoché raggiunti i limiti fisici delle dimensioni di un transistor. La seconda alternativa non è invece applicabile, poiché esistono limiti nella quantità dissipabile di calore e, inoltre, si otterrebbe un'inefficienza troppo elevata in termini di consumi.

La soluzione al problema di continuare ad aumentare le prestazioni, è consistita, quindi, nel dirigersi verso il parallelismo: creare cioè processori composti da più unità di elaborazione, chiamati processori multi-core. I processori multi-core sono una delle possibili architetture parallele.

Tale soluzione però, rispetto al semplice incremento della frequenza del processore, comporta la necessità di adattare l'applicazione all'architettura multi-core, adattamento che non sempre è possibile e che, in ogni caso, potrebbe richiedere ingenti sforzi al programmatore.

1.2 Architetture parallele

Le architetture parallele vengono frequentemente classificate attraverso la tassonomia di Flynn, che individua quattro categorie, in base al numero di istruzioni e dati gestibili contemporaneamente dall'architettura:

SISD Single Instruction stream Single Data stream. È l'architettura classica dei processori single-core (senza estensioni SIMD). Il sistema può eseguire una singola istruzione per volta su un singolo dato [98, pag. 29].

SIMD Single Instruction stream Multiple Data stream. Sono sistemi paralleli che eseguono una singola istruzione per volta, ma possono applicarla contemporaneamente su più dati [98, pag. 30]. Ricadono in questa categoria i processori dotati di estensioni SIMD (come MMX, SSE o AVX) e le GPU, composte da migliaia di core che possono applicare parallelamente le stesse operazioni su dati diversi.

MIMD Multiple Instruction stream Multiple Data stream. Sono sistemi che possono eseguire contemporaneamente più istruzioni diverse su più dati. Solitamente sono costituiti da un insieme di unità di calcolo indipendenti, ognuna delle quali esegue le proprie istruzioni indipendentemente dalle altre [98, pag. 32]. Ricadono in questa categoria, ad esempio, i processori multi-core ed i cluster di elaboratori.

MISD Multiple Instruction stream Single Data stream. Sono sistemi che possono eseguire contemporaneamente più istruzioni diverse sugli stessi dati. Attualmente non esistono sistemi che adottano questa architettura.

I sistemi attualmente in commercio sono principalmente di categoria MIMD, spesso con la possibilità aggiunta di operare come SIMD, per massimizzare le prestazioni in particolari contesti di calcolo.

1.2.1 Classificazione dei sistemi MIMD

I sistemi MIMD sono a loro volta classificabili in due categorie, in base alla modalità con cui i dati vengono condivisi fra le varie unità di calcolo.

Memoria condivisa

In questo caso, ogni unità di calcolo può accedere direttamente ad ogni locazione di memoria. L'accesso alla memoria avviene, solitamente, attraverso un bus condiviso o tramite una griglia di commutazione [98, pag. 35]; le griglie di commutazione sono più veloci dei bus, ma richiedono costi maggiori per la loro realizzazione [98, pag. 37]. Le

varie unità di calcolo comunicheranno tra loro scambiandosi delle strutture dati direttamente in memoria, ma dovranno anche adottare delle politiche di sincronizzazione per regolare l'accesso ad essa. Esistono alcune varianti di sistemi di questo tipo, contraddistinte dalle differenze di velocità di accesso a diverse aree di memoria:

UMA (Uniform Memory Access) Il sistema ricade in questa categoria se ogni unità di calcolo può accedere ad ogni area di memoria con la medesima velocità. Ricadono in questa categoria i sistemi dotati di un singolo processore multi-core, ma esistono anche processori multi-core NUMA, solitamente quelli dotati di un numero molto elevato di core.

NUMA (Non-Uniform Memory Access) Il sistema ricade in questa categoria se alcune unità di calcolo possono accedere più velocemente ad alcune aree di memoria e più lentamente ad altre. Solitamente, ogni unità o gruppo di unità possiede un'area di memoria direttamente connessa, a cui vi può accedere velocemente; nel caso uno dei nodi debba accedere ad un'area direttamente connessa ad un altro gruppo, comunicherà con un hardware di interconnessione apposito, che permette di accedere alle aree di memoria di tutti gli altri gruppi ma in maniera più lenta. Ricadono in questa categoria i sistemi composti da più processori multi-core.

Attualmente, i sistemi paralleli più diffusi a memoria condivisa sono quelli dotati di un singolo processore multi-core, architettura ormai adottata da ogni processore in commercio. Ogni core del processore, infatti, ha la propria indipendenza, ma può accedere ad ogni locazione di memoria del sistema senza restrizioni.

Memoria distribuita

In questo caso, ogni unità di calcolo possiede una propria memoria privata e può accedere direttamente solamente ad essa. Ogni unità di calcolo può però scambiare dati con le altre unità attraverso una rete di interconnessione. Ogni unità esegue quindi le elaborazioni sulla propria memoria, e, nel caso abbia bisogno di un risultato o di un dato in possesso di un'altra unità, dovrà richiederlo e reperirlo attraverso la rete di interconnessione. Questo meccanismo evita di dover sincronizzare gli accessi in memoria, ma rende più complesso lo scambio di dati fra i diversi nodi. Esistono diverse varianti di sistemi di questo tipo, solitamente differenziate dal grado di accoppiamento ed eterogeneità dei vari nodi:

MPP (Massively Parallel Processing) Il sistema ricade in questa categoria se i nodi sono fortemente accoppiati, con hardware specifico di alto livello ed interconnessi da una rete ad alte prestazioni, per offrire le massime capacità di calcolo possibili con la tecnologia attualmente disponibile.

Cluster computing Il sistema ricade in questa categoria se i nodi sono fortemente accoppiati, costituiti da hardware generico ma omogeneo ed interconnessi da una rete LAN Ethernet. Un cluster, rispetto a MPP, richiede costi minori di realizzazione, poiché viene utilizzato un hardware generico, sia per la realizzazione dei nodi che della rete; nonostante questo, le prestazioni ottenibili sono comunque elevate, grazie alla possibilità di ridimensionare il cluster, e quindi le prestazioni, aggiungendo o rimuovendo nodi.

Grid computing Il sistema ricade in questa categoria se i nodi sono debolmente accoppiati, dispersi geograficamente e con hardware eterogeneo. È adatto a tutte quelle applicazioni che richiedono grandi capacità di calcolo ma che non hanno forti esigenze in termini di reattività o di sincronizzazione tra i vari nodi di elaborazione.

L'architettura più diffusa, in ambito commerciale, tra quelle appena descritte, è sicuramente quella del cluster computing.

1.2.2 Limiti delle architetture parallele

Le architetture parallele appena descritte offrono sicuramente, in termini di performance, notevoli miglioramenti rispetto ad un'architettura single-core SISD. Si manifesta però un vincolo fondamentale: l'applicazione deve essere in grado di sfruttare tale architettura parallela, altrimenti non potrà trarne beneficio. Ad esempio, dovrà utilizzare istruzioni SIMD per sfruttare le estensioni SIMD del processore; utilizzare più processi o thread per sfruttare un'architettura MIMD a memoria condivisa; utilizzare librerie, paradigmi o architetture applicative apposite per sfruttare un'architettura MIMD a memoria distribuita, come, ad esempio, il paradigma ad attori o le architetture a microservizi.

1.3 Scaling

Con il termine scaling si intende il ridimensionamento del sistema, variando il numero di risorse e, quindi, le capacità computazionali, da esso offerte. Lo scaling può essere classificato in due categorie: scale up e scale out.

1.3.1 Scale up

Consiste nell'aumentare le capacità offerte dal sistema rimpiazzando alcune risorse con altre più prestanti. Ad esempio, installando una CPU più potente, installando una memoria RAM o dischi più capienti o sostituendo una scheda di rete con una più

veloce. Questo tipo di scaling è molto efficace nel momento in cui viene rilasciata una nuova tecnologia hardware più prestante delle precedenti, ma pone dei limiti alle prestazioni massime raggiungibili: ad esempio, successivamente all'installazione della CPU più veloce presente sul mercato, non sarà possibile aumentare ulteriormente le prestazioni. Inoltre, la tolleranza ai guasti, a seguito dell'operazione di scale up, rimarrà pressoché la medesima, a meno che il tipo di hardware adottato non migliori anche tale caratteristica. L'operazione contraria allo scale up è lo scale down, che consiste nel ridurre le capacità offerte dal sistema rimpiazzando alcune risorse con altre meno prestanti.

1.3.2 Scale out

Consiste nell'aumentare le capacità offerte dal sistema aggiungendo altre risorse che opereranno parallelamente a quelle già presenti. Ad esempio, nel contesto di un cluster di elaboratori, consiste nell'aggiungere ulteriori nodi al sistema, che incrementeranno le capacità computazionali totali. Questo tipo di scaling non ha i limiti stringenti dello scale up, permettendo di raggiungere prestazioni molto superiori. In compenso però, il software dovrà essere in grado di sfruttare in parallelo più repliche dello stesso tipo di risorsa. Inoltre, la tolleranza ai guasti, a seguito dell'operazione di scale out, sarà incrementata, poiché esisteranno più istanze dello stesso tipo di risorsa. L'operazione contraria allo scale out è lo scale in, che consiste nel ridurre le capacità offerte dal sistema eliminando alcune delle risorse presenti.

1.4 Cluster computing

Come affermato nella Sezione 1.2, un cluster è un insieme omogeneo di elaboratori che comunicano tra loro, attraverso una rete LAN, per raggiungere un obiettivo. L'obiettivo potrebbe consistere nell'eseguire un'applicazione, distribuendola sui vari nodi, per massimizzare le performance. Attualmente, la diffusione e adozione delle architetture basate su cluster è in costante crescita, in vari ambiti applicativi, come big data, high performance computing, machine learning e, in generale, in tutti quegli ambiti in cui è necessario realizzare applicazioni che possano offrire un servizio altamente performante e tollerante ai guasti.

I punti di forza dell'architettura cluster computing sono, principalmente, i seguenti: i limiti delle architetture MIMD a memoria condivisa, i bassi costi di realizzazione, la facilità di ridimensionamento del sistema, le elevate prestazioni e la tolleranza ai guasti, la crescente disponibilità di software capace di sfruttare e gestire le risorse di tale architettura.

1.4.1 Le cause della diffusione

Come descritto nella Sezione 1.2, considerati i limiti dei processori single-core, ci si è orientati verso l'utilizzo di architetture parallele MIMD a memoria condivisa: i processori multi-core. In realtà, anche quest'ultima architettura possiede dei limiti: non è infatti possibile aumentare con facilità il numero di core evitando di creare un collo di bottiglia sul bus di interconnessione tra i core e la memoria [98, pag. 46]. Una possibile soluzione è rappresentata dalle architetture NUMA, che richiedono comunque ingenti sforzi di progettazione e realizzazione. Per superare tali limiti, ci si è quindi diretti verso architetture parallele MIMD a memoria distribuita, cioè sistemi composti da più nodi indipendenti interconnessi tra loro da una rete di comunicazione a scambio di messaggi. Solitamente, in un sistema di questo tipo, ogni nodo è composto da un processore multi-core, per cui lo si potrebbe classificare come una soluzione ibrida fra MIMD a memoria condivisa e MIMD a memoria distribuita. Nella pratica, ogni sistema MIMD a memoria distribuita è composto da nodi MIMD a memoria condivisa, per cui tale classificazione non è diffusamente utilizzata.

Tra le varie architetture MIMD a memoria distribuita, quella più diffusa è il cluster computing. Le cause di tale diffusione sono imputabili, sicuramente, ai costi bassi ed alla facilità di realizzazione: è possibile realizzare un cluster attraverso del commodity hardware [98, pag. 35], cioè hardware generico, facilmente reperibile in commercio e facilmente intercambiabile con hardware di produttori diversi [30]. Inoltre, i vari nodi saranno interconnessi da una rete Ethernet, per cui, anche l'hardware di rete adottabile, potrà essere generico, reperibile in commercio e facilmente intercambiabile. Tale genericità, dal punto di vista dell'hardware e della rete, permette facilmente di aggiungere o rimuovere nodi dal sistema, ridimensionando a piacere le capacità computazionali offerte.

Un cluster permette quindi di realizzare un sistema ad elevatissime prestazioni, con la possibilità di aumentarle ulteriormente aggiungendo nodi, e tollerante ai guasti. Infatti, nel caso in cui uno dei nodi entrasse in stato di errore, il resto del sistema continuerebbe ad operare correttamente, continuando ad offrire il servizio richiesto. Architetture di questo genere sono quindi indispensabili, nel contesto attuale, in cui i servizi digitali sono sempre più diffusi ed esigenti in termini di prestazioni ed affidabilità.

1.4.2 Cluster management software

Bisogna tenere in considerazione che, per sfruttare appieno le capacità di un cluster, è necessario che il software sia in grado di operare su di esso.

Una prima soluzione potrebbe essere quella di realizzare un software in grado di operare su un'architettura distribuita, suddividendo il software in diversi componenti, installando e configurando ognuno di essi su ogni nodo e gestendo manualmente, lato

applicazione, ogni singola problematica legata allo scambio di messaggi o all'allocazione di risorse. Una strategia di questo tipo sarebbe sicuramente adottabile, ma presenta una serie di lati negativi. Infatti, ogni nuova applicazione dovrebbe implementare una propria logica per poter sfruttare le risorse del sistema, senza contare che la gestione delle risorse del cluster, come, ad esempio, a quale nodo assegnare l'applicazione, dovrebbe essere effettuata manualmente. Inoltre, in caso di malfunzionamenti su un determinato nodo, sarebbe necessario che l'amministratore riavviasse manualmente le parti dell'applicativo fallite su altri nodi.

Queste problematiche di assegnazione delle risorse, gestione delle comunicazioni, riavvio di applicazioni fallite, bilanciamento del carico, gestione dello storage, scaling automatico, sono comuni ad ogni applicazione distribuita. Sarebbe utile quindi definire un livello che risolvesse tali problematiche per ogni applicazione, permettendo allo sviluppatore di astrarre dalle problematiche di basso livello e di concentrarsi sulla logica applicativa. Tale livello è solitamente chiamato cluster management software, cioè un software installato su ogni nodo del cluster, che gestisce le risorse di ogni singola macchina ed offre alle applicazioni un insieme di servizi per potervi accedere. Solitamente tali software gestiscono il cluster in maniera centralizzata attraverso un nodo denominato master, che monitora e coordina gli altri nodi denominati worker.

Esistono diverse alternative di cluster management software, differenziati, principalmente, dall'ambito applicativo, come, ad esempio, orchestrazione di container o big data, che devono gestire. Infatti, in un contesto di orchestrazione di container il cluster management software dovrà offrire le modalità per gestire l'allocazione dei vari container sui vari nodi, monitorare il loro stato di salute e riavviarli quando necessario; in un contesto big data, invece, dovrà offrire le modalità per distribuire la computazione, per l'accesso ai dati e per la raccolta dei risultati. Quindi, in base all'ambito applicativo, le responsabilità del cluster management software potrebbero cambiare.

Alcuni dei principali cluster management software attualmente disponibili sono i seguenti [66]:

Apache Mesos Un kernel di sistema distribuito. Mesos applica gli stessi principi del kernel Linux, ma ad un differente livello di astrazione. Il kernel Mesos viene eseguito su ogni macchina ed offre alle applicazioni delle API per la gestione delle risorse e dello scheduling nell'ambito dell'intero data center [8].

Kubernetes Orchestratore di container open-source che offre meccanismi per il deployment, la manutenzione e lo scaling di applicazioni; progettato, originariamente da Google [59].

Docker Swarm Orchestratore di container che permette di raggruppare più Docker engine in esecuzione su nodi diversi in un singolo Docker engine virtuale [1][42].

HashiCorp Nomad Un semplice e flessibile orchestratore che permette il deployment e la gestione di applicazioni sia containerizzate che non containerizzate, su infrastrutture sia cloud che on-premises [95].

OpenHPC Un insieme di tool per il provisioning, per la gestione delle risorse e per lo sviluppo necessari alla realizzazione e gestione di cluster High Performance Computing (HPC) Linux [31].

Hadoop YARN Yet Another Resource Negotiator. Gestisce le risorse computazionali di un cluster e le alloca alle applicazioni che le richiedono [7].

1.5 Infrastrutture cloud

I software descritti nella Sezione 1.4.2 offrono molti vantaggi dal punto di vista delle applicazioni e dell'amministratore del cluster. Restano, comunque, alcune difficoltà, che potrebbero scoraggiare la realizzazione di un sistema di questo tipo. In particolare, per realizzare un cluster, dal punto di vista fisico, è necessario reperire ed installare l'hardware, per realizzare i vari nodi e l'infrastruttura di rete. Tali operazioni sono, sicuramente, molto onerose, sia dal punto di vista dei costi, che della progettazione e della manutenzione. Inoltre, dopo aver realizzato fisicamente il cluster, è necessario configurare ed installare il software di gestione sui vari nodi, operazione che richiede elevate competenze tecniche.

La soluzione a questi problemi può essere individuata nel cloud computing, cioè un servizio o un insieme di servizi che permettono di richiedere dinamicamente un insieme di risorse; tali risorse, potrebbero essere, ad esempio, macchine complete sulle quali eseguire delle applicazioni o, perfino, un intero cluster di macchine preconfigurate.

1.5.1 Risorse on demand

Una delle caratteristiche essenziali del cloud computing è la possibilità di richiedere dinamicamente determinate risorse, tramite servizi che potrebbero essere esposti all'utilizzatore attraverso delle API. Queste risorse vengono reperite da una pool, condivisa tra tutti gli utilizzatori. Nel momento in cui, un utilizzatore richiede determinate risorse, queste vengono selezionate dalla pool e assegnate al richiedente. Le risorse potranno poi essere rilasciate dall'utilizzatore e ritornare nella pool in qualsiasi momento. Ciò permette all'utilizzatore di ridimensionare dinamicamente la quantità di risorse a disposizione per le proprie applicazioni, in base alle necessità.

In generale, le risorse offerte da un'infrastruttura cloud possono essere classificate in tre categorie di base: elaborazione, memoria e rete [93, cap. 3]. La combinazione di queste può definire risorse più complesse, offerte dal cloud provider per velocizzare e

facilitare il deployment di determinate entità frequentemente utilizzate o per offrire al cliente servizi di più alto livello. Tale questione verrà approfondita nella Sezione 1.5.2.

Elaborazione

È la categoria che comprende tutte quelle risorse che offrono capacità computazionale per l'esecuzione di applicazioni. Si differenziano in base al tipo di istanza computazionale offerta, che potrebbe essere, ad esempio:

Macchina fisica Il cloud provider permette di allocare nuove macchine fisiche dinamicamente su richiesta. Cloud provider di questo tipo vengono chiamati anche bare-metal cloud, approfonditi nel Capitolo 3.

Macchina virtuale Il cloud provider permette di allocare nuove macchine virtuali, cioè macchine che offrono le stesse funzionalità di macchine fisiche, ma che non possiedono un hardware fisico dedicato. L'hardware fisico è gestito da un software, denominato hypervisor. Tale hypervisor può creare e gestire un insieme di macchine virtuali, fornendo loro parte dell'hardware fisico sotto forma di hardware virtualizzato. Attraverso l'hardware virtualizzato, più macchine virtuali possono condividere lo stesso hardware fisico, mantenendo, comunque, un elevato isolamento tra di loro. Le macchine virtuali, rispetto alle macchine fisiche, sono molto più facili da installare, configurare e gestire; per questo, sono ampiamente utilizzate in ambito cloud. Inoltre, il software in esecuzione all'interno della macchina virtuale, grazie alla virtualizzazione dell'hardware, sarà eseguito allo stesso modo su qualsiasi hardware fisico, garantendo un'elevata portabilità. Va sottolineato però, che la virtualizzazione dell'hardware e la necessità di un sistema operativo per ogni macchina, porta ad un notevole spreco di risorse.

Container Il cloud provider permette di allocare dei container, cioè pacchetti software contenenti l'applicazione, le sue dipendenze e tutto il necessario per poterla eseguire in maniera isolata ed indipendente dal sistema operativo sottostante. Rispetto alle macchine virtuali, i container possono sprecare meno risorse, poiché non virtualizzano l'hardware, ma si appoggiano, invece, direttamente sul sistema operativo della macchina che li ospita. Nonostante ciò, l'isolamento viene comunque garantito, anche se in maniera ridotta, grazie a particolari meccanismi del sistema operativo che li ospita.

Cluster Il cloud provider permette di allocare interi cluster, cioè gruppi di macchine, virtuali o fisiche, gestite come un unico sistema. Le varie macchine potrebbero non essere configurate, cioè essere semplicemente un gruppo di macchine sulla stessa rete, provviste esclusivamente del sistema operativo. In questo caso, sarà

compito dell'amministratore, installare, manualmente o attraverso particolari tool, il cluster management software che ritiene più adatto. Altrimenti, il cloud provider potrebbe offrire un cluster preconfigurato, già dotato di un cluster management software ed immediatamente utilizzabile. In quest'ultimo caso si può parlare di Cluster as a Service [93, cap. 14].

Memoria

È la categoria che comprende tutte quelle risorse che offrono capacità di archiviazione, come dischi, volumi persistenti, database o repository. Le risorse di elaborazione potrebbero usufruire di questo tipo di risorse per salvare persistentemente i propri dati, che potrebbero consistere, ad esempio, in interi dischi relativi a macchine virtuali in esecuzione, in file di configurazione, in segreti come password e chiavi o in singoli record di un database. Spesso, le risorse di archiviazione sfruttano tecnologie di replicazione e sharding, per essere tolleranti ai guasti e per migliorare le performance.

Rete

È la categoria che comprende tutte quelle risorse che permettono di modellare la rete presente fra le varie risorse di elaborazione. Alcune delle principali risorse di rete potrebbero essere, ad esempio, subnet di rete o LAN virtuali, router di rete, switch, gateway o firewall. In generale, la possibilità di definire e riconfigurare tramite software la rete presente tra i vari nodi di un sistema, viene detta Software Defined Networking (SDN).

1.5.2 Modelli di servizio

Come descritto nella Sezione 1.5.1, il punto di forza delle infrastrutture cloud è la loro capacità di offrire, tramite un servizio, risorse on demand. In base alle modalità ed al tipo di risorse offerte, cioè se il cloud provider fornisce semplicemente risorse base o se invece fornisce risorse e servizi più complessi, è possibile classificare i cloud provider, come specificato dal NIST [128], in tre categorie:

Infrastructure as a Service (IaaS) Viene fornita al cliente la possibilità di richiedere risorse base, cioè di elaborazione, memoria o rete. Il cliente potrà utilizzare tali risorse per eseguire un qualsiasi software, come un particolare sistema operativo o un'applicazione. Il cliente non dovrà gestire l'infrastruttura cloud sottostante, ma avrà un controllo completo delle risorse richieste [128].

Platform as a Service (PaaS) Viene fornita al cliente la possibilità di effettuare il deployment, sull'infrastruttura cloud, di applicazioni, create sfruttando librerie, servizi o tool supportati dal provider. Il cliente non avrà il controllo né

dell'infrastruttura cloud sottostante né delle risorse base utilizzate dal provider per eseguire le applicazioni dell'utente; ha però il completo controllo delle proprie applicazioni e, eventualmente, la possibilità di configurare l'ambiente di esecuzione [128].

Software as a Service (SaaS) Viene fornita al cliente la possibilità di usufruire di applicazioni in esecuzione sull'infrastruttura cloud del provider. L'utente potrà accedere all'applicazione sfruttando, ad esempio, un'interfaccia web o un'application programming interface (API). Il cliente non avrà il controllo né dell'infrastruttura cloud sottostante né delle risorse base utilizzate dal provider per eseguire l'applicazione e neppure il controllo dell'applicazione stessa. Potrà però avere il controllo, eventualmente, di alcune configurazioni utente dell'applicazione [128].

Queste categorie potrebbero essere considerate come un'architettura a livelli, in cui il livello base è composto dai servizi IaaS e, al di sopra di esso, si posizionano prima il livello PaaS e poi SaaS. Ognuno dei livelli, per realizzare i propri servizi, potrebbe sfruttare quelli del livello sottostante. In realtà, ciò non è obbligatorio: si potrebbe, ad esempio, realizzare un modello SaaS senza implementare i livelli sottostanti, anche se, probabilmente, sarebbero richiesti sforzi maggiori, poiché non si ha la possibilità di sfruttare gli automatismi dei livelli sottostanti.

Oltre a queste categorie di base, stanno emergendo ulteriori categorie, che spesso, sono una combinazione o un caso specifico di quelle appena descritte. Il numero di queste categorie è in continuo aumento, poiché, spesso, vengono coniate per fini puramente commerciali e non perché ve ne sia realmente la necessità. In generale, ricadono nel cosiddetto XaaS (Anything as a Service), cioè fornire qualsiasi cosa attraverso un servizio. Tra queste:

Function as a Service (FaaS) Il cliente può inviare direttamente al cloud provider il codice delle funzioni da eseguire. Tali funzioni verranno eseguite senza che l'utente debba installare o configurare l'ambiente di esecuzione. Questo tipo di modello è conosciuto anche con il nome di serverless computing. Un esempio di cloud provider che offre questo modello è AWS Lambda.

Cluster as a Service Viene fornita al cliente la possibilità di richiedere interi cluster, preconfigurati o meno. Alcuni esempi di cloud provider che offrono questo modello sono Amazon EKS o Google Kubernetes Engine; entrambi permettono di richiedere un cluster preconfigurato con Kubernetes.

Database as a Service (DBaaS) Viene fornita al cliente la possibilità di accedere ad un servizio di database, senza dover configurare l'infrastruttura sottostante. Un esempio di cloud provider che offre questo modello è Amazon Relational Database Service (RDS).

Container as a Service (CaaS) Viene fornita al cliente la possibilità di effettuare il deployment, sull'infrastruttura cloud, di applicazioni containerizzate. Rispetto a PaaS, l'utente ha un controllo maggiore delle risorse base utilizzate dal provider, poiché può specificare le risorse e l'ambiente necessari al container per operare correttamente. Contrariamente a IaaS, che utilizza come risorsa base macchine virtuali o bare-metal, CaaS utilizza, come risorsa base, i container. Si posiziona quindi tra un modello IaaS e PaaS. Un esempio di cloud provider che offre questo modello è AWS Fargate.

Metal As A Service Viene fornita al cliente la possibilità di richiedere ed allocare intere macchine fisiche. Metal As A Service corrisponde al nome del tool per realizzare bare-metal cloud offerto da Canonical, MAAS, approfondito nel Capitolo 3.

Uno degli obiettivi di questa trattazione sarà quello di effettuare il deployment di un cluster di elaboratori in maniera automatica. Tale automatismo potrà essere raggiunto, senza troppe difficoltà, sfruttando i servizi messi a disposizione da un cloud. In particolare, sarà necessario adottare un'infrastruttura cloud che offra, almeno, il modello IaaS, per poter allocare le macchine con le quali realizzare il cluster. Inoltre, poiché le macchine dovranno essere fisiche e non virtuali, sarà necessario adottare tecnologie per realizzare un bare-metal cloud. Successivamente, sfruttando il livello IaaS appena definito, si realizzerà un modello riconducibile al Cluster as a Service, permettendo, in maniera automatica, di realizzare un cluster Kubernetes preconfigurato.

1.5.3 Modelli di deployment

Ogni infrastruttura cloud può essere classificata in base al numero ed al tipo di soggetti coinvolti nella condivisione delle risorse fisiche. Il NIST individua quattro categorie [128]:

Public Cloud I servizi vengono offerti attraverso una rete aperta al pubblico. L'infrastruttura cloud è quindi condivisa tra più clienti e, per questo, offre dei livelli di sicurezza ed isolamento potenzialmente non sufficienti per applicazioni critiche. L'hardware dell'infrastruttura cloud è gestito interamente dal cloud provider e si trova nella sua sede. Attualmente, i principali provider cloud pubblici sono Amazon Web Services, Google Cloud Platform e Microsoft Azure.

Private Cloud L'infrastruttura cloud è dedicata interamente ad un singolo cliente. Poiché l'hardware non è condiviso offre i livelli di sicurezza ed isolamento massimi. L'hardware dell'infrastruttura cloud potrebbe essere gestito da un provider esterno o direttamente dal cliente e potrebbe essere collocato

internamente alla sede del cliente o all'esterno. Un'infrastruttura cloud privata gestita interamente da un'organizzazione offre, sicuramente, il grado più elevato di sicurezza ma richiede anche ingenti sforzi economici e competenze interne. In quest'ultimo caso, l'organizzazione, per realizzare l'infrastruttura, dovrà adottare tecnologie e software appositi. Attualmente, i principali software per la realizzazione di cloud privati sono OpenStack, CloudStack, VMware vSphere o, per la realizzazione di bare-metal cloud, Canonical Metal As A Service (MAAS).

Community Cloud L'infrastruttura cloud è dedicata ad un gruppo di organizzazioni, con obiettivi comuni, che hanno scelto di cooperare per la realizzazione dell'infrastruttura cloud. L'hardware dell'infrastruttura cloud potrebbe essere gestito da una o più organizzazioni del gruppo e potrebbe essere collocata in sede o presso un provider esterno. Offre un grado di sicurezza intermedio tra il public cloud ed il private cloud.

Hybrid Cloud L'infrastruttura cloud è composta da due o più distinte infrastrutture cloud (public, private o community), che mantengono la loro indipendenza ma che cooperano per raggiungere un determinato obiettivo.

Per raggiungere l'obiettivo principale di questa trattazione, che prevede di realizzare un cluster con hardware fisico gestito privatamente, sarà necessario adottare il modello private cloud, sfruttando uno dei possibili software per realizzare cloud di questo tipo.

1.6 Architettura a microservizi

Come già accennato nella Sezione 1.4.2, il software, per sfruttare le capacità di un'architettura a memoria distribuita, dovrà adottare tecnologie, paradigmi e stili architetturali appositi, che permettano di ottenere tutti i vantaggi che i sistemi distribuiti possono offrire. Esistono alcuni software, denominati middleware, come, ad esempio, i cluster management software, che offrono alle applicazioni un insieme di servizi per facilitare alcune operazioni sul sistema distribuito, solitamente relative alla comunicazione e gestione dei dati.

Sicuramente, come affermato nella Sezione 1.4.2, questi software sono necessari in un sistema distribuito, poiché permettono all'applicazione di evitare di risolvere problematiche comuni in questo tipo di sistemi. Non possono però consentire all'applicazione di ignorare la presenza del sistema distribuito sottostante: l'applicazione dovrà comunque adottare un'architettura applicativa che ne permetta la distribuzione su più nodi, se si desidera sfruttare al massimo i vantaggi di un sistema distribuito. Tali vantaggi sono riconducibili, principalmente, alle performance ed alla tolleranza ai guasti.

Nel corso degli anni, la diffusione e adozione di sistemi distribuiti è aumentata notevolmente, anche grazie alla facilità della loro realizzazione grazie ai servizi cloud; di conseguenza, sono stati sviluppati e perfezionati anche i paradigmi di programmazione e gli stili architetturali per sfruttarli appieno. Tra questi, quella che attualmente risulta essere la più innovativa, è l'architettura a microservizi.

1.6.1 I problemi del software monolitico

Originariamente, le applicazioni erano sviluppate organizzandole in un'unica unità di deployment; tutta la logica applicativa veniva cioè concentrata in un singolo processo applicativo, che non poteva essere separato.

Un'architettura di questo genere si è rivelata molto problematica, sia dal punto di vista dello sviluppo, che dal punto di vista del deployment e della scalabilità [121, cap. 1.1.3]. Infatti, aggiungendo funzionalità all'applicativo si finisce, inevitabilmente, con l'aumentarne la sua complessità, incrementando il numero di dipendenze all'interno del codice. Tale complessità si ripercuote sulla facilità di sviluppo, dato che il programmatore dovrà conoscere l'intero sistema per intervenire sul codice. Nel caso non lo conoscesse, rischierebbe di realizzare un codice di bassa qualità, aumentando ulteriormente la difficoltà di intervento per eventuali modifiche successive.

Inoltre il deployment risulta essere lento e difficoltoso. Infatti, ogni volta che vengono aggiunte determinate funzionalità o modifiche in alcune funzioni dell'applicativo, è necessario effettuare nuovamente il deployment dell'intero sistema, poiché questo non è divisibile.

Lo scaling dell'applicazione risulta, inoltre, difficoltoso ed inefficiente. Infatti, nel caso una singola istanza dell'applicazione non fosse sufficiente a soddisfare le richieste, sarebbe necessario aggiungerne altre. Ad esempio, si potrebbe effettuare il deployment di più istanze su più nodi diversi ed utilizzare un load balancer per bilanciare il carico fra di loro. Sebbene questa modalità possa funzionare, esistono alcuni problemi. Innanzitutto, le varie istanze devono essere programmate in modo tale da supportare la replicazione, per ogni funzione dell'applicazione. Inoltre, non è possibile effettuare lo scaling di singole funzionalità. Ad esempio, se solo una delle funzionalità dell'applicazione non avesse le performance necessarie per soddisfare le richieste, sarebbe comunque necessario effettuare lo scaling dell'intera applicazione, incrementando anche il numero delle altre funzionalità che offrivano già prestazioni accettabili.

Un ulteriore problema riguarda infine le tecnologie utilizzate. Se il monolite è stato utilizzato sfruttando determinate librerie, framework o linguaggi di programmazione, sarà molto difficile cambiare la tecnologia utilizzata, poiché ogni funzionalità potrebbe avere dipendenze con esse. Se invece, le varie funzionalità fossero state separate e rese indipendenti, sarebbe possibile sfruttare tecnologie differenti per ognuna di esse.

1.6.2 Caratteristiche dell'architettura a microservizi

L'idea alla base dell'architettura a microservizi è quella di scomporre l'applicazione in più microservizi indipendenti, ognuno con una singola responsabilità ben definita. I vari microservizi coopereranno tra loro per realizzare la logica applicativa. Tale cooperazione avverrà sfruttando una rete di comunicazione, attraverso protocolli standard. La decomposizione viene effettuata sulla base delle funzionalità di business, andando a decomporre il dominio del problema in sottodomini quasi indipendenti.

L'architettura a microservizi risolve gran parte delle problematiche delle architetture monolitiche. Infatti, suddividendo l'applicazione in parti distinte, debolmente accoppiate tra loro, si semplificano le operazioni di sviluppo e di deployment. Gruppi di sviluppatori distinti possono infatti operare indipendentemente sul proprio microservizio, ottenendo un maggior controllo sulla complessità, e possono effettuare il deployment indipendentemente dagli altri microservizi. Anche le operazioni di scaling sono più efficienti: se una specifica funzionalità dell'applicazione ha necessità di prestazioni maggiori, potranno essere aggiunte unità relative esclusivamente a quella funzionalità. Inoltre, le tecnologie utilizzate per implementare ogni singolo microservizio potranno essere differenti, in base alle necessità dello specifico microservizio.

I principi alla base dell'architettura a microservizi sono due [135, pag. 17]:

Responsabilità singola Ogni singola unità, in questo caso microservizio, deve possedere una ed una sola responsabilità. Non deve verificarsi il caso in cui più unità condividono le stesse responsabilità o un'unità possiede più di una responsabilità [135, pag. 18].

Autonomia Ogni microservizio deve essere fornito in un pacchetto software contenente tutte le dipendenze, tra cui le librerie, e l'ambiente di esecuzione, che potrebbe consistere, ad esempio, nel web server utilizzato o nel sistema operativo necessario. Questo permette al microservizio di avere un deployment indipendente, sia dagli altri microservizi che dall'ambiente di esecuzione, e di avere l'autonomia per implementare ed eseguire una particolare funzionalità di business [135, pag. 18].

Ogni microservizio dovrà inoltre possedere le seguenti caratteristiche [135, pag. 20]:

Service contract Ogni microservizio deve specificare il proprio contratto agli utilizzatori, che definisce quale servizio offre e quali sono le modalità per usufruirne.

Loose coupling I microservizi devono essere debolmente accoppiati tra loro; le interazioni devono avvenire esclusivamente attraverso le API che il microservizio offre, nascondendo i dettagli implementativi. L'implementazione

di ogni microservizio potrà quindi essere modificata in maniera trasparente agli utilizzatori del servizio offerto.

Service abstraction Il microservizio è l'entità di astrazione di base, che incapsula, oltre alla logica applicativa, anche le librerie e l'ambiente di esecuzione.

Service reuse Ogni microservizio deve essere realizzato nell'ottica della riutilizzabilità. Un microservizio dovrebbe quindi offrire, quando possibile, un servizio generico, riutilizzabile da altri microservizi.

Statelessness Ogni microservizio non dovrebbe mantenere uno stato interno, né uno stato relativo alla comunicazione con gli utilizzatori del servizio. Nel caso in cui sia necessario mantenere uno stato, questo deve essere mantenuto da un servizio di database esterno. L'assenza di stato facilita lo scaling e la tolleranza ai guasti, poiché permette di eliminare o aggiungere una qualsiasi istanza del microservizio senza il rischio di perdere dati o di interferire con le comunicazioni esistenti.

Discoverability Ogni microservizio dovrebbe avere la possibilità di annunciare la propria presenza a chi potrebbe avere la necessità di usufruire del servizio offerto.

Interoperability Ogni microservizio dovrebbe essere interoperabile con gli altri, attraverso l'utilizzo di protocolli e messaggi standard. Fra questi, HTTP, REST, JSON o gRPC.

Composeability La possibilità di realizzare microservizi componendo i servizi offerti da altri microservizi.

Qualora si riuscisse a soddisfare i principi e le caratteristiche appena descritte, sarebbe possibile realizzare un'applicazione organizzata in modo tale da poter essere facilmente eseguita su un sistema distribuito. Infatti, ogni microservizio potrebbe essere eseguito su un nodo fisico differente e replicato in base alle performance necessarie. Per distribuire il carico tra un gruppo di microservizi replicati su potrebbe adottare un load balancer. Inoltre, poiché i vari microservizi sono stateless, risulta semplice scalare le performance o fare fronte ai guasti. Infatti, se un nodo ospitante alcune istanze si disconnettesse, non verrebbero persi dati, e le istanze non più in esecuzione potrebbero essere facilmente sostituite. L'unica eccezione riguarda servizi di database, che, per la loro natura, non possono essere stateless. Tali servizi però adottano particolari tecniche di replicazione per garantire la costante disponibilità dei dati, anche a fronte di guasti. Per questo, i microservizi che necessitano di mantenere uno stato, dovrebbero delegare la sua gestione ad un servizio di database esterno.

I microservizi offrono molti vantaggi, ma presentano anche alcuni aspetti negativi [121, pag. 17]. Una difficoltà evidente consiste nel determinare in quale modo

suddividere l'applicazione, cioè quali microservizi individuare. Questa scelta non è banale, e non esiste un metodo ben definito e sempre efficace per attuarla.

Inoltre, i sistemi distribuiti sono molto più complessi dei sistemi centralizzati. Gli sviluppatori devono infatti gestire le interazioni tra i vari microservizi, considerando che un determinato microservizio potrebbe non essere sempre disponibile. Ogni microservizio inoltre ha il proprio servizio di database e, ciò, potrebbe causare temporanee inconsistenze dei dati. Inoltre, il deployment e la gestione di un numero molto elevato di microservizi richiede, inevitabilmente, un ambiente di orchestrazione, che deve essere installato e configurato.

Per questo, l'adozione di un'architettura a microservizi non è sempre la scelta migliore; è consigliabile adottarla solamente in quei casi in cui l'applicazione è caratterizzata da un'elevata complessità ed ha necessità di elevate prestazioni e garanzie di tolleranza ai guasti.

1.6.3 Deployment dei microservizi

Effettuare il deployment di un'applicazione monolitica è relativamente semplice, poiché è necessario effettuare il deployment di una singola applicazione che sfrutta un singolo framework ed un singolo linguaggio di programmazione. Nel caso dei microservizi potrebbe essere invece necessario effettuare il deployment di decine o centinaia di applicazioni distinte, ognuna potenzialmente scritta tramite framework e linguaggi di programmazione differenti [121, pag. 26]. Sarebbe improponibile effettuare manualmente il deployment di un numero così elevato di applicazioni, senza considerare i possibili conflitti, fra le librerie ed i framework utilizzati, che potrebbero manifestarsi se più microservizi distinti dovessero essere eseguiti sullo stesso nodo. Per risolvere questi problemi, ogni microservizio dovrebbe incapsulare, oltre alla logica applicativa, anche tutte le dipendenze, tra cui le librerie, e l'ambiente di esecuzione, come già affermato nella Sezione 1.6.2. Inoltre, per evitare, su uno stesso nodo, conflitti tra le diverse tecnologie utilizzate dai vari microservizi, sarebbe necessario un qualche sistema di isolamento.

Una possibile soluzione sarebbe quella di utilizzare una macchina virtuale per ogni microservizio, ma, in tal caso, si manifesterebbe un elevato spreco di risorse, causato dall'overhead introdotto dai sistemi operativi di ogni macchina.

La tecnologia più adatta e diffusa per il deployment di microservizi risulta essere quella dei container. I container, rispetto alle macchine virtuali, introducono molto meno overhead, ma garantiscono comunque un elevato grado di isolamento. Per effettuare il deployment di un container su un nodo, è necessaria la presenza di un container runtime engine, come Docker, containerd o LXC. Per effettuare il deployment dei vari microservizi si potrebbe quindi eseguire il deployment dei vari container sui vari nodi provvisti di un container runtime engine. L'utilizzo dei container permetterebbe, sicuramente, di risolvere il problema dell'installazione e

configurazione dell'ambiente di esecuzione, ma, la gestione manuale di ogni container su ogni singolo nodo risulterebbe comunque onerosa. Per questo sono nati i cosiddetti orchestratori di container, come Kubernetes, già introdotti nella Sezione 1.4.2. In generale, questi orchestratori permettono di gestire automaticamente i vari container sui vari nodi del cluster ed offrono servizi aggiuntivi, facilitando notevolmente il deployment di applicazioni composte da microservizi. Gli orchestratori di container saranno descritti in maniera più approfondita nel Capitolo 5. La possibilità di richiedere un cluster preconfigurato con un orchestratore di container attraverso i servizi cloud, come già descritto nella Sezione 1.5, ha reso il deployment delle applicazioni a microservizi ulteriormente più semplice ed accessibile, incrementandone la diffusione.

Capitolo 2

Hardware in ambito server e cloud

In questo capitolo verranno descritte le due famiglie di processori più diffuse sul mercato: i processori con architettura x86-64 ed i processori con architettura Arm. Tali architetture verranno quindi messe a confronto, cercando di determinare i rispettivi pregi e difetti.

Si tenterà inoltre di determinare quali sono i fattori che stanno progressivamente portando all'adozione dell'architettura Arm anche in ambito desktop e server.

Verranno inoltre introdotte le tecnologie che stanno rendendo le piattaforme Arm, in termini di compatibilità software con i vari sistemi operativi, sempre più simili alle piattaforme x86-64. Tali tecnologie sono riconducibili agli standard definiti da Arm, sotto il nome di Arm SystemReady, che prevede l'utilizzo di UEFI ed ACPI.

2.1 Instruction set architectures

Come già affermato nella Sezione 1.1, i processori single-core hanno incontrato dei limiti per quanto riguarda l'aumento delle performance e, per questo, si è passati a processori multi-core. Attualmente, i processori multi-core hanno ormai soppiantato completamente i processori single-core nella maggior parte degli ambiti, come quello desktop, server, mobile e, perfino, in alcuni casi, IoT (Internet of Things).

Le prestazioni e le funzionalità di un processore non dipendono però solamente dalla sua architettura parallela, ma anche da altre caratteristiche, legate alla sua organizzazione interna, alle modalità con cui opera ed al tipo di istruzioni che mette a disposizione.

In generale, i processori possono essere classificati in base alla loro ISA (Instruction Set Architecture), cioè, un modello astratto[53] che definisce le istruzioni supportate ed i modi di operare del processore. L'ISA può essere considerata come un'interfaccia tra il processore ed il software: un software realizzato per un determinato ISA, potrà essere eseguito da un qualsiasi processore che rispetta tale ISA[32]. Nel caso invece, lo stesso

software, dovesse essere eseguito da processori con ISA diverso, sarebbe necessario convertirlo per la differente architettura. Per risolvere questo problema, solitamente il software viene scritto con linguaggi di più alto livello, e, attraverso un processo di compilazione per una determinata ISA, trasformato in un file binario eseguibile.

Spesso le ISA sono abbinate ai termini 32 bit o 64 bit; termini che indicano la capacità, in bit, dei registri utilizzati dal processore, solitamente di quelli che mantengono gli indirizzi relativi alla memoria centrale. Le CPU a 32 bit sono ormai obsolete, ed utilizzate solo per particolari contesti relativi a sistemi embedded; nonostante ciò, alcune delle ISA a 64 bit continuano a mantenere la retrocompatibilità per versioni precedenti della stessa ISA a 32 bit.

Nel corso degli anni sono state definite innumerevoli ISA. In generale sono classificabili in due grandi categorie: CISC e RISC

2.1.1 CISC (Complex Instruction Set Computer)

Le ISA di questa categoria definiscono un numero elevato di istruzioni complesse, termine con cui ci si riferisce ad un'istruzione, che, quando eseguita, comporta l'esecuzione di un numero elevato di operazioni a basso livello da parte del processore [33]. Il processore dovrebbe quindi fornire un elevato numero di istruzioni di alto livello, che verrebbero poi scomposte dall'hardware in una serie di operazioni più semplici; tale decomposizione avverrebbe attraverso un livello di microcodice, cioè un livello di interpretazione capace di tradurre le istruzioni di alto livello in una serie di operazioni eseguibili direttamente dalla CPU [88]. Tale traduzione viene effettuata direttamente dall'hardware e la sua efficienza può essere massimizzata direttamente dal produttore e consentire così di raggiungere elevate prestazioni. In compenso, però, deve essere presente, all'interno del processore, il circuito capace di effettuare tale traduzione; tale circuito può essere notevolmente complesso e, naturalmente, richiede energia per poter operare. Per questo i processori di questa categoria non riescono a raggiungere un'elevata efficienza in termini energetici [132]. Attualmente, l'ISA più diffusa di questa categoria è l'architettura x86, a 32 bit, e la sua evoluzione x86-64, a 64 bit.

2.1.2 RISC (Reduced Instruction Set Computer)

Le ISA di questa categoria definiscono un numero ridotto di istruzioni semplici. Tali istruzioni corrispondono ad operazioni base che il processore può direttamente eseguire e, solitamente, possono essere compiute in un singolo ciclo di clock [120]. Il livello di interpretazione, presente nelle architetture CISC, non è quindi presente, e ciò permette di ottenere un'efficienza energetica maggiore [132]. Le prestazioni offerte da queste architetture non si discostano molto dalle architetture CISC, ma, in compenso, permettono di realizzare processori molto meno esigenti dal punto di vista energetico.

Per questo, sono adatti per tutti quei contesti dove l'efficienza energetica è di centrale importanza, come in ambito mobile o sistemi embedded. Attualmente, l'ISA più diffusa di questa categoria è l'architettura Arm, nelle sue varianti a 32 bit, AArch32 e 64 bit, AArch64, anche nota come arm64.

2.2 x86-64 e Arm

Attualmente, il mercato delle CPU è dominato da due architetture: l'architettura x86-64 e l'architettura Arm. I principali produttori di processori x86-64 sono, attualmente, solamente due, AMD ed Intel. I produttori di processori Arm sono invece molto più numerosi, fra questi, Broadcom, Texas Instruments, Qualcomm, Samsung, Apple e molti altri [9].

2.2.1 Diffusione di x86-64

L'architettura x86-64 venne introdotta, per la prima volta nel 2003, come evoluzione a 64 bit dell'architettura x86, introdotta nel 1978 [139]. La specifica iniziale di x86, nelle varianti a 16 e 32 bit, fu definita da Intel; nel 2003, AMD rilasciò un'architettura compatibile con x86 ma con estensioni a 64 bit, denominata AMD64. Tale architettura ebbe molto successo e costrinse Intel ad adottarla nei propri processori [139]. I processori x86 sono sempre stati realizzati da questi due produttori, Intel e AMD. Intel realizzò la specifica iniziale dell'ISA x86 attraverso i propri processori; fu poi AMD a copiare tale ISA con una propria implementazione compatibile. Accadde invece il contrario nella definizione dell'ISA x86 a 64 bit: fu AMD la prima a realizzarla ed Intel fu costretta, a sua volta, ad implementarla. Questi produttori hanno il completo controllo del mercato di questo tipo di processori, poiché sono loro stessi a progettarli ed a realizzarli nelle proprie fabbriche, come frutto di vari decenni di ricerca e sviluppo.

Nel corso degli anni, l'architettura x86 e, successivamente x86-64, divenne sempre più diffusa in ambito desktop [139]. Poiché la gran parte degli sviluppatori possedeva una macchina x86, il software iniziò ad essere sviluppato quasi esclusivamente per quell'architettura. Questo causò la diffusione di x86 e x86-64 anche in ambito server e, più recentemente nel cloud computing [139]. L'architettura x86 non divenne quindi la tecnologia predominante in ambito server per le sue caratteristiche tecniche. Negli anni Novanta, infatti esistevano altre architetture altrettanto prestanti, come PowerPC, che potevano offrire prestazioni alla pari, se non superiori, di x86. Fu, invece, la vasta diffusione di x86 in ambito desktop, la presenza di una vasta comunità di sviluppatori e di software compatibile [138] a permettere ad Intel di migliorare costantemente i propri prodotti e di ridurre i costi, rendendo x86 l'architettura più conveniente per l'ambito server [46].

Attualmente, il mercato dei PC desktop e server è dominato dall'architettura x86-64. La totalità dei sistemi operativi, sia server che desktop, come Linux e Windows, e dei software in circolazione, supportano, come prima scelta, questa architettura. La stessa situazione si verifica in ambito cloud: ad esempio, Google Cloud Platform, che è uno dei maggiori cloud provider, fornisce, attualmente, solamente macchine con architettura x86-64 [37].

2.2.2 Diffusione di Arm

L'architettura Arm, a differenza di x86, non è frutto della realizzazione di processori da parte di un particolare produttore, come nel caso di Intel. Al contrario, Arm Ltd, cioè l'azienda che progetta queste architetture, si limita a venderne le licenze d'utilizzo, a chiunque volesse realizzare il proprio processore Arm [9]. Arm Ltd, quindi, non gestisce la produzione dei processori Arm, e non vende prodotti finiti. L'acquirente, inoltre, può scegliere fra più pacchetti di licenze, per poter acquistare solamente le componenti di cui necessita: ad esempio, se intendesse realizzare un processore personalizzato che rispettasse l'ISA di Arm, potrebbe limitarsi ad acquistare l'ISA; se invece non volesse realizzare la propria implementazione, potrebbe acquistare dei design realizzati da Arm, come le CPU Cortex o le GPU Mali. Successivamente all'acquisto della licenza, il produttore potrà quindi personalizzare il sistema che vuole realizzare, aggiungendo o rimuovendo componenti per adattarlo alle necessità [10].

L'architettura Arm, come x86, non è recente: venne introdotta nel 1985 [9]. In quegli anni però, come descritto nella Sezione 2.2.1, prese il sopravvento l'architettura x86, per la diffusa convinzione, nei primi anni 2000, che con il passaggio dei PC Apple ad Intel, l'era dei processori non x86, in ambito desktop, fosse finita [10]. Nel 1990 furono venduti circa 20 milioni di personal computer, e la produzione aumentò del 15% ogni anno fino al 2011, quando si raggiunsero i 365 milioni [10] e ognuno di questi PC possedeva un processore con architettura x86. I processori Arm erano comunque presenti sul mercato, ad esempio nei cellulari[16], ma non erano in competizione con i personal computer.

La situazione iniziò a mutare dal 2008, con l'introduzione di iPhone da parte di Apple. IPhone possedeva un processore single-core Arm 32-bit @ 412MHz. Fino al 2008, l'unica modalità per accedere ai servizi digitali era attraverso un personal computer; con l'introduzione di iPhone si passò all'era degli smartphone.

Negli anni successivi, diversi altri produttori di cellulari realizzarono i propri smartphone, sempre basati su processori Arm, considerando la necessità di efficienza dal punto di vista energetico. Gli smartphone iniziarono a diffondersi sempre più, eliminando, per la maggior parte degli utenti, la necessità di possedere un personal computer. La crescente diffusione di questi dispositivi mobili incrementò le vendite, scatenando un processo di miglioramento continuo da parte dei produttori. Tale miglioramento, negli ultimi anni, ha avuto un'accelerazione notevole, permettendo ai

processori Arm di superare, sotto alcuni aspetti, i processori di Intel e AMD. Ad esempio, il processore M1 di Apple è già stato realizzato con un processo di produzione a 5 nanometri, mentre Intel produce ancora processori a 14 nanometri. Di conseguenza, i processori Arm hanno raggiunto prestazioni paragonabili, se non superiori, ai processori desktop x86, garantendo minori consumi e costi inferiori.

Attualmente Arm domina solamente il mercato mobile ed embedded, costituito, ad esempio, da smartphone, tablet, smartwatch, smart tv o dispositivi di rete. Di questo passo, però, l'architettura Arm potrebbe diventare una valida alternativa anche in ambito desktop e server. I processori Arm possono infatti offrire un migliore rapporto consumi prestazioni rispetto a x86 e, vista la loro crescente diffusione, si assisterà probabilmente ad una costante diminuzione dei costi. Inoltre, l'architettura Arm può essere personalizzata dai produttori, permettendo di realizzare processori con componenti specifici per determinate elaborazioni. Tale possibilità ricade sotto il termine di heterogeneous computing, che indica la possibilità di realizzare un sistema composto da più core eterogenei, specifici per determinati compiti. Ad esempio, inserire nella CPU core ad alte prestazioni e core a risparmio energetico, per migliorare l'efficienza, o aggiungere core specifici per le operazioni di machine learning. Questa possibilità permetterebbe di superare di gran lunga, in determinati contesti, le prestazioni di x86.

Costi inferiori ed un miglior rapporto consumi prestazioni potrebbero giustificare la diffusione di Arm, nel prossimo futuro, anche in ambito desktop e server. Attualmente, server e desktop basati sull'architettura Arm sono già disponibili in commercio, ma non sono ancora minimamente diffusi quanto l'architettura x86. Anche alcuni cloud provider stanno iniziando a fornire macchine Arm, come AWS Graviton [101], ma questo tipo di soluzioni costituiscono comunque una minoranza.

La possibile diffusione di Arm in ambito desktop e server non sarà determinata, esclusivamente, dalla disponibilità hardware, ma anche dalla compatibilità software, che dovrà seguire questa tendenza. Infatti, se i sistemi operativi ed i software server non offriranno il supporto per Arm, si estingueranno i vantaggi appena descritti.

Il fatto che Arm sia un ISA completamente diverso da x86, comporta la necessità, per tutto lo stack del software, partendo dal firmware, il sistema operativo, fino ad arrivare alle singole applicazioni, di essere compatibili con tale architettura. Un'incompatibilità, anche parziale, che potrebbe rendere il sistema instabile, o determinare anche una minima deviazione del comportamento dello stesso software rispetto a x86, scoraggerebbe l'adozione di questa architettura. Per questo, il sistema operativo dovrà essere capace di sfruttare correttamente, attraverso i propri driver e tutto il software di basso livello, l'hardware del sistema Arm; sarebbe inoltre ideale riuscire ad utilizzare un generico sistema operativo, senza dover applicare correzioni o aggiungere componenti software per renderlo compatibile con la specifica piattaforma Arm. Per risolvere tale problema, Arm ha delineato un insieme di standard, sotto il

nome di Arm SystemReady, che, se rispettati dai produttori di sistemi Arm e riconosciuti dai vari sistemi operativi, garantiscono la compatibilità, nativa, tra i due.

2.3 Compatibilità tra hardware Arm e sistemi operativi

Come già affermato nella Sezione 2.2.2, i sistemi Arm possono essere personalizzati dai singoli produttori, che possono aggiungere o rimuovere componenti hardware a piacere. Solitamente, tali componenti, che potrebbero essere, ad esempio, i core della CPU, controller USB, interfacce UART, schede di rete, non sono facilmente individuabili dal sistema operativo, almeno per quanto riguarda i sistemi Arm [136], senza adottare strategie apposite.

2.3.1 Storia

Fino al 2010, per permettere ai sistemi operativi Linux di utilizzare l'hardware della piattaforma Arm, veniva ricompilato il kernel, personalizzandolo per lo specifico sistema, aggiungendo i driver necessari. Il kernel è il componente software principale di un sistema operativo che ha il controllo completo del sistema, gestendo direttamente l'hardware e fungendo da intermediario con le applicazioni; un device driver è un componente software che contiene la logica per interagire con un particolare dispositivo hardware ed è utilizzato dal kernel. La definizione di un kernel personalizzato per ogni sistema Arm comportò alcuni problemi: primo fra tutti, la proliferazione incontrollata di driver. Solitamente infatti, ogni produttore, per adattare il kernel, realizzava la propria versione degli stessi driver.

Per risolvere tale problema, nel 2010, venne introdotta una nuova soluzione, il devicetree. L'idea alla base del devicetree è quella di scorporare dal kernel la logica per renderlo compatibile con una particolare configurazione hardware. In particolare, il devicetree è una struttura dati che descrive i componenti hardware di un particolare sistema, in modo tale che il kernel possa utilizzarli e gestirli [38]. In questo modo, per rendere il kernel compatibile con un particolare sistema, non sarà più necessario ricompilarlo, ma sarà necessario solamente adattare il devicetree alla diversa configurazione. Il kernel, al momento del boot del sistema, andrà a leggere le informazioni inerenti al devicetree, determinando i dispositivi hardware disponibili ed i driver da utilizzare. Questa soluzione prevede, comunque, che il driver per un determinato dispositivo sia già presente nel kernel; il vantaggio è che offre l'opportunità di specificare un insieme di configurazioni per consentire al driver di operare su una particolare configurazione hardware, permettendo di rimuovere dal kernel un insieme di file sorgente e di impostazioni di compilazione specifici per quel sistema [38]. Il devicetree viene fornito al kernel dal bootloader, cioè il primo software avviato durante il boot, che si occupa di inizializzare l'hardware e successivamente di

passare il controllo del sistema al kernel. Il bootloader più diffuso per i sistemi Arm è U-Boot, che è capace di fornire al kernel, nel momento in cui gli passa il controllo, il devicetree.

2.3.2 Arm SystemReady

Dal 2012, iniziarono ad essere commercializzati i primi sistemi basati su arm64 [2]. Nel caso di arm64, soprattutto in ambito server, si optò per una direzione differente. Arm decise di definire un insieme di standard, che, se rispettati, potessero permettere ad un generico sistema operativo di essere avviato, senza particolari modifiche o configurazioni, sul sistema Arm, garantendo una condizione di intercompatibilità simile a quella già presente per i sistemi x86. Nonostante in ambiente Arm fosse già diffuso l'utilizzo di U-Boot e del devicetree, Arm si indirizzò verso l'adozione di due differenti standard: UEFI per quanto riguarda il firmware e ACPI per quanto riguarda la descrizione delle componenti hardware, al posto del devicetree. Inizialmente tali standard erano stati definiti esclusivamente per i sistemi server, con il nome di Arm ServerReady. Dal 2020, la definizione di tali standard è stata estesa anche a dispositivi di altre categorie, come sistemi IoT o embedded-server, con il nome di Arm SystemReady. Gli standard da rispettare sono stati raggruppati in due categorie:

Base System Architecture (BSA) Specifica i requisiti hardware minimi necessari per il deployment di un sistema operativo. Il software di sistema, cioè, ad esempio, i sistemi operativi, gli hypervisor ed il firmware potranno quindi basarsi su un hardware con caratteristiche standardizzate. Oltre al BSA base, è stata definita un'estensione apposita per i sistemi di tipo server: Server Base System Architecture (SBSA).

Base Boot Requirements (BBR) Specifica i requisiti da rispettare dal punto di vista del firmware, cioè di tutto quel software, realizzato dai produttori di board Arm, che si interpone tra l'hardware ed il sistema operativo. Questi standard sono inerenti a UEFI ed ACPI, descritti, rispettivamente, nella Sezione 2.4 e nella Sezione 2.5. In base al tipo di sistema, i requisiti BBR sono stati suddivisi in quattro sottocategorie:

SBBR (Server Base Boot Requirements) Definisce i requisiti che devono essere soddisfatti dall'hardware di categoria server. Impone di utilizzare UEFI ed ACPI, vietando esplicitamente l'uso di devicetree. Attualmente, Windows Server, Windows 10, Red Hat Enterprise Linux (RHEL), Amazon Linux e Oracle Linux richiedono SBBR. Altri sistemi operativi, come, ad esempio, VMware ESXi, SUSE Linux Enterprise Server (SLES), CentOS, Ubuntu, Kylin OS, NetBSD e FreeBSD supportano SBBR.

L'implementazione di riferimento è chiamata EDK2, fornita da TianoCore; in ogni caso, SBBR non impone EDK2 [15, cap. 4.1].

ESBBR Come SBBR ma con possibili eccezioni [15, cap. 4.2]. Al momento nessuna eccezione è stata ancora specificata.

EBBR (Embedded Base Boot Requirements) Definisce un insieme di requisiti ridotti rispetto a SBBR. Adatto per tutti quei dispositivi che, per limitazioni hardware o di altro genere, non possono soddisfare per intero SBBR. EBBR richiede un ambiente UEFI ridotto, e permette l'utilizzo di devicetree, a patto che solamente uno dei due, tra ACPI e devicetree, sia utilizzato [43]. Attualmente, Fedora, Debian, openSUSE, SLES e Ubuntu supportano EBBR. Solitamente, l'implementazione di EBBR viene realizzata attraverso U-Boot; in ogni caso, un'implementazione basata su EDK2 non è vietata [15, cap. 4.3].

LBBR Definisce un insieme di requisiti per quei sistemi basati su LinuxBoot. LinuxBoot è un firmware alternativo basato sul kernel Linux. LinuxBoot è utilizzato da alcuni datacenter che adottano il modello Open Compute Project (OCP) [15, cap. 4.4].

Gli standard Arm SystemReady individuano quindi quattro categorie di certificazioni, ottenute combinando le specifiche BSA e BBR:

SystemReady SR Arm SystemReady SR certifica che il server funzionerà immediatamente, senza necessità di configurazioni o modifiche, garantendo un'interoperabilità nativa con sistemi operativi, hypervisor e software. I sistemi conformi a SystemReady SR devono rispettare BSA, SBSA e SBBR [14].

SystemReady ES Arm SystemReady ES (Embedded ServerReady) certifica che il sistema implementa un insieme minimo di funzionalità hardware e software, sulle quali, un eventuale sistema operativo o hypervisor, potrà fare affidamento. I sistemi conformi a SystemReady ES devono rispettare BSA e SBBR [11].

SystemReady IR Arm SystemReady IR (IoTReady) è una certificazione per dispositivi IoT basati su system on chip con architettura Arm A-profile. Assicura l'interoperabilità con Linux e altri sistemi operativi per dispositivi embedded. I sistemi conformi a SystemReady IR devono rispettare BSA ed EBBR [12].

SystemReady LS Arm SystemReady LS (LinuxBoot ServerReady) è una certificazione che assicura che la piattaforma hardware sia adatta per lo sviluppo sullo stack LinuxBoot. I sistemi conformi a SystemReady LS devono rispettare BSA ed LBBR [13].

2.4 UEFI

Nella Sezione 2.3 sono stati introdotti gli standard proposti da Arm per garantire l'intercompatibilità tra la piattaforma hardware ed un qualsiasi sistema operativo. Per garantire ciò è necessario adottare uno standard dal punto di vista del firmware della piattaforma. Uno standard già notevolmente diffuso in questo ambito, che è stato scelto da Arm per i propri sistemi, è UEFI.

2.4.1 Storia

La specifica UEFI è l'evoluzione di EFI. La specifica EFI (Extensible Firmware Interface) venne introdotta, per la prima volta, da parte di Intel, nella seconda metà degli anni 90, per la realizzazione dei primi sistemi con processore Itanium. Prima dell'introduzione di EFI, i sistemi adottavano il cosiddetto legacy BIOS, che era però caratterizzato da varie limitazioni, non adatte per i nuovi processori Itanium. Per risolvere tali limitazioni, Intel, nel 1998, iniziò lo sviluppo di una nuova specifica, denominata Intel Boot Initiative, che diventò poi la specifica EFI [133]. L'obiettivo di EFI era quello di definire un'interfaccia più moderna, rispetto al BIOS, con un sistema di driver modulari che premettesse di aggiungere e rimuovere componenti con facilità [124, pag. 121]. La specifica EFI venne adottata da vari produttori leader, evolvendosi, nel 2005, nella specifica UEFI (Unified Extensible Firmware Interface). La specifica UEFI è mantenuta dallo Unified EFI Forum, ma la proprietà della specifica originale EFI rimane, comunque, ad Intel. Dal 2005 in poi, la specifica UEFI si è evoluta, aggiungendo nuove funzionalità. Parallelamente alla sua evoluzione, venne realizzato EDK (EFI Development Kit), cioè un insieme di strumenti di sviluppo open-source, per aiutare gli sviluppatori nella realizzazione firmware basati su UEFI. EDK si è poi evoluto in EDK2, che, attualmente, è mantenuto dalla comunità di TianoCore [133].

2.4.2 Definizione

La specifica UEFI (Unified Extensible Firmware Interface) descrive un'interfaccia tra il sistema operativo (OS) ed il firmware della piattaforma hardware [134, pag. 1]. Tale interfaccia è fornita sotto forma di tabelle, che contengono informazioni relative alla piattaforma hardware, di servizi di boot, utilizzabili dal bootloader del sistema operativo e da servizi di runtime, utilizzabili direttamente dal sistema operativo. Tali tabelle e servizi, forniscono un ambiente standardizzato per poter avviare un qualsiasi sistema operativo. La specifica UEFI è solamente un'interfaccia che definisce esclusivamente quali servizi e strutture il firmware della piattaforma hardware dovrà fornire e quali servizi e strutture il sistema operativo potrà sfruttare. La scelta delle modalità con cui il firmware fornirà tali servizi e delle modalità con cui il sistema li sfrutterà, è lasciata interamente agli sviluppatori del firmware e del sistema operativo.

L'obiettivo della specifica UEFI è quello di definire una modalità di interazione, attraverso un'interfaccia standard, che permetta al sistema operativo ed al firmware della piattaforma hardware di scambiarsi esclusivamente le informazioni necessarie per l'avvio del sistema operativo. Se il firmware adotterà tale interfaccia, sarà possibile avviare un sistema operativo compatibile con essa, senza la necessità di ulteriori personalizzazioni del firmware o del sistema operativo. Inoltre, tale interfaccia permette la modifica dell'implementazione del firmware, come, ad esempio, dei driver dei dispositivi, in maniera trasparente, senza influire sul processo di boot.

La specifica UEFI è applicabile ad una vasta gamma di dispositivi, dai sistemi mobile ai server. UEFI definisce un insieme di servizi principali e di protocolli; la selezione di questi ultimi, potrà evolvere nel corso del tempo per essere adattata alle esigenze di una particolare classe di dispositivi. UEFI garantisce inoltre, nei confronti dei produttori, un'elevata possibilità di personalizzazione.

2.4.3 Funzionalità

UEFI è un livello software, interposto tra il livello del firmware e quello del sistema operativo, molto complesso. Infatti, non si limita esclusivamente nell'effettuare l'avvio del sistema operativo: solitamente, un bootloader di primo livello, dopo aver lasciato il completo controllo al bootloader del sistema operativo, avrebbe terminato il suo compito e, quindi, verrebbe rimosso dalla memoria. UEFI è invece un livello software che, oltre a contribuire al processo di boot del sistema tramite dei boot services, rimane in memoria centrale, offrendo un insieme di servizi al sistema operativo in esecuzione, chiamati runtime services. In generale, UEFI è un livello che permette di astrarre notevolmente dall'hardware sottostante, offrendo numerose funzionalità; alcune delle più importanti sono le seguenti:

Boot service I boot service sono disponibili solamente durante il processo di boot, e forniscono delle interfacce per interagire con i vari dispositivi e con determinate funzionalità di sistema [134, pag. 7].

Runtime service I runtime service sono offerti al sistema operativo, assicurando un appropriato livello di astrazione rispetto alla piattaforma hardware, per l'accesso a determinate risorse che potrebbero essere necessarie al sistema operativo durante la sua normale esecuzione [134, pag. 7]. Questi servizi includono, ad esempio, il reperimento della data e del tempo attuale ed il salvataggio di variabili sulla NVRAM (Non-Volatile RAM) [133].

Variabili su NVRAM La NVRAM è una memoria non volatile, presente nella piattaforma hardware, sulla quale possono essere salvati determinati valori, sotto forma di variabili, come coppie chiave e valore; tali valori potranno rimanere disponibili tra riavvi successivi del sistema. Questo servizio permette quindi la

condivisione di informazioni tra il firmware UEFI ed il sistema operativo. Un esempio di informazione salvata sulla NVRAM è l'ordine di boot: un sistema potrebbe avere più sistemi operativi installati su differenti dischi o partizioni. Il boot manager di UEFI deve quindi conoscere quali sono le possibili scelte di boot ed in quale ordine tentare di avviarle. Le scelte di boot vengono salvate in variabili con nome Boot seguito da un numero, come Boot0001 o Boot0002; l'ordine di boot è salvato invece nella variabile BootOrder [50]. La variabile BootOrder può essere modificata sia dal firmware UEFI, sia dal sistema operativo, per definire, al momento dell'avvio della macchina, quale opzione di boot tentare di avviare.

Time services Permette al sistema operativo di impostare l'orario di sistema, sfruttando il real-time clock hardware.

EFI system partition La partizione EFI, chiamata anche ESP, è la partizione che può essere presente su uno o più dischi rigidi collegati al sistema, contenente le applicazioni UEFI ed i file a loro necessari. Tale partizione deve essere in un formato FAT, mentre la tabella di partizionamento può essere in formato MBR o GPT. La possibilità di utilizzare GPT permette di effettuare il boot da dischi con capacità maggiore di due TiB, limite imposto dai bios legacy che permettevano il boot solamente da dischi MBR.

Applicazioni Il firmware UEFI realizza un livello software che permette l'esecuzione di vere e proprie applicazioni, chiamate applicazioni UEFI. Le applicazioni UEFI solitamente, sono file con estensione ".efi", che risiedono nella partizione EFI del disco rigido. Ricadono in questa categoria i bootloader, realizzati per UEFI, per l'avvio dei sistemi operativi. In particolare, il più popolare in ambito Linux è GRUB, mentre per Windows è Windows Boot Manager. I bootloader possono essere individuati automaticamente dal firmware UEFI, che genererà le opzioni di boot di conseguenza. Per permettere ciò, solitamente i boot loader risiedono in un percorso specifico della ESP, cioè "/efi/boot/bootaa64.efi" per le architetture arm64 e "/efi/boot/bootx64.efi" per le architetture x86-64 [133].

Network boot La specifica UEFI comprende il supporto per effettuare il boot da rete di un sistema operativo attraverso il Preboot eXecution Environment (PXE). Per implementare PXE sono necessari vari protocolli, tra cui, il protocollo IP, UDP, DHCP, TFTP e opzionalmente anche HTTP [133]. Tali protocolli sono quindi supportati da UEFI, a patto che siano presenti i driver per poter utilizzare le interfacce di rete.

Device drivers Poiché il firmware UEFI deve fornire servizi di alto livello, dovrà possedere la logica per interagire con l'hardware del sistema, ad esempio con i

bus di sistema, con i controller USB, con le schede di rete o con i dischi rigidi. Tale logica deve essere incapsulata in appositi driver che permettano di interagire con i vari componenti del sistema. I driver scritti per UEFI devono però essere progettati solamente per poter accedere ai dispositivi prima della fase di boot, e non per sostituire i driver ad alte prestazioni che saranno poi presenti nei vari sistemi operativi [134, pag. 1].

UEFI shell UEFI mette a disposizione una shell, utilizzabile per eseguire un insieme di comandi, tra cui, ad esempio, visualizzare i dischi connessi, esplorare le directory ed editare file, modificare configurazioni o utilizzare funzioni di rete, e molte altre. Tale shell è assimilabile ad una shell Linux, ma con comandi specifici aggiuntivi, propri dell'ambiente UEFI. In base al produttore della piattaforma hardware, la shell potrebbe essere già presente tra le opzioni di boot, altrimenti è necessario installarla manualmente caricando il file compilato della shell nella partizione EFI [133].

2.4.4 Compatibilità

La specifica UEFI è compatibile, attualmente, con quasi la totalità delle ISA presenti sul mercato: è compatibile con IA-32, Itanium, x64, AArch32, AArch64 e RISC-V [134, cap. 2.3]. Questo non significa che tutti i produttori di piattaforme hardware adottino UEFI, ma la compatibilità della specifica ne permetterebbe un'implementazione. Attualmente UEFI è adottato dalla totalità di produttori di schede madri x86-64. Per quanto riguarda Arm, UEFI non è diffuso quanto la controparte x86-64, ma potrebbe diventarlo in futuro, almeno, per quanto riguarda l'ambito desktop e server.

Anche dal punto di vista software, la compatibilità con UEFI è molto diffusa: il kernel Linux è compatibile con EFI dagli anni 2000, Mac OS X sfrutta UEFI dalla versione 10.4 e Windows ha introdotto il supporto a UEFI dalla versione Vista SP1 [133]. Anche in ambito virtualizzazione e cloud, UEFI è molto diffuso: VMware Workstation, vSphere ESXi, VirtualBox, QEMU/KVM, Microsoft Hyper-V e Google Cloud Platform, permettono di realizzare macchine guest con un firmware UEFI virtualizzato [133].

2.5 ACPI

Nella Sezione 2.3 sono stati introdotti gli standard proposti da Arm per garantire l'intercompatibilità tra la piattaforma hardware ed un qualsiasi sistema operativo. Per garantire ciò è necessario adottare una modalità standard che permetta al sistema operativo di individuare tutto l'hardware presente sul sistema. Una possibile soluzione,

almeno per quanto riguarda Arm, poteva essere il devicetree. ACPI fornisce un servizio simile a quello fornito dal devicetree, ma non è limitato all'ambiente Arm, essendo già notevolmente diffuso nei sistemi x86-64.

2.5.1 Storia

Prima della comparsa di ACPI, vi fu una proliferazione incontrollata di soluzioni personalizzate, proposte singoli produttori, relative al power management ed alla configurazione ed individuazione dell'hardware di sistema. Ogni soluzione proprietaria richiedeva il supporto specifico del sistema operativo, causando una frammentazione che ha alimentato il desiderio di realizzare una soluzione standardizzata [131].

Intel, Microsoft, Toshiba, HP e Phoenix, decisero quindi di unire le forze per unificare il sistema di power management attraverso lo standard ACPI; la versione 1.0 venne rilasciata nel 1996. ACPI, progressivamente, andò a sostituire sia sistemi di power management utilizzati precedentemente, come Advanced Power Management (APM), sia tecnologie per la configurazione ed individuazione dell'hardware di sistema, come PNPBIOS e MultiProcessor Specification (MPS) [131].

Intel adottò ACPI nei propri sistemi dall'Aprile del 1998, mentre Microsoft aggiunse il supporto ad ACPI in Windows da giugno del 1998 [94, pag. 348]. Nel 2000 venne rilasciata la versione 2.0, che aggiunse il supporto ai processori 64 bit e multi-core. Nel 2004, venne rilasciata la versione 3.0, che aggiunse il supporto alle interfacce SATA ed al bus PCI Express. La versione 4.0, nel 2009, aggiunse il supporto ad USB 3.0, mentre la versione 5.0, nel 2011, lo aggiunse all'architettura Arm [4]. Nel 2013, la specifica ACPI venne presa in carico dall'UEFI Forum, la stessa associazione di imprese che, attualmente, gestisce la specifica UEFI. Questo, per agevolare la sincronizzazione tra le interfacce UEFI ed ACPI, oltre che a favorire la partecipazione di sviluppatori open-source. Venne quindi creato l'ACPI Specification Working Group (ASWG), un gruppo dell'UEFI Forum con il compito di gestire ed evolvere la specifica ACPI [131]. Attualmente, l'ultima versione della specifica ACPI è la 6.4, rilasciata il 22 gennaio 2021.

2.5.2 Definizione

Advanced Configuration and Power Interface (ACPI) è un framework per la gestione del power management e per la configurazione ed individuazione dell'hardware di sistema, indipendente dall'architettura e considerabile come un sottosistema risiedente all'interno del sistema operativo [5, pag. 36]. ACPI definisce dei meccanismi flessibili e standardizzati per l'individuazione dei dispositivi presenti nel sistema, per l'OSPM (Operating System configuration and Power Management), per la gestione delle temperature (thermal management) e per le funzionalità RAS (Reliability, Availability and Supportability) [131]. L'OSPM è un componente, che dovrà essere presente

all'interno del sistema operativo, che si occuperà di tutte le operazioni inerenti alla configurazione ed al power management dei dispositivi, considerando le informazioni ACPI provenienti dal firmware della piattaforma. La specifica ACPI può essere quindi considerata come un'interfaccia, tra il sistema operativo ed il firmware, per tutte quelle operazioni inerenti al power management ed alla configurazione dell'hardware.

2.5.3 Funzionalità

Le funzionalità principali offerte da ACPI sono due:

Individuazione e configurazione dell'hardware ACPI può fornire al sistema operativo la lista dell'hardware presente nel sistema, incluso tutto quell'hardware che non sarebbe direttamente individuabile. Un concetto al centro della specifica ACPI è quello di namespace. L'ACPI namespace è una rappresentazione gerarchica di tutti i dispositivi ACPI presenti nel sistema, la cui radice è il bus di sistema. Il namespace verrà utilizzato dal sistema operativo sia per caricare che per configurare il driver corretto per ogni dispositivo ACPI [5, pag. 42]. Il namespace permette quindi al sistema operativo, di individuare e configurare tutto l'hardware del sistema.

Power Management ACPI, al contrario delle soluzioni basate su BIOS legacy, fornisce il controllo completo del power management al sistema operativo. Quando il sistema operativo è in esecuzione, potrà quindi decidere, autonomamente, di disattivare specifici dispositivi o impostarli in uno stato di risparmio energetico, per massimizzare l'efficienza dal punto di vista del consumo di energia. Ad esempio, per ogni dispositivo del sistema, ACPI definisce quattro stati, da D0 a D3. D0 indica che il dispositivo è completamente operativo e, quindi, il suo consumo di energia sarà massimo. D1 e D2 sono stati intermedi, in cui il dispositivo potrebbe trovarsi in uno stato di risparmio energetico, mentre D3 indica che il dispositivo è spento. La logica per effettuare il passaggio da uno stato all'altro dipende dall'hardware sottostante; ACPI fornisce quindi dei metodi, al sistema operativo, che contengono la logica per operare con quel determinato hardware, che devono essere definiti dal produttore della piattaforma. Ad esempio, il metodo "_PS0" permette di impostare il dispositivo allo stato D0, mentre "_PS3" permette di impostarlo allo stato D3 [5, pag. 467].

La specifica ACPI definisce tre componenti principali [4]:

Tabelle ACPI Le tabelle ACPI contengono tutte le informazioni sull'hardware del sistema. Sono realizzate dal produttore del firmware della piattaforma hardware, e vengono fornite al sistema operativo per informarlo sulla configurazione del sistema.

BIOS ACPI La parte realizzata dal produttore della piattaforma hardware, genera le tabelle ACPI e le carica in memoria centrale, dove potranno essere lette dal sistema operativo.

Registri ACPI I registri ACPI corrispondono a particolari indirizzi, accessibili in modo analogo a quelli relativi alla memoria centrale, che permettono di interagire con l'hardware. Tali indirizzi, potrebbero risiedere in diversi spazi, tra cui, System I/O, System memory, PCI configuration, SMBus, Embedded controller o Functional Fixed Hardware [5, pag. 101]. Manipolando il valore di tali registri, ACPI potrà leggere o modificare lo stato dei dispositivi hardware, per gestire le operazioni di power management o di configurazione.

Tabelle ACPI

Per implementare le funzionalità descritte nella Sezione 2.5.3, devono essere scambiate delle informazioni tra il firmware ed il sottosistema ACPI in esecuzione all'interno del sistema operativo. Tali informazioni vengono scambiate attraverso una serie di tabelle, che descrivono la piattaforma hardware al sistema operativo (OSPM). Il numero di tabelle definito da ACPI è molto elevato, e, solitamente, non tutte sono necessarie. Le tabelle necessarie sono determinate dal sistema operativo e dall'architettura della CPU. Le principali tabelle ACPI sono:

Root System Description Pointer (RSDP) Puntatore alla tabella XSDT o RSDT. Tale puntatore è impostato dal firmware della piattaforma hardware al momento del boot, e permette al sistema operativo di individuare la locazione, in memoria, della tabella XSDT o della tabella RSDT. Solitamente, i sistemi operativi preferiscono la tabella XSDT, che è una evoluzione della tabella RSDT. Nei sistemi UEFI, il puntatore RSDP è ottenuto dalla tabella EFI, fornita dal firmware UEFI [5, pag. 140].

Root System Description Table (RSDT) Tabella che contiene gli indirizzi, in memoria, della maggior parte delle altre tabelle ACPI. RSDT è la prima tabella letta dal sistema operativo per individuare le altre [5, pag. 62]. Tra i vari riferimenti, contiene quello alla tabella FADT.

eXtended System Description Table (XSDT) Fornisce le stesse informazioni della tabella RSDT, ma può contenere indirizzi di dimensione superiore a 32 bit [5, pag. 63].

Fixed ACPI Description Table (FADT) Tabella che fornisce un insieme di valori di lunghezza fissa che descrivono funzionalità statiche dell'hardware relative ad ACPI, tra cui, l'indirizzo in memoria della tabella DSDT. Oltre a tale indirizzo,

contiene gli indirizzi, in memoria, di vari registri standard necessari ad ACPI per operare.

Differentiated System Description Table (DSDT) Tabella che definisce, attraverso una struttura gerarchica, l'insieme dei dispositivi hardware presenti nel sistema. Tale definizione avviene attraverso un insieme di blocchi, chiamati Differentiated Definition Block, che contengono la definizione di vari oggetti. Un Definition Block consiste in un insieme di dati definiti in formato AML (ACPI Machine Language), che contengono informazioni riguardanti l'hardware, sotto forma di oggetti AML contenenti dati, metodi AML o altri oggetti AML. Il sistema operativo utilizzerà tali informazioni per generare il namespace.

AML e ASL

AML è il linguaggio utilizzato per definire i vari Definition Block della tabella DSDT. Solitamente, ogni Definition Block di più alto livello definisce un particolare dispositivo hardware. Tale blocco, avrà, all'interno, ulteriori blocchi, che definiranno ulteriori proprietà del dispositivo; ad esempio, in quale modo interagire, attraverso metodi, con l'hardware, o in quale area di memoria andare ad operare. Un esempio di metodo è "_PS0", che deve definire la logica per permettere di impostare il dispositivo allo stato D0, descritto precedentemente. AML è un linguaggio Turing-completo [4], per cui, parte della logica per interagire con i dispositivi, può essere inserita all'interno della tabella DSDT. Il codice AML, corrispondente alla tabella DSDT, viene generato compilando un linguaggio di più alto livello, ASL (ACPI Source Language). I produttori della piattaforma hardware definiscono, quindi, tutti i dispositivi presenti nel sistema, nella tabella DSDT, attraverso ASL; al momento della compilazione del firmware, ASL verrà tradotto in codice AML.

Poiché AML è codice eseguibile, il sistema operativo dovrà essere in grado di interpretarlo. Per questo, ogni sistema operativo compatibile con ACPI, deve possedere un interprete di codice AML. Il sistema operativo, interpretando la tabella DSTD, genererà il namespace, contenente tutti i dispositivi ACPI connessi al sistema.

Ogni Definition Block relativo ad un dispositivo, contiene, inoltre, una proprietà "_HID", che definisce l'hardware ID. Il sistema operativo enumererà i dispositivi connessi e caricherà il relativo driver utilizzando tale proprietà. La possibilità di caricare automaticamente il driver e configurare il dispositivo ricade sotto il termine di Plug and Play. Inoltre, il driver di sistema per un determinato dispositivo ACPI potrà sfruttare l'interprete AML per eseguire i metodi o accedere ai registri, specificati nella tabella DSDT, relativi a quel dispositivo.

2.5.4 Compatibilità

La specifica ACPI supporta una vasta gamma di piattaforme, inclusi laptop, tablet, smartphone, workstation, desktop e server. La specifica include, inoltre, un primo supporto per dispositivi System-on-Chip con architettura Arm [131]. Dal punto di vista dei produttori di piattaforme hardware, ACPI è molto diffuso: i nuovi sistemi operativi, come, ad esempio, i sistemi operativi Windows dalla versione Vista, richiedono un BIOS che rispetti la specifica ACPI [4]; per questo, i produttori di piattaforme hardware hanno già diffusamente adottato tale tecnologia. Anche il kernel Linux supporta ACPI, già dalla versione 2.6, uscita nel 2003 [4].

Capitolo 3

Bare-metal cloud

Come già accennato nel capitolo introduttivo, uno degli obiettivi di questa trattazione sarà la realizzazione di un cluster basato su un insieme di macchine fisiche. Per riuscire ad automatizzare i processi di installazione del sistema operativo e di deployment del cluster management software, già descritti nella Sezione 1.4.2, sarà necessario adottare delle tecnologie che permettano la realizzazione di un bare-metal cloud privato. Con il termine bare-metal cloud si intende un servizio cloud, che permetta, su richiesta, di ottenere macchine fisiche pronte all'uso; sarà il software del bare-metal cloud a gestire, installare e configurare automaticamente le singole macchine.

In questo capitolo si andranno quindi a delineare le principali caratteristiche e funzionalità di un software che permetta la realizzazione di un bare-metal cloud; saranno quindi individuate le principali alternative attualmente disponibili.

Infine, verranno descritte le modalità di operare e le funzioni messe a disposizione dal software scelto, in questo ambito, per la realizzazione di questo progetto: Canonical Metal As A Service (MAAS).

3.1 Definizione

Un servizio di bare-metal cloud permette di installare o reinstallare un determinato sistema operativo su macchine fisiche, in maniera totalmente automatica e senza interventi manuali sull'hardware.

I bare-metal cloud ricadono nella categoria IaaS (Infrastructure as a Service), poiché permettono di richiedere risorse base, cioè di elaborazione, memoria e rete, sulle quali il cliente potrà eseguire un qualsiasi software. La caratteristica principale che contraddistingue questo tipo di cloud è la natura fisica, e non virtuale, delle risorse richiedibili.

La possibilità di richiedere server fisici, invece di macchine virtuali, porta ad alcuni vantaggi [93, cap. 3], che, in determinati ambiti, potrebbero essere notevolmente importanti:

- la possibilità di eseguire il software direttamente sulla macchina fisica elimina completamente l'overhead legato alla virtualizzazione. Questa caratteristica potrebbe essere molto utile per tutte quelle tipologie di software che hanno esigenze prestazionali molto elevate;
- le tecniche di virtualizzazione mascherano l'hardware fisico con un hardware virtualizzato. In alcuni contesti, le applicazioni potrebbero richiedere di accedere direttamente all'hardware fisico. La possibilità di richiedere una macchina fisica risolve questo problema. In realtà, alcune tecnologie di virtualizzazione permettono di assegnare il completo controllo di un dispositivo fisico ad una macchina virtuale; tale tecnica è chiamata device passthrough, ma l'hardware e l'hypervisor devono supportarla;
- la possibilità di allocare una macchina fisica permette di assegnare l'intero hardware ad una determinata applicazione. Poiché l'hardware non è più condiviso, viene eliminata la possibilità che applicazioni differenti, in esecuzione sulle proprie macchine virtuali, interferiscano tra loro. Per questo, anche l'isolamento tra i vari utenti e le loro applicazioni è migliorato.

Naturalmente, gestire delle macchine fisiche è più complesso rispetto alla gestione di macchine virtuali. Per questo, una soluzione di questo tipo dovrebbe essere adottata solamente per specifici ambiti. Alcuni delle principali problematiche sono:

- impossibilità di migrare una determinata macchina da una locazione all'altra, come invece può avvenire nel contesto delle macchine virtuali, che possono essere spostate facilmente da un host all'altro;
- l'hardware, non essendo virtualizzato, potrebbe presentare alcune differenze tra le varie macchine, provocando, potenzialmente, dei problemi al momento del deployment del software. Inoltre, tale differenza, potrebbe compromettere la possibilità di utilizzare immagini preinstallate;
- salvare lo stato di una macchina fisica, per effettuare operazioni di backup e ripristino, è molto più complesso;
- l'hardware, non essendo condiviso, potrebbe essere non sfruttato appieno.

In realtà, questi ultimi problemi elencati, potrebbero essere attenuati dal software di bare-metal cloud utilizzato, o dal software che verrà installato sulle varie macchine fisiche.

In generale, i software per la realizzazione di un bare-metal cloud prevedono la presenza di uno o più nodi di gestione, denominati controller, che offriranno il servizio di allocazione on demand delle macchine fisiche, gestendo il processo di installazione e configurazione. Solitamente, tali controller, alla richiesta di allocazione di una nuova macchina, operano nella seguente modalità [93, cap. 3]:

1. il controller sceglie una delle possibili macchine, non ancora allocate, che rispetti gli eventuali vincoli della richiesta;

2. il controller avvia un server DHCP (Dynamic Host Configuration Protocol) per fornire un indirizzo IP alla macchina, ed un server TFTP (Trivial File Transfer Protocol) o HTTP per fornirgli i file di boot;

3. la macchina viene avviata, costringendo il firmware ad effettuare un boot da rete PXE (Preboot Execution Environment); questo solitamente avviene attraverso specifiche tecnologie che ne permettono l'avvio e la gestione da remoto;

4. la macchina acquisisce quindi un indirizzo di rete e l'indirizzo del server TFTP, da cui scaricherà i vari file di boot; tali file avvieranno il processo di installazione del sistema operativo su uno dei dischi fisici;

5. la macchina viene dunque riavviata, avviando il sistema operativo appena installato;

6. infine, viene avviato il processo di configurazione del sistema operativo, attraverso degli strumenti di configurazione automatica.

3.2 Principali alternative

Attualmente esistono vari software che permettono la realizzazione di bare-metal cloud. Alcuni dei principali sono i seguenti:

OpenStack Ironic Software di bare-metal provisioning open-source integrato nel programma OpenStack, che permette di effettuare il provisioning di macchine fisiche invece di macchine virtuali [54].

RackHD Fornisce degli strumenti e delle API open-source che gli sviluppatori possono utilizzare per automatizzare la gestione e l'orchestrazione dell'hardware [105].

Foreman Uno strumento open-source per la gestione del ciclo di vita di server sia fisici che virtuali. L'amministratore potrà facilmente automatizzare operazioni ripetitive, effettuare il deployment di applicazioni e gestire server, sia localmente che sul cloud [48].

Digital Rebar Una piattaforma proprietaria, per l'automazione di data center, imparziale rispetto all'hardware, che permette la gestione ed il provisioning di infrastrutture sotto forma di codice (Infrastructure as Code) [39].

Collins Uno strumento che permette di modellare un'infrastruttura attraverso un insieme di asset; un asset potrebbe essere, ad esempio, un server, uno switch, un router o una configurazione. Componendo questi oggetti attraverso un insieme di tag, informazioni e transizioni di stato, permette di modellare ed automatizzare dinamicamente un'infrastruttura [29].

Cobbler Un server di installazione Linux che permette la configurazione rapida di ambienti di installazione basati sulla rete. Cobbler può aiutare nelle operazioni di provisioning, gestione del DNS e del DHCP, aggiornamento dei pacchetti software, power management e gestione delle configurazioni [28].

FAI - Fully Automatic Installation Uno strumento che permette di effettuare deployment di massa di sistemi operativi Linux. Permette di installare e configurare sistemi Linux e pacchetti software su macchine fisiche o su macchine virtuali, da ambienti composti da poche unità fino ad infrastrutture su larga scala [47].

Tinkerbell Uno strumento per il provisioning di macchine bare-metal. Tinkerbell è composto da cinque microservizi: Boots (DHCP server), Hegel (metadata service), OSIE (in-memory Operating System Installation Environment) , PBnJ (Power and Boot service) e Tink (workflow engine) [129].

MAAS Uno strumento open-source che permette di realizzare infrastrutture data center partendo da server bare-metal. Permette di individuare, ingaggiare, effettuare il deployment e riconfigurare dinamicamente una rete molto estesa di macchine. Permette di convertire l'hardware fisico in un data center distribuito, coeso e flessibile, minimizzando i tempi e gli sforzi necessari [69].

Tra queste alternative si è scelto di utilizzare MAAS (Metal As A Service). Tale scelta è giustificata dalle seguenti caratteristiche:

- è un software ben manutenuto; vengono aggiunte funzionalità e corretti problemi molto frequentemente;
- è stato sviluppato da Canonical Ltd., la stessa società che gestisce la distribuzione Linux Ubuntu; per questo, la compatibilità tra MAAS ed Ubuntu è sempre garantita, anche per le versioni più recenti;
- la documentazione disponibile è di alta qualità;

- la comunità è sufficientemente vasta, e mette a disposizione un forum per la risoluzione dei problemi;
- il codice è open-source, e, per questo, è possibile modificarlo per risolvere eventuali problemi o aggiungere funzionalità;
- Canonical Ltd. mette a disposizione un ulteriore tool di Infrastructure as Code, chiamato Juju, per il deployment di infrastrutture, come, ad esempio, Kubernetes. MAAS e Juju sono fortemente integrati tra loro, facilitando il deployment e la configurazione di applicazioni sulle macchine fisiche offerte da MAAS.

Ovviamente, questo non significa che MAAS sia il miglior software per realizzare bare-metal cloud, ma, indubbiamente, costituisce una valida alternativa.

3.3 MAAS (Metal As A Service)

MAAS (Metal As A Service), è un software che permette l'installazione e la configurazione automatica di macchine fisiche, realizzato da Canonical Ltd., la stessa società che gestisce la distribuzione Linux Ubuntu. In generale, MAAS permette di gestire un insieme di macchine fisiche in maniera analoga a come un hypervisor o un servizio cloud gestisce le proprie macchine virtuali: è possibile effettuare il deployment o la distruzione di macchine come se fossero ospitate, ad esempio, su un servizio cloud pubblico, come Amazon AWS, Google GCE, o Microsoft Azure.

MAAS integra al suo interno un insieme di servizi, tra cui un'interfaccia Web, un insieme di API (Application Programming Interface), una CLI (Command Line Interface), un ambiente di boot PXE (Preboot Execution Environment), un sistema di gestione delle immagini dei sistemi operativi ed un insieme di script di configurazione, che permettono all'amministratore di gestire, in maniera semplice e veloce, l'installazione e la configurazione di un determinato sistema operativo su un insieme di macchine fisiche. In realtà, MAAS è in grado di gestire anche macchine virtuali, analogamente a come gestisce quelle fisiche; purché queste possano effettuare il boot di rete tramite PXE.

Le funzionalità che verranno descritte nelle sezioni seguenti, faranno riferimento alla versione 2.9. Probabilmente, in futuro, alcune di esse verranno modificate ed altre aggiunte.

3.3.1 Principali tecnologie

MAAS, per poter installare e configurare le varie macchine da remoto, senza la necessità di dover preinstallare del software su di esse, utilizza varie tecnologie. Le principali, sono:

PXE (Preboot Execution Environment) Consiste in una modalità standard per permettere, a client che supportano PXE, il boot di un software, che potrebbe essere un sistema operativo, attraverso la rete [100]. Tali file vengono forniti da un server, che può gestire il boot di più client. L'unico requisito che il client deve soddisfare è quello di possedere una scheda di rete abilitata per il PXE, che sia utilizzabile dal firmware presente su quel sistema. PXE si basa su due protocolli standard, DHCP (Dynamic Host Configuration Protocol) per l'assegnazione dell'indirizzo di rete e TFTP (Trivial File Transfer Protocol) per lo scambio dei file di boot. UEFI supporta PXE, per cui, un client con tale firmware, sarà abilitato al suo utilizzo.

BMC (Baseboard Management Controller) PXE permette di avviare del software su una macchina senza sistema operativo. Permane, però, il problema, su come controllare, da remoto, l'accensione e lo spegnimento della macchina. Tale problema viene risolto sfruttando dei BMC, cioè dei microcontrollori esterni, presenti sulla scheda madre della macchina, che permettono di controllarla da remoto, attraverso un canale di comunicazione esterno. MAAS ha la capacità di comunicare con tali dispositivi, che solitamente sono presenti nelle macchine di classe server, per gestirne l'accessione, lo spegnimento e altri tipi di configurazioni.

Curtin e cloud-init Attraverso PXE è possibile avviare, su una macchina priva di software, un sistema operativo. Tale sistema è però solamente in esecuzione; per questo è necessaria una tecnologia che permetta di installarlo su uno dei dischi rigidi del sistema. Tale operazione viene effettuata attraverso il tool Curtin, che permette di definire le operazioni di installazione attraverso degli script. Successivamente all'installazione, sarà necessario configurare il sistema. Tale operazione viene effettuata attraverso il tool cloud-init. Cloud-init è un pacchetto software che contiene un insieme di utility necessarie all'inizializzazione di istanze cloud. MAAS, tramite cloud-init, specifica un insieme di impostazioni e script da eseguire per configurare il sistema operativo.

3.3.2 Interazione con l'utente

MAAS offre tre modalità di interazione: attraverso un'interfaccia web, attraverso una command line interface o tramite delle chiamate API HTTP RESTful.

L'interfaccia web è la modalità più immediata, per l'amministratore, per interagire con MAAS. Alcune delle funzionalità che l'interfaccia mette a disposizione, sono le seguenti:

Visualizzazione delle Macchine Mostra tutte le macchine gestite da MAAS, con le loro caratteristiche ed il loro stato. Selezionandole, è possibile visualizzare i loro

dettagli, tra cui, le informazioni sull'hardware, sulla rete, sullo storage e la lista eventi accaduti. Alcune proprietà possono essere configurate prima del deployment, come, ad esempio, il partizionamento dei dischi, la versione del kernel da utilizzare o la configurazione del BMC. Selezionando una macchina è inoltre possibile invocare un'azione, per avviare, ad esempio, il deployment di un sistema operativo.

Visualizzazione dei controller Similmente alle macchine, permette di visualizzare lo stato o configurare alcune impostazioni relative ai vari controller MAAS presenti nell'infrastruttura.

Visualizzazione dei pod KVM Permette di visualizzare i vari pod KVM, cioè macchine che possono avviare delle macchine virtuali KVM (Kernel-based Virtual Machine). Selezionando un determinato pod, è possibile creare una nuova macchina virtuale o visualizzare quelle attive.

Visualizzazione delle immagini Permette di visualizzare e selezionare le immagini dei sistemi operativi disponibili per il deployment.

Visualizzazione delle subnet Permette di visualizzare o modificare la configurazione delle varie reti gestite da MAAS.

Visualizzazione delle impostazioni Permette di visualizzare o modificare le impostazioni generali di MAAS.

L'interfaccia web offre varie funzionalità, ma, alcune funzionalità specifiche, sono disponibili solamente attraverso la command line interface.

Le stesse funzionalità vengono inoltre offerte tramite delle API HTTP RESTful. Solitamente, queste sono utilizzate da software esterni, come, ad esempio, Juju o la CLI appena descritta, che necessitano di accedere ai servizi offerti da MAAS.

3.3.3 Controller

MAAS, per poter offrire i propri servizi, dovrà essere installato su uno o più nodi. Saranno tali nodi che offriranno i servizi di MAAS, gestendo il processo di installazione e configurazione delle altre macchine. Tali nodi vengono chiamati controller e sono organizzati in maniera gerarchica per massimizzare le prestazioni.

I controller MAAS si dividono in due categorie:

Region controller Il region controller gestisce un insieme molto vasto di macchine, solitamente, un intero data center. Il region controller fornisce il servizio DNS, le API e l'interfaccia web. Inoltre, comunica con un database PostgreSQL, sul quale salva le informazioni di configurazione e di stato necessarie. Il region controller

è il componente che interagisce con l'utente, ricevendo le richieste di deployment o eliminazione di una particolare macchina [73].

Rack controller Il rack controller gestisce un insieme meno numeroso di macchine, solitamente un singolo rack. Il rack controller fornisce il servizio DHCP, TFTP e di power management; fornisce, inoltre, alle macchine che devono effettuare l'installazione, l'immagine del sistema operativo. Il rack controller è il componente che interagisce con le macchine da gestire; esso fornisce i servizi di rete ed i file necessari al boot, garantendo prestazioni elevate [73].

In un'infrastruttura gestita da MAAS devono essere presenti almeno un region controller ed un rack controller, che comunicheranno tra loro e rimarranno sincronizzati per operare come un unico sistema. Per infrastrutture non eccessivamente estese, è consigliabile installare sia il region controller che il rack controller sulla stessa macchina fisica. Il pacchetto di installazione di MAAS, di default, applica quest'ultima configurazione [3].

I vari controller possono supportare, inoltre, un meccanismo di high availability e load balancing, replicandoli su più nodi.

3.3.4 Ciclo di vita delle macchine

Le macchine, cioè gli elaboratori che dovranno essere gestiti e configurati, sono le risorse alla base di MAAS. Ogni macchina ha un particolare ciclo di vita, che prevede il passaggio attraverso una serie di stati [72]:

La macchina non è presente La macchina fisica non è mai stata accesa e MAAS non è al corrente della sua esistenza.

New La macchina fisica è stata accesa per la prima volta. La macchina ha effettuato il boot da rete attraverso i servizi PXE offerti da MAAS. Il sistema operativo avviato sulla macchina ha eseguito una serie di script per raccogliere alcune informazioni sull'hardware ed informare il controller MAAS della propria esistenza. Non è possibile effettuare il deployment di un sistema operativo sulle macchine in questo stato. Prima è necessario effettuare l'operazione di commission, che andrà ad individuare in modo completo e dettagliato l'hardware presente sul sistema, oltre che ad eseguire eventuali script di test o benchmark dell'hardware. Al termine della fase di commissioning, la macchina passerà allo stato di Ready.

Commissioning La macchina sta eseguendo delle operazioni di commissioning, cioè di individuazione dell'hardware e di esecuzione di script di test o di benchmark. Tale stato può essere raggiunto tramite un'azione di commission, quando la macchina si trova in stato di New o Ready, o alla prima accensione della macchina, prima di entrare nello stato di New.

Ready La macchina è disponibile per il deployment di un sistema operativo. Questo stato viene raggiunto se l'azione di commissioning va a buon fine.

Deploying La macchina sta eseguendo le operazioni di deployment di un sistema operativo.

Deployed La macchina ha portato a termine con successo l'installazione e configurazione del sistema operativo e può essere utilizzata.

Allocated La macchina è stata riservata ad un particolare utente di MAAS. Solamente tale utente potrà effettuare il deployment della macchina.

Failed Commissioning La fase di commissioning è fallita. Qualche script eseguito dal sistema operativo potrebbe essere fallito, oppure potrebbe essere stata eseguita un'azione di abort.

Failed Deployment La fase di deploying è fallita. Qualche script eseguito dal sistema operativo potrebbe essere fallito, oppure potrebbe essere stata eseguita un'azione di abort.

Broken La macchina è stata categorizzata come broken, o non funzionante. MAAS eviterà quindi di utilizzare tale macchina per eventuali deployment.

Rescue mode La macchina è entrata in una fase di recovery. MAAS ha avviato la macchina con un sistema operativo ephemeral, cioè in esecuzione interamente memoria senza l'utilizzo di dischi rigidi. Questo permette all'amministratore di accedervi tramite SSH e di analizzare e correggere eventuali problemi, anche se il sistema operativo originale, installato sul disco rigido, risulta corrotto o non funzionante.

Entering rescue mode La macchina sta eseguendo le operazioni per entrare nel rescue mode.

Exiting rescue mode La macchina sta eseguendo le operazioni per uscire dal rescue mode.

Locked Qualsiasi operazione sulla macchina è bloccata, a meno che non venga sbloccata con un'operazione di unlock. Questo permette di evitare modifiche accidentali.

Il passaggio da uno stato all'altro può avvenire tramite azioni, invocate dall'amministratore o da tool automatici, come, ad esempio, Juju, su una o più macchine [72]:

Commission Effettua il commissioning di un nodo nello stato New, portandolo prima allo stato di Commissioning e poi, se tutto è andato a buon fine, nello stato di Ready.

Deploy Effettua il deployment di un sistema operativo sulla macchina; è possibile specificare la versione del sistema operativo da installare, altrimenti sarà installata la versione di default. Durante la procedura, MAAS installa il sistema operativo automaticamente, configurando le interfacce di rete, le partizioni dei dischi e le impostazioni di sistema. Questa azione, che comprende quella di Power on, cambia lo stato della macchina da Ready a Deployed.

Release Questa azione, che comprende quella di Power off, cambia lo stato di un nodo da Deployed a Ready, rendendolo disponibile per altri deployment.

Delete La macchina viene eliminata dalle macchine gestite da MAAS. Quando la macchina verrà avviata nuovamente, MAAS la considererà una nuova macchina, nello stato di New.

Abort Permette di annullare un'azione di Commission o Deploy.

Rescue mode Porta la macchina allo stato di Entering rescue mode e, successivamente, allo stato Rescue mode.

Exit rescue mode Porta la macchina allo stato di Exiting rescue mode e, successivamente, allo stato della macchina antecedente all'entrata in Rescue mode.

Mark broken Imposta la macchina nello stato di Broken. Include l'azione di Power off.

Mark fixed Riporta lo stato di una macchina Broken allo stato di Ready.

Lock Imposta la macchina nello stato di Locked.

Unlock Rimuove la macchina dallo stato di Locked.

Override failed Se la macchina fallisce degli script di test dell'hardware non sarà più utilizzabile. Con questa azione la macchina viene sbloccata e considerata funzionante.

Power on Accende la macchina tramite il relativo BMC.

Power off Spegne la macchina tramite il relativo BMC.

Test hardware Esegue una serie di script sulla macchina per verificare che l'hardware funzioni correttamente.

3.3.5 Modalità di gestione delle macchine

Aggiunta di una nuova macchina

L'aggiunta di una nuova macchina può avvenire in maniera manuale o automatica [70]. Nel caso dell'aggiunta manuale, l'amministratore dovrà specificare manualmente le proprietà della macchina, tra cui, il nome, il dominio, l'architettura del processore, la versione del kernel Linux da utilizzare, l'indirizzo MAC ed il tipo di BMC. Successivamente, MAAS avvierà automaticamente l'operazione di commission.

Alternativamente, MAAS può aggiungere nuove macchine in maniera automatica, attraverso un'operazione di enlist. Le operazioni che MAAS compie, per aggiungere automaticamente una macchina con firmware UEFI, sono le seguenti:

1. MAAS avvia il servizio DHCP e TFTP, impostando l'ambiente di boot PXE;

2. il firmware UEFI della macchina è configurato in modo tale da avere, come prima opzione di boot, PXE. La macchina viene quindi avviata;

3. la macchina acquisisce il proprio indirizzo di rete, l'indirizzo del server TFTP ed il nome del file di boot da scaricare attraverso il server DHCP. MAAS seleziona tale file in base all'architettura ed al tipo di firmware che il client ha dichiarato al momento della richiesta DHCP. Nel caso di client UEFI, tale file è il bootloader GRUB (GNU GRand Unified Bootloader), che permette l'avvio del sistema operativo, offrendo varie funzionalità e configurazioni; nel caso di UEFI, è avviabile come applicazione UEFI, descritte nella Sezione 2.4.3. Ad esempio, viene fornito "grubaa64.efi" nel caso di un client con architettura arm64, "bootx64.efi" nel caso di un client con architettura x64;

4. il firmware UEFI presente sul client scarica, tramite TFTP, il file ".efi" contenente GRUB e lo avvia come applicazione UEFI. GRUB, appena avviato, ricerca, di default, il file di configurazione "grub/grub.cfg", nel quale sono specificate le possibili opzioni di boot. Poiché GRUB è stato avviato da TFTP, tale file verrà scaricato dal server TFTP. MAAS, in base all'operazione che vuole far compiere alla macchina, fornirà un file "grub.cfg" differente;

5. nel caso della prima accensione, MAAS fornisce al client un file di configurazione che indica a GRUB come scaricare il kernel Linux e l'initrd (Initial ramdisk) dal server TFTP. GRUB passa quindi il controllo al kernel Linux appena scaricato;

6. il kernel Linux scarica, tramite il server HTTP di MAAS, un'immagine Squashfs, che contiene il file system del sistema operativo da avviare. Solitamente, nel caso di MAAS, tale sistema operativo è Ubuntu. Questa modalità di avvio del sistema viene detta ephemeral, poiché la totalità dei file relativi al sistema operativo è caricata nella RAM, senza la necessità di un disco rigido;

7. avendo a disposizione tutti i file di Ubuntu, cloud-init, cioè un tool di configurazione automatica, può essere avviato;
8. vengono avviati gli script cloud-init, che individuano l'hardware della macchina ed informano il MAAS controller. MAAS registra la macchina nello stato di New;
9. la macchina viene spenta;
10. l'amministratore, prima di passare alla fase di commissioning, potrà configurare la macchina appena aggiunta tramite l'interfaccia web di MAAS o la CLI: ad esempio, potrà modificare il nome della macchina e definire le impostazioni riguardanti il BMC, nel caso in cui, quest'ultimo, non fosse stato individuato automaticamente.

Commissioning

Le operazioni effettuate da MAAS per portare a termine un'operazione di commissioning sono analoghe a quelle di enlist [71]. La differenza ricade negli script cloud-init, che, in questo caso, saranno in numero maggiore e potranno comprendere anche una serie di test per verificare il corretto funzionamento dell'hardware: ad esempio, dei test S.M.A.R.T per verificare la salute dei dischi rigidi.

Deployment

Le operazioni effettuate da MAAS per portare a termine un'operazione di deployment sono simili a quelle di enlist [74]. La differenza ricade negli script cloud-init, che, in questo caso, dovranno avviare l'operazione di installazione del sistema. Tale installazione avviene attraverso lo strumento Curtin, che eseguirà le varie operazioni di installazione copiando sul disco rigido l'immagine del sistema operativo Squashfs già scaricata. Al termine dell'installazione, la macchina verrà riavviata, per poter avviare il sistema operativo appena installato.

La macchina effettuerà nuovamente il boot tramite PXE. Le operazioni effettuate saranno analoghe a quelle di enlist fino al punto 4. A questo punto, MAAS fornirà al client un file di configurazione che indicherà a GRUB di ricercare, tra i vari dischi connessi al sistema, il bootloader del sistema operativo appena installato. Il controllo del sistema verrà quindi passato a tale bootloader con un'operazione di chainload. Il nuovo bootloader si occuperà, quindi, di avviare il sistema operativo installato sul disco rigido. Successivamente, saranno avviati gli script cloud-init per configurare il sistema.

3.3.6 Sistemi operativi

MAAS permette il deployment di vari sistemi operativi. Quello che garantisce il maggior grado di compatibilità con MAAS risulta essere Ubuntu, che è il sistema operativo di default, utilizzato per le operazioni di enlist, commission e deploy. MAAS permette di effettuare il deployment anche di CentOS, Windows, RHEL ed ESXi [75]. Effettuare il deployment di sistemi operativi diversi da Ubuntu risulta, però, molto più complesso. Infatti, per Ubuntu, viene fornito un repository di immagini preconfigurate per MAAS, mentre nel caso degli altri sistemi operativi è necessario realizzare l'immagine manualmente, attraverso particolari tool; inoltre, in quest'ultimo caso, è richiesta una licenza di supporto enterprise, chiamata Ubuntu Advantage.

MAAS, come impostazione di default, permette di scaricare, dall'interfaccia grafica o dalla CLI, un insieme di immagini Ubuntu, differenziate in base alla versione ed all'architettura per la quale sono state compilate. Tali immagini provengono dalla sorgente "https://images.maas.io/ephemeral-v3/stable/". Eventualmente, è possibile specificare una sorgente differente per avere accesso ad altre immagini.

Attualmente, nella sorgente di default, sono disponibili tutte le release di Ubuntu dalla 12.04 alla 21.04, compilate per le architetture amd64, arm64, armhf, i386, ppc64el e s390x. Non sempre sono possibili tutte le combinazioni di versione e architettura, ma le principali, cioè amd64 e arm64, sono disponibili per qualsiasi versione. Selezionando le release e le architetture desiderate, MAAS scaricherà le immagini sui vari controller, rendendole disponibili per il deployment.

Come descritto nella Sezione 3.3.5, MAAS avvia un sistema operativo anche durante le fasi di enlist e commission. Anche per tali fasi, si potrà selezionare la release di Ubuntu da utilizzare.

Kernel

Una caratteristica molto importante delle immagini Ubuntu è la versione del kernel. Solitamente, una determinata release viene fornita con più versioni del kernel Linux. Nelle impostazioni di MAAS, è possibile selezionare, fra le immagini scaricate, quale release e quale versione del kernel utilizzare, di default, per le operazioni di commission e di deploy. Le operazioni di enlist utilizzeranno la stessa versione specificata per le operazioni di commission. Esiste, però, un vincolo: le operazioni di enlist e commission potranno utilizzare solamente una release di Ubuntu LTS (Long-Term Support), cioè versioni con supporto a lungo termine che vengono rilasciate ogni due anni. Il deployment potrà, invece, avvenire con una versione qualsiasi. Eventualmente, l'azione di deploy potrà specificare una release o una versione del kernel differente da quella di default.

La possibilità di specificare la versione del kernel permette di ottenere un maggior grado di compatibilità con l'hardware o di ottenere nuove funzionalità, aggiunte dai

kernel Linux più recenti. I kernel disponibili nelle immagini Ubuntu sono classificati in quattro categorie [83]:

General availability (GA) Corrisponde alla versione del kernel che viene fornita, di default, con una determinata release di Ubuntu. Ogni release di Ubuntu è associata ad una determinata versione major del kernel, che, in base alle policy di Ubuntu, non verrà mai aggiornata ad un'altra versione major. Ad esempio, poiché Ubuntu 20.04 è basato sul kernel Linux 5.4, la versione ga-20.04 corrisponderà, sempre, al kernel Linux 5.4.

Hardware enablement (HWE) Un kernel hwe permette di installare un kernel Linux più recente su una determinata release di Ubuntu, in modo da ottenere la maggiore compatibilità hardware e le funzionalità aggiuntive dei nuovi kernel Linux. Ad esempio, attualmente, la versione hwe-20.04, possiede il kernel Linux 5.8.

Hardware enablement (pre-release) Una release edge del kernel HWE; corrisponde ad una versione più recente rispetto al kernel HWE, ma, essendo in fase sperimentale, potrebbe essere instabile.

Low latency Una versione del kernel basata sul kernel GA o HWE, che utilizza, però, una configurazione più aggressiva, per ridurre la latenza. Viene categorizzato come un kernel soft real-time.

3.3.7 Power Management

MAAS, per poter controllare appieno le varie macchine, deve avere la possibilità di controllare la loro alimentazione, cioè essere in grado di accenderle, di spegnerle e di verificare il loro stato. Solitamente, le macchine di classe server dispongono di dispositivi appositi, chiamati BMC (Baseboard Management Controller), che offrono tali funzionalità. Un BMC è un microcontrollore esterno, presente sulla scheda madre del server, che permette di gestire e monitorare il sistema da remoto, attraverso un canale di comunicazione dedicato.

Attualmente, esistono varie tipologie di BMC; solitamente, sono soluzioni proprietarie con i propri protocolli e le proprie funzionalità. MAAS supporta vari BMC, attraverso specifici driver, necessari per autenticarsi e comunicare, tramite il giusto protocollo, con lo specifico Baseboard Management Controller [80].

Prima del commissioning di una macchina, MAAS richiede obbligatoriamente di selezionare il driver BMC da utilizzare. Non sempre, però, le macchine possiedono un BMC: MAAS, per questo, offre anche un driver di tipo "Manual". Utilizzando tale driver, MAAS non potrà effettuare il power management della macchina, e delegherà all'amministratore l'onere di accendere e spegnere le macchine manualmente.

3.3.8 Rete

MAAS, per poter realizzare uno o più ambienti di boot PXE, deve essere capace di modellare e gestire alcuni concetti rete.

Subnet

Corrisponde ad una determinata sottorete di indirizzi IP. MAAS permette di definire due tipi di subnet: managed e unmanaged [82]. Nelle subnet managed, può intervenire il server DHCP di MAAS per fornire degli indirizzi di rete alle macchine, mentre nelle subnet unmanaged, il server DHCP di MAAS non potrà intervenire; si ricade in quest'ultimo caso quando la rete è già gestita da un altro server DHCP.

Gli indirizzi possono essere forniti alle interfacce di rete delle macchine in tre modalità [71]:

Auto assign MAAS assegna all'interfaccia un indirizzo IP statico casuale.

Static assign L'amministratore assegna all'interfaccia un indirizzo IP statico.

DHCP MAAS assegna all'interfaccia un indirizzo IP non statico, fornito dal server DHCP.

Per ogni subnet è possibile, inoltre, definire dei range di indirizzi, che possono essere riservati o dinamici. In base al tipo di subnet, il comportamento dei range di indirizzi riservati cambia [76]:

Subnet managed con range riservato MAAS non assegnerà mai gli indirizzi specificati da questo range, né come indirizzi DHCP né come indirizzi di rete statici per le macchine.

Subnet unmanaged con range riservato MAAS utilizzerà gli indirizzi specificati da questo range per assegnarli staticamente alle macchine. Non verranno però utilizzati dal DHCP.

Subnet managed con range dinamico Gli indirizzi di questo range verranno utilizzati esclusivamente dal server DHCP. Non saranno utilizzati come indirizzi di rete statici per le macchine.

Subnet unmanaged con range dinamico MAAS non assegnerà mai gli indirizzi specificati da questo range, né come indirizzi DHCP né come indirizzi di rete statici per le macchine.

VLAN

Una rete locale LAN fisica può essere suddivisa in più reti locali virtuali, denominate VLAN (Virtual Local Area Network). Tale funzionalità deve però essere supportata dai dispositivi di rete dell'infrastruttura, cioè dagli switch e dai router. L'utilizzo di VLAN permette di creare reti locali separate, ma che condividono lo stesso hardware di rete. MAAS supporta le VLAN, e può quindi gestire macchine che si trovano in tali reti.

All'interno di una VLAN possono essere create più subnet, descritte precedentemente. Ad ogni VLAN può essere assegnato un server DHCP, che gestirà tutti gli indirizzi dinamici delle subnet managed assegnate a tale VLAN. Le VLAN possono essere raggruppate in fabric; un fabric permette di interconnettere fra loro le varie VLAN che ne fanno parte [72].

DHCP snippet

MAAS, come già accennato, possiede un server DHCP interno che può essere utilizzato per gestire gli indirizzi dinamici delle subnet gestite da MAAS. Il server DHCP di MAAS è l'elemento chiave per permettere il boot PXE delle macchine. Esso è configurato in modo da fornire, ai client PXE, le opzioni DHCP necessarie al boot, come l'indirizzo del server TFTP ed il nome del file di boot da scaricare. La scelta del file di boot da fornire al client viene effettuata dal server DHCP, sulla base dell'architettura dichiarata dal client al momento della richiesta DHCP. Tale scelta è definita nel file di configurazione del server DHCP, che può essere dinamicamente modificato da MAAS.

È data però l'opzione di modificare il comportamento del server DHCP definendo degli snippet [77], cioè dei file di configurazione parziali, che vengono aggiunti al file di configurazione principale del server DHCP. Il server DHCP di MAAS è basato su dhcpd, che viene configurato attraverso un file "dhcpd.conf"; gli snippet dovranno aderire al formato di tale file.

Proxy

MAAS può fornire, alle macchine che gestisce, un servizio di proxy HTTP e HTTPS, per accedere, ad esempio, ai pacchetti software forniti attraverso APT (Advanced Packaging Tool) [81]. Il proxy effettua un caching delle richieste HTTP, garantendo performance elevate nel caso più macchine richiedessero la stessa risorsa. In un contesto come quello di MAAS, una situazione tale è molto probabile, ed un sistema di caching permette di migliorare notevolmente le performance ed il carico sulla rete. Il proxy può essere configurato in tre modi:

Interno Il proxy è gestito interamente da MAAS

Esterno Il proxy è esterno a MAAS; il traffico in uscita dalle macchine sarà diretto verso tale proxy.

Disabilitato Il proxy è disabilitato.

NTP

MAAS fornisce un servizio NTP (Network Time Protocol) alle macchine che gestisce ed ai propri controller [78]. Questo permette di mantenere sincronizzati i tempi tra i vari controller ed anche tra le diverse macchine gestite. Il region controller si sincronizza con un server NTP esterno. I rack controller si sincronizzano con il region controller. Infine, ogni macchina è sincronizzata con il proprio rack controller.

3.3.9 Pod KVM

Al momento del deployment di una macchina, MAAS permette di configurarla come pod KVM (Kernel-based Virtual Machine). MAAS installerà, automaticamente, tutto il software necessario per renderla tale. Un pod KVM permette di creare macchine virtuali, che potranno essere gestite, da MAAS, come ogni altra macchina fisica. La macchina configurata come pod KVM metterà a disposizione le proprie risorse hardware, cioè CPU, RAM e storage, per l'esecuzione di una o più macchine virtuali, che potranno essere dinamicamente create da MAAS [85].

Successivamente al deployment del pod KVM, verrà aggiunta la rispettiva voce, nella pagina KVM dell'interfaccia di MAAS. Dall'interfaccia sarà possibile creare una nuova macchina virtuale da eseguire sul pod, con determinate risorse. La macchina virtuale, una volta creata, verrà quindi aggiunta alle macchine gestite da MAAS e verrà avviata l'operazione di commission. La macchina sarà inoltre preconfigurata per essere gestita attraverso il driver BMC per virsh. Virsh è un tool che permette la gestione, anche da remoto, di macchine virtuali KVM.

MAAS può inoltre creare macchine virtuali per soddisfare determinate richieste: ad esempio, se un tool esterno, come Juju, richiedesse una macchina con determinate caratteristiche, MAAS potrebbe decidere di creare una nuova macchina virtuale con tali caratteristiche su uno dei pod che gestisce.

Inoltre, è possibile aggiungere pod KVM non creati attraverso MAAS; ma, in questo caso, è necessario specificare le credenziali virsh o LXD, necessarie per controllare l'hypervisor sulla macchina.

Capitolo 4

Infrastructure as Code

Come descritto nel Capitolo 3, un bare-metal cloud permette di automatizzare il processo di installazione e gestione di un insieme di macchine fisiche, permettendo di allocare tali macchine su richiesta.

Come già accennato nel capitolo introduttivo, uno degli obiettivi di questa trattazione consisterà nella realizzazione di un cluster di macchine fisiche; le risorse di tale cluster verranno gestite attraverso un particolare cluster management software, Kubernetes, le cui componenti dovranno essere installate sulle varie macchine.

I bare-metal cloud permettono di allocare, in maniera automatica, le varie macchine che dovranno fare parte del cluster. Permane, però, il problema relativo all'installazione ed alla configurazione delle componenti del cluster management software. Sicuramente, l'amministratore potrebbe, attraverso un bare-metal cloud, allocare le macchine necessarie per realizzare il cluster, procedendo poi, manualmente, all'installazione e configurazione delle componenti del cluster management software su ognuna di esse.

Tale strategia avrebbe, però, una serie di svantaggi. Infatti, sebbene i bare-metal cloud permettano di installare, automaticamente, il sistema operativo sulle macchine, sarebbe comunque l'amministratore a doverle selezionare per l'utilizzo; inoltre, le operazioni manuali di installazione e configurazione del software su tali macchine sarebbero ripetitive, soggette ad errori, lente e difficilmente ripetibili.

Per questo, negli ultimi anni, sono nati degli strumenti per automatizzare queste operazioni, identificabili attraverso il termine Infrastructure as Code.

In questo capitolo si andrà quindi a definire cosa si intende per Infrastructure as Code e quali siano le principali caratteristiche degli strumenti che adottano tale modello. Saranno quindi individuate le principali alternative attualmente disponibili.

Infine, verranno descritte le principali caratteristiche dei due strumenti di Infrastructure as Code scelti per la realizzazione di questo progetto: Canonical Juju per il deployment e la configurazione del cluster management software sul cluster di

macchine fisiche; Terraform per il deployment e la configurazione delle applicazioni gestite dal cluster management software.

4.1 Definizione

L'idea alla base degli strumenti di IaC (Infrastructure a Code), è quella di scrivere ed eseguire un codice per definire, aggiornare o distruggere la propria infrastruttura [20, cap. 1], intendendo per infrastruttura l'insieme di risorse hardware, quindi server o dispositivi di rete, e software, cioè particolari ambienti di esecuzione o applicativi, la cui integrazione permette l'offerta di specifici servizi. Ad esempio, se un'organizzazione volesse fornire un determinato servizio digitale ai propri clienti, dovrebbe definire e realizzare un'infrastruttura, composta da server, dispositivi di rete e applicazioni, alla base della fornitura di tale servizio.

Con l'avvento dei servizi cloud, l'allocazione delle risorse necessarie per la realizzazione di tali infrastrutture si è semplificata notevolmente, eliminando la necessità di gestire, manualmente, i problemi legati all'hardware di un'infrastruttura fisica.

Nonostante ciò, la possibilità di realizzare o apportare delle modifiche ad un'infrastruttura in maniera facile, veloce e sicura, non deriva meramente dall'adozione di un servizio cloud [93, cap. 1]. Infatti, se le operazioni necessarie per effettuare il deployment e la configurazione dell'infrastruttura risultassero scoordinate e caotiche in un ambiente tradizionale, risulterebbero tali anche in un ambiente cloud; anzi, l'adozione dei servizi cloud non farebbe altro che velocizzare la propagazione di tale situazione caotica [93, cap. 1]. Una condizione di questo genere si verifica, solitamente, quando l'infrastruttura viene gestita manualmente: l'amministratore installa manualmente un determinato sistema operativo, con determinate versioni dei pacchetti software e determinate configurazioni. Solamente l'amministratore conosce i dettagli dell'infrastruttura, che, nel tempo, viene adattata alle esigenze delle applicazioni che devono sfruttarla. Con l'aumentare delle dimensioni di un'infrastruttura gestita manualmente, aumentano i problemi: i vari amministratori potrebbero compiere errori nel configurare nuove macchine, le macchine aggiunte più recentemente potrebbero possedere dei pacchetti software più recenti che potrebbero interferire con il comportamento delle applicazioni, oppure, nel tempo, potrebbero essere state applicate configurazioni o installati software di cui ci si è poi dimenticati. Un'infrastruttura di questo tipo risulta praticamente ingestibile: al momento del deployment di una nuova versione dell'applicazione, o dell'aggiunta di un nuovo server, o della modifica dell'infrastruttura di rete o dell'aggiornamento del sistema operativo, è necessario intervenire manualmente, risolvendo gli inevitabili problemi causati dall'eterogeneità che le configurazioni di tali macchine hanno raggiunto (configuration drift). L'intera infrastruttura potrebbe essere comunque funzionante, ma

le modifiche e gli aggiornamenti sarebbero scoraggiati dall'incapacità di poterla ricreare velocemente da zero se si verificasse qualche grave problema.

L'obiettivo degli strumenti di Infrastructure as Code (IaC) è quella di definire qualsiasi proprietà, configurazione e risorsa dell'infrastruttura attraverso del codice, cioè dei file che ne contengono la descrizione in un qualche tipo di linguaggio. L'approccio IaC si basa, principalmente, sulle pratiche dello sviluppo software [93, cap. 1]. L'idea è quella di definire, analogamente allo sviluppo software, un insieme di procedure consistenti e ripetibili per effettuare il deployment o la modifica di un'infrastruttura; la modifica del codice che descrive l'infrastruttura si ripercuote, automaticamente, sull'infrastruttura reale.

Una strategia di questo tipo permette di risolvere i problemi derivanti da una gestione manuale, grazie alle seguenti caratteristiche [20, cap. 1]:

Riduzione delle operazioni degli amministratori Gli amministratori non devono più gestire manualmente l'installazione e la configurazione delle singole macchine. Sarà lo strumento di IaC che si occuperà di effettuare, automaticamente, tali operazioni, sulla base del codice che descrive l'infrastruttura desiderata.

Velocità e ripetibilità Poiché il processo di deployment è automatico, la sua velocità sarà maggiore rispetto a quella di un amministratore che opera manualmente, non soggetto ad errori manuali e totalmente ripetibile. In caso di necessità, lo strumento IaC potrebbe infatti ricreare l'intera infrastruttura da zero, con le stesse identiche caratteristiche e configurazioni di quella precedente.

Documentazione Il codice che descrive l'infrastruttura, oltre ad essere utilizzato dallo strumento IaC per effettuarne il deployment, permette anche al team di sviluppo, o, in generale, a chiunque ne fosse interessato, di comprendere lo stato e le caratteristiche dell'infrastruttura.

Controllo delle versioni Il codice che descrive l'infrastruttura potrebbe essere gestito da strumenti di controllo delle versioni, come Git. Questo permette di tenere traccia delle modifiche apportate all'infrastruttura, e, nel caso si verificassero particolari problemi, effettuare un rollback alla versione precedente.

Validazione Poiché le modifiche vengono prima apportate al codice, è possibile, prima di effettuare il deployment delle modifiche sull'infrastruttura reale, effettuare un processo di validazione, per ridurre al minimo la probabilità che possa verificarsi un problema nell'ambiente di produzione. Solitamente, si crea, attraverso lo strumento IaC, un'infrastruttura di test con le nuove modifiche, per verificare l'assenza di problemi.

Riutilizzo Il codice che descrive l'infrastruttura potrebbe essere definito in maniera generica, e ciò permetterà di riutilizzare alcune parti, la cui affidabilità è stata dimostrata nel corso del tempo, per infrastrutture diverse.

Gli strumenti IaC, solitamente, ma non sempre, permettono di creare l'infrastruttura definita nel codice sfruttando i servizi messi a disposizione da un cloud provider. In generale, gli strumenti IaC sono classificabili sulla base delle seguenti caratteristiche [20, cap. 1]:

Provisioning o gestione delle configurazioni Uno strumento IaC che si occupa del provisioning permette di creare, installare e configurare una nuova risorsa dell'infrastruttura. L'operazione di gestione delle configurazioni è parte del provisioning, ma si riferisce anche a tutte quelle operazioni di riconfigurazione delle risorse già presenti. Ad esempio, un tool di gestione delle configurazioni potrebbe permettere di riconfigurare allo stesso modo i sistemi operativi in esecuzione su un insieme di macchine. Un tool di provisioning potrebbe invece permettere, sfruttando i servizi di un cloud provider, di allocare automaticamente le varie risorse dell'infrastruttura, come macchine o dispositivi di rete, e di configurarle.

Infrastruttura mutabile o immutabile Solitamente, gli strumenti IaC di gestione delle configurazioni modificano le risorse già presenti nell'infrastruttura, mutando alcune delle loro proprietà. In questo caso ci si riferisce, quindi ad un'infrastruttura mutabile. Altri strumenti IaC, invece, nel momento in cui devono apportare una modifica ad una risorsa, la distruggono e la ricreano completamente. In quest'ultimo caso ci si riferisce ad un'infrastruttura gestita in maniera immutabile. Le operazioni sulle infrastrutture gestite in questo modo potrebbero essere meno efficienti, poiché la risorsa deve essere ricreata; in compenso, però, si evitano possibili problemi causati dall'accumulo, nel tempo, di continue modifiche alla stessa risorsa, che potrebbero produrre comportamenti non previsti.

Linguaggi imperativi o dichiarativi Il codice che descrive l'infrastruttura potrebbe essere specificato, in base allo strumento IaC adottato, tramite un linguaggio imperativo o dichiarativo. Nel caso dell'utilizzo di un linguaggio imperativo, sarà necessario specificare tutte le operazioni da compiere per ottenere l'infrastruttura voluta. Nel caso dell'utilizzo di un linguaggio dichiarativo, sarà invece sufficiente dichiarare lo stato finale desiderato dell'infrastruttura; sarà lo strumento IaC ad individuare le operazioni da compiere per raggiungere tale risultato. I linguaggi imperativi, poiché consentono di specificare le singole operazioni da compiere, permettono una maggiore espressività per risolvere particolari problemi; tuttavia essi rendono più difficile comprendere quale sarà lo

stato finale del sistema, che dipenderà sia dallo stato attuale sia dalle operazioni specificate. I linguaggi dichiarativi permettono invece di prevedere facilmente quale sarà lo stato finale del sistema, poiché esso dovrà corrispondere a quanto dichiarato nel codice: lo strumento IaC individuerà le operazioni da compiere per raggiungere lo stato finale partendo dallo stato attuale.

Presenza o meno di un nodo master Alcuni strumenti IaC prevedono la presenza di un nodo master, che mantiene le configurazioni e distribuisce gli aggiornamenti alle varie risorse dell'infrastruttura. L'amministratore comunica, attraverso un software client, esclusivamente con il nodo master, specificando i comandi e le configurazioni da applicare alle varie risorse. Altri strumenti IaC non prevedono tale nodo: sarà il client a dialogare direttamente con le varie risorse da configurare o con i provider che potranno fornire tali risorse.

Presenza o meno di agenti Alcuni strumenti IaC prevedono l'installazione di un agente sulle varie risorse da gestire. Tale agente monitorerà lo stato della risorsa e applicherà le configurazioni fornite dallo strumento IaC. Altri strumenti non prevedono la presenza di tale agente: le configurazioni verranno applicate sfruttando metodologie standard.

4.2 Principali alternative

Attualmente esistono vari software che permettono di adottare il modello Infrastructure as Code. Tali software si differenziano tra loro, principalmente, per l'ambito applicativo e per le modalità di operare. Alcuni dei principali sono i seguenti:

Ansible Uno strumento che permette di automatizzare l'intero ciclo di vita di un'applicazione e le operazioni di continuous delivery. Tra queste, il provisioning, la gestione delle configurazioni, il deployment di applicazioni ed il continuous delivery [6]. Ansible non prevede l'uso di agenti né di nodi master. Il linguaggio adottato permette un approccio sia di tipo imperativo che dichiarativo.

Chef Uno strumento che permette di automatizzare la configurazione dell'infrastruttura, assicurando che ogni sistema sia configurato correttamente e consistentemente. I vari server gestiti da Chef sono continuamente confrontati con lo stato desiderato, assicurando la correzione automatica di eventuali configuration drift ed applicando globalmente gli eventuali cambiamenti di configurazione [25]. Chef prevede l'uso di agenti e di nodi master. Il linguaggio adottato è di tipo imperativo.

Puppet Uno strumento che permette di gestire ed automatizzare l'infrastruttura o i processi di lavoro in maniera molto semplice ma anche molto efficace [103].

Puppet prevede l’uso di agenti e di nodi master. Il linguaggio adottato è di tipo dichiarativo.

Otter Uno strumento che permette il provisioning di server e la gestione delle configurazioni [97]. Otter prevede l’uso di agenti e di nodi server. Il linguaggio adottato permette un approccio sia di tipo imperativo che dichiarativo.

Salt Uno strumento che utilizza dei semplici file YAML, leggibili facilmente dall’utente, combinati ad una automazione di tipo event-driven per effettuare il deployment e la configurazione di infrastrutture complesse [122]. Salt prevede l’uso di agenti e di nodi master. Il linguaggio adottato permette un approccio sia di tipo imperativo che dichiarativo.

AWS CloudFormation Uno strumento che offre una semplice modalità per modellare una collezione di risorse, offerte da AWS o da terze parti; permette quindi di effettuare il provisioning, veloce e consistente, di tali risorse, gestendo, successivamente, il loro ciclo di vita [17]. Il linguaggio adottato è di tipo dichiarativo.

OpenStack Heat Uno strumento, composto da un sistema di orchestrazione, che permette di avviare un insieme di applicazioni cloud composite, tramite l’utilizzo di template sotto forma di file di testo, che possono essere trattati come codice [96]. Il linguaggio adottato è di tipo dichiarativo.

Juju Uno strumento che permette di effettuare il deployment, la configurazione, la gestione, la manutenzione e lo scaling di applicazioni cloud, in maniera veloce ed efficiente, sia su cloud pubblici che su server fisici [57]. Juju prevede l’uso di agenti e di nodi master. Il linguaggio adottato è di tipo dichiarativo.

Terraform Uno strumento di Infrastructure as Code open-source che fornisce un workflow consistente, composto dalle fasi di Write, Plan ed Apply, per gestire centinaia di servizi cloud [125]. Terraform non prevede l’uso di agenti né di nodi master. Il linguaggio adottato è di tipo dichiarativo.

Nell’ambito di questo progetto, era necessario scegliere degli strumenti IaC che permettessero di:

- installare e configurare, automaticamente, il cluster management software (Kubernetes) sulle varie macchine gestite da MAAS;
- successivamente all’installazione di Kubernetes, facilitarne la gestione, permettendo, ad esempio, di installare, eliminare o gestire le applicazioni containerizzate.

Tra queste alternative si è scelto di utilizzare, per l'installazione del cluster management software, Canonical Juju, che è dotato di caratteristiche che ne giustificano la scelta:

- è un software ben manutenuto; vengono aggiunte funzionalità e corretti problemi molto frequentemente;
- la documentazione disponibile è di alta qualità;
- la comunità è sufficientemente vasta, e mette a disposizione un forum per la risoluzione dei problemi;
- è fortemente integrato con MAAS, poiché prodotto dalla stessa organizzazione, Canonical Ltd; è predisposto per utilizzare e gestire, automaticamente, le macchine offerte da MAAS, semplificando e velocizzando le operazioni di deployment;
- è predisposto per effettuare il deployment di un'infrastruttura Kubernetes, senza la necessità di definire script o configurazioni personalizzate per tale scopo;
- ha un approccio di tipo dichiarativo: è possibile definire un modello che specifica le applicazioni desiderate, le loro relazioni, le loro configurazioni ed i requisiti hardware delle macchine sulle quali dovranno essere installate; Juju si occuperà, automaticamente, di realizzare tale modello.

Sicuramente, Juju non era l'unica alternativa possibile per effettuare il deployment di Kubernetes; ad esempio, anche Kubespray, tramite Ansible, poteva soddisfare tale esigenza. Tuttavia, le caratteristiche appena descritte, e, in particolare, l'integrazione con MAAS, hanno incoraggiato l'adozione di questa soluzione.

Per quanto riguarda la gestione delle applicazioni Kubernetes, si è scelto di utilizzare Terraform. Tale scelta è giustificata dalle seguenti caratteristiche:

- è un software che non richiede un'infrastruttura aggiuntiva, permettendo di gestire le varie risorse senza la necessità di agenti o di nodi master;
- è compatibile con Kubernetes: permette di automatizzare e facilitare la gestione dei vari oggetti Kubernetes, come applicazioni o configurazioni;
- permette un approccio dichiarativo, consentendo di effettuare il deployment o la distruzione delle risorse con un singolo comando.

Anche in questo caso, Terraform non era l'unica alternativa possibile, poiché la gestione degli oggetti Kubernetes potrebbe avvenire, facilmente, anche senza adottare un particolare strumento IaC, sfruttando esclusivamente gli strumenti messi a disposizione da Kubernetes, come il tool kubectl. Terraform consente, però, di facilitare e velocizzare molte operazioni, e queste opportunità ne giustificano la scelta.

4.3 Juju

Juju è uno strumento di Infrastructure as Code, che permette di effettuare il deployment e la gestione di applicazioni distribuite, sfruttando le risorse messe a disposizione da un servizio cloud, che potrebbe essere offerto, indifferentemente, da un provider pubblico, come Amazon AWS, o privato, come MAAS. Juju, utilizzerà tale cloud provider per allocare le macchine necessarie, sulle quali installerà poi le varie applicazioni.

Le applicazioni vengono distribuite sotto forma di charm, cioè di pacchetti software, contenenti, oltre all'applicazione, tutta la logica relativa al processo di installazione, la logica per reagire a cambiamenti di configurazione o per interagire con altri charm. L'interazione fra charm è abilitata definendo delle relazioni. I vari charm e le relazioni che sussistono tra essi, sono raccolti in un modello, che può essere definito attraverso un linguaggio dichiarativo.

Successivamente alla definizione del modello, sarà Juju ad occuparsi di tutte le operazioni di deployment, ottenendo le varie macchine, installando le applicazioni, cioè i charm, e configurandoli.

Per operare, Juju necessita di un nodo, denominato controller, per ogni cloud provider utilizzato; inoltre, verranno installati, su ogni macchina allocata, un insieme di agenti, che si occuperanno di monitorare e gestire la macchina stessa ed i vari charm in esecuzione.

Le funzionalità che verranno descritte nelle sezioni seguenti, faranno riferimento alla versione 2.9. Probabilmente, in futuro, alcune di esse verranno modificate ed altre aggiunte.

4.3.1 Cloud provider e controller

Juju, per poter operare, necessita di un servizio che permetta l'allocazione di macchine su richiesta, sulle quali, poi, verrà effettuato il deployment delle varie applicazioni. Tale servizio può essere fornito da un cloud provider, pubblico o privato; l'unico vincolo, è che consenta di allocare macchine o container. Juju supporta varie piattaforme cloud [58]:

- Amazon AWS
- Microsoft Azure
- Google GCE
- Oracle
- Rackspace
- LXD

- Kubernetes
- VMware vSphere
- OpenStack
- MAAS

Juju, per connettersi ad uno o a più di questi cloud, dovrà possedere le relative credenziali. Successivamente, Juju procederà con la fase di bootstrap del controller. Juju, infatti, per poter gestire le macchine relative ad un determinato cloud provider, necessita di un nodo sulla loro stessa rete, che possa comunicare direttamente con tali macchine. Il nodo in questione è chiamato controller, e deve essere creato attraverso un'operazione di bootstrap: Juju allocherà una nuova macchina tramite il cloud provider, e vi installerà il software relativo al controller.

Il controller è il nodo centrale di gestione per tutte le macchine appartenenti ad un determinato cloud, poiché gestisce tutte le operazioni di installazione e configurazione richieste dall'amministratore su quelle determinate macchine. Inoltre, mantiene un database interno dove memorizza lo stato dei modelli, delle applicazioni, delle macchine e degli utenti, inerenti al cloud che gestisce [35]; tali informazioni e le possibili azioni su di esse sono esposte attraverso un insieme di API WebSocket.

Solitamente, si configura almeno un controller per ogni cloud configurato, ma, opzionalmente, è possibile utilizzare un unico controller per più cloud. Poiché il controller è, a tutti gli effetti, un possibile single point of failure, è possibile configurare più controller per cloud, tramite la cosiddetta modalità high availability. In tale stato, se il controller principale (master) dovesse fallire, sarebbe rimpiazzato immediatamente da un controller secondario.

Tra i cloud supportati compare anche Kubernetes. Nel caso si utilizzasse tale cloud, Juju effettuerà il deployment delle applicazioni, che dovranno essere charm compatibili a tale scopo, sul cluster Kubernetes configurato. Questa modalità non ha nulla a che vedere con l'utilizzo di Juju per effettuare il deployment di un cluster Kubernetes sulle macchine offerte da un cloud provider.

Alternativamente, Juju può operare anche senza un servizio cloud, sebbene con alcune limitazioni. Tale possibilità, chiamata "manual cloud" [58], permette di utilizzare Juju su un insieme di macchine non gestite da alcuna infrastruttura cloud. Juju potrà accedere a tali macchine, tramite ssh, per installare e configurare i vari charm; non potrà però aggiungere o rimuovere macchine dinamicamente, in base alle necessità.

4.3.2 Interazione con l'utente

Juju offre tre modalità di interazione:

Juju client (CLI) Lo strumento principale che Juju offre all'amministratore o ad un utente, per impartire comandi o controllare lo stato del sistema. Consiste in una command line interface, disponibile dal momento dell'installazione del pacchetto software Juju. Tramite tale CLI è possibile eseguire la totalità dei comandi disponibili per le operazioni di monitoraggio o di deployment. Ad esempio, permette di configurare nuovi cloud ed effettuare il bootstrap di nuovi controller. Il Juju client comunicherà, quindi, con i vari controller, per reperire le informazioni di stato o inoltrare i comandi dell'amministratore.

Dashboard Ogni controller Juju mette a disposizione una dashboard, cioè un'interfaccia web che permette di monitorare lo stato dei modelli, delle applicazioni e delle macchine. Tale interfaccia permette, però, solamente la visualizzazione dello stato; non permette di effettuare alcun tipo di azione, che dovrà invece essere impartita attraverso la CLI [55]. Per facilitare l'accesso alla CLI, la dashboard mette a disposizione una web CLI, sulla quale potranno essere impartiti i vari comandi.

WebSocket API Ogni controller Juju mette a disposizione delle API WebSocket, che permettono di accedere allo stato dei modelli, delle applicazioni o delle macchine e di effettuare azioni su esse. Tale modalità viene utilizzata, solitamente, da strumenti di automazione o programmi esterni, che necessitano di comunicare con Juju.

4.3.3 Deployment di applicazioni

Juju permette di effettuare il deployment di applicazioni distribuite su più macchine. I concetti principali alla base del deployment di applicazioni sono i seguenti:

Modello

Un modello è assimilabile ad un workspace, poiché raggruppa tutte le entità, cioè applicazioni, configurazioni, macchine e relazioni, di cui dovrà essere costituito il deployment. Tale modellazione permette, inoltre, di astrarre dai dettagli implementativi, consentendo di concentrarsi, esclusivamente, sulla scelta delle applicazioni e sulle loro relazioni.

Un modello è associato ad un singolo controller, ma un controller può possedere e gestire più modelli.

Un modello può essere definito tramite due modalità:

- definendo un file YAML, che specifica, in maniera dichiarativa, tutte le applicazioni, le relazioni fra esse, le configurazioni e le macchine necessarie al deployment. Tale file YAML verrà quindi importato attraverso la CLI;

- creando un modello vuoto ed aggiungendo le varie entità manualmente tramite la CLI.

Gli approcci appena descritti non sono esclusivi: è possibile importare un modello tramite un file YAML per poi effettuare delle modifiche attraverso la CLI. Dopo aver effettuato tali modifiche, è possibile esportare il relativo YAML, che potrà essere utilizzato per realizzare un deployment identico.

Non appena il modello viene definito, Juju inizia, automaticamente, le operazioni di deployment, per raggiungere lo stato specificato in esso.

Macchine e Container

Una macchina corrisponde ad un elaboratore offerto dal cloud provider, su di essa verranno installate le eventuali applicazioni, seguendo le indicazioni del modello. Nel modello è possibile specificare i requisiti della macchina, in termini di CPU, RAM e storage, necessari ad ospitare una determinata applicazione. Juju utilizzerà il cloud provider per allocare una macchina con tali caratteristiche.

Inoltre, è possibile specificare, nel modello, delle macchine virtuali, da creare all'interno di altre macchine già definite. Tali macchine sono gestite, da Juju, come ogni altra macchina, ma vengono chiamate container. Esse possono essere basate su LXD o KVM (Kernel-based Virtual Machine).

Sui container è possibile effettuare il deployment di una qualsiasi applicazione, analogamente alle altre macchine; il vantaggio dei container è quello di permettere la condivisione, fra più applicazioni, della stessa macchina, garantendo, comunque, un elevato grado di isolamento.

Charm e Application

Le applicazioni gestibili da Juju sono distribuite attraverso dei pacchetti, chiamati charm. Un charm è un pacchetto software che contiene, oltre all'applicazione stessa, anche la logica necessaria per reagire a determinati eventi, che potrebbero consistere, ad esempio, nell'avvio dell'installazione, nel cambiamento di una configurazione o nell'aggiunta di una relazione. Quando il modello viene modificato, o l'amministratore specifica un'azione, i vari charm reagiranno di conseguenza, eseguendo il codice relativo all'evento accaduto. Questo permette ai charm di gestire autonomamente le specifiche operazioni di deployment o di manutenzione, in base all'applicazione che ospitano.

Ogni charm è caratterizzato da:

Logica applicativa L'insieme di eseguibili e script che implementano o permettono di installare e configurare l'applicazione che il charm ospita.

Configurazioni Un charm può esporre un insieme di configurazioni, che permettono di riconfigurare l'applicazione che ospitano. Tali configurazioni sono specificate nel relativo modello Juju. Al variare di una configurazione, il charm reagisce a tale evento ed avvia le operazioni di riconfigurazione dell'applicazione.

Risorse L'applicazione che il charm ospita potrebbe avere la necessità di accedere a risorse esterne, come, ad esempio, archivi o pacchetti snapcraft. Al momento della definizione del modello, è possibile specificare tali risorse. Poiché tali risorse sono esterne al charm, è possibile modificarle o sostituirle senza la necessità di ricompilare il charm.

Azioni Ogni charm può esporre un insieme di azioni, che potrebbero permettere, ad esempio, di eseguire dei comandi relativi all'applicazione che ospitano. Tali azioni possono essere invocate dall'amministratore attraverso la CLI.

Relazioni Ogni charm può esporre un insieme di relazioni, che permettono al charm di associarsi con altri charm, riconoscendosi reciprocamente, ed avviando un processo di configurazione automatica.

I charm vengono distribuiti attraverso un apposito servizio offerto da Juju, il Charm Store, che mantiene sia i charm ufficiali sia quelli realizzati dalla community. Come comportamento di default, Juju reperirà i charm necessari da tale sorgente. Alternativamente, è anche possibile specificare, al momento della definizione del modello, un charm salvato localmente.

Il charm store mette anche a disposizione dei bundle, cioè dei modelli composti da più charm, preconfigurati e relazionati tra loro, che realizzano una determinata infrastruttura. Tali bundle vengono distribuiti come un qualsiasi modello preconfigurato, cioè attraverso un file YAML che specifica i vari charm, le configurazioni, le relazioni e le macchine necessarie.

Nei modelli di Juju viene fatto riferimento anche al termine applicazione; in generale, i termini applicazione e charm sono da considerare quasi come sinonimi: un'applicazione è associata ad uno ed un solo charm.

Application Unit

Al momento del deployment, si potrebbero desiderare più istanze della stessa applicazione. Juju modella tale necessità attraverso le application unit. Un'application unit rappresenta una singola istanza dell'applicazione, che potrà essere avviata su una determinata macchina. Sulle macchine, infatti, è possibile effettuare il deployment delle unit e non direttamente delle applicazioni. Ad un'applicazione possono essere associate più unit, distribuite su macchine differenti. Una macchina può comunque ospitare più unit di applicazioni differenti. Tutte le unità di un'applicazione

condivideranno lo stesso charm, le stesse relazioni, le stesse configurazioni e le stesse risorse.

Tra le varie unità di un'applicazione, Juju elegge un leader, che fungerà da sorgente autoritaria per lo stato dell'applicazione e della configurazione.

Juju permette anche di eseguire operazioni di scaling di un'applicazione, aggiungendo o rimuovendo le relative unità e inoltre mette a disposizione il concetto di charm subordinato [34]. Un charm subordinato deve essere relazionato ad un charm principale. Non è possibile effettuare direttamente il deployment di unità relative a charm subordinati. Le unità del charm subordinato verranno create parallelamente alla creazione delle unità del charm principale, con corrispondenza uno ad uno.

Relazioni, interfacce ed endpoint

Juju permette di relazionare più applicazioni tra loro, in modo che queste possano giungere a conoscenza della loro presenza reciproca e possano configurarsi di conseguenza. Ogni charm, avrà una logica personalizzata per reagire e riconfigurarsi al momento della definizione di una relazione. Il concetto alla base delle relazioni è quello di interfaccia. L'interfaccia consiste nel protocollo di comunicazione utilizzato dai charm relazionati tra loro. Due charm differenti, potrebbero essere predisposti per comunicare tra loro, attraverso tale particolare interfaccia. In un'interazione di questo tipo, solitamente, un charm fornisce un servizio mentre l'altro lo utilizza. L'interfaccia deve quindi specificare la logica di comunicazione per tre tipi di interazione, chiamati ruoli [34]:

Requires Il charm richiede un particolare servizio da parte di un altro charm.

Provides Il charm può offrire un particolare servizio ad un altro charm.

Peers Due charm possono scambiarsi informazioni tra loro in modalità peer-to-peer; questo ruolo può essere utilizzato solo fra unità della stessa applicazione.

La terna costituita da ruolo, nome ed interfaccia definisce un endpoint. Ogni charm può possedere più endpoint. Rispettando i ruoli e le interfacce, cioè associando endpoint di tipo requires con endpoint di tipo provides, che utilizzano la stessa interfaccia, è possibile relazionare i vari charm. Gli endpoint possiedono anche un nome, poiché il charm potrebbe possedere più coppie ruolo-interfaccia identiche, ma aventi scopi differenti.

Una relazione non deve essere considerata come sinonimo di comunicazione di rete: una relazione permette infatti a due charm di scambiarsi informazioni in maniera ben definita, ma solitamente tali informazioni sono necessarie solamente per le operazioni di configurazione. Ad esempio, potrebbero consistere nell'indirizzo di rete o nelle credenziali per accedere ad un determinato servizio. Successivamente alla

configurazione, il charm potrà accedere ai servizi offerti da un altro charm tramite una normale comunicazione di rete, esterna al concetto di relazione.

Agenti

Juju avvia, su ogni macchina che gestisce, dei software di monitoraggio e gestione, chiamati agenti. Juju utilizza due tipi di agenti [34]:

Machine agent Opera a livello della macchina, monitorando e gestendo le sue risorse; permette di creare eventuali container ed avvia gli unit agent.

Unit agent Opera a livello della singola application unit ed è responsabile di tutte le attività relative alla singola applicazione.

Il controller Juju, per monitorare lo stato del modello o eseguire particolari azioni, comunica con gli agenti in esecuzione sulle varie macchine.

4.4 Terraform

Terraform è uno strumento di Infrastructure as Code open-source, realizzato da HashiCorp. Terraform permette di automatizzare le operazioni di allocazione e configurazione, tramite un apposito linguaggio dichiarativo, di un insieme di risorse, messe a disposizione da uno o più cloud provider.

A differenza di Juju, che può effettuare direttamente il provisioning di un'infrastruttura, Terraform si limita a sfruttare i servizi di provisioning, o di altro tipo, offerti da un determinato cloud provider.

Terraform può comunicare con i servizi cloud attraverso delle componenti software distinte, chiamate provider. Un provider abilita la comunicazione con un determinato servizio cloud, rendendo disponibili i possibili template di risorse che tale cloud può istanziare.

Attraverso uno o più file di configurazione, definiti attraverso un linguaggio dichiarativo specifico, HCL (HashiCorp Configuration Language), è possibile dichiarare le risorse che si intende istanziare, definendo, per ognuna, le specifiche configurazioni. Terraform utilizzerà tali file per allocare, automaticamente, tutte le risorse dichiarate sui rispettivi cloud provider. HCL, inoltre, mette a disposizione alcuni costrutti, come variabili o espressioni condizionali, offrendo la possibilità di configurare le varie risorse dinamicamente o di organizzarle in porzioni di codice riutilizzabili, chiamate moduli.

Terraform, per operare, non richiede alcun tipo di infrastruttura aggiuntiva: è esclusivamente il client Terraform che istanzia le risorse, comunicando direttamente con il cloud provider.

Il vantaggio che offre Terraform è la possibilità di definire e gestire l'intera infrastruttura, composta da molteplici risorse relative ad uno o più cloud provider, attraverso un insieme di file di configurazione. Tali file garantiscono tutti i vantaggi derivanti dal modello IaC (Infrastructure as Code), come la ripetibilità ed il controllo versione.

Il limite di Terraform è definito dai cloud-provider: non potranno infatti essere istanziate risorse che il cloud-provider non può offrire.

4.4.1 Comandi e Workflow

Terraform, per operare, necessita di due componenti: il client Terraform, cioè l'eseguibile utilizzabile tramite CLI che implementa tutta la logica applicativa, ed i file di configurazione, che definiscono i cloud provider da utilizzare e le risorse da creare.

Il client Terraform mette a disposizione cinque comandi principali, che dovranno essere eseguiti nella cartella contenente i file di configurazione:

init Inizializza la cartella, scaricando i file aggiuntivi necessari per utilizzare i provider specificati nei file di configurazione; è il primo comando da eseguire dopo aver scritto i file di configurazione.

validate Verifica la presenza di eventuali errori di sintassi nei file di configurazione.

plan Mostra quali azioni verrebbero effettuate, cioè quali risorse verrebbero create, modificate o rimosse, se venisse applicata la configurazione sullo stato attuale dell'infrastruttura.

apply Permette di applicare la configurazione attuale sull'infrastruttura. Prima di procedere, mostra l'output del comando plan e richiede esplicita conferma.

destroy Distrugge l'intera infrastruttura, rimuovendo tutte le risorse specificate dai file di configurazione. Prima di procedere richiede esplicita conferma

Attraverso tali comandi, è possibile usufruire del workflow tipico di Terraform, composto dalle seguenti fasi [125]:

Write Si descrive l'infrastruttura che si vuole realizzare attraverso dei file di configurazione definiti tramite il linguaggio HCL (HashiCorp Configuration Language);

Plan ogniqualvolta si effettua una modifica ai file di configurazione, si esegue il comando plan per verificare quali modifiche saranno applicate, prima di apportarle realmente all'infrastruttura;

Apply si esegue il comando apply per applicare lo stato desiderato, specificato nei file di configurazione, all'infrastruttura.

4.4.2 Providers

Un provider è un componente software, esterno al Terraform client, che implementa la logica per interagire con un determinato cloud provider. Oltre alla logica, il provider fornisce tutti i template necessari per definire le risorse relative a quel determinato cloud.

I provider che si desidera utilizzare devono essere dichiarati nei file di configurazione, tramite la parola chiave provider, seguita dal nome che lo identifica. Nel blocco di codice relativo al provider, è possibile specificare ulteriori configurazioni specifiche per quel provider, come, ad esempio, le credenziali di accesso o la regione del cloud nella quale le risorse verranno create.

Terraform supporta una moltitudine di provider, non solo per interfacciarsi a servizi cloud, ma anche relativi a servizi o protocolli generici. Alcuni di questi, sono, ad esempio:

- Amazon Web Services (AWS)
- Microsoft Azure
- Google Cloud Platform
- Kubernetes
- HTTP
- DNS

Ad esempio, il provider Kubernetes permetterà di definire ed allocare le risorse relative a Kubernetes, mentre il provider HTTP, permetterà di ottenere il contenuto di una pagina HTTP. La possibilità di combinare le risorse messe a disposizione da provider differenti permette di definire infrastrutture molto articolate e complesse.

La lista completa dei provider è visualizzabile nel Terraform Registry [127], il servizio messo a disposizione da Terraform che mantiene e documenta tutti i provider disponibili. I provider potrebbero essere sia ufficiali, cioè realizzati da HashiCorp, sia realizzati dalla community. La possibilità, per la community, di realizzare provider, ha permesso di estendere notevolmente il grado di compatibilità offerto da Terraform.

4.4.3 HashiCorp Configuration Language

HCL (HashiCorp Configuration Language) è il linguaggio dichiarativo utilizzato per definire i vari file di configurazione. In realtà, sarebbe possibile definirli anche attraverso dei file JSON, anche se, tale modalità, è sconsigliata, poiché JSON è caratterizzato da una sintassi molto prolissa e non specifica per IaC.

I file di configurazione HCL, per essere riconosciuti dal client Terraform, devono possedere l'estensione ".tf". HCL permette di dichiarare varie entità:

Provider (provider) Attraverso la parola chiave provider è possibile specificare il cloud provider da utilizzare.

Risorsa (resource) Le risorse sono le entità principali per la definizione di un'infrastruttura. Il tipo di risorsa e, conseguentemente, le relative proprietà specificabili, dipendono dal cloud provider utilizzato. Una risorsa potrebbe essere, ad esempio, un server, un dispositivo di rete o una configurazione, in base a quello che il servizio cloud permette di allocare.

Sebbene provider distinti possano offrire risorse simili, ad esempio, macchine server, non è possibile definire un codice generico che possa essere applicato a cloud differenti: nonostante il tipo di risorsa sia simile, cloud provider differenti, solitamente, hanno le proprie modalità per gestire ed offrire tali risorse.

Attraverso la parola chiave resource è possibile specificare una particolare risorsa, che verrà allocata sul relativo cloud provider al momento dell'apply. All'interno del blocco di codice relativo alla risorsa, è possibile poi specificare gli eventuali parametri di configurazione, che potranno anche essere generati dinamicamente sfruttando le potenzialità del linguaggio HCL.

Sorgente dati (data) Attraverso la parola chiave data, è possibile definire una sorgente dati, cioè un insieme di valori e proprietà, accessibili all'interno del codice. Il tipo di sorgente dati e, conseguentemente, le relative proprietà specificabili, dipendono dal cloud provider utilizzato.

Solitamente, le informazioni fornite da una sorgente dati provengono direttamente dal cloud provider, e sono necessarie per operare determinate scelte. Ad esempio, il provider Kubernetes definisce una sorgente dati per ottenere la lista di tutti i namespace disponibili.

Moduli (module) Tramite HCL, è possibile organizzare il codice in componenti generici e riutilizzabili, denominati moduli. Solitamente un modulo contiene varie risorse, che, combinate tra loro, realizzano una risorsa di più alto livello. Se tale risorsa di alto livello è stata modellata correttamente attraverso un modulo, sarà possibile riutilizzarla, adattandola alle specifiche necessità.

In realtà, tutto il codice Terraform è un modulo: infatti, un modulo è semplicemente una cartella contenente dei file di configurazione; non è però garantito che tale codice sia stato definito in maniera generica e riutilizzabile.

Attraverso la parola chiave module, è possibile istanziare un particolare modulo, fornendogli i parametri necessari.

Variabili di input (variable) Attraverso la parola chiave variable, è possibile definire delle variabili inizializzabili dall'esterno; che potrebbero essere inizializzate

tramite il comando apply, specificando i relativi valori. Questo permette di definire delle infrastrutture configurabili, senza la necessità di modificare i file di configurazione. Le variabili di input sono solitamente utilizzate dai moduli per ottenere i vari parametri di configurazione dall'esterno.

Valori di output (output) Attraverso la parola chiave output, è possibile specificare dei valori che verranno esposti all'esterno, successivamente all'operazione di apply; ad esempio, si potrebbe fornire all'esterno l'indirizzo di rete che un particolare server ha ottenuto successivamente alle operazioni di allocazione. I valori di output sono solitamente utilizzati dai moduli per restituire determinate informazioni ai propri utilizzatori.

Valori locali (locals) Attraverso la parola chiave locals è possibile dichiarare un insieme di valori o espressioni, che potranno essere utilizzate all'interno del codice per evitare ripetizioni o per renderlo facilmente configurabile.

Espressioni e funzioni HCL mette a disposizione una serie di espressioni logiche, operatori o funzioni, che possono essere utilizzate all'interno del codice per generare, dinamicamente, determinati valori o blocchi di codice.

4.4.4 Stato

Terraform, ogni volta che viene eseguito, salva delle informazioni relative alle risorse che ha creato in un file di stato. Come impostazione di default, tale file viene mantenuto localmente, nella cartella contenente i file di configurazione, con il nome "terraform.tfstate".

Tale file registra la corrispondenza, in formato JSON, tra le risorse specificate nei file di configurazione e le risorse create sul cloud provider [20, cap. 3].

Questo permette a Terraform di comprendere quale sia lo stato attuale dell'infrastruttura, e, a fronte di modifiche del file di configurazione, di determinare su quali risorse del cloud provider intervenire.

L'utilizzo di un file di stato locale potrebbe essere accettabile per realizzare un progetto personale, ma, nel caso più sviluppatori dovessero collaborare, sarebbe necessario definire delle opportune modalità per condividere tale file. Per risolvere questo problema, Terraform permette di specificare il backend da utilizzare; in base a tale configurazione, è possibile specificare dove mantenere tale file di stato. Ad esempio, potrebbe essere salvato su particolari servizi cloud, come Amazon S3, Azure Storage o Google Cloud Storage.

Capitolo 5

Container orchestration

Come già affermato nel Capitolo 1, per facilitare la gestione delle risorse relative ad un cluster di elaboratori, è necessario adottare un cluster management software. Una categoria di cluster management software, attualmente molto diffusa, che permette il deployment e la gestione di applicazioni organizzate in microservizi, è quella dei container orchestrator. Per queste caratteristiche, si è deciso di adottare un cluster management software appartenente a tale categoria, per la realizzazione di questo progetto.

Nei capitoli 3 e 4, sono state introdotte le principali tecnologie per realizzare un cluster di macchine fisiche e per effettuare il deployment di un cluster management software; in questo capitolo verranno approfonditi i vantaggi e le funzionalità offerte da tali cluster management software, in particolare, dei container orchestrator.

Si andrà quindi a definire cosa si intende per container orchestration e quali siano le principali caratteristiche dei cluster management software che offrono tali funzionalità; saranno quindi individuate le principali alternative attualmente disponibili.

Infine, verranno descritte le principali caratteristiche e funzionalità offerte dal software di container orchestration scelto per questo progetto: Kubernetes.

5.1 Definizione

I container orchestrator sono particolari cluster management software, che permettono di gestire le risorse di un cluster in modo tale da renderle facilmente utilizzabili da applicazioni containerizzate, cioè composte da container. Solitamente, tali applicazioni, adottano un'architettura a microservizi, già descritta nella Sezione 1.6.

Un container è un pacchetto software che offre un grado di isolamento e portabilità simile a quello delle macchine virtuali, ma che introduce un overhead, in termini di prestazioni, molto inferiore. Questi vantaggi hanno reso tale tecnologia ampiamente adottata nell'ambito delle applicazioni composte da microservizi, poiché, solitamente,

ogni microservizio necessita della propria indipendenza e del proprio ambiente di esecuzione. In generale, l'utilizzo dei container per il deployment delle applicazioni comporta una serie di vantaggi:

Portabilità Il container ospita, oltre all'applicazione, anche l'intero ambiente di esecuzione, composto dalle librerie di sistema, dagli strumenti e, in generale, da tutto il software necessario all'applicazione. Questo permette l'esecuzione dello stesso container, in maniera trasparente all'applicazione, su una qualsiasi piattaforma hardware.

In realtà, si presentano, comunque, dei limiti [68, pag. 15]: i container condividono il kernel Linux della macchina su cui vengono eseguiti; se l'applicazione avesse la necessità di particolari funzionalità offerte da una specifica versione del kernel, ma la macchina possedesse un kernel differente, si verificherebbe una situazione di incompatibilità.

Inoltre, i container non risolvono il problema della portabilità fra piattaforme hardware con architetture differenti: non sarà possibile, ad esempio, eseguire un container realizzato per architettura x86-64 su una macchina con architettura arm64.

In ogni caso, entrambi i problemi possono essere attenuati: il primo, tentando di mantenere sempre aggiornate le macchine, per offrire una versione recente del kernel Linux; il secondo, compilando le immagini dei container sia per x86-64 che per arm64. Al momento del deployment, il sistema individuerà l'immagine da utilizzare sulla base dell'architettura della macchina.

Prestazioni Le macchine virtuali comportano un notevole spreco di risorse, causato dalla necessità di virtualizzare l'hardware e di eseguire, al loro interno, un ulteriore sistema operativo. Sicuramente, le macchine virtuali offrono il grado massimo di portabilità ed isolamento, ma non altrettanto può dirsi dal punto di vista delle prestazioni. I container non virtualizzano l'hardware né eseguono un proprio sistema operativo, poiché si basano su delle funzionalità di isolamento offerte dal kernel Linux della macchina. Questo permette di ottenere prestazioni comparabili a quelle di un'applicazione in esecuzione direttamente sulla macchina fisica.

Isolamento I container sfruttano un insieme di funzionalità di isolamento offerte direttamente dal kernel Linux, chiamate Linux Namespaces e Linux Control Groups (cgroups) [68, pag. 11]. I Linux Namespaces permettono di isolare un determinato processo, fornendogli una vista ridotta delle risorse del sistema, come, ad esempio, dei file, degli altri processi o delle interfacce di rete. I Linux Control Groups permettono invece di limitare la quantità di risorse, in termini di CPU, memoria o accesso alla rete, che un determinato processo può utilizzare.

Distribuzione Le applicazioni containerizzate possono essere distribuite facilmente e velocemente attraverso la creazione di immagini, che possono essere condivise sfruttando degli appositi servizi di registry. Un'immagine include, al suo interno, un intero file system, che mantiene i file relativi all'applicazione ed al suo ambiente di esecuzione. Da un'immagine è possibile avviare un container, che è un processo isolato attraverso le funzionalità del kernel. Un'immagine può quindi essere considerata come un modello, accessibile in sola lettura, attraverso la quale è possibile avviare uno o più container, cioè istanze della stessa applicazione. Ogni container avviato dalla stessa immagine rimarrà, comunque, isolato dagli altri, e potrà possedere una configurazione differente; il container potrà accedere ad un qualsiasi file contenuto nell'immagine, ma le eventuali modifiche rimarranno ad esso esclusive. Il formato di tali immagini è stato definito, in maniera standard, dalla Open Container Initiative.

Per effettuare la distribuzione di tali immagini, vengono sfruttati particolari servizi, chiamati registry. Un registry è un repository che mantiene le varie immagini, permettendone la condivisione. Uno sviluppatore può realizzare l'immagine di una determinata applicazione e caricarla sul registry; successivamente, chiunque vorrà eseguire tale applicazione, potrà scaricare la relativa immagine e creare il container. Esistono sia servizi di registry pubblici, cioè utilizzabili da chiunque senza la necessità di autenticarsi per ottenere le immagini, come Docker Hub, sia privati.

Per poter avviare e gestire applicazioni containerizzate su una determinata macchina è necessaria la presenza di un container runtime engine, come Docker, containerd, cri-o o LXC, cioè un software permetta di automatizzare tutte quelle operazioni relative alla gestione delle immagini e dei container.

Per poter avviare e gestire applicazioni containerizzate su più nodi è invece necessario adottare un container orchestrator.

I container orchestrator offrono dei servizi che consentono di gestire l'intero ciclo di vita dei container di un'applicazione, permettendone il deployment sui vari nodi del cluster, la gestione ed il monitoraggio. In generale, i container orchestrator offrono le seguenti funzionalità:

Deployment L'orchestratore ha il controllo dei vari container runtime engine in esecuzione sui vari nodi; questo gli permette di avviare un particolare container su uno qualsiasi dei nodi presenti nel cluster.

Scheduling L'orchestratore può scegliere su quale nodo istanziare un particolare container in base alle risorse richieste da esso ed a quelle disponibili.

High Availability L'orchestratore può monitorare lo stato dei vari container in esecuzione; se uno di questi fallisse, potrebbe eseguire un'operazione di riavvio automatico.

Bilanciamento del carico L'orchestratore può avviare più istanze dello stesso container, potenzialmente su nodi differenti, e distribuire le richieste provenienti dall'esterno, in maniera equa, tra di loro.

Migrazione L'orchestratore può migrare determinati container, da un nodo ad un altro, in maniera trasparente, per fare fronte a malfunzionamenti o per esigenze di scheduling; a seguito della migrazione, le configurazioni o gli eventuali dati a cui il container ha accesso potranno rimanere invariati.

Scaling L'orchestratore può variare, dinamicamente, il numero di istanze di container in esecuzione, per fare fronte ad eventuali variazioni di carico;

Facilitare l'interazione tra container L'orchestratore può mettere a disposizione meccanismi per facilitare l'individuazione reciproca e la comunicazione tra i container relativi alla stessa applicazione.

Esposizione dei servizi L'orchestratore può mettere a disposizione dei meccanismi per permettere, ad uno o più container, di esporre i propri servizi all'esterno della rete relativa al cluster.

Storage persistente L'orchestratore può mettere a disposizione dei meccanismi per permettere, ai vari container, di memorizzare persistentemente i propri dati, che rimarranno disponibili anche a fronte di riavvi o migrazioni.

Configurazione delle applicazioni L'orchestratore può mettere a disposizione dei meccanismi per permettere la configurazione delle applicazioni ospitate dai vari container, ad esempio, permettendo di specificare i parametri di invocazione o determinate variabili d'ambiente.

5.2 Principali alternative

Attualmente, esistono vari software che offrono funzionalità di orchestrazione di container. Alcuni dei principali, già parzialmente descritti nella Sezione 1.4.2, sono i seguenti:

Kubernetes Orchestratore di container open-source che offre meccanismi per il deployment, la manutenzione e lo scaling di applicazioni; progettato, originariamente da Google [59].

Docker Swarm Orchestratore di container che permette di raggruppare più Docker engine in esecuzione su nodi diversi in un singolo Docker engine virtuale [1][42].

HashiCorp Nomad Un semplice e flessibile orchestratore che permette il deployment e la gestione di applicazioni sia containerizzate che non containerizzate, su infrastrutture sia cloud che on-premises [95].

Apache Marathon Una piattaforma di orchestrazione di container per Apache Mesos. Offre varie funzionalità, tra cui, High Availability, differenti container runtime, applicazioni stateful, service discovery, load balancing e monitoraggio [86].

CoreOS Fleet Il progetto è attualmente deprecato, ma permetteva di aggregare più macchine i un singolo insieme di risorse, sul quale potevano essere avviati servizi sotto forma di container. Fleet poteva gestire i fallimenti delle macchine automaticamente e permetteva un efficiente riutilizzo delle risorse [36].

Sebbene, attualmente, siano disponibili varie alternative, Kubernetes è diventato de facto, negli ultimi anni, lo standard nell'ambito dell'orchestrazione di container. Molte soluzioni alternative, tra cui, ad esempio, CoreOS Fleet, sono state deprecate in favore di Kubernetes; altre soluzioni, come Docker Swarm, Nomad e Marathon sono, invece, attivamente manutenute, ed offrono particolari vantaggi, ma non arrivano, sicuramente, al grado di diffusione e maturità che Kubernetes ha raggiunto.

Kubernetes risulta quindi la scelta più adatta, e, quasi obbligata, per la realizzazione di questo progetto.

5.3 Kubernetes

Kubernetes è un sistema open-source per la gestione di applicazioni containerizzate su un insieme di macchine, fornendo meccanismi che permettono il deployment, la manutenzione e lo scaling di tali applicazioni [59]. Il progetto Kubernetes derivò da alcune soluzioni software realizzate, originariamente, da Google.

Google fu, probabilmente, una delle prime aziende che dovette affrontare le problematiche legate al deployment ed alla gestione di un'innumerevole quantità di componenti software su migliaia di server, per poter offrire i propri servizi. Per questo, si rese conto della necessità di un sistema che facilitasse e velocizzasse tali operazioni su una così ampia scala. Nel corso degli anni, Google sviluppò una soluzione interna, chiamata Borg, e, successivamente, un altro sistema chiamato Omega [68, pag. 16]. Tali soluzioni, permettevano agli sviluppatori ed agli amministratori di gestire, facilmente, un deployment di applicazioni su larga scala e di utilizzare, più efficientemente, le risorse dell'infrastruttura.

Borg e Omega furono mantenuti segreti per circa un decennio, fino al 2014, quando Google introdusse Kubernetes, un progetto open-source basato sull'esperienza acquisita tramite tali progetti [68, pag. 16].

Kubernetes permette di eseguire un insieme di applicazioni containerizzate, distribuendole, automaticamente, su centinaia o migliaia di macchine, come se fossero un singolo enorme computer [68, pag. 17]. Consente di astrarre dall'infrastruttura sottostante, semplificando le operazioni di sviluppo e di deployment: le modalità rimarranno le medesime, ad esempio, sia nel caso in cui il cluster sia composto da 10 nodi sia in quello in cui sia composto da 100 nodi; in quest'ultimo caso, l'unica differenza, consisterà nella disponibilità di una maggiore quantità di risorse. Per offrire tale astrazione, Kubernetes modella un insieme di concetti, chiamati oggetti Kubernetes, che permettono di definire le risorse o i comportamenti relativi al deployment di una determinata applicazione containerizzata. Gli oggetti Kubernetes necessari per un determinato deployment possono essere descritti in un file YAML, che può essere interpretato da Kubernetes per realizzare tale deployment.

5.3.1 Architettura

Un'infrastruttura Kubernetes adotta il modello cluster computing, cioè un insieme di elaboratori, possibilmente costituiti da hardware generico, interconnessi tra loro tramite una rete locale (LAN). Ogni elaboratore costituisce un nodo del cluster, e mette a disposizione le proprie risorse hardware, in termini di CPU, RAM e storage. Kubernetes utilizzerà tali risorse per eseguire, in maniera distribuita sui vari nodi, le varie applicazioni containerizzate. Un'infrastruttura Kubernetes è composta da due tipologie di nodi: il Kubernetes Master ed il Kubernetes Worker.

Kubernetes Master (Kubernetes Control Plane)

Il Kubernetes Master è il nodo che gestisce e controlla l'intero cluster Kubernetes, rendendolo funzionante. Esso è composto da vari componenti, che possono essere eseguiti su un singolo nodo master o distribuiti su più nodi e, eventualmente, replicati, per garantire tolleranza ai guasti (high availability). Tali componenti sono i seguenti:

Database distribuito (etcd) Un database chiave valore che permette il salvataggio dei dati in maniera distribuita, garantendo un'elevata tolleranza ai guasti; viene utilizzato per salvare tutte le informazioni relative alla configurazione del cluster e relative ai vari oggetti Kubernetes. Viene utilizzato esclusivamente dall'API server per salvare o reperire le informazioni necessarie.

API server Il componente centrale utilizzato da tutti gli altri componenti e da eventuali software client esterni, come kubectl [68, pag. 316]; offre la modalità principale per interagire con il cluster Kubernetes. Mette a disposizione un insieme di API HTTP RESTful che permettono di creare, leggere, aggiornare o cancellare (CRUD) l'intero stato del cluster, composto dai vari oggetti Kubernetes. Tali informazioni vengono gestite attraverso etcd.

Scheduler Effettua le scelte relative alla distribuzione delle applicazioni sui vari nodi del cluster, considerando le risorse disponibili sui vari nodi e le risorse richieste dai container dell'applicazione. Non si occupa direttamente di allocare un container di un'applicazione su un determinato nodo, ma si limita a salvare le proprie scelte di scheduling sull'API server [68, pag. 319]; saranno i nodi worker del cluster che reagiranno al cambiamento di tali informazioni, allocando adeguatamente i vari container.

Controller Manager L'API server e lo scheduler sono componenti passivi, si limitano cioè a mantenere e modificare lo stato dei vari oggetti Kubernetes. Sono quindi necessari dei componenti attivi, che effettuino delle operazioni di controllo, per garantire che lo stato delle varie risorse del cluster converga a quello specificato nell'API server [68, pag. 321]. Tali componenti, chiamati controller, sono raggruppati ed eseguiti dal Controller Manager. Ogni controller effettua operazioni specifiche per la gestione di una determinata categoria di oggetti Kubernetes.

Kubernetes Worker

Il Kubernetes Worker è un nodo sul quale verranno eseguiti i container relativi ad una determinata applicazione. Ogni nodo worker, sotto la guida del nodo master, eseguirà tutte le operazioni di esecuzione e monitoraggio relative ai container che gestisce, tramite i seguenti componenti:

Kubelet Il componente responsabile di ogni risorsa presente sul nodo worker [68, pag. 326]. Comunica con l'API server del nodo master, registrandosi, reagendo ad eventi di scheduling ed avviando i relativi container, ed inviando informazioni di monitoraggio relative ai container in esecuzione. Inoltre, verifica, periodicamente, che tutti i vari container siano in salute, riavviandoli nel caso di malfunzionamenti; su indicazione dell'API server, può eliminare i container in esecuzione.

Kubernetes Service Proxy (kube-proxy) Componente che redirige le connessioni provenienti dall'esterno verso i container che offrono il servizio richiesto [68, pag. 327]. Nel caso esistano più container che offrono lo stesso servizio, il kube-proxy effettua un'operazione di load balancing. Questo componente è necessario per la definizione dei servizi, cioè particolari endpoint a cui eventuali client esterni possono connettersi, per usufruire, trasparentemente, dei servizi offerti dalle applicazioni in esecuzione su un insieme di container.

Container Runtime (containerd, Docker o altri) Il software che permette l'esecuzione e la gestione, in maniera semplificata, sulla singola macchina, dei

vari container. Si occupa di tutte quelle operazioni che riguardano il reperimento e la gestione delle immagini, permettendo di creare e configurare, da esse, le istanze dei vari container. Sono software esterni dal progetto Kubernetes, genericamente adottati, in vari contesti, per la gestione di container su una singola macchina.

Deployment di applicazioni

Per eseguire un'applicazione su Kubernetes, è prima necessario organizzarla in un insieme di immagini di container; se l'applicazione è stata progettata e sviluppata sulla base di un'architettura a microservizi, tale operazione sarà triviale. Successivamente, è necessario caricare tali immagini su un servizio di registry, in modo tale che i vari nodi worker del cluster abbiano la possibilità di ottenere tali immagini.

A tal punto, è possibile definire una descrizione dell'applicazione, da sottoporre all'API server. Tale descrizione viene definita, solitamente, in formato YAML, delineando i vari oggetti Kubernetes necessari, che descriveranno le varie componenti e le risorse necessarie all'applicazione, tra cui, ad esempio, le immagini dei container da utilizzare, come i vari container saranno raggruppati, quali servizi andranno ad esporre e quante repliche dovranno essere create.

In realtà, l'API server accetta solamente descrizioni in formato JSON, ma, solitamente, viene utilizzato uno strumento a riga di comando, chiamato kubectl, che permette di definire le descrizioni da applicare in formato YAML. Sarà kubectl a convertire automaticamente tali descrizioni in formato JSON. Kubectl fornisce, inoltre, un insieme di comandi per interagire facilmente e velocemente con l'API server di un cluster Kubernetes.

Quando tale descrizione verrà fornita all'API server, saranno creati i relativi oggetti Kubernetes. I vari controller faranno quindi convergere lo stato del cluster a quello specificato nella descrizione. Se gli oggetti Kubernetes specificati prevedevano l'allocazione di nuovi container, lo scheduler allocherà i vari gruppi di container su determinati nodi worker, in base alle risorse disponibili. A tal punto, il componente Kubelet di ogni nodo worker selezionato impartirà al container runtime di reperire una determinata immagine e di creare il relativo container.

5.3.2 Principali oggetti Kubernetes

Kubernetes, per permettere di definire le modalità di deployment e le risorse necessarie alle applicazioni, modella un insieme di oggetti, che possono essere creati e gestiti attraverso l'API server. I principali, sono:

Pod Il pod è l'unità fondamentale per il deployment di un'applicazione containerizzata. Kubernetes non permette infatti di effettuare direttamente il

deployment di container, ma lo permette solamente attraverso dei pod, cioè delle entità che possono contenere uno o più container. I container appartenenti allo stesso pod verranno sempre istanziati sullo stesso nodo, come se fossero una singola entità. Questo meccanismo permette di raggruppare tutti quei container che devono eseguire componenti dell'applicazione fortemente legate fra loro, e che, quindi, necessitano di comunicare come se fossero eseguiti sulla stessa macchina. I vari container dello stesso pod condivideranno lo stesso ambiente di esecuzione, come se i relativi processi venissero eseguiti all'interno dello stesso container, ma rimanendo, comunque, in qualche modo, isolati [68, pag. 57]. Questo permette ai processi dei vari container, ad esempio, di comunicare tramite IPC (Inter-Process Communication) o tramite l'interfaccia di rete "localhost".

I container di un'applicazione dovrebbero essere raggruppati in un unico pod solamente nel caso ve ne fosse una reale necessità; se così non fosse, sarebbe sempre consigliabile creare un pod differente per ogni container dell'applicazione.

Namespace Un namespace permette di raggruppare tra loro un insieme di oggetti Kubernetes, realizzando una sorta di cluster virtuale. Gli oggetti appartenenti ad un determinato namespace saranno logicamente separati da tutti gli oggetti appartenenti agli altri namespace, come se appartenessero ad un differente cluster. Questo permette, ad esempio, di mantenere separati gli oggetti relativi ad applicazioni distinte, o di realizzare determinati ambienti di testing.

In realtà, non tutti gli oggetti Kubernetes sono associabili ad un namespace: alcuni, come ad esempio i nodi o i volumi, sono globali. Infatti, pod di namespace diversi potrebbero essere schedulati sullo stesso nodo, oppure, potrebbero utilizzare uno stesso volume [123, pag. 11]. I pod di un determinato namespace hanno, comunque, la capacità di accedere ai servizi offerti da pod di altri namespace; nel caso si desiderasse un isolamento completo, sarebbe necessario configurare specifiche politiche di rete, adottando specifici CNI (Container Network Interface).

Label, annotazioni e selettori Le label sono delle coppie chiave valore che permettono di raggruppare tra loro un insieme di oggetti, solitamente i pod [123, pag. 8]. Una label specifica una proprietà che permette di classificare l'oggetto, ad esempio, per determinare se appartenga o meno ad un determinato gruppo. Il formato di una label deve rispettare, inoltre, determinate caratteristiche, per permettere di essere valutata velocemente. Tali informazioni sono molto importanti per tutti quegli oggetti, come i ReplicaSet o i Deployment, che operano su gruppi di oggetti dinamici e necessitano quindi di identificare i membri di tali gruppi [123, pag. 8].

Le annotazioni sono simili alle label, ma permettono di associare all'oggetto dei metadati che non verranno utilizzati direttamente da Kubernetes; per questo, le annotazioni non devono rispettare alcun tipo di formato specifico.

Infine, un label selector permette di specificare le condizioni, su un insieme di label, che devono essere rispettare per rientrare in un determinato gruppo. I label selector sono utilizzati, ad esempio, dai ReplicaSet o dai Deployment per determinare i membri del gruppo sul quale devono operare.

ReplicationController e ReplicaSet Sia i ReplicationController che i ReplicaSet gestiscono un gruppo di pod identificato da un particolare label selector, assicurando che un determinato numero di repliche di quel determinato gruppo sia sempre in esecuzione [123, pag. 10]. Nel caso il numero di pod in esecuzione fosse inferiore al numero desiderato, a causa, ad esempio, di un malfunzionamento di un nodo, il ReplicationController o il ReplicaSet creerebbero, automaticamente, nuovi pod, attraverso uno specifico template. Le repliche che gestiscono hanno esattamente le stesse caratteristiche e sono perfettamente intercambiabili tra loro, poiché, solitamente, sono stateless, cioè non mantengono uno stato; per questo, possono essere aggiunte ed eliminate automaticamente senza particolari accorgimenti.

I ReplicationController ed i ReplicaSet offrono le stesse funzionalità, ma i ReplicaSet sono considerati un'evoluzione dei ReplicationController e destinati a sostituirli. I ReplicaSet, a differenza dei ReplicationController, permettono una maggiore espressività nella definizione del label selector.

DaemonSet I DaemonSet sono simili ai ReplicaSet, ma, a differenza di questi ultimi, garantiscono l'esecuzione di una e una sola replica per ogni nodo del cluster, o, alternativamente, su tutti i nodi che rispettano un determinato selettore. I DaemonSet non utilizzano lo scheduler, poiché conoscono già dove allocare le repliche; se un nodo viene rimosso dal cluster, il DaemonSet non ricrea la replica eliminata. Al contrario, se un nodo viene aggiunto al cluster, il DaemonSet crea automaticamente una nuova replica su di esso.

Job e CronJob I ReplicaSet ed i DaemonSet permettono di replicare un insieme di pod che offrono un particolare servizio, cioè pod che rimangono costantemente in esecuzione. I Job permettono invece di creare un insieme di pod, il cui compito è quello di completare una determinata operazione, per poi terminare. Quando un pod appartenente al Job termina, non viene riavviato, come nel caso dei ReplicaSet, poiché si assume che il container abbia completato l'operazione che doveva eseguire. I pod di un Job sono invece riavviati nel caso si verificasse un fallimento, di un nodo o di un container relativi a tale Job.

Un CronJob è un Job che viene eseguito automaticamente con cadenza periodica.

ConfigMap e Segreti Una ConfigMap è un oggetto che permette di memorizzare un insieme di configurazioni, sotto forma di record chiave valore, che saranno accessibili da tutti i pod che la utilizzeranno. I valori di una ConfigMap potranno essere forniti ai processi relativi ai container di un pod attraverso variabili d'ambiente o attraverso un volume.

I Segreti sono simili alle ConfigMap e possono essere utilizzati allo stesso modo; offrono, però dei meccanismi aggiuntivi di sicurezza, che li rendono adatti per memorizzare informazioni sensibili, come, ad esempio, credenziali o certificati.

StatefulSet A differenza dei ReplicaSet, che gestiscono repliche identiche ed intercambiabili, gli StatefulSet permettono di gestire gruppi formati da pod contraddistinti da caratteristiche uniche. Ogni pod del gruppo ha cioè particolari caratteristiche, che devono essere mantenute anche a fronte di riavvi o operazioni di scaling. Per questo, i vari pod non sono intercambiabili tra loro, poiché ognuno possiede la propria identità. Ogni pod del gruppo è caratterizzato da un insieme di proprietà, come, ad esempio, l'hostname, l'indice del pod ed i volumi associati; per un determinato valore di tali proprietà, lo StatefulSet assicura la presenza di uno ed un solo pod. Non potrà accadere, ad esempio, che due pod con lo stesso hostname coesistano in uno stesso momento.

Questo tipo di oggetti è molto utile per gestire tutti quei pod che devono mantenere uno stato, come, ad esempio, i pod relativi ad un database distribuito. In questi casi, ogni pod deve poter essere identificato univocamente, in modo tale che esista la possibilità di riassociargli il relativo volume in caso di riavvi o operazioni di scaling.

Deployment Un Deployment è simile ad un ReplicaSet, ma offre funzionalità aggiuntive per gestire e controllare le operazioni di aggiornamento di un'applicazione. Tale operazione potrebbe consistere, ad esempio, nella modifica della versione dell'immagine relativa al container che i vari pod dovranno utilizzare. Un Deployment permette di automatizzare tali operazioni di aggiornamento, rimuovendo le vecchie repliche e creando quelle aggiornate.

Al momento della creazione di un Deployment, vengono creati uno o più ReplicaSet associati. I vari pod verranno gestiti da tali ReplicaSet e non direttamente dal Deployment. Al momento dell'aggiornamento, il Deployment coordinerà il ReplicaSet della vecchia versione e quello della nuova, per sostituire, gradualmente, le vecchie repliche con le nuove. Inoltre, il Deployment fornisce delle funzionalità di rollback: nel caso la nuova versione presentasse dei problemi, sarebbe possibile tornare, facilmente, ad una versione precedente.

5.3.3 Servizi e load balancing

Kubernetes offre un insieme di modalità per esporre, all'esterno o nei confronti di altri pod, i servizi offerti dalle applicazioni in esecuzione sui vari container. Tali modalità sono realizzate dagli oggetti Kubernetes chiamati servizi. Un servizio permette di definire un punto di accesso, singolo e permanente, nei confronti di un gruppo di pod, identificati da un selettore, che forniscono lo stesso servizio [68, pag. 121]. Ogni servizio è caratterizzato da un indirizzo IP e da un numero di porta, che rimarranno tali fino a quando il servizio non verrà rimosso. Gli eventuali client potranno connettersi a tale IP e porta; le relative connessioni verranno redirette verso uno dei pod che forniscono tale servizio [68, pag. 121]. In questo modo, i client non devono conoscere la locazione dei singoli pod, ma, solamente, l'indirizzo esposto dal servizio; questo permette a Kubernetes di creare, eliminare o migrare i vari pod in maniera trasparente ai vari client. I vari client devono cioè potersi connettere, allo stesso modo, ad un determinato servizio, sia nel caso in cui il servizio sia offerto da un singolo pod, sia nel caso sia offerto da centinaia o migliaia di pod, distribuiti su più nodi. L'indirizzo IP e la porta relativi ad un determinato servizio possono essere reperiti, dai pod che intendono utilizzarlo, tramite variabili d'ambiente o tramite il DNS.

Il comportamento di default dei servizi è quello di esporre il servizio offerto da un gruppo di pod a qualsiasi altro pod del cluster che volesse utilizzarlo. In questa modalità, chiamata ClusterIP, è possibile accedere a tale servizio esclusivamente attraverso la rete virtuale interna del cluster Kubernetes.

Alternativamente, è possibile esporre un servizio all'esterno, per permettere a client esterni al cluster Kubernetes di usufruire dei servizi offerti dai vari pod. Esistono tre alternative:

NodePort Il servizio viene esposto da ogni nodo del cluster su una determinata porta. Per usufruire del servizio è sufficiente connettersi ad uno dei nodi del cluster Kubernetes su tale porta. In questo caso il servizio è accessibile dall'esterno poiché esposto sull'indirizzo IP esterno del nodo, e non esclusivamente su un indirizzo della rete virtuale interna del cluster. I servizi NodePort sfruttano, comunque, un servizio ClusterIP, al quale vengono redirette le connessioni. Quando un client si connette ad un determinato nodo, la connessione viene rediretta al servizio ClusterIP, che la redirige su uno dei relativi pod; tale pod potrebbe essere in esecuzione su un nodo differente da quello che il client aveva originariamente contattato. Il servizio ClusterIP effettua un'operazione di load balancing tra i vari pod, ma, se tutti i client contattassero un singolo nodo del cluster per accedere al servizio, si verificherebbe, comunque, un collo di bottiglia. Per risolvere tale problema è necessario adottare un servizio di tipo LoadBalancer.

LoadBalancer Espone il servizio esternamente sfruttando un load balancer offerto dal cloud provider. Alla creazione di un servizio LoadBalancer viene creato, automaticamente, un servizio NodePort con il relativo sevizio ClusterIP. Il LoadBalancer del cloud provider distribuirà il carico su tutti i nodi del cluster che espongono il servizio tramite il relativo NodePort.

Ingress Mentre i servizi operano a livello di trasporto, cioè a livello del protocollo TCP o UDP, gli oggetti Ingress operano a livello applicativo, cioè HTTP. Questo permette offrire funzionalità aggiuntive, permettendo, ad esempio, di redirigere il traffico sulla base dell'URL presente nella richiesta.

Il servizio di tipo LoadBalancer risulta essere una delle scelte migliori per esporre, all'esterno, un particolare servizio. In questo caso deve essere presente, però, il supporto del cloud provider, che deve offrire il load balancer da utilizzare.

Un possibile load balancer: MetalLB

MetalLB è un'applicazione che può essere eseguita, come una qualsiasi altra applicazione containerizzata, su un cluster Kubernetes. Il deployment di tale applicazione è composto da vari oggetti Kubernetes, che forniscono un'implementazione di un load balancer basata su particolari funzionalità offerte dalla rete. Tale load balancer permette la definizione di servizi LoadBalancer anche su cluster che non hanno il supporto di un cloud provider per tale funzionalità.

MetalLB è adatto per tutti quei cluster Kubernetes che non si basano su un cloud provider, o che ne utilizzano uno che non offre risorse di tipo load balancer, come, ad esempio, un cluster bare-metal composto da nodi fisici non gestiti.

MetalLB, per operare, deve risolvere due problematiche: l'allocazione degli indirizzi e l'annuncio di tali indirizzi all'esterno [87].

L'indirizzo IP del load balancer, nel caso si utilizzasse un cloud provider, verrebbe allocato dall'infrastruttura cloud. Nel caso, invece, di MetalLB, è MetalLB stesso che deve occuparsi di tale allocazione. In generale, è possibile configurare MetalLB tramite una ConfigMap, specificando un range di indirizzi che MetalLB potrà utilizzare per creare i propri load balancer. Tali indirizzi dovranno essere dedicati per tale utilizzo, evitando eventuali conflitti.

Successivamente alla definizione degli indirizzi che MetalLB potrà utilizzare, è necessario annunciare la presenza di tali indirizzi all'esterno, in modo tale che l'infrastruttura di rete sia consapevole della presenza di tali indirizzi e che gli eventuali client possano utilizzarli per connettersi ai vari servizi. MetalLB offre due modalità per risolvere tale problema:

Layer 2 In questa modalità, una macchina del cluster ottiene la proprietà di un servizio, ed utilizza protocolli standard per la determinazione della

corrispondenza tra indirizzi fisici (MAC) ed indirizzi di rete (IP), come ARP (Address Resolution Protocol), per rendere raggiungibile l'indirizzo IP del load balancer. Infatti, se un eventuale client non ricevesse le informazioni ARP per ottenere l'indirizzo fisico, non potrebbe comunicare con l'indirizzo IP del load balancer. Dal punto di vista della rete, è come se il nodo che ha la proprietà del servizio possedesse un ulteriore indirizzo IP, pubblicato tramite ARP, sul quale riceve il traffico del servizio LoadBalancer.

Questa modalità non implementa, in realtà, un vero e proprio servizio di load balancing. Infatti, tutto il traffico del servizio viene inviato ad un singolo nodo del cluster, che redirigerà poi il traffico ai vari pod tramite i servizi offerti da Kubernetes. L'unico vantaggio che questa modalità offre, rispetto, ad esempio, ad un servizio NodePort, è il failover automatico: se il nodo che detiene la proprietà del servizio fallisse, MetalLB migrerebbe, automaticamente, le varie funzionalità di load balancing su un altro nodo, modificando, tramite ARP, l'indirizzo fisico associato all'indirizzo IP del load balancer.

Questa modalità è particolarmente vantaggiosa poiché può funzionare su una qualsiasi infrastruttura di rete, ma è caratterizzata da alcuni lati negativi: il nodo che detiene la proprietà del servizio limita la banda massima del load balancer, causando un possibile collo di bottiglia; inoltre, il failover potrebbe non essere istantaneo, poiché si basa sulle modalità con cui i sistemi operativi client gestiscono il protocollo ARP.

BGP Nella modalità BGP (Border Gateway Protocol), tutti i nodi del cluster avviano una sessione di peering BGP con i router dell'infrastruttura di rete, definendo come tali router devono inoltrare il traffico ai vari IP che offrono il servizio [87]. L'utilizzo di BGB permette un reale load balancing tra i vari nodi del cluster, ma richiede un'infrastruttura di rete che supporti tali protocolli.

5.3.4 Volumi

Kubernetes offre un insieme di modalità per fornire, ai container in esecuzione sui vari pod, la possibilità di utilizzare uno storage permanente. I vari container potranno utilizzare i volumi per mantenere dati che non devono essere persi, in caso di riavvi, o che devono essere condivisi tra i container di uno stesso pod. I volumi sono componenti di un pod e, per questo, sono definiti nella sua specifica; non possono essere creati o eliminati come un oggetto di alto livello, poiché sono legati al pod in cui sono stati definiti. Ogni container del pod potrà accedere ai relativi volumi, definendo, nella specifica del pod, un'operazione di mount sullo specifico container. Esistono vari tipi di volumi, in base alle specifiche necessità; tra questi [68, pag. 162]:

emptyDir Una cartella vuota, utilizzabile, ad esempio, per condividere file tra i container di uno stesso pod.

hostPath Un volume che permette di accedere ad una particolare cartella nel filesystem del nodo worker.

gitRepo Un volume che viene inizializzato con il contenuto di un repository Git.

nfs Un volume che permette di accedere al contenuto di un filesystem NFS (Network File System).

configMap e secret Un volume che permette di accedere alle informazioni contenute in un oggetto ConfigMap o Secret.

volumi relativi a specifici servizi di storage Esistono varie tipologie di volumi per permettere ai container di accedere a dati memorizzati su particolari servizi di storage o file system, come, ad esempio, gcePersistentDisk (Google Compute Engine Persistent Disk), awsElasticBlockStore (Amazon Web Services Elastic Block Store Volume) o cephfs (Ceph File System).

persistentVolumeClaim Una modalità per accedere a volumi pre-allocati o allocati dinamicamente; verranno approfonditi successivamente.

Le tipologie di volumi appena descritte devono essere configurate appositamente sulla base delle tecnologie disponibili e dell'infrastruttura presente. Ad esempio, nel caso di un volume nfs, sarebbe necessario specificare il percorso di rete relativo a tale servizio. Tale modalità non permetterebbe quindi di definire la specifica di un pod indipendentemente dall'infrastruttura sottostante, minando il grado di astrazione e portabilità che Kubernetes si è imposto di offrire.

Per risolvere tale problema, sono stati introdotti i seguenti oggetti:

PersistentVolume L'amministratore definisce, attraverso le API Kubernetes, degli oggetti PersistentVolume, che contengono tutti i dettagli relativi all'infrastruttura per accedere alle particolari tecnologie di memorizzazione. Un PersistentVolume è analogo ai volumi precedentemente descritti, ma la sua definizione non avviene all'interno della specifica del pod, ma in un oggetto separato, da parte dell'amministratore del cluster. L'amministratore specifica le caratteristiche e le modalità di accesso relative ad ogni PersistentVolume. I PersistentVolume saranno quindi volumi disponibili all'utilizzo da parte dei vari pod.

PersistentVolumeClaim Un PersistentVolumeClaim è un oggetto Kubernetes che descrive l'esigenza di un particolare PersistentVolume. Un PersistentVolumeClaim specifica tutte le caratteristiche che dovranno essere rispettate da un particolare PersistentVolume, e potrà essere utilizzato da un pod per ottenere un PersistentVolume che rispetti tali caratteristiche.

StorageClass I singoli PersistentVolume possono essere creati manualmente dall'amministratore, ma, nel caso di cluster di grandi dimensioni, tale soluzione sarebbe improponibile. Per questo, è stata introdotta una modalità che permette di realizzare automaticamente i PersistentVolume necessari. L'amministratore, invece di definire i singoli PersistentVolume, effettua il deployment di un provisioner e definisce uno o più oggetti StorageClass. I PersistentVolumeClaim potranno fare riferimento a tale StorageClass per allocare dinamicamente un nuovo PersistentVolume con le caratteristiche richieste. L'allocazione verrà effettuata dal provisioner associato a tale StorageClass. Kubernetes offre un insieme di provisioner per i servizi di storage più popolari, ma è anche possibile definire dei provisioner personalizzati.

I concetti appena descritti offrono un elevato grado di astrazione nei confronti della tecnologia di memorizzazione effettivamente utilizzata: i vari pod, attraverso un PersistentVolumeClaim, dichiarano la necessità di un volume, con determinate caratteristiche, ad esempio, con una determinata capacità, ma astraendo dalle specifiche tecnologie; il cluster Kubernetes, quindi, allocherà o reperirà un volume che rispetti tali caratteristiche.

Un possibile servizio di storage per volumi: Ceph

Ceph è un sistema distribuito open-source per la memorizzazione di dati, che può essere eseguito su cluster di macchine composte da hardware generico (commodity hardware); è capace di offrire un servizio di storage affidabile, basandosi su hardware potenzialmente inaffidabile.

Ceph non ha single point of failure, poiché tutti i vari componenti sono replicati, e garantisce la disponibilità, la consistenza e la correttezza dei dati sfruttando tecniche di replicazione distribuita o di erasure coding.

Un cluster Ceph può essere definito come uno unified storage system, poiché offre i propri servizi di storage in tre modalità: object, block e file storage.

I componenti principali di Ceph sono i seguenti [21]:

RADOS (Reliable Autonomic Distributed Object Store) È il livello che offre le funzionalità base di Ceph. Gli altri servizi, come RGW, RBD o CephFS, si baseranno su questo livello per offrire le proprie funzionalità. RADOS offre un servizio per la memorizzazione di oggetti affidabile, consistente, altamente disponibile, autonomo, distribuito e scalabile, capace di gestire, automaticamente, malfunzionamenti o riconfigurazioni. Ad esempio, si assicura che i dati siano correttamente replicati tra i vari nodi, gestendo le operazioni di allocazione, migrazione, ribilanciamento o riparazione. RADOS è composto da tre componenti principali:

- **ceph-mon (Monitor)** Mantiene un insieme di mappe relative allo stato del cluster. Tali mappe sono di importanza critica, poiché mantengono lo stato del cluster e sono necessarie per coordinare gli altri componenti del cluster. Costituisce l'autorità per quanto riguarda le operazioni di autenticazione e collocazione dei dati. A causa della sua importanza, è consigliabile configurare almeno tre ceph-mon per cluster, ma non più di sette.
- **ceph-mgr (Manager)** Monitora lo stato del cluster, aggregando un insieme di metriche in tempo reale, come, ad esempio, l'utilizzo della rete, dei dischi fisici o delle altre risorse relative ai vari nodi. Tali informazioni vengono esposte attraverso una dashboard ed attraverso delle API RESTful. Solitamente, in un cluster Ceph, sono presenti due manager, uno attivo ed uno in standby, in attesa di subentrare in caso di malfunzionamenti.
- **ceph-osd (Object Storage Daemon)** Si occupa di salvare fisicamente i dati sui dispositivi fisici, come HDD o SSD. Memorizza i dati e gestisce la loro replicazione, riparazione e bilanciamento, cooperando con gli altri ceph-osd. Risponde alle richieste di accesso ai dati. Un cluster Ceph dovrebbe possedere almeno tre osd, per garantire prestazioni e tolleranza ai guasti; cluster di grandi dimensioni potrebbero possedere migliaia di nodi osd.

LIBRADOS Una libreria che permette alle applicazioni di accedere direttamente a RADOS. Supporta C, C++, Java, Python, Ruby e PHP.

Object Storage service (RGW) Basandosi su RADOS, fornisce un servizio che permette, attraverso un'API HTTP RESTful, di salvare e gestire un insieme di oggetti. In generale, un oggetto è composto da un insieme di dati e da metadati che descrivono tale oggetto; gli oggetti sono, inoltre, raggruppati in bucket. Le API messe a disposizione da RGW sono compatibili sia con le API di Amazon S3 che con quelle di OpenStack Swift.

Block Device service (RBD) Basandosi su RADOS, fornisce un servizio che permette di allocare dei dispositivi di memorizzazione virtuale, a cui è possibile accedere, analogamente a dispositivi fisici, leggendo o scrivendo i singoli blocchi di dati. Su un tale dispositivo sarà quindi possibile utilizzare un qualsiasi file system. Ceph mantiene i dati relativi a tali dispositivi distribuiti su più nodi, per migliorare le performance e garantire la disponibilità; inoltre offre delle funzionalità per realizzare snapshot o per clonare un particolare dispositivo. Questo tipo di servizio è ampiamente utilizzato nelle infrastrutture cloud per gestire gli hard disk delle macchine virtuali.

File System service (CephFS) Basandosi su RADOS, mette a disposizione un file system di rete, distribuito, che rispetta gli standard POSIX. Tale file system potrà

quindi essere condiviso tra più client, che potranno accedervi contemporaneamente.

Alcuni dei servizi di storage offerti da Ceph possono essere utilizzati per la creazione di volumi Kubernetes. Questo permetterà, ai vari pod, di accedere ad uno storage persistente, caratterizzato da tutti i vantaggi di prestazioni e tolleranza ai guasti offerti da Ceph.

Inoltre, sono disponibili alcuni provisioner per Kubernetes che permettono la creazione dinamica, attraverso le StorageClass, di PersistentVolume basati su alcuni dei servizi offerti da Ceph; in particolare, è possibile creare volumi basati su RBD e formattati con un particolare file system, come xfs o ext4, oppure creare volumi basati su cephfs. I volumi basati su RDB permettono le operazioni di scrittura e lettura ad un solo nodo Kubernetes (ReadWriteOnce), mentre i volumi basati su cephfs permettono anche operazioni di scrittura contemporanee da parte di nodi differenti (ReadWriteMany).

5.3.5 Autoscaling

Kubernetes offre un insieme di modalità per effettuare delle operazioni di scaling automatiche, permettendo di variare automaticamente le risorse associate a determinate applicazioni in base al carico di lavoro. Kubernetes offre tre modalità di scaling automatico:

Horizontal pod Autoscaler Permette di scalare automaticamente il numero di pod relativi ad un ReplicaSet, un Deployment o un StatefulSet sulla base di un insieme di metriche specificate dall'amministratore, come, ad esempio, l'utilizzo della CPU.

Il controller relativo all'horizontal pod autoscaler ottiene, periodicamente, i valori correnti delle metriche relative ai pod che deve scalare; successivamente calcola il numero di pod necessari per rendere il valore corrente della metrica prossimo a quello desiderato, cioè il valore specificato al momento della definizione dell'autoscaler; infine, modifica il numero di repliche della risorsa che gestisce, cioè del ReplicaSet, del Deployment o dello StatefulSet; saranno poi i controller relativi a tali risorse ad aggiungere o eliminare i vari pod.

Vertical pod Autoscaler Alcune applicazioni non possono essere scalate orizzontalmente, cioè aggiungendo ulteriori unità. In tali casi è necessario effettuare uno scaling verticale, aggiungendo cioè risorse alle unità già in esecuzione. Tale possibilità è offerta dal Vertical pod Autoscaler. Esso permette di ridimensionare automaticamente le risorse assegnate ai singoli pod, in termini, ad esempio, di CPU o memoria, rimuovendo risorse ai pod che ne hanno allocate

più del necessario (scale down) e fornendole a quelli che ne hanno la necessità (scale up).

Cluster Autoscaler L'horizontal pod autoscaler permette di aggiungere pod in caso di necessità. Non può però aggiungere pod illimitatamente, poiché, oltre un certo limite, verrebbero saturate tutte le risorse di tutti i nodi del cluster, rendendo impossibile allocare nuovi pod. Questo problema non è causato esclusivamente dall'horizontal pod autoscaler, ma potrebbe verificarsi anche aggiungendo manualmente nuovi pod, saturando completamente le risorse del cluster. In una situazione del genere, se il cluster Kubernetes fosse in esecuzione su un insieme di macchine non gestite, sarebbe necessario aggiungere manualmente nuovi nodi. Nel caso, invece, in cui il cluster fosse ospitato da un particolare cloud provider, sarebbe possibile aggiungere nodi richiedendoli all'infrastruttura cloud. Il cluster autoscaler gestisce quest'ultimo caso. Infatti, nella situazione in cui fossero presenti pod non schedulabili a causa di risorse insufficienti, esso utilizzerà le API del cloud provider per allocare ed aggiungere nuovi nodi al cluster (scale out); analogamente, potrà rimuovere automaticamente dal cluster tutti i nodi sottoutilizzati (scale in). Il cluster autoscaler supporta vari cloud provider, come, ad esempio, Google Kubernetes Engine o Amazon Web Services; alcuni provider non sono invece ancora supportati, ma potrebbero essere aggiunti in futuro. Eventualmente, è, comunque, possibile implementare il proprio supporto per un determinato cloud provider, aggiungendo la relativa logica.

5.3.6 Controllo degli accessi

Kubernetes offre un insieme di modalità per gestire gli accessi all'API server, cioè definire chi può interagire con esso e quali operazioni può effettuare. Il processo di autenticazione, necessario per stabilire l'identità del client, viene svolto da un insieme di plug-in, per determinare l'utente o il ServiceAccount che ha effettuato la richiesta; successivamente, viene avviato il processo di autorizzazione, anch'esso realizzato tramite appositi plug-in. Uno dei più utilizzati è RBAC (Role-Based Access Control), che permette di definire ruoli e associazioni. I principali concetti inerenti al controllo degli accessi sono i seguenti:

Utenti, gruppi e ServiceAccounts Gli utenti corrispondono agli umani, come amministratori o sviluppatori, che vogliono interagire con Kubernetes. L'API server è in grado di autenticare un particolare utente, ma non gestisce il processo di creazione o eliminazione degli utenti; tale processo deve essere gestito esternamente.

I ServiceAccount sono invece necessari per autenticare le applicazioni in esecuzione nei vari pod; in tal modo, un'applicazione potrà autenticarsi sull'API server per interagire con esso, ed eseguire esclusivamente le operazioni consentite a quell'account. Al contrario degli utenti, la gestione dei ServiceAccount, cioè la loro creazione ed eliminazione, viene effettuata direttamente da Kubernetes: i ServiceAccount sono oggetti analoghi a tutti gli altri oggetti Kubernetes ed appartengono ad uno specifico namespace; per questo, i pod appartenenti ad un determinato namespace potranno utilizzare esclusivamente i ServiceAccount appartenenti a tale namespace.

Ogni utente o ServiceAccount può appartenere ad uno o più gruppi, ottenendo tutti i permessi associati ad essi.

Roles e RoleBindings I Role permettono di specificare quali azioni sono consentite e su quali risorse. Le azioni consistono, ad esempio, nella lettura, creazione, aggiornamento o cancellazione. Le risorse su cui applicare tali azioni potrebbero invece consistere in un'intera categoria, come ad esempio, i pod o i servizi o in una specifica istanza. I Role sono associati ad uno specifico namespace; per questo, garantiranno i permessi solamente alle risorse di tale namespace.

I RoleBindings permettono di associare i permessi definiti da un Role, ad un utente, ad un ServiceAccount o ad un gruppo. Anche in questo caso, i RoleBindings sono associati ad uno specifico namespace e, per questo, potranno associare solamente ruoli appartenenti a tale namespace. Alternativamente, un RoleBinding potrebbe referenziare un ClusterRole, limitando l'accesso alle sole risorse del namespace a cui il RoleBinding appartiene.

ClusterRoles and ClusterRoleBindings I ClusterRole sono analoghi ai Role, ma permettono di specificare i permessi su risorse prive di namespace, come i nodi, o sulle risorse di una determinata categoria per tutti i namespace del cluster, come i pod.

I ClusterRoleBindings sono analoghi ai RoleBindings, ma permettono di associare i ClusterRole, garantendo i permessi sull'intero cluster.

Capitolo 6

Definizione degli obiettivi

In questo capitolo saranno definiti i principali obiettivi di questo progetto, tenendo conto delle architetture, degli standard e delle tecnologie descritte nei capitoli precedenti.

In generale, l'obiettivo principale consisterà nella realizzazione di un cluster Kubernetes, gestito automaticamente, utilizzando dell'hardware a bassissimo costo. Questo progetto non si porrà come obiettivo quello di realizzare un sistema utilizzabile in un ambiente di produzione, poiché, in tal caso, sarebbe consigliabile utilizzare dell'hardware di categoria server, o di affidarsi ad un cloud provider pubblico.

Il sistema che ci si propone di realizzare sarà invece diretto a tutti coloro, come sviluppatori o studenti, che volessero realizzare un cluster Kubernetes entry level gestito automaticamente, senza utilizzare tecnologie di virtualizzazione ed evitando di affidarsi ad un cloud provider pubblico. Questa necessità potrebbe essere giustificata, ad esempio, dai minori costi, nel lungo periodo, rispetto all'utilizzo di un cloud provider pubblico, o da specifiche esigenze che impongono l'utilizzo di hardware fisico.

Oltre a ciò, la realizzazione di questo progetto permetterà, a chi volesse realizzarlo, di approfondire e mettere in pratica vari strumenti software, relativi alla gestione di macchine fisiche, al tema dell'Infrastructure as Code ed all'orchestrazione di container.

6.1 Realizzazione fisica del cluster

Il primo problema da affrontare consisterà nella realizzazione fisica del cluster, individuando l'hardware fisico da utilizzare, dal punto di vista della rete, delle macchine e dei dispositivi di memorizzazione. L'hardware scelto dovrà essere caratterizzato da bassi costi, ma dovrà anche garantire la compatibilità con i software di gestione che verranno adottati per automatizzare i vari processi di installazione e configurazione.

Attualmente, il tipo di elaboratori che offre i costi minori è quello dei cosiddetti single-board computer (SBC), cioè computer completi implementati su una singola scheda. Attualmente, il leader per eccellenza di questa categoria è il Raspberry Pi 4 B, costituito da un processore con architettura Arm. Questo SBC permette l'esecuzione di vari sistemi operativi, ma non offre tutte le funzionalità rese disponibili da computer ordinari con architettura x86-64. Uno degli obiettivi consisterà quindi nel determinare se tale SBC possa essere utilizzato per questo progetto, configurando o adattando, eventualmente, il software presente a livello del firmware. In caso negativo, potrebbero essere adottati SBC o mini-PC a basso costo differenti, eventualmente basati su architettura x86-64.

6.2 Realizzazione di un bare-metal cloud

Un ulteriore obiettivo che ci si pone è quello di riuscire a gestire le macchine fisiche scelte in maniera completamente automatica, riducendo al minimo gli interventi manuali, di configurazione o installazione, sulle singole macchine. L'obiettivo consisterà quindi nel realizzare, attraverso appositi strumenti, un'infrastruttura cloud privata, basata sul modello IaaS, che permetta, analogamente ad un cloud provider pubblico, di richiedere risorse on demand. Tali risorse consisteranno nelle macchine fisiche, che, al momento della richiesta, dovranno essere automaticamente installate e configurate. I cloud che offrono servizi di questo tipo sono chiamati bare-metal cloud, già descritti nel Capitolo 3.

Lo strumento scelto per raggiungere tale obiettivo è MAAS (Metal As A Service), per sfruttare tale software e, eventualmente, adattarlo, per gestire le macchine del cluster, che saranno, presumibilmente, Raspberry Pi 4 B. Si tenterà di determinare, quindi, se MAAS sia in grado, anche a seguito di eventuali adattamenti o aggiunte di funzionalità, di gestire automaticamente tali SBC. In caso negativo, si potrebbero utilizzare SBC o mini-PC a basso costo differenti.

Un traguardo ragguardevole, consisterebbe nell'adattare MAAS per permettergli di controllare il power management dei vari Raspberry, o, in generale, delle macchine scelte, attraverso la realizzazione di un driver e di un BMC personalizzato, basato su un qualche microcontrollore. Sarebbe, inoltre, auspicabile, ridurre al minimo le operazioni di configurazione delle singole macchine, permettendo, idealmente, di ridurre gli interventi manuali esclusivamente alle operazioni di connessione dell'alimentazione, delle interfacce di rete e del BMC; la gestione di tutte le altre operazioni dovrebbe avvenire automaticamente attraverso MAAS.

6.3 Deployment automatico del cluster Kubernetes

Un ulteriore obiettivo è quello di riuscire ad effettuare, automaticamente, il deployment di Kubernetes sulle macchine offerte da MAAS. MAAS, infatti, si occuperà esclusivamente dell'installazione dei sistemi operativi; le procedure successive, inerenti all'installazione dei vari componenti relativi ad una infrastruttura Kubernetes, composta da nodi master e worker, dovranno essere automatizzate il più possibile. In tal modo, sarà possibile realizzare, basandosi sui servizi IaaS offerti da MAAS, un'infrastruttura cloud privata capace di offrire un modello riconducibile al Cluster as a Service, simile ad Amazon EKS o Google Kubernetes Engine; un cloud di questo tipo permette cioè di richiedere, ad esempio, un cluster Kubernetes preconfigurato e funzionante.

Attualmente, per effettuare dei deployment automatizzati di Kubernetes su un insieme di nodi, sono disponibili vari strumenti, solitamente basati su tecniche di Infrastructure as Code, descritte nel Capitolo 4.

Lo strumento scelto per raggiungere questo obiettivo è Canonical Juju. L'adozione di tale strumento si propone di determinare se sia possibile effettuare un deployment automatico di un cluster Kubernetes su un insieme di macchine, gestite, a più basso livello, da MAAS. Tali macchine saranno, presumibilmente, dei Raspberry.

Un ulteriore traguardo consisterà nel riuscire ad aggiungere o rimuovere, attraverso Juju, nodi worker, in maniera semiautomatica: le operazioni di installazione e configurazione dovranno essere eseguite automaticamente, ma l'operazione di scaling complessiva dovrà avvenire su indicazione dell'amministratore.

Infine, si tenterà, attraverso Terraform, di facilitare e velocizzare il deployment di eventuali applicazioni o oggetti Kubernetes, da eseguire sul cluster realizzato.

6.4 Scaling automatico del cluster Kubernetes

Un cluster Kubernetes, come già descritto nella Sezione 5.3.5, potrebbe ritrovarsi a corto di risorse, cioè non possedere più nodi sui quali poter schedulare nuovi pod. Solitamente, nel caso in cui il cluster Kubernetes fosse installato su un insieme di macchine non gestite, sarebbe necessario aggiungere manualmente nuovi nodi. In questo progetto, poiché le macchine sono gestite da particolari strumenti che permettono un elevato grado di automatismo, potrebbe essere possibile aggiungere, automaticamente, nuove macchine al cluster Kubernetes in caso di necessità. Uno degli obiettivi consisterà quindi nel determinare se sia possibile, e con quali modalità, aggiungere automaticamente nodi al cluster Kubernetes, realizzando un sistema di scaling automatico. Per realizzare tale sistema si potranno sfruttare le funzionalità di scaling messe a disposizione da Juju, che permetteranno di richiedere nuove macchine a MAAS e di configurarle automaticamente.

Kubernetes mette già a disposizione un componente che si occupa di queste operazioni, chiamato cluster autoscaler. Attualmente, però, tale componente è compatibile esclusivamente con particolari cloud provider pubblici. Si potrebbe quindi tentare di aggiungere a tale componente la logica per comunicare con Juju. Alternativamente, si potrebbe adottare un sistema di scaling esterno a Kubernetes.

6.5 Load balancer e volumi persistenti su Kubernetes

I cluster Kubernetes, per poter definire dei servizi di tipo LoadBalancer, devono avere il supporto da parte dell'eventuale cloud provider su cui si basano. MAAS e Juju non offrono tale supporto e, per questo, un obiettivo aggiuntivo consisterà nell'individuare e adottare una tecnologia che permetta, in questi casi, di definire tali servizi di tipo LoadBalancer; la tecnologia che verrà presa in considerazione sarà MetalLB, installabile come applicazione Kubernetes.

Un'importante funzionalità delle infrastrutture Kubernetes è quella che permette di configurare ed utilizzare volumi persistenti. Un'infrastruttura Kubernetes può allocare nativamente volumi persistenti, locali ai singoli nodi worker; tale modalità non è però consigliabile, poiché non garantisce disponibilità o tolleranza ai guasti. Un obiettivo aggiuntivo consisterà quindi nell'individuazione e adozione di una tecnologia che permetta di offrire un servizio di storage distribuito, performante e tollerante ai guasti, utilizzabile per creare volumi Kubernetes. Tale soluzione software dovrà essere installata, con procedure automatiche, sugli stessi nodi che compongono il cluster Kubernetes. Inoltre, il cluster Kubernetes dovrà essere configurato adeguatamente, definendo le opportune StorageClass e provisioner, per permettere l'allocazione di tali volumi.

La tecnologia che verrà presa in considerazione sarà Ceph, che si tenterà di installare, analogamente a Kubernetes, tramite Juju.

6.6 Valutare la compatibilità del software con l'architettura Arm

Come precedentemente descritto, le macchine che si tenterà di utilizzare per la realizzazione del cluster saranno dei Raspberry Pi 4 B. Tali SBC sono costituiti da un processore con architettura Arm ed, inoltre, non offrono tutte le funzionalità, dal punto di vista del firmware, che le normali macchine x86-64 possono offrire. Per questo, un obiettivo di questo progetto consisterà anche nel valutare, allo stato attuale, la compatibilità del software utilizzato nei confronti di tali Raspberry Pi 4 B, e, in generale, nei confronti dell'architettura Arm.

In particolare, sarà necessario valutare, e, eventualmente, correggere, la compatibilità di tali SBC nei confronti di MAAS e dei sistemi operativi da esso utilizzati, di Juju, Kubernetes, MetalLB e Ceph.

Capitolo 7

Analisi del problema e progettazione del cluster

In questo capitolo verranno descritte le principali problematiche individuate e le conseguenti scelte progettuali adottate.

In particolare, si è scelto di realizzare il cluster adottando, dal punto di vista hardware, dei Raspberry Pi 4 B, cioè dei single-board computer che verranno resi compatibili nei confronti di MAAS sfruttando un'apposita implementazione open-source del firmware UEFI EDKII. Verrà quindi delineata una modalità che permetta di fornire tale firmware attraverso PXE, senza la necessità di installarlo sulla microSD dei vari Raspberry.

Inoltre, verrà progettato un semplice Baseboard Management Controller (BMC), basato sul microcontrollore ESP32, che permetta, a MAAS, di accendere, spegnere e controllare lo stato dei vari Raspberry. Le operazioni di accensione e spegnimento verranno impartite ai vari Raspberry tramite l'interfaccia GPIO, aggiungendo un device di tipo power button alle tabelle ACPI del firmware UEFI. L'ESP32 verrà programmato in linguaggio Python, sfruttando un apposito interprete per microcontrollori chiamato MicroPython.

Il deployment di Kubernetes e del servizio di storage, Ceph, verrà effettuato attraverso appositi modelli definiti tramite Juju. Verrà inoltre esteso il cluster autoscaler di Kubernetes, in modo tale che possa, attraverso chiamate a procedura remota (gRPC), invocare le API esposte da Juju per aggiungere o rimuovere nodi fisici, forniti da MAAS. Inoltre, si prevederà di utilizzare Terraform per effettuare il deployment di alcune applicazioni Kubernetes, come MetalLB ed il cluster autoscaler realizzato.

7.1 Hardware

Uno degli obiettivi pone di adottare dell'hardware a basso costo. Per questo, si è scelto di basare la realizzazione fisica del cluster, dal punto di vista delle macchine, su dei Raspberry Pi 4 B. Tali macchine sono i single-board computer (SBC) attualmente più diffusi, garantendo bassi costi ed un elevato supporto, sia dal punto di vista del software che della community.

I Raspberry Pi sono prodotti dalla Raspberry Pi Foundation, un'organizzazione di beneficenza britannica. La prima versione di Raspberry venne introdotta nel 2012, riscuotendo un elevato successo; da allora, l'hardware di questi SBC è stato costantemente migliorato.

Attualmente, l'ultima versione di tali SBC, il Raspberry Pi 4 B, offre le seguenti caratteristiche hardware [107]:

- Broadcom BCM2711 con processore Quad core Cortex-A72 (ARM v8) a 64-bit @ 1.5GHz
- 2 GB, 4 GB o 8 GB di SDRAM LPDDR4-3200
- Wi-Fi 802.11ac a 2.4 GHz e 5.0 GHz, Bluetooth 5.0
- Gigabit Ethernet
- 2 porte USB 3.0 e 2 porte USB 2.0
- GPIO header con 40 pin
- Slot microSD per il sistema operativo
- 2 porte micro-HDMI per gli eventuali display
- Alimentabile a 5V DC (minimo 3A), tramite un connettore USB-C

Per quanto riguarda lo storage sul quale installare il sistema operativo, potrebbe essere adottata la modalità classica offerta dai Raspberry, cioè l'utilizzo di una scheda microSD. Come verrà però descritto successivamente, per rendere compatibile MAAS con i Raspberry sarà necessario adottare un'apposita implementazione open-source del firmware UEFI EDKII. Tale implementazione, attualmente, non permette l'utilizzo della scheda microSD per l'installazione del sistema operativo. Il sistema operativo dovrà quindi essere installato su un disco esterno, connesso ad una delle porte USB del Raspberry. Per garantire costi ridotti, si è scelto di sfruttare delle pendrive USB, sulle quali verrà installato il sistema operativo e tutto il software relativo a Kubernetes.

Oltre a Kubernetes, ci si è posti come obiettivo quello di installare Ceph, un servizio di storage distribuito. Ogni nodo di un cluster Ceph richiede almeno un disco

dedicato, sul quale memorizzerà parte dei dati distribuiti. Per questo, ogni Raspberry che fungerà da nodo Ceph dovrà essere dotato di una seconda pendrive, che verrà gestita, esclusivamente, dal servizio di storage.

Inoltre, sarà necessario reperire i vari alimentatori, la struttura di supporto che ospiterà fisicamente i vari Raspberry, il microcontrollore per realizzare il BMC, uno switch di rete ed i vari patch cord per connetterlo ai vari Raspberry.

Il cluster che verrà realizzato sarà composto da quattro nodi ma, per le eventuali prove di scaling, sarà utilizzato un quinto nodo; per questo, alcuni componenti hardware dovranno essere replicati quattro o cinque volte. L'hardware utilizzato è elencato in Tabella 7.1.

Componente	**Quantità**
Raspberry Pi 4 B da 8 GB (Figura 7.1)	4/5
Sandisk Extreme Go da 64 GB, per il sistema operativo (Figura 7.2)	4/5
Sandisk Ultra Flair da 32 GB, per Ceph (Figura 7.2)	3
Alimentatore USB-C 5V 3A	4/5
Struttura verticale a quattro livelli per Raspberry (Figura 7.3)	1
Switch Gigabit Ethernet a 8 porte	1
Cavo patch UTP Cat 5e	4/5
Microcontrollore Espressif ESP32 per la realizzazione del BMC (Figura 7.4)	1
Cavo da micro-HDMI a HDMI, tastiera USB e microSD per le operazioni di debug e configurazione	1

Tabella 7.1: componenti hardware utilizzati per la realizzazione del cluster.

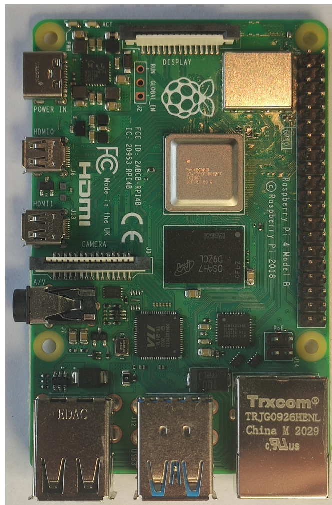

Figura 7.1: Raspberry Pi 4 B da 8 GB.

Figura 7.2: da sinistra a destra: Sandisk Extreme Go da 64 GB e Sandisk Ultra Flair da 32 GB.

Figura 7.3: struttura verticale a quattro livelli per Raspberry.

Figura 7.4: microcontrollore Espressif ESP32.

7.2 Possibili modalità di deployment

La realizzazione di cluster Kubernetes basati su Raspberry Pi è una pratica relativamente comune. Le modalità standard per il deployment di tali cluster sono, principalmente, due:

Flash manuale dei singoli Raspberry Il sistema operativo, come, ad esempio, Raspberry Pi OS o Ubuntu, viene istallato manualmente su ogni nodo, effettuando un'operazione di flash delle relative immagini sulle varie schede microSD. Questa è la modalità standard per installare un sistema operativo su un Raspberry Pi. Successivamente, attraverso particolari tool di configurazione, come Ansible, o, alternativamente, manualmente, vengono installati, su ogni nodo, tutti i componenti di Kubernetes. Alternativamente potrebbero essere create delle immagini preconfigurate con i vari componenti di Kubernetes, ma dovrebbero essere comunque installate manualmente sulle singole schede microSD. Questa modalità potrebbe essere valida per cluster di piccole dimensioni, ma, per cluster notevolmente estesi sarebbe improponibile installare manualmente il sistema operativo su ogni singola scheda microSD. Inoltre, le operazioni di correzione di malfunzionamenti o di applicazione di aggiornamenti, che richiedessero la reinstallazione completa dei vari nodi, sarebbero da effettuare manualmente, ripristinando le singole schede microSD.

Utilizzo di PXE ed NFS I Raspberry Pi offrono una modalità per effettuare il boot da rete, attraverso PXE. Tramite PXE, ogni Raspberry ottiene i file relativi al kernel Linux che, in seguito all'avvio, accederà al proprio file system attraverso la rete, tramite NFS (Network File System). Questo sistema permette di evitare di utilizzare le varie schede microSD, poiché i file system dei vari Raspberry vengono mantenuti in un server NFS centrale. Attraverso NFS viene fornito il file system relativo all'immagine di un particolare sistema operativo per Raspberry; tale file system sarà ad uso esclusivo del singolo nodo, e, per questo, dovrà essere presente, sul server NFS, una copia per ogni nodo del cluster. Analogamente alla modalità descritta precedentemente, l'immagine fornita potrebbe essere preconfigurata con i componenti Kubernetes o, alternativamente, configurata successivamente attraverso tool di configurazione. Questo sistema risolve le problematiche inerenti al flash manuale delle schede microSD, ma rende i vari nodi dipendenti dal server NFS: un eventuale malfunzionamento del server o relativo alla rete renderebbe inutilizzabile il cluster; inoltre, tale server, potrebbe causare un collo di bottiglia.

In questo progetto non verrà adottata nessuna delle modalità appena descritte. Infatti, la gestione ed il deployment dei vari nodi avverranno automaticamente attraverso un particolare software, chiamato MAAS (Metal As A Service); i vari componenti relativi a Kubernetes saranno poi installati attraverso un altro strumento, chiamato Juju.

Questa modalità di deployment di un cluster di Raspberry è inedita, poiché, solitamente, MAAS viene utilizzato per gestire macchine di categoria server, che offrono molte più funzionalità rispetto a SBC low cost, come i Raspberry. Anche MAAS utilizza PXE, ma la gestione delle macchine avviene in maniera differente

rispetto a quanto descritto precedentemente; infatti, non viene utilizzato NFS. Il deployment attraverso MAAS accomuna i vantaggi delle modalità descritte precedentemente: infatti, su ogni nodo verrà installato, localmente, il file system del sistema operativo, rendendo la macchina completamente indipendente; in compenso, però, la gestione dei processi di installazione o reinstallazione del sistema operativo potrà avvenire da remoto, evitando di dover intervenire manualmente su ogni singolo nodo.

Nella sezione successiva, verrà approfondita questa modalità di deployment, descrivendo le tecnologie adottate e le soluzioni necessarie per rendere MAAS compatibile con i Raspberry Pi.

7.3 MAAS

MAAS, come già descritto nel Capitolo 3, è uno strumento che permette di realizzare bare-metal cloud, gestendo, automaticamente e da remoto, il processo di installazione e configurazione di Ubuntu su un insieme di macchine fisiche.

Sorge però il problema relativo a come rendere compatibile tale strumento con i Raspberry Pi. Infatti, i sistemi operativi avviabili sui Raspberry non sono standard, poiché comprendono, al loro interno, un kernel Linux appositamente modificato per poter essere eseguito su tale hardware. Tali sistemi operativi vengono forniti attraverso delle immagini preconfigurate, che, attraverso un'operazione di flash, devono essere caricate sulla scheda microSD del Raspberry. Differentemente da quanto avviene per le macchine ordinarie basate x86-64, non viene avviato alcun processo di installazione del sistema operativo, poiché esso è preinstallato nell'immagine fornita.

MAAS, come già descritto nel Capitolo 3, opera differentemente, poiché utilizza delle immagini standard di Ubuntu per avviare, tramite PXE, il processo di installazione. Sebbene tali immagini siano disponibili anche per architettura arm64, non è possibile installarle direttamente sui Raspberry, poiché non comprendono le modifiche necessarie per renderle compatibili.

Fortunatamente, la community ha sviluppato un firmware UEFI per il Raspberry Pi 4, che risolve tale problema.

7.3.1 Raspberry Pi 4 UEFI Firmware

Il Raspberry Pi 4 UEFI Firmware è un'implementazione open-source, basata sul firmware UEFI TianoCore EDKII, che permette l'esecuzione di un firmware che rispetta gli standard UEFI su un Raspberry Pi 4. Come già descritto nella Sezione 2.4, UEFI si interpone, in questo caso, fra il firmware closed source [119] del Raspberry Pi 4 ed il sistema operativo, garantendo intercompatibilità tra le due parti. Il Raspberry Pi 4 UEFI Firmware è compatibile con gli standard Arm SystemReady, in particolare con

SBBR (Server Base Board Boot Requirements), descritti nella Sezione 2.3; poiché esso rispetta tali standard, adotta ACPI come tecnologia per l'individuazione dell'hardware. Ciò permette di avviare sul Raspberry un qualsiasi sistema operativo compatibile con tali standard, tra cui le immagini Ubuntu offerte da MAAS. UEFI, inoltre, permette di effettuare il boot da rete tramite PXE, garantendo la completa compatibilità con MAAS.

Attualmente, il Raspberry Pi 4 UEFI Firmware è però ancora in fase sperimentale, ed è soggetto ad alcune limitazioni [115]:

- il sistema operativo non potrà accedere né alla scheda microSD né all'interfaccia Wi-Fi, a causa dell'assenza, nel kernel Linux, dei relativi driver ACPI; per questo, non sarà possibile installare il sistema operativo, tramite MAAS, sulla scheda microSD, ma sarà necessario utilizzare una pendrive esterna;
- per poter utilizzare la scheda di rete è necessaria, almeno, la versione 5.7 del kernel Linux;
- per poter utilizzare più di 3 GB di RAM, sui Raspberry con 4 GB o 8 GB, è necessaria, almeno, la versione 5.8 del kernel Linux, che possiede una patch per aggirare un problema hardware del Raspberry relativo al Direct Memory Access. Di default, il firmware UEFI limita la RAM a 3 GB; se si utilizza un sistema operativo con tale patch applicata, è possibile disabilitare tale limite dalle impostazioni di UEFI.

Installazione e configurazione di UEFI su microSD

Il Raspberry Pi 4 UEFI Firmware può essere installato, analogamente ad un qualsiasi sistema operativo per Raspberry, caricando i relativi file sulla scheda microSD. In particolare, sono necessari:

start4.elf e fixup4.dat I file binari relativi al firmware closed source del Raspberry. Sono i primi file che vengono caricati dal bootloader del Raspberry, presente sulla EEPROM, per proseguire con l'operazione di boot. Solitamente, "start4.elf" carica direttamente il kernel del sistema operativo da avviare, ma non nel caso di UEFI, in cui, invece, dovrà caricare il firmware UEFI.

config.txt Un file di configurazione, letto da "start4.elf", che definisce vari parametri di configurazione. Nel caso del firmware UEFI, specificherà al firmware del Raspberry di caricare il binario contenente il firmware UEFI, e non direttamente il kernel di un eventuale sistema operativo.

File devicetree (.dtb) e overlay (.dto) I file ".dtb" descrivono tutti i dispositivi hardware del Raspberry, nel formato relativo ai devicetree del kernel Linux; gli

overlay permettono invece di aggiungere, a tale devicetree, particolari dispositivi aggiuntivi. Questi file sono sempre presenti nelle immagini dei sistemi operativi realizzate per Raspberry, poiché, tali sistemi operativi, basati sul kernel Linux, sono configurati per utilizzare devicetree come tecnologia per l'individuazione dell'hardware.

Il firmware UEFI può, invece, utilizzare esclusivamente ACPI per descrivere l'hardware; se il firmware UEFI viene avviato impostando l'utilizzo esclusivo di ACPI, che è l'impostazione di default, tali file verranno ignorati e potranno essere omessi. Alternativamente, è possibile avviare il firmware UEFI anche in modalità devicetree, e, in tal caso, questi file saranno necessari.

RPI_EFI.fd Il file binario contenente il firmware UEFI. I file descritti precedentemente sono file ufficiali, distribuiti nelle varie immagini dei sistemi operativi utilizzabili sui Raspberry. "RPI_EFI.fd" è invece il file derivato dalla compilazione del progetto relativo al firmware UEFI TianoCore EDKII per Raspberry Pi 4. Il firmware del Raspberry, caricando questo file, passerà il controllo a UEFI; sarà quindi UEFI ad occuparsi delle successive operazioni di boot del sistema operativo.

All'avvio del Raspberry, successivamente al caricamento di tali file sulla scheda microSD, verrà quindi avviato il firmware UEFI. UEFI mette a disposizione un'interfaccia di configurazione, mostrata in Figura 7.5, accessibile, analogamente ai computer ordinari, premendo il tasto ESC nei secondi successivi all'avvio.

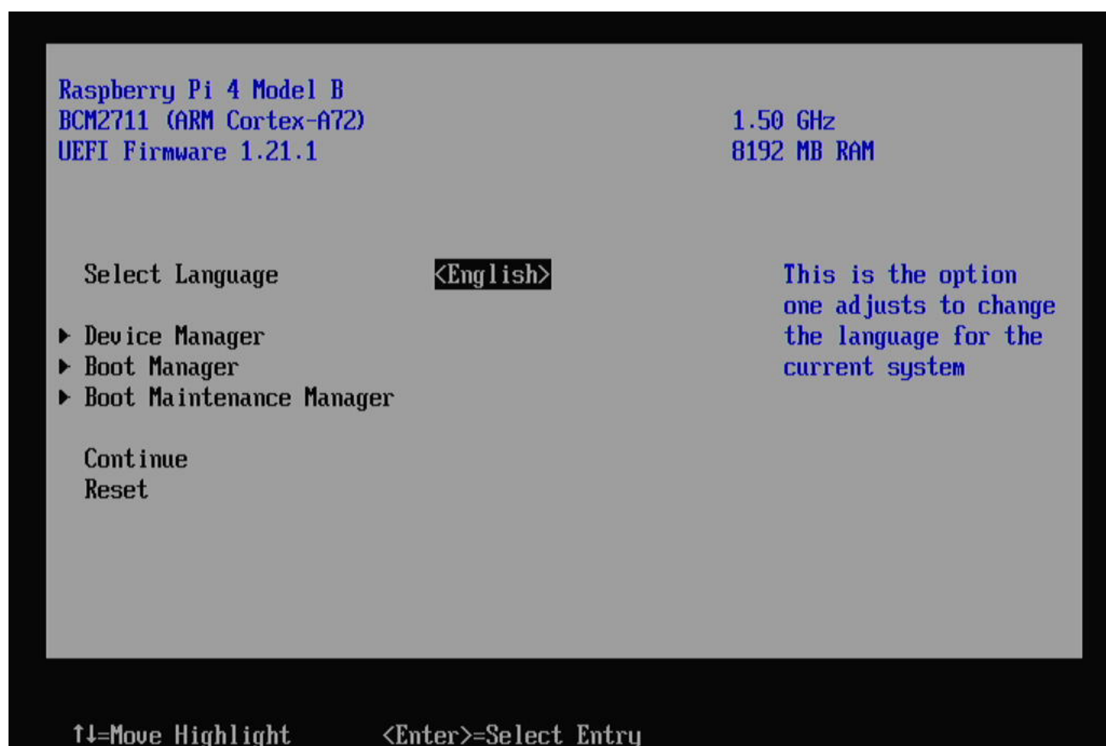

Figura 7.5: interfaccia di configurazione del Raspberry Pi 4 UEFI Firmware, accessibile premendo il tasto ESC nei secondi successivi all'avvio.

L'interfaccia permette di modificare varie impostazioni, tra cui, l'ordine di boot, memorizzato in una lista. UEFI, infatti, all'avvio, genera tutte le possibili opzioni di boot, ad esempio, per effettuare un boot da rete tramite PXE o per avviare un determinato bootloader presente nella partizione ESP di un disco connesso al sistema. Quando UEFI si avvierà, analogamente ai computer ordinari, tenterà di effettuare il boot dalla prima opzione della lista di boot.

Per rendere i vari Raspberry compatibili con MAAS, sarà quindi necessario installare UEFI nella scheda microSD di ognuno di essi, configurando l'ordine di boot in modo tale che l'avvio tramite PXE risulti essere la prima opzione della lista, come mostrato in Figura 7.6.

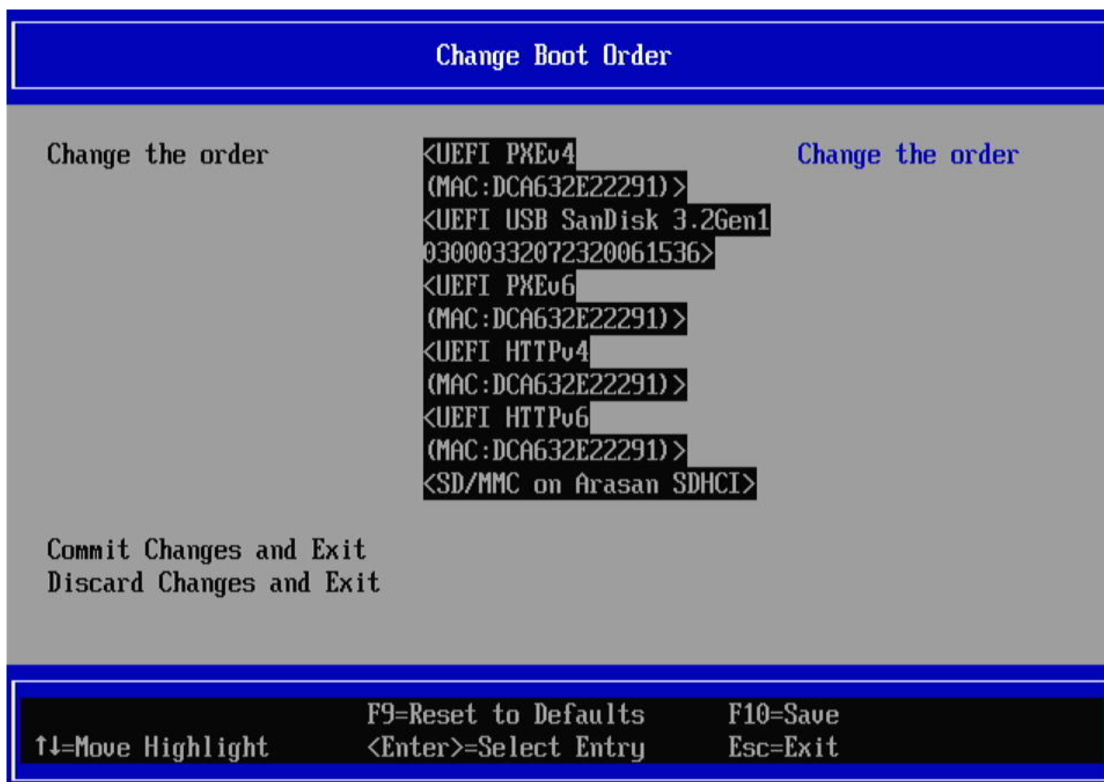

Figura 7.6: menu del Raspberry Pi 4 UEFI Firmware per la configurazione dell'ordine di boot; in questo caso l'opzione PXEv4 è in testa.

Per quanto riguarda le limitazioni di UEFI, è necessario che MAAS fornisca delle immagini di Ubuntu con, almeno, la versione 5.8 del kernel Linux. Attualmente, le immagini MAAS per arm64 di Ubuntu 20.04, con kernel hwe, dispongono del kernel Linux 5.8, che soddisfa i requisisti del firmware UEFI. Poiché Ubuntu 20.04 è una versione LTS, potrà essere utilizzata anche per le operazioni di enlist e commissioning, garantendo la compatibilità con il firmware UEFI per tutte le operazioni effettuate da MAAS.

Come già descritto nella Sezione 3.3.5, MAAS gestirà i Raspberry differentemente in base all'operazione da compiere: per le operazioni di enlist, commissioning e deployment fornirà un file di configurazione (grub.cfg) che istruirà il bootloader GRUB al caricamento, tramite TFTP ed HTTP, di un sistema operativo ephemeral, cioè caricato totalmente sulla memoria RAM. Sarà tale sistema ad avviare le operazioni di individuazione dell'hardware, di test e di installazione. La sequenza di tali azioni, oltre alle operazioni di avvio del firmware UEFI descritte precedentemente, sono mostrate in Figura 7.7.

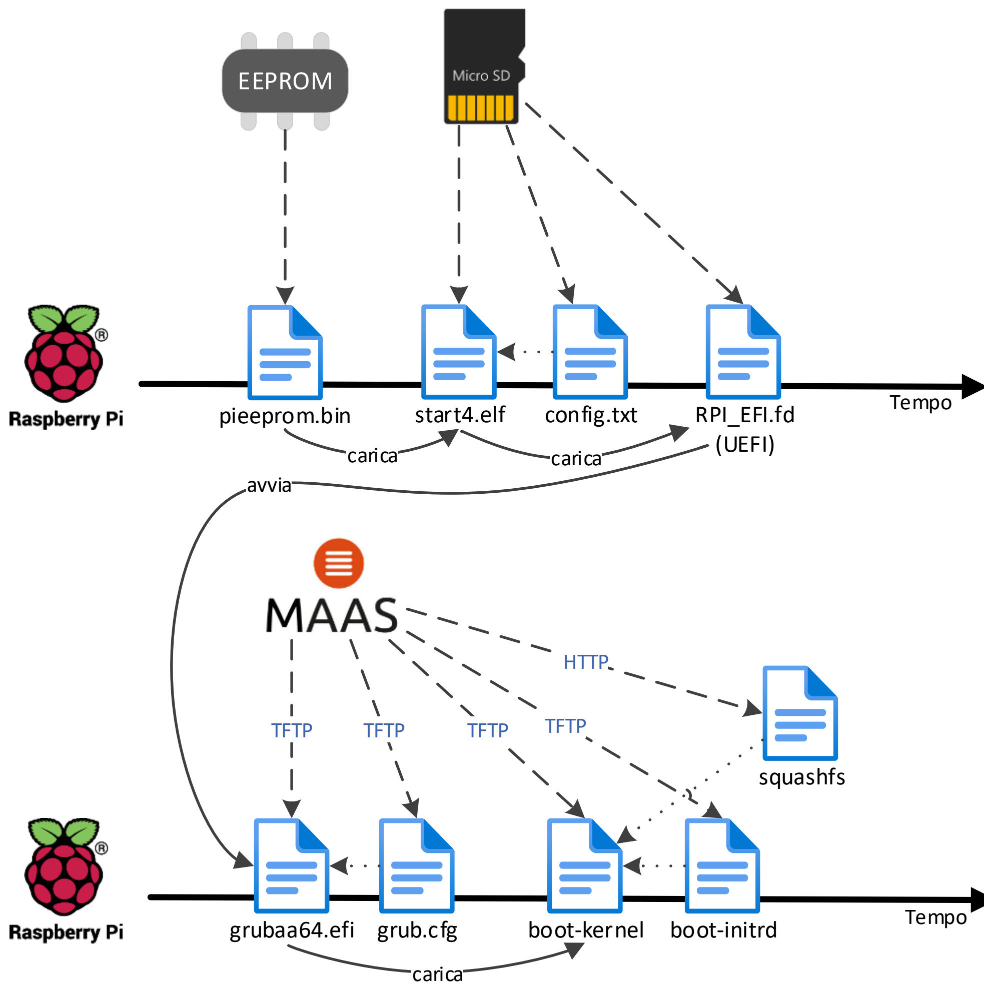

Figura 7.7: sequenza di boot di un Raspberry, con firmware UEFI su microSD, per le operazioni di enlist, commissioning e deployment. Fonti delle icone e dei loghi: [67][51][69].

Per quanto riguarda l'avvio, da parte di MAAS, di una macchina, con il sistema operativo già installato su uno dei dischi, che, nel caso dei Raspberry, saranno delle pendrive, viene eseguita la sequenza mostrata in Figura 7.8. Le operazioni relative al caricamento di UEFI sono identiche al caso precedente, con la differenza che il file di configurazione (grub.cfg) fornito da MAAS costringerà il bootloader GRUB al caricamento del sistema operativo presente su uno dei dischi connessi al sistema.

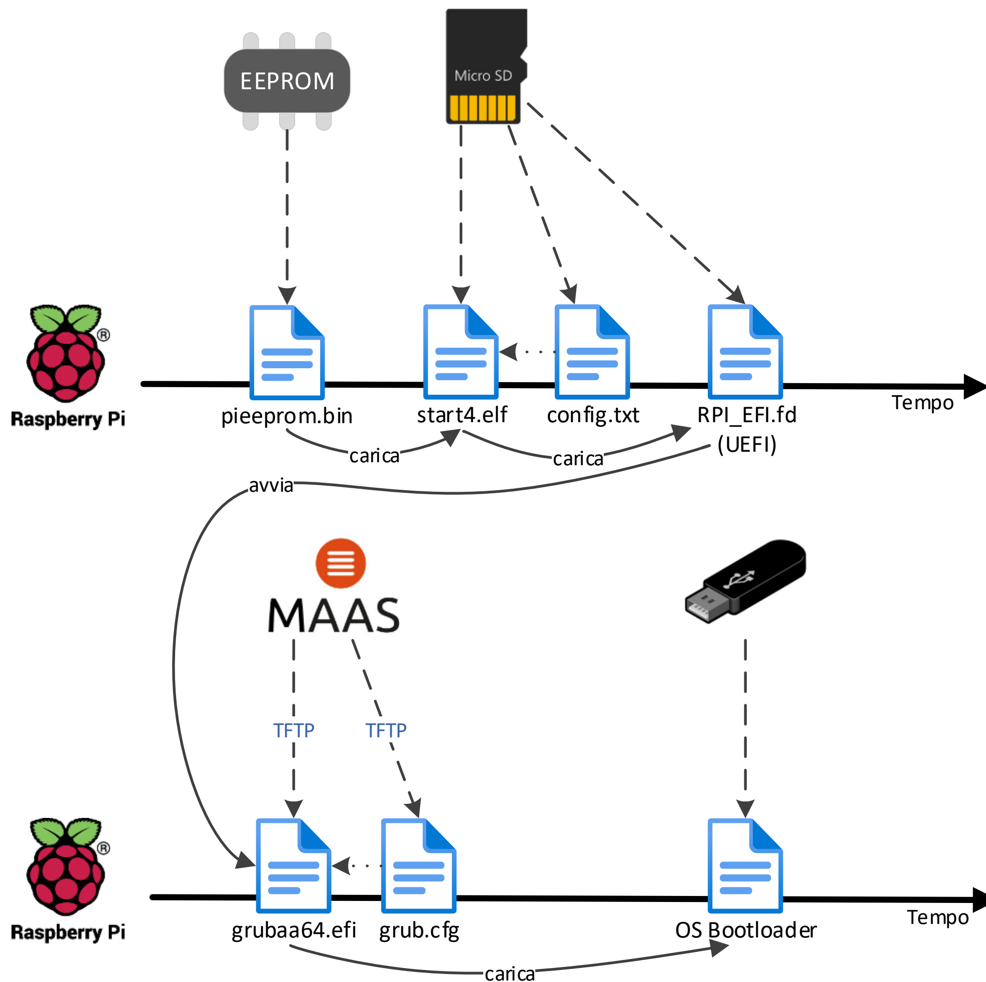

Figura 7.8: sequenza di boot di un Raspberry, con firmware UEFI su microSD, per l'avvio del sistema operativo installato. Fonti delle icone e dei loghi: [67][51][69][52].

Avvio di UEFI senza microSD

Il Raspberry Pi 4 UEFI Firmware rende i vari Raspberry compatibili con MAAS, ma permane un lato negativo: tale firmware deve essere installato manualmente su ogni microSD di ogni Raspberry e, successivamente configurato. Sebbene tale operazione debba essere eseguita una sola volta, al momento dell'installazione fisica del cluster, risulta, comunque, alquanto onerosa: oltre alla copia dei file sulle varie microSD, è necessario configurare il firmware UEFI, connettendo ogni Raspberry ad un display

e modificando le impostazioni tramite una tastiera USB; in particolare, è necessario disabilitare il limite relativo ai 3 GB di RAM, e modificare l'ordine di boot impostando PXE come prima opzione.

Poiché le impostazioni del firmware UEFI, tra cui l'ordine di boot, sono memorizzate direttamente all'interno del file "RPI_EFI.fd", si potrebbe pensare di modificare le impostazioni su un singolo Raspberry, per poi copiare sugli altri il file "RPI_EFI.fd" con le nuove configurazioni. In realtà, tale metodo non può funzionare, poiché le opzioni di boot PXE sono legate all'indirizzo MAC della scheda di rete presente sul singolo Raspberry: se l'ordine di boot venisse modificato su un altro Raspberry, le opzioni PXE sarebbero inutilizzabili, poiché legate al MAC di tale Raspberry. In tal caso, UEFI rigenererebbe le opzioni di boot, ripristinando l'ordine di default, che non prevede PXE come prima opzione della lista.

Per evitare l'installazione manuale del firmware UEFI su ogni microSD è possibile sfruttare il boot da rete PXE offerto dal bootloader presente sulla EEPROM di ogni Raspberry. Come impostazione di default, tale bootloader ricerca sulla microSD il firmware del Raspberry, "start4.elf", caricandolo in memoria e passandogli il controllo. Esso, però, può anche essere configurato differentemente, in modo tale che carichi il file "start4.elf", ad esempio, da un disco USB o da un ambiente PXE.

Anche il bootloader presente sulla EEPROM del Raspberry permette un insieme di configurazioni, tra cui l'ordine di boot, cioè l'ordine delle sorgenti da cui tenterà di caricare "start4.elf" e gli altri file necessari al boot, come "fixup4.dat", "config.txt" ed i file da caricare specificati in tale "config.txt", come, ad esempio, il kernel del sistema operativo (kernel7l.img) o UEFI (RPI_EFI.fd).

In particolare, è possibile impostare l'ordine di boot modificando la proprietà "BOOT_ORDER" della EEPROM [106]. Tale proprietà è rappresentata da un valore intero, che, letto dal valore meno significativo a quello più significativo, cioè da destra a sinistra, nella sua rappresentazione esadecimale, indica l'ordine delle opzioni di boot. Ogni cifra della rappresentazione esadecimale indica un'opzione; in particolare:

0x1 Scheda microSD.

0x2 Network boot tramite PXE.

0x3 RPIBOOT.

0x4 Boot da disco USB.

0x5 Boot da disco USB 2.0 connesso al connettore USB Type-C.

0xe Stop, interrompe il processo di boot mostrando un errore.

0xf Restart, comincia nuovamente il ciclo di boot partendo dalla prima opzione.

L'ordine di boot di default non prevede PXE. In questo caso specifico, sarà quindi necessario impostare tale proprietà al valore "0xf21". Il Raspberry tenterà di caricare il firmware prima dalla scheda microSD (0x1), poi da PXE (0x2) ed infine reitererà nuovamente il ciclo (0xf). I vari Raspberry non possiederanno una scheda microSD e, per questo, reperiranno il firmware del Raspberry ed il firmware UEFI da PXE. In caso di particolari malfunzionamenti all'ambiente PXE, o della necessità di riconfigurare la EEPROM, sarà comunque possibile riprendere il controllo dei singoli nodi, installando, su di essi, una scheda microSD.

Le configurazioni della EEPROM possono essere modificate tramite un apposito strumento, rpi-eeprom, presente nel principale sistema operativo per Raspberry, il Raspberry Pi OS. Per questo, al momento dell'installazione fisica del cluster, sarà necessario avviare, manualmente, tale sistema operativo su ogni Raspberry, per modificare l'ordine di boot della EEPROM. Questa operazione, sicuramente, compromette leggermente il livello di automatismo desiderato, ma non è possibile evitarla, poiché i Raspberry, come impostazione di default, possiedono il boot da rete disabilitato.

Impostando l'ordine di boot in modo tale che il Raspberry tenti di caricare il firmware da rete, tramite PXE, sarà possibile evitare di installare UEFI su ogni singola scheda microSD. Successivamente, il firmware UEFI procederà con un secondo boot da rete PXE, guidato da MAAS, già descritto precedentemente. Sarà però necessario risolvere il problema legato all'ordine di boot del firmware UEFI: tutti i Raspberry otterranno, tramite PXE, la medesima immagine del firmware UEFI (RPI_EFI.fd), che non possiederà alcuna opzione di boot; ogni firmware UEFI, rendendosi conto di non possedere alcuna opzione di boot, procederà alla loro generazione e, successivamente, tenterà di avviare la prima. Sarà quindi necessario individuare una modalità che permetta di generare tali opzioni in modo tale che quella relativa a PXE risulti essere in testa alla lista. In questo modo, ogni firmware UEFI genererebbe l'opzione di boot PXE relativa alla propria scheda di rete, rendendola utilizzabile ed avviandola automaticamente. Inoltre, il file "RPI_EFI.fd" dovrà essere preconfigurato con il limite relativo ai 3 GB di RAM disabilitato.

L'ambiente di boot PXE si basa sulla presenza di un server DHCP, per fornire gli indirizzi di rete, e TFTP, per fornire i file di boot. Il bootloader presente sulla EEPROM del Raspberry, per effettuare il boot da rete, richiede la presenza di due opzioni nell'offerta DHCP:

DHCP Option 43 (Vendor-specific information) Deve essere impostata a "Raspberry Pi Boot".

DHCP Option 66 (TFTP server name) Deve contenere l'indirizzo del server TFTP che conterrà i file relativi al firmware del Raspberry, tra cui "start4.elf", "fixup4.dat", "config.txt" ed "RPI_EFI.fd".

Il server DHCP che verrà utilizzato sarà quello offerto da MAAS, sia per il boot PXE relativo al bootloader presente sulla EEPROM che per il boot PXE relativo a UEFI. Sarà necessario però realizzare un DHCP snippet di MAAS, che permetta di identificare i client corrispondenti ai bootloader relativi alla EEPROM, assegnando le opportune opzioni DHCP. Per quanto riguarda il server TFTP, per fornire i file relativi al firmware, sarà possibile utilizzare sia il server TFTP di MAAS, copiando i relativi file nella root del server, sia un server TFTP esterno, dedicato a tale scopo. La scelta dovrà essere specificata nell'opzione 66 dello snippet DHCP.

La sequenza delle operazioni di boot effettuate da ogni Raspberry, per caricare il firmware UEFI da TFTP ed avviare le azioni relative alle fasi di enlist, commissioning e deployment, sono mostrate in Figura 7.9.

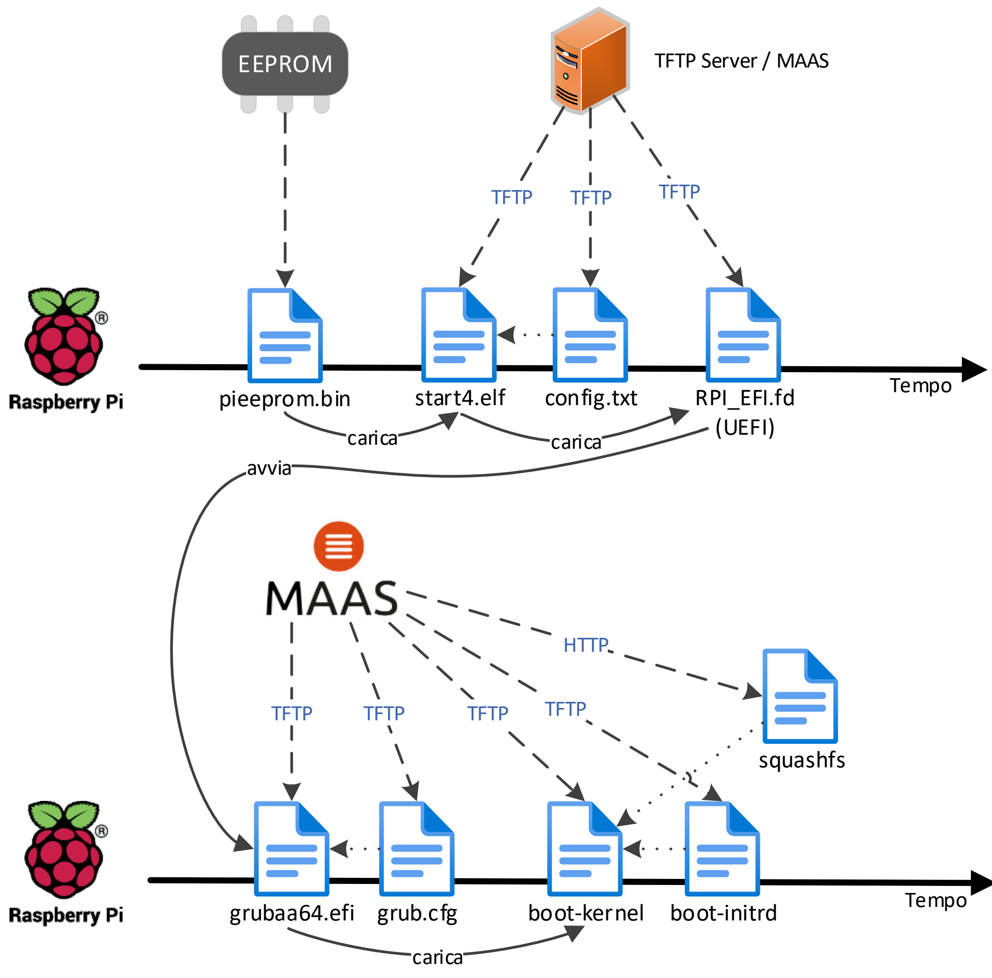

Figura 7.9: sequenza di boot di un Raspberry, con firmware UEFI fornito da TFTP, per le operazioni di enlist, commissioning e deployment. Fonti dei loghi: [67][69].

La sequenza delle operazioni di boot effettuate da ogni Raspberry, per caricare il firmware UEFI da TFTP ed avviare un sistema operativo installato su uno dei dischi connessi al sistema, sono mostrate in Figura 7.10.

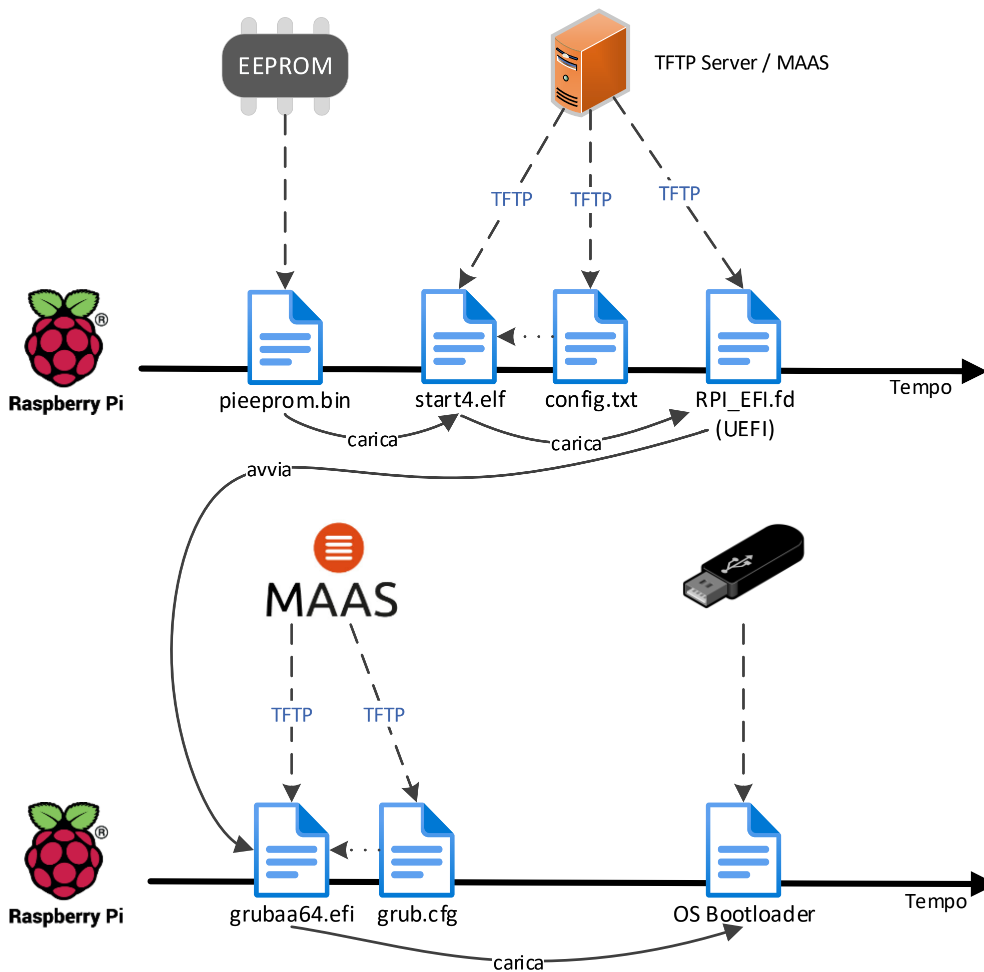

Figura 7.10: sequenza di boot di un Raspberry, con firmware UEFI fornito da TFTP, per l'avvio del sistema operativo installato. Fonti delle icone e dei loghi: [67][69][52].

7.3.2 Progettazione del BMC

MAAS, sfruttando il firmware UEFI, può installare e configurare Ubuntu automaticamente sui vari Raspberry. Per avere, però, un controllo completo delle macchine, dovrebbe possedere una modalità per controllare l'accensione o lo spegnimento delle stesse. MAAS, per permettere ciò, offre un insieme di power-driver, che contengono la logica per interagire con particolari BMC, presenti, solitamente, sulle macchine di categoria server. Senza tali componenti, sarebbe necessario effettuare manualmente le operazioni di accensione o spegnimento dei singoli

Raspberry. Sebbene una soluzione di questo genere non sia, comunque, improponibile, almeno per cluster di dimensioni ridotte, si è scelto di definire una modalità di power management personalizzata, per raggiungere l'obiettivo che impone di minimizzare gli interventi manuali. Tale modalità sarà composta da un BMC personalizzato e dal rispettivo driver, che permetterà a MAAS di utilizzarlo per controllare il power management dei vari Raspberry.

Possibili alternative

Le operazioni di accensione e spegnimento dei vari Raspberry potrebbero avvenire attraverso varie tecnologie:

Senza BMC Questa soluzione non prevedere di realizzare alcun tipo di BMC fisico, ma di sfruttare esclusivamente tecnologie software. Le operazioni di spegnimento di un Raspberry potrebbero quindi essere effettuate, ad esempio:

- implementando un componente software che dovrà essere eseguito su ogni nodo; tale software esporrà, attraverso una particolare tecnologia di accesso, come API HTTP o altro, un metodo per spegnere la macchina. Tale software, al momento della richiesta, si limiterà ad eseguire il comando di poweroff. Questo software potrebbe essere installato sui vari Raspberry sfruttando gli script cloud-init eseguiti da MAAS al momento dell'installazione; il power driver di MAAS si limiterà ad effettuare una richiesta di spegnimento a tale software;
- connettendosi alla macchina tramite SSH ed eseguendo il comando di poweroff. Il power driver di MAAS si limiterebbe ad effettuare tale operazione. In questo caso, è necessario però definire una modalità per reperire le chiavi di accesso SSH relative al particolare nodo.

Le operazioni di controllo dello stato potrebbero essere eseguite, ad esempio:

- tentando di contattare un componente software realizzato appositamente, in esecuzione sulla macchina; in caso di risposta, la macchina verrà considerata come accesa;
- effettuando un ping alla macchina; in caso di risposta, la macchina verrà considerata come accesa;
- tentando di connettersi a SSH; in caso di risposta, la macchina verrà considerata come accesa.

Si presenta, però, il problema dell'accensione; operazione che, solitamente, potrebbe avvenire, via software, da remoto, attraverso il Wake-on-LAN (WoL). I Raspberry non supportano però tale tecnologia, né valide alternative. Per questo,

una soluzione che non preveda l'utilizzo di un BMC fisico, non permetterà, probabilmente, di gestire le operazioni di accensione dei vari Raspberry.

BMC connesso ad uno o più relè Questa soluzione prevede di realizzare un BMC fisico, attraverso un qualche microcontrollore, che permetta di controllare l'alimentazione dei vari Raspberry attraverso un insieme di relè. Questa soluzione permette di effettuare le operazioni di accensione, poiché i Raspberry sono realizzati in modo tale da accedersi non appena ricevono l'alimentazione; inoltre, lo stato del Raspberry corrisponderà allo stato del relè. Si presenta però il problema legato allo spegnimento: sebbene sia possibile spegnere un Raspberry rimuovendo direttamente l'alimentazione, l'operazione è fortemente sconsigliata, poiché si rischierebbe di corrompere il file system del sistema operativo.

BMC connesso ai GPIO del Raspberry Questa soluzione prevede di realizzare un BMC fisico, attraverso un qualche microcontrollore, connesso all'interfaccia GPIO del Raspberry. Il Raspberry Pi 4, al momento dello spegnimento attraverso un comando di poweroff, entra in uno stato di risparmio energetico, da cui può essere risvegliato connettendo, momentaneamente, il pin GPIO3 a massa. Tale comportamento, abilitato di default, può essere eventualmente disabilitato modificando la proprietà "WAKE_ON_GPIO" relativa alla EEPROM. L'accensione dei vari Raspberry potrebbe quindi avvenire cambiando, attraverso il microcontrollore, lo stato di tale pin del GPIO.

Le operazioni di controllo dello stato potrebbero invece avvenire monitorando il pin GPIO14, che corrisponde al pin TX della seriale UART0. Tale interfaccia seriale, se abilitata tramite "enable_uart=1" nel file "config.txt", sarà attiva quando il Raspberry sarà acceso e disattiva quando spento. Quando l'interfaccia UART sarà attiva, il GPIO14 sarà impostato allo stato logico alto (3.3V). Il microcontrollore potrà quindi leggere il valore di tale pin per determinare lo stato del Raspberry. Sarà però necessario tenere in considerazione che le trasmissioni di dati su tale interfaccia causano un temporaneo cambiamento dello stato logico di tale pin, che potrebbe interferire con il processo di lettura.

Il Raspberry Pi mette inoltre a disposizione un particolare devicetree overlay, chiamato "gpio-shutdown.dtbo", che permette di impostare un pin del GPIO in modo tale che, collegandolo momentaneamente a massa, notifichi il sistema operativo per avviare le operazioni di spegnimento.

Purtroppo, a causa delle incompatibilità introdotte dal firmware UEFI con il devicetree, non è possibile, almeno attualmente, adottare tale overlay. Nel caso si decidesse di adottare questa alternativa, sarà quindi necessario individuare una soluzione per risolvere tale problema.

Soluzione ibrida Questa soluzione prevede di utilizzare parte delle soluzioni precedentemente descritte. Ad esempio, si potrebbero gestire le operazioni di accensione attraverso un insieme di relè, mentre quelle relative allo spegnimento tramite un software in esecuzione sui vari nodi.

Soluzione scelta: accensione e spegnimento dei Raspberry tramite GPIO

La soluzione che si è scelto di realizzare ricade nel caso del BMC connesso ai GPIO del Raspberry. Tale scelta è giustificabile dalle seguenti motivazioni:

- l'installazione del software ad hoc su ogni nodo potrebbe essere difficoltosa; inoltre, tale software deve essere compatibile con lo specifico sistema operativo;
- le chiavi SSH per accedere ad una macchina potrebbero non essere facilmente reperibili;
- l'utilizzo di relè non permette uno spegnimento corretto.

Come già descritto, le operazioni di accensione e di controllo dello stato sono facilmente realizzabili. Per quanto riguarda lo spegnimento, non esistono, però, molte alternative, per le ragioni qui di seguito riportate:

- il firmware UEFI supporta il devicetree ed i relativi overlay, ma non garantisce che il sistema operativo possa essere avviato correttamente. Infatti, il firmware UEFI, come comportamento di default, per rispettare gli standard SBBR di Arm, utilizza esclusivamente ACPI e non devicetree. La soluzione ottimale consisterebbe quindi nell'evitare di utilizzare devicetree, favorendo ACPI. Per questo, non sarà possibile adottare l'overlay "gpio-shutdown.dtbo", che avrebbe permesso lo spegnimento attraverso il GPIO;
- avviando il firmware UEFI in modalità ACPI, non è possibile, dal sistema operativo, controllare il GPIO, poiché, attualmente, il kernel Linux non possiede ancora i driver ACPI relativi a tale interfaccia del Raspberry Pi 4. Per questo, non è possibile realizzare un software che possa leggere lo stato di un particolare pin, eseguendo il comando di shutdown nel caso tale stato cambiasse.

La soluzione che si tenterà di adottare consisterà nel definire un cosiddetto power button nella tabella DSDT di ACPI. Tale estensione dovrà fornire la stessa funzionalità fornita dal devicetree overlay gpio-shutdown, ma, in questo caso, utilizzando ACPI.

ACPI permette infatti di definire, tra i vari device della tabella DSDT, un cosiddetto power button, identificato dall'HID "PNP0C0C". Quando il sistema operativo verrà avviato, leggendo la tabella DSDT, entrerà a conoscenza di tale device e caricherà, per quel dispositivo, i driver corrispondenti al power button. Tale driver, alla ricezione

di una notifica con valore "0x80", inizierà le operazioni di spegnimento del sistema. La notifica avviene tramite il metodo "Notify", all'interno del codice AML relativo alla tabella DSDT. Sarà quindi necessario realizzare una soluzione in AML che, all'accadere di un evento su un determinato pin del GPIO, come, ad esempio, il cambio di stato da alto a basso, notifichi il power button, per avviare le operazioni di spegnimento.

Le modifiche effettuate alla tabella DSDT dovranno quindi essere integrate nel firmware UEFI, ricompilando il file "RPI_EFI.fd".

Questa soluzione risulta la migliore, poiché permetterebbe di effettuare lo shutdown di un qualsiasi sistema operativo che supporta ACPI, senza la necessità di installare un software ad hoc su ogni singolo nodo.

Scelta dell'hardware

La soluzione delineata sarà composta da un microcontrollore, connesso ai GPIO dei vari Raspberry del cluster. In generale, tale microcontrollore ha un unico requisito: possedere un'interfaccia di rete dedicata, che permetta a MAAS di comunicare con esso. Potrebbe quindi essere adottato, ad esempio, un Arduino con uno shield Ethernet, un altro Raspberry Pi o, in generale, un qualsiasi elaboratore dotato di un'interfaccia GPIO ed un'interfaccia di rete.

Per la realizzazione di questo progetto, si è scelto di utilizzare il microcontrollore ESP32, prodotto da Espressif. Tale microcontrollore dispone di un'interfaccia Wi-Fi, di 48 pin GPIO, anche se non tutti utilizzabili, e, per la sua categoria, di un'elevata potenza di calcolo. Inoltre, è supportato da MicroPython, un interprete che permette di eseguire codice Python su vari microcontrollori.

L'utilizzo di un'interfaccia Wi-Fi potrebbe non essere la scelta più appropriata per garantire affidabilità e disponibilità. In ogni caso, se fossero richieste tali garanzie si potrebbe adottare un microcontrollore dotato, ad esempio, di una scheda Ethernet.

Progettazione API

L'ESP32, cioè il BMC, esporrà un insieme di API RESTful, che permetteranno di verificare lo stato dei vari Raspberry o di avviare le operazioni di accensione e spegnimento. In particolare, verranno esposte le seguenti risorse:

/rpis Il metodo GET su tale risorsa dovrà restituire un oggetto JSON, contenente lo stato di ogni Raspberry connesso al BMC. Tale oggetto JSON conterrà un insieme di proprietà, il cui nome corrisponderà all'identificativo univoco del Raspberry. Ognuna di tali proprietà conterrà un oggetto con la proprietà "power", che potrà assumere due valori: "on" o "off". Un esempio di output è il seguente:

```
{"0": {"power": "off"}, "1": {"power": "on"}}
```

/rpis/<id> Il metodo GET su tale risorsa dovrà restituire lo stato del Raspberry identificato dall'id specificato nell'URL. Tale stato sarà composto da un oggetto JSON con la proprietà "power", che potrà assumere due valori: "on" o "off". Un esempio di output è il seguente:

```
{"power": "off"}
```

Il metodo PUT su tale risorsa permetterà di modificare lo stato del relativo Raspberry, avviando un'operazione di accensione o spegnimento. All'interno del body della richiesta, sarà necessario specificare la proprietà power, in formato "x-www-form-urlencoded". Tale proprietà potrà possedere due valori: "on" o "off". Nel caso di "on" verrà avviata un'operazione di accensione della macchina, mentre nel caso di "off" di spegnimento. Se l'operazione è stata accettata, verrà restituito il seguente JSON:

```
{"message": "accepted"}
```

L'accettazione della richiesta non garantisce, però, il cambiamento di stato della risorsa. Infatti, se, ad esempio, il particolare Raspberry non rispondesse alle richieste di spegnimento, rimarrà nello stato di "on". Sarà il client che dovrà gestire opportunamente tali situazioni.

Le varie chiamate alle API avranno un effetto sui pin dell'ESP32, che, a loro volta, cambieranno lo stato dell'interfaccia GPIO dei vari Raspberry connessi. Ad esempio, un'operazione di accensione provocherà il cambiamento di stato da alto a basso, per un breve periodo, del pin dell'ESP32 connesso al GPIO3 di un particolare Raspberry, provocandone l'accensione.

MAAS Power driver

Dovrà inoltre essere realizzato il power driver che permetterà a MAAS di interagire con le API del BMC precedentemente descritte, in modo da poter controllare il power management dei vari Raspberry del cluster. Tale driver dovrà essere selezionabile nelle impostazioni delle macchine di MAAS, e necessiterà dell'indirizzo IP del BMC e dell'identificativo del relativo Raspberry. Tale associazione, tra la macchina MAAS e l'identificativo utilizzato dalle API del BMC per controllarla, dovrà essere realizzata manualmente, al momento della prima configurazione, che avverrà successivamente all'operazione di enlist ma prima del commissioning.

Interfacciamento tra Raspberry ed ESP32

L'ESP32 dovrà essere connesso, tramite la propria interfaccia GPIO, direttamente alle interfacce GPIO dei quattro Raspberry del cluster. Sarà necessario realizzare tre collegamenti per ognuno di essi, in particolare:

- la massa dell’ESP32 dovrà essere in comune alle masse dei vari Raspberry;
- l’ESP32 utilizzerà un proprio pin di output per connettersi al GPIO3 del Raspberry, in modo da inviare il segnale di accensione o spegnimento. Il GPIO3 sarà sempre impostato dal Raspberry come pin di input, con valore logico, quando scollegato, alto, attraverso una resistenza di pull-up. L’ESP32 dovrà portare tale pin temporaneamente a massa, provocando un evento che comporterà l’accensione o lo spegnimento del Raspberry;
- l’ESP32 utilizzerà un proprio pin di input per verificare lo stato del GPIO14, che sarà nello stato logico alto quando il Raspberry sarà acceso.

Un esempio di tale collegamento è mostrato, per un singolo Raspberry, in Figura 7.11, in cui le masse sono in comune, il pin 12 dell’ESP32 è connesso al GPIO14 del Raspberry, ed il pin 13 al GPIO3.

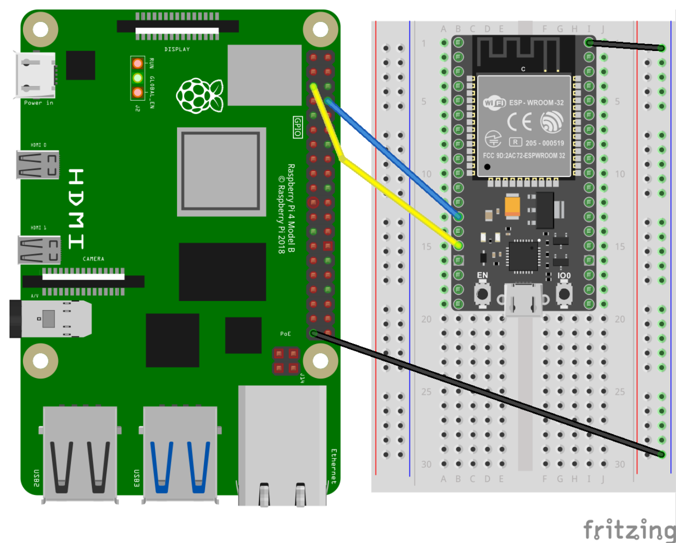

Figura 7.11: schema di collegamento tra uno dei Raspberry del cluster e l’ESP32. Fonti dei modelli: [92][91].

Architettura complessiva

Complessivamente, l’ESP32 sarà connesso, tramite l’interfaccia GPIO, ad ogni Raspberry del cluster. Esso esporrà quindi delle API HTTP, precedentemente descritte, che permetteranno a MAAS, tramite un apposito power driver da sviluppare, di controllare il power management dei singoli Raspberry. Tale architettura è mostrata in Figura 7.12.

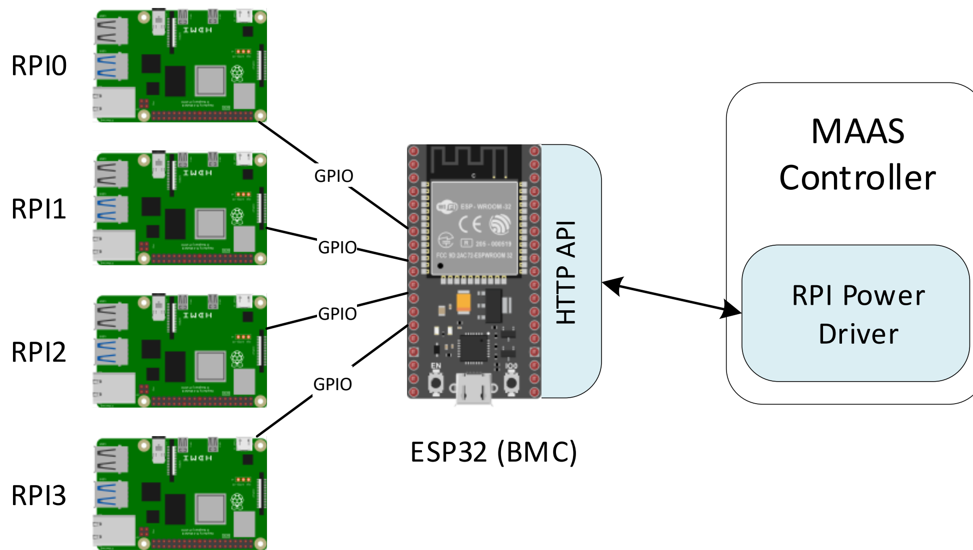

Figura 7.12: architettura del sistema relativo al power management dei Raspberry. Fonti dei modelli: [92][91].

7.4 Juju

MAAS, grazie al firmware UEFI ed al BMC che si andrà a realizzare, avrà il controllo completo dei singoli Raspberry, permettendo di installarli e configurarli su richiesta. Tale servizio, verrà sfruttato da Juju, per il deployment dei modelli relativi a Kubernetes ed a Ceph, distribuiti sotto forma di bundle. Questi bundle potranno poi essere modificati o integrati tra loro in base alle specifiche esigenze.

7.4.1 Kubernetes Core

Juju offre due bundle Kubernetes: Kubernetes Core [62] e Charmed Kubernetes [23]. La differenza principale consiste nel fatto che, nel bundle Charmed Kubernetes, sono presenti più nodi Kubernetes master e il traffico diretto agli API server di tali nodi master viene distribuito attraverso un ulteriore nodo di load balancing. Il bundle Charmed Kubernetes è, sicuramente, più completo e affidabile, ma richiede un numero molto elevato di nodi: almeno due master, un load balancer ed i vari nodi worker.

Considerato il numero di Raspberry a disposizione, si è scelto di sfruttare il bundle Kubernetes Core, che non comprende il load balancer e permette di realizzare un cluster Kubernetes composto da due soli nodi.

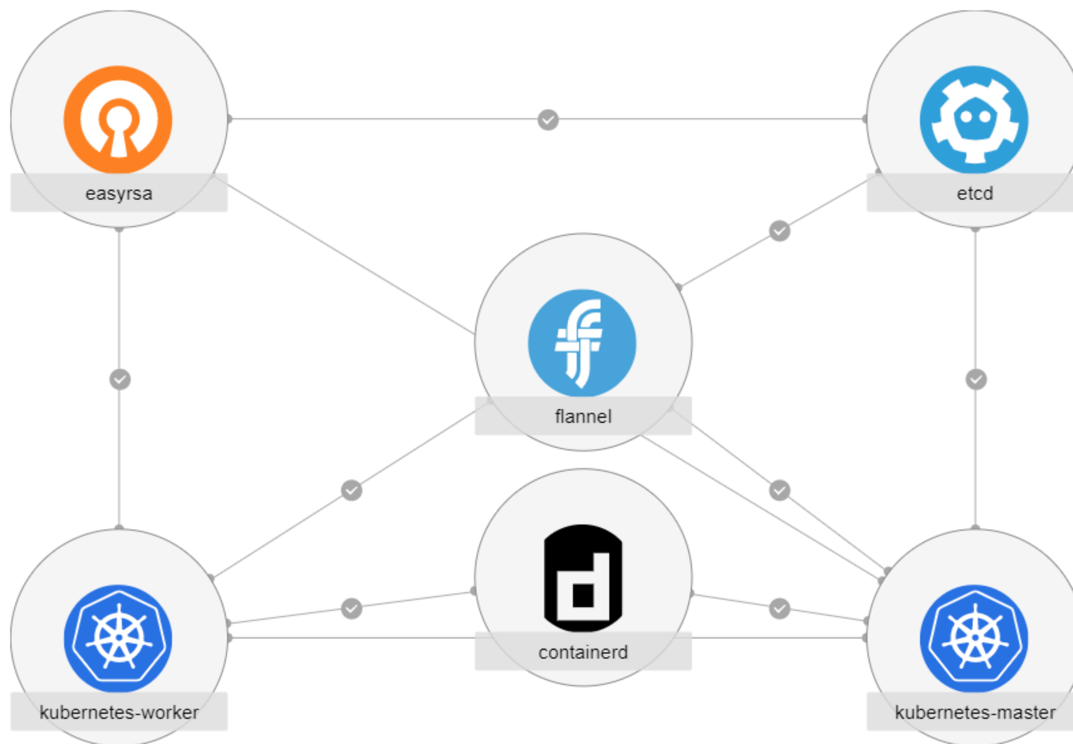

Figura 7.13: charm e relazioni inclusi nel bundle Kubernetes Core. I nodi del grafo rappresentano i charm, mentre gli archi rappresentano le relazioni che sussistono tra essi. Fonte: [62].

Il bundle Kubernetes Core, mostrato in Figura 7.13, è composto dai seguenti charm:

kubernetes master Questo charm comprende i vari componenti relativi ad un nodo Kubernetes master, tra cui, l'API server, lo scheduler ed il controller manager. Comprende Heapster per il monitoraggio del cluster e, opzionalmente, la dashboard. Non comprende però etcd, che viene distribuito attraverso un charm distinto. Questo offre la liberà di installare etcd su nodi differenti rispetto al master.

kubernetes worker Questo charm comprende i vari componenti relativi ad un nodo Kubernetes worker, tra cui Kubelet e kube-proxy. non comprende però il container runtime, poiché esso viene distribuito attraverso un charm subordinato distinto. Questo permette di intercambiare facilmente il container runtime, senza la necessità di modificare il Kubernetes worker.

etcd Questo charm comprende il database distribuito, etcd, utilizzato dal Kubernetes master. Come configurazione di default prevede un solo nodo, ma è comunque

possibile creare più unità e distribuirle su più nodi, per garantire prestazioni e tolleranza ai guasti.

containerd Questo charm subordinato comprende containerd, che, attualmente, è il container runtime principale di Kubernetes; viene distribuito come charm subordinato, relazionato ai nodi master e worker; per questo, verrà installato su ognuno di tali nodi.

flannel Questo charm comprende flannel, uno dei plugin CNI (Container Network Interface) più utilizzati. Tale plugin permette di realizzare la rete virtuale interna di Kubernetes, fornendo ad ogni nodo una sottorete che i vari container potranno utilizzare. Viene distribuito come charm subordinato, relazionato ai nodi master e worker; per questo, verrà installato su ognuno di tali nodi.

easyrsa Questo charm comprende EasyRSA, che fungerà da certificate authority fornendo certificati self signed ad ogni unità del cluster che li richiederà.

Come configurazione di default, il bundle Kubernetes Core prevede due soli nodi:

- un nodo master, sul quale saranno eseguite le unit relative ai charm Kubernetes master, etcd ed easyrsa;
- un nodo worker, sul quale sarà eseguita la unit relativa al charm Kubernetes worker.

Tale bundle, poiché il cluster progettato sarà composto da quattro nodi, verrà quindi modificato, aggiungendo altri due nodi worker.

Il numero dei nodi worker può essere anche modificato dinamicamente, effettuando un'operazione di scaling. Attraverso il comando

```
juju add-unit kubernetes-worker
```

sarà possibile aggiungere al cluster un nodo worker. Juju richiederà a MAAS una nuova macchina ed inizierà il processo di installazione e configurazione.

Analogamente, attraverso il comando

```
juju remove-unit kubernetes-worker/3
```

è possibile rimuovere uno specifico nodo worker, in questo caso il nodo 3. I processi in esecuzione su tale nodo verranno terminati; successivamente MAAS spegnerà la macchina, rendendola disponibile per altri utilizzi.

7.4.2 Ceph Base

Juju permette anche il deployment dei vari componenti del servizio di storage distribuito Ceph. Tali componenti sono distribuiti sotto forma di charm, come, ad esempio, Ceph Osd, Ceph Mon, Ceph Radosgw, Ceph Proxy o Ceph Fs.

I componenti di base, necessari per realizzare un cluster Ceph minimale, sono compresi nel bundle Ceph Base [22], mostrato in Figura 7.14.

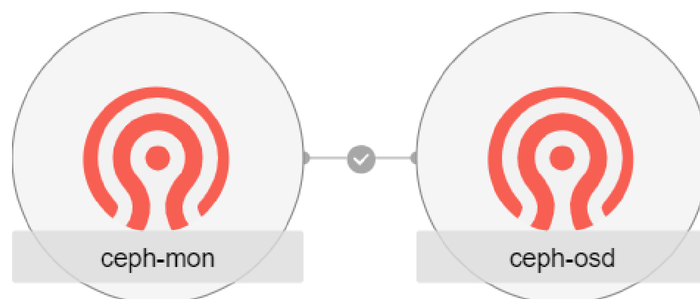

Figura 7.14: charm e relazioni inclusi nel bundle Ceph Base. I nodi del grafo rappresentano i charm, mentre gli archi rappresentano le relazioni che sussistono tra essi. Fonte: [22].

Questo bundle comprende:

ceph-mon Questo charm comprende il componente Monitor di un cluster Ceph, già descritto nella Sezione 5.3.4. In generale, mantiene lo stato del cluster.

ceph-osd Questo charm comprende il componente Object Storage Daemon di un cluster Ceph, già descritto nella Sezione 5.3.4. In generale, gestisce i dischi fisici connessi al sistema.

Ceph Base, come configurazione di default prevede tre nodi, ognuno con una unit ceph-mon ed una unit ceph-osd. Le unit ceph-mon verranno installate su macchine LXD. Le unit ceph-osd dovranno invece essere eseguite su macchine con un disco fisico dedicato a Ceph, accessibile tramite "/dev/sdb". Tali dischi corrisponderanno alle pendrive impiegate appositamente per tale scopo.

Questa configurazione di default è adatta al cluster che si andrà a realizzare; infatti, le unità dei tre nodi Ceph potranno essere installate sui tre nodi worker.

Inoltre, è possibile aggiungere una relazione Juju tra il charm ceph-mon ed il charm Kubernetes master. In tal modo, il cluster Kubernetes verrà automaticamente configurato per utilizzare volumi persistenti basati sui servizi offerti da Ceph.

7.5 Autoscaler

Attraverso MAAS e Juju sarà possibile realizzare, automaticamente, un cluster Kubernetes composto da Raspberry Pi. Una delle funzionalità più interessanti offerte da Juju, è la possibilità di impartire delle azioni di scaling, per aumentare o diminuire il

numero di unità relative a determinate applicazioni, come, ad esempio, il kubernetes-worker. Tali azioni automatizzano l'allocazione e la configurazione delle macchine da aggiungere al cluster Kubernetes, ma possono essere impartite esclusivamente dall'amministratore. In questa sezione, si determinerà una modalità che permetta di avviare tali azioni di scaling automaticamente, in base al carico di lavoro del cluster Kubernetes.

7.5.1 Possibili alternative

Il sistema di scaling automatico del cluster Kubernetes potrebbe essere implementato scegliendo tra due strategie principali:

Sistema esterno a Kubernetes Consiste nell'implementare un sistema esterno che monitora ogni nodo del cluster. Ad esempio, potrebbe monitorare l'utilizzo medio di CPU tra tutti i nodi; nel caso le risorse utilizzate superino una certa soglia, verrà avviata una procedura di scaling. Sebbene questo metodo possa portare a buoni risultati, permette di effettuare lo scaling solamente se le risorse sono effettivamente utilizzate; non può infatti individuare il caso in cui il cluster Kubernetes sia saturo, non perché le risorse sono effettivamente utilizzate, ma perché riservate all'utilizzo da parte di determinati pod.

Sistema interno a Kubernetes Consiste nell'implementare un sistema che interagisce direttamente con le API di Kubernetes, sfruttando il suo sistema di monitoraggio dei nodi. Un sistema di questo tipo risolve le criticità del caso precedente, poiché può individuare il caso in cui le risorse del sistema siano sature perché già riservate all'utilizzo di altri pod. Nel caso in cui nuovi pod non potessero essere schedulati, procederà all'operazione di scaling.

La strategia scelta per la realizzazione dell'autoscaler consisterà nel realizzare il sistema internamente a Kubernetes, sfruttando ed estendendo una soluzione già ampiamente diffusa: il Kubernetes cluster autoscaler [60]. Per completezza, verrà però accennata una possibile soluzione relativa all'altra strategia, basata sul charm Juju chiamato Charmscaler.

7.5.2 Juju Charmscaler

Charmscaler [24] è un charm sviluppato dalla community di Juju, che permette di effettuare automaticamente delle operazioni di scaling su un determinato modello Juju, sulla base dell'utilizzo medio di CPU dei nodi che compongono il cluster. Tale charm sfrutta una logica sviluppata da Elastisys per effettuare le varie decisioni di scaling. Inoltre, sfrutta i seguenti charm:

telegraf Un charm subordinato, che dovrà essere relazionato all'applicazione i cui nodi dovranno essere monitorati. Telegraf è un'applicazione che permette di collezionare un insieme di metriche relative al nodo sul quale è eseguito. In questo caso, collezionerà le metriche relative all'utilizzo della CPU.

influxdb Un charm che permette installare InfluxDB, un database per serie temporali. Su tale database verranno memorizzate le varie metriche raccolte da telegraf. Il Charmscaler utilizzerà le informazioni memorizzate sul database per valutare un'eventuale operazione di scaling.

Il Charmscaler, per poter effettuare le operazioni di scaling, dovrà, inoltre, possedere l'indirizzo e le credenziali del controller Juju e l'identificativo del modello sul quale apportare le relative modifiche.

Questo charm permetterebbe di automatizzare le operazioni di scaling sul cluster che si andrà a realizzare, ma, per le ragioni descritte precedentemente, si è scelto di favorire l'adozione del Kubernetes cluster autoscaler.

7.5.3 Estensione del Kubernetes cluster autoscaler

Il Kubernetes cluster autoscaler supporta vari cloud provider, fra cui, però, non è presente Juju. Per risolvere tale problema è possibile estenderlo, cioè aggiungere la logica necessaria che gli permetta di utilizzare Juju per aggiungere o rimuovere nodi fisici. Il codice sorgente del cluster autoscaler, realizzato tramite il linguaggio di programmazione Go, è open-source, e può essere liberamente consultato o modificato [60].

Interfacce da implementare

L'estensione del cluster autoscaler può avvenire estendendo l'interfaccia CloudProvider, definita nel file "cloud_provider.go" [61] nel repository dell'autoscaler. I metodi principali di tale interfaccia sono:

NodeGroups Deve restituire un array contenente tutti i node group del cloud provider.

NodeGroupForNode Deve restituire il node group a cui appartiene un particolare nodo del cluster.

I node group sono gruppi di nodi che possiedono le stesse caratteristiche ed hanno lo stesso insieme di label. Il cloud provider può quindi raggruppare i propri nodi in gruppi, in modo che possano essere gestiti separatamente dal cluster autoscaler di Kubernetes. Le operazioni scaling, da parte dell'autoscaler, verranno effettuate su un particolare node group, in base alle caratteristiche che i nodi di tale categoria possono offrire.

I node group sono modellati attraverso l'interfaccia NodeGroup, che dovrà essere implementata. I metodi principali di tale interfaccia sono:

Id Deve restituire l'identificativo univoco del NodeGroup.

MaxSize Deve restituire il numero massimo di nodi che il gruppo può raggiungere.

MinSize Deve restituire il numero minimo di nodi che il gruppo può raggiungere.

TargetSize Deve restituire il numero di nodi che il cloud provider desidera raggiungere, attualmente, per tale gruppo. Questo valore potrebbe non corrispondere al numero di nodi attualmente presenti nel cluster Kubernetes, poiché alcuni potrebbero ancora essere in fase di creazione o eliminazione. Prima o poi, però, il numero di nodi del cluster Kubernetes dovrà convergere a tale valore.

IncreaseSize Deve permettere di aumentare il numero di nodi del gruppo, avviando le procedure di creazione di nuovi nodi sul cloud provider. Questo metodo, quando chiamato, dovrà attendere l'effettiva creazione dei nodi.

DeleteNodes Deve permettere di eliminare specifici nodi del gruppo, avviando le procedure di eliminazione da parte cloud provider. Questo metodo, quando chiamato, dovrà attendere l'effettiva eliminazione dei nodi.

DecreaseTargetSize Deve permettere di ridurre il TargetSize. Non deve, però, eliminare nodi già attivi del cluster, ma, solamente, nodi che sono ancora in fase di creazione da parte del cloud provider.

Nodes Deve restituire la lista dei nodi che appartengono a questo gruppo. Ogni nodo della lista sarà identificato da un Id, che dovrà corrispondere alla proprietà providerID, nella sezione spec, dei nodi presenti nelle API Kubernetes. Questo metodo permette all'autoscaler di associare i nodi presenti nelle API Kubernetes ai nodi del cloud provider.

Cluster autoscaler come client gRPC

L'obiettivo sarà allora quello di realizzare un'implementazione di tali interfacce, che permetta all'autoscaler di interagire con il modello Kubernetes di Juju. Ad esempio, quando l'autoscaler richiederà un nuovo nodo tramite in metodo IncreaseSize, dovrà essere eseguita, da Juju, un'azione di scaling, analoga al comando "juju add-unit kubernetes-worker".

Sorge, però, una problematica: l'interazione con il controller Juju dovrà avvenire attraverso le API WebSocket che espone; gli sviluppatori di Juju hanno realizzato delle librerie per interagire, facilmente, con tali API, ma sono state implementate, esclusivamente, per due linguaggi: per Python, chiamata python-libjuju [104], e per JavaScript, chiamata js-libjuju [56]. Poiché il cluster autoscaler è stato implementato in

Go, sarà necessario individuare una strategia che permetta, al codice Go, di invocare metodi in codice Python o JavaScript.

Oltre a questa problematica se ne aggiunge una ulteriore: la ricompilazione del codice relativo all'autoscaler risulta particolarmente onerosa, in quanto necessita di importare varie librerie attraverso alcuni script e di ricreare l'immagine del container da eseguire nel relativo pod Kubernetes.

Per questo, la community dell'autoscaler, ha recentemente proposto una soluzione [40], che permetterebbe di evitare tale procedura per implementare un cloud provider personalizzato. Tale soluzione si basa sull'utilizzo di gRPC, cioè un framework, sviluppato da Google, che implementa un sistema di chiamata a procedura remota. Tale framework permette, ad un client gRPC, di invocare i metodi esposti da un server gRPC; client e server potrebbero essere in esecuzione su nodi distinti, messi in comunicazione da una rete TCP/IP. Inoltre, il linguaggio di programmazione utilizzato dal client potrebbe differire da quello utilizzato dal server: l'unico vincolo, è che le due parti rispettino il protocollo gRPC.

L'idea proposta dalla community è quella di realizzare un'implementazione del CloudProvider come client gRPC, che potrà essere configurato per connettersi ad un server gRPC esterno, che implementerà la vera logica del cloud provider. Questa strategia permetterebbe di evitare di ricompilare il cluster autoscaler per aggiungere un determinato cloud provider. Infatti, sarebbe necessario, esclusivamente implementare la logica del server gRPC, che, a sua volta, comunicherebbe con le API del servizio cloud.

L'utilizzo di gRPC risolverebbe anche la problematica legata ai linguaggi di programmazione differenti tra le due parti. Per questo, si è scelto di adottare tale strategia per implementare la logica di interazione del cluster autoscaler con Juju.

Purtroppo, attualmente, tale soluzione è ancora solo una proposta e, per questo, la relativa implementazione non è stata realizzata. Sarà quindi necessario implementare anche il client gRPC, estendendo direttamente il cluster autoscaler.

Protocol Buffers e gRPC

Il framework gRPC si basa sui cosiddetti Protocol Buffers. I Protocol Buffers permettono di definire, attraverso dei file ".proto", tutte le strutture dati che dovranno essere serializzate. Tali strutture dati possono essere definite attraverso la parola chiave "message". Ogni "message" può contenere, al suo interno, tipi di dato base, come interi o stringhe, o, ricorsivamente, altri messaggi.

Attraverso uno specifico compilatore, chiamato protoc, sarà possibile generare il relativo codice sorgente, in un qualsiasi linguaggio, che permetterà di leggere o scrivere tali strutture dati su una qualsiasi sorgente.

All'interno del file ".proto" è inoltre possibile dichiarare dei servizi, attraverso la parola chiave "service". Ogni servizio definisce l'insieme dei metodi che espone,

attraverso la parola chiave "rpc". Tali metodi potranno accettare in input o restituire in output le strutture dati dichiarate precedentemente attraverso la parola chiave "message".

Saranno tali servizi che definiranno in metodi invocabili dal client gRPC sul server gRPC. Il compilatore protoc permetterà di compilare, in un qualsiasi linguaggio, anche il codice relativo ai servizi, specificando, durante la compilazione, un particolare argomento. Tramite tale codice sarà possibile creare un server gRPC o un client gRPC, che sfrutteranno il codice relativo alla serializzazione per effettuare il passaggio, attraverso la rete, dei parametri o dei risultati.

Per estendere il cluster autoscaler attraverso gRPC, sarà quindi necessario definire tale file ".proto", che dovrà contenere tutte le strutture dati ed i metodi necessari per implementare le interfacce CloudProvider e NodeGroup.

gRPC server e Juju Python client

Per implementare il server gRPC, che dovrà comunicare con le API Juju, si è scelto il linguaggio Python, sfruttando la libreria python-libjuju [104]. Questo componente realizzerà, quindi, un server gRPC, che rimarrà in attesa della connessione da parte del cluster autoscaler. Alla ricezione di una particolare invocazione di metodo gRPC, sfrutterà la libreria python-libjuju per impartire particolari comandi al Juju controller, in base alla logica del metodo.

Ad esempio, quando il cluster autoscaler invocherà il metodo IncreaseSize, verrà inoltrata la relativa richiesta al server gRPC, che, a sua volta, tramite python-libjuju, impartirà al Juju controller di aggiungere un'unità kubernetes-worker. L'aggiunta di tale unità scatenerà l'allocazione, da parte di MAAS, di una nuova macchina, che verrà automaticamente installata e configurata per entrare a far parte del cluster Kubernetes.

In ogni caso, python-libjuju, per poter impartire tali comandi dovrà possedere l'indirizzo e le credenziali per accedere al Juju controller, oltre che l'identificativo del modello relativo a Kubernetes, per poter leggere lo stato o apportare le relative modifiche.

Questo componente sarà separato dal cluster autoscaler, e potrà quindi essere eseguito su un pod Kubernetes dedicato o su un nodo esterno al cluster Kubernetes. L'unico requisito è che possa comunicare con il controller Juju.

Architettura complessiva

Complessivamente, il cluster autoscaler verrà esteso aggiungendo il cloud provider relativo a Juju. Tale cloud provider sarà composto da un client gRPC, che inoltrerà le chiamate ad un componente esterno, chiamato Juju client, in esecuzione su un nodo differente e composto da un server gRPC. Tale componente, implementato in Python, definirà la logica dei metodi relativi all'autoscaler impartendo particolari comandi al

Juju controller, sfruttando la libreria python-libjuju [104]. Il Juju controller sfrutterà quindi i servizi di MAAS per richiedere o rimuovere i relativi nodi fisici. Il cluster autoscaler verrà eseguito all'interno di un pod Kubernetes, poiché dovrà accedere all'API server con determinati permessi; il Juju client potrà invece essere eseguito sia su un altro pod, sia su una macchina dedicata. Le interazioni tra i vari componenti sono riassunte in Figura 7.15.

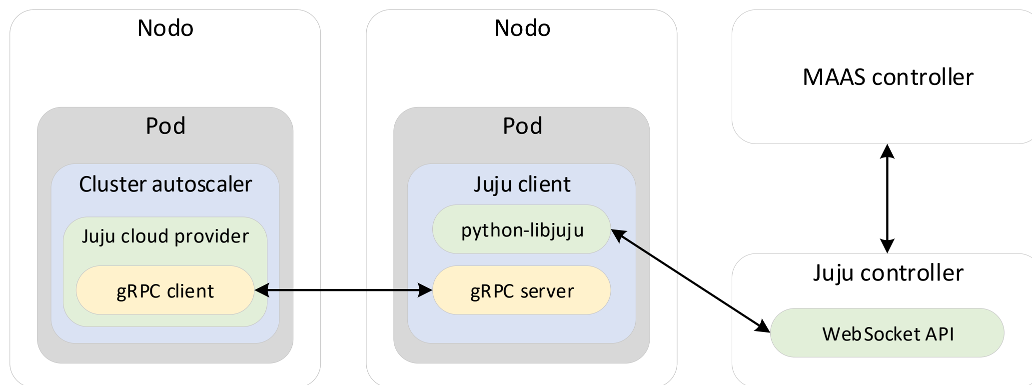

Figura 7.15: interazione tra i componenti inerenti al cluster autoscaler.

7.6 Terraform

Successivamente alla realizzazione del cluster Kubernetes, sarà necessario effettuare il deployment di alcune applicazioni, come MetalLB ed il cluster autoscaler, oltre che ad applicazioni di prova per verificare il corretto funzionamento del sistema. Il deployment delle applicazioni Kubernetes e dei relativi oggetti, come, ad esempio, dei ReplicaSet, dei Deployment, degli StatefulSet o delle ConfigMap, verrà facilitato dall'utilizzo dello strumento di Infrastructure as Code chiamato Terraform.

Terraform, tra i vari provider offerti, dispone di quello relativo a Kubernetes, che permette di effettuare il deployment delle relative risorse.

All'interno del file di configurazione, sarà sufficiente specificare il seguente codice:

```
provider "kubernetes" {
}
```

Esso dichiarerà l'utilizzo del provider Kubernetes, rendendo disponibili le relative risorse. Tale provider, come configurazione di default, utilizzerà le credenziali presenti nel file ".kube/config" della home directory dell'utente per accedere alle API Kubernetes; alternativamente, è possibile specificare, nel blocco di codice del provider, una modalità di accesso differente.

I file di configurazione Terraform non sono eccessivamente diversi rispetto ai file YAML applicabili tramite kubectl. Essi hanno, però, come vantaggio, quello di permettere di definire un codice dinamico, organizzabile in moduli ed in grado di interagire con altri provider. Terraform, inoltre, permette di velocizzare e facilitare le operazioni di deployment o di eliminazione delle varie risorse tramite un workflow ben definito.

Il deployment di MetalLB avverrà attraverso un modulo Terraform sviluppato dalla community [126], mentre il deployment del cluster autoscaler prevederà la realizzazione di un file di configurazione Terraform che definirà il relativo Deployment Kubernetes.

7.7 Infrastruttura di rete e deployment dei controller

I vari Raspberry del cluster dovranno essere connessi ad una rete LAN totalmente dedicata. Assegnare semplicemente i vari nodi ad una sottorete IP non sarebbe infatti sufficiente, poiché il protocollo DHCP, implementato dal server DHCP del controller MAAS per assegnare gli indirizzi, utilizza il broadcast di rete a livello di data link, cioè al livello a cui operano gli switch. Sarà quindi necessario realizzare un dominio di broadcast dedicato al cluster, per evitare che il DHCP di MAAS interferisca con eventuali altri dispositivi o che, similmente, l'indirizzo dei vari Raspberry venga assegnato da altri server DHCP presenti sulla rete.

Nel progetto che si andrà a realizzare, per evitare l'acquisto di ulteriori dispositivi di rete dedicati al cluster, si è deciso di sfruttare le VLAN (Virtual Local Area Network), cioè reti LAN virtuali che condividono gli stessi dispositivi di rete fisici, ma che permettono di separare completamente il relativo traffico. Nello specifico, si è scelto di utilizzare la VLAN con id 10, anche se un qualsiasi identificativo sarebbe stato ammissibile.

Considerando le ridotte dimensioni del cluster che verrà realizzato, si è scelto di co-locare il MAAS region controller ed il rack controller su una stessa macchina; ci si riferirà a tale macchina con il termine generico di MAAS controller.

Il MAAS controller, ospitando il server DHCP ed i server relativi ad altri servizi, dovrà possedere un'interfaccia connessa alla VLAN dedicata al cluster, con un indirizzo IP statico. Nello specifico, si è scelto l'indirizzo "192.168.10.1/24".

Il MAAS controller verrà configurato in maniera tale da gestire la subnet "192.168.10.0/24". Tale subnet, che sarà di categoria managed, comprenderà due range di indirizzi:

192.168.10.100-192.168.10.150 Il range dinamico che verrà utilizzato dal server DHCP di MAAS per assegnare gli indirizzi di rete ai nodi non ancora configurati.

192.168.10.151-192.168.10.160 Il range riservato, non utilizzato da MAAS, che verrà dedicato ai load balancer di MetalLB.

Ai vari nodi, successivamente alla fase di enlist, verrà assegnato un indirizzo statico casuale (auto assign), appartenente alla subnet "192.168.10.0/24", escludendo i range appena descritti. La rete "192.168.10.0/24" avrà come gateway l'indirizzo "192.168.10.254/24", che permetterà di accedere ad Internet o ad altre reti locali.

Il BMC sarà connesso, tramite Wi-Fi, ad una rete differente, separata da quella del cluster, ma comunque accessibile attraverso il gateway. Ciò garantisce una maggiore sicurezza, permettendo, eventualmente, al gateway, di filtrare il traffico diretto al BMC.

Considerato l'esiguo numero di Raspberry disponibili, si è scelto di effettuare il deployment del controller MAAS e del Controller Juju su macchine virtuali. L'hypervisor che gestirà tali macchine fornirà ad esse uno switch virtuale, connesso alla rete VLAN fisica a cui saranno connessi i vari Raspberry. La configurazione della macchina virtuale relativa al controller MAAS dovrà avvenire manualmente; al contrario, la configurazione del controller Juju avverrà automaticamente creando una macchina virtuale avviabile tramite PXE. Tale macchina verrà aggiunta, analogamente ai Raspberry, a MAAS, permettendo a Juju di effettuare, su di essa, il bootstrap del controller.

L'infrastruttura di rete adottata è mostrata in Figura 7.16

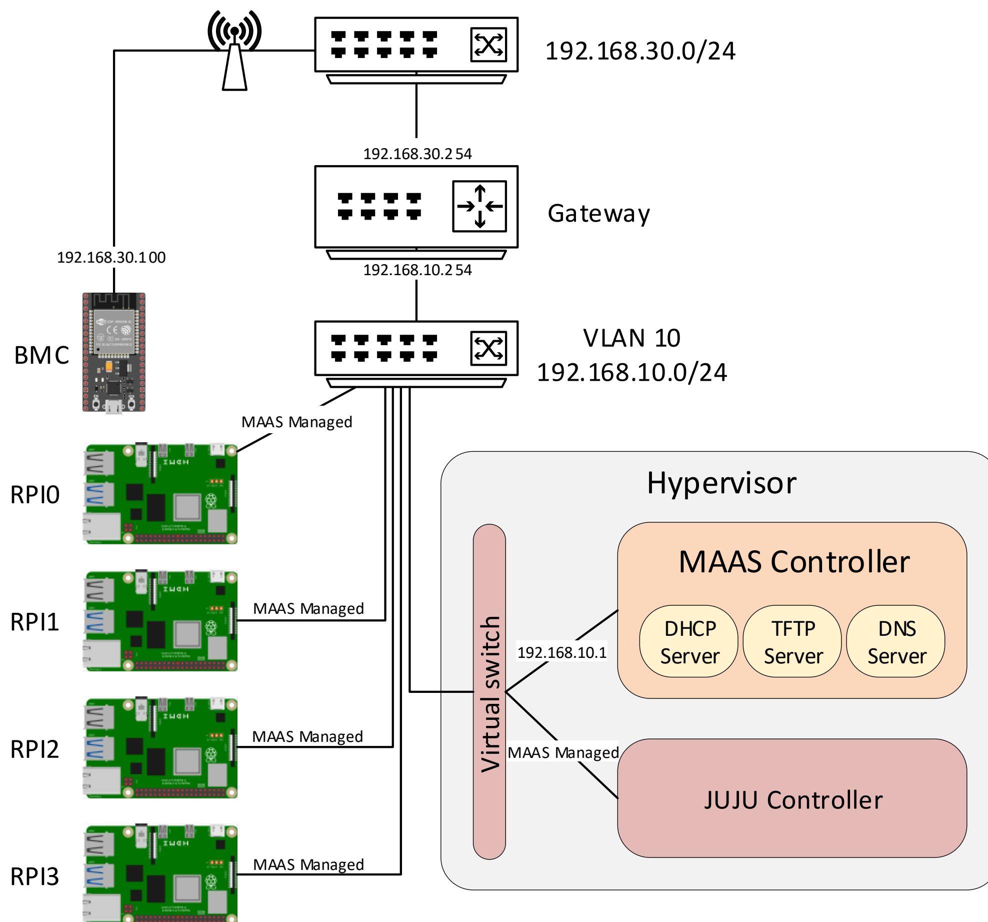

Figura 7.16: infrastruttura di rete. Fonti dei modelli: [92][91].

7.8 Infrastruttura complessiva

Complessivamente, il MAAS controller si occuperà di fornire ai vari Raspberry il firmware UEFI e, successivamente, di installare e configurare il sistema operativo Ubuntu su di essi. MAAS potrà inoltre controllare le operazioni di accensione e spegnimento dei singoli Raspberry sfruttando i servizi offerti dal BMC realizzato.

Il Juju controller si occuperà, invece, di gestire le operazioni di installazione e configurazione, sui Raspberry offerti da MAAS, di Ceph e Kubernetes. Kubernetes sfrutterà i servizi offerti da Ceph per allocare volumi persistenti. Attraverso Terraform

verrà effettuato il deployment di alcune applicazioni Kubernetes, tra cui MetalLB ed il cluster autoscaler.

Il cluster autoscaler sfrutterà gRPC per comunicare con il componente Juju client, che, a sua volta, sfrutterà le API del Juju controller per avviare eventuali operazioni scaling. Queste operazioni avranno degli effetti sul modello Juju, che, a sua volta, avrà degli effetti sul controller MAAS, che potrà allocare nuovi nodi o eliminare quelli esistenti.

L'architettura complessiva del sistema che verrà realizzato è mostrata in Figura 7.17.

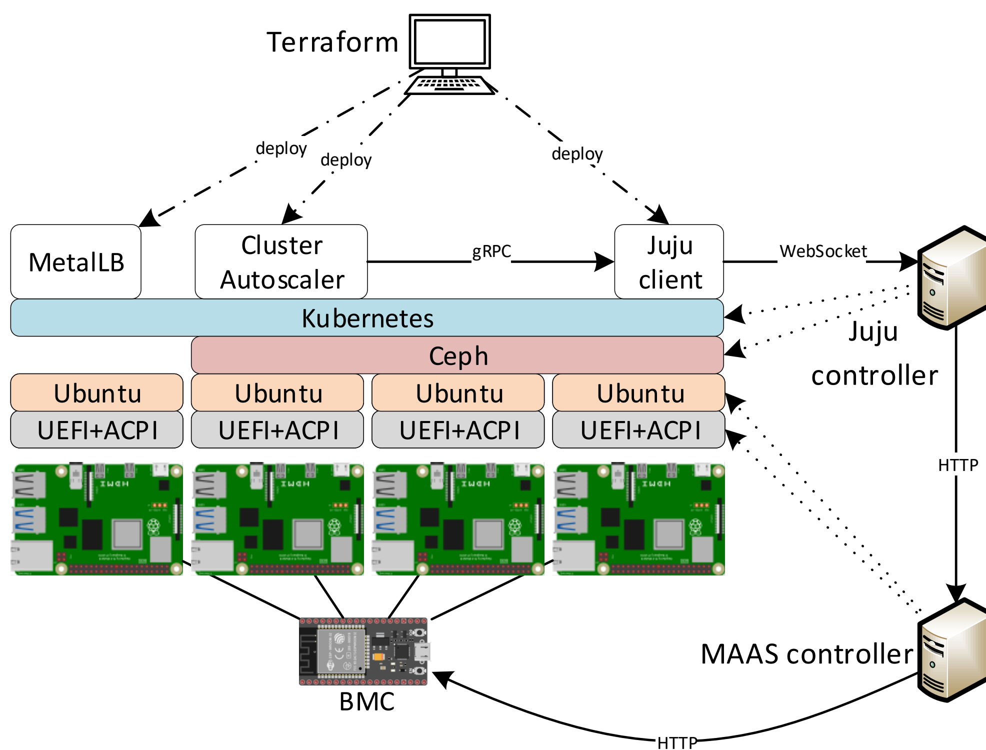

Figura 7.17: architettura complessiva del sistema. Fonti dei modelli: [92][91].

Capitolo 8

Realizzazione del cluster

In questo capitolo verranno descritti i principali problemi, che si sono verificati durante la fase di realizzazione e le relative soluzioni adottate. Saranno inoltre descritte tutte le operazioni compiute e le principali scelte implementative adottate per realizzare ciò che era stato definito in fase di progettazione.

8.1 Installazione fisica

L'installazione fisica del cluster è avvenuta senza particolari problemi. Come prima operazione è stata assemblata la struttura di supporto del cluster. Il kit relativo a tale struttura comprendeva vari dissipatori da installare su ogni Raspberry ed una ventola per ognuno di essi. Su ogni Raspberry sono stati quindi installati i dissipatori, ed è stata connessa la ventola ai pin 4 e 6, corrispondenti a 5V e massa. Inoltre, sono stati preinstallati, per ognuno di essi, tre jumper, sui pin 5, 8 e 39 (GPIO3, GPIO14 e massa), per permettere la successiva connessione del BMC.

Procedendo in questo modo per ogni livello, fissando i Raspberry con i relativi supporti, è stata ottenuta la struttura completa. Sono stati quindi connesse le varie pendrive USB, gli alimentatori ed i cavi di rete. Lo switch di rete, connesso ai vari Raspberry, è stato connesso ad un altro switch managed, configurato per inoltrare il relativo traffico sulla VLAN10. Sarà su tale VLAN che verrà reso disponibile il controller MAAS ed il controller Juju.

Il cluster fisico realizzato è mostrato in Figura 8.1 e in Figura 8.2.

Figura 8.1: cluster fisico realizzato.

Figura 8.2: cluster fisico realizzato, vista laterale.

8.2 Ricompilazione e configurazione del firmware UEFI

Per rendere il sistema funzionante, è stato necessario risolvere alcuni problemi relativi al firmware UEFI. Inoltre, è stato necessario aggiungere sia il power button ad ACPI, per permettere al BMC di spegnere il sistema, sia la logica necessaria al riordinamento automatico delle opzioni di boot e, infine, ricompilare il firmware per rendere disponibili tutte le modifiche realizzate. Oltre a ciò, è stato necessario modificare la configurazione della EEPROM di ogni Raspberry per abilitare il boot da rete.

Tutte le considerazioni descritte e le modifiche realizzate fanno riferimento alla versione 1.21 del firmware UEFI per Raspberry [108]. Probabilmente, in futuro, le problematiche citate o le funzionalità che è stato necessario aggiungere non rimarranno tali.

8.2.1 Problemi riscontrati

In seguito all'installazione del firmware UEFI sono stati riscontrati due problemi fondamentali, che compromettevano la compatibilità con MAAS. In particolare:

Driver UEFI della scheda di rete

Il Raspberry Pi 4 UEFI Firmware, nella versione 1.21, riesce ad effettuare il boot da rete, attraverso l'ambiente PXE messo a disposizione da MAAS. Il bootloader GRUB viene correttamente scaricato ed avviato, ma, successivamente, il processo di boot si interrompe. GRUB infatti, successivamente al suo avvio, tenta di scaricare, attraverso TFTP, il file di configurazione "grub.cfg"; tale operazione, però, fallisce, mostrando l'errore "couldn't send network packet". Apparentemente, tale errore è legato ai driver UEFI della scheda di rete: infatti, utilizzando i driver presenti nella versione 1.16 del firmware UEFI, tutto funziona correttamente.

Per modificare i driver è necessario clonare il repository del firmware UEFI [115], che contiene, al suo interno, altri repository relativi ai sorgenti EDKII. I sorgenti relativi al driver della scheda di rete si trovano nel repository edk2-platforms, nella cartella "Silicon/Broadcom/Drivers/Net/BcmGenetDxe/". I file contenuti in tale cartella, relativi alla versione 1.16 del firmware UEFI, che corrisponde al commit "8dd78ea11a" del repository edk2-platforms [19], dovranno sostituire i medesimi file relativi alla versione 1.21 del firmware.

Connessione di tutte le sorgenti di boot

Come già descritto nella Sezione 3.3.5, quando MAAS deve avviare una macchina già installata, fornisce un file "grub.cfg" che ordina a GRUB di ricercare il bootloader del sistema operativo installato tra tutti i dischi connessi al sistema e di avviarlo. Quando il firmware UEFI viene avviato normalmente, cioè senza entrare nella configurazione, applica però una strategia per velocizzare l'operazione di boot. Tale strategia, chiamata fast boot, prevede di collegare, nell'ambiente UEFI, solamente la sorgente da cui è stato effettuato il boot. Per questo, impostando il boot da PXE, non verrà connesso alcun disco del sistema. Le applicazioni UEFI avviate tramite PXE, tra cui GRUB, non potranno quindi accedere a tali dischi. Questo comportamento è stato introdotto dal seguente commit: [99].

A causa di ciò, il processo di boot del sistema operativo si bloccava, poiché GRUB non era in grado di individuare il disco sul quale risiedeva il bootloader del sistema operativo installato. Per risolvere tale problema è stato necessario aggiungere, alla riga 635 del file "Platform/RaspberryPi/Library/PlatformBootManagerLib/PlatformBm.c" [116], contenuto nel repository edk2-platforms, la seguente linea di codice:

```
EfiBootManagerConnectAll();
```

L'invocazione di tale funzione permette connettere tutte le sorgenti di boot disponibili, consentendo a GRUB di accedere ai vari dischi connessi al sistema.

8.2.2 Aggiunta del power button in ACPI

Come descritto in fase di progettazione, è stato aggiunto, alle tabelle ACPI del firmware UEFI, un power button, associato ad un pin dell'interfaccia GPIO, per permettere lo spegnimento del sistema senza installare del software aggiuntivo sul sistema operativo. Un power button è un dispositivo che rappresenta il bottone di accensione e spegnimento della macchina, ma, in questo caso, verranno considerate esclusivamente le funzionalità di spegnimento.

I vari dispositivi ACPI sono dichiarati nella tabella DSDT. Il codice sorgente ASL relativo a tale tabella si trova nel repository edk2-platforms, nel file "Platform/RaspberryPi/AcpiTables/Dsdt.asl" [112]. Alla fine di tale file, è stato quindi dichiarato un device di tipo power button, attraverso il seguente codice:

```
Device (PWRB) {

  Name(_HID, "PNP0C0C")

}
```

In particolare, PWRB è un nome generico, necessario per riferirsi a tale device in altre parti del codice. Il valore "PNP0C0C" della proprietà HID (Hardware Identifier), indica invece che tale device è un power button [5, pag.323]. Il sistema operativo, leggendo tale HID, caricherà quindi, per tale dispositivo, i driver corrispondenti.

La mera definizione di tale dispositivo è, però, inutile, in quanto deve essere anche presente la logica che permetta di determinare quando il pulsante viene effettivamente premuto. Al momento della pressione del pulsante fisico, il power button deve essere notificato, attraverso il metodo "Notify", con il valore "0x80". Il driver del sistema operativo associato al power button, alla ricezione di tale notifica, avvierà le operazioni di spegnimento del sistema.

In questo caso, la pressione del pulsante fisico corrisponde all'evento di cambio di stato del GPIO3 del Raspberry: al verificarsi di un cambiamento dello stato da alto a basso (falling edge) del GPIO3, dovrà essere notificato il power button, che avvierà le operazioni di spegnimento.

GPIO-signaled ACPI Events

La specifica di ACPI descrive una modalità per reagire agli eventi provenienti dal GPIO [5, pag.315], definendo un oggetto "_AEI" ed un metodo "_Exx" all'interno del device ACPI rappresentante il GPIO. L'oggetto "_AEI" permette di specificare quali interrupt dell'interfaccia GPIO devono essere gestiti come eventi ACPI. Il metodo

"_Exx" permette invece di specificare un handler che verrà eseguito al verificarsi di tali eventi; all'interno di questo handler può essere, ad esempio, notificato un particolare device, come il power button.

Le tabelle ACPI del Raspberry Pi 4 UEFI Firmware possiedono tale device relativo al GPIO, nel file "Platform/RaspberryPi/AcpiTables/GpuDevs.asl" [114], con HID "BCM2845". Sfortunatamente, però, Linux non possiede ancora il driver ACPI per tale dispositivo. Per questo, il codice specificato all'interno di tale device non verrà mai eseguito.

Altrimenti, sarebbe stato possibile notificare il power button a seguito di un evento sul GPIO3, tramite il codice in Elenco 8.1.

Elenco 8.1: codice da aggiungere al device GPIO, nel file "GpuDevs.asl", per notificare il power button all'accadere di un evento falling edge sul GPIO3; questa strategia non è purtroppo applicabile poiché Linux non possiede ancora i driver per tale device.

```
Name (_AEI, ResourceTemplate () {
  GpioInt (Edge, ActiveLow, Exclusive, PullUp, 0, "\\_SB_.GDV0.GPI0")
  {
    3 // GPIO3
  }
})
Method (_E03) { // Handles GPIO3 event
  Notify (\_SB_.PWRB, 0x80) // Notify power button
}
```

Nel codice in Elenco 8.1, GpioInt definisce un particolare interrupt del GPIO a cui ACPI potrà reagire; in particolare definisce un evento di tipo falling edge sul GPIO3. Il metodo "_E03", tramite il codice 03, specifica che reagirà al rispettivo evento con codice 3, che corrisponde al GPIO3. Al verificarsi di tale evento, verrà notificato il power button con "0x80".

Poiché il kernel Linux non possiede i driver ACPI del "BCM2845", è stato necessario realizzare un codice AML capace di interagire direttamente con i registri del GPIO del Raspberry Pi 4.

Raspberry Pi 4 GPIO

Per interagire direttamente con l'interfaccia GPIO del BCM2711, senza l'utilizzo di appositi driver, è necessario comprendere le sue modalità di funzionamento.

L'interfaccia GPIO del BCM2711 è composta da 58 linee, suddivise in tre gruppi (bank) [18, pag.64]. Il bank 0 comprende i GPIO dallo 0 al 27, il bank 1 dal 28 al 45 mentre il bank 2 dal 46 al 57. I GPIO del primo bank sono quelli esposti sull'header esterno del Raspberry, mentre gli altri non sono esposti, poiché vengono utilizzati per gestire periferiche interne.

L'interfaccia GPIO possiede quattro linee di interrupt dedicate, che scateneranno un interrupt al momento della modifica dei bit relativi ai registri di stato di event detect, descritti successivamente. Ogni bank ha una propria linea di interrupt dedicata; la quarta linea è invece condivisa tra tutti i bank, scatenando un interrupt alla modifica dei bit event detect di uno qualsiasi dei tre bank.

L'accesso all'interfaccia GPIO avviene tramite un insieme di registri, che permettono di impostare la funzionalità del pin, determinare il verificarsi di eventi o verificare lo stato dei vari pin del GPIO.

Ogni registro è composto da 32 bit; ogni bit del registro si riferisce ad un particolare pin del GPIO. Solitamente, ma per alcuni registri esistono varianti, il bit i-esimo, partendo dal meno significativo, specifica il GPIO i-esimo. Poiché i pin del GPIO sono 57, sono necessari almeno due registri da 32 bit per referenziarli completamente.

Tali registri sono referenziabili partendo dall'indirizzo base "0x7e200000" [18, pag.65], mappato, per il BCM2711, all'indirizzo "0xfe200000". I registri disponibili sono elencati in Tabella 8.1; per maggiori dettagli sul loro utilizzo è possibile consultare la relativa documentazione [18, pag.65].

Offset	Nome	Descrizione
0x00	GPFSEL0	Function Select 0
0x04	GPFSEL1	Function Select 1
0x08	GPFSEL2	Function Select 2
0x0c	GPFSEL3	Function Select 3
0x10	GPFSEL4	Function Select 4
0x14	GPFSEL5	Function Select 5
0x1c	GPSET0	Pin Output Set 0
0x20	GPSET1	Pin Output Set 1
0x28	GPCLR0	Pin Output Clear 0
0x2c	GPCLR1	Pin Output Clear 1
0x34	GPLEV0	Pin Level 0
0x38	GPLEV1	Pin Level 1
0x40	GPEDS0	Pin Event Detect Status 0
0x44	GPEDS1	Pin Event Detect Status 1
0x4c	GPREN0	Pin Rising Edge Detect Enable 0
0x50	GPREN1	Pin Rising Edge Detect Enable 1
0x58	GPFEN0	Pin Falling Edge Detect Enable 0
0x5c	GPFEN1	Pin Falling Edge Detect Enable 1
0x64	GPHEN0	Pin High Detect Enable 0
0x68	GPHEN1	Pin High Detect Enable 1
0x70	GPLEN0	Pin Low Detect Enable 0
0x74	GPLEN1	Pin Low Detect Enable 1
0x7c	GPAREN0	Pin Async. Rising Edge Detect 0
0x80	GPAREN1	Pin Async. Rising Edge Detect 1
0x88	GPAFEN0	Pin Async. Falling Edge Detect 0
0x8c	GPAFEN1	Pin Async. Falling Edge Detect 1
0xe4	GPIO_PUP_PDN_CNTRL_REG0	Pull-up / Pull-down Register 0
0xe8	GPIO_PUP_PDN_CNTRL_REG1	Pull-up / Pull-down Register 1
0xec	GPIO_PUP_PDN_CNTRL_REG2	Pull-up / Pull-down Register 2
0xf0	GPIO_PUP_PDN_CNTRL_REG3	Pull-up / Pull-down Register 3

Tabella 8.1: registri GPIO del BCM2711. Fonte: [18, pag.65].

I registri principali che dovranno essere utilizzati per implementare il power button sono i seguenti:

Function Select (GPFSEL) Tramite tali registri, il GPIO3 dovrà essere impostato in modalità di input.

Pull-up Register (GPIO_PUP_PDN_CNTRL_REG) Tramite tali registri, il GPIO3 dovrà essere impostato con una resistenza di pull-up; in tal modo, nello stato scollegato, possiederà un valore logico alto.

Pin Falling Edge Detect Enable (GPFEN) Tramite tali registri, dovrà essere abilitata la rilevazione di eventi falling edge sul GPIO3, cioè quando tale pin passa da uno stato logico alto ad uno basso; questo evento corrisponderà alla pressione del pulsante di spegnimento.

Pin Event Detect Status (GPEDS) Il bit in posizione i-esima conterrà il valore 1 se si è verificato un evento del tipo definito dagli altri registri, come GPFEN, sul GPIO i-esimo. In questo caso, se il registro GPEDS0 conterrà un 1 nel bit corrispondente al GPIO3, significherà che su tale pin si è verificato un evento di falling edge. Il cambiamento di stato di questi registri scatena un interrupt sulla relativa linea di interrupt. Ognuno dei bit di questo registro può essere reimpostato a 0, scrivendo un 1 nella relativa posizione (W1C, write one clear).

La configurazione iniziale del GPIO3, comprendente l'impostazione del pin come pin di output, della resistenza di pull-up e la rilevazione di eventi falling edge, verrà effettuata aggiungendo il relativo codice nell'inizializzazione del firmware UEFI. La logica di reazione a tali eventi sarà invece implementata nella tabella DSDT di ACPI.

Configurazione del GPIO3 su UEFI

Al momento dell'inizializzazione del firmware UEFI, dovrà essere configurato il GPIO3 in modo tale da impostarlo come pin di input, da abilitare la resistenza di pull-up e la rilevazione di eventi falling edge. Per realizzare ciò, è stato inserito, nel file "Platform/RaspberryPi/Drivers/ConfigDxe/ConfigDxe.c" [111] del repository edk2-platforms, alla riga 590, il codice mostrato in Elenco 8.2.

Elenco 8.2: codice da aggiungere al file "ConfigDxe.c" per configurare il GPIO3.

```
//ACPI Power Button
GpioPinFuncSet(3, GPIO_FSEL_INPUT);
GpioSetPull(3, GPIO_PULL_UP);
GpioPinFallingEdgeSet(3);
```

La funzione per impostare il rilevamento degli eventi falling edge non era disponibile. Per questo è stato necessario implementare la funzione GpioPinFallingEdgeSet. Tale funzione è stata aggiunta al file "Silicon/Broadcom/Bcm283x/Library/GpioLib/GpioLib.c" [113] ed al relativo "GpioLib.h", tramite il codice mostrato in Elenco 8.3.

Elenco 8.3: codice da aggiungere al file "GpioLib.c" per implementare la funzione GpioPinFallingEdgeSet, che permette di abilitare il rilevamento degli eventi falling edge

su un particolare pin del GPIO. Basato sul codice delle funzioni già presenti nel file "GpioLib.c" [113].

```
VOID
GpioPinFallingEdgeSet (
  IN  UINTN Pin
  )
{
  EFI_PHYSICAL_ADDRESS Reg;
  UINTN RegIndex;
  UINTN SelIndex;

  ASSERT (Pin < GPIO_PINS);

  RegIndex = Pin / 32;
  SelIndex = Pin % 32;

  Reg = GPIO_GPFEN0;
  Reg += RegIndex * sizeof (UINT32);
  MmioWrite32 (Reg, 1 << SelIndex);
}
```

Il codice mostrato in Elenco 8.3, realizzato sulla base delle funzioni già presenti nel file "GpioLib.c", si limita ad impostare ad 1, sul registro GPFEN, il bit relativo al GPIO fornitogli in input.

ACPI Generic Event Device

Per reagire agli eventi relativi al GPIO, è stato utilizzato il dispositivo Generic Event Device. Tale device permette di generare eventi ACPI alla ricezione, da parte dell'OSPM, di determinati interrupt di sistema. Tali interrupt devono essere specificati nell'oggetto "_CRS". Attraverso il metodo "_EVT", è possibile definire gli handler che dovranno essere eseguiti al verificarsi degli eventi ACPI generati; all'interno di tale handler può essere, ad esempio, notificato un particolare device, come il power button. L'interrupt a cui si deve reagire corrisponde a quello relativo alla prima linea di interrupt dell'interfaccia GPIO, che viene scatenato ogniqualvolta si verifica un cambiamento di stato del registro GPEDS0, sui pin relativi al bank 0. Sulla base della configurazione definita precedentemente, un evento falling edge sul GPIO3 scatenerà questo interrupt. Attraverso il Generic Event Device si reagirà ad esso, e, nel caso l'interrupt fosse stato causato dal GPIO3, si procederà alla notifica del power button.

Il codice AML relativo al Generic Event Device è stato aggiunto alla fine del file "Platform/RaspberryPi/AcpiTables/Dsdt.asl" [112], successivamente alla definizione del power button. Tale codice è mostrato in Elenco 8.4.

Elenco 8.4: codice da aggiungere al file "Dsdt.asl" per implementare un Generic Event Device capace di reagire agli eventi del GPIO3. Basato su parte del codice presente nel file "Platform/RaspberryPi/AcpiTables/SsdtThermal.asl" [118].

```
Device (GED1) {

    Name(_HID, "ACPI0013")

    OperationRegion (GPIO, SystemMemory, GPIO_BASE_ADDRESS, 0x1000)
    Field (GPIO, DWordAcc, NoLock, Preserve) {
        Offset (0x1C),
        GPS0, 32, //0x1C  GPSET0  GPIO Pin Output Set 0
        GPS1, 32, //0x20  GPSET1  GPIO Pin Output Set 1
        RES1, 32, //0x24
        GPC0, 32, //0x28  GPCLR0  GPIO Pin Output Clear 0
        GPC1, 32, //0x2C  GPCLR1  GPIO Pin Output Clear 1
        RES2, 32, //0x30
        GPL1, 32, //0x34  GPLEV0  GPIO Pin Level 0
        GPL2, 32, //0x38  GPLEV1  GPIO Pin Level 1
        RES3, 32, //0x3C
        GPE0, 32, //0x40  GPEDS0  GPIO Pin Event Detect Status 0
        GPE1, 32  //0x44  GPEDS1  GPIO Pin Event Detect Status 1
    }


    Name(_CRS, ResourceTemplate () {
        Interrupt (ResourceConsumer, Level, ActiveHigh, Shared) {
            BCM2386_GPIO_INTERRUPT0
        }
    })


    Method (_EVT, 1) {
        Switch (Arg0)
        {
            Case (BCM2386_GPIO_INTERRUPT0) {
                if (GPE0 & (1 << 3)) { // Check GPIO3
                    Store (1 << 3, GPE0) // Clear register
                    Notify (\_SB_.PWRB, 0x80) // Notify power button
                }
            }
        }
    }
}
```

Nell'Elenco 8.4, l'HID "ACPI0013" identifica device come un Generic Event Device. L'OperationRegion definisce un'area di memoria sulla quale si potrà operare e che, in questo caso, corrisponde all'area relativa ai registri del GPIO. Il Field permette di assegnare un nome a determinati blocchi di byte di un'area di memoria, per poterli

referenziare facilmente all'interno del codice AML. È stato assegnato quindi, un nome ai blocchi corrispondenti a vari registri del GPIO. L'oggetto "_CRS" specifica l'interrupt "BCM2386_GPIO_INTERRUPT0", che corrisponde all'interrupt prodotto dalla prima linea di interrupt del GPIO a cui si dovrà reagire. Il metodo "_EVT" specifica gli handler da eseguire al verificarsi degli interrupt contenuti nell'oggetto "_CRS". Nel caso dell'interrupt "BCM2386_GPIO_INTERRUPT0", viene controllato il registro GPEDS0 per verificare che tale interrupt sia stato scatenato dal GPIO3. In caso affermativo, viene reimpostato a 0 il relativo bit del registro e notificato il power button con "0x80", per avviare le operazioni di spegnimento.

Tramite lo strumento "acpi_listen", che permette di intercettare gli eventi ACPI del sistema, è stato possibile verificare il corretto funzionamento del power button, come mostrato in Figura 8.3.

```
ubuntu@rpi0:~$ acpi_listen
button/power PBTN 00000080 00000000
button/power PNP0C0C:00 00000080 00000001
Connection to 192.168.10.59 closed by remote host.
Connection to 192.168.10.59 closed.
```

Figura 8.3: evento ACPI che si verifica quando il GPIO3 viene portato temporaneamente a massa.

8.2.3 Ordinamento automatico delle opzioni di boot

Come già descritto nella Sezione 7.3.1, il firmware UEFI avviato su ogni Raspberry, successivamente alla generazione delle opzioni di boot, dovrà modificare l'ordine di boot automaticamente in modo tale che l'opzione relativa a PXE risulti essere la prima della lista. Tale comportamento è stato implementato sfruttando la funzione, messa a disposizione da EDKII, EfiBootManagerSortLoadOptionVariable, che permette di riordinare la lista di boot attraverso un comparatore. Basandosi sul codice preesistente [44], che definiva un comparatore capace di impostare l'UEFI Shell come prima opzione di boot, è stato realizzato un comparatore analogo, capace di assegnare una priorità più alta, durante l'ordinamento, all'opzione UEFI PXEv4. Tale codice è mostrato in Elenco 8.5.

Elenco 8.5: codice che implementa un comparatore tra le opzioni di boot UEFI, assegnando una priorità più elevata all'opzione UEFI PXEv4. Basato sul seguente codice: [44].

```
/**
  Returns the priority number.
  @param BootOption
**/
UINTN
```

```
BootOptionPriority (
  CONST EFI_BOOT_MANAGER_LOAD_OPTION *BootOption
  )
{
  //
  // Make sure PXEv4 is first
  //
  if (StrnCmp (BootOption->Description,
               L"UEFI PXEv4", StrLen (L"UEFI PXEv4")) == 0) {
    return 0;
  }
  return 100;
}

INTN
EFIAPI
CompareBootOption (
  CONST EFI_BOOT_MANAGER_LOAD_OPTION  *Left,
  CONST EFI_BOOT_MANAGER_LOAD_OPTION  *Right
  )
{
  return BootOptionPriority (Left) - BootOptionPriority (Right);
}
```

In particolare, la funzione BootOptionPriority restituisce la priorità relativa ad una determinata opzione di boot: se l'opzione ha, come descrizione, la stringa "UEFI PXEv4", viene restituito un valore più basso, che, durante l'ordinamento, determinerà la posizione all'inizio della lista. La funzione CompareBootOption implementa il comparatore, sfruttando BootOptionPriority. La funzione EfiBootManagerSortLoadOptionVariable utilizza tale comparatore per determinare l'ordine della lista. Il codice che invoca tale funzione è mostrato in Elenco 8.6. Tale codice è stato inserito dopo la riga 647 del file "Platform/RaspberryPi/Library/PlatformBootManagerLib/PlatformBm.c" [116], contenuto nel repository edk2-platforms.

Elenco 8.6: codice che ordina le opzioni di boot sulla base del comparatore CompareBootOption, definito precedentemente. Basato sul seguente codice: [44].

```
EfiBootManagerRefreshAllBootOption();
EfiBootManagerSortLoadOptionVariable(
                              LoadOptionTypeBoot,
                              (SORT_COMPARE) CompareBootOption
                              );
```

Tramite queste modifiche, il firmware UEFI, all'accensione, avvierà automaticamente la prima opzione di boot, che corrisponderà a "UEFI PXEv4", come mostrato in Figura 8.4.

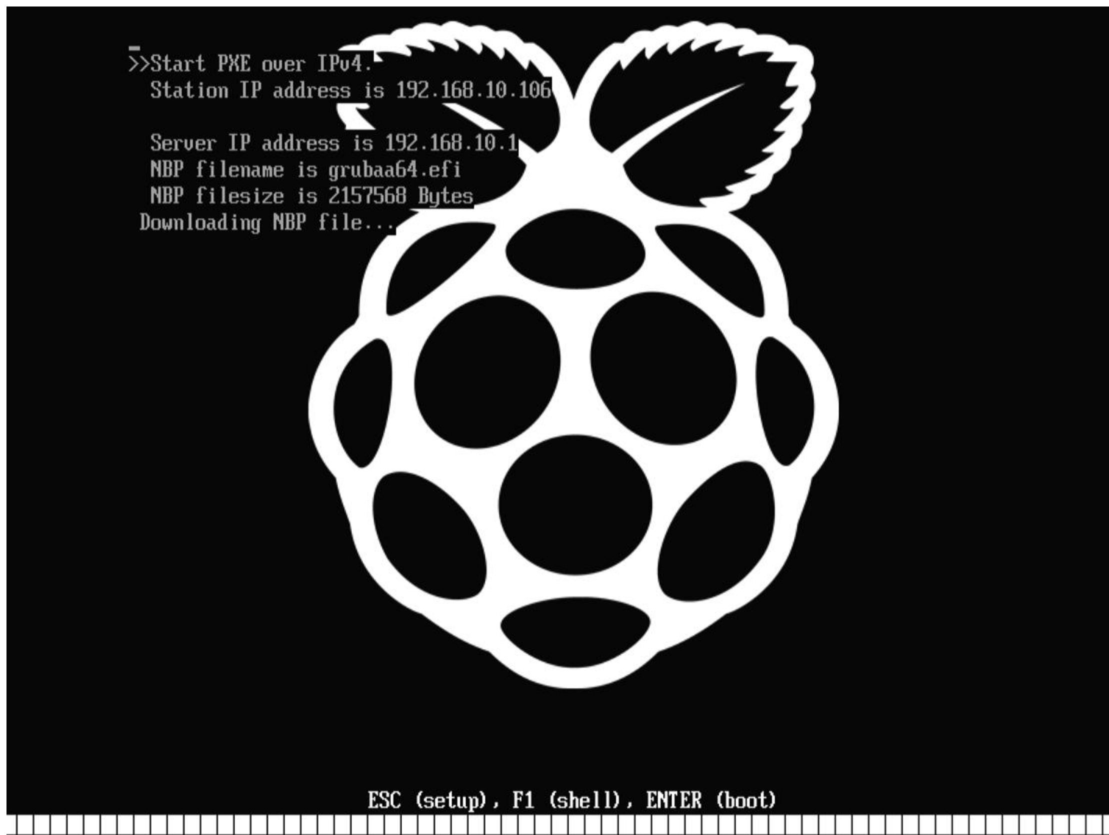

Figura 8.4: avvio automatico dell'opzione di boot "UEFI PXEv4" da parte del Raspberry Pi 4 UEFI Firmware.

8.2.4 Rimozione del limite relativo alla RAM

Come impostazione di default, la RAM utilizzabile del Raspberry viene limitata a 3 GB. Per modificare tale configurazione è stato sufficiente modificare il file "Platform/RaspberryPi/RPi4/RPi4.dsc" [117], contenuto nel repository edk2-platforms, impostando a 0 il valore alla fine della riga 502.

8.2.5 Ricompilazione del firmware UEFI

Per ricompilare il firmware UEFI, applicando le modifiche precedentemente descritte e producendo il corrispondete file "RPI_EFI.fd", è sufficiente:

1. clonare il repository "Raspberry Pi 4 UEFI Firmware Images" [115] al commit relativo alla versione 1.21;

2. eseguire i comandi specificati nella sezione install del file "appveyor.yml" [109]; installare i pacchetti gcc e g++;

3. modificare lo script "build_firmware.sh" [110] aggiungendo, all'inizio del file, la riga "#!/bin/bash"; specificare la versione della build, che verrà mostrata all'interno del firmware UEFI, dichiarando il valore della variabile "APPVEYOR_REPO_TAG_NAME". Ad esempio, aggiungendo il seguente codice dopo la riga 6 dello script:

```
export APPVEYOR_REPO_TAG_NAME="1.21.1"
```

4. eseguire lo script "build_firmware.sh";

5. al termine della compilazione, reperire il file "Build/RPi4/RELEASE_GCC5/FV/RPI_EFI.fd";

6. reperire i file "start4.elf", "fixup4.dat" e "config.txt" dalla release del firmware 1.21 [108].

8.2.6 Riconfigurazione EEPROM

Come descritto nella Sezione 7.3.1, la EEPROM di ogni Raspberry deve essere riconfigurata per abilitare il boot da rete. Per effettuare tale riconfigurazione è necessario:

1. installare Raspberry Pi OS Lite (32-bit) su una microSD, ed avviare il Raspberry con tale sistema operativo;

2. eseguire il comando "sudo rpi-eeprom-update" per verificare la presenza di aggiornamenti relativi alla EEPROM; in caso affermativo, aggiornare la EEPROM con il comando "sudo rpi-eeprom-update -a" e riavviare il sistema;

3. editare la configurazione della EEPROM tramite il comando "sudo -E rpi-eeprom-config –edit", impostando la proprietà "BOOT_ORDER" a "0xf21";

4. riavviare il sistema;

5. spegnere il sistema e rimuovere la microSD;

6. ripetere le operazioni per ogni Raspberry del cluster, utilizzando la stessa microSD, senza reinstallare il sistema operativo.

Tramite queste modifiche, il bootloader presente sulla EEPROM dei Raspberry, all'accensione, nel caso la microSD non sia presente, tenterà di caricare i file di boot relativi al firmware tramite PXE, come mostrato in Figura 8.5.

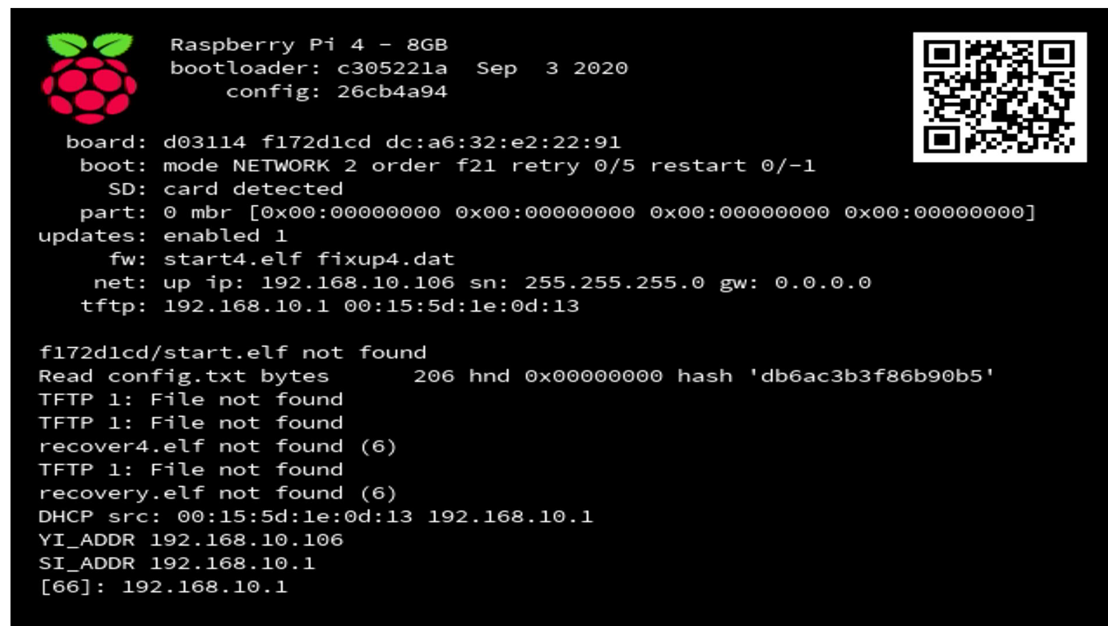

Figura 8.5: reperimento dei file di boot, relativi al firmware, tramite PXE, da parte del bootloader presente sulla EEPROM dei Raspberry.

8.3 Realizzazione BMC

La realizzazione del BMC è avvenuta senza riscontrare particolari problemi, seguendo ciò che era stato definito in fase di progettazione.

8.3.1 Programmazione microcontrollore tramite MicroPython

La programmazione dell'ESP32 è avvenuta sfruttando MicroPython [89], cioè un interprete open-source capace di eseguire codice Python su vari microcontrollori, tra cui l'ESP32.

Solitamente, l'ESP32 viene programmato in linguaggio C++, sfruttando l'SDK fornito da Espressif chiamato ESP-IDF (Espressif IoT Development Framework) [45]. Tale SDK, a meno di non utilizzare un ambiente di sviluppo preimpostato, come l'Arduino IDE, richiede una considerevole fase di set-up. Le librerie in esso contenute sono molto ricche e permettono di controllare appieno le funzionalità dell'MCU, ma,

di contro, sono molto articolate e complesse, richiedendo una notevole conoscenza dell'hardware e del linguaggio C++.

Considerando il tipo di applicazione da realizzare e quindi la non necessità di effettuare operazioni complesse a basso livello, si è deciso di utilizzare MicroPython [89].

MicroPython è un interprete Python, scritto in linguaggio C, che può essere eseguito su vari microcontrollori. Per installarlo è necessario effettuare un'operazione di flash, che permetterà di caricare l'immagine dell'interprete sulla memoria del microcontrollore. Successivamente, sarà possibile fornirgli del codice Python3, che verrà eseguito senza necessità di previa compilazione.

Il codice può essere fornito, in maniera interattiva, tramite la REPL messa a disposizione sull'interfaccia UART, o in maniera batch, caricando i vari file Python da eseguire sulla memoria del microcontrollore. Anche il caricamento dei file avviene tramite l'interfaccia UART, ma, in questo caso, i dati inviati saranno memorizzati, in maniera persistente, su un file system gestito da MicroPython. Il microcontrollore, all'avvio, eseguirà il file "main.py", che dovrà contenere l'entry point dell'applicativo. MicroPython mette inoltre a disposizione una serie di librerie di base per interagire, in maniera semplice e veloce, con l'hardware del microcontrollore. Gli unici svantaggi consistono nelle performance, che, rispetto ad un applicativo scritto attraverso il software development kit di Espressif, risultano notevolmente ridotte, e nell'impossibilità di ottenere un controllo completo dell'hardware dell'MCU. MicroPython ha però l'enorme vantaggio di permettere di realizzare applicativi in maniera agile, grazie all'elevata velocità di apprendimento delle librerie ed all'utilizzo di un linguaggio di alto livello con tipizzazione dinamica.

Considerato quindi il tipo di applicazione da realizzare, che non ha particolari esigenze di performance ma che deve gestire problematiche di interazione di alto livello, fornendo un insieme di API HTTP, si è scelto di prediligere MicroPython.

Attraverso un'apposita libreria, chiamata TinyWeb [130], è stato realizzato un server HTTP che implementa le API definite in fase di progettazione. Una chiamata a tali API avrà degli effetti sui pin del microcontrollore: un'operazione GET comporterà la lettura dello stato del pin connesso al GPIO14 di uno dei Raspberry; un'operazione PUT comporterà, invece, il cambiamento di stato, per un breve periodo, del pin connesso al GPIO3, scatenando l'accensione o lo spegnimento del relativo Raspberry,

8.3.2 Interfacciamento con Raspberry

Come descritto in fase di progettazione, i GPIO3 dei vari Raspberry sono impostati, quando scollegati, allo stato logico alto, attraverso una resistenza di pull-up. L'ESP32 dovrà portare i relativi pin, temporaneamente, a massa, per provocare un evento di cambio di stato sul GPIO dei Raspberry. Poiché i pin dell'ESP32 sono stati connessi

direttamente ai GPIO3 dei vari Raspberry, è stato necessario impostarli in una modalità di output open-drain, che prevede il seguente comportamento:

Stato logico basso Il pin viene impostato a massa.

Stato logico alto Il pin abilita una resistenza ad alta impedenza, rendendolo, in pratica, scollegato.

La modalità open-drain permette, quindi, all'ESP32 di simulare un insieme di pulsanti connessi ai GPIO3 dei vari Raspberry. Un pulsante infatti, quando non premuto, renderebbe il pin del Raspberry scollegato, mentre, quando premuto, lo imposterebbe a massa.

MicroPython permette di impostare un pin in tale modalità, tramite la costante "Pin.OPEN_DRAIN" [90].

I pin relativi alla lettura del GPIO14, sono stati invece impostati in modalità di input, con una resistenza di pull-down. In questo modo, quando il relativo Raspberry sarà privo di alimentazione ed il corrispettivo GPIO14 sarà in uno stato scollegato, il pin dell'ESP32 sarà impostato ad uno stato logico basso, che corrisponde allo stato spento del Raspberry.

I pin utilizzati per interfacciare l'ESP32 con i vari Raspberry del cluster sono riassunti in Tabella 8.2. In Figura 8.6 sono mostrati i relativi collegamenti fisici realizzati.

Raspberry	Raspberry GPIO	ESP32 GPIO
0	14 (output)	12 (input, pull-down)
	3 (input, pull-up)	13 (output, open-drain)
1	14 (output)	27 (input, pull-down)
	3 (input, pull-up)	26 (output, open-drain)
2	14 (output)	33 (input, pull-down)
	3 (input, pull-up)	25 (output, open-drain)
3	14 (output)	23 (input, pull-down)
	3 (input, pull-up)	22 (output, open-drain)

Tabella 8.2: pin utilizzati per l'interfacciamento tra il BMC ed i vari Raspberry del cluster.

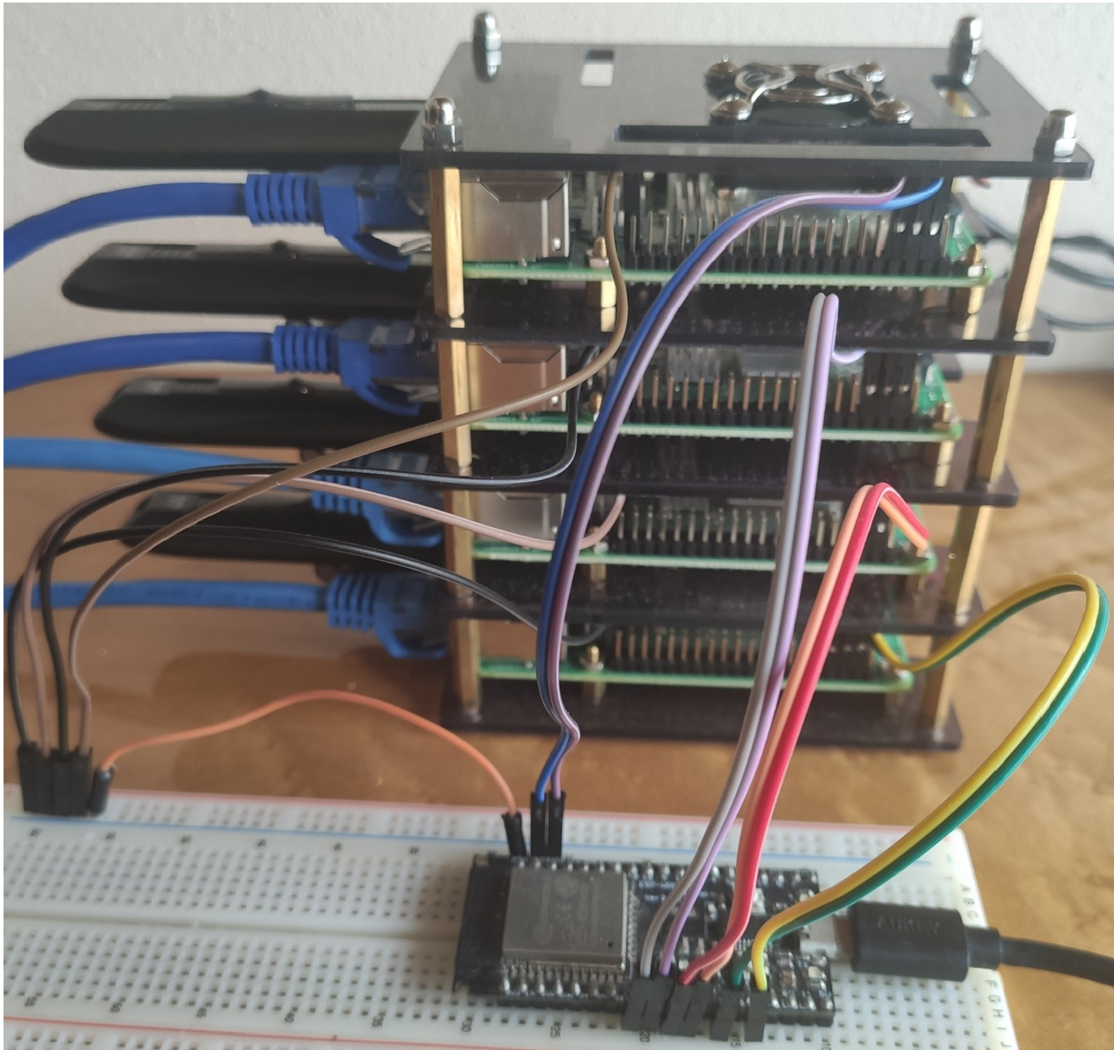

Figura 8.6: interfacciamento tra il BMC ed i vari Raspberry del cluster.

8.3.3 Realizzazione del MAAS Power Driver

I driver utilizzati da MAAS per il power management delle macchine sono raccolti nella cartella "src/provisioningserver/drivers/power" del relativo repository [79]. Per implementare un driver aggiuntivo, è stato necessario aggiungere il corrispondente file Python, aggiungendo, inoltre, la relativa dichiarazione nel file "registry.py", collocato nella medesima cartella.

Successivamente all'installazione di MAAS tramite snapcraft, tale cartella si troverà nel percorso "/snap/maas/current/lib/python3.8/site-packages/provisioningserver/drivers/power/".

Poiché snapcraft distribuisce i propri pacchetti software sfruttando delle immagini di file system accessibili in modalità di sola lettura, non è possibile aggiungere o modificare direttamente i file relativi a tale cartella. Per risolvere tale problema esistono due soluzioni: ricompilare il pacchetto snapcraft con le relative modifiche o effettuare un'operazione di mount di un'altra cartella, con permessi di scrittura, su tale percorso. L'aggiunta dei file relativi al driver personalizzato è avvenuta tramite quest'ultima modalità:

1. la cartella "power" è stata copiata nella home directory; su tale copia, a differenza dell'originale, sarà possibile effettuare modifiche;

2. alla copia di tale cartella è stato quindi aggiunto il file relativo al nuovo power driver ed è stato modificato il file "registry.py";

3. attraverso il comando

   ```
   sudo mount --bind -o nodev,ro ~/power
     /snap/maas/current/lib/python3.8/site-packages/
     provisioningserver/drivers/power
   ```

   è stata effettuata un'operazione di mount della copia sul percorso della cartella originale;

4. attraverso il comando "sudo snap restart maas.supervisor" è stato riavviato il servizio relativo al controller MAAS, in modo tale che potesse caricare i nuovi file sorgente.

Lo svantaggio di questa modalità è la necessità di rieseguire il comando di mount e di restart di MAAS successivamente ad ogni riavvio della macchina; in compenso, però, è possibile installare il pacchetto snapcraft dalle sorgenti originali, che lo manterranno aggiornato, senza necessità di ricompilarlo.

L'implementazione di un power driver avviene tramite una classe Python, che definisce i metodi per interagire con il BMC. I principali, tra tali metodi, sono i seguenti:

power_query MAAS utilizza questo metodo per verificare lo stato della macchina. In questo caso, il metodo utilizza la libreria Requests per effettuare un'operazione di GET sulle API offerte dall'ESP32.

power_on MAAS utilizza questo metodo per accendere la macchina. In questo caso, il metodo utilizza la libreria Requests per effettuare un'operazione di PUT sulle API offerte dall'ESP32. Successivamente, attende il cambiamento di stato della macchina, reperendolo periodicamente tramite il metodo "power_query". Nel caso la richiesta venisse rifiutata o il cambiamento di stato non avvenisse, varrà lanciata un'eccezione di tipo PowerError.

power_off MAAS utilizza questo metodo per spegnere la macchina. L'implementazione è analoga a "power_on".

Oltre a tali metodi, la classe permette di definire un insieme di impostazioni, i cui rispettivi valori potranno essere forniti al momento della configurazione del driver su una specifica macchina di MAAS. In questo caso, sono state definite due impostazioni: l'indirizzo dell'ESP32, per raggiungere le API, e l'identificativo del Raspberry, per specificare al BMC su quale Raspberry del cluster verranno effettuate le operazioni. Queste impostazioni saranno definibili dall'interfaccia web di MAAS, come mostrato in Figura 8.7.

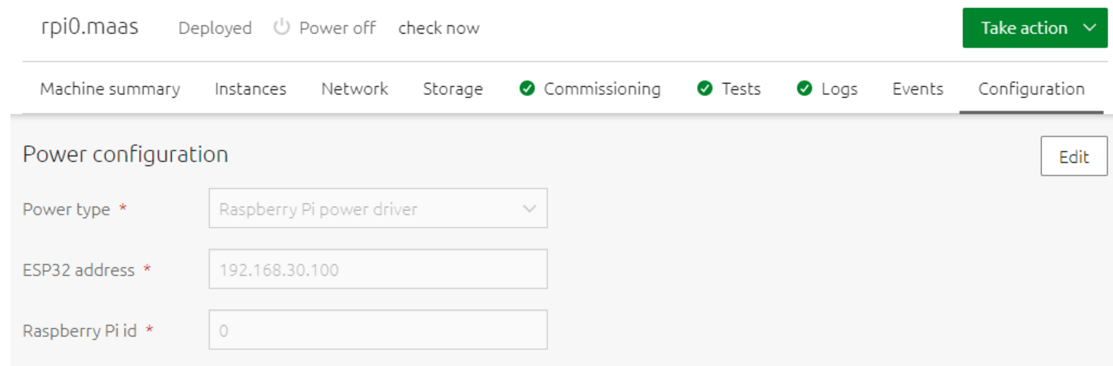

Figura 8.7: interfaccia web di MAAS per la configurazione del power driver, relativa alla macchina rpi0.

Impostando e configurando adeguatamente il power driver per ogni Raspberry del cluster, è stato possibile verificare il loro stato o avviare le operazioni di accensione o spegnimento tramite l'interfaccia web di MAAS, come mostrato in Figura 8.8.

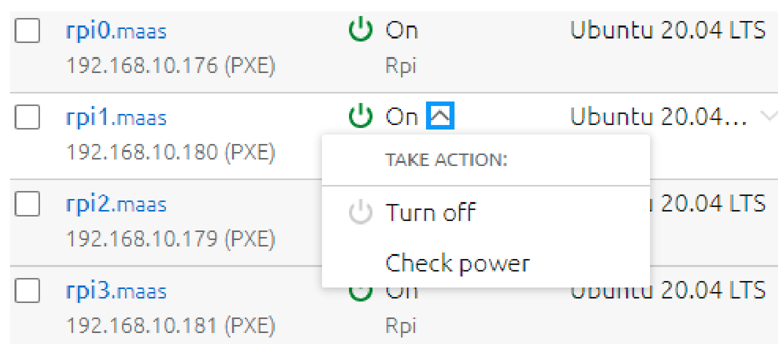

Figura 8.8: interfaccia web di MAAS che permette di verificare lo stato dei vari Raspberry o di avviare le operazioni di accensione o spegnimento.

8.4 Installazione MAAS

L'installazione e la configurazione del controller MAAS sono avvenute senza riscontrare particolari problemi, seguendo ciò che era stato definito in fase di progettazione.

8.4.1 Deployment e configurazione del controller

Come descritto in fase di progettazione, il controller MAAS è stato installato su una macchina virtuale. Nello specifico, si è scelto di utilizzare, come hypervisor, Hyper-V, presente nei sistemi operativi Windows più recenti. In ogni caso, un qualsiasi altro hypervisor sarebbe risultato adatto allo scopo.

Attraverso Hyper-V, è stata quindi creata una macchina di prima generazione, con 4 core, 4 GB di RAM ed un'interfaccia di rete impostata sulla VLAN 10, per abilitare la comunicazione con la rete del cluster. Successivamente, è stata quindi avviata l'installazione di una release di Ubuntu server, nello specifico, di Ubuntu Groovy 20.10. Durante la configurazione iniziale, l'interfaccia di rete è stata impostata sull'indirizzo statico "192.168.10.1/24", con gateway "192.168.10.254/24", come specificato in fase di progettazione.

Al termine dell'installazione di Ubuntu, sono state compiute le seguenti operazioni:

1. MAAS 2.9 è stato installato attraverso snapcraft, tramite il comando:
   ```
   sudo snap install maas --channel=2.9/stable
   ```
2. il database PostgreSQL, necessario a MAAS per memorizzare le proprie configurazioni, è stato installato tramite il comando:
   ```
   sudo snap install maas-test-db --channel=2.9/beta
   ```
3. i controller MAAS sono stati inizializzati, fornendo l'URL del database, tramite il comando:
   ```
   sudo maas init region+rack --database-uri maas-test-db:///
   ```
4. l'utente amministratore è stato creato, tramite il comando:
   ```
   sudo maas createadmin
   ```
5. tramite le credenziali appena specificate, è stato effettuato l'accesso all'interfaccia web; sono state importate le chiavi SSH per permettere l'accesso alle macchine installate da MAAS e sono state selezionate le immagini Ubuntu da scaricare. Nello specifico, è stata selezionata la release di Ubuntu 20.04 per le architetture amd64 e arm64. Nel menu subnet, è stato abilitato il server DHCP e sono stati

specificati i vari range di indirizzi, come descritto nella fase di progettazione. Nelle impostazioni, è stato selezionato, come default, il kernel hwe-20.04 per le operazioni di commissioning, e Ubuntu 20.04 per il deployment;

6. attraverso il comando

   ```
   maas login <profilo> http://192.168.10.1:5240/MAAS <API Key>
   ```

   è stato configurato un profilo, per poter accedere, attraverso la CLI di MAAS, al controller, sfruttando l'utente appena creato. L'API Key è stata reperita dall'interfaccia web di MAAS nelle impostazioni utente;

7. attraverso il comando

   ```
   maas <profilo> maas set-config name=node_timeout value=60
   ```

   è stato aumentato a 60 minuti il tempo massimo ammissibile per le operazioni di deployment. Infatti, la configurazione di default di 30 minuti, a causa dell'utilizzo di pendrive non particolarmente veloci, si è rivelata, spesso, non sufficiente.

Definizione del DHCP snippet

Come descritto nella fase di progettazione, è stato necessario realizzare un DHCP snippet per identificare i client corrispondenti ai bootloader relativi alla EEPROM, assegnando le opportune opzioni DHCP. Tale snippet deve aderire al formato di "dhcpd.conf", il file di configurazione di dhcpd.

Lo snippet realizzato è mostrato in Elenco 8.7.

Elenco 8.7: snippet per il server DHCP di MAAS che permette di definire le opzioni 43 e 66 nel caso il client sia il bootloader relativo alla EEPROM di un Raspberry.

```
if option vendor-class-identifier="PXEClient:Arch:00000:UNDI:002001"{
    # Option 43
    option vendor-encapsulated-options "Raspberry Pi Boot";

    #Option 66
    option tftp-server-name "192.168.10.1";
}
```

Se la richiesta DHCP contiene l'opzione vendor class identifier con valore "PXEClient:Arch:00000:UNDI:002001", significa che il client è il bootloader relativo alla EEPROM del Raspberry. In tal caso, vengono quindi impostate le opzioni 43 e 66. L'opzione 66 indica, come server TFTP, quello offerto da MAAS, che dovrà fornire il firmware UEFI. Per questo, è stato necessario aggiungere, a tale server, i relativi file.

I file "RPI_EFI.fd", "start4.elf", "fixup4.dat" e "config.txt" sono stati quindi copiati nella root del server TFTP di MAAS, localizzata in "/var/snap/maas/common/maas/boot-resources/current/".

Lo snippet è stato applicato a MAAS attraverso l'interfaccia web, tramite il menu DHCP snippets presente nelle impostazioni. Esso è stato inoltre definito come di tipo "Subnet", applicandolo esclusivamente alla subnet del cluster, cioè "192.168.10.0/24".

8.4.2 Enlist e configurazione dei vari Raspberry

Applicando le configurazioni e le modifiche precedentemente descritte, è stato impostato l'ambiente necessario ai vari Raspberry per effettuare il boot da rete e, in generale, per poter essere gestisti da MAAS.

I Raspberry sono stati quindi accesi manualmente, in modo da avviare, su MAAS, l'operazione di enlist. Al termine di tale fase, è stato quindi configurato, dall'interfaccia web di MAAS, per ogni Raspberry, il power driver precedentemente realizzato. Inoltre, sono stati impostati sia il nome delle varie macchine sia gli eventuali tag. Successivamente, è stata avviata la fase di commissioning, per permettere ai vari Raspberry di entrare nello stato di Ready, cioè per renderli disponibili al deployment, come mostrato in Figura 8.9. Grazie al BMC realizzato, tutte le operazioni di accensione e spegnimento delle macchine, successive alla fase di enlist, sono state effettuate automaticamente.

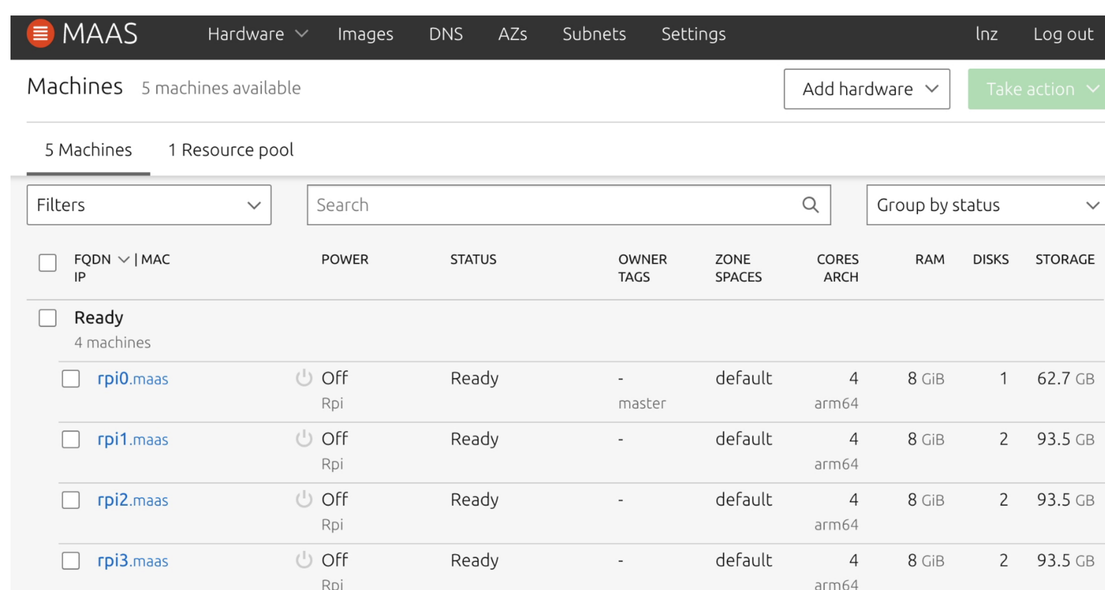

Figura 8.9: interfaccia web di MAAS che mostra i vari Raspberry nello stato di Ready.

8.4.3 Configurazione dei Raspberry come pod KVM

MAAS offre la possibilità di effettuare il deployment di pod KVM, permettendo di virtualizzare le risorse delle relative macchine. Per la realizzazione del cluster,

considerate le limitate risorse a disposizione dei vari Raspberry, si è scelto di installare i componenti di Kubernetes direttamente sulle macchine fisiche. In ogni caso, adottando le dovute modifiche, sarebbe stato possibile configurare i vari Raspberry come pod KVM, tramite le seguenti operazioni:

1. dall'interfaccia web di MAAS è stato avviato il deployment di uno dei Raspberry, abilitando l'opzione "Register as MAAS KVM host". Così facendo, è stato specificato a MAAS di eseguire, oltre alle operazioni di installazione del sistema operativo, anche le operazioni relative all'installazione e configurazione del software per la gestione delle macchine virtuali;

2. al termine delle operazioni di installazione, il Raspberry è comparso nella lista degli host KVM. Selezionandolo, sarebbe stato possibile, normalmente, creare, su tale host, una nuova macchina virtuale. Purtroppo, però, il template XML libvirt utilizzato da MAAS per la definizione delle macchine virtuali non è risultato adatto ai vari Raspberry. Per questo, è stato necessario modificarlo;

3. tale template si trova nel file "/snap/maas/current/lib/python3.8/site-packages/provisioningserver/drivers/pod/virsh.py" [84], all'interno della variabile "DOM_TEMPLATE_ARM64". È stato necessario modificare la riga 160, impostando il gic alla versione 2. Infatti, il gic (Generic Interrupt Controller) presente sul Raspberry Pi 4 B, chiamato GIC-400, implementa solamente la specifica GICv2 di Arm. Inoltre, a riga 176 e 177, è stato necessario impostare il bus su "virtio". Per applicare tali modifiche, analogamente alla modifica dei file relativi al power driver, è stato necessario effettuare un'operazione di mount di una cartella modificabile, riavviando, successivamente, il servizio di MAAS. Tale modifica ha quindi abilitato la creazione di macchine virtuali sui Raspberry configurati come pod KVM.

8.5 Installazione Juju

Anche l'installazione e la configurazione del controller Juju sono avvenute senza riscontrare particolari problemi, seguendo ciò che era stato definito in fase di progettazione. Si è verificato, però, qualche problema relativamente al deployment di Ceph.

8.5.1 Deployment e configurazione del controller

Il controller Juju, come descritto in fase di progettazione, è stato installato, analogamente a MAAS, su una macchina virtuale Hyper-V. Attraverso Hyper-V, è stata quindi creata una macchina di seconda generazione, con 2 core, 4 GB di RAM ed

un'interfaccia di rete impostata sulla VLAN 10, per abilitare la comunicazione con la rete del cluster. Le macchine Hyper-V di seconda generazione utilizzano il firmware UEFI e permettono di effettuare il boot PXE attraverso interfacce di rete ordinarie, senza ricorrere ad interfacce di rete legacy. Per questo, risultano essere, per MAAS, più facili da gestire. La macchina è stata quindi configurata, da Hyper-V, impostando il boot da rete come prima opzione. Successivamente, la macchina è stata accesa, per avviare, su MAAS, l'operazione di enlist. Infine, dopo aver impostato il power driver manuale, è stato avviato il commissioning, per renderla disponibile al deployment.

Successivamente, sul MAAS controller, sono state eseguite le seguenti operazioni:

1. la CLI di Juju 2.9 è stata installata tramite il seguente comando:
   ```
   sudo snap install juju --channel=2.9/candidate --classic
   ```

2. tramite il comando
   ```
   juju add-cloud --local
   ```
   è stato configurato il cloud relativo a MAAS, specificando "maas" come cloud type, "rpi-cluster" come nome del cloud e "http://192.168.10.1:5240/MAAS" come API endpoint;

3. tramite il comando
   ```
   juju add-credential rpi-cluster
   ```
   sono state quindi aggiunte le credenziali per accedere al controller MAAS, specificando le relative API Key alla richiesta del "maas-oauth";

4. tramite il comando
   ```
   juju bootstrap rpi-cluster rpi-cluster-controller
     --constraints "arch=amd64"
   ```
   è stato effettuato, sul cloud gestito da MAAS precedentemente configurato, il bootstrap del controller Juju. Attraverso il vincolo "arch=amd64", è stata costretta l'installazione del controller sull'unica macchina con architettura amd64, che risulta essere la macchina virtuale precedentemente configurata;

5. la macchina virtuale relativa al controller Juju, poiché sprovvista di un BMC, è stata avviata manualmente, per permettere, a MAAS, di avviare le operazioni di deployment.

Successivamente al bootstrap del controller, Juju è stato in grado, attraverso MAAS, di utilizzare i vari Raspberry per effettuare il deployment dei propri modelli.

8.5.2 Deployment del modello relativo a Ceph e Kubernetes

I bundle relativi a Kubernetes Core [62] e Ceph Base [22], sono stati uniti in un unico modello, che richiede quattro macchine, corrispondenti ai quattro Raspberry del cluster. Uno dei Raspberry funge da Kubernetes master, ospitando le unità relative ad easyrsa, etcd e kubernetes-master; tale macchina è stata associata al Raspberry senza disco dedicato a Ceph, definendo, su MAAS, per quella macchina, un tag con valore "master". Gli altri tre nodi fungono invece da Kubernetes worker, ospitando anche le unità relative a ceph-mon e ceph-osd. L'applicazione ceph-mon è, inoltre, relazionata al kubernetes-master; relazione che comporta l'avvio, da parte del kubernetes-master, di un insieme di pod, che rendono disponibili un insieme di servizi necessari all'interazione tra il cluster kubernetes e Ceph. Tra essi, vi è, ad esempio, il provisioner, che permette l'allocazione automatica di volumi Ceph. Inoltre, vengono configurate automaticamente due storage class, utilizzabili dai pod Kubernetes: "storageclass.storage.k8s.io/ceph-xfs" e "storageclass.storage.k8s.io/ceph-ext4", che permettono l'utilizzo di block device Ceph, formattati, rispettivamente in xfs o ext4.

Il modello realizzato è mostrato in Elenco 8.8.

Elenco 8.8: modello Juju utilizzato per il deployment di Kubernetes e Ceph sui vari Raspberry.

```
series: focal
machines:
  '0':
    constraints: cores=4 mem=4G root-disk=16G tags=master
    series: focal
  '1':
    constraints: cores=4 mem=4G root-disk=16G
    series: focal
  '2':
    constraints: cores=4 mem=4G root-disk=16G
    series: focal
  '3':
    constraints: cores=4 mem=4G root-disk=16G
    series: focal
services:
  containerd:
    charm: cs:~containers/containerd-100
    resources: {}
  easyrsa:
    charm: cs:~containers/easyrsa-342
    num_units: 1
    resources:
      easyrsa: 5
    to:
    - lxd:0
  etcd:
```

```
    charm: cs:~containers/etcd-546
    num_units: 1
    options:
      channel: 3.4/stable
    resources:
      core: 0
      etcd: 3
      snapshot: 0
    to:
    - '0'
  flannel:
    charm: cs:~containers/flannel-513
    resources:
      flannel-amd64: 653
      flannel-arm64: 650
      flannel-s390x: 637
  kubernetes-master:
    charm: cs:~containers/kubernetes-master-926
    constraints: cores=4 mem=4G root-disk=16G
    expose: true
    num_units: 1
    options:
      channel: 1.20/stable
    resources:
      cdk-addons: /home/lnz/cdk-addons_1.20.0_arm64.snap
      core: 0
      kube-apiserver: 0
      kube-controller-manager: 0
      kube-proxy: 0
      kube-scheduler: 0
      kubectl: 0
    to:
    - '0'
  kubernetes-worker:
    charm: cs:~containers/kubernetes-worker-718
    constraints: cores=4 mem=4G root-disk=16G
    expose: true
    num_units: 3
    options:
      channel: 1.20/stable
    resources:
      cni-amd64: 690
      cni-arm64: 681
      cni-s390x: 693
      core: 0
      kube-proxy: 0
      kubectl: 0
      kubelet: 0
    to:
```

```
    - '1'
    - '2'
    - '3'
  ceph-mon:
    charm: cs:ceph-mon-51
    num_units: 3
    options:
      expected-osd-count: 3
      source: distro
    to:
    - lxd:1
    - lxd:2
    - lxd:3
  ceph-osd:
    charm: cs:ceph-osd-306
    num_units: 3
    options:
      osd-devices: /dev/sdb
      source: distro
    to:
    - '1'
    - '2'
    - '3'
relations:
- - kubernetes-master:kube-api-endpoint
  - kubernetes-worker:kube-api-endpoint
- - kubernetes-master:kube-control
  - kubernetes-worker:kube-control
- - kubernetes-master:certificates
  - easyrsa:client
- - kubernetes-master:etcd
  - etcd:db
- - kubernetes-worker:certificates
  - easyrsa:client
- - etcd:certificates
  - easyrsa:client
- - flannel:etcd
  - etcd:db
- - flannel:cni
  - kubernetes-master:cni
- - flannel:cni
  - kubernetes-worker:cni
- - containerd:containerd
  - kubernetes-worker:container-runtime
- - containerd:containerd
  - kubernetes-master:container-runtime
- - ceph-osd:mon
  - ceph-mon:osd
- - ceph-mon:admin
```

```
  - kubernetes-master:ceph-storage
- - ceph-mon:client
  - kubernetes-master:ceph-client
```

Come mostrato in Elenco 8.8, la risorsa cdk-addons è stata specificata manualmente, riferendosi ad un pacchetto snapcraft locato in un percorso locale. L'operazione si è resa necessaria per risolvere un problema legato ai pod avviati automaticamente dalla relazione tra il kubernetes-master e ceph-mon.

Sostituzione delle immagini relative ai container dei servizi Ceph

I vari pod, avviati dal kubernetes-master per interagire con Ceph, entravano in uno stato di errore, legato all'errata architettura per cui erano state realizzate le immagini dei relativi container. Gli oggetti relativi a tali pod, infatti, facevano riferimento ad immagini, ospitate su vari servizi di registry, compilate esclusivamente per l'architettura amd64. Questo non permetteva di avviare i relativi pod sui Raspberry, che possiedono un'architettura arm64.

I riferimenti alle immagini dei vari container sono stati quindi modificati, specificando delle immagini compilate per arm64. Le definizioni degli oggetti Kubernetes relativi a tali pod, sono contenuti nella risorsa cdk-addons [63] del charm kubernetes-master. Tale risorsa è un pacchetto snapcraft, che contiene al suo interno vari file YAML. Tali file descrivono gli oggetti Kubernetes da creare al momento dell'aggiunta della relazione tra il kubernetes-master e ceph-mon.

Tali file YAML sono stati quindi modificati, ricreando il pacchetto snapcraft, tramite le seguenti operazioni:

1. tramite il comando

   ```
   UBUNTU_STORE_ARCH=arm64
        snap download cdk-addons --channel=1.20/stable
   ```

   è stato scaricato il file relativo al pacchetto snapcraft, chiamato "cdk-addons_5076.snap". Definendo la variabile "UBUNTU_STORE_ARCH", è stato specificato di ottenere la versione del pacchetto realizzata per arm64;

2. tramite il comando

   ```
   unsquashfs cdk-addons_5076.snap
   ```

   è stato estratto il contenuto del pacchetto;

3. sono stati modificati i file estratti, specificando i nuovi riferimenti alle immagini dei container. In particolare, sono stati modificati i seguenti file:

   - templates/cephfs/csi-cephfsplugin.yaml

- templates/cephfs/csi-cephfsplugin-provisioner.yaml
- templates/cinder-csi-controllerplugin.yaml
- templates/cinder-csi-nodeplugin.yaml
- templates/csi-rbdplugin.yaml
- templates/csi-rbdplugin-provisioner.yaml
- templates/kube-state-metrics-deployment.yaml

All'interno di tali file sono stati modificati i riferimenti alle seguenti immagini:

- csi-node-driver-registrar
- cephcsi
- csi-provisioner
- csi-resizer
- csi-attacher
- csi-snapshotter
- kube-state-metrics

I nuovi riferimenti sono stati realizzati ricercando le rispettive immagini, compilate per arm64, su Docker Hub;

4. tramite il comando

```
snap pack .
```

eseguito all'interno della cartella contenente il file "snapcraft.yaml", è stato ricompilato il pacchetto snapcraft, nel file "cdk-addons_1.20.0_arm64.snap";

5. tale file è stato quindi specificato, nel modello Juju, come risorsa del kubernetes-master. Nel caso il deployment del modello fosse già avvenuto, è possibile modificare tale risorsa attraverso i seguenti comandi:

```
juju attach-resource kubernetes-master
    cdk-addons=./cdk-addons_1.20.0_arm64.snap

juju run-action kubernetes-master/0 upgrade
```

Deployment del modello

Il deployment del modello realizzato è avvenuto tramite le seguenti operazioni, eseguite dal MAAS controller, che possiede la CLI Juju:

1. tramite il comando

   ```
   juju add-model k8s-ceph
   ```

   è stato creato un nuovo modello Juju, chiamato "k8s-ceph";

2. tramite il comando

   ```
   juju deploy ./k8s-ceph.yaml
   ```

   è stato effettuato, sul modello precedentemente creato, il deployment del bundle contenuto nel file "k8s-ceph.yaml", corrispondente all'Elenco 8.8;

3. Juju ha quindi avviato, automaticamente, tramite MAAS, l'allocazione, l'installazione e la configurazione dei vari Raspberry del cluster. Lo stato dei vari Raspberry, relativamente a MAAS, al termine di queste operazioni, è mostrato in Figura 8.10;

4. tramite il comando

   ```
   juju status
   ```

   è stato possibile visualizzare lo stato di avanzamento delle operazioni di deployment e gli eventuali problemi. L'output di questo comando è mostrato in Figura 8.11;

5. al termine del deployment, tramite il comando

   ```
   juju scp kubernetes-master/0:config ~/.kube/config
   ```

   è stato possibile ottenere il file di configurazione Kubernetes, che specifica le credenziali di accesso per permettere, ad esempio, a kubectl o Terraform, di accedere al cluster Kubernetes;

6. tramite il comando

   ```
   juju add-unit kubernetes-worker
   ```

   è stato possibile effettuare un'operazione di scaling, aggiungendo al cluster Kubernetes un nuovo nodo worker, ospitato, ad esempio, da un quinto Raspberry. Anche in questo caso, le operazioni di installazione e configurazione del nuovo nodo sono state effettuate automaticamente da MAAS e Juju;

7. tramite il comando

   ```
   juju remove-unit kubernetes-worker/<unit id>
   ```

   è stato possibile rimuovere il nodo precedentemente aggiunto.

MAAS Hardware Images DNS AZs Subnets Settings lnz Log out

Machines 5 machines available Add hardware Take action

5 Machines 1 Resource pool

Filters Search Group by status

FQDN | MAC IP	POWER	STATUS	OWNER TAGS	ZONE SPACES	CORES ARCH	RAM	DISKS	STORAGE
Deployed 5 machines								
juju-controller.maas 192.168.10.9 (PXE)	Unknown Manual	Ubuntu 20.04 LTS	lnz virtual	default	2 amd64	4 GiB	1	136.4 GB
rpi0.maas 192.168.10.176 (PXE)	On Rpi	Ubuntu 20.04 LTS	lnz master	default	4 arm64	8 GiB	1	62.7 GB
rpi1.maas 192.168.10.180 (PXE)	On Rpi	Ubuntu 20.04 LTS	lnz	default	4 arm64	8 GiB	2	93.5 GB
rpi2.maas 192.168.10.179 (PXE)	On Rpi	Ubuntu 20.04 LTS	lnz	default	4 arm64	8 GiB	2	93.5 GB
rpi3.maas 192.168.10.181 (PXE)	On Rpi	Ubuntu 20.04 LTS	lnz	default	4 arm64	8 GiB	2	93.5 GB

MAAS name: maas MAAS
MAAS version: 2.9.0 (9137-g.8e920a12b)

CANONICAL

Local documentation · Legal information · Give feedback

© 2021 Canonical Ltd. Ubuntu and Canonical are registered trademarks of Canonical Ltd.

Figura 8.10: stato delle macchine gestite da MAAS al termine delle operazioni di deployment.

```
Model  Controller              Cloud/Region          Version  SLA          Timestamp
k8s    rpi-cluster-controller  rpi-cluster/default   2.9-rc2  unsupported  20:20:10Z

App                Version  Status  Scale  Charm              Store       Rev  OS      Message
ceph-mon           15.2.7   active      3  ceph-mon           charmstore   53  ubuntu  Unit is ready and clustered
ceph-osd           15.2.7   active      3  ceph-osd           charmstore  308  ubuntu  Unit is ready (1 OSD)
containerd         1.3.3    active      4  containerd         charmstore  100  ubuntu  Container runtime available
easyrsa            3.0.1    active      1  easyrsa            charmstore  342  ubuntu  Certificate Authority connected.
etcd               3.4.5    active      1  etcd               charmstore  546  ubuntu  Healthy with 1 known peer
flannel            0.11.0   active      4  flannel            charmstore  513  ubuntu  Flannel subnet 10.1.78.1/24
kubernetes-master  1.20.4   active      1  kubernetes-master  charmstore  926  ubuntu  Kubernetes master running.
kubernetes-worker  1.20.4   active      3  kubernetes-worker  local         0  ubuntu  Kubernetes worker running.

Unit                  Workload  Agent  Machine  Public address  Ports            Message
ceph-mon/0            active    idle   1/lxd/0  192.168.10.183                   Unit is ready and clustered
ceph-mon/1*           active    idle   2/lxd/0  192.168.10.184                   Unit is ready and clustered
ceph-mon/2            active    idle   3/lxd/0  192.168.10.185                   Unit is ready and clustered
ceph-osd/0*           active    idle   1        192.168.10.179                   Unit is ready (1 OSD)
ceph-osd/1            active    idle   2        192.168.10.180                   Unit is ready (1 OSD)
ceph-osd/2            active    idle   3        192.168.10.181                   Unit is ready (1 OSD)
easyrsa/0*            active    idle   0/lxd/0  192.168.10.182                   Certificate Authority connected.
etcd/0*               active    idle   0        192.168.10.176  2379/tcp         Healthy with 1 known peer
kubernetes-master/0*  active    idle   0        192.168.10.176  6443/tcp         Kubernetes master running.
  containerd/2        active    idle            192.168.10.176                   Container runtime available
  flannel/2           active    idle            192.168.10.176                   Flannel subnet 10.1.57.1/24
kubernetes-worker/0*  active    idle   1        192.168.10.179  80/tcp,443/tcp   Kubernetes worker running.
  containerd/0*       active    idle            192.168.10.179                   Container runtime available
  flannel/0*          active    idle            192.168.10.179                   Flannel subnet 10.1.78.1/24
kubernetes-worker/1   active    idle   2        192.168.10.180  80/tcp,443/tcp   Kubernetes worker running.
  containerd/1        active    idle            192.168.10.180                   Container runtime available
  flannel/1           active    idle            192.168.10.180                   Flannel subnet 10.1.73.1/24
kubernetes-worker/2   active    idle   3        192.168.10.181  80/tcp,443/tcp   Kubernetes worker running.
  containerd/3        active    idle            192.168.10.181                   Container runtime available
  flannel/3           active    idle            192.168.10.181                   Flannel subnet 10.1.70.1/24

Machine  State    DNS             Inst id              Series  AZ       Message
0        started  192.168.10.176  rpi0                 focal   default  Deployed
0/lxd/0  started  192.168.10.182  juju-04d390-0-lxd-0  focal   default  Container started
1        started  192.168.10.179  rpi2                 focal   default  Deployed
1/lxd/0  started  192.168.10.183  juju-04d390-1-lxd-0  focal   default  Container started
2        started  192.168.10.180  rpi1                 focal   default  Deployed
2/lxd/0  started  192.168.10.184  juju-04d390-2-lxd-0  focal   default  Container started
3        started  192.168.10.181  rpi3                 focal   default  Deployed
3/lxd/0  started  192.168.10.185  juju-04d390-3-lxd-0  focal   default  Container started
```

Figura 8.11: output del comando "juju status", che mostra lo stato del modello Juju realizzato al termine delle operazioni di deployment.

8.6 Estensione del Kubernetes cluster autoscaler

L'estensione del Kubernetes cluster autoscaler è avvenuta, anch'essa, senza particolari problemi. La difficoltà maggiore è stata riscontrata nella compilazione del file ".proto", in modo tale che il codice generato fosse compatibile con la specifica versione della libreria gRPC importata dal progetto dell'autoscaler.

8.6.1 Definizione e compilazione della specifica Protobuf

Per permettere la chiamata a procedura remota, tramite gRPC, tra il cluster autoscaler ed il Juju Python client, è stato definito un file ".proto", contenente la dichiarazione delle strutture dati e dei servizi necessari. Le strutture dati ed i metodi rpc definiti corrispondono ai metodi principali delle interfacce CloudProvider e NodeGroup del cluster autoscaler. I metodi opzionali, poiché non si sono rivelati necessari, non sono stati definiti. Il file ".proto" realizzato, basato sulla relativa proposta della community [40], è mostrato in Elenco 8.9.

Elenco 8.9: file ".proto" che definisce le strutture dati ed i servizi gRPC necessari per l'invocazione remota dei metodi relativi al cluster autoscaler sul Juju Python client. Codice basato sulla relativa proposta della community: [41].

```
syntax = "proto3";

package clusterautoscaler.cloudprovider.v1;

import "google/protobuf/descriptor.proto";

option go_package = "juju-grpc-cloudprovider";

message Node {
    string id = 1;
}

message NodeGroup {
    string id = 1;
    int32 minSize = 2;
    int32 maxSize = 3;
    string debug = 4;
}

message NameRequest {
}

message NameResponse {
    string name = 1;
}

message NodeGroupsRequest {
}

message NodeGroupsResponse {
    repeated NodeGroup nodeGroups = 1;
}

message NodeGroupForNodeRequest {
    Node node = 1;
}

message NodeGroupForNodeResponse {
    repeated NodeGroup nodeGroup = 1;
}



message NodeGroupMaxSizeRequest {
    string id = 1;
```

```
}

message NodeGroupMaxSizeResponse {
    int32 maxSize = 1;
}

message NodeGroupMinSizeRequest {
    string id = 1;
}

message NodeGroupMinSizeResponse {
    int32 minSize = 1;
}

message NodeGroupTargetSizeRequest {
    string id = 1;
}

message NodeGroupTargetSizeResponse {
    int32 targetSize = 1;
}

message NodeGroupIncreaseSizeRequest {
    int32 delta = 1;
    string id = 2;
}

message NodeGroupIncreaseSizeResponse {
}

message NodeGroupDeleteNodesRequest {
    repeated Node nodes = 1;
    string id = 2;
}

message NodeGroupDeleteNodesResponse {
}

message NodeGroupDecreaseTargetSizeRequest {
    int32 delta = 1;
    string id = 2;
}

message NodeGroupDecreaseTargetSizeResponse {
}

message NodeGroupNodesRequest {
    string id = 1;
}
```

```
message NodeGroupNodesResponse {
    repeated Instance instances = 1;
}

message Instance {
    string id = 1;
    InstanceStatus status = 2;
}

message InstanceStatus {
    enum InstanceState {
        InstanceRunning = 0;
        InstanceCreating = 1;
        InstanceDeleting = 2;
    }
    InstanceState status = 1;
    InstanceErrorInfo errorInfo = 2;
}

message InstanceErrorInfo {
    string errorCode = 1;
    string errorMessage = 2;
    int32 instanceErrorClass = 3;
}

service CloudProvider {
  // CloudProvider specific RPC functions
  rpc Name(NameRequest)
    returns (NameResponse) {}

  rpc NodeGroups(NodeGroupsRequest)
    returns (NodeGroupsResponse) {}

  rpc NodeGroupForNode(NodeGroupForNodeRequest)
    returns (NodeGroupForNodeResponse) {}

  // NodeGroup specific RPC functions
  rpc NodeGroupMaxSize(NodeGroupMaxSizeRequest)
  returns (NodeGroupMaxSizeResponse) {}

  rpc NodeGroupMinSize(NodeGroupMinSizeRequest)
  returns (NodeGroupMinSizeResponse) {}

  rpc NodeGroupTargetSize(NodeGroupTargetSizeRequest)
    returns (NodeGroupTargetSizeResponse) {}

  rpc NodeGroupIncreaseSize(NodeGroupIncreaseSizeRequest)
    returns (NodeGroupIncreaseSizeResponse) {}
```

```
  rpc NodeGroupDeleteNodes(NodeGroupDeleteNodesRequest)
    returns (NodeGroupDeleteNodesResponse) {}

  rpc NodeGroupDecreaseTargetSize(NodeGroupDecreaseTargetSizeRequest)
    returns (NodeGroupDecreaseTargetSizeResponse) {}

  rpc NodeGroupNodes(NodeGroupNodesRequest)
    returns (NodeGroupNodesResponse) {}
}
```

Alcune importanti caratteristiche relative alle strutture dati ed ai metodi definiti sono le seguenti:

- i nodi vengono identificati esclusivamente dal loro nome, che corrisponde al nome della macchina su MAAS;
- le strutture dati di input, relative ai metodi dell'interfaccia NodeGroup, contengono un campo id che specifica su quale NodeGroup deve essere chiamato il metodo. In questo caso esisterà un solo NodeGroup, con id "rpis";
- il metodo NodeGroupNodes restituisce la lista dei nodi appartenenti ad uno specifico NodeGroup. Ogni nodo è identificato da un id, che, in questo caso, corrisponde al nome della macchina su MAAS. Tale id deve però anche corrispondere alla proprietà providerID dei corrispettivi oggetti di tipo "Node" relativi alle API Kubernetes.

La compilazione di tale file ".proto", per generare il codice sorgente in Go e Python, è avvenuta, su una macchina Ubuntu, tramite le seguenti operazioni:

1. tramite il comando
   ```
   sudo snap install go --classic
   ```
   è stato installato il compilatore del linguaggio Go;
2. è stata creata la cartella relativa al GOPATH, nel percorso "~/go";
3. tramite il comando
   ```
   export GOPATH=~/go
   ```
   è stato impostato il GOPATH a tale cartella;
4. la versione 1.20 del Kubernetes cluster autoscaler utilizza una libreria gRPC Go incompatibile con il codice generato dall'ultima versione del compilatore grpc-go. Nello specifico, le nuove versioni del compilatore generano codice con la costante SupportPackageIsVersion7; la libreria gRPC dell'autoscaler, invece,

può riconoscere solamente la costante SupportPackageIsVersion6. Per questo, è stato necessario installare manualmente una versione meno recente del compilatore grpc-go, cioè la versione 1.31.1 [49]. Il contenuto della cartella "grpc-go-1.31.1", contenuta nell'archivio relativo a tale release, è stato copiato nel percorso "$GOPATH/src/google.golang.org/grpc";

5. il comando

   ```
   go mod tidy
   ```

   è stato eseguito all'interno delle cartelle "$GOPATH/src/google.golang.org/grpc" e "$GOPATH/src/google.golang.org/grpc/cmd/protoc-gen-go-grpc", per scaricare le relative dipendenze Go;

6. tramite i comandi

   ```
   go install google.golang.org/grpc/cmd/protoc-gen-go-grpc/...
   go install google.golang.org/protobuf/cmd/protoc-gen-go
   ```

   sono stati compilati gli eseguibili protoc-gen-go-grpc e protoc-gen-go, nella cartella "$GOPATH/bin";

7. tramite il comando

   ```
   export PATH="$GOPATH/bin:$PATH"
   ```

   è stata aggiunta la cartella con gli eseguibili compilati al PATH;

8. è stato installato il compilatore protoc 3.14.0 [102], copiando le cartelle bin ed include, presenti nell'archivio della release, nei percorsi, rispettivamente, "/usr/bin" e "/usr/include";

9. è stato installato Python e pip; tramite il comando

   ```
   python -m pip install grpcio-tools
   ```

   è stato installato il compilatore per la generazione di codice Python;

10. tramite il comando

    ```
    protoc -I='.' --go_out=. --go-grpc_out=.
      juju-grpc-cloudprovider.proto
    ```

    è stato generato il codice Go relativo al file "juju-grpc-cloudprovider.proto", contenente ciò che è stato mostrato in Elenco 8.9;

11. tramite il comando

    ```
    python -m grpc_tools.protoc -I='.' --python_out=.
      --grpc_python_out=. juju-grpc-cloudprovider.proto
    ```

    è stato generato, analogamente, il relativo codice Python.

8.6.2 Estensione e compilazione del cluster autoscaler

L'estensione del Kubernetes cluster autoscaler si è basata sulla versione 1.20, disponibile nel branch "cluster-autoscaler-release-1.20" del relativo repository [60]. Sono state eseguite le seguenti operazioni:

1. all'interno della cartella "cluster-autoscaler/cloudprovider", è stata creata una nuova cartella, chiamata grpc;

2. all'interno di tale cartella, sono stati copiati i file sorgente, derivati dal file ".proto", precedentemente generati;

3. sono state realizzate le estensioni delle interfacce CloudProvider e NodeGroup. Tali implementazioni si limitano a connettersi al server gRPC e ad inoltrare le relative chiamate a metodo. Ciò viene effettuato attraverso la libreria gRPC, già presente all'interno del cluster autoscaler, e dai file sorgente precedentemente generati;

4. sono stati modificati i file "cluster-autoscaler/cloudprovider/builder/builder_all.go" e "cluster-autoscaler/cloudprovider/cloud_provider.go", aggiungendo la dichiarazione del cloud provider realizzato;

5. le dipendenze del progetto vengono gestite manualmente tramite lo script "cluster-autoscaler/hack/update-vendor.sh". Tale script deve essere eseguito prima di avviare l'operazione di build, specificando, tramite l'argomento -r, l'hash del commit relativo al repository di Kubernetes dal quale verranno importate le dipendenze [27]. Nel caso della versione 1.20 del cluster autoscaler deve essere quindi eseguito, prima di effettuare la compilazione, il seguente comando:

   ```
   ./hack/update-vendor.sh
       -re3de62298a730415c5d2ab72607ef6adadd6304d
   ```

6. per effettuare la compilazione è stato necessario copiare la cartella "autoscaler", cioè l'intero repository nel quale si trova la cartella cluster-autoscaler, nel percorso "$GOPATH/src/k8s.io"; successivamente è stato possibile compilare il progetto tramite il comando

   ```
   make build
   ```

   eseguito dalla cartella "$GOPATH/src/k8s.io/autoscaler/cluster-autoscaler";

7. installando Docker ed eseguendo il comando

   ```
   make make-image
   ```

dalla medesima cartella, è stato possibile costruire l'immagine del container, contenente il cluster autoscaler con le estensioni realizzate. Per realizzare un'immagine per arm64, senza utilizzare emulatori, è stato necessario eseguire le operazioni di compilazione e realizzazione dell'immagine su uno dei Raspberry. Tale immagine è stata quindi caricata su Docker Hub, per permettere la successiva realizzazione, tramite Terraform, del relativo pod Kubernetes.

8.6.3 Realizzazione del Juju Python Client

Il componente Juju Python Client utilizza il codice gRPC Python precedentemente generato per rispondere alle richieste del cluster autoscaler. L'implementazione dei metodi gRPC, che comprendono l'interazione con il Juju controller, è avvenuta sfruttando la libreria python-libjuju. Tale libreria, per interagire con il Juju controller, necessita delle credenziali di accesso relative ad un utente Juju, che abbia i permessi di lettura e scrittura sul modello precedentemente realizzato. Per questo è stato necessario creare un nuovo utente Juju, tramite le seguenti operazioni:

1. tramite i comandi

   ```
   juju add-user autoscaler
   juju change-user-password autoscaler
   ```

   è stato creato l'utente autoscaler ed è stata definita la relativa password per l'autenticazione;

2. tramite il comando

   ```
   juju grant autoscaler write k8s-ceph
   ```

   è stato fornito all'utente autoscaler il permesso di scrittura sul modello k8s-ceph;

3. tramite il comando

   ```
   juju show-controller
   ```

   sono state reperite le ulteriori informazioni necessarie a python-libjuju per accedere al controller. In particolare, l'API endpoint, l'UUID del modello k8s-ceph ed il certificato relativo alla certificate authority.

Python-libjuju, tramite l'API endpoint, l'UUID del modello, il certificato relativo alla certificate authority, il nome utente "autoscaler" e la relativa password, potrà accedere al controller Juju ed effettuare eventuali modifiche sul modello k8s-ceph, ad esempio, aggiungendo o rimuovendo unità kubernetes-worker.

In particolare, i metodi gRPC relativi alle interfacce del cluster autoscaler, cioè CloudProvider e NodeGroup, sono stati implementati, attraverso python-libjuju, come segue:

NodeGroups Restituisce una lista composta da un singolo NodeGroup, identificato dall'id "rpis". Il cloud provider è stato infatti realizzato per gestire, esclusivamente, un unico cluster di Raspberry identici tra loro.

NodeGroupForNode Reperisce, dal modello Juju, i nomi delle macchine sulle quali è in esecuzione un'unità kubernetes-worker. Nel caso l'id del nodo fornito in input corrisponda ad uno di tali nomi, restituisce il NodeGroup "rpis", altrimenti una lista vuota.

NodeGroupMaxSize Specificando il NodeGroup "rpis", restituisce un valore costante. Nel caso specifico, tale valore corrisponderà a 4. Significa che i kubernetes-worker potranno essere estesi fino a quattro unità.

NodeGroupMinSize Specificando il NodeGroup "rpis", restituisce un valore costante. Nel caso specifico, tale valore corrisponderà a 3. Infatti, il numero di kubernetes-worker non potrà scendere sotto le 3 unità, poiché i tre nodi che ospitano anche i servizi relativi a Ceph non dovranno essere eliminati.

NodeGroupTargetSize Specificando il NodeGroup "rpis", restituisce il numero di macchine sulle quali è in esecuzione, o verrà messa in esecuzione, un'unità kubernetes-worker; comprende, inoltre, anche le macchine sulle quali era in esecuzione un'unità kubernetes-worker, ma che sono, attualmente, in fase di eliminazione. Tale metodo considera, infatti, anche i nodi le cui operazioni di deployment o di rimozione non sono ancora terminate.

NodeGroupIncreaseSize Specificando il NodeGroup "rpis", aggiunge al modello Juju una o più unità kubernetes-worker, in base al delta, positivo, specificato. Attende, quindi, l'associazione delle relative macchine fisiche alle unità aggiunte e, successivamente, attende il termine delle operazioni di deployment. Durante il deployment, le macchine vengono conteggiate nel NodeGroupTargetSize; inoltre, sono temporaneamente aggiunte ad una lista, per tenere traccia del loro stato, necessario per implementare il metodo NodeGroupNodes.

NodeGroupDeleteNodes Specificando il NodeGroup "rpis", rimuove del modello Juju tutte le macchine con un nome corrispondente agli id specificati nella lista di input. Attende, quindi, che le macchine siano effettivamente rimosse. Durante la rimozione, le macchine vengono conteggiate nel NodeGroupTargetSize; inoltre, sono temporaneamente aggiunte ad una lista, per tenere traccia del loro stato, necessario per implementare il metodo NodeGroupNodes.

NodeGroupDecreaseTargetSize Specificando il NodeGroup "rpis", rimuove del modello Juju una o più macchine in fase di deployment, in base al delta, negativo, specificato. Successivamente all'individuazione degli id delle

macchine da rimuovere, invoca il metodo NodeGroupDeleteNodes per effettuare la rimozione. Per individuare le macchine che possono essere rimosse, utilizza la lista popolata dal metodo NodeGroupIncreaseSize, che tiene traccia delle macchine in fase di deployment.

NodeGroupNodes Specificando il NodeGroup "rpis", restituisce la lista delle macchine relative alle unità kubernetes-worker. Ogni macchina è identificata da un id, che corrisponde al nome della macchina. Inoltre, ogni macchina specifica uno stato, che indica se la macchina è attiva, in fase di deployment, di eliminazione o in stato di errore. Tali informazioni vengono ottenute sia dallo stato del modello Juju, sia dalle liste di stato gestite dai metodi NodeGroupIncreaseSize e NodeGroupDeleteNodes.

8.6.4 Definizione del providerID nei kubernetes-worker

Il cluster autoscaler, per determinare lo stato dei nodi Kubernetes, sfrutta le informazioni restituite dal metodo "Nodes" relativo ai NodeGroup del rispettivo cloud provider. Tali informazioni devono essere associate agli oggetti relativi ai nodi presenti nelle API Kubernetes. L'associazione viene effettuata attraverso il providerID: il cluster autoscaler suppone che l'id delle macchine restituite dal metodo "Nodes" corrisponda alla proprietà providerID, specificata all'interno dei rispettivi oggetti "Node" di Kubernetes.

Sfortunatamente, Juju, al momento della registrazione dei nodi kubernetes-worker sulle API Kubernetes, non definisce tale proprietà. Per questo, è stato necessario modificare il charm kubernetes-worker, in modo da impostare tale proprietà. Sono state effettuate le seguenti operazioni:

1. tramite il comando

   ```
   sudo snap install charm --classic
   ```

   è stato installato lo strumento per la gestione dei charm Juju;

2. tramite il comando

   ```
   charm pull cs:~containers/kubernetes-worker-718
        ~/kubernetes-worker-718
   ```

   è stato reperito il codice relativo alla versione 718 del charm kubernetes-worker, salvandolo nella home directory;

3. il file "kubernetes-worker-718/reactive/kubernetes_worker.py" [65], è stato modificato, aggiungendo, alla riga 808, il seguente codice:

   ```
   else:
        kubelet_opts['provider-id'] = get_node_name()
   ```

Tale codice imposta il nome del nodo, che corrisponde al nome della macchina gestita da MAAS, alla proprietà providerID del relativo oggetto "Node" di Kubernetes. In questo modo, poiché anche il metodo "Nodes" implementato identifica le macchine tramite il loro nome, sarà possibile, per il cluster autoscaler, associare le macchine del cloud provider ai relativi oggetti "Node" di Kubernetes;

4. il charm modificato è stato quindi aggiunto al modello Juju, tramite il comando:

```
juju upgrade-charm kubernetes-worker
    --path ~/kubernetes-worker-718
```

5. a causa dell'adozione di un charm localizzato in un percorso locale, è stato necessario scaricare localmente anche le relative risorse di tale charm, in particolare, di "cni-arm64"; le altre risorse snapcraft, se non specificate, saranno reperite automaticamente dalle sorgenti pubbliche. L'archivio ".tgz" relativo alla risorsa "cni-arm64" è stato quindi reperito dal Charm Store [64], scaricandolo nella home directory e rinominandolo "681.tgz"; tale risorsa, successivamente, è stata assegnata al charm kubernetes-worker tramite il seguente comando:

```
juju attach-resource kubernetes-worker cni-arm64=~/681.tgz
```

8.6.5 Configurazione dei nodi Kubernetes

I tre nodi del cluster relativi ai kubernetes-worker ospitano anche le unità relative a Ceph. Per questo, il cluster autoscaler non dovrà, nel caso di un'operazione di scale in, eliminare tali nodi. Per risolvere tale problema è stato necessario, attraverso il comando

```
kubectl annotate node <nome nodo>
    --overwrite cluster-autoscaler.kubernetes.io/safe-to-evict=false
```

annotare tali nodi, tramite kubectl, con l'annotazione "safe-to-evict=false". Il cluster autoscaler, come specificato nella relativa documentazione [27], non rimuoverà i nodi con tale caratteristica, anche se essi risultassero sottoutilizzati.

8.7 Terraform

Come descritto in fase di progettazione, il deployment di alcune applicazioni sul cluster Kubernetes è stato effettuato attraverso Terraform.

8.7.1 Installazione e configurazione di Terraform

L'installazione di Terraform risulta notevolmente semplice: è sufficiente ottenere, sia su sistemi operativi Windows che su Linux, il file eseguibile di Terraform, aggiungendolo al PATH.

Successivamente, sarà possibile utilizzare i comandi init, plan, apply, o destroy, su un determinato file di configurazione. Per la connessione al cluster Kubernetes realizzato è stato utilizzato il provider kubernetes, fornendogli il file di configurazione contenente le credenziali di accesso. Tale file è stato ottenuto tramite il comando

```
juju scp kubernetes-master/0:config ~/.kube/config
```

8.7.2 Deployment di MetalLB

Il deployment di MetalLB è avvenuto, attraverso Terraform, sfruttando un apposito modulo offerto dalla community [126]. Il file di configurazione Terraform realizzato è mostrato in Elenco 8.10.

Elenco 8.10: file di configurazione Terraform utilizzato per deployment di MetalLB, tramite il relativo modulo offerto dalla community [126].

```
provider "kubernetes" {
  version = "1.13.3"
}

module "metallb" {
  source  = "colinwilson/metallb/kubernetes"
  version = "0.1.2"
}

resource "kubernetes_config_map" "metallb_config_map" {
  metadata {
    name = "config"
    namespace = "metallb-system"
  }

  data = {
    config = <<EOF
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 192.168.10.151-192.168.10.160
    EOF
  }
}
```

In particolare, oltre alla dichiarazione del modulo relativo a MetalLB, è stata definita una ConfigMap Kubernetes, contenente le configurazioni necessarie a MetalLB per operare. Tali configurazioni impostano MetalLB in modo che utilizzi la modalità Layer2, sfruttando, per la creazione degli eventuali load balancer, il range di indirizzi "192.168.10.151-192.168.10.160" riservato da MAAS.

8.7.3 Deployment del cluster autoscaler

Il deployment del cluster autoscaler è avvenuto, tramite Terraform, definendo un Deployment Kubernetes. Esso prevede la creazione di un pod, che eseguirà l'immagine del cluster autoscaler precedentemente creata. Tale pod necessita, però, degli adeguati permessi per accedere a determinate risorse esposte dalle API Kubernetes: ad esempio, deve poter accedere ai nodi del cluster ed allo stato degli altri pod, per individuare quelli non schedulabili. Questi permessi sono stati definiti attraverso un insieme di Role e ClusterRole, associati, attraverso RoleBinding e ClusterRoleBinding ad un determinato ServiceAccount, chiamato "cluster-autoscaler".

In generale, tali permessi sono comuni a tutti i cloud provider. Per questo, la loro definizione si è basata sui file YAML utilizzati per il deployment del cluster autoscaler su altri cloud provider, come AWS. Nello specifico, sono state applicate, tramite kubectl, le definizioni, da riga 1 a 118, del file YAML di esempio relativo ad AWS [26]. Tale porzione ha permesso di creare i vari ruoli e di associarli al ServiceAccount "cluster-autoscaler", che dovrà essere utilizzato dal relativo pod per poter operare. Sebbene tali risorse potessero essere applicate attraverso Terraform, considerata la loro staticità e gli sforzi che sarebbero stati necessari per convertirle in linguaggio HCL, si è scelto di applicarle tramite kubectl.

L'oggetto relativo al Deployment Kubernetes, basato anch'esso sul file YAML precedentemente enunciato, è stato invece applicato attraverso un file di configurazione Terraform, mostrato in Elenco 8.11.

Elenco 8.11: file di configurazione Terraform utilizzato per il deployment del cluster autoscaler.

```
provider "kubernetes" {
  version = "1.13.3"
}

resource "kubernetes_deployment" "depca" {
  metadata {
    name = "cluster-autoscaler"
    namespace= "kube-system"
    labels = {
      app = "cluster-autoscaler"
    }
  }
```

```
  spec {
    replicas = "1"
    selector {
      match_labels = {
        app = "cluster-autoscaler"
      }
    }
    template {
      metadata {
        labels = {
          app = "cluster-autoscaler"
        }
        annotations = {
          "prometheus.io/scrape" = "true"
          "prometheus.io/port" = "8085"
        }
      }
      spec {
        service_account_name = "cluster-autoscaler"
        automount_service_account_token = true
        container {
          image = "lorenzomondani/cluster-autoscaler"
          name = "cluster-autoscaler"
          image_pull_policy = "Always"
          resources {
            limits {
              cpu = "100m"
              memory = "300Mi"
            }
            requests {
              cpu = "100m"
              memory = "300Mi"
            }
          }
          command = [
            "./cluster-autoscaler",
            "--v=4",
            "--stderrthreshold=info",
            "--cloud-provider=grpc",
            "--skip-nodes-with-local-storage=false",
            "--max-node-provision-time=120m",
            "--leader-elect=false"
          ]
        }
      }
    }
  }
}
```

Le caratteristiche principali del Deployment Kubernetes realizzato sono le seguenti:

1. il pod relativo al deployment utilizza il ServiceAccount "cluster-autoscaler", per ottenere tutti i permessi necessari;

2. è stato necessario definire la proprietà "automount_service_account_token = true", in modo che il pod avesse accesso al token relativo al ServiceAccount. Terraform, contrariamente agli YAML kubectl, imposta tale proprietà, come valore di default, a falso;

3. il pod utilizza l'immagine relativa al cluster autoscaler con l'estensione precedentemente realizzata, compilata per arm64 e caricata su Docker Hub;

4. il comando di avvio dell'eseguibile relativo al cluster autoscaler specifica vari argomenti:

   –cloud-provider=grpc Impone di utilizzare il cloud provider grpc, realizzato precedentemente.

   –max-node-provision-time=120m Il cluster autoscaler, come impostazione di default, successivamente ad un'azione di scale out, attende l'apparizione del nuovo nodo per 15 minuti. L'aggiunta di un nuovo nodo, a causa dell'hardware e delle tecnologie utilizzate, potrebbe però richiedere molto più tempo. Per questo, è stato impostato il tempo massimo di 120 minuti. Infatti, l'installazione e la configurazione di un nuovo Raspberry, solitamente, richiedono circa 60 minuti, a causa della ridotta velocità delle pendrive utilizzate.

   –leader-elect=false Poiché il deployment prevede un solo pod, sono state disabilitate le funzionalità relative all'elezione di un leader.

   Ulteriori dettagli relativi agli argomenti specificabili sono disponibili nelle FAQ del cluster autoscaler [27].

Capitolo 9

Verifica e valutazione dei risultati

In questo capitolo sarà descritto l'insieme delle operazioni di verifica che sono state condotte per determinare il grado di raggiungimento degli obiettivi.

I risultati ottenuti saranno quindi valutati, evidenziando gli eventuali problemi riscontrati.

Infine, saranno individuati alcuni dei possibili sviluppi futuri, legati alla risoluzione di tali problemi, al miglioramento delle funzionalità realizzate o alle tematiche non affrontate.

9.1 Realizzazione fisica del cluster

Le macchine scelte per la realizzazione del cluster, cioè i Raspberry Pi 4 B, si sono rivelate pienamente adatte alle particolari esigenze di questo progetto, grazie allo loro accessibilità, dal punto di vista dei costi, e all'esteso supporto software offerto dalla comunità. Attraverso l'adozione e la modifica del firmware UEFI è stato infatti possibile renderli compatibili con MAAS e, in generale, con tutto il software utilizzato.

La potenza computazionale offerta dalla CPU di tali SBC è risultata sufficiente per l'esecuzione del sistema operativo Ubuntu e dei vari componenti Ceph e Kubernetes. Probabilmente, la variante scelta di tali SBC, cioè quella che prevede 8 GB di memoria RAM, non è stata, invece, quella più adatta. Considerando infatti le risorse offerte dalla CPU, risulta improbabile saturare 8 GB di memoria RAM, avviando un gran numero di pod ma continuando a garantire prestazioni elevate. Per questo, sarebbe stato forse più vantaggioso contenere i costi dei singoli nodi, adottando la variante da 4 GB.

Dal punto di vista dello storage, le pendrive adottate sono risultate adatte allo scopo, offrendo costi ridotti ma prestazioni accettabili. Il punto debole di tali dispositivi sono proprio le prestazioni: perché sebbene sia possibile eseguire tutto il software relativo ad Ubuntu e Kubernetes, è richiesto un tempo notevolmente elevato per le operazioni di installazione e configurazione, che può superare anche i 60 minuti. Gran parte di

tale periodo è necessario per le operazioni di partizionamento, da parte di MAAS, dei vari dischi. Sebbene le prestazioni offerte dalla CPU dei Raspberry possano anch'esse influire sulle tempistiche di installazione, probabilmente, l'adozione di dischi più veloci, come dischi SSD, avrebbe garantito un notevole vantaggio da questo punto di vista.

9.2 Bare-metal cloud tramite MAAS

Tramite l'adozione del software MAAS e la realizzazione del BMC capace di controllare il power management dei vari Raspberry, è stato possibile realizzare un bare-metal cloud, che permette di installare e configurare le vari macchine, automaticamente, su richiesta.

Tramite il firmware UEFI è stato possibile rendere i vari Raspberry compatibili con l'ambiente di boot PXE e con le immagini Ubuntu generiche offerte da MAAS. I Raspberry, al momento dell'accensione, reperiscono, tramite il server DHCP e TFTP di MAAS, il firmware UEFI e, successivamente, i file necessari all'avvio del sistema operativo Ubuntu. In seguito alle numerose prove di boot dei vari Raspberry effettuate tramite tale modalità, non sono state individuate particolari criticità. Molto raramente, si sono verificate delle situazioni di blocco durante il processo di boot, sia al momento del reperimento dei file da parte di GRUB, sia al momento del boot del kernel di Ubuntu; in qualche caso, infatti, MAAS forniva al Raspberry una versione del kernel inferiore alla 5.8, che non possedeva i driver della scheda di rete. Non è chiaro quali siano state le cause di tali eventi, ma si presume siano legati a malfunzionamenti di MAAS o del firmware UEFI. In ogni caso, poiché tali errori si sono presentati molto raramente, non hanno destato forte preoccupazione. Sicuramente, però, nel caso si avesse la necessità di realizzare un sistema affidabile, a bassa probabilità di errore, sarebbe necessario risolvere tali problematiche. In tal caso, attualmente, considerato lo stato ancora sperimentale del firmware UEFI, si consiglierebbe di utilizzare un hardware differente, ad esempio di categoria server, o di gestire i vari Raspberry manualmente, senza MAAS, tramite le ordinarie procedure di boot.

Sfruttando il boot da rete, sia per fornire i file relativi al firmware UEFI, sia, per fornire i file relativi al sistema operativo, è stato possibile ridurre al minimo gli interventi manuali sulle varie macchine. Le uniche operazioni da compiere manualmente sono consistite nella connessione dell'alimentazione, delle interfacce di rete, del BMC e della riconfigurazione della EEPROM.

Il BMC realizzato ha permesso di gestire, da parte di MAAS, le operazioni di accensione e spegnimento dei vari Raspberry. Anche in questo caso, in seguito a varie prove effettuate, non sono state individuate particolari criticità. Il BMC è stato infatti in grado di accendere, spegnere e controllare lo stato dei vari Raspberry, tramite la loro interfaccia GPIO. Devono essere però rispettati alcuni vincoli: il Raspberry, quando è spento, deve essere nella modalità a risparmio energetico, in cui entra successivamente

ad uno shutdown del sistema; il sistema operativo utilizzato, per poter inviare la notifica di spegnimento, deve supportare ACPI ed il device ACPI GED. Nel caso il sistema operativo non supportasse tale device ACPI, o entrasse in uno stato di errore tale da comportare il blocco dell'intero sistema operativo, come un kernel panic, non sarà possibile spegnerlo. In ogni caso, Ubuntu supporta tale device ACPI, permettendo al BMC di spegnere la macchina.

Le API del BMC implementate non offrono alcun sistema di sicurezza; si assume infatti che tale dispositivo sia connesso ad una rete ad accesso limitato. Sarebbe comunque possibile estendere l'implementazione per definire una logica di controllo degli accessi.

Il microcontrollore scelto, l'ESP32, è risultato adatto allo scopo, grazie all'elevata quantità di pin GPIO che offre ed all'interfaccia Wi-Fi, tramite la quale può esporre, a MAAS, le API HTTP. Il punto debole di tale soluzione riguarda la potenziale inaffidabilità dell'interfaccia Wi-Fi, che potrebbe non riuscire a garantire un canale di comunicazione sufficientemente stabile.

9.3 Deployment e scaling tramite Juju

Juju, sfruttando i Raspberry offerti da MAAS, ha permesso il deployment automatico dei componenti relativi a Ceph e Kubernetes. Al momento del deployment del modello Juju, MAAS alloca automaticamente le macchine, le accende tramite il BMC ed avvia il processo di boot tramite DHCP e TFTP.

Il processo di deployment del cluster Kubernetes e Ceph sui vari Raspberry è risultato quindi totalmente automatico, senza comportare particolari problemi. Infatti, i vari charm Juju utilizzati sono risultati compatibili con l'architettura Arm, e, in generale, con l'hardware dei Raspberry.

Tramite Juju e MAAS è stato quindi possibile realizzare un servizio cloud riconducibile al modello Cluster as a Service, che permette cioè di allocare, su richiesta, un intero cluster preconfigurato con Kubernetes.

Juju, tramite i comandi

```
juju add-unit kubernetes-worker
juju remove-unit kubernetes-worker/<unit id>
```

ha permesso, inoltre, di effettuare operazioni di scaling del cluster Kubernetes. Le operazioni di aggiunta e rimozione di un nodo sono anch'esse gestite automaticamente, sfruttando MAAS, che, a sua volta, utilizza il BMC e l'ambiente di boot da rete per controllare le relative macchine. Anche in questo caso non sono sorti particolari problemi, a parte l'ingente quantità di tempo richiesto per effettuare le operazioni di installazione e configurazione di una nuova macchina.

Il cluster Kubernetes realizzato è risultato perfettamente funzionante: è stato possibile, tramite Terraform, effettuare il deployment di alcune applicazioni, senza riscontrare particolari problemi.

9.4 Load balancing tramite MetalLB

La verifica del funzionamento di MetalLB è avvenuta definendo un servizio Kubernetes di tipo LoadBalancer. Tale servizio crea un load balancer che espone, sulla porta 80, i server HTTP esposti sulla porta 8080 di un insieme di pod, gestiti da un ReplicationController. Il container eseguito da tali pod espone una semplice pagina HTML, che fornisce il nome del pod sul quale è in esecuzione. L'idea di utilizzare tali pod è venuta dal libro "Kubernetes in Action" [68, pag. 28]. Nel caso il LoadBalancer funzionasse correttamente, sarebbe possibile, connettendosi alla porta 80 del suo indirizzo, ottenere una pagina web indicante il nome del pod che ha fornito tale pagina; effettuando un'altra richiesta, tale nome dovrebbe cambiare.

Tali risorse sono state applicate al cluster tramite il file di configurazione Terraform mostrato in Elenco 9.1.

Elenco 9.1: file di configurazione Terraform che definisce un ReplicationController i cui pod sono esposti tramite un servizio Kubernetes di tipo LoadBalancer.

```
provider "kubernetes" {
  version = "1.13.3"
}

resource "kubernetes_namespace" "kubias-ns" {
  metadata {
    name = "kubias"
  }
}

resource "kubernetes_replication_controller" "kubias-rc" {
  metadata {
    name = "kubia"
    namespace = kubernetes_namespace.kubias-ns.metadata.0.name
    labels = {
      app = "kubia"
    }
  }
  spec {
    replicas = 4
    selector = {
      app = "kubia"
    }
    template {
      metadata {
```

```
        labels = {
          app = "kubia"
        }
      }
      spec {
        container {
          image = "lorenzomondani/kubia"
          name = "kubia"
          port {
            container_port = 8080
          }
        }
      }
    }
  }
}

resource "kubernetes_service" "kubias-lb" {
  metadata {
    namespace = kubernetes_namespace.kubias-ns.metadata.0.name
    name = "kubias-lb"
  }

  spec {
    selector = {
      app = "kubia"
    }
    port {
      port        = 80
      target_port = 8080
    }

    type = "LoadBalancer"
  }
}
```

Il file mostrato in Elenco 9.1 definisce un ReplicationController che crea quattro istanze del pod precedentemente descritto. Il servizio LoadBalancer crea quindi un load balancer, che distribuirà le richieste tra tali pod.

Questo load balancer è stato creato automaticamente da MetalLB. Esso, infatti, era già stato configurato sul cluster come unica applicazione capace di soddisfare le richieste di allocazione dei servizi di tipo LoadBalancer. MetalLB ha quindi creato il load balancer, assegnandogli uno degli indirizzi riservati che erano stati precedentemente configurati su MAAS e nella relativa ConfigMap.

Connettendosi a tale indirizzo sulla porta 80 è stato possibile accedere alle pagine web offerte dai vari pod, dimostrando il corretto funzionamento di MetalLB.

9.5 Volumi persistenti tramite Ceph

La verifica del funzionamento dei volumi Ceph è avvenuta definendo una risorsa StatefulSet. Tale risorsa crea un singolo pod, a cui è associato un VolumeClaim. Nel caso Ceph fosse stato configurato correttamente, tale VolumeClaim allocherà, automaticamente, tramite il relativo provisioner, un nuovo volume persistente Ceph.

Tale StatefulSet è stato applicato al cluster tramite il file di configurazione Terraform mostrato in Elenco 9.2.

Elenco 9.2: file di configurazione Terraform che definisce uno StatefulSet con un pod MongoDB per effettuare i test relativi ai volumi persistenti Ceph.

```
provider "kubernetes" {
  version = "1.13.3"
}

resource "kubernetes_namespace" "mongo-ns" {
  metadata {
    name = "mongo-ns"
  }
}

resource "kubernetes_stateful_set" "mongo-ss" {
  metadata {
    name = "mongo"
    namespace = kubernetes_namespace.mongo-ns.metadata.0.name
  }
  spec {
    service_name = "mongo"
    replicas = 1
    selector {
      match_labels = {
        app = "mongo"
      }
    }
    template {
      metadata {
        labels = {
          app = "mongo"
        }
      }
      spec {
        container {
          name = "mongo"
          image = "mongo"
          port {
            container_port = 8081
          }
```

```
          volume_mount {
            mount_path = "/data/db"
            name = "mongo-data"
          }
        }
      }
    }
    volume_claim_template {
      metadata {
        name = "mongo-data"
      }
      spec {
        access_modes = ["ReadWriteOnce"]
        resources {
          requests = {
            storage = "1Gi"
          }
        }
      }
    }
  }
}
```

Il file mostrato in Elenco 9.1 definisce un pod che ospita un container MongoDB, cioè un particolare database NoSQL. Questo pod, per poter salvare i dati relativi a tale database, necessita di un volume persistente. Le caratteristiche di tale volume sono specificate nella sezione "volume_claim_template". Questo template specifica la dimensione del volume, ma non specifica la StorageClass da utilizzare, e, per questo, verrà utilizzata quella di default. Al momento dell'installazione di Ceph e della definizione della relazione tra il charm ceph-mon ed il kubernetes-master, sono stati creati automaticamente, da Juju, le StorageClass ed i provisioner relativi a Ceph. Come StorageClass di default è stata impostata "ceph-xfs", che permette di allocare dei Block Device Ceph (RBD) formattati in xfs. I pod relativi allo StatefulSet definito hanno quindi allocato, automaticamente tali volumi.

Verificando, tramite kubectl, le proprietà del pod creato dallo StatefulSet, è stato possibile confermare l'avvenuta creazione ed associazione del volume Ceph.

Per verificare il corretto funzionamento dello StatefulSet e dei volumi Ceph associati al relativo pod, sono state eseguite le seguenti operazioni:

1. tramite i comandi

   ```
   kubectl exec -it mongo-0 mongo -n mongo-ns
   use data
   db.data.insert({data:'volume test'})
   db.data.find()
   ```

ci si è connessi al pod gestito dallo StatefulSet, eseguendo il comando "mongo" per accedere alla shell di MongoDB. È stato quindi salvato, all'interno della collezione "data" del database "data", un particolare JSON. Tramite il comando find è stato poi verificato che l'oggetto fosse stato effettivamente salvato;

2. tramite il comando

   ```
   kubectl delete pod mongo-0 -n mongo-ns
   ```

   è stato eliminato il pod a cui ci si era connessi precedentemente. Tale pod, essendo gestito dallo StatefulSet, è stato quindi automaticamente ricreato, riassociandogli il volume del pod precedente;

3. tramite i comandi

   ```
   kubectl exec -it mongo-0 mongo -n mongo-ns
   use data
   db.data.find()
   ```

   ci si è connessi al nuovo pod verificando il contenuto della collezione "data". La presenza dell'oggetto precedentemente salvato ha quindi confermato il corretto funzionamento dei volumi persistenti Ceph.

I dati relativi ai volumi Ceph vengono salvati, in maniera distribuita, tra le pendrive dedicate a tale scopo, di cui i nodi worker del cluster sono stati dotati. L'utilizzo di tali pendrive non ha evidenziato particolari criticità.

9.6 Kubernetes cluster autoscaler

La verifica del funzionamento del cluster autoscaler è avvenuta definendo un ReplicationController che potesse saturare, tramite i relativi pod, le risorse computazionali del cluster.

Tale ReplicationController è stato applicato al cluster tramite il file di configurazione Terraform mostrato in Elenco 9.3.

Elenco 9.3: file di configurazione Terraform che definisce un ReplicationController per effettuare i test di scaling automatico.

```
provider "kubernetes" {
  version = "1.13.3"
}

resource "kubernetes_namespace" "kubias-ns" {
  metadata {
    name = "kubias"
  }
```

```
}

resource "kubernetes_replication_controller" "kubias-rc" {
  metadata {
    name = "kubia"
    namespace = kubernetes_namespace.kubias-ns.metadata.0.name
    labels = {
      app = "kubia"
    }
  }
  spec {
    replicas = 9
    selector = {
      app = "kubia"
    }
    template {
      metadata {
        labels = {
          app = "kubia"
        }
      }
      spec {
        container {
          image = "lorenzomondani/kubia"
          name = "kubia"
          port {
            container_port = 8080
          }
          resources {
            limits {
              cpu = "1000m"
              memory = "300Mi"
            }
            requests {
              cpu = "1000m"
              memory = "300Mi"
            }
          }
        }
      }
    }
  }
}
```

Per effettuare i test di scaling automatico è stato introdotto un quinto Raspberry, che è stato connesso al BMC ed aggiunto a MAAS. I pod relativi al ReplicationController, mostrato in Elenco 9.3, richiedono l'allocazione di un intero core di CPU; poiché ogni nodo non possiede esattamente 4 core disponibili potrà eseguire al massimo tre di tali

pod. Impostandone, quindi, il numero a 9, sono state saturate le risorse dei tre nodi worker disponibili.

Successivamente, è stato modificato tale valore a 10: a tal punto, il cluster autoscaler, poiché il decimo pod non poteva essere schedulato, ha iniziato le operazioni di scaling, aggiungendo, tramite Juju, il nuovo nodo.

Infine, il numero di repliche è stato reimpostato a 9, rendendo il nodo precedentemente aggiunto sottoutilizzato e scatenando, quindi, un'operazione di scale in da parte del cluster autoscaler.

Sia le operazioni di scale out che di scale in automatiche hanno quindi avuto successo, senza causare particolari problemi.

9.7 Compatibilità del software con l'architettura Arm

La valutazione relativa alla compatibilità con l'architettura Arm è risultata molto positiva: sebbene, in alcuni casi, sia stato necessario intervenire, la totalità del software utilizzato è risultato pienamente compatibile con tale architettura.

Infatti, le immagini Ubuntu offerte da MAAS sono compilate per arm64, così come i charm Juju relativi a Kubernetes e Ceph e l'applicazione Kubernetes MetalLB. Ormai quasi la totalità delle immagini di container, presenti sui principali registri pubblici, è compilata sia per amd64 che per arm64 e questo ha permesso di creare facilmente pod Kubernetes che potessero essere eseguiti sul cluster di Raspberry realizzato.

Inoltre, tramite il firmware UEFI, è stato possibile adeguare il Raspberry Pi 4 B agli standard System Ready di Arm, incrementando notevolmente la compatibilità di tale piattaforma arm64 con i principali sistemi operativi in circolazione, tra cui, Ubuntu.

L'unico problema che è stato rilevato riguarda le immagini dei container relativi ai servizi Ceph, che non erano state compilate per arm64, nonostante il relativo charm fosse stato realizzato per tale architettura. In ogni caso il problema si è rivelato facilmente risolvibile e probabilmente verrà corretto dagli sviluppatori nel prossimo futuro.

In generale, tutti i software per arm64 adottati hanno dimostrato un comportamento identico alla controparte x86-64, senza evidenziare particolari criticità.

9.8 Possibili sviluppi futuri

I risultati ottenuti soddisfano pienamente gli obiettivi preposti; ciononostante, esiste ancora un ampio margine di miglioramento.

Uno dei componenti ulteriormente migliorabili potrebbe essere il BMC. Esso, infatti, non permette lo spegnimento della macchina nel caso questa si ritrovasse in uno stato di blocco completo, come un kernel panic. In tal caso, potrebbero essere utilizzati

dispositivi aggiuntivi, come, ad esempio, dei relè, per spegnere forzatamente i Raspberry, rimuovendo l'alimentazione. Il BMC effettuerebbe, comunque, un tentativo di spegnimento tramite il GPIO; se però, dopo un certo intervallo di tempo, lo stato della macchina risultasse il medesimo, rimuoverebbe l'alimentazione per forzare lo spegnimento. Inoltre, il BMC realizzato sfrutta il pin TX della seriale per verificare lo stato del Raspberry. Sebbene tale soluzione abbia portato a buoni risultati, sarebbe più opportuno adottare una strategia ad-hoc, ad esempio aggiungendo un'opportuna logica all'interno del firmware UEFI per cambiare lo stato di un pin al momento dell'accessione e dello spegnimento. Il BMC, infatti, se venissero inviati dati sull'interfaccia seriale, potrebbe leggere uno stato errato. Un' ulteriore problematica legata al BMC riguarda particolari fenomeni di accensione o spegnimento involontario dei Raspberry. Tali fenomeni sono causati da interferenze sulle connessioni tra il BMC ed i vari Raspberry, che provocano un temporaneo cambiamento di stato dei GPIO. Sebbene tali interferenze siano rare, si potrebbe tentare di introdurre una soluzione hardware o software per mitigare tale problema.

Sebbene l'ESP32 sia stato adatto alla realizzazione del BMC, potrebbe risultare relativamente ingombrante e difficile da installare fisicamente, poiché deve essere connesso a più Raspberry del cluster. Sarebbe interessante progettare un BMC di dimensioni molto più ridotte, ad esempio attraverso un ESP8266 in formato ESP-01, installandone uno per ogni Raspberry, direttamente sull'interfaccia GPIO. Ciò renderebbe i singoli Raspberry indipendenti, con il proprio BMC, facilitandone l'installazione e la dislocazione.

A causa delle limitazioni attuali del kernel Linux e del firmware UEFI, è stato possibile installare i sistemi operativi esclusivamente su dischi USB. Probabilmente, in futuro, questa limitazione verrà rimossa e, in tal caso, sarebbe interessante riuscire ad installare le varie componenti software sul supporto di memorizzazione standard dei Raspberry, cioè la scheda microSD.

Un ulteriore miglioramento potrebbe riguardare l'installazione e la configurazione della macchina virtuale relativa al controller MAAS. Infatti, tali procedure sono state eseguite manualmente, ma esistono strumenti che permetterebbero di automatizzare anche tali operazioni. Ad esempio, sarebbe possibile utilizzare il tool Vagrant, realizzato da HashiCorp, per la creazione automatica, attraverso un file di configurazione, della macchina virtuale. Successivamente, il software relativo al controller MAAS potrebbe essere installato automaticamente attraverso Ansible.

Anche le operazioni di configurazione, su MAAS, dei singoli Raspberry, successive alla fase di enlist, sono state eseguite manualmente. Tali operazioni sono consistite nel definire, per ogni macchina, il nome, il power driver da utilizzare e gli eventuali tag. Sarebbe comodo eseguire anche tali operazioni automaticamente, sfruttando, ad esempio, degli strumenti per la gestione delle configurazioni.

Per quanto riguarda Juju, sarebbe inoltre interessante tentare di effettuare, sul cluster

di Raspberry realizzato, il deployment di un'infrastruttura differente da Kubernetes: ad esempio di OpenStack, Hadoop o Spark, per le quali sono disponibili i relativi bundle Juju preconfigurati.

Per quanto riguarda l'autoscaler, le tempistiche necessarie per la configurazione di un nuovo nodo sono risultate molto elevate. Ciò è stato causato sia dalle caratteristiche hardware, sia dalla necessità di reinstallare completamente il sistema operativo. Per risolvere tale problema, si potrebbe tentare di preinstallare le macchine con i componenti relativi al Kubernetes worker; al momento di un'operazione di scale out, l'autoscaler si limiterebbe ad accedere la macchina, senza avviare il processo di installazione. Al momento di uno scale in, la macchina verrebbe semplicemente spenta. Per quanto riguarda l'hardware, invece, si potrebbero utilizzare dei dischi più veloci, come degli SSD, per velocizzare il processo di installazione.

L'implementazione dell'autoscaler realizzata è relativamente limitata, poiché progettata per risolvere questo caso specifico, relativo ad un singolo cluster di Raspberry. Sarebbe interessante estendere tale implementazione per realizzarne un cloud provider Juju generico, capace di gestire efficacemente i NodeGroup e tutte le funzionalità più avanzate. Inoltre, quando verrà rilasciato il codice ufficiale relativo al cloud provider implementato tramite gRPC [40], sarebbe opportuno adattare, di conseguenza, il Juju Python client realizzato.

Una tematica non affrontata dettagliatamente in questo progetto riguarda la sicurezza. Molti dei software utilizzati, come MAAS, Juju e Kubernetes, offrono vari meccanismi per il controllo degli accessi o per separare, logicamente, le reti relative ad applicazioni diverse.

MAAS e Juju permettono, ad esempio, di configurare dei network space per mantenere separate le reti delle macchine, mentre Kubernetes, sfruttando il plug-in CNI Calico al posto di flannel, potrebbe isolare il traffico relativo ad una determinata applicazione, tramite gli oggetti NetworkPolicy. Sarebbe interessante tentare di applicare queste funzionalità al cluster realizzato, per verificare il grado di isolamento ottenibile tra le varie applicazioni.

Anche alcune caratteristiche legate alla sicurezza dell'autoscaler e del BMC sarebbero migliorabili. Le chiamate gRPC tra il cluster autoscaler ed il Juju Python client realizzati avvengono infatti in modalità non sicura; si potrebbero quindi sfruttare le modalità offerte da gRPC per autenticare e rendere sicure tali comunicazioni. Anche il BMC realizzato espone le proprie API senza adottare particolari tecniche di autenticazione o di crittografia e, per questo, sarebbe anch'esso migliorabile sotto questo punto di vista.

Conclusioni

Gli obiettivi preposti sono stati completamente raggiunti. Infatti, è stato possibile costruire, tramite processi completamente automatici, un cluster Kubernetes, composto da nodi fisici a basso costo realizzati tramite il single-board computer (SBC) Raspberry Pi 4 B. In particolare, è stato creato un cluster composto da quattro Raspberry Pi, comprendenti un nodo Kubernetes master e tre Kubernetes worker, oltre che ad un Raspberry Pi aggiuntivo per effettuare le prove di scaling. I tre nodi Kubernetes worker sono stati configurati in modo tale da poter ospitare anche i servizi di storage relativi a Ceph, che hanno permesso l'allocazione, da parte di Kubernetes, di volumi persistenti, distribuiti e tolleranti ai guasti. L'adozione del nodo aggiuntivo si è rivelata necessaria per evitare di rimuovere, durante le operazioni di scale in, un nodo ospitante i servizi Ceph.

La gestione dei vari SBC è avvenuta attraverso lo strumento, offerto da Canonical, chiamato MAAS (Metal As A Service), che ha permesso di automatizzare i processi di installazione e configurazione delle macchine, permettendo di realizzare, di fatto, un'infrastruttura bare-metal cloud privata. Per permettere a MAAS di gestire correttamente i vari Raspberry, è stato però necessario adottare un'ulteriore tecnologia: il Raspberry Pi 4 UEFI Firmware, cioè un'implementazione open-source basata sul firmware UEFI TianoCore EDKII. Tale firmware ha permesso ai vari Raspberry di effettuare il boot da rete UEFI PXE dall'ambiente di boot configurato da MAAS. Inoltre, poiché rispetta gli standard Arm SystemReady SBBR (Server Base Boot Requirements), basati su UEFI ed ACPI, esso ha permesso di effettuare il boot e l'installazione delle immagini Ubuntu generiche, compilate per arm64, offerte da MAAS. Senza il firmware UEFI, MAAS non avrebbe potuto gestire i vari Raspberry, poiché esso può installare esclusivamente immagini di sistemi operativi standard; i Raspberry, invece, permettono solamente l'esecuzione di sistemi operativi appositamente modificati per tale hardware. L'adozione del firmware UEFI ha permesso di risolvere, facilmente, tale incompatibilità, rendendo i vari Raspberry simili a delle macchine ordinarie, sulle quali è possibile installare un qualsiasi sistema operativo generico.

Per permettere ai vari Raspberry di essere gestiti tramite MAAS è stato quindi necessario fornirgli il firmware UEFI, configurandolo in modo che il boot PXE fosse la

prima opzione nell'ordine di boot. Il Raspberry Pi 4 UEFI Firmware viene solitamente installato sulla microSD del Raspberry e, successivamente, configurato. Tale configurazione, che comprende l'ordine delle opzioni di boot, viene quindi mantenuta sul relativo file salvato sulla microSD. Per evitare di installare e configurare manualmente il firmware UEFI su ogni singolo Raspberry, si è deciso di fornirlo, a sua volta, attraverso PXE. La EEPROM di ogni Raspberry è stata quindi riconfigurata, per abilitare il boot da rete. È stato quindi configurato il server DHCP e TFTP di MAAS in modo tale che potesse fornire i file relativi al firmware UEFI. Nella strategia individuata, ogni Raspberry effettua quindi un boot da rete per reperire ed avviare UEFI, che, a sua volta, effettua un ulteriore boot da rete per avviare il sistema operativo fornito da MAAS. Per applicare tale strategia è stato però necessario modificare il firmware UEFI, in modo tale che, al momento della generazione delle opzioni di boot, inserisse quella relativa a PXE in testa.

Oltre a ciò, è stato necessario risolvere due problemi legati al firmware UEFI. Il primo ha riguardato i driver UEFI della scheda di rete, che nelle ultime versioni del firmware non risultano completamente funzionanti; il secondo ha riguardato la connessione di tutte le sorgenti di boot, per permettere a MAAS di individuare il bootloader del sistema operativo installato. Tali problemi sono stati risolti modificando opportunamente il codice sorgente del firmware UEFI.

A causa dell'assenza, nel kernel Linux, dei driver ACPI relativi al controller della scheda microSD, è stato necessario installare i sistemi operativi su dischi esterni, connessi all'interfaccia USB. Per contenere i costi, si è scelto quindi di adottare delle pendrive USB.

Per permettere a MAAS di gestire le operazioni di accensione e spegnimento dei singoli nodi è stato realizzato un apposito BMC (Baseboard Management Controller), tramite il microcontrollore ESP32 e l'ambiente di sviluppo MicroPython. Tale BMC realizzato espone un'API HTTP, e utilizza l'interfaccia GPIO dei vari Raspberry per impartire i comandi di accensione o spegnimento. Il Raspberry permette, nativamente, l'accensione tramite il GPIO3, ma non lo spegnimento. Se fosse stato utilizzato un sistema operativo realizzato per Raspberry, basato sulla tecnologia devicetree per l'individuazione dell'hardware, sarebbe stato possibile applicare un overlay, offerto dagli sviluppatori, per definire un pulsante di spegnimento associato ad un particolare pin del GPIO. Poiché il firmware UEFI si basa su ACPI e non su devicetree, è stato necessario implementare tale pulsante modificando opportunamente la tabella DSDT di ACPI, definendo un device "power button" ed un device "Generic Event Device" capace di interagire direttamente con i registri del GPIO.

È stato quindi realizzato il relativo driver necessario a MAAS per comunicare con le API del BMC.

Grazie al firmware UEFI ed al BMC realizzato, è stato quindi possibile fornire a MAAS il completo controllo dei Raspberry del cluster.

Successivamente, tramite Juju, è stato possibile installare, in maniera completamente automatica, tutti i componenti software relativi a Kubernetes e Ceph, definendo un apposito modello dell'infrastruttura da realizzare. Tale automatismo è stato possibile grazie ai servizi di bare-metal cloud offerti da MAAS. La possibilità di allocare e configurare un cluster Kubernetes su richiesta ha permesso di realizzare, di fatto, un servizio cloud privato riconducibile al modello Cluster as a Service.

L'installazione di Ceph ha permesso l'allocazione, da parte di Kubernetes, di volumi persistenti e distribuiti, tramite i provisioner e le StorageClass configurate automaticamente dai charm Juju. È stato però necessario modificare manualmente i riferimenti alle immagini dei container relative ai servizi Ceph, poiché quelle fornite non erano compilate per arm64.

Successivamente è stato affrontato il tema dello scaling automatico dei Kubernetes worker. Juju permette di effettuare operazioni di scaling, aggiungendo o rimuovendo le unità, dal modello, relative ai Kubernetes worker. Tali operazioni devono però essere specificate dall'amministratore. È stato quindi sviluppato un sistema che ha permesso di eseguire tali operazioni di scaling automaticamente, sulla base delle risorse disponibili, agendo direttamente sul modello Juju. In particolare, è stato esteso il cluster autoscaler di Kubernetes, abilitandolo alla comunicazione con Juju per aggiungere o rimuovere nodi worker. Per risolvere il problema legato all'intercompatibilità tra il codice del cluster autoscaler e quello della libreria python-libjuju, necessaria per interagire con il controller Juju, sono state sfruttate le chiamate a procedura remota gRPC, sulla base di una proposta della comunità nata per facilitare l'estensione dell'autoscaler. Per rendere compatibile l'autoscaler con i nodi worker configurati da Juju, è stato però necessario modificare il relativo charm, in modo tale che impostasse la proprietà providerID, necessaria per associare i nodi Kubernetes ad i nodi del provider.

La gestione delle risorse Kubernetes e il deployment delle relative applicazioni, tra cui il cluster autoscaler e MetalLB, è stata facilitata e velocizzata dallo strumento di IaC (Infrastructure as Code) Terraform. Tramite tale strumento, è stato possibile dichiarare le risorse Kubernetes necessarie al deployment delle varie applicazioni e di applicarle o eliminarle, velocemente, al cluster.

MetalLB ha abilitato la possibilità di allocare, sul cluster Kubernetes realizzato, dei servizi di tipo LoadBalancer, capaci di esporre all'esterno della rete del cluster, su un unico indirizzo, i servizi offerti da un determinato gruppo di pod.

La totalità del software utilizzato è risultato perfettamente compatibile con l'architettura arm64 e, in generale, con l'hardware offerto dai Raspberry, senza mostrare particolari criticità.

I risultati ottenuti sono stati più che soddisfacenti, anche se alcune delle funzionalità realizzate o delle strategie adottate lasciano comunque spazio ad eventuali miglioramenti e sviluppi futuri.

Le tecnologie adottate sono risultate perfettamente adatte al raggiungimento degli obiettivi, permettendo di realizzare e gestire, facilmente e velocemente, un cluster Kubernetes, a diversi livelli di astrazione, partendo dall'hardware, tramite MAAS, fino a raggiungere il livello applicativo, tramite Juju e Terraform.

Sicuramente, non si vuole affermare che la soluzione proposta sia l'alternativa migliore in un ambito di produzione. In tal caso sarebbe infatti molto più conveniente affidarsi ad un servizio cloud pubblico, o, nel caso si desiderasse realizzare il proprio cloud privato, adottare un hardware di categoria server. Ma l'obiettivo principale di questo progetto consisteva nel realizzare un cluster Kubernetes entry level, gestito automaticamente ma al contempo accessibile dal punto di vista dei costi, non per essere adottato in ambienti di produzione, ma rivolto a tutti coloro che volessero realizzare un proprio cluster per scopi di studio o per lo sviluppo di applicazioni distribuite.

Non si vuole nemmeno affermare che il modo migliore per gestire un cluster di Raspberry sia attraverso MAAS e Juju: è stato semplicemente dimostrato che tale modalità può portare a buoni risultati. Tali software richiedono, comunque, alcuni sforzi per il set-up iniziale, e necessitano della presenza di macchine aggiuntive sulle quali dovranno essere eseguiti i vari controller. Essi potrebbero essere quindi adatti per cluster di grandi dimensioni, ma, nel caso si avessero a disposizione solamente pochi nodi, potrebbe risultare più conveniente l'installazione manuale, affiancata da altri strumenti, che non richiedono la presenza di un controller, come Ansible.

Inoltre, per rendere i vari Raspberry compatibili con MAAS, è stato necessario adottare il firmware UEFI, che, sebbene introduca un elevato grado di compatibilità, è ancora in fase sperimentale, e, per questo, potrebbe non essere adatto a contesti in cui è richiesta un'alta affidabilità. In tal caso, potrebbe essere più conveniente gestire le macchine manualmente, installando i sistemi operativi appositamente realizzati per il Raspberry Pi.

In ogni caso, tramite questo progetto, è stato possibile studiare e adottare vari software di gestione delle risorse, che hanno permesso di ottenere, su un insieme di macchine fisiche a basso costo, un livello di automatismo paragonabile a quello offerto dai principali cloud provider pubblici.

Bibliografia

[20] Yevgeniy Brikman. *Terraform: Up & Running: Writing Infrastructure As Code*. 2ª ed. Oreilly & Associates Inc, 2019. 339 pp. ISBN: 9781492046905.

[68] Marko Luksa. *Kubernetes in Action*. 1ª ed. Manning Publications, 2018. 594 pp. ISBN: 9781617293726.

[93] Kief Morris. *Infrastructure as Code: Dynamic Systems for the Cloud Age*. 2ª ed. O'Reilly Media, 2020. 430 pp. ISBN: 9781098114671.

[94] Scott M. Mueller. *Upgrading and Repairing PCs*. 22ª ed. Que Pub, 2015. 1162 pp. ISBN: 9780789756107.

[98] Peter Pacheco. *An Introduction to Parallel Programming*. 1ª ed. Morgan Kaufmann, 2011. 392 pp. ISBN: 9780123742605.

[121] Chris Richardson. *Microservices Patterns: With examples in Java*. 1ª ed. Manning Publications, 2018. 522 pp. ISBN: 9781617294549.

[123] Gigi Sayfan. *Mastering Kubernetes*. 3ª ed. Packt Publishing Ltd, 2020. 643 pp. ISBN: 9781839213083.

[124] Jiming Sun et al. *Embedded Firmware Solutions: Development Best Practices for the Internet of Things*. 1ª ed. Apress, 2015. 224 pp. ISBN: 9781484200704.

[135] Rajesh R V. *Spring 5.0 Microservices*. 2ª ed. Packt Publishing, 2018. 408 pp. ISBN: 9781787127685.

Sitografia

[1] *8 Container Orchestration Tools to Know*. URL: https://web.archive.org/web/20201108104014/https://www.linux.com/news/8-open-source-container-orchestration-tools-know/ (visitato il 18/01/2021).

[2] *AArch64*. URL: https://web.archive.org/web/20210113033937/https://en.wikipedia.org/wiki/AArch64 (visitato il 25/01/2021).

[3] *About MAAS*. URL: https://web.archive.org/web/20210128114839/https://maas.io/docs/snap/2.9/ui/about-maas (visitato il 28/01/2021).

[4] *Advanced Configuration and Power Interface*. URL: https://web.archive.org/web/20210126162540/https://en.wikipedia.org/wiki/Advanced_Configuration_and_Power_Interface (visitato il 26/01/2021).

[5] *Advanced Configuration and Power Interface (ACPI) Specification*. URL: https://web.archive.org/web/20210126093316/https://uefi.org/sites/default/files/resources/ACPI_Spec_6_4_Jan22.pdf (visitato il 26/01/2021).

[6] *Ansible*. URL: https://web.archive.org/web/20210201183601/https://www.ansible.com/ (visitato il 01/02/2021).

[7] *Apache Hadoop*. URL: https://web.archive.org/web/20210115060724/https://en.wikipedia.org/wiki/Apache_Hadoop (visitato il 18/01/2021).

[8] *Apache MESOS*. URL: https://web.archive.org/web/20210116043803/http://mesos.apache.org/ (visitato il 18/01/2021).

[9] *ARM architecture*. URL: https://web.archive.org/web/20210119203159/https://en.wikipedia.org/wiki/ARM_architecture (visitato il 22/01/2021).

[10] *ARM Servers: Mobile CPU Architecture For Datacentres?* URL: https://www.toptal.com/back-end/arm-servers-armv8-for-datacentres (visitato il 22/01/2021).

[11] *Arm SystemReady ES*. URL: https://web.archive.org/web/20201120140840/https://developer.arm.com/architectures/system-architectures/arm-systemready/es (visitato il 25/01/2021).

[12] *Arm SystemReady IR*. URL: https://web.archive.org/web/20201020114631/https://developer.arm.com/architectures/system-architectures/arm-systemready/ir (visitato il 25/01/2021).

[13] *Arm SystemReady LS*. URL: https://web.archive.org/web/20201020123632/https://developer.arm.com/architectures/system-architectures/arm-systemready/ls (visitato il 25/01/2021).

[14] *Arm SystemReady SR*. URL: https://web.archive.org/web/20210125165518/https://developer.arm.com/architectures/system-architectures/arm-systemready/sr (visitato il 25/01/2021).

[15] *Arm® Base Boot Requirements 1.0*. URL: https://web.archive.org/web/20210125160057/https://documentation-service.arm.com/static/5fb7e4b4d77dd807b9a80c87?token= (visitato il 25/01/2021).

[16] *ARMed for the living room*. URL: https://web.archive.org/web/20201012060744/https://www.cnet.com/news/armed-for-the-living-room/ (visitato il 22/01/2021).

[17] *AWS CloudFormation*. URL: https://web.archive.org/web/20210201183622/https://aws.amazon.com/cloudformation/?nc1=h_ls (visitato il 01/02/2021).

[18] *BCM2711 ARM Peripherals*. URL: https://web.archive.org/web/20210218124004/https://datasheets.raspberrypi.org/bcm2711/bcm2711-peripherals.pdf (visitato il 19/02/2021).

[19] *BcmGenetDxe 1.16*. URL: https://web.archive.org/web/20210218115243/https://github.com/tianocore/edk2-platforms/tree/8dd78ea11a38d2d7e8031c4783e9f2ca5569956b/Silicon/Broadcom/Drivers/Net/BcmGenetDxe (visitato il 18/02/2021).

[21] *Ceph Architecture*. URL: https://web.archive.org/web/20210209152504/https://docs.ceph.com/en/latest/architecture/ (visitato il 09/02/2021).

[22] *Ceph Base*. URL: https://web.archive.org/web/20210223110941/https://jaas.ai/ceph-base/bundle/3 (visitato il 23/02/2021).

[23] *Charmed Kubernetes*. URL: https://web.archive.org/web/20210216101750/https://jaas.ai/charmed-kubernetes/bundle/559 (visitato il 16/02/2021).

[24] *Charmscaler*. URL: https://web.archive.org/web/20210217144603/https://jaas.ai/charmscaler/5 (visitato il 17/02/2021).

[25] *Chef*. URL: https://web.archive.org/web/20210201183605/https://www.chef.io/products/chef-infra (visitato il 01/02/2021).

[26] *Cluster autoscaler AWS Example Deployment*. URL: https://web.archive.org/web/20210224164022/https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/examples/cluster-autoscaler-one-asg.yaml (visitato il 24/02/2021).

[27] *Cluster autoscaler FAQ*. URL: https://web.archive.org/web/20210203052952/https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md (visitato il 24/02/2021).

[28] *Cobbler*. URL: https://web.archive.org/web/20210127182442/http://cobbler.github.io/ (visitato il 27/01/2021).

[29] *Collins*. URL: https://web.archive.org/web/20210127182135/https://tumblr.github.io/collins/index.html (visitato il 27/01/2021).

[30] *Commodity Hardware*. URL: https://web.archive.org/web/20200922111047/https://susedefines.suse.com/definition/commodity-hardware/ (visitato il 18/01/2021).

[31] *Community building blocks for HPC systems*. URL: https://web.archive.org/web/20210105123155/https://openhpc.community/ (visitato il 18/01/2021).

[32] *Comparison of instruction set architectures*. URL: https://web.archive.org/web/20210121190140/https://en.wikipedia.org/wiki/Comparison_of_instruction_set_architectures (visitato il 21/01/2021).

[33] *Complex instruction set computer*. URL: https://web.archive.org/web/20201218220047/https://en.wikipedia.org/wiki/Complex_instruction_set_computer (visitato il 22/01/2021).

[34] *Concepts and terms*. URL: https://web.archive.org/web/20201127204840/https://juju.is/docs/concepts-and-terms (visitato il 02/02/2021).

[35] *Controllers*. URL: https://web.archive.org/web/20210202161707/https://juju.is/docs/controllers (visitato il 02/02/2021).

[36] *CoreOS fleet*. URL: https://web.archive.org/web/20201104221400/https://coreos.com/fleet/docs/latest/ (visitato il 05/02/2021).

[37] *CPU platforms*. URL: https://web.archive.org/web/20210110112607/https://cloud.google.com/compute/docs/cpu-platforms (visitato il 22/01/2021).

[38] *Device tree*. URL: https://web.archive.org/web/20210125104305/https://en.wikipedia.org/wiki/Device_tree (visitato il 25/01/2021).

[39] *Digital Rebar*. URL: https://web.archive.org/web/20210127182129/https://rackn.com/rebar/ (visitato il 27/01/2021).

[40] *doc: proposal custom cloud provider over gRPC*. URL: https://web.archive.org/web/20210211193231/https://github.com/kubernetes/autoscaler/pull/3140 (visitato il 17/02/2021).

[41] *doc: proposal custom cloud provider over gRPC merge commit*. URL: https://web.archive.org/web/20210223151706/https://github.com/kubernetes/autoscaler/commit/9cffba500c3a9bc9efd725100dd62b48adc8fd72 (visitato il 23/02/2021).

[42] *Docker Swarm*. URL: https://web.archive.org/web/20210309162514/https://docs.docker.com/engine/swarm/ (visitato il 09/03/2021).

[43] *Embedded Base Boot Requirements (EBBR) Specification*. URL: https://web.archive.org/web/20210125155226/https://arm-software.github.io/ebbr/ (visitato il 25/01/2021).

[44] *EmulatorPkg: Make the shell be the first boot option*. URL: https://web.archive.org/web/20210219142546/https://github.com/tianocore/edk2/commit/0e92957eaa50be08ebbd8f0fd4e989b380704466 (visitato il 19/02/2021).

[45] *Espress if IoT Development Framework*. URL: https://web.archive.org/web/20210127021724/https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/ (visitato il 22/02/2021).

[46] *Exiting x86: Apple & Microsoft Embrace Arm-based PC*. URL: https://web.archive.org/web/20210122160741/https://wikibon.com/apple-microsoft-embrace-arm-based-pc/ (visitato il 22/01/2021).

[47] *FAI - Fully Automatic Installation*. URL: https://web.archive.org/web/20210127182146/https://fai-project.org/ (visitato il 27/01/2021).

[48] *Foreman*. URL: https://web.archive.org/web/20210127182205/https://www.theforeman.org/ (visitato il 27/01/2021).

[49] *grpc-go 1.31.1*. URL: https://web.archive.org/web/20210223173233/https://github.com/grpc/grpc-go/releases/tag/v1.31.1 (visitato il 23/02/2021).

[50] *How can an OS change the boot order?* URL: https://web.archive.org/web/20210126122424/https://superuser.com/questions/936430/how-can-an-os-change-the-boot-order (visitato il 26/01/2021).

[51] *Icona microSD*. URL: https://web.archive.org/web/20210308160125/https://icon-icons.com/icon/micro-sd/23556 (visitato il 08/03/2021).

[52] *Icona pendrive*. URL: https://web.archive.org/web/20210308174110/https://freesvg.org/usb-thumb-drive-4-vector-image (visitato il 08/03/2021).

[53] *Instruction set architecture*. URL: https://web.archive.org/web/20210121190520/https://en.wikipedia.org/wiki/Instruction_set_architecture (visitato il 21/01/2021).

[54] *Ironic*. URL: https://web.archive.org/web/20210127182041/https://wiki.openstack.org/wiki/Ironic (visitato il 27/01/2021).

[55] *JAAS Dashboard, the new Juju GUI*. URL: https://web.archive.org/web/20201126085154/https://discourse.charmhub.io/t/jaas-dashboard-the-new-juju-gui/2978 (visitato il 02/02/2021).

[56] *js-libjuju*. URL: https://web.archive.org/web/20210217101040/https://github.com/juju/js-libjuju (visitato il 17/02/2021).

[57] *Juju*. URL: https://web.archive.org/web/20210201183632/https://juju.is/docs/what-is-juju (visitato il 01/02/2021).

[58] *Juju Clouds*. URL: https://web.archive.org/web/20201108014117/https://juju.is/docs/clouds (visitato il 02/02/2021).

[59] *Kubernetes (K8s)*. URL: https://web.archive.org/web/20210116044342/https://github.com/kubernetes/kubernetes (visitato il 18/01/2021).

[60] *Kubernetes cluster autoscaler*. URL: https://web.archive.org/web/20210217092355/https://github.com/kubernetes/autoscaler/tree/cluster-autoscaler-release-1.20/cluster-autoscaler (visitato il 17/02/2021).

[61] *Kubernetes cluster autoscaler cloud_provider.go*. URL: https://web.archive.org/web/20210216143521/https://github.com/kubernetes/autoscaler/blob/cluster-autoscaler-release-1.20/cluster-autoscaler/cloudprovider/cloud_provider.go (visitato il 17/02/2021).

[62] *Kubernetes Core*. URL: https://web.archive.org/web/20210216144105/https://jaas.ai/kubernetes-core/bundle/1163 (visitato il 16/02/2021).

[63] *Kubernetes master cdk-addons*. URL: https://web.archive.org/web/20210223112109/https://snapcraft.io/cdk-addons (visitato il 23/02/2021).

[64] *Kubernetes Worker Charm*. URL: https://web.archive.org/web/20210224150034/https://jaas.ai/u/containers/kubernetes-worker/718 (visitato il 24/02/2021).

[65] *Kubernetes Worker Charm kubernetes_worker.py*. URL: https://web.archive.org/web/20210224124136/https://github.com/charmed-kubernetes/charm-kubernetes-worker/blob/355a26526770b8a9c70477f85ef9d5a178cf7963/reactive/kubernetes_worker.py (visitato il 24/02/2021).

[66] *List of cluster management software*. URL: https://web.archive.org/web/20210118174306/https://en.wikipedia.org/wiki/List_of_cluster_management_software (visitato il 18/01/2021).

[67] *Logo Raspberry*. URL: https://web.archive.org/web/20210308161152/https://www.raspberrypi.org/trademark-rules/ (visitato il 08/03/2021).

[69] *MAAS*. URL: https://web.archive.org/web/20210126210157/https://maas.io/ (visitato il 27/01/2021).

[70] *MAAS Add machines*. URL: https://web.archive.org/web/20210129100940/https://maas.io/docs/snap/2.9/ui/add-machines (visitato il 29/01/2021).

[71] *MAAS Commission machines*. URL: https://web.archive.org/web/20210129102734/https://maas.io/docs/snap/2.9/ui/commission-machines (visitato il 29/01/2021).

[72] *MAAS Concepts and terms*. URL: https://web.archive.org/web/20210128144815/https://maas.io/docs/concepts-and-terms (visitato il 28/01/2021).

[73] *MAAS Controllers*. URL: https://web.archive.org/web/20210128113213/https://maas.io/docs/snap/2.9/ui/controllers (visitato il 28/01/2021).

[74] *MAAS Deploy machines*. URL: https://web.archive.org/web/20210129102728/https://maas.io/docs/snap/2.9/ui/deploy-machines (visitato il 29/01/2021).

[75] *MAAS Images*. URL: https://web.archive.org/web/20210129122626/https://maas.io/docs/snap/2.9/ui/images (visitato il 29/01/2021).

[76] *MAAS IP ranges*. URL: https://web.archive.org/web/20210129173744/https://maas.io/docs/snap/2.9/ui/ip-ranges (visitato il 29/01/2021).

[77] *MAAS Managing DHCP*. URL: https://web.archive.org/web/20201129103317/https://maas.io/docs/snap/2.9/ui/managing-dhcp (visitato il 29/01/2021).

[78] *MAAS NTP services*. URL: https://web.archive.org/web/20210129182024/https://maas.io/docs/snap/2.9/ui/ntp-services (visitato il 29/01/2021).

[79] *MAAS power drivers*. URL: https://web.archive.org/web/20210222115748/https://github.com/maas/maas/tree/2.9/src/provisioningserver/drivers/power (visitato il 22/02/2021).

[80] *MAAS Power management*. URL: https://web.archive.org/web/20210129123113/https://maas.io/docs/snap/2.9/ui/power-management (visitato il 29/01/2021).

[81] *MAAS Proxy*. URL: https://web.archive.org/web/20210129182013/https://maas.io/docs/snap/2.9/ui/proxy (visitato il 29/01/2021).

[82] *MAAS Subnet management*. URL: https://web.archive.org/web/20210129173339/https://maas.io/docs/snap/2.9/ui/subnet-management (visitato il 29/01/2021).

[83] *MAAS Ubuntu kernels*. URL: https://web.archive.org/web/20210129120932/https://maas.io/docs/snap/2.9/cli/ubuntu-kernels (visitato il 29/01/2021).

[84] *MAAS virsh.py*. URL: https://web.archive.org/web/20210224182306/https://github.com/maas/maas/blob/2.9/src/provisioningserver/drivers/pod/virsh.py (visitato il 24/02/2021).

[85] *MAAS VM hosting*. URL: https://web.archive.org/web/20201129101301/https://maas.io/docs/snap/2.9/ui/vm-hosting (visitato il 29/01/2021).

[86] *Marathon*. URL: https://web.archive.org/web/20210131171428/https://mesosphere.github.io/marathon/ (visitato il 05/02/2021).

[87] *MetalLB*. URL: https://web.archive.org/web/20210209104020/https://metallb.universe.tf/concepts/ (visitato il 09/02/2021).

[88] *Microcode*. URL: https://web.archive.org/web/20201211173259/https://en.wikipedia.org/wiki/Microcode (visitato il 22/01/2021).

[89] *MicroPython*. URL: https://web.archive.org/web/20210218193129/https://micropython.org/ (visitato il 22/02/2021).

[90] *MicroPython class Pin – control I/O pins*. URL: https://web.archive.org/web/20210116125330/http://docs.micropython.org/en/latest/library/machine.Pin.html (visitato il 22/02/2021).

[91] *Modello ESP32*. URL: https://web.archive.org/web/20210308174333/https://forum.fritzing.org/t/esp32s-hiletgo-dev-boad-with-pinout-template/5357 (visitato il 08/03/2021).

[92] *Modello Raspberry Pi 4*. URL: https://web.archive.org/web/20210308174256/https://forum.fritzing.org/t/raspberry-pi-4-model-b/8622/18 (visitato il 08/03/2021).

[95] *Nomad*. URL: https://web.archive.org/web/20210112011544/https://github.com/hashicorp/nomad (visitato il 18/01/2021).

[96] *OpenStack Heat*. URL: https://web.archive.org/web/20210201183626/https://wiki.openstack.org/wiki/Heat (visitato il 01/02/2021).

[97] *Otter*. URL: https://web.archive.org/web/20210201183612/https://inedo.com/otter (visitato il 01/02/2021).

[99] *Platform/RaspberryPi: don't connect all devices on an ordinary boot*. URL: https://web.archive.org/web/20210218144747/https://github.com/tianocore/edk2-platforms/commit/c8000ecccc83b728baf04ced2fedb870bc3bc1b3 (visitato il 18/02/2021).

[100] *Preboot Execution Environment.* URL: https://web.archive.org/web/20210128150651/https://en.wikipedia.org/wiki/Preboot_Execution_Environment (visitato il 28/01/2021).

[101] *Processore AWS Graviton.* URL: https://web.archive.org/web/20210122194152/https://aws.amazon.com/it/ec2/graviton/ (visitato il 22/01/2021).

[102] *Protocol Buffers v3.14.0.* URL: https://web.archive.org/web/20210207011633/https://github.com/protocolbuffers/protobuf/releases/tag/v3.14.0 (visitato il 23/02/2021).

[103] *Puppet.* URL: https://web.archive.org/web/20210201183609/https://puppet.com/ (visitato il 01/02/2021).

[104] *python-libjuju.* URL: https://web.archive.org/web/20210217101138/https://github.com/juju/python-libjuju (visitato il 17/02/2021).

[105] *RackHD.* URL: https://web.archive.org/web/20210127182059/https://rackhd.github.io/ (visitato il 27/01/2021).

[106] *Raspberry Pi 4 bootloader configuration.* URL: https://web.archive.org/web/20210302103601/https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711_bootloader_config.md (visitato il 02/03/2021).

[107] *Raspberry Pi 4 Tech Specs.* URL: https://web.archive.org/web/20210210090914/https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/ (visitato il 11/02/2021).

[108] *Raspberry Pi 4 UEFI Firmware 1.21.* URL: https://web.archive.org/web/20201205181535/https://github.com/pftf/RPi4/releases/tag/v1.21 (visitato il 18/02/2021).

[109] *Raspberry Pi 4 UEFI Firmware appveyor.yml.* URL: https://web.archive.org/web/20210219160705/https://github.com/pftf/RPi4/blob/8b4bb587b1df7ce0ba3517dfde492964a7500802/appveyor.yml (visitato il 19/02/2021).

[110] *Raspberry Pi 4 UEFI Firmware build_firmware.sh.* URL: https://web.archive.org/web/20210219161931/https://github.com/pftf/RPi4/blob/8b4bb587b1df7ce0ba3517dfde492964a7500802/build_firmware.sh (visitato il 19/02/2021).

[111] *Raspberry Pi 4 UEFI Firmware ConfigDxe.c*. URL: https://web.archive.org/web/20210219111753/https://github.com/tianocore/edk2-platforms/blob/867efd012b2f2fa98854eaee896832e0ff2e52c4/Platform/RaspberryPi/Drivers/ConfigDxe/ConfigDxe.c (visitato il 19/02/2021).

[112] *Raspberry Pi 4 UEFI Firmware Dsdt.asl*. URL: https://web.archive.org/web/20210218152501/https://github.com/tianocore/edk2-platforms/blob/867efd012b2f2fa98854eaee896832e0ff2e52c4/Platform/RaspberryPi/AcpiTables/Dsdt.asl (visitato il 18/02/2021).

[113] *Raspberry Pi 4 UEFI Firmware GpioLib.c*. URL: https://web.archive.org/web/20210219112419/https://github.com/tianocore/edk2-platforms/blob/867efd012b2f2fa98854eaee896832e0ff2e52c4/Silicon/Broadcom/Bcm283x/Library/GpioLib/GpioLib.c (visitato il 19/02/2021).

[114] *Raspberry Pi 4 UEFI Firmware GpuDevs.asl*. URL: https://web.archive.org/web/20210218174913/https://github.com/tianocore/edk2-platforms/blob/867efd012b2f2fa98854eaee896832e0ff2e52c4/Platform/RaspberryPi/AcpiTables/GpuDevs.asl (visitato il 18/02/2021).

[115] *Raspberry Pi 4 UEFI Firmware Images*. URL: https://web.archive.org/web/20210212170355/https://github.com/pftf/RPi4 (visitato il 12/02/2021).

[116] *Raspberry Pi 4 UEFI Firmware PlatformBm.c*. URL: https://web.archive.org/web/20210218145118/https://github.com/tianocore/edk2-platforms/blob/867efd012b2f2fa98854eaee896832e0ff2e52c4/Platform/RaspberryPi/Library/PlatformBootManagerLib/PlatformBm.c (visitato il 18/02/2021).

[117] *Raspberry Pi 4 UEFI Firmware RPi4.dsc*. URL: https://web.archive.org/web/20210219153925/https://github.com/tianocore/edk2-platforms/blob/867efd012b2f2fa98854eaee896832e0ff2e52c4/Platform/RaspberryPi/RPi4/RPi4.dsc (visitato il 19/02/2021).

[118] *Raspberry Pi 4 UEFI Firmware SsdtThermal.asl*. URL: https://web.archive.org/web/20210301103601/https://github.com/tianocore/edk2-platforms/blob/867efd012b2f2fa98854eaee896832e0ff2e52c4/Platform/RaspberryPi/AcpiTables/SsdtThermal.asl (visitato il 01/03/2021).

[119] *Raspberry Pi Firmware*. URL: https://web.archive.org/web/20210202084852/https://github.com/raspberrypi/firmware (visitato il 12/02/2021).

[120] *Reduced instruction set computer*. URL: https://web.archive.org/web/20210114023849/https://en.wikipedia.org/wiki/Reduced_instruction_set_computer (visitato il 22/01/2021).

[122] *Salt*. URL: https://web.archive.org/web/20210201183618/https://github.com/saltstack/salt (visitato il 01/02/2021).

[125] *Terraform*. URL: https://web.archive.org/web/20210110180612/https://www.terraform.io/ (visitato il 01/02/2021).

[126] *Terraform metalLB module*. URL: https://web.archive.org/web/20210217153656/https://registry.terraform.io/modules/colinwilson/metallb/kubernetes/latest (visitato il 17/02/2021).

[127] *Terraform Registry*. URL: https://web.archive.org/web/20210102091811/https://registry.terraform.io/browse/providers (visitato il 02/02/2021).

[128] *The NIST Definition of Cloud Computing*. URL: https://web.archive.org/web/20210113175843/https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (visitato il 19/01/2021).

[129] *Tinkerbell*. URL: https://web.archive.org/web/20210127182155/https://tinkerbell.org/ (visitato il 27/01/2021).

[130] *TinyWeb*. URL: https://web.archive.org/web/20210224185439/https://github.com/belyalov/tinyweb/tree/new_uasync_io (visitato il 22/02/2021).

[131] *UEFI ACPI FAQS*. URL: https://web.archive.org/web/20210126165936/https://www.uefi.org/faq (visitato il 26/01/2021).

[132] *Understanding the Differences Between X86 and ARM Processors*. URL: https://web.archive.org/web/20201201023349/https://www.techidence.com/understanding-the-differences-between-x86-and-arm-processors/ (visitato il 22/01/2021).

[133] *Unified Extensible Firmware Interface*. URL: https://web.archive.org/web/20210126094911/https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface (visitato il 26/01/2021).

[134] *Unified Extensible Firmware Interface (UEFI) Specification*. URL: https://web.archive.org/web/20210112043544/https://uefi.org/sites/default/files/resources/UEFI%20Spec%202.8B%20May%202020.pdf (visitato il 26/01/2021).

[136] *Why do embedded systems need device tree while pcs don't?* URL: https://web.archive.org/web/20210125095150/https://unix.stackexchange.com/questions/399619/why-do-embedded-systems-need-device-tree-while-pcs-dont (visitato il 25/01/2021).

[137] *Why has CPU frequency ceased to grow?* URL: https://web.archive.org/web/20201125022507/https://software.intel.com/content/www/us/en/develop/blogs/why-has-cpu-frequency-ceased-to-grow.html (visitato il 15/01/2021).

[138] *Why was the PowerPC architecture unable to keep up with Intel x86?* URL: https://www.quora.com/Why-was-the-PowerPC-architecture-unable-to-keep-up-with-Intel-x86 (visitato il 22/01/2021).

[139] *x86*. URL: https://web.archive.org/web/20210116124406/https://en.wikipedia.org/wiki/X86 (visitato il 22/01/2021).

[140] *x86-64*. URL: https://web.archive.org/web/20210115184205/https://en.wikipedia.org/wiki/X86-64 (visitato il 22/01/2021).