POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Definizione e prototipazione di servizi auto-orchestranti su infrastrutture 5G

Relatore
prof. Fulvio Risso

Supervisore:
ing. Gabriele Castellano

Candidato
Emanuele Fia

Dicembre 2017

A tutte le persone che hanno contribuito
al raggiungimento di questo risultato.

Indice

Elenco delle figure

Capitolo 1

Introduzione

I fornitori di servizi (cosiddetti *over the top* come YouTube, Netflix, Facebook, ecc.) oggi si trovano in un mercato fortemente competitivo, dove gli utenti possono facilmente rivolgersi alla concorrenza. Questo crea l'esigenza di fornire servizi sempre più efficienti (che consumano le risorse strettamente necessarie al fine di ridurre i costi operativi), e che riescono a limitare le interruzioni e le riduzioni della qualità, al fine di evitare la perdita di quote di mercato.

La rapida introduzione di tecnologie come l'*Internet of Things* in diversi settori (industria, città, abitazioni, automobili, sanità, agricoltura, ecc.), la progressiva diffusione dello streaming ad alta risoluzione e la necessità di spostare enormi quantità di dati (ad esempio il trasferimento di un genoma da un ospedale verso un laboratorio analisi) richiederanno modifiche sostanziali nelle infrastrutture attuali.

Dalle attività nei vari enti di Standardizzazione internazionale (3GPP, ITU, IEEE, ecc.) e dai movimenti di mercato emerge che i cambiamenti che verranno introdotti dalle reti 5G non si limiteranno ad un aumento delle prestazioni [1] delle reti di accesso wireless (in termini di banda e latenza). Si prevede un nuovo ecosistema che faciliterà la creazione di servizi end-to-end e introdurrà ulteriore flessibilità nelle infrastrutture.

La flessibilità è già stata parzialmente implementata nelle infrastrutture di cloud computing (ad esempio la possibilità di creare e rimuovere macchine virtuali con facilità) ma al momento non è ancora disponibile nelle reti di accesso e geografiche, dove generalmente i tempi di provisioning sono lunghi.

Si prevede pertanto che le tecnologie *Software Defined Networks* (SDN), *Network Function Virtualization* (NFV) e *Software Defined Radio* (SDR) avranno un ruolo fondamentale nell'introduzione di elasticità nelle infrastruttura. Gli operatori avranno una gestione più precisa dell'infrastruttura (tramite software di controllo che implementeranno le politiche desiderate) ma avranno anche la possibilità di esporre delle *Application Programming Interface* (API) verso i loro clienti che potranno, entro certi limiti, riconfigurare rapidamente i servizi offerti dall'infrastruttura.

É previsto un importante incremento di dispositivi ai bordi della rete e che potranno richiedere diverse necessità tecnologiche (banda, consumi, latenza, ecc.). Un'importante innovazione che probabilmente verrà introdotta è il Fog computing [2] che prevede di inserire nodi di elaborazione in prossimità dei nodi di accesso alle reti (central office o base station).

L'aggiunta di nodi di elaborazione al confine dell'infrastruttura può introdurre alcune problematiche rispetto alla soluzione centralizzata dei datacenter come ad esempio la gestione dell'alimentazione e delle relative precauzioni (oltre alla necessità di ridurre l'impatto ambientale [3]) e l'aumento dei tempi di intervento dovuti alla distribuzione geografica dei dispositivi. Queste difficoltà spingeranno gli operatori a limitare al minimo la capacità di calcolo disponibile nei nodi periferici e quindi condivideranno la necessità dei fornitori di servizi di ridurre lo spreco di risorse.

Questa tesi propone un nuovo paradigma che ha l'obbiettivo di migliorare la qualità dei servizi ospitati su di un'infrastruttura attraverso una gestione collaborativa delle risorse tra i due soggetti. L'attuale mancanza di interazioni tra infrastruttura e servizio e la volontà di mantenere il più possibile semplici le implementazioni di queste due entità ha portato l'utilizzo di un paradigma chiamato *autoscaling* che compensa la sua facile implementazione con la sua poco accurata distribuzione delle risorse ai servizi. Questa gestione approssimativa comporta uno consumo inutile delle risorse che implica un aumento dei costi operativi.

L'elaborato è strutturato come segue:

- **Capitolo 2:** Presenta le tecnologie attualmente utilizzate per gestire le risorse dei servizi in un'infrastruttura, elenca i loro principali problemi e infine viene spiegato l'obbiettivo della tesi.
- **Capitolo 3:** Descrive le tecnologie più importanti utilizzate.
- **Capitolo 4:** Illustra la soluzione proposta da questa tesi, spiegando l'architettura dei servizi auto-orchestranti.
- **Capitolo 5:** Analizza le parti più importanti del prototipo sviluppato per effettuare la validazione
- **Capitolo 6:** Mostra le prove compiute per validare l'architettura proposta e riporta i risultati ottenuti.
- **Capitolo 7:** Espone le conclusioni e i possibili sviluppi futuri.

Capitolo 2

Stato dell'arte

In questo capitolo è descritta la principale strategia attualmente utilizzata dagli orchestratori per gestire l'aggiunta e la rimozione dinamica di risorse nei servizi in esecuzione su di un'infrastruttura. Sono inoltre elencati i principali problemi dell'attuale soluzione e l'obiettivo di questa tesi.

2.1 Autoscaling

La quantità di risorse utilizzate da un servizio dipendono generalmente dal suo carico di lavoro, che può variare nel corso del tempo. Associare in modo statico e univoco delle risorse ad un servizio è controproducente per vari motivi. Le risorse assegnate rimarrebbero fruibili esclusivamente dal servizio associato che dovrebbe pagarle anche quando non sono utilizzate.

Per risolvere questo problema al momento viene utilizzata una metodologia chiamata *autoscaling*. L'autoscaling si divide in due tipologie: verticale e orizzontale. In entrambi i casi l'orchestratore controlla periodicamente dei parametri generici disponibili a livello infrastrutturale che indicano utilizzo delle risorse assegnate (ad esempio l'utilizzo di CPU di una macchina virtuale) e al verificarsi di alcune situazioni ritenute critiche (ad esempio il superamento di una determinata soglia) vengono eseguite alcune azioni generiche. Nel caso di autoscaling verticale, queste azioni generalmente prevedono l'incremento o la rimozione di una risorsa (ad esempio vengono incrementate le CPU disponibili ad una VM). L'autoscaling orizzontale invece prevede la creazione di un'ulteriore istanza che lavorerà in parallelo con l'istanza principale. Questa seconda ipotesi implica che il servizio sia predisposto a lavorare con più istanze attive contemporaneamente, eseguite in parallelo.

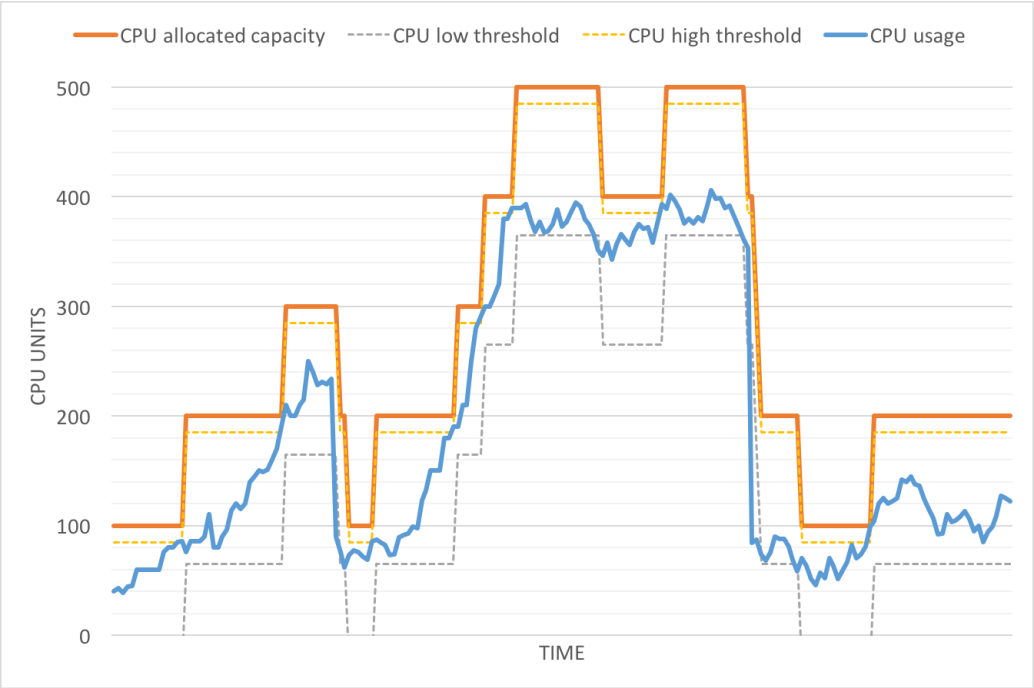

Figura 2.1. Esempio di risorse utilizzate dall'autoscaling.

2.2 Vantaggi e svantaggi dell'autoscaling

Il principale vantaggio dell'autoscaling è sicuramente la sua facile implementazione e configurazione. L'implementazione di questa metodologia va fatta esclusivamente nell'infrastruttura ed è completamente trasparente al servizio. Per configurare l'autoscaling è sufficiente definire i parametri da monitorare (utilizzo di CPU, RAM, ecc..), le soglie (ad esempio l'utilizzo della CPU al 50% per più di 10 secondi), le azioni da intraprendere (ad esempio incrementare i core CPU disponibili di una unità) ed eventualmente dei limiti (ad esempio fino ad un massimo di 4). Questi valori possono essere associati ad offerte commerciali che prevedono limiti massimi maggiori a fronte di un costo maggiore.

Il punto di forza dell'autoscaling è la sua generalità, che però risulta essere contemporaneamente un vantaggio e uno svantaggio. É da considerare un vantaggio perché la configurazione è sicuramente più semplice, così come la sua l'implementazione. É uno svantaggio perché la mancata considerazione della semantica del servizio implica il monitoraggio di parametri generici che, per loro natura, descrivono in modo approssimativo le necessità del servizio.

L'impossibilità di controllare parametri più indicativi (ad esempio il numero di utenti che attualmente connessi) è dovuta alla mancanza di comunicazione tra l'infrastruttura e il servizio e alla volontà di mantenere semplice l'implementazione

dell'autoscaling. Tuttavia, questa limitazione impedisce all'infrastruttura di capire in modo accurato lo stato e le risorse necessarie al servizio, che devono essere quindi dedotte dai parametri generici.

Esiste inoltre l'impossibilità, da parte dell'infrastruttura, di informare il servizio circa la non disponibilità di una o più risorse necessarie e di notificare la presenza di alcuni malfunzionamenti (ad esempio latenza elevata tra due punti, ecc..). La mancanza di questa interazione impedisce al servizio di adattarsi alle criticità riscontrate dall'infrastruttura.

2.2.1 Primo esempio: autoscaling basato su parametri generici

Supponiamo di essere il fornitore di un servizio che consente di convertire il formato di alcuni file. Vogliamo inoltre garantire un tempo massimo per la conversione di 1 ora (SLA). Il servizio viene realizzato tramite un software in esecuzione in una virtual-machine istanziata su di un'infrastruttura che supporta l'autoscaling. Il software che si occupa della conversione dei file è stato programmato in modo da utilizzare tutta la CPU disponibile al fine di velocizzare l'elaborazione.

Alla ricezione della prima richiesta di conversione, il software inizia a utilizzare tutta la CPU e l'infrastruttura di conseguenza reagisce aumentando ripetutamente la CPU disponibile fino a raggiungere un limite predefinito. Questi ripetuti incrementi di CPU velocizzano in modo significativo il processo di conversione. Questo però potrebbe non essere il comportamento desiderato. L'obbiettivo è fornire un servizio che rispetta gli SLA definiti che però non sono noti all'infrastruttura. Il software del servizio lavora alla massima velocità possibile perché non conosce lo stato dell'infrastruttura (ad esempio le risorse attuali potrebbero ad un certo punto diminuire o aumentare). L'infrastruttura a sua volta, non conosce le esigenze del servizio (SLA): non sa quante conversioni sono in corso, quando sono iniziate, entro quanto devono finire e come influiscono le risorse disponibili sulle conversioni in corso.

Il caso in esempio può essere esteso prevenendo due tipologie di utenti: free e premium. Quelli classificati come free non pagano nessun abbonamento ma non hanno SLA mentre quelli premium pagano un abbonamento più alto e richiedono uno SLA di 30 minuti. In questo caso, al fine di rispettare gli SLA degli utenti premium, il servizio potrebbe essere interessato anche a pagare un compenso extra all'infrastruttura (per ottenere le risorse necessarie al completamento delle attività nei tempi richiesti) rispetto al caso degli utenti free dove si vuole risparmiare economicamente il più possibile. Quest'ultima logica non può essere implementata con gli orchestratori attuali perché non riuscirebbero ad accedere e interpretare lo stato del servizio.

2.2.2 Secondo esempio: mancata collaborazione del servizio con l'infrastruttura

Consideriamo un servizio composto esclusivamente da una macchina virtuale che ospita un webserver con il supporto alla compressione delle risposte HTTP. La macchina virtuale del servizio considerato viene istanziata su di un Host dove sono già in esecuzione altre macchine virtuali. In un determinato momento una macchina virtuale legata ad un altro servizio con priorità maggiore inizia a consumare tutta le risorse CPU disponibili sulla macchina Host.

Contemporaneamente, il server web del servizio che stiamo considerando, inizia a ricevere un elevato numero di richieste HTTP con compressione che saturano la CPU disponibile. L'autoscaling non riesce ad aumentare le risorse disponibili perché le risorse CPU sull'host sono esaurite. Quello che succede dunque è una degradazione della qualità del servizio offerto perché le richieste vengono soddisfatte in un tempo maggiore.

Se esistesse una collaborazione tra i servizi e l'infrastruttura, diverse potrebbero essere le soluzioni a questa problematica. Si potrebbe richiedere al servizio che sta consumando la maggior parte delle risorse sull'host di posticipare o dilazionare le attività nel tempo (consideriamo come attività ad esempio la compressione dei file di log o comunque qualsiasi attività non real-time di tipo batch), oppure si potrebbe chiedere al web-server di disattivare la compressione HTTP visto che l'infrastruttura sta affrontando una criticità. La qualità del servizio erogata sarà leggermente inferiore (ci sarà un maggior consumo di banda) ma complessivamente accettabile.

2.3 ContainerPilot

ContainerPilot [4] è un orchestratore che si occupa di gestire il ciclo di vita delle applicazioni basate sull'architettura a microservizi. I principali problemi che questo software vuole affrontare riguardano la risoluzione delle dipendenze tra microservizi e la necessità di far scalare agevolmente le applicazioni. ContainerPilot utilizza i servizi offerti da Consul [5] che è sostanzialmente un database sviluppato per facilitare le operazioni di *Service Discovery*.

Il *Service Discovery* è un pattern largamente utilizzato che consente a un microservizio di notificare a tutti gli altri la sua disponibilità. Questo avviene tramite l'invio di un messaggio di registrazione al server che si occupa di gestire il catalogo dei servizi disponibili (in questo caso specifico Consul). Gli altri microservizi dovranno interrogare il catalogo per ottenere l'elenco di quali servizi sono disponibili e di chi li offre. Esistono due strategie distinte per effettuare questa interrogazione: passiva o attiva.

La strategia passiva può essere implementata in due modi: tramite delle richieste DNS o tramite un proxy. Ne primo caso il microservizio viene configurato per

utilizzare i servizi di un determinato nome DNS, che verrà risolto da un server DNS opportunamente modificato in modo da interrogare il catalogo dei servizi. Nel caso del proxy invece, il microservizio è configurato per utilizzare il nome DNS del proxy e quest'ultimo andrà a controllare il catalogo e inoltrerà le richieste al microservizio opportuno. In entrambi i casi il microservizio non dovrà interrogare direttamente il catalogo per trovare i servizi disponibili e non dovrà gestire il loadbalancing, quindi la sua implementazione sarà sicuramente più semplice.

Queste due soluzioni però presentano alcuni problemi: i proxy aggiungono un nuovo *point of failure* nell'architettura e introducono della latenza nelle conversazioni tra i microservizi, i server DNS invece rendono più complicata una gestione corretta del loadbalancing.

L'approccio attivo invece prevede d'inglobare il *Service Discovery* e la gestione del loadbalancing direttamente dentro il microservizio. Questo elimina il problema della latenza introdotta dai proxy e del 'point of failure" ma introduce della complessità aggiuntiva all'interno del microservizio. Spostare questa logica può aiutare a migliorare le performance complessive dell'applicazione grazie alle nuove informazioni che il microservizio può sfruttare. ContainerPilot si propone come un software in grado di facilitare l'implementazione di servizi con *Service Discovery* attivo.

ContainerPilot va installato all'interno dei microservizi e si comporta come se fosse il software "init", avviando tutti i processi necessari per far funzionare il microservizio. Tramite un file di configurazione è possibile specificare quali applicativi lanciare all'avvio, come controllare il loro stato di salute (*healthy*, *unhealthy*), cosa fare quando lo stato diventa *unhealthy*, quali sono le dipendenze del microservizio e quali altri cambiamenti nel catalogo sono rilevanti per il microservizio. Le modifiche individuabili nel catalogo possono essere solo di tre tipi: *healthy* se il servizio era precedentemente in uno stato *unknown* o *unhealthy* ed è diventato *healthy*, *unhealthy* se il servizio è diventato *unhealthy* o irraggiungibile oppure *changed* se il servizio ha cambiato stato o è cambiato l'elenco delle istanze disponibili.

Tramite questo file di configurazione quindi è possibile eseguire degli script che aggiornano la configurazione di un software quando si verificano degli eventi (ad esempio cambia l'elenco dei microservizi che implementano un servizio). Un esempio pratico può essere il caso di un loadbalancer HTTP che deve aggiornare la sua configurazione quando vengono aggiunti o tolti dei server di backend.

2.3.1 Similitudini con il lavoro svolto in questa tesi

L'approccio utilizzato da ContainerPilot sotto alcuni aspetti assomiglia alla strategia proposta in questa tesi ovvero tentare di aumentare la consapevolezza dei servizi al fine di permettergli di trovare una configurazione globalmente migliore.

Ad esempio, con alcune piccole modifiche, si potrebbe inserire una segnalazione dall'infrastruttura quando quest'ultima si trova nella condizione di dover rimuovere

delle risorse a un servizio.

Questo potrebbe essere gestito tramite ContainerPilot, prevedendo una notifica dell'infrastruttura verso Consul per avvisarlo che un microservizio sarà presto spento e dunque va rimosso dal catalogo. In questo modo tutti i microservizi che lo hanno come dipendenza possono riconfigurarsi per non considerarlo. Questo tipo approccio risulta comunque limitato perché impedisce al servizio d'influenzare l'infrastruttura (ad esempio quale microservizio spegnere). A fronte di un cambiamento ContainerPilot può eseguire degli script dove quindi è possibile inserire della logica aggiuntiva (ad esempio cambiare la configurazione dell'applicazione controllata da ContainerPilot in base al numero delle istanze di un altro microservizio).

2.4 Obiettivo della tesi

Questa tesi propone un nuovo paradigma collaborativo tra infrastruttura e servizi con l'intenzione di rimuovere le limitazioni e le problematiche della metodologia basata su autoscaling.

Le principali innovazioni sono:

- Permettere una più precisa gestione delle risorse.
- Permettere all'infrastruttura di notificare al servizio eventuali criticità (ad esempio l'aumento di latenza tra due punti della rete oppure la carenza di una determinata risorsa) al fine di permettere al servizio di adattarsi.
- Permettere una decisione collaborativa tra infrastruttura e servizio nelle operazioni da effettuare al seguito di una criticità (ad esempio scegliere se scalare orizzontalmente o verticalmente).

Gli aspetti di diversità tra l'approccio scelto in questa tesi e quello di ContainerPilot sono:

- l'approccio utilizzato da ContainerPilot è completamente distribuito e non considera il servizio nella sua interezza.
- viene considerata esclusivamente l'affidabilità e la configurazione del servizio e non vengono esaminate le risorse utilizzate.
- non sono previste interazioni o collaborazioni con l'infrastruttura.

Nei successivi capitoli vengono illustrati i vantaggi di questo nuovo metodo e le problematiche che introduce. Infine viene proposta un'architettura prototipale realizzata allo scopo di controllare se effettivamente questo nuovo paradigma introduca vantaggi apprezzabili e per verificare l'eventuale presenza di problemi.

Capitolo 3

Strumenti utilizzati

In questo capitolo vengono illustrate le principali tecnologie e i software utilizzati per definire l'architettura e sviluppare il prototipo di questa tesi.

3.1 Universal Node

L' Universal Node [6] fornisce le stesse funzionalità degli orchestratori di infrastrutture come OpenStack [7] o Kubernetes [8] ma si limita al controllo di un singolo dispositivo. Può essere utilizzato per verificare sia le capacità di elaborazione (computing) che le primitive di networking.

L'Universal Node si basa sul formalismo del Network Functions Forwarding Graph (NFFG) [9]. Questo grafo permette d'indicare un elenco di Virtual Network Function (VNF) da eseguire e le specifiche su come inoltrare i dati tra le VNF tramite delle flowrule simili a quelle definite nel protocollo OpenFlow [10]. Il software include inoltre un server HTTP utilizzato per esporre un'interfaccia REST. Al momento, tramite queste API, è possibile istanziare,aggiornare o eliminare gli NFFG dall'Universal Node.

Il *Compute manager* riceve l'elenco delle VNF da eseguire, mentre il *Network manager* riceve l'elenco delle flowrule da realizzare. Per la gestione del computing è possibile scegliere tra diversi backend: macchine virtuali con KVM [11], container con Docker [12] oppure processi DPDK [13]. Per le funzionalità di networking è invece possibile scegliere tra OpenVSwitch [14], xDPd [15] e ERFS [16].

Nel momento in cui l'Universal Node decide di avviare una determinata VNF procede nel seguente modo: se nella descrizione è indicato un template, questo viene scaricato dal *Datastore*. Il template contiene diverse informazioni tra cui: le risorse richieste dalla VNF (come CPU, RAM, ecc.), il backend da utilizzare (KVM, Docker, ecc.) e l'immagine del disco da utilizzare per avviare la macchina virtuale oppure il Dockerfile. Diversamente, se il template non è indicato nella VNF, deve essere necessariamente indicata la *functional_capability*. In questo caso l'Universal Node

scarica tutti i template che implementano quella determinata *functional_capability* e tramite delle logiche interne ne sceglie una.

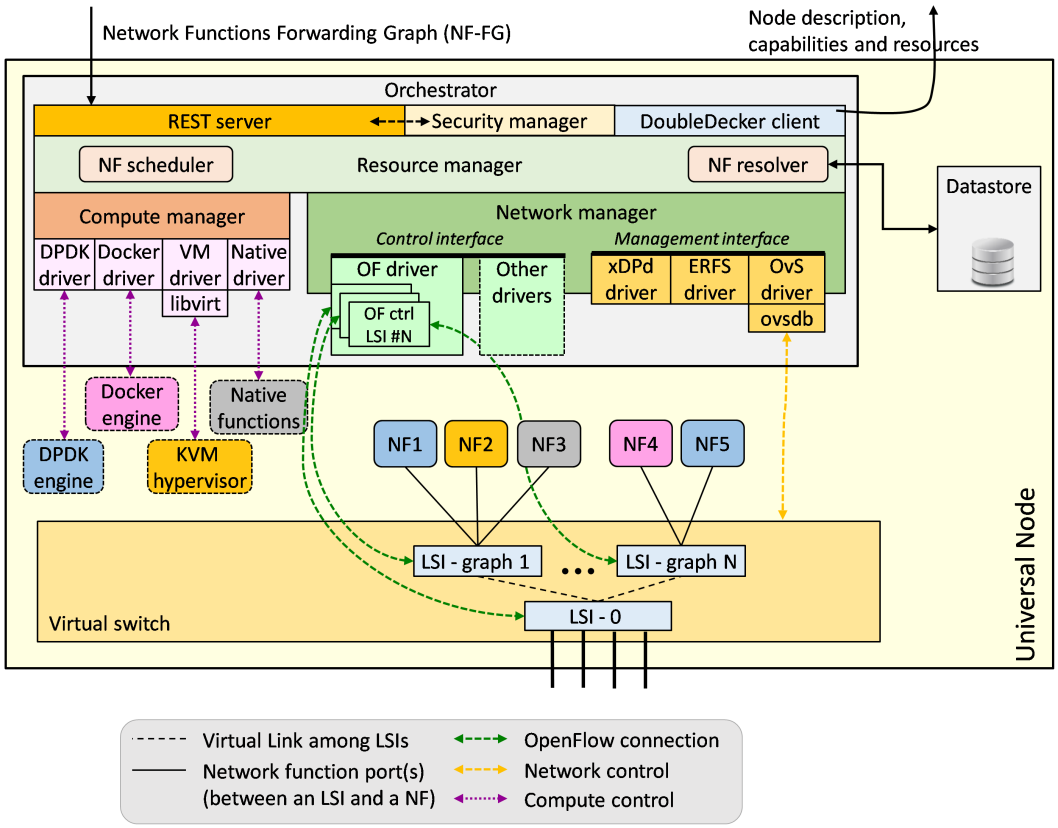

Figura 3.1. Architettura dell'Universal Node (Fonte: Universal Node github repository [6]).

3.2 KVM

KVM [11] (Kernel-based Virtual Machine) è un modulo per la virtualizzazione disponibile all'interno del kernel Linux dalla versione 2.6. É sostanzialmente composto da un modulo che permette di sfruttare le estensioni dedicate alla virtualizzazione dell'hardware x86 presenti nei processori moderni(Intel VT o AMD-V). La comunicazione con il modulo da parte delle applicazioni user-space avviene tramite un device virtuale (/dev/kvm). KVM consente di eseguire nativamente diverse macchine virtuali con diversi sistemi operativi senza la necessità di effettuare alcuna modifica (purché siano compatibili con l'architettura x86).

3.3 QEMU

QEMU [17] (Quick EMUlation) è un software di emulazione che consente di eseguire sistemi operativi sviluppati per determinate piattaforme su altre (ad esempio eseguire sistemi operativi sviluppato per ARM su piattaforme x86) tramite la tecnica dynamic binary translation [18]. QEMU consente di emulare, oltre alla CPU, anche tutto l'hardware connesso a un'architettura standard (dischi, schede di rete, interfacce grafiche, ecc.). QEMU può funzionare anche senza KVM ma le performance, in questo caso, possono essere molto scarse. Utilizzandolo invece, QEMU riesce a sfruttare direttamente le primitive di virtualizzazione presenti nel kernel evitando l'emulazione della CPU.

3.4 Libvirt

Libvirt [19] rappresenta un insieme di software che permettono una più facile gestione degli ambienti di virtualizzazione. Diverse sono le piattaforme supportate: KVM, Xen, VirtualBox, LXC, VMWare, ecc...

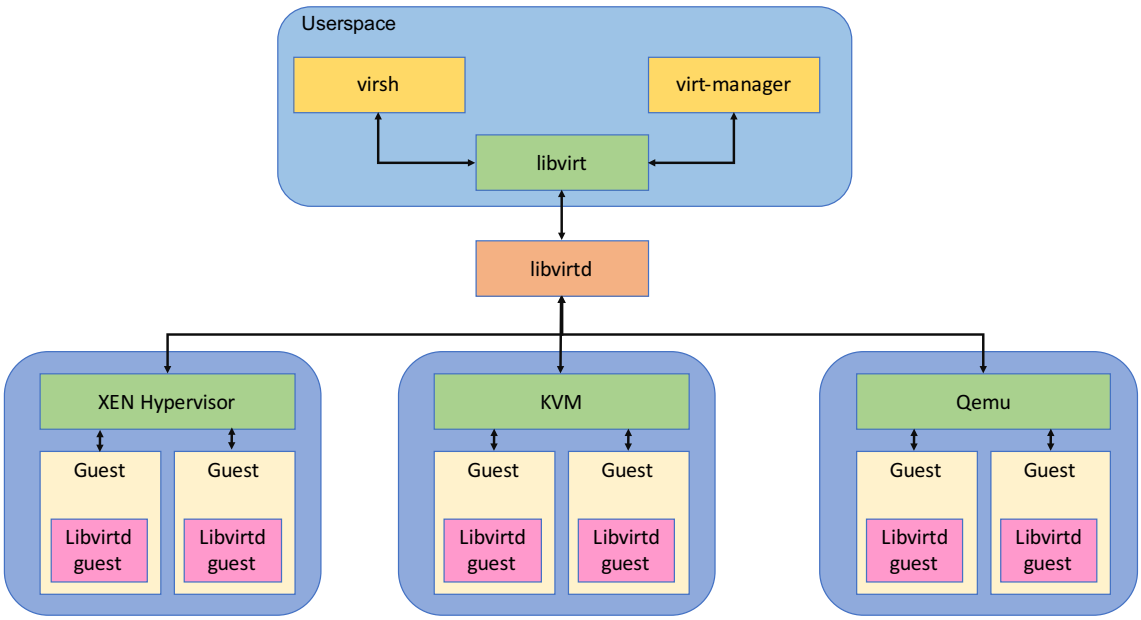

Figura 3.2. Architettura di libvirt.

La suite di strumenti messi a disposizione di libvirt sono: un demone (libvirtd) installato sulla macchina host, un'interfaccia a linea di comando (virsh), un insieme di librerie disponibili per diversi linguaggi di programmazione (c, c++, Python, PHP, ecc.) e un demone (libvirtd-guest) da installare nelle macchine guest per

poter effettuare delle modifiche nelle VM senza doverle riavviare. L'obbiettivo di libvirt è realizzare un'interfaccia unica e indipendente per il controllo delle macchine virtuali.

3.5 Configuration Orchestrator e Agent

Il Configuration Orchestrator [20] e il Configuration Agent [21] sono i due componenti utilizzati per la gestione della configurazione e dello stato delle VNF. L'orchestratore è istanziato una volta sola mentre l'Agent viene inserito all'interno di ogni Virtual Network Function.

Il principale vantaggio di questa architettura è che garantisce uno stato di astrazione tra la configurazione e lo stato di una VNF e la sua implementazione. Ad ogni VNF è associata una funzionalità di rete (ad esempio NAT, Firewall, DHCP server, ecc.) e a ognuna di queste corrisponde un modello YANG [22] che descrive la loro configurazione e il loro stato.

I compiti dell'Agent sono: notificare l'avvio della VNF tramite il message bus DoubleDecker [23], ricevere la configurazione generica e convertirla in una comprensibile al software della Virtual Network Function e viceversa. Il Configuration Orchestrator invece riceve, tramite il message bus, i messaggi che indicano l'avvio di una nuova VNF, gli fornisce una configurazione iniziale e espone delle API per consentire ad applicativi esterni di modificare e interrogare la configurazione e lo stato delle VNF.

Purtroppo questa architettura prevede anche una modifica all'interno dell'orchestratore che avvia le VNF. Infatti è necessario inserire dentro il disco delle Virtual Network Function alcuni file per consentire all'Agent di avviarsi: le chiavi di DoubleDecker, l'indirizzo IP del message broker e alcune informazioni che caratterizzano la VNF. Questi verranno utilizzati dall'Agent per connettersi al message bus e notificare l'avvio della Virtual Network Function al Configuration Orchestrator.

3.6 YANG, PyAng, JSON schema plugin e Jsonschema2pojo

Il linguaggio di modellazione YANG [22] è stato sviluppato nell'ambido della definizione del protocollo NETCONF (network configuration protocol) [24]. É stato ideato per descrivere la configurazione e lo stato di componenti di rete ma può essere utilizzato anche in altri ambiti, grazie alla sua elevata flessiblità nella definizione dei tipi di dato. YANG rappresenta i dati tramite una struttura ad albero e consente di codificarli in qualunque modo (ad esempio JSON o XML). Sono presenti dei tipi di dato predefiniti ma è possibile aggiungerne altri, anche complessi (ad esempio il tipo mac-address o il tipo indirizzo IPv6).

Nel prototipo realizzato in questa tesi, è stato necessario generare delle classi Java corrispondenti agli oggetti JSON utilizzati dal Configuration Service per riuscire ad interagire con le configurazioni e lo stato delle VNF. Gli oggetti JSON seguono i criteri definiti in un modello YANG. Al fine di procedere alla generazione automatica di queste classi è stato scelto di utilizzare Pyang [25], software generalmente usato per validare i modelli YANG, e il suo plugin JSON Schema [26] per generare uno schema JSON equivalente al modello YANG. Infine dallo schema JSON sono state generate le classi Java tramite il programma jsonschema2POJO [27].

3.7 Wowza

Wowza [28] (Wowza Streaming Engine) è il software di streaming utilizzato per la validazione dell'architettura proposta in questa tesi. Il software consente di fare il setup di un servizio di streaming VOD (Video on Demand) o live con estrema facilità.

Tutta la configurazione avviene tramite un portale web dove è possibile configurare diverse applicazioni (canali), ognuna con una configurazione indipendente dalle altre. Wowza include un transcoder che permette il transrate (cambio del bitrate di un flusso audio o video) e il transcoding (cambio del codec).

Diversi sono i codec disponibili dentro Wowza (Video: H265, H264, VP9, VP8, VP6, Audio: AAC, MP3, Speex, Opus, Vorbis) e molti sono i protocolli di streaming supportati (RTMP, HDS, HLS, MPEG-DASH e RTMP/RTP).

Inoltre il software mette a disposizione delle API per Java e delle REST API che consentono di modificare la configurazione, d'interrogare lo stato attuale delle trasmissioni e di ottenere alcune statistiche.

3.8 VLC e LibVLC

VLC [29] è un riproduttore multimediale basato sulla libreria libVLC [30]. Sia il media player che la libreria fanno parte di un progetto open-source e sono completamente gratuiti. VLC è in grado di riprodurre un elevato numero di formati, audio e video e riesce inoltre a strutturare le accelerazioni fornite da molti hardware. É disponibile per i sistemi operativi Windows, MacOS, Linux, Android e iOS.

VLC può essere utilizzato come client per poter ascoltare o visualizzare un flusso audio o video proveniente da un file, da un dispositivo (come ad esempio una webcam) o da un flusso di rete. E' possibile utilizzare questo software anche per avviare delle trasmissioni su protocolli unicast o multicast e dispone inoltre anche di un modulo di transcodifica.

In questa tesi è stato utilizzato per creare un piccolo applicativo per misurare la qualità di una trasmissione in live streaming.

Capitolo 4

Servizi auto-orchestranti

In questo capitolo viene proposto un nuovo paradigma per una gestione cooperativa delle risorse tra l'infrastruttura e i servizi.

Al fine di risolvere i problemi trattati nel Capitolo 2 è necessario introdurre delle interazioni tra l'infrastruttura e il servizio. La soluzione proposta in questa tesi prevede un nuovo componente, inserito tra i due soggetti, che andrà a risolvere le situazioni problematiche individuate.

4.1 Premessa

In questa sezione vengono inserite alcune definizioni che verranno poi utilizzate successivamente.

Le architetture a microservizi[31] sono attualmente quelle più usate per lo sviluppo dei servizi e perciò, nella tesi, è sottinteso l'utilizzo di questo paradigma. Questa architettura prevede che le applicazioni siano estremamente distribuite (invece di essere monolitiche), composte da una serie di entità che comunicano tra di loro che si chiamano microservizi e che sono specializzati in determinate attività. Il primo vantaggio di questo paradigma è la possibilità di scalare più velocemente (aumentando il numero di microservizi che compiono una determinata funzione). Inoltre facilita la manutenzione e l'aggiunta di nuove funzionalità (vanno modificati esclusivamente i microservizi coinvolti). I servizi monolitici possono essere comunque considerati come un applicazione composta da un unico microservizio.

Ognuna di queste entità può contenere una configurazione che permette di cambiare alcuni aspetto del microservizio(ad esempio abilitare o disabilitare la registrazione dei log in un webserver).

Ad ogni microservizio è inoltre associato uno *stato* che rappresenta l'insieme delle informazioni che descrivono in modo completo la sua situazione attuale.

L'infrastruttura è amministrata tramite un orchestratore che è sotto il controllo del operatore di rete. Nella situazione attuale l'orchestratore riceve una descrizione

del servizio da istanziare con eventualmente un elenco statico delle risorse necessarie. L'infrastruttura procede autonomamente nella scelta di dove istanziare i microservizi e come amministrare le risorse non garantite (nel caso di autoscaling, controllando periodicamente dei parametri generici).

4.2 Servizi auto-orchestranti

Con *servizio auto-orchestrante* intendiamo un servizio tradizionale arricchito di alcune funzionalità che consentono una gestione condivisa delle risorse con l'orchestratore dell'infrastruttura.

Questa gestione condivisa permette ad entrambe le parti di raggiungere la migliore configurazione possibile: il servizio può chiedere all'infrastruttura le risorse necessarie o altri cambiamenti mentre l'infrastruttura può richiedere variazioni al servizio (ad es. ridurre l'utilizzo di CPU). Con queste nuove funzionalità, al verificarsi di determinati eventi nei microservizi o nell'infrastruttura, il servizio può reagire con delle azioni specifiche.

I microservizi, oltre a notificare eventuali eventi nel caso di cambiamenti nel loro stato, possono anche accettare nuove configurazioni (ad es. ridurre la qualità del servizio offerto) provenienti dal modulo di auto-orchestrazione.

Di seguito vengono indicate le interazioni previste tra i vari componenti.

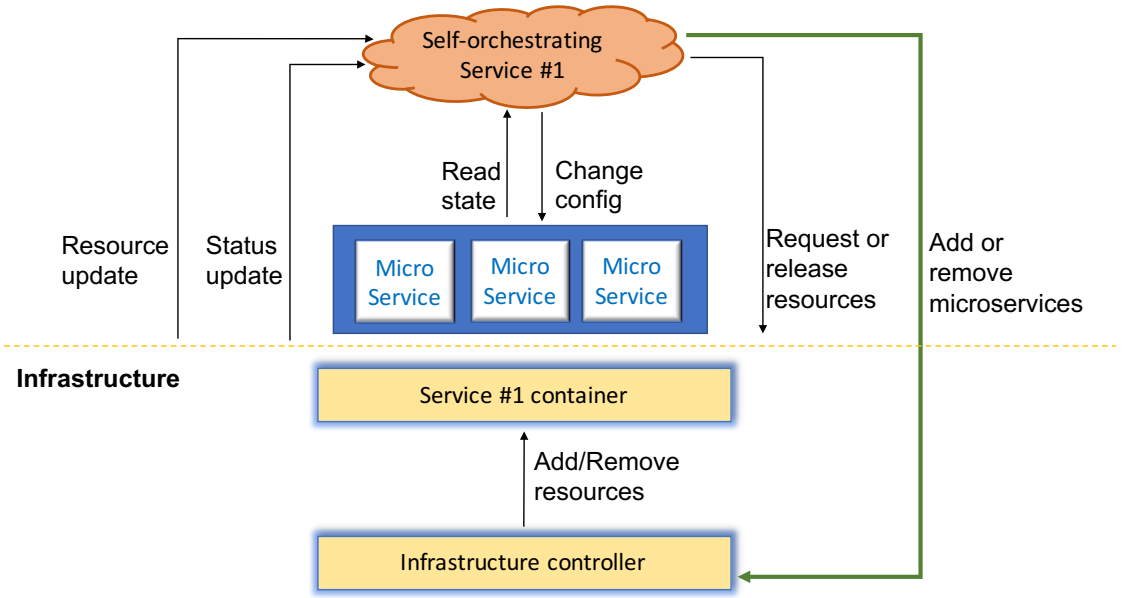

Figura 4.1. Servizi auto-orchestranti

4.2.1 Interazioni con i microservizi

Il modulo di auto-orchestrazione deve riuscire a leggere la configurazione e lo stato attuale dei microservizi perché queste informazioni vengono utilizzate per capire il loro funzionamento interno, i parametri caratteristici e le risorse necessarie ai vari microservizi per poter funzionare senza problemi. Deve anche essere prevista la possibilità da parte del modulo di poter modificare la configurazione dei microservizi per adattarli alle risorse attualmente disponibili (ad esempio disattivando la compressione delle richieste HTTP se tutte le CPU sono occupate).

4.2.2 Interazioni con l'infrastruttura

Il modulo di auto-orchestrazione può ricevere diverse informazioni dall'infrastruttura: queste verranno utilizzate per trovare la migliore configurazione possibile (considerando anche lo stato attuale del servizio).

Le interazioni principali che devono essere necessariamente disponibili sono quelle che consentono al modulo di auto-orchestrazione di aggiungere o rimuovere risorse o microservizi.

Opzionalmente, l'infrastruttura può anche fornire una topologia che può essere eventualmente virtualizzata per rispondere alla volontà degli operatori di nascondere dettagli implementativi o topologici. La topologia, anche se virtualizzata, permette al servizio di conoscere dettagli che potrebbero consentirgli di calcolare configurazioni ulteriormente ottimizzate.

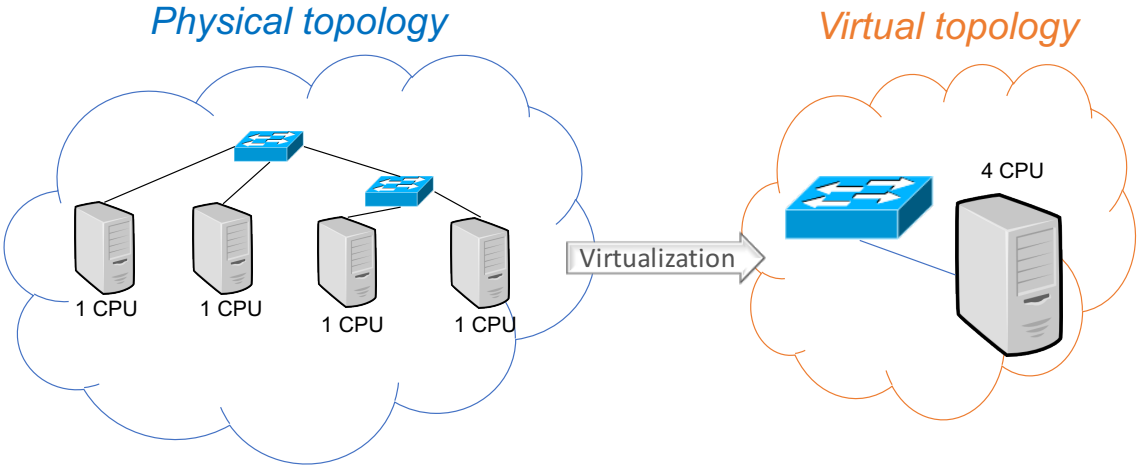

Figura 4.2. Differenze tra infrastruttura fisica e virtuale

Un esempio pratico di quanto questa informazione possa migliorare la configurazione si può trovare in un servizio che rileva temperature da una serie di sensori

collocati geograficamente in un'area limitata. In questo caso il modulo di auto-orchestrazione potrebbe decidere di avviare il microservizio di raccolta dati nel nodo di rete più vicino ai sensori (architettura Fog), anche se con capacità di calcolo inferiori, al fine di aumentare la resistenza ai guasti (ad esempio l'isolamento dei sensori dal software di raccolta). Una situazione analoga si può verificare quando c'è necessità di istanziare microservizi di backup.

L'aspetto più interessante di questo nuovo modo di gestire i servizi è che ciascuno di essi può influenzare molte decisioni dell'infrastruttura e ha un controllo più preciso delle risorse utilizzate e disponibili: può decidere quando aggiungere o rimuovere risorse ad un microservizio a seconda del suo stato (ad es. il numero di utenti connessi è aumentato) oppure può sostituire l'implementazione di un microservizio con un'altra (ad es. perché in alcune situazioni richiede meno risorse per poter funzionare).

L'infrastruttura analogamente può anche richiedere al modulo di auto-orchestrazione di rilasciare risorse allocate precedentemente perché richieste da un altro servizio con maggiore priorità. In questo caso il servizio potrà eventualmente riorganizzarsi per adeguarsi alla riduzione di risorse disponibili evitando interruzioni o eccessive perdite di qualità. Può anche essere previsto l'invio di eventi personalizzati dall'infrastruttura verso il modulo di auto-orchestrazione.

4.2.3 Modifiche richieste negli orchestratori delle infrastrutture

Introducendo le funzionalità di auto-orchestrazione in un servizio, alcune attività che prima erano svolte esclusivamente nell'orchestratore dell'infrastruttura vengono spostate nel servizio o gestite in modo condiviso.

L'orchestratore dell'infrastruttura deve continuare ad offrire delle API che permettano di istanziare, modificare e rimuovere servizi (che ad esempio possono essere descritti tramite Dockerfile, Kubernetes Deployment, NF-FG, ecc..), mentre viene mantenuto l'*autoscaling* esclusivamente per quei servizi che non implementeranno le funzionalità di auto-orchestrazione. L'infrastruttura quindi non deve più occuparsi del monitoraggio dei parametri generici dei servizi, perché questa mansione viene effettuata all'interno dal servizio stesso.

Nell'infrastruttura viene aggiunto un nuovo componente denominato "gestore delle risorse". Questo modulo si occupa di amministrare le risorse disponibili all'interno dell'infrastruttura. Le principali attività che dovrà svolgere sono:

- ricevere dall'infrastruttura l'elenco delle risorse disponibili, che possono variare nel tempo (ad esempio perché nell'infrastruttura vengono aggiunti nuovi nodi)
- comunicare (facoltativamente) ai servizi le risorse disponibili e la topologia dell'infrastruttura

- verificare se, le richieste provenienti dai servizi di aggiunta o rimozione di risorse, siano legittime (ad esempio, può un determinato servizio richiedere altre 10 CPU?) e decide se accettarle o rifiutarle (in parte o in toto)

- comunicare con l'infrastruttura per assicurarsi che effettivamente il servizio possa utilizzare al massimo le risorse che gli sono state concesse (ad esempio, limitando la quantità di RAM accessibile oppure inserendo delle code sulle interfacce di rete)

- eventualmente risolvere i conflitti dovuti a richieste concorrenti

- gestire le diverse priorità dei servizi (ad esempio nel caso in cui un servizio ad alta priorità richieda una risorsa che non è disponibile, il modulo potrà richiedere ad un servizio con priorità inferiore di rilasciarla).

Il modulo, al fine di riuscire ad implementare le attività riportate nel precedente elenco, sarà implementato seguendo le esigenze dell'operatore di rete. Se l'infrastruttura è utilizzata per un servizio commerciale del tipo *IaaS* (Infrastructure as a service) il gestore delle risorse dovrà interagire con uno o più componenti al fine di capire quali priorità assegnare, quali risorse devono essere accessibili e quali sono i limiti massimi delle risorse assegnabili.

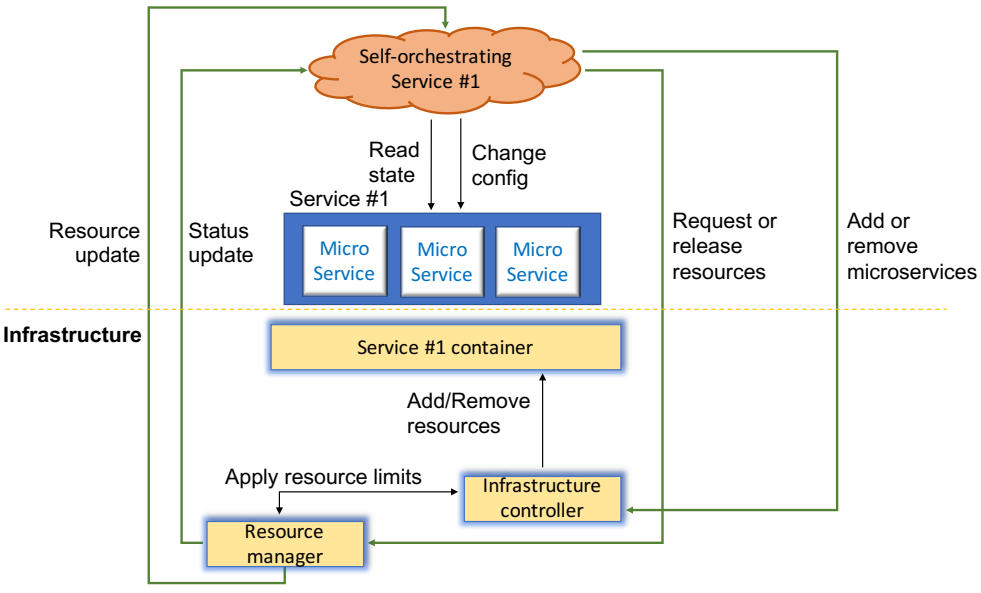

Figura 4.3. Orchestratore infrastruttura con gestore risorse

4.3 Moduli di auto-orchestrazione

Nell'ambito di questa tesi sono state valutate due modalità per aggiungere le funzionalità di auto-orchestrazione ad un servizio tradizionale: *Embedded* o *Auto-generated*.

4.3.1 Servizio con auto-orchestratore embedded

L'approccio più immediato per aggiungere la funzionalità di auto-orchestrazione ad un servizio è quello che prevede di inserirla direttamente nel codice sorgente del servizio originale.

Il principale vantaggio di questa implementazione è la sua facilità di realizzazione. Ignorando le difficoltà tecnologiche, all'interno del software di un servizio è relativamente facile capire quando sono necessarie nuove risorse, quando non lo sono più e quando e come cambiare il suo comportamento al fine di adattarlo alle risorse disponibili.

Questo approccio ovviamente contiene molti aspetti negativi: il più importante è sicuramente la necessità di dover modificare il codice sorgente originale del servizio. In alcuni casi questo potrebbe non essere possibile (si pensi ad esempio ai software proprietari) e inoltre il codice del servizio viene "appesantito" con una parte importante di logica che potrebbe non essere compatibile con la natura del servizio stesso (si pensi ad esempio a servizi in ambito IoT sviluppati per funzionare su hardware a basse prestazioni). La gestione delle risorse potrebbe comunque risultare non banale all'interno del servizio nonostante sia potenzialmente possibile accedere a tutto il suo stato. Inoltre lo sviluppo di queste funzionalità potrebbe richiedere competenze diverse da quelle possedute dal programmatore del servizio originale.

Un ulteriore svantaggio è la difficile riutilizzabilità del codice che aggiunge le funzionalità di auto-orchestrazione.

```
1  void on_client_connected()
2  {
3      ...
4      infrastructure.addCPU();
5      ...
6  }
7
8  void on_client_disconnected()
9  {
10     ...
11     infrastructure.remCPU();
12     ...
13 }
14
```

```
15 void on_infrastructure_CPU_overload()
16 {
17     ...
18     updateQuality(getActualQuality()-1);
19     ...
20 }
21
22 void on_infrastructure_CPU_overload_resolved()
23 {
24     ...
25     updateQuality(getActualQuality()+1);
26     ...
27 }
```

Listing 4.1. Pseudo codice di un servizio con auto-orchestratore embedded

4.3.2 Auto-orchestrazione detached

Generalmente è difficile convincere i fornitori di servizi ad aggiungere complessità nei loro software perché questo renderebbe più complicata la manutenzione. Per questo motivo, nella tesi, si è approfondito un altro metodo, dove il modulo di auto-orchestrazione non è integrato ma è esterno ed è inoltre automaticamente generato dall'infrastruttura. In questo caso il servizio si limita a esporre delle API che consentono la modifica della sua configurazione e l'interrogazione del suo stato attuale.

Come già evidenziato in precedenza, il modulo di auto-orchestrazione non può essere generico ma, al fine di risolvere i problemi elencati nel Capitolo 2 deve essere necessariamente specializzato per il servizio in questione. Per rispondere a questa esigenza, il servizio verrà fornito insieme ad un modello che servirà a specializzare il modulo di auto-orchestrazione.

Il modulo dovrà supportare il più ampio numero possibile di interfacce per collegarsi alle API messe a disposizione dai microservizi per interrogare e modificare il loro stato e la loro configurazione (ad esempio REST API, messaggi su un message broker, socket TCP, web-socket, ecc..).

La separazione del modulo di auto-orchestrazione dal servizio originale risolve i problemi esposti precedentemente, in particolare quelli relativi alla manutenzione del codice ma anche quelli legati all'inserimento di ulteriore logica che potrebbe appesantire eccessivamente il servizio.

Questo modo di procedere infatti, non appesantisce il servizio e risulta particolarmente utile in alcune situazioni, tipo l'IoT. Il software del modulo di auto-orchestrazione potrebbe essere troppo complesso per dei nodi IoT che di solito possiedono limitate capacità di calcolo per soddisfare dei requisiti di consumi energetici.

Con questa implementazione, invece, il servizio non dovrà rinunciare alle funzionalità aggiunte dall'auto-orchestrazione, ma il modulo verrà istanziato su un nodo vicino ai sensori IoT. Nel caso di sistemi particolarmente sparsi si può ipotizzare anche l'esistenza di un modulo di auto-orchestrazione distribuito.

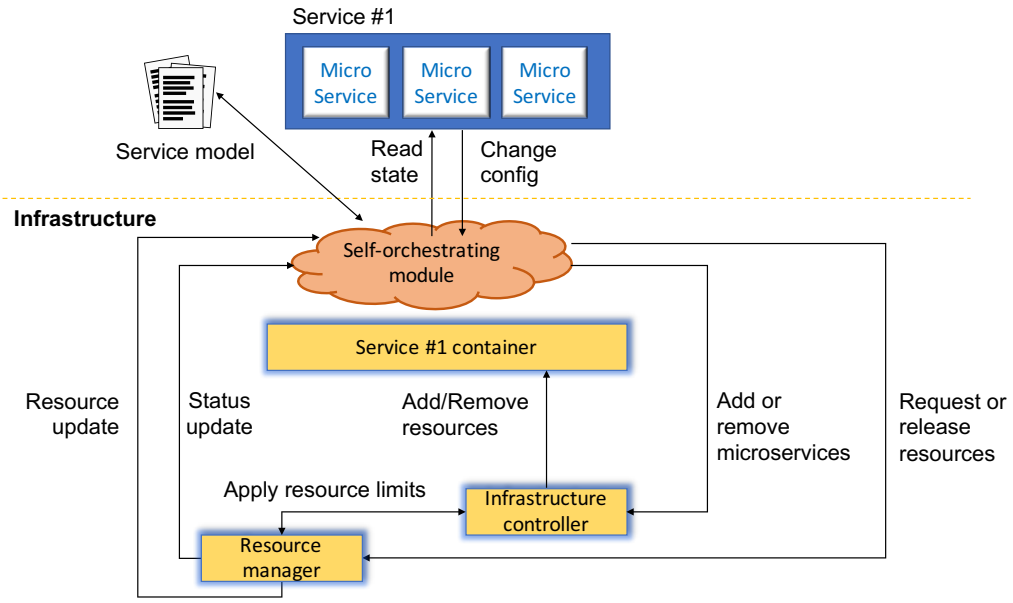

Figura 4.4. Illustrazione del modulo di auto-orchestrazione detached.

4.3.3 Modello

Come indicato nella sezione precedente, per specializzare un modulo di orchestrazione generico e fornito dall'infrastruttura è necessario che il servizio fornisca un modello che descriva come reagire a determinati eventi. Una descrizione separata facilita la migrazione di un servizio tradizionale verso uno auto-orchestrato riducendo lo sforzo necessario per la transizione.

Il modello dovrà contenere queste informazioni:

- la relazione tra lo stato e la configurazione del servizio e le risorse necessarie.

  Queste informazioni sono utilizzate dal modulo di auto-orchestrazione per richiedere le risorse necessarie all'infrastruttura e nel caso in cui non siano disponibili, per poter cambiare la configurazione del servizio fino al raggiungimento di una che, in base allo stato corrente, richieda un numero di risorse disponibili nell'infrastruttura.

- come interfacciarsi con i vari microservizi per cambiare la loro configurazione e per monitorare i loro eventi.

In questa sezione il modello descrive come interrogare e configurare i vari microservizi: quali protocolli usare (http rest, web-socket, socket tcp, ecc.), come costruire i messaggi (il formato dei dati, quali campi includere e come valorizzarli) e come interpretare le risposte (il formato dei dati, il significato dei vari campi ed eventuali trasformazioni da effettuare).

- quali eventi andranno ascoltati e le azioni da intraprendere a seguito di questi.

  Le azioni possono prevedere delle modifiche da effettuare al servizio prima di rimuovere una determinata risorsa, quando viene aggiunto un nuovo microservizio oppure quando si vuole sostituire l'implementazione di un microservizio con un'altra.

  Consideriamo ad esempio un servizio di web hosting composto da due tipi di microservizi: una serie di webserver e un loadbalancer. Al raggiungimento di un determinato numero di connessioni (evento: cambiamento nello stato del microservizio loadbalancer) viene aggiunta una nuova istanza del microservizio webserver (azione: istanziato un nuovo microservizio) e contemporaneamente viene modificata la configurazione del loadbalancer per informarlo che un nuovo webserver è stato istanziato (azione: modificata la configurazione di un microservizio).

4.4 Problemi di sicurezza

In questa sezione si analizzano gli eventuali problemi di sicurezza che si potrebbero verificare durante l'implementazione dei servizi auto-orchestranti.

L'aggiunta di canali di comunicazione tra l'infrastruttura e il servizio aumenta anche la superficie di attacco di eventuali malintenzionati. Le informazioni provenienti dall'infrastruttura potrebbero essere utilizzate in modo fraudolento per capire quando l'infrastruttura si trova in una situazione critica e quindi potenzialmente più vulnerabile ad attacchi DDoS.

La topologia potrebbe rilevare eventuali punti deboli che però potrebbero essere nascosti utilizzando la virtualizzazione topologica.

Un altro problema che potrebbe verificarsi è il caso in cui un servizio continui a richiedere risorse anche se non sono effettivamente necessarie. Una strategia per risolvere questa situazione potrebbe essere un accordo commerciale tra il proprietario dell'infrastruttura e il fornitore del servizio che prevede un compenso economico per ogni risorsa richiesta.

La seconda strategia illustrata nella sezione precedente, che sarà poi quella effettivamente implementata, prevede un modulo di auto-orchestrazione separato e fornito dall'infrastruttura. Questa soluzione ha alcuni svantaggi dal punto di vista della sicurezza, infatti in ogni momento l'infrastruttura conosce lo stato e la configurazione di tutti i microservizi. Questo significa rendere potenzialmente accessibili

informazioni private (dati degli utenti, dettagli implementativi, strategie commerciali) al proprietario dell'infrastruttura. Inoltre, tramite il modello fornito insieme al servizio, l'infrastruttura conosce anche come il servizio reagisce a fronte di determinati eventi e queste informazioni potrebbero rivelare ulteriori dettagli. Nel caso del modulo integrato invece queste informazioni rimarrebbero private all'interno del servizio e tutte le comunicazioni tra il modulo e i microservizi possono essere cifrate.

Capitolo 5

Implementazione

In questo capitolo viene illustrato il prototipo che è stato realizzato secondo le specifiche dell'architettura descritta nel Capitolo 4.

Lo sviluppo del prototipo è avvenuto in tre fasi:

- prima fase: definizione di un formalismo per descrivere le esigenze di un servizio generico (come e cosa scrivere nel modello).
- seconda fase: realizzazione di un modulo di auto-orchestrazione di tipo *detached* che quindi deve modificare il suo comportamento in base al contenuto del modello che sarà fornito insieme al servizio.
- terza fase: implementazione delle interfacce mancanti all'interno del software di orchestrazione dell'infrastruttura scelto.

Come orchestratore dell'infrastruttura è stato scelto l'Universal Node perchè open-source e facilmente modificabile. A seguito di questa decisione per la descrizione dei servizi è stato necessariamente scelto il formalismo NFFG (Network Function-Forwarding Graph) [9]. Alle VNF (Virtual Network Function) è stato associato il significato di microservizio mentre all'NFFG quello di servizio. L'Universal Node può utilizzare diversi back-end di virtualizzazione (Docker, KVM, ecc..), ma in questo caso è stato considerato esclusivamente KVM e quindi solo VNF implementate come macchine virtuali.

Allo stato attuale questo orchestratore permette di aggiungere o togliere VNF (microservizi) da un NFFG (servizio) ma non fornisce nessuna API per modificare le risorse assegnate alle VNF se implementate come macchine virtuali KVM, alle quali viene assegnato un quantitativo di risorse preconfigurato e non modificabile.

Nell'ambito dello sviluppo del prototipo, all'infrastruttura è stato aggiunto il *gestore delle risorse* che si occupa di gestire le risorse disponibili e si assicura che i servizi possano consumare al massimo le risorse assegnate.

Per velocizzare lo sviluppo del prototipo, invece di effettuare il parsing del modello direttamente dal modulo di auto-orchestrazione, è stato introdotto un compilatore che riceve come input il modello e produce delle classi Java che vengono poi utilizzate dal modulo di auto-orchestrazione. In alcuni casi il software sviluppato presenta delle semplificazioni rispetto all'architettura descritta nel Capitolo 4, ad esempio per accedere allo stato dei microservizi è previsto solamente l'utilizzo del *Configuration Service* e non è possibile utilizzare altri protocolli.

5.1 Modello per la descrizione del servizio

Il modello è la parte più importante di questa architettura perché consente una facile migrazione dalla soluzione tradizionale alla soluzione basata su servizi auto-orchestranti. Lo scopo del modello è fornire tutte le informazioni necessarie al modulo di auto-orchestrazione al fine di ottimizzare il funzionamento del servizio.

Per rendere semplice la scrittura del modello, è stato scelto un approccio perlopiù dichiarativo. Il maggiore ostacolo si è rivelato essere la libreria Jackson che impone determinati vincoli su come strutturare gli oggetti JSON per consentire una conversione in oggetti Java. Il modello è stato implementato come un oggetto JSON composto da 7 macro aree.

```
1  {
2      "name": "my_service",
3      "default_nffg_filename":"nffg.json",
4      "variables": [...],
5      "state": [...],
6      "events": [...],
7      "actions": [...],
8      "service_description": [...],
9      "templates": [...]
10 }
```

Listing 5.1. Esempio di un modello rappresentato in JSON.

- In una prima parte (**name** e **default_nffg_filename**) è possibile definire alcune proprietà generali: il nome del servizio e il file che contiene il grafo NFFG da istanziare all'avvio.

- Nella seconda parte (**variables**) è possibile definire delle variabili che definiscono lo stato del modulo di auto-orchestrazione. Queste potranno ad esempio essere utilizzate per particolari elaborazioni sullo stato dei microservizi.

- Nella terza parte (**state**) è possibile definire quali campi all'interno dello stato di un determinato tipo di microservizio bisogna monitorare (ad esempio la NatTable di un NAT).
- Nella quarta parte (**events**) si definiscono quali sono gli eventi che si vogliono catturare a seguito della variazione di uno stato (ad esempio si vuole monitorare l'aggiunta di una nuova entry nella NatTable) e quali sono le azioni da eseguire quando si verifica.
- Nella quinta parte (**actions**) è possibile definire un elenco di comandi che saranno eseguiti per gestire degli eventi.
- Nella sesta parte (**service_description**) viene descritto il servizio: l'elenco dei microservizi dei quali è composto, se esistono più implementazioni per uno stesso microservizio (ad esempio due VNF diverse che eseguono la stessa operazione sui dati ma con risorse differenti), e quali risorse utilizzino.
- Nell'ultima parte (**templates**) possono essere elencati dei template che verranno applicati ai microservizi quando saranno istanziati. Un template consiste in una configurazione iniziale e un elenco dei collegamenti che vengono creati all'avvio del microservizio (ad esempio connettere in automatico la porta di management della VNF allo switch di management)

É importante comprendere da subito la differenza tra le azioni collegate agli eventi e la descrizione del servizio. Quest'ultima serve per insegnare al modulo di auto-orchestrazione le necessità del servizio in termini di risorse e come reagire nel caso in cui queste siano limitate, cercando di preservare la qualità del servizio.

Gli eventi invece servono per gestire tutte le situazioni "di contorno" come ad esempio la modifica di una flowrule quando una VNF viene aggiunta o rimossa oppure per aggiornare delle variabili di stato.

5.1.1 Templates

Il campo **templates** contiene una lista di oggetti **template**.

Gli oggetti **template** sono così composti:

- **id**: stringa che identifica in modo univoco il template.
- **default_ports_connection**: contiene un array associativo dove la chiave è il nome della porta del microservizio a cui è associato il template e il valore corrisponde al nome della porta alla quale va connessa.
- **default_configuration**: nome del file che contiene la configurazione di default per i microservizi associati a questo template.

I template facilitano l'aggiunta e la rimozione delle VNF, non è più necessario aggiungere esplicitamente una flowrule per collegare le porte. Nel caso mostrato nel listato 5.2 la porta "lanPort" del microservizio a cui è associato il template viene collegata alla prima porta disponibile di nome "port" sulla VNF "SWITCH_LAN". Analogamente la porta "managementPort" viene connessa alla porta "port" della VNF "SWITCH_MAN".

```
{
    "id": "template_transcoder",
    "default_ports_connections": {
        "lanPort": "SWITCH_LAN:port",
        "managementPort": "SWITCH_MAN:port"
    },
    "default_configuration":
       "transcoder_defaultconfig.json"
}
```

Listing 5.2. Esempio di un template.

5.1.2 Tipi di dato ParamDescription e MacroDescription

ParamDescription e **MacroDescription** sono due tipi di dato che possono essere utilizzati per descrivere il contenuto di un campo di un oggetto JSON. **ParamDescription** è un tipo che permette di definire un *placeholder* ovvero un valore che verrà valutato a run-time. **MacroDescription** invece descrive un comando da eseguire che non necessariamente rappresenta un valore. Ad esempio può essere utilizzato per sommare il contenuto di due variabili, cambiare il contenuto una variabile di stato, modificare la configurazione di un microservizio, ecc...

Attraverso il tipo **ParamDescription** è possibile descrivere sia dei valori costanti di tipo elementare (**String**, **Integer**, **Double**, ecc.) sia altri di tipo dinamico (ad esempio che dipendono dal contenuto delle variabili di stato dell'orchestatore, dallo stato dell'infrastruttura o dei microservizi).

Gli oggetti che estendono **ParamDescription** devono necessariamente avere un campo di nome **type** che contiene una delle seguenti stringhe:

- **String**, **Integer**, **Double** o **Boolean**: rappresentano gli omonimi tipi di dato. In questi casi l'oggetto presenta un ulteriore campo di nome **value** che contiene il valore effettivo che si vuole assegnare al parametro.

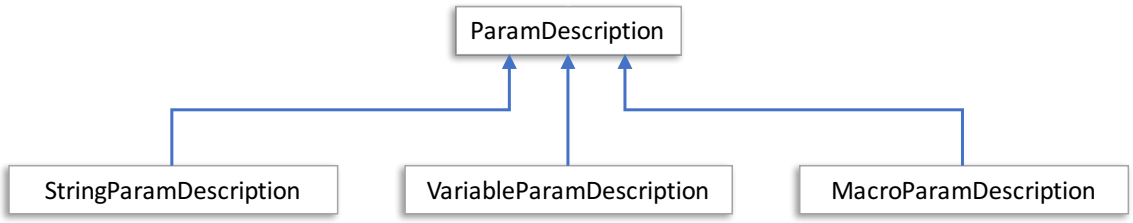

Figura 5.1. Parziale rappresentazione dei tipi che estendono ParamDescription.

```
{
    "type": "String",
    "value": "00:11:22:33:44:55"
}
```

Listing 5.3. Esempio di parametro che contiene una stringa (**StringParamDescription**). A runtime verrà sostituito dal valore "00:11:22:33:44:55"

```
{
    "type": "Integer",
    "value": 60000
}
```

Listing 5.4. Esempio di parametro che contiene un interno (**IntegerParamDescription**). A runtime verrà sostituito dal valore 60000.

- **Variable**: quando si vuole assegnare ad un ParamDescription il valore di una variabile del modulo di auto-orchestrazione, sarà necessario inserire nel campo **variable** una stringa equivalente al nome della variabile.

```
{
    "type": "Variable",
    "variable": "counter"
}
```

Listing 5.5. Esempio di parametro che contiene il contenuto di una variabile (**VariableParamDescription**). A runtime verrà sostituito dal valore della variabile "counter" (che deve essere definita nella sezione **Variables**).

- **Macro**: quando si vogliono effettuare delle trasformazioni ai dati (ad esempio eseguire la somma di due valori), si vuole accedere ad un particolare campo di

un oggetto (ad esempio ottenere la dimensione di una collezione) o si vogliono eseguire altre operazioni complesse è possibile utilizzare una **MacroDescription**. In questo caso è necessario valorizzare il campo **macro** con un oggetto **MacroDescription**.

Come già indicato precedentemente, a causa delle limitazioni di Jackson, non è possibile inserire direttamente un oggetto di tipo **MacroDescription** in un campo che prevede il tipo **ParamDescription**, per questo motivo è stato annidato all'interno di quest'ultimo.

```
1 {
2     "type": "Macro",
3     "macro": { ... }
4 }
```

Listing 5.6. Esempio di parametro che contiene il risultati di una macro (*MacroParamDescription*). A runtime verrà sostituito con il risultato della **Macro** specificata nel campo marco (il risultato verrà ricalcolato ogni volta che verrà richiesto di conoscere il valore di questo **ParamDescription**)

Se consideriamo i listing 5.3 e 5.4, l'utilizzo del tipo **ParamDescription** può risultare superfluo ma invece è giustificato da più motivazioni:

- la libreria Jackson, utilizzata per il parsing, prevede che a tutti i campi di ogni oggetto JSON sia associato un preciso tipo di dato. Evitando l'utilizzo dei **ParamDescription** risulterebbe impossibile assegnare a un campo un valore di tipo diverso a seconda della situazione.
- se fosse consentito assegnare direttamente un valore di tipo **String** a un campo, dove ora è previsto il **ParamDescription**, si creerebbero delle situazioni di ambiguità: il valore rappresenterebbe una stringa costante o il nome di una variabile?
- il tipo **ParamDescription** permette anche di manipolare i dati prima che questi siano utilizzati (ad esempio l'aggiunta a una stringa di un suffisso), annidando al suo interno una **MacroDescription** che descrive come modificarli.

Il tipo **MacroDescription** serve per descrivere dei comandi da eseguire. Il risultato di questi comandi può eventualmente rappresentare un valore (ad esempio il risultato di una somma). Gli oggetti di tipo **MacroDescription** possono quindi descrivere sia come modificare un determinato valore o variabile e sia eseguire delle azioni (ad esempio aggiungere una nuova FlowRule).

Gli oggetti che estendono **MacroDescription** devono necessariamente avere un campo di nome **type** che contiene una delle seguenti stringhe:

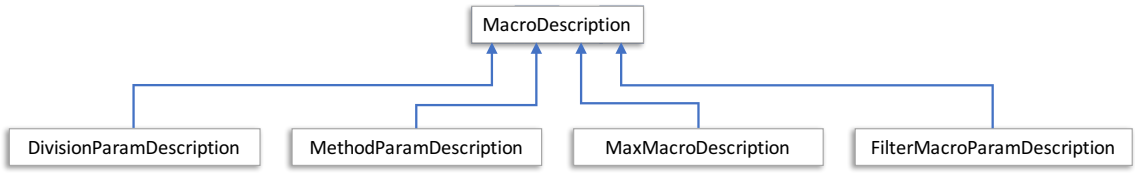

Figura 5.2. Parziale rappresentazione dei tipi che estendono MacroDescription.

- **New**: istanzia un nuovo oggetto e eventualmente lo assegna a una variabile, se necessario passa i parametri contenuti nel campo **params** (lista di **ParamDescription**) al costruttore.

```
{
        "macro": "new",
        "type": "Host",
        "assign_to": "myHost",
        "params":
        [
        {
            "type":"String",
            "value": "192.168.1.1"
        },
        {
            "type":"String",
            "value": "00:11:22:33:44:55"
        }
    ]
}
```

Listing 5.7. Istanzia un nuovo oggetto di tipo **Host** e lo assegna alla variabile **myHost**. Il codice Java equivalente è: Host myHost = new Host(“192.168.1.1”,“00:11:22:33:44:55”) .

- **Method**: esegue il metodo indicato nel campo **method** (di tipo stringa) sull’oggetto specificato nel campo **on** (di tipo **ParamDescription**). Durante lo sviluppo di questo prototipo sono stati considerati metodi validi tutti quelli presenti sull’oggetto Java corrispondente a quello contenuto nel campo **on**.

```
{
    "macro": "method",
    "method": "size",
```

```
4       "on": {
5           "type": "Variable",
6           "variable": "myListOfHost"
7       }
8   }
```

Listing 5.8. Esegue il metodo **size()** sull'oggetto contenuto nella variabile **myListOfHost**. Il codice Java equivalente è: myListOfHost.size() .

- **Division**: esegue l'operazione di divisione tra il valore contenuto nel campo **divider** (di tipo **ParamDescription**) e il contenuto del capo **dividend** (di tipo **ParamDescription**).

```
1   {
2       "macro": "division",
3       "divider": {
4           "type":"Integer",
5           "value": 10
6       },
7       "divisor": {
8           "type":"Double",
9           "value": 2.5
10      }
11  }
```

Listing 5.9. Esegue l'operazione 10/2.5 .

- **Concat**: esegue il concatenamento delle stringhe presenti nel campo **params** (lista di **ParamDescription**).

```
1   {
2       "macro": "concat",
3       "params": [
4           {
5               "type": "String",
6               "value": "prefix"
7           },
8           {
9               "type": "Variable",
10              "variable": "myString"
11          },
12          {
```

```
13                 "type": "String",
14                 "value": "suffix"
15             },
16         ]
17 }
```

Listing 5.10. Esegue il concatenamento delle stringe contenute nel campo **params**. Il codice Java equivalente è: “prefix” + myString + “suffix” .

- **Max** o **Min**: selezionano rispettivamente il valore massimo o il valore minimo contenuti nella collezione indicata nel campo **on** (di tipo **ParamDescription**).

```
1 {
2     "macro": "max",
3     "on": {
4         "type": "Variable",
5         "variable": "myListOfInteger"
6     }
7 }
```

Listing 5.11. Ricerca e rappresenta il valore massimo contenuto nella lista **myListOfInteger**.

- **Filter**: esegue un operazione di filtraggio sulla collezione specificata nel campo **on** (di tipo **ParamDescription**) restituendo una lista contenente solo gli elementi uguali all’oggetto indicato nel campo **equal_to** (di tipo **ParamDescription**).

```
1 {
2     "macro": "filter",
3     "on": {
4         "type": "Variable",
5         "variable": "myListOfHost"
6     },
7     "filter_method": "getAddress",
8     "equal_to": {
9         "type": "String",
10        "value": "192.168.100.10"
11    }
12 }
```

Listing 5.12. Rappresenta una lista che contiene gli elementi della lista **myListOfHost** il cui valore ritornato dal metodo **getAddress()** risulta essere **“192.168.100.10”**.

Se le macro restituiscono un valore, è possibile assegnarlo a una variabile temporanea. Per fare questo è sufficiente valorizzare il campo **assign_to** (stringa) con il nome e **type** (stringa) con il tipo della variabile.

5.1.3 Variables

In questo campo è possibile definire le variabili di stato del modulo di auto-orchestrazione.

Il campo contiene una lista di oggetti **variable** che sono così composti:

- **name**: stringa che contiene il nome della variabile.
- **type**: stringa con il tipo della variabile che può essere scelto tra **String**, **Integer**, **Boolean**, **List** e **Map** .
- **key_of**: stringa che identifica il tipo della chiave della mappa, necessario solo se **type** è **Map**.
- **value_of**: stringa che identifica il tipo del valore della mappa, necessario solo se **type** è **Map**.
- **list_of**: stringa che identifica il tipo del contenuto nella lista, necessario solo se **type** è **List**.
- **params**: lista di parametri (**ParamDescription**) da passare al costruttore della variabile per l’inizializzazione.

```
1  [
2      {
3          "name": "hosts",
4          "type": "ArrayList",
5          "list_of": "Host"
6      },
7      {
8          "name": "lbport",
9          "type": "String",
10         "params": [
11             {
```

```
12                      "type": "String",
13                      "value": "vnf:SWITCH_LAN:port1:0"
14                  }
15              ]
16      }
17  ]
```

Listing 5.13. Esempio di contenuto del campo variables.

5.1.4 State

In questa sezione è possibile definire quali proprietà nello stato di un microservizio si vogliono monitorare.

- **id**: stringa che contiene e identifica lo stato.
- **type**: stringa che determina il tipo di controllo da effettuare sullo stato del servizio. Al momento è disponibile solamente il tipo **polling**.
- **check_every**: stringa da specificare nel caso di polling, che indica ogni quanto effettuare il controllo.
- **check_on**: lista di stringhe che contengono i tipi che rappresentano i microservizi che si vogliono monitorare.
- **check_method**: stringa che identifica il metodo da eseguire sugli oggetti che rappresentano i microservizi per accedere alla proprietà che si desidera monitorare. Negli sviluppi futuri questo campo potrebbe rappresentare ad esempio il path (XML/YANG) per raggiungere la proprietà.
- **check_type**: specifica il tipo di dato contenuto nella proprietà da monitorare.
- **list_of**: nel caso in cui la proprietà sia una lista, questa stringa identifica il tipo del dato contenuto nella lista.

```
1  {
2      "id": "nat_session_state",
3      "type": "polling",
4      "check_every": "100ms",
5      "check_method": "getNatSession",
6      "check_type": "list",
7      "param_type": "NatSession",
8      "check_on": [
```

```
9           "NatVNF"
10      ]
11  }
```

Listing 5.14. Esempio di oggetto contenuto nella lista del campo state.

5.1.5 Events

In questa sezione del modello è possibile associare precise variazioni nello stato ad un elenco di azioni da eseguire.

- **id**: stringa che contiene identifica l'evento.
- **on**: stringa che contiene l'id dello stato a cui è associato l'evento. Se questo campo non è valorizzato vuol dire che si sta definendo un evento associato all'infrastruttura.
- **type**: stringa che specifica l'evento che si vuole monitorare.

  Se si tratta di un evento collegato ad uno stato i valori ammessi in questo campo sono: **NEW**, **DEL** e **CHANGE** che rispettivamente vengono scatenati quando viene aggiunto un nuovo elemento (da una lista), quando uno viene cancellato (da una lista) e quando uno di questi cambia.

  Se si tratta invece di un evento collegato all'infrastruttura i valori ammessi sono: **NEW_RESOURCE**, **REMOVING_RESOURCE**, **REMOVED_RESOURCE**, **NEW_VNF**, **REMOVING_VNF** e **REMOVED_VNF**.
- **actions**: lista di stringhe che rappresentano gli identificativi delle azioni da eseguire quando si verifica l'evento.

```
1  {
2      "id": "on_new_nat_session",
3      "on": "nat_session_state",
4      "type": "NEW",
5      "actions": [
6          "action_session_new"
7      ]
8  }
```

Listing 5.15. Esempio di oggetto contenuto nella lista del campo event.

5.1.6 Actions

Il campo **actions** contiene un elenco di comandi che sono eseguiti quando si verificano un evento.

- **id**: stringa che contiene identifica l'azione.
- **params**: lista di oggetti che associano il tipo di dato ricevuto come parametro ad una variabile temporanea.
- **steps**: lista di MacroDescription che contiene, in sequenza, le macro da eseguire.

```
{
    "id": "action_session_new",
    "params":
    [
        {
            "name": "nat",
            "type": "NatVNF"
        },
        {
            "name": "session",
            "type": "NatSession"

        }
    ],
    "steps":
    [
        {
            "macro": "new",
            "type": "Host",
            "assign_to": "session_host",
            "params":
            [
                {
                    "type": "Macro",
                    "macro": {
                        "macro": "method",
                        "method": "getSrc_address",
                        "on": {
                            "type": "Variable",
                            "variable": "session"
```

```
31                              }
32                          }
33                      },
34                      {
35                          "type": "Macro",
36                          "macro": {
37                              "macro": "method",
38                              "method": "getSrcMac_address",
39                              "on": {
40                                  "type": "Variable",
41                                  "variable": "session"
42                              }
43                          }
44                      }
45                  ]
46          },
47          {
48                  "macro": "method",
49                  "method": "put",
50                  "on": {
51                      "type": "Variable",
52                      "variable": "active_hosts"
53                  },
54                  "params":
55                  [
56                      {
57                          "type": "Variable",
58                          "variable": "session_host"
59                      }
60                  ]
61          }
62      ]
63  }
```

Listing 5.16. Esempio di azione che aggiunge in una variabile di stato dell'orchestratore (active_hosts) un nuovo elemento in base alle modifiche avvenute nello stato di un microservizio.

5.1.7 Service description

Questa è la parte fondamentale del modello. Il formalismo scelto è ispirato a quello proposto nell'articolo [32]. Il servizio viene descritto come un insieme di servizi elementari, ognuno dei quali rappresenta una determinata funzionalità. Ogni servizio elementare può essere realizzato attraverso più implementazioni. Il compito del modulo di auto-orchestrazione sarà di riuscire a implementare tutti i servizi elementari, scegliendo l'implementazione migliore (dal punto di vista delle risorse consumate e della qualità del servizio erogato).

Nei casi previsti da questa tesi, il servizio elementare corrisponde alla capacità che deve offrire un microservizio (ad esempio NAT, DHCP, ecc.), mentre l'implementazione corrisponde ad una sua specifica versione (ad esempio NAT implementato con IPTables oppure con RRAS). E' possibile inoltre specificare delle gerarchie di servizi auto-orchestranti specificando come implementazione di un servizio elementare un altro servizio auto-orchestrato. In questo caso il modulo di auto-orchestrazione espoterà delle API che verranno utilizzate dal modulo di auto-orchestrazione principale.

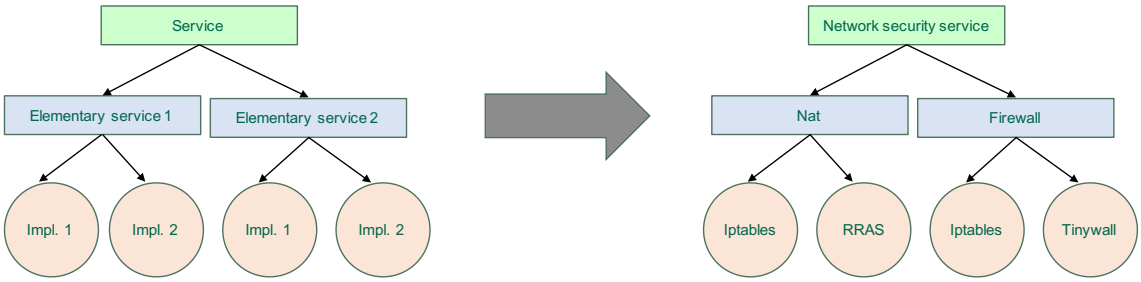

Figura 5.3. Rappresentazione grafica del formalismo previsto.

Ogni servizio elementare è descritto da due proprietà:

- **name**: una stringa che rappresenta il nome del servizio elementare.
- **Implementations**: una lista d'implementazioni del servizio elementare.

```
1 {
2     "name": "transcoder",
3     "implementations": [...]
4 }
```

Listing 5.17. Esempio di un servizio elementare

Ogni implementazione è descritta nel seguente modo:

- **name**: una stringa che rappresenta il nome dell'implementazione.

- **microservice_type**: identificativo del template (NFFG) della VNF che realizza l'implementazione.
- **microservice_template**: id del template da applicare al microservizio.
- **configurations**: lista delle possibili configurazioni utilizzabili e loro relativa qualità offerta.
- **resources_used**: lista che specifica quante risorse sono necessarie al servizio, in base al suo stato.

```
{
    "name": "trcd1",
    "microservice_type": "TranscoderVNF",
    "microservice_template": "template_transcoder",
    "configurations": [...],
    "resources_used": [...]
}
```

Listing 5.18. Esempio di implementazione.

Ogni configurazione è descritta nel seguente modo:

- **qos_value**: un intero che rappresenta la qualità del servizio offerta da questa configurazione (0 minima e 100 massima).
- **configuration**: contiene una lista di **MacroDescription** che agendo sulla variabile **configuration** modificano la configurazione corrente con quella desiderata. Le Macro vengono eseguite in sequenza, dalla prima all'ultima.

```
{
    "qos_value":100,
    "configuration": [
        {
            "macro": "method",
            "method": "setEnabled",
            "on": {
                "type": "Variable",
                "variable": "configuration"
            },
            "params": [
                {
                    "type": "Boolean",
                    "value": true
```

```
15                      }
16                  ]
17              }
18      ]
19  }
```

Listing 5.19. Esempio di configurazione con QoS=100 che prevedere di impostare "enabled" della configurazione sul valore true.

Ogni requisito di risorse è descritto nel seguente modo:

- **resource_type**: è una stringa che rappresenta il tipo di risorsa di cui si stanno descrivendo i requisiti. Al momento è solo supportato solo il tipo CPU.
- **limit**: contiene una lista di **CPUResourceRequirement** (Attualmente è l'unico ResourceRequirement supportato). Ogni CPUResourceRequirement prevede un campo di nome **resources_needed** che indica il numero di risorse necessarie (che può essere un valore fisso oppure calcolato in base allo stato) e un campo **condition** che permette di specificare una condizione affinchè il requisito sia considerato valido. Questo può essere utile per definire dei vincoli diversi a seconda di una condizione.

```
1  {
2      "resource_type":"CPU",
3      "limit":
4      [
5          {
6              "resources_needed":
7              {
8                  "type":"Integer",
9                  "value":4
10             },
11             "condition":
12             {
13                 "operand1": {
14                     "type": "Macro",
15                     "macro": {
16                         "macro": "method",
17                         "method": "getTranscoderEnabled",
18                         "on": {
19                             "type": "Variable",
20                             "variable": "configuration"
21                         }
```

```
22                          }
23                      },
24                      "operand2":
25                      {
26                          "type":"Boolean",
27                          "value": "true"
28                      },
29                      "compare_mode": "equal_to"
30                  }
31              },
32              [...]
33      ]
34  }
```

Listing 5.20. Esempio di requisito sulla risorsa CPU. Se il campo 'transcoderEnabled' è impostato su true, sono necessarie 4 CPU.

Le definizione delle risorse necessarie è separata rispetto alla definzione della configurazione perchè esse possono dipendere non solo dalla configurazione in uso ma anche dallo stato dei vari microservizi. Ad esempio un microservizio potrebbe richiedere una percentuale di CPU proporzionale al numero di client connessi.

5.2 Compilatore

L'eseguibile del compilatore prevede tre parametri: il file che contiene il modello, il package Java di base da utilizzare per le classi generate e la directory di output. Il compito di questo programma è generare delle classi Java per il modulo di auto-orchestrazione.

Il software che implementa il compilatore utilizza la libreria Jackson per fare il parsing dell'oggetto JSON presente dentro il file del modello. Ad ogni oggetto JSON è associata una classe, opportunamente etichettata con le annotazioni previste dalla libreria (ad esempio @JSONProperty).

Le classi che rappresentano gli oggetti JSON del modello devono implementare una di queste due interfacce: **GenerateJavaCode** o **GenerateJavaClass**. La prima viene utilizzata quando l'oggetto JSON (ad esempio **MacroDescription** e **ParamDescription**) deve essere tradotto in istruzioni Java, la seconda quando deve essere convertito in una classe.

Le classi che sono generate implementano le seguenti classi astratte:

- una classe che estende **AbstractInfrastructureEventHandler**: si occupa di eseguire le azioni previste nel modello a seguito di eventi nell'infrastruttura.

- una classe che estende **AbstractSelfOrchestrator**: nel suo costruttore vengono istanziate le variabili di stato e salvate in un oggetto "variables" accessibile da ogni oggetto. Vengono inoltre istanziate tutte le classi che corrispondono ai servizi elementari.
- alcune classi che estende **AbstractElementaryService**: una per ogni servizio elementare definito nel modello. Nel costruttore vengono istanziate tutte le classi corrispondenti alle implementazioni del servizio elementare
- alcune classi che estende **AbstractImplementation**: una per ogni implementazione definita nel modello. Nel costruttore vengono istanziate tutte le classi corrispondenti ai requisiti di risorse dell'implementazione. Inoltre viene anche implementato un metodo **getConfiguration(Integer qos)** che ritorna la configurazione della VNF associata all'implementazione con un qos uguale a quello specificato come parametro.
- alcune classi che implementano **ResourceRequirement**: una per ogni requisito di risorse definito nel modello (quindi ognuna di queste classi è associata ad un tipo di risorsa). In questa classe viene implementato il metodo **minimum(VNFState state)** dove **state** rappresenta lo stato dell'implementazione e il valore ritornato rappresenta il numero minimo di risorse necessarie all'implementazione con lo stato (che include la configurazione) ricevuto come parametro.
- alcune classi che implementano **VNFTemplate**: una per ogni template definito nel modello.

Il significato di queste classi sarà più evidente quando verrà descritto il funzionamento interno del modulo di auto-orchestrazione.

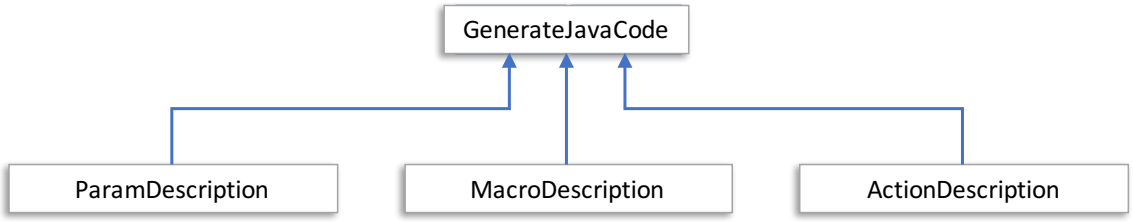

Figura 5.4. Rappresentazione delle classi che estendono GenerateJavaCode.

```
7 public class FilterMacroDescription extends
    MacroDescription
8 {
```

```
9
10      @JsonProperty("filter_method")
11      String filter_method;
12      @JsonProperty("equal_to")
13      ParamDescription equal_to;
14
15      @JsonProperty("on")
16      ParamDescription on;
17
18      @Override
19      public String getJavaCode(boolean from_root, int
           tabs, SelfOrchestratorModel model)
20      {
21          String java =
               super.getJavaCode(from_root,tabs,model);
22
23          java += "(";
24          java += on.getJavaCode(false,tabs,model );
25          java += ".stream().filter(x ->
               x."+filter_method+"().equals("+equal_to.getJavaCode(fals
               )+")";
26          java += ")";
27
28          java +=
               ".collect(Collectors.toCollection(ArrayList::new))
               )";
29
30          return java;
31      }
32
33  }
```

Listing 5.21. Esempio di un oggetto Java che implementa l'interfaccia GenerateJavaCode. In questo listato viene mostrato com'è generato il codice Java relativo alla macro filter del modello.

```
7   public class ElementaryServiceDescription implements
       GenerateJavaClass
8   {
9       @JsonProperty("name")
10      public String name;
11      @JsonProperty("implementations")
```

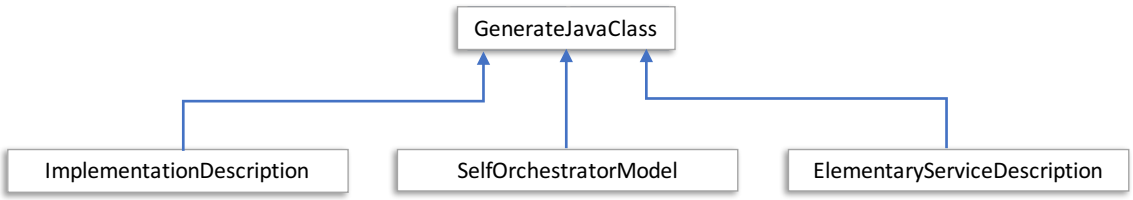

Figura 5.5. Parziale rappresentazione delle classi che estendono GenerateJavaClass.

```
12      public List<ImplementationDescription>
           implementations;
13
14      public String getJavaClass(String
           prefix,SelfOrchestratorModel model,String pack)
15      {
16          String java = "package "+pack+";\n" +
17                  "\n" +
18                  PackageGenerator.getPackage() +
19                  "\n" +
20                  "//Autogenerted class\n" +
21                  "public class
                       "+getJavaClassName(prefix)+" extends
                       AbstractElementaryService\n" +
22                  "{\n" +
23                  "\tpublic
                       "+getJavaClassName(prefix)+"(Variables
                       var)\n" +
24                  "\t{\n" +
25                  "\t\tsuper(var);\n" +
26                  "\t\tList<Class<? extends VNFTemplate>>
                       resourceTemplates = new
                       ArrayList<>();\n";
27
28          for(InfrastructureVNFTemplateDescription
               template : model.getTemplates()) {
29              java += "\t\tresourceTemplates.add(" +
                   template.getJavaClassName(prefix) +
                   ".class);\n";
30          }
31
```

```
32          for(ImplementationDescription implementation :
               implementations)
33          {
34              java += "\t\timplementations.add(new " +
                   implementation.getJavaClassName(prefix+name)
                   + "(var,resourceTemplates));\n";
35          }
36
37          java += "\t\tname = \""+name+"\";\n" +
38                  "\t}\n" +
39                  "}";
40          return java;
41      }
42
43
44      @Override
45      public String getJavaClassName(String prefix)
46      {
47          String className = prefix+name;
48          return
               Character.toUpperCase(className.charAt(0)) +
49                   className.substring(1) +
50                   "ElementaryService";
51      }
52  }
```

Listing 5.22. Esempio di un oggetto Java che implementa l'interfaccia GenerateJavaClass. In questo listato viene mostrato com'è generata la classe Java realtiva ad un servizio elementare.

5.3 Modulo di auto-orchestrazione

In questa sezione viene descritto il funzionamento del modulo di auto-orchestrazione realizzato per questo prototipo.

Le classi principali che compongono questo software sono:

- il **Framework**: è il cuore del modulo, si occupa della coordinazione dei vari componenti e del passaggio delle informazioni da un componente all'altro.
- il **Infrastructure**: è la classe che si occupa di dialogare con l'orchestratore dell'infrastruttura.

- il **SelfOrchestrator**: è una classe generata dal compilatore che rappresenta la descrizione del servizio e contiene l'elenco dei servizi elementari che lo compongono.

- il **ResourceManager**: è la classe che deve cercare la migliore configurazione utilizzabile con le risorse disponibili.

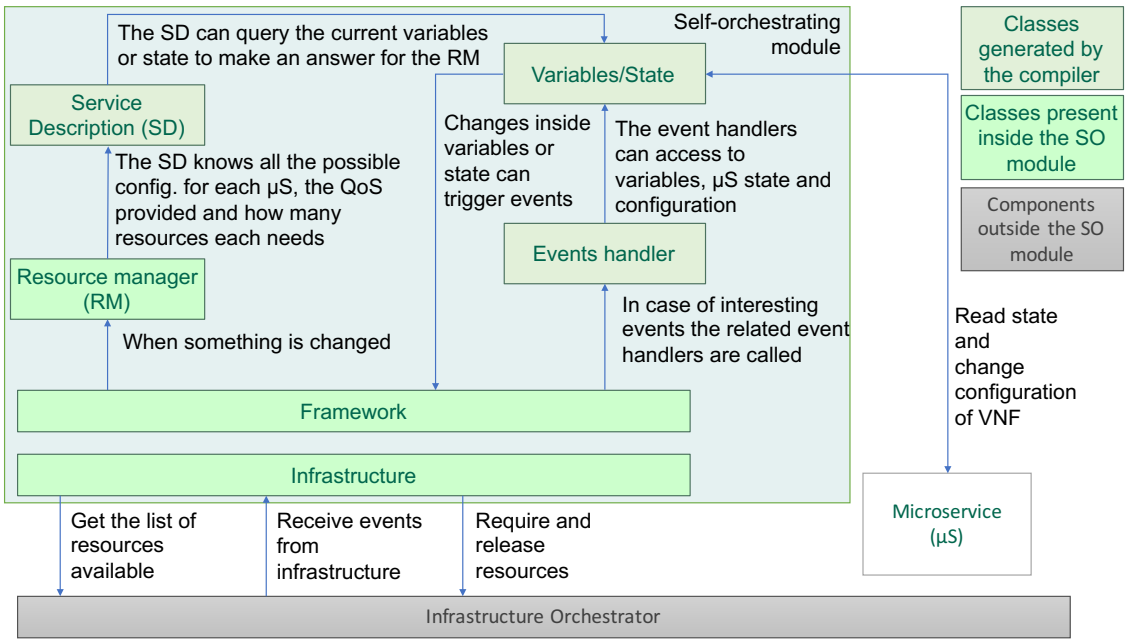

Figura 5.6. Architettura interna del modulo di auto-orchestrazione.

All'avvio del modulo viene istanziato il **Framework**. Dopo la sua creazione si occupa di istanziare tutti gli altri componenti e richiede all'infrastruttura di caricare il grafo NFFG iniziale (specificato all'interno del modello) accessibile tramite il metodo *getInitNFFGFilename()* sull'istanza di **SelfOrchestrator**.

Subito dopo questa operazione, viene invocato il metodo *newServiceConfiguration()* dell'oggetto **ResourceManager** che si occupa di trovare la migliore configurazione possibile del servizio, considerando il suo stato attuale e le risorse disponibili nell'infrastruttura. Questo metodo verrà richiamato ogni qual volta si verifichi un cambiamento all'interno dello stato di un microservizio, di una variabile di stato del modulo di auto-orchestrazione o dell'infrastruttura.

Dopo aver calcolato la prima configurazione, il modulo richiama il metodo *mainLoop()* sull'oggetto **Framework**. Viene quindi avviato un loop infinito che verifica l'esistenza di eventi provenienti dall'infrastruttura o cambiamenti all'interno dei vari microservizi. Nel caso in cui si sia verificata almeno una delle due condizioni, viene

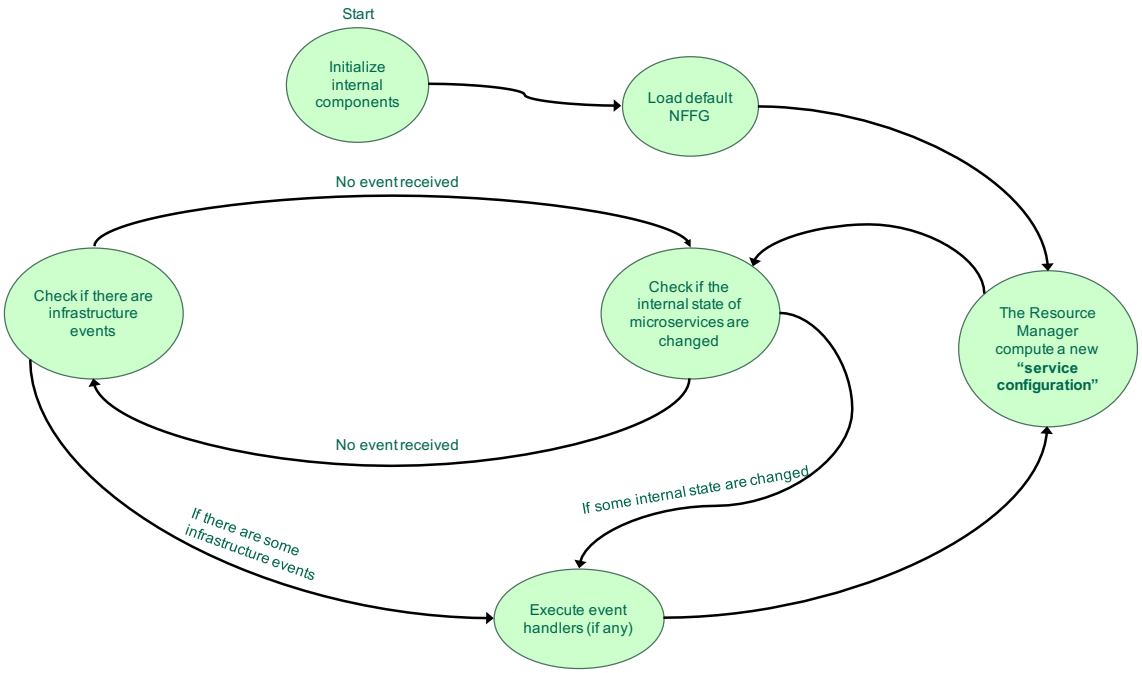

Figura 5.7. Diagramma di stato del modulo di auto-orchestrazione.

invocato nuovamente il metodo *newServiceConfiguration()* del **ResourceManager** per verificare se le esigenze del servizio sono cambiate. Un esempio potrebbe essere il caso di un utente che si disconnette da un servizio che richiede un quantitativo di RAM proporzionale al numero di utenti connessi. In questo caso, al termine del metodo *newServiceConfiguration()* le risorse eccedenti verranno rilasciate. Oppure, nel caso in cui si verificasse un evento dell'infrastruttura che annuncia un cambiamento nelle risorse disponibili, il metodo *newServiceConfiguration()* produrrebbe la migliore configurazione globale per adattare il servizio alle nuove risorse disponibili.

5.3.1 Algoritmo di ricerca della miglior configurazione globale

L'algoritmo utilizzato all'interno della funzione *newServiceConfiguration()* al momento non esplora correttamente tutto lo spazio delle soluzioni possibili ma rimane comunque valido per gli scopi previsti dalla validazione.

L'implementazione di questa funzione inizia con una chiamata al metodo *updateResources()* sull'oggetto **infrastructure** che recupera dall'infrastruttura l'elenco delle risorse disponibili. Viene inoltre inizializzato un oggetto di tipo **ResourceSet** che serve a mantenere un'associazione tra le risorse disponibili e a chi sono state assegnate durante l'esecuzione dell'algoritmo.

Successivamente viene avviato un ciclo su tutti i servizi elementari che compongono il servizio. La funzione dovrà scegliere un'implementazione, una configurazione e un elenco di risorse necessarie per ogni servizio elementare.

A ogni iterazione viene letto lo stato attuale della VNF che al momento sta implementando il servizio elementare rappresentato dall'oggetto **elementaryService** (se la VNF non è ancora stata avviata viene ritornato uno stato di default).

```
0 List<Resource> resources =
     infrastructure.updateResourceAvailable();
1
2 ResourceSet resourceSet = new ResourceSet(resources);
3 List<ElementaryService> elementaryServices =
     framework.getSelfOrchestrator().getElementaryServices();
4 for (ElementaryService elementaryService :
     elementaryServices )
5 {
6      BestImplementation bestImplementation = null;
7      StateVNF current_state =
          elementaryService.getCurrentVNFState();
```

Listing 5.23. Parte 1 del metodo *newServiceConfiguration()*.

Successivamente viene avviato un altro ciclo (listing 5.24) su tutti i valori possibili del parametro qos, partendo dal valore massimo (100) fino al minimo (0). Il loop si ferma se il parametro raggiunge il valore minimo o se viene trovata una configurazione valida.

```
8      for (Integer qos = 100 ; qos >= 0 ; qos--)
9      {
10          if ( bestImplementation != null &&
11               bestImplementation.getQos() > qos ) break;
```

Listing 5.24. Parte 2 del metodo *newServiceConfiguration()*.

In seguito inizia un altro ciclo (listing 5.25) sull'elenco di tutte le implementazioni disponibili per il servizio elementare in esame. Per ogni iterazione viene chiamato il metodo *getConfiguration* sull'oggetto **implementation** utilizzando come parametro la variabile qos (definita alla riga 7 del listing 5.23).

Successivamente viene avviato un ulteriore ciclo che si occupa di verificare se le risorse necessarie per l'implementazione, considerando l'attuale stato e la configurazione proposta, sono disponibili. In caso affermativo le risorse necessarie vengono etichettate come utilizzate nel **resourceSet**, diversamente il programma passa ad analizzare l'implementazione successiva.

```
12          for (Implementation implementation :
               elementaryService.getImplementations())
13          {
14              StateVNF new_state =
                   implementation.getConfiguration(current_state,qos);
15
16              Boolean invalid= false;
17              for (ResourceRequirement resourceRequirement
                   :
                   implementation.getResourceRequirement()) {
18                Integer min =
                       resourceRequirement.minimum(new_state);
19                if ( min > resourceSet.getFreeOfType(
                       resourceRequirement.getResourceType()
                       ).size() ) {
20                  //Not satified.
21                  invalid = true;
22                  break;
23                }
24                resourceSet.setResourceUsed(elementaryService,
                       implementation , resources , min );
25              }
26
27              if (invalid) continue;
```

Listing 5.25. Parte 3 del metodo *newServiceConfiguration()*.

A seguito di queste operazioni, come mostrato nel listing 5.26, viene valutato se la configurazione ottenuta ha una qualità maggiore o se ha un minor consumo di risorse rispetto a quelle calcolate in precedenza. In caso positivo, viene salvata.

```
28              Integer implementationSize =
                   resourceSet.getResourcesOf(implementation).size();
29              Double bestQos = -1;
30              Integer usedSize = -1;
31              if (bestImplementation != null) {
32                  bestQos = bestImplementation.getQos();
33                  usedSize =
                       bestImplementation.getResourcesUsed().size();
34              }
35
36              Double currentQos = new Double(qos);
37              Integer
```

```
38              if (    bestImplementation == null ||
39                      bestQos < currentQos ||
40                      (
41                          bestImplementation.getQos().equals(
                               currentQos ) &&
42                          usedSize < implementationSize)
43                      )
44              {
45                  bestImplementation = new
                       BestImplementation(
46                                  currentQos,
47                                  implementation,
48                                  new_state,
49                                  resourceSet.
                                       getResourcesOf(implementation));
50              }
51          }
52      }
```

Listing 5.26. Parte 4 del metodo *newServiceConfiguration()*.

In quest'ultima parte (listing 5.27), al termine del ciclo iniziato nel listing 5.24 viene impostata la migliore configurazione trovata. Infine, quanto tutti i servizi elementari sono stati valutati, viene richiesto all'infrastruttura di applicare le modifiche richieste.

```
51      elementaryService.setCurrentImplementation(
           bestImplementation.getImplementation());
52
53      infrastructure.useVNF(elementaryService,
54                  bestImplementation.getImplementation(),
55                  bestImplementation.getState(),
56                  bestImplementation.getResourcesUsed());
57  }
58  infrastructure.commit();
```

Listing 5.27. Parte 5 del metodo *newServiceConfiguration()*.

Com'è possibile notare dalla descrizione precedente, l'algoritmo tenta di massimizzare singolarmente la qualità del servizio di ogni servizio elementare e non ha una visione globale con il risultato di assegnare un maggior numero di risorse ai servizi elementari che risultano posizionati all'inizio della lista.

Una valida alternativa a questo semplice algoritmo di assegnazione potrebbe essere l'euristica proposta nell'articolo [32] basata su una variante del problema dello zaino.

5.4 Modifiche richieste all'interno dell'orchestratore dell'infrastruttura

Come indicato nel capitolo 4, nell'infrastruttura va aggiunto un nuovo componente denominato "gestore delle risorse" che espone delle nuove API utilizzate dal modulo di auto-orchestrazione per ricevere informazioni dall'infrastruttura e per gestire le risorse. Nel prototipo il modulo è stato sviluppato esternamente all'orchestratore Universal Node per velocizzarne l'implementazione.

Tre sono le interfacce che il gestore delle risorse mette a disposizione:

- **ResourceAvailable**: interfaccia facoltativa che permette al modulo di auto-orchestrazione di ricevere l'elenco delle risorse disponibili.
- **ResourceUpdate**: interfaccia utilizzata dal modulo di auto-orchestrazione per richiedere/rilasciare risorse utilizzate.
- **Event**: interfaccia utilizzata per comunicare al modulo di auto-orchestrazione che si sono verificati degli eventi nell'infrastruttura (ad esempio una risorsa utilizzata dal servizio presto non sarà più disponibile).

Queste tre interfacce sono state sviluppate in PHP e corrispondono a delle richieste HTTP GET o POST.

Il modulo di auto-orchestrazione esegue un polling sull'interfaccia degli eventi, e nel caso in cui ve ne siano, esegue determinate operazioni. L'utilizzo del polling è accettabile perché il software è stato sviluppato con l'obbiettivo di fornire un prototipo e non un applicativo commerciale. Diversamente si sarebbe optato per l'utilizzo di un MessageBus (ad esempio DoubleDecker [23] che è già integrato dentro l'Universal Node) con la pubblicazione degli eventi in un determinato topic.

Inoltre, durante lo sviluppo del prototipo, è sempre stato assunto un solo servizio in esecuzione e come tipologia di risorse esclusivamente la CPU.

5.4.1 Interfaccia ResourceAvailable

L'interfaccia **ResourceAvailable** legge da un file "resources.json", utilizzato come database,un oggetto JSON con l'elenco delle risorse disponibili e lo invia, quando richiesto, al modulo di auto-orchestrazione. Nel prototipo questo file è statico ed è stato realizzato manualmente: in una versione definitiva andrebbe sostituito con chiamate all'orchestratore dell'infrastruttura ma attualmente l'UniversalNode non fornisce questi dati. Questa interfaccia, in un ambiente reale, dovrebbe inoltre interagire con altri componenti di controllo al fine di limitare le risorse disponibili ai vari servizi in base a logiche commerciali/interne.

```
1    {
2        "type" : "CPU",
3        "id" : "CPU0",
4        "used" : true,
5        "usedBy" : "trcd1"
6    },
7    {
8        "type" : "CPU",
9        "id" : "CPU1",
10       "used" : false,
11       "usedBy" : null
12   },
```

Listing 5.28. Esempio di rappresentazione delle risorse disponibili in JSON ritornato dall'interfaccia **ResourceAvailable**.

I parametri che descrivono le risorse sono:

- **Type**: stringa che rappresenta il tipo della risorsa (ad esempio CPU, MEMORIA, BANDA). In questo prototipo non sono stati specificati altri parametri che descrivono la risorsa (come ad esempio la frequenza della CPU) ma si suppone che sia un valore noto (ad esempio le CPU sono tutte a 3Ghz).
- **Id**: un identificativo univoco per la risorsa.
- **Used**: valore booleano che indica se la risorsa è utilizzata o meno.
- **UsedBy**: stringa che identifica il microservizio che sta utilizzando la risorsa.

5.4.2 Interfaccia ResourceUpdate

Questa interfaccia consente al modulo di auto-orchestrazione di rilasciare o richiedere risorse dall'infrastruttura.

La richiesta contiene un oggetto JSON identico a quello restituito dalla API **ResourceAvailable** ma con alcuni campi modificati in base alle esigenze del servizio (quindi il modulo di auto-orchestrazione va a modificare il campo **Used** mettendolo a false per rilasciare una risorsa o mettendolo a true per richiederla).

Il codice che gestisce questa interfaccia, oltre ad aggiornare il file utilizzato come database, si occupa anche applicare effettivamente gli aggiornamenti richiesti. Questo avviene attraverso la libreria `libvirt`, disponibile anche per PHP.

```
0  <?php
1  $RETRY = 40;
```

```
2  $conn = libvirt_connect("qemu:///system", false,
3              ["username","password"]);
4  if ( $conn === false )
5  {
6          header("HTTP/1.0 500 Errore");
7          die();
8  }
9
10 $resources_json = file_get_contents('php://input');
11 $resources = json_decode($resources_json);
12
13 foreach ( countResources($resources) as $microservice =>
       $resource )
14 {
15     $res = lookup_by_vnf_name($microservice,$conn);
16     foreach ($resource as $type => $number )
17     {
18         if ( $type == "CPU" )
19         {
20             for ($t=0; $t < $RETRY; $t++ )
21             {
22                 $ret = libvirt_domain_change_vcpus($res,
                       $number, 1+8); // live+guest
23                 if ( $ret )
24                 {
25                     break;
26                 }
27                 sleep(2);
28             }
29             if (!$ret)
30             {
31                 header("HTTP/1.0 500 Errore");
32                 die();
33             }
34         }
35     }
36 }
37
38 file_put_contents("resources.json",$resources_json);
```

Listing 5.29. Parte del codice che realizza l'interfaccia ResourceUpdate.

La funzione **countResources** si occupa di convertire l'elenco delle risorse in una mappa a due livelli con la prima chiave che corrisponde all'id del microservizio e come seconda il tipo di una risorse e restituisce il numero di risorse di quel tipo richieste per quel determinato microservizio.

```
0 //$c = numero di risorse del tipo $tipo_risorsa
     utilizzate da $id_microservizio
1 $c = $mappa[$id_microservizio][$tipo_risorsa]
2
3 //$c = numero di risorse del tipo 'CPU' utilizzate da
     'transcoder'
4 $c = $mappa['transcoder']['CPU']
```

Listing 5.30. Esempio di mappa prodotta dalla funzione countResources.

La funzione **lookup_by_vnf_name()** si occupa di ottenere un oggetto `libvirt` che rappresenta la macchina virtuale associata alla VNF partendo dal suo id.

Il pezzo di codice riportato mostra come viene applicato il limite alle CPU utilizzabili da un servizio. Per ogni microservizio viene calcolato il numero di risorse CPU che ha richiesto e viene chiamata la funzione **libvirt_domain_change_vcpus** per limitare il numero massimo di CPU utilizzabili. Questa chiamata può fallire se la macchina virtuale non ha ancora completato il boot e per questo motivo è inserita all'interno di un ciclo che effettua diversi tentativi.

5.4.3 Limitazione della CPU disponibile ad una macchina virtuale

Il metodo con cui effettivamente la funzione **libvirt_domain_change_vcpus** va a limitare la CPU disponibile ad una macchina virtuale è relativamente semplice. L'Universal Node nel caso di virtualizzazione con KVM utilizza un template base per definire le caratteristiche base di tutte le macchine virtuali. Questo template è stato modificato aggiungendo un device virtuale di tipo channel. Questo consente al demone di libvirt, in esecuzione sull'host, di parlare con il demone libvirt-guest che deve essere installato nella macchina virtuale.

Il kernel Linux ha la possibilità di mettere off-line le CPU (tutte tranne la CPU0) semplicemente scrivendo 0 dentro il file virtuale /sys/devices/system/cpu/cpuX/online (dove X rappresenta il numero della CPU). Questo è esattamente quello che fa il demone libvirt-guest sulla macchina guest, quando riceve una richiesta di ridurre il numero di CPU allocate (tramite il device channel) dal demone libvirtd in esecuzione sull'host.

Questo meccanismo è particolarmente efficiente perché non interrompe la funzionalità della macchina virtuale. I principali difetti di questa procedura è l'impossibilità di aggiungere più CPU rispetto a quelle presenti al bootstrap ma anche la

necessaria complicità con il sistema Guest (che deve installare un software aggiuntivo e deve essere leale verso l'host). Questa limitazione è accettabile considerando lo stato prototipale del codice e comunque eventuali comportamenti sleali del Guest risulterebbero individuabili nell'Host.

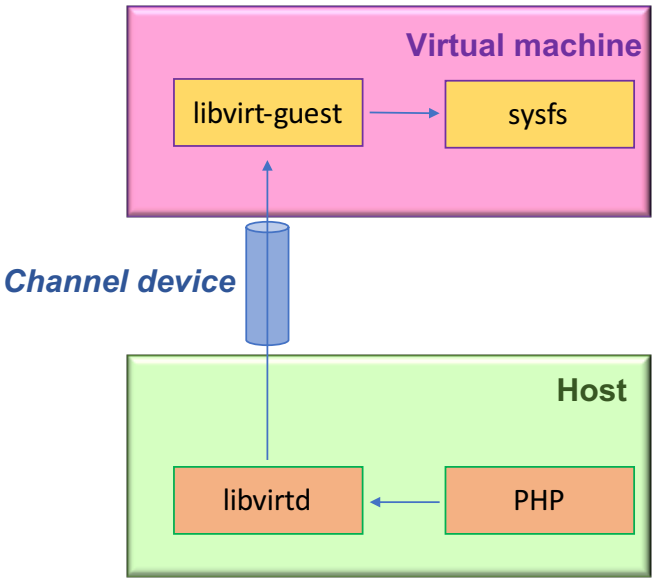

Figura 5.8. Libvirt e la gestione delle cpu nelle macchine virtuali.

5.4.4 Generazione di eventi provenienti dall'infrastruttura

Siccome l'infrastruttura al momento non è in grado di generare eventi, è stato creato un piccolo programma che, se invocato con determinati parametri, ne simula la generazione nell'infrastruttura per un determinato servizio. Questo tipo di implementazione prototipale è comunque compatibile con il nostro scopo di validare il comportamento di un sistema auto-orchestrante in condizioni realistiche. Al fine di realizzare il prototipo, questo applicativo è limitato alla generazione di eventi relativi all'aggiunta o alla rimozione di CPU ad un determinato servizio.

L'implementazione si basa sulla scrittura del file "event.json" con un oggetto JSON molto semplice. Come si può notare dal codice, il software che genera l'evento non modifica il file delle risorse. Questo aggiornamento verrà fatto solo dopo che l'evento sarà correttamente consumato e inoltrato al chiamante (per evitare problemi di sincronismo).

Il formato del file JSON è il seguente:

- **eventId**: identificativo univoco dell'evento.

- **eventType**: tipologia dell'evento che al momento può contenere solo due valori: NONE per indicare che non ci sono eventi (durante il polling) oppure UPDATE_RESOURCES se l'elenco delle risorse disponibili è cambiato.
- **serviceId**: id del servizio a cui bisogna segnalare l'evento.
- **updateNum**: campo utilizzato internamente per generare il nuovo elenco delle risorse.

```
0 {
1     "eventId": 1,
2     "eventType": "NONE",
3     "serviceId": 1,
4     "updateNum": null
5 }
```

Listing 5.31. Oggetto JSON che rappresenta un evento.

```
0  <?php
1  if (count($argv) <= 2 )
2  {
3      echo "USAGE: ".$argv[0]." num_res service_id\n";
4      die();
5  }
6
7  $obj = new stdClass();
8  $obj->eventId = 0;
9  $obj->eventType = "UPDATE_RESOURCES";
10 $obj->updateNum = $argv[1];
11 $obj->serviceId = $argv[2];
12
13 file_put_contents("event.json",json_encode($obj));
14 ?>
```

Listing 5.32. Parte del codice che implementa l'applicativo che genera eventi.

5.4.5 Interfaccia Event

L'interfaccia **countResources** viene interrogata periodicamente dal modulo di auto-orchestrazione per verificare se vi siano nuovi eventi nell'infrastruttura.

Nel prototipo questa interfaccia legge il contenuto del file "event.json", se necessario aggiorna in modo opportuno il database delle risorse, trasmette l'evento letto al chiamante e resetta il contenuto del file.

```php
<?php

//Reading event file
$json = file_get_contents("event.json");
$obj = json_decode($json);

if ( $obj->eventType == "UPDATE_RESOURCES" )
{
    $resources=array();

    for($i=0;$i<$obj->updateNum;$i++)
    {
        $resources[]=[
            "id"=>"CPU".$i,
            "type"=>"CPU",
            "usedBy"=>null,
            "isUsed"=>false
        ];
    }
    $json2=json_encode($resources);

    //Updating resources available
    file_put_contents("resources.json",$json2);

    $obj->eventType = "NONE";
    $obj->updateNum = 0;
    //Updateing event file
    $ret =
       file_put_contents("event.json",json_encode($obj));
}
echo $json;
?>
```

Listing 5.33. Parte del codice che realizza l'interfaccia Event.

E' facilmente intuibile che al momento l'interfaccia degli eventi funzioni unicamente se vi è un solo servizio in esecuzione sull'infrastruttura (non viene fatto nessun controllo sul campo *serviceId*) e inoltre è possibile inserire un unico evento nella coda di quelli da notificare.

5.5 Agent di configurazione per Wowza

Wowza è il software di streaming che è stato utilizzato durante la validazione di questo prototipo. Questo software è stato integrato con l'infrastruttura del Configuration Service, tramite un Agent appositamente sviluppato.

Sviluppare il Configuration Agent per Wowza è stato relativamente semplice. Wowza ha già al suo interno un webservice REST che consente di gestire completamente la sua configurazione e che permette di avere un elevato numero di informazioni sul suo stato (client connessi, flussi utilizzati, statistiche di utilizzo, ecc.).

Il modello utilizzato in questo prototipo è semplice e non completo ma permette di eseguire i test previsti nella validazione. La prima parte (**container interfaces**) è dedicata alla configurazione di rete delle interfacce di rete della macchina virtuale o container che ospiterà il software Wowza. La parte relativa alla configurazione di Wowza in questa versione contiene solo due campi: **"enabled"** che consente di abilitare o disabilitare completamente il componente di transcodifica e **"template"** che permette di indicare il nome del profilo di transcodifica da utilizzare ("transcoder" e "transcoder light", la cui differenza verrà spiegata nel dettaglio nel capitolo 6).

```
102     container transcoder{
103         description "name='transcoder";
104         leaf enabled {
105             description "name='Transcoder enabled or
                    disabled";
106             type boolean
107         }
108         leaf template {
109             description "name='Transcoder used template";
110             type string
111         }
112     }
```

Listing 5.34. Porzione del template YANG che descrive la configurazione di Wowza(transcoder).

```
0
1 class TranscoderService():
2
3         def _get_config(self):
4                 self.wait()
5                 headers = {'Accept': 'application/json' }
6                 try:
```

```
7                           resp =
                               requests.get(self.url,headers=headers)
8                           resp.raise_for_status()
9
10                          json = resp.json()
11                          return json
12
13                  except HTTPError as err:
14                          raise err
15                  except Exception as err:
16                          raise err
17
18          def _get_transcoder_template(self,config):
19                  if  ( "transcoderConfig" in config ):
20                          if ( "liveStreamTranscoder" in
                               config["transcoderConfig"] ):
21                                  return
                                       config["transcoderConfig"]["temp
22                  return ""
23
24          def get_transcoder_template(self):
25                  config = self._get_config()
26                  transcoder =
                       self._get_transcoder_template(config)
27
28                  return transcoder
```

Listing 5.35. Parte della classe Python che implementa l'interfaccia tra il Configuration Agente e Wowza.

5.6 VLC quality meter

VLC quality meter è l'applicativo che è stato sviluppato per effettuare le misurazioni di qualità richieste nella validazione e si basa sulla libreria `libVLC`.

All'avvio il software inizia la riproduzione del flusso ricevuto da riga di comando e comincia a registrare in un file di report in formato CSV alcune informazioni.

Tramite un'opportuna callback, VLC quality meter riceve tutte le informazioni di logging provenienti dai vari moduli del riproduttore multimediale. I messaggi ricevuti sono quindi filtrati tramite delle opportune regole basate sul loro contenuto (ad esempio i messaggio che contengono la stringa "playback in danger of stalling") e vengono salvati nel report. Oltre ai messaggi di logging viene intercettato anche

l'evento *libvlc_MediaPlayerBuffering* che si verfica tutte le volte in cui il buffer di VLC si svuota (e quindi si blocca la riproduzione).

Inoltre, ogni due secondi, vengono scritte nel report le seguenti informazioni: il frame rate della riproduzione (calcolando facendo la differenza tra i frame che erano stati visualizzati nell'iterazione precedente e quelli attualmente visualizzati e dividendo per il tempo trascorso tra le due rilevazioni), le informazioni sulla dimensione delle immagini riprodotte, lo stato del riproduttore musicale (PLAYING, STOPPED, BUFFERING, ecc.), il numero di frame persi (perchè arrivati fuori sequenza) e il bitrate della riproduzione.

Se durante la misurazione VLC entra in uno stato di errore (dovuto a problemi di rete, connessione persa con il server, ecc.) oppure se il frame rate rimane a zero per più di 10 secondi, la riproduzione viene riavviata per evitare che il programma rimanga bloccato senza effettuare misure per troppo tempo.

Il report prodotto da questo software viene successivamente rielaborato da un programma Python per renderlo compatibile con GnuPlot.

```
60          /* Create a new item */
61          m = libvlc_media_new_location (inst, argv[1]);
62          /* Create a media player playing environement */
63          mp = libvlc_media_player_new_from_media (m);
64          libvlc_event_manager_t* eMan =
               libvlc_media_player_event_manager(mp);
65          libvlc_event_attach(eMan,
66              libvlc_MediaPlayerBuffering,
67              onVlcBuffering,NULL);
68          /* play the media_player */
69          libvlc_media_player_play (mp);
70          while (1)
71          {
72              state_media=libvlc_media_get_state(m);
73              state_player=libvlc_media_player_get_state(mp);
74              int err = libvlc_media_get_stats(m,&stats);
75              if (err)
76              {
77                  if(last_check_time != 0 )
78                  {
79                      fps=stats.i_displayed_pictures-last_displayed;
80                      fps=fps/(now_time-last_check_time);
81                  }
82                  libvlc_video_get_size(mp,0,&width,&height);
83                  FILE* fp=fopen("report.csv","a");
```

```
84                  fprintf(fp,"%d;",now_time);
85                  fprintf(fp,"STATUS;");
86                  fprintf(fp,"%d;",stats.i_lost_pictures+lost_sum);
87                  fprintf(fp,"%d;",stats.i_displayed_pictures);
88                  fprintf(fp,"%f;",fps);
89                  fprintf(fp,"%d;",height);
90                  fprintf(fp,"%d;",width);
91                  fprintf(fp,"%f;",stats.f_demux_bitrate*8000);
92                  fprintf(fp,"%s;",state2string[state_media]);
93                  fprintf(fp,"%s;\n",state2string[state_player]);
94                  fclose(fp);
```

Listing 5.36. Parte della codice sorgente del software VLC quality meter (vlc.c).

Capitolo 6

Validazione

Le prove eseguite per la validazione verificano la qualità di un servizio di streaming con e senza il modulo di auto-orchestrazione e utilizzano il prototipo descritto nel capitolo precedente. Le prove sono state effettuate in particolare per verificare se le nuove interazioni con l'infrastruttura aumentino effettivamente la qualità offerta da un servizio.

6.1 Modulo di transcodifica: a cosa serve?

Per riuscire a comprendere in maniera corretta i risultati evidenziati in queste prove è necessario ricordare per quale motivo è indispensabile introdurre un software di transcoding, componente molto esigente in termini di risorse computazioniali, nelle piattaforme di streaming. Il modulo di transcoding si occupa di tre funzionalità: *trans-sizing*, *trans-rating* e *trans-coding*. Il *trans-sizing* rappresenta le funzionalità di ridimensionamento delle dimensioni dei frame. Tramite Il *trans-rating* si riduce il bitrate di una trasmissione che però risulta con una minore qualità ma anche una riduzione della banda richiesta. Con il *trans-coding* si indica l'attività di ricodificare un flusso con un nuovo codec. Questa operazione è necessaria perchè alcuni codec potrebbero non essere disponibili su un dispositivo oppure perchè alcuni codec potrebbero richiedere più risorse computazionali e energia (minore durata della batteria di un dispositivo mobile).

Abilitare il *trans-coding* è dunque fondamentale per ampliare la platea dei dispositivi che possono accedere al servizio e per massimizzare la sua qualità.

I protocolli di streaming adattativi eseguono periodicamente delle misurazioni per stimare la banda disponibile tra il client e il server e se rilevano una banda non sufficiente per il bitrate attuale scelgono automaticamente di utilizzare un flusso con bitrate minore. Il protollo però, per poter funzionare, ha necessità che il server fornisca diversi flussi per la stessa trasmissione ma con differenti bitrate.

Per tutti questi motivi è necessario avere un modulo di transcodifica in un servizio di streaming.

6.2 Banco di prova

Le prove sono state effettuate in un ambiente di test configurato come illustrato nella figura 6.1.

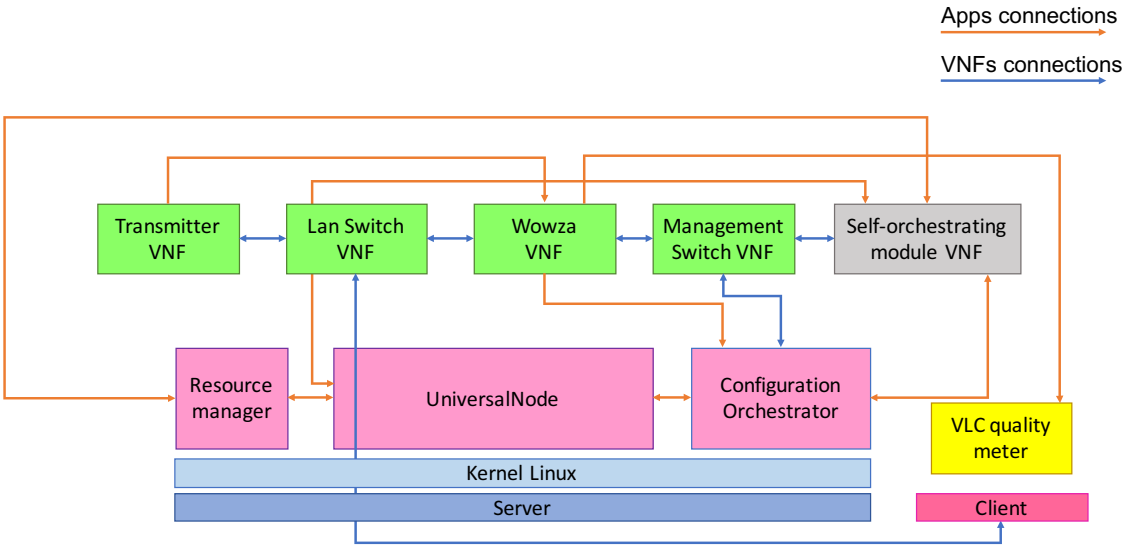

Figura 6.1. Ambiente di validazione.

L'infrastruttura utilizzata per le prove è basta su due macchine. La prima (Intel i7-6700 CPU 3.40GHz, RAM 32GiB e SO Debian 9.2) esegue l'orchestratore Universal Node e rappresenta un'infrastruttura, mentre la seconda è utilizzata come client per lo streaming e effettua delle misurazioni di qualità del servizio.

Le VNF utilizzate sono quattro: due implementano un normale switch layer2 ethernet (uno per la rete di management e uno per far comunicare il client, il trasmettore e Wowza), un trasmettitore video e Wowza come server di streaming.

Il trasmettitore video esegue il software FFmpeg [33] che legge un file su disco e lo trasmette tramite il protocollo RTMP a Wowza simulando il comportamento di una videocamera. Wowza riceve il video trasmesso da FFmpeg, lo converte in diversi formati come previsto dalla configurazione del suo transcoder, e lo rende disponibile agli utenti. Il client si collega ad uno dei flussi resi disponibili da Wowza e esegue il software di misura descritto nel Capitolo 5. É importante inoltre sottolineare che il modulo di transcodifica consuma risorse per produrre i flussi richiesti in output anche se non utilizzati da nessun client

Il trasmettitore invia un flusso video H264 con frame di dimensione 1080x640 pixel con 24 frame/secondo e un bitrate di 1330Kbit/s.

Wowza è configurato con due profili di transcodifica. Il primo, denominato *transcoding* (attivo di default), prevede i seguenti flussi in uscita:

- video: H264 1080x640 1000Kbps, audio: AAC 96Kbps
- video: H263 1080x640 1000Kbps, audio: AAC 96Kbps
- video: VP8 1080x640 1000Kbps, audio: AAC 96Kbps
- video: H264 640x360 200Kbps, audio: AAC 96Kbps
- video: VP8 640x360 200Kbps, audio: AAC 96Kbps
- video: H264 360x240 350Kbps, audio: AAC 96Kbps
- video: VP8 360x240 350Kbps, audio: AAC 96Kbps
- video: H264 284x160 200Kbps, audio: AAC 96Kbps
- video: H264 176x144 150Kbps, audio: AAC 96Kbps

Il secondo profilo, chiamato *transcoding light*, prevede in uscita esclusivamente il flusso con video H264 360x240 200Kbps e audio AAC 96Kbps.

In entrambi i test, il client è configurato per collegarsi al servizio di streaming tramite il protocollo HLS (HTTP live streaming) [34] sul flusso H264 con frame di dimensioni pari a 640x320 pixel e con bitrate pari a 300 Kbits/s.

6.3 Prove effettuate e risultati

Le prove effettuate durante questa validazione vogliono verificare il comportamento di un servizio quando a questo vengono rimosse delle risorse per essere allocate ad un altro servizio con priorità maggiore.

I test prevedono che la macchina virtuale di Wowza (VNF) sia avviata con 4 CPU, successivamente, a causa di una sua logica interna, l'infrastruttura decide di rimuovere 2 CPU che verranno poi restituite in seguito.

I test sono effettuati prima su un servizio senza le capacità di auto-orchestrazione e successivamente su un servizio con queste funzionalità.

Nel caso del servizio auto-orchestrante, nel modello sono state definite due configurazioni valide. La prima, con maggiore QoS, prevedeva di utilizzare la configurazione di default per il transcoder e con la necessità di 4 CPU. La seconda invece, con minore QoS, configurava il transcoder per utilizzare il profilo *transcoder light* e

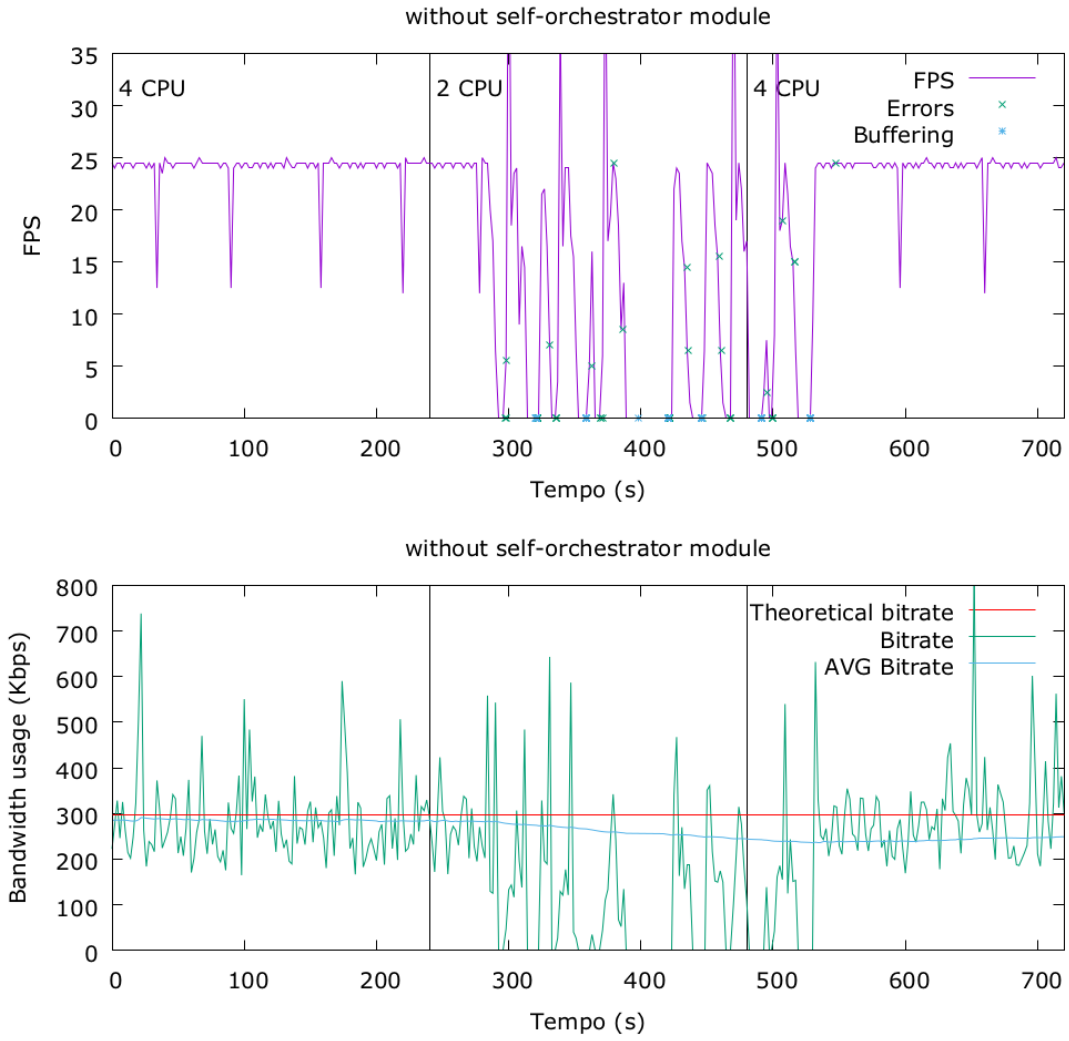

Figura 6.2. Risultati ottenuti senza l'utilizzo del modulo di auto-orchestrazione.

richiedeva solamente 2 CPU.

Come è possibile notare nelle figure 6.2 e 6.3 il comportamento iniziale con 4 CPU (da 0 a 240 secondi) è identico in entrambi i casi. La linea fucsia indica i frame per secondo della riproduzione che rimangono costanti a circa 24fps in tutti e due i grafici (tranne in alcuni punti a causa di un errore introdotto dalla misurazione). La linea rossa rappresenta il bitrate terico della riproduzione, quella verde rappresenta il bitrate misurato e quella azzurra il bitrate medio. Anche in questo caso, nel primo quadrante di entrambi i grafici non si notano differenze.

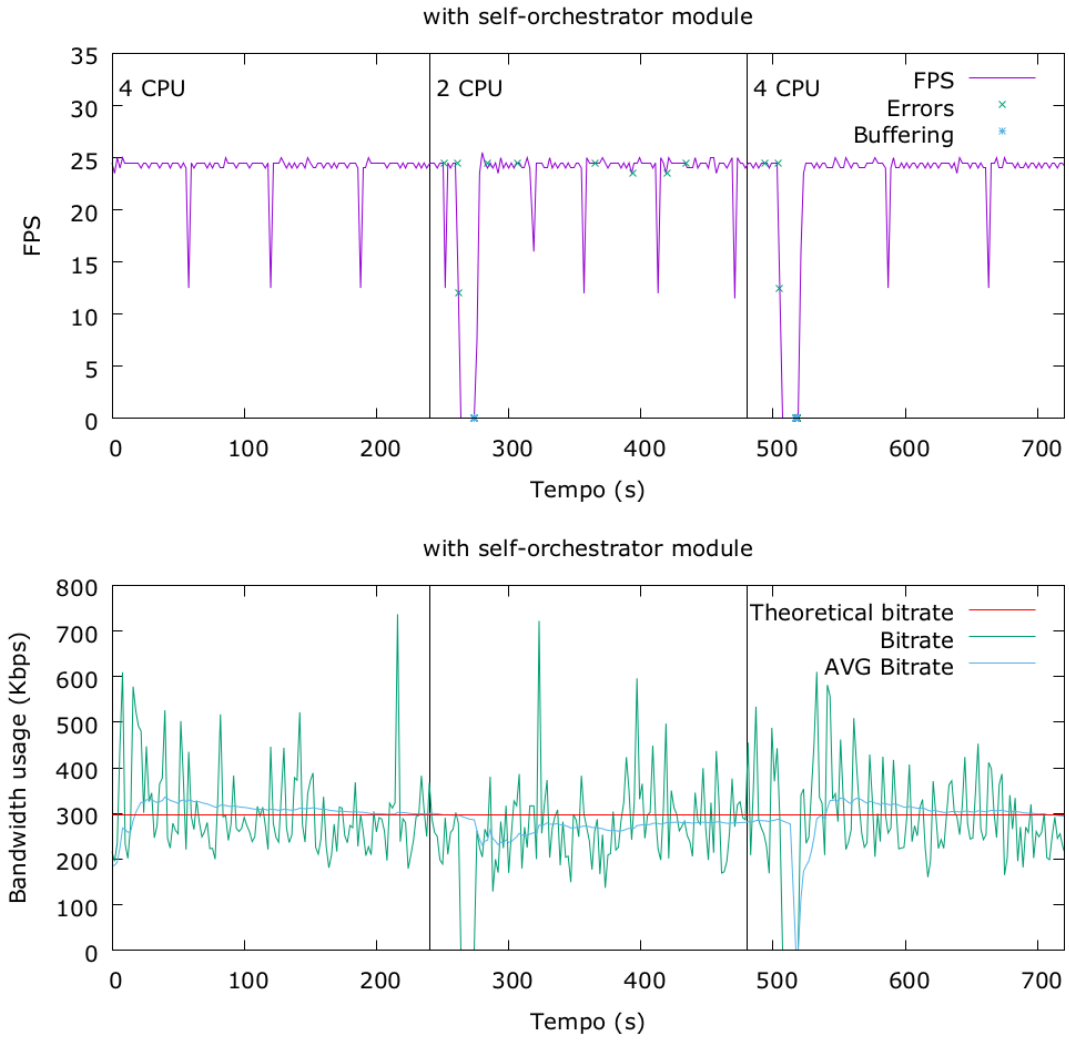

Figura 6.3. Risultati ottenuti con l'utilizzo del modulo di auto-orchestrazione.

Il comportamento cambia a 240 secondi circa, quando l'orchestratore dell'infrastruttura (più precisamente il gestore delle risorse), rimuove 2 CPU alla VNF del transcodificatore. Nella figura 6.2 (servizio non auto-orchestrante), il servizio continua a funzionare senza problemi per altri 20 secondi circa (grazie ai frames già presenti nel playback buffer del client), dopo i quali la qualità inizia a subire un importante degrado: il frame rate raggiunge anche lo zero, ovvero la riproduzione si blocca. Questo comportamento si può anche notare dal grafico del bitrate, che si riduce fortemente in questo quadrante.

Nella figura 6.3, relativa al servizio auto-orchestrante, si vede un comportamento diverso. Dopo pochi secondi dalla riduzione delle CPU il frame rate crolla a zero.

Questo, come indicato nella sottosezione 5.5, è dovuto al cambio del profilo del trancoder che richiede un riavvio dell'applicazione Wowza provocando la disconnessione di tutti i client. Quando il servizio torna disponibile (in qualche secondo) VLC si ricollega. Come si può notare dalla figura 6.3 i frame per secondo, in questo caso, rimangono a circa 25fps e non ci sono interruzioni. Il cambio di profilo del transcoder di Wowza ha ridotto il consumo di risorse richieste dall'applicazione (riducendo anche il numero di flussi disponibili) e ha mantenuto usufruibile il servizio. L'aspetto negativo di questo cambiamento è la probabile esclusione di nuovi futuri client dal servizio se questi non riuscissero a supportare l'unico flusso rimasto disponibile (ad esempio perchè non disponogno di un quantitativo di banda sufficiente). Una possibile correzione di questo comportamento in un'applicazione commerciale potrebbe essere la realizzazione di un modello più complesso che preveda di disattivare solo il minimo numero di flussi necessari, considerando ad esempio quelli meno utilizzati oppure disattivare determinati bitrate e mantenere diversi codec per mantenere la compatibilità con più dispositivi.

Passati 480 secondi dall'inizio della validazione, l'infrastruttura restituisce le 2 CPU rimosse precedentemente al servizio. Nel caso della figura 6.2 si nota che dopo circa 50 secondi il servizio ritorna con una qualità uguale a quella iniziale, il frame rate torna a circa 24fps e il bit rate rimane mediamente intorno ai 300 Kbit/s.

Nella figura 6.3 si nota nuovamente un interruzione, che avviene qualche decina di secondi dopo l'evento generato dall'infrastruttura. E' nuovamente dovuto al riavvio di Wowza richiesto dal modulo di auto-orchestrazione per ripristinare la configurazione originale del transcoder. Dopo questo evento si può notare che il frame rate ritorna al valore di 24fps e il bit rate si attesta sui 300 Kbit/s.

Se consideriamo come qualità del servizio la non interruzione della riproduzione (quindi un frame rate costante) possiamo affermare che lo streaming offerto dal servizio con il modulo di auto-orchestrazione è sicuramente migliore rispetto a quello senza. Le uniche due interruzioni riscontrate nel servizio auto-orchestrante sono dovute ad una limitazione del software utilizzato, che in una situazione reale potrebbe essere sostituito da un altro che non ha questa esigenza.

In una situazione reale, il profilo *transcoder light* potrebbe essere sostituito da un altro, creato appositamente dal modulo di auto-orchestrazione, che prevede di disattivare esclusivamente i flussi che non sono utilizzati da nessun client (compatibilmente con le risorse disponibili).

Capitolo 7

Conclusioni

In questa tesi è stata proposto un nuovo paradigma per la gestione dei servizi. É stato sviluppato anche un prototipo con l'obiettivo di verificare se l'introduzione dei servizi orchestranti comporta dei vantaggi tangibili nella gestione dei servizi. La strategia proposta mira a risolvere le inefficenze delle soluzioni tradizionali nella gestione delle risorse oltre ad aumentare la consapevolezza dello stato globale del sistema nel servizio al fine di permettergli di adattarsi alle situazioni critiche cercando di mantenere la qualità più elevata possibile.

I risultati ottenuti nella validazione sono sicuramente positivi e incoraggiano l'approfondimento delle potenzialità di questo paradigma. Il prototipo sviluppato è certamente una semplificazione di quello che può essere un applicativo definitivo e non considera una serie di situazioni che potrebbero risultare di difficile gestione.

Un esempio potrebbe essere l'implementazione della API che invia l'elenco delle risorse disponibili ad un servizio, interfaccia su cui si basa fortemente il prototipo. Per un orchestratore d'infrastruttura potrebbe essere difficile rispondere a questa interrogazione, sopratutto nel caso d'infrastrutture con molti servizi. L'elenco delle risorse presenti nell'infrastruttura è certamente un informazione nota all'orchestratore, così come a quale servizio sono assegnate quelle utilizzate. Potrebbe però essere difficile notificare a un servizio quante sono effettivamente le risorse che può ancora richiedere. Un orchestratore potrebbe infatti voler offrire la stessa risorsa a più servizi, situazione che potrebbe creare delle problematiche. Per questo motivo questa interfaccia è stata indicata come facoltativa nell'architettura. Senza questa interfaccia, il servizio può fare delle richieste di risorse all'infrastruttura eventualmente verranno rifiutate perchè non disponibili o non utilizzabili per vincoli commerciali.

Volendo semplificare lo sviluppo del prototipo, l'unico protocollo utilizzabile per interrogare lo stato di un microservizio o per modificare la sua configurazione è quello previsto dal *Configuration Service*. Questa scelta ha conseguentemente richiesto lo sviluppo di un Agent per il software Wowza non rispettando interamente le considerazioni a supporto del modulo di auto-orchestrazione *detached*. Infatti una delle ragioni che hanno motivavano l'utilizzo di questa tipologia di moduli era basata

sul fatto che non erano richieste modifiche nel microservizio. Questa situazione è dovuta esclusivamente a esigenze pratiche nello sviluppo del prototipo, che non si dovrebbero verificare nel caso di utilizzo di un software definitivo perchè Wowza espone delle REST API che consentono di controllare ogni aspetto del software.

Oltre a questo aspetto però, sono emerse anche altre difficoltà. A ogni cambiamento della configurazione del software di transcodifica, presente all'interno di Wowza, è necessario riavviare l'applicazione con la conseguente disconnessione di tutti i client. Questo è sicuramente un comportamento non desiderato anche perché, nella logica dei servizi auto-orchestranti, il cambiamento della configurazione è un evento molto frequente. Purtroppo l'unico modo per risolvere questa situazione è modificare il software Wowza rimuovendo questo comportamento. Essendo però un software *closed-source* questa operazione può essere fatta esclusivamente dall'azienda produttrice. Questa problematica si scontra nuovamente con la motivazione a supporto dei moduli di auto-orchestrazione *detached* che prevedeva di non effettuare modifiche al software del microservizio. Questo problema potrebbe parzialmente risolto se l'applicazione utilizzata dal client prevedesse delle riconnessioni in caso di problemi (parzialmente perché l'utente noterebbe comunque delle interruzioni).Questa soluzione potrebbe essere praticabile perché nelle infrastrutture 5G sono previsti servizi *End-To-End* in cui anche l'applicazione client è parte integrante del servizio.

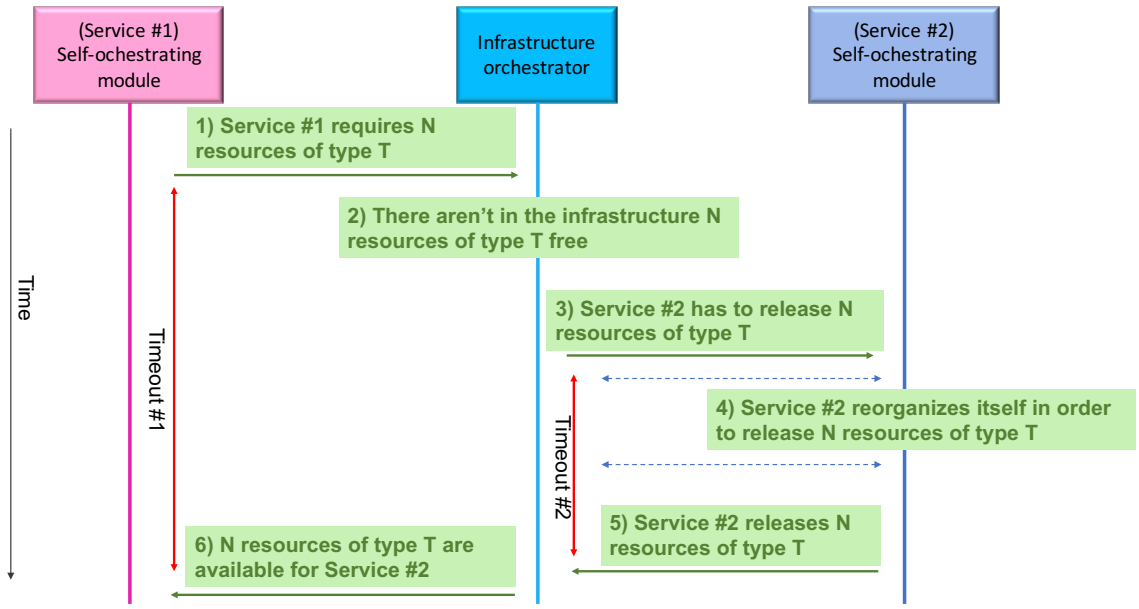

Figura 7.1. Sequenza temporale delle interazioni che avvengono quando un servizio richiede delle risorse che non sono immediatamente disponibili nell'infrastruttura.

La gestione dei timeout è un elemento che non è stato considerato durante l'implementazione del prototipo. Nel caso considerato durante la validazione, un servizio a priorità maggiore richiede ulteriori risorse che però non sono disponibili nell'infrastruttura perché già tutte allocate. L'orchestratore dell'infrastruttura quindi procede a richiedere ad altri servizi di rilasciare un quantitativo di risorse equivalente a quelle richieste in modo da soddisfare l'esigenza del servizio a priorità maggiore. Come ben visibile dalla figura 7.1 è necessario che durante lo sviluppo, sia del modulo di auto-orchestrazione, sia delle modifiche all'interno dell'orchestratore dell'infrastruttura, vengano considerati, implementati e rispettati opportuni timeout. L'aspetto più interessante di questo elemento è che potrebbe essere utilizzato dagli operatori di rete come parametro per definire delle offerte commerciali. Tanto più tempo un servizio si vuole riservare per riorganizzarsi e rilasciare le risorse richieste dall'orchestatore (Timeout#2), tanto più costoso sarà l'utilizzo dell'infrastruttura. Allo stesso modo, più un servizio sarà disposto ad aspettare prima di ottenere le risorse richieste (Timeout#1), tanto più economico sarà il suo piano commerciale.

Gli aspetti più interessanti su cui si potranno indirizzare gli sviluppi futuri di questo paradigma possono riguardare lo sviluppo di un algoritmo per la scelta della configurazione migliore in base allo stato e alle risorse disponibili. Come già indicato nel capitolo 5 l'algoritmo utilizzato nel prototipo è molto semplificato e non esplora tutto lo spazio delle soluzioni possibili, oltre a essere molto inefficiente. Molti miglioramenti sono inoltre fattibili nel formalismo descritto in questa tesi per il modello che rappresenta il servizio.

Bibliografia

[1] D Soldani e A Manzalini. "A 5G Infrastructure for "Anything-as-a-Service"". In: *Journal of Telecommunications System & Management* 3.2 (2014), p. 1.

[2] OpenFog Consortium. *Definition of Fog Computing*. URL: https://www.openfogconsortium.org/resources/#definition-of-fog-computing.

[3] Willem Vereecken et al. "Overall ICT footprint and green communication technologies". In: *Communications, Control and Signal Processing (ISCCSP), 2010 4th International Symposium on*. IEEE. 2010, pp. 1–6.

[4] *ContainerPilot*. URL: https://www.joyent.com/containerpilot.

[5] *Consul*. URL: https://www.consul.io.

[6] *Universal Node public repository*. URL: https://github.com/netgroup-polito/un-orchestrator.

[7] *OpenStack*. URL: https://www.openstack.org.

[8] *Kubernates*. URL: https://kubernetes.io.

[9] *NFFG: Network Function Forwarding Graph*. URL: https://github.com/netgroup-polito/nffg-library/blob/master/README_NFFG.

[10] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38 (2008), pp. 69–74.

[11] *KVM: Kernel-based Virtual Machine*. URL: http://www.linux-kvm.org.

[12] *Docker*. URL: https://www.docker.com.

[13] *DPDK*. URL: http://dpdk.org.

[14] *OpenVSwitch*. URL: http://openvswitch.org.

[15] *xDPd*. URL: https://github.com/bisdn/xdpd.

[16] Gergely Pongrácz. "ERFS: The fastest x86 OpenFlow dataplane in the world". In: *16th IEEE International Conference on High Performance Switching and Routing, HPSR 2015, Budapest, Hungary, July 1-4, 2015*. 2015, pp. 128–129. DOI: 10.1109/HPSR.2015.7483094. URL: https://doi.org/10.1109/HPSR.2015.7483094.

[17] *QEMU: Quick EMUlator*. URL: https://www.qemu.org.

[18] Mark Probst. "Dynamic binary translation". In: *UKUUG Linux Developer's Conference*. Vol. 2002. sn. 2002.

[19] *Libvirt*. URL: https://libvirt.org.

[20] *Configuration orchestrator*. URL: https://github.com/netgroup-polito/frog4-config-orch.

[21] *Configuration agent*. URL: https://github.com/netgroup-polito/frog4-configurable-vnf.

[22] Martin Björklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020. IETF, ott. 2010, pp. 1–173. URL: https://tools.ietf.org/html/rfc6020.

[23] *DoubleDecker public git repository*. URL: https://github.com/Acreo/DoubleDecker.

[24] R. Enns et al. *Network Configuration Protocol (NETCONF)*. RFC 6241. IETF, giu. 2011, pp. 1–113. URL: https://tools.ietf.org/html/rfc6241.

[25] *PyAng*. URL: https://github.com/cmoberg/pyang-json-schema-plugin.

[26] *Pyang JSON schema plugin*. URL: https://github.com/cmoberg/pyang-json-schema-plugin.

[27] *Jsonschema2POJO*. URL: http://www.jsonschema2pojo.org.

[28] *Wowza streaming engine*. URL: https://www.wowza.com.

[29] *VLC*. URL: https://www.videolan.org/.

[30] *LibVLC*. URL: https://www.videolan.org/vlc/libvlc.html.

[31] Dmitry Namiot e Manfred Sneps-Sneppe. "On micro-services architecture". In: *International Journal of Open Information Technologies* 2.9 (2014), pp. 24–27.

[32] Martin Moser. "Declarative scheduling for optimally graceful QoS degradation". In: *Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on*. IEEE. 1996, pp. 86–94.

[33] *FFMPEG*. URL: https://ffmpeg.org.

[34] *HLS*. URL: https://tools.ietf.org/html/rfc8216.