ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

SUPPORTO A MICRO-SERVIZI PER CONTROLLER SDN AD ALTA SCALABILITÀ E AFFIDABILITÀ

TESI DI LAUREA MAGISTRALE

in

Mobile Systems M

Relatore:
Prof. Luca Foschini

Candidato:
Daniel Barattini

Correlatori:
Prof. Paolo Bellavista
Ing. Domenico Scotece

SESSIONE II
ANNO ACCADEMICO 2019/2020

Alla mia Famiglia

Che mi ha sempre sostenuto

In questo lungo e faticoso percorso

Sommario

L'attuale architettura di rete sta mostrando, nel tempo, un numero sempre più elevato di limiti derivanti principalmente dall'integrazione del piano di controllo e del piano di inoltro all'interno dei dispositivi di rete che la compongono. Uno dei più promettenti paradigmi che consente di superare queste limitazioni è SDN (Software-Defined Networking), che si basa sull'idea di estrarre il piano di controllo dai dispositivi di rete per inserirlo all'interno di un nuovo componente logicamente centralizzato: il controller SDN. Le attuali e più popolari implementazioni di controller SDN utilizzano però un'architettura monolitica che limita l'affidabilità e la scalabilità del sistema. L'obbiettivo di questa Tesi consiste quindi nell'implementazione di un prototipo di controller SDN che fornisca supporto a micro-servizi e nella sua relativa sperimentazione.

Indice

Introduzione 1

1. Scenario tecnologico e Software-Defined Networks 4
   1.1 I Piani Della Rete 6
   1.2 Switch Tradizionale 7
      1.2.1 Tabelle di Routing 7
      1.2.2 Funzionamento di uno switch tradizionale 8
   1.3 Architettura SDN 9
      1.3.1 Infrastruttura di rete 9
      1.3.2 Controller 10
      1.3.3 Piano di controllo centralizzato e distribuito 11
      1.3.4 Applicazioni di monitoraggio e controllo 12
   1.4 Protocollo Openflow 13
      1.4.1 Switch Openflow 14
      1.4.2 Flow Entry 16
      1.4.3 Pipeline 18
      1.4.4 Group Table e Meter Table 19
      1.4.5 Messaggi OpenFlow 20
   1.5 Panoramica di controller SDN 21
      1.5.1 ONOS 21
      1.5.2 OpenDaylight 24
      1.5.3 Ryu 25

2. Scomposizione di controller SDN in micro-servizi: letteratura e lavori correlati 27
   2.1 Verso la disgregazione del piano di controllo 27
   2.2 I vantaggi della scomposizione del piano di controllo in micro-servizi 28
   2.3 Progettazione del piano di controllo scomposto 29
   2.4 Meccanismi per la distribuzione di eventi 31
      2.4.1 Distribuzione di eventi publish-subscribe 32
      2.4.2 Distribuzione di eventi point-to-point 33
   2.5 µONOS 34

3. Scomposizione di controller SND in micro-servizi: progettazione 38
   3.1 Tolleranza ai guasti 39
   3.2 Scalabilità dei singoli micro-servizi 40
   3.3 Micro-servizi come VNFs 41
      3.3.1 Open Source MANO 41
      3.3.2 Monitoraggio di VNFs in OSM 43
      3.3.3 Tolleranza ai guasti di VNFs in OSM 43
      3.3.4 Scalabilità di VNFs in OSM 44

4. Scomposizione del controller SDN Ryu in micro-servizi: implementazione 45
4.1 Tecnologie utilizzate 45
4.1.1 Mininet 46
4.1.2 Docker 47
4.1.3 Vagrant 49
4.2 Analisi architetturale e funzionale del controller SDN Ryu 49
4.2.1 Applicazione Ryu 50
4.2.2 Funzionamento di Ryu 51
4.3 Implementazione del prototipo 54
4.3.1 Architettura del prototipo 54
4.3.2 Funzionamento del prototipo 56
4.4 Trasferimento dei micro-servizi all'interno di container Docker 60
4.4.1 Container per il middleware 60
4.4.2 Container per il micro-servizio di routing 61
4.5 Trasferimento dei micro-servizi all'interno di Vagrant box 61
4.5.1 Vagrant box per il middleware 62
4.5.2 Vagrant box per il micro-servizio di routing 62

5. Esperimenti e risultati ottenuti 64
5.1 Ambiente di testing 66
5.2 Comportamento di ping inviati all'interno dell'ambiente di testing 67
5.3 Affidabilità 69
5.3.1 Ambiente di Test 69
5.3.2 Esperimento 70
5.3.3 Risultati ottenuti 70
5.4 Latenza 72
5.4.1 Ambiente di Test 72
5.4.2 Esperimento 73
5.4.3 Risultati ottenuti 74

6. Conclusioni e sviluppi futuri 76

Ringraziamenti 78

Bibliografia e sitografia 79

Introduzione

L'attuale architettura di rete, che costituisce anche le fondamenta di Internet, è basata su dispositivi, quali switch e router, che sono necessari per il trasferimento di dati tra end-point e che integrano al loro interno sia un piano di controllo, che decide come gestire ed indirizzare il traffico di rete, sia un piano dati, che effettua operazioni di inoltro del traffico basandosi sulle decisioni prese dal piano di controllo. L'unione di queste due funzionalità all'interno di un singolo dispositivo di rete porta ad un'integrazione verticale che riduce la flessibilità del sistema e che si pone come ostacolo ai processi di innovazione ed evoluzione dell'intera infrastruttura.

Poiché ogni dispositivo possiede un proprio piano di controllo, questo risulta essere completamente distribuito imponendo ad un operatore che volesse adottare nuove politiche di rete di riconfigurare manualmente ogni singolo componente utilizzando comandi a basso livello e tipicamente vendor-specific: ne risulta una gestione ed una manutenzione della rete molto complessa e costosa che sfocia nell'impossibilità di adattare il comportamento dell'infrastruttura in seguito a guasti o a cambiamenti di carico. Ulteriore conseguenza della distribuzione del piano di controllo è rappresentata dal fatto che ogni dispositivo di rete possiede una visione parziale della stessa ed è, quindi, in grado di prendere decisioni che non sempre risultano ottimali.

L'insieme di tutte queste problematiche porta la rete tradizionale ad essere complessa, costosa e difficile da mantenere. L' infrastruttura corrente risulta inoltre essere eccessivamente statica a fronte della dinamicità delle moderne applicazioni portando ad un'inevitabile e necessaria ricerca di modelli di rete più flessibili e dinamici.

Uno dei più interessanti tra i nuovi paradigmi proposti per superare le limitazioni appena descritte [1] è rappresentato da Software-Defined Networking (SDN) [2] che ha come obbiettivo principale l'estrazione del piano di controllo dai dispositivi di rete in modo da renderli meri esecutori delle decisioni che vengono prese e comunicate da un nuovo elemento di controllo logicamente centralizzato: il controller SDN. Questo componente consente di astrarre l'infrastruttura sottostante rendendola direttamente programmabile da livello applicativo: è infatti possibile imporre nuove politiche di rete o monitorare lo stato del sistema tramite l'utilizzo di applicazioni che interagiscono con il controller. SDN si basa sulla gestione centralizzata effettuata da un componente che ha una visione olistica dell'infrastruttura e che è quindi in grado di prendere decisioni potenzialmente migliori rispetto al caso tradizionale.

Per le comunicazioni tra controller SDN e l'infrastruttura sottostante è considerato oramai uno standard de facto OpenFlow [3], un protocollo di comunicazione aperto che consente di interagire con il piano di inoltro dei dispositivi di rete, come switch e router, per monitorarne lo stato e per definirne il comportamento da intraprendere per la gestione di determinati flussi di dati. Per le comunicazioni tra controller e livello applicativo non è invece ancora riuscito ad imporsi alcuno standard impedendo, nella maggior parte dei casi, la portabilità delle applicazioni.

Le attuali implementazioni di controller SDN si basano su un approccio monolitico che ne riduce l'affidabilità e la dinamicità: essendo racchiusi all'interno di un unico componente software, una personalizzazione delle loro funzionalità in base allo scenario d'uso ed ai requisiti di performance richiesti risulta difficile. L'aggiunta di nuove funzionalità può diventare un processo laborioso, dipendente dall'implementazione del controller e replicare singoli componenti per aumentare la scalabilità e la tolleranza rispetto a guasti del sistema risulta essere non fattibile.

L'obiettivo di questa tesi consiste nell'implementazione e nell'analisi di un prototipo di controller SDN basato su architettura a micro-servizi formato da un componente principale (o nucleo) che racchiude le funzionalità fondamentali del controller e che, essendo altamente flessibile, consente di aggiungerne facilmente

di nuove, anche in maniera dinamica, sotto forma di micro-servizi. Come punto di partenza per l'implementazione del controller è stato utilizzato Ryu, un controller SDN aperto e basato su componenti. Per sfruttare al meglio la nuova architettura proposta, sia il nucleo che gli altri micro-servizi sono stati implementati come Virtualized Network Functions (VNFs) utilizzando dei container Docker (un'unità di software standardizzata contenente il codice da eseguire e tutte le sue dipendenze), sfruttando l'idea alla base della Network Function Virtualization (NFV): virtualizzare le funzionalità di rete in modo da consentire la loro gestione ed orchestrazione all'interno di una Network Functions Virtualization Infrastructure (NFVI) utilizzando un Network Functions Virtualization MANagement and Orchestration framework (NFV-MANO) che permette di effettuare l'istanziazione, anche dinamica, di singoli micro-servizi (e non dell'intero sistema come avveniva nel caso monolitico) con possibilità di replica in modo da garantire maggiore dinamicità ed affidabilità del controller. Per facilitare la sperimentazione con altre tecnologie di virtualizzazione, il nucleo ed i vari micro-servizi sono stati infine racchiusi ciascuno all'interno di una propria Vagrant box (immagine di una macchina virtuale preconfigurata contenente il codice da eseguire e tutte le sue dipendenze) che possono essere definite indipendentemente dal provider di virtualizzazione scelto: VirtualBox, Hyper-V, Docker, VMware, ecc.

Il lavoro di tesi partirà quindi da un'analisi approfondita del paradigma SDN in cui verranno anche presentate le principali implementazioni di controller in modo da identificare quali siano le funzionalità fondamentali da inserire all'interno del nucleo del prototipo, per poi passare, nel capitolo 2, allo studio delle pubblicazioni scientifiche e dei lavori inerenti alla scomposizione del controller SDN in micro-servizi. All'interno del capitolo 4 e del capitolo 5 verranno presentate, rispettivamente, la progettazione e l'implementazione del prototipo. I risultati ottenuti nel corso dei vari esperimenti effettuati saranno invece oggetto di approfondimento nel capitolo 6. Infine, nell'ultimo capitolo, verranno presentate le conclusioni ed i possibili sviluppi futuri.

1. Scenario tecnologico e Software-Defined Networks

SDN è uno tra i nuovi e più promettenti paradigmi di comunicazione che, tramite la separazione del piano di controllo dal piano di inoltro attualmente racchiusi all'interno di un unico dispositivo di rete, permette di ottenere un'infrastruttura più flessibile, dinamica ed aperta. La sua ideazione deriva da anni di ricerca guidati dalla volontà di riprogettare l'infrastruttura di rete corrente in modo da renderla programmabile e di più facile gestione e manutenzione [4]. I vantaggi derivanti dall'utilizzo di questo nuovo paradigma hanno attirato l'interesse dell'industria portando, nel 2011, aziende quali Google, Facebook, Yahoo, Microsoft, Verizon e Deutsche Telekom a fondare la Open Networking Foundation (ONF) con l'obiettivo di promuovere e sviluppare standard aperti in modo da facilitarne l'adozione [5, 6]. Con lo stesso obiettivo è stato fondato nel 2013, da parte della Linux Foundation, l'OpenDaylight Project: un framework opensource guidato dalla community e supportato dalle aziende, che ha avuto tra i suoi membri fondatori Big Switch Networks, Brocade, Cisco, Citrix, Ericsson, IBM, Juniper Networks, Microsoft, NEC, Red Hat e VMware [7, 8].

SDN, usato in combinazione con NFV, consente di ottenere la programmabilità, la flessibilità e la modularità richieste per la creazione di reti logiche (virtuali) end-to-end (E2E) sopra ad una stessa infrastruttura comune. Queste reti logiche, chiamate Network Slices, consentono alla nuova rete 5G di creare slices su richiesta e su misura per la soddisfazione dei requisiti critici (comunicazione ultra-affidabile a bassa latenza, comunicazione IoT massiva e banda larga mobile aumentata) richiesti da un particolare caso d'uso [9].

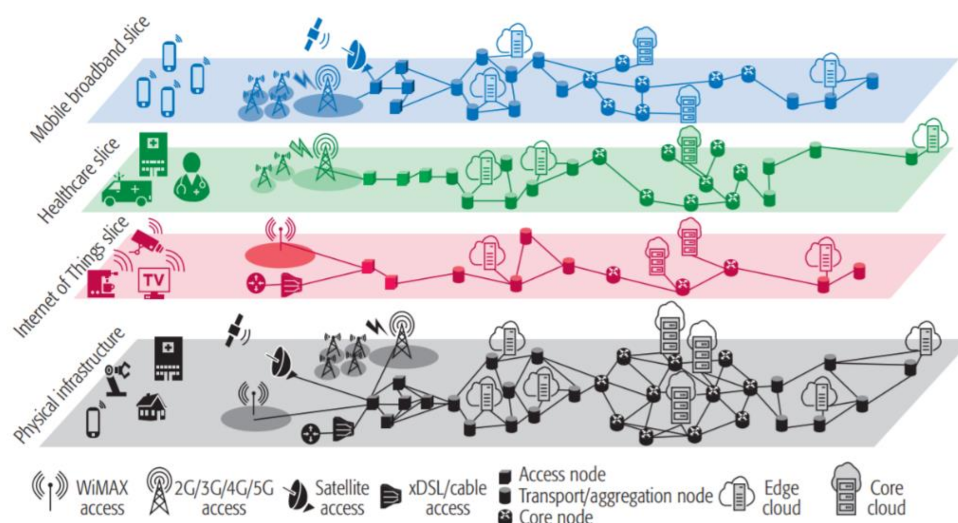

Figura 1- 5G network slices [9]

Per comprendere al meglio i vantaggi derivanti dall'adozione di SDN risulta fondamentale descrivere inizialmente quali siano i piani di cui è composta la rete, in particolare cosa siano i piani di controllo e di inoltro menzionati in precedenza; quindi descrivere il funzionamento di uno switch tradizionale, ossia il più semplice tra i componenti alla base dell'infrastruttura di rete corrente e, successivamente, presentare l'architettura logica ed il funzionamento di un sistema SDN anche tramite la descrizione del protocollo OpenFlow, oramai sua parte integrante.

Poiché, come anticipato nel capitolo precedente, l'obiettivo di questa tesi consiste nello scomporre un controller in micro-servizi, particolare attenzione verrà data al piano di controllo; verranno quindi presentate le sue varianti centralizzate e distribuite assieme alle architetture dei principali controller SDN: Onos, OpenDaylight e Ryu.

1.1 I Piani Della Rete

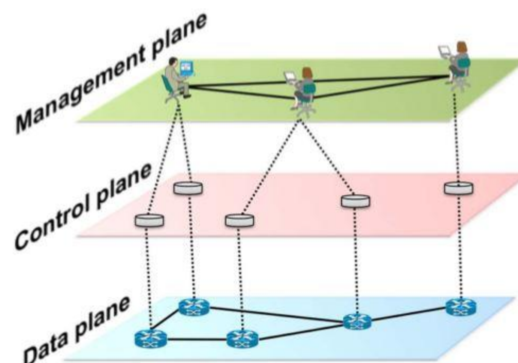

Figura 2 - scomposizione in piani dell'infrastruttura di rete [2]

Un'infrastruttura di rete, come mostrato in figura 2, può essere suddivisa in base alle funzionalità in tre piani distinti: piano dei dati o piano di inoltro (data plane), piano di controllo (control plane) e piano di amministrazione o applicativo (management plane).

All'interno del piano dei dati si posizionano i dispositivi che hanno il compito di inoltrare (in maniera efficiente) il traffico all'interno della rete. Nel piano di controllo sono presenti, invece, protocolli che consentono di manipolare le tabelle di routing integrate all'interno di ciascun dispositivo di rete appartenente al piano dei dati. Il piano di amministrazione, infine, è popolato da applicazioni software che consentono di monitorare e configurare da remoto il piano dei dati tramite interazioni con il piano di controllo.

Per mostrare il funzionamento del sistema complessivo e delle interazioni tra i diversi piani della rete si può considerare, a titolo di esempio, una situazione in cui una nuova policy viene definita da un'applicazione all'interno del piano di amministrazione. Le nuove regole vengono comunicate al piano di controllo che, dopo averle ricevute, le impone all'interno del piano dei dati che, a questo punto, è in grado di applicare la nuova policy inoltrando il traffico ricevuto a seguito delle regole imposte dai piani superiori.

1.2 Switch Tradizionale

Uno switch è un dispositivo di rete di livello 2 del modello ISO/OSI, che ragiona quindi a livello di datalink ed è in grado di ispezionare gli indirizzi MAC (Media Access Control) che identificano univocamente ogni dispositivo connesso alla rete. Il suo compito consiste nell'effettuare l'inoltro di pacchetti ricevuti tramite una porta d'ingresso verso una determinata porta d'uscita.

Per indirizzare il pacchetto verso la porta corretta, all'interno dello switch è presente una tabella di routing che viene gestita ed utilizzata dallo switch stesso. L'integrazione di queste due funzionalità all'interno dello stesso dispositivo equivale all'integrazione del piano di controllo e del piano di inoltro che ha portato la rete tradizionale ad essere complessa, statica e difficile da gestire.

1.2.1 Tabelle di Routing

Una tabella di routing consente ad uno switch di associare l'indirizzo MAC di un dispositivo, ottenuto tramite l'analisi dell'intestazione di un pacchetto ricevuto, ad una determinata porta di rete, utilizzata come porta d'uscita.

Ogni volta che viene ricevuto un pacchetto con indirizzo MAC di destinazione corrispondente ad una entry, questo verrà inoltrato attraverso la porta di rete a cui l'indirizzo MAC è associato.

A tutti gli elementi appartenenti alla tabella di routing è associato un time-out che, allo scadere, ne determinano la rimozione. Questo meccanismo viene utilizzato in modo da rendere l'infrastruttura a cui lo switch appartiene maggiormente flessibile e scalabile.

1.2.2 Funzionamento di uno switch tradizionale

Quando uno switch riceve un pacchetto inviato da un altro dispositivo (che può essere un router, un altro switch o un end-point) presso una sua porta d'ingresso, ne analizza l'intestazione in modo da estrarre l'indirizzo MAC di mittente e destinatario. Successivamente verifica se all'interno della sua tabella di routing siano presenti entry corrispondenti agli indirizzi ottenuti. Nel caso in cui non sia presente nessuna entry relativa all'indirizzo MAC del mittente, questa viene aggiunta in modo da collegare l'indirizzo MAC del mittente alla porta d'ingresso del pacchetto.

Per determinare su quale porta debba essere inoltrato il pacchetto ricevuto viene utilizzato il MAC del destinatario: nel caso in cui esista una entry che associa l'indirizzo MAC del destinatario ad una determinata porta dello switch, il pacchetto può essere inoltrato tramite la porta d'uscita indicata, in caso contrario, lo switch sarà costretto, non potendo determinare quale porta consenta di raggiungere il dispositivo di destinazione, ad inoltrare il pacchetto su tutte le sue porte d'uscita. Se il destinatario del pacchetto corrente invierà, in seguito, una risposta al mittente, lo switch aggiungerà una nuova entry all'interno della sua tabella di routing che gli permetterà di riconoscere anche questo percorso precedentemente sconosciuto.

Come si può evincere dal suo funzionamento, un dispositivo di rete tradizionale possiede solo una visione locale del sistema e non è in grado, quindi, di prendere sempre decisioni di routing che risultino essere le migliori possibili.

Oltre al funzionamento di base appena descritto, è anche possibile aggiungere manualmente da remoto delle entry all'interno di una tabella di routing in modo da poter imporre specifiche policy di rete. Questo processo, però, richiede l'interazione con uno switch per volta ed impone l'utilizzo di comandi di basso livello e dipendenti dal modello e dalla tipologia di switch. Queste limitazioni rendono difficile la gestione della rete e rendono statica l'infrastruttura.

1.3 Architettura SDN

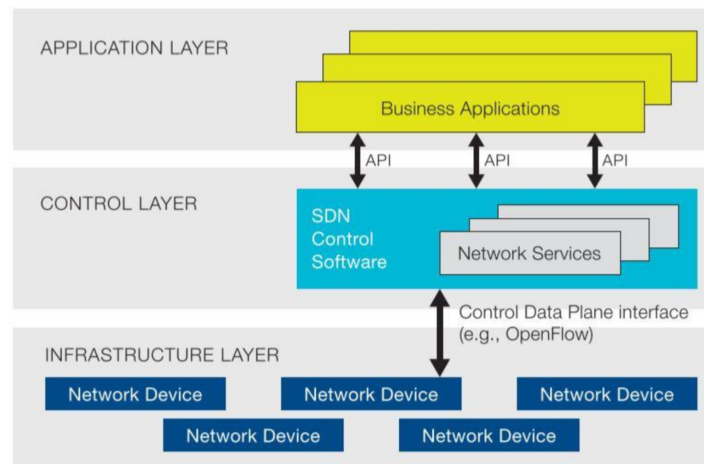

Figura 3 - architettura logica SDN [6]

L'architettura SDN, come mostrato in figura 3, estrae il piano di controllo dai dispositivi di rete in modo da trasferire l'intelligenza dell'infrastruttura all'interno di un controller logicamente centralizzato.

Questa separazione delle funzionalità porta il sistema ad essere composto da tre componenti principali: un'infrastruttura di rete, che costituisce il data plane, un controller logicamente centralizzato, che rappresenta il control plane, ed una o più applicazioni di monitoraggio e gestione, che popolano il management plane.

1.3.1 Infrastruttura di rete

L'infrastruttura di rete è composta da dispositivi, quali switch e router, che hanno il solo compito di inoltrare pacchetti ricevuti in base a regole dettate dal piano di controllo e che rappresentano, quindi, il data plane. Queste regole possono essere imposte da un controller logicamente centralizzato tramite manipolazione di particolari tabelle di routing memorizzate all'interno dei dispositivi di rete, come avviene utilizzando il protocollo OpenFlow che verrà approfondito successivamente.

1.3.2 Controller

Il controller è un componente logicamente centralizzato che rappresenta il cuore dell'architettura SDN. Il suo compito principale consiste nel fare da intermediario tra le applicazioni (che vogliono monitorare e controllare l'infrastruttura di rete) e i dispositivi di rete (che inoltrano il traffico in base alle regole imposte).

Dal punto di vista del controller sono quindi necessarie diverse interfacce di comunicazione: un'interfaccia per comunicare con i dispositivi di rete (southbound interface), una per comunicare con le applicazioni (northbound interface) e, nel caso di controller distribuito, un'interfaccia per comunicare con gli altri controller (westbound ed eastbound interface). L'unica tra tutte le interfacce in cui è riuscito ad imporsi uno standard, anche se de facto, è la southbound interface che viene tipicamente implementata utilizzando il protocollo OpenFlow.

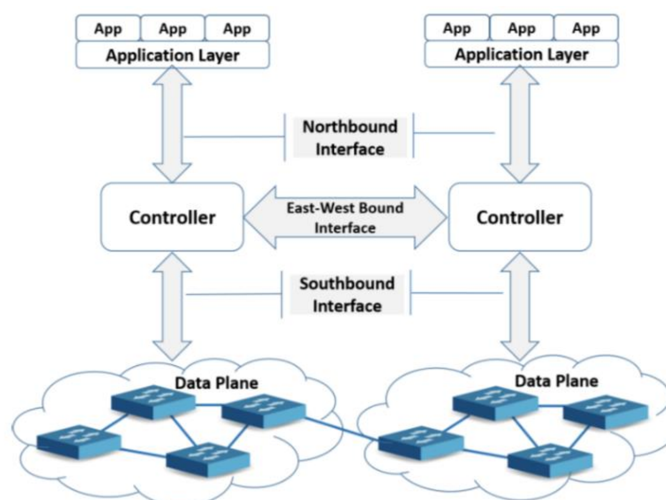

Figura 4 - interfacce controller SDN [10]

Tramite l'utilizzo della southbound interface, un controller può installare una entry all'interno della tabella di routing di un dispositivo di rete. Tale installazione può avvenire con due modalità: proattiva o reattiva.

Seguendo l'approccio proattivo, una entry può essere preinstallata all'interno di una tabella di routing prima ancora della ricezione del primo pacchetto appartenente ad un flusso dati. I vantaggi derivanti da questo modello consentono di avere un ritardo

d'istallazione trascurabile ed una riduzione delle interazioni tra infrastruttura e controller.

Utilizzando, invece, il modello reattivo, una entry può essere installata solamente in seguito alla ricezione del primo pacchetto appartenente ad un nuovo flusso, dopo che questo è stato inoltrato al controller dal dispositivo di rete che l'ha ricevuto. Il modello reattivo comporta un elevato round-trip-time ma consente una maggior flessibilità per la presa di decisioni relative ad un flusso in quanto permette di tenere in considerazione requisiti di Quality of Service (QoS) e condizioni relative al traffico di rete.

Per installare una nuova entry, un controller verifica inizialmente le policy imposte dal livello applicativo, calcola quindi il percorso che il flusso dovrà seguire in termini di dispositivi di rete, e procede infine con l'installazione di una nuova entry per ogni switch appartenente a tale percorso [11].

Essendo logicamente centralizzato, il controller consente il monitoraggio ed il controllo del data plane da parte del management plane tramite interazioni con un singolo punto di controllo. In questo modo risulta possibile superare le difficoltà di gestione e di manutenzione emerse nel caso tradizionale.

È da sottolineare, però, che un controller SDN è solo logicamente centralizzato: fisicamente può infatti essere centralizzato o distribuito. Questa seconda variante è tipicamente preferibile in quanto consente di evitare l'effetto collo di bottiglia derivante dall'utilizzo di una singola istanza.

1.3.3 Piano di controllo centralizzato e distribuito

L'implementazione del piano di controllo può seguire un approccio centralizzato, tramite l'utilizzo di un singolo controller, oppure un approccio distribuito, tramite l'utilizzo di più controller che possono essere coordinati tra loro.

Il singolo controller SDN fisicamente centralizzato rappresenta la più semplice delle possibilità ma, a differenza del caso distribuito, costituisce un single point of failure e comporta bassa scalabilità ed affidabilità del sistema.

Poiché a fronte di una complessità maggiore un approccio distribuito porta a superare i problemi legati al modello centralizzato, negli ultimi anni la ricerca sta procedendo in questa direzione.

È possibile effettuare una classificazione di controller SDN distribuiti [12, 13] in base all'approccio utilizzato:

- *Fisicamente Distribuito*: Il control plane è formato da più controller fisicamente distribuiti. Questo approccio è usato tipicamente per collegare diversi domini: una rete di grosse dimensioni può essere, per esempio, suddivisa in reti più piccole, ognuna controllata da un controller locale. Tutti i controller possiedono la stessa visione globale della rete e si aggiornano a vicenda periodicamente
- *Logicamente Distribuito*: Questo approccio è molto simile al precedente. L'unica differenza risiede nel fatto che ciascun controller non aggiorna gli altri periodicamente, ma solamente nel caso in cui si verifichino modifiche all'interno della rete
- *Gerarchicamente Distribuito*: Il control plane è formato da uno o più livelli di controller disposti secondo una struttura gerarchica. I controller, in questo caso, possono essere specializzati nell'eseguire una determinata funzionalità e possono prendere decisioni avendo a disposizione solamente una visione parziale della rete. La radice del sistema può rappresentare in questo caso un single point of failure
- *Ibrido*: Il piano di controllo può anche essere sviluppato mescolando alcuni tra gli altri approcci descritti in modo da combinarne i benefici e da evitarne le limitazioni

1.3.4 Applicazioni di monitoraggio e controllo

Le applicazioni di monitoraggio e controllo si posizionano all'interno del management plane e possono essere sviluppate con l'obiettivo di monitorare lo stato della rete o di imporre nuove policy tramite interazioni con il piano di controllo. Non esistendo alcuno standard per le interazioni tra management plane e control

plane, la maggior parte delle applicazioni viene sviluppata specificatamente utilizzando le API messe a disposizione da una determinata implementazione di controller non risultando, quindi, spesso compatibile con altre implementazioni.

1.4 Protocollo Openflow

OpenFlow è un protocollo aperto proposto inizialmente da un gruppo di ricercatori presso l'università di Stanford come un modo che consentisse di testare ed utilizzare protocolli sperimentali all'interno delle reti appartenenti ai campus [14] e che si è oramai imposto come standard de facto per le interazioni tra controller SDN e rete sottostante. Attualmente la sua standardizzazione (correntemente alla versione 1.5.1 [15]) è seguita dalla Open Networking Foundation (ONF).

Il funzionamento del protocollo è basato sul concetto di flusso, rappresentato, ad esempio, da tutti i pacchetti aventi stesso mittente o stesso destinatario espresso sotto forma di indirizzo MAC o indirizzo IP all'interno dell'intestazione, oppure da tutti i pacchetti ricevuti da una stessa porta d'ingresso o, ancora, da tutti i pacchetti relativi ad una specifica connessione TCP.

L'adozione di OpenFlow consente di manipolare dall'esterno particolari tabelle di routing, definite flow table, presenti all'interno di ciascun dispositivo di rete garantendo accesso diretto al piano d'inoltro. Ogni elemento appartenente ad una flow table, definito flow entry, definisce in primo luogo delle regole di corrispondenza che vengono utilizzate per determinare il flusso di appartenenza di un pacchetto ricevuto e, in base al riscontro ottenuto, come tale pacchetto debba essere gestito. Tramite l'utilizzo di OpenFlow è possibile ottenere quindi un controllo sull'infrastruttura altrimenti impossibile da raggiungere con il modello tradizionale: i flussi di dati scambiati tra due end point sono infatti costretti, in quest'ultimo caso, a seguire lo stesso percorso di rete indipendentemente dai diversi requisiti richiesti.

Oltre al concetto di flow table, all'interno del protocollo vengono definiti anche un insieme di messaggi che, scambiati con i dispositivi di rete, consentono ad un

controller di analizzarne lo stato e di definire nuove regole di routing. Tali dispositivi devono però fornire supporto al protocollo OpenFlow: l'esempio più semplice consiste in un'evoluzione dello switch tradizionale chiamata switch OpenFlow.

1.4.1 Switch Openflow

Uno switch OpenFlow, come definito nell'ultima versione (1.5.1) della specifica [15] e come mostrato nella figura 5, è composto da una o più flow table (ordinate in modo da creare una pipeline per il processamento dei pacchetti ricevuti) e da una group table utilizzate per l'analisi e la gestione di pacchetti. Le comunicazioni tra switch e controller e la gestione dello switch da parte del controller vengono effettuate utilizzando canali dedicati e sicuri (è possibile criptare le comunicazioni utilizzando il Transport Layer Security - TLS) tramite scambio di messaggi specificati all'interno del protocollo OpenFlow; sfruttando questo modello di comunicazione un controller è in grado di aggiungere, aggiornare o eliminare flow entry all'interno di una flow table.

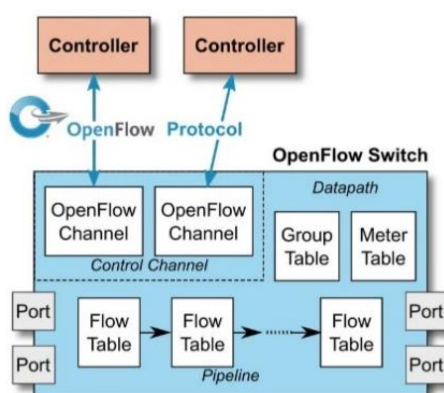

Figura 5 - componenti principali di uno switch OpenFlow [15]

Uno switch OpenFlow può essere di due tipi:

- *OpenFlow-only*: dispositivi che forniscono supporto esclusivamente al protocollo OpenFlow. Sono in grado di processare pacchetti unicamente all'interno della pipeline composta da flow table, definita pipeline OpenFlow

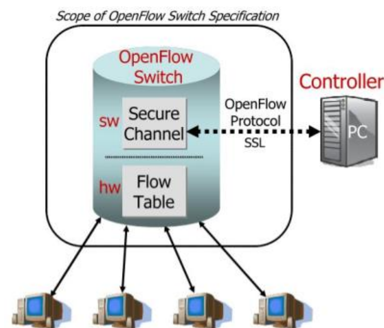

Figura 6 – struttura logica di uno switch OpenFlow-only [3]

- *OpenFlow-hybrid*: switch tradizionali estesi con funzionalità OpenFlow, composti quindi da una pipeline di processamento tradizionale e da una OpenFlow. Questa tipologia di switch deve anche fornire un meccanismo di classificazione che sia in grado di selezionare la pipeline sulla quale debba essere processato il traffico in ingresso

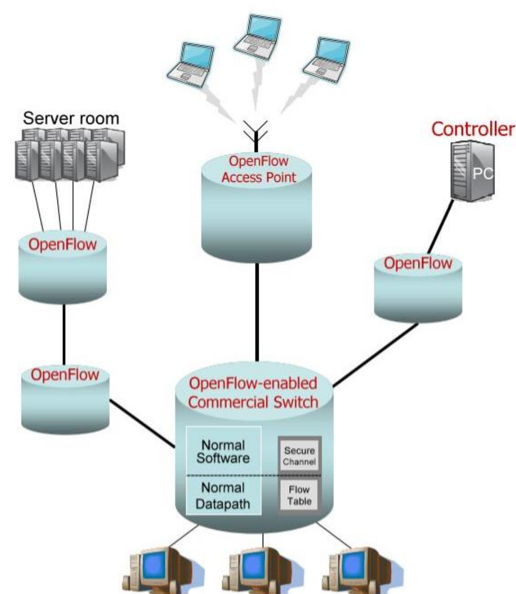

Figura 7 – struttura logica di uno switch OpenFlow-hybrid [3]

Così come uno switch tradizionale, anche uno switch OpenFlow è dotato di porte, fisiche o logiche, utilizzate per trasferire pacchetti verso altri dispositivi di rete. Considerando per semplicità uno switch OpenFlow-only, un pacchetto viene ricevuto tramite una porta d’ingresso, viene processato all’interno della pipeline OpenFlow e, se indicato all’interno dell’insieme di istruzioni da eseguire sul flusso

a cui il pacchetto appartiene, viene inoltrato verso un altro dispositivo tramite una porta d'uscita.

Uno switch OpenFlow possiede, oltre a porte standard, alcune porte riservate associate a particolari funzionalità:

- *ALL*: rappresenta tutte le porte che uno switch può utilizzare per l'inoltro di un pacchetto (esclusa la porta d'ingresso del pacchetto stesso)
- *CONTROLLER*: rappresenta il canale di comunicazione con un controller e può essere utilizzata sia come porta d'ingresso che come porta d'uscita. I pacchetti inoltrati verso il controller vengono prima incapsulati all'interno di messaggi di tipo packet-in e, successivamente, trasferiti tramite protocollo di comunicazione OpenFlow
- *TABLE*: rappresenta il punto d'ingresso della pipeline OpenFlow.
- *IN_PORT*: rappresenta la porta da cui è stato ricevuto il pacchetto
- *ANY*: rappresenta una porta qualunque dello switch.
- *LOCAL* (opzionale): consente ad entità remote di interagire con lo switch mediante rete OpenFlow
- *NORMAL* (opzionale): rappresenta l'inoltro di un pacchetto utilizzando la pipeline tradizionale di uno switch ibrido
- *FLOOD* (opzionale): rappresenta il flooding eseguito sulla pipeline tradizionale di uno switch ibrido

1.4.2 Flow Entry

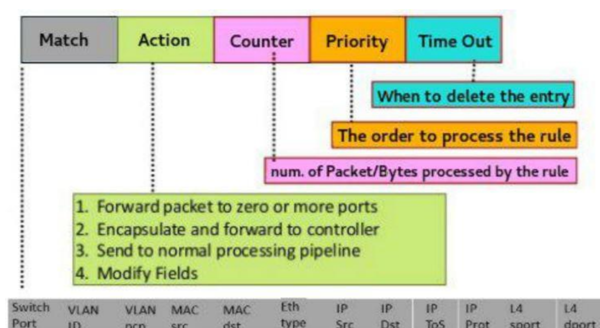

Figura 8 - anatomia di una flow entry [16]

Ogni flow entry è composta da:

- *Campi di corrispondenza* (*match fields*): determinano quale sia il flusso di appartenenza di un pacchetto in base a filtri quali porta di ingresso o valori assegnati a campi presenti all'interno della sua intestazione (ad esempio indirizzo MAC, indirizzo IP, etc.)
- *Priorità (priority)*: se all'interno di una stessa flow table si verificano corrispondenze con più flussi viene selezionato quello con priorità maggiore
- *Contatori (counters)*: contengono informazioni statistiche e rappresentano tipicamente il numero di pacchetti ed il numero di byte che sono stati riconosciuti come appartenenti ad un flusso. Sono definiti anche altri tipi di contatore
- *Istruzioni (instructions)*: consentono di modificare l'insieme di azioni da eseguire su un pacchetto riconosciuto come appartenente ad un flusso o di modificare il suo processamento all'interno della pipeline
- *Timeouts*: tempo massimo trascorso dall'installazione o trascorso dall'ultimo riscontro ottenuto prima che una entry venga rimossa da una flow table
- *Cookie*: insieme di dati opachi definiti e gestiti da un controller. Non vengono utilizzati durante il processamento del pacchetto
- *Flags*: modificano le modalità di gestione di una flow entry

Una flow entry che omette i valori associati a tutti i campi e possiede priorità pari a 0 viene utilizzata solo quando un pacchetto ricevuto non ha riscontri con nessun'altra flow entry appartenente ad una flow table: per questo motivo viene chiamata table miss. Una delle azioni che è possibile intraprendere in questo caso consiste nell'incapsulare il pacchetto non riconosciuto all'interno di un messaggio packet-in e nell'inoltrarlo, tramite un canale dedicato, ad un controller in modo che quest'ultimo possa analizzarlo ed installare una flow entry all'interno di una flow table, se necessario, affinché il dispositivo di rete associato sia in grado successivamente di riconoscere il nuovo flusso.

Le istruzioni che si possono specificare all'interno di una flow entry sono composte da un set minimo che può essere esteso. Le azioni che un qualsiasi dispositivo OpenFlow deve mettere a disposizione sono:

- *Inoltro di un pacchetto appartenente ad un flusso verso una determinata porta*: consente la trasmissione di pacchetti ricevuti verso altri dispositivi collegati
- *Incapsulamento ed inoltro di un pacchetto appartenente ad un flusso verso un controller*: tipicamente utilizzata per inoltrare il primo pacchetto ricevuto appartenente ad un flusso sconosciuto in modo da consentire ad un controller di effettuare una sua analisi e di scegliere se debba essere aggiunto o meno un nuovo flusso all'interno di una flow table. In alcuni esperimenti può anche essere utilizzata per inviare pacchetti al controller affinché sia quest'ultimo ad effettuarne il processamento
- *Scarto del pacchetto*: può essere usata per aggiungere funzionalità di sicurezza ad un dispositivo OpenFlow; ad esempio in modo da evitare attacchi di tipo DoS (Denial of Service) o per ridurre il numero di messaggi inoltrati durante trasmissioni broadcast

Switch di tipo OpenFlow-hybrid possono mettere a disposizione un'azione aggiuntiva che consente di trasferire pacchetti dalla pipeline OpenFlow a quella tradizionale tramite l'utilizzo di porte riservate quali NORMAL e FLOOD.

1.4.3 Pipeline

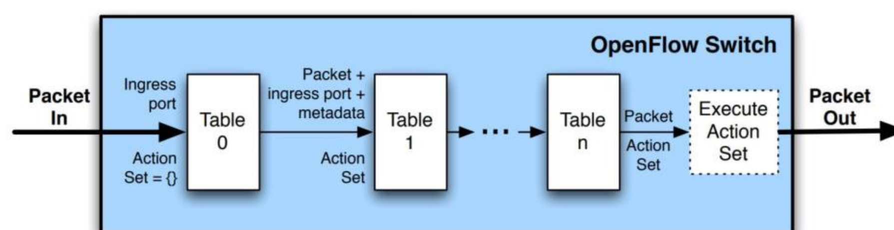

Figura 9 – flusso dei pacchetti all'interno della pipeline OpenFlow [17]

La verifica di appartenenza di un pacchetto ad un determinato flusso comincia cercando un riscontro con i campi di corrispondenza definiti all'interno della prima flow table della pipeline, chiamata flow table 0. Nel caso in cui ne venga trovato almeno uno viene selezionato il flusso con priorità maggiore, vengono quindi eseguite le istruzioni ad esso associate (elaborazioni del pacchetto, aggiornamento dell'insieme di azioni o aggiornamento dei metadati) ed infine, se indicato, il pacchetto ed i relativi metadati prodotti durante la sua elaborazione vengono passati ad una successiva flow table andando a formare quella che viene definita pipeline OpenFlow.

Quando una flow table non esprime la volontà di trasferire il pacchetto elaborato presso una flow table successiva, vengono eseguite tutte le azioni definite all'interno dell'insieme corrispondente, aggiornato durante le varie fasi di processamento, che possono portare all'inoltro del pacchetto elaborato tramite una determinata porta d'uscita verso altri dispositivi connessi alla rete. In caso non si sia trovato alcun riscontro utilizzando la prima flow table, viene selezionata la flow entry table miss che specifica quali azioni debbano essere intraprese in questo caso. Se nessuna entry di tipo table miss è definita all'interno della tabella, allora il pacchetto corrente viene semplicemente scartato.

1.4.4 Group Table e Meter Table

In aggiunta alle flow table, all'interno di uno switch OpenFlow sono anche presenti, come descritto inizialmente, una group table ed una meter table.

La group table consente di raggruppare un insieme di operazioni che possono essere applicate a diversi flussi in un unico punto a cui una flow entry può puntare. Il suo utilizzo consente di specificare azioni di inoltro addizionali.

La meter table consente di eseguire operazioni su un determinato flusso in base alle prestazioni misurate.

1.4.5 Messaggi OpenFlow

OpenFlow, oltre a specificare quale debba essere la struttura logica di uno switch, definisce un protocollo di comunicazione utilizzato per le interazioni tra controller e switch. I messaggi supportati sono suddivisi in tre categorie: messaggi controller-to-switch, messaggi asincroni e messaggi simmetrici.

I messaggi controller-to-switch vengono inviati da un controller e possono richiedere, o non richiedere, una risposta da parte di uno switch. Le principali tipologie di messaggio presenti all'interno di questa categoria sono:

- *Features*: un controller può inviare una features request per richiedere l'identità e le funzionalità supportate da uno switch. In questo caso lo switch che ha ricevuto il messaggio deve rispondere con una features reply, comunicando le informazioni richieste. Questo scambio di messaggi avviene tipicamente durante l'inizializzazione del canale di comunicazione OpenFlow che collega controller e switch
- *Configurazione*: consente ad un controller di richiedere o modificare i parametri di configurazione di uno switch
- *Modifica-Stato*: tramite questa tipologia di messaggio un controller è in grado di modificare lo stato di uno switch, ad esempio aggiungendo o rimuovendo flow entry all'interno di una flow table
- *Leggi-Stato*: permette ad un controller di raccogliere informazioni relative ad uno switch, per esempio la configurazione corrente, statistiche e funzionalità
- *Packet-out*: messaggi utilizzati da un controller per inoltrare un pacchetto su una determinata porta di uno switch

I messaggi asincroni vengono inviati da uno switch per aggiornare un controller sui cambiamenti relativi alla rete o al proprio stato. In questa categoria i messaggi principali sono:

- *Packet-in*: consente di delegare l'analisi di un pacchetto al controller. Questa tipologia di messaggio può essere utilizzata, ad esempio, quando non vengono trovati riscontri relativi ad un pacchetto all'interno di una flow table
- *Flusso-Rimosso*: consente di aggiornare un controller sull'avvenuta rimozione di una flow entry all'interno di una flow table
- *Stato-porta*: permette di informare il controller su un cambiamento apportato su una determinata porta

I messaggi simmetrici possono essere inviati sia da un controller che da uno switch. Appartengono a questa categoria i messaggi:

- *Hello*: sono i primi messaggi che vengono scambiati durante l'inizializzazione del canale OpenFlow e consentono la negoziazione della versione del protocollo da usare durante le successive comunicazioni
- *Echo*: una echo request può essere inviata alla controparte sia da un controller che da uno switch. Se il ricevente non risponde con una echo reply significa che non è più attivo
- *Error*: messaggi utilizzati per segnalare un errore alla controparte

1.5 Panoramica di controller SDN

Con l'obbiettivo di individuare quali siano i componenti fondamentali di un controller SDN, verranno analizzate le sue più note implementazioni: ONOS, OpenDaylight e Ryu.

1.5.1 ONOS

ONOS (Open Network Operating System) [18] è un progetto open source guidato dalla Open Networking Foundation (ONF) con l'obiettivo di creare un sistema operativo SDN utilizzabile in ambiente industriale. Per conseguire tale obiettivo, ONOS è stato progettato in modo da fornire alta scalabilità, affidabilità ed elevate prestazioni.

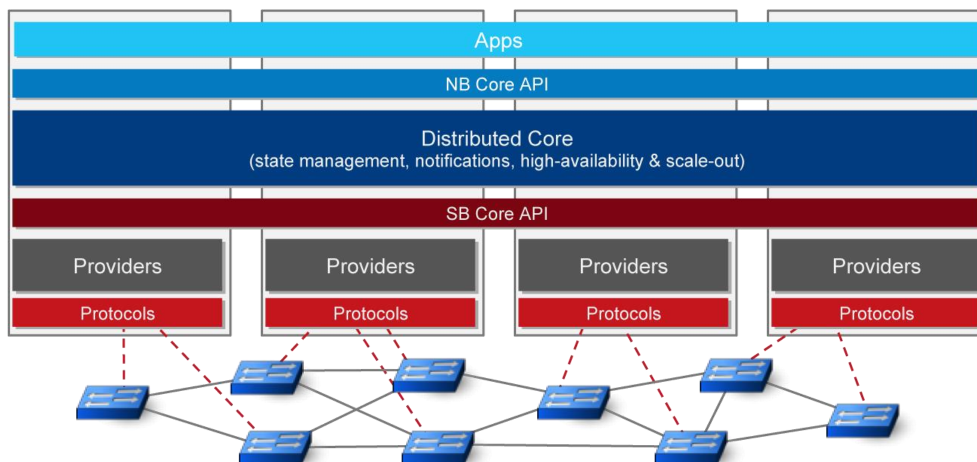

Figura 10 - architettura ONOS [18]

L'architettura alla base di ONOS, mostrata in figura 10, è suddivisa in strati che riprendono l'architettura logica SDN. Seguendo un approccio top down, troviamo:

- *Le applicazioni*: applicazioni per il monitoraggio ed il controllo della rete sviluppate dagli utenti
- *Northbound interface*: consente alle applicazioni di interagire con il nucleo di ONOS tramite le API che quest'ultimo mette a disposizione
- *Nucleo distribuito*: rappresenta il piano di controllo, ossia il vero e proprio controller SDN
- *Southbound interface*: consente le comunicazioni tra nucleo ed infrastruttura di rete sottostante. ONOS fornisce diversi adattatori (OpenFlow, NetConf, OVSDB, TL1, etc.) che permettono al sistema di essere indipendente da protocolli di southbound

Per poter garantire scalabilità ed affidabilità, ONOS utilizza un'architettura fisicamente distribuita: vengono installate diverse istanze di ONOS in modo da formare un cluster simmetrico. Queste sono tutte identiche e ciascuna di esse è in grado di sostenere il carico di lavoro di una qualunque altra in caso di guasto o per effettuare bilanciamenti di carico.

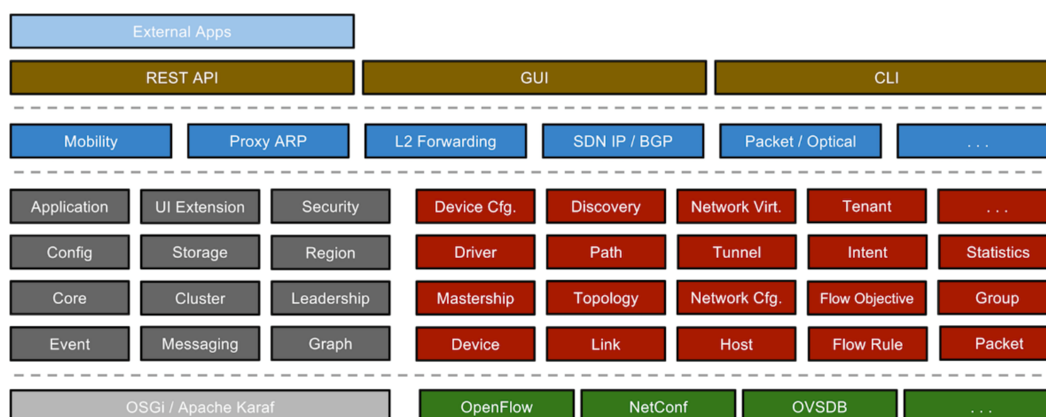

Figura 11- sottosistemi ONOS [19]

I diversi strati di ONOS, come mostrato in figura 11, sono formati da un insieme di moduli (o sottosistemi): l'utilizzo di un'architettura modulare consente di poter aggiungere nuove funzionalità, sotto forma di moduli, a quelle che vengono inizialmente offerte dal nucleo del sistema.

La progettazione di un nuovo modulo da inserire all'interno del nucleo ruota attorno a due componenti tra loro cooperanti: il componente Manager, che si occupa del flusso di lavoro nord/sud, ed il componente Store, che si occupa invece del flusso di lavoro est/ovest, incluse le comunicazioni tra cluster. Un evento emesso all'interno di un cluster viene propagato, tramite il componente Store locale, sia al componente Manager locale che ai componenti Store presenti all'interno degli altri cluster. Un componente Manager, in base alla natura dell'evento, può distribuirlo a tutti i listener locali oppure eseguire azioni sull'ambiente [20].

Il kernel, i moduli che compongono il nucleo e le applicazioni ONOS sono tutti scritti in Java ed implementati come bundles che è possibile caricare all'interno di container Karaf OSGi, dove OSGi è un sistema Java basato su componenti che consente di installare ed eseguire dinamicamente i diversi moduli all'interno di una Java Virtual Machine (JVM) [21].

1.5.2 OpenDaylight

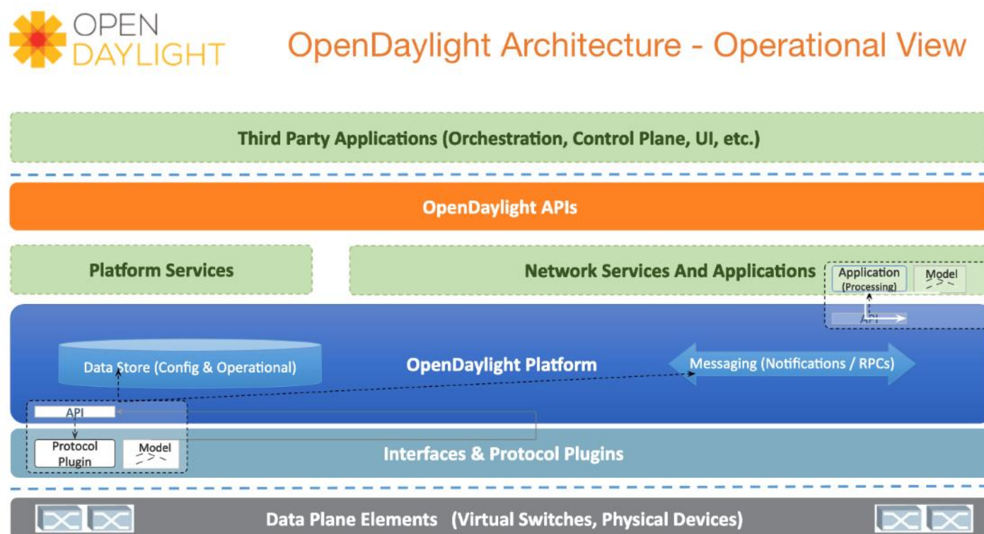

Figura 12 - architettura OpenDaylight [22]

OpenDaylight (ODL) è un progetto open source fondato dalla Linux Foundation con l'intento di produrre un controller SDN utilizzabile in ambiente industriale. L'OpenDaylight controller, successivamente rinominato piattaforma OpenDaylight, consiste in un framework scritto in Java e progettato per offrire agli utenti massima flessibilità per la costruzione di controller che soddisfino le proprie esigenze. Il design modulare consente di comporre un controller selezionando solo i moduli d'interesse con la possibilità di aggiungerne di nuovi, anche a runtime, grazie all'utilizzo di Apache Karaf [23].

L'OpenDaylight controller viene eseguito all'interno di una sua Java Virtual Machine (JVM). Per comunicare con le applicazioni mette a disposizione due tecnologie: se l'applicazione è interna allo stesso spazio d'indirizzamento del controller, viene utilizzato il framework OSGi, altrimenti vengono utilizzate delle API REST. Per interagire con i dispositivi di rete il controller offre supporto, sotto forma di plugin, a diversi protocolli di southbound (OpenFlow, OVSDB, NETCONF, etc.).

Le richieste provenienti da servizi interni ed esterni vengono ricevute da un Service Abstraction Layer (SAL) che, funzionando come un grosso registro contenente informazioni su tutti i servizi offerti dai moduli e sui plugin di southbound interni al controller, è in grado di collegarli alle applicazioni o ad altri servizi che ne hanno bisogno.

È possibile pensare al SAL come formato da due componenti [24]:

- Un datastore composto da:
  - *Config Datastore*: mantiene lo stato desiderato della rete
  - *Operational Datastore*: mantiene la rappresentazione dello stato attuale della rete basato sulle informazioni ricevute dai dispositivi di rete
- Un canale di comunicazione basato su messaggi che fornisce un modo per i vari servizi e plugin di southbound di inviare notifiche e comunicare gli uni con gli altri.

Per aumentare scalabilità e affidabilità del controller è possibile distribuire OpenDaylight all'interno di cluster composti da un minimo di tre macchine [25].

1.5.3 Ryu

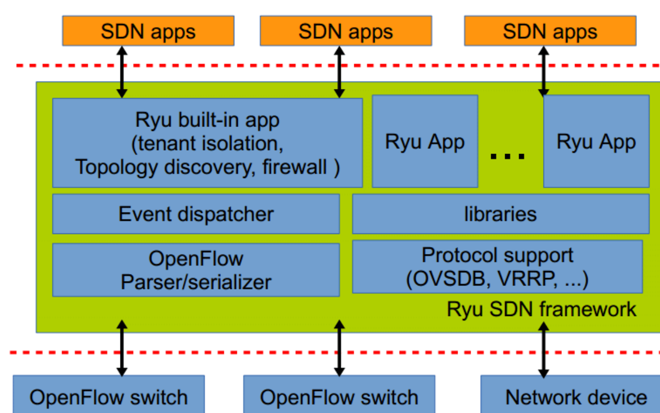

Figura 13 - architettura Ryu [26]

Ryu [27] è un framework SDN opensource scritto in Python basato su componenti che mette a disposizione API ben definite per consentire a sviluppatori di creare

facilmente applicazioni di controllo e di monitoraggio della rete sotto forma di applicazioni Ryu. Per il suo funzionamento utilizza un distributore di eventi che, conoscendo le diverse applicazioni e gli eventi che ciascuna di esse produce o consuma, è in grado di collegare produttori e consumatori. Infine, Ryu fornisce supporto a diversi protocolli di southbound (OpenFlow, NetConf, OF-config, etc.) in modo da poter selezionare il protocollo più adatto in base al caso d'uso.

La sua semplicità e l'utilizzo di un'architettura basata su componenti l'hanno reso il punto di partenza ideale per l'implementazione del prototipo di controller SDN con architettura a micro-servizi. Per questo motivo un'analisi architetturale e funzionale più approfondita verrà effettuata nel paragrafo 4.2.

2. Scomposizione di controller SDN in micro-servizi: letteratura e lavori correlati

Attualmente non sono molti i lavori inerenti alla scomposizione di un controller SDN in micro-servizi: al momento è presente una sola pubblicazione [28] in cui viene descritta l'idea, i vantaggi ed una possibile scomposizione del piano di controllo e la Open Networking Foundation ha iniziato un progetto, chiamato µONOS, con l'intenzione di trasformare l'architettura alla base di ONOS da monolitica a micro-servizi.

2.1 Verso la disgregazione del piano di controllo

I controller SDN attuali sono basati su un'architettura monolitica che integra tutti i servizi e le applicazioni all'interno di un unico programma e che, unitamente alla mancanza di standard per l'interfaccia di northbound, impone agli sviluppatori di dover utilizzare, per la realizzazione di applicazioni di monitoraggio e controllo della rete, solamente le interfacce ed i servizi offerti da una determinata versione di controller SDN, rendendo le applicazioni sviluppate dipendenti dallo specifico controller e dal linguaggio utilizzato per la sua implementazione. Questa limitazione porta ad una riduzione della portabilità delle applicazioni di management che in molti casi, per poter essere utilizzate con altri controller, devono essere completamente riscritte.

Un ragionamento simile vale per lo sviluppo di moduli che consentono di aggiungere funzionalità a controller basati su architettura modulare (come ONOS, OpenDaylight e Ryu). Anche in questo caso essendo dipendenti dal controller e dal linguaggio usato per la sua implementazione i moduli sviluppati, per essere utilizzati all'interno di altri controller, devono essere nella maggior parte dei casi riscritti.

All'interno delle attuali implementazioni di controller SDN, i sottosistemi di cui sono composti sono strettamente accoppiati rendendo il sistema poco affidabile, in quanto un errore di un singolo sottosistema può portare alla caduta dell'intero controller, e non consentendo la replica dei singoli sottosistemi.

Per superare queste limitazioni è necessaria una nuova architettura che consenta di progettare e implementare le nuove generazioni di controller SDN come insieme di micro-servizi cooperanti.

2.2 I vantaggi della scomposizione del piano di controllo in micro-servizi

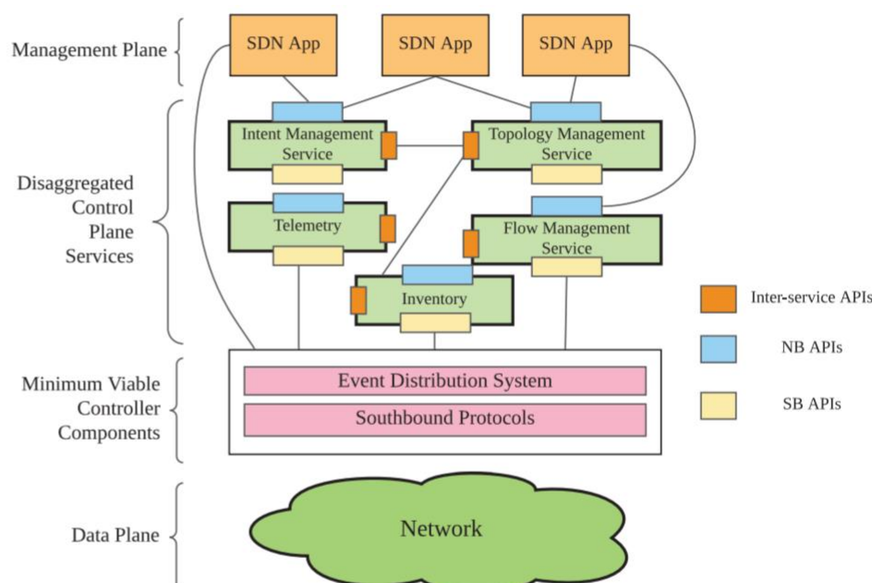

Figura 14 - architettura del piano di controllo scomposto [28]

Un controller SDN monolitico può essere scomposto, come mostrato in figura 14, in un insieme di micro-servizi cooperanti. Il cuore della nuova architettura è costituito da un nucleo che fornisce le funzionalità fondamentali per il funzionamento del controller, mentre tutti gli altri servizi possono essere aggiunti come micro-servizi al suo esterno.

I benefici derivanti dalla trasformazione del piano di controllo da monolitico a micro-servizi consistono in:

- *Scalabilità dei singoli componenti*: è possibile far scalare orizzontalmente singoli micro-servizi indipendentemente da tutti gli altri
- *Aumento di portabilità*: i programmatori possono scegliere arbitrariamente il linguaggio di programmazione, la tecnologia e le librerie da utilizzare per lo sviluppo di applicazioni di management o di micro-servizi da aggiungere al controller
- *Aumento di affidabilità*: la caduta di un singolo micro-servizio non ha effetti sul resto del sistema. Grazie alla nuova architettura, un micro-servizio può essere riparato e ripristinato senza dover ricompilare l'intero controller e senza dover riavviare altri micro-servizi

2.3 Progettazione del piano di controllo scomposto

Il primo passo verso la disgregazione del piano di controllo consiste nell'identificare quali siano i componenti e le funzionalità minime che devono essere inserite all'interno del nucleo: per far questo si può iniziare con un'analisi delle operazioni eseguite da un controller generico nel corso del suo funzionamento.

Quando un controller riceve una richiesta da parte di un'applicazione o da parte di un servizio interno al piano di controllo, la soddisfa comunicando, tramite protocolli di southbound, con i dispositivi di rete ed utilizza le eventuali informazioni ottenute nel corso della comunicazione per rispondere alle richieste iniziali. Quando invece riceve una notifica da parte di un dispositivo di rete riguardo ad un evento avvenuto

all’interno dell’infrastruttura, un controller può inoltrare tale evento ai servizi e alle applicazioni che si sono registrate presso di lui per la sua ricezione.

Partendo da questa breve analisi è possibile dichiarare che un controller con funzionalità minime deve mettere a disposizione due funzionalità essenziali:

- *Un sistema per la distribuzione di eventi*: utilizzato per inviare notifiche riguardanti l’infrastruttura di rete (aggiornamenti di flow entry, nuovi collegamenti di rete, aggiunta di dispositivi, eccezioni, ecc.) ad altri micro-servizi. Il distributore di eventi deve fornire API standard indipendenti da un qualunque linguaggi di programmazione in modo da poter inoltrare eventi a processi esterni indipendentemente da come questi siano stati implementati
- *Supporto a protocolli di southbound*: fondamentali per consentire al nucleo del controller di poter comunicare con i dispositivi di rete sottostanti

All’interno del modello di piano di controllo scomposto, tutti i servizi di cui si compone (ad esempio per la raccolta di informazioni sulla topologia della rete, per la gestione dei flussi, ecc.) sono implementati come micro-servizi che è possibile eseguire all’interno di container, ad esempio basati su tecnologia Docker, in modo da consentirne l’orchestrazione tramite l’uso di tecnologie come Kubernetes. Ogni servizio appartenente al piano di controllo mette a disposizione tre interfacce di comunicazione:

- *Northbound*: per comunicare con le applicazioni
- *Inter-service*: per comunicare con altri micro-servizi appartenenti al piano di controllo tramite standard come gRPC o REST
- *Southbound*: per comunicare con i componenti del nucleo del controller

Nel modello proposto anche le applicazioni appartenenti al management plane sono implementate come micro-servizi e possono, quindi, essere eseguite anch’esse all’interno di container.

2.4 Meccanismi per la distribuzione di eventi

Tramite l'invio di un evento viene segnalata da parte di un dispositivo di rete una modifica relativa allo stato della rete. Le tipologie di evento più comuni in questo ambito sono:

- *Packet Exception*: si innesca ogniqualvolta il nucleo del controller riceve un pacchetto da un dispositivo di rete che non ha trovato nessun riscontro all'interno della propria tabella di inoltro
- *Topology*: ogni volta che la topologia di rete cambia (ad esempio in seguito alla modifica di un collegamento) il nucleo del controller riceve un evento di questo tipo dal dispositivo di rete che si è accorto di tale cambiamento. Questo tipo di evento può essere specializzato in base alla tipologia di modifica riscontrata: collegamento, dispositivo, porta, ecc.
- *Flow Rule*: viene inoltrato tutte le volte che una flow table viene modificata in seguito all'aggiunta, rimozione o aggiornamento di una regola

Gli eventi appena descritti possono essere distribuiti sia ad applicazioni appartenenti al piano di management sia a micro-servizi interni al piano di controllo. I meccanismi per la distribuzione di eventi possono essere di due tipi in base al paradigma di comunicazione utilizzato: publish-subscribe e point-to-point.

2.4.1 Distribuzione di eventi publish-subscribe

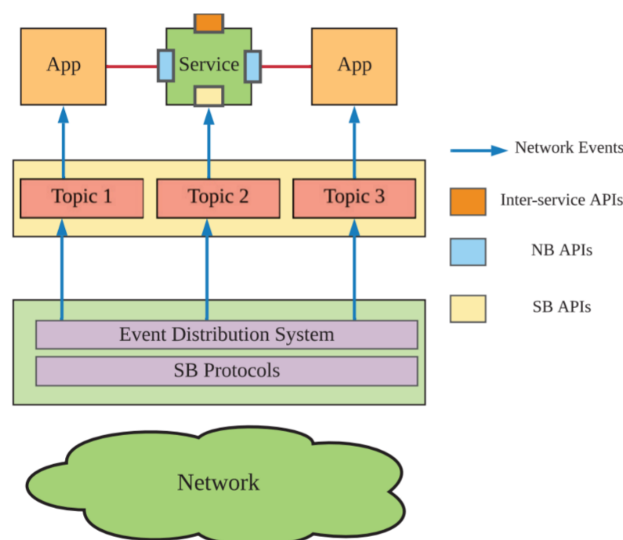

Figura 15 - architettura del sistema di distribuzione di eventi basato sul paradigma publish-subscribe [28]

Il modello publish-subscribe, come mostrato in figura 15, si basa sull'utilizzo di un componente esterno chiamato broker di eventi. Tale componente è in grado memorizzare al suo interno delle topic: raccoglitori di eventi riguardanti un determinato argomento (nel caso del controller è possibile utilizzare topic diverse in base alla tipologia di evento). Un broker di eventi interagisce con due attori principali: produttori e consumatori di eventi. Il produttore, in questo caso il nucleo del controller, è in grado di pubblicare un evento all'interno della topic più adatta. I consumatori, in questo caso applicazioni e servizi, possono invece iscriversi presso un broker in modo da poter essere notificati ogniqualvolta un nuovo evento viene aggiunto ad una topic d'interesse.

Per l'implementazione di un broker di eventi basato sul paradigma publish-subscribe possono essere utilizzati framework come Apache Kafka, RabbitMQ e ActiveMQ.

Questo paradigma di comunicazione consente di avere basso accoppiamento, in quanto un produttore non deve obbligatoriamente conoscere un consumatore e viceversa, e l'utilizzo di un broker per la distribuzione di eventi fornisce vantaggi in termini di scalabilità, resilienza e manutenibilità.

2.4.2 Distribuzione di eventi point-to-point

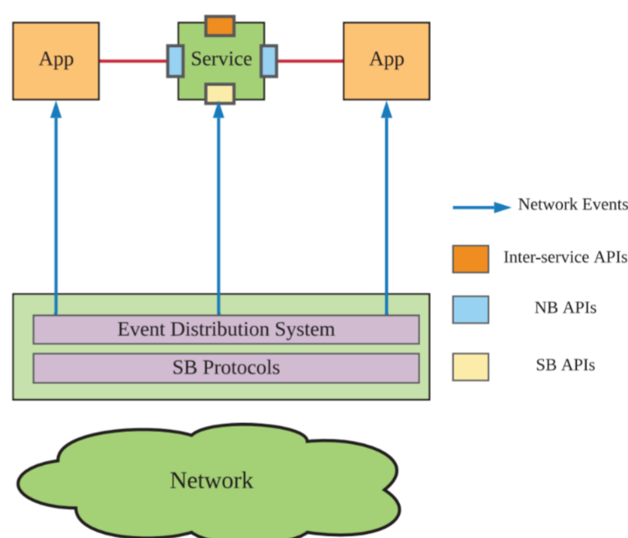

Figura 16 - architettura di un sistema di distribuzione di eventi basato sul paradigma point-to-point [28]

Il modello di comunicazione point-to-point, come mostrato in figura 16, si basa sull'utilizzo di un canale diretto per l'invio di eventi ad ogni singola applicazione o servizio. Il sistema trasmette notifiche di tipo push ad ogni processo che si è iscritto, presso il distributore, per ricevere notifiche relative a determinati tipi di evento.

Le comunicazioni di questo tipo possono avvenire utilizzando framework come gRPC oppure utilizzando API REST.

Rispetto al caso publish-subscribe, il paradigma point-to-point ha come vantaggio un overhead ridotto che può diminuire la latenza relativa alla trasmissione dei messaggi.

2.5 µONOS

µONOS è uno dei progetti annunciati dalla ONF (Open Networking Foundation) per l'implementazione di un nuovo stack open source SDN di prossima generazione (Next Gen SDN – NG-SDN) [29] che consenta un vero controllo della rete, configurazione zero touch (Zero Touch Provisioning – ZTP: automazione di setup e manutenzione dei dispositivi di rete per garantire l'integrità del sistema) e

verificabilità/sicurezza della rete. Questa traslazione verso SDN di nuova generazione fornisce agli operatori un controllo completo della rete.

I componenti appartenenti al nuovo stack proposto e mostrato in figura 17 sono:

- *µONOS* [30]: piattaforma di configurazione e controllo SDN logicamente centralizzata e cloud-native che gestisce una rete composta da switch Stratum-enabled
- *Stratum* [31]: sistema operativo leggero per switch che supporta interfacce Next-Gen SDN (P4Runtime – *Programming Protocol-Independent Packet Processors Runtime*, gNMI – *gRPC Network Management Interface*, gNOI – *gRPC Network Operations Interface*, gRIBI – *gRPC Routing Information Base Interface*) e modelli Open Config
- *NG-SDN Verification Engine*: consente la verifica del corretto funzionamento della rete tramite misurazioni eseguite su pacchetti e flussi
- *NG-SDN Operationalization Framework*: permette di gestire la rete con toolchain CI/CD (Continuous Integration/Continuous Delivery) consentendo aggiornamenti rapidi, automazione per le implementazioni, verifica e rollback

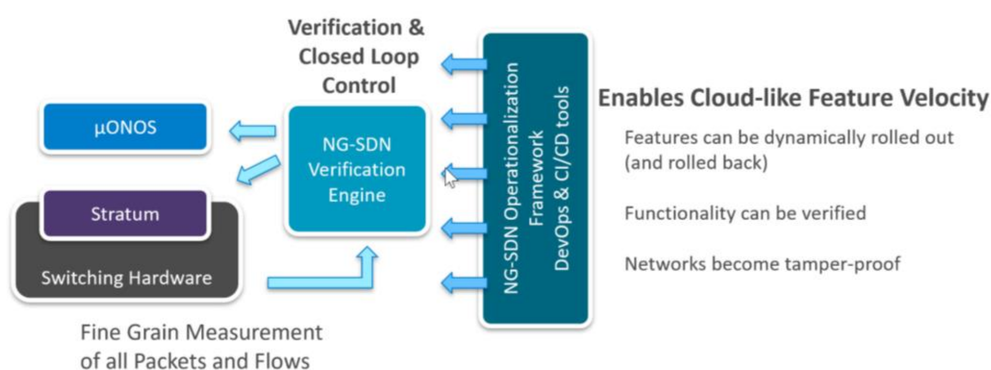

Figura 17 - architettura logica NG-SDN [29]

L'obiettivo di µONOS, ancora in fase di sviluppo, consiste nel fornire una piattaforma che consenta operazioni di configurazione, controllo e monitoraggio della rete. La sua architettura è compatibile con quella di ONOS ma, al posto di essere basata su un singolo e grosso monolite al cui interno sono presenti molte

APIs integrate, è basata su un'architettura a micro-servizi. Come ulteriore differenza rispetto all'implementazione monolitica, la nuova piattaforma, per poter comunicare con Stratum, non fornisce supporto solamente ad interfacce gRPC, ma anche ad interfacce NG-SDN come gNMI, gNOI, P4Runtime e gRIBI.

Figura 18 - µONOS repositories

Il codice di µONOS [32] è suddiviso, come mostrato in figura 18, su più repositories, ognuna creata per contenere un singolo micro-servizio in modo da consentirne l'utilizzo indipendente dagli altri. I principali repository attualmente presenti includono:

- *onos-config*: sottosistema per la configurazione dei dispositivi di rete principalmente tramite interfacce gNMI e gNOI
- *onos-topo*: sottosistema che tiene traccia della topologia di rete
- *onos-control*: sottosistema per il controllo dei dispositivi di rete principalmente tramite interfacce P4Runtime
- *onos-gui*: interfaccia grafica
- *onos-cli*: interfaccia per linea di comando

Per l'implementazione dei componenti il linguaggio preferito e consigliato è Go [33], in quanto fornisce un eccellente integrazione con gRPC (che costituisce anche la base delle altre interfacce NG-SDN gNMI,gNOI e gRIBI) e facilita la gestione e la distribuzione di eventi.

Una possibile configurazione per il deployment di µONOS è mostrata in figura 19.

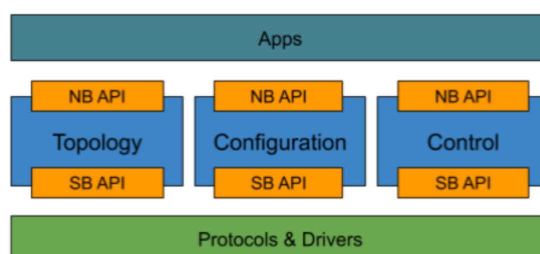

Figura 19 - possibile configurazione di μONOS [35]

μONOS è una piattaforma cloud-native: ciò significa che la sua collezione di micro-servizi può essere impacchettata all'interno di container che è possibile orchestrate utilizzando tecnologie come Kubernetes. Per facilitare questa modalità d'utilizzo, i diversi componenti forniti con μONOS sono disponibili anche come container Docker scaricabili da DockerHub [34].

Un esempio di architettura basata su NG-SDN è mostrato in figura 20. Come si può notare, il sistema è scomponibile su tre livelli principali, tutti orchestrati tramite Kubernetes:

- Il livello più alto rappresenta il management plane e contiene tutte le applicazioni scritte dagli utenti che interagiscono con interfacce gRPC-like o P4Runtime con μONOS
- Il livello centrale rappresenta il piano di controllo ed è popolato dai micro-servizi scelti appartenenti a μONOS
- Il livello più basso rappresenta il piano dei dati composto dai dispositivi di rete al cui interno è installato Stratum

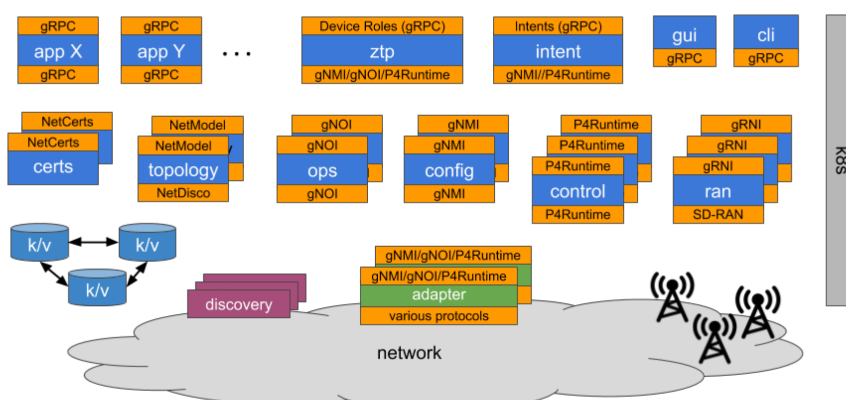

Figura 20 - esempio di deploy NG-SDN [35]

Essendo ancora in fase di sviluppo inizialmente è stato deciso, per semplicità e per minimizzare il numero di componenti utilizzati per la trasmissione di eventi riguardanti cambiamenti dello stato della rete ai vari componenti, di utilizzare comunicazioni di tipo point-to-point tramite interfacce gRPC-like. È però già stata programmata per iterazioni successive la sostituzione di questa tipologia di comunicazione con un modello publish-subscribe [35].

3. Scomposizione di controller SND in micro-servizi: progettazione

Per lo sviluppo di un controller SDN con supporto a micro-servizi è necessario rendere l'infrastruttura di rete osservabile e controllabile dall'esterno: per fare ciò risulta necessario l'utilizzo di un componente principale, il nucleo del controller, che faccia da middleware tra l'infrastruttura di rete sottostante ed i micro-servizi. Questo componente dovrebbe utilizzare per le comunicazioni con gli altri micro-servizi protocolli di northbound che siano i più aperti possibili in modo da garantire massima elasticità per il loro sviluppo e facilità di integrazione. Per questo motivo, uno dei migliori candidati per questa tipologia di comunicazione è REST.

Per le comunicazioni con l'infrastruttura di rete è invece possibile utilizzare un sistema a plugin che permetta di selezionare il protocollo di southbound più adatto in base alla tipologia dei dispositivi di rete appartenenti all'infrastruttura.

L'utilizzo di un'architettura basata su micro-servizi consente di rendere il sistema tollerante ai guasti e di poter scalare i singoli componenti.

L'implementazione dei micro-servizi come VNFs consente inoltre di poterli inserire all'interno di un'sistema di gestione ed orchestrazione per NFV che permette di poter sfruttare appieno i vantaggi forniti da tecnologie di virtualizzazione e cloud.

3.1 Tolleranza ai guasti

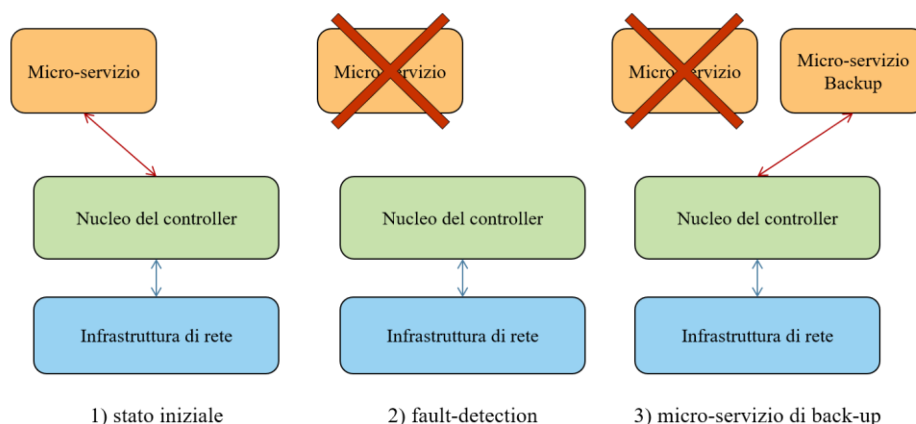

Figura 21 – meccanismo di tolleranza ai guasti

Per rendere il controller tollerante ai guasti dev'essere fornito di un sistema che consenta di individuare e reagire a cadute dei singoli micro-servizi: tale sistema può funzionare in maniera proattiva o reattiva.

Il sistema proattivo dev'essere in grado di monitorare lo stato di funzionamento dei singoli micro-servizi, ad esempio tramite lo scambio di messaggi periodici (o heartbeat). Nel caso in cui un componente non dovesse rispondere entro lo scadere di un time-out, il sistema dovrà essere in grado di istanziare anticipatamente una nuova copia del micro-servizio caduto in modo tale che non si verifichino interruzioni di servizio.

L'unico svantaggio derivante dall'adozione del modello proattivo consiste nell'aumento dell'overhead di comunicazione dovuto allo scambio dei messaggi periodici.

Il sistema reattivo interviene invece istanziando una copia del micro-servizio caduto in seguito alla sua mancata risposta ad una richiesta inviata dal nucleo. L'utilizzo del modello reattivo ha come vantaggio un overhead ridotto rispetto al caso proattivo, al costo però di un aumento di latenza dovuta all'istanziazione del nuovo micro-servizio che può sfociare, nel caso peggiore, in una breve interruzione di servizio.

3.2 Scalabilità dei singoli micro-servizi

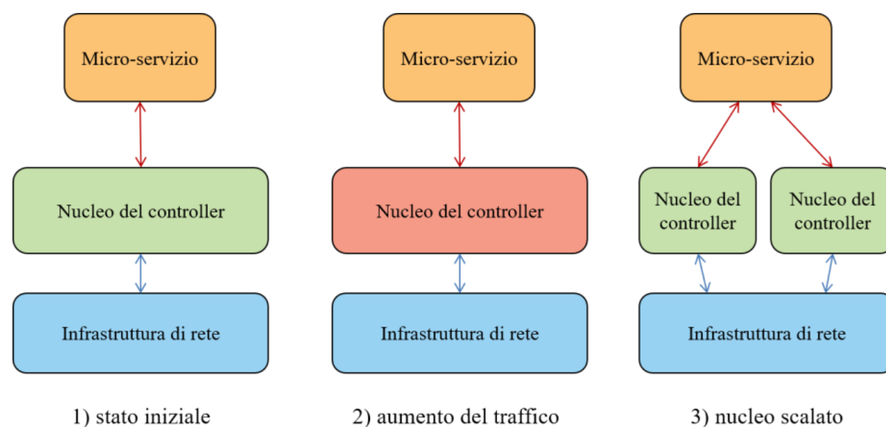

Figura 22 - meccanismo di scalabilità dei singoli micro-servizi

Essendo il controller proposto basato su un'architettura a micro-servizi, sia il nucleo che gli altri componenti possono essere scalati in maniera indipendente. Questa caratteristica consente di non dover replicare l'intero controller, come accade con le attuali implementazioni, ma solo i micro-servizi d'interesse.

Il sistema, per garantire la scalabilità automatizzata dei singoli componenti, dovrebbe essere in grado di monitorare una metrica di interesse e di poter definire delle soglie su tali valori in modo tale che, una volta superate, venga richiesta l'istanziazione di un'ulteriore copia del micro-servizio monitorato.

Ad esempio, se monitorando il nucleo del controller si dovesse riscontrare un aumento del traffico di rete che vada oltre ad una determinata soglia predefinita, il sistema dovrebbe istanziare automaticamente una nuova copia del nucleo del controller, e non di altri componenti. Lo stesso meccanismo può essere sfruttato nel caso in cui un determinato micro-servizio debba eseguire computazioni pesanti e non sia in grado di soddisfare tutte le richieste ricevute da altri componenti: in questo caso verrà richiesta la replica del solo micro-servizio d'interesse senza la necessità di dover replicare il nucleo o altri micro-servizi.

3.3 Micro-servizi come VNFs

Per poter sfruttare al massimo tutti i vantaggi forniti dalle moderne tecnologie di virtualizzazione e di cloud è possibile implementare tutti i componenti del controller come VNFs in modo da poterli inserire all'interno di sistemi di gestione ed orchestrazione per NFV che consentono di garantire l'affidabilità del sistema e la scalabilità dei singoli micro-servizi.

L'attuale standard de facto per questa tipologia di sistemi è Open Source MANO.

3.3.1 Open Source MANO

Open Source MANO (OSM) [36] è un progetto ETSI (European Telecomunications Standards Institute) [37] nato con l'obiettivo di sviluppare uno stack software NFV open source allineato con la specifica NFV ETSI [38].

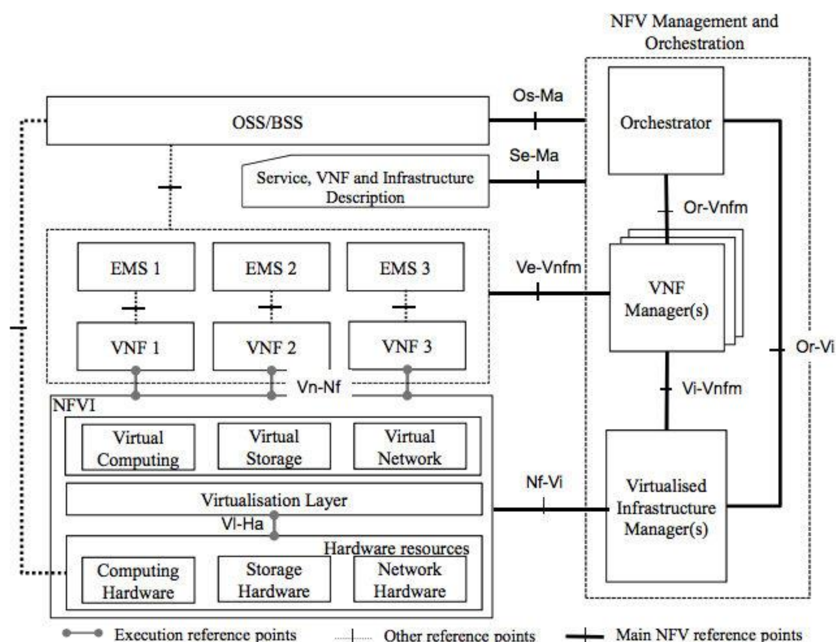

Figura 23 - specifica dell'infrastruttura ETSI NFV [38]

I componenti dell'infrastruttura ETSI NFV sono [39]:

- *VNFM (Virtual Network Function Manager)*: responsabile del ciclo di vita delle istanze di VNFs, si occupa della loro creazione, eliminazione e scalabilità
- *NFVO (NFV Orchestrator)*: responsabile del ciclo di vita dei servizi di rete, applica politiche di utilizzo di risorse e richiede l'istanziazione di VNFs tramite interazione con il VNFM corrispondente
- *NFVI (NFV Infrastructure)*: mette a disposizione le risorse di computazione, memorizzazione e rete che è possibile utilizzare per l'istanziazione di VNFs
- *VIM (Virtualized Infrastructure Manager)*: responsabile del ciclo di vita delle risorse fornite dalla NFVI. Tipicamente è un sistema per la gestione di cloud che espone API in modo da consentire operazioni standard CRUD sulle risorse. OpenStack [40] rappresenta l'implementazione standard de facto

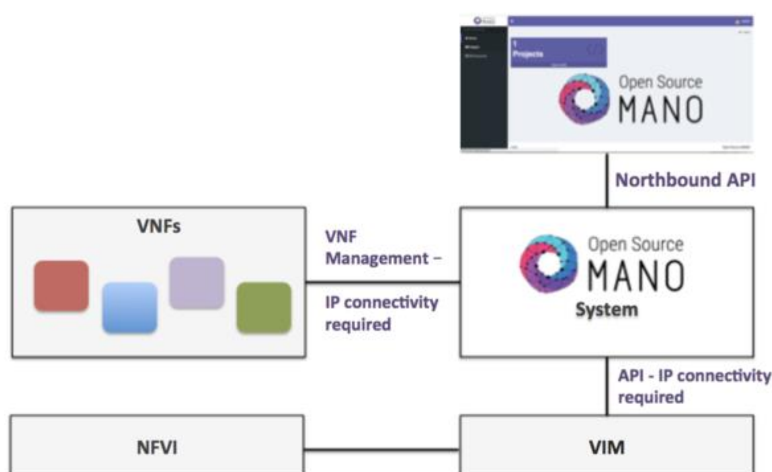

Figura 24 – posizionamento di OSM all'interno dell'infrastruttura ETSI NFV [41]

Come si può notare dalla figura 24, OSM assume i ruoli di VNFM e NFVO all'interno dell'infrastruttura ETSI NFV.

3.3.2 Monitoraggio di VNFs in OSM

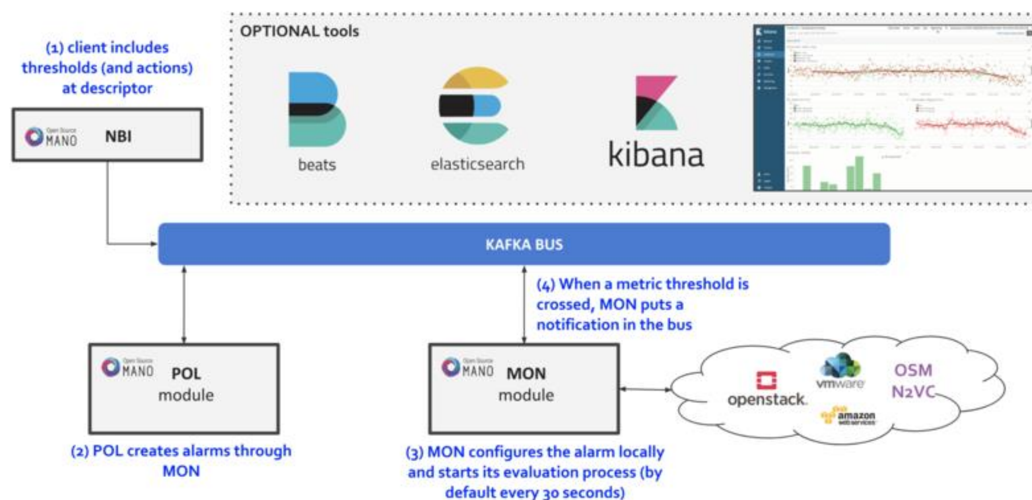

Figura 25 - diagramma di riferimento per la tolleranza ai guasti [42]

OSM mette a disposizione un sistema che consente di monitorare il funzionamento e le performance di VNFs e di poter generare e gestire allarmi nel caso in cui vengano superate le soglie predefinite o si verifichino determinati eventi.

Più in dettaglio, un client può definire all'interno del descrittore di una VNF tramite North Bound Interface (NBI) delle condizioni che, quando verificate, richiedono di eseguire delle azioni ad esse associate. Una volta che queste condizioni sono state definite, il modulo per la gestione di policy (POLicy manager module) imposta degli allarmi per ciascuna di esse all'interno del modulo di monitoraggio (MONitoring module) che, a sua volta, memorizza gli allarmi ed inizia il processo di monitoraggio delle metriche associate. Quando il modulo MON rileva che una delle condizioni definite si è verificata, invia una notifica all'interno del bus Kafka [43]: tale notifica viene consumata dal modulo POL che richiede l'esecuzione delle azioni associate all'allarme.

3.3.3 Tolleranza ai guasti di VNFs in OSM

Sfruttando il sistema di monitoraggio e di allarmi fornito da OSM, è possibile definire delle regole che impongano per ogni micro-servizio il monitoraggio del loro corretto funzionamento, ad esempio, in maniera proattiva, tramite misurazioni

dell'heartbit. Una volta che il sistema si è accorto della caduta di un determinato micro-servizio, richiederà l'esecuzione delle azioni specificate all'interno del suo descrittore. La più semplice di queste potrebbe essere la richiesta d'istanziazione di una nuova copia del micro-servizio caduto.

3.3.4 Scalabilità di VNFs in OSM

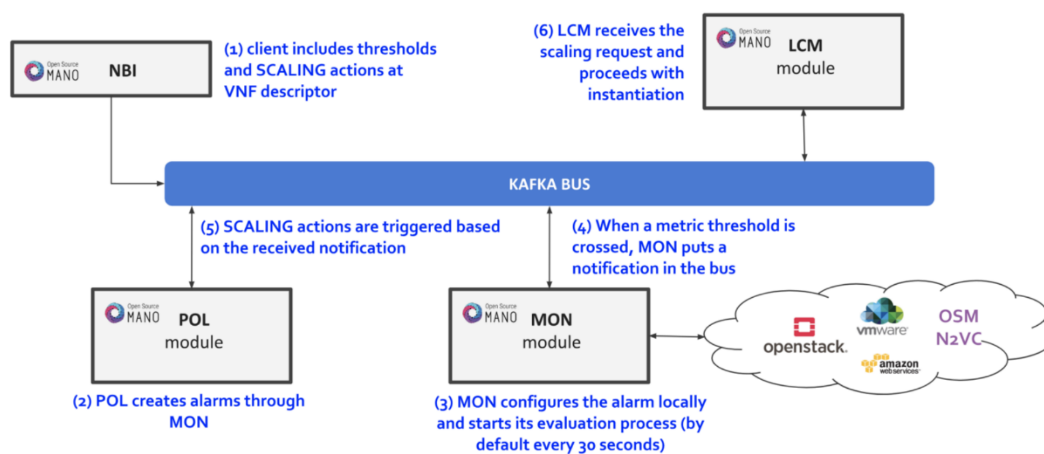

Figura 26 - diagramma della feature OSM Autoscaling [44]

OSM mette a disposizione una nuova feature di Autoscaling [44] che, sfruttando il sistema di monitoraggio e di allarmi, consente di modificare, una volta che sono state superate delle soglie (ad esempio la quantità di traffico) o si sono verificati degli eventi predefiniti, il numero di istanze relative ad un determinato micro-servizio.

La feature sfrutta per il funzionamento un nuovo modulo, LCM (Live Cycle Management), che riceve ed esegue le richiesta di incremento o dectemento del numero di istanze di una determinata VNF.

4. Scomposizione del controller SDN Ryu in micro-servizi: implementazione

All'interno di questo capitolo verrà discussa l'implementazione del prototipo di controller SDN basato su micro-servizi. Per prima cosa verranno mostrate e descritte le tecnologie utilizzate e verrà effettuata un'analisi approfondita dell'architettura e del funzionamento del controller SDN Ryu, introdotto nel paragrafo 1.5.3, in quanto costituirà il punto di partenza del progetto. Verranno quindi descritti il modello e l'implementazione del prototipo realizzato e verranno mostrate anche due versioni di deployment basate rispettivamente su macchine virtuali e su container che permettono di facilitare l'utilizzo dei micro-servizi su cloud e la loro orchestrazione.

4.1 Tecnologie utilizzate

Per verificare il corretto funzionamento e le prestazioni di un qualsiasi controller SDN, è necessario avere a disposizione un'infrastruttura di rete che può essere di due tipi: fisica o virtuale. Poiché per l'implementazione del prototipo e per la verifica del suo funzionamento risulta necessario avere a disposizione un ambiente flessibile e a basso costo, si è optato per la seconda opzione utilizzando Mininet: un emulatore di reti virtuali.

Per poter garantire maggior flessibilità all'intero sistema, si è resa necessaria una tecnologia per la creazione di container che consentisse di rendere i micro-servizi

appartenenti al prototipo orchestrabili e facilmente utilizzabili all'interno di un'infrastruttura cloud: Docker.

Infine, per facilitare la sperimentazione con diverse tecnologie di virtualizzazione si è reso necessario uno strumento che consentisse la creazione di macchine virtuali preconfigurate, indipendenti dalla tecnologia di virtualizzazione scelta: Vagrant.

4.1.1 Mininet

Mininet [45] è un progetto open source che consente la creazione di reti virtuali, anche di grosse dimensioni, all'interno di un qualsiasi laptop o PC in modo da fornire un ambiente di sviluppo e di test per SDN. La sua adozione consente:

- prototipazione rapida
- test di tecnologie complesse senza l'obbligo di dover configurare una rete fisica
- lavoro contemporaneo e indipendente di più sviluppatori su una stessa topologia di rete

Mininet mette a disposizione una Command-Line Interface (CLI), con cui è possibile richiedere l'istanziazione di una nuova rete virtuale ed interagire con essa per ottenere informazioni o per effettuare test, e delle API Python estensibili, utilizzabili per definire nuove topologie custom e per effettuare sperimentazioni.

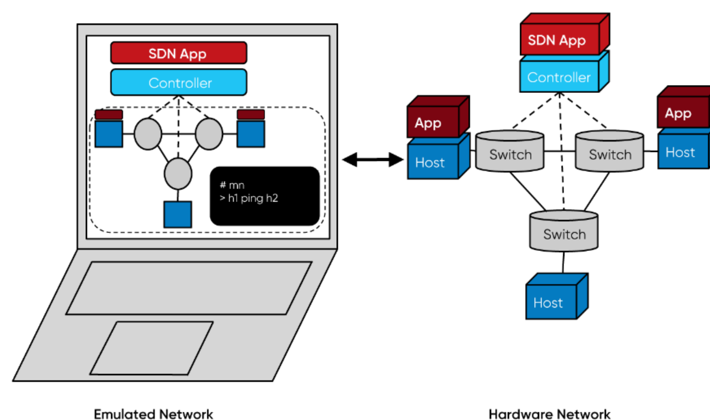

Figura 27 - emulazione di una rete fisica con Mininet [45]

Come visibile in figura 27, una rete Mininet consente il collegamento di un controller SDN dall'esterno e consiste in:

- *Hosts*: funzionano come macchine reali ed hanno il ruolo di end-point
- *Switch*: è possibile specificare la tipologia di switch da utilizzare. Gli switch utilizzati di default forniscono supporto ad OpenFlow
- *Collegamenti*: consentono di emulare i collegamenti tra i diversi componenti della rete

Uno dei maggiori svantaggi derivanti dall'utilizzo di Mininet consiste nel fatto che emulando grosse reti all'interno di un'unica macchina le performance dell'emulatore dipendano direttamente dalle risorse fornite dall'host.

4.1.2 Docker

Docker [46] è un progetto open source nato con l'obiettivo di semplificare e automatizzare il processo di deployment di applicazioni sotto forma di container standardizzati.

I container [47] sono un unità software leggera contenente il codice di un'applicazione e tutte le sue dipendenze (strumenti, librerie di sistema, file di configurazione, ecc.) in modo da poter eseguire un programma software in maniera uniforme e isolata all'interno di un qualunque ambiente d'esecuzione.

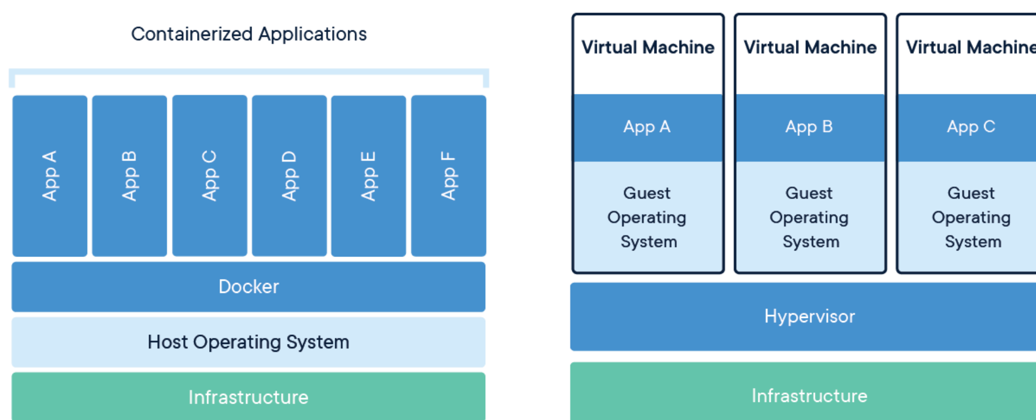

Figura 28 - confronto tra container e macchina virtuale [47]

A differenza di una macchina virtuale, come mostrato in figura 28, un container non si basa sulla virtualizzazione dell'hardware ma sulla virtualizzazione del sistema operativo: non richiedendo la presenza di un sistema operativo completo ed evitando il carico computazionale legato agli hypervisor, i container risultano essere più portabili, leggeri ed efficienti delle macchine virtuali. Grazie ai vantaggi appena descritti, all'interno di una stessa macchina fisica o virtuale possono essere eseguiti più container condividendo lo stesso kernel.

Un container Docker è definito da un'immagine, ovvero un template contenente le istruzioni per la creazione di un container che possono essere condivise anche tramite Docker Hub [48], e dalle opzioni di configurazione definite durante la fase di creazione o d'avvio. La conversione da immagine a container avviene in fase d'esecuzione che, nel caso di container Docker, consiste nel momento in cui le immagini vengono eseguite all'interno di un Docker Engine [49].

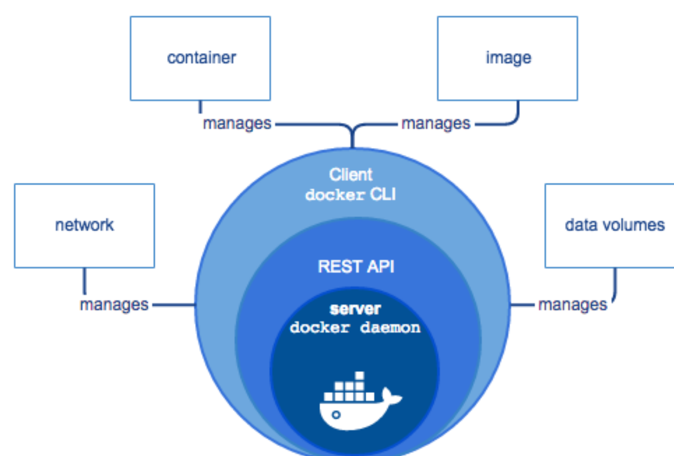

Figura 29 - architettura logica di Docker Engine [49]

Il Docker Engine è un'applicazione open source client-server composta da:

- *Server*: processo demone che ha il compito di costruire, eseguire e distribuire i container Docker
- *API REST*: specificano interfacce di programmazione che è possibile utilizzare per comunicare e interagire con il processo server
- *Command Line Interface (CLI)*: utilizza le API REST per consentire l'interazione con il server tramite linea di comando

4.1.3 Vagrant

Vagrant [50] è uno strumento open source che può essere utilizzato per la costruzione e la distribuzione di ambienti virtuali preconfigurati, riproducibili e portabili chiamati Vagrant box.

Le Vagrant box possono essere definite all'interno di un Vagrantfile, ovvero un file dichiarativo (scritto in Ruby) che consente di configurare e preparare un'ambiente virtuale tramite la descrizione di tutti i requisiti, i pacchetti, le configurazioni del sistema operativo, gli utenti ed altro partendo da una Vagrant box di base che può essere scelta, ad esempio, tra quelle presenti all'interno del catalogo pubblico di Vagrant box [51].

Una volta definita una Vagrant box, questa può essere istanziata, utilizzando il comando *vagrant up*, utilizzando il provider di default VirtualBox [52] o un qualunque altro provider come Hyper-V, Docker e VMware. Al termine della configurazione e della preparazione dell'ambiente virtuale da parte del provider, è possibile interagirvi tramite ssh [53].

4.2 Analisi architetturale e funzionale del controller SDN Ryu

Come anticipato nel paragrafo 1.5.3, Ryu mette a disposizione un controller SDN formato da un insieme estensibile di componenti e delle API Python che consentono lo sviluppo di applicazioni di controllo e gestione della rete, chiamate applicazioni Ryu.

Per descrivere al meglio l'architettura ed il funzionamento di Ryu, l'analisi seguente è stata suddivisa in due parti: nella prima parte viene fornita una descrizione più dettagliata di cosa si intende con applicazione Ryu e di quali siano i suoi componenti fondamentali, mentre nella seconda parte vengono analizzati tutti i componenti che consentono il funzionamento del controller.

4.2.1 Applicazione Ryu

Un'applicazione Ryu è uno script Python al cui interno è definita una classe estensione della classe RyuApp che la rende, tra le altre cose, Observable e potenziale Observer di altre applicazioni Ryu. L'utilizzo del pattern Observer viene sfruttato dal dispatcher di eventi per poter mettere in comunicazione produttori e consumatori registrati al suo interno. Durante la fase di avvio di un'applicazione Ryu, questa deve registrarsi presso il dispatcher esplicitando quali siano gli eventi che può produrre o che è interessata a consumare. Sarà poi compito del dispatcher registrare i consumatori di eventi come Observer dei produttori.

Per indicare quali siano gli eventi d'interesse di una Ryu app, i metodi definiti al suo interno possono essere decorati con annotazioni del tipo *@set_ev_cls(Nome_Evento, Stato_Dispatcher)* indicanti che il metodo decorato dev'essere invocato al verificarsi dell'evento *Nome_Evento*. Se si volesse, per esempio, invocare il metodo *packet_in_handler* definito all'interno di una applicazione Ryu ogni volta che si verifica l'evento OFPPacketIn, che rappresenta la ricezione da parte dal controller di un messaggio OpenFlow di tipo PacketIn, bisognerà scrivere:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
        ...
```

All'interno dell'annotazione bisogna indicare anche lo *Stato_Dispatcher*, che rappresenta la fase in cui si deve trovare lo switch che ha generato l'evento. Le fasi possibili sono:

1. *HANDSHAKE_DISPATCHER*: fase in cui lo switch ed il controller si presentano tramite lo scambio di messaggi di *HELLO*
2. *CONFIG_DISPATCHER*: fase in cui il controller richiede quali siano le *features* offerte da uno switch e lo switch gliele fornisce
3. *MAIN_DISPATCHER*: fase operativa dello switch

4. *DEAD_DISPATCHER*: rappresenta la chiusura di connessione tra lo switch ed il controller

All'interno di un metodo ascoltatore di eventi, ogni volta che un evento viene ricevuto è possibile estrarre da esso le informazioni desiderate. Nel caso di ricezione di un evento OFPPacketIn, per esempio, tipicamente si estraggono in sequenza:

1. La rappresentazione del *packet_in* allegata all'evento: *msg = ev.msg*
2. Il *datapath* (oggetto rappresentante di un dispositivo di rete) che ha generato l'evento: *dp = msg.datapath*
3. Il *parser OpenFlow* necessario per la costruzione di messaggi da inviare al *datapath*: *ofproto_parser = dp.ofproto_parser*

Nel caso in cui si volesse inviare un messaggio ad un dispositivo di rete, dopo aver costruito il messaggio utilizzando un parser adatto, ad esempio quello estratto con ofproto_parser, questo può essere inviato al dispositivo di rete desiderato tramite l'invocazione del metodo *datapath.send_msg(messaggio)*. Il messaggio generato ed inviato può essere di qualunque tipo (es: un *packet_out*, una entry per la flow table, ecc.).

All'interno del repository contenente il codice di Ryu sono anche presenti alcuni esempi di applicazioni Ryu che è possibile utilizzare per aggiungere funzionalità al controller. Per esempio, la Ryu app *simple_switch* consente di effettuare semplici operazioni di routing all'interno della rete.

4.2.2 Funzionamento di Ryu

È possibile istallare Ryu in due modi: tramite comando pip o partendo dal codice sorgente [27]. Indifferentemente dal tipo di installazione, è possibile invocare Ryu da riga di comando tramite invocazione del *ryu-manager*, al quale è possibile passare, in maniera opzionale, una lista di applicazioni Ryu che si vogliono utilizzare. Nel caso non venga fornita nessuna applicazione, verrà caricata di default la sola applicazione *ofp_handler*.

All’interno del *ryu-manager* viene utilizzato un componente, chiamato *AppManager*, che, in sequenza:

1. Carica tutte le applicazioni Ryu passate come argomento durante l’invocazione di *ryu-manager* e tutte le applicazioni Ryu da cui queste dipendono, in quanto emettitrici di eventi d’interesse. Qualunque applicazione Ryu che utilizzi almeno un’annotazione *@set_ev_cls* risulterà dipendente dall’applicazione *ofp_handler* e, quindi, anche quest’ultima verrà caricata.
2. Crea i contesti necessari alle applicazioni Ryu
3. Istanzia tutte le applicazioni Ryu caricate precedentemente all’interno dei corrispettivi contesti e registra tali istanze come *brick*. Prima di avviare le applicazioni Ryu su thread dedicati, l’*AppManager* registra i *brick* consumatori di eventi presso quelli che li producono sfruttando il pattern *Observer* implementato all’interno della classe *RyuApp*, estesa da tutte le applicazioni Ryu (figura 30).

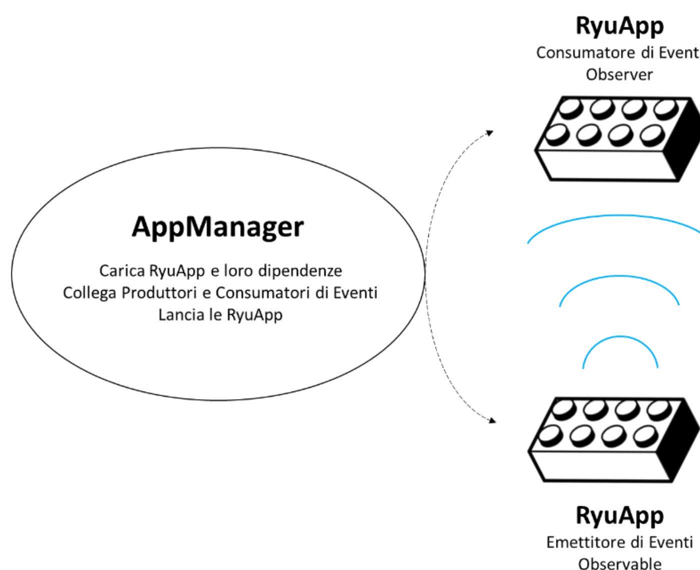

Figura 30 - funzionamento di AppManager

ofp_handler è un’applicazione Ryu fondamentale per il corretto funzionamento del controller in quanto consente di interagire, tramite protocollo OpenFlow, con l’infrastruttura di rete sottostante. Al suo interno viene istanziata la classe

OpenFlowController che ha il compito di occuparsi delle funzionalità di più basso livello del controller.

OpenFlowController esegue su un thread dedicato e, in fase di inizializzazione, crea un canale (SSL o TCP) utilizzato per comunicare con gli switch OpenFlow. Ogni volta che l'*OpenFlowController* si accorge della presenza di un nuovo switch OpenFlow collegato al canale, crea una sua rappresentazione virtuale tramite istanziazione della classe *Datapath*.

In fase di inizializzazione, il *Datapath* crea un canale bidirezionale per comunicare con lo switch OpenFlow corrispondente utilizzando socket e indirizzo passatogli dall'*OpenFlowController*. A questo punto, ogni volta che il *Datapath* riceverà un messaggio da parte dello switch, sfrutterà il meccanismo di notifica di *ofp_handler* per avvisare tutte le applicazioni Ryu interessate a tale evento indicando anche lo *Stato_Dispatcher*. Un'Applicazione Ryu può inviare un messaggio ad uno switch OpenFlow utilizzando il metodo *send_msg* fornito del *Datapath* ad esso associato.

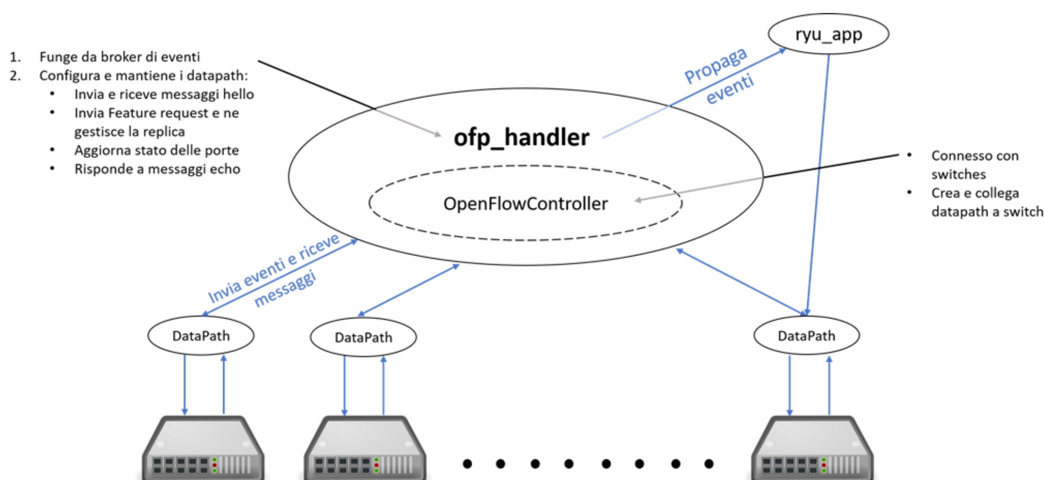

Figura 31 - architettura logica di Ryu

Oltre a contenere l'*OpenFlowController*, l'*ofp_handler* risponde ad alcuni eventi generati dai *Datapath* in modo da nascondere le interazioni di più basso livello tra controller SDN e switch OpenFlow e da mantenere sincronizzati i *Datapath* con gli switch corrispondenti. In particolare, *ofp_handler* reagisce agli eventi:

- *EventOFPHello*: risponde, dopo aver cambiato lo stato del *Datapath* corrispondente da *HANDSHAKE_DISPATCHER* a *CONFIG_DISPATCHER*, con un messaggio *OFPFeatureRequest* richiedendo le feature dello switch
- *EventOFPSwitchFeatures*: memorizza nel *Datapath* l'identificatore e le porte dello switch corrispondente e cambia lo stato del *Datapath* in *MAIN_DISPATCHER*
- *EventOFPEchoRequest*: risponde con un *EchoReply*
- *EventOFPEchoReply*: risponde con un *ack*
- *EventOFPPortStatus*: aggiorna lo stato delle porte del *Datapath* e notifica gli osservatori con un evento di tipo *EventOFPPortStateChange(Datapath,Cambiamento,Numero_porta)*
- *EventOFPErrorMsg*: stampa il messaggio d'errore

4.3 Implementazione del prototipo

Il punto di partenza per la scomposizione del controller in micro-servizi consiste nell'individuazione del nucleo che faccia da intermediario (o middleware) tra l'infrastruttura di rete e tutti gli altri processi, siano essi appartenenti al piano di controllo o al piano delle applicazioni. In questo modo l'intera infrastruttura di rete risulta essere osservabile e controllabile dall'esterno, fornendo nuove opportunità di sviluppo.

4.3.1 Architettura del prototipo

Il prototipo è stato implementato con l'obiettivo di dimostrare la possibilità di utilizzo di un middleware che consenta l'interazione di processi esterni al controller con l'infrastruttura di rete sottostante in modo da poter scomporre il piano di controllo in micro-servizi e conseguentemente aumentare la flessibilità e l'affidabilità del sistema.

Per dimostrare il funzionamento del prototipo quest'ultimo è stato implementato, nella sua versione di base, come composizione di due micro-servizi che interagiscono tra loro sfruttando API REST (figura 32).

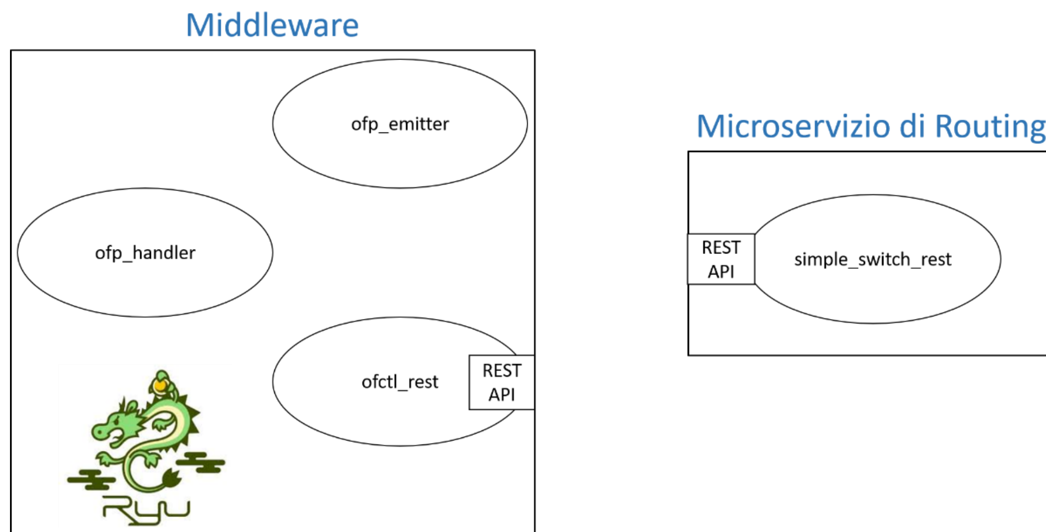

Figura 32 - architettura del prototipo

Il primo di questi micro-servizi rappresenta il nucleo del controller, il middleware del sistema, e consente di osservare e controllare dall'esterno l'infrastruttura di rete sottostante. Questo componente è formato da: il framework Ryu, che carica e collega le diverse applicazioni, l'applicazione *ofp_handler*, che consente l'interazione con l'infrastruttura di rete sottostante sfruttando il protocollo Openflow, l'applicazione *ofp_emitter*, che inoltra verso l'esterno tutti gli eventi generati da *ofp_handler*, e *ofctl_rest*, che permette il controllo dell'infrastruttura di rete dall'esterno interagendo con le API REST che mette a disposizione.

Il secondo micro-servizio, *simple_switch_rest*, è stato implementato come esempio di micro-servizio esterno al nucleo del controller in modo da poterne dimostrare il funzionamento. La sua implementazione consiste in un'estensione dell'applicazione Ryu *simple_switch*, che esegue un semplice algoritmo di routing, a cui è stata aggiunta la capacità di ricevere eventi di tipo *PacketIn* inviati da *ofp_emitter*, di installare flow entry e di inviare pacchetti ai dispositivi di rete utilizzando *ofctl_rest* come intermediario.

4.3.2 Funzionamento del prototipo

Si consideri, a titolo di esempio, una nuova infrastruttura di rete in cui tutti gli switch OpenFlow sono collegati al nucleo del controller. Ognuno degli switch contiene al suo interno una flow table in cui è presente solamente la entry table-miss in cui viene specificato che, per ogni pacchetto ricevuto e non appartenente a nessun flusso di quelli definiti all'interno della flow table, dev'essere inviato un messaggio di tipo PacketIn al nucleo del controller. Per semplificare la spiegazione si consideri che, allo stato attuale, sono già stati scambiati tra gli switch e *ofp_handler* i messaggi necessari per l'inizializzazione della connessione e, quindi, sono già stati creati i *DataPath* rappresentanti di ogni switch.

A questo punto, uno degli switch appartenenti all'infrastruttura riceve un pacchetto da un dispositivo ad esso collegato per il suo inoltro. Per prima cosa lo switch verifica se il pacchetto ricevuto ha corrispondenze all'interno della propria flow table: essendo questa inizialmente vuota, lo switch, seguendo le istruzioni definite nella entry table-miss, invia un messaggio di tipo *PacketIn* al nucleo del controller, sfruttando il protocollo OpenFlow. Il *DataPath* che rappresenta lo switch all'interno del controller riceve il messaggio e, utilizzando il meccanismo per la distribuzione di eventi fornito dal framework Ryu, lo inoltra sotto forma di evento attraverso *ofp_handler* a tutte le applicazioni interessate alla sua ricezione. Poiché l'applicazione *ofp_emitter*, interna al nucleo del controller, è iscritta per la ricezione di eventi di tipo *PacketIn*, riceve l'evento ed in sequenza lo converte in formato JSON e lo invia come contenuto di un messaggio *POST* all'URI */packetin* dell'unico micro-servizio esterno correntemente iscritto per la sua ricezione: *simple_switch_rest* (figura 33). L'evento *PacketIn* in formato JSON contiene:

- *buffer_id*: nel caso in cui uno switch OpenFlow non sia in grado di inoltrare un pacchetto all'interno di un evento, questo viene memorizzato in un buffer intero allo switch e viene indicato nell'evento l'identificore del buffer. Nel caso in cui il pacchetto venga invece inoltrato all'interno dell'evento, questo campo contiene il valore OFP_NO_BUFFER

- *data*: contiene il pacchetto che ha generato l'evento codificato in *base64*
- *in_port*: indica la porta dello switch da cui è stato ricevuto il pacchetto che ha generato l'evento
- *reason*: motivo per cui il pacchetto è stato inviato al controller. Nel caso di *PacketIn* contiene il valore *OFPR_NO_MATCH*
- *total_len*: dimensione del pacchetto inoltrato
- *dpid*: identificatore del datapath rappresentante dello switch che ha generato l'evento

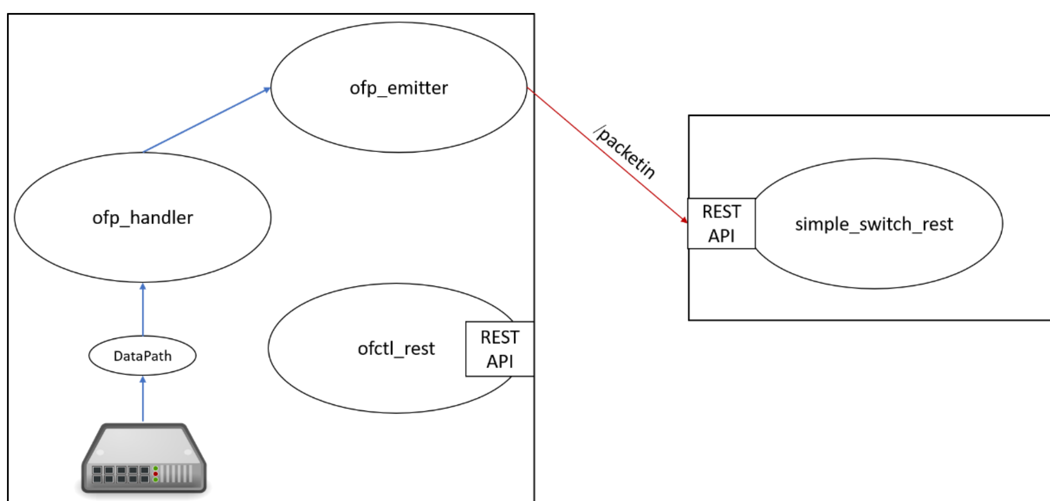

Figura 33 - propagazione evento PacketIn

simple_switch_rest riceve l'evento, estrae e memorizza le informazioni d'interesse e verifica se è a conoscenza di quale porta debba essere utilizzata dallo switch per l'inoltro del pacchetto. In caso positivo viene costruita in JSON la descrizione di una flow entry inviata come contenuto di un messaggio *POST* all'URI */stats/flowentry/add* di *ofctl_rest* in cui è possibile specificare:

- *dpid*: datapath id. Identificatore intero univoco del datapath che rappresenta lo switch in cui si vuole installare la nuova regola
- *match*: campi che definiscono le regole di corrispondenza per riconoscere i pacchetti appartenenti al flusso. Devono essere espressi sotto forma di dizionario (es. *{"in_port":1}* indica che tutti i pacchetti ricevuti dalla porta 1 dello switch appartengono al flusso)

- *cookie*: intero utilizzato dal controller in maniera opaca
- *idle_timeout*: tempo in secondi trascorso senza riscontri prima della rimozione della entry
- *hard_timeout*: tempo in secondi trascorso prima della rimozione della entry
- *priority*: priorità assegnata alla regola
- *flags*: intero per specificare flag OpenFlow
- *actions*: lista di dizionari contenente le istruzioni da eseguire sui pacchetti appartenenti al flusso (es. *[{"type":"OUTPUT","port":2}]* indica che il pacchetto dev'essere inoltrato verso la porta 2 dello switch)
- *buffer_id*: se il pacchetto non è stato incapsulato all'interno del messaggio *PacketIn* ma memorizzato all'interno di un buffer, questo campo specifica l'identificatore del buffer in cui si trova il pacchetto su cui bisogna eseguire le azioni definite in actions. In caso contrario contiene *OFP_NO_BUFFER*

ofctl_rest, dopo aver ricevuto il messaggio, utilizza le istruzioni in esso contenute per costruire la flow entry e per installarla all'interno dello switch indicato interagendo con il *DataPath* corrispondente (figura 34). In seguito all'istallazione della nuova regola, *simple_switch_rest* invia un secondo messaggio all'URI */sendpacket* di *ofctl_rest* contenente la descrizione in JSON per la ricostruzione di un pacchetto simile a quello inizialmente ricevuto dallo switch, ma al cui interno viene specificata la porta d'uscita da utilizzare per il suo inoltro. Questa descrizione contiene:

- *dpid*: identificatore del datapath a cui inoltrare il pacchetto
- *in_port*: intero che identifica la porta dello switch da cui è stato ricevuto il pacchetto
- *actions*: lista di dizionari contenente le istruzioni da eseguire sul pacchetto corrente
- *data*: pacchetto codificato in *base64*

ofctl_rest riceve la richiesta, crea il pacchetto seguendo le istruzioni fornite e lo inoltra, interagendo con il *Datapath*, allo switch indicato che provvederà ad inoltrarlo verso la porta indicata (figura 35). Questo secondo passaggio risulta

essere necessario in quanto senza di esso il pacchetto contenuto nell'evento *PacketIn* andrebbe perso. Nel caso in cui invece *simple_switch_rest* non fosse a conoscenza di quale porta lo switch debba utilizzare per l'inoltro del pacchetto, costruisce in JSON la descrizione di un pacchetto simile a quello ricevuto dallo switch, ma al cui interno viene specificata come porta d'uscita *OFPP_FLOOD* (ciò può essere fatto inserendo all'interno del campo *actions* il valore *[{"type":"OUTPUT","port":OFPP_FLOOD}]*), che rappresenta la porta speciale *ALL* presente all'interno degli switch OpenFlow e consente l'inoltro del pacchetto su tutte le porte esclusa quella indicata come porta d'ingresso. La descrizione del pacchetto viene inviata con un messaggio *POST* all'URI */sendpacket* di *ofctl_rest* che, similmente al caso precedente, costruisce il pacchetto e lo inoltra, interagendo con il *DataPath*, allo switch indicato che provvederà ad inoltrarlo tramite la sua porta *ALL* (figura 35).

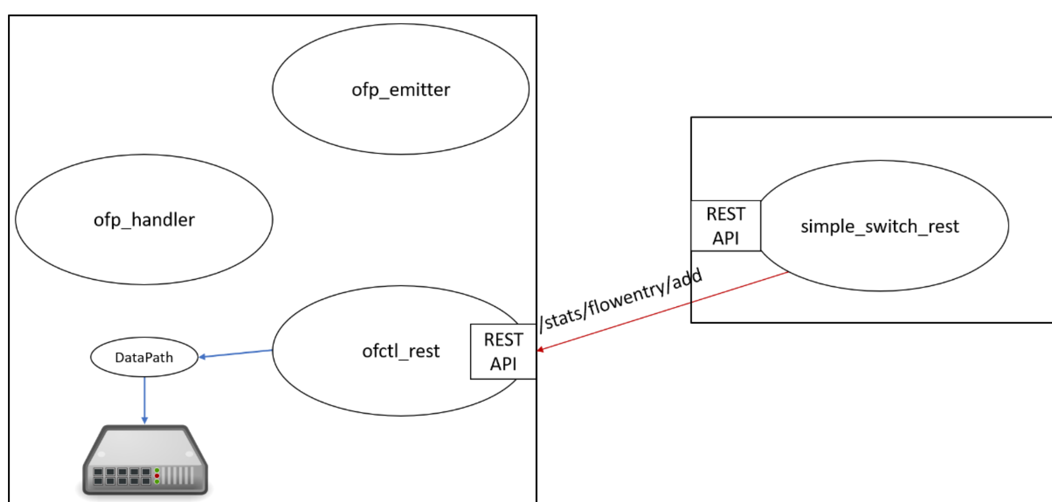

Figura 34 - installazione di una flow entry

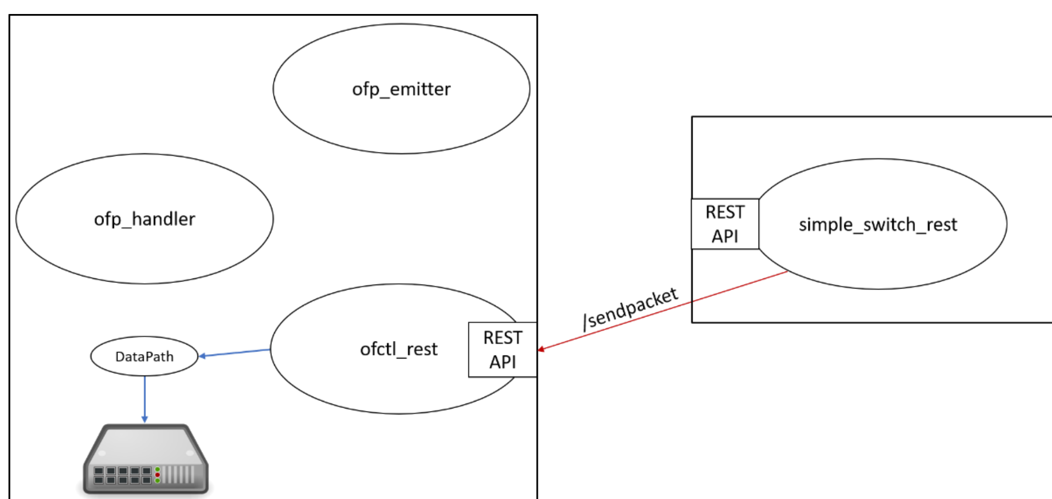

Figura 35 - inoltro di un pacchetto

4.4 Trasferimento dei micro-servizi all'interno di container Docker

Per aumentare l'affidabilità e la scalabilità del sistema, risulta necessario l'incapsulamento dei vari micro-servizi all'interno di container in modo tale da consentirne, tramite l'utilizzo di un orchestratore, l'istanziazione e la replicazione su richiesta in seguito a variazioni di carico o caduta di micro-servizi. Con questo obiettivo sono stati prodotti due Dockerfile, uno per il nucleo ed uno per il micro-servizio di routing, in modo da definire i due container di base.

4.4.1 Container per il middleware

Il Dockerfile che descrive il container del nucleo del controller utilizza come container di base *python:3* [54] a cui vengono aggiunte le due applicazioni Ryu *ofp_emitter* e *ofctl_rest*. All'interno del container viene istallato Ryu, tramite il comando *pip*, che mette a disposizione il framework per il caricamento delle applicazioni e l'applicazione di base *ofp_handler*. Dopo aver caricato ed istallato tutto il necessario, viene eseguito il comando *ryu-manager*, disponibile in seguito all'istallazione di Ryu, a cui vengono passate come argomento le due applicazioni *ofp_emitter* e *ofctl_rest*.

Per consentire la comunicazione con Mininet e con il micro-servizio di routing dev'essere richiesta, in fase di avvio, rispettivamente la mappatura delle porte d'ingresso del container 6633 (porta di default utilizzata da Mininet per il collegamento con un controller SDN esterno) e 8080 (porta utilizzata dal server REST del nucleo del controller per consentire la comunicazione tramite API) sulle porte della macchina host.

4.4.2 Container per il micro-servizio di routing

Il Dockerfile in cui viene descritto il container del micro-servizio di routing utilizza anch'esso come container di base *python:3* a cui viene aggiunto lo script python *simple_switch_rest* e alcune librerie estratte da Ryu che consentono l'analisi dei pacchetti dati incapsulati negli eventi ricevuti. All'interno del container viene istallata la libreria python Flask, utilizzata per l'implementazione del server REST necessario per la ricezione degli eventi emessi dal nucleo del controller. Per consentire la ricezione di tali eventi, la porta 8090 (porta utilizzata dal server REST) dev'essere mappata in fase di avvio su una porta dell'host. Al termine della preparazione del container, viene semplicemente eseguito lo script *simple_switch_rest* sfruttando il supporto a python fornito dal container di base.

4.5 Trasferimento dei micro-servizi all'interno di Vagrant box

Per facilitare la sperimentazione con diverse tecnologie di virtualizzazione risulta molto utile l'incapsulamento dei vari micro-servizi all'interno di Vagrant box. Il loro utilizzo consente infatti di poter definire Vagrantfile indipendenti dal provider di virtualizzazione e facilmente configurabili. Per questo motivo sono stati definiti, così come nel caso Docker, due Vagrantfile: uno per il nucleo ed uno per il micro-servizio di routing.

4.5.1 Vagrant box per il middleware

Il Vagrantfile con cui viene definita la Vagrant box contenente il nucleo del controller, utilizza come Vagrant box di base *ubuntu/bionic64* [55]. Il posizionamento del Vagrantfile all'interno della stessa directory in cui sono state definite le due applicazioni Ryu *ofp_emitter* e *ofctl_rest*, consente di poter accedere a queste due applicazioni anche all'interno della Vagrant box (più precisamente è possibile ritrovare gli stessi file contenuti nella directory contenente il Vagrantfile anche all'interno della directory */vagrant* della Vagrant box). Similmente al caso Docker, nel Vagrantfile viene richiesta l'installazione del framework Ryu tramite l'utilizzo del comando *pip* di cui a sua volta, non essendo presente all'interno della box di base, dev'essere richiesta l'installazione.

Per consentire la comunicazione con Mininet e con il micro-servizio di routing devono essere richiesta, all'interno del Vagrantfile, la mappatura delle porte d'ingresso della Vagrant box 6633 e 8080 sulle porte della macchina host.

Una volta istanziata e configurata la Vagrant box utilizzando il provider scelto, è possibile accedervi tramite *ssh* (per comodità è possibile sfruttare per questo scopo il comando *vagrant ssh*). All'interno della Vagrant box è quindi possibile eseguire il comando *ryu-manager* a cui vengono passate come argomento le due applicazioni *ofp_emitter* e *ofctl_rest* contenute all'interno della cartella */vagrant*.

4.5.2 Vagrant box per il micro-servizio di routing

Il Vagrantfile in cui viene descritta la Vagrant box contenente il micro-servizio di routing utilizza anch'essa come Vagrant box di base *ubuntu/bionic64*. Il Vagrantfile è stato posizionato in questo caso all'interno della stessa directory contenente lo script python *simple_switch_rest* e le librerie da esso utilizzate. Come per la Vagrant box contenente il nucleo, viene richiesta anche in questo caso all'interno del Vagrantfile l'installazione di *pip*, strumento necessario per l'installazione della libreria python Flask utilizzata per l'implementazione del server REST all'interno *simple_switch_rest*. Per il corretto funzionamento del server, dev'essere richiesta

all’interno del Vagrantfile anche la mappatura della porta 8090 su una porta dell’host.

Una volta che la Vagrant box è stata istanziata e configurata è possibile richiedere, interagendo con *ssh*, l’esecuzione dello script *simple_switch_rest* in modo da attivare il micro-servizio di routing.

5. Esperimenti e risultati ottenuti

Per poter valutare le performance del prototipo in termini di affidabilità, scalabilità e latenza, e poter confrontare i risultati ottenuti con quelli relativi all'approccio tradizionale, sono stati eseguiti diversi esperimenti utilizzando quattro modelli di controller:

- *Controller SDN standard*: utilizza la versione standard del framework Ryu assieme all'applicazione di routing *simple_switch*

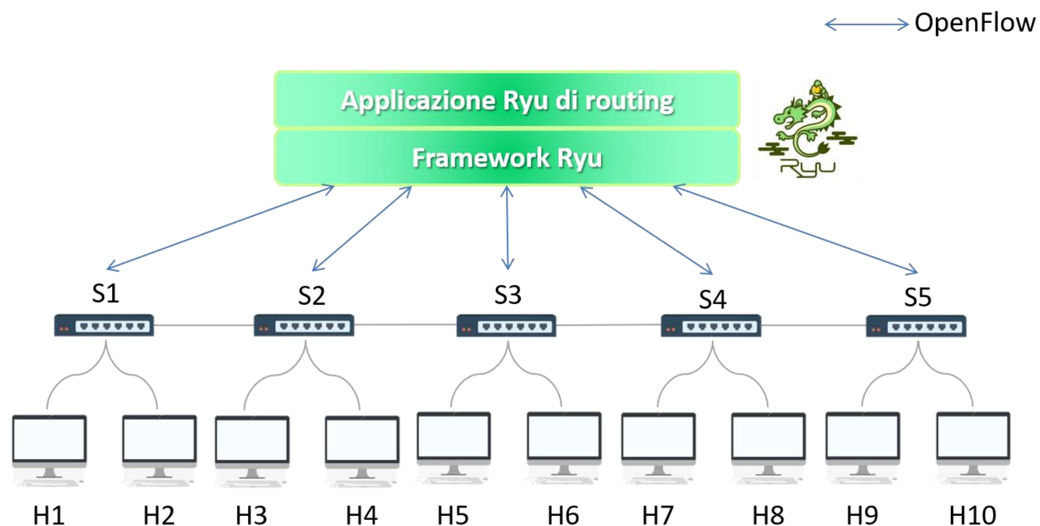

Figura 36 - controller SDN standard

- *Controller SDN basato su micro-servizi*: utilizza l'implementazione del prototipo composta dal nucleo e dal micro-servizio di routing *simple_switch_rest*

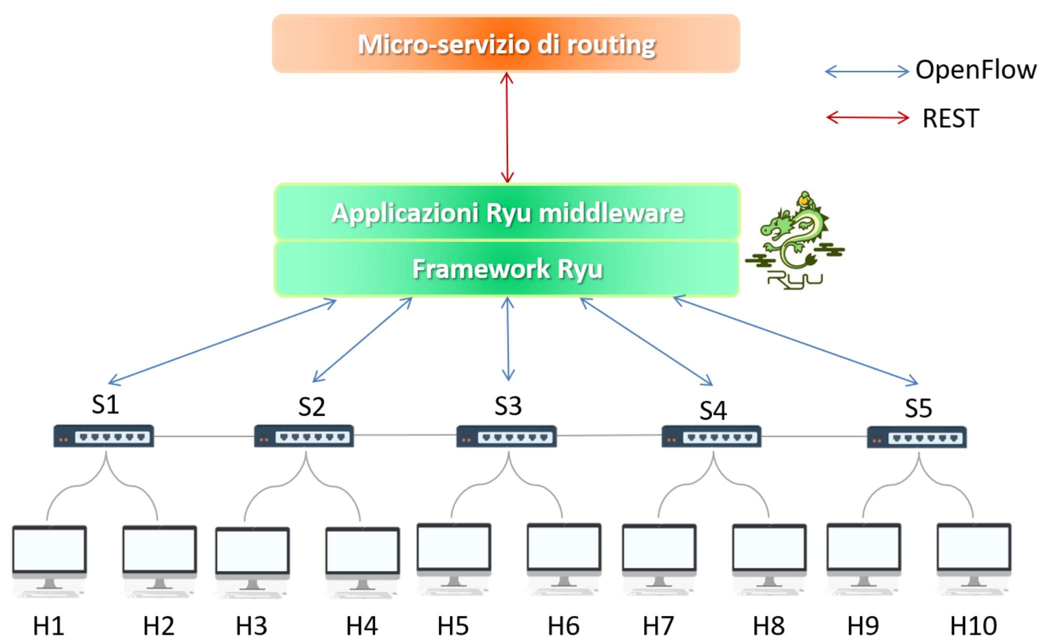

Figura 37 - controller SDN basato su micro-servizi

- *Controller SDN basato su container*: utilizza gli stessi elementi del controller SDN basato su micro-servizi racchiusi all'interno di container Docker

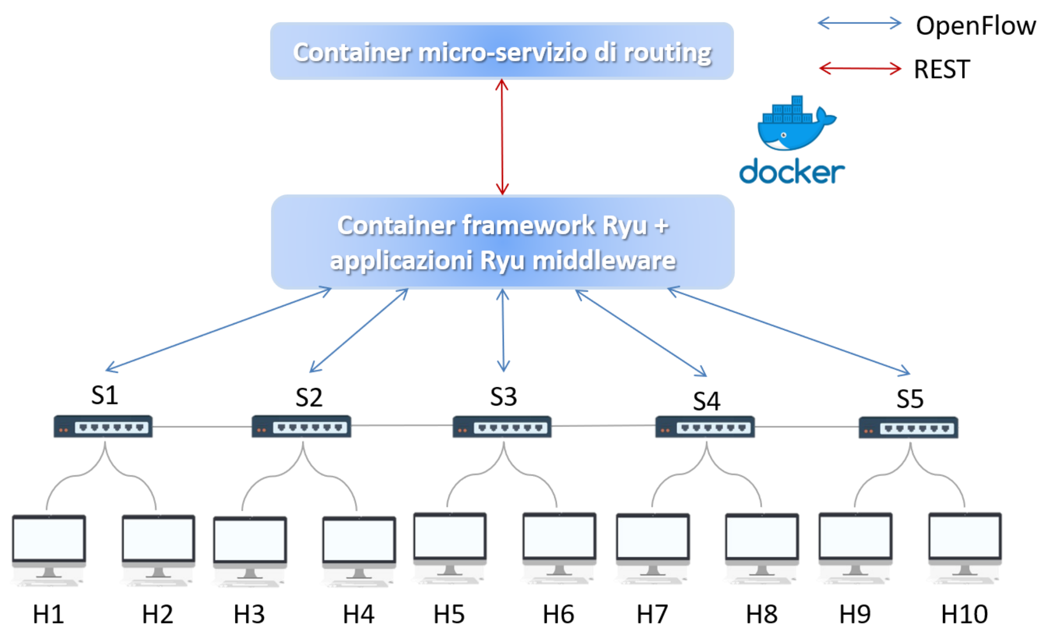

Figura 38 - controller SDN basato su container

- *Controller SDN basato su macchine virtuali*: utilizza gli stessi elementi del controller SDN basato su micro-servizi racchiusi all'interno di Vagrant box istanziate utilizzando come provider VirtualBox

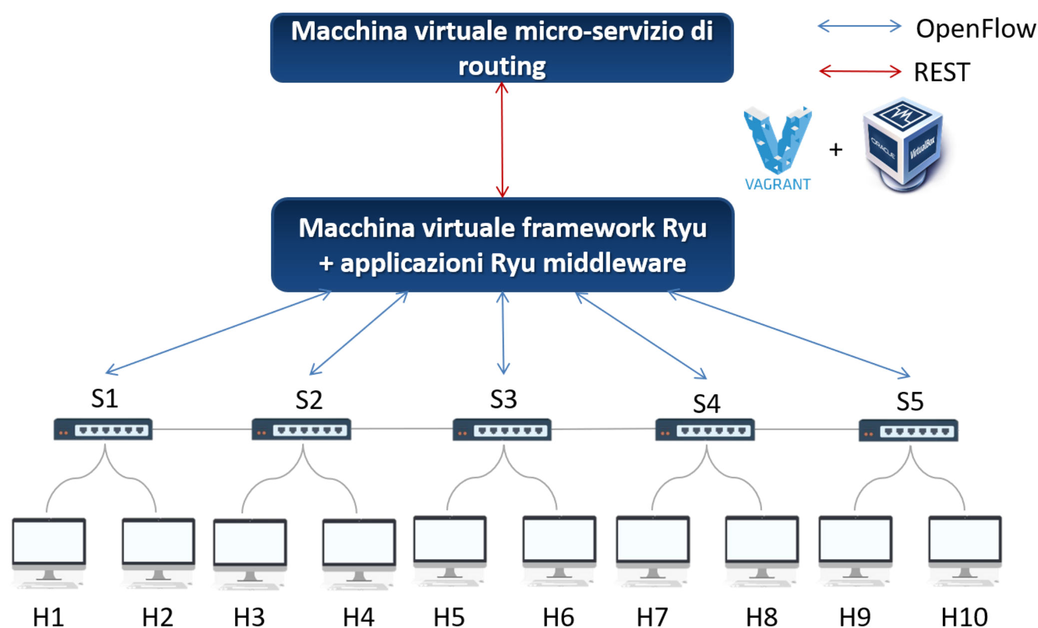

Figura 39 - controller SDN basato su macchine virtuali

5.1 Ambiente di testing

I diversi esperimenti sono stati eseguiti utilizzando una macchina virtuale Linux (MX 19.2) con 6 GB di RAM e 3 vCPU al cui interno sono stati installati Mininet (2.3.0d6), Docker Community edition (19.03.11), Ryu SDN Framework (4.34) e Python 3 (3.7.3).

La topologia della rete virtuale utilizzata per gli esperimenti è stata creata sfruttando le API Python messe a disposizione da Mininet ed è composta da 5 switch (S) e 10 hosts (H) collegati come in figura 40.

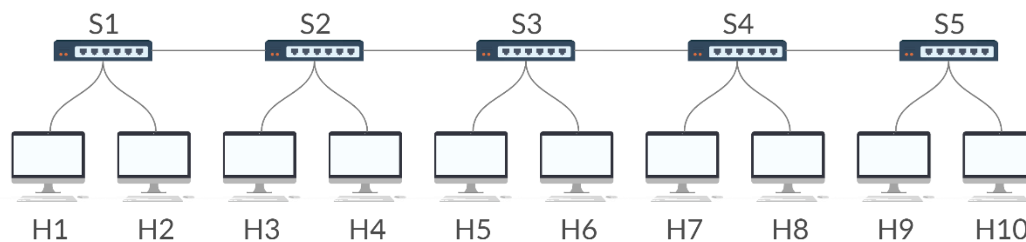

Figura 40 - topologia rete Mininet

Per la raccolta dei dati è stato sfruttato lo strumento di amministrazione *ping* che, tramite l'invio periodico di pacchetti ICMP *Echo Request* e ricezione delle relative risposte sotto forma di *Echo Reply*, ha consentito di verificare la raggiungibilità di un host da parte di un altro ed il tempo impiegato da un pacchetto per raggiungere un dispositivo di destinazione e tornare indietro all'origine.

La presentazione degli esperimenti effettuati è anticipata da una descrizione passo-passo di cosa accade all'interno della rete durante l'utilizzo dello strumento *ping*, utile per facilitare la comprensione dei risultati ottenuti in seguito all'esecuzione degli esperimenti.

5.2 Comportamento di ping inviati all'interno dell'ambiente di testing

A titolo di esempio si consideri una rete virtuale Mininet appena creata con topologia descritta in precedenza e in cui l'host H1 vuole utilizzare lo strumento di amministrazione *ping* per verificare la raggiungibilità dell'host H10 e misurare il ritardo della comunicazione. La descrizione seguente del comportamento di *ping* è valida per ciascuna delle tre tipologie di controller definite in precedenza: l'unica differenza consiste nella metodologia di comunicazione tra il middleware del sistema ed il servizio di routing.

Il primo messaggio inviato da H1 consiste in una *ARP REQUEST*: H1 è a conoscenza del solo indirizzo IP di H10 e non del suo indirizzo MAC, che è necessario specificare all'interno dell'intestazione dei pacchetti *ECHO REQUEST*.

La richiesta ARP viene ricevuta dal primo switch (S1) che, non possedendo nessuna regola per l'inoltro del pacchetto ricevuto, la invia al nucleo del controller SDN sotto forma di *PacketIn* sfruttando OpenFlow. Una volta che il nucleo ha ricevuto il messaggio, trasmette un evento di tipo *PacketIn* al servizio di routing utilizzando la tecnologia specifica dell'implementazione (per il controller SDN standard viene utilizzato il distributore di eventi messo a disposizione da Ryu mentre negli altri tre casi viene utilizzato il protocollo REST). Il servizio di routing memorizza, sfruttando le informazioni presenti nell'intestazione del pacchetto, l'associazione tra porta d'ingresso da cui lo switch S1 ha ricevuto la *ARP REQUEST* e l'indirizzo MAC di H1, restituisce quindi il pacchetto allo switch S1 tramite il nucleo del controller richiedendo il suo inoltro su tutte le porte d'uscita. L'unica porta d'uscita che non è stata utilizzata come porta d'ingresso dal pacchetto in esame è quella che collega S1 ad S2.

Lo stesso processo si ripete, consentendo al controller di memorizzare un percorso inverso con destinazione H1, fino a quando il pacchetto non viene consegnato allo switch S5 che, dopo averlo inviato al controller e dopo aver ricevuto richiesta di inoltrarlo verso tutte le sue porte d'uscita, lo inoltra verso H10 che riconosce l'indirizzo IP ed invia una risposta sotto forma di *ARP RESPONSE* contenente il suo indirizzo MAC.

A questo punto S5 riceve l'*ARP REPLY* da H10 e richiede al controller su quale porta debba essere inoltrato. Il nucleo del controller lo inoltra al servizio di routing che, essendo a conoscenza del percorso verso H1 precedentemente memorizzato, installa una nuova regola all'interno di S5 in cui viene specificato che tutti i pacchetti con destinatario H1 devono essere inoltrati verso la porta che connette S5 ad S4. Lo stesso processo viene ripetuto, consentendo al controller di memorizzare il percorso inverso con destinazione H10, fino alla ricezione del pacchetto *ARP REPLY* da parte di H1. A questo punto H1, essendo a conoscenza dell'indirizzo MAC di H10, può iniziare la vera e propria operazione di ping inviando, ogni secondo, una nuova *ECHO REQUEST*.

Il primo di questi pacchetti genera ancora una volta degli eventi *PacketIn* in quanto all'interno degli switch sono state installate solo regole relative al percorso da H10 ad H1 e non del percorso inverso. La generazione di tali eventi porta all'istallazione di regole per la creazione del percorso da H1 ad H10. Da questo momento fino allo scadere delle regole scritte all'interno degli switch, tutti i pacchetti aventi come destinatario H1 o H10 inviati da uno qualsiasi degli hosts appartenenti alla rete possono essere consegnati dall'infrastruttura di rete senza ulteriori interazioni con il controller.

5.3 Affidabilità

Il test sull'affidabilità ha come obiettivo la dimostrazione che, anche in caso di fallimento di un servizio, il controller sia in grado di funzionare sfruttando, al posto del servizio fallito, una sua copia di backup. La differenza tra l'approccio basato su micro-servizi e quello basato su architettura monolitica consiste nel fatto che in questo secondo caso, come visto nei precedenti capitoli, la replica e la distribuzione di singoli servizi risulta essere impossibile ed il fallimento di un solo componente può portare alla caduta dell'intero controller.

5.3.1 Ambiente di Test

L'affidabilità aumentata del prototipo è stata dimostrata utilizzando il controller SDN basato su container, in quanto caso di maggiore interesse. Il sistema utilizzato per l'esecuzione dell'esperimento è formato da:

- rete virtuale Mininet: composta da 5 switch e 10 hosts
- un container contente il middleware del controller
- un container contentente il micro-servizio di routing
- un secondo container di backup del micro-servizio di routing

5.3.2 Esperimento

Per l'esecuzione dell'esperimento l'host H1 utilizza il comando *ping* specificando come destinatario l'host H10; le flow entry vengono installate all'interno degli switch con un time-out pari a 20 secondi ed il container del middleware è inizialmente configurato per comunicare con il container di routing principale. Dopo 40 secondi dall'avvio dell'esperimento il container di routing principale, dopo aver installato per la seconda volta, interagendo con il container del nucleo, delle regole di routing relative al collegamento tra H1 e H10, è stato interrotto in modo da simulare una caduta del servizio.

Una volta scadute le regole di routing installate, il middleware del controller inizia a ricevere nuovamente dei messaggi *PacketIn* che prova ad inoltrare, sotto forma di eventi, al container di routing principale.

Non ottenendo nessuna risposta nell'arco di 0,05 secondi, il middleware si accorge della caduta del servizio ed inizia quindi ad inoltrare gli eventi verso il container di routing di backup, inizialmente predisposto ed avviato.

5.3.3 Risultati ottenuti

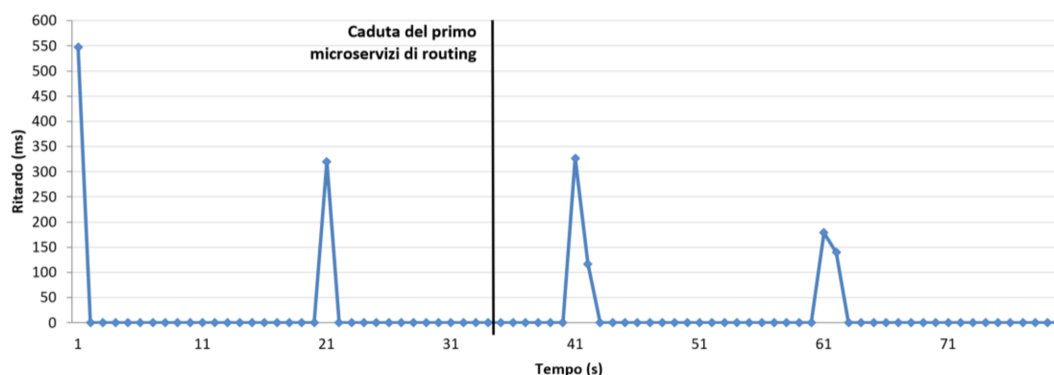

Figura 41 - risultati test affidabilità

Nella figura 41 sono riportati i risultati relativi al test di affidabilità. All'interno dell'immagine è possibile individuare quattro picchi principali che rappresentano quattro fasi distinte in cui l'infrastruttura di rete interagisce con il controller:

1) *Primo pacchetto*: presenta il valore di latenza maggiore in quanto comprende al suo interno lo scambio dei messaggi ARP, la prima installazione delle regole di routing e l'invio e la ricezione del primo *ping*
2) *Aggiornamento delle regole*: il *ping* inviato successivamente allo scadere del time-out di 20 secondi assegnato alle regole installate durante la fase 1 richiede, per giungere a destinazione, un aggiornamento da parte del controller delle regole di routing appena scadute. Tale aggiornamento avviene in questa fase
3) *Passaggio al micro-servizio di routing di backup*: allo scadere del time-out relativo alle regole installate durante la fase 2, il nucleo del controller si accorge, cercando di inoltrare le richieste di aggiornamento delle regole, della caduta del micro-servizio di routing principale ed inizia quindi ad inoltrare gli eventi verso il micro-servizio di routing di backup. Durante l'elaborazione dell'evento *PacketIn* relativo alla prima *ECHO REQUEST*, il nuovo micro-servizio di routing non è in grado di installare nessuna regola all'interno dell'infrastruttura di rete in quanto non è a conoscenza di nessun accoppiamento tra porte d'uscita dello switch che ha generato l'evento e l'indirizzo MAC di destinazione dell'*ECHO REQUEST* e richiede quindi a tutti gli switch di inoltrare la richiesta su tutte le porte d'uscita. Una volta che l'*ECHO REQUEST* è giunta a destinazione, il controller deve gestire l'inoltro della prima *ECHO REPLY*. In questa fase il micro-servizio di routing può utilizzare le informazioni raccolte durante l'inoltro della *ECHO REQUEST* per aggiornare le regole di inoltro relative al percorso H10-H1. Poiché fino a questo punto non sono state ancora aggiornate le regole relative al percorso H1-H10, affinché la *ECHO REQUEST* successiva giunga a destinazione, è necessaria un'ulteriore interazione con il controller: questo il motivo dell'aumento della latenza relativa al quarantaduesimo *ping* inviato.
4) *Aggiornamento delle regole da parte del micro-servizio di backup*: poiché le installazioni delle regole di inoltro relative al percorso H1-H10 e H10-H1 sono state effettuate nel corso della fase precedente in seguito alla gestione

di due *ping* successivi, i time-out a loro associati risultano quindi sfalsati di 1 secondo. Per questo motivo, durante l'inoltro del sessantunesimo *ping*, risultano scadute solamente le regole relative al percorso inverso (H10-H1) e non quelle del percorso diretto (H1-H10) interessando esclusivamente la *ECHO REPLY* e non la *ECHO REQUEST*: questo è il motivo del ritardo così basso rispetto ai casi precedenti. La latenza generata dall'aggiornamento delle regole relative al percorso H1-H10 si mostra invece nella gestione del *ping* successivo.

Dai risultati dell'esperimento si può notare che, a differenza di ciò che accadrebbe utilizzando il controller SDN standard, l'interruzione del container contenente il micro-servizio di routing principale non comporta un'interruzione di servizio ma solamente un leggero aumento nella latenza complessiva dei primi due *ping*.

5.4 Latenza

L'esternalizzazione del micro-servizio di routing effettuata dal prototipo comporta un aumento della latenza nella ricezione dei pacchetti dati che richiedono, prima del loro inoltro, interazioni tra switch e controller. L'obiettivo di questo esperimento consiste nel misurare quantitativamente l'entità di tale aumento in modo da poter effettuare un confronto sulle performance ottenute nell'utilizzo delle quattro diverse tipologie di controller.

5.4.1 Ambiente di Test

I test sono stati ripetuti 30 volte per ciascuna delle quattro tipologie di controller definite inizialmente (Controller SDN standard, basato su micro-servizi, basato su container e basato su macchine virtuali) ed i risultati ottenuti ne rappresentano la media. La topologia di rete Mininet utilizzata è sempre quella composta da 5 switch e 10 hosts.

5.4.2 Esperimento

L'esperimento è composto da più parti e consiste nell'utilizzare lo strumento d'amministrazione *ping* per misurare il tempo trascorso dall'invio di un pacchetto *ECHO REQUEST* alla ricezione del relativo pacchetto *ECHO REPLY*. Le fasi dell'esperimento sono 3:

1) *Primo pacchetto*: misurazione della latenza relativa al primo *ping* inviato. All'interno della misura ottenuta è contenuto anche il ritardo relativo allo scambio di messaggi ARP per il riconoscimento dell'indirizzo MAC della controparte
2) *Flusso normale*: misurazione della latenza dei *ping* inviati successivamente all'istallazione, da parte del controller, di regole di routing all'interno dell'infrastruttura di rete. In questa fase, non essendo richieste interazioni con il controller, ci si aspetta una latenza costante per tutte e quattro le tipologie di controller considerate
3) *Pacchetto per l'aggiornamento delle regole*: misurazione della latenza di un *ping* inviato successivamente allo scadere del time-out associato alle regole di routing installate. In questo caso la misura ottenuta non contiene al suo interno lo scambio dei messaggi ARP

Per poter effettuare un'analisi più accurata, l'esperimento è stato ripetuto, in tutte le sue fasi, in modo da poter misurare la latenza nella comunicazione tra gli host H1-H3, in cui gli switch interessati sono solo due (S1 ed S2), e la latenza nella comunicazione tra gli host H1-H10, in cui gli switch interessati sono cinque, ossia tutti quelli appartenenti all'infrastruttura di rete.

5.4.3 Risultati ottenuti

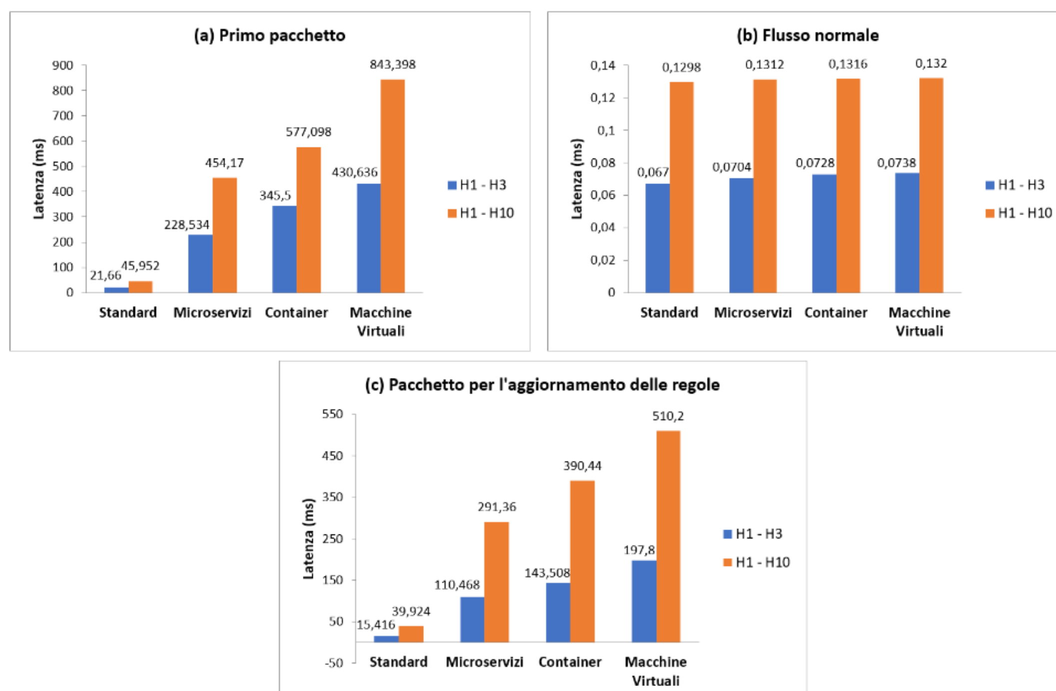

Figura 42 - risultati test latenza

Dai risultati ottenuti per il primo pacchetto (figura 42a), si può notare che il controller SDN basato su micro-servizi presenta un ritardo, sia nel caso di comunicazione H1-H3 che H1-H10, di circa 10 volte superiore rispetto al controller SDN standard. Questo aumento in termini di latenza è dovuto all'utilizzo del protocollo di comunicazione REST per le interazioni tra il middleware ed il micro-servizio di routing. L'incapsulamento di questi due componenti all'interno di container Docker comporta un ulteriore incremento della latenza pari a circa 100ms mentre il loro incapsulamento all'interno di macchine virtuali comporta un ulteriore incremento della latenza pari a circa 100ms rispetto al caso Docker nella comunicazione H1-H3 che diventa di circa 250ms nel caso di comunicazione H1-H10. Un andamento simile, ma meno marcato, è riscontrabile anche dai risultati relativi ai pacchetti inviati in seguito alla scadenza delle regole di routing precedentemente istallate (figura 42c).

Dai risultati ottenuti da questi due esperimenti si può affermare che, in caso di installazione di tipo reattivo di regole di routing da parte del controller (e

generalizzando, nel caso in cui prima dell'inoltro di un pacchetto questo debba essere inviato a servizi esterni al nucleo del controller per essere processato), l'approccio monolitico, grazie alla sua gestione locale degli eventi, consente di avere una latenza minore rispetto a quello basato su micro-servizi. Confrontando i risultati ottenuti per le diverse tipologie di controller, il risultato migliore è stato infatti ottenuto dal controller SDN standard seguito, in ordine, dalla versione basata su micro-servizi, da quella basata su container e da quella basata su macchine virtuali.

I risultati ottenuti per il flusso normale (figura 42b) dimostrano, come atteso, che i pacchetti inviati successivamente all'installazione delle regole di routing o, più in generale, che non richiedano interazioni con il controller prima del loro inoltro, si ha una latenza pressoché identica per tutte e quattro le tipologie di controller.

In conclusione, utilizzando il prototipo si può riscontrare un aumento in termini di latenza esclusivamente nei casi in cui sia richiesta un'interazione con il controller prima dell'inoltro di pacchetti, mentre in tutti gli altri casi la latenza misurata risulta essere la stessa del caso standard.

6. Conclusioni e sviluppi futuri

All’interno di questo elaborato sono state inizialmente descritte le motivazioni che stanno spingendo grandi aziende ad investire nella ricerca e nello sviluppo di nuove tecnologie che consentano di superare le limitazioni imposte dall’utilizzo dell’infrastruttura di rete corrente; è stato quindi presentato SDN come uno dei più promettenti paradigmi per conseguire tale obiettivo e sono state discusse le implementazioni di controller SDN più popolari sottolineando come siano tutte basate su un’architettura monolitica che non permette di sfruttare appieno il potenziale fornito da moderne tecnologie di cloud e di virtualizzazione.

L’idea di trasformare i controller SDN da singoli monoliti ad insiemi di micro-servizi cooperanti tra loro è supportata, come discusso all’interno dell’elaborato, anche dalla letteratura e da lavori correlati. Dopo aver fornito una panoramica sul paradigma SDN ed aver discusso dell’utilità del lavoro svolto, è stato presentato il prototipo di controller SDN basato su micro-servizi formato da un nucleo (middleware) che ha il compito di fare da intermediario tra l’infrastruttura di rete ed i vari micro-servizi e che, sfruttando il concetto di container ed il supporto fornito da infrastrutture cloud, consente di ottenere, come dimostrato nel corso degli esperimenti svolti, maggior affidabilità e scalabilità rispetto al caso monolitico al costo di un aumento in termini di latenza da pagare in determinate situazioni. In aggiunta alle caratteristiche appena descritte il prototipo, grazie alla sua scomposizione in micro-servizi, consente di adattare l’architettura del controller, anche dinamicamente, in base allo scenario d’uso e a variazioni riscontrate all’interno dell’infrastruttura.

In conclusione, in base ai risultati ottenuti, il prototipo di controller SDN con supporto a micro-servizi è da considerarsi come punto di partenza per future sperimentazioni. Con l'obiettivo di ridurre il costo, in termini di latenza, riscontrato nel corso degli esperimenti potranno essere sperimentate tecnologie alternative a REST: in particolare, gli sforzi di ricerca dovranno concentrarsi sull'utilizzo del framework gRPC che, a fronte di una minore apertura, consentirà di ottenere performance migliori.

Sarà inoltre possibile affrontare in modo più esauriente l'inserimento del controller SDN all'interno di un contesto NFV tramite sperimentazioni con OSM andando ad esplorare in maniera più approfondita gli strumenti che mette a disposizione per la scalabilità e la tolleranza ai guasti automatizzati.

Un ulteriore spunto di ricerca è rappresentato dalla sperimentazione con comunicazioni di tipo publish-subscribe che permettono di sfruttare i vantaggi offerti da distributori di eventi affermati come Apache Kafka.

Infine, per consentire la sperimentazione con sistemi più complessi, sarà possibile sviluppare ulteriori micro-servizi che consentano di aggiungere funzionalità al nucleo del controller.

Ringraziamenti

Siamo arrivati anche alla fine di questo percorso. Dico “siamo” perché molto probabilmente da solo non ce l’avrei mai fatta. Dedico quindi questa parte della mia tesi a ringraziare tutte le persone che mi hanno sostenuto ed aiutato lungo la strada.

Un ringraziamento speciale va al prof. Luca Foschini ed al prof. Paolo Bellavista per avermi indirizzato e guidato in questo interessante lavoro di tesi e per avermi trasmesso nel corso delle loro lezioni importanti insegnamenti e dato utili consigli. Ringrazio inoltre l’ingegner Domenico Scotece che è stato fondamentale per la buona riuscita di questo lavoro e che si è sempre reso disponibile per chiarimenti e consigli.

Ringrazio la mia famiglia, che mi ha sempre sostenuto. In particolar modo ringrazio mia madre Katia e mio padre Sandro perché senza di loro tutto questo non sarebbe stato possibile. Ringrazio anche mia sorella Deborah che mi ha sempre fornito supporto morale. Un ringraziamento speciale va alla mia fidanzata Paola, che mi ha supportato e sopportato, come sempre.

Ringrazio i miei compagni di corso, con cui ho instaurato amicizie indimenticabili. In particolare, ringrazio Mauro R., Matteo C. e Marco M. con cui ho condiviso la maggior parte di questo percorso. Ringrazio inoltre il team Q4ttro: Mauro R., Matteo C., Marco M., Simone B., Luca P., Federico D.G. e Federico G.

Infine, ringrazio tutto il personale dell’Università di Bologna. In particolare, ringrazio tutti i professori che ho incontrato lungo il mio percorso e che mi hanno contaminato con la loro passione e la loro conoscenza.

Grazie a Tutti!

Daniel

Bibliografia e sitografia

[1] H. Kim and N. Feamster, "Improving network management with software defined networking," in *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114-119, February 2013, doi: 10.1109/MCOM.2013.6461195.

[2] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," in *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14-76, Jan. 2015, doi: 10.1109/JPROC.2014.2371999.

[3] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. “OpenFlow: enabling innovation in campus networks”. In: ACM SIGCOMM Computer Communication Review 38.2 (2008), pp. 69–74.

[4] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: an intellectual history of programmable networks. SIGCOMM Comput. Commun. Rev. 44, 2 (April 2014), 87–98.

[5] Open Networking Foundation: https://www.opennetworking.org/ (Data di accesso: 18/08/2020)

[6] Open Networking Foundation. “Software-Defined Networking: The New Norm for Networks”. In: ONF White Paper, April 2012.

[7] Open Daylight: https://www.opendaylight.org/ (Data di accesso: 18/08/2020)

[8] The Linux Foundation. “Industry Leaders Collaborate on OpenDaylight Project, Donate Key Technologies to Accelerate Software-Defined Networking”. April 2013. https://www.linuxfoundation.org/press-release/2013/04/industry-leaders-collaborate-on-opendaylight-project-donate-key-technologies-to-accelerate-software-defined-networking/ (Data di accesso: 18/08/2020)

[9] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca and J. Folgueira, "Network Slicing for 5G with SDN/NFV: Concepts, Architectures, and Challenges," in *IEEE Communications Magazine*, vol. 55, no. 5, pp. 80-87, May 2017, doi: 10.1109/MCOM.2017.1600935.

[10] J. H. Cox et al., "Advancing Software-Defined Networks: A Survey," in IEEE Access, vol. 5, pp. 25487-25526, 2017, doi: 10.1109/ACCESS.2017.2762291.

[11] Y. Jarraya, T. Madi and M. Debbabi, "A Survey and a Layered Taxonomy of Software-Defined Networking," in *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1955-1980, Fourthquarter 2014, doi: 10.1109/COMST.2014.2320094.

[12] M. Nkosi, A. Lysko, L. Ravhuanzwo, T. Nandeni and A. Engelberencht, "Classification of SDN distributed controller approaches: A brief overview," *2016 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, Durban, 2016, pp. 342-344, doi: 10.1109/ICACCE.2016.8073772.

[13] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, ''A roadmap for traffic engineering in SDN-OpenFlow networks,'' Comput. Netw., vol. 71, pp. 1–30, Oct. 2014.

[14] McKeown, Nick & Anderson, Tom & Balakrishnan, Hari & Parulkar, Guru & Peterson, Larry & Rexford, Jennifer & Shenker, Scott & Turner, Jonathan. (2008). OpenFlow: Enabling innovation in campus networks. Computer Communication Review. 38. 69-74. doi: 10.1145/1355734.1355746.

[15] Open Networking Foundation. “OpenFlow Switch Specification Version 1.5.1”. March 2015. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf (Data di accesso: 20/08/2020)

[16] Letteri, Ivan. (2018). Performance of Botnet Detection by Neural Networks in Software-Defined Networks (https://www.ivanletteri.it/2020/01/31/itasec2020/). 10.1007/978-3-030-01689-0_4.

[17] Open Networking Foundation. “OpenFlow Switch Specification Version 1.3.0”. June 2012. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf (Data di accesso: 20/08/2020)

[18] ONOS: https://www.opennetworking.org/onos/ (Data di accesso 21/08/2020)

[19] ONOS – System Components: https://wiki.onosproject.org/display/ONOS/System+Components (Data di accesso: 28/08/2020)

[20] Overview of ONOS architecture: https://wiki.onosproject.org/display/ONOS/Overview+of+ONOS+architecture (Data di accesso: 28/08/2020)

[21] ONOS Wiki: https://wiki.onosproject.org/display/ONOS/ONOS (Data di accesso: 28/08/2020)

[22] OpenDaylight Sodium: https://www.opendaylight.org/what-we-do/current-release/sodium (Data di accesso: 31/09/2020)

[23] Apache Karaf: https://karaf.apache.org/ (Data di accesso: 31/08/2020)

[24] OpenDaylight concepts and tools: https://docs.opendaylight.org/en/stable-sodium/getting-started-guide/concepts_and_tools.html (Data di accesso: 31/08/2020)

[25] OpenDaylight Clustering: https://docs.opendaylight.org/en/stable-sodium/getting-started-guide/clustering.html (Data di accesso: 31/08/2020)

[26] Yamahata, Isaku.: Ryu: SDN framework and Python experience: Pycon APAC 2013, Japan, September 2014. https://www.slideshare.net/yamahata/ryu-sdnframeworkupload (Data di accesso: 01/09/2020)

[27] Ryu: https://ryu-sdn.org/ (Data di accesso: 01/09/2020)

[28] D. Comer and A. Rastegarnia, "Toward Disaggregating the SDN Control Plane," in IEEE Communications Magazine, vol. 57, no. 10, pp. 70-75, October 2019, doi: 10.1109/MCOM.001.1900063.

[29] NG-SDN: https://www.opennetworking.org/ng-sdn/ (Data di accesso: 03/09/2020)

[30] μONOS: https://docs.onosproject.org/ (Data di acceso: 03/09/2020)

[31] Stratum: https://www.opennetworking.org/stratum/ (Data di accesso: 03/09/2020)

[32] μONOS Repository: https://github.com/onosproject/ (Data di accesso 03/09/2020)

[33] Go: https://golang.org/ (Data di accesso 03/09/2020)

[34] μONOS – Docker Hub: https://hub.docker.com/u/onosproject/ (Data di accesso 03/09/2020)

[35] μONOS - Next Generation ONOS: https://docs.google.com/document/d/1IZz_8EG1AII3JYmTYla585Gbpe9dfSwChO8lEkehp4A/mobilebasic (Data di accesso 03/09/2020)

[36] ETSI – Open Source Mano: https://osm.etsi.org/ (Data di accesso 25/09/2020)

[37] ETSI: https://www.etsi.org/ (Data di accesso 25/09/2020)

[38] ETSI, "Network Functions Virtualisation (NFV); Architectural Framework"; in ETSI GS NFV 002 V1.2.1 (2014-12): https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf (Data di accesso 25/09/2020)

[39] P. Bellavista, L. Foschini, "05 – 5G and Mobile Edge Computing", (2019-2020), slides del corso Mobile Systems M – Università di Bologna: http://lia.disi.unibo.it/Courses/sm1920-info/lucidi/05-5G_MobileEdgeComputing(1x).pdf (Data di accesso: 25/09/2020)

[40] OpenStack: https://www.openstack.org/ (Data di accesso 25/09/2020)

[41] OSM – Quickstart: https://osm.etsi.org/docs/user-guide/01-quickstart.html (Data di accesso: 25/09/2020)

[42] OSM – Fault Management: https://osm.etsi.org/wikipub/index.php/OSM_Fault_Management (Data di accesso: 25/09/2020)

[43] Apache Kafka: https://kafka.apache.org/ (Data di accesso: 25/09/2020)

[44] OSM Autoscaling: https://osm.etsi.org/wikipub/index.php/OSM_Autoscaling (Data di accesso: 25/09/2020)

[45] Mininet: https://www.opennetworking.org/mininet/ (Data di accesso 04/09/2020)

[46] Docker: https://www.docker.com/ (Data di accesso 04/09/2020)

[47] Container Docker: https://www.docker.com/resources/what-container (Data di accesso 04/09/2020)

[48] Docker Hub: https://hub.docker.com/ (Data di accesso 04/09/2020)

[49] Docker overview: https://docs.docker.com/get-started/overview/ (Data di accesso 04/09/2020)

[50] Vagrant: https://www.vagrantup.com/ (Data di accesso 25/09/2020)

[51] Catalogo pubblico di Vagrant box: https://app.vagrantup.com/boxes/search (Data di accesso 25/09/2020)

[52] VirtualBox: https://www.virtualbox.org/ (Data di accesso 25/09/2020)

[53] "Introduction to Vagrant": https://www.vagrantup.com/intro (Data di accesso 25/09/2020)

[54] Python – Docker Official Image: https://hub.docker.com/_/python (Data di accesso 04/09/2020)

[55] Vagrant box – ubuntu/bionic64: https://app.vagrantup.com/ubuntu/boxes/bionic64 (Data di accesso 25/09/2020)