POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Orchestrazione SDN in ambito geografico: stato dell'arte e prospettive future

Relatore
prof. Fulvio Risso

Candidato
Paolo Magliona

Dicembre 2017

Alla mia famiglia.

Indice

Elenco delle figure v

1 Introduzione 2

2 Scenario 5
2.1 Il problema della connettività end-to-end in ambito di reti geografiche 5
2.2 Caso d'uso e soluzione proposta 6

3 Background 8
3.1 ONOS: Open Network Operating System 8
3.2 OpenStack 9
3.3 Docker 11
3.4 NETCONF 12
3.5 YANG 14

4 CORD 16
4.1 Caratteristiche generali 16
4.2 Hardware 17
4.3 Software 18
4.4 Processo di trasformazione del Central Office in CORD 19
4.4.1 Disaggregazione e virtualizzazione dei device legacy 20
4.4.2 Framework dei servizi 21
4.5 Infrastruttura della rete CORD 24
4.6 Underlay Fabric 25
4.7 Virtual Network Overlay 26
4.7.1 Composizione di servizi 26
4.7.2 Il ruolo della VTN 27

5 E-CORD 28
5.1 Caratteristiche generali 28
5.2 Hardware 29
5.3 Software 30

5.4 L'applicazione Carrier Ethernet 32

6 XOS 33
6.1 Caratteristiche generali 33
6.2 Software 34
6.3 Data model e framework di sincronizzazione 36
6.4 Ambiente di sviluppo 37

7 Implementazione di un servizio XOS 39
7.1 Creazione di un servizio XOS 39
7.2 Service Graph 42
7.2.1 Modificare il Service Graph 42
7.2.2 Servizio di esempio 43
7.2.3 Deploy del servizio di esempio 45
7.3 Considerazioni generali 45

8 ONOS YANG Tools 48
8.1 Caratteristiche generali 48
8.2 YANG Compiler 49
8.3 YANG Serializer 51
8.4 YANG Runtime 51

9 Implementazione di un driver ONOS basato su YANG data model 54
9.1 Studio preliminare 54
9.2 Descrizione generale del driver 55
9.3 YANG data model 56
9.3.1 Generazione automatica di classi Java da YANG data model . 56
9.3.2 Serializzazione dei dati in formato XML 57
9.4 Struttura del driver e delle classi Java 59
9.5 Integrazione del driver in ONOS 63
9.6 Funzionamento 66
9.7 Considerazioni generali 73

10 Implementazione di una applicazione ONOS 75
10.1 Descrizione generale dell'applicazione 75
10.2 Uso del servizio di network configuration 76
10.3 Struttura dell'applicazione e delle classi Java 78
10.4 Integrazione dell'applicazione in ONOS 78
10.5 Funzionamento 81

11 Risultati ottenuti 87
11.1 Banco di Prova 88
11.2 Analisi prestazionale 89

12 Conclusioni e sviluppi futuri 91

Appendice A Creazione dell'ambiente front-end di XOS 93
A.1 Prerequisiti 93
A.2 Ambiente front-end di XOS 93

Appendice B Creazione di un driver ONOS basato su YANG data model 96
B.1 Prerequisiti 96
B.2 Procedimento 96

Bibliografia 101

Elenco delle figure

2.1 Scenario di connettività end-to-end su rete geografica tra due branch office con l'uso delle tecnologie CORD e E-CORD. 6

3.1 Struttura dell'architettura di ONOS (Fonte: Onosproject [1]). 9
3.2 OpenStack Framework (Fonte: OpenStack website [2]). 10
3.3 Docker Engine (Fonte: Docker website [4]). 11
3.4 Layer del protocollo NETCONF. 12

4.1 Hardware di riferimento costruito da commodity server, I/O blade e switch (Fonte: CORD wiki [9]). 17
4.2 Hardware di riferimento visto come due rack virtuali (Fonte: "*Central Office Re-Architected as a Data Center*" [10]). 18
4.3 Componenti software open-source in CORD (Fonte: "*Central Office Re-Architected as a Data Center*" [10]). 19
4.4 Central Office legacy che include tre dispositivi (CPE, OLT, BNG) da virtualizzare e disaggregare (Fonte: "*Central Office Re-Architected as a Data Center*" [10]). 19
4.5 Esempio di service graph CORD (Fonte: CORD wiki [9]). 22
4.6 Trellis: infrastruttura della rete CORD (Fonte: CORD wiki [9]). . . . 24
4.7 Underlay fabric in un singolo POD CORD (Fonte: CORD wiki [9]). . 25
4.8 Composizione di servizi dentro CORD (Fonte: CORD wiki [9]). . . . 26

5.1 E-CORD nella Wide Area Network (Fonte: CORD wiki [9]). 29
5.2 Implementazione del POD di riferimento di E-CORD (Fonte: CORD wiki [9]). 30
5.3 Controllore ONOS globale che gestisce il controllo dei siti CORD e della rete di trasporto del POD di E-CORD. 31

6.1 XOS visto come OS che gestisce i service controller (Fonte: "*XOS: An Extensible Cloud Operating System*" [12]). 34
6.2 Anatomia di un servizio in XOS (Fonte: "*XOS: An Extensible Cloud Operating System*" [12]). 34

6.3 Diagramma a blocchi della struttura software di XOS (Fonte: "*XOS: An Extensible Cloud Operating System*" [12]). 35
6.4 Componenti del service control plane di XOS (Fonte: CORD wiki [9]). 37

7.1 Rappresentazione della toolchain di xosgen che converte un file xproto in un file Python (Fonte: CORD wiki [9]). 40
7.2 Rappresentazione del Service Graph contenente il servizio example-service visto dalla GUI di XOS. 46

8.1 Architettura generale di funzionamento di ONOS YANG Tools (Fonte: CORD wiki [9]). 49
8.2 Struttura logica del funzionamento di YANG Compiler (Fonte: CORD wiki [9]). 50
8.3 Flusso di compilazione di YANG Compiler (Fonte: CORD wiki [9]). . 50
8.4 Processo di serializzazione tra YANG Runtime e YANG Serializer (Fonte: CORD wiki [9]). 52
8.5 Architettura di YANG Runtime (Fonte: CORD wiki [9]). 52

9.1 Rappresentazione a blocchi del funzionamento del driver basato su modelli YANG. 57
9.2 Rappresentazione delle relazioni tra Behaviour, YANG Tools e protocollo NETCONF in ONOS. 59
9.3 Rappresentazione a blocchi dei componenti ONOS con cui il driver interagisce. 68

10.1 Rappresentazione a blocchi del funzionamento dell'applicazione visto ad alto livello. 76
10.2 Rappresentazione a blocchi dei componenti ONOS con cui la applicazione interagisce. 82

11.1 Banco di prova utilizzato per le analisi prestazionali. 88
11.2 Distribuzione dei tempi impiegati per l'assegnazione di VLAN e la configurazione delle interfacce in modalità trunk e access. 90

Capitolo 1

Introduzione

Negli ultimi anni si è assistito a una rapida evoluzione nel mondo delle reti in seguito all'introduzione di tecnologie quali SDN (Software Defined Networking) e NFV (Network Function Virtualization). Infatti esse hanno modificato radicalmente il concetto di architettura di rete separando il piano di controllo da quello di trasporto dei dati e introducendo con la virtualizzazione un nuovo modo di sfruttare le funzionalità svolte dagli apparati di rete. Ciò ha permesso di adottare un modello di rete dinamico, flessibile e scalabile, in grado di garantire una più semplice gestione e una più rapida velocità di sviluppo delle risorse.

Difatti l'infrastruttura di rete, tradizionalmente composta in larga misura da componenti hardware, ha sempre richiesto grossi sforzi di manutenzione e rinnovamento per potersi adattare ai cambiamenti del futuro. Pertanto l'introduzione di queste nuove tecnologie basate sul software e sulla virtualizzazione hanno favorito una gestione più flessibile delle risorse e dei servizi, abbattendo le problematiche e le limitazioni dovute alla staticità della componentistica hardware.

Recentemente l'idea di utilizzare la tecnologia delle SDN si è diffusa tra i fornitori di servizi di rete, in particolare nel contesto geografico della *Wide Area Network* (WAN), rete di trasporto che connette fra loro più reti locali e/o metropolitane tramite una dorsale detta *backbone*. La specifica applicazione di questa tecnologia in questo ambito prende il nome di SD-WAN (Software Defined-Wide Area Network) e permette agli operatori di rete di trarre i vantaggi introdotti dalle SDN, introducendo flessibilità e scalabilità per i servizi offerti ai clienti *enterprise*.

Uno degli obiettivi principali della tecnologia SD-WAN è quello di offrire i servizi già disponibili nella WAN ma in maniera più efficiente e flessibile, riducendo notevolmente i costi di gestione. Il servizio chiave di questa tecnologia è quello di fornire connettività ai clienti *enterprise* che hanno la necessità di collegare tra loro diversi *branch office* o *data center* disposti geograficamente in luoghi diversi. Inoltre grazie al paradigma della virtualizzazione è possibile inserire agilmente ulteriori servizi business come gestione della banda, WAN accelerator o servizi cloud, a seconda delle esigenze del cliente *enterprise*.

In questo scenario, i nodi dell'infrastruttura di rete chiamati *Central Office*, posti tra la rete di accesso residenziale e il *backbone*, sono principalmente composti da apparati fisici piuttosto complessi e difficilmente aggiornabili. Per questo motivo è sorta la necessità di trasformare i *Central Office* in *data center*, in grado di virtualizzare le funzioni svolte dagli apparati fisici. Questo permette all'operatore di seguire il paradigma delle SDN, fornendo una gestione più agile dei servizi, abolendo di fatto i costi e le limitazioni dovuti dall'hardware legacy.

Nell'ambito delle possibili soluzioni, una proposta di particolare interesse può essere identificata in CORD (Central Office Re-Architected as a Data Center), progetto open-source che ha come scopo proprio quello di trasformare il tradizionale *Central Office* in un *data center*. In questo modo, tramite l'uso di hardware ampiamente disponibile sul mercato, è possibile virtualizzare i servizi che prima venivano forniti da specifici apparati fisici.

Pertanto una volta sostituito l'hardware legacy dei *Central Office* con i *data center* e implementate le funzionalità degli apparati fisici con una loro controparte software, è necessario un ulteriore elemento di orchestrazione in grado di gestire ad alto livello i servizi e la connettività tra i vari *branch office*.

In questo contesto una proposta di particolare interesse può essere identificata in E-CORD (Enterprise-CORD), progetto open-source in via di sviluppo creato con lo scopo di orchestrare un servizio di connettività tra due o più *branch office* aziendali andando a gestire i siti CORD dispositi all'interno dell'infrastruttura della rete e i servizi business richiesti dai clienti *enterprise*.

Questa tesi si propone innanzitutto di analizzare nel dettaglio le nuove tecnologie di CORD e E-CORD, andando ad esplorare le loro caratteristiche hardware e software e le funzionalità che sono in grado di fornire. In seguito si procederà ad esplorare la loro capacità di orchestrazione dei servizi tramite l'uso di un framework di alto livello chiamato XOS, un componente chiave dell'architettura CORD e E-CORD.

Inoltre, dopo aver definito uno specifico scenario di connettività end-to-end tra due host posti in due *branch office* diversi, usando il controllore di rete ONOS, verrà proposta una soluzione per gestire il dispositivo posto nella parte di rete di accesso a cui gli host sono collegati.

Infine, dopo aver implementato in ONOS il driver e la applicazione necessari per configurare il dispositivo e realizzare la connettività end-to-end, verranno analizzati i vantaggi portati dalla soluzione proposta tramite la misurazione dei risultati sperimentali ottenuti.

L'elaborato è strutturato come segue:

- **Capitolo 2:** espone il problema della connettività end-to-end nell'ambito delle reti geografiche, lo scenario preso in considerazione e la soluzione proposta in questo lavoro di tesi.
- **Capitolo 3:** presenta una panoramica delle architetture sulle quali questa tesi si basa e dei componenti utilizzati.
- **Capitolo 4:** descrive l'architettura di CORD, progetto che ridisegna il *Central Office* (CO) di una compagnia di telecomunicazioni come un data center e che viene trattato nello scenario presentato nel lavoro di tesi.
- **Capitolo 5:** descrive l'architettura di E-CORD, progetto in via di sviluppo che fornisce orchestrazione di servizi e servizi di connettività su rete geografica e che viene trattato nello scenario presentato nel lavoro di tesi.
- **Capitolo 6:** descrive l'architettura di XOS, framework contenuto nell'architettura CORD, presentando le sue componenti software.
- **Capitolo 7:** analizza nel dettaglio le componenti necessarie per implementare un servizio XOS e mostra come modificare un grafo di servizio.
- **Capitolo 8:** descrive la suite di *tool* di ONOS denominata "*YANG Tools*" e analizza nel dettaglio le sue componenti strutturali.
- **Capitolo 9:** mostra l'implementazione di un driver in ONOS nello scenario analizzato in questo lavoro di tesi.
- **Capitolo 10:** mostra l'implementazione di una applicazione in ONOS di supporto al driver scritto in questo lavoro di tesi.
- **Capitolo 11:** esamina i risultati ottenuti e le prestazioni del lavoro svolto.
- **Capitolo 12:** espone le conclusioni e presenta alcuni possibili sviluppi futuri sulla base del lavoro svolto in questa tesi.
- **Appendice A:** descrive i passi necessari per effettuare il deploy dell'ambiente front-end di XOS.
- **Appendice B:** descrive i passi che portano alla creazione di un driver ONOS basato su modelli YANG e alla generazione automatica delle classi Java che rappresentano tali modelli.

Capitolo 2

Scenario

2.1 Il problema della connettività end-to-end in ambito di reti geografiche

Nell'ambito delle reti geografiche è di particolare interesse per le aziende il servizio di connettività end-to-end tra i vari *branch office*.

Tale servizio è attualmente fornito dagli operatori di rete tramite l'uso di tecnologie WAN (Wide Area Network) legacy come ad esempio MPLS (Multiprotocol Label Switching). Tali tecnologie per quanto performanti e scalabili, sono però piuttosto costose e basate su una componente hardware del *Central Office* di difficile aggiornamento o espansione.

L'introduzione della tecnologia SD-WAN (Software Defined-Wide Area Network) permette agli operatori di rete di gestire più agilmente il servizio di connettività, abbattendo i costi e definendo, tramite un approccio basato sul software, una più semplice gestione di tale servizio e di altri servizi correlati (e.g.: gestione banda, acceleratore WAN, etc.).

Nell'ambito delle possibili soluzioni, quelle analizzate in questo lavoro di tesi possono essere identificate in CORD (Central Office Re-Architected as a Data Center), per permettere la virtualizzazione delle funzionalità del *Central Office*, e E-CORD per la orchestrazione globale del servizio di connettività *enterprise* o di ulteriori servizi business tra i vari siti CORD.

Con l'uso di CORD l'operatore di rete può diminuire considerevolmente i costi di manutenzione e aggiornamento dell'hardware, andandolo a sostituire con un più generico *data center* che ha lo scopo di virtualizzare le funzionalità svolte dagli specifici apparati fisici del *Central Office*.

Inoltre tramite l'uso di E-CORD l'operatore di rete può gestire l'orchestrazione globale dei servizi e fornire facilmente la connettività tra due host aziendali collegati a diversi siti CORD tramite una interfaccia grafica di alto livello messa a disposizione

da XOS, il componente responsabile della orchestrazione dei servizi di CORD e E-CORD.

2.2 Caso d'uso e soluzione proposta

In questa tesi si è voluto analizzare il problema della connettività end-to-end a livello due tra vari host dei *branch office* nel contesto delle reti geografiche che fanno uso delle tecnologie CORD e E-CORD.

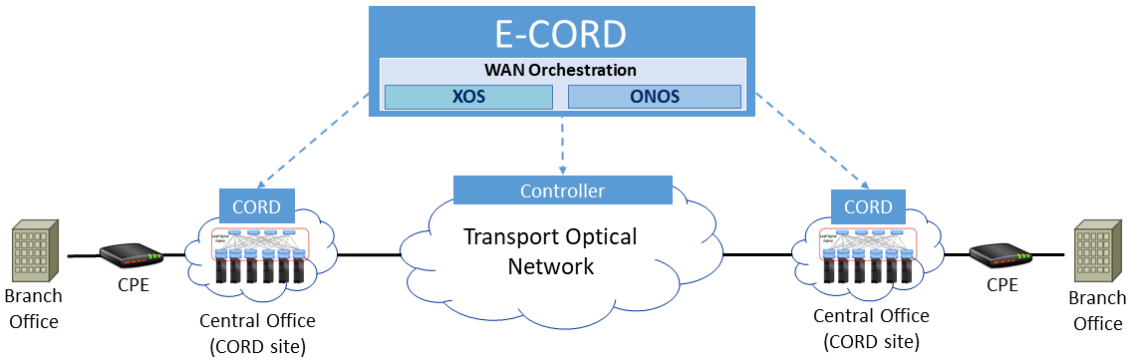

Figura 2.1. Scenario di connettività end-to-end su rete geografica tra due branch office con l'uso delle tecnologie CORD e E-CORD.

Pertanto si è definito uno specifico scenario che implementa il servizio di connettività end-to-end tra due host aziendali (Figura 2.1). Esso è composto da due *Central Office* che usano la tecnologia di CORD e da una rete ottica di trasporto che li collega. Nella parte della rete di accesso di entrambi i lati sono disposti i CPE a cui si collegano gli host aziendali di diversi *branch office*. In questo specifico caso d'uso i CPE sono collegati direttamente alla *switching fabric* di CORD e non vengono virtualizzati. Per realizzare il servizio di connettività, E-CORD si pone a capo di questo scenario andando ad orchestrare i controllori che gestiscono i due siti CORD e la rete di trasporto.

Per valutare la gestione e la possibilità di modifica del grafo dei servizi necessaria per implementare il servizio di connettività, si è proceduto con un primo approccio implementativo tramite l'uso di XOS, il componente responsabile della orchestrazione dei servizi. Tale approccio, non ha permesso però di configurare le risorse sottostanti (e.g.: ONOS, OpenStack) che gestiscono rispettivamente l'hardware e le funzionalità fornite dai servizi all'interno dei siti CORD.

Dunque, prendendo come riferimento lo scenario presentato, si è preferito focalizzare l'attenzione a livello di rete di accesso per gli host, in particolare sui CPE (Customer Premises Equipment), i dispositivi che collegano gli host aziendali ai siti CORD.

Infatti per poter implementare il servizio di connettività è necessario configurare questi dispositivi in modo da assegnare ai pacchetti particolari tag VLAN chiamati c-tag (*customer-tag*) che identificano il cliente aziendale all'interno dell'infrastruttura E-CORD.

Tali dispositivi non sono gestiti dal controllore ONOS globale di E-CORD, pertanto in questo lavoro di tesi si è deciso di integrare la loro gestione in un controllore ONOS dedicato. Si è quindi proceduto ad implementare in ONOS un driver e una applicazione volti alla configurazione di tali dispositivi, basati sull'uso di *data model* YANG e del protocollo NETCONF.

In questo modo si è esteso lo scenario SD-WAN controllato da E-CORD, aggiungendo la possibilità di controllo del CPE a livello di ONOS.

Tale controllo può così essere esercitato a seconda delle necessità dall'operatore di rete, integrandolo in E-CORD, o dal cliente aziendale, che si può così occupare in autonomia della configurazione dei dispositivi di accesso.

Capitolo 3

Background

In questo capitolo vengono introdotte le principali tecnologie che sono state analizzate nel corso dello studio dello stato dell'arte proposto in questa tesi, in modo da chiarirne i concetti fondamentali.

3.1 ONOS: Open Network Operating System

ONOS (Open Network Operating System) [1] è un progetto open-source il cui obiettivo è creare un sistema operativo di rete basato sulla tecnologia Software Defined Networking (SDN). In particolare ONOS fornisce il *control plane* per una SDN gestendo i componenti della rete (e.g: switch, link) e facendo girare programmi software o moduli che forniscono servizi di comunicazione tra host o reti. Il kernel e il core di ONOS, come anche le applicazioni, sono scritte in linguaggio Java come bundle che vengono caricati in un container Karaf OSGi. Quest'ultimo è un componente di sistema per Java che permette l'installazione e l'attivazione di moduli in una singola JVM (Java Virtual Machine).

Per permettere agli sviluppatori di interagire con il sistema operativo e di sviluppare applicazioni, ONOS fornisce un set di astrazioni e modelli di alto livello. Per evitare che il sistema diventi legato ad una specifica configurazione o ad un particolare protocollo, qualsiasi software in diretto contatto con librerie specifiche di protocollo o che si trova in stretta interazione con l'ambiente di rete viene isolato all'interno di un particolare strato denominato *driver*.

Le applicazioni possono essere gestite non solo tramite CLI, ma anche essere caricate, attivate e disattivate dinamicamente tramite API REST o GUI. Ciascuna applicazione può usufruire di una serie di servizi messi a disposizione da ONOS per poter apprendere lo stato della rete, controllare il flusso di traffico che l'attraversa, consentire la propria configurazione dall'esterno tramite REST, CLI o GUI, richiedere che particolari pacchetti le vengano recapitati e effettuare operazioni sui pacchetti stessi.

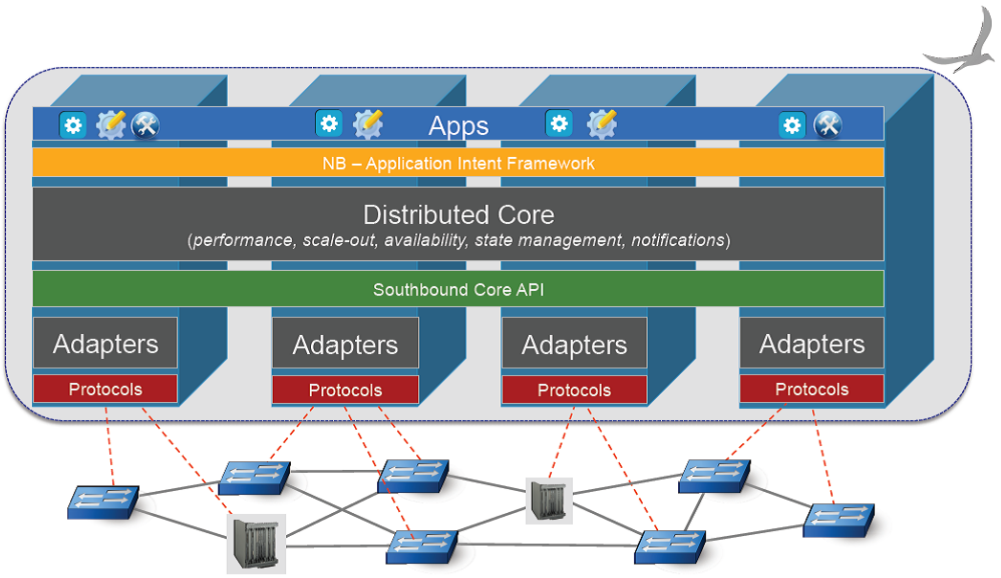

Figura 3.1. Struttura dell'architettura di ONOS (Fonte: Onosproject [1]).

I servizi offerti dalla piattaforma possiedono un certo numero di astrazioni ad alto livello, che permettono di apprendere lo stato della rete e controllare il flusso di traffico che la attraversa. Una di queste astrazioni è il *network graph*, che fornisce informazioni riguardanti la struttura e la topologia della rete. Il *flow objective* invece consente alle applicazioni di direzionare il flusso di traffico attraverso un dispositivo specifico, senza la necessità di essere a conoscenza della tabella del dispositivo stesso. Similmente, l'*intent* permette agli sviluppatori la possibilità di controllare la rete specificando ciò che si vuole realizzare, piuttosto che specificare come realizzarlo.

All'interno dell'implementazione di CORD, ONOS svolge un ruolo fondamentale come sistema operativo di rete che gestisce la *switching fabric*. Infatti ospita una collezione di applicazioni di controllo che implementano una serie di servizi ed è responsabile di integrare reti virtuali nella *fabric*.

Nel lavoro di tesi ONOS è stato utilizzato come controllore di rete SDN per configurare il dispositivo all'edge della rete dello scenario preso in considerazione. In particolare è stato sviluppato sia un driver che una applicazione di supporto col fine di configurare il dispositivo tramite il protocollo NETCONF.

3.2 OpenStack

OpenStack [2] è un progetto IaaS (Infrastructure as a Service) open-source il cui obbiettivo è creare un sistema operativo per il Cloud in grado di controllare grandi insiemi di risorse di computing, storage e networking in ogni parte di un data center.

L'architettura di OpenStack [3] può essere pensata come un framework composto da diversi moduli. Essa si basa su tre componenti principali:

- **OpenStack Compute (*Nova*)**: è un controller per il cloud computing di un sistema IaaS.
- **OpenStack Image(*Glance*)**: fornisce alle macchine virtuali i servizi di storage, registrazione e distribuzione delle immagini.
- **OpenStack Object (*Swift*)**: viene utilizzato per creare un sistema di storage ridondante e scalabile per memorizzare diversi petabyte di dati.

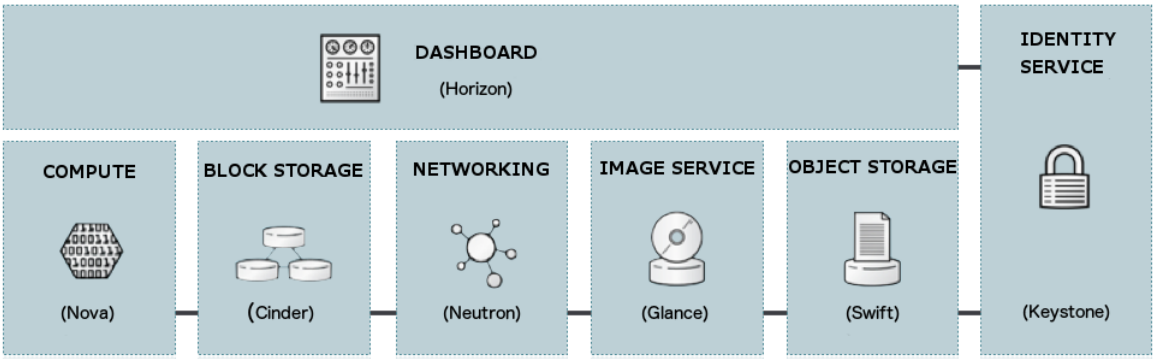

Figura 3.2. OpenStack Framework (Fonte: OpenStack website [2]).

Esistono poi altri componenti che aggiungono servizi mancanti o migliorano quelli già esistenti, come ad esempio:

- **OpenStack Block Storage (*Cinder*)**: fornisce uno storage persistente a livello di dispositivi a blocchi per il loro utilizzo da parte delle istanze di OpenStack Compute (*Nova*).
- **OpenStack Identity (*Keystone*)**: fornisce il sistema di autenticazione di OpenStack.
- **OpenStack Dashboard (*Horizon*)**: fornisce una interfaccia grafica per l'accesso e la gestione delle risorse fornite dal Cloud.
- **OpenStack Networking (*Neutron*)**: fornisce il sistema per la gestione delle reti e degli indirizzi IP.

OpenStack è uno dei tre pilastri fondamentali dell'implementazione di CORD, insieme a ONOS e XOS. Infatti oltre a rendere l'infrastruttura di CORD secondo il paradigma Cloud del *IaaS* (Infrastructure as a Service), viene utilizzato per la gestione dei cluster. Inoltre è responsabile della creazione e del provisioning di macchine virtuali e di reti virtuali.

3.3 Docker

Docker [4] è una piattaforma per lo sviluppo, la distribuzione e l'esecuzione di applicazioni. Permette di separare le applicazioni dall'infrastruttura consentendone una semplice gestione. Sfruttando Docker per la distribuzione, il trasporto ed il test del codice in modo rapido, è possibile ridurre in modo significativo il ritardo tra la scrittura del codice stesso e la messa in produzione.

Le applicazioni girano in un ambiente isolato chiamato container: l'isolamento e la sicurezza permettono l'esecuzione di diversi *container* simultaneamente nello stesso host, in numero maggiore rispetto alle virtual machines grazie alla natura leggera dei container, i quali girano senza il carico extra di un *hypervisor*.

Docker mette a disposizione dei *tool* ed una piattaforma per la gestione del ciclo di vita dei container permettendo l'incapsulamento delle applicazioni, la distribuzione, il trasporto e l'installazione in ambiente di produzione (sia in data center che nel cloud).

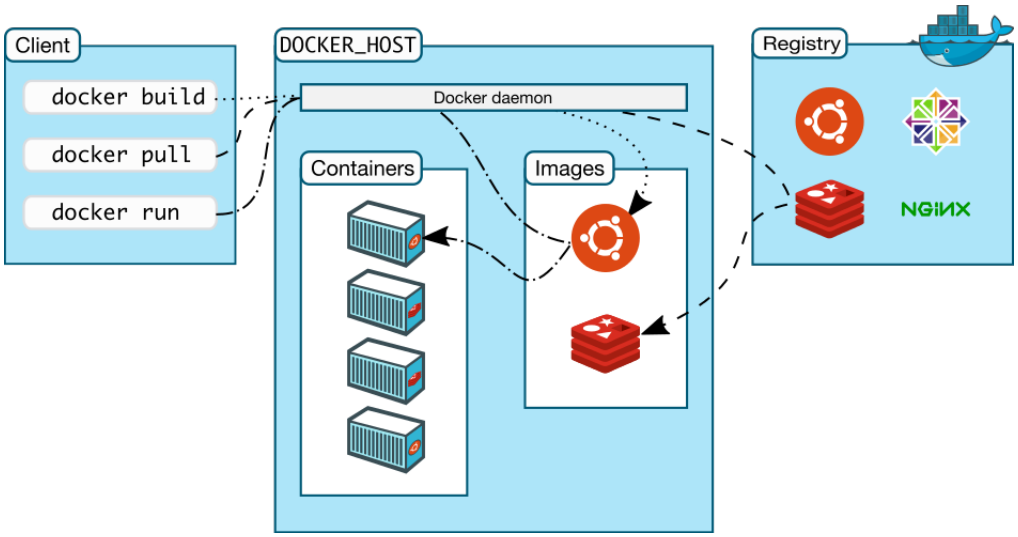

Figura 3.3. Docker Engine (Fonte: Docker website [4]).

La parte principale del sistema, che prende il nome di "*Docker Engine*" (Figura 3.3) è un'applicazione client/server composta da:

- un server di tipo *long-running program* chiamato processo demone: si occupa di creare e gestire i container, le immagini la rete e il volume dati.
- un'interfaccia client a linea di comando (CLI): l'utente usa la CLI per interagire con il client docker il cui compito è quello di ricevere i comandi, interpretarli e gestire lo scambio dei messaggi da e verso il demone. Un client può comunicare con diversi demoni.

- una REST API: definisce l'interfaccia che il client deve utilizzare per poter parlare con ed istruire il demone.

Un container è un'istanza in esecuzione di un'immagine, rappresentata da un template *read-only* contenente le istruzioni per la creazione dei container. Ad esempio un'immagine potrebbe contenere un sistema operativo Ubuntu in cui gira un *server web* Apache ed una *web application* scritta dall'utente.

Le immagini vengono memorizzate in un registro, che può essere pubblico o privato e può trovarsi sulla stessa macchina che ospita il demone o il client o su un server separato.

All'interno dell'implementazione di CORD, Docker viene utilizzato sia per istanziare XOS, OpenStack e XOS in container dedicati sia per fare il deploy e l'interconnessione di servizi.

3.4 NETCONF

NETCONF (Network Configuration Protocol) [5] è un protocollo definito da IETF (Internet Engineering Task Force) che fornisce i meccanismi per installare, manipolare e cancellare le configurazioni di un dispositivo di rete. Esso fa uso di una codifica dati basata su XML (eXtensible Markup Language) sia per i dati di configurazione sia per i messaggi scambiati dal protocollo. Le operazioni sono realizzate secondo il paradigma del RPC (Remote Procedure Call).

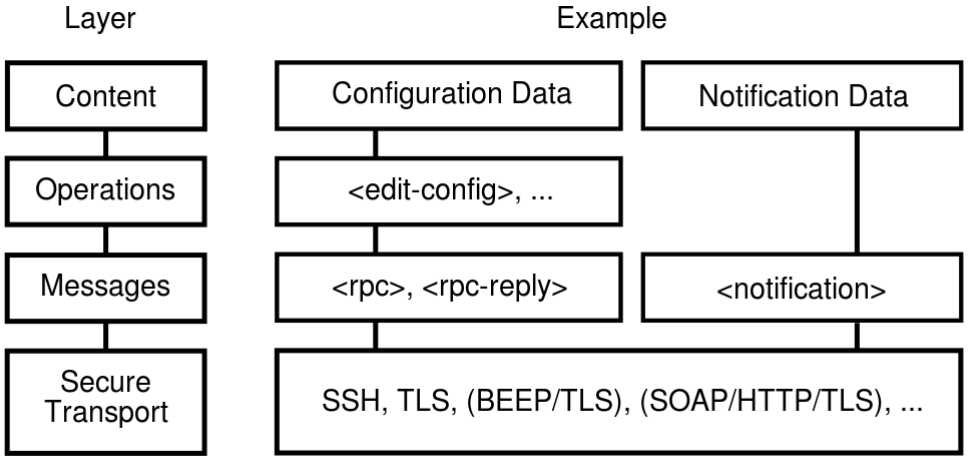

Figura 3.4. Layer del protocollo NETCONF.

Il funzionamento tipico del protocollo vede un client che codifica un RPC in XML e lo manda ad un server usando una sessione sicura. Il server risponde con un messaggio anche esso codificato in XML anche in caso di errore nella richiesta.

Il protocollo NETCONF può essere concettualmente partizionato in quattro *layer* (Figura 3.4):

1. **Content Layer**: contiene i dati di configurazione e di notifica.
2. **Operation Layer**: definisce un set di operazioni base per acquisire e modificare i dati di configurazione.
3. **Message Layer**: fornisce un meccanismo per codificare RPC e notifiche.
4. **Secure Transport Layer**: fornisce il trasporto sicuro e affidabile di messaggi tra un client e un server.

Il protocollo NETCONF fornisce un piccolo set di operazioni per gestire le configurazioni del dispositivo e acquisire informazioni sullo suo stato. Queste operazioni di base permettono di acquisire, configurare, copiare e cancellare *datastore* di configurazione. Il *datastore* è il luogo concettuale dove vengono immagazzinate o da cui vengono acquisite le informazioni.

Le operazioni di base definite dal protocollo sono:

- **<get>**: acquisisce informazioni sull'attuale configurazione e stato del dispositivo.
- **<get-config>**: acquisisce tutte o parte delle informazioni di uno specifico datastore di configurazione.
- **<edit-config>**: modifica un datastore di configurazione creando, cancellando, incorporando o sostituendo parte del contenuto.
- **<copy-config>**: copia un intero datastore di configurazione e lo sovrascrive sopra un altro datastore di configurazione.
- **<delete-config>**: cancella un datastore di configurazione.
- **<lock>**: blocca un intero datastore di configurazione di un dispositivo.
- **<unlock>**: rilascia un datastore di configurazione precedentemente bloccato con una operazione di *<lock>*.
- **<close-session>**: richiede la terminazione di una sessione NETCONF.
- **<kill-session>**: forza la terminazione di una sessione NETCONF.

Il *Message Layer* fornisce un meccanismo di framing semplice e indipendente dal *Transport Layer* per codificare:

- **<rpc>**: messaggio di richiesta, modifica o cancellazione di informazioni inviato dal client.
- **<rpc-reply>**: messaggio di risposta inviato dal server.
- **<notification>**: messaggio contenente le notifiche di un particolare evento.

Nel lavoro di tesi il protocollo NETCONF è stato utilizzato per configurare un dispositivo posto all'edge della rete dello scenario preso in considerazione.

3.5 YANG

YANG [6] è un linguaggio di modellazione dati nato per la definizione del formato dei dati inviati attraverso il protocollo NETCONF (Network Configuration Protocol [5]). Può essere utilizzato sia per modellare dati di configurazione sia dati di stato dei componenti di rete. Tale linguaggio, essendo indipendente dal protocollo con cui viene utilizzato, può essere convertito in qualunque formato di codifica (e.g.: XML, JSON) supportato dal protocollo di configurazione di rete.

Il linguaggio YANG definisce il modello in maniera modulare, rappresentando la struttura dati tramite un modello ad albero. Sono messi a disposizione una serie di tipi di dato predefiniti, da cui è possibile derivare tipi addizionali tramite estensione o restrizione. Tipi di dati più complessi possono essere ottenuti tramite *grouping*, cioè definendo un insieme di oggetti appartenenti ai tipi elementari.

I componenti principali di un modello YANG sono:

- **Module**: oggetto base che definisce il formato che una certa categoria di dati deve seguire. Un data model può essere formato da più moduli. La definizione di un modulo comprende i seguenti statement:
  - *header*: descrive il modulo;
  - *revision*: individua la versione del modulo tramite una data;
  - *definition*: rappresenta il corpo del modulo.

  Un *module* può contenere al suo interno più **submodules**.

  La direttiva *include* permette di referenziare elementi appartenenti ad un altro submodule, mentre *import* consente di farlo con elementi appartenenti ad un altro modulo.
- **Node**: elemento che contiene i dati. Può essere di quattro tipi diversi:

  - **leaf**: contiene un dato primitivo (e.g.: un intero, una stringa) e non ha nodi figli.
  - **leaf-list**: consiste in una lista di nodi leaf (tutti dello stesso tipo).
  - **container**: contiene altri nodi, siano essi leaf o altri container.
  - **list**: lista di nodi container (tutti dello stesso tipo).

- **Grouping**: definisce una struttura dati riutilizzabile più volte, sia all'interno del modulo stesso sia in moduli esterni, tramite la direttiva **uses**. Alcuni elementi del grouping possono essere modificati tramite la direttiva **refine**.

- **Choice**: definisce i nodi di diverso tipo che in un determinato punto del modello si possano trovare solamente in maniera esclusiva.

- **Augment**: definisce l'estensione di moduli esterni da parte del modulo corrente. Bisogna specificare il path del modulo da estendere, i nodi da aggiungere e dei vincoli che permettono di limitare l'estensione solo a particolari casi.

Nel corso del lavoro di tesi il linguaggio YANG è stato utilizzato per definire i *data model* contenenti le informazioni necessarie per la configurazione di un dispositivo.

Capitolo 4

CORD

In questo capitolo viene presentata l'architettura di CORD e vengono analizzate nel dettaglio le sue componenti sia hardware che software.

4.1 Caratteristiche generali

CORD (Central Office Re-Architected as a Data Center) [7] è un progetto open-source di ONF (Open Networking Foundation) [8] creato con lo scopo di sostituire hardware proprietario e chiuso con software che viene eseguito su *commodity server*, switch e dispositivi di accesso. In particolare CORD ridisegna il *Central Office* (CO) di una compagnia di telecomunicazioni come un data center. Nel fare questo vengono unificate tre tecnologie diverse:

- **Software Defined Network (SDN)**: separa il control plane della rete dal data plane e rende il primo programmabile. Questo semplifica la infrastruttura della rete e permette l'uso di white-box switch.
- **Network Function Virtualization (NFV)**: sposta il data plane da dispositivi fisici a virtual machine (VM) che girano su server. Questo permette di sostituire costosi dispositivi fisici con *commodity hardware*, riducendo i costi e permettendo una più agile orchestrazione basata su software.
- **Cloud**: definisce le procedure migliori per servizi scalabili facendo leva su soluzioni basate sul software e composizione di servizi. Ciò permette agli operatori di rete di effettuare il deploy di servizi più rapidamente.

In breve l'obiettivo di CORD non è solo quello di sostituire i dispositivi hardware specifici con una controparte software, ma anche quello di rendere il *Central Office* come parte integrante di una più grande strategia di Cloud di una compagnia di telecomunicazioni, rendendo accessibile una ricca collezione di servizi per clienti *residential*, *mobile* e *enterprise*.

4.2 Hardware

La componente hardware di CORD è basata sull'uso di una collezione di *commodity server* e *white-box switch*. Si tratta di server e switch relativamente poco costosi e ampiamente disponibili sul mercato. Questi elementi hardware sono poi organizzati in una unità rack denominata POD, la quale viene in seguito installata presso un *Central Office*. L'implementazione hardware di riferimento è basata su una particolare configurazione caratterizzata da:

- **Server**: Open Compute Project (OCP)-qualified QUANTA STRATOSS210-X12RS-IU, ognuno dei quali configurato con 128 GB di RAM, 2 x 300 GB hard-disk e 1 x NIC dual-port da 40GE.
- **Switch**: Open Compute Project (OCP)-qualified e OpenFlow-enabled Accton 6712, ognuno dei quali configurato con 32 x 40GE porte, usati sia per la *leaf* sia per la *spine* della *fabric* di CORD.
- **I/O Blade**: OCP-contributed AT&T Open GPON-NFV OLT Line Card.

Figura 4.1. Hardware di riferimento costruito da commodity server, I/O blade e switch (Fonte: CORD wiki [9]).

La *switching fabric* (Figura 4.1) è organizzata in una topologia a maglia chiamata *leaf-spine* ed è ottimizzata per far scorrere il traffico tra la rete di accesso che connette i clienti al *Central Office* e i link di upstream che collegano il *Central Office* al backbone dell'operatore. Le line card GPON-NFV OLT garantiscono la connettività alla rete di accesso: esse vanno a sostituire l'hardware proprietario con una soluzione open e software-defined.

I server, gli switch e gli OLT blade sono disposti su un singolo rack fisico ma assemblati come se fossero due rack virtuali (Figura 4.2). I server e gli OLT blade sono interconnessi da una *fabric leaf-spine*, la quale consiste in due switch di spine e due switch di leaf per ognuno dei due rack virtuali. Inoltre, come mostrato nella Figura 4.2, lo switch *"Leaf-2"* è connesso a un router di upstream e gli OLT blade

sono connessi via GPON agli ONT (Optical Network Terminals) e agli home router. I server utilizzano Ubuntu LTS 14.04 come sistema operativo e includono Open-vSwitch (OvS). Gli switch utilizzano il software open-source Atrium, il quale include Open Network Linux, Indigo OpenFlow Agent (OF 1.3) e OpenFlow Data Plane Abstraction (OF-DPA) di Broadcom.

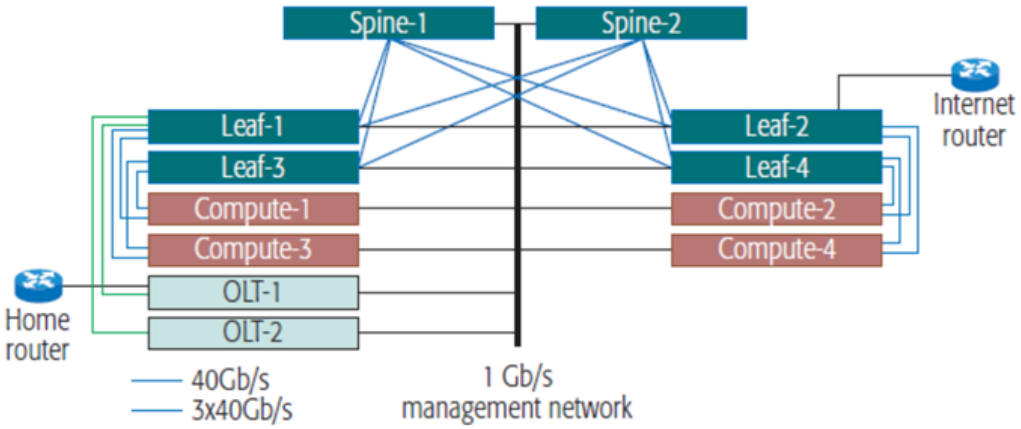

Figura 4.2. Hardware di riferimento visto come due rack virtuali (Fonte: "*Central Office Re-Architected as a Data Center*" [10]).

4.3 Software

Per quanto concerne la componente software, l'implementazione di riferimento di CORD si basa su quattro progetti open-source (Figura 4.3):

- **OpenStack**: framework che fornisce la gestione dei cluster e rende l'infrastruttura del core come un IaaS (Infrastructure as a Service). Inoltre è responsabile della creazione e del provisioning di macchine virtuali (VM) e di reti virtuali.
- **Docker**: fornisce un container su cui fare il deploy e l'interconnessione dei servizi. Inoltre ha un ruolo fondamentale nel deploy di CORD stesso (e.g: OpenStack, ONOS, XOS sono istanziati in container Docker).
- **ONOS**: sistema operativo di rete che gestisce sia switch software sia la *fabric* di switch fisici. Contiene al suo interno una sere di applicazioni di controllo che implementano servizi e gestiscono la *fabric*.
- **XOS**: framework per l'assemblaggio e la composizione di servizi. Unisce servizi di infrastruttura (forniti da OpenStack), servizi di control plane (forniti da ONOS) e qualunque altro servizio di cloud (eseguito in macchine virtuali o container).

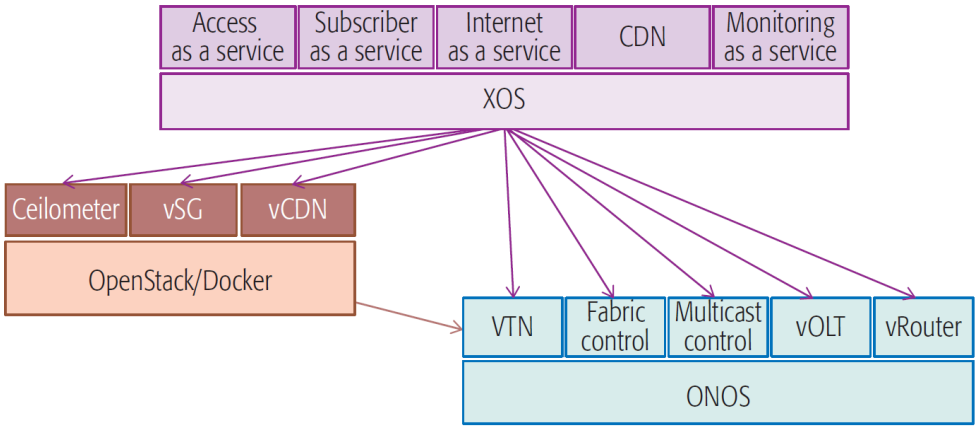

Figura 4.3. Componenti software open-source in CORD (Fonte: "*Central Office Re-Architected as a Data Center*" [10]).

4.4 Processo di trasformazione del Central Office in CORD

Una volta stabilita la configurazione hardware e software presentata nelle sezioni precedenti, trasformare il *Central Office* odierno in CORD significa effettuare un processo composto da due passi.

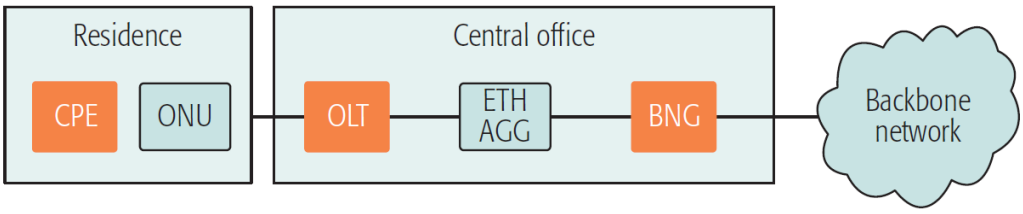

Figura 4.4. Central Office legacy che include tre dispositivi (CPE, OLT, BNG) da virtualizzare e disaggregare (Fonte: "*Central Office Re-Architected as a Data Center*" [10]).

Il primo passo è quello di disaggregare e virtualizzare i dispositivi fisici utilizzati attualmente nei *Central Office* (Figura 4.4). Con ciò si intende sostituire tutti i dispositivi hardware costruiti in modo specifico per il *Central Office* nella loro controparte software che viene eseguita su *commodity hardware*, hardware ampiamente disponibile sul mercato.

Il secondo passo è quello di fornire un framework dentro il quale possano essere inseriti i risultanti elementi disaggregati, producendo un sistema ent-to-end coerente.

4.4.1 Disaggregazione e virtualizzazione dei device legacy

I dispositivi legacy contenuti all'interno del *Central Office* e nell'area residenziale sono in quantità numerosa, ma CORD si prefigge la disaggregazione e la virtualizzazione solo di alcuni di questi per realizzare una implementazione di riferimento più semplice possibile. In particolare i *device legacy* presi in considerazione sono (come evidenziati in Figura 4.4):

- **CPE** (Customer Premises Equipment)
- **OLT** (Optical Line Termination)
- **BNG** (Broadband Network Gateway)

Lo switch *Ethernet Aggregation* presente in Figura 4.4 non viene disaggregato e virtualizzato in quanto viene effettivamente poi rimpiazzato dalla *switching fabric* sotto il controllo di ONOS. L'OLT è un apparato molto costoso del *Central Office* e coinvolge diversi rack di hardware proprietario e chiuso dove termina l'accesso per decine di migliaia di *subscribers*. Virtualizzare l'OLT significa eliminare definitivamente questo hardware dedicato e molto oneroso in termini di costi. Anche le CPE distribuiti nelle case dei *subscribers* sono in quantità molto numerosa: la virtualizzazione eliminerebbe anche in questo caso il costo operazionale e di distribuzione, specialmente quando un upgrade di un servizio comporta la sostituzione dell'hardware. I BNG sono router complessi e costosi i quali aggregano al loro interno quasi tutte le funzionalità fornite dal *Central Office*. Per questo motivo è difficile poterli evolvere in maniera rapida e efficiente in termini di costi.

La disaggregazione e virtualizzazione di ognuno di questi dispositivi fisici risulta in tre elementi:

1. Uso di *commodity server* e *white-box switch*.
2. Un elemento SDN per la funzione di control plane.
3. Un elemento NFV per la funzione di data plane.

Entrambi gli elementi SDN e NFV sono implementati da software eseguito su *commodity server*: si considera NFV se il processamento dei pacchetti viene eseguito interamente via software, mentre si considera SDN se il software controlla anche i *white-box switch* e gli I/O blade con una interfaccia come OpenFlow.

Le versioni virtualizzate di CPE, OLT e BNG prendono il nome rispettivamente di: vSG, vOLT e vRouter.

vSG (virtual Subscriber Gateway)

La versione virtualizzata del CPE è chiamata vSG (virtual Subscriber Gateway). Il CPE in questo contesto viene inteso come un *home router* installato nella sede del *subscriber*. Come un normale CPE, il vSG permette l'esecuzione di un bundle di funzioni pensate per il *subscriber* (e.g: DHCP, NAT, Firewall) ma esse vengono eseguite nel *commodity hardware* del *Central Office* piuttosto che nella sede del *subscriber*. In particolare viene creato un container per ogni *subscriber* in cui viene eseguito Linux che implementa un bundle di funzioni di base, come ad esempio Firewall o filtri di Parental Control. Esiste ancora un dispositivo presente nella sede del subscriber ma si tratta di un *bare-metal switch*: uno switch generico senza particolari funzioni pensate per il *subscriber*.

vOLT (virtual Optical Line Termination)

L'OLT legacy termina i suoi link ottici nel *Central Office*: ogni punto di terminazione fisico aggrega un insieme di connessioni di diversi *subscriber*. La virtualizzazione viene raggiunta attraverso una specifica I/O blade che implementa dei chip sotto controllo di un programma a controllo remoto via OpenFlow. Questo programma viene chiamato vOLT (virtual OLT) e viene eseguito in ONOS e implementa tutte le altre funzionalità normalmente contenute in un OLT legacy. Il vOLT implementa anche l'autenticazione e gestisce le VLAN che connettono i dispositivi dei *subscriber* alla *switching fabric* del *Central Office*.

vRouter (virtual Router)

Il BNG legacy fornisce i mezzi attraverso i quali i subscriber possono connettersi a Internet. Esso mette a disposizione un indirizzo IP *routable* per ogni *subscriber* e raggruppa in un bundle un insieme di funzioni (e.g.: QoS, VPN, MPLS). Il BNG virtualizzato di CORD, denominato vRouter (virtual Router), viene implementato come un programma di controllo installato dentro ONOS che gestisce i flussi che scorrono nella *switching fabric* per conto di ogni *subscriber*. Molte delle funzioni ausiliare riunite nel BNG non sono riprodotte dal vRouter. In generale è più accurato pensare il vRouter come un router virtuale privato e personale per ogni *subscriber*. In questa ottica la *switching fabric* può essere vista come un grosso router distribuito. Un'altra funzionalità principale del vRouter riguarda quella di parlare protocolli di routing con altri router esterni.

4.4.2 Framework dei servizi

Dopo aver effettuato la virtualizzazione dei dispositivi legacy è necessario gestire questi elementi software tramite una orchestrazione che permetta di creare un sistema end-to-end funzionante e controllabile. Così come tutti i dispositivi hardware

di un *Central Office* devono essere cablati fisicamente tra loro, così anche le loro controparti software hanno la necessità di essere collegate tra loro. Questo processo viene definito *service orchestration*.

Anziché utilizzare un modello che concateni semplicemente le VM insieme tra loro, in CORD viene usato un approccio chiamato XaaS (Everything as a Service). Secondo questo approccio vOLT, vSG e vRouter sono visti non solo come la controparte virtualizzata dei tre corrispondenti dispositivi fisici, ma come servizi stessi. Con questo nuovo punto di vista essi implementano i seguenti servizi:

- **vOLT**: è un programma di controllo eseguito dentro ONOS. Implementa l'*Access-as-a-service*, dove ogni *tenant* acquisisce un *subscriber VLAN*.
- **vSG**: è una funzione di data plane distribuita tra un insieme di container. Implementa il *Subsriber-as-a-service*, dove ogni *tenant* acquisisce un *subscriber bundle*.
- **vRouter**: è un programma di controllo eseguito dentro ONOS. Implementa l'*Internet-as-a-service*, dove ogni *tenant* acquisisce una sottorete *routable*.

In questa modo le funzioni di controllo (eseguite su ONOS), le funzioni di data plane (eseguite su VM) e altri vari servizi Cloud globali (e.g.: CDN (Content Distribution Network)) vengono eseguiti come servizi scalabili. Inoltre anche l'infrastruttura stessa viene gestita come un servizio (IaaS).

Il *Central Office* legacy rappresentato in Figura 4.4 viene quindi ridisegnato nel *service graph* mostrato in Figura 4.5.

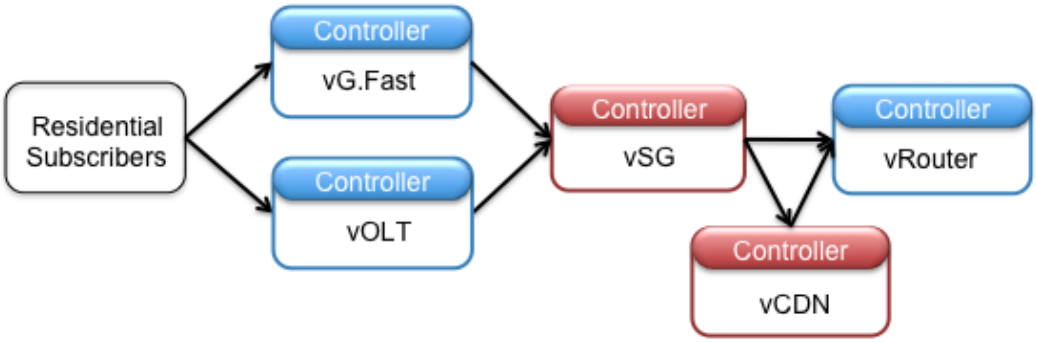

Figura 4.5. Esempio di service graph CORD (Fonte: CORD wiki [9]).

La Figura 4.5 include anche un servizio vG.Fast per rappresentare una seconda tecnologia di accesso utilizzabile in alternativa al vOLT e il vCDN come servizio cloud. I blocchi rappresentati in blu sono relativi a servizi di *control plane*, mentre quelli rossi sono relativi a servizi di *data plane*.

Il *subscriber* è *tenant* del *service graph* nella sua interezza. Questo significa che una volta che il *service graph* mostrato in Figura 4.5 viene istanziato in CORD, il *subscriber* è in grado di controllare la sua sottoscrizione (e.g.: impostare il parental control per bloccare l'accesso ad alcuni siti) senza nessuna consapevolezza di quale servizio implementi certe funzionalità.

Layer di astrazione

Il *service graph* mostrato in Figura 4.5 rappresenta la specifica di alto livello che fornirebbe un operatore di rete. Questa specifica però deve essere mappata nei server, switch e I/O blade sottostanti. La realizzazione di questa mappatura è una diretta conseguenza di un insieme di astrazioni annidate tra loro che CORD stratifica sopra i componenti mostrati in Figura 4.3.

Usando un approccio top-down, CORD definisce le seguenti astrazioni e i corrispondenti meccanismi che le implementano:

- **Service Graph**: rappresenta l'insieme delle relazioni di dipendenza tra un set di servizi. CORD modella la composizione di servizi con una relazione di *tenancy* tra un servizio del provider e un servizio del tenant (e.g.: cliente).
- **Service**: rappresenta un programma scalabile e *multi-tenant*. CORD modella il servizio come un *service controller* che esporta una interfaccia *multi-tenant* e un insieme di istanze di servizi che sono collettivamente istanziate a loro volta in una *slice*.
- **Slice**: rappresenta il container in cui vengono eseguiti i servizi, specificando anche come le risorse vengono incoroporate nella infrastruttura sottostante. CORD modella la *slice* come un insieme di macchine virtuali (VM) e di reti virtuali (VN). Le VM sono implementate dai componenti sottostanti (OpenStack e Docker), mentre le VN sono implementate da ONOS.
- **Virtual Network**: rappresenta l'interconnessione che mette in comunicazione un insieme di istanze.

Il meccanismo che permette a CORD di supportare le reti virtuali viene implementato da una coppia di applicazioni di controllo che viene eseguita dentro ONOS:

1. **VTN (Virtual Tenant Network)**: applicazione di controllo che installa regole di flusso negli OvS (Open vSwitch) che sono eseguiti su ogni server per implementare l'indirizzamento diretto o indiretto.
2. **Segment Routing**: applicazione di controllo che implementa i flussi aggregati tra i server attraverso la *switching fabric*.

4.5 Infrastruttura della rete CORD

L'infrastruttura della rete CORD è una combinazione della *leaf-spine fabric* di *underlay*, delle reti virtuali di *overlay* e del controllo SDN unificato (tramite ONOS) di queste due componenti. L'insieme di questi tre elementi prende il nome di Trellis (Figura 4.6).

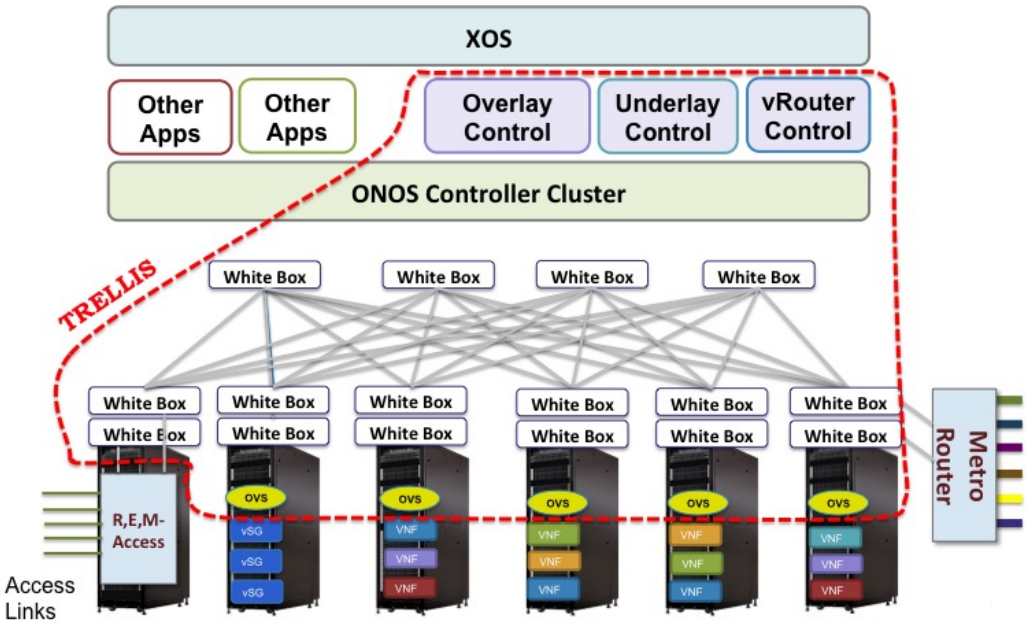

Figura 4.6. Trellis: infrastruttura della rete CORD (Fonte: CORD wiki [9]).

In Figura 4.6 è mostrato un unico cluster per il controller ONOS, ma nell'implementazione di riferimento di CORD in realtà sono presenti due cluster:

1. **onos-cord**: cluster del controllore ONOS responsabile dell'infrastruttura di *overlay* (reti virtuali e composizione di servizi) e dell'infrastruttura di accesso. Questo cluster contiene al suo interno le applicazioni VTN, che gestisce la parte di *overlay*, e vOLT, che gestisce la parte di accesso.

2. **onos-fabric**: cluster del controllore ONOS responsabile di controllare l'*underlay fabric* e di interfacciarsi con i router di *upstream*. Questo cluster contiene al suo interno le applicazioni *Segment Routing*, che gestisce la *fabric* di switch, e *vRouter*, che gestisce il routing di *upstream*.

4.6 Underlay Fabric

La *underlay fabric* di switch viene controllata da una applicazione ONOS chiamata "*Segment Routing*". Tale applicazione viene usata per trasportare il traffico incapsulato con header VxLAN tra i servizi della rete di *overlay*. La *underlay fabric* si comporta come una rete IP/MPLS che instrada il traffico tra le sotto-reti dei rack contenenti i server. Il traffico dentro una sotto-rete (e quindi dentro un rack) è di livello 2. Invece per il traffico tra server in rack differenti, la *fabric* effettua routing di livello 3 con MPLS.

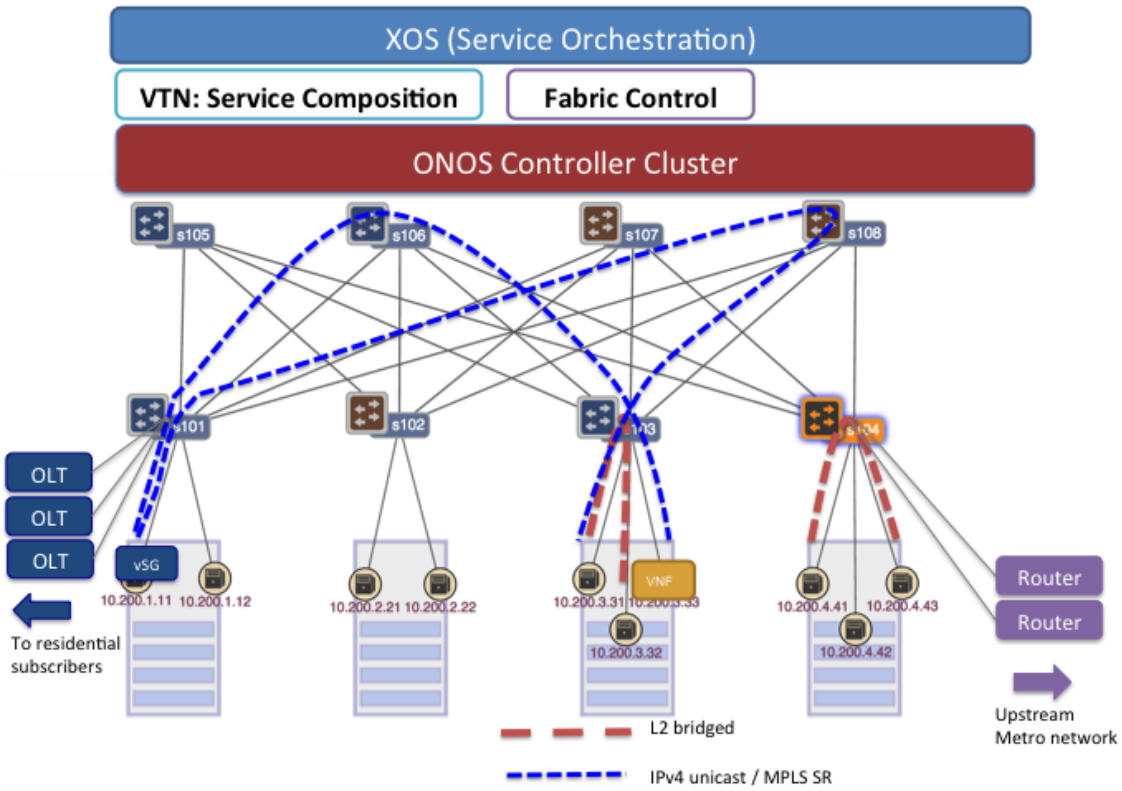

Figura 4.7. Underlay fabric in un singolo POD CORD (Fonte: CORD wiki [9]).

In Figura 4.7, ogni rack di server ha la sua sotto-rete del tipo "*10.200.x.0/24*". I server nello stesso possiedono indirizzi IP appartenenti alla stessa sotto-rete (per esempio *10.200.1.11* e *10.20.1.12* nel primo rack in Figura 4.7). Il traffico tra questi server è *bridged* dallo switch *leaf switch* a cui il rack è connesso. Invece il traffico destinato ad altri rack viene instradato dallo stesso *leaf switch* verso gli *spine switch* ma a livello 3 e con l'uso di etichette MPLS. Gli *spine switch* si occuperanno poi di instradare correttamente il traffico alla destinazione in base all'etichetta MPLS presente nei pacchetti.

4.7 Virtual Network Overlay

In CORD i servizi sono istanziati dall'operatore di rete usando XOS (Figura 4.8). A sua volta XOS presenta a ONOS un *service graph* per il traffico del *subscriber*. Questo grafo viene decomposto in regole di flusso che sono programmate nell'infrastruttura di rete di CORD tramite l'applicazione "*VTN*" di ONOS.

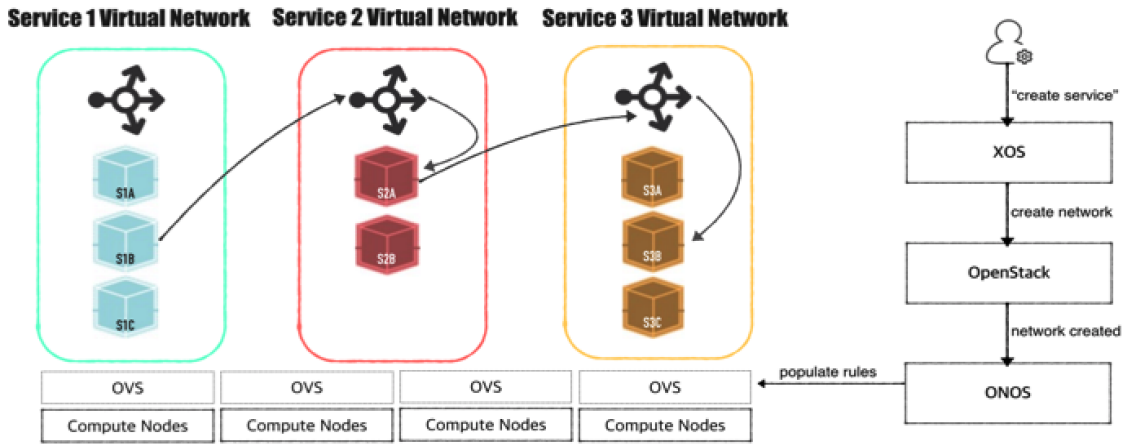

Figura 4.8. Composizione di servizi dentro CORD (Fonte: CORD wiki [9]).

4.7.1 Composizione di servizi

In CORD la composizione di servizi viene implementata usando le reti di *overlay* e la virtualizzazione di rete. La descrizione seguente analizza la composizione di servizi ad alto livello.

I servizi possiedono le proprie reti virtuali (VN) e le macchine virtuali (VM) o container che istanziano il servizio sono parte della stessa rete virtuale. Ogni nodo di *compute* opsita al suo interno VM o container che sono connessi a OvS che agiscono come *hypervisor switch* programmabili tramite OpenFlow. Ogni rete virtuale (o servizio) ha il suo *load-balancer* distribuito in ogni OvS nella rete. Il compito dei *load-balancer* è quello di selezionare una VM che istanzia il servizio, tra tutte le VM all'interno della rete virtuale del servizio.

XOS e l'applicazione VTN di ONOS agiscono in maniera coordinata tra di loro per mantenere aggiornato lo stato dell'infrastruttura virtuale. Infatti quando vengono aggiunte o eliminate delle VM per un servizio e vengono quindi create catene di servizi, l'applicazione VTN aggiorna le regole di flusso negli OvS per realizzare l'instradamento del traffico per il servizio voluto dal *subscriber*.

Per trasportare il traffico del *subscriber* tra i vari servizi viene utilizzata *VxLAN encapsulation*. Tramite l'uso di *tunnel* VxLAN i servizi vengono interconnessi tra di loro.

4.7.2 Il ruolo della VTN

Per definire una configurazione di CORD è necessario stabilire un insieme di servizi organizzati in un *service graph*. Ogni servizio è connesso a una o più *management network* e a una o più *data network*.

L'applicazione VTN di ONOS definisce due *management network*. Una di queste (denominata *management host*) permette a istanze poste in diversi *compute node* di parlare tra di loro. Invece la *data network* connette tutte le istanze di un servizio tra di loro. Interconnettendo la *data network* ad altre reti è possibile collegare tra loro i servizi che compongono il *service graph* da attuare.

Il compito dell'applicazione VTN è quello di programmare gli switch software (e.g: OvS) per inoltrare pacchetti da/a porte delle istanze di un servizio.

Essa svolge un ruolo fondamentale nella gestione dell'infrastruttura di *overlay* di CORD connettendo le istanze alle reti e fornendo una rete privata per un insieme di istanze.

Capitolo 5

E-CORD

In questo capitolo viene presentata l'architettura di E-CORD e vengono analizzate nel dettaglio le sue componenti sia hardware che software. Si sottolinea che E-CORD è attualmente in via di sviluppo e al momento esiste solo come *proof of concept*.

5.1 Caratteristiche generali

E-CORD (Enterprise - CORD) [7] è un progetto open-source di ONF (Open Networking Foundation) [8] attualmente in via di sviluppo, creato con lo scopo di offrire servizi di connettività alle aziende (clienti *enterprise*) su rete metropolitana (MAN - *Metropolitan Area Network*) ma soprattutto su rete geografica (WAN - *Wide Area Network*) usando solamente software open-source e *commodity hardware*.

E-CORD si sviluppa sopra l'infrastruttura CORD presentata nel capitolo precedente e si concentra sull'offrire la connettività tra i vari siti CORD (*Central Office* ridisegnati come datacenter) disposti geograficamente in luoghi differenti.

Sfruttando le potenzialità di CORD al posto dei *Central Office* legacy, i *service provider* possono offrire servizi di connettività aziendale (L2VPN e L3VPN) e in più permettere di includere VNF (Virtual Network Function) e ulteriori servizi cloud utili ai clienti *enterprise*.

I clienti *enterprise* stessi possono collegarsi ad una interfaccia grafica e usare E-CORD per creare rapidamente reti on-demand tra qualunque numero di filiali aziendali e configurare VNF *on-the-fly* (e.g.: firewall, tool di analisi di traffico, router virtuali, acceleratori WAN, etc.) come servizi che vengono forniti e mantenuti dentro la rete del *service provider* (Figura 5.1).

Il servizio di connettività end-to-end tra due filiali aziendali poste in luoghi geografici differenti prende il nome di "*E-Line*". Mentre se si tratta di connettività tra più di due filiali allora si parla di "*E-LAN*" (Figura 5.1).

L'obiettivo che si prefigge E-CORD è quello di fornire alle aziende una orchestrazione a livello globale dei servizi di connettività su rete geografica, garantendo

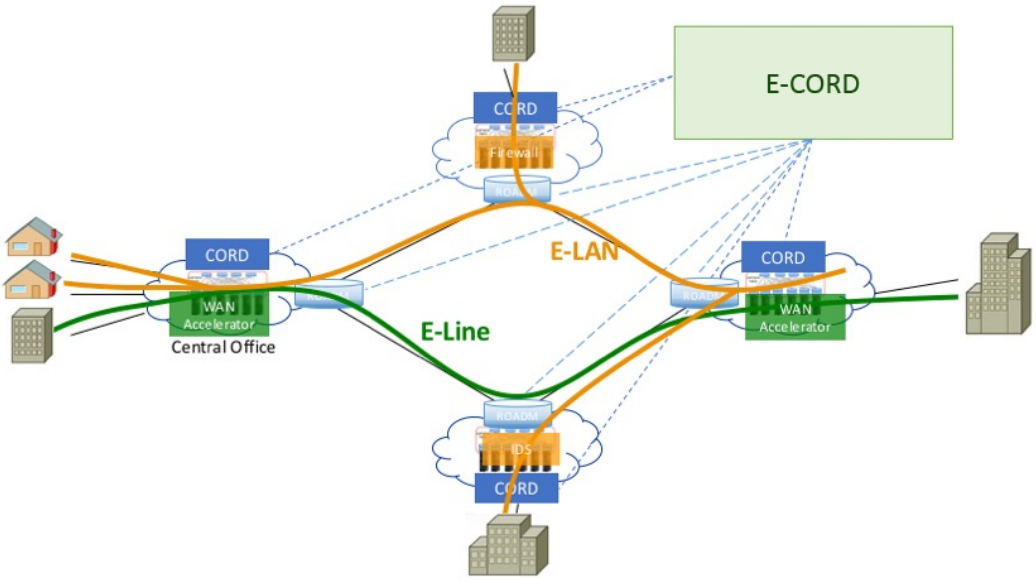

Figura 5.1. E-CORD nella Wide Area Network (Fonte: CORD wiki [9]).

allo stesso tempo l'elasticità e la scalabilità offerta dai siti CORD nell'usufruire di servizi VNF o cloud forniti dal *service provider*.

5.2 Hardware

L'infrastruttura hardware di E-CORD è formata da un insieme di siti CORD connessi tra loro con una rete di trasporto. Quest'ultima può essere di qualunque tipo (da una rete ottica a un semplice switch) e può essere composta sia da equipaggiamento legacy sia da *white-box*. L'insieme dei componenti che vanno a comporre E-CORD viene denominato "*POD*".

Benché sia possibile connettere tra loro un ampio numero di siti CORD, l'implementazione hardware di riferimento è basata su una particolare configurazione più semplice (Figura 5.2). In questa implementazione il POD di E-CORD contiene tre siti CORD connessi tra loro da un *packet switch* come nodo di trasporto.

Per semplificarne la rappresentazione grafica, ogni sito CORD riportato in Figura 5.2 è in realtà un *half-POD*, cioè contiene la metà dei componenti hardware che la configurazione di riferimento di un sito CORD dovrebbe contenere. I componenti contenuti dall'*half-POD* dei siti CORD sono:

- 1 *fabric switch*

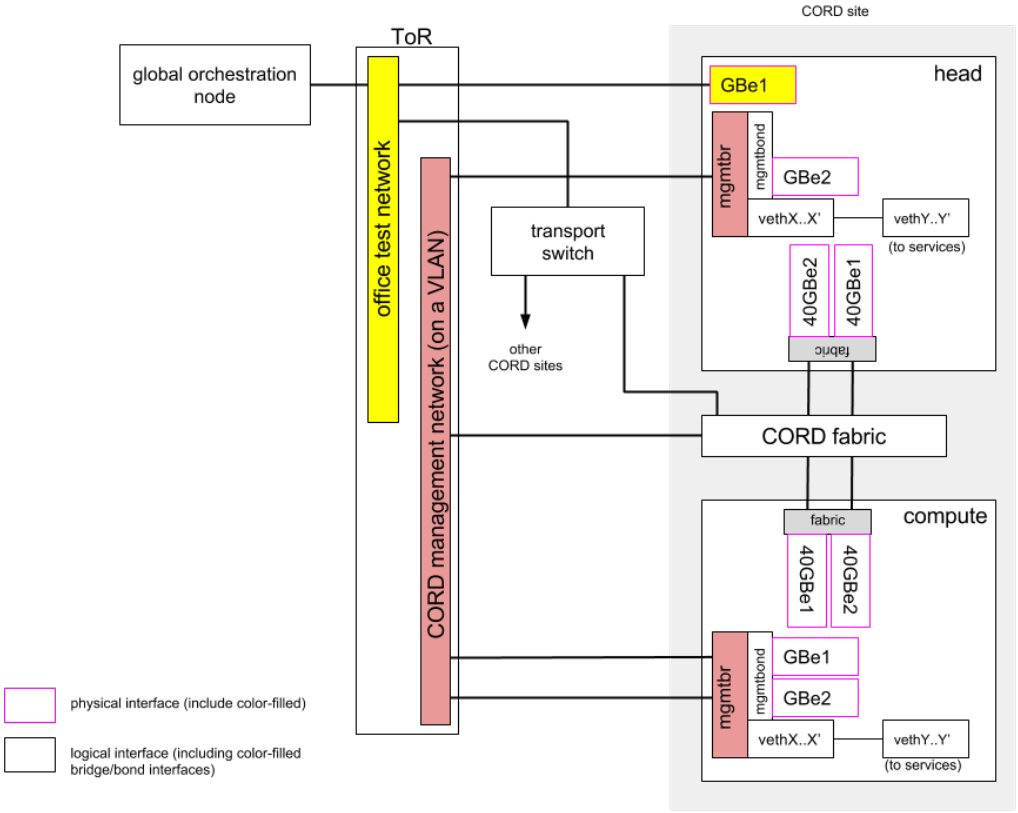

Figura 5.2. Implementazione del POD di riferimento di E-CORD (Fonte: CORD wiki [9]).

- 1 *head node*
- 1 *compute node*

Il POD di E-CORD include inoltre un nodo addizionale in cui vengono eseguiti i componenti dell'orchestratore globale, il quale è indirettamente connesso ai siti CORD e allo switch di trasporto attraverso la *management network*.

Il nodo globale interagisce con l'*head node* di ogni sito e la rete di trasporto, in maniera tale da connettersi sia alla rete dei *Central Office* sia allo switch dedicato alla rete di trasporto.

5.3 Software

La componente software di E-CORD può essere suddivisa in due parti distinte: quella che compone i siti CORD e quella che compone il nodo globale di orchestrazione del POD di E-CORD.

I vari siti CORD che compongono il POD di E-CORD si basano su quattro progetti open-source (presentati in maniera più approfondita nella sezione "*Software*" del Capitolo 4). Questi sono i responsabili della gestione del *control plane* e del *data plane* di CORD e sono:

- **OpenStack**
- **Docker**
- **ONOS**
- **XOS**

Invece il nodo globale che interagisce con i vari siti CORD e la rete di trasporto si basa su una installazione semplificata di XOS (Capitolo 6), priva dei componenti di OpenStack. Infatti questo nodo non ha necessità di fornire servizi di infrastruttura tramite VM o VN, ma piuttosto si interessa della orchestrazione dei servizi posti già all'interno dei siti CORD e della loro inter-connettività.

Dunque in questa particolare implementazione di XOS, ONOS svolge un ruolo fondamentale, in quanto deve occuparsi della gestione del control plane per i siti CORD e per la rete di trasporto (Figura 5.3).

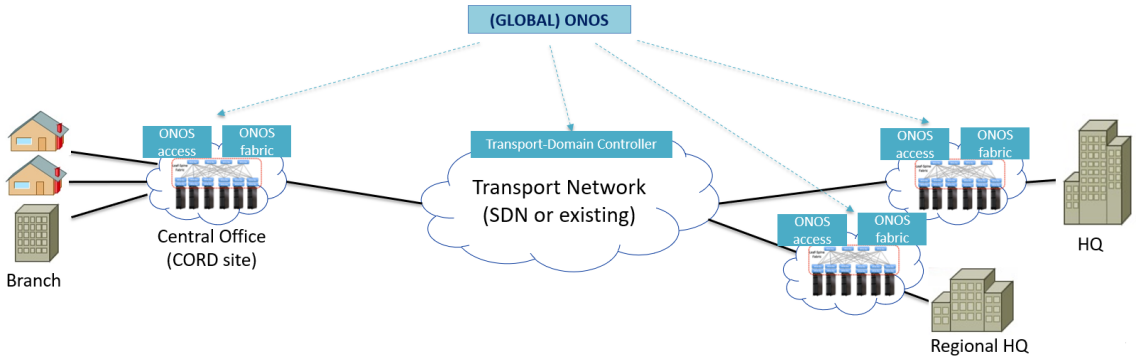

Figura 5.3. Controllore ONOS globale che gestisce il controllo dei siti CORD e della rete di trasporto del POD di E-CORD.

In E-CORD, il framework di XOS viene quindi usato parzialmente e ha una funzionalità più di alto livello: gestire l'orchestrazione della connettività e dei servizi tra più filiali aziendali disposte geograficamente in luoghi differenti fornendo ai clienti *enterprise* una interfaccia grafica semplice da utilizzare.

5.4 L'applicazione Carrier Ethernet

A livello topologico il controllore ONOS globale vede i vari siti CORD come dei *Big Switch* e appoggiandosi all'applicazione "*Carrier Ethernet*" crea degli EVC (Ethernet Virtual Circuit) che mettono in comunicazione i siti CORD tra loro.

Questi circuiti virtuali rispettano le normative definite dal MEF (Metro Ethernet Forum) [11] riguardanti i servizi *Carrier Ethernet* e permettono anche l'impostazione di profili di banda (*bandwidth profile*) e degli UNI (User Network Interface), responsabili del punto di demarcazione fisico tra *subscriber* e *service provider*.

L'applicazione è composta dalle seguenti classi Java:

- **CarrierEthernetManager**: compie la validazione, installazione (se possibile) e rimozione del servizio *Carrier Ethernet*.
- **CarrierEthernetVirtualConnection**: rappresenta un EVC del servizio *Carrier Ethernet* includendo il relativo insieme di UNI, profili di banda e tipo di servizio.
- **CarrierEthernetBandwidthProfile**: rappresenta un profilo di banda di uno *ingress* UNI e include attributi rilevanti come *Committed Information Rate* (CIR), *Committed Burst Size* (CBS), *Excess Information Rate* (EIR) and *Excess Burst Size* (EBS).
- **CarrierEthernetUni**: rappresenta uno UNI e include attributi rilevanti come il relativo profilo di banda e gli id della CE-VLAN.
- **CarrierEthernetProvisioner**: gestisce la connettività nel dominio della rete a pacchetto.
- **CarrierEthernetPacketNodeManager**: classe astratta usata per controllare i *packet node* attraverso la rete in base al loro protocollo di controllo e al tipo di connettività.

Capitolo 6

XOS

In questo capitolo viene presentata l'architettura di XOS e vengono analizzate nel dettaglio le sue componenti software.

6.1 Caratteristiche generali

XOS è un framework costruito sulla base dei due progetti open-source OpenStack e ONOS in grado di gestire i servizi scalabili che vengono eseguiti dentro CORD. Per questo è un componente vitale dell'architettura di quest'ultimo e al suo interno viene definito come il *service abstraction layer*, ovvero il livello di astrazione dei servizi. XOS rende possibile l'assemblaggio e la composizione di servizi e nel fare questo unisce la gestione di applicazioni SDN di controllo, di funzioni virtuali NFV e di servizi cloud. Inoltre esso definisce una astrazione dei servizi che fornisce una interfaccia unificata a un insieme di servizi.

XOS è strutturato secondo il paradigma del Everything-as-a-Service (XaaS): ogni sua componente sottostante viene visto come un servizio. Sotto questa filosofia di pensiero sia applicazioni di controllo di rete fornite da ONOS sia VM istanziate in OpenStack vengono indistintamente trattati come servizi.

Più in generale si può pensare a XOS come il sistema operativo di CORD, in quanto al suo interno possiede lo stesso ruolo di un sistema operativo tradizionale in un computer: fornisce le astrazioni di programmazione generale per supportare un ampio spettro di applicazioni, gestendo allo stesso tempo le sottostanti risorse hardware e i servizi software.

Tale sistema operativo viene posto a gestione di un insieme di servizi (Figura 6.1). XOS si basa su un modello secondo il quale tutti i servizi incorporati al suo interno forniscono un *service controller* che esporta una interfaccia programmabile, mentre il servizio vero e proprio viene implementato da un numero elasticamente scalabile di *service istance* (Figura 6.2). Le istanze dei servizi possono essere potenzialmente distribuite su insieme di cluster disposti geograficamente in luoghi diversi.

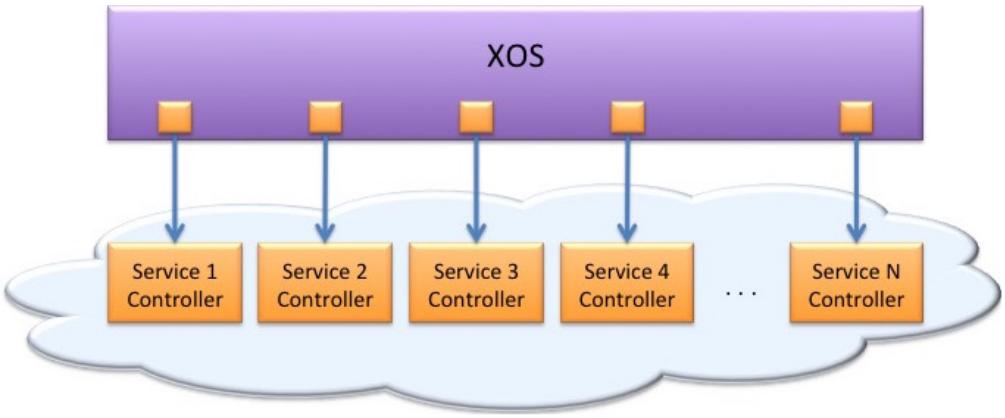

Figura 6.1. XOS visto come OS che gestisce i service controller (Fonte: "*XOS: An Extensible Cloud Operating System*" [12]).

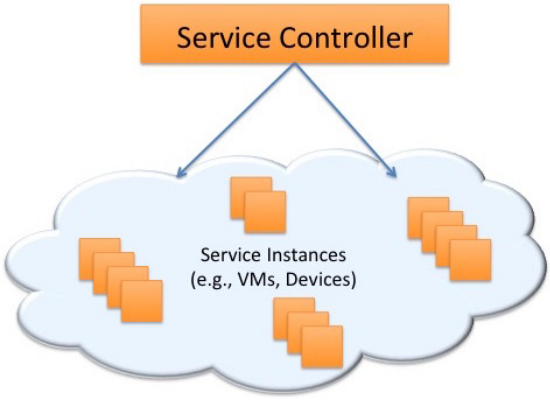

Figura 6.2. Anatomia di un servizio in XOS (Fonte: "*XOS: An Extensible Cloud Operating System*" [12]).

Questa separazione tra *service controller* e *service istance* è centrale nel design di XOS. Il *service controller* mantiene lo stato autoritativo del servizio ed si occupa di configurare le istanze sottostanti. Inoltre esporta una interfaccia globale per il servizio, utilizzabile da chi interagisce col controller (e.g.: operatore), mantenendo nascosti i dettagli dell'implementazione.

6.2 Software

XOS è organizzato in tre strati principali: *Data Model*, *View* e *Controller Framework* (Figura 6.3).

Al centro vi è il *Data Model*, che registra lo stato logicamente centralizzato del sistema. È proprio il *Data Model* che lega tutti i servizi insieme tra loro, permettendogli di operare in maniera efficiente. La centralizzazione logica di questo stato è raggiunta attraverso una separazione tra lo stato autoritativo e lo stato operazionale. Il *Data Model* incapsula gli oggetti astratti, le relazioni e le operazioni di questi oggetti. Le operazioni sono esportate sia con una interfaccia RESTful HTTP, sia tramite una libreria (*xoslib*), la quale fornisce una interfaccia di programmazione di più alto livello.

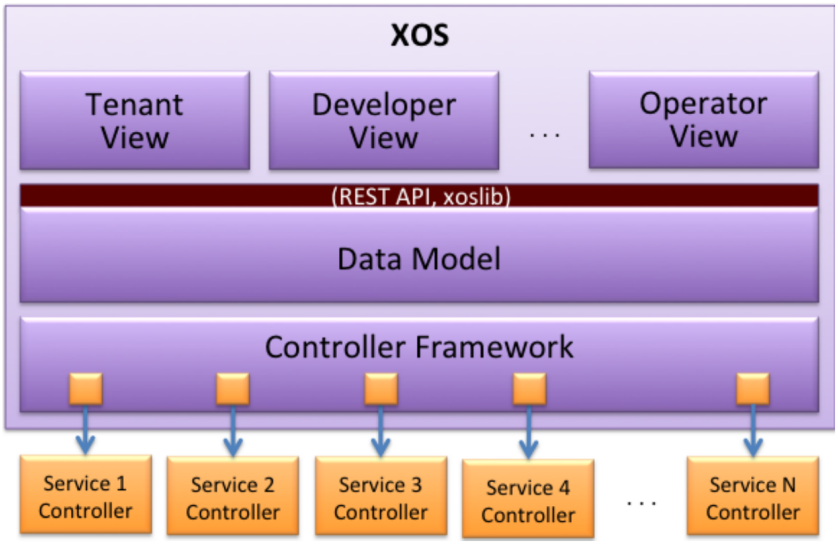

Figura 6.3. Diagramma a blocchi della struttura software di XOS (Fonte: "*XOS: An Extensible Cloud Operating System*" [12]).

Sopra il *Data Model*, le *View* (viste) definiscono la lente attraverso la quale gli utenti interagiscono con XOS. Per esempio in Figura 6.3 è possibile osservare una *View* su misura per i *tenant*, un'altra per gli sviluppatori e un'altra ancora per gli operatori.

Infine il *Controller Framework* è responsabile di mantenere lo stato rappresentato da un insieme distribuito di *service controller* in sincronia con lo stato autoritativo mantenuto dal *Data Model*. Il *Controller Framework* è un componente critico di XOS: vincola lo stato autoritativo logicamente centralizzato al resto del sistema, sincronizza le *policy* specificate nei livelli più elevati con i meccanismi di più basso livello e le mantiene in uno stato coerente tra loro.

Il *Data Model* è implementato in Django, le *View* sono programmi Javascript che vengono eseguiti nel browser dell'utente, mentre *xoslib* è una libreria client/server accessibile tramite API REST. Il *Controller Framework* è un programma basato su

Ansible per gestire le configurazioni di più basso livello con i controllori delle risorse di back-end (e.g.: ONOS, OpenStack).

6.3 Data model e framework di sincronizzazione

XOS rappresenta il grafo dei servizi con una implementazioni basata essenzialmente su due parti:

1. Un **data model** dichiarativo che definisce lo stato associato ad ogni servizio insieme ad una interfaccia per operare sulla *tenancy* dei servizi.

2. Un **framework di sincronizzazione** (basato sul concetto di *Synchronizer*) che direziona come questo stato dichiarativo viene tradotto in operazioni sulle risorse sottostanti (e.g.: OpenStack, ONOS).

Per mantenere lo stato operazionale delle risorse sottostati in sincronia con lo stato del modello, il framework di sincronizzazione si divide a sua volta in un meccanismo composto da tre parti (Figura 6.4):

1. *Synchronizer*

2. *Model Policy plugin*

3. *Ansible*

Ansible [13] è un software open-source che consente di automatizzare le procedure di configurazione e gestione all'interno di sistemi UNIX-like.

Nel contesto di XOS viene utilizzato per eseguire una serie di azioni necessarie per portare lo stato operazionale delle risorse di back-end in linea con il corrispondente stato autoritativo del data model. Per descrivere queste azioni viene utilizzato il linguaggio YAML (YAML Ain't Markup Language) [14], adibito alla serializzazione di dati utilizzabile da esseri umani.

Le informazioni che descrivono le azioni da prendere sono contenute in vere e proprie "ricette" (*recipe*) chiamate *playbook*. È tramite l'esecuzione dei *playbook* che XOS opera direttamente sulle risorse di back-end.

Il *Synchronizer* ha il compito di generare i *playbook* che Ansible usa per pilotare le risorse sottostanti. Per fare ciò, il *Synchronizer* prima analizza il data model determinando l'esistenza degli oggetti e delle loro dipendenze. Poi tramite gli appositi *model policy plugin* genera i playbook.

Una volta generati i *playbook* in grado di configurare le risorse di back-end in maniera allineata allo stato del data model, Ansible li esegue andando a operare sulle risorse sottostanti (e.g.: ONOS, OpenStack).

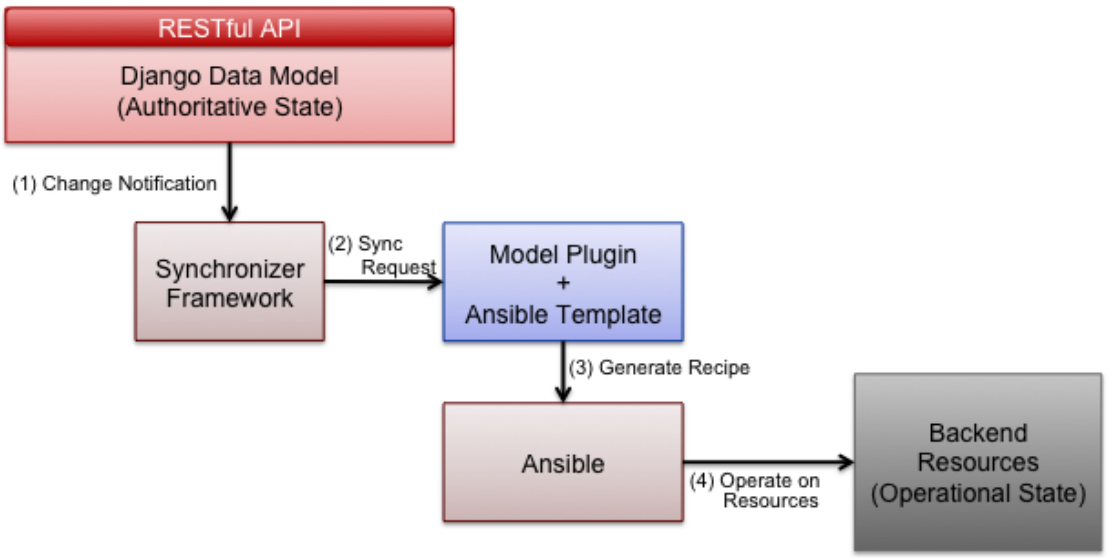

Figura 6.4. Componenti del service control plane di XOS (Fonte: CORD wiki [9]).

6.4 Ambiente di sviluppo

Per poter creare e lavorare sui servizi XOS in tutte le loro componenti (compresa l'interazione con le risorse sottostanti) è necessario avere CORD o CORD-in-a-Box come ambiente di sviluppo.

CORD-in-a-Box è un ambiente virtualizzato dentro il quale vengono installati tutti i componenti del POD di CORD in forma di macchine virtuali. La comodità di CORD-in-a-Box è quella di poter avere una versione virtualizzata di CORD all'interno di una singola macchina fisica (server).

I requisiti hardware per eseguire CORD-in-a-Box sono:

- Server 64-bit AMD64/x86-64 con:
  - 48GB+ RAM
  - 12+ core CPU
  - 200GB+ Hard-Disk

Inoltre è possibile lavorare solamente sugli aspetti di modellazione del grafo di servizio, GUI o API senza la necessità di avere ONOS o OpenStack sottostanti, creando una VM con solo XOS installato.

In questa versione XOS funzionerà solamente come front-end e non potrà utilizzare nessuna delle risorse di back-end. Lavorare sulla versione front-end di XOS permette di agevolare i tempi di build e la modellazione del *service graph*.

Infine in questa versione i *Synchronizer* sono supportati solo parzialmente: possono essere creati ma non possono interagire con le risorse sottostanti in quanto assenti.

Capitolo 7

Implementazione di un servizio XOS

In questo capitolo vengono analizzate nel dettaglio le componenti necessarie per implementare un servizio XOS. Inoltre viene mostrato come modificare il *Service Graph* in modo da aggiungere tale servizio nella catena dei servizi.

7.1 Creazione di un servizio XOS

Come descritto nel capitolo precedente (Capitolo 6), XOS si appoggia su un framework che interagisce sia con lo stato autoritativo del sistema (*data model*) sia con lo stato operazionale (*sincronizzazione*) che opera sulle risorse di back-end. Dunque per implementare un servizio XOS è necessario scrivere le componenti che permettono al sistema di incorporarlo all'interno dell'ambiente XOS.

Tali componenti sono:

1. *Xproto (Django data model)*
2. *REST API*
3. *TOSCA API*
4. *Synchronizer*
5. *Ansible playbook*
6. *On-boarding Specification*

Xproto

Xproto è una variante di *Google Protocol Buffers* [15], un linguaggio simile ad XML pensato per serializzare dati strutturati, creato appositamente per incorporare le funzionalità del *data modeling* di XOS. L'obiettivo di Xproto è quello di codificare i *data model* di XOS e facilitare la generazione del codice che dipende da essi. In particolare, partendo da un file in formato "*.xproto*", tramite un programma chiamato "*xosgen*" viene generato automaticamente il *data model Django* [1], il quale prima della release di CORD 3.0 doveva essere scritto manualmente. Una volta scritto il modello in linguaggio *Xproto*, questo viene tradotto in una rappresentazione intermedia che a sua volta viene convertita in un file Python contenente il modello Django corrispondente.

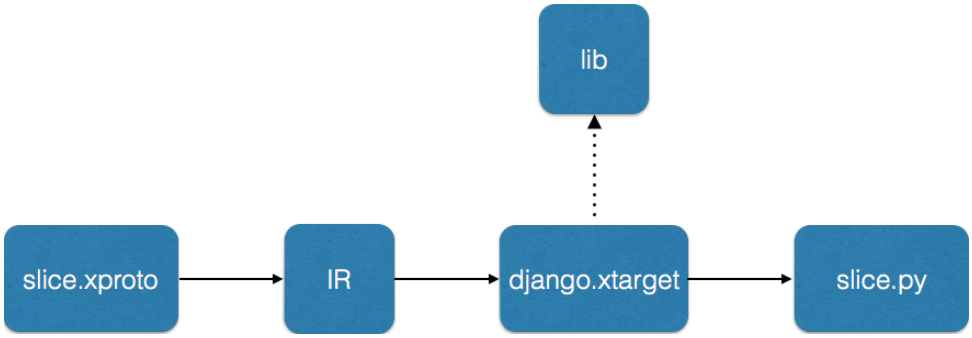

Figura 7.1. Rappresentazione della toolchain di xosgen che converte un file xproto in un file Python (Fonte: CORD wiki [9]).

In Figura 7.1 viene mostrato un esempio del funzionamento del procedimento di generazione automatica del data model. Il file "*slice.xproto*" viene prima trasformato in una rappresentazione intermedia (IR - Intermediate Representation) e poi passato a sua volta ad un template "*django.xtarget*" scritto in *Jinja2* [2] che prende la rappresentazione intermedia e genera, appoggiandosi a una libreria, il file finale "*slice.py*" scritto in Python e contenente il *data model Django*.

REST API e TOSCA API

In un altro file Python vengono definite le REST API per permettere alle componenti client di poter interagire col servizio.

Esistono inoltre due tipi di API:

[1] Django [16] è un web framework open source scritto in linguaggio Python per lo sviluppo di applicazioni web che segue il pattern Model-View-Controller.

[2] Jinja2 [17] è un motore di template per Python.

1. *Service API*
2. *Tenant API*

Le prime sono globali per l'intero servizio, le secondo sono specifiche ad un particolare *tenant*.

Viene inoltre definita una risorsa TOSCA per permettere al servizio di essere configurato tramite linguaggio TOSCA [3]. Il primo passo nel creare una TOSCA API è quello di definire i TOSCA *custom type* utilizzati dal servizio. Esiste un file scritto in *linguaggio m4* [4], il quale utilizzando delle macro genera il file TOSCA in formato "*.yaml*" che rappresenta il template della configurazione del servizio. Il file "*.yaml*" contiene quindi la definizione dei *custom type* utilizzati dal servizio che vengono poi utilizzati in fase di build per configurare il servizio con le caratteristiche desiderate.

Synchronizer e Ansible playbook

I *Synchronizer* sono processi che girano continuamente per verificare cambiamenti ai modelli. Quando un *Synchronizer* rileva un cambiamento al modello, lo applicherà a tutto il sistema sottostante richiamando un *playbook* Ansible. Per stabilire che azioni deve compiere Ansible, generalmente si crea un *playbook* generico che racchiude delle informazioni generali per il servizio e un insieme di "*role*" (ruoli) che vengono poi definiti singolarmente in altri file.

I ruoli vengono scritti per alleggerire il *playbook* principale. Ogni singolo ruolo richiama tutta una serie di "*task*" definiti all'interno dei file che caratterizzano i ruoli stessi. I "*task*" sono le azioni che Ansible deve effettivamente svolgere interagendo col sistema sottostante (e.g.: comandi shell, chiamate REST, etc). In alternativa si possono scrivere i "*task*" direttamente all'interno del *playbook* in maniera esplicita senza l'uso dei ruoli.

On-boarding Specification

Ogni servizio deve avere una "*recipe*" (ricetta) di *on-boarding*. Per *on-boarding* si intende la fase in cui il servizio viene effettivamente caricato su XOS e reso attivo. La ricetta di on-boarding viene scritta utilizzando le specifiche TOSCA e contiene la lista delle risorse necessarie per il funzionamento del servizio. In particolare contiene i percorsi dei file creati prima (e.g.: *data model Django*, *Synchronizer*, *custom type* TOSCA, risorsa TOSCA, servizio REST, etc.).

[3] TOSCA (Topology and Orchestration Specification for Cloud Applications) [18] è un linguaggio sviluppato da OASIS per descrivere la topologia dei servizi web per il cloud, i componenti, le relazioni e i processi che li gestiscono.

[4] m4 [19] è un linguaggio macro general-purpose.

7.2 Service Graph

Il *Service Graph* mostra tutti i servizi attivi e caricati su XOS e le dipendenze di *tenancy* che esistono sono tra loro. Esso può essere consultato per mezzo della GUI di XOS accessibile tramite browser. Il senso delle frecce che collegano un servizio all'altro all'interno del grafo, stabilisce il rapporto di *tenancy* che vi è tra di loro. Ad esempio un servizio *vOLT* da cui parte una freccia che punta al servizio *vSG*, indica che il *vOLT* è *tenant* di *vSG*.

Esiste la possibilità di vedere un "*fine-grained*" *Service Graph*, una versione dettagliata del grafo che mostra tutti i servizi e i *tenant* e anche le reti utilizzate. Nel caso di R-CORD (Residential-CORD) sono visibili i servizi: *vOLT, vSG, vRouter, ONOS_Fabric, ONOS_CORD* e *vtr*.

Utilizzando la versione front-end di XOS è possibile caricare un "profilo" di R-CORD privo della parte di back-end (e.g: "*mock-rcord*") che permette di visualizzare gli stessi servizi a livello di grafo e modello senza la necessità di avere le risorse operazionali sottostanti.

Per "profilo" si intende l'entità che definisce l'insieme dei servizi che caratterizzano un particolare POD CORD. Ad esempio il profilo denominato "*rcord*", definisce i servizi residenziali utilizzati da R-CORD e contiene quindi quelli citati in precedenza (e.g: *vOLT, vSG*, *vRouter*, etc.). Invece il profilo "*mcord*" orientato più al deploy di un CORD da integrare nel mondo del *mobile*, denominato M-CORD (Mobile-CORD), contiene ulteriori servizi come ad esempio il *vRAN* (virtual Radio Access Network). Inoltre è possibile creare nuovi profili CORD personalizzati in cui inserire specifici servizi a seconda delle esigenze dell'operatore di rete.

Nel lavoro svolto in questa tesi è stato preso in considerazione il profilo "*mock-rcord*", il quale contiene i servizi che caratterizzano R-CORD e permette di lavorare sulla versione front-end di XOS.

7.2.1 Modificare il Service Graph

Per poter inserire nuovi servizi e stabilire i rapporti di *tenancy* tra questi e i servizi già esistenti è necessario effettuare alcune modifiche al codice.

In particolare è necessario eseguire i seguenti passaggi:

1. Creare il nuovo servizio (data model, REST API, TOSCA API, Synchronizer, etc.) e aggiungerlo alla cartella "*xos_services*".

2. Aggiungere nome e percorso al profilo CORD utilizzato (e.g.: il profilo "*mock-rcord.yaml*" che contiene la lista di tutti i servizi che compongono quel POD di CORD).

3. Creare una *entry* nel *data model* dei servizi CORD per il nuovo servizio configurando le sue specifiche definite nella risorsa TOSCA del servizio e importare il TOSCA *custom type* del nuovo servizio.

4. Definire sempre nel *data model* dei servizi CORD le relazioni di *tenancy* tra il nuovo servizio e gli altri già esistenti.

7.2.2 Servizio di esempio

In questo lavoro di tesi si è voluto analizzare la possibilità di modificare il *Service Graph*.

In particolar modo si è posta l'attenzione sull'inserimento di un servizio nel *Service Graph* e sulla possibilità di avere biforcazioni all'interno del grafo.

Quindi si è deciso di implementare un servizio di esempio e di utilizzare la versione front-end di XOS, in quanto sufficiente per la parte di modellazione del grafo e meno onerosa in termini di risorse hardware richieste. Prendendo come modello base quello di "*mock-rcord*" si è provato a modificare il *Service Graph* di default composto da: *vOLT, vSG* e *vRouter*.

In particolare si è provato ad aggiungere un nuovo servizio chiamato "*exampleservice*" e a collegarlo al *vSG*. Tale servizio è uno di quelli già esistenti e disponibili in R-CORD, ma di *default* non viene utilizzato. Esso non svolge nessun comportamento particolare e non interagisce con le risorse di back-end, per questo motivo è privo della parte relativa al *Synchronizer*.

Si è quindi proceduto a:

1. Aggiungere nome e percorso di "*exampleservice*" al profilo "*mock-rcord.yaml*" (Listato 7.1).

```
59 xos_services:
60   - name: volt
61     path: onos-apps/apps/olt
62   - name: onos
63     path: orchestration/xos_services/onos-service
64   - name: vrouter
65     path: orchestration/xos_services/vrouter
66   - name: vsg
67     path: orchestration/xos_services/vsg
68   - name: vtr
69     path: orchestration/xos_services/vtr
70   - name: fabric
71     path: orchestration/xos_services/fabric
72   - name: exampleservice
```

```
73      path: orchestration/xos_services/exampleservice
```

Listing 7.1. Definizione di nome e percorso del servizio exampleservice.

2. Importare il TOSCA *custom type* di "*exampleservice*" in "*cord-services.yaml.j2*" (Listato 7.2).

```
5  imports:
6     - custom_types/xos.yaml
7     - custom_types/vtr.yaml
8     - custom_types/exampleservice.yaml
```

Listing 7.2. Importazione del TOSCA custom type del servizio exampleservice.

3. Creare una *entry* che definisce le specifiche del servizio in "*cord-services.yaml.j2*" (Listato 7.3).

```
53 # CORD Services
54     service#exampleservice:
55       type: tosca.nodes.ExampleService
56       properties:
57         view_url:
              /admin/exampleservice/exampleservice/$id$/
58         kind: exampleservice
59         service_message: hello
```

Listing 7.3. Definizione delle specifiche del servizio exampleservice.

4. Aggiungere la relazione di *tenancy* tra *vSG* e "*exampleservice*" sempre nel file "*cord-services.yaml.j2*" (Listato 7.4).

```
100     service#vsg:
101       type: tosca.nodes.VSGService
102       requirements:
103         - vrouter_tenant:
104             node: service#vrouter
105             relationship:
                    tosca.relationships.TenantOfService
106         - exampleservice_tenant:
107             node: service#exampleservice
108             relationship:
                    tosca.relationships.TenantOfService
```

Listing 7.4. Definizione della relazione di tenancy tra il servizio exampleservice e il vSG.

7.2.3 Deploy del servizio di esempio

Una volta terminate le modifiche al codice si procede col *deploy* del front-end di XOS specificando il profilo desiderato (in questo caso "*mock-rcord*") e richiamando un *playbook* Ansible che andrà a caricare in automatico tutti i servizi e tutte le componenti di XOS necessarie.

Per una descrizione dettagliata sulla creazione dell'ambiente di front-end di XOS si rimanda alla Appendice A.

Il comando bash da eseguire per procedere al *deploy* del profilo "*mock-rcord*" è il seguente:

```
1 $ ansible-playbook -i inventory/mock-rcord
    deploy-xos-playbook.yml
```

In caso fosse già attivo un profilo, per apportare i cambiamenti e visualizzarli nella GUI è necessario fare prima il *teardown* di XOS sempre tramite un *playbook* Ansible e poi fare nuovamente il *deploy* di XOS sempre specificando il profilo desiderato (in questo esempio "*mock-rcord*").

Il comando bash da eseguire per procedere al *teardown* del profilo "*mock-rcord*" è il seguente:

```
1 $ ansible-playbook -i inventory/mock-rcord
    teardown-playbook.yml
```

Una volta completato il *deploy* tramite l'esecuzione del *playbook* contenente le istruzioni per caricare i servizi e le componenti di XOS, nella homepage della GUI di XOS è possibile vedere la rappresentazione grafica del *Service Graph* (Figura 7.2).

Come mostrato nella Figura 7.2, si è riusciti a modificare il grafo iniziale contenente *vOLT, vSG* e *vRouter*, inserendovi il servizio "*exampleservice*" e raggiungendo l'obiettivo iniziale di creare una biforcazione del *Service Graph*.

È possibile vedere dal grafo come il *vSG* sia il *tenant* del *vRouter* e del servizio "*exampleservice*". Infatti esaminando il senso delle frecce che connettono questi servizi tra loro, è possibile constatare che esse sono uscenti dal *vSG* e puntano verso il *vRouter* e il servizio "*exampleservice*".

7.3 Considerazioni generali

Essendo CORD ancora in via di sviluppo, non è ancora stata rilasciata una documentazione di XOS per gli sviluppatori. Essa infatti è un elemento fondamentale per un progetto open-source come CORD per permettere alla community degli sviluppatori di poter implementare servizi o sviluppare nuove funzionalità.

Inoltre sono necessarie delle API con cui interfacciarsi per poter interagire con la piattaforma di XOS, ma attualmente esse non sono state definite in modo esplicito

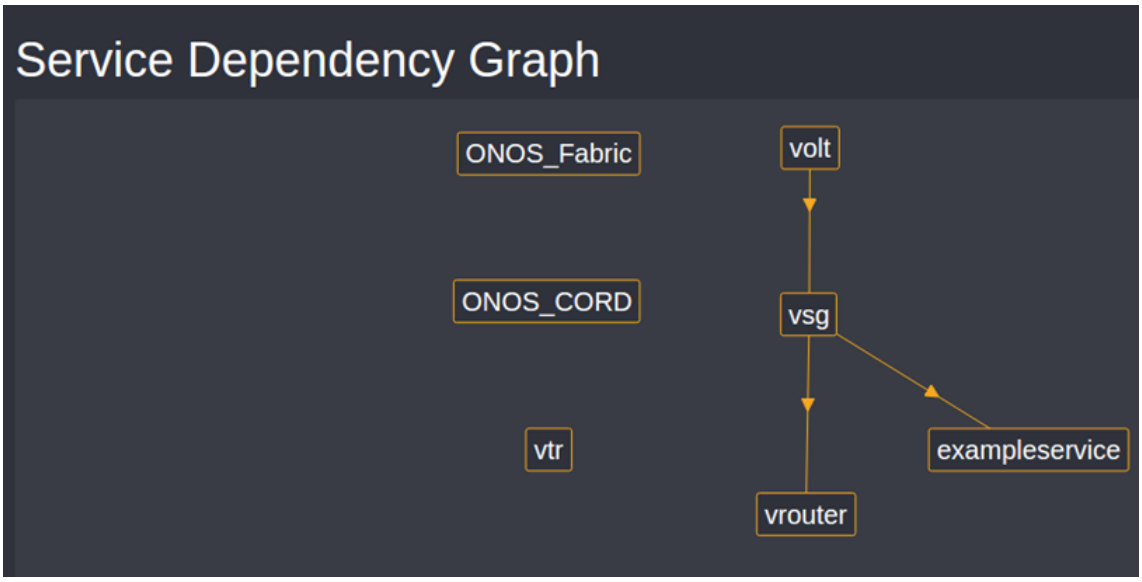

Figura 7.2. Rappresentazione del Service Graph contenente il servizio exampleservice visto dalla GUI di XOS.

proprio a causa della natura in via di sviluppo di CORD. Per questo motivo, non avendo a disposizione l'insieme delle procedure disponibili all'interno di XOS, nè gli sviluppatori nè gli operatori di rete possono interagire attivamente col framework.

In particolare la mancanza di documentazione sugli elementi fondamentali necessari per la creazione di un servizio (e.g.: Xproto, template TOSCA, Synchronizer etc.) e la modifica di un *Service Graph*, ha portato grosse difficoltà nella implementazione del servizio di esempio mostrato nelle sezioni precedenti.

Pertanto in questo lavoro di tesi si è deciso di utilizzare un approccio di tipo *trial and error*, in cui sono stati fatti diversi tentativi prima di giungere ad una soluzione funzionante.

Inoltre l'elevata modularità del codice distribuito in numerosi *playbook* Ansible e, più in generale, il codice di difficile lettura, non agevolano la comprensione dell'istanziazione delle VM in OpenStack e dell'interconnessione dei servizi tra loro.

I file YAML che contengono i *playbook* generalmente presentano al loro interno poche righe di codice in cui vengono eventualmente richiamati in maniera ricorsiva altri *role* o *task*, ognuno dei quali svolge parte delle azioni necessarie per effettuare l'*on-boarding* dei servizi. Questi file sono presenti in grande quantità e benché da un lato semplifichino la automazione e l'esecuzione dei processi di *on-boarding* di XOS, dall'altro rendono difficoltoso per uno sviluppatore inserirsi in questa catena di chiamate a funzioni e comprendere quali *playbook* modificare per poter implementare i propri servizi o aggiungere funzionalità.

Per questo motivo si è deciso di concentrare il lavoro di tesi relativo all'implementazione di un servizio XOS solamente sulla versione front-end di XOS. In questa versione si è potuto esplorare il livello superiore di XOS che permette di visualizzare e gestire i servizi a livello di grafo, senza interazione con le risorse operazionali sottostanti.

Quindi un ulteriore sviluppo del servizio mostrato nelle sezioni precedenti potrebbe essere quello di integrare gli elementi relativi alla parte di back-end di XOS, implementando la gestione del framework di sincronizzazione e l'interazione con OpenStack e ONOS.

Capitolo 8

ONOS YANG Tools

In questo capitolo viene presentata la suite di *tool* di ONOS denominata "*YANG Tools*" e vengono analizzate nel dettaglio le sue componenti strutturali.

8.1 Caratteristiche generali

YANG Tools [20] è un progetto di infrastruttura di ONOS con lo scopo di sviluppare le *tool chain* e le librerie necessaria per fornire supporto alle applicazioni JAVA basate sull'uso di NETCONF e YANG. In particolare esso si pone l'obiettivo di astrarre le applicazioni dal processamento sintattico delle informazioni ottenute dal mondo esterno.

In questo modo fornisce alle applicazioni un framework in cui esse necessitano solamente di implementare la logica di business, supportando pienamente qualunque linguaggio di interfacciamento (e.g.: REST, NETCONF, etc.). Per logica di business in questo contesto si intende la logica applicativa che rende l'applicazione in grado di interagire con un dispositivo tramite il protocollo NETCONF [5].

I *data model* contenenti le informazioni di configurazione dei dispositivi su cui si basa *YANG Tools* sono scritti in linguaggio YANG [6].

Come mostrato in Figura 8.1, le entità principali che pilotano il funzionamento di *YANG Tools* sono:

- **Model**
- **YANG Compiler**
- **YANG Serializer**
- **YANG Runtime**

I *model* sono una rappresentazione interna dei modelli YANG e vengono usati da differenti componenti di *YANG Tools* (e.g.: *YANG Compiler*, *YANG Runtime*). In particolare essi sono composti da:

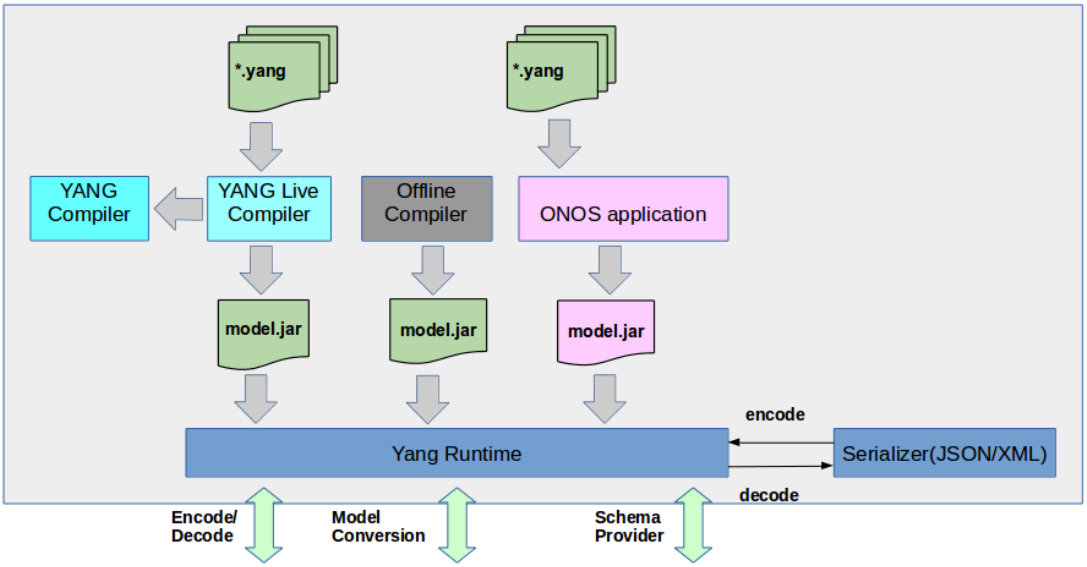

Figura 8.1. Architettura generale di funzionamento di ONOS YANG Tools (Fonte: CORD wiki [9]).

- *DataNode*: fornisce una base per mantenere le informazioni in una struttura ad albero, la quale corrisponde ad un set di modelli YANG di base e di *augmentation*.

- *ResourceId*: entità che identifica una risorsa nella struttura ad albero. Di fatto è una lista di nodi chiave contenente il nome del nodo locale e il corrispondente namespace del modulo base.

- *ModelObject*: fornisce una base comune per tutti gli oggetti JAVA che vengono generati da un modello YANG. Il collegamento tra l'oggetto JAVA e il DataNode viene eseguito da *YANG Runtime*.

8.2 YANG Compiler

YANG Compiler è il componente di *YANG Tools* responsabile di effettuare la traduzione da modelli YANG a classi JAVA (Figura 8.2).

Questo componente non dipende da nessun framework di *building* specifico (e.g.: Buck, Maven, Ant) ed è possibile eseguirlo in una macchina virtuale *standalone*.

YANG Compiler consiste di un *parser* per l'interpretazione del linguaggio YANG e di un generatore di codice JAVA a sua volta composto da *linker* e *translator*.

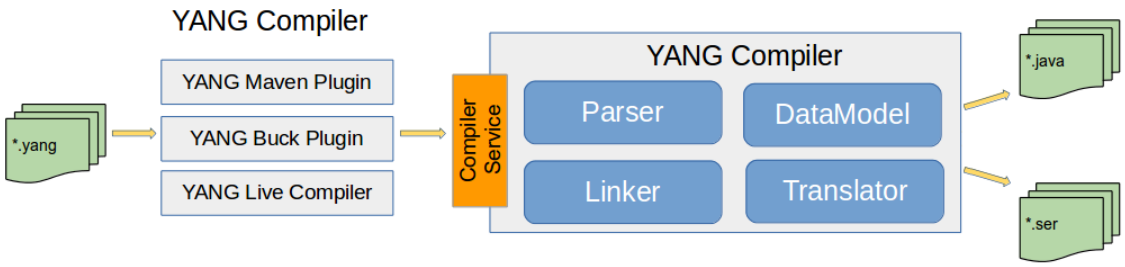

Figura 8.2. Struttura logica del funzionamento di YANG Compiler (Fonte: CORD wiki [9]).

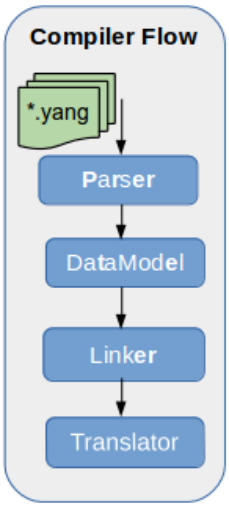

Figura 8.3. Flusso di compilazione di YANG Compiler (Fonte: CORD wiki [9]).

Il flusso di compilazione che un modello YANG deve attraversare è rappresentato in Figura 8.3. In particolare i vari componenti di *YANG Compiler* che si occupano del lavoro di traduzione sono:

- *Parser*: esegue il *parse* dei file YANG in input e produce il *DataModel*. Compila i file YANG basandosi sulla versione 1.0 della grammatica di YANG usando il tool ANTLR[1] e aggiorna l'albero dei *data model*.

[1]ANTLR (ANother Tool for Language Recognition) [21] è un generatore di parser.

- *DataModel*: è la rappresentazione astratta del modello YANG in una struttura JAVA ad albero. Questa viene poi serializzata e usata dallo YANG Runtime per comprendere le informazioni dello *schema*.
- *Linker*: collega le varie dipendenze dei costrutti YANG nei vari file generati (e.g.: dipendenze dovute da import, include, etc.).
- *Translator*: genera il codice JAVA corrispondente al *data model*.

Riassumendo, il processo prende un insieme di file YANG e li compila in un insieme di classi JAVA e li inserisce in un *package* JAR. I file YANG originali vengono inclusi come risorse all'interno del *package*.

8.3 YANG Serializer

YANG Serializer è una entità utilizzata da *YANG Runtime* capace di effettuare la codifica e la decodifica dei *DataNode* (rappresentazioni *in-memory* dei modelli YANG) in e da varie forme esterne di rappresentazione (e.g.: XML, JSON, etc.).

Tale componente è capace di decodificare la rappresentazione esterna (e.g.: JSON, XML) di un modello di configurazione da uno specifico *input stream* in una rappresentazione *in-memory*. Inoltre è in grado di effettuare l'operazione inversa: codificare la rappresentazione interna *in-memory* di un modello di configurazione in una rappresentazione esterna (e.g.: XML, JSON). Il processo di serializzazione viene richiamato da *YANG Runtime* ed è rappresentato in Figura 8.4.

In YANG Tools sono inclusi due *serializer*:

1. *JSON Serializer*: converte in/da formato JSON da/in DataNode.
2. *XML Serializer*: converte in/da formato XML da/in DataNode.

8.4 YANG Runtime

YANG Runtime è il componente che serve come registro dei vari modelli YANG nel sistema e il suo scopo primario è quello di effettuare la serializzazione e la deserializzazione tra vari formati esterni (e.g.: XML, JSON, etc.) e la loro rappresentazione interna in JAVA (basata sui *DataNode*).

Gli utenti e le applicazioni possono registrare nuovi modelli e nuovi *serializer* a *runtime*.

YANG Runtime è indipendente da quaunque protocollo di trasporto (e.g.: RESTCONF, NETCONF) e da qualunque meccanismo di *storage* o di messaggistica. Le implementazioni dei protocolli che si rivolgono all'esterno (e.g.: NETCONF) dipendono da *YANG Runtime* per serializzare e deserializzare i *payload* dei messaggi da

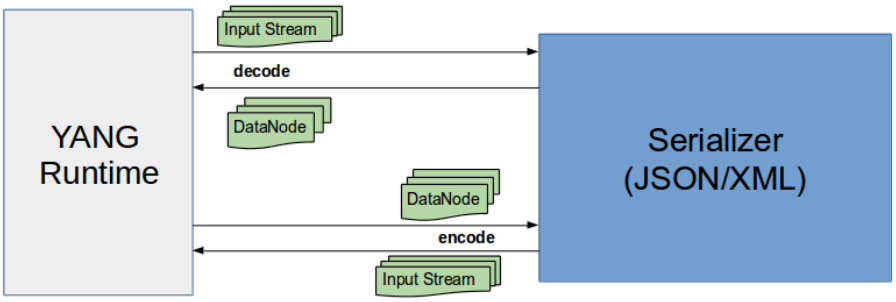

Figura 8.4. Processo di serializzazione tra YANG Runtime e YANG Serializer (Fonte: CORD wiki [9]).

scambiare. Come *YANG Compiler*, anche *YANG Runtime* non dipende da nessun framework di *building* specifico (e.g.: Buck, Maven, Ant) ed è possibile eseguirlo in una macchina virtuale *standalone*.

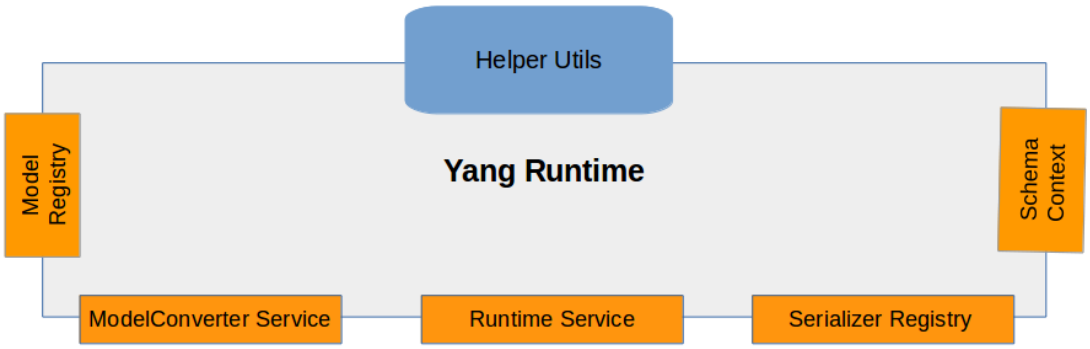

Figura 8.5. Architettura di YANG Runtime (Fonte: CORD wiki [9]).

YANG Runtime si struttura a sua volta in diversi componenti (Figura 8.5):

- *Serializer Registry*: servizio utilizzato per registrare e cancellare i *serializers* disponibili. All'attivazione della applicazione "*yang*" di ONOS vengono registrati di default i *serializer* per i formati XML e JSON. Gli utenti possono registrare *serializer* per nuovi formati di data.

- *Model Registry*: servizio utilizzato per registrare modelli YANG. Quando una applicazione di ONOS vuole fare uso di YANG Tools deve registrare i modelli

YANG che vuole usare nel sistema, in maniera da poter essere poi utilizzati da *YANG Runtime*.

- *Runtime Service*: servizio per la codifica e la decodifica tra rappresentazioni interne e modelli esterni. Contiene i metodi *encode* e *decode* responsabili rispettivamente per le operazioni di codifica e di decodifica.
- *ModelConverter Service*: fornisce un meccanismo per effettuare la conversione tra *DataNode* e oggetti JAVA. Inoltre è capace di includere qualunque possibile *augmentation* presente nei *DataNode* e di creare la struttura ad albero da un dato oggetto JAVA.
- *SchemaContext Provider*: ritorna una entità che fornisce il contesto dello *schema* corrispondente ad una certa risorsa.
- *Helper Utils*: insieme di *utility* utilizzate da *YANG Runtime* e da *YANG Serializer* che permettono di ottenere informazioni sui modelli YANG o di creare *DataNode*.

Capitolo 9

Implementazione di un driver ONOS basato su YANG data model

In questo capitolo viene mostrata l'implementazione di un driver in ONOS nello scenario analizzato in questo lavoro di tesi.

9.1 Studio preliminare

Per comprendere come implementare un driver ONOS basato su modelli YANG e il funzionamento della generazione automatica delle classi Java a partire dai modelli YANG, prima di tutto è stata necessaria una fase di studio preliminare.

In questa fase sono stati analizzati i driver già presenti nel *code base* di ONOS e a seguito di tale analisi si è compreso il procedimento necessario per creare un driver ONOS basato su *data model* YANG e per generare automaticamente le classi Java di tali modelli.

Questo procedimento viene esaminato nel dettaglio nell'Appendice B.

Tra i diversi driver esaminati, particolare attenzione è stata prestata al driver del dispositivo "*Microsemi Edge Assure 1000 SFP-NID*", utilizzato all'interno dell'architettura CORD. Infatti questo driver presenta caratteristiche similari a quelle ricercate e soprattutto anche esso basa il suo funzionamento su *data model* YANG e sull'autogenerazione delle classi Java.

Da esso sono quindi stati tratti gli elementi strutturali che hanno portato alla costruzione del driver sviluppato in questo lavoro di tesi.

In un primo approccio di implementazione è stato creato un driver di esempio chiamato "*examplenetconfdriver*" dove, oltre a implementare le caratteristiche elencate precedentemente riguardo i modelli YANG e le classi Java autogenerate, si è cercato di analizzare il funzionamento di tali classi autogenerate a partire da un

modello YANG del protocollo BGP preso da *OpenConfig* [22]. In particolare si è cercato di sviluppare la parte di codice relativa all'uso di tali classi autogenerate per impostare rotte BGP a un dispositivo generico tramite il protocollo NETCONF.

Ciò ha portato ad una maggiore comprensione della rappresentazione interna del modello e dell'uso delle classi Java autogenerate all'interno del codice.

È possibile consultare il codice del driver di esempio sviluppato all'interno del repository del gruppo di ricerca Netgroup del Politecnico di Torino [23].

Si è quindi poi utilizzato "*examplenetconfdriver*" come base per lo sviluppo del driver trattato in questa tesi e che viene descritto nella sua interezza all'interno di questo capitolo.

9.2 Descrizione generale del driver

Il driver sviluppato in questo lavoro di tesi ha come scopo generale quello di permettere ad un dispositivo di ricevere configurazioni o inviare informazioni sulla configurazione attuale tramite il protocollo NETCONF.

Una particolarità che caratterizza tale driver è quella di basarsi sull'uso di modelli YANG che descrivono il *data model* relativo alla configurazione del dispositivo. Tramite l'uso della suite di *tool* di ONOS denominata *YANG Tools*, questi modelli permettono la generazione automatica di una loro rappresentazione in oggetti JAVA all'interno del codice, la quale a sua volta permette di automatizzare la generazione degli RPC da inviare al dispositivo con il protocollo NETCONF.

Nello scenario di utilizzo presentato nelle sezioni precedenti, il driver sviluppato si deve occupare di pilotare un dispositivo posto agli *edge* della rete controllata da E-CORD, nella parte della rete di accesso. Questo dispositivo è il CPE che collega un *host* aziendale ad un sito CORD.

Per fare ciò è necessario che tale dispositivo sia in grado di assegnare un "*c-tag*" (customer tag), un tag VLAN che definisce all'interno dell'architettura CORD l'identità di un *customer*.

Per realizzare l'assegnazione dei tag VLAN, è necessario impostare le porte collegate agli host in modalità "*access*", in modo da aggiungere il tag VLAN corretto ai pacchetti in ingresso e toglierlo a quelli in uscita, e poi impostare la porta collegata con il sito CORD in modalità "*trunk*", in modo da permettere il transito dei pacchetti con i tag VLAN specificati sia in invio che in ricezione.

Quindi è stato sviluppato un driver che gestisce l'impostazione delle interfacce in modalità "*access*" o "*trunk*" e l'assegnazione di tag VLAN alle interfacce del CPE posto nella rete di accesso.

Il CPE di riferimento utilizzato in questo lavoro di tesi è il router *Tiesse Imola 5262-IKF*.

Il router *Tiesse Imola 5262-IKF* permette di *default* l'uso di una particolare modalità chiamata *LAN Splitting*: quando questa modalità è attiva ogni porta dello

switch del router diventa *routed* e coincide col nome dell'interfaccia (e.g.: la porta numero 2 coincide con l'interfaccia "eth2"). Grazie a questa associazione la configurazione delle porte in modalità *access* o *trunk* è più semplice sia a livello di comandi da CLI sia a livello di codice del driver.

La configurazione dei tag VLAN e le modalità da assegnare alle interfacce viene ricevuta da una applicazione ONOS di supporto, la quale passa le informazioni ricevute al driver, richiamandone i metodi che attuano la configurazione del dispositivo tramite protocollo NETCONF. Tale applicazione viene trattata nel dettaglio nel Capitolo 10.

Il codice del driver sviluppato, che permette di configurare il dispositivo *Tiesse Imola*, può essere consultato all'interno del repository del gruppo di ricerca Netgroup del Politecnico di Torino [24].

9.3 YANG data model

Il dispositivo *Tiesse Imola* è accompagnato da una serie di *data model* scritti in linguaggio YANG che rappresentano i modelli astratti necessari per la configurazione del dispositivo tramite il protocollo NETCONF.

Ad esempio il modello "*tiesse-ethernet.yang*" contiene la descrizione delle interfacce del router e i vari parametri che si possono assegnare ad esse, mentre il modello "*tiesse-vlan.yang*" contiene la descrizione dei parametri per definire le VLAN da associare eventualmente ad una interfaccia.

Tali modelli vengono prima compilati con *Yang Compiler* in modo da generare automaticamente le classi Java che ne rappresentano il contenuto. Tramite le istanze di queste classi Java, il programmatore può gestire il contenuto dei dati da configurare e poi appoggiarsi a *YANG Runtime* e *YANG Serializer* per serializzare in formato XML i dati da inserire nel RPC da inviare al dispositivo (Figura 9.1).

In ONOS, tramite l'uso di *YANG Tools* e della libreria "*netconf*", è possibile automatizzare la creazione degli RPC da inviare al server NETCONF del dispositivo in maniera da separare la gestione della sintassi dei messaggi dall'implementazione nel codice della configurazione da inviare.

In questo modo il programmatore può concentrarsi sugli oggetti Java che rappresentano i dati che andranno a configurare il dispositivo e lasciare a *YANG Runtime* e *YANG Serializer* il compito di creare il messaggio sintatticamente corretto da inviare in formato XML.

9.3.1 Generazione automatica di classi Java da YANG data model

Grazie all'uso di *YANG Compiler* è possibile, partendo dai modelli YANG, generare le corrispondenti classi Java che li rappresentano. Questo permette di lavorare a

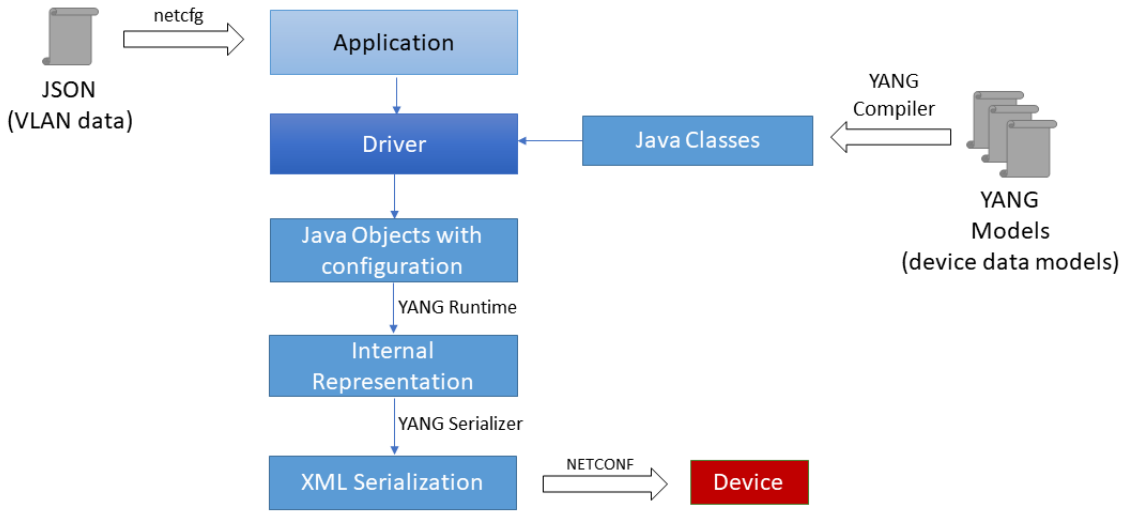

Figura 9.1. Rappresentazione a blocchi del funzionamento del driver basato su modelli YANG.

livello di codice tramite oggetti che conterranno al loro interno i valori necessari per la configurazione, piuttosto che realizzare manualmente i messaggi XML da inviare al server NETCONF per ogni tipo di configurazione.

Per permettere a *YANG Compiler* di effettuare la compilazione dei modelli relativi al driver è necessario inserire all'interno della cartella "*models*" di ONOS, una cartella che possiede il nome stesso del driver. All'interno di essa devono essere posti i modelli YANG contenenti i *data model* necessari per il driver. Per i dettagli sulla procedura necessaria per creare un driver ONOS basato su modelli YANG e generare automaticamente le classi Java si rimanda alla Appendice B.

9.3.2 Serializzazione dei dati in formato XML

YANG Runtime è l'elemento ONOS che si occupa della trasformazione degli oggetti Java ottenuti dai *data model* in rappresentazione interna (*DataNode*) e della loro serializzazione in formato XML.

Gli oggetti Java utilizzati che sono stati ottenuti dai *data model*, vengono prima convertiti *runtime* in *ModelObjectData* e poi trasformati nella rappresentazione interna del modello, il *DataNode*.

In seguito il *DataNode* viene a sua volta serializzato in formato XML, il linguaggio utilizzato nella comunicazione degli RPC nel protocollo NETCONF e salvato in una stringa. Tale stringa viene messa come corpo dell'RPC del protocollo NETCONF (e.g.: un *<edit-config>*), il quale viene poi inviato al dispositivo.

All'interno del driver scritto in questo lavoro di tesi è possibile vedere questo procedimento all'interno del metodo "*encodeMoToXmlStr*" contenuto nel file "*AbstractYangServiceImpl.java*":

```
392     protected final String encodeMoToXmlStr(
           ModelObjectData yangObjectOpParamFilter, List<
           AnnotatedNodeInfo> annotations) throws
           NetconfException {
393
394         //Convert the param to DataNode
395         ResourceData rd = ((ModelConverter)
               yangModelRegistry).createDataNode(
               yangObjectOpParamFilter);
396
397         DefaultCompositeData.Builder cdBuilder =
398                         DefaultCompositeData.builder().
                               resourceData(rd);
399         if (annotations != null) {
400             for (AnnotatedNodeInfo ani : annotations) {
401                 cdBuilder.addAnnotatedNodeInfo(ani);
402             }
403         }
404         //Convert the CompositeData to XML
405         CompositeStream cs = xSer.encode(cdBuilder.build
               (), yCtx);
406
407         try {
408             ByteSource byteSource = new ByteSource() {
409                 @Override
410                 public InputStream openStream() throws
                       IOException {
411                     return cs.resourceData();
412                 }
413             };
414             //Returns the XML data as a string
415             return byteSource.asCharSource(Charsets.UTF_8
                   ).read();
416         } catch (IOException e) {
417             throw new NetconfException("Error decoding
                   CompositeStream to String", e);
418         }
419     }
```

Listing 9.1. Trasformazione degli oggetti Java ottenuti dai modelli YANG in DataNode e serializzazione dei dati in formato XML.

Nel Listato 9.1 è possibile vedere nella riga 417 la trasformazione da oggetto Java a rappresentazione interna del modello in *DataNode*. A questi dati vengono aggiunti eventuali annotazioni se necessario creando un *CompositeData* (righe 419-425) che poi viene codificato in formato XML (riga 426) e ritornato come stringa (riga 437).

9.4 Struttura del driver e delle classi Java

Il codice del driver è strutturato in due sezioni principali: quella relativa ai *behaviour* implementati nel driver che fanno uso delle rappresentazioni Java dei modelli YANG e quella relativa alla comunicazione con il dispositivo tramite il protocollo NETCONF (Figura 9.2).

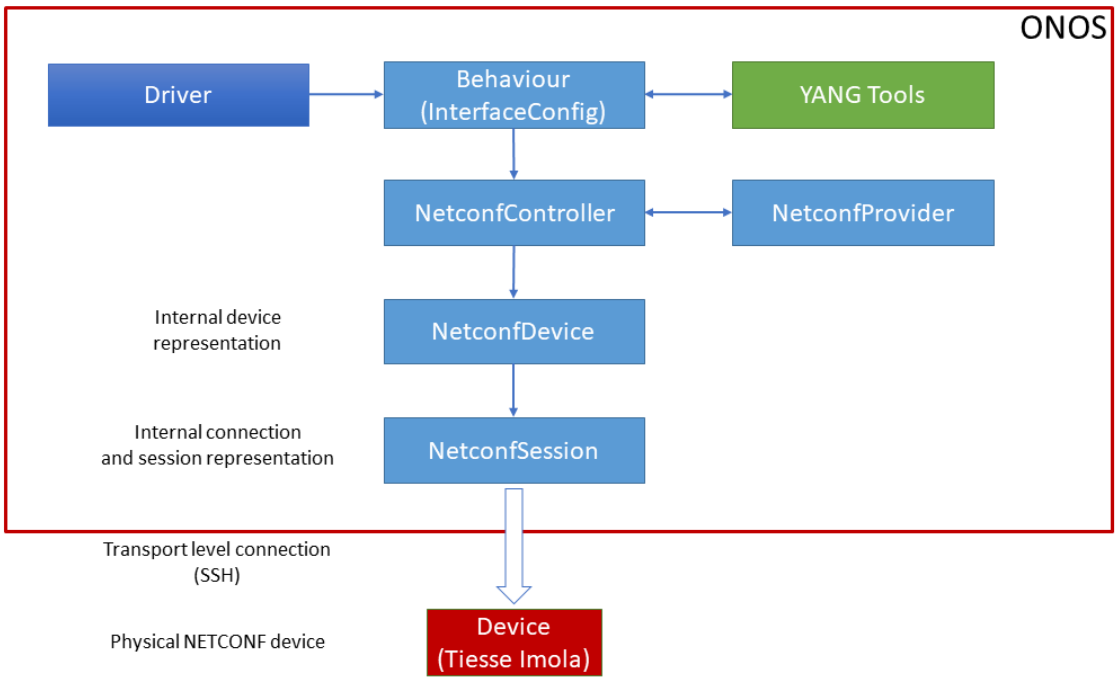

Figura 9.2. Rappresentazione delle relazioni tra Behaviour, YANG Tools e protocollo NETCONF in ONOS.

Nella prima sezione vengono usate le classi Java autogenerate da *YANG Compiler* per creare degli oggetti di configurazione mentre nella seconda sezione vengono utilizzati *YANG Runtime* e *YANG Serializer* per elaborare e codificare i dati in formato XML da inviare al dispositivo. Per inviare gli RPC contenenti i dati codificati

in XML tramite il protocollo NETCONF, si è utilizzata la libreria "*netconf*" già inclusa nel *code base* di ONOS.

I *behaviour* sono delle interfacce che descrivono i servizi che è possibile ottenere da un *device*. In genere i *behaviour* di default per essere utilizzati su uno specifico dispositivo hanno necessità di essere implementati appositamente per comunicare con esso.

Ad esempio il *behaviour* "*DeviceDescriptionDiscovery*", che permette a ONOS di conoscere i dettagli generali (e.g.: serial number, produttore, etc.) o la descrizione delle porte di un *device*, ha una implementazione che varia in base al dispositivo con cui deve comunicare per ottenere certe informazioni.

Inoltre è necessario estendere queste interfacce per aggiungere nuovi metodi che sono necessari per implementare funzionalità non previste di base. Questo è il caso del *behaviour* "*InterfaceConfig*' che descrive i mezzi per configurare le interfacce su un dispositivo. Esso infatti, nello sviluppo di questo driver, è stato esteso per aggiungere la funzionalità di assegnare VLAN e parametri (e.g.: indirizzo ip, netmask, etc.) alle interfacce del *Tiesse Imola*.

I *behaviour* implementati nel driver di cui si vuole far conoscere la presenza al *Driver Subsystem* di ONOS devono essere contenuti in un file in formato XML all'interno della cartella "*resources*". Nel Listato 9.2 sono rappresentati i *behaviour* esportati dal driver scritto in questo lavoro di tesi.

```
17 <drivers>
18     <driver name="examplenetconfdriver-netconf" extends="
          netconf" manufacturer="Tiesse" hwVersion="Imola">
19         <behaviour api="org.onosproject.net.device.
              DeviceDescriptionDiscovery"
20             impl="org.onosproject.drivers.
                  examplenetconfdriver.
                  MyDeviceDeviceDescription" />
21         <behaviour api="org.onosproject.net.behaviour.
              ConfigGetter"
22             impl="org.onosproject.drivers.
                  examplenetconfdriver.NetconfConfigGetter"
                  />
23         <behaviour api="org.onosproject.net.behaviour.
              Pipeliner"
24             impl="org.onosproject.drivers.
                  examplenetconfdriver.MyDevicePipeliner" />
25         <behaviour api="org.onosproject.net.flow.
              FlowRuleProgrammable"
```

```
26              impl="org.onosproject.drivers.
                   examplenetconfdriver.
                   MyDeviceFlowRuleProgrammable" />
27          <behaviour api="org.onosproject.net.behaviour.
               MeterQuery"
28              impl="org.onosproject.drivers.
                   examplenetconfdriver.FullMetersAvailable"
                   />
29          <behaviour api="org.onosproject.drivers.
               examplenetconfdriver.InterfaceConfigTiesse"
30              impl="org.onosproject.drivers.
                   examplenetconfdriver.
                   InterfaceConfigTiesseImpl" />
31      </driver>
32  </drivers>
```

Listing 9.2. Behaviour esportati dal driver per il dispositivo Tiesse Imola.

Poiché il driver sviluppato per il dispositivo *Tiesse Imola* è basato a sua volta sul driver di esempio "*examplenetconfdriver*" sviluppato in fase di studio preliminare, sono presenti in realtà alcuni *behaviour* che però non sono stati implementati in questo specifico caso d'uso, in quanto non necessari. Tali *behaviour* sono però stati lasciati disponibili nell'eventualità in cui sia necessario il loro uso in un futuro sviluppo del driver.

Le classi Java del driver sviluppato che compongono la prima sezione strutturale riguardante i *behaviour* sono:

- **FullMetersAvailable.java**: classe che implementa il *behaviour* "*MeterQuery*" e che definisce il massimo *meter* disponibile per il *device*. Questa classe non è utilizzata in questo caso d'uso.

- **InterfaceConfigTiesse.java**: interfaccia che estende il *behaviour* "*InterfaceConfig*" che permette la configurazione delle interfacce. In particolare aggiunge un metodo che permette l'assegnazione di VLAN e parametri (e.g.: indirizzo ip, netmask, etc.) alle interfacce.

- **InterfaceConfigTiesseImpl.java**: classe che implementa il *behaviour* "*InterfaceConfigTiesse*" e che fornisce i metodi per configurare le interfacce sul dispositivo *Tiesse Imola*. In particolare permette di definire le interfacce in modalità *access* o *trunk* e di assegnare VLAN e parametri (e.g.: indirizzo ip, netmask, etc.) alle interfacce.

- **MyDeviceDeviceDescription.java**: classe che implementa il *behaviour* "*DeviceDescriptionDiscovery*" e che è responsabile della creazione delle descrizioni del dispositivo e delle sue porte.

- **MyDeviceDriversLoader.java**: classe che estende la classe astratta "*AbstractDriverLoader*" e che ha lo scopo di registrare dentro ONOS i *behaviour* che il driver supporta passando alla classe estesa il file in formato XML (mostrato nel Listato 9.2) contenente queste informazioni.

- **MyDeviceFlowRuleProgrammable.java**: classe che implementa il behaviour "*FlowRuleProgrammable*" e che gestisce l'insieme delle regole di flusso da impostare sul dispositivo. Questa classe non è utilizzata in questo caso d'uso.

- **MyDeviceMeterProvider.java**: classe che implementa il behaviour "*MeterProvider*" e che usa un controller NETCONF per gestire i *meter*. Questa classe non è utilizzata in questo caso d'uso.

- **MyDevicePipeliner.java**: classe che implementa il *behaviour* "*Pipeliner*" e definisce i metodi per supportare l'uso dei *FlowObjective* nel dispositivo. Questa classe non è utilizzata in questo caso d'uso.

- **NetconfConfigGetter.java**: classe che implementa il *behaviour* "*ConfigGetter*" e che permette di ottenere dal dispositivo la configurazione dei *datastore* (e.g.: *running*, *candidate*, etc.) NETCONF del dispositivo.

Le classi Java del driver sviluppato che compongono la seconda sezione strutturale riguardante la comunicazione con il dispositivo tramite il protocollo NETCONF sono:

- **InterfaceConfigTiesseNetconfService.java**: interfaccia che definisce i metodi per gestire le sessioni NETCONF e per configurare il dispositivo *Tiesse Imola*.

- **MyNetconfDriverModelRegistrator.java**: classe che estende la classe astratta "*AbstractYangModelRegistrator*". Essa si occupa di registrare i modelli YANG, che rappresentano i *data model* necessari per configurare il dispositivo *Tiesse Imola*, nel *Model Registry* contenuto in ONOS e usato da *YANG Runtime*.

- **InterfaceConfigTiesseManager.java**: classe che implementa i metodi descritti dall'interfaccia "*InterfaceConfigTiesseNetconfService.java*". Tali metodi sono dei metodi *wrapper* e hanno lo scopo di trasformare gli oggetti derivati dalle classi Java autogenerate dai modelli YANG in "*ModelObjectData*". Tale

rappresentazione dei dati viene a sua volta inviata ai metodi della classe "*AbstractYangServiceImpl.java*' che si occupa di elaborarla e di implementare la effettiva comunicazione NETCONF col dispositivo.

- **AbstractYangServiceImpl.java**: classe astratta che implementa i metodi che vanno a configurare il dispositivo nella sessione NETCONF specificata. In particolare si occupano di serializzare i dati contenuti nel *ModelObjectData* in formato XML e di includerlo nel RPC dell'*<edit-config>* che viene inviato al dispositivo tramite il protocollo NETCONF per impostare la configurazione. Per implementare l'*<edit-config>* viene utilizzata la libreria "*netconf*" di ONOS.

9.5 Integrazione del driver in ONOS

Per poter integrare il driver sviluppato all'interno del *code base* di ONOS è stato necessario seguire un preciso procedimento. Questa sezione spiega i passi che portano sia alla generazione automatica delle classi Java a partire dai modelli YANG del *Tiesse Imola*, sia alla integrazione del driver in ONOS.

Tale procedimento consiste in:

1. Aggiungere una nuova cartella che possiede il nome del driver (in questo caso "*examplenetconfdriver"*) in "*onos/models*". Creare all'interno della cartella appena aggiunta una serie di sottocartelle: "*src/main/yang*". All'interno dell'ultima sottocartella "*yang*" devono essere inseriti i file YANG contenenti i *data model* del dispositivo *Tiesse Imola* necessari per il driver, facendo attenzione a inserire anche eventuali file YANG richiesti al loro interno nel campo *import*.

2. Inserire in "*onos/models/examplenetconfdriver*" un file BUCK col seguente contenuto:

```
1    yang_model(
2    app_name =
        'org.onosproject.models.examplenetconfdriver',
3    title = 'examplenetconfdriver YANG Model',
4    )
```

3. Aggiungere una nuova cartella che possiede il nome del driver (in questo caso "*examplenetconfdriver"*) in "*onos/drivers*". Creare all'interno della cartella appena aggiunta la serie di sottocartelle "*src/main/java/org/onosproject/examplenetconfdriver*", dove "*org/onosproject/examplenetconfdriver*" sarebbe

il package relativo alle classi Java esaminate nella sezione precedente. L'informazione relativa al package è contenuta nel file "*package-info.java*", il quale contiene semplicemente:

```
1    package
        org.onosproject.driver.examplenetconfdriver
```

Tutte le classi Java che compongono il driver (esaminate nella sezione precedente) devono essere inserite all'interno della cartella "*src/main/java/org/onosproject/examplenetconfdriver*".

4. Inserire in "*onos/drivers/examplenetconfdriver*" un file BUCK che specifichi le dipendenze di compilazione, di test, con le altre applicazioni e i dettagli del driver. Il file BUCK del driver sviluppato in questo lavoro di tesi contiene:

```
1  COMPILE_DEPS = [
2  '//lib:CORE_DEPS',
3  '//lib:ONOS_YANG',
4  '//drivers/utilities:onos-drivers-utilities',
5  '//protocols/netconf/api:onos-protocols-netconf-api',
6  '//protocols/netconf/ctl:onos-protocols-netconf-ctl',
7  '//models/examplenetconfdriver:onos-models-
     examplenetconfdriver',
8  '//lib:org.apache.karaf.shell.console',
9  '//incubator/api:onos-incubator-api',
10 ] + YANG_TOOLS
11
12 TEST_DEPS = [
13 '//lib:TEST_ADAPTERS',
14 '//core/api:onos-api-tests',
15 '//drivers/netconf:onos-drivers-netconf-tests',
16 '//utils/osgi:onlab-osgi-tests',
17 '//incubator/net:onos-incubator-net'
18 ]
19
20 APPS = [
21 'org.onosproject.yang',
22 'org.onosproject.config',
23 'org.onosproject.netconf',
24 'org.onosproject.netconfsb',
25 'org.onosproject.drivers.netconf',
26 'org.onosproject.models.examplenetconfdriver'
```

```
27 ]
28
29 osgi_jar_with_tests (
30 deps = COMPILE_DEPS,
31 test_deps = TEST_DEPS,
32 resources_root = 'src/main/resources',
33 resources = glob(['src/main/resources/**']),
34 )
35
36 onos_app (
37 app_name = 'org.onosproject.drivers.
     examplenetconfdriver',
38 title = 'examplenetconfdriver device drivers',
39 category = 'Drivers',
40 url = 'http://onosproject.org',
41 description = 'ONOS examplenetconfdriver device
     drivers application.',
42 required_apps = APPS,
43 )
```

I campi “COMPILE_DEPS” e in “TEST_DEPS” contengono le dipendenze necessarie per la compilazione ed eventuali test. Le applicazioni necessarie invece sono elencate sotto il campo “APPS”.

5. Aggiungere le *entry* del driver e dei modelli YANG nel file “*onos/modules.def*”. Per scrivere correttamente le *entry* è necessario sostituire i "/" contenuti nel path del file con "-" e aggiungere "-oar" come suffisso. (e.g.: “*onos/drivers/examplenetconfdriver*” diventa “*onos-drivers-examplenetconfdriver-oar*”).

   In questo caso è necessario aggiungere a “*onos/modules.def*”:

```
1     ONOS_DRIVERS = [
2     ...
3
4     '//drivers/examplenetconfdriver:onos-drivers-
        examplenetconfdriver-oar'
5
6     ...
7     ]
8
9     MODELS = [
10    ...
```

```
11
12      '//models/examplenetconfdriver:onos-models-
            examplenetconfdriver-oar'
13
14      ...
15      ]
```

Dopo avere effettuato questa procedura è possibile eseguire la build di ONOS tramite BUCK. Infine una volta avviato ONOS, per attivare il driver è necessario scrivere nella CLI di ONOS il comando:

```
1 $  app activate
    org.onosproject.drivers.examplenetconfdriver
```

9.6 Funzionamento

Una funzione fondamentale del driver è quella di fornire a ONOS i dettagli riguardanti il dispositivo che lo utilizza, in questo caso il *Tiesse Imola*. Per connettere tale dispositivo a ONOS e farlo apparire all'interno della sua topologia di rete, è necessario iniettare una serie di informazioni tramite un file di configurazione in formato JSON. Il contenuto del file JSON di configurazione utilizzato per connettere il dispositivo *Tiesse Imola* è mostrato nel Listato 9.3

```
1  {
2    "devices": {
3      "netconf:10.10.77.29:831": {
4        "netconf": {
5          "ip": "10.10.77.29",
6          "port": 831,
7          "username": "root",
8          "password": "tiesseadm"
9        },
10       "basic": {
11         "driver": "examplenetconfdriver-netconf"
12       }
13     }
14   }
15 }
```

Listing 9.3. File in formato JSON per la connessione del dispositivo Tiesse Imola.

Il servizio ONOS che si occupa di ricevere tali informazioni e di elaborarle è denominato "*Network Configuration*". In particolare tale file è necessario per permettere a ONOS di conoscere la capacità del dispositivo di utilizzare il protocollo NETCONF, utilizzato in questo lavoro di tesi per configurare il dispositivo *Tiesse Imola*.

I *behaviour* implementati nel driver sono caricati e registrati dentro ONOS in fase di attivazione del driver stesso.

Pertanto nel momento in cui ONOS, tramite il file JSON, viene a conoscenza della presenza del dispositivo e del driver che lo pilota, è in grado di sapere la lista di tutti i *behaviour* che il driver implementa.

Dunque una volta iniettata la configurazione che definisce il dispositivo *Tiesse Imola*, ONOS recupera le informazioni che lo descrivono richiamando i metodi del *behaviour* "*DeviceDescriptionDiscovery*". Nel driver sviluppato, tale *behaviour* viene implementato nella classe "*MyDeviceDeviceDescription.java*". In particolare il metodo "*discoverDeviceDetails()*" fornisce le caratteristiche hardware e software del *Tiesse Imola* (e.g.: serial number, versione software, etc.). Mentre il metodo "*discoverPortDetails()*" fornisce all'interno di una lista le descrizioni delle porte contenute nel *Tiesse Imola*.

Un'altra funzione fondamentale all'interno del funzionamento del driver è quella di registrare i modelli YANG utilizzati all'interno del *Model Registry* di *YANG Runtime*. Ciò avviene durante la fase di attivazione del driver, in cui vengono registrati i modelli YANG necessari per la configurazione del dispositivo. La classe "*MyNetconfDriverModelRegistrator.java*" si occupa della registrazione di tali modelli, specificando uno per uno i modelli YANG relativi al driver che devono essere registrati nel *Model Registry* di *YANG Runtime*. In Figura 9.3 vengono mostrati i componenti ONOS con cui il driver interagisce.

Successivamente il driver non esegue nessun'altra azione se non viene richiamato dall'applicazione "*tiessemanager*" descritta nel Capitolo 10. Infatti tale applicazione, ricevuta una configurazione delle interfacce del *Tiesse Imola* tramite un file JSON, richiama i metodi contenuti nel driver nella classe "*InterfaceConfigTiesseImpl.java*". In tale classe viene implementato il *behaviour* "*InterfaceConfigTiesse*" responsabile della configurazione delle interfacce del dispositivo.

In particolare i metodi utilizzati per configurare le interfacce del *Tiesse Imola* sono: "*addAccessMode()*" e "*addVlanAndIpAddrAndNetmaskToInterface()*".

Entrambi i metodi lavorano su oggetti creati a partire dalle classi Java autogenerate con *YANG Compiler*, le quali rappresentano i modelli YANG del dispositivo. In questo modo usando la rappresentazione Java dei modelli "*tiesse-ethernet.yang*" e "*tiesse-vlan.yang*" è possibile impostare ad una interfaccia la modalità *access* oppure assegnare alle interfacce una VLAN e definire i parametri che la caratterizzano.

Nel dispositivo *Tiesse Imola* una interfaccia è posta in modalità *trunk* indirettamente: se viene aggiunta più di una VLAN ad una stessa interfaccia essa

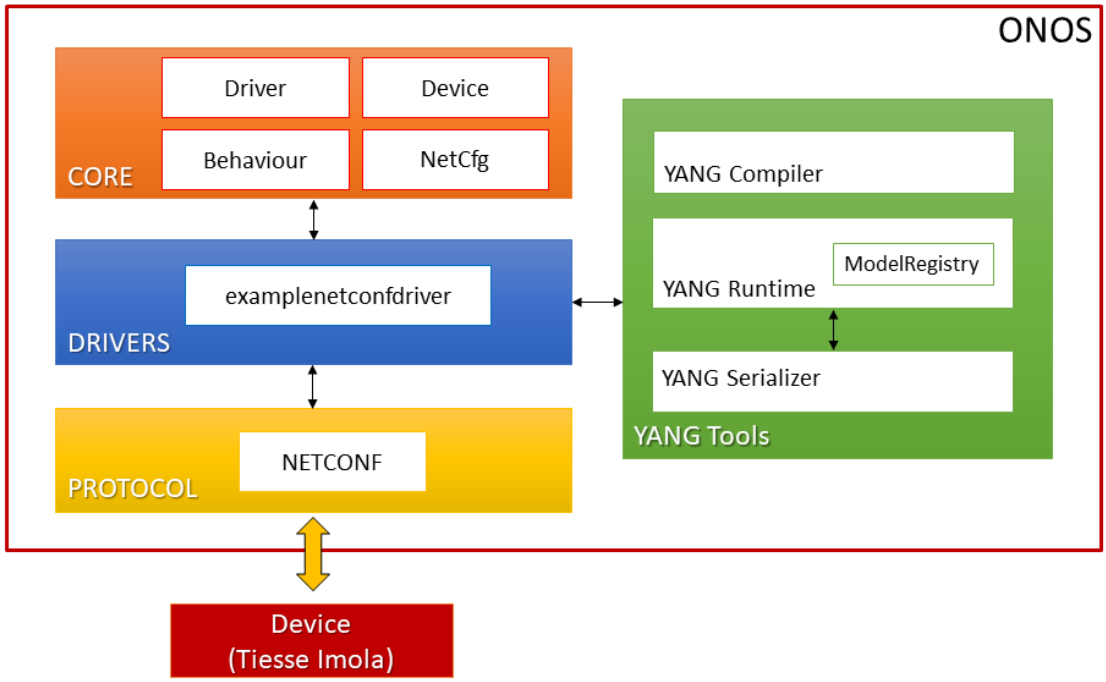

Figura 9.3. Rappresentazione a blocchi dei componenti ONOS con cui il driver interagisce.

automaticamente viene considerata come *trunk*.

Nel Listato 9.4 è mostrata una parte del codice del metodo "*addAccessMode()*", in cui usando le classi autogenerate dai modelli YANG viene creato l'oggetto contenente la configurazione della interfaccia *eth0* e viene richiamato il metodo "*setTiesseEthernet()*" per andare ad applicarla al dispositivo tramite il protocollo NETCONF.

```
0 public boolean addAccessMode(String intf, VlanId vlanId)
    {
1         NetconfController controller = checkNotNull(
              handler().get(NetconfController.class));
2
3         NetconfSession session = controller.getDevicesMap
              ().get(handler().data().deviceId()).getSession
              ();
4
```

```
5           InterfaceConfigTiesseNetconfService
               interfaceConfigTiesseNetconfService = (
               InterfaceConfigTiesseNetconfService)
               checkNotNull(handler().get(
               InterfaceConfigTiesseNetconfService.class));
6
7           TiesseEthernetOpParam tiesseEthernet = new
               TiesseEthernetOpParam();
8
9           if(intf.equals("eth0")) {
10              Eth0 eth0 = new DefaultEth0();
11              AccessVlan accessVlan = new DefaultAccessVlan
                   ();
12              String vlanIdString = vlanId.toString();
13              Long vidLong = Long.parseLong(vlanIdString);
14              accessVlan.vid(vidLong);
15
16              eth0.accessVlan(accessVlan);
17              eth0.active(Onoff.fromString("on"));
18              tiesseEthernet.eth0(eth0);
19          }
20
21          ...
22
23          boolean reply;
24          try {
25              reply = interfaceConfigTiesseNetconfService.
                   setTiesseEthernet(tiesseEthernet, session,
                    DatastoreId.RUNNING, intf);
26          } catch (NetconfException e) {
27              log.error("Failed to configure VLAN ID {} on
                   device {} intf {}.",
28                      vlanId, handler().data().deviceId()
                           , intf, e);
29              return false;
30          }
31
32          return reply;
33      }
```

Listing 9.4. Creazione dell'oggetto tiesseEthernet per configurare una interfaccia in modalità access e invocazione del metodo setTiesseEthernet() che invia la configurazione al dispositivo.

Nel Listato 9.5 è mostrata una parte del codice del metodo "*addVlanAndIpAddrAndNetmaskToInterface()*", in cui usando le classi autogenerate dai modelli YANG viene creato l'oggetto contenente la configurazione della VLAN da assegnare alla interfaccia specificata e dei suoi parametri (e.g.: indirizzo ip, netmask, broadcast). Inoltre viene richiamato il metodo "*setTiesseVlan()*" per andare ad applicare tale configurazione al dispositivo tramite il protocollo NETCONF.

```
37      public boolean addVlanAndIpAddrAndNetmaskToInterface(
           String intf, VlanId vlanId, String ipAddress,
           String netmask, String broadcast) {
38
39          NetconfController controller = checkNotNull(
               handler()
40                  .get(NetconfController.class));
41
42          NetconfSession session = controller.getDevicesMap
               ().get(handler()
43                  .data().deviceId()).getSession();
44
45          InterfaceConfigTiesseNetconfService
               interfaceConfigTiesseNetconfService =
46                  (InterfaceConfigTiesseNetconfService)
                       checkNotNull(handler().get(
                       InterfaceConfigTiesseNetconfService.
                       class));
47
48          TiesseVlanOpParam tiesseVlan = new
               TiesseVlanOpParam();
49          Vlan vlan = new DefaultVlan();
50          Vlans vlans = new DefaultVlans(); //list of vlans
51
52          String vlanIdString = vlanId.toString();
53          Long vidLong = Long.parseLong(vlanIdString);
54          vlans.vid(vidLong);
55          vlans.protocol("802.1q");
56          vlans.yangAutoPrefixInterface(TsInterfaces.
               fromString(intf));
```

```
57
58          Ipv4Address ipAddr = Ipv4Address.fromString(
               ipAddress);
59          Netmask netmaskVar = Netmask.fromString(netmask);
60          Ipv4Address ipBroadcast = Ipv4Address.fromString(
               broadcast);
61          vlans.ipaddr(ipAddr);
62          vlans.netmask(netmaskVar);
63          vlans.broadcast(ipBroadcast);
64
65          vlans.active(Onoff.fromString("on"));
66
67          vlan.addToVlans(vlans);
68          tiesseVlan.vlan(vlan);
69
70          boolean reply;
71          try {
72              reply = interfaceConfigTiesseNetconfService.
                   setTiesseVlan(tiesseVlan, session,
                   DatastoreId.RUNNING);
73          } catch (NetconfException e) {
74              log.error("Failed to configure VLAN ID {} on
                   device {} intf {}.",
75                        vlanId, handler().data().deviceId(),
                           intf, e);
76              return false;
77          }
78          return reply;
79      }
```

Listing 9.5. Creazione dell'oggetto tiesseVlan per assegnare una VLAN ad una interfaccia e invocazione del metodo setTiesseVlan() che invia la configurazione al dispositivo.

I metodi "*setTiesseEthernet()*" e "*setTiesseVlan()*" della classe "*InterfaceConfigTiesseManager.java*", sono dei metodi *wrapper*, responsabili di trasformare rispettivamente gli oggetti *tiesseEthernet* e *tiesseVlan* in *ModelObjectData*, che richiamano a loro volta i metodi "*setNetconfObjectTiesseEthernet()*" (Listato 9.6) e "*setNetconfObjectTiesseVlan()*" (Listato 9.7) contenuti nella classe "*AbstractYangServiceImpl.java*". Sono questi ultimi due metodi che si occupano di serializzare la configurazione in formato XML e di inviare un RPC di tipo <edit-config> al dispositivo tramite una sessione NETCONF.

```
354     protected final boolean
           setNetconfObjectTiesseEthernet(
355              ModelObjectData moConfig, NetconfSession
                    session, DatastoreId targetDs,
356              List<AnnotatedNodeInfo> annotations) throws
                    NetconfException {
357
358          ...
359
360          String xmlQueryStr = encodeMoToXmlStr(moConfig,
                annotations);
361
362          String xmlQueryStrWithPrefix =
                tiesseEthernetRpcAddPrefix(xmlQueryStr);
363
364          return session.editConfig(targetDs, null,
                xmlQueryStrWithPrefix);
365     }
```

Listing 9.6. Serializzazione in formato XML della configurazione del ModelObjectData che rappresenta l'oggetto tiesseEthernet e invio al dispositivo del RPC <edit-config> che la contiene.

```
379     protected final boolean setNetconfObjectTiesseVlan(
380              ModelObjectData moConfig, NetconfSession
                    session, DatastoreId targetDs,
381              List<AnnotatedNodeInfo> annotations) throws
                    NetconfException {
382
383          ...
384
385          String xmlQueryStr = encodeMoToXmlStr(moConfig,
                annotations);
386
387          String xmlQueryStrWithPrefix =
                tiesseVlanRpcAddPrefix(xmlQueryStr);
388
389          return session.editConfig(targetDs, null,
                xmlQueryStrWithPrefix);
390     }
```

Listing 9.7. Serializzazione in formato XML della configurazione del ModelObjectData che rappresenta l'oggetto tiesseVlan e invio al dispositivo dell'RPC <edit-config> che la contiene.

9.7 Considerazioni generali

Il driver è stato pensato per mettere a disposizione i metodi e le interfacce necessarie per impostare le VLAN sulle interfacce di un dispositivo a partire da modelli YANG, basandosi sul protocollo NETCONF per la parte di configurazione. Per questo motivo esso non deve salvare al suo interno un particolare stato del dispositivo o una particolare configurazione, ma deve essere il mezzo tramite il quale le configurazioni vengono attuate.

Dunque è stato necessario disaccoppiare il codice relativo alla definizione dei metodi che permettono di attuare la configurazione del dispositivo, dal codice relativo alla ricezione ed elaborazione dei dati di configurazione.

Pertanto si è proceduto allo sviluppo di una applicazione di supporto in grado di ricevere, tramite il servizio di *Network Configuration* di ONOS, un file in formato JSON contenente la configurazione delle VLAN. Tale applicazione viene trattata nel dettaglio nel Capitolo 10.

I modelli YANG utilizzati all'interno del driver per la autogenerazione delle classi Java, hanno causato dei problemi durante la loro fase di compilazione con *YANG Compiler*. Infatti i modelli utilizzati in questo lavoro di tesi non sono stati pensati originariamente per essere tradotti in classi Java da ONOS e inoltre sono stati sviluppati per seguire le regole di sintassi e semantica della versione 1.1 di YANG.

YANG Compiler invece è in grado di interpretare e tradurre solamente i modelli della versione 1.0 di YANG. Per questo motivo si è dovuto modificare il campo "*version*" all'interno dei modelli YANG da 1.1 a 1.0 e verificare che i modelli venissero poi tradotti in maniera corretta. Inoltre alcuni dei campi all'interno dei modelli contenevano delle definizioni che *YANG Compiler* non era in grado di interpretare sintatticamente e quindi rendevano impossibile generare una loro traduzione corrispondente in Java. I simboli "+" e "-" sono un esempio di questo problema in quanto non vengono attualmente tradotti da *YANG Compiler*. Per risolvere gli errori di compilazione che ciò comportava si è dovuto procedere a sostituirli con caratteri. Ad esempio i campi che definivano l'orario di sistema del dispositivo all'interno di un modello YANG, contenevano elementi del tipo "GMT+1" o "GMT-2" che sono stati cambiati in "GMTp1" o "GMTm2" per permettere la loro interpretazione.

Quindi *YANG Compiler* presenta dei limiti nella traduzione dei modelli YANG e per questo è stato necessario effettuare delle piccole modifiche per permettere la loro corretta compilazione e la successiva generazione delle classi Java corrispondenti.

Infine un altro problema riscontrato è stato quello della sintassi richiesta dal server NETCONF installato sul dispositivo *Tiesse Imola* per i messaggi XML degli RPC. Infatti il server si aspetta nei campi di configurazione contenuti negli RPC un prefisso davanti ad ognuno dei tag XML e inoltre esso varia a seconda del tipo di configurazione che si vuole effettuare.

Questi particolari prefissi non sono deducibili dai modelli YANG, perciò il messaggio serializzato in maniera automatica da *YANG Serializer* non include nessuno di essi. Nel caso in cui venga inviato tale messaggio XML privo di prefissi, il server NETCONF del dispositivo restituisce un RPC reply contenente l'errore di tipo "*unknown element*", in quanto non in grado di interpretare la richiesta in maniera corretta.

Pertanto si è dovuto procedere alla creazione di un algoritmo di *matching* basato su espressioni regolari in cui, in caso di assenza del prefisso richiesto dal server nei tag XML, viene modificato il contenuto di ogni messaggio inviato in maniera tale da aggiungere il prefisso dove necessario.

Questa problematica mostra più in generale come, benché ONOS produca dei messaggi XML che rispettano gli standard YANG definiti nello RFC [6], è possibile che invece i produttori dei dispositivi si discostino parzialmente da tali specifiche. Dunque in questi casi è necessario sviluppare soluzioni apposite (e.g.: algoritmo per aggiungere il prefisso ai tag XML), per permettere il corretto funzionamento del driver ONOS e rispettare così eventuali specifiche proprietarie a cui fa riferimento il dispositivo.

Inoltre in caso di variazioni all'interno dei modelli YANG dovuti all'aggiornamento delle funzionalità messe a disposizione dal dispositivo, è necessario rettificare il codice per renderlo coerente con le nuove funzionalità immesse nei *data model*. Infatti una modifica all'interno dei modelli implica un cambiamento nelle classi Java autogenerate in ONOS. Ciò comporta eventuali aggiornamenti o modifiche della struttura del codice a seconda della portata del cambiamento del modello.

Tuttavia le modifiche da apportare in ONOS al codice di un driver *model-driven* come quello sviluppato in questo lavoro di tesi, si rivelano essere comunque minori rispetto a quelle che sarebbero necessarie nel caso di un driver *API-driven* in cui tutte le funzionalità devono essere definite manualmente, inclusi i messaggi XML da inviare al dispositivo.

L'uso di un driver *model-driven*, basato quindi sull'uso di *data model* YANG, toglie però generalità alle API di northbound rispetto a quella che fornirebbe un driver *API-driven*. Infatti le applicazioni che interagiscono con i driver basati su modelli, come ad esempio quella sviluppata in questo lavoro di tesi (i.e.: *tiessemanager*), non sono più generali poiché devono conoscere a priori le funzionalità specifiche del driver a cui fare riferimento.

Capitolo 10

Implementazione di una applicazione ONOS

In questo capitolo viene mostrata l'implementazione di una applicazione in ONOS a supporto del driver sviluppato nel Capitolo 9.

10.1 Descrizione generale dell'applicazione

L'applicazione sviluppata in questo lavoro di tesi ha come scopo generale quello di ricevere la configurazione delle VLAN da applicare al dispositivo e, dopo averne effettuato il *parsing*, richiamare i metodi del driver che vanno ad attuare tale configurazione sul dispositivo.

Essa è pensata per funzionare insieme al driver sviluppato nel Capitolo 9 e per implementare la logica che gestisce i dati della configurazione del dispositivo.

La configurazione delle VLAN è contenuta in un file JSON che viene iniettato in ONOS tramite l'uso del servizio di "*Network Configuration*". Essa viene successivamente analizzata e salvata in due opportune strutture dati.

La prima è denominata "*AccessData*" e contiene le informazioni relative ad una interfaccia da impostare in modalità *access*. La seconda è denominata "*TrunkData*" contiene le informazioni relative alle interfacce da impostare in modalità *trunk*.

Inoltre la applicazione è pensata per configurare più di un dispositivo con un singolo file JSON. In questo modo si va a ridurre notevolmente i tempi di attuazione di una configurazione su uno scenario in cui nel lato della rete di accesso si hanno più dispositivi da configurare con uno stesso controllore ONOS.

In Figura 10.1 è mostrata una rappresentazione ad alto livello del funzionamento dell'applicazione all'interno dello scenario presentato nelle sezioni precedenti.

Il codice della applicazione sviluppata può essere consultato all'interno del repository del gruppo di ricerca Netgroup del Politecnico di Torino [24].

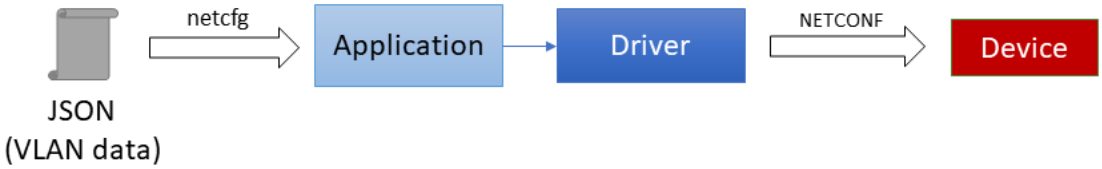

Figura 10.1. Rappresentazione a blocchi del funzionamento dell'applicazione visto ad alto livello.

10.2 Uso del servizio di network configuration

Per fornire all'applicazione la configurazione delle VLAN da elaborare e far attuare sul dispositivo, viene utilizzato il servizio di "*Network Configuration*" di ONOS.

Durante l'attivazione dell'applicazione infatti viene aggiunto un *listener* al servizio di configurazione, in maniera da restare in ascolto all'evento di *push* di un file JSON contenente la configurazione delle VLAN.

In tale file JSON sono contenute le impostazioni da attuare sul dispositivo, sia per le interfacce da impostare in modalità *access* sia per quelle in modalità *trunk*. Inoltre sono contenuti i parametri quali indirizzo ip, netmask e broadcast da impostare alla sotto-interfaccia relativa alla VLAN da assegnare.

Nel Listato 10.1 viene mostrato un file JSON di esempio contenente la configurazione delle VLAN da assegnare alle interfacce del dispositivo *Tiesse Imola*.

In particolare le interfacce "*eth2*" e "*eth4*" vengono impostate in modalità "*access*", assegnandogli rispettivamente i tag VLAN 222 e 444. In questo modo i pacchetti in ingresso a quelle interfacce vengano taggati col rispettivo tag, mentre quelli in uscita da esse vengono privati di tali tag.

L'interfaccia "*eth3*" viene invece impostata in modalità "*trunk*" per permettere il passaggio sia in ingresso che in uscita dei pacchetti con i tag VLAN 222 e 444.

```
1  {
2    "apps":{
3      "org.onosproject.tiessemanager":{
4        "vlansconfig":{
5          "vlans": [
6              {
7                  "mode": "ACCESS",
8                  "interface":"eth2",
9                  "port":"2",
10                 "vlan":"222",
11                 "ipaddress": "192.168.20.2",
12                 "netmask" : "255.255.255.0",
```

```
13                      "broadcast" : "192.168.20.255"
14                  },
15                  {
16                      "mode": "ACCESS",
17                      "interface":"eth4",
18                      "port":"4",
19                      "vlan":"444",
20                      "ipaddress": "192.168.40.4",
21                      "netmask" : "255.255.255.0",
22                      "broadcast" : "192.168.40.255"
23                  },
24                  {
25                      "mode": "TRUNK",
26                      "interface" : "eth3",
27                      "port":"3",
28                      "vlan":"222",
29                      "ipaddress": "192.168.30.31",
30                      "netmask" : "255.255.255.0",
31                      "broadcast" : "192.168.30.255"
32                  },
33                  {
34                      "mode": "TRUNK",
35                      "interface" : "eth3",
36                      "port":"3",
37                      "vlan":"444",
38                      "ipaddress": "192.168.30.32",
39                      "netmask" : "255.255.255.0",
40                      "broadcast" : "192.168.30.255"
41                  }
42              ]
43          }
44      }
45  }
46 }
```

Listing 10.1. File in formato JSON contenente la configurazione delle VLAN da applicare sul dispositivo Tiesse Imola.

10.3 Struttura dell'applicazione e delle classi Java

Il codice della applicazione è strutturato in due sezioni principali: quella relativa alla definizioni delle strutture dati e al *parsing* del file JSON e quella relativa alla logica in cui l'applicazione resta in ascolto di nuove configurazioni, elabora i dati e a richiama il driver per attuare la configurazione delle VLAN.

Le classi Java dell'applicazione sviluppata che compongono la prima sezione sono:

- **AccessData.java**: classe che contiene la definizione della struttura dati utilizzata per le interfacce da impostare in modalità "*access*". I campi che la compongono sono: interfaccia, porta, tag VLAN, indirizzo ip, netmask e broadcast. Sono inoltre definiti tutti i metodi getter e setter per accedere ai vari campi in lettura e in scrittura.
- **TrunkData.java**: classe che contiene la definizione della struttura dati utilizzata per le interfacce da impostare in modalità "*trunk*". I campi che la compongono sono: interfaccia, porta, tag VLAN, indirizzo ip, netmask e broadcast. Sono inoltre definiti tutti i metodi getter e setter per accedere ai vari campi in lettura e in scrittura.
- **TiesseConfig.java**: classe responsabile del parsing del JSON contenente la configurazione delle VLAN e della creazione delle liste di tipo "*AccessData*" e "*TrunkData*" che contengono i dati ricevuti.

Le classi Java dell'applicazione sviluppata che compongono la seconda sezione sono:

- **TiesseConfigService.java**: interfaccia che definisce i metodi "*accessModeData()*" e "*trunkModeData()*' che ritornano rispettivamente la lista degli oggetti di tipo "*AccessData*" e "*TrunkData*".
- **TiesseConfigManager.java**: classe che implementa l'interfaccia "*TiesseConfigService.java*" e i metodi "*accessModeData()*" e "*trunkModeData()*". Inoltre è responsabile della registrazione di un *listener* per restare in ascolto di file JSON e della logica necessaria per attuare le configurazioni ricevute sul dispositivo *Tiesse Imola*. Quest'ultima azione viene svolta andando a richiamare i metodi "addAccessMode()" e "addVlanAndIpAddrAndNetmaskToInterface()" del *behaviour* "*InterfaceConfigTiesse*" del driver.

10.4 Integrazione dell'applicazione in ONOS

Per poter integrare l'applicazione sviluppata all'interno di ONOS è stato necessario seguire un preciso procedimento. Tale procedimento consiste in:

1. Aggiungere una nuova cartella che possiede il nome della applicazione (in questo caso "*tiessemanager*") in "*onos/apps*". Creare all'interno della cartella appena aggiunta la serie di sottocartelle "*src/main/java/org/onosproject/tiessemanager*", dove "*org/onosproject/tiessemanager*" sarebbe il package relativo alle classi Java esaminate nella sezione precedente. L'informazione relativa al package è contenuta nel file "*package-info.java*", il quale contiene semplicemente:

```
1    package org.onosproject.tiessemanager
```

   Tutte le classi Java che compongono l'applicazione (esaminate nella sezione precedente) devono essere inserite all'interno della cartella "*src/main/java/org/onosproject/tiessemanager*".

2. Inserire in "*onos/apps/tiessemanager*" un file BUCK che specifichi le dipendenze di compilazione, di test, con le altre applicazioni e i dettagli della applicazione. Il file BUCK dell'applicazione sviluppata in questo lavoro di tesi contiene:

```
1    COMPILE_DEPS = [
2    '//lib:CORE_DEPS',
3    '//lib:ONOS_YANG',
4    '//lib:JACKSON',
5    '//protocols/netconf/api:onos-protocols-netconf-
       api',
6    '//protocols/netconf/ctl:onos-protocols-netconf-
       ctl',
7    '//lib:org.apache.karaf.shell.console',
8    '//incubator/api:onos-incubator-api',
9    '//drivers/examplenetconfdriver:onos-drivers-
       examplenetconfdriver'
10   ] + YANG_TOOLS
11
12   TEST_DEPS = [
13   '//lib:TEST_ADAPTERS',
14   '//core/api:onos-api-tests',
15   '//utils/osgi:onlab-osgi-tests',
16   '//incubator/net:onos-incubator-net'
17   ]
18
19   APPS = [
20   'org.onosproject.yang',
```

```
21      'org.onosproject.config',
22      'org.onosproject.netconf',
23      'org.onosproject.netconfsb',
24      'org.onosproject.drivers.netconf'
25      ]
26
27      osgi_jar_with_tests (
28      deps = COMPILE_DEPS,
29      test_deps = TEST_DEPS
30      )
31
32      onos_app (
33      app_name = 'org.onosproject.tiessemanager',
34      title = 'tiessemanager app',
35      category = 'Traffic Steering',
36      url = 'http://onosproject.org',
37      description = 'ONOS tiessemanager YANG
           application.',
38      required_apps = APPS,
39      )
```

I campi “COMPILE_DEPS” e in “TEST_DEPS” contengono le dipendenze necessarie per la compilazione ed eventuali test. Le applicazioni necessarie invece sono elencate sotto il campo “APPS”.

3. Aggiungere la *entry* della applicazione nel file “*onos/modules.def*”. Per scrivere correttamente la *entry* è necessario sostituire i "/" contenuti nel path del file con "-" e aggiungere "-oar" come suffisso. (e.g.: “*onos/apps/tiessemanager*” diventa “*onos-apps-tiessemanager-oar*”).

   In questo caso è necessario aggiungere a “*onos/modules.def*”:

```
1       ONOS_APPS = [
2       ...
3
4        '//apps/tiessemanager:onos-apps-tiessemanager-
            oar'
5
6       ...
```

Dopo avere effettuato questa procedura è possibile eseguire la build di ONOS tramite BUCK. Infine una volta avviato ONOS, è necessario attivare il driver "*examplenetconfdriver*") prima della applicazione "*tiessemanager*") . Per attivare l'applicazione è necessario scrivere nella CLI di ONOS il comando:

```
1 $  app activate org.onosproject.tiessemanager
```

10.5 Funzionamento

Al momento dell'attivazione della applicazione (Listato 10.2), prima di tutto viene registrato il nome dell'applicazione "*org.onosproject.tiessemanager*") nel servizio "*CoreService*", responsabile dell'interazione con il *core* del controllore.

In seguito viene registato nel servizio "*NetworkConfigRegistry*" la *factory* necessaria per riconoscere le configurazioni destinate all'applicazione e interpretarle correttamente (Listato 10.3).

```
95      public void activate() {
96          appId = coreService.getAppId(APP_NAME);
97          configRegistry.registerConfigFactory(
               configFactory);
98          configService.addListener(configListener);
99          log.info("Started");
100     }
```

Listing 10.2. Attivazione della applicazione in ONOS.

```
79      private final ConfigFactory configFactory =
80              new ConfigFactory(SubjectFactories.
                   APP_SUBJECT_FACTORY, TiesseConfig.class, "
                   vlansconfig") {
81                  @Override
82                  public TiesseConfig createConfig() {
83                      return new TiesseConfig();
84                  }
85              };
```

Listing 10.3. Creazione della factory necessaria per il NetworkConfigRegistry.

Infine viene creato un *listener* (Listato 10.4), il quale viene registrato nel servizio di "*Network Configuration*", responsabile della ricezione delle configurazioni tramite il push di un file JSON. In tal modo l'applicazione resta in attesa dell'evento in cui

viene ricevuto un file JSON destinato ad essa, nel quale è contenuta la configurazione delle VLAN. In Figura 10.2 vengono mostrati i componenti ONOS con cui la applicazione interagisce.

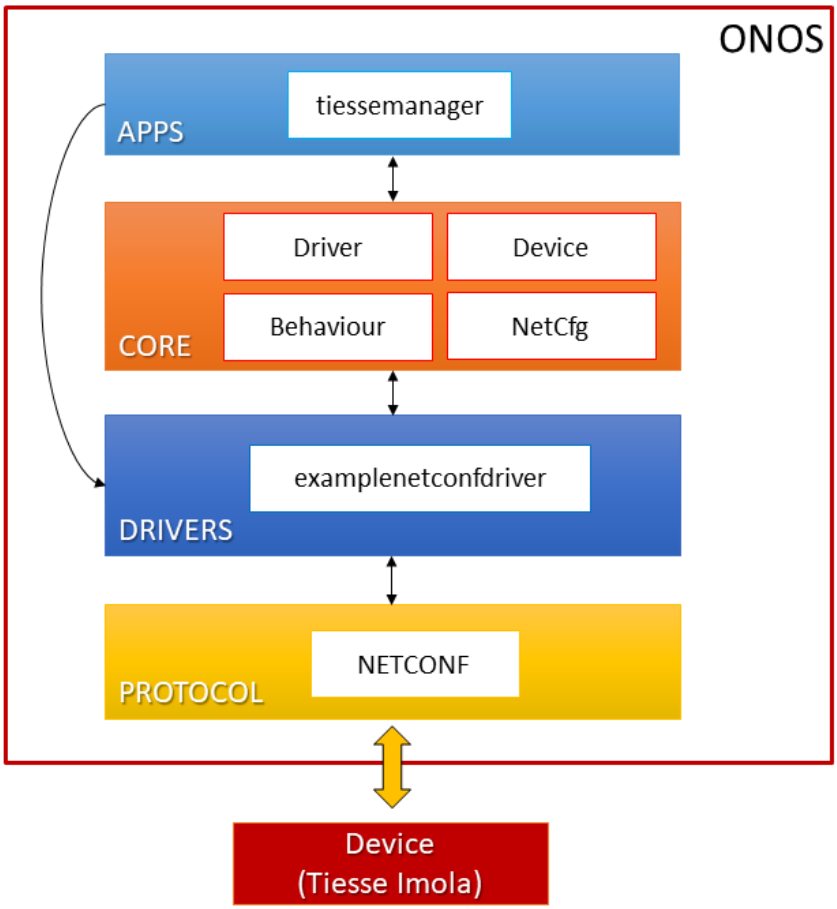

Figura 10.2. Rappresentazione a blocchi dei componenti ONOS con cui la applicazione interagisce.

All'occorrere dell'evento viene lanciato un thread (*eventExecutor.execute()*) che va a eseguire il metodo "*readConfig()*", che si occupa della logica necessaria per fare il *parsing* della configurazione. Successivamente esso va a richiamare a sua volta il metodo "*ConfigTiesseDevice()*" responsabile della logica di attuazione della configurazione sul dispositivo.

L'uso del thread è necessario per mantenere l'applicazione reattiva all'occorrere di altri eventi durante la fase di elaborazione.

```
272     private class InternalConfigListener implements
          NetworkConfigListener {
```

```
273         @Override
274         public void event(NetworkConfigEvent event) {
275             if (!event.configClass().equals(TiesseConfig.
                    class)) {
276                 return;
277             }
278             log.debug("Listener called: {}", event.type()
                    );
279             switch (event.type()) {
280                 case CONFIG_ADDED:
281                     log.info("Network configuration added
                            ");
282                     eventExecutor.execute(
                            TiesseConfigManager.this::
                            readConfig);
283                     break;
284                 case CONFIG_UPDATED:
285                     log.info("Network configuration
                            updated");
286                     eventExecutor.execute(
                            TiesseConfigManager.this::
                            readConfig);
287                     break;
288                 default:
289                     break;
290             }
291         }
292     }
```

Listing 10.4. Creazione del listener necessario per far ascoltare alla applicazione gli eventi di ricezione di file di configurazione.

Il metodo "*readConfig()*" (Listato 10.5) prende la configurazione ricevuta tramite il file JSON iniettato in ONOS e procede ad effettuarne il *parsing*, andando a creare riempire delle liste contenenti i dati sia per le interfacce da impostare in modalità "*access*", sia per quelle da impostare in modalità "*trunk*". Successivamente richiama il metodo "*ConfigTiesseDevice()*", responsabile di attuare la configurazione ricevuta sul dispositivo.

```
128     private void readConfig() {
129         TiesseConfig config = configRegistry.getConfig(
                appId, TiesseConfig.class);
130         accessModeDataList = config.getAccessModeData();
```

```
131         trunkModeDataList = config.getTrunkModeData();
132         ConfigTiesseDevice();
133     }
```

Listing 10.5. Creazione del listener necessario per far ascoltare alla applicazione gli eventi di ricezione di file di configurazione.

Il metodo "*ConfigTiesseDevice()*" (Listato 10.6) implementa la logica necessaria per attuare la configurazione ricevuta al dispositivo *Tiesse Imola*. Nel caso siano presenti più dispositivi *Tiesse* da configurare, un ciclo si occupa di attuare la configurazione ricevuta in ognuno di essi.

I dispositivi interessati vengono individuati tramite la verifica del supporto del *behaviour* "*InterfaceConfigTiesse*" definito nel driver "*examplenetconfdriver*" sviluppato in questo lavoro di tesi. In caso affermativo l'algoritmo crea un *handler* del driver per quel *device* e procede a richiamare i metodi "*addAccessMode()*" e "*addVlanAndIpAddrAndNetmaskToInterface()*". Nel caso di interfacce da impostare in modalità "*access*", entrambi i metodi vengono richiamati. Mentre nel caso di interfacce da impostare in modalità "*trunk*", è sufficiente richiamare solo il metodo "*addVlanAndIpAddrAndNetmaskToInterface()*", in quanto se non specificato diversamente, il dispositivo *Tiesse Imola* interpreta in automatico l'interfaccia come "*trunk*".

```
140     private void ConfigTiesseDevice() {
141
142         Iterable<Device> devices = deviceService.
               getAvailableDevices();
143
144         for (Device device : devices) {
145         if (device.manufacturer().equals("Tiesse")) {
146         if (device.is(InterfaceConfigTiesse.class)){ //
               InterfaceConfigTiesse behaviour is supported
               by the device
147
148         DriverHandler handler = driverService.
               createHandler(device.id());
149         InterfaceConfigTiesse interfaceConfig = handler.
               behaviour(InterfaceConfigTiesse.class);
150
151         if (!accessModeDataList.isEmpty()) {
152             for (AccessData accessModeData:
                   accessModeDataList){
153                 String intf = accessModeData.getIntf();
154                 String port = accessModeData.getPort();
```

```
155                 String accessVlanString = accessModeData.
                        getVlan();
156                 String accessIpAddr = accessModeData.
                        getIpaddress();
157                 String accessNetmask = accessModeData.
                        getNetmask();
158                 String accessBroadcast = accessModeData.
                        getBroadcast();
159
160                 VlanId accessVlanId = VlanId.vlanId(Short
                        .parseShort(accessVlanString));
161
162                 interfaceConfig.addAccessMode(intf,
                        accessVlanId); //set intf in access
                        mode
163
164                 interfaceConfig.
                        addVlanAndIpAddrAndNetmask ToInterface
                        (intf,accessVlanId,accessIpAddr,
                        accessNetmask,accessBroadcast); //set
                        vlan with ip addr, netmask and
                        broadcast
165                 }
166         }
167         if (!trunkModeDataList.isEmpty()) {
168             for (TrunkData trunkModeData:
                    trunkModeDataList) {
169
170                 String intf = trunkModeData.getIntf();
171                 String port = trunkModeData.getPort();
172                 String trunkVlanString = trunkModeData.
                        getVlan();
173                 String trunkIpAddr = trunkModeData.
                        getIpaddress();
174                 String trunkNetmask = trunkModeData.
                        getNetmask();
175                 String trunkBroadcast = trunkModeData.
                        getBroadcast();
176
177                 VlanId trunkVlanId = VlanId.vlanId(Short.
                        parseShort(trunkVlanString))
```

```
178
179                     interfaceConfig.
                           addVlanAndIpAddrAndNetmaskToInterface(
                           intf,trunkVlanId,trunkIpAddr,
                           trunkNetmask, trunkBroadcast); //set
                           vlan with ip addr, netmask and
                           broadcast
180
181                                 }
182                              }
183                     }
184                 }
185             }
186     }
```

Listing 10.6. Attuazione della configurazione delle interfacce e delle VLAN per ogni dispositivo presente.

Dopo aver applicato la configurazione sul dispositivo (o più dispositivi, se presenti), l'applicazione continua ad essere in ascolto per la ricezione di nuovi file di configurazione. In caso venga iniettato un nuovo file JSON, l'applicazione lancerà un nuovo thread che richiamerà il metodo metodo "*readConfig()*" e ripeterà i passi spiegati in precedenza.

Capitolo 11

Risultati ottenuti

Nella prima parte del lavoro svolto in questa tesi, in cui è stata focalizzata l'attenzione su CORD e E-CORD, si è riusciti a implementare un servizio XOS e a modificare un *Service Graph*. In particolare si è lavorato a livello di modello e si è riusciti a modificare il *Service Graph* standard composto da vOLT, vSG e vRouter, riuscendo ad aggiungere un ulteriore servizio e ad ottenere anche una biforcazione all'interno del grafo. Infatti ciò permette di implementare ulteriori servizi utili per il cliente *enterprise*, come ad esempio analisi del traffico o gestione della banda. Tramite la biforcazione del grafo è quindi possibile scegliere una catena di servizi differente a seconda delle esigenze del cliente.

Il lavoro svolto è stato validato con l'uso di una versione *front-end* di XOS, in cui le risorse sottostanti (e.g.: ONOS, OpenStack) sono assenti. In questo modo si è potuto esplorare le possibilità di modellazione del *Service Graph* per quanto riguarda la creazione dei servizi, la loro modifica ed inserimento nel grafo e i rapporti di *tenancy* tra loro. L'implementazione dell'interazione con le risorse sottostanti e dell'attuazione del *Service Graph* non è stato preso in considerazione nel lavoro svolto, a causa della mancanza della documentazione per sviluppatori.

Nella seconda parte del lavoro svolto in questa tesi è stata focalizzata l'attenzione sull'implementazione in ONOS di un driver e di una applicazione per la gestione e la configurazione di un dispositivo posto nel lato *access* dello scenario presentato precedentemente.

Si è riusciti con successo a raggiungere gli obiettivi iniziali di scrivere un driver basato su *data model* YANG e su l'uso del protocollo NETCONF per la configurazione del dispositivo. Infatti il driver e l'applicazione sviluppati sono in grado di: interagire correttamente con il router *Tiesse Imola*, inviare i messaggi necessari per attuare la configurazione e configurare con successo il dispositivo.

Un vantaggio che proviene dall'uso del controllore ONOS per configurare il dispositivo è sicuramente la automatizzazione del processo di configurazione, oltre che una notevole riduzione dei tempi.

Inoltre l'applicazione sviluppata è in grado di configurare più dispositivi, se presenti, con la configurazione desiderata.

In particolare è possibile impostare le interfacce del dispositivo in modalità *access* o *trunk* e assegnare i tag VLAN desiderati per permettere il corretto instradamento del traffico a livello due tra le porte del *device*. Questa distizione a livello di tag VLAN è necessaria per permettere successivamente all'infrastruttura CORD di distinguere i clienti *enterprise* e associargli i servizi che essa stessa mette a disposizione.

Un altro vantaggio portato dall'uso dell'applicazione e del driver è quello della rimozione dell'errore umano a livello di configurazione.

Infatti una configurazione manuale del dispositivo può essere soggetta ad errori logici o sintattici che si manifestano durante la esecuzione dei comandi tramite CLI del dispositivo. Tali errori costringono alla ripetizione della procedura e quindi a un considerevole aumento dei tempi necessari per la procedura manuale di configurazione del dispositivo.

Per valutare i vantaggi introdotti dal lavoro svolto in termini di tempo rispetto ad una configurazione manuale, sono stati eseguiti dei test prestazionali per capire con quale velocità avvenga una configurazione completa del router *Tiesse Imola* nello scenario analizzato.

11.1 Banco di Prova

Le analisi prestazionali sono state svolte sul banco di prova mostrato in Figura 11.1.

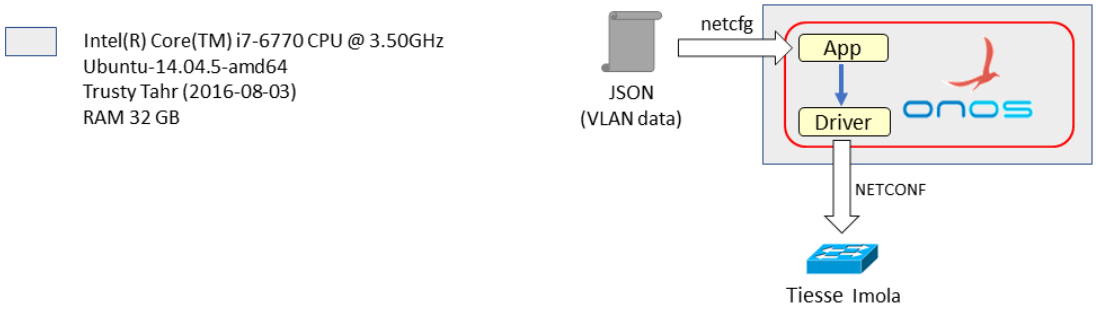

Figura 11.1. Banco di prova utilizzato per le analisi prestazionali.

Il router *Tiesse Imola* dispone di 6 porte Ethernet (0 - 5). Una di esse è utilizzata per connettere il dispositivo al PC su cui viene eseguito ONOS. Quindi le porte disponibili sono 5.

11.2 Analisi prestazionale

Nello scenario di rifermento della rete d'accesso in cui viene inserito il Tiesse Imola, la porta che si affaccia verso il sito CORD viene configurata in modalità *trunk*, mentre le restanti 4 vengono utilizzate in modalità *access* per collegare gli host.

Si è dunque proceduto ad effettuare diversi test in cui vengono configurate le interfacce come descritto nello scenario di riferimento per misurarne le tempistiche e trovare un valore temporale medio.

Ogni test effettuato consiste nell'invio al dispositivo *Tiesse Imola* di una configurazione completa delle interfacce e nella misurazione sia del tempo totale impiegato per attuarla, sia dei singoli tempi impiegati per impostare le interfacce in modalità *trunk* o *access* e per assegnargli un tag VLAN. Il file JSON utilizzato per iniettare in ONOS la configurazione contiene le VLAN da assegnare ad una interfaccia in modalità *trunk* e a quattro interfacce in modalità *access*.

Poiché si ha il *LAN splitting* attivato sul dispositivo Tiesse, assegnando un tag VLAN ad una interfaccia, si afferma indirettamente che essa sia in modalità *trunk*. Invece per impostare una interfaccia in modalità *access*, è necessario configurarla esplicitamente con un ulteriore comando.

Pertanto per configurare una interfaccia in modalità *trunk* è necessario inviare al dispositivo un solo RPC di configurazione contenente il tag VLAN da assegnare. Invece per configurare una interfaccia in modalità *access* sono necessari due RPC: uno per specificare che si tratta di modalità *access* e l'altro per assegnare il tag VLAN ad essa. Per questo motivo l'attuazione della configurazione delle interfacce in modalità *access* è più onerosa in termini di tempo.

	Average Configuration Time [ms]	Variance	Standard Deviation
Trunk mode	350	184,6	13,6
Access mode	450	329,7	18,1

Tabella 11.1. Tempi medi per l'attuazione della configurazione sulle interfacce del dispositivo, varianza e deviazione standard delle misure effettuate.

In accordo con i risultati mostrati in Tabella 11.1, i tempi medi risultanti per effettuare la configurazione di una interfaccia *trunk* e di una interfaccia *access* sono:

- 350 ms per l'assegnazione del tag VLAN a una interfaccia in modalità *trunk*.
- 450 ms per impostare una interfaccia in modalità *access* (100 ms) e per assegnargli il tag VLAN (350 ms).

Dunque prendendo come riferimento la configurazione di riferimento (1 *trunk*, 4 *access*) si è ottenuto che il tempo totale medio impiegato per attuare la configurazione completa sul dispositivo *Tiesse Imola* è di 2,15 secondi.

Il calcolo della varianza e della deviazione standard sulle misure effettuate (Tabella 11.1) mostrano una certa variabilità dei tempi di configurazione. Quest'ultima è data maggiormente dal tempo che il dispositivo impiega per attuare la configurazione sulle interfacce, piuttosto che dal tempo di esecuzione del codice in ONOS o dal RTT (Round Trip Time) dei pacchetti inviati.

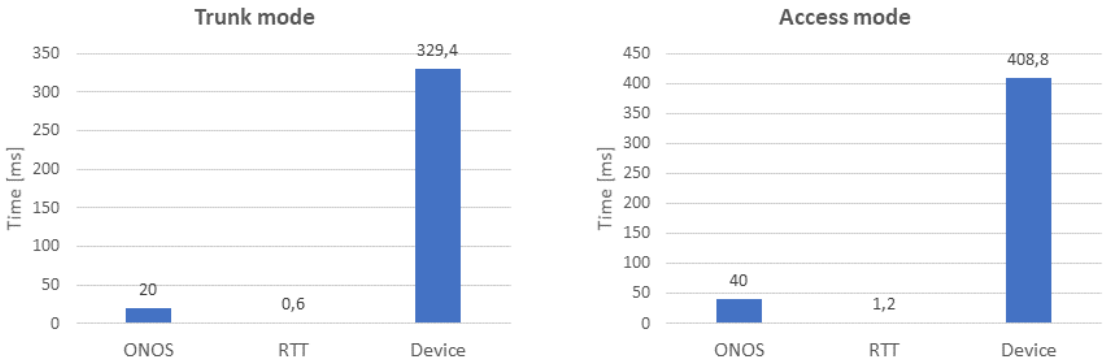

Figura 11.2. Distribuzione dei tempi impiegati per l'assegnazione di VLAN e la configurazione delle interfacce in modalità trunk e access.

In Figura 11.2 è mostrata la distribuzione dei tempi impiegati per l'assegnazione della VLAN e per la configurazione di una interfaccia rispettivamente in modalità *trunk* e in modalità *access*. Da questi grafici è possibile notare come il tempo richiesto per configurare le interfacce sia per lo più occupato dal dispositivo *Tiesse Imola* nell'attuare la configurazione.

Nella macchina utilizzata per effettuare questi test, l'esecuzione del solo codice ONOS dell'applicazione e del driver è dell'ordine delle decine di millisecondi, mentre il *Round Trip Time* (RTT) dei pacchetti inviati è all'incirca di un millisecondo. Dunque si può dedurre che il dispositivo *Tiesse Imola*, una volta ricevuti gli RPC contenenti la configurazione delle VLAN, impiega circa 2 secondi in totale per attuarla sulle interfacce.

I risultati ottenuti mostrano che la soluzione proposta dalla applicazione e dal driver ONOS sviluppati, usando il protocollo NETCONF per inviare RPC di configurazione, introducono miglioramenti prestazionali piuttosto importanti.

Infatti si passa da una configurazione manuale soggetta a possibili errori che richiede tempi dell'ordine di minuti, a una configurazione automatica che rimuove l'errore umano e riduce i tempi di attuazione all'ordine dei secondi. Inoltre la soluzione sviluppata scala da un punto di vista temporale in maniera lineare in presenza di più dispositivi da configurare.

Capitolo 12

Conclusioni e sviluppi futuri

In questa tesi sono state innanzitutto analizzate le tecnologie CORD e E-CORD all'interno dell'ambito delle reti geografiche. Esplorando i componenti che le costituiscono si è riusciti a comprendere nel dettaglio il loro funzionamento e i vantaggi che esse portano nell'ambito delle SD-WAN.

La tecnologia di CORD, che si pone l'obiettivo di ristrutturare i *Central Office* in *data center*, si è rivelata particolarmente interessante in quanto permette all'operatore di rete di ridurre radicalmente i costi operativi e di gestione e di mettere a disposizione una vasta collezione di servizi ai clienti grazie al paradigma della virtualizzazione.

Similmente E-CORD, benché si tratti di un *proof of concept*, pone le basi necessarie all'operatore di rete che al giorno d'oggi ha bisogno di un flessibile e rapido sistema di orchestrazione a livello globale di una rete WAN, il quale nello scenario analizzato è necessario per orchestrare le SDN sottostanti e implementare servizi quali la connettività end-to-end tra vari siti CORD.

Essendo entrambi progetti attualmente in via di sviluppo, specialmente E-CORD ancora in fase sperimentale, non è stato possibile realizzare una loro applicazione completa nello scenario analizzato di connettività end-to-end in ambito di reti geografiche.

Infatti l'implementazione del servizio XOS sviluppato in questa tesi consente una gestione a livello di front-end del modello del *Service Graph*, ma la mancanza di documentazione e di API necessarie per interagire con la piattaforma non hanno permesso di effettuare il *deploy* del grafo sulle risorse di back-end sottostanti (e.g.: ONOS, OpenStack).

Pertanto successivamente si è deciso di focalizzare l'attenzione sulla parte della rete d'accesso dei clienti aziendali, in particolar modo sull'integrazione in ONOS del controllo del CPE. Infatti nello specifico caso d'uso preso in considerazione, il controllo esercitato dall'orchestrazione di E-CORD si ferma ai siti CORD e non prevede la gestione del CPE.

Si è dunque proceduto a implementare in ONOS la gestione di tali dispositivi tramite lo sviluppo di un driver basato sull'uso di modelli YANG e di una applicazione specifica per la gestione e la configurazione delle loro interfacce fisiche. Quest'ultima è infatti necessaria per permettere al traffico di essere instradato correttamente all'interno dello scenario preso in considerazione e per definire i *c-tag* (customer-tag) che distinguono i clienti aziendali all'interno della piattaforma di E-CORD.

Uno dei vantaggi che porta la soluzione sviluppata è sicuramente l'automazione della configurazione di uno o più CPE. Infatti l'applicazione realizzata è in grado tramite un algoritmo di rilevare tutti i CPE controllati da ONOS e applicare loro la configurazione desiderata. Inoltre la scelta di basare lo sviluppo del driver su *data model* YANG consente in ONOS di generare in maniera automatica le classi JAVA che li rappresentano, permettendo di semplificare lo sviluppo del codice necessario per configurare i dispositivi.

I test effettuati per valutare le prestazioni della configurazione del dispositivo tramite ONOS risultano promettenti in quanto essa, rispetto a una configurazione manuale, permette di ridurre il tempo richiesto all'ordine dei secondi, oltre che elimina la possibilità di errore umano.

In particolare nel driver sviluppato si è focalizzata l'attenzione sulla implementazione delle funzionalità necessarie per svolgere la configurazione delle interfacce e delle VLAN sul CPE. Un possibile sviluppo futuro a riguardo potrebbe essere l'inserimento di ulteriori funzionalità all'interno del driver andando ad integrare per esempio la gestione di protocolli di routing o di altre *features* che permettono una più approfondita configurazione del dispositivo.

Similmente sarebbe possibile estendere la logica dell'applicazione per renderla in grado di interpretare file di configurazioni più dettagliati, che non si limitano a contenere solo i dati necessari per attuare la configurazione delle VLAN e delle interfacce.

Un ulteriore sviluppo futuro potrebbe essere quello di integrare il controllore ONOS che gestisce il CPE in E-CORD, problema che in questo lavoro di tesi non è stato trattato. Infatti in questo modo, grazie alla sua struttura gerarchica di gestione dei controllori, E-CORD sarebbe in grado di gestire servizi quali ad esempio quello della connettività end-to-end fino al CPE, che può essere così controllato e configurato remotamente dall'operatore di rete.

A fronte dei risultati ottenuti si aprono quindi ulteriori prospettive di sviluppo relative alla orchestrazione SDN nell'ambito delle reti geografiche.

Appendice A

Creazione dell'ambiente front-end di XOS

Questa sezione spiega i passi che portano al deploy dell'ambiente front-end di XOS nel profilo "*mock-rcord*". Tale profilo va a generare il *Service Graph* di base di un R-CORD (Residential-CORD) che consiste in: *vOLT, vSG* e *vRouter*.

L'ambiente di front-end è privo degli elementi operazionali sottostanti (e.g: OpenStack, ONOS), ma permette di lavorare sulla modellazione del *Service Graph* e a interagire con la GUI di XOS.

A.1 Prerequisiti

Il sistema operativo necessario per eseguire l'ambiente di front-end di XOS è Ubuntu 14.04. La versione di CORD utilizzata è la 3.0.

Prima di tutto è necessario scaricare CORD 3.0 dal suo repository su Github [25]. Inoltre è necessario avere installato Vagrant [26], il software per creare l'ambiente di sviluppo in una macchina virtuale.

È possibile installarlo tramite il comando bash:

```
1 $  sudo apt-get install vagrant
```

A.2 Ambiente front-end di XOS

Tutti i comandi seguenti assumono di essere eseguiti all'interno della directory "*cord/build/platform-install*".

Da questa cartella eseguire il comando bash:

```
1 $  vagrant up head-node
```

Con questo comando viene creata una macchina virtuale su *Vagrant*, con cui viene condivisa la cartella "*cord/*" della macchina fisica. Ciò permette di poter fare le modifiche al codice localmente: esse verranno poi automaticamente riflesse nella VM, in maniera da velocizzare il processo di *testing* del codice.

Una volta creata, è possibile collegarsi alla macchina virtuale eseguendo il comando bash:

```
1 $  vagrant ssh head-node
```

All'interno della VM spostarsi nella cartella "*platform-install*" col comando bash:

```
1 $  cd ~/cord/build/platform-install
```

Da qui è possibile eseguire il *deploy* del profilo "*mock-rcord*" con la configurazione di front-end di XOS tramite il comando bash:

```
1 $  ansible-playbook -i inventory/mock-rcord
     deploy-xos-playbook.yml
```

Impostare l'ambiente iniziale e lanciare XOS per la prima volta richiede all'incirca 30 minuti.

Le credenziali necessarie per accedere alla GUI di XOS tramite broswer sono autogenerate e possono essere trovate in un file posto all'interno della VM nella cartella "*credentials/*". Tale file possiede come nome lo username e come contenuto la password. Lo username per l'admin di XOS è "*xosadmin@opencord.org*".

Nel caso si vogliano fare dei cambiamenti nel codice e si vuole che tali cambiamenti vengano riflessi nei container in cui viene eseguito XOS, è possibile eseguire il *redeploy* del profilo con il comando bash:

```
1  $  ansible-playbook -i inventory/mock-rcord
      redeploy-xos-playbook.yml
```

Fare nuovamente il *build* e il *deploy* dei container usando questo metodo impiega molto meno tempo della fase iniziale. Questo comando mantiene il database esistente nello stato in cui era prima del *redeploy*.

Se invece si vuole fare il *teardown* dell'ambiente, per esempio per cambiare profilo o per lanciare il profilo corrente con un reset del database, si può eseguire il comando bash:

```
1 $  ansible-playbook -i inventory/mock-rcord
     teardown-playbook.yml
```

Questo comando distrugge tutti i container docker creati. Dopo il *teardown* dell'ambiente è possibile fare cambiamenti al codice e fare il *deploy* dello stesso profilo usato in precedenza o di un altro.

Appendice B

Creazione di un driver ONOS basato su YANG data model

Questa sezione spiega i passi che portano alla creazione di un driver ONOS basato su modelli YANG e alla generazione automatica delle classi Java che rappresentano tali modelli. Questa procedura permette al driver creato di poter utilizzare, dopo la compilazione tramite *YANG Compiler*, le classi Java automaticamente generate all'interno del codice.

B.1 Prerequisiti

Prima di tutto è necessario scaricare ONOS dal suo repository su Github [27].

La versione di ONOS di riferimento utilizzata è la 1.12.0 e come tool di build di ONOS viene utilizzato BUCK [28].

B.2 Procedimento

I passi necessari per creare il driver basato su YANG *data model* e generare automaticamente le classi Java da tali modelli sono:

1. Aggiungere una nuova cartella che possiede il nome del driver in "*onos/models*". Creare all'interno della cartella appena aggiunta una serie di sottocartelle: "*src/main/yang*". All'interno dell'ultima sottocartella "*yang*" devono essere inseriti i file YANG contenenti i *data model* per il driver, facendo attenzione a inserire anche eventuali file YANG richiesti al loro interno nel campo *import*.

2. Inserire in "*onos/models/nomedriver*" (dove al posto di *nomedriver* si intende il nome del driver che si sta creando) un file BUCK col seguente contenuto:

```
yang_model(
app_name = 'org.onosproject.models.nomedriver',
title = 'nomedriver YANG Model',
)
```

3. Aggiungere una nuova cartella che possiede il nome del driver in "*onos/drivers*". Creare all'interno della cartella appena aggiunta una serie di sottocartelle: "*src/main/java*". La cartella "*java*" non può essere vuota e deve contenere il package relativo alle classi Java che si vogliono creare. Ad esempio creando il package "*org.onosproject.drivers.nomedriver*", si va a inserire le classi java all'interno della cartella "*src/main/java/org/onosproject/nomedriver*". Una volta creato il package è necessario aggiungere il file "*package-info.java*" contenente semplicemente:

```
package org.onosproject.driver.nomedriver
```

   A questo punto si possono creare tutte le classi Java necessarie e inserirle all'interno della cartella "*src/main/java/org/onosproject/nomedriver*".

4. Inserire in "*onos/drivers/nomedriver*" (dove al posto di *nomedriver* si intende il nome del driver che si sta creando) un file BUCK che specifichi le dipendenze di compilazione, di test, con le altre applicazioni e i dettagli del driver che si sta andando a creare. Un file BUCK di base deve avere il seguente contenuto:

```
    COMPILE_DEPS = [
    '//lib:CORE_DEPS',
    '//lib:ONOS_YANG',
    '//drivers/utilities:onos-drivers-utilities',
    '//models/nomedriver:onos-models-nomedriver',
    ] + YANG_TOOLS

    TEST_DEPS = [
    '//lib:TEST_ADAPTERS',
    '//utils/osgi:onlab-osgi-tests'
    ]

    APPS = [
    'org.onosproject.yang',
    #     'org.onosproject.yang-gui',
```

```
16      'org.onosproject.models.nomedriver'
17      ]
18
19      osgi_jar_with_tests (
20      deps = COMPILE_DEPS,
21      test_deps = TEST_DEPS
22      )
23
24      onos_app (
25      app_name = 'org.onosproject.drivers.nomedriver',
26      title = 'nomedriver sample device driver',
27      category = 'Drivers',
28      url = 'http://onosproject.org',
29      description = 'ONOS nomedriver device driver.',
30      required_apps = APPS,
31      )
```

Nel caso si vada poi ad utilizzare ulteriori librerie o protocolli, le loro dipendenze vanno aggiunte in "COMPILE_DEPS" e in "TEST_DEPS" se necessario. Invece se sono necessari ulteriori applicazioni di supporto esse vanno elencate sotto "APPS".

5. Aggiungere le *entry* del driver e del modello nel file "*onos/modules.def*". Per scrivere correttamente le *entry* è necessario sostituire i "/" contenuti nel path del file con "-" e aggiungere "-oar" come suffisso. (e.g.: "*onos/drivers/nomedriver*" diventa "*onos-drivers-nomedriver-oar*" e "*onos/models/nomedriver*" diventa "*onos-drivers-nomedriver-oar*"). In questo caso è necessario aggiungere a "*onos/modules.def*":

```
1  ONOS_DRIVERS = [
2  ...
3
4   '//drivers/nomedriver:onos-drivers-nomedriver-oar'
5
6  ...
7  ]
8
9  MODELS = [
10 ...
11
12 '//models/nomedriver:onos-models-nomedriver-oar'
13
```

```
14 ...
15 ]
```

6. Lanciare BUCK per eseguire la build del codice. Per fare questo è necessario spostarsi da shell prima nella cartella "*onos*":

```
1     $  cd onos
```

   E poi lanciare il comando:

```
1     $  tools/build/onos-buck build onos --show-output
```

Se la build viene effettuata correttamente i file Java vengono generati automaticamente dai file YANG all'interno della cartella "*onos/buck-out/gen/models/nomedriver/onos-models-nomedriver-yang#srcs__yang-gen*". In questo modo è possibile utilizzare i file Java generati all'interno del codice del driver.

Bibliografia

[1] *ONOS project website*. URL: http://onosproject.org/.

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

[3] Omar SEFRAOUI, Mohammed AISSAOUI e Mohsine ELEULDJ. «OpenStack: Toward an Open-Source Solution for Cloud Computing». In: *International Journal of Computer Applications* 55 (2012), pp. 38–42.

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

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

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

[7] *CORD website*. URL: https://opencord.org/.

[8] *Open Networking Foundation website*. URL: https://www.opennetworking.org/.

[9] *CORD wiki website*. URL: https://wiki.opencord.org/.

[10] Larry Peterson et al. «Central Office Re-Architected as a Data Center». In: *IEEE Communications Magazine* 54 (2016), pp. 96–101.

[11] *Metro Ethernet Forum website*. URL: https://www.mef.net/.

[12] Larry Peterson et al. «XOS: An Extensible Cloud Operating System». In: *BigSystem '15 Proceedings of the 2nd International Workshop on Software-Defined Ecosystems* (2015), pp. 23–30.

[13] *Ansible website*. URL: https://www.ansible.com/.

[14] *YAML website*. URL: http://yaml.org/.

[15] *Google Protocol Buffers website*. URL: https://developers.google.com/protocol-buffers/.

[16] *Django project website*. URL: https://www.djangoproject.com/.

[17] *Jinja2 website*. URL: http://jinja.pocoo.org/.

[18] *TOSCA website*. URL: https://www.oasis-open.org/committees/tosca/.

[19] *GNU m4 website*. URL: https://www.gnu.org/software/m4/m4.html.

[20] *YANG Tools repository*. URL: https://github.com/opennetworkinglab/onos-yang-tools.

[21] *ANTLR website*. URL: http://www.antlr.org/.

[22] *OpenConfig website*. URL: http://openconfig.net/.

[23] *Example Driver repository*. URL: https://github.com/netgroup-polito/onos/tree/netconf_driver.

[24] *Driver/App Tiesse repository*. URL: https://github.com/netgroup-polito/onos/tree/netconf_app_tiesse.

[25] *CORD 3.0 repository*. URL: https://github.com/opencord/cord/tree/cord-3.0.

[26] *Vagrant website*. URL: https://www.vagrantup.com/.

[27] *ONOS repository*. URL: https://github.com/opennetworkinglab/onos.

[28] *Buck website*. URL: https://buckbuild.com/.