



POLITECNICO DI TORINO
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Orchestrazione di funzioni di sicurezza in ambito NFV

Relatori

prof. Antonio Lioy
dott. Fulvio Valenza

Candidato

Ignazio PEDONE

ANNO ACCADEMICO 2017-2018

Indice

1	Introduzione	1
2	Network Function Virtualisation (NFV)	4
2.1	La Virtualizzazione	4
2.2	Introduzione ad NFV	5
2.2.1	Nascita ed evoluzione di NFV	6
2.2.2	Vantaggi di NFV rispetto alle soluzioni tradizionali	7
2.2.3	Relazione di NFV con i concetti di Cloud ed SDN	8
2.3	Architettura NFV	11
2.3.1	Virtual Network Function (VNF)	12
2.3.2	Network Service e Network Forwarding Graph	14
2.3.3	Data Model e descrittori	16
2.3.4	Requisiti del paradigma NFV	17
2.3.5	Network Function Virtualisation Infrastructure (NFVI)	18
2.4	Sicurezza in NFV	21
2.4.1	Punti di forza e vulnerabilità	21
2.4.2	Analisi dei rischi	22
2.4.3	Best Practices	25
2.5	Sfide future per NFV	27
2.5.1	Gestione ed Orchestrazione	28
2.5.2	Efficienza Energetica	28
2.5.3	Prestazioni	29
2.5.4	Allocazione delle risorse	30
2.5.5	Sicurezza	30
2.5.6	Modellazione delle risorse, delle funzioni e dei servizi	31

3	Architettura NFV MANO	33
3.1	Funzionalità	33
3.2	Architettura	33
3.2.1	NFV Orchestrator	34
3.2.2	VNF Manager	35
3.2.3	Virtual Infrastructure Manager	35
3.2.4	Element Management System	37
3.2.5	Repository	37
3.3	Progetti NFV MANO	37
3.3.1	Open Source MANO	38
3.3.2	Open Baton	44
3.3.3	OPNFV	46
3.3.4	NFV over Open DC/OS	47
4	Policy di sicurezza	49
4.1	Il concetto di Policy di sicurezza	49
4.1.1	Introduzione alla terminologia	49
4.2	Policy-based management (PBM)	50
4.2.1	Architettura	51
4.2.2	Modellazione delle policy	51
4.2.3	Policy Management Tool	54
4.3	PBM nel progetto SECURED	55
4.3.1	Introduzione al progetto SECURED	55
4.3.2	Modellazione delle risorse di sicurezza	56
4.3.3	Linguaggi: definizione	61
4.3.4	Policy Manager e Workflow	66
4.3.5	Modello geometrico e Policy Analysis	69
5	Obiettivi e metodi	73
5.1	Definizione degli obiettivi	73
5.2	Analisi degli strumenti	75
5.3	Design e Implementazione	75
5.4	Test e Sviluppi futuri	75
6	Analisi degli strumenti	77
6.1	OpenStack	77
6.1.1	Architettura base	78
6.1.2	Servizi aggiuntivi	81
6.2	Open Source MANO	84
6.2.1	Workflow	84

6.2.2	Descrittori	88
6.2.3	Configurazione delle VNF	93
6.2.4	Cloud-Init	94
6.3	Juju	96
6.3.1	Terminologia e definizioni	96
6.3.2	Charm	99
6.3.3	Architettura	101
6.3.4	Proxy charm	102
6.4	Light Virtualization	106
6.4.1	LXD	107
6.4.2	Docker	108
7	Design della soluzione	111
7.1	Design di alto livello	111
7.1.1	Architettura complessiva	111
7.1.2	Security Service Manager	112
7.1.3	Security Site	113
7.2	Policy Services	114
7.2.1	Service discovery e authentication	114
7.3	Security Site	115
7.3.1	Architettura	115
7.3.2	Progettazione della rete	116
7.4	Progettazione della vNSF	117
7.4.1	Architettura della vNSF	117
7.4.2	Configurazione della vNSF	118
7.5	Progettazione del Network Security Service	120
7.6	Security Service Controller	120
7.7	Security Service Manager: controllo degli accessi	122
7.8	Workflow e logica della soluzione	122
7.8.1	Lato utente	123
7.8.2	Lato service provider	123
8	Implementazione e Testing	126
8.1	Proof-of-Concept generale	126
8.2	Security Service Manager	127
8.3	Esempio reingegnerizzazione moduli SECURED: UPR	128
8.4	Infrastructure e VIM	129
8.5	Network Security Service	130
8.5.1	Mapping dei virtual link di OSM su OpenStack	130
8.6	Sviluppo di una vNSF	130

8.6.1	Packet Filter	131
8.6.2	Variante per ottimizzare le risorse	131
8.6.3	Configurazioni con Ansible	132
8.7	Testing	132
8.7.1	Tecnologie utilizzate e scenari di test	133
8.7.2	Cloud-image vs Monolithic Image	134
8.7.3	Scenario cloud-image mono-site o multi-site	136
8.7.4	Scenario con l'introduzione di Juju e calcolo del suo overhead	137
9	Sviluppi futuri	142
9.1	OSM vs container	142
9.1.1	VIM EMU	143
9.1.2	Prestazioni a confronto	145
9.2	Forwarding graph e SDN in OSM	147
9.2.1	Richiami al Virtual Network Functions Forwarding Graph in OSM	147
9.2.2	Senario di esempio multi PoP	148
9.3	Ottimizzazione delle prestazioni: SR-IOV, DPDK ed EPA	149
10	Conclusioni	152
	Bibliografia	154
A	Manuale utente: soluzione proposta	157
A.1	NSS Deployer	157
A.2	Dashboard Open Source MANO	158
A.3	Dashboard OpenStack	162
A.4	Dashboard Juju	164
B	Manuale sviluppatore: Open Source MANO	166
B.1	Installazione e configurazione Open Source MANO	166
B.1.1	Installazione dipendenze OSM	166
B.1.2	Installazione dipendenze OSM: opzionale	167
B.1.3	Installazione OSM (Release THREE)	170
B.2	Descrizione UI e CLI di OSM	171
B.2.1	User Interface (UI)	171
B.2.2	Command Line Interface (CLI)	174
B.3	Operazioni di base con OSM	177
B.3.1	Integrazione di un VIM	177
B.3.2	Onboarding dei descrittori	177
B.3.3	Deploy di un servizio di rete	178
B.4	Esempi di NSD e VNFD	179

B.4.1	VNF Descriptor di una vNSF	179
B.4.2	NS Descriptor di un NSS	184
B.5	OSM vs Docker: vim-emu	187
B.5.1	Operazioni preliminari con vim-emu	188
B.5.2	Possibili estensioni alla nostra soluzione	189
B.5.3	Strumenti di supporto: son-emu	190
B.6	Juju	193
B.6.1	UI	193
B.6.2	CLI	198
B.6.3	Juju & OSM: Proxy Charm	201
B.7	Operazioni utili per sviluppi futuri	205
B.7.1	Integrazione SDN Controller (SDN assist/EPA)	205
B.7.2	Ansible charm	207
B.7.3	RO extention	207
C	Manuale sviluppatore: User Policy Repository (UPR)	208
C.1	Installazione	208
C.1.1	Introduzione e creazione container LXD	208
C.1.2	Installazione UPR	209
C.1.3	Documentazione ed utilizzo	211
D	Manuale sviluppatore: OpenStack	212
D.1	Installazione con DevStack	212
D.1.1	Installazione	212
D.1.2	Utilizzo e configurazione	213
D.2	Integrazione con OpenSource MANO	214

Capitolo 1

Introduzione

Gli ultimi anni sono stati teatro di numerosi cyber attacchi, con implicazioni considerevoli talvolta legate ad eventi nel mondo reale e alle infrastrutture, talvolta con risvolti direttamente connessi alla politica. Un esempio può essere ciò che è accaduto nel dicembre del 2015 in Ucraina, dove centinaia di migliaia di persone sono rimaste senza elettricità per diverse ore nel bel mezzo dell'inverno a causa di un malware. Ancora la famosa vicenda dei "Panama Papers", dove nell'aprile del 2016 sono stati sottratti 11.5 milioni di documenti allo studio legale Mossack Fonseca, esponendo conti offshore appartenenti a migliaia di persone di rilievo in tutto il mondo. Le implicazioni politiche di quest'ultima vicenda, considerando il coinvolgimento di numerosi politici e leader di stato mondiali, sospettati di evasione fiscale, sono state di proporzioni smisurate. Basti pensare che quest'ultimo è stato un argomento centrale nelle elezioni americane del 2016.

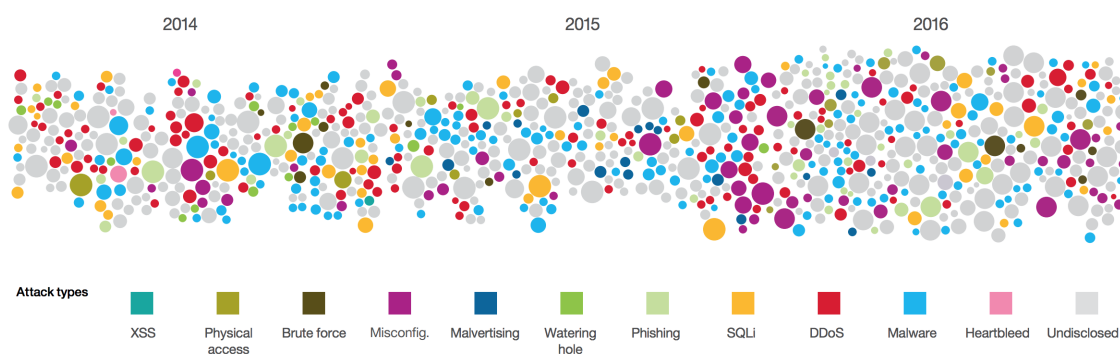


Figura 1.1: Attacchi informatici dal 2014 al 2016, catalogati per data, tipologia e impatto (fonte: [1]).

Come riportato in figura 1.1, nel triennio 2014-2016 gli attacchi informatici hanno subito un'evoluzione, legata al numero e all'impatto di tali attacchi in termini di costi per le società coinvolte e la quantità di dati sottratti. Tali attacchi coinvolgono in molti casi gli utenti, basti pensare al caso di Yahoo nel Dicembre del 2016, quando la compagnia annunciò delle falle di sicurezza e la relativa sottrazione di dati relativi a miliardi di account a partire dal 2013 (se ne contano 3 miliardi ad oggi).

Queste problematiche illustrano in modo particolare come possa essere devastante l'impatto di un cyber attacco e come sia sempre più rilevante il problema della sicurezza informatica. Esiste però un'altra prospettiva da considerare ovvero quella dell'utente.

Il singolo utente, oggi, si trova ad utilizzare un numero esponenzialmente crescente di dispositivi intelligenti, tra smartphone, smartwatch, smart TV e dispositivi per l'IoT. Ognuno di questi è connesso, per poter scambiare dati con altri dispositivi ed assumere così il suo comportamento

“intelligente”. Un bracciale in grado di tener traccia dei parametri vitali e di comunicarli a servizi di emergenza in caso di pericolo, elettrodomestici in grado di automatizzare, comunicando tra loro, le faccende domestiche quotidiane; questi sono tutti scenari che vanno a dipingere il nostro futuro e il modo con cui ci rapportiamo alla tecnologia.

Scenari del genere prevedono che l’utente condivida e quindi metta in rete un quantità di dati personali consistente. All’aumentare della quantità di dati e delle connessioni dei dispositivi, però, aumenta anche la superficie di attacco e le possibili vulnerabilità presenti nei sistemi utilizzati. Questo si traduce, in termini di sicurezza, in una necessità sempre più capillare di sistemi che siano aggiornati e ben configurati, per prevenire eventuali fughe di dati sensibili. Spesso il singolo utente, però, non è in grado o non è dotato dei sistemi necessari per poter sopperire a tale esigenza. Una possibile soluzione a questo problema si basa sull’utilizzo del paradigma *SECaaS* (*Security-as-a-Service*). Esso può fornire dei servizi di sicurezza personalizzati, sempre aggiornati e con tempi di messa in opera trascurabili rispetto alle attuali soluzioni di sicurezza.

Questo paradigma prevede l’utilizzo di funzioni virtuali di sicurezza (vNSF), che forniscono delle specifiche funzionalità di sicurezza (firewall, reverse proxy, VPN, etc.). Tali funzioni possono essere correlate tra loro e garantire così dei servizi di sicurezza più complessi. Tutto questo può essere ospitato dall’infrastruttura del Service Provider, ad esempio sull’ultimo nodo di rete prima del terminale dell’utente stesso. Questo consentirebbe all’utente di richiedere un servizio personalizzato sfruttando un approccio policy-driven e al Service Provider di offrire il servizio in modo virtuale con la relativa ottimizzazione di costi e prestazioni.

Il principale obiettivo di questa tesi è di sviluppare un approccio SECaaS attraverso l’utilizzo del paradigma *Network Function Virtualisation* (NFV), cercando di ottimizzare l’utilizzo di risorse e delineando anche quali potranno essere le possibili evoluzioni future. Il paradigma NFV sta riscuotendo un grande successo tra i Service Provider; operatori come Telefonica, Vodafone, AT&T e molti altri ancora stanno testando soluzioni legate a questa nuova tecnologia, che permette, attraverso l’utilizzo di un’infrastruttura virtuale, di sfruttare a pieno le risorse fisiche e diminuire considerevolmente i costi di gestione, manutenzione e aggiornamento. I vantaggi sono considerevoli e fanno di NFV una probabile “killer technology” per il prossimo futuro.

Per valutare la fattibilità e la bontà delle scelte effettuate e per provare che il sistema complessivo sia valido e semplice da estendere in futuro, è stato sviluppato un *Proof-of-Concept* (PoC). Tale implementazione è basata sulle ultime tecnologie NFV disponibili, che sono lo *standard de facto* per la realizzazione di infrastrutture NFV e sistemi per la gestione e l’orchestrazione di funzioni di rete virtuali.

Il risultato finale converge in un sistema NFV MANO (NFV Management & Orchestration), guidato da policy definite dall’utente, che è in grado, in modo del tutto automatico, di inizializzare e configurare tutte le funzioni virtuali di sicurezza necessarie a garantire la sicurezza del singolo utente.

Questo lavoro si divide nei seguenti capitoli e nelle relative argomentazioni:

- **Capitolo 2:** presentazione del paradigma NFV (cap. 2).
- **Capitolo 3:** presentazione degli strumenti di gestione ed orchestrazione in NFV (cap. 3).
- **Capitolo 4:** presentazione del management basato su policy (cap. 4).
- **Capitolo 5:** presentazione degli obiettivi, contributi e criticità di questo lavoro (cap. 5).
- **Capitolo 6:** analisi degli strumenti utilizzati per l’implementazione della soluzione (cap. 6).
- **Capitolo 7:** design della soluzione (cap. 7).
- **Capitolo 8:** implementazione e testing della soluzione (cap. 8).
- **Capitolo 9:** sviluppi futuri (cap. 9).
- **Appendice A:** manuale utente della soluzione proposta (cap. A).

- **Appendice B:** manuale dello sviluppatore di Open Source MANO (app. [B](#)).
- **Appendice C:** manuale dello sviluppatore dello User Policy Repository (app. [C](#)).
- **Appendice D:** manuale dello sviluppatore di OpenStack (app. [D](#)).

Capitolo 2

Network Function Virtualisation (NFV)

In questo capitolo verrà presentato il paradigma di *Network Function Virtualisation (NFV)* e tutti i concetti ad esso correlati.

2.1 La Virtualizzazione

Nell'ultima decade il concetto di Virtualizzazione è diventato centrale per molti settori dell'IT (Information Technology) e continua a crescere rapidamente grazie anche alla spinta tecnologica proveniente dalla produzione di hardware sempre più performante. In sostanza essa può essere definita come la creazione di una rappresentazione virtuale (basata tipicamente sul software) di una qualsiasi risorsa come dispositivi di rete, supporti di memorizzazione, CPU e sistemi operativi. Il principale vantaggio di questo approccio è un notevole risparmio in termini di costi per l'IT e una maggiore agilità ed efficienza nella gestione delle risorse hardware.

Mettendo da parte i recenti sviluppi, i primi esempi di utilizzo di tecniche di virtualizzazione risalgono alla fine degli anni '60, quando la IBM sviluppò il CP/CMS, un sistema operativo time-sharing, ovvero capace di condividere le risorse di calcolo a disposizione, dividendo l'attività della CPU in intervalli temporali da assegnare sequenzialmente a vari processi di uno o più utenti. In questo caso specifico il sistema era multi-utente e la CPU centrale veniva utilizzata per rispondere alle richieste dei singoli utenti, passando rapidamente da uno all'altro e dando così l'impressione agli stessi di avere a disposizione un computer centrale interamente per sé. Sembra chiaro come, l'idea di disaccoppiare il comportamento delle risorse hardware e software così come viste dall'utente dalla loro realizzazione fisica, sia da molti anni un punto fondamentale nel mondo dei sistemi di elaborazione delle informazioni.

Durante gli anni successivi, la virtualizzazione, in relazione soprattutto alla diffusione dei Personal Computer (PC) al decorrere degli anni '80, venne trascurata e messa da parte, infatti, pochi sono gli esempi di utilizzo di questa tecnica soprattutto in ambito commerciale. Due tra i maggiori prodotti riconosciuti in questo periodo furono Simultask e Merger/386, entrambi sviluppati dalla Locus Computing Corporation, i quali avevano l'obiettivo di eseguire MS-DOS come sistema operativo guest. Un altro esempio anche se limitato si ha nel 1988, quando Insignia Solutions realizzò Soft PC, che era in grado di eseguire DOS sulle piattaforme Sun e Macintosh.

Una volta trascorsi gli anni bui per la virtualizzazione, precisamente alla fine degli anni '90, abbiamo una nuova ondata di prodotti e soluzioni basate sulla virtualizzazione, basti pensare a Connectix Virtual PC per Macintosh, programma anch'esso in grado di ospitare sistemi operativi guest e distribuito originariamente come applicazione per System 7. Successivamente Connectix rilasciò una versione del suo software per Windows e pochi anni dopo la Microsoft acquistò la compagnia e continuò lo sviluppo delle moderne versioni di Virtual PC. Nel frattempo nel 1999

VMware fece il suo ingresso sul mercato con il suo primo prodotto VMware Workstation, un software che permetteva la creazione e l'esecuzione simultanea di più macchine virtuali sullo stessa macchina fisica. Contestualmente, sempre negli anni '90, alcune compagnie dell'ambito telecomunicazioni, che fino ad allora offrivano un punto unico di connessione dedicato, iniziarono ad offrire delle connessioni VPN (Virtualized Private Network), che con la stessa qualità di servizio di quelle dedicate avevano un costo inferiore. Questo consentiva, a dispetto del costruire delle infrastrutture fisiche per la connessione di più utenti, la possibilità di condividere il medesimo accesso all'infrastruttura fisica. Questo scenario permetteva alle compagnie di dirigere opportunamente il traffico, di introdurre reti migliori e avere più controllo sull'utilizzo della banda.

Dopo qualche anno, con l'avvento del nuovo millennio, la crescita delle tecniche di virtualizzazione ha ricevuto un impulso significativo dalla diffusione di internet. Il passaggio logico, a questo punto, consisteva nel portare la virtualizzazione "on-line". Per permettere tale passaggio è stato necessario creare risorse virtuali a partire dai server fisici a disposizione, richiedendo storicamente due fasi. La prima, giustificata dai costi elevati dei server, consisteva nella tendenza a far coesistere più applicazioni sul medesimo server. Questo implicava l'andare a virtualizzare risorse del server fisico, allo scopo di dividerle tra i vari applicativi. Successivamente, invece, con l'abbattimento dei costi dell'hardware e l'aumento della richiesta di servizi, si è passati alla necessità di combinare tra loro più server "economici", per sopperire al bisogno in termini di risorse da parte degli applicativi. Questo secondo passaggio ha posto le basi per la nascita del *cloud computing* (sez. 2.2.3).

Attualmente, tra le varie declinazioni della virtualizzazione, ne individuiamo per i nostri scopi tre di particolare interesse: una è il Cloud Computing, la seconda il Software-Defined Networking (sez. 2.2.3) e la terza la Network Function Virtualisation. Di seguito andremo a dettagliare quest'ultima per gli scopi di questo lavoro di tesi.

2.2 Introduzione ad NFV

Il paradigma di Network Function Virtualisation (NFV) ha guadagnato un ruolo decisivo nella progettazione delle moderne architetture di rete. Tale ruolo è sintomo di una particolare attenzione, da parte di NFV, alle caratteristiche di flessibilità e ottimizzazione di costi e prestazioni, cruciali in uno scenario come quello attuale. Scenario dove gli Internet Service Provider (ISP) si trovano ad affrontare sfide sempre più pressanti; in primo luogo nel rispondere ad esigenze sempre più spinte di servizi da parte dei clienti; in secondo luogo nel dover sopperire a tali esigenze senza un aumento proporzionale dei profitti. La Network Function Virtualisation si pone l'obiettivo, sfruttando l'evoluzione delle tecnologie di virtualizzazione presenti nell'ambito IT, di trasformare il modo in cui oggi gli operatori progettano le reti. Il fine ultimo è quello di avere a disposizione una versione virtuale dei dispositivi di rete attualmente presenti sul mercato, così da poterli allocare su infrastrutture fisiche standard su larga scala (server, switches e dispositivi di memorizzazione in fig. 2.1). Per raggiungere tale obiettivo è necessario implementare le funzioni di rete (Virtual Network Function o VNF (sez. 2.3.1)) via software, un software compatibile con questa nuova infrastruttura e sufficientemente scalabile. Le funzioni di rete, così create, devono poter essere spostate ed istanziate in diverse locazioni della rete a seconda delle esigenze e senza la necessità di installare dei nuovi dispositivi fisici.

La Network Function Virtualisation, ha riscosso notevole successo, sia nel mondo dell'industria che in quello accademico, come importante passo avanti rispetto al modo con cui le compagnie di telecomunicazione offrivano i propri servizi. Disaccoppiando le funzioni di rete dai dispositivi fisici dove vengono eseguite, NFV ha il potenziale per ridurre in modo significativo le spese operative (OPEX) e le spese per l'acquisto di risorse (CAPEX). A questo si aggiunge la semplificazione delle procedure per la distribuzione dei servizi e il contestuale risparmio di tempo per la messa in opera. A partire da queste premesse, è utile far presente che il paradigma NFV si trova ancora nel suo stato embrionale, ciò significa che potenzialmente è in grado di offrire un gran numero di opportunità per la ricerca, sia in termini di sviluppo di nuove tecnologie, sia in termini di ottimizzazione di quelle esistenti. Non avendo poi raggiunto la completa standardizzazione, è il momento adatto per valutare quali potrebbero essere, da una parte le soluzioni migliori dal punto di vista tecnologico, dall'altra eventuali modifiche da apportare alla sua architettura per renderla più efficiente e sicura.

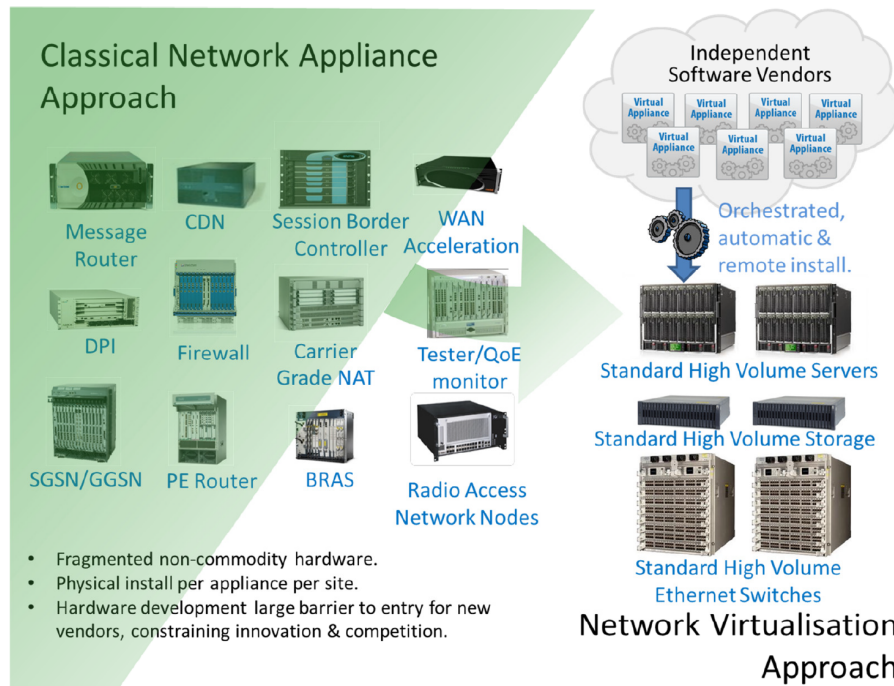


Figura 2.1: Approccio di NFV, passaggio da dispositivi fisici specifici a commodity server.

2.2.1 Nascita ed evoluzione di NFV

Il concetto di NFV nasce a partire dall'ETSI (The European Telecommunications Standards Institute), ovvero l'istituto di standardizzazione europeo in ambito telecomunicazioni. Quest'ultimo ha creato un Industry Specification Group (ISG), ovvero un gruppo di lavoro che conta aziende e università, con l'obiettivo di realizzare un'architettura comune di NFV e delle specifiche tecniche per ogni componente della stessa. Tra le aziende che partecipano all'ISG sono presenti i nomi di alcune tra le maggiori compagnie di telecomunicazione a livello mondiale: AT&T, BT, China Mobile, Deutsche Telekom, Orange, Telefónica e Verizon. A partire dall'ottobre 2012, quando ha preso vita l'iniziativa, in soli dodici mesi più di 150 aziende, tra compagnie di telecomunicazioni, operatori di rete e fornitori di dispositivi, vi hanno preso parte.

Dopo soli due anni l'ETSI contava 245 compagnie individuali implicate nel progetto, di cui 37 tra i maggiori service provider del mondo e aveva completato la prima fase del suo lavoro, pubblicando undici specifiche tecniche su NFV. Queste specifiche tecniche costituivano la prima documentazione ETSI rilasciata su NFV e tale pubblicazione risale all'ottobre del 2013. I temi trattati da queste specifiche comprendevano una panoramica sull'infrastruttura NFV, l'architettura complessiva del framework, la descrizione di come doveva avvenire la computazione, il ruolo dell'hypervisor e la distinzione dei vari domini all'interno dell'infrastruttura. Era inoltre presente una sezione relativa all'orchestrazione e alla gestione delle funzioni di rete, per noi di particolare interesse. Tali specifiche verranno presentate e dettagliate nella sezione 2.3, sottoforma di descrizione dell'architettura NFV.

Ad oggi l'ETSI continua il suo percorso di standardizzazione e contestualmente porta avanti un progetto con diverse compagnie tra cui Telefónica, progetto teso all'implementazione di uno stack software per la gestione ed orchestrazione delle funzioni virtuali di rete (Open Source MANO). Tale progetto oltre ad essere open source raccoglie l'obiettivo, comune a molte aziende di telecomunicazioni, di allineare le proprie infrastrutture allo standard proposto dall'ETSI, implementando un'architettura fortemente modulare ed integrabile, che cerca di sopperire alle richieste del nascente mercato delle reti NFV.

Utile alla consultazione dello standard risulta notare che l'attuale gruppo ETSI NFV ISG ha diversi sottogruppi di lavoro. Di seguito andiamo ad elencarli e a descriverne la funzione:

- *Software Architecture (SWA)*: descrizione delle funzioni di rete virtuali.

- *Management and Orchestration (MAN)*: Descrizione del sistema di gestione ed orchestrazione delle funzioni di rete.
- *Performance and Portability (PER)*: normative per ottenere buone performance e portabilità.
- *Infrastructure (INF)*: descrizione dell'infrastruttura NFVI.
- *Testing and Open Source (TST)*: report sui test effettuati per la validazione.
- *Solutions (SOL)*: si occupano dei protocolli di comunicazione e del data model.
- *Reliability and Availability (REL)*: documentazione sulla resistenza ai guasti e alla disponibilità del servizio.
- *Interface and Architecture (IFA)*: descrizione delle interfacce tra i vari moduli architetturali.
- *Evolution and Ecosystem (EVE)*: si occupa degli aspetti più avanzati, legati soprattutto all'evoluzione di NFV.
- *Security (SEC)*: si occupa della sicurezza di NFV.

Oltre all'ETSI, sia il 3GPP (Third Generation Partnership Group) che l'IETF (Internet Engineering Task Force), sono stati coinvolti attivamente in NFV. Il primo con il gruppo di lavoro SA5, che ha fatto come oggetto di studio la gestione delle funzioni di rete virtuali, con l'obiettivo primario di investigare che impatto avesse l'architettura proposta dall'ETSI sul proprio modello di riferimento; il secondo con il gruppo di lavoro SFC (Service Function Chaining), che voleva investigare come guidare il traffico attraverso le funzioni di rete virtualizzate.

2.2.2 Vantaggi di NFV rispetto alle soluzioni tradizionali

Tipicamente la fornitura dei servizi di rete da parte delle compagnie di telecomunicazione è basata sull'installazione di dispositivi e accessori fisici (aventi hardware e software specializzati), per ogni singola funzione di rete. In aggiunta, abbiamo l'esigenza di coordinare tali funzioni di rete a formare un servizio più complesso e di gestire e configurare ognuna di queste. Ciò avviene, nel caso fisico, attraverso un amministratore che, manualmente, va a configurare ogni singola funzione e a preoccuparsi della gestione della connettività di rete a livello sottostante. Un tale dispendio di tempo e risorse si traduce in una scarsa flessibilità del servizio, basti pensare che per aggiungere una singola funzione di rete va installato nuovo hardware nell'infrastruttura e bisogna procedere con la riconfigurazione manuale. Gli aspetti negativi di questo *modus operandi* sono dunque in sintesi: la poca elasticità e scalabilità del servizio, i tempi necessari per instaurarne di nuovi e la grande dipendenza dall'hardware specializzato.

La poca dinamicità nella fornitura dei servizi da parte delle aziende di telecomunicazione non si sposa molto bene con la richiesta sempre più ingente di servizi da parte degli utenti, servizi che sono anche molto spesso di breve durata. Con le attuali tecnologie sopperire a tale domanda si traduce in pratica con la necessità di un continuo acquisto e messa in opera di nuove infrastrutture, oltre che in una richiesta ai tecnici di un continuo e rapido aggiornamento delle loro competenze, aumentando di conseguenza sia le OPEX che le CAPEX per i Service Provider. Ovviamente i costi derivanti da tutte queste operazioni non possono ricadere sugli utenti finali, per ragioni di concorrenza, quindi sta ai Service Provider andare ad ottimizzare i costi e le prestazioni delle loro architetture di rete al fine di non andare in perdita.

NFV propone ai Service Provider una maggiore flessibilità e agilità riguardo alle loro capacità di lanciare e gestire servizi di rete, il tutto introducendo le seguenti differenze rispetto al passato:

- *Disaccoppiamento del software dall'hardware*: questo permette ai dispositivi di rete di non essere più una composizione di hardware e software integrati e quindi di separare lo sviluppo di queste due parti. Il concetto è quello di mantenere un'infrastruttura hardware standard, su cui far girare uno strato software di virtualizzazione (nel maggior parte dei casi un *Hypervisor*). Una volta virtualizzate le risorse hardware, è possibile eseguire del software generico, che nel nostro caso va a coincidere con le funzioni di rete, abbandonando così il concetto di avere per ogni dispositivo hardware specializzato una funzione di rete.

- *Dispiegamento di funzioni di rete flessibili:* separare hardware e software permette di ri-assegnare e condividere le risorse dell'infrastruttura. Hardware e software, quindi, possono svolgere funzioni differenti in tempi diversi. Questo semplifica il lavoro degli operatori di rete nel dispiegare servizi e gli consente di farlo in modo più veloce e sulla stessa infrastruttura fisica esistente. Risulta possibile, inoltre, istanziare componenti su ogni nodo della rete, compatibile con l'architettura NFV, e le connessioni possono essere configurate in modo dinamico.
- *Scalamento dinamico:* utilizzare funzioni di rete, implementate via software, consente di avere una maggiore flessibilità nello scalare le performance e questo con una granularità maggiore. In sostanza quello che viene fatto è allocare più o meno risorse, compatibilmente con le prestazioni richieste dalla VNF stessa. Una puntualizzazione sta nel fatto che anche nello scenario precedente con dispositivi fisici è possibile scalare le funzioni di rete, ma in quel caso avviene in modo più complesso richiedendo della movimentazione nell'infrastruttura fisica. Infine, è anche possibile avere degli scenari ibridi in cui coesistano sia le funzioni di rete tradizionali che virtuali.

2.2.3 Relazione di NFV con i concetti di Cloud ed SDN

Il concetto di NFV è legato anche ad altri paradigmi del modo delle telecomunicazioni, il cui obiettivo è quello di aumentare l'agilità e la flessibilità della fornitura di servizi. Tra quelli fondamentali troviamo il Cloud Computing e il Software-defined networking. Tutti questi concetti, hanno in comune tra loro, come pilastro fondamentale, la virtualizzazione, intesa come astrazione di differenti tipi di risorse: di computazione nel caso del cloud computing, di rete nel caso di SDN e di funzioni di rete nel caso di NFV. Di seguito andremo ad illustrare gli obiettivi di tali tecnologie e le relazioni che intercorrono con NFV (fig. 2.2).

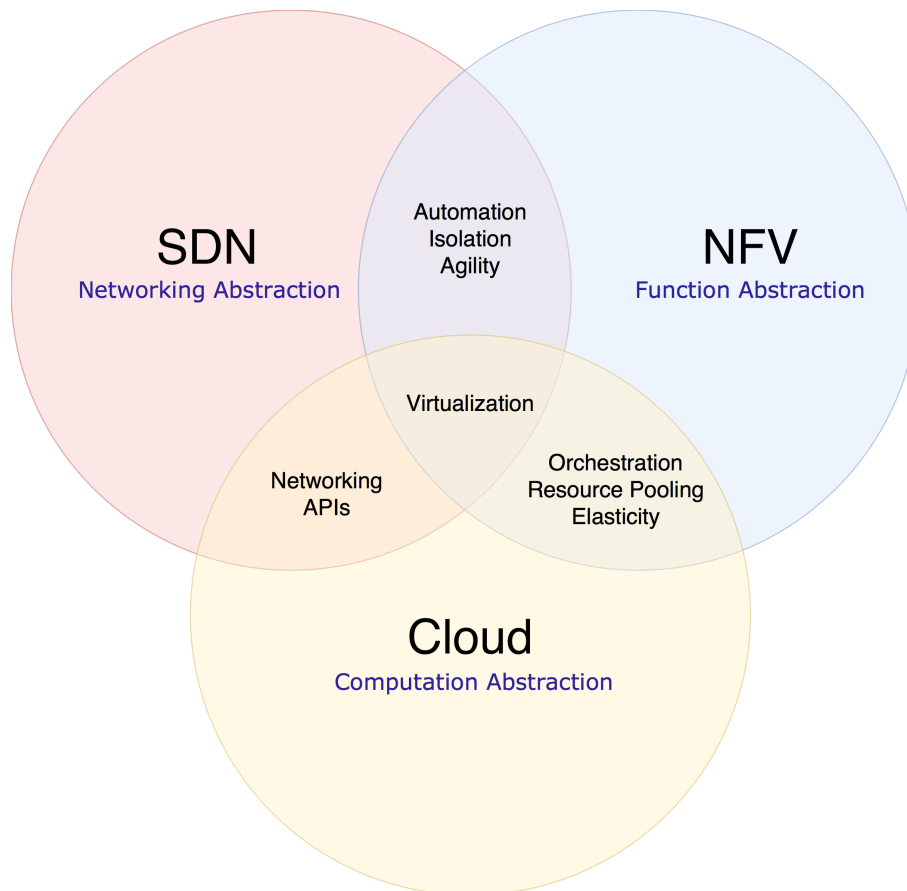


Figura 2.2: Punti in comune tra NFV, SDN e Cloud.

Cloud Computing

Secondo la definizione del National Institute of Standards and Technology (NIST) [8], “*Il Cloud Computing è un modello per abilitare, tramite la rete, l’accesso diffuso, agevole e a richiesta, ad un insieme condiviso e configurabile di risorse di elaborazione (ad esempio reti, server, memoria, applicazioni e servizi) che possono essere acquisite e rilasciate rapidamente e con minimo sforzo di gestione o di interazione con il fornitore di servizi*”. Il cloud computing è basato su cinque caratteristiche fondamentali, di seguito riportate:

- *Self-service su richiesta*: il Service Provider deve fornire risorse di calcolo, di rete o di storage su richiesta a un cliente in modo automatico, senza necessità di intervento umano.
- *Ampio accesso alla rete*: le risorse devono essere disponibili in rete e devono essere accessibili tramite meccanismi standard che promuovono l’utilizzo di piattaforme eterogenee per l’accesso, sia client leggeri che pesanti.
- *Condivisione delle risorse*: i fornitori condividono le risorse al fine di servire più clienti, utilizzando un modello multi-tenant, dove sia le risorse fisiche che virtuali sono assegnate dinamicamente in risposta alla domanda da parte dei clienti.
- *Elasticità rapida*: le risorse possono essere acquisite e rilasciate in modo elastico, in molti casi automaticamente, scalando rapidamente per allinearsi alla domanda.
- *Servizio misurato*: i sistemi Cloud controllano in modo automatico le risorse e le ottimizzano, utilizzando delle metriche ad un livello di astrazione adatto al tipo di servizio (ad es. memoria, computazione, larghezza di banda e utenti attivi).

Sono presenti inoltre tre modelli diversi di servizio:

- *Software as a Service (SaaS)*: gli utenti utilizzano le applicazioni del fornitore in esecuzione su di un’infrastruttura cloud.
- *Platform as a Service (PaaS)*: gli utenti possono istanziare sull’infrastruttura cloud del fornitore delle applicazioni compatibili con la stessa in termini di linguaggi di programmazione, librerie, servizi e strumenti utilizzati.
- *Infrastructure as Service (IaaS)*: gli utenti possono reperire risorse di computazione (ad es. memoria, elaborazione e rete), da poter utilizzare per istanziare arbitrariamente del software, che può includere sistemi operativi e applicazioni.

Una volta viste le caratteristiche e definizioni di cloud computing, andiamo a capire quali sono i punti in comune e le differenze con il concetto di Network Function Virtualisation.

NFV parte da un concetto molto simile a quello di cloud computing, infatti, entrambi i paradigmi hanno come obiettivo comune l’esecuzione di alcuni applicativi (o servizi) su server standard su larga scala. Ancora l’ottimizzazione delle risorse utilizzate, la necessità di una forte scalabilità e l’utilizzo di sistemi di orchestrazione trovano dei punti in comune tra i due.

Questo potrebbe indurre a pensare di poter utilizzare strumenti identici per l’implementazione dei due paradigmi, ma non è così; infatti ci sono anche delle profonde differenze tra questi, vediamo di seguito alcune:

- *Lo Scopo*: nel caso di NFV abbiamo l’esigenza di astrarre funzioni di rete, che hanno uno scopo particolare e dei requisiti specifici. Rispetto all’astrarre risorse per servizi generici, come nel caso del cloud, c’è una significativa differenza. Si può vedere, in un certo senso, NFV come un caso molto particolare di SaaS.
- *Le Prestazioni*: il data plane, nelle caso delle funzioni di rete virtuali, viene stressato in termini di prestazioni, rispetto al caso del cloud.

- *L'orientamento alle reti*: l'astrazione delle risorse di rete è di fondamentale importanza e necessita di considerazioni più ampie nel caso NFV.
- *La scalabilità e affidabilità del servizio*: oltre a richiedere scalabilità come nel caso del cloud, NFV necessita di un disponibilità e affidabilità di gran lunga superiore, si parla di *carrier-grade*, quindi ben oltre la disponibilità five-nines (del 99,999 %).

Queste sono le ragioni per cui le tecnologie cloud non sono direttamente applicabili a NFV, anche se i loro aspetti in comune rendono tali tecnologie un ottimo punto di partenza. Ad esempio molti degli strumenti di orchestrazione ed infrastrutturali citati e utilizzati nel presente lavoro provengono dal cloud computing, uno tra tutti è OpenStack.

Software-defined networking (SDN)

Il Software-defined networking è un paradigma che sta riscontrando un notevole successo nell'ambito delle telecomunicazioni. Il suo obiettivo principale è quello di disaccoppiare il *control plane*, ovvero tutta la parte dedicata alla gestione delle regole di inoltro dei pacchetti, e il *data plane* ovvero l'effettivo flusso di tali pacchetti attraverso la rete. Disaccoppiando queste due entità delle attuali reti è possibile avere un'estrema flessibilità nella gestione delle topologie di rete ed inoltre prevedere dei meccanismi automatici di configurazione, ad esempio basati su policy di alto livello.

Valutiamo ora quali sono i principi fondamentali di tale paradigma:

- *Programmabilità*: questo rispecchia il principio di disaccoppiamento visto in precedenza, ovvero avere la possibilità di programmare i dispositivi in modo semplice eventualmente anche automatico.
- *Agilità*: flessibilità nella gestione del traffico che permette agli amministratori di cambiare repentinamente i piani di inoltro a seconda delle esigenze.
- *Gestione centralizzata*: un solo controller ha la capacità di gestire una topologia complessa e gestire tutti gli *switch virtuali* (*vSwitch*) al fine di ottenere un comportamento desiderato del flusso di pacchetti.
- *Basata su standard liberi*: risulta importante essere svincolati da protocolli proprietari in modo da poter gestire dispositivi di fornitori diversi con il medesimo controller.

Da tali principi viene definita l'architettura SDN illustrata ad alto livello in figura 2.3, dove si evince che sono presenti sostanzialmente tre strati diversi:

- *Application layer*: questo primo livello è costituito da tutte le componenti applicative che servono a gestire il control plane ad un livello d'astrazione molto alto, questo livello ha poi la possibilità di interfacciarsi con il livello sottostante attraverso le API messe a disposizione dal protocollo SDN utilizzato ad esempio *OpenFlow*.
- *Control layer*: questo livello è rappresentato dall' SDN Controller, anch'esso dipendente ovviamente dalla tipologia di protocollo utilizzato, uno dei più quotati come già ribadito è OpenFlow. Il controller è il punto centrale dell'architettura SDN e comunica attraverso due interfacce, la *northbound interface* verso l'application Layer e la *southbound interface* verso l'infrastructure layer; rispettivamente una con il livello superiore per ottenere i comandi da uno o più applicativi e l'altra con il livello inferiore per innestare le regole all'interno dei dispositivi (ad es. vSwitch).
- *Infrastruction layer*: questo livello contiene tutti i dispositivi che vengono programmati attraverso il controller, tipicamente degli vSwitch, sui quali sarà presente un *agent*, che può dipendere dal particolare hardware o meno e quindi essere eseguito su dispositivi standard (come nel caso di *Open vSwitch*). Questi agent ricevono i comandi dal controller, comandi che in sostanza non sono altro che delle regole di inoltro da applicare al dataplane, e li applicano alle tabelle dei vSwitch.

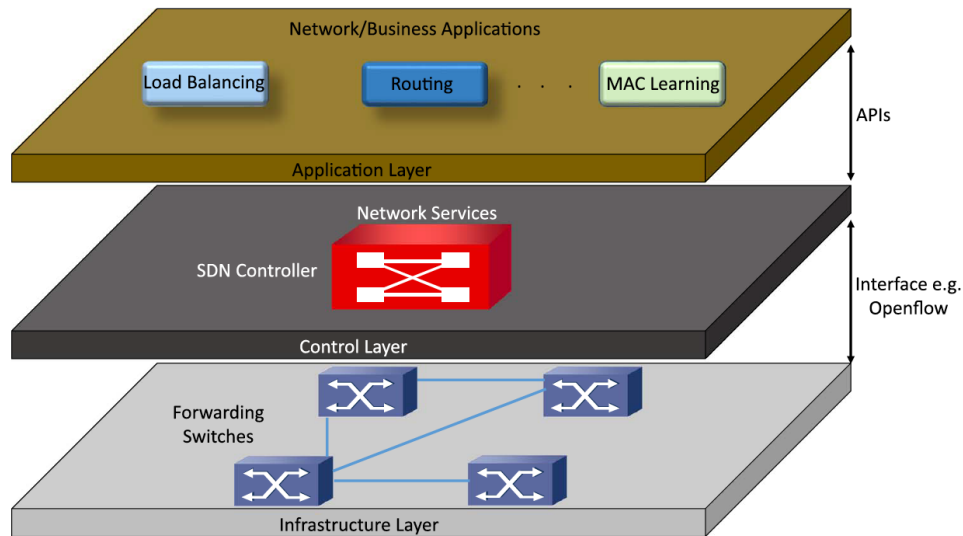


Figura 2.3: Architettura SDN (fonte: [semantic scholar](#)).

Abbiamo delineato unicamente le caratteristiche fondamentali di SDN, senza perderci in dettagli tecnici, ai quali faremo, anche se non in modo completo, riferimento nei capitoli successivi. Quello che però ci interessa particolarmente è valutare qual'è il rapporto che questo paradigma ha con NFV.

Per questo aspetto esistono due forme secondo cui le due tecnologie si influenzano a vicenda. La prima consiste nel beneficio che potrebbe trarre la tecnologia SDN se il controller fosse implementato come una funzione di rete, come parte quindi della catena di servizio (sez. 2.3.2). Questo sarebbe un netto vantaggio, in quanto, il controller godrebbe di tutti i benefici in termini di elasticità e scalabilità tipici di una VNF.

Allo stesso modo sarebbe utile disporre di SDN per gestire il dataplane delle VNF, ovvero coadiuvare le VNF della catena con un meccanismo secondo il quale la topologia di rete venga gestita in modo dinamico da un SDN controller, quindi con tutti i vantaggi derivanti dal disaccoppiamento (data/control plane) precedentemente descritto. Nei successivi capitoli soprattutto quando tratteremo delle architetture NFV reali e dell'implementazione proposta ci concentreremo anche su questo secondo aspetto, in quanto SDN risulta un notevole vantaggio per la gestione del control plane.

2.3 Architettura NFV

In questa sezione verranno presentati i requisiti e le caratteristiche fondamentali di NFV in accordo con lo standard ETSI presentato in precedenza. Questi verranno poi declinati nell'architettura che viene schematizzata in figura 2.4 e descritta nel corso della sezione.

L'architettura, innanzitutto, si basa su tre elementi fondamentali:

- *Descrittori e Data Model*: questo rappresenta l'insieme dei dati necessari per la caratterizzazione degli elementi base del mondo NFV: funzioni e servizi di rete.
- *Network Function Virtualisation Infrastructure (NFVI)*: rappresenta l'infrastruttura sia fisica che logica messa a disposizione per l'esecuzione delle funzioni di rete (sez. 2.3.5).
- *NFV Management and Orchestration (NFV MANO)*: questo elemento racchiude i moduli che hanno la funzione di gestione ed orchestrazione delle funzioni di rete e sarà discusso in dettaglio nel capitolo 3.

essa è composta da moduli software, che vengono poi eseguiti in contenitori messi a disposizione dallo strato di virtualizzazione, e da interfacce, sia con le altre VNF che con l'orchestrazione.

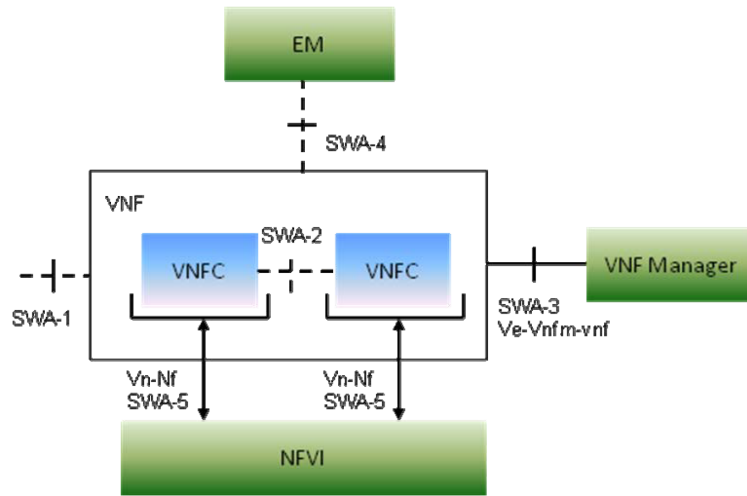


Figura 2.5: Architettura di una Virtual Network Function (fonte: [3]).

Per quanto riguarda i moduli software, bisogna tener conto del fatto che i fornitori di VNF possono progettare la stessa con una struttura monolitica o dividere le varie parti in moduli più semplici chiamati *VNF Component (VNFC)*. Tutto questo avviene in funzione di una serie di fattori relativi alle performance, alla scalabilità, alla disponibilità del servizio e alla sicurezza. Una volta sviluppati questi moduli, gli stessi devono essere incapsulati in una o più immagini software (es. qcow2, iso, etc.), le quali saranno verosimilmente eseguite attraverso i *Virtualization Container*, rappresentati in 2.6. Questi sono un'astrazione logica di un contenitore software isolato, messo a disposizione dalla sottostante infrastruttura NFVI e in grado di ospitare ed eseguire VNF, il tutto prescindendo dalla tecnologia utilizzata per la virtualizzazione.

Passando alle interfacce, invece, possiamo individuarne di diversi tipi con diverse funzionalità che andiamo a dettagliare di seguito:

- *SWA-1*: questa è un'astrazione delle interfacce di comunicazione inter-VNF, quindi garantisce la comunicazione tra la VNF stessa e le altre, al fine di ottenere un servizio di rete più complesso.
- *SWA-2*: si riferisce alle interfacce interne, ovvero dedite alla comunicazione tra i vari VNFC. Queste interfacce sono definite in modo specifico dai fornitori delle funzioni di rete e tipicamente non sono visibili agli utilizzatori delle VNF. Per tali interfacce vengono utilizzati i meccanismi di comunicazione messi a disposizione dal livello sottostante (NFVI), ma esistono altri metodi di comunicazione, che possono essere sfruttati per migliorare le prestazioni in particolar modo in termini di latenza, ad esempio comunicazioni basate su canale o meccanismi a memoria condivisa. In quest'ultimo caso con meccanismi di condivisione della memoria potrebbero esserci problemi di sicurezza, sono da utilizzare, quindi, solo in contesti controllati.
- *SWA-3*: rappresenta l'interfaccia di comunicazione con il VNF Manager e con l'orchestratore, viene quindi utilizzata per la parte di *management* delle VNF.
- *SWA-4*: quest'ultima interfaccia è utilizzata per la comunicazione con l'*Element Management* e quindi la gestione a tempo di esecuzione della VNF.
- *SWA-5*: rappresenta un'astrazione dell'insieme di interfacce che vengono messe a disposizione per la comunicazione tra ogni singola VNF e l'Infrastruttura sottostante (NFVI). Queste interfacce sono varie e dipendono in modo considerevole dalla tipologia della VNF e dai servizi che richiede a livello sottostante, in termini di computazione, memoria e rete.

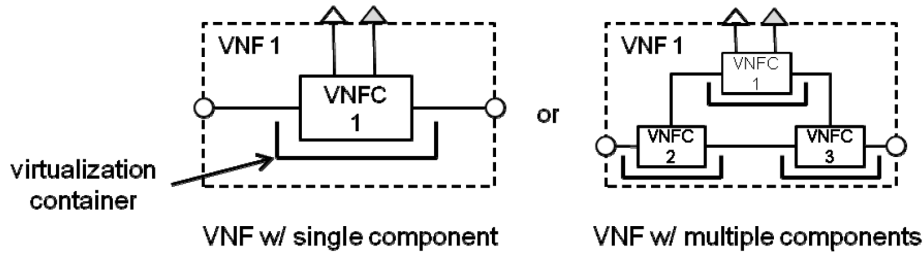


Figura 2.6: Esempi di Virtualization Container nel caso di un singolo componente o più componenti (fonte: [3]).

Laddove non è stato specificato il meccanismo di comunicazione, utilizzato da quella tipologia di interfaccia, si sottintenda che esso dipende dall'NFVI sottostante e che alcuni esempi e tecnologie verranno mostrate nei capitoli di implementazione (cap. 7 e 8).

2.3.2 Network Service e Network Forwarding Graph

Una volta definita la VNF, andiamo a valutare come essa possa essere collocata in un sistema più complesso, al fine di garantire un servizio di rete *end-to-end* (E2E) e a questo scopo introduciamo il concetto di Network Service (NS).

Un Network Service è composto da un insieme di VNF, atte a creare con la loro interazione un servizio di rete più complesso (es. *Deep Packet Inspection* (DPI) via software, *IPsec*, Virtual Private Network (VPN), etc.). Il punto di partenza è il concetto di *Service Function Chaining* (SFC), descritto dall'IETF [4] come uno strumento in grado di definire un insieme ordinato di funzioni di rete e di vincoli, da applicare al flusso di pacchetti come risultato del processo di *classificazione*. La classificazione è un metodo utilizzato per regolare il flusso dei pacchetti attraverso alcune predefinite funzioni di rete e viene eseguita da un componente chiamato *classificatore*. Come si evince dalla figura 2.7, eliminando i dettagli superflui, l'idea è quella di avere una serie di funzioni di rete (ad es. *firewall*, *antivirus*, *video optimizer* e *parental control*) e di regole di inoltro dei pacchetti, che permettano di attraversare unicamente le funzioni desiderate. Un sottoinsieme di queste funzioni va a costituire quella che viene chiamata "*catena*", mentre il flusso di pacchetti, attraverso le funzioni della catena, viene rappresentato sempre in figura 2.7 con delle linee blu e rosse marcate. Nell'esempio trattato, si individuano due catene, una composta da firewall e video optimizer (linea blu marcata), l'altra composta da firewall, antivirus e parental control (linea rossa marcata). L'inoltro dei pacchetti viene effettuato mediante il paradigma SDN (sez. 2.2.3), attraverso l'applicazione di regole ai vSwitch per mezzo dell'SDN controller, tale metodologia ci sarà utile come strumento per la gestione automatica del traffico di rete nello scenario futuro NFV che tratteremo nei capitoli degli sviluppi futuri (cap. 9). Il servizio di rete o Network service sarà quindi completamente definito da una o più catene e da un insieme di regole di instradamento dei pacchetti.

L'ETSI [2], invece, per la definizione di un NS, parla di *Virtual Network Function Forwarding Graph* (VNFFG), quest'ultimo è un grafo (come in figura 2.8), i cui nodi rappresentano le varie funzioni di rete e gli archi i loro collegamenti. Questi possono essere unidirezionali, bidirezionali, multicast e/o broadcast, e vanno a definire i vincoli di flusso per i pacchetti, che saranno poi gestiti da un controller (ad es. con SDN).

La figura 2.8 mostra come viene modellato un servizio end-to-end dallo standard ETSI. I nodi della parte superiore rappresentano le varie VNF e le linee tratteggiate i link logici tra le stesse. Le risorse logiche, in termini di VNF e link, vengono poi mappate con quelle fisiche nella parte bassa della figura 2.8, in modo del tutto trasparente al servizio di rete. Questo processo viene effettuato grazie allo strato di virtualizzazione e ad un orchestratore per l'ottimizzazione dell'allocazione. Il disaccoppiamento che ne deriva, tra risorse logiche e fisiche, è fondamentale per l'architettura NFV e permette di eseguire stesse istanze di funzioni di rete su differenti risorse fisiche, anche geograficamente dislocate in luoghi diversi, il tutto per migliorare le prestazioni e la disponibilità

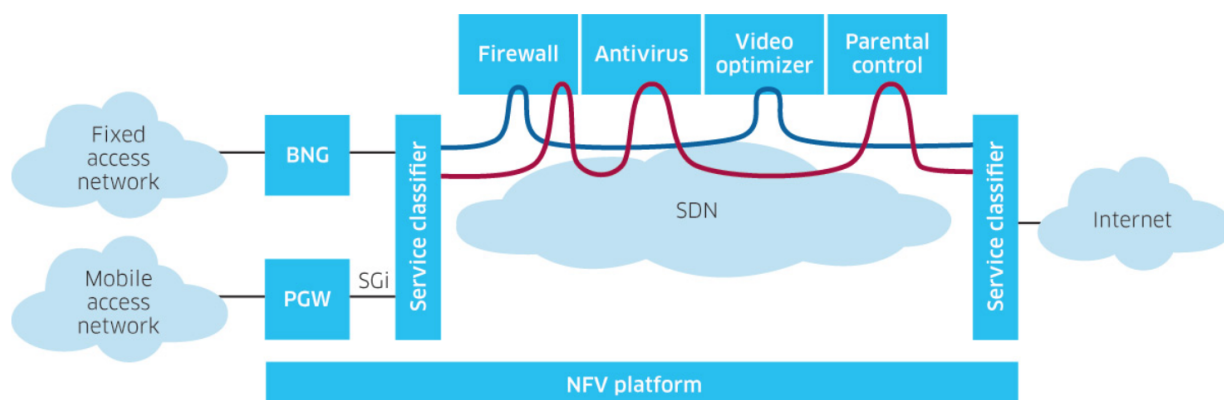


Figura 2.7: Esempio di Service Function Chaining (fonte: [sdx central](#)).

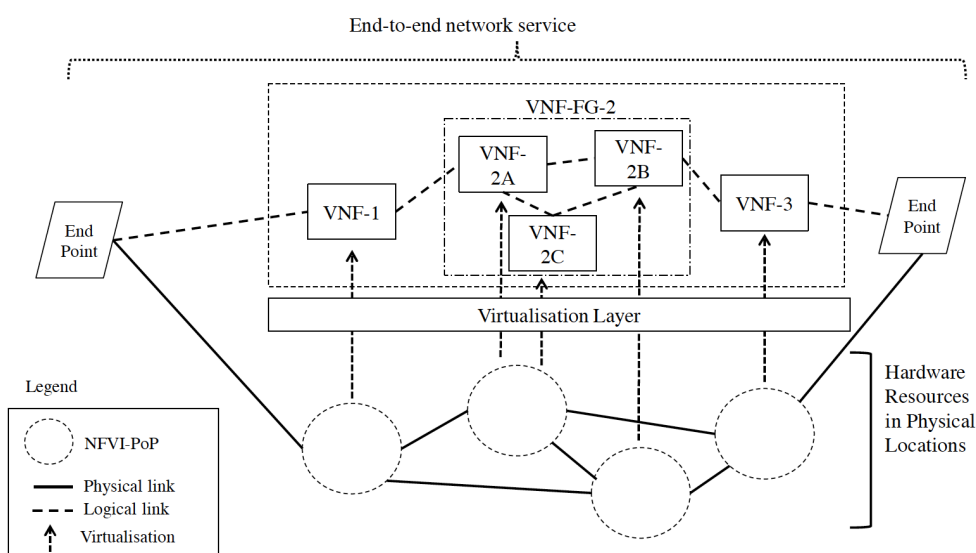


Figura 2.8: Esempio di Virtualisation Network Function Forwarding Graph o VNFFG secondo lo standard ETSI (fonte: [2]).

del servizio di rete. In ogni caso le risorse fisiche e l'infrastruttura di supporto devono essere visibili per ragioni di configurazione, monitoraggio e risoluzione di problematiche.

Il Virtual Network Function Forwarding Graph è molto simile al concetto di SFC, infatti, anch'esso prevede dei meccanismi atti a controllare l'inoltro dei pacchetti attraverso delle specifiche VNF, quindi il processo di classificazione e i classificatori. La differenza sta nel fatto che l'VNFFG può essere visto come una sorta di estensione del concetto di SFC, questo a partire dalla definizione di due strumenti di modellazione molto utilizzati in NFV:

- *Il Forwarding Service Path (FSP)*: questo permette di modellare un insieme ordinato di VNF e le relative interfacce che un flusso di pacchetti deve attraversare per seguire quello specifico percorso. Questo ricalca completamente il concetto di SFC, infatti si va a modellare la "catena" del servizio desiderato.
- *Il Classifier*: questo permette attraverso delle *regole di matching* prestabilite di scegliere quali tra i Forwarding Service Path attraversare, quindi in sostanza effettuare un controllo del flusso dei pacchetti attraverso un percorso piuttosto che un altro.

Un VNFFG può essere quindi visto in alternativa come un insieme di FSP e uno o più classifier che selezionano su quale dei percorsi inoltrare i pacchetti, tutto questo viene effettuato, come detto in precedenza, con l'utilizzo di tecnologie basate su SDN. Una volta definiti tali strumenti, diventa palese come il caso più semplice di un VNFFG è quello con un singolo FSP e quindi una singola "catena", in questo senso può essere visto come un'estensione dell'SFC. Come ultima nota a margine si evidenzia la possibilità di avere più grafi innestati.

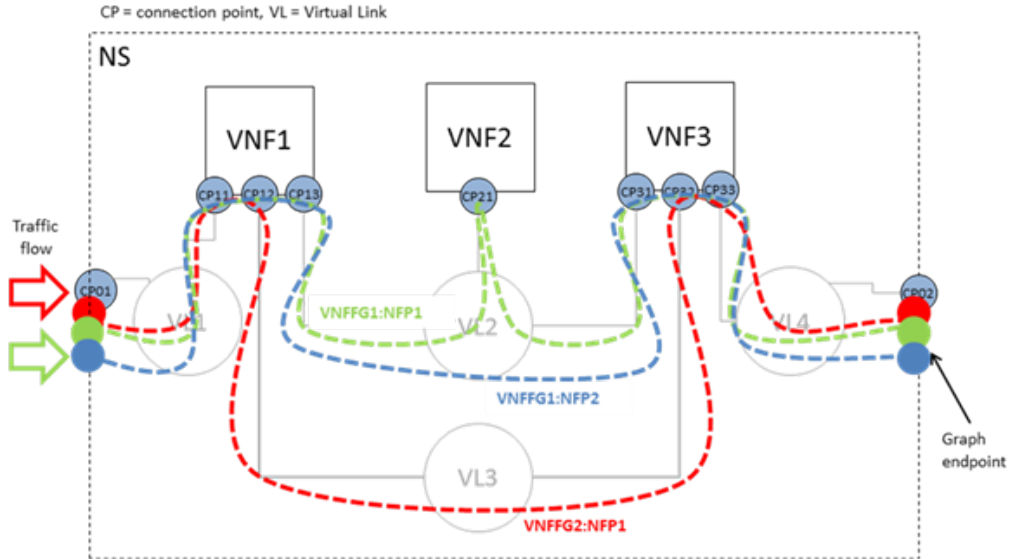


Figura 2.9: Esempio di un Virtual Network Function Forwarding Graph in un'architettura reale (fonte: [15]).

2.3.3 Data Model e descrittori

Continuiamo in questa sezione l'introduzione ai moduli essenziali dell'architettura NFV. In questo caso parliamo del data model, ovvero lo strumento di modellazione di un servizio di rete, in altri termini la descrizione di tutte le caratteristiche necessarie per il *deploy*, la configurazione e il monitoraggio delle VNF e del Network Service. Gli elementi costitutivi del modello di dati NFV sono i descrittori e nelle architetture reali ne troviamo essenzialmente di cinque tipi:

- *VNF Descriptor (VNFD)*: tale descrittore si occupa di caratterizzare i dettagli identificativi della funzione di rete (ad es. Nome, ID, Fornitore), i suoi punti di connessione e tutti i requisiti in termini di risorse richiesti per la sua esecuzione. Per quanto riguarda i punti di connessione abbiamo due tipi di interacce: interne ed esterne. Le prime sono quelle che permettono la comunicazione tra i vari VNFC, le altre quelle che si occupano di quella con le altre VNF o gli *end-point*. I requisiti della VNF corrispondono al solito principalmente a quelli di computazione (quante CPU allocare), alla memoria (quanta RAM o disco) o alla rete (creazione di sottoreti o altri elementi di rete utili). In realtà a seconda della VNF è possibile anche modellare come la stessa venga configurata, quindi stabilire quali siano eventuali script di configurazione o i VNF Manager da utilizzare. Risulta, inoltre, possibile utilizzare tecnologie specifiche messe a disposizione dall'infrastruttura, come ad esempio *Enhanced Platform Awareness (EPA)* di Intel, o altre come *SR-IOV*. Queste possono essere configurate attraverso il modello dati, così come è possibile fornire dei dettagli sulla scalabilità della funzione di rete e molti altri parametri che valuteremo di seguito nella fase implementativa (cap. 9). In ogni caso sembra chiaro come il descrittore definisca in modo chiaro e completo la VNF.
- *Virtual Link Descriptor (VLD)*: questo descrittore si occupa della descrizione dei collegamenti di rete tra le varie VNF, gli end-point ed eventualmente i componenti interni di una funzione di rete.

- *VNF Forwarding Graph Descriptor (VNFFGD)*: questo modella due strumenti descritti in precedenza, i Forwarding Service Path e i Classifier.
- *Physical Network Function Descriptor (PNFD)*: serve per descrivere eventuali funzioni di rete fisiche presenti nel servizio di rete, quindi attributi legati alla retrocompatibilità, ai requisiti di rete e alle performance.
- *NS Descriptor (NSD)*: questo è il descrittore che modella tutto il Network Service end-to-end in termini di aggregazione dei precedenti descrittori trattati. Delinea, inoltre, il modo in cui tutti questi elementi interagiscono tra di loro e definisce dettagli infrastrutturali, quali datacenter utilizzati per il deploy, informazioni aggiuntive sulla topologia di rete e altro ancora che approfondiremo, come sempre, andando ad analizzare i descrittori di in un'architettura reale nei capitoli di analisi dell'implementazione (cap. 6).

A partire dai descrittori elencati, l'obiettivo di NFV è quello di avere tutti i dati necessari, per istanziare, gestire e monitorare le VNF a disposizione. Questo è un obiettivo complesso, in quanto NFV nella sua implementazione reale è per sua natura *multi-site* (utilizza diversi datacenter ed infrastrutture). La diretta conseguenza è quella di avere dei descrittori estremamente ricchi di parametri e configurazioni, questo per poter tener conto di tutta una serie di differenze tecnologiche, derivanti dalla forte eterogeneità delle soluzioni esistenti. Ci si riferisce in questo caso alla grande quantità di sistemi di orchestrazione, di VNF Manager e controllori SDN esistenti e alla ovvia tendenza nel voler utilizzare tecnologie preesistenti per implementare l'architettura di NFV. Una situazione del genere porta a delle serie difficoltà nel riuscire a convergere ad un modello generico per descrivere ad alto livello sia le VNF che i NS. Molti sforzi sono in questa direzione soprattutto da parte dello standard ETSI, che dedica un intero working group a questo scopo, quello chiamato Solutions.

2.3.4 Requisiti del paradigma NFV

Risulta essere di vantaggio, prima della presentazione degli aspetti architetturali di NFV, rifarsi ai documenti dello ETSI NFV [5], relativi ai requisiti desiderati dalla virtualizzazione. Tali requisiti non includono aspetti quali la definizione di protocolli ed interfacce, ma vogliono analizzare quali sono le differenze introdotte da NFV rispetto al passato e valutare quali caratteristiche risultano desiderabili nella definizione di un architettura. Di seguito riportiamo quali sono i principali:

- *Portabilità*: la capacità di caricare ed eseguire le funzioni di rete in ambienti eterogenei, ovvero su datacenter differenti anche se popolati da dispositivi standard.
- *Prestazioni*: definire in modo completo quali sono le caratteristiche necessarie alla descrizione dei requisiti, in termini di performance per le funzioni di rete.
- *Gestione ed Orchestrazione*: definire i meccanismi secondo i quali viene gestito ed orchestrato il ciclo di vita delle VNF, delle risorse infrastrutturali e delle operazioni su di esse.
- *Elasticità*: provvedere a dei meccanismi per scalare (*scale up/down*) in modo semplice le risorse hardware in relazione al traffico.
- *Sicurezza*: necessità di analizzare quali possono essere gli eventuali attacchi all'infrastruttura NFV, che non sono previsti nelle precedenti architetture tradizionali.
- *Resilienza e stabilità delle reti*: assicurare la capacità di un servizio di rete nel resistere ai guasti e tornare alla normale operatività, limitando eventuali disservizi.
- *Continuità del servizio*: analizzare quali meccanismi permettano di assicurare due delle caratteristiche fondamentali di un servizio di rete, ovvero disponibilità e continuità, in conformità alle specifiche dello stesso e del Service Level Agreement (SLA).
- *Operazioni*: la capacità di automatizzare le funzioni operative, quali capacità della rete di adattarsi al carico, aggiornamenti software e interventi per guasti.

- *Efficienza energetica*: utilizzare tecniche per minimizzare il consumo di energia, in considerazione di uno scenario reale di server su vasta scala.
- *Migrazione e coesistenza con le piattaforme esistenti*: tener conto dell'attuale periodo di transizione degli scenari di rete, dove coesistono reti non-virtualizzate con reti virtualizzate e far sì che questo non abbia un impatto negativo sui servizi erogati all'utente.

2.3.5 Network Function Virtualisation Infrastructure (NFVI)

L' NFVI è il modulo che rappresenta l'infrastruttura di un ambiente NFV, ovvero fornisce tutte le risorse hardware e software per l'istanziamento delle funzioni di rete. In primo luogo fornisce l'hardware, quale diverse CPU, dispositivi di memorizzazione e di rete standard e in secondo luogo il software, da applicare a tale hardware per la virtualizzazione delle risorse (tipicamente un *Hypervisor*). L'infrastruttura NFVI è concepita per essere distribuita, quindi i nodi NFVI sono decentrati in varie locazioni per supportare i requisiti di località e latenza, necessari in taluni casi d'uso, questo aggiunge una notevole flessibilità all'architettura complessiva.

La figura 2.10 fornisce schematicamente qualche dettaglio sulla tipica struttura astratta di NFVI. In sostanza, essa è composta alla base da risorse fisiche, rappresentate da tutta una serie di dispositivi commerciali: server standard (ad es. CPU Intel, NIC, ect.), dispositivi di storage e switch per la gestione dei livelli L2-L3 di rete. Al di sopra dello strato fisico, è presente uno strato di virtualizzazione, costituito come vedremo dall'hypervisor e al di sopra ancora le risorse virtuali, che attraverso l'hypervisor sono mappate su quelle fisiche.

Per introdurre un po' di terminologia relativa allo standard ETSI, possiamo isolare in NFVI tre diversi domini [6]:

- *Compute Domain*: il seguente dominio è costituito dai moduli che in figura 2.10 sono chiamati *Compute Hardware* e *Storage Hardware*.
- *Hypervisor Domain*: il dominio dell'hypervisor è costituito dalla virtualizzazione del Compute Domain, quindi i moduli *Virtual Compute* e *Virtual Storage*, questo comprende anche parte dello strato di virtualizzazione.
- *Network Domain*: questo dominio comprende il *Network Hardware* e il *Virtual Networking*, risorse sia logiche che fisiche della parte di rete.

Per ulteriori dettagli e nomenclatura rimandiamo allo standard [6], dove viene descritta NFVI e vengono citati altri documenti dello standard utili ad approfondire problematiche e soluzioni relative alla stessa. Parte di queste problematiche le affronteremo nei capitoli successivi di implementazione (cap. 6), andando a confrontarci con un'infrastruttura reale. Nel frattempo, però, diamo un'idea di quali possono essere le scelte fondamentali da effettuare quando si vuole progettare un'infrastruttura NFV.

Quando vogliamo implementare un'infrastruttura NFV dobbiamo compiere due scelte fondamentali: come implementare il *networking* e quale tecnologia utilizzare per virtualizzare le funzioni di rete. La prima scelta riguardante il Network Domain, di solito ricade su tecnologie basate su linux, le principali sono i *linux bridge*, *macvlan* e *Open vSwitch (OVS)*. Esistono anche soluzioni proprietarie come *VMware vDS (Virtual Distributed Switch)*, ma noi ci concentreremo su quelle open e basate su linux. Una volta effettuata la scelta delle tecnologie per i vSwitch, va valutato come implementare il control plane e spesso si ricorre a tecnologie basate su SDN. Un esempio classico, che viene sfruttato da prodotti largamente utilizzati come OpenStack, è l'utilizzo di switch OVS, che vengono poi controllati attraverso il protocollo OpenFlow da un SDN controller (ad es. Floodlight). Questo è uno dei tanti modi per implementare il networking, ovviamente come sempre quando andremo a descrivere un'architettura reale vedremo qualche dettaglio in più, inoltre analizzeremo anche come queste tecnologie vengono integrate all'interno degli orchestratori (cap. 9).

La seconda scelta riguarda la tecnologia di virtualizzazione delle funzioni di rete, quindi in particolar modo l'Hypervisor Domain. Possiamo isolare due possibili scelte tecnologiche:

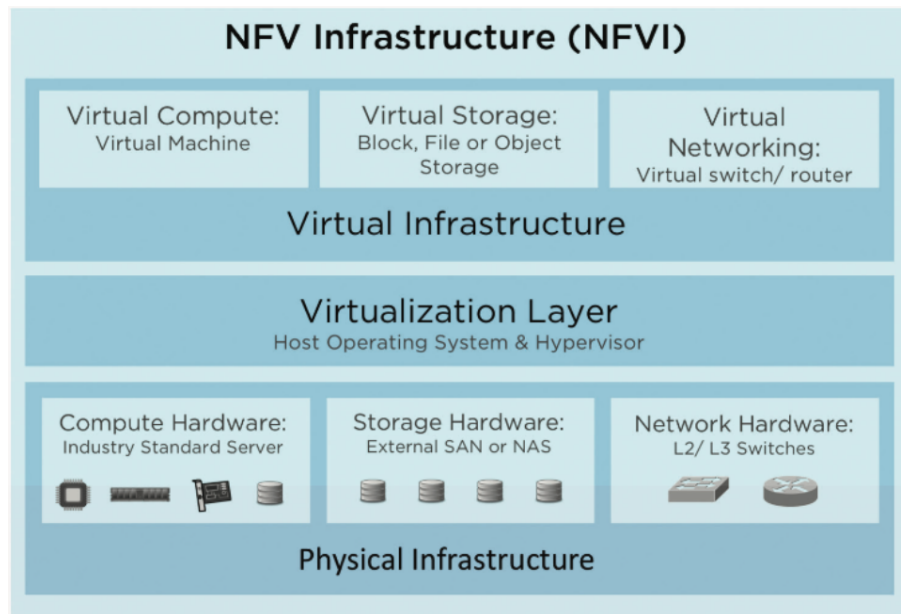


Figura 2.10: Struttura ad alto livello di una Network Function Virtualisation Infrastructure (fonte: [sdx central](#)).

- *Hypervisor (VM)*: questa tipologia di virtualizzazione prevede l'utilizzo di un hypervisor, ovvero un software installato sul sistema operativo (o direttamente sull'hardware, dipende dal tipo di virtualizzazione), che astrae le risorse fisiche e permette l'installazione di un nuovo sistema operativo *guest* su quello preesistente *host*. Molte CPU attuali, con architetture x86, x86-64 e ARMv7, rispettivamente da intel, AMD e ARM, supportano nel loro instruction set delle istruzioni speciali per assistere la virtualizzazione. Questo aiuta il completo isolamento tra il kernel del sistema operativo *guest* e quello dell'*host*. Il risultato di questo tipo di virtualizzazione è l'avere a disposizione una serie di macchine virtuali o *virtual machine (VM)*, che ospitano tipicamente una VNF.
- *Virtualizzazione leggera (Container)*: in questo caso la virtualizzazione avviene direttamente a livello di sistema operativo *host*. Sfruttando dei meccanismi messi a disposizione dal quest'ultimo, infatti, è possibile garantire l'isolamento dei vari applicativi che vengono eseguiti. Nel caso particolare del sistema operativo linux, sono messi a disposizione a livello di kernel due strumenti fondamentali, quali *cgroups* e *namespaces*. Questi consentono da un lato di limitare le risorse allocate ad un singolo processo (o un insieme) e dall'altro di limitare gli accessi del singolo processo solo alle risorse attribuite al namespace a cui appartiene. Ogni sistema operativo mette a disposizione dei meccanismi di isolamento, ad esempio esistono l'equivalente dei container per Windows, Solaris o macOS. Esiste altresì il concetto di *sandbox*, che è un altro meccanismo per ottenere risultati analoghi. Nel nostro caso ci concentreremo sul sistema operativo linux.

Una volta data una descrizione generale dei due macro insiemi di tecniche di virtualizzazione, andiamo a valutarne quali sono i vantaggi e gli svantaggi al fine di effettuare una scelta consapevole nella successiva implementazione. Ovviamente in questa fase tralasceremo le particolari scelte tecnologie che presenteremo, almeno in parte, successivamente (es. KVM, Docker, LXC, LXD, CoreOS) nei capitoli 3 e 6.

La virtualizzazione attraverso VM necessita di un tempo di istanziazione molto superiore rispetto ai container, tenendo conto della configurazione dell'hypervisor, del boot del sistema operativo *guest* e dell'allineamento delle dipendenze tra le varie VNF. Le prestazioni a tempo di esecuzione dipendono fortemente dalle risorse allocate per la VM e devono sempre tener conto del passaggio attraverso il sistema operativo *guest* e l'hypervisor per la comunicazione con le altre. Quest'ultimo aspetto aumenta in modo considerevole la latenza relativa all'I/O e di conseguenza risulta un collo

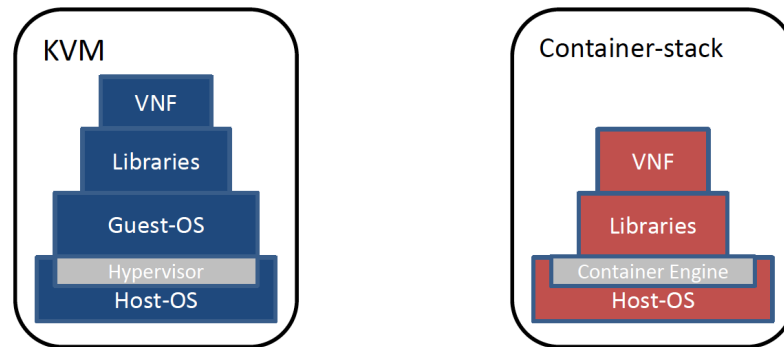


Figura 2.11: Differenze tra stack nel caso VM (KVM) e nel caso container (fonte: [linkedin](#)).

di bottiglia fondamentale, per un sistema che si propone di virtualizzare funzioni di rete. Una soluzione a tale problema consiste nell'utilizzo di nuove tecnologie come la tecnologia SR-IOV, che consente alla singola macchina virtuale di accedere al dispositivo fisico senza passare attraverso tutto lo strato di virtualizzazione. Dettagli relativi a questa tecnologia verranno presentati più avanti (cap. 3), ma in sostanza l'obiettivo è di avvicinarsi alle prestazioni di una macchina fisica in termini di latenza di I/O. Ovviamente data la mole di lavoro per instanziare una VM, anche la portabilità e la scalabilità della stessa VNF risultano fortemente compromesse. Per quanto riguarda la sicurezza, invece, la superficie di attacco nel caso delle VM risulta notevolmente minore rispetto a quella della virtualizzazione leggera, in quanto il livello di isolamento nelle VM, considerando lo strato di hypervisor e che ogni singola istanza non condivide il kernel del sistema operativo, risulta molto più elevato rispetto al caso della virtualizzazione leggera.

Nella virtualizzazione leggera, le prestazioni in termini di velocità di startup e di flessibilità rispetto alla precedente tecnologia sono nettamente superiori. Un container ha dei tempi di startup trascurabili rispetto ad una VM ed inoltre, non dipendendo direttamente dall'hardware, può essere virtualizzato a prescindere dall'architettura del processore. Questo è dovuto al fatto che gli applicativi vengono eseguiti direttamente sul sistema operativo host, senza passare per l'hypervisor ed un ulteriore sistema operativo guest. I vantaggi in termini di prestazioni e di consumo di energia sono considerevoli, inoltre le immagini software necessarie per l'avvio dei container sono nettamente più piccole (passiamo da alcuni GB per le VM a decine di MB per i container). L'aspetto negativo è relativo alla sicurezza, infatti, la condivisione del kernel del sistema operativo risulta, da una parte il più grande vantaggio in termini di prestazioni, dall'altra la sua più grande vulnerabilità, in quanto aumenta notevolmente la superficie d'attacco. Bisogna, inoltre, considerare che i namespace di linux sono una soluzione ancora relativamente immatura, e tutte le soluzioni che prevedono l'utilizzo di container (es. Docker, CoreOS, LXD, lxc, etc.) sfruttano questi meccanismi per l'isolamento dell'host. L'utilizzo dello stesso sistema operativo fa convergere sul kernel linux tutte le funzionalità di isolamento, compromettere quindi lo stesso ha come conseguenza la possibile compromissione di tutti gli applicativi sui container. Senza l'effort della CPU e dell'hypervisor per l'isolamento, tutta la sicurezza si sposta sul software, questo crea ulteriori implicazioni soprattutto se l'hardware è esposto direttamente (ad es. schede video o audio). Il kernel risulta quindi un *single point of failure* per le prestazioni, la stabilità e la sicurezza. Un altro aspetto da considerare è relativo alla questione orchestrazione, infatti, sono presenti moltissime tecnologie (es. Rancher, MESOS/Aurora-Marathon, Docker Swarm, LXD, Openstack Magnum, Kubernetes/Borg, etc.), che si pongono l'obiettivo di coordinare il deploy, la gestione e il monitoraggio dei container. Ancora adesso, però, ci sono da considerare molti aspetti relativi alla sicurezza di queste tecnologie, rappresentando di fatto una sfida aperta per il futuro. Anche valutando tutti gli aspetti negativi, c'è da dire che i container sono una tecnologia estremamente promettente per la virtualizzazione delle VNF e di conseguenza molti sforzi, da parte degli esperti del settore, sono rivolti a tale tecnologia, come valida alternativa per sostituire o coadiuvare la virtualizzazione basata su hypervisor [7].

Adesso ci siamo limitati a dare una visione di superficie sulle tecnologie, i dettagli verranno presentati nei successivi capitoli implementativi.

2.4 Sicurezza in NFV

Come abbiamo visto nelle sezioni precedenti, NFV è un paradigma che presenta numerosi benefici in termini di costi, di ottimizzazione delle risorse e di scalabilità, questo ha però un prezzo da pagare in termini di sicurezza. Un paradigma basato sulla virtualizzazione delle funzioni di rete ha dei rischi, che sono dati dall'intersezione tra l'insieme di rischi della tradizionali funzioni fisiche di rete e quello delle problematiche relative allo strato di virtualizzazione.

In questa sezione il nostro obiettivo sarà, da una parte quello di andare ad analizzare quali sono le principali minacce relative all'architettura presentata nei precedenti capitoli, dall'altra andare a valutare alcune tecniche di mitigazione delle stesse.

Un'interessante considerazione riguarda la prospettiva secondo cui andremo ad affrontare il problema sicurezza: nello specifico uno dei modi è andare ad analizzare le minacce per ogni singolo componente dell'architettura, quindi dettagliare il problema della "sicurezza di NFV". Questa è una delle prospettive ed è l'argomento del presente capitolo.

Un'altra interessante prospettiva, invece, è quella valutare e garantire la sicurezza dell'utente del servizio di rete, ovvero di far sì che il servizio stesso di rete sia in grado di offrire dei meccanismi e degli strumenti per fare sicurezza. Questa prospettiva permetterebbe di sfruttare i mezzi, messi a disposizione da NFV, per implementare delle funzionalità di sicurezza attraverso delle VNF specializzate e quindi di garantire un servizio di sicurezza all'utente. Tale servizio può essere orchestrato e gestito in modo centralizzato, in accordo con il paradigma *Security as Service (SECaaS)*. Le tematiche relative a tale prospettiva verranno per il momento abbandonate e riprese più avanti quando parleremo del cuore del lavoro di tesi.

2.4.1 Punti di forza e vulnerabilità

La sicurezza di NFV è strettamente legata alla sottostante infrastruttura di telecomunicazione, per questo motivo ha due implicazioni fondamentali: una riguardo alla resilienza del sistema e l'altra alla qualità del servizio offerto. La prima è legata al fatto che la sicurezza è parte integrante degli strumenti atti a garantire la disponibilità del servizio, in quanto difesa contro alcune tipologie di malfunzionamenti indotti. Una compromissione di VNF o dell'hypervisor potrebbe avere conseguenze devastanti sull'infrastruttura e sulle capacità del sistema di allocare risorse in modo corretto. La seconda è relativa all'overhead introdotto dai meccanismi di sicurezza sul sistema complessivo e quindi concerne il possibile degrado di prestazioni introdotto dalla sicurezza. Questo dev'essere tenuto sotto stretto controllo, sempre in considerazione delle prestazioni "carrier-grade" richieste da NFV.

Le considerazioni sulla sicurezza non si fermano ovviamente alla sottostante infrastruttura, ma si estendono a tutti i componenti architetturali chiave di NFV. Un esempio è il *Virtual Infrastructure Manager (VIM)* (come vedremo nel capitolo 3 è uno strumento dedicato alla gestione dell'infrastruttura) e il suo elemento chiave, ovvero l'hypervisor. Quest'ultimo ha numerose implicazioni in termini di possibili attacchi, basti pensare che l'acquisizione del controllo della VM o del sistema operativo guest, da parte dell'attaccante, potrebbe dar luogo all'estrapolazione di dati sensibili o alla distruzione degli stessi.

Una caratteristica che abbiamo sottolineato di NFV è la sua capacità di istanziare funzioni di rete, in modo elastico e rapido, preservando una notevole flessibilità e scalabilità. Tutto questo può essere fatto anche in modo automatico e/o in risposta a determinati eventi, senza l'intervento di un amministratore di sistema. Queste caratteristiche, paradossalmente, espongono NFV a minacce significative, che concernono vulnerabilità nell'istanziamento, nella configurazione (o configurazione malevola) e nell'orchestrazione delle VNF. Ancora ne derivano minacce riguardo altri aspetti, come l' SDN Controller, che potrebbe una volta compromesso esporre il sistema ad un malevolo controllo di flusso dei pacchetti e numerosi attacchi correlati. Un esempio di attacco dovuto all'estrema flessibilità di NFV è il *DNS amplification attack*, che sarà presentato nella prossima sezione 2.4.2 di analisi dei rischi.

Risulta interessante una considerazione sulle VNF, le quali, in accordo allo standard ETSI, possono essere fornite da vari produttori e devono avere la possibilità di interoperare tra di loro.

Questo espone il sistema ad ulteriori falle di sicurezza per l'infrastruttura, da una parte dovute a possibili incompatibilità tra le VNF sconosciute al sistemista, dall'altra perché alcune di esse potrebbero essere infettate e portare con sé codice malevolo.

Una nota positiva che mostra come vi sia un punto di incontro tra le problematiche di sicurezza e le caratteristiche di flessibilità ed elasticità di questo paradigma, risiede nel fatto che per sua natura NFV risulta più efficace nel rispondere ad attacchi di tipo *Distributed Denial of Service (DDoS)*. Questo tipo di attacchi cerca di saturare le risorse a disposizione di un Service Provider, atte ad erogare un servizio specifico. NFV, in quanto capace di allocare velocemente nuove risorse per tale servizio, di istanziare firewall e *IDS* su richiesta, di bloccare o ridirigere il traffico malevolo, risulta particolarmente adatta a contrastare questo tipo di attacchi. Uno scenario tipico è illustrato in figura 2.12, dove andiamo ad analizzare un possibile attacco DDoS ad un'infrastruttura *Evolved Packet Core (EPC)* e al suo componente *Virtual Mobility Management Entity (vMME)*, nodo che si occupa della gestione della connettività LTE di un sistema mobile.

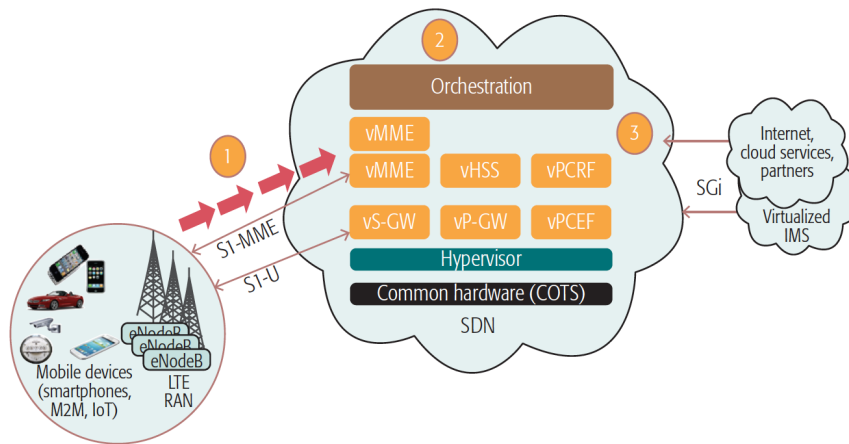


Figura 2.12: Scenario di DDoS attack in Evolved Packet Core (fonte: [9]).

Nel seguente scenario di attacco, abbiamo un sistema che presenta il nodo di management vMME virtualizzato e un orchestratore in grado di istanziare nuovi nodi dello stesso tipo, a seconda delle esigenze. L'attaccante potrebbe creare una *botnet*, una rete di nodi infetti utilizzata per scopi malevoli (tipicamente attacchi DDoS) e costituita da un nodo *master* e tanti nodi *slave*, il primo in grado di istruire gli slave su alcuni specifici task da eseguire. In questo caso la botnet sarebbe costituita da dispositivi mobile infetti da un "remote-reboot malware", un software in grado di eseguire il reboot del dispositivo, su comando del nodo master. L'attaccante, quindi, potrebbe istruire tutti i dispositivi nel riavviarsi contemporaneamente, questo genererebbe traffico malevolo (fig. 2.12 passo 1), in termini di un eccessivo numero di richieste di riavvio del servizio al vMME. Quest'ultimo potrebbe non essere in grado di soddisfarle tutte, come conseguenza del DDoS attack. In risposta a tale attacco l'orchestratore creerebbe nuove VM per scalare il servizio del vMME (fig. 2.12 passo 2), al fine di sostenere tale numero di richieste e andrebbe quindi a mitigare l'attacco, il tutto mentre viene monitorata ed investigata la natura di tale attacco (fig. 2.12 passo 3).

2.4.2 Analisi dei rischi

Esistono differenti rischi relativi alla sicurezza di NFV, la maggior parte legati all'infrastruttura NFVI, tra quelli principali analizzeremo, in accordo con l'articolo [9], i seguenti:

- problemi di isolamento;
- validazione della topologia di rete e problemi di implementazione;
- violazione di normative;

- elusione dei meccanismi di protezione degli attacchi *Denial of Service*;
- violazione dei file di log;
- amministratori malevoli.

Un primo rischio è dato dalla compromissione di una o più VNF o dell'Hypervisor stesso. Questo può creare ingenti rischi alla sicurezza complessiva del sistema, vediamo un esempio in figura 2.13 di una tipologia di attacco di questo genere, ovvero un *VM escape attack*.

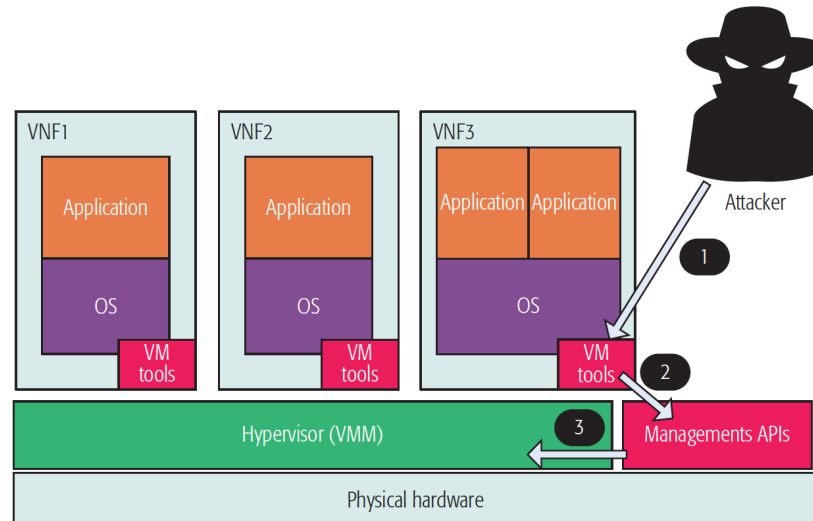


Figura 2.13: Esempio di VM Escape Attack (fonte: [9]).

L'attaccante dapprima può compromettere la VNF, guadagnando così l'accesso al sistema operativo (fig. 2.13 passo 1). Successivamente, attraverso l'uso di opportuni strumenti e la connessione alla rete di gestione della VNF, può essere in grado di accedere alle API di gestione dell'hypervisor (fig. 2.13 passo 2), questo porta all'ultimo passo ovvero all'accesso e al controllo dell'hypervisor stesso (fig. 2.13 passo 3). Questo attacco è reso possibile da un cattivo isolamento tra le VNF e l'hypervisor stesso.

Un esempio è dato da del codice malevolo eseguito su una VNF che è in grado di mandare opportuni pacchetti per raggiungere una condizione di *heap overflow* [10] (vulnerabilità VMware) con una conseguente compromissione dello strato di virtualizzazione, questo permetterebbe di eseguire del codice arbitrario sull'hypervisor e accedere di conseguenza all'host.

Un'altra possibilità di attacco è data da VNF che potrebbero richiedere la capacità di orchestrare altre VNF, per far questo è necessario fornire loro accesso alle API dello strato di virtualizzazione relative all'orchestrazione (in breve al VIM). Questo potrebbe permettere ad un attaccante di fare breccia in una VNF e guadagnare accesso allo strato di virtualizzazione sottostante.

Per quanto concerne il discorso relativo alle topologie di rete, si parte dal presupposto che con la virtualizzazione si ha la possibilità di creare velocemente dei dispositivi di rete e configurarli. La medesima cosa vale per dei router virtuali, i quali potrebbero essere per qualche errore umano, collegati direttamente alle funzioni di rete senza l'utilizzo di alcun firewall. Inoltre è possibile che un attaccante comprometta la VNF che ne ospita uno e limiti le restrizioni del traffico quanto basta, per garantirsi un sufficiente accesso per eseguire un attacco. Ancora è possibile che un attaccante estrapoli delle informazioni sensibili relative all'infrastruttura multi-site, ovvero fatta da più datacenter localizzati geograficamente in punti diversi. Questo potrebbe lasciar spazio alla migrazione di alcune macchine virtuali in punti meno sicuri e ancora dare possibilità di molteplici attacchi.

Veicolo di attacco possibile, sono anche le differenze tra le varie normative presenti in vari configurazioni geografiche e quindi lo sfruttare delle restrizioni territoriali per indurre ad esempio

un Service Provider ad eseguire dei servizi di rete laddove non consentito. Per meglio esemplificare tale situazione si osservi la figura 2.14, dove è presentato un attacco attraverso il quale alcune VNF vengono spostate da server statunitensi ad infrastrutture presenti sul territorio russo. Questo può ingenerare come già detto violazioni di accordi o normative che potrebbero portare all'applicazione di penali o alla soppressione del servizio.

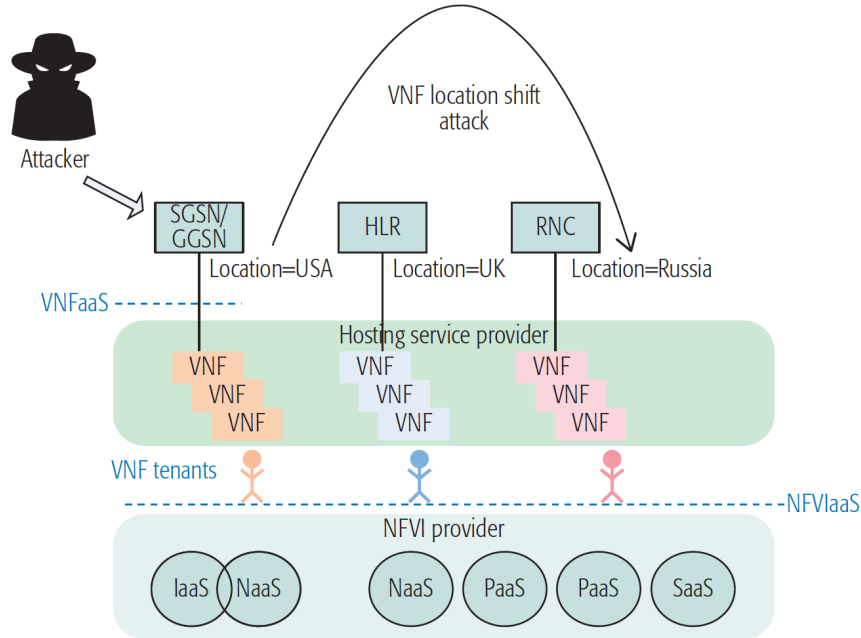


Figura 2.14: Esempio di attacco VNF location shift attack (fonte: [9]).

Per quanto riguarda gli attacchi *Denial of Service (DoS)*, esiste anche la possibilità che una VNF sia vittima di un attacco simile che ha come mira quella di andare ad esaurire le risorse in termini di rete, CPU e memoria messe a disposizione dall'hypervisor. Il concetto è quello di infettare una o più VNF, con del software malevolo, e ridirigere un enorme quantità di traffico su VNF che sono presenti sullo stesso hypervisor o su hypervisor diversi. Questo ovviamente dovrebbe essere mitigato, come prima accennato, dall'elasticità di NFV, ma in qualche caso tale caratteristica potrebbe essere sfruttata per amplificare l'attacco. Andiamo a presentare un esempio pratico in figura 2.15, dove sempre nello scenario di un EPC, scopriamo come è possibile infliggere un attacco di tipo DNS amplification attack.

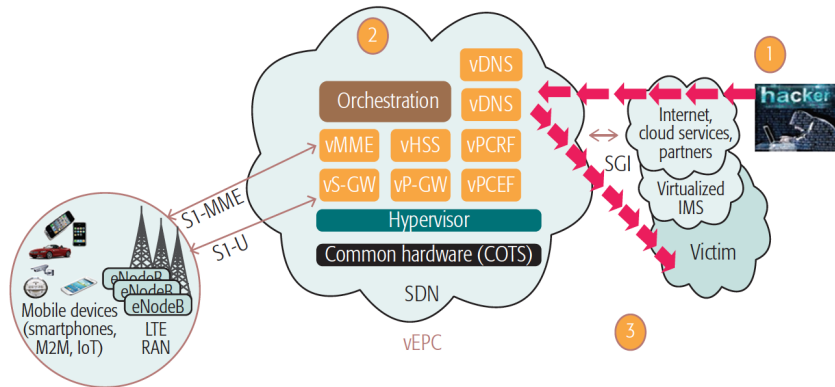


Figura 2.15: Esempio attacco DNS Amplification attack (fonte: [9]).

In questo scenario l'infrastruttura mette a disposizione dei DNS virtuali (vDNS), che sono in grado di sopperire ad un aumento del traffico di richieste DNS, meccanismo che potrebbe essere

sfruttato da un eventuale attaccante. Lo stesso, infatti, potrebbe falsificare l'indirizzo IP (IP spoofing) di un certo numero di vittime e successivamente lanciare un gran numero di richieste DNS (DNS query) utilizzando tali indirizzi (fig. 2.15 passo 1). A questo punto, dato l'ingente numero di richieste, il meccanismo di scalabilità dell'architettura impone l'istanziamento di altre macchine virtuali a supporto della VNF vDNS (fig. 2.15 passo 2), questo serve a sopperire alla crescita di traffico, ma allo stesso tempo aumenta notevolmente l'efficacia dell'attacco. Una volta messo in atto tale meccanismo, infatti, le query DNS avranno risposta, quindi le vittime riceveranno una quantità enorme di *DNS response* (fig. 2.15 passo 3), amplificati ulteriormente dal meccanismo di scalabilità NFV. Tutto questo sfocia ovviamente in un sovraccarico delle VNF bersaglio e una indisponibilità del servizio.

I file di log potrebbero essere un'ulteriore problematica durante un attacco. Questo è dovuto al fatto che le VNF compromesse potrebbero generare un enorme quantità di log e rendere difficile la loro interpretazione da parte dell'hypervisor (soprattutto quando le prime righe sono eliminate). Quando l'infrastruttura è compromessa, poi, la stessa potrebbe rendere i file di log accessibili da parte dell'attaccante e attraverso le correlazioni tra quelli di varie VNF (magari di vari fornitori), estrarre delle informazioni sensibili.

L'ultima problematica riguarda eventuali amministratori di sistema malevoli e quindi plausibilmente intenti ad accedere a informazioni sensibili degli utenti. Un amministratore di sistema ha potenzialmente accesso ai *dump* di memoria delle VM dell'utente, questo consente allo stesso di estrarre gli identificativi degli utenti, le password e le chiavi SSH per l'autenticazione, compromettendo in modo evidente la privacy dell'utente. Ancora un amministratore in uno scenario di attacco potrebbe estrarre i dati dell'utente dal volume del disco, gestito dal servizio di storage del cloud. In questo caso per eseguire tale attacco è possibile creare una copia di backup del drive della VM e utilizzare degli strumenti open source, quali *kpartx* e *vgscan*, per estrarre informazioni sensibili.

2.4.3 Best Practices

A partire dagli esempi dei possibili attacchi all'infrastruttura e agli altri componenti dell'architettura NFV, passiamo adesso ad elencare alcune delle tecniche di mitigazione e delle precauzioni da intraprendere per evitare che i suddetti attacchi vadano a buon fine. Vediamone alcune di seguito in accordo con [9]:

- *Boot Integrity Measurement*;
- mettere in sicurezza l'hypervisor e la rete virtuale;
- creare delle zone sicure attraverso la separazione del traffico;
- utilizzare i meccanismi di sicurezza del kernel linux;
- *hypervisor introspection*;
- criptare i dati sui volumi;
- firmare digitalmente le immagini software;
- sicurezza di NFV MANO;
- *remote attestation*.

Una prima difesa contro le minacce dei vari attaccanti consiste nell'accertare l'integrità del sistema sul quale vengono eseguite le VM e successivamente le VNF. Questo avviene attraverso diverse tecnologie ad esempio il *Trusted Platform Module (TPM)*, un coprocessore crittografico in grado di offrire funzionalità quali la generazione e la memorizzazione di chiavi crittografiche (RSA), generazione di numeri pseudo-casuali, cifratura e decifratura con RSA e generazione e verifica di hash (SHA-1 o SHA-256). Tale modulo può essere utilizzato per effettuare quello che in gergo viene

chiamato *boot integrity measurement*, ovvero, attraverso delle funzionalità del TPM (sostanzialmente generazione di hash, registri dedicati e una NV-RAM), è possibile effettuare misurazioni atte a validare l'integrità del sistema, questo a partire dal BIOS fino al sistema operativo utilizzato. Tali misurazioni possono essere effettuate solo quando il sistema è resettato o riavviato e le stesse vengono validate in accordo ad una politica di controllo (LCP) propria del TPM o attraverso una server di attestazione remota. Non scendiamo ulteriormente nei dettagli di tali tecnologie perché esula dagli scopi del presente lavoro, contestualmente però risulta di interesse per capire quali sono le tecnologie che vengono utilizzate e a che scopo.

Un ulteriore buona norma, per quanto riguarda la sicurezza dell'hypervisor e la connettività di rete, è quella di tenere in considerazione il fatto che l'hypervisor risulta essere l'elemento chiave per la virtualizzazione e l'isolamento delle varie VNF sull'infrastruttura fisica, quindi un aggiornamento costante dello stesso, con annessa l'applicazione delle patch di sicurezza, può prevenire eventuali minacce. Un altro vantaggio potrebbe essere quello di abilitare i servizi, solo quando effettivamente necessari, ad esempio SSH e l'accesso remoto ad un servizio potrebbero non essere sempre necessari e, disabilitarli, potrebbe ridurre la superficie d'attacco. Ancora gli amministratori di sistema hanno la possibilità di accedere ad innumerevoli strumenti di gestione dell'infrastruttura, bisogna quindi rendere sicuri i loro account ed utilizzare una politica robusta di scelta delle password.

Creare una netta divisione del traffico tra le varie VNF, riduce, in caso di attacco, la possibilità che la compromissione di una di queste ricada sulle altre. Risulta buona norma, inoltre, separare la rete dedicata ad operazioni di gestione, da quella relativa al traffico dati. Ancora è possibile separare il traffico delle VLAN in diversi gruppi e disabilitare quelle che non vengono utilizzate, lo stesso discorso è applicabile alle VNF/VM che possono essere suddivise in diverse zone per isolarne il traffico. Tali zone possono essere protette creando delle diverse politiche di accesso attraverso un firewall dedicato, un esempio di creazione di diverse zone è dato dal concetto di *demilitarized zone (DMZ)*.

Risulta di interesse valutare anche quali possono essere i benefici, come prima elencato, nell'utilizzo di alcuni strumenti legati al kernel linux, in modo da rendere più sicuro lo strato del sistema operativo, onde evitare che l'attaccante possa far breccia nello stesso. Una prima considerazione sta nel fatto che il sistema operativo è fondamentale nell'isolamento tra le varie applicazioni, questo è vero sia nel caso di hypervisor che di container, quindi, meccanismi di sicurezza integrati nel kernel potrebbero risultare molto utili. Una risposta a questa esigenza è stata data dalla *National Security Agency (NSA)*, che ha sviluppato un modulo del kernel linux chiamato *SELinux*. Tale modulo garantisce un robusto isolamento tra i vari *tenant* (coloro che occupano parte delle risorse per i loro applicativi), come già accennato fondamentale nel caso di virtualizzazione direttamente sull'host. È anche presente in SELinux un'estensione di *libvirt*, chiamata *Secure Virtualization (sVirt)*, per integrare le funzionalità di *mandatory access control (MAC)* nel caso di hypervisor basati sul kernel linux. Tale funzionalità insieme ad altre messe a disposizione da sVirt consentono l'isolamento tra i processi della VM e i dati. Sono anche presenti altri strumenti sempre relativi al kernel linux per mettere in sicurezza le applicazioni, alcuni esempi sono dati da *hidepd*, per prevenire l'accesso non autorizzato ad alcuni processi e *GRSecurity* che previene attacchi di corruzione della memoria.

Interessante risulta anche andare a monitorare e validare il software che viene eseguito all'interno delle VM, questo è lo scopo dell'Hypervisor introspection. Tale tecnica permette, come farebbe un IDS, di accedere allo stato della VM e di valutare se presenti o meno rootkit o boot kit all'interno della stessa. I primi sono degli strumenti software utili a compiere un attacco informatico ed appropriarsi del controllo delle risorse, gli altri sono un'estensione dei primi e sono più difficili da individuare in quanto si nascondono sul disco e vengono avviati ad ogni boot. L'Hypervisor introspection è una tecnica potente per mettere in sicurezza una VM, ma dato l'accesso che ha allo stato delle VM può essere anche una potenziale vulnerabilità. Tra le librerie per mettere in atto tale tecnica abbiamo LibVMI, implementata in linguaggio c, tale libreria permette di ispezionare la VM su diversi punti (controlli sulla memoria, visualizzare il contenuto dei registri della vCPU e la registrazione delle interruzioni).

Per quanto riguarda il discorso sicurezza dei volumi è possibile, nel caso in cui si vogliano mantenere al sicuro i dati sensibili delle varie VM, effettuare una crittazione del contenuto dei volumi legati alle VM stesse. In seconda istanza dato che molti dati potrebbero anche essere

contenuti nell'area di swap, area di memoria su disco che serve di supporto al sistema operativo per ragioni di prestazioni, è auspicabile criptare anche il suo contenuto.

Le minacce si estendono anche alle immagini software, infatti, basta poco per inserire del codice malevolo all'interno delle stesse prima che esse vengano caricate su un repository. Per questo motivo è buona norma che le stesse vengano firmate e validate prima del lancio della VNF.

Un'altra considerazione di particolare interesse come visto in elenco è il discorso della gestione e dell'orchestrazione, in quanto il modulo dedicato (NFV MANO) è pieno di punti sensibili, come vedremo nel capitolo 3, dall'*Element Management System (EMS)* al VNF manager, fino alla gestione dei descrittori (VNFD, NSD, etc.). Tali moduli vanno messi in sicurezza, ad esempio per quanto riguarda i descrittori risulta buona pratica quella di stabilire dei limiti alla scalabilità delle funzioni di rete, in quanto la stessa, come abbiamo visto in precedenza, potrebbe essere utilizzata per degli attacchi di tipo *DNS amplification*. Inoltre, ultima considerazione su questo punto potrebbe essere quella di garantire un sistema per la gestione della sicurezza di NFV all'interno dei classici ambienti di orchestrazione (MANO).

L'ultima pratica di sicurezza applicabile è quella di utilizzare la remote attestation dei nodi, ovvero utilizzare il TPM precedentemente descritto per verificare l'affidabilità della piattaforma NFV in termini, ad esempio, dei compute node utilizzati per l'istanziamento delle VNF. Tale tecnica potrebbe essere messa in atto e gestita direttamente all'interno dell'ambiente di orchestrazione. In pratica esistono alcune tecnologie che implementano la remote attestation, una tra tutte è *openCIT*, un software open source disponibile su GitHub.

Rischio per la sicurezza	Target	Tecniche di mitigazione
Compromissione dell'hypervisor	Piattaforma	Separazione del traffico dati e quello di gestione, aggiornamento periodico dell'hypervisor
Falla nell'isolamento	Piattaforma/VNF	Hypervisor introspection, creazione di zone di sicurezza
Integrità della piattaforma	Piattaforma	TPM boot integrity, remote attestation
Attacco DDos	VNF	Istanziamento strategico delle VNF per difendere contro questo attacco, limitare la scalabilità con i descrittori (VNFD)
Amministratori malevoli	VNF	Crittografia Volumi/swap, firma delle immagini, attenta politica di gestione degli accessi e delle operazioni consentite
Problemi legati alle normative	VNF	Geo-tagging mediante remote attestation

Tabella 2.1: Rischi per la sicurezza con relativi target e tecniche di mitigazione.

Un interessante riassunto dei rischi e delle tecniche di mitigazione disponibili è dato dalla tabella 2.1.

2.5 Sfide future per NFV

Abbiamo valutato nelle sezioni precedenti quali sono i requisiti richiesti da NFV, in termini di prestazioni, e quali possono essere le criticità dell'architettura, in termini di sicurezza. Scopo di

questa sezione è approfondire queste ed altre criticità, con l'obiettivo di valutare quali possono essere i punti su cui si possa lavorare per migliorare nel complesso l'architettura. In accordo con [11] le sfide future per NFV appartengono alle seguenti aree:

- gestione ed orchestrazione;
- efficienza energetica;
- prestazioni;
- allocazione delle risorse;
- sicurezza;
- modellazione delle risorse, delle funzioni e dei servizi di rete.

Di seguito andremo a dettagliare quali sono le sfide per ognuna di queste aree.

2.5.1 Gestione ed Orchestrazione

Rispetto alle tradizionali reti, NFV ha una flessibilità e una velocità nell'istanziamento e distribuzione dei servizi molto maggiori. In un tipico scenario abbiamo un insieme di funzioni di rete, appartenenti a fornitori diversi, che devono essere eseguite su risorse di server magari differenti e collocati in punti geograficamente diversi.

Risulta chiaro che un paradigma di questo tipo ha necessità di un forte investimento sull'orchestrazione di tali funzioni, in quanto c'è bisogno che le stesse vengano istanziate in uno stato coerente, a seconda delle richieste e in un modo del tutto automatico. Inoltre, considerare la possibilità di migrazione di una VNF, da una VM all'altra, fa sorgere un'ulteriore caratteristica desiderabile: avere a disposizione dei meccanismi di monitoring come strumenti per coadiuvare il management del servizio end-to-end complessivo. Il concetto di orchestrazione è centrale in NFV, perché la maggior parte di soluzioni di gestione ed orchestrazione ad oggi si fermano al concetto di cloud computing. Ciò vuol dire che strumenti come Openstack, AWS, Kubernetes, Mesos sono specifici per l'orchestrazione di risorse e per garantire un servizio complesso a partire da microservizi, ma non tengono conto in modo completo delle esigenze di NFV, come la creazione di flussi dinamici di traffico e le prestazioni carrier-grade.

Questo impone soluzioni specifiche di orchestrazione, come nel caso ETSI e del progetto Open Source MANO, che tengano conto di questi aspetti. Quindi il binomio NFV e SDN, l'accelerazione di alcune funzioni di rete attraverso tecnologie come EPA, SR-IOV e ancora tanto altro, tra cui il concetto di forte distribuzione delle risorse su datacenter diversi (multi-site) e quello di un'orchestrazione gerarchica, per riuscire a coordinare tale distribuzione, sono aspetti che necessitano di una soluzione ad hoc che, in parte, è ancora una sfida per il futuro. Tutto questo sarà in parte approfondito nel prossimo capitolo 3 sull'orchestrazione di NFV, dove saranno presentate delle soluzioni di orchestrazione disponibili in modo open, e successivamente nei capitoli di implementazione (cap. 6 e 9), con uno studio specifico relativo a come utilizzare tali tecnologie in uno scenario reale.

2.5.2 Efficienza Energetica

Uno degli obiettivi della virtualizzazione delle funzioni di rete, come abbiamo visto, è l'ottimizzazione complessiva delle risorse. Tale ottimizzazione, che si traduce in un complessivo risparmio nella quantità dei server presenti nell'infrastruttura fisica, ha un forte impatto sul consumo di energia complessivo da parte dei Service Provider. L'impatto energetico, negli ultimi anni è diventato una delle sfide più pressanti delle nuove architetture di rete, infatti, in accordo con l'iniziativa Global e-Sustainability Initiative (GeSI) [11], si è riscontrato che qualora il cloud, inteso come infrastruttura complessiva a livello mondiale, fosse un paese, sarebbe il sesto al mondo in quanto a consumo di energia. Questi ultimi dati sono destinati ad aumentare nei prossimi anni fino ad

un incremento del 63% fino al 2020. Questo è significativo anche rispetto alle politiche di salvaguardia ambientale. Un'altra prospettiva riguarda i Service Provider e le implicazioni economiche che questo enorme dispendio produce. Si stima, infatti, che il 10% dei costi sostenuti dagli stessi sia relativo all'energia elettrica, con similari tassi di crescita per l'orizzonte 2020, sembra ragionevole, quindi, cercare di ottimizzare sempre più l'efficienza energetica dell'architetture di rete per mitigare l'impatto di tale spesa sui Service Provider.

NFV, secondo i dati ottenuti da un framework proposto dai Bells Labs [11], per la misurazione dell'efficienza energetica delle VNF e basato sulle previsione di crescita del traffico, consente un considerevole risparmio energetico per svariate funzioni di rete nella versione virtualizzata rispetto all'implementazione fisica.

	Traffic (EXABYTES/MONTH)	Total Efficiency (MBITS/J)	Total Power (MWATTS)	Power Savings (MWATTS)	Cummulative Savings (2013 - 2018) GJ
Baseline Network	1,153.05	0.0328510	116,203	0.0	
Virtual EPC	1,153.05	0.0422222	92,159.8	24,044.1	5.0×10^9
Virtual CPE	1,205.11	0.0352130	113.500	2,703.63	5.5×10^9
Virtual RAN	1,227.88	0.0463708	89,599.5	26,604.4	7.5×10^9
Virtual Video CDN	810.22	0.0346562	80,029.3	36,174.6	7.5×10^9
Virtual Broadband Network Gateway	1,169.69	0.0333016	116.260	-76.794	-1.7×10^7
Virtual Provider Edge	1,151.91	0.0328255	116,180	22.9517	3.8×10^6

Figura 2.16: Risparmio energetico introdotto da NFV (fonte: [11]).

La tabella in figura 2.16 mostra come, per la maggior parte delle funzioni di rete virtuali prese in considerazione, ci sia un risparmio energetico notevole. Per ogni singola funzione analizzata, è disponibile un dato sulla quantità di traffico mensile medio, l'efficienza energetica espressa in MBITS/Joule, il totale della potenza consumata, il risparmio energetico ed infine una previsione sul risparmio energetico delle stesse per il periodo 2013-2018.

Questo ci dà un'idea su come NFV possa migliorare in modo concreto l'efficienza energetica, ovviamente questo contributo va ancora perfezionato e contestualizzato rispetto al trade-off sempre presente tra l'efficienza energetica ed il suo impatto sulle prestazioni delle VNF.

2.5.3 Prestazioni

L'obiettivo principale di NFV è quello di eseguire le VNF su dei server standard, in gergo “commodity server”. Assunto questo, abbiamo una situazione per cui i produttori di hardware si trovano a produrre senza sapere quali VNF verranno eseguite sullo stesso e, contemporaneamente, gli sviluppatori di VNF devono assicurare che il loro software sia compatibile con i commodity server. In breve diventa complicato assicurare un completo supporto da parte dell'hardware a ogni tipo di VNF, senza una forte standardizzazione. A questo si aggiunge il fatto che, talune funzioni di rete, necessitano di notevoli prestazioni (es. DPI, Dedup e NAT) e quindi risulta un vantaggio eseguirle su un hardware specializzato. Questo complica notevolmente il discorso performance in NFV, perché, oltre alla compatibilità, bisogna valutare quanto è necessario avvicinarsi alle prestazioni delle funzioni fisiche, quali funzioni si ha intenzione di virtualizzare e quali tecniche posso utilizzare in alcuni casi per ridurre il gap di prestazioni con il loro corrispettivo fisico.

A queste problematiche cerca di dare risposta l'ETSI con un'analisi nello standard sul discorso prestazioni [12]. In questo documento non solo si evince, da alcuni test effettuati su funzioni quali DPI, C-RAN e BRAS, che molte funzioni virtuali possono garantire un throughput elevato (più di 80 Gbps per server in un ambiente virtualizzato), ma che riescono anche a dare caratteristiche di predicibilità e consistenza rispetto alla corrispettiva versione fisica.

Questa è una risposta in ogni caso parziale, perché, come già accennato, tutta una serie di funzioni (es. Dedup) hanno delle esigenze particolari in termini di prestazioni. Esigenze che un ambiente virtualizzato tipicamente non soddisfa. Una soluzione a questo problema nel caso riportato da [11] è l'utilizzo, contestualmente all'ambiente virtualizzato, di acceleratori hardware,

basati su tecnologia FGPA, e di interfacce PCI e SR-IOV. Queste tecnologie, che in parte analizzeremo più avanti, permettono di raggiungere prestazioni di gran lunga superiori e di avvicinarsi alle prestazioni nel caso di dispositivi fisici con hardware specializzato.

Ovviamente in questo concetto c'è una contropartita, ovvero l'utilizzo di queste soluzioni specifiche limita la flessibilità di NFV, perché tipicamente la programmabilità di alcuni dispositivi di supporto è limitata e la configurazione di alcune interfacce introdotte è meno flessibile. La ricerca di un trade-off tra questi due aspetti (utilizzo di hardware specifico e portabilità delle VNF) resta ancora un sfida aperta per NFV.

2.5.4 Allocazione delle risorse

Per raggiungere un'applicabilità su larga scala di NFV è necessario utilizzare le risorse in modo efficiente. Questo si traduce in un'esigenza spinta nel progettare ed implementare degli algoritmi efficienti, per determinare su quali risorse fisiche convenga allocare le funzioni virtuali. Tali algoritmi devono tener conto di tutta una serie di fattori tra cui il bilanciamento del carico, il risparmio energetico e i tempi di ripresa in seguito a guasti. Fare questo significa modellare questo scenario come un problema di ottimizzazione e molte sono in letteratura le risposte a tale esigenza di modellazione. Un esempio tra tutti è dato da [13], dove l'autore dell'articolo ha proposto un modello per formalizzare la catena del servizio di rete. Tale modellazione parte da un Forwarding Graph, che successivamente verrà mappato su risorse fisiche, tenendo conto del fatto che le risorse a disposizione sono limitate e ogni VNF ha dei requisiti diversi. A questo scopo viene formulato il problema come *Mixed Integer Quadratically Constrained Program (MIQCP)* e viene osservato dall'autore che per ottenere dei buoni risultati bisogna tener conto, nell'allocazione delle risorse, dell'obiettivo di tale ottimizzazione (es. throughput costante, latenza, etc.).

Oltre al problema di ottimizzazione esistono anche problematiche di contorno, una fondamentale è la necessità di far sì che la VNF utilizzi solo le risorse che gli sono effettivamente necessarie. Nella maggior parte dei Proof-of-Concept (PoC) questo non viene fatto, infatti, allocare una VM ad esempio per alcune funzioni "leggere" (quali ad es. un DHCP in un CPE) potrebbe essere eccessivo, tenendo conto dell'overhead di un intero sistema operativo guest. Inoltre, molte funzioni non hanno necessariamente l'esigenza di essere strettamente isolate l'una dall'altra. Al fine di ridurre l'impatto di questa problematica si sta cercando di passare come già accennato nei capitoli precedenti, dall'utilizzo delle VM a quello dei linux container. Una delle alternative prese maggiormente in considerazione è la tecnologia Docker, al fine di ottenere un isolamento automatico delle risorse e utilizzare alcuni dei meccanismi descritti in sez. 2.3.5 per partizionare la memoria, le risorse di rete e i processi. L'utilizzo di queste tecnologie permette di eliminare l'overhead del sistema operativo guest e di garantire complessivamente prestazioni migliori. L'unica sfida ancora aperta riguarda la sicurezza, ma molti sforzi di ricerca si concentrano proprio su quest'ultima. Sui container c'è anche la problematica aperta relativa all'orchestrazione, ovvero esistono numerosi strumenti a disposizione per effettuare la stessa come Docker Swarm, Apache Mesos, Google Borg/Kubernetes, ma quella che va ancora ben indagata è la loro capacità di garantire prestazioni carrier-grade. Uno studio del 2015 [14], suggerisce che Borg (Orchestratore di cluster, sviluppato da Google) ad esempio, ha una latenza di startup di 25s e un disponibilità del servizio four-nines (ovvero il 99,99%), quindi andrebbe proposta una metodologia per migliorare tali prestazioni al fine di ottenere il carrier-grade.

Esistono altri punti aperti sull'allocazione delle risorse, a partire dalla discussione su come devono essere condivise le risorse fisiche tra le varie VNF, fino ad arrivare a come garantire l'ottimizzazione dell'allocazione delle risorse del sistema online, tenendo conto di tutte le problematiche di multi-dominio, sostenibilità delle reti, gestione dinamica delle VNF e dell'NS.

2.5.5 Sicurezza

Abbiamo descritto nelle sezioni precedenti le minacce e i possibili attacchi a NFV. Ovviamente questo non rappresentava che un sottoinsieme delle possibili vulnerabilità e delle sfide aperte in ambito sicurezza per questo paradigma.

La prima sfida significativa per NFV in questo ambito è quella di definire un'interfaccia standard nell'architettura ETSI NFV, che permetta di istanziare funzioni di rete, che reagiscano direttamente alle varie minacce in tempo reale. Questa funzionalità dovrebbe essere in grado di comunicare con il modulo di orchestrazione e seguire le buone norme descritte nella sezione precedente.

Altre sfide riguardano la possibilità di monitorare e gestire in modo sicuro le VNF durante la migrazione, rendendo possibile il mantenimento delle configurazioni e dello stato delle stesse. Questo potrebbe rappresentare un problema data la dinamicità e flessibilità di NFV. Ancora di interesse potrebbe essere implementare il concetto di “trust” tra differenti fornitori di VNF, ovvero dare la possibilità di mantenere la “chain of trust” tra fornitori diversi e garantire l'affidabilità delle VNF. Un'altra sfida è quella relativa al portare la remote attestation a run-time, per ora è solo possibile effettuare attestazione a boot-time.

Queste sono tra le principali sfide e, unite a quelle descritte nella sezione precedente, sono da integrare agli sforzi per rendere il framework NFV più sicuro.

2.5.6 Modellazione delle risorse, delle funzioni e dei servizi

Il potenziale di NFV risiede nella sua abilità di offrire un'automatizzazione di alto livello e un'estrema flessibilità. Resta il fatto che le VNF provengono da fornitori diversi, questo significa che per raggiungere la capacità di istanziare servizi su larga scala, è necessario avere a disposizione dei descrittori (delle VNF e dei NS) che siano basati su uno standard open, che siano ben strutturati e che descrivano il servizio di rete o la funzione in modo completo. Questo deve tener conto di vari aspetti oltre quelli citati, ovvero alcune VNF potrebbero avere delle esigenze diverse in termini di risorse, avere richieste riguardo ad accelerazione hardware o essere eseguite su diversi VIM. A questo punto, la faccenda si può complicare ulteriormente, se ad esempio andiamo a cambiare la tipologia di infrastruttura sottostante per la virtualizzazione (es. passaggio a container). Un descrittore completo fornisce quindi dei parametri standard per l'istanziamento, la configurazione e la scalabilità della VNF e del servizio di rete, che tenga conto di tutti gli aspetti citati.

A partire da questo preambolo esistono differenti proposte dello standard ETSI [15] per la modellazione di tali descrittori e riportiamo di seguito i tre principali:

- *Topology & Orchestration Standard for Cloud Application (TOSCA)*: TOSCA è un linguaggio basato sullo standard OASIS, per descrivere una topologia di un servizio web basato su cloud, i suoi componenti, le relazioni e i processi che lo gestiscono.
- *NETCONF/YANG*: NETCONF è un protocollo definito dall'IETF [16], per “installare, manipolare e cancellare le configurazioni dei dispositivi di rete”. Le operazioni di tale protocollo sono realizzate su un livello RPC (Remote Procedure Call) attraverso una codifica xml, che mette a disposizione una serie di operazioni di base per modificare e richiedere configurazioni su un dispositivo di rete. Ancora è basato su un linguaggio di modellazione quale YANG, lo stesso viene utilizzato sia per i dati di configurazione che quelli relativi allo stato.
- *Information Framework (SID)*: SID è un componente del TM Forum's Framework che ha come obiettivo quello di mettere a disposizione un modello informativo e un dizionario comune per tutte le informazioni condivise tra diverse entità, al fine di gestire clienti, posizioni ed elementi di rete. L'entità sono ben caratterizzate dal fatto che gli attributi descrivono non solo se stesse ma anche il loro comportamento in termini di operazioni.

Tutti questi modelli però non sono stati sviluppati con l'idea di essere utilizzati per NFV, di conseguenza ognuno di questi modelli presenta delle mancanze per definire in modo completo tutto ciò che è necessario per le funzioni virtuali e i servizi di rete.

Un esempio è dato da TOSCA, che ha bisogno di essere migliorato sotto gli aspetti relativi al supporto della portabilità e dei servizi federati. NETCONF deve estendere il suo supporto ad amministratori multipli e ad uno scenario multi-dominio, ancora TOSCA manca di supporto relativo management a run-time. Questo ci porta ad uno scenario in cui ad esempio, utilizzando

TOSCA, si possa istanziare, modificare o eliminare dei router virtuali ma non ho alcuna possibilità di modificarne le configurazioni.

Uno scenario plausibile potrebbe essere quello di combinare questi diversi approcci, al fine di ottenere tutte le specifiche richieste dallo standard. In uno scenario del genere, ad esempio, TOSCA potrebbe sopperire alle funzionalità precedentemente viste e, combinato con NETCONF/YANG, lasciare a quest'ultimo l'onere di mettere a disposizione delle API a run-time per configurare sia le funzioni che i servizi di rete.

Capitolo 3

Architettura NFV MANO

In questo capitolo approfondiremo il modulo di *Management and Orchestration* dell'architettura NFV, ovvero NFV MANO.

3.1 Funzionalità

Nei precedenti capitoli, a partire dai concetti di VNF e NS, sono stati introdotti i componenti di un'architettura NFV. Parlando dell'infrastruttura, abbiamo compreso quali sono le principali tecnologie per la virtualizzazione delle funzioni di rete e quali possono essere alcune delle problematiche. In riferimento al data model, abbiamo poi visto come modellare le caratteristiche e i requisiti dei servizi di rete e delle relative funzioni, senza tralasciare gli strumenti accessori come i link virtuali e i forwarding graph. A questo punto saremmo in grado di istanziare un servizio di rete, ma non abbiamo ancora un'idea precisa su come coordinare tutti gli strumenti visti in precedenza e soprattutto su come configurare e gestire le funzioni di rete. Il tassello mancante all'architettura NFV è quello che descriveremo di seguito: il modulo di gestione ed orchestrazione (MANO). Tale modulo ci fornirà, attraverso i suoi componenti, delle metodologie per coordinare le operazioni di deploy del servizio di rete e meccanismi per la gestione delle funzioni di rete. Di seguito vediamo quali sono le fondamentali funzionalità offerte da un modulo NFV MANO [15]:

- gestione delle risorse virtuali di NFVI: disponibilità, allocazione e rilascio;
- gestione delle problematiche e delle prestazioni di NFVI;
- istanziazione, scalamento, aggiornamento, modifica e terminazione di una VNF;
- *on-boarding* di una VNF o di un NS;
- istanziazione, scalamento, modifica e terminazione di un NS;
- creazione, cancellazione, interrogazione e modifica di un VNFFG;

Queste sono le funzionalità fondamentali offerte da un sistema MANO, ovviamente ne esistono anche altre accessorie, ad esempio quelle che prevedono il monitoraggio delle funzioni virtuali e dei servizi di rete. Tali funzionalità consentono, attraverso delle metriche predefinite, di valutare quando effettuare determinate operazioni: un esempio classico è lo scalamento.

3.2 Architettura

In questa sezione andremo a definire quali sono i componenti fondamentali dell'architettura del modulo MANO all'interno di un framework NFV e ne descriveremo in dettaglio le funzioni.

Come si vede dalla figura 3.1 i componenti fondamentali sono:

state allocate per ogni singola VNF, questo permette di utilizzarle successivamente anche ai fini della gestione di queste ultime. Tali dati vengono immagazzinati all'interno di repository, accessibili da parte dell'RO, che corrispondono ai moduli architetturali "NFV Instances" e "NFVI Resources" della figura 3.1.

Il Network Service Orchestrator, invece, lavora a livello più alto, fungendo da una parte da tramite tra il VNF Manager (interfaccia Or-Vnfm) e l'RO, dall'altra da supervisore, dialogando anche con moduli di gestione a più alto livello come l'OSS/BSS (interfaccia Os-Ma-nfvo). L'SO si occupa anche della fase di on-boarding, ovvero carica in alcuni repository adatti (NS catalogue e VNF Catalogue) tutti i descrittori dei servizi e delle funzioni di rete necessari al deploy, tenendo conto anche delle eventuali risorse di supporto necessarie (immagini software delle VNF e altri file di supporto). L'SO è anche il modulo che si occupa dell'istanziamento del Network Service, quindi a partire dai dati contenuti nei suoi database, è in grado di contattare l'RO e di far allocare le risorse necessarie per ogni singola VNF. Dopo questo passaggio, può utilizzare i riferimenti alle VNF messi a disposizione dall'RO per gestire le funzioni di rete, questo o attraverso il VNF Manager o direttamente attraverso il VIM (interfaccia Or-Vi). Queste rappresentano le funzionalità di base, mentre nelle architetture di progetti attuali, come vedremo in seguito, esistono altre funzioni che vengono messe a disposizione da questo sottomodulo dell' NFVO.

3.2.2 VNF Manager

Il VNF Manager è il modulo che si occupa della gestione del ciclo di vita della singola VNF. Ogni VNF è associata ad un VNF manager e quest'ultimo può coordinare anche più funzioni di rete contemporaneamente. Nel caso base, si presuppone che le operazioni, che possono essere eseguite su di una singola funzione di rete, siano applicabili anche alle altre. Questo si traduce nel creare un sottoinsieme di funzionalità comuni (ad. esempio inizio/terminazione/scalamento), applicabile a tutte le VNF gestite dal quel VNFM. Tale assunzione non risulta sempre efficace, infatti, molte funzioni di rete richiedono delle configurazioni particolari, per questo è talvolta richiesta una flessibilità maggiore da parte VNFM o addirittura sono richiesti VNF Manager diversi. Tralascieremo per ora i dettagli sulle singole operazioni di gestione, che questo gestore è in grado di effettuare, in quanto successivamente vedremo degli esempi pratici. Quello che ci interessa notare, è che molte volte il VNF Manager è coadiuvato da un ulteriore elemento descritto in sezione 3.2.4, ovvero l'*Element Management System (EM o EMS)*.

3.2.3 Virtual Infrastructure Manager

Il Virtual Infrastructure Manager (VIM) è l'ultimo componente che divide la parte di MANO dall'infrastruttura NFVI. Esso si occupa della gestione ad alto livello di tutte o di parte delle risorse NFVI (Compute, Storage e Network), nel secondo caso si parla di VIM specializzato.

Il VIM presenta due interfacce: una *northbound interface* verso i componenti a livello più alto (VNFM e NFVO, attraverso le interfacce Vi-Vnfm e Or-Vi) e una *southbound interface* verso i componenti a livello più basso (NFVI attraverso l'interfaccia Nf-Vi). La prima espone le API che definiscono le operazioni applicabili alle risorse dell'infrastruttura (es. allocazione, terminazione e scalamento), la seconda, invece, dialoga con gli hypervisor di NFVI per mettere in atto le operazioni richieste. In sintesi Il VIM mantiene un inventario delle varie risorse, valida le richieste provenienti dalla parte di orchestrazione e le comunica all'infrastruttura quando esse risultano coerenti con lo stato della stessa.

Abbiamo parlato di VIM specializzato, potrebbe essere quindi di interesse citare uno dei casi particolari che ritroviamo spesso nelle architetture reali: il *WAN Infrastructure Manager (WIM)*. Per capire di cosa si occupa, dobbiamo rivisitare il concetto di *Network Controller*. Quest'ultimo è un componente dell'infrastruttura che si occupa della gestione della connettività a livello 2/3 e può essere un gestore interno al singolo NFVI-PoP o ancora gestire la connettività tra diversi NFVI-PoP. Nell'ultimo caso, il modulo specializzato a livello di NFV MANO e in grado di colloquiare con questo tipo di network controller, prende proprio il nome di WIM. In aggiunta, quando si parla di Network Controller si ci riferisce ad un componente in grado di gestire il control plane del

traffico tra le varie VNF, che esse siano collocate geograficamente nello stesso PoP o meno. Un esempio di Network Controller, a cui abbiamo accennato nel precedente capitolo 2, è dato dall'SDN Controller, un componente che rivedremo e approfondiremo nel corso del lavoro di tesi.

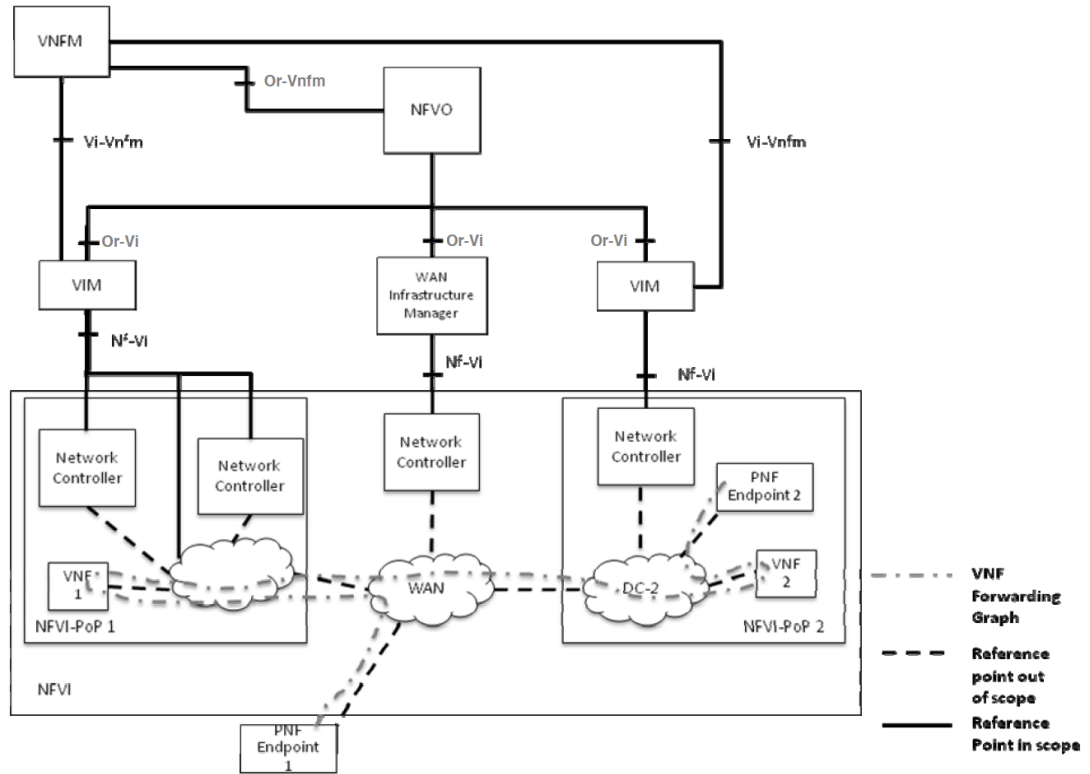


Figura 3.2: Esempio di Network Controller (fonte: [15]).

In figura 3.2 abbiamo un esempio di utilizzo di diversi network controller in un'architettura NFV. Nella parte alta vi sono tutti i componenti NFV MANO, descritti in precedenza (VNFM, NFVO, VIM e WAN), mentre nella parte bassa ritroviamo l'infrastruttura NFVI. Quest'ultima è composta da diversi NFVI-PoP, fisicamente localizzati in punti diversi; al loro interno troviamo dei Network Controller, che gestiscono il traffico interno delle varie VNF e gli end-point fisici. All'esterno dei due PoP descritti in figura, notiamo che è presente un Network Controller guidato da un WIM. Quest'ultimo non indirizza direttamente il traffico delle VNF, ma è in grado di gestire la connettività tra i vari PoP. Il sistema NFV MANO sfrutta ognuno dei VIM e dei WIM per gestire il control plane complessivo ed è in grado di creare dei percorsi del traffico articolati, non solo tra le VNF intra-PoP, ma anche tra le VNF e gli end-point di diversi PoP. Tutto questo avviene a livello superiore (Network Service Orchestrator) attraverso la modifica del VNFFG, il quale definisce il percorso che devono effettuare i pacchetti tra le varie funzioni di rete e dispositivi fisici di tutto il servizio di rete. Successivamente vengono istruiti i moduli sottostanti VIM/WIM, i quali propagano le regole ai controller, modificando di fatto il traffico dei pacchetti all'interno dell'infrastruttura NFVI. Nel nostro caso particolare in figura 3.2, il traffico viene indirizzato dall'end-point 1 fino all'end-point 2, attraverso il percorso delineato dal Forwarding Graph in figura 3.2.

Oltre al caso specializzato, il VIM di solito si presenta come uno strumento in grado di gestire tutti i domini che abbiamo illustrato nella sezione di NFVI. Più avanti nell'implementazione delle nostre soluzioni NFV, andremo ad approfondire qual'è la struttura completa di un VIM e come viene implementata (cap. 6).

3.2.4 Element Management System

L'Element Management (EM) o Element Management System (EMS) è responsabile delle funzioni di gestione FCAPS (Fault, Configuration, Accounting, Performance e Security) della VNF. Più nel dettaglio le funzionalità che offre sono:

- configurazione della funzione di rete assolta dalla VNF;
- gestione delle falle della funzione di rete assolta dalla VNF;
- statistiche sull'utilizzo della VNF;
- collezione di dati sulle prestazioni della funzione di rete assolta dalla VNF;
- gestione della sicurezza per la VNF.

Spesso questo modulo, per assolvere alle funzionalità descritte, potrebbe dover comunicare con il VNF manager (attraverso l'interfaccia Ve-Vnfm-em), al fine di ottenere informazioni sulle risorse NFVI. Tipicamente l'EM è un modulo software che viene eseguito sullo stesso NFVI-PoP dove sono presenti le VNF e può essere incorporato nella funzione di rete stessa. Come vedremo successivamente, alcuni progetti MANO tendono a vederlo come modulo esterno alla VNF, questo per avere a disposizione una maggiore flessibilità dell'architettura complessiva e consentire l'aggiornamento della VNF e dell'EM in modo indipendente. Inoltre, tale scelta permette a più Element Management di interagire tra di loro, per gestire in modo più complesso un insieme di funzioni di rete.

3.2.5 Repository

In un sistema NFV MANO molteplici informazioni vengono immagazzinate per modellare il servizio e la gestione del suo ciclo di vita. In questa sezione presentiamo quali sono i principali repository e quali dati contengono.

Esistono quattro tipologie di repository:

- *NS Catalogue*: questa base di dati contiene tutte le informazioni relative ai descrittori dei servizi di rete (NSD), dei link virtuali (VLD) e dei VNF Forwarding Graph (VNFFGD).
- *VNF Catalogue*: questo repository contiene tutte le informazioni relative alle VNF, in termini di descrittori (VNFD), immagini software e file di supporto.
- *NFV Instances repository*: contiene le informazioni sugli NS e le VNF che sono stati istanziati, questo è ovviamente utile nella gestione del ciclo di vita del servizio e delle funzioni di rete.
- *NFVI Resources repository*: contiene tutta una serie di informazioni riguardo alle risorse dell'infrastruttura che sono a disposizione.

3.3 Progetti NFV MANO

Una volta chiari i concetti base su cui poggia il paradigma NFV e capito il ruolo fondamentale del sistema di gestione e orchestrazione, presentiamo di seguito alcuni dei progetti e delle architetture implementate attualmente disponibili:

- *Open Source MANO*: questo è un progetto coordinato direttamente dall'ETSI, che si pone l'obiettivo di sviluppare un insieme di software open source, per la gestione e l'orchestrazione (MANO) secondo il modello ETSI NFV. Tale progetto è distribuito sotto licenza Apache v2 e verrà dettagliato più avanti (sez. 3.3.1).

- *Open Baton*: questo è un framework open, allineato allo standard NFV MANO, altamente estensibile e personalizzabile. I dettagli verranno presentati più avanti. (sez. 3.3.4)
- *OPNFV*: il progetto si propone di realizzare una completa infrastruttura NFVI, che abbia tutte le caratteristiche necessarie per offrire le prestazioni desiderate dal paradigma. Verrà approfondito in dettaglio in sezione 3.2.3
- *Tracker*: Tracker è un progetto ufficiale di OpenStack, che ha come obiettivo quello di estendere le funzionalità di quest'ultimo, ampliando la sua architettura con un VNF Manager e un NFV Orchestrator. Anch'esso risulta allineato allo standard ETSI MANO e dispone delle funzionalità necessarie per fornire un servizio end-to-end. Faremo una panoramica sull'architettura di OpenStack più avanti nei capitoli implementativi (cap. 6).
- *T-NOVA*: questo è nato in relazione al programma di finanziamento europeo FP7 ed ha come obiettivo la progettazione ed implementazione di un framework, in grado di adempiere a tutte le funzioni di un MANO. La peculiarità sta nel fatto che oltre a permettere il deploy di VNF all'operatore, tale framework dovrebbe consentire, on-demand e as-a-Service, agli utenti del servizio di creare delle proprie funzioni di rete (gateway, proxy, firewall, etc.).
- *GigaSpaces Cloudify*: è un progetto open source basato sullo standard TOSCA. In origine è stato introdotto come semplice orchestratore di risorse (come OpenStack HEAT), successivamente è stato adattato per NFV. Cloudify ha un'architettura molto estendibile e può interagire con diversi VIM come AWS, Openstack e Microsoft Azure. Viene distribuito open source sotto licenza Apache v2.
- *ONAP*: Open Network Automation Platform è un progetto coordinato dalla Linux Foundation, inerente ad una piattaforma per l'orchestrazione e la gestione delle funzioni di rete virtuali. Tale progetto nasce dalla combinazione tra ECOMP e OPEN-O, il primo un progetto di AT&T (Enhanced Control, Orchestration, Management & Policy) e il secondo sempre della Linux Foundation (Open Orchestrator). L'unione di queste due iniziative, crea un framework complessivo che consente l'orchestrazione di funzioni di rete in tempo reale e basata su policy.
- *CORD/XOS*: Central Office Re-architected as a Datacenter è un progetto nato come uno dei casi d'uso dell'SDN Controller ONOS. Successivamente si è evoluto e ha avuto come obiettivo quello di combinare NFV, SDN e l'agilità del cloud, al fine di creare un framework di gestione ed orchestrazione delle funzioni di rete. Come SDN controller utilizza per l'appunto ONOS, come VIM OpenStack e come Orchestrator XOS.
- *NFV over Open DC/OS*: questa è un'architettura NFV MANO interessante, perché sfrutta il sistema Mesosphere DC/OS per inizializzare delle funzioni di rete sotto forma di linux container. Vedremo i dettagli nella sezione 3.3.4.

3.3.1 Open Source MANO

Open Source MANO (OSM) è un framework costituito da un insieme di software, atto a garantire le funzioni MANO in un'architettura NFV. OSM è costituito da tre componenti principali:

- *Resource Orchestrator (RO)*: questo è un modulo che svolge un sottoinsieme delle funzioni dell'NFV Orchestrator, in particolare l'orchestrazione delle risorse dell'infrastruttura NFVI, in accordo con la definizione di RO della precedente sezione.
- *VNF Configuration & Abstraction (VCA)*: il VCA è il modulo che si occupa della gestione del ciclo di vita delle funzioni di rete virtuali, in particolare ha la possibilità a run-time di configurare e gestire le stesse. Tale modulo non rappresenta un software specifico, ma tende a fungere da contenitore per uno svariato numero di software, che consentono di effettuare il management delle VNF. All'interno di tale modulo possiamo ritrovare le definizioni di VNF Manager e Element Management System.

- *Service Orchestrator (SO)*: l'SO si occupa, innanzitutto, della coordinazione dei due precedenti moduli. Quest'ultimo, infatti, gestisce in modo completo il ciclo di vita del servizio di rete, partendo dall'on-boarding dei descrittori, fino alla gestione on-line del servizio di rete.

Nelle prossime sezioni daremo uno sguardo più nel dettaglio ai moduli presentati.

Resource Orchestrator (RO)

Il Resource Orchestrator di OSM è OpenMANO, un software sviluppato da Telefonica, uno tra i principali membri del gruppo ETSI ISG NFV MANO. Esso consente l'orchestrazione di risorse attraverso uno o più VIM ed è compatibile con le soluzioni più di rilievo presenti sul mercato e in modo open.

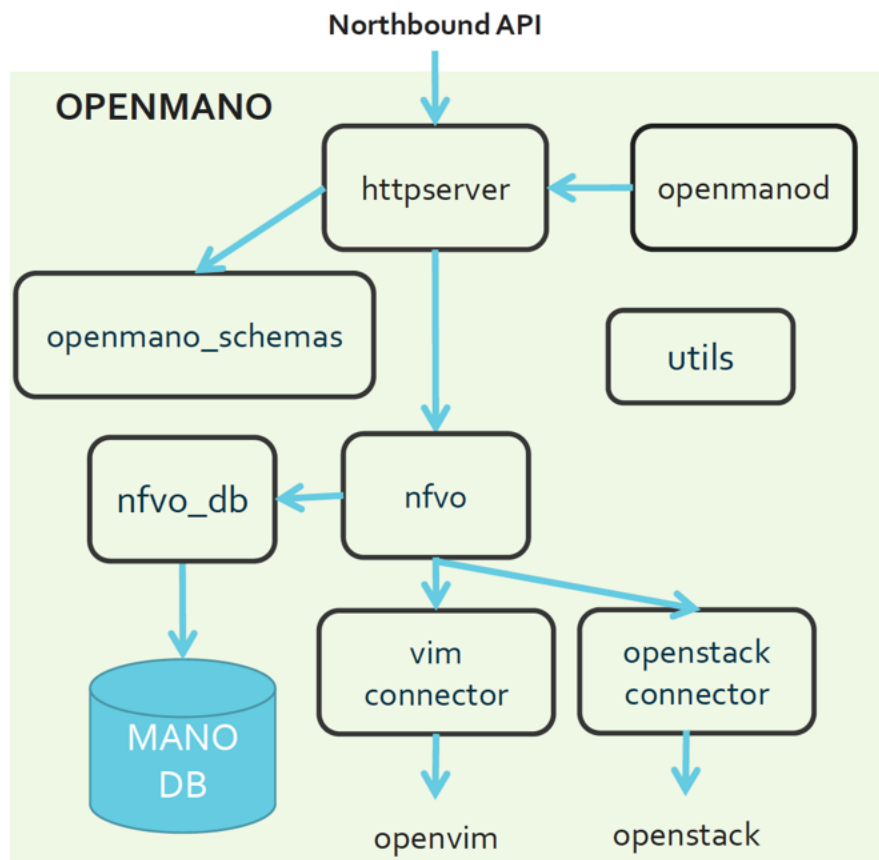


Figura 3.3: Architettura OpenMANO (fonte: [osm wiki](#)).

Come si evince dalla figura 3.3, OpenMANO è composto da un serie di moduli software sviluppati in linguaggio Python. Il cuore di tali moduli è l'*nfvo.py* che implementa tutti i metodi per la creazione, terminazione e gestione di quelli che chiama *scenario* e *instance*. Gli scenario corrispondono al nostro concetto di Network Service, mentre le instance alle VNF. Bisogna contestualizzare questo RO, per capire il perché di una apposita terminologia rispetto allo standard. OpenMANO nasce prima del progetto Open Soruce MANO, per questo motivo era stato pensato come orchestratore complessivo di tutte le risorse necessarie per un servizio di rete. Successivamente, le crescenti esigenze di funzionalità di un framework MANO, hanno portato all'abbandono di orchestratori così semplici. Tralasciando questa piccola digressione, andiamo ad analizzare gli altri moduli:

- *openmanod.py*: funge da main e si occupa della gestione complessiva degli altri moduli.
- *httpserver.py*: è costituito da un thread che gestisce la northbound API, in termini di richieste http.

- *openmano_schemas.py*: modulo che serve alla validazione delle richieste da parte delle API e delle risposte in formato JSON. Questo avviene attraverso la libreria JSON schema.
- *utils*: rappresenta tutta una serie di funzioni di utilità, utilizzate dagli altri moduli.
- *nfvo_db.py*: si occupa della gestione di un database (MANO DB), contenente tutti i cataloghi di tutte le *instance* e gli *scenario* a disposizione. Ovviamente tali cataloghi non sono quelli di OSM, ma sono una versione comprensibile all'RO, consentendo, di fatto, il riutilizzo di OpenMANO per fare da tramite con i VIM.
- *vim/openstack connector.py*: questi sono una serie di moduli che mappano le richieste di risorse NFVI, provenienti dal modulo *nfvo.py*, in API specifiche per il tipo di VIM utilizzato. In figura 3.3 sono presenti solo due dei VIM supportati dall'attuale OpenMANO, che in realtà ha esteso tale supporto anche ad altri VIM come Amazon Web Services (AWS) e VMware.

L'utilizzo, infine, del modulo RO è garantito dalla northbound interface, la quale espone delle API che si possono consultare nella documentazione ufficiale [17].

Una piccola nota a margine sta nel fatto che OpenMANO, almeno inizialmente, era concepito per un solo VIM, lo stesso era sviluppato da Telefonica e prendeva il nome di OpenVIM. Questo Virtual Infrastructure Manager non è molto diffuso, quindi è caduto quasi subito in disuso. Anche il consorzio di OSM è passato per le demo ufficiali all'utilizzo di OpenStack, ma non ha abbandonato del tutto il progetto OpenVIM, il quale viene proposto come alternativa soprattutto in fase di testing.

VNF Configuration & Abstraction (VCA)

Il componente VCA rappresenta un'astrazione del VNF Manager. In esso infatti è possibile allocare, teoricamente, uno o più VNFM ed EM, che si occupano della gestione delle VNF, a prescindere dal VIM utilizzato per il deploy effettivo del servizio. Vedremo più avanti, in particolar modo nei capitoli successivi di implementazione, che tale astrazione non è sempre una scelta vincente o quantomeno non implementata in modo impeccabile, questo soprattutto nel caso di infrastruttura NFVI basata su container.

Open Source MANO prevede un VNF Manager di default, questo è JUJU, un software sviluppato dalla Canonical, membro attivo del consorzio per lo sviluppo di OSM. L'utilizzo di JUJU è strettamente raccomandato dai creatori di Open Source MANO, questo ha come obiettivo quello di far convergere tutte le risorse, dedite alla gestione delle VNF, in un solo applicativo. In un ecosistema fortemente frammentato come quello di NFV, dove le VNF provengono da diversi fornitori e le soluzioni applicative sono le più disparate, sembra ovvio il vantaggio di utilizzare un unico VNF Manager. Questa soluzione però ha numerosi vantaggi e svantaggi che approfondiremo avanti nella parte di analisi dell'implementazione (cap. 6 e 8).

La cosa che ci interessa adesso, per la definizione dell'architettura OSM, è che JUJU ricalca completamente i concetti di VNF Manager e Element Manager, identificati dai suoi moduli architetturali *juju controller* e *proxy charm*.

Service Orchestrator (SO)

Il Service Orchestrator è il modulo che garantisce, innanzitutto, le funzionalità descritte nelle sezioni precedenti tipiche di un SO: NS/VNF on-boarding e ciclo di vita del NS. Come SO, OSM utilizza RIFT.ware, un software scritto in python, capace di garantire, oltre a quelle di base, numerose funzionalità aggiuntive. RIFT.ware, infatti, è in grado di effettuare una traduzione dei descrittori, da svariati formati (yaml, xml, JSON, etc.), attraverso un modello dati YANG, in un formato che può direttamente essere utilizzato dai moduli python dei componenti di OSM. Ancora, controlla ad alto livello l'interazione tra JUJU (VNF Manager) e il RO (openMANO) e mette a disposizione una comoda GUI per gestire nel complesso il framework di OSM.

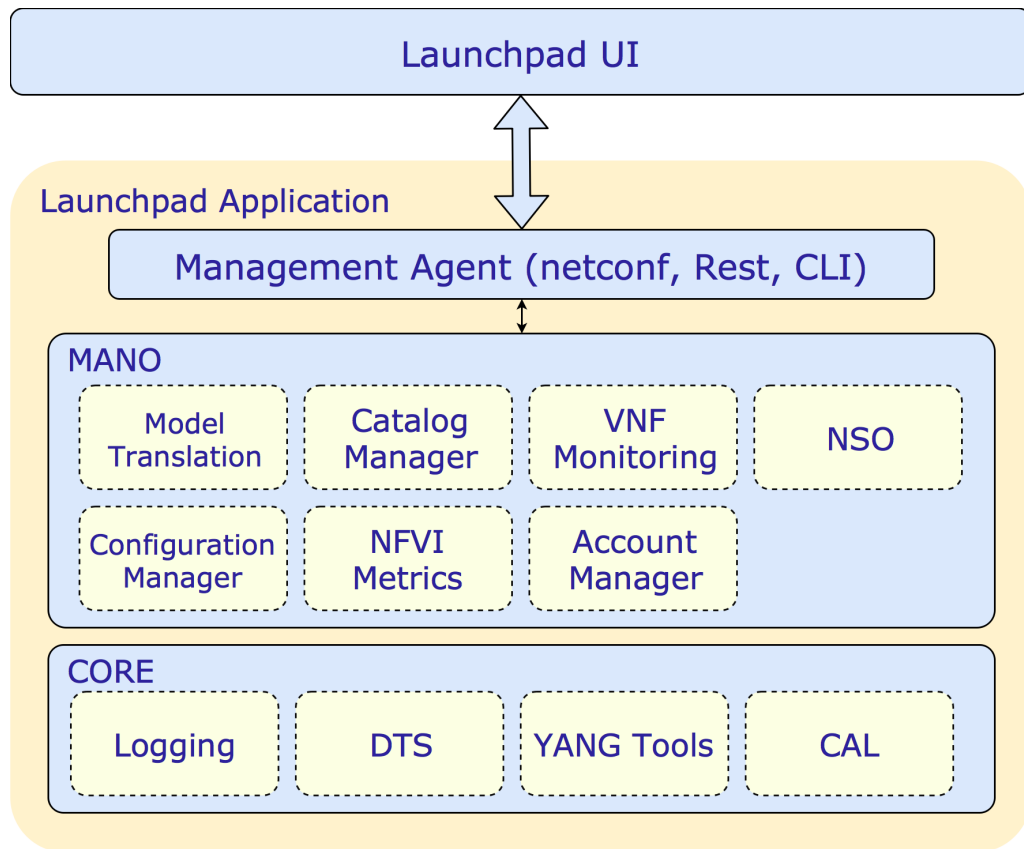


Figura 3.4: Architettura Launchpad dell'SO.

L'architettura di questo software è abbastanza complessa e costituita da numerosi moduli, noi restringeremo il campo alle funzionalità per noi di interesse. Le principali sono racchiuse nel modulo chiamato *Launchpad* ed illustrato in figura 3.4.

Tutti i moduli, compreso quello in figura 3.4, si poggiano su due strati architetturali, uno chiamato *CORE* e l'altro *MANO*. Il primo contiene i componenti base, su cui poggiano tutti i plug-in di gestione dell'orchestratore:

- *Logging*: sistema di logging per il modulo SO.
- *DTS*: Data Trasformation Services per il supporto alle basi di dati.
- *YANG Tools*: strumenti per la gestione del Data Model YANG.
- *CAL*: insieme di moduli interfaccia per i vari VIM.

Il secondo, contiene tutta una serie di plug-in, che espletano le funzionalità del Service Orchestrator descritte in precedenza, questi sono:

- *Model Translation*: sistema di logging per il modulo SO.
- *Catalog Manager*: gestori dei cataloghi di VNFD e NSD, corrispondenti in sostanza ai descrittori.
- *VNF Monitoring*: modulo di monitoraggio, verrà brevemente descritto in seguito.
- *NSO*: plug-in che permette l'interazione con il Resource Orchestrator OpenMANO.
- *Configuration Manager*: gestore della configurazione delle VNF, in sostanza l'interazione con il VNFM Juju.

- *NFVI Metrics*: metriche relative all'infrastruttura NFVI, potenzialmente utilizzate per gestione di eventi o scalamento/terminazione/riallocazione delle VNF.
- *Account Manager*: gestione degli accessi alla gestione di OSM, via GUI o CLI. Questi dettagli verranno affrontati in seguito nell'appendice B.

Tornando al Launchpad, questo, a partire dai plug-in descritti, offre un Agent che accetta richieste secondo diversi protocolli e modalità (Netconf, REST, CLI) e le propaga opportunamente agli strati inferiori. Tali richieste riguardano nello specifico l'istanziamento di un servizio di rete, delle VNF e dei moduli di gestione, a partire dai descrittori e dai file di supporto caricati nella fase di onboarding, fase che si presuppone già eseguita al momento del lancio del servizio di rete.

Scopo di questo capitolo è dare una panoramica sull'architettura degli orchestratori, in seguito, quando andremo ad implementare la nostra soluzione, vedremo come questi moduli effettivamente interagiscono, mostrando vari workflow (cap. 7).

Architettura complessiva

Andiamo ora a riassumere l'architettura OSM attraverso due immagini, la prima in figura 3.5 dal punto di vista dello standard, la seconda in figura 3.6 come sintesi ad alto livello.

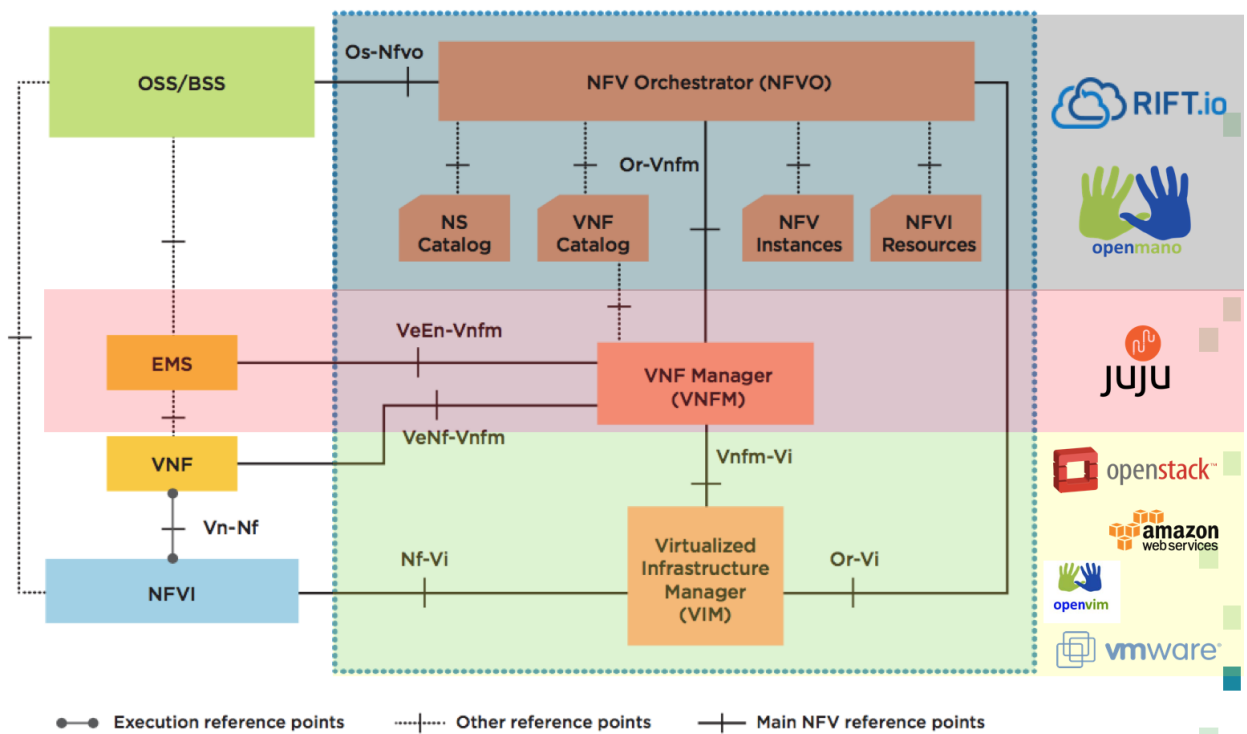


Figura 3.5: Mapping di OSM con ETSI NFV MANO.

Nel primo caso, abbiamo il mapping dei software costituenti e compatibili con OSM. Dall'alto in figura 3.5 quelli relativi all'orchestrazione (Riftware e OpenMANO), poi quelli relativi alla gestione delle VNF (Juju) ed infine i VIM compatibili (OpenStack, AWS, OpenVIM e VMware).

Nel secondo caso abbiamo il framework completo, composto dall'SO, dall'RO e dal VNFM, illustrati in figura 3.6 e delimitati dall' "OSM scope". Nella stessa figura, è interessante notare come questi tre elementi si interfaccino con l'infrastruttura NFVI, delimitata dalla linea tratteggiata. Nella parte bassa della figura, infatti, sono rappresentati, sulla sinistra l'NFVI (i Compute Node, nodi fisici dove vengono allocate le VNF) e sulla destra due dei VIM compatibili con OpenMANO. Questi ultimi, guidati dall'RO, gestiscono i nodi fisici, allocando le risorse necessarie per le VNF.

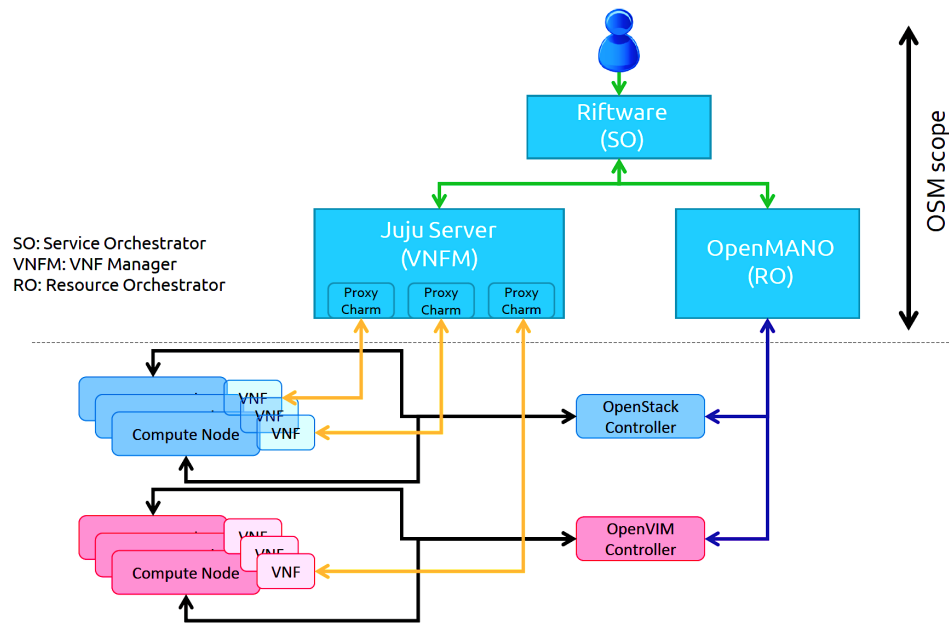


Figura 3.6: Architettura complessiva OSM (fonte: [osm wiki](https://wiki.osm.eu/)).

Successivamente Juju, attraverso i suoi *proxy charm* (corrispondono all'EM in NFV MANO), che vedremo in seguito, configurano e gestiscono le funzioni di rete virtuali.

Open Source MANO, per tutta una serie di motivi, è stato scelto per implementare un caso d'uso in questo lavoro di tesi, quindi dettagli ulteriori verranno dati successivamente (cap. 6).

Moduli sperimentali

Esistono due moduli sperimentali, della *release OSM THREE*, che descriviamo brevemente di seguito:

- *Monitoring Module (MON)*: l'ultima versione di OSM prevede questo modulo di monitoraggio, che si pone l'obiettivo fondamentale di non essere in competizione, ma di supporto a tutta una serie di strumenti di monitoring esistenti: OpenStack Aodh, OpenStack Gnocchi, Amazon CloudWatch e VMware vRealize. Tale modulo, in sostanza, coordina e configura gli strumenti esterni appena citati e definisce una serie di eventi azionabili. Questi eventi possono essere lanciati dagli strumenti esterni stessi o dagli NS e dalle VNF che sono online. Per l'implementazione del bus dei messaggi è stato utilizzato Apache Kafka, una piattaforma di stream processing a bassa latenza e alta velocità [18].
- *Multi-PoP NFVI/VIM Emulation Platform (VIM EMU)*: tipicamente i servizi di rete sono complessi e richiedono, in taluni casi, l'utilizzo di più PoP. La difficoltà di valutare il corretto funzionamento, di fronte a tale complessità, dei servizi dispiegati, ha portato OSM a supportare una piattaforma per l'emulazione di un VIM Multi-PoP. Questo permette, prima di istanziare su larga scala un servizio di rete, di effettuare dei test su una piattaforma contenuta (anche un semplice laptop). Tale emulatore lega tecnologie quali Docker, Mininet e containernet, attraverso il progetto Sonata [19]. Nei capitoli implementativi (cap. 9) faremo qualche esempio e daremo maggiori dettagli su questo progetto.

Integrazione con SDN

Open Source MANO di base consente l'utilizzo di diversi SDN controller, per supportare il discorso di integrazione con il Software Defined Networking, discusso nel precedente capitolo 2. Elenchiamo di seguito i controller supportati:

- Floodlight
- ONOS
- OpenDayLight

Come sempre andremo successivamente a dettagliare più in profondità le caratteristiche e le funzionalità avanzate nei capitoli di implementazione (cap. 9).

3.3.2 Open Baton

Open Baton è una piattaforma open source, allineata allo standard ETSI NFV MANO, per la gestione ed orchestrazione delle funzioni di rete virtuali.

Non scenderemo troppo nei dettagli, ma ci limiteremo ad illustrare quali sono i componenti architetturali principali e i vantaggi di tale piattaforma.

Innanzitutto come si evince dall'architettura in figura 3.7, Open Baton è composto da un orchestratore (NFVO) e da diversi moduli software, collegati tra loro dal *message broker* RabbitMQ. Tale software fornisce strumenti per la comunicazione tra i vari componenti, attraverso il protocollo *Advanced Message Queuing Protocol (AMQP)*. I moduli software sono divisi rispetto alla loro area di gestione: OSS, NFVI e VNF Managers.

Operational Support System

Questo sottoinsieme di componenti software comprende un modulo di gestione dei guasti (FM system), un motore per lo scalamento automatico (AE system), un motore per il *Network Slicing* (NSE) e un *Service Function Chaining Orchestrator* (SFCO). Per Open Baton, questi sono tutti strumenti di supporto, che possono essere tutti o in parte omessi a seconda delle esigenze. Ad esempio il *Network Slicing* entra in gioco solo quando ho a che fare con un'orchestrazione multi-dominio, ovvero con più porzioni (slices) di rete e servizi da gestire con diversi orchestratori. In tal caso è necessaria un'orchestrazione gerarchica, che non è argomento di questo lavoro di tesi.

VNF Managers

Qui vi sono i sottomoduli che si occupano della gestione delle VNF. Nel caso di Open Baton vi sono diversi *VNFM adapter* a seconda del tipo di VIM utilizzato e delle esigenze specifiche. Una delle possibili scelte ricade su Juju come nel caso di Open Source MANO, ma sono possibili anche altre scelte come l'utilizzo del *Docker VNFM Adapter*, per un'infrastruttura basata su container, ed esiste, infine, la possibilità di sviluppare il proprio VNF Manager, questo attraverso delle librerie messe a disposizione da Open Baton in Java o Python.

Gestione NFVI

Per la gestione dell'infrastruttura NFVI occorrono, ovviamente, dei VIM e, per adattarli all'architettura di Open Baton, dei VIM Driver. Qui abbiamo il punto forse più interessante di questo orchestratore: dei driver specifici per il deploy delle VNF su docker e come visto nella sezione precedente uno specifico VNFM per questo. Infine, anche Open Baton come OSM, prevede una serie di strumenti per il monitoraggio, nel caso specifico Zabbix.

Analisi conclusive

Open Baton ha sostanzialmente due vantaggi, uno dettato dall'avanguardia nel proporre dei driver e un manager per docker, l'altro nella possibilità di avere dei VNF Manager generici con degli

Element Management System (EMS, equivalente all'EM presentato in precedenza) interni alle VNF, così da evitare l'overhead nell'istanziamento di ulteriori VM o container.

Il rovescio della medaglia sta nel fatto che l'integrazione con SDN non è prevista, quindi, per la gestione della connettività a livello 2, c'è molta meno flessibilità rispetto ad OSM. Per ogni ulteriore dettaglio di interesse è possibile consultare il sito del progetto [20].

3.3.3 OPNFV

Open Platform for NFV (OPNFV) è un progetto volto alla creazione di una piattaforma, che acceleri l'adozione di NFV. Tale progetto open source risulta molto interessante e in rapida crescita, per questo motivo accenniamo qualcosa dell'architettura e dei vantaggi.

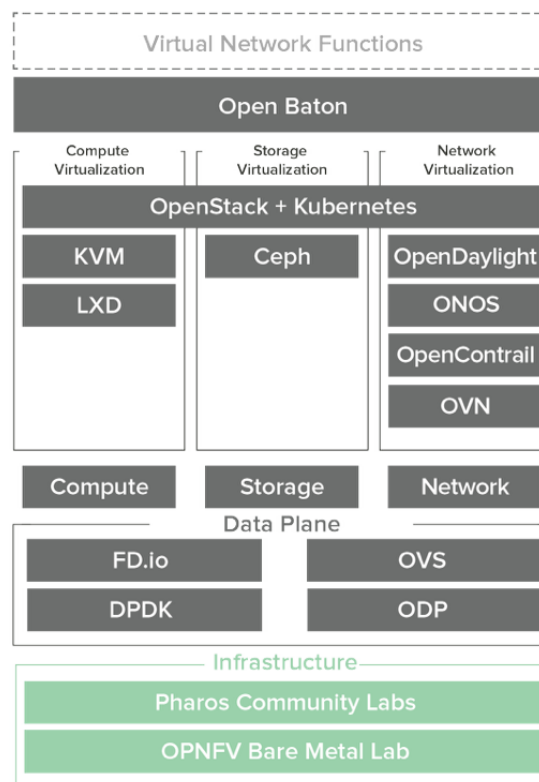


Figura 3.8: Architettura OPNFV (fonte: [opnfv](https://opnfv.org)).

In figura 3.8 è illustrata l'architettura di OPNFV su più livelli, dall'infrastruttura all'orchestrazione delle funzioni di rete. La prima nota sta nel fatto che, a differenza dei precedenti progetti, questa piattaforma mira a creare un'infrastruttura complessiva, a prescindere dal sistema di orchestrazione utilizzato. Questo vuol dire che si preoccupa di problemi molto seri di NFV, come l'analisi delle prestazioni carrier-grade e l'impatto di un'infrastruttura NFV su larga scala, argomenti che vengono di solito, almeno in parte, accantonati da molte piattaforme presenti sul mercato in modo open.

A partire dal basso, sempre in figura 3.8, abbiamo l'infrastruttura composta da due livelli: *OPNFV Bare Metal Lab* e *Pharos Community Labs*. Questo è di forte interesse perché non solo OPNFV si occupa dell'infrastruttura software, ma, con il progetto “Pharos”, è in grado di mettere a disposizione un'infrastruttura hardware distribuita geograficamente, eterogenea e composta da server appartenenti a 16 laboratori in tutto il mondo. Questa struttura così eterogenea crea un *test bed* eccezionale per il testing dei servizi di rete e il miglioramento del codice di OPNFV.

Salendo di un livello abbiamo il *Data Plane*, questo sfrutta alcune tecnologie come Open vSwitch (OVS), OpenDataPlane (ODP), Data Plane Development Kit (DPDK) e The Fast Data Project

(FD.io). Qualcuna tra queste tecnologie ritornerà successivamente nei prossimi capitoli (cap. 9), nel frattempo concentriamoci su qualcosa di grande interesse: la gestione dei domini dell'hypervisor.

È particolarmente interessante, infatti, l'insieme di soluzioni adottate per ognuno dei domini presentati nel precedente capitolo 3: compute, storage e network. In primis, per quanto riguarda la virtualizzazione delle funzioni di rete, vengono utilizzate due tecnologie: KVM e LXD. La prima *Kernel-based Virtual Machine* è un'infrastruttura di virtualizzazione del kernel linux, che supporta la virtualizzazione completa tramite le tecnologie *Intel-VT* o *AMD-V*. La seconda è un hypervisor basato su container, in particolare sulla tecnologia *LXC (Linux container)*. LXD offre delle funzionalità molto simili a quelle delle VM, ma con una velocità e densità nettamente superiori. Per quanto riguarda lo storage vi è l'utilizzo di *Ceph*, una piattaforma libera per lo storage su un cluster distribuito, che si occupa nativamente della replicazione dei dati e della ripresa in caso di guasti. Infine, per quanto riguarda il Control Plane della rete, vengono utilizzate tecnologie quali OpenDayLight, ONOS, OpenContrail e Open Virtual Network (OVN), di queste tecnologie ne vedremo alcune in dettaglio successivamente.

Una volta descritte le tecnologie dei domini dell'hypervisor, andiamo a valutare quali sono i VIM che vengono presi in considerazione, ovvero OpenStack e Kubernetes. Il primo rappresenta una tecnologia *IaaS* molto diffusa e che approfondiremo nei capitoli di analisi dell'implementazione (cap. 6). È costituito da un insieme di moduli applicativi e supporta principalmente infrastrutture basate su VM. Il secondo è *Kubernetes*, un progetto open source per il dispiegamento, scalamento e la gestione automatici di servizi su container. Nato dal progetto *Borg* di Google, Kubernetes è uno dei più promettenti orchestratori di risorse su container attualmente sviluppati ed è compatibile con numerose tecnologie di container, ad esempio LXD e Docker.

Un ultimo sguardo va alla parte di orchestrazione, infatti, tale piattaforma utilizza software di terze parti per la gestione ad alto livello. In parole povere, alcuni dei servizi di NFV MANO descritti in precedenza vengono affidati ad orchestratori esterni, tra quelli compatibili abbiamo: Open Baton, Open-O e OSM.

In ultima analisi questa risulta una delle piattaforme più promettenti allo stato attuale, l'unico risvolto negativo è la difficoltà nell'utilizzarla per scenari contenuti di testing, in quanto i requisiti di sistema richiesti per l'installazione di tale piattaforma sono ingenti.

3.3.4 NFV over Open DC/OS

L'ultimo progetto di cui discutiamo è un concept di un istituto di ricerca in Taiwan [21], che mira ad utilizzare Open DC/OS di Mesosphere come orchestratore e gestore di risorse in un'architettura NFV.

Innanzitutto DC/OS è un sistema operativo basato su Apache Mesos e consente di gestire diverse macchine come una singola. DC/OS automatizza la gestione delle risorse, lo scheduling dei processi e semplifica l'installazione e la gestione dei servizi distribuiti.

Per capire di cosa si tratta dobbiamo fare un passo indietro e parlare di Apache Mesos [22]. Questo è un progetto nato per gestire cluster di computer. Come si evince dalla figura 3.9, sono presenti un nodo master e dei nodi slave; il primo ha un processo demone che si occupa della gestione degli slave; gli altri sono degli esecutori e hanno un agent che comunica con il master. Esiste poi la possibilità di integrarlo con Marathon [23], un framework per l'orchestrazione e la gestione di container. Quest'ultimo permette, attraverso il nodo master di Apache Mesos, di inizializzare, terminare e scalare dei container, ad esempio basati su Docker come in figura 3.9. Marathon espone una REST API per consentire tutte le operazioni sopra citate. Sempre in figura 3.9, esistono altri componenti accessori, ma rimandiamo per ulteriori dettagli alla pagina del progetto [22].

I componenti appena descritti sono alla base del sistema operativo di cui sopra ed esauriscono gli elementi che ci servono per presentare l'architettura in figura 3.10. L'interesse come si evince da tale immagine, non è rivolto all'utilizzo di DC/OS come un NFV MANO, ma come gestore dell'infrastruttura NFVI (VIM, Mesos Master) e come VNFM (Marathon). Infatti attraverso il Mesos Master potremmo gestire diversi nodi fisici dell'infrastruttura NFVI e con Marathon guidare l'allocazione e gestione dei container su tali nodi. Il tutto potrebbe essere gestito ancora

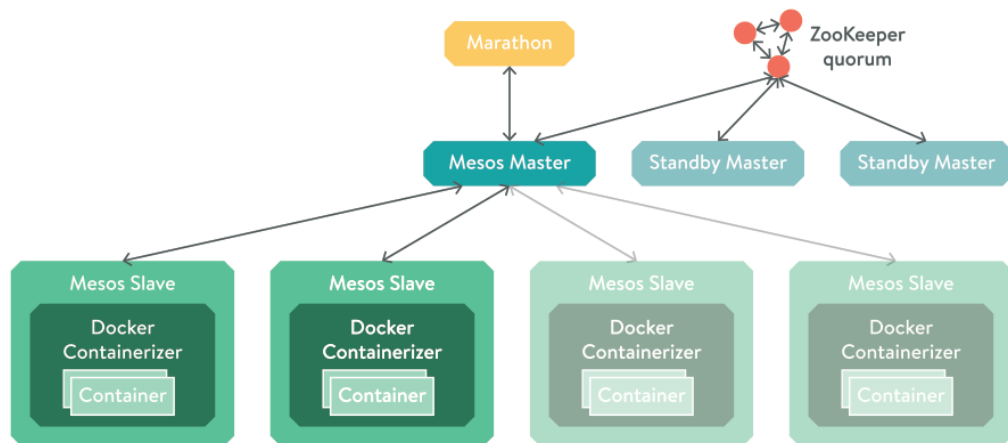


Figura 3.9: Apache Mesos con Marathon (fonte: [livewyer](#)).

a livello superiore da uno degli NFV MANO sopra descritti, in modo semplice e attraverso le API REST di Marathon. Questo è possibile in considerazione del fatto che i framework di gestione ed orchestrazione che abbiamo presentato, in particolare OSM e Open Baton, già offrono la possibilità di sviluppare dei plug-in che fungano da VIM Driver e VNF Manager; risultano quindi già pensati per collegare Marathon e Mesos a quello che in precedenza abbiamo chiamato Service Orchestrator.

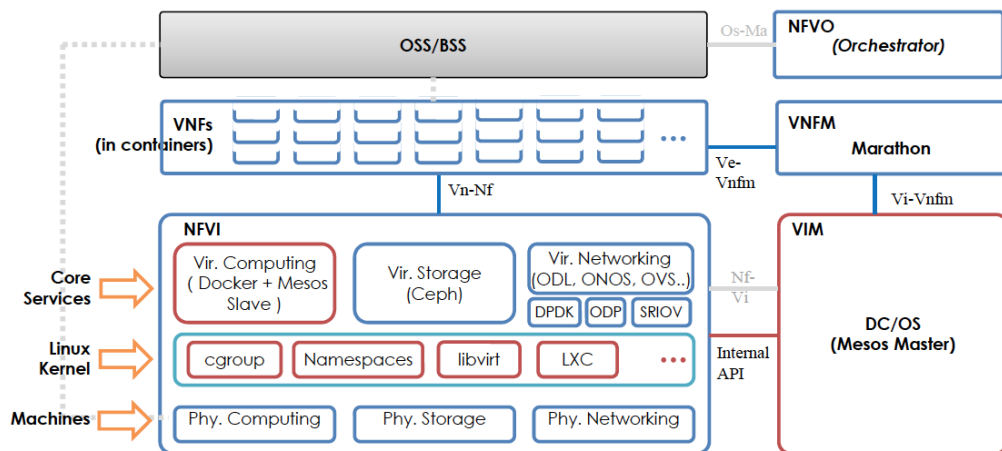


Figura 3.10: Architettura di NFV over Open DC/OS (fonte: [\[21\]](#)).

Capitolo 4

Policy di sicurezza

In questo capitolo approfondiremo il concetto di Policy e lo contestualizzeremo all'interno del progetto SECURED.

4.1 Il concetto di Policy di sicurezza

Come si evince da [24] una possibile definizione di policy viene data da D.Clark e D.Wikson nell'articolo [25]:

“Una policy di sicurezza specifica gli obiettivi legati alla sicurezza che il sistema deve raggiungere e le minacce alla quali deve resistere. Per esempio, un obiettivo di sicurezza ad alto livello spesso specifica che il sistema dovrebbe prevenire la rivelazione o sottrazione non autorizzata di informazioni, dovrebbe prevenire la modifica non autorizzata di informazioni, e ancora dovrebbe prevenire la negazione del servizio.”

Un'altra possibile definizione è stata presentata da D.C. Robinson e M.S. Sloman nell'articolo [26]:

“Una policy di gestione definisce l'insieme di regole per raggiungere alcuni obiettivi. Ad esempio, una policy di controllo degli accessi è un insieme di regole che definiscono le risorse a cui un utente può accedere e una policy di gestione dei guasti definisce dove un guasto deve essere riportato e alcune azioni di ripresa. La policy è definita dall'amministratore di sistema.”

Una definizione attuale di policy viene data dall'IETF (Internet Engineering Task Force) nell'RFC-3198 [27]:

“Una policy può essere definita da due prospettive: (i) un obiettivo definito, percorso o metodo di azione per guidare o determinare le decisioni presenti e future. Le policy sono implementate o eseguite in un contesto particolare (come le policy definite in un'unità aziendale); (ii) le policy come un insieme di regole per amministrare, gestire e controllare l'accesso alle risorse di rete”.

4.1.1 Introduzione alla terminologia

L'RFC-3198 [27] offre anche delle definizioni e spiegazioni relative a termini nell'ambito delle politiche di sicurezza, ne riportiamo alcuni utili di seguito:

- **Policy rule (Regola):** è il tassello fondamentale di un sistema basato su policy. Una regola è concorde al paradigma ECA (Evento-Condizione-Azione). Una regola in tale paradigma può essere espressa nella forma: “quando l'Evento occorre, se la Condizione è verificata, allora compi l'Azione”.

- **Policy event (Evento):** è definito come qualsiasi importante occorrenza nel tempo, relativa ad un cambiamento del sistema gestito, e/o nell'ecosistema relativo al sistema gestito. Viene utilizzato per determinare se la Condizione di una Policy Rule può essere valutata o meno. Alcuni esempi di Evento includono operazioni svolte dall'utente (es. logon, logoff, aggiunta, modifica, cancellazione).
- **Policy condition (Condizione):** è una rappresentazione dello stato necessario e/o dei prerequisiti che definiscono se le azioni di una Policy Rule devono essere eseguite. Quando le condizioni associate ad una policy sono valutate TRUE, allora le regole, in accordo anche con altre strategie di decisione e priorità, devono essere applicate.
- **Network fields (Campi di rete):** rappresentano i possibili valori dei corrispondenti campi in un pacchetto che combaciano con una regola. Un esempio può essere quello degli header del pacchetto (IP sorgente, IP destinazione, numero della porta). Alcune di queste informazioni possono essere utili a designare gli eventi e le condizioni da gestire.
- **Policy decision (Decisione):** ci sono due prospettivi riguardo alla “decisione”: (i) una prospettiva di processo, coincidente con la valutazione della condizione di una regola; (ii) una prospettiva di risultato, coincidente con l'applicazione delle azioni, quando le condizioni di una policy sono TRUE.
- **Policy enforcement (Applicazione delle regole):** rappresenta l'esecuzione delle decisioni della policy.
- **Policy action (Azione):** è la definizione di cosa dev'essere fatto per applicare una regola, quando le condizioni di quest'ultima sono positive. Le azioni di una policy possono sfociare nell'esecuzione di una o più operazioni per modificare e configurare il traffico e le risorse di rete.
- **Policy abstraction (Astrazione):** una policy può essere rappresentata a livelli diversi, a partire dagli obiettivi operativi alla configurazione specifica di un dispositivo. La traduzione tra i vari livelli di “astrazione” potrebbe richiedere informazioni al di fuori del contesto della policy, come ad esempio i parametri di configurazione della rete e degli host.
- **Security controls:** sono delle applicazioni o moduli software in una rete. Questi implementano le funzionalità necessarie all'applicazioni di una policy di sicurezza di rete. I Security controls possono ispezionare il traffico della rete e bloccare alcuni pacchetti, o in alternativa modificarli, cambiando le informazioni dell'header. Come esempio, un packet filter, un firewall stateful e un firewall a livello applicazione sono tutti strumenti utilizzati per controllare il traffico, mentre un gateway IPsec, un VPN (Virtual Private Network) terminator e dispositivi NAT/NATP sono capaci di modificare il traffico.
- **Policy refinement:** è il processo che determina le risorse necessarie per soddisfare i requisiti della policy, per tradurre le policy di alto livello in configurazioni di basso livello che possono essere applicate dal sistema e per verificare che l'insieme di policy di basso livello o configurazioni sposi i requisiti delle policy di alto livello.
- **Policy group (Gruppo):** è una classe che rappresenta un contenitore, aggregando sia regole di policy che altri gruppi. Questo permette di raggruppare regole all'interno di una policy.

Questo serviva a dare un'idea complessiva della terminologia e di alcune definizioni nell'ambito delle policy. In figura 4.1 abbiamo un esempio di una regola.

4.2 Policy-based management (PBM)

Una volta introdotta la terminologia di base andiamo a vedere come è strutturato un sistema di gestione basato su policy.

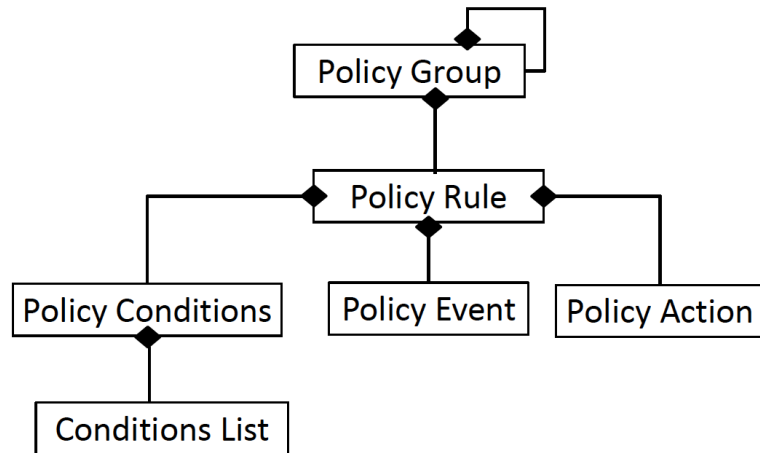


Figura 4.1: Struttura di una Policy Rule (fonte: [24]).

4.2.1 Architettura

Una delle più popolari architetture di Policy Based Management (PBM) è quella proposta dall'IETF [28], questa si basa su quattro elementi fondamentali:

- **Policy Management Tool (PMT):** è un'entità che viene utilizzata dagli amministratori per specificare le policy da applicare alla rete.
- **Policy Repository (PR):** le policy che vengono generate dal PMT vengono memorizzate all'interno di questa entità.
- **Policy Enforcement Point (PEP):** l'elemento dove le policy vengono applicate (Policy enforcement).
- **Policy Decision Point (PDP):** questo è l'elemento più importante dell'architettura, poiché questo modulo è quello che prende le decisioni basate sulle policy dal PR e le comunica al PEP.

Una sintesi dell'architettura è presentata in figura 4.2, dove si evince che i moduli presentati comunicano attraverso diversi protocolli (LDAP, SNMP e COPS). Le policy introdotte nel PMT risponderanno al paradigma ECA (Evento-Condizione-Azione) e avranno altri componenti come i soggetti, i target e gli eventi scatenanti.

4.2.2 Modellazione delle policy

Come riportato dall'articolo [29], la maggior parte dei sistemi di gestione basati su policy definisce la stessa come una singola entità a sé stante. Questo concetto non risulta molto efficace nel caso di sistemi di gestione di rete su larga scala, o in generale per i sistemi complessi. Il motivo è dato dal fatto che le policy, anche curando aspetti diversi (gestione operativa, gestione clienti, configurazione dispositivi, etc.), presentano numerose correlazioni e dipendenze. Questo ci suggerisce che, prese individualmente le policy incapsulano un aspetto specifico (o una vista) del sistema complessivo, prese insieme, invece, collaborano per forzare il sistema verso un determinato comportamento desiderato. A questo punto può essere di interesse, come si evince in figura 4.3 suddividere in vari livelli la modellazione delle policy, consentendo quindi di poter raggruppare le stesse a seconda di una particolare vista del sistema complessiva. Dall'alto le policy di alto livello, fino ad arrivare a

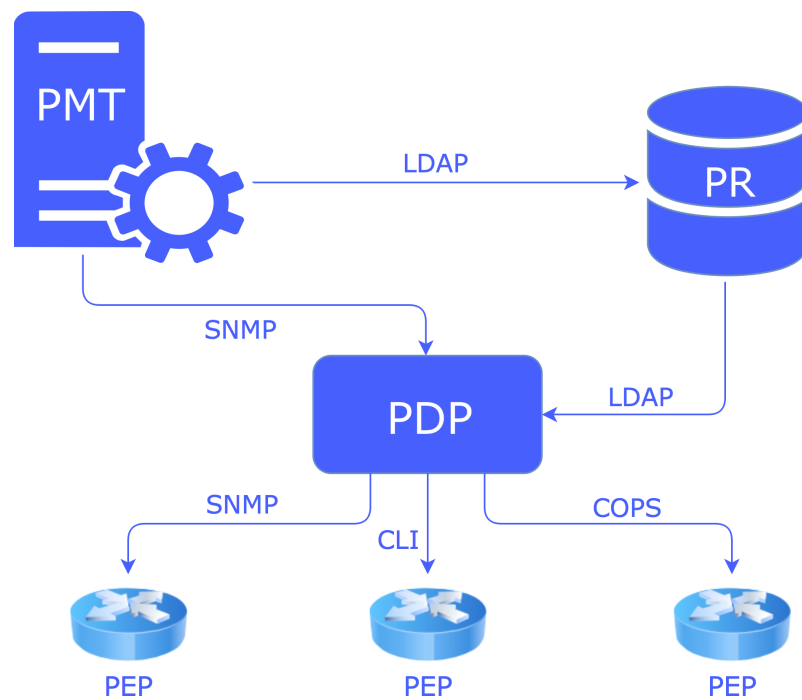


Figura 4.2: Architettura PBM dell'IETF.

quelle di basso livello e ai dispositivi da configurare per ottenere il comportamento desiderato dal sistema.

Andiamo adesso a vedere una proposta di modello di creazione di policy [30], che ci darà un'idea su come stratificare le policy e quali sono i vari attori in gioco.

Livelli di astrazione

Come accennato, in un sistema PBM le policy possono essere rappresentate e specificate in modi diversi, al fine di ottenere la scalabilità e flessibilità necessaria dal sistema. Il modello presentato propone i seguenti livelli di astrazione:

- *Le policy di alto livello*, queste policy includono:
 - gli obiettivi della compagnia o dell'organizzazione;
 - il Service Level Agreement (SLA) tra provider oppure tra provider e clienti;
 - le esigenze di chi è coinvolto nell'utilizzo della rete: utenti, applicazioni, servizi, provider e operatori di rete.
- *Le policy di medio livello*: queste sono le policy che sono applicate dagli amministratori nel PMT attraverso le regole. Il tutto utilizzando o il linguaggio specifico per la policy o utilizzando un'interfaccia (grafica o da console). Le regole seguono lo stesso paradigma definito in precedenza.
- *Le policy di basso livello*: queste policy sono definite per il dispositivo specifico, con una specifica configurazione e rappresentano il livello più basso di una policy perché sono applicate direttamente all'elemento di rete coinvolto nella policy (es. PEP). Queste policy sono concepite in un linguaggio e formato comprensibili solo al dispositivo specifico, per questo vengono tradotte in tale formato dal PDP. Solitamente queste policy non vengono specificate, perché questo farebbe perdere l'obiettivo di gestione complessivo, dovendo sopperire ad una enorme quantità di dettagli provenienti da dispositivi tutti diversi.

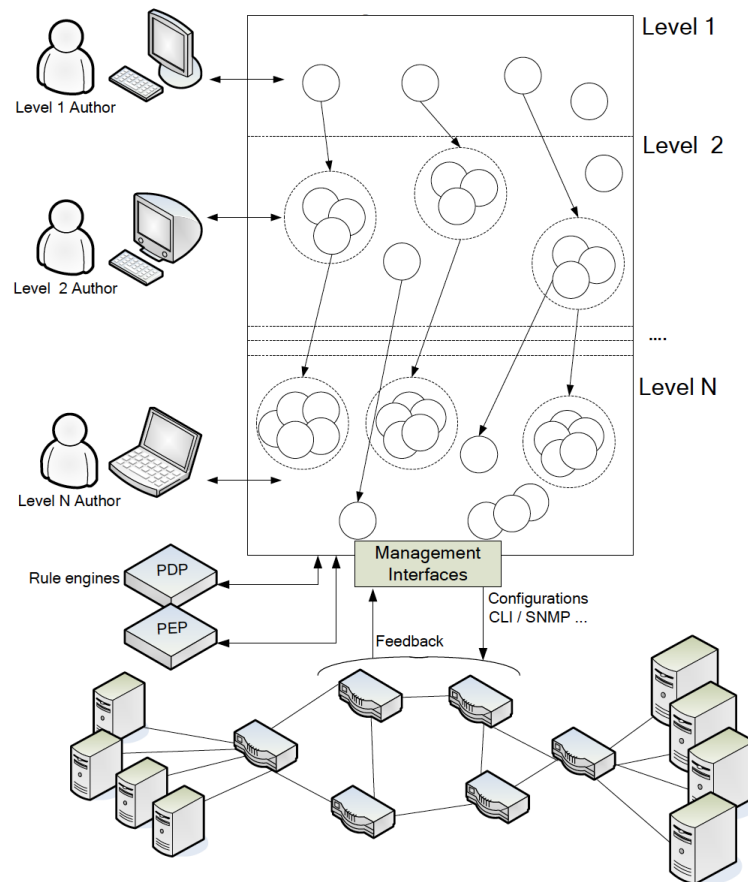


Figura 4.3: Policy modellate su più livelli (fonte: [29]).

Processo di creazione delle policy

Vediamo adesso il processo di creazione della policy dove isoliamo tre fasi fondamentali:

- *Definizione delle policy astratte di alto livello:* In questa fase sono presi in considerazione gli obiettivi di gestione, ovvero i requisiti del sistema PBM. In questa fase vengono presi in considerazione dettagli per il monitoraggio e la manutenzione delle policy.
- *Policy refinement:* per ottenere gli obiettivi delineati nella fase precedente, occorre raffinare le policy scendendo di un livello di astrazione, traducendole quindi in delle operazioni da effettuare per soddisfare tali obiettivi. Questa fase fa uso di ulteriori dettagli: analisi dei requisiti, le applicazioni che sono istanziate sulla rete, gli utenti e le risorse a disposizione.
- *Definizione e specifica delle Policy rule:* In questa ultima fase le policy sono ancora tradotte in obiettivi di basso livello (come la configurazione di un dispositivo specifico). Queste policy possono essere espresse in linguaggi di specifica di policy (es. Ponder, XACML, etc.) o iniettate nei dispositivi attraverso delle interfacce grafiche o da console.

Attori

Gli attori in gioco nella creazione di un sistema SBM sono i seguenti:

- *Designer di Policy:* queste persone sono coloro i quali progettano e formalizzano le policy astratte di alto livello. Tra questi abbiamo:

- *manager*: conoscono le regole, gli obiettivi e l'organizzazione della struttura;
 - *specialisti di rete*: Conoscono le tecnologie a basso livello che possono essere utilizzate (Meccanismi per la qualità del servizio, sicurezza e tecnologie di rete);
 - *amministratori*: conoscono come definire gli obiettivi di gestione e cosa supporta il sistema basato su policy.
- *Amministratori delle policy*: sono i responsabili dell'implementazione delle policy sul sistema PBM. Si occupano della loro gestione e della manutenzione dei meccanismi di policy enforcement. Risulta loro responsabilità provvedere al fornire record periodici ai manager e specialisti di sicurezza.
 - *Utenti*: viene loro richiesto quali sono i requisiti necessari dalla loro prospettiva, in quanto risultano gli utilizzatori finali dell'infrastruttura, in termini di servizi e applicazioni.

4.2.3 Policy Management Tool

Risulta di interesse, una volta descritti i vari livelli di astrazione delle policy e come esse vengono specificate, andare a dettagliare qual'è l'architettura e quali sono le funzionalità offerte da un Policy Management Tool [31].

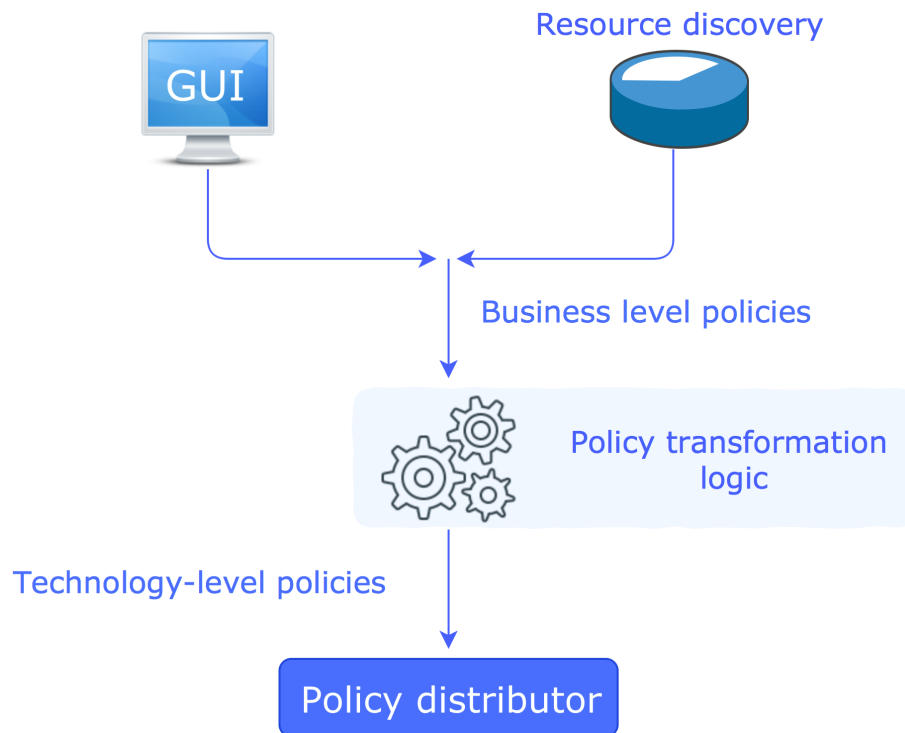


Figura 4.4: Policy management tool.

Come si evince dalla figura 4.4 i tipici moduli architetturali che compongono un PMT sono:

- *User Interface*: l'interfaccia utente, che può essere sia da command line che grafica, consente agli amministratori di inserire le *business level policies*, ovvero le policy ad alto livello.
- *Resource discovery*: è un componente che determina la topologia di rete, gli utenti e le applicazioni a disposizione.
- *Policy transformation logic*: è il componente responsabile della verifica della consistenza e correttezza delle policy inserite dagli amministratori di sistema, anche in relazione alla topologia di rete esistente. Inoltre tale componente, successivamente alla fase di validazione,

traduce le policy di business level in quelle da distribuire ai vari dispositivi della rete, questo tipicamente in un linguaggio di medio livello, che sarà successivamente tradotto dal Policy distributor o dal PDP.

- *Policy distributor*: è responsabile della distribuzione delle policy ai vari dispositivi della rete. A seconda che sia presa o meno in considerazione l'architettura IETF, questo strumento può occuparsi della traduzione delle policy in linguaggio adatto a configurare i dispositivi. Nel caso IETF assume questo comportamento, in caso contrario la traduzione delle policy da un linguaggio di medio livello a uno a basso livello potrebbe essere effettuata sul dispositivo stesso.

Questi sono i blocchi che si ritrovano in un Policy management tool o Policy manager, anche se il cuore fondamentale del concetto di PBM risiede nella logica di traduzione delle policy all'interno del modulo di trasformazione (fig. 4.4). È in questa logica che sono incapsulati tutti i concetti di rappresentazione e gestione delle policy.

Abbiamo finora fornito una panoramica di alto livello su come funziona un sistema di gestione basato su policy, nella prossima sezione forniremo maggiori dettagli su quali possono essere le logiche implementative, dando un esempio pratico nell'ambito del progetto SECURED.

4.3 PBM nel progetto SECURED

4.3.1 Introduzione al progetto SECURED

SECURED è stato un progetto europeo finanziato nell'ambito del programma FP7 (finanziamenti europei 2007-2013), che ha avuto inizio nell'ottobre del 2013 ed è terminato nel settembre del 2016. Molti partner, accademici e relativi ad aziende di telecomunicazione, hanno partecipato al progetto, tra i principali abbiamo HP, Telefonica, il Politecnico di Torino e l'università della Catalogna. Il nome "SECURity at the network EDge" esemplifica l'obiettivo di tale progetto, infatti, lo scopo era quello di far convergere tutti gli applicativi e gli strumenti di sicurezza dai terminali (PC, Smartphone, etc.) al primo nodo di rete disponibile (fig. 4.5). Questo consente di "scaricare" tali dispositivi dal carico e dalla responsabilità relativi alla sicurezza, assegnando questo compito ad un nodo sicuro ed affidabile, il *Network Edge Device (NED)*.

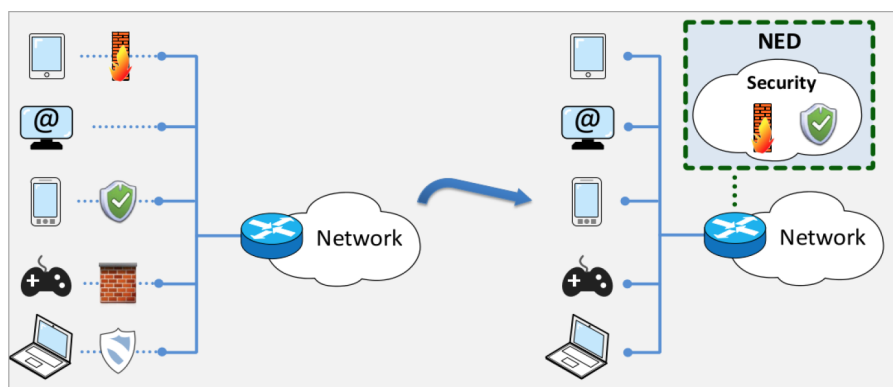


Figura 4.5: Modifica architetturale nella topologia di rete introdotta da SECURED (fonte: [progetto SECURED](#)).

Le componenti del modello proposto da SECURED sono tre:

- *NED*: nodo affidabile che è in grado di fornire all'utente un *Trusted Virtual Domain (TVD)*.
- *Personal Security Application (PSA)*: sono delle vNSF (Virtualized Network Security Functions), ovvero funzioni di rete virtuali progettate e configurate per scopi di sicurezza (Packet Filter, IDS, parental control, etc.)

- *Policy di sicurezza*: queste in SECURED sono utilizzate per gestire i meccanismi di deploy e configurazione delle PSA.

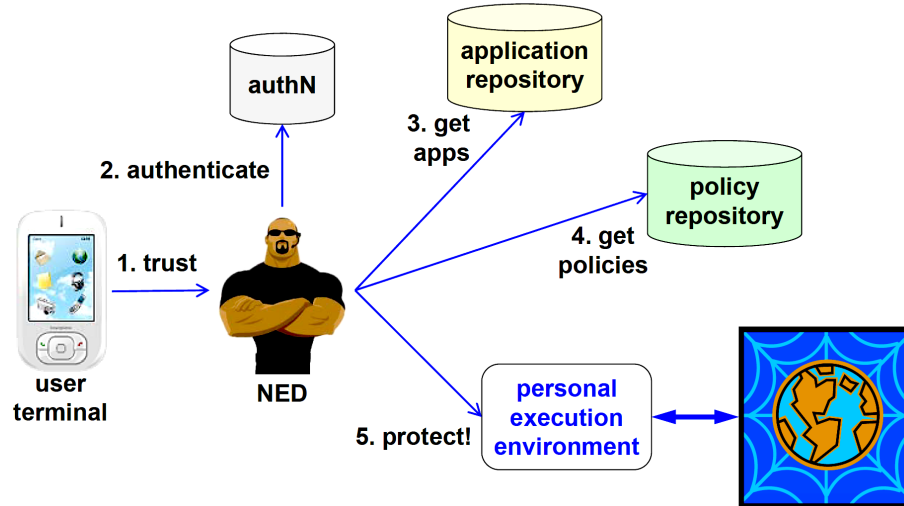


Figura 4.6: Architettura modello SECURED (fonte: [progetto SECURED](#)).

Come si evince dalla figura 4.6, per rendere sicuro l'accesso alla rete del terminale in questione si attraversano le seguenti fasi:

- *Trust*: il terminale dell'utente verifica, comunicando con il NED attraverso un canale sicuro, che lo stesso non sia stato compromesso e non presenti minacce (attraverso tecniche di Trusted Computing).
- *Authenticate*: in questa fase l'utente si autentica al NED attraverso il modulo di autenticazione authN all'interno del NED.
- *Get apps*: una volta finita la fase di autenticazione, il NED può reperire le PSA necessarie a fornire sicurezza all'utente dall'application repository, nel quale vengono messe in relazione le PSA desiderate con l'utente. Il NED conosce ovviamente i dati dell'utente dalla fase precedente.
- *Get policies*: sempre a partire dai dati dell'utente il NED può reperire le policy da scegliere attraverso il policy repository.
- *Protect*: una volta effettuate tutte le fasi preliminari, possono essere a questo punto istanziate tutte le PSA di sicurezza al fine di creare quello che in SECURED viene chiamato il *personal execution environment*.

Lo scenario tipico è rappresentato in figura 4.7, dove l'utente, una volta svolte le fasi sopra indicate, si ritrova un collegamento sicuro con la rete esterna. Tale collegamento prevede l'istanziamento di una service function chain, dove le singole VNF sono le PSA, che offrono i diversi servizi di sicurezza selezionati dall'utente. L'uso del paradigma NFV/SDN è tipico di tale approccio ed è in linea con quello presentato nei capitoli precedenti (cap. 2 e 3).

Noi non lavoreremo nello specifico contesto del progetto SECURED, ma ci è utile presentarlo, in quanto utilizzeremo buona parte del modello di gestione delle policy implementata in tale contesto.

4.3.2 Modellazione delle risorse di sicurezza

Le risorse a disposizione del sistema per attuare i meccanismi di sicurezza sono le PSA, che noi chiameremo in modo più generico vNSF. Il modello che utilizza SECURED per astrarre queste

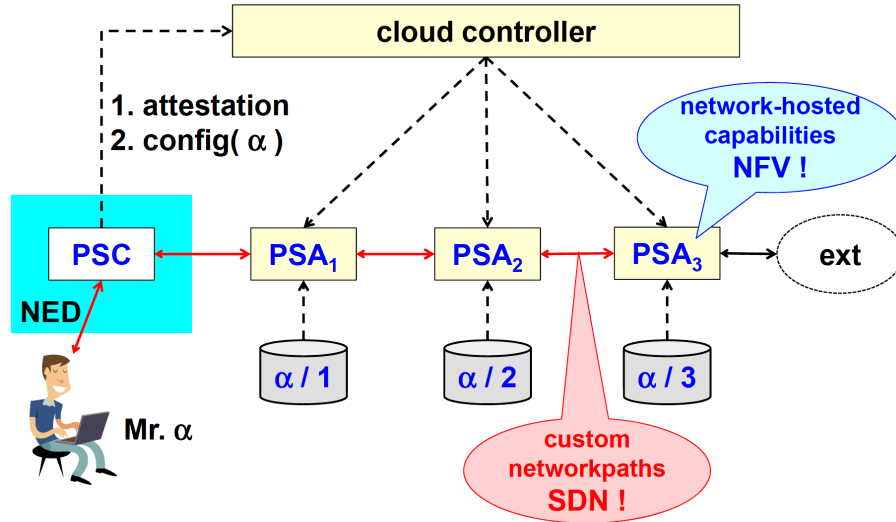


Figura 4.7: Approccio di sicurezza in SECURED (fonte: [progetto SECURED](#)).

funzionalità è basato sul concetto di **Capability**. Una **Capability** è una funzionalità astratta di sicurezza che corrisponde tipicamente ad un modulo software all'interno di una vNSF. La classe **Capability** è strettamente legata a quella **ITResource**, quest'ultima è la rappresentazione del modulo software specifico (es. Netfilter, Iptables) da utilizzare per ottenere quella **Capability**. Il mapping tra queste due classi viene effettuato attraverso il processo di policy refinement in SECURED, che rappresenta un punto centrale, in quanto permette di valutare come ottenere un determinato comportamento, descritto inizialmente dalle policy di alto livello, attraverso l'utilizzo di un insieme di **ITResource**.

Il modello proposto da SECURED è rappresentato nelle figure 4.8 e 4.9. Tale modello parte dalla classe astratta **Capability** e viene declinato successivamente in tutte le classi che la estendono, le quali rappresentano in ultima analisi una classe o una specifica funzionalità di sicurezza. Una nota sta nel fatto che ogni classe presenta l'attributo **may affect**, il quale indica se una **Capability** può o meno influenzarne un'altra. Nel modello astratto di SECURED è possibile isolare due sottoinsiemi di classi: un primo livello di estensione con le classi astratte e dei livelli successivi con funzionalità specifiche. Le classi astratte racchiudono funzionalità di sicurezza e le elenchiamo e presentiamo di seguito:

- **AddressTranslationCapability;**
- **AuthenticationCapability;**
- **DataProtectionCapability;**
- **AuthorizationCapability;**
- **RoutingCapability;**
- **IdentityProtectionCapability;**
- **ResourceScannerCapability;**
- **TrafficAnalysisCapability;**

AddressTraslationCapability e RoutingCapability

La classe **AddressTranslationCapability** aggrega le funzionalità di mapping dinamico tra gli indirizzi e gli Uniform Resource Locator (URL). Queste funzionalità, come si evince dal modello, vengono svolte tipicamente da dispositivi quali proxy e NAT (Network Address Translation). Le classi specifiche che la estendono sono di conseguenza **ForwardProxyCapability** e

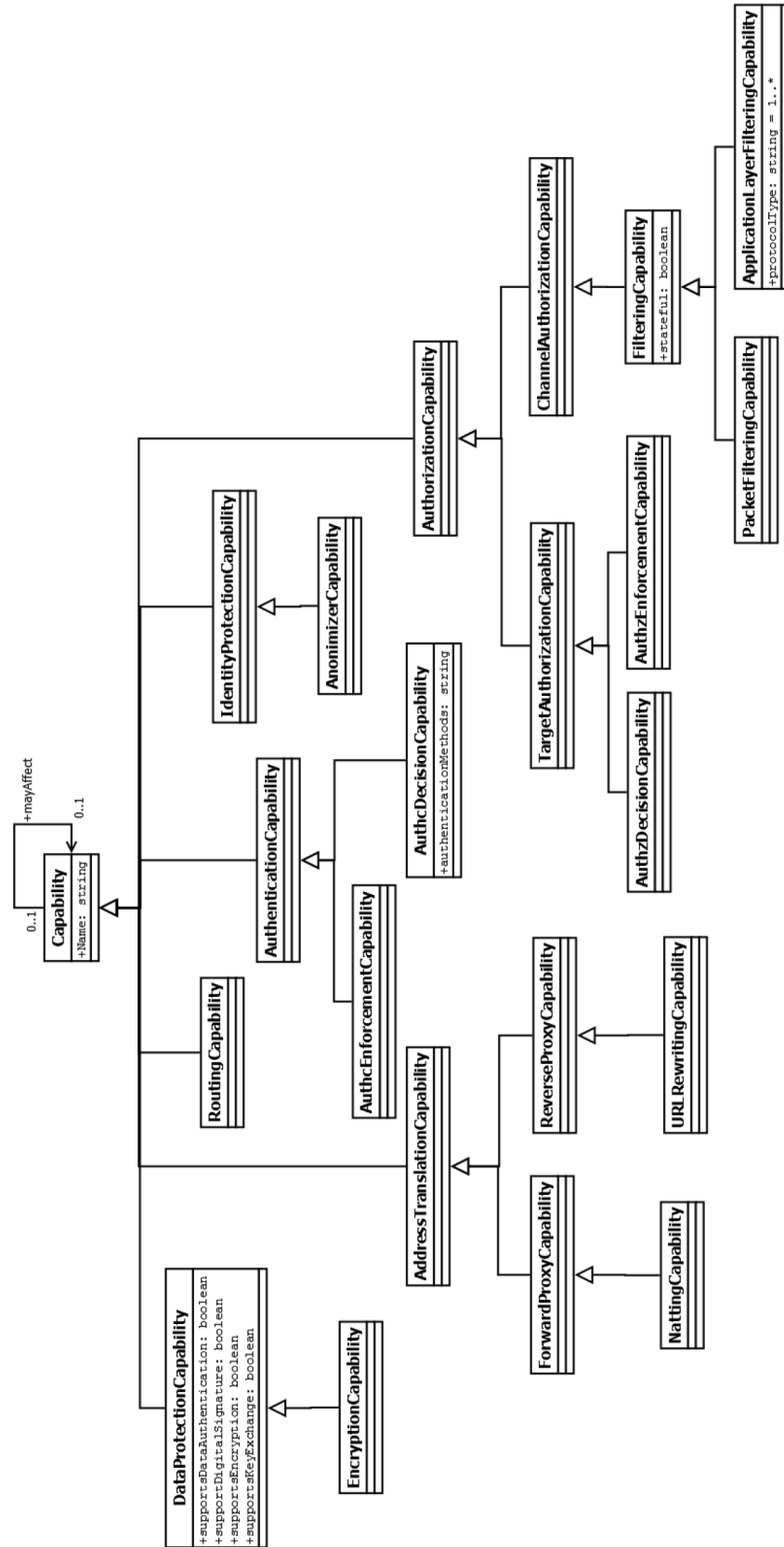


Figura 4.8: Modello astratto delle Capability 1/2 (fonte: [32]).

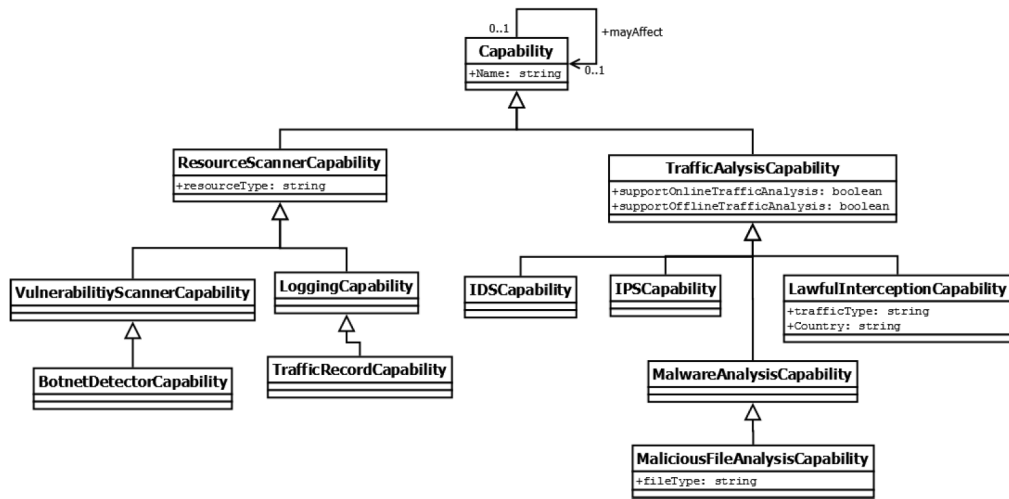


Figura 4.9: Modello astratto delle Capability 2/2 (fonte: [32]).

ReverseProxyCapability, che a loro volta vengono estese da NattingCapability e URLRewritingCapability.

La classe RoutingCapability specifica l'abilità di smistare i pacchetti, attraverso i nodi di rete connessi all'elemento in questione.

AuthenticationCapability e AuthorizationCapability

Autenticazione e autorizzazione, anche rappresentando due concetti diversi sono molto legate tra loro, ad esempio non è possibile autorizzare un utente senza prima averlo autenticato. Queste due sono descritte come due entità diverse, rispettivamente le classi astratte AuthenticationCapability e AuthorizationCapability. La prima viene estesa da due classi AuthcDecisionCapability e AuthcEnforcementCapability, questa separazione ha come obiettivo quello di poter modellare un scenario in cui viene previsto un meccanismo di autenticazione centralizzato. Tale meccanismo prevede un *authentication enforcement point* (es. WPA2 Wireless Access Point), interno ad un PEP, che invia richieste di autenticazione ad un *authentication decision point*, all'interno di un PDP. Questo *authentication decision point* può utilizzare diversi meccanismi di autenticazione (sfida asimmetrica, username/password, zero knowledge, etc.), questi sono definiti dall'attributo authenticationMethods.

La classe astratta AuthorizationCapability viene estesa da due classi: TargetAuthorizationCapability e ChannelAuthorizationCapability. Questa differenza dev'essere introdotta per distinguere i sistemi di autorizzazione che gestiscono l'accesso a risorse sotto il loro diretto controllo, dai dispositivi che prendono decisioni su canale come i firewall. Nel primo caso vengono definite due classi figlie AuthzDecisionCapability e AuthzEnforcementCapability, le quali sono state separate per una logica anloga all'autenticazione. Nel secondo caso viene definita la classe figlia FilteringCapability, che rappresenta una funzionalità di filtraggio di pacchetti e presenta un attributo stateful, per indicare se nel filtraggio viene considerata o meno, la storia dei pacchetti processati fino a quel momento. Questa classe viene ulteriormente estesa in base ai livelli ISO/OSI interessati: PacketFilteringCapability fino al livello 4 e ApplicationLayerFilteringCapability per il livello 7 (un attributo protocolType viene definito per indicare il protocollo applicazione in esame).

DataProtectionCapability

La classe astratta DataProtectionCapability indica una ITResource in grado di proteggere i dati a livello applicazione. Presenta una serie di attributi per stabilire le modalità di azione sui

dati:

- **supportsEncryption**: stabilisce se la risorsa può o meno offrire meccanismi di crittografia simmetrica (es. con AES)
- **supportsDigitalSignature**: stabilisce se la risorsa ha la capacità di firmare digitalmente dati o documenti (es. con RSA, DSS)
- **supportsDataAuthenticationAndIntegrity**: indica se la risorsa può offrire meccanismi di integrità ed autenticazione utilizzando algoritmi simmetrici (es. utilizzando HMAC)
- **supportsKeyExchange**: stabilisce se la risorsa può provvedere ad uno scambio sicuro di chiavi simmetriche (ad es. Diffie-Hellman o RSA)

Infine è prevista una classe figlia **EncryptionCapability**, che rappresenta una risorsa in grado di cifrare dati.

IdentityProtectionCapability

Tale classe rappresenta la capacità di proteggere l'identità di un utente o di una risorsa. Ha una classe figlia **ResourceScannerCapability**, che assicura l'anonimato, un esempio è dato dal progetto TOR [34].

ResourceScannerCapability

Questa classe aggrega le capacità di scansione e rilevamento di contenuti malevoli. Questa classe astratta viene estesa dalle seguenti sottoclassi:

- **VulnerabilityScannerCapability**: modella la capacità di rilevare vulnerabilità nella rete.
- **LoggingCapability**: modella l'abilità di tracciamento di connessioni o siti visitati.
- **TrafficRecordCapability**: modella la capacità di catturare e memorizzare il traffico di rete.
- **BotnetDetectorCapability**: modella la capacità di analizzare la rete e rilevare la presenza di botnet.

TrafficAnalysisCapability

Questa classe astratta aggrega le capacità di analisi del traffico di rete, al fine di effettuare delle statistiche o prendere delle decisioni, ad esempio eliminare pacchetti malevoli attraverso la firma (IPS). Tale classe viene estesa dalle seguenti sottoclassi:

- **IDSCapability**: rappresenta un Intrusion Detection System (IDS) ed opera offline.
- **IPSCapability**: rappresenta un Intrusion Prevention System (IPS) ed opera online.
- **MalwareAnalysisCapability**: modella la capacità di analizzare e rilevare malware e virus su una risorsa. Viene ancora specializzata da **MaliciousFileAnalysisCapability** per analizzare i file (es. PDF).
- **LawfullInterceptionCapability**: modella le capacità di analizzare ed eliminare il traffico per sottostare a delle normative. La tipologia di traffico da analizzare è indicato dall'attributo **traffic-Type**.

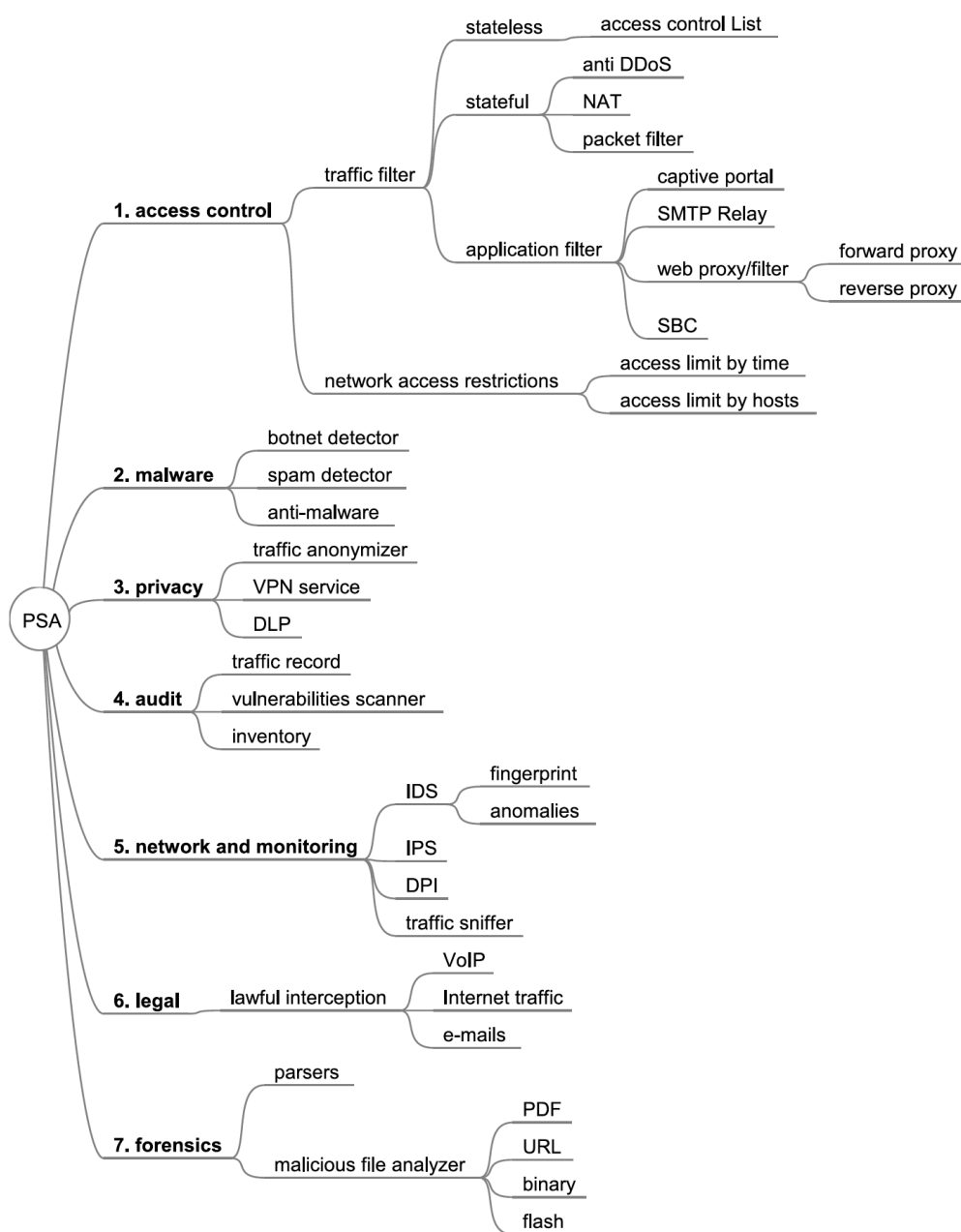


Figura 4.10: Categorie di vNSF (PSA) (fonte: [32]).

Mapping delle Capability di sicurezza sulle vNSF

Può essere di interesse valutare in figura 4.10 come vengono categorizzate le vNSF (PSA), e dopo mappare le capability sulle stesse. In tabella sintetizziamo questo mapping.

4.3.3 Linguaggi: definizione

L'obiettivo di SECURED è spostare i controlli di sicurezza dal terminale dell'utente alle vNSF. Per ottenere le funzionalità richieste, però, occorre che le funzioni di sicurezza siano scelte e configurate correttamente. Questo compito risulta complesso in virtù dell'eterogeneità delle configurazioni e dei linguaggi utilizzati per la loro specifica sui diversi dispositivi di rete. Per dominare tale complessità sono richieste tipicamente delle competenze da amministratore di sistema e talvolta anche queste

Categoria	Capability	vNSF
Access control	PacketFilteringCapability	<i>stateless/stateful inspections</i>
	ApplicationLayerCapability	<i>application filter</i>
	ForwardProxyCapability	<i>forward proxy</i>
	ReverseProxyCapability	<i>reverse proxy</i>
Malware	MalwareAnalysisCapability	<i>spam detector, anti.malware</i>
	BotnetDetectorCapability	<i>botnet detector</i>
Privacy	AnonimizerCapability	<i>traffic anonymizer</i>
	DataProtectionCapability	<i>VPN service</i>
	ApplicationLayerFiltering	<i>Data Leakage Prevention (DLP)</i>
Audit	TrafficRecordCapability	<i>traffic record</i>
	VulnerabilityScannerCapability	<i>vulnerability scanner, inventory</i>
Network and Monitoring	IDSCapability	<i>IDS</i>
	IPSCapability	<i>IPS</i>
	TrafficRecordCapability	<i>traffic sniffer</i>
	ApplicationLayerFiltering, TrafficAnalysisCapability	<i>Deep Packet Inspection (DPI)</i>
Legal	LawfulInterceptionCapability	<i>lawful interception</i>
Forensics	MaliciousFileAnalysisCapability	<i>malicious file analyzer</i>

Tabella 4.1: Mapping Capability su vNSF.

potrebbero non bastare su sistemi molto articolati. Risulta utile quindi creare linguaggi di specifica delle policy ad alto livello orientati all'utente, in modo da poter specificare le stesse con chiarezza e semplicità. In SECURED viene perseguito proprio tale obiettivo e viene declinato con la creazione di diversi linguaggi su più livelli di astrazione:

- High-level Security Policy Language (HSPL)
- Medium-level Security Policy Language (MSPL)

Di seguito in accordo con la figura 4.11 daremo dei dettagli molto schematici su come sono stati definiti i vari linguaggi e quali sono le relazioni tra gli stessi.

HSPL

L' *High-level Policy Language* è un linguaggio di alto livello per la specifica di policy. Tale linguaggio ha come obiettivo quello di essere orientato all'utente, ovvero di stabilire cosa è permesso o meno a quest'ultimo. Un linguaggio di questo tipo non fa trasparire né i dettagli delle configurazioni né le configurazioni stesse. I tre requisiti fondamentali nella definizione dell'HSPL sono:

- *semplicità*: la definizione delle policy dev'essere intuitiva e l'utente dev'essere assistito in questo processo (es. completamento automatico).
- *flessibilità*: dev'essere utilizzabile per ogni applicazioni di sicurezza, pur supportando condizioni specifiche (vincoli di tempo, tipi di contenuto e di traffico).
- *estendibilità*: dev'essere possibile estenderlo, questo in termini di introduzione di nuove policy e condizione specifiche.

La struttura di tale linguaggio è la seguente:

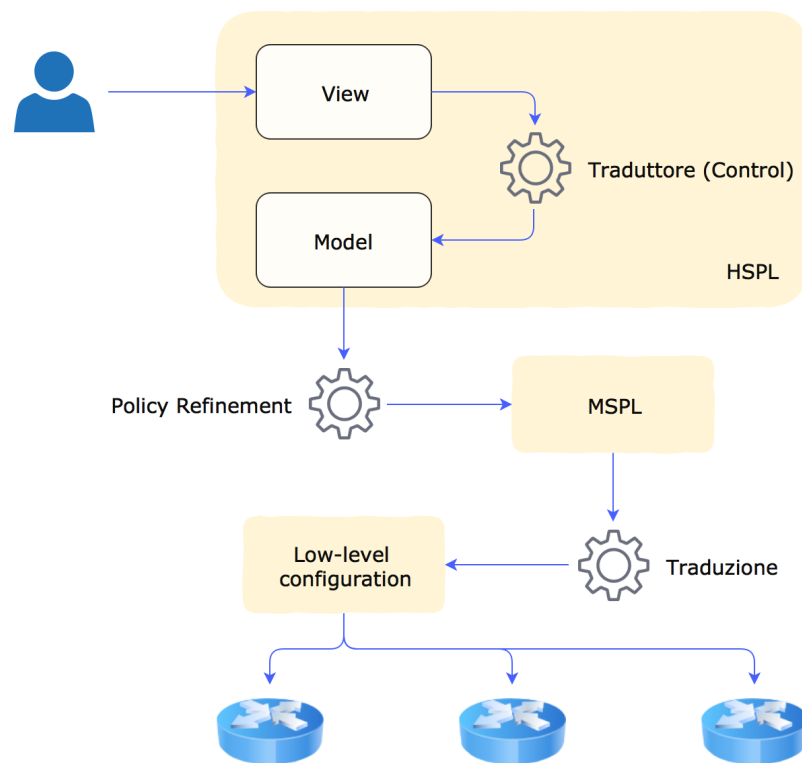


Figura 4.11: Processi di traduzione in SECURED.

[soggetto] azione oggetto [(tipo,valore) ... (tipo,valore)]

Dove:

- **Soggetto:** è l'utente che ha bisogno di accedere per effettuare alcune operazioni su un oggetto.
- **Azione:** è l'operazione che deve essere effettuata dall'utente.
- **Oggetto:** è l'entità su cui dev'essere applicata l'azione.
- **(tipo,valore):** rappresenta una condizione opzionale per creare vincoli restrittivi relativi all'azione.

I possibili **soggetti** che possiamo avere sono:

- *specific user*: un singolo utente (es. Alice).
- *current user*: il caso dell'omissione dello stesso citato in precedenza.
- *user group*: un gruppo di utenti (es "children", "marketing employees")

Le possibili **azioni** sono:

- *is/are not authorized to access*: negare una tipologia generica o specifica di traffico.
- *is/are authorized to access*: consentire una tipologia generica o specifica di traffico.
- *enable(s)*: attivare un determinato controllo (antivirus, firewall).
- *remove(s)*: intercettare e bloccare del traffico specifico.

- *reduce(s)*: limitare l'utilizzo di banda.
- *check(s) over*: analizzare le vulnerabilità.
- *count(s)*: tracciare e limitare il numero di connessioni.
- *protect(s) confidentiality*: assicurare la confidenzialità di un particolare flusso di dati.
- *protect(s) integrity*: assicurare l'integrità di un particolare flusso di dati.
- *protect(s) confidentiality integrity*: assicurare entrambe le precedenti.

Gli **oggetti** possono essere:

- *Internet traffic*: traffico riguardante indirizzi IP diversi da quelli privati.
- *DNS traffic, DNS-request, DNS-response*: traffico DNS, le DNS-request e i DNS-response.
- *intranet traffic*: traffico riguardante indirizzi IP della rete interna.
- *VoIP traffic*.
- *3G/4G traffic*.
- *all traffic*: aggregazione di tutte le tipologie di traffico (Internet, Intranet, VoIP, etc.).
- *public identity*: per mascherare l'identità degli utenti.
- *resource "x"*: traffico riguardante la risorsa "x".
- *basic parental control*: parental control con caratteristiche base che non possono essere personalizzate dall'utente.
- *advanced parental control*: parental control personalizzabile dall'utente.
- *antivirus*.
- *logging*: per tracciare l'attività dell'utente.
- *IDS, IPS*: tecniche di *Intrusion detection* e *Intrusion prevention*.
- *DDos attack protection*: protezione da attacchi *Distributed Denial of Service*.
- *email scanning, file scanning*.
- *tracking techniques, advertisement*: un esempio è dato dalle tecniche di tracciamento di utenti e pubblicità.
- *bandwidth*: per controllo della banda.
- *connection*: per controllo del numero di connessioni.
- *security status*: questo permette di visualizzare qual'è lo stato di sicurezza della rete appartenente al NED.

Le condizioni sono una serie di parametri opzionali, che possono essere inseriti in coda ad un comando del linguaggio. Seguono come visto precedentemente la struttura della coppia (**tipo**, **valore**) e i principali sono:

- *time*: per restringere l'applicazione della policy a vincoli temporali (intervallo di tempo, tempo specifico);
- *specific URL*: per restringere l'applicazione della policy ad un URL specifico o ad un sito;
- *type of content*: per restringere la policy ad un particolare tipo di contenuto;

- Un ulteriore dettaglio sull’HSPL è la possibilità di esprimere le frasi in un linguaggio ancora più semplice e vicino a quello naturale, questo per permettere agli utenti non esperti di essere agevolati nella definizione delle policy. L’idea di base è quella di avere come in figura 4.11 un HSPL che risponde al paradigma MVC (*Model-View-Controller*): la *view* permette l’inserimento del linguaggio naturale e il *controller* lo trasforma in HSPL (che rappresenta il *model*) per come l’abbiamo presentato. Ad esempio una frase in linguaggio naturale del tipo:

viene trasformata in:

MSPL

```

classDiagram
    class Capability {
        -Name: string
    }
    class Configuration {
    }
    class ITResource {
    }
    class ResolutionStrategy {
    }
    class RuleSetConfiguration {
        -Name: string
    }
    class ConfigurationAction {
    }
    class Priority {
        -value: integer
    }
    class ExternalData {
    }
    class ConfigurationRule {
        -Name: string
        -isCNF: boolean
    }
    class ConfigurationCondition {
        +isCNF: boolean
    }

    Capability "1" --> "*" ITResource : + provides
    Capability "1" --> "*" Configuration : + configuration
    Configuration "1" --> "*" ITResource : + configuration
    Configuration <|-- RuleSetConfiguration
    RuleSetConfiguration "1" --> "1" ResolutionStrategy : + resolutionStrategy
    RuleSetConfiguration "1" --> "0..1" ConfigurationAction : + defaultAction
    RuleSetConfiguration "1" --> "1" ConfigurationRule : + configurationRule
    ResolutionStrategy "1" --> "1..*" ExternalData : + externalData
    ExternalData "1" --> "1" ConfigurationRule : + externalData
    ConfigurationRule "1" --> "1" ConfigurationAction : + configurationRuleAction
    ConfigurationRule "1" --> "1" ConfigurationCondition : + configurationCondition
    ConfigurationCondition "1" --> "1" ConfigurationCondition : + isCNF
  
```

Per comprendere l’astrazione delle configurazioni illustriamo ora, rifacendoci alla figura 4.12, il meta-modello utilizzato da SECURED. Esistono per questo modello tre componenti fondamentali:

- **ITResource**: questo rappresenta il tassello fondamentale, ovvero il pezzo di software che implementa un controllo di sicurezza.
- **Capability**: una specifica funzionalità di sicurezza che può essere fornita, viene descritta nella sezione precedente.
- **Configuration**: rappresenta un'astrazione dei settaggi per la configurazione, che sono indipendenti da un dato fornitore o prodotto.

Ad ogni **ITResource** può essere assegnata nessuna o più **Configuration**, quest'ultime assegnate ad esattamente una **Capability** con un rapporto uno a uno. È possibile specificare diverse configurazioni per una stessa **ITResource**, questo per ottenere diverse funzionalità di sicurezza. La classe **Configuration**, inoltre, può essere divisa in sottoclassi per fornire configurazioni di capability specifiche (ad es. filtraggio pacchetti o firewall con la classe **FilteringConfiguration**). La classe **RuleSetConfiguration** è una specializzazione di **Configuration**, e viene usata per assegnare una configurazione ad un PDP. Questo tipo di configurazioni rappresenta il risultato del processo di *policy refinement*, ovvero un insieme di regole (**ConfigurationRule**) che rappresentano la policy MSPL per una vNSF. Una **ConfigurationRule** è composta da una o più **ConfigurationAction** che la applicano e da un insieme di **ConfigurationCondition**. Queste ultime sono dei predicati booleani che esprimono le condizioni come nel paradigma ECA visto in precedenza. L'attributo **isCNF** presente sia in **ConfigurationRule** che in **ConfigurationCondition** esprime la stessa semantica, ovvero quando è **true** tutte le condizioni devono essere valutate positivamente per applicare l'azione, contrariamente ne basta una. Quando un evento in un PEP necessita di una decisione, lo stesso contatta il PDP e gli invia una serie di dati di contesto utili, questo, poi, valuta le condizioni e invia la decisione presa, in base alle condizioni, al PEP. Esistono due casi particolari: quando nessuna regola può essere applicata o quando più regole possono essere applicate. In questo caso si parla di *policy conflict* ed esistono diverse strategie di risoluzione. Innanzitutto per la prima problematica è sufficiente impostare una regola di default, quando non vi sono dei *match* utili. Per quanto riguarda il secondo caso, in modo specifico quando parliamo di filtraggio di pacchetti, una regola potrebbe essere la *First Matching Rule (FMR)*. Quest'ultima, quando avviene il *matching* di più pacchetti con le regole definiti stabilisce un criterio di priorità ed applica la prima dell'ordine costituito. In ragione di questi ulteriori dati da modellare, utili ai criteri di decisione, viene modellata l'entità **ExternalData**, ovvero un insieme di dati utili per le decisioni da effettuare.

4.3.4 Policy Manager e Workflow

Un altro aspetto di interesse del progetto SECURED è il *SECURED Policy Manager* (SPM). L'SPM è il punto centrale della gestione delle policy e funge da Policy Management Tool, orchestrando tutte le funzionalità, principali e collaterali, del sistema di sicurezza basato su policy. Faremo una panoramica sull'architettura e tutte le funzionalità, concentrandoci su quelle di interesse per il lavoro di tesi, risparmiando dettagli non strettamente inerenti a tale lavoro.

Come si evince dalla figura 4.13, l'Architettura dell'SPM è basata innanzitutto su quattro componenti fondamentali:

- **H2M Service**: questo modulo rappresenta un servizio che si occupa del *policy refinement*, traducendo le policy dall'HSPL all'MSPL. Inoltre, in tale processo, viene selezionato un insieme ottimo di vNSF per applicare le policy HSPL, se queste non sono già state decise dall'utente.
- **M2L Service**: questo modulo rappresenta un servizio che si occupa del *policy refinement*, traducendo le policy dall'MSPL in configurazioni applicabili alle vNSF, questo avviene dopo la fase di reconciliation.
- **Policy Reconciliation Service**: questo servizio consente la riconciliazione delle policy provenienti da diversi attori, dopo che l'utente si è connesso al NED, i dettagli verranno trattati nella prossima sottosezione.

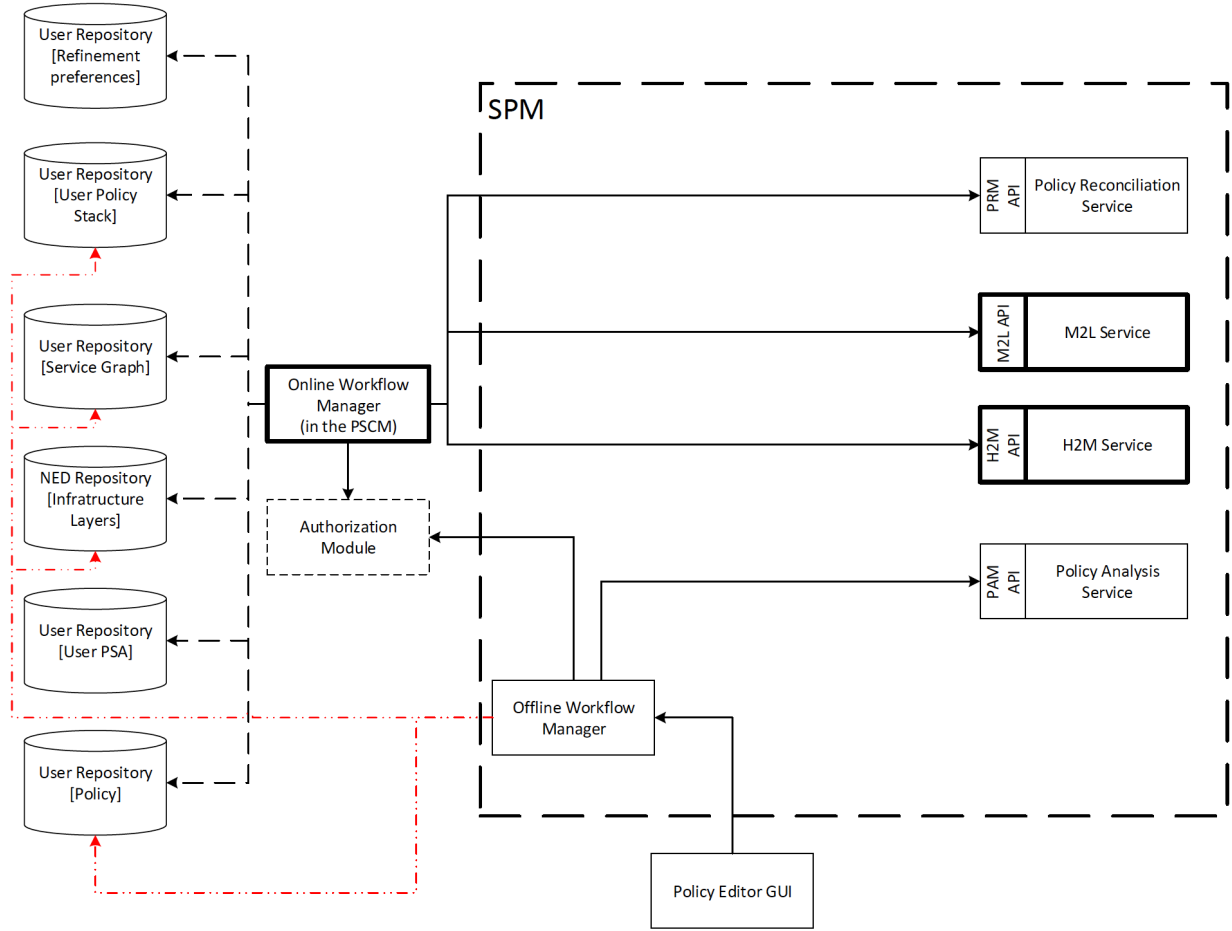


Figura 4.13: Architettura SPM (fonte: [33]).

- *Policy Analysis Service*: questo servizio consente di effettuare l'analisi sulle policy MSPL provenienti dallo stesso o da diversi attori, anche questo sarà presentato in dettaglio nella prossima sottosezione.

Per il corretto funzionamento del sistema proposto da SECURED è necessario introdurre i seguenti moduli:

- *Online Workflow Manager*: è un sottomodulo di un componente dell'architettura di SECURED chiamato *Personal Security Controller Manager (PSCM)*. Questo Workflow Manager si occupa di gestire il flusso di chiamate necessario al funzionamento dei principali servizi presenti in SECURED, in particolare relativi al policy refinement e alla policy reconciliation.
- *Offline Workflow Manager*: consente all'utente di inserire/modificare le policy di alto livello nell'apposito repository e di gestire le operazioni offline (come la policy analysis), valutandone anche i risultati.
- *Policy Editor GUI*: strumento in grado di interfacciare l'utente con l'Offline Workflow Manager.
- *Authorization Module*: determina se un componente del framework è autorizzato o meno ad accedere ai repository.
- *Un insieme di repository*: fanno da supporto a tutte le operazioni di gestione ed applicazione delle policy.

Risulta di nostro interesse concentrarci sul workflow complessivo del policy, a partire dall'introduzione delle policy di alto livello nei repository fino al raggiungimento della fase, al termine di tutto il flusso, in cui vengono generate le configurazioni e scelte le vNSF necessarie alla creazione della Service Function Chain, indispensabile a sua volta per creazione del servizio di sicurezza. Questo workflow, come già accennato nell'architettura, è declinato in due componenti: un online workflow manager e un offline workflow manager. Approfondendo questi due moduli, avremo tutto ciò che ci serve per comprendere il funzionamento complessivo del cuore di SECURED: il policy manager. Prima di entrare nei dettagli, dobbiamo introdurre ulteriori concetti complementari:

- *Service Graph*: è un grafo, i cui nodi rappresentano i possibili controlli di sicurezza che soddisfano i requisiti per l'applicazione delle policy. Gli archi sono, invece, i flussi di traffico fra i vari controlli, il tutto in una logica sovrapponibile al concetto di SFC introdotto in precedenza.
- *Application Graph*: è un grafo, i cui nodi rappresentano le vNSF (o PSA) che soddisfano i requisiti per l'applicazione delle policy. Gli archi rappresentano, ugualmente, i flussi di traffico.
- *Reconciled Application Graph*: è un Application Graph, frutto della reconciliation di policy provenienti da attori diversi.

Online Workflow Manager

Quando un'utente si autentica correttamente al framework di SECURED, l'online workflow manager è in grado di effettuare quattro operazioni:

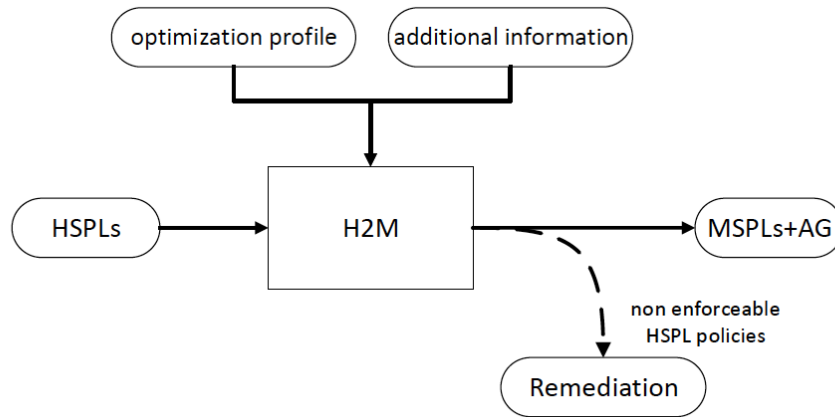


Figura 4.14: Policy-driven refinement (fonte: [33]).

- Preleva tutte le policy HSPL dagli appositi repository (in questo caso l'User Policy Repository o UPR) e chiama il servizio H2M per effettuare il policy refinement. Questo processo a partire dall' HSPL restituisce l'Application Graph e le MSPL, il tutto può avvenire secondo 2 approcci diversi:
 - *Policy-driven approach*: questo approccio risulta il più intuitivo e consiste nel fornire al modulo H2M le HSPL e il criterio di ottimizzazione desiderato per le PSA. Il sistema si occuperà di effettuare un'analisi di *non-enforceability* che identifica eventuali incompatibilità tra le policy specificate e le vNSF a disposizione. Nel caso in cui alcune risultino non compatibili ha luogo la *remediation*, ovvero una strategia per gestire questa mancanza. (fig. 4.14)
 - *Application-driven approach*: alternativamente può essere utilizzato quest'approccio, dove l'utente, oltre a fornire le HSPL al modulo H2M, si preoccupa di indicare anche

quali sono le vNSF che vuole utilizzare. Questo approccio richiede che venga effettuata anche un'analisi preliminare di *early non-enforceability* sulla compatibilità delle vNSF specificate in relazione alle policy HSPL indicate. (fig. 4.15)

- Si occupa della *reconciliation* delle policy MSPL. Partendo dal presupposto che più attori possono specificare le policy per un singolo utente (con un criterio di priorità), è chiaro che occorre riconciliare le differenti policy specificate. Per questa ragione, questo workflow manager chiama il servizio di reconciliation, che effettua l'omonima operazione, e successivamente reinserisce le policy riconciliate nell'UPR, nello specifico vengono memorizzati un Reconciled Application Graph e le nuove MSPL.
- Quando necessario, traduce le policy MSPL, attraverso la chiamata al servizio M2L, in configurazioni a basso livello per le vNSF.
- Comunica con un modulo di gestione superiore, da cui è controllato e fornisce allo stesso i risultati della traduzione delle policy di basso livello e le configurazioni per le vNSF.

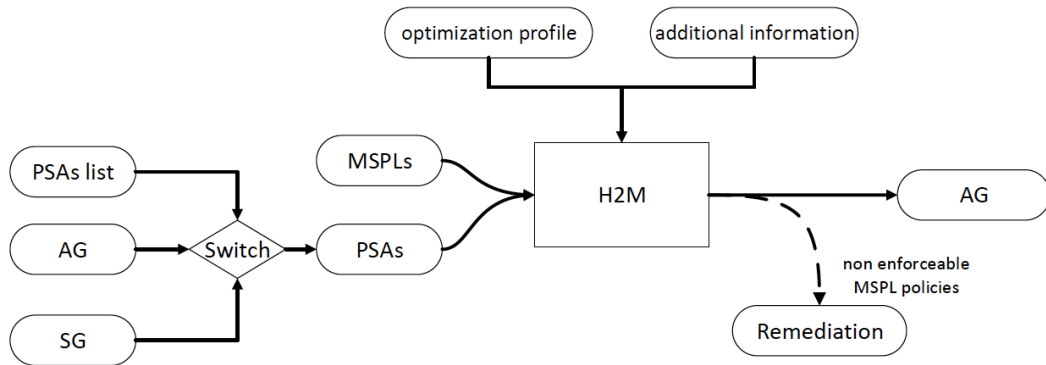


Figura 4.15: Application-driven refinement (fonte: [33]).

Una sintesi delle mansioni dell'online workflow manager e del processo di traduzione sono dati dalla figura 4.16.

Offline Workflow Manager

L'offline workflow manager è un componente interno all'SPM e si occupa di gestire le seguenti operazioni:

- Interfaccia la GUI per la manipolazione delle policy con i repository, ad esempio tramite questo manager è possibile caricare le policy in HSPL per un determinato utente.
- Gestisce le chiamate al servizio di Policy Analysis che vedremo in seguito.

In figura 4.16, come già accennato, vi è un riassunto del processo di traduzione delle policy. In tale immagine viene sottolineata la differenza logica tra i due manager e si evince come l'offline workflow manager non sia direttamente coinvolto nella traduzione delle policy. I suoi compiti bensì sono da considerarsi come operazioni a supporto, le quali vengono effettuate in momenti diversi dal processo di traduzione ed applicazione delle policy.

4.3.5 Modello geometrico e Policy Analysis

Una nota, allo scopo di completare la trattazione delle policy nel progetto SECURED, è data dalla definizione del modello geometrico e dalla trattazione del concetto di Policy Analysis.

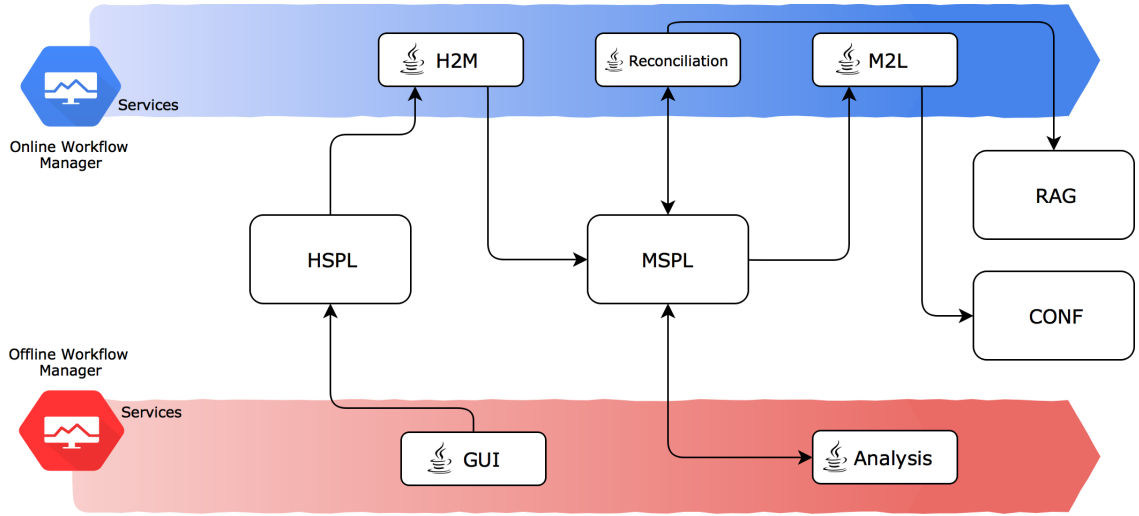


Figura 4.16: Processo di trasformazione delle policy e ruolo dei Workflow Manager.

Modello Geometrico

Con riferimento ai deliverable [35], definiamo la Policy come una quadrupla (R, \mathfrak{R}, E, d) dove:

- $R = \{r_i\}_i, i \in [1, n]$ è l'insieme di regole (composto da n regole),
- $\mathfrak{R} : 2^R \rightarrow A$ è la funzione di risoluzione, utilizzata per decidere l'azione da applicare,
- E è l'insieme dei dati esterni associati alle regole,
- d è l'azione di default da applicare quando non ci sono *match* tra il pacchetto e le regole definite.

Una volta definito l'insieme di regole, possiamo modellare, attraverso la funzione $match_R(x)$, l'operazione di verifica del *matching* tra il pacchetto e le regole stesse, tale funzione ha come valore di ritorno la regola che ha avuto *match* positivo. Non dimenticando i casi particolari discussi in precedenza, abbiamo che i valori possibili di ritorno della funzione di risoluzione \mathfrak{R} sono:

$$p(x) = \begin{cases} d, & \text{se } match_R(x) = \emptyset \\ a_i, & \text{se } match_R(x) = r_i \\ \mathfrak{R}(\{r_1, r_m, \dots\}), & \text{se } match_R(x) = \{r_1, r_m, \dots\} \end{cases} \quad (4.1)$$

Dalla 4.1, si evince che nel caso particolare di nessun match viene applicata la regola di default e in quello di match multiplo una strategia di risoluzione. Tale strategia è basata tipicamente sull'utilizzo di dati esterni, questi ultimi modellati dall'insieme E , definito come: $E = \{f_j : R \rightarrow X_j\}_j$. Questo insieme non è altro che il mapping delle regole su degli attributi esterni alle stesse. Da qui è possibile definire la strategia di risoluzione come una funzione composta $\epsilon_E(r_i) = (r_i, f_1(r_i), f_2(r_i), \dots)$, questa ha come elementi oltre alla regola stessa una serie di attributi che possono essere utilizzati per prendere una decisione. Un caso pratico lo ritroviamo nella strategia FMR, sopra citata. Quest'ultima consiste nel definire una funzione π , in grado di associare un indice ad ognuna delle regole. L'indice creato servirà come criterio di priorità per l'applicazione delle regole. Un esempio è dato dalla policy $p = (R, FMR, \{\pi\}, DENY)$, costituita dall'insieme di regole R , dalla funzione di risoluzione FMR, dalla funzione utilizzata dalla strategia di risoluzione e dalla regola di default DENY. Andiamo a vedere come viene risolta:

$$FMR = \{(r_x, \pi(r_x), (r_y, \pi(r_y), \dots)) \rightarrow a_k \quad (4.2)$$

tale che

$$\pi(r_k) = \min_{r \in \{r_x, r_y, \dots\}} \{\pi(r)\} \quad (4.3)$$

Nota a margine, definiamo il criterio di equivalenza tra policy come:

$$\forall x \in \mathcal{S}, \mathfrak{R}_1(\text{match}_{r_1}(x)) = \mathfrak{R}_2(\text{match}_{r_2}(x)) \quad (4.4)$$

se soddisfatto due policy si dicono equivalenti, definiremo \mathcal{S} successivamente.

Abbiamo definito le regole secondo il paradigma ECA (Evento-Condizione-Azione) quando abbiamo presentato il modello IETF. Andiamo adesso a modellare in SECURED condizioni e azioni.

Definiamo una generica clausola condizionale

$$c_i = s_{i1} \times \dots \times s_{im} \subseteq \mathcal{S}_1 \times \dots \times \mathcal{S}_m = \mathcal{S} \quad (4.5)$$

come l'iper-rettangolo di m dimensioni, ottenuto come prodotto cartesiano tra le condizioni s_{ij} , definite nei loro campi di appartenenza \mathcal{S}_j , chiamati *selettori*. Il prodotto $\mathcal{S}_1 \times \dots \times \mathcal{S}_m$ è definito come *spazio di decisione*.

Un pacchetto risulta essere una sequenza finita di bit, ragion per cui lo modelliamo come un insieme discreto finito \mathbb{P} . Una *condizione* su di un dato *selettore* è verificata se il valore del pacchetto mappato sul rispettivo *selettore* appartiene alla condizione stessa. Un esempio è riportato in figura 4.17, dove il pacchetto x_1 appartiene ad entrambe le condizioni s_1 e s_2 , quindi c'è stato un *match* con la regola. Per il pacchetto x_2 , invece, non è così in quanto appartiene ad una sola delle due.

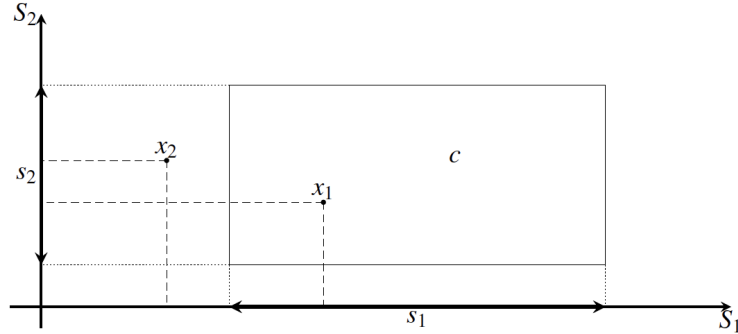


Figura 4.17: Rappresentazione geometrica di una regola (fonte: [35]).

Classifichiamo come segue le varie tipologie di selettori:

- *Exact match*: selettori che rappresentano insieme non ordinabili, quindi il controllo è solo per uguaglianza.
- *Range-based*: in questo caso abbiamo selettori su insiemi ordinabili, quindi possiamo definire il concetto di intervallo di valori.
- *Regex-based*: selettori che permettono controlli basati su espressioni regolari.
- *Custom-match*: sono selettori che effettuano controlli personalizzati, basati su algoritmi ad hoc.

Risulta possibile, avendo ora gli strumenti geometrici necessari, definire delle operazioni su insiemi al fine di capire qual'è l'iper-rettangolo di interesse per una determinata regola. In particolare ci concentriamo sull'operazione di intersezione tra due o più condizioni (es. $s_1 \cap s_2$), questo come già detto ci permette di valutare l'iper-rettangolo che definisce lo spazio di applicazione di una regola. Tramite questo strumento è possibile individuare alcune anomalie presenti nelle policy rule definite, ad esempio quando gli iper-rettangoli di due regole si intersecano o sovrappongono. La rilevazione delle anomalie è quella che chiamiamo Policy Analysis e che vedremo a breve.

L'ultimo tassello da definire è l'azione. La stessa è una funzione che trasforma un pacchetto nella forma $a : \mathbb{P} \rightarrow \mathbb{P}$. L'insieme delle azioni è definito come $\mathcal{A} = a : \mathbb{P} \rightarrow \mathbb{P}$. Un valore particolare dell'insieme \mathbb{P} è il valore \emptyset che modella il pacchetto da scartare. Alcuni esempi di azioni sono riportati di seguito:

- *ALLOW*: corrisponde alla funzione identità;
- *LOG/MONITOR*: corrisponde ugualmente alla funzione identità, in virtù del fatto che i dati vengono solo letti;
- *DENY*: trasforma un pacchetto in quello nullo;
- *REDIRECT*: applica la funzione identità sull'interfaccia a cui inviare il traffico;
- *MODIFY*: aggiunge ulteriori informazioni a quelle contenute nella clausola di condizione;
- *NEW HEADER*: modella la modifica delle informazioni di intestazione;
- *ENCRYPT*: modella la cifratura del pacchetto riportando nella clausola i campi cifrati e/o il livello a cui avviene la cifratura.

Policy Analysis

La *Policy Analysis* è una tecnica di rilevazione delle anomalie riguardanti le policy e nel progetto SECURED viene effettuata dal modulo *Policy Analysis Service*.

Tale modulo presenta varie funzionalità:

- *intra-policy anomalies*: rileva anomalie nella stessa policy.
- *inter-policy anomalies*: rileva anomalie tra più policy relative alla stessa Capability.
- *inter-function anomalies*: rileva anomalie tra policy specificate per vNSF che hanno Capability diverse.
- *inter-actor anomalies*: rileva anomalie tra policy specificate da attori diversi.

In particolare in SECURED:

- Le anomalie intra-policy vengono rilevate dal sottomodulo di Single Function Analysis (SFA).
- Le anomalie inter-function vengono rilevate dal sottomodulo di Inter Function Analysis (IFA).
- Le anomalie inter-actor e inter-policy vengono rilevate dal sottomodulo di Multiple user Inter Function Analysis (MIFA).

Capitolo 5

Obiettivi e metodi

In questo capitolo andremo a definire gli obiettivi del presente lavoro di tesi e presenteremo gli strumenti che ci permetteranno di perseguirli.

5.1 Definizione degli obiettivi

Come abbiamo visto nei capitoli 2 e 3, NFV è diventata una soluzione molto promettente nell'ambito delle reti di telecomunicazione. Contestualmente, anche l'approccio di gestione di sistemi di sicurezza basato su policy si è rivelato molto efficace, basti pensare al progetto SECURED presentato nel capitolo 4.

L'idea cardine di questo lavoro di tesi è quindi quella di combinare le due soluzioni appena citate, progettando ed implementando un sistema, basato sulla tecnologia NFV, in grado di istanziare ed orchestrare funzioni di rete virtuali di sicurezza (vNSF) e di ottenere un Network Security Service personalizzato per l'utente. Tale sistema sarà policy-driven, ovvero guidato dalle policy che sono state definite a partire dal progetto SECURED nel capitolo 4.

Un sistema così concepito ha la capacità di allocare, su uno o più nodi di rete, tutte le risorse e le funzionalità necessarie (es. firewall, IDS, DPI) per ottenere uno scenario personalizzato di sicurezza. L'utente, che è al centro di tale sistema, può definire come abbiamo visto nell'ambito di SECURED tutta una serie di policy di alto livello, che corrispondono alle sue esigenze in termini di funzionalità di sicurezza. Queste verranno gestite dal sistema e convertite in un servizio di sicurezza ad hoc. Questo servizio sarà a sua volta composto da una serie di vNSF, accuratamente scelte dal processo di ottimizzazione delle policy e configurate in linea con le esigenze dell'utente.

Un approccio del genere viene utilizzato, con sempre maggior frequenza, dai fornitori di servizi di rete, che in molti casi si trovano di fronte a clienti che non hanno le capacità infrastrutturali di sopperire, in modo efficace, alle loro esigenze in termini di sicurezza. Questo porta i provider a proporre dei servizi, in modalità cloud e di facile distribuzione, che offrono i meccanismi di sicurezza necessari e richiesti dal cliente specifico. Tutto questo è conveniente in termini economici da ambedue le parti, infatti, il service provider si ritrova un'infrastruttura da sfruttare ed è in grado di offrire questo servizio a pagamento, mentre il cliente dall'altra parte ha dei costi contenuti per ottenere un servizio che altrimenti non potrebbe permettersi. In più con l'utilizzo di tecnologie NFV, il provider ha la capacità di mantenere molto bassi i costi per erogare tale servizio, per tutta una serie di motivi discussi nel capitolo 2 ed è altresì capace di aggiornare e configurare le funzionalità di sicurezza in modo dinamico, offrendo un servizio che mantiene sempre costanti affidabilità e sicurezza. Il cliente è per questo motivo scaricato da ognuna di queste responsabilità, l'unica operazione che deve effettuare è concordare il livello e la qualità del servizio.

Questo nuovo approccio alla sicurezza, offerta come servizio cloud, prende il nome di Security-as-Service (SECaaS). In figura 5.1 è rappresentata ad alto livello l'architettura di tale paradigma e come essa viene applicata nello scenario di questo lavoro di tesi. L'utente, come già in precedenza visto, non è in grado di avere a disposizione un alto livello di sicurezza, fortemente personalizzato

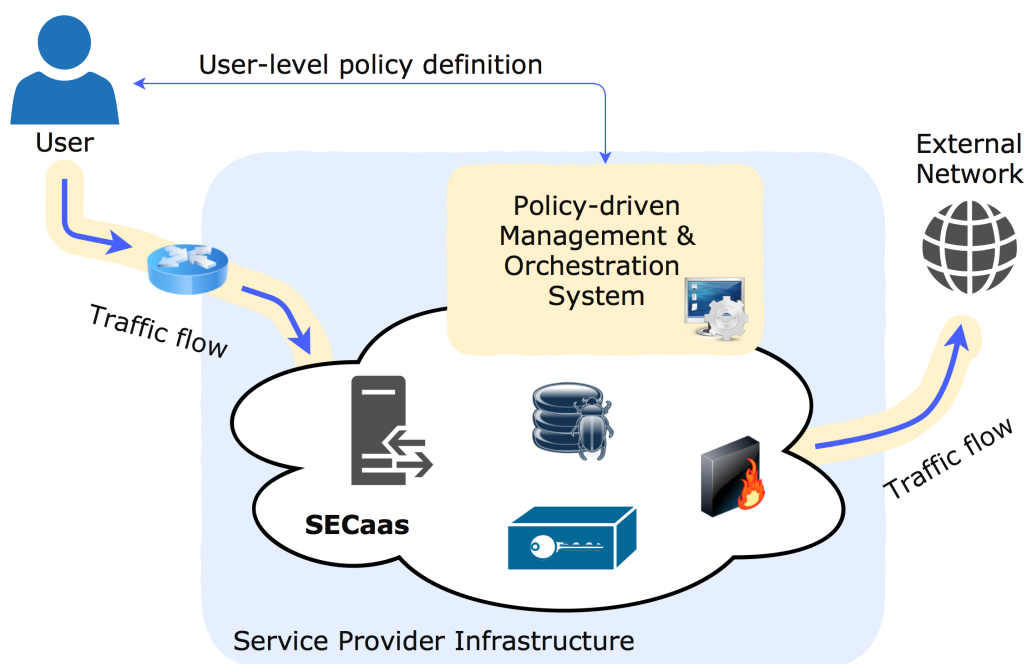


Figura 5.1: Paradigma Security-as-Service.

e costantemente aggiornato a seconda delle sue esigenze. A questo punto si rivolge al suo service provider che gli garantisce, attraverso la sua infrastruttura cloud, tutta una serie di servizi a seconda delle sue preferenze (esprese tramite le User-level policy). Una volta istanziate le risorse necessarie, il service provider ha messo a disposizione dell'utente un collegamento sicuro, basato sul livello di sicurezza desiderato.

Anche entrando a pieno titolo in questo paradigma, il lavoro che presentiamo ha delle peculiarità interessanti rispetto alle soluzioni attualmente a disposizione. In prima istanza, la forte centralità dell'utente nel processo di creazione del servizio personalizzato e il processo di ottimizzazione per garantire il soddisfacimento dei requisiti di sicurezza, in secondo luogo la forte spinta all'utilizzo di tecnologie performanti per poter garantire dei tempi di istanziazione e configurazione del servizio con bassa latenza. Tutti questi aspetti verranno trattati nel dettaglio nei capitoli relativi alla progettazione ed implementazione della soluzione.

Abbiamo visto che questo progetto parte dal lavoro fatto in SECURED, la differenza risiede principalmente nelle tecnologie utilizzate e nella riprogettazione del workflow, al fine di ottimizzarlo e renderlo compatibile con il sistema di orchestrazione utilizzato. Sembra opportuno, arrivati a questo punto, delineare in modo netto quali sono state le componenti del progetto SECURED riutilizzate e quali gli sforzi e le innovazioni introdotte per la realizzazione di questo nuovo sistema.

Verranno ripresi all'interno di questo progetto:

- Parte dei moduli software relativi al sistema di gestione delle policy di SECURED (es. SPM, Policy Analysis, H2L service);
- Gli obiettivi finali e l'approccio che prevede la centralità dell'utente in tutto il sistema.

Il valore aggiunto è dato da questi elementi:

- Reingegnerizzazione dei moduli software e delle architetture presenti in SECURED;
- Messa in opera di un'infrastruttura basata su strumenti cloud, nello specifico OpenStack;
- Utilizzo di un nuovo strumento di orchestrazione e management, nello specifico Open Source MANO;

- Utilizzo di un nuovo approccio per la configurazione automatica delle funzioni di rete virtuali;
- Discussione e presentazione di alcune soluzioni tecnologiche per ottimizzare il processo di istanziamento del servizio.

Il percorso per raggiungere tali obiettivi si può delineare nelle seguenti fasi:

- Analisi dello stato dell'arte e degli strumenti da utilizzare (dettagli in sez. 5.2);
- Design e implementazione della nuova soluzione (dettagli in sez. 5.3);
- Test della soluzione e una corposa analisi sulle prospettive future. (dettagli in sez. 5.4);

5.2 Analisi degli strumenti

In questa fase sono stati analizzati a fondo diversi strumenti nell'ambito della virtualizzazione delle funzioni di rete.

Dal punto di vista dell'infrastruttura abbiamo indagato le differenze tra la *full-virtualization*, basata su VM, e la *light virtualization*, basata su container, per capire quali potessero essere le effettive applicazioni e quali gli strumenti più adatti per la messa in opera della nostra infrastruttura. Da questo punto di vista abbiamo scelto per il nostro Proof-of-Concept OpenStack, che presenteremo nel dettaglio nel capitolo 6 e daremo anche le motivazioni di tale scelta.

Per quanto riguarda il data plane, sono state valutate varie tecnologie soprattutto in merito all'ottimizzazione dell'NFVI in termini di I/O, tra le principali SR-IOV e dpdk. Queste sono molto legate alla prospettiva di un binomio NFV-SDN e presenteremo nel capitolo 9 i possibili vantaggi derivanti dalla combinazione di tali tecnologie.

Sul fronte gestione ed orchestrazione, abbiamo analizzato nel capitolo 3 quali potessero essere le scelte in termini di sistemi MANO e quali i vantaggi derivanti dalle varie scelte. Nel capitolo 6 andremo ad approfondire Open Source MANO e a giustificare il perché è stato scelto questo sistema per l'orchestrazione.

Ancora c'è stato un processo di reingegnerizzazione dei moduli software di SECURED, come visto in precedenza, per questo nel capitolo 6 andremo a valutare la tecnologia LXD e capire come è possibile utilizzarla per ottenere dei container molto vicini al concetto di VM, utili ai nostri scopi per rendere modulare ed altamente integrabile parte del framework di SECURED.

In conclusione, questa fase è stata di fondamentale importanza per tracciare le linee guida su cui, successivamente, avrebbero poggiate le fasi di progettazione ed implementazione. Capire, infatti, quello che tecnologicamente era possibile fare nell'immediato e quello che invece sarebbe stato a disposizione come sviluppo futuro, è stata una discriminante chiave per realizzare un PoC attuale e delineare la sua evoluzione per eventuali sforzi futuri.

5.3 Design e Implementazione

Nella fase di progettazione si è proposta un'architettura quanto più modulare ed integrabile possibile, al fine di supportare nuove integrazioni future. Successivamente si è provveduto alla creazione di alcuni PoC, con lo scopo di valutare la fattibilità di ciò che si era teorizzato. Queste due fasi verranno dettagliate rispettivamente nei capitoli 7 e 8.

5.4 Test e Sviluppi futuri

Quest'ultima fase è relativa in parte ai test della soluzione implementata, questo secondo diversi indicatori presentati nel capitolo 8, e in altra parte agli sviluppi futuri. Questi sono reputati

cruciali in considerazione di due fattori fondamentali; il primo relativo allo stato embrionale della tecnologia NFV e il secondo alle potenzialità di alcune soluzioni tecnologiche emergenti, legate sempre ad NFV. Fondamentale risulta quest'ultima fase, soprattutto perché parlare di NFV equivale solo in parte all'orchestrazione di VNF, più in generale, invece, significa illustrare quali sono le vere sfide in termini di prestazioni e di sicurezza, il tutto per capire quali potranno essere, e quali no, le tecnologie vincenti nel prossimo futuro (as es. EPA, SDN, SR-IOV, dpdk, SELinux, Light Virtualization). Questi argomenti sono stati trattati in parte nei primi capitoli [2](#) e [3](#) e in modo più diffuso nel capitolo [9](#).

Capitolo 6

Analisi degli strumenti

In questo capitolo analizzeremo gli strumenti utilizzati per l'implementazione della soluzione e parte di quelli legati agli sviluppi futuri.

6.1 OpenStack

OpenStack è una piattaforma libera ed open source per il cloud computing. Nasce da una collaborazione tra NASA e Rackspace, dove la prima contribuisce con il progetto Nebula (poi diventato Nova) e la seconda con Cloud Files (poi diventato Swift). La prima release viene rilasciata per il sistema operativo Ubuntu nel 2011 e da lì in poi introdotto in varie distribuzioni, tra cui Red Hat nel 2012. Ad oggi viene sviluppato, distribuito ed adottato con l'ausilio della *OpenStack Foundation*, un'organizzazione non profit che lo promuove a livello globale.

Con riferimento ai modelli di servizio cloud, presentati nel capitolo 2, OpenStack appartiene alla categoria *Infrastructure-as-a-Service*, infatti, mette a disposizione agli utenti tipicamente server e risorse virtuali. In relazione al modello NFV, invece, OpenStack comprende parte dell'infrastruttura NFVI (strato di virtualizzazione e *Virtual Infrastructure*) e il modulo VIM dell' NFV MANO. A prescindere dalla collocazione, nell'uno o nell'altro modello, è interessante isolare quali sono le funzionalità che questa piattaforma mette a disposizione:

- un pool di risorse virtuali (Compute, Network e Storage) a partire da macchine fisiche distribuite sulla rete;
- permette la gestione di macchine virtuali e del loro ciclo di vita (es. istanziazione, terminazione, sospensione, migrazione), attraverso l'utilizzo delle risorse citate;
- offre connettività alle macchine virtuali, consentendo così la comunicazione e l'accesso alle stesse;
- consente di gestire volumi virtuali, immagini software e altre risorse, utili al ciclo di vita della VM;
- supporta SDN per la comunicazione tra i vari nodi, attraverso l'utilizzo del modulo di gestione di rete;
- offre numerosi moduli aggiuntivi per monitoraggio delle risorse, l'orchestrazione e la configurazione delle VM, e ancora alcuni per l'orchestrazione in ambito NFV.

In base alle funzionalità ad alto livello descritte, andiamo a dettagliare ora l'architettura di questo strumento e a valutare quali sono state le ragioni della sua adozione nello sviluppo della nostra infrastruttura NFV.

6.1.1 Architettura base

OpenStack è formato da vari moduli che offrono servizi diversi, alcuni di base ed essenziali per il funzionamento, altri aggiuntivi, per ottenere delle funzionalità avanzate e accessorie. Esistono numerose versioni di questa piattaforma, i cui aggiornamenti vengono rilasciati con cadenza semestrale. Ogni rilascio, trattandosi di un progetto open, introduce novità sostanziali e spesso modifica anche l'assetto dei moduli che vengono utilizzati, per questo motivo molti preferiscono utilizzare delle versioni datate, al fine di avere una piattaforma più stabile a disposizione. La versione più utilizzata attualmente risulta Mitaka, la quale è già *End-of-Life* da svariati mesi. Andiamo adesso a descrivere i vari moduli (fig. 6.1) ed il funzionamento complessivo di questa piattaforma.

Nova (Compute)

Nova è il componente fondamentale di OpenStack, in quanto si occupa della gestione e automazione di un pool di risorse virtuali, al fine di creare istanze di macchine virtuali o *VM instance*. Questo modulo non è un hypervisor, ma supporta numerose tecnologie di virtualizzazione, come KVM, VMware, Xen, Hyper-V e ancora vanta un supporto, se pur ancor limitato, ai container, sia in tecnologia LXD che Docker. Per fornire tale supporto, tipicamente nova installa un *agent* sull'hypervisor così da renderlo compatibile con l'ambiente OpenStack.

In figura 6.1, possiamo notare che i moduli indispensabili per nova (OpenStack Compute) sono:

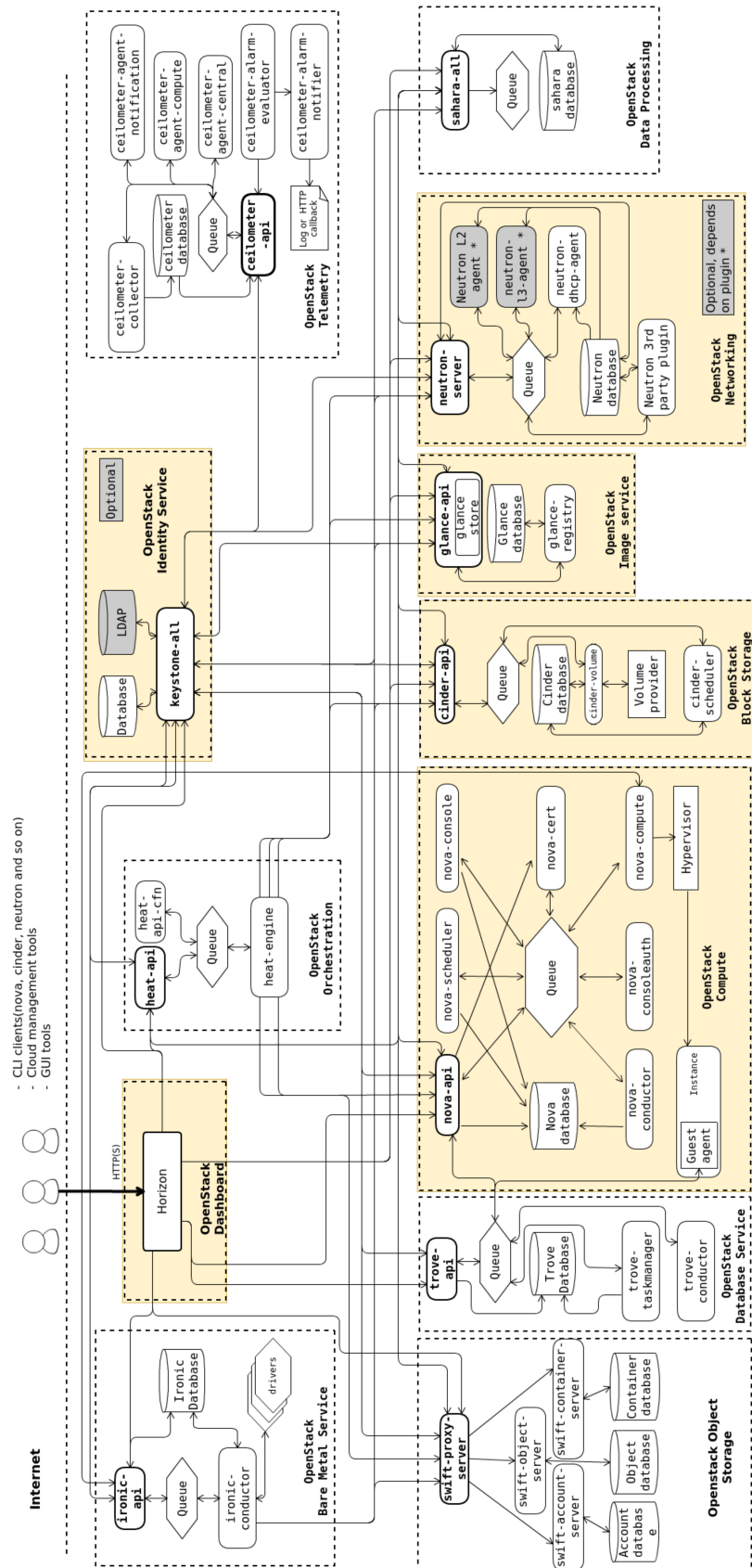
- *queue*: Corrisponde ad un servizio di comunicazione che può essere offerto da un *message broker*, come ad esempio RabbitMQ attraverso il protocollo AMQP;
- *nova-scheduler*: decide su quali host indirizzare le varie istanze;
- *nova-conductor*: si occupa della gestione delle richieste che devono essere coordinate (operazioni sulle istanze) e funge da database proxy;
- *nova-compute*: gestisce la comunicazione tra l'hypervisor e le macchine virtuali;
- *nova-api*: riceve le richieste http, le converte in un formato opportuno e le comunica agli altri moduli.

Nova è quindi il punto nevralgico della piattaforma, andando a gestire il dialogo tra varie tecnologie di virtualizzazione (hypervisor) e le esigenze in termini di gestione delle macchine virtuali. Un'altra caratteristica interessante è data dal fatto che nova è progettato per scalare orizzontalmente, questo significa che è possibile avere a disposizione, come in altri casi in OpenStack, più nodi di computazione (macchine fisiche diverse). In tal caso esisterà un nodo centrale (detto *Controller*) che ospiterà parte dei servizi di nova, come lo scheduler, e più nodi di computazione (*Compute*), dove sarà installato l'hypervisor specifico e nova-compute.

Neutron (Networking)

Neutron è l'elemento che si occupa della gestione di rete, quindi della connessione tra le varie VM e verso l'esterno.

Come si evince dalla figura 6.2, uno scenario tipico di connessione può vedere come attori diversi *compute node* e diversi *nodi di rete*. Dove per compute node si intende un nodo su cui è presente il nova-compute e per nodo di rete, un nodo dove è presente tutto lo stack di rete, necessario per fornire la connettività. Scendendo nel dettaglio è possibile creare, attraverso neutron, sia delle reti interne all'ambiente di OpenStack (come quella di management in figura), tipicamente non raggiungibili dall'esterno, sia delle reti che forniscono connettività verso l'esterno (External). Per effettuare queste operazioni, neutron utilizza i seguenti moduli, che ritroviamo in parte in figura 6.1 e in altra parte in figura 6.2:

Figura 6.1: Architettura Logica OpenStack Pike (in giallo i moduli base) (fonte: [docs openstack](https://docs.openstack.org)).

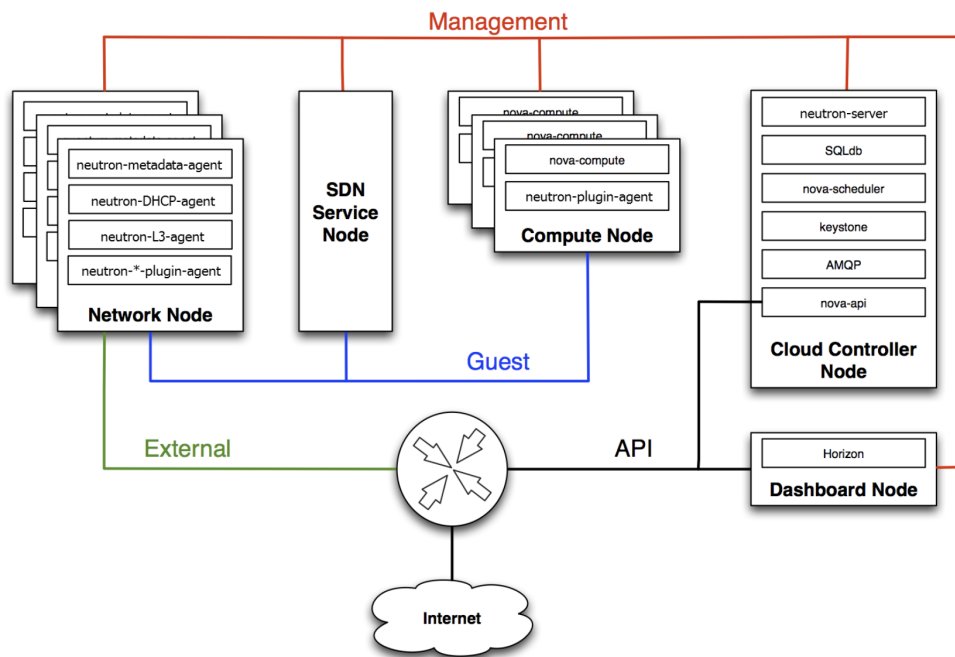


Figura 6.2: Neutron e nodi di rete in OpenStack (fonte: [docs openstack](#)).

- *neutron server*: questo è un servizio eseguito sul *controller node* e si occupa della gestione delle chiamate API (riguardanti la parte di networking), coordinando i vari agent e plug-in al fine di fornire la connettività richiesta.
- *neutron-plugin agent*: viene eseguito su ogni compute node e gestisce localmente la configurazione di tutti gli switch virtuali sul nodo.
- *neutron-DHCP agent*: viene eseguito su ogni nodo di rete e funge da DHCP per quel nodo.
- *neutron-L3 agent*: viene eseguito su ogni nodo di rete e fornisce connettività di livello 3, tra le VM o verso l'esterno.
- *SDN service node*: fornisce dei servizi aggiuntivi, che consentono la gestione via software delle reti presenti sui nodi. Un esempio potrebbe essere un SDN Controller che si preoccupa di cambiare le regole all'interno delle tabelle OpenFlow dei vSwitch, sulla base di decisioni a livello applicativo. In ogni caso bisogna, se non integrato, fornire il plugin adeguato per la comunicazione tra questo modulo e il neutron server.

Tutto questo ci serve a capire che OpenStack, utilizzando i moduli introdotti, gestisce le reti in modo estremamente modulare, consentendo una grande flessibilità nelle configurazioni di rete. Vedremo degli esempi pratici di configurazione più avanti nell'appendice [B](#).

Cinder (Block storage)

Cinder è il modulo che fornisce storage persistente alle VM instance. Di base lo storage a disposizione per una VM è effimero, ovvero svanisce alla terminazione della stessa. Cinder, per contro, mette a disposizione la possibilità di associare dei volumi persistenti alle istanze. Tra le varie funzionalità a disposizione nelle API fornite ci sono, oltre a quelle di gestione ordinaria (creazione, cancellazione volume), delle funzionalità avanzate, come la possibilità di estendere i volumi, clonarli o fare degli *snapshot* (istantanee dello stato della VM). Ci sono varie possibilità per implementare lo storage, o si utilizza quello locale di Linux *Local Volume Manager (LVM)*, oppure si fa riferimento a servizi come *Swift* e *Ceph*, che sono soluzioni di storage ad oggetti, caratterizzate da un'estrema scalabilità e disponibili tra i servizi aggiuntivi di OpenStack.

Keystone (Identity)

Keystone è utilizzato per l'autenticazione e autorizzazione dei vari servizi di OpenStack. Questo servizio è organizzato come un gruppo di tanti servizi esposti su uno o più end-point. Quando si cerca di accedere ad uno dei servizi di OpenStack, la chiamata verrà autenticata e validata con le credenziali dell'utente attraverso il servizio Identity, se questa validazione va a buon fine viene creato e restituito un token e l'URL dell'endpoint che espone il servizio desiderato. Con queste informazioni, l'utente può effettuare la chiamata, inserendo nel campo dell'header "X-Auth-Token" il token appena ricevuto, e riuscendo così ad utilizzare il servizio.

Esistono poi numerosi dettagli sulle politiche di controllo degli accessi, che però esulano dai nostri scopi, ciò che ci interessa è che il Keystone è l'end-point che ci garantisce l'accesso ai servizi di OpenStack.

Glance (Image)

Glance è un servizio che si occupa della memorizzazione delle immagini software delle VM. Tale immagini servono nella fase di boot per il lancio dell'istanza. Per ragioni di scalabilità è possibile utilizzare come *back-end*, ugualmente al caso di cinder, i servizi swift e ceph.

Horizon (Dashboard)

Horizon rappresenta l'interfaccia grafica di OpenStack ed è uno strumento molto potente per configurare, in modo semplice, tutti i servizi della piattaforma. Vedremo alcuni esempi di come viene utilizzata nell'appendice A, anche se la maggior parte delle configurazioni è stata fatta, per maggiore comodità e flessibilità, attraverso la *Command Line Interface* CLI e resa disponibile attraverso degli script.

6.1.2 Servizi aggiuntivi

Introduciamo adesso alcuni servizi, non strettamente necessari per le operazioni di base, che però risultano ugualmente interessanti.

Heat

Heat fornisce un servizio di orchestrazione alla piattaforma OpenStack. Nello specifico, l'obiettivo di tale servizio è quello di creare uno strumento, utilizzabile sia dall'uomo che dalla macchina, che sia in grado di lanciare e gestire applicazioni composte e complesse sulla piattaforma OpenStack. Questo viene effettuato attraverso l'utilizzo di template, sotto forma di file di testo (formato yaml), che consentono di specificare tutta una serie di caratteristiche delle applicazioni da istanziare. Tale formato è molto simile e mantiene una compatibilità con AWS CloudFormation, che ha lo stesso scopo di heat, ma nel contesto del cloud Amazon Web Services.

Nell'esempio in figura 6.3, abbiamo un template heat che consente il deploy di una singola istanza di OpenStack. A partire dall'alto, è possibile specificare una serie di parametri, che verranno passati ai servizi di OpenStack come caratteristiche per la creazione dell'istanza. In particolare è possibile passare il nome della coppia di chiavi RSA per creare il collegamento sicuro con la stessa, tipicamente viene effettuata l'*injection* della chiave pubblica all'interno della VM per potervi accedere successivamente via SSH. Viene ancora passato come parametro l'ID dell'immagine software precaricata nel repository di glance, in modo da poter così effettuare il boot della VM a partire da essa. Infine viene specificato il tipo di *flavour*, che corrisponde ad un oggetto che astrae le caratteristiche necessarie per l'esecuzione di quella VM (numero di CPU, quantità di RAM e dimensione del volume in GB). Le risorse (resources) corrispondono alle istanze effettive che si vogliono creare, in questo caso una, che prende come parametri quelli definiti sopra. La tipologia di istanza è identificata come `OS::Nova::Server`, questo sta ad indicare che si tratta di una VM da

```
1 description: Simple template to deploy a single compute instance
2
3 parameters:
4   key_name:
5     type: string
6     label: Key Name
7     description: Name of key-pair to be used for compute instance
8   image_id:
9     type: string
10    label: Image ID
11    description: Image to be used for compute instance
12  instance_type:
13    type: string
14    label: Instance Type
15    description: Type of instance (flavor) to be used
16
17 resources:
18   my_instance:
19     type: OS::Nova::Server
20     properties:
21       key_name: { get_param: key_name }
22       image: { get_param: image_id }
23       flavor: { get_param: instance_type }
```

Figura 6.3: Esempio di template Heat per OpenStack.

istanziare su uno dei compute node di nova. Ovviamente è possibile creare con questo meccanismo dei servizi molto più complessi, ad esempio le tipologie di risorse sono varie, potrebbero anche riguardare la rete e quindi la creazione di reti di supporto attraverso neutron. Anche i parametri sono i più svariati e permettono di personalizzare le istanze in un modo estremamente flessibile. Per una guida completa si rimanda alla documentazione di OpenStack [36].

Celiometer

Celiometer è parte del progetto Telemetry ed è un servizio volto all'accumulo dei dati relativi all'utilizzo delle risorse, sia fisiche che virtuali. Telemetry più in generale è costituito anche da altri componenti, tra cui Aodh, Gnocchi e Panko, ed è volto alla collezione di varie metriche, utili a diversi scopi nel framework di OpenStack. I dati vengono ottenuti, in parte sondando lo stato dell'infrastruttura, in altra parte attraverso le notifiche provenienti dagli altri servizi di OpenStack. Le utilità di tale servizio possono essere le più disparate, di solito è utilizzato per calcolare l'importo dovuto dai clienti (attraverso le ore di utilizzo delle istanze), ma può essere adoperato anche per migliorare la gestione dell'infrastruttura e valutare alcune situazioni di allarme, laddove è necessario effettuare operazioni specifiche.

Trove

Trove è un Database-as-a-Service per OpenStack, ovvero permette agli utenti di creare ed utilizzare database relazionali e non, in modo semplice e veloce. Tutto questo avviene scaricandoli dall'onere della gestione amministrativa, come l'istanziamento, la configurazione, il backup e il monitoraggio della base di dati.

Sahara

Sahara ha come obiettivo quello di fornire un modo veloce ed efficace, per effettuare il deploy di cluster su OpenStack. Alcuni esempi sono dati da Hadoop, Spark e Storm.

Magnum

Magnum è un servizio sviluppato dall'OpenStack Containers Team, al fine di poter effettuare l'orchestrazione dei container, in modo simile ad altri prodotti come Docker Swarm, Kubernetes e Apache Mesos. Magnum utilizza heat per orchestrare una o più immagini software, dove sono contenuti Docker e Kubernetes. Il *magnum conductor* poi utilizza Kubernetes per la gestione dei docker su tale VM istanziata, come si vede in modo semplificato in figura 6.4.

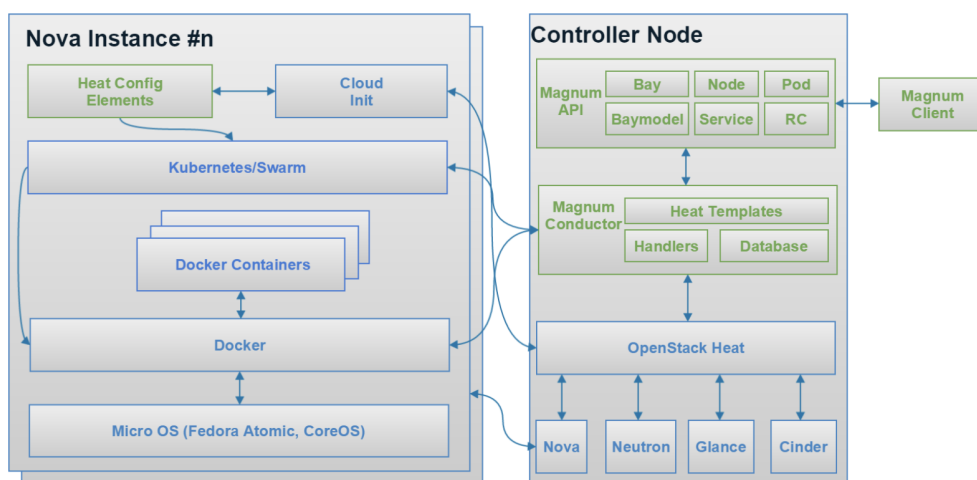


Figura 6.4: Magnum Service (fonte: [wiki openstack](https://wiki.openstack.org/wiki/Magnum)).

Un approccio del genere, dove vari container vengono istanziati su una stessa VM, è molto utilizzato anche per conciliare le esigenze di sicurezza e flessibilità nella gestione dei servizi.

IroniC

IroniC è un servizio di OpenStack che mira ad offrire la possibilità di istanziare macchine fisiche, anziché macchine virtuali. Il tutto ovviamente presuppone l'utilizzo di opportuni driver sulle macchine fisiche che fungono da compute node. Questo servizio può essere di interesse, nel caso interesse si abbiano esigenze relative alle performance o nel caso si abbia necessità di accesso a dispositivi hardware che non possono essere virtualizzati.

Tracker

Tracker è un servizio estremamente interessante, perché porta l'orchestrazione NFV in OpenStack. Come si evince dalla figura 6.5, infatti, l'idea di Tracker è quella di sfruttare tutta la gestione dell'infrastruttura messa a disposizione da OpenStack e inserire in aggiunta solo i moduli essenziali all'orchestrazione delle funzioni virtuali. Nella parte alta della figura è mostrata l'API, che consente di manipolare lo strumento Tracker con la GUI solita di OpenStack (Horizon) o attraverso CLI. Nella parte bassa è presente una seconda interfaccia (Infra Driver), che corrisponde a quella in grado di far comunicare Tracker con gli altri servizi di OpenStack, al fine di istanziare il servizio di rete. Tracker per fare questo utilizza heat, infatti, come abbiamo visto, tale strumento può essere molto comodo per orchestrare istanze virtuali, in modo del tutto automatico e prestandosi all'utilizzo tramite script (per via del formato yaml). Una volta definite le vie di comunicazione di Tracker, vediamo che all'interno è composto dagli elementi essenziali per la gestione di un framework NFV:

un catalogo di risorse, un orchestratore di risorse e un VNF Manager. Per quanto riguarda il data model tutti i descrittori e le configurazioni vengono espresse tramite il linguaggio TOSCA, accennato nel capitolo 2. Tutte le operazioni vengono messe in opera attraverso gli strumenti di orchestrazione inferiori, ovvero attraverso heat e le chiamate al Keystone.

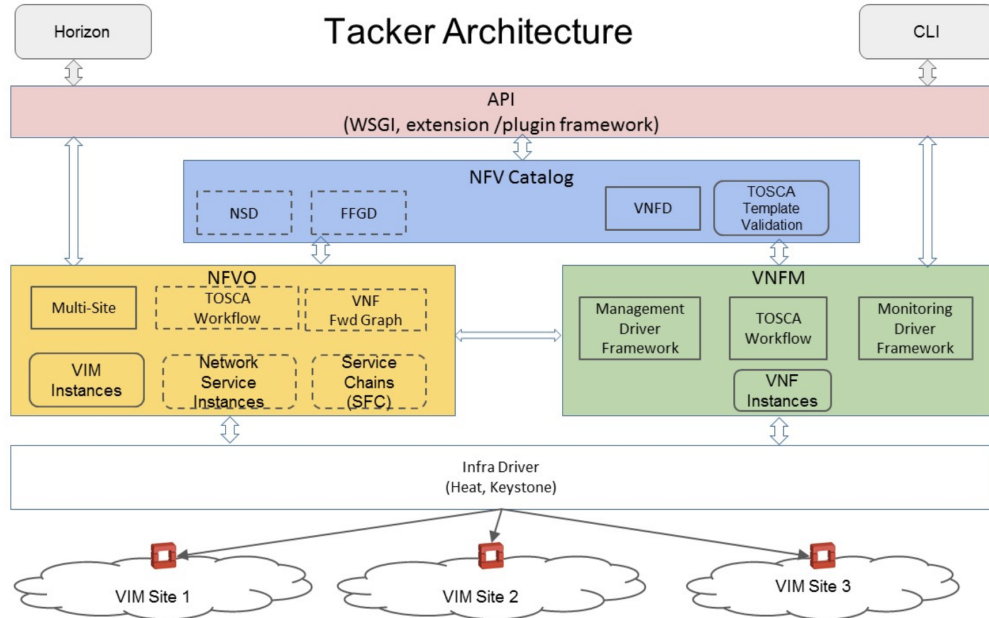


Figura 6.5: Tracker Architecture (fonte: [wiki openstack](https://wiki.openstack.org/wiki/TrackerArchitecture)).

6.2 Open Source MANO

Abbiamo presentato, nel capitolo 3, Open Source MANO da un punto di vista meramente architettonico, illustrando quali fossero i tre componenti fondamentali: RO, SO e VCA. Andiamo adesso ad approfondire OSM sotto una prospettiva diversa, più funzionale e legata ad aspetti pratici, che ci torneranno utili nella presentazione della nostra soluzione. Per quanto riguarda, invece, la documentazione ed esempi di utilizzo pratico rimandiamo all'appendice B.

6.2.1 Workflow

Le funzionalità che un sistema di gestione ed orchestrazione NFV deve garantire sono in estrema sintesi tre:

- Modellazione di un servizio di rete, attraverso un'interfaccia grafica o mediante dei descrittori in formato testo.
- Dispiegamento automatico di un servizio di rete, a partire dal modello.
- Configurazione automatica di ognuna della VNF del servizio in oggetto, attraverso strumenti dell'orchestratore stesso o di terze parti.

Open Source MANO mette a disposizione, per essere in linea con le suddette funzionalità, un'interfaccia grafica e dei descrittori specifici per la modellazione, un orchestratore di alto livello (SO) per l'istanziamento del servizio di rete e, infine, diversi strumenti per la configurazione delle VNF e del servizio di rete stesso.

Per quanto riguarda l'interfaccia grafica nello specifico, vedremo tutte le funzionalità messe a disposizione dalla stessa nell'appendice B. I descrittori, compresi di ogni loro caratteristica e funzionalità, saranno presentati nella prossima sezione 6.2.2, mentre le configurazioni in dettaglio nella 6.2.3.

Qui ci occuperemo, invece, di presentare le operazioni che consentono il dispiegamento del servizio di rete, contestualizzandolo nel workflow complessivo di OSM. Il tutto a partire dalla fase di on-boarding dei descrittori, fino alla fase in cui il servizio di rete è online ed adeguatamente configurato.

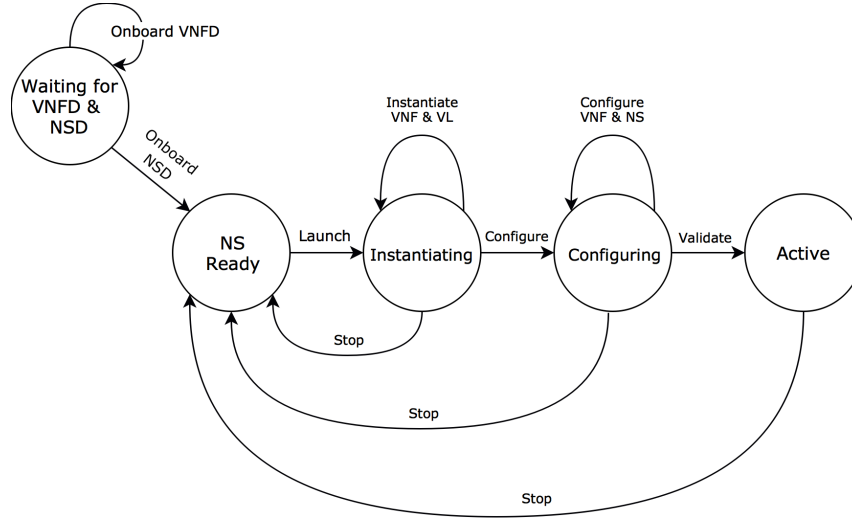


Figura 6.6: Fasi Deployment di un Network Service.

Come si evince dalla figura 6.6, tutto il processo inizia con l'on-boarding, prima dei VNF Descriptor e successivamente dei Network Service Descriptor. Una volta effettuata questa operazione, è possibile lanciare il servizio di rete, passando così alla fase di istanziazione (*Instantiating*). In tale fase vengono inizializzati, attraverso il VIM, le VNF e i link virtuali, passando una volta terminata questa fase alla configurazione delle VNF. Nella fase di *Configuring*, infatti, vengono istanziate le risorse necessarie alla gestione delle VNF e del servizio di rete, inoltre vengono applicate le configurazioni iniziali desiderate. Quando questa serie di fasi viene portata a termine, possiamo ritenere il nostro servizio di rete nello stato di *Active*, quindi attivo e online.

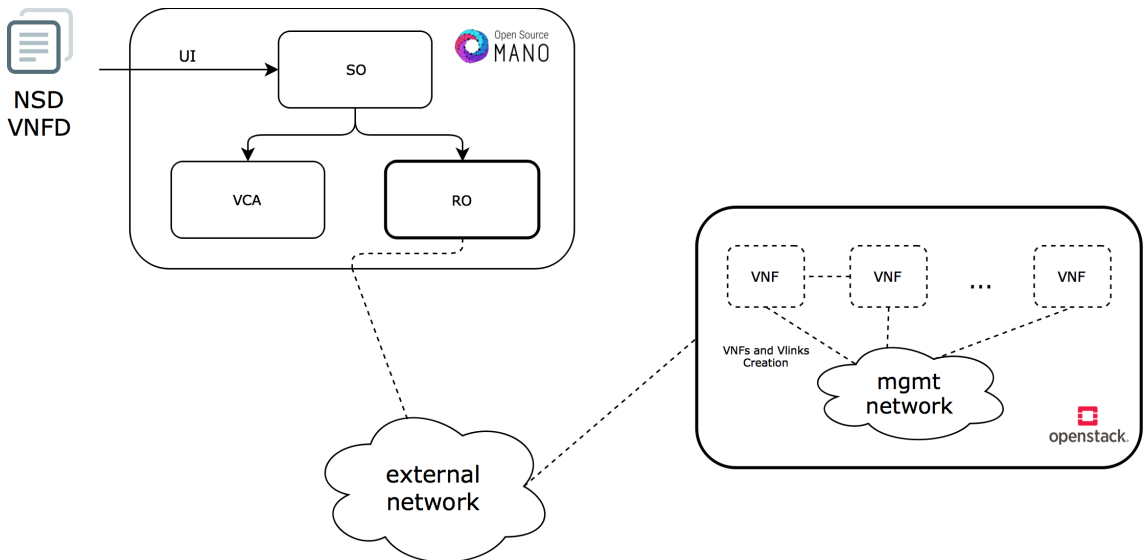
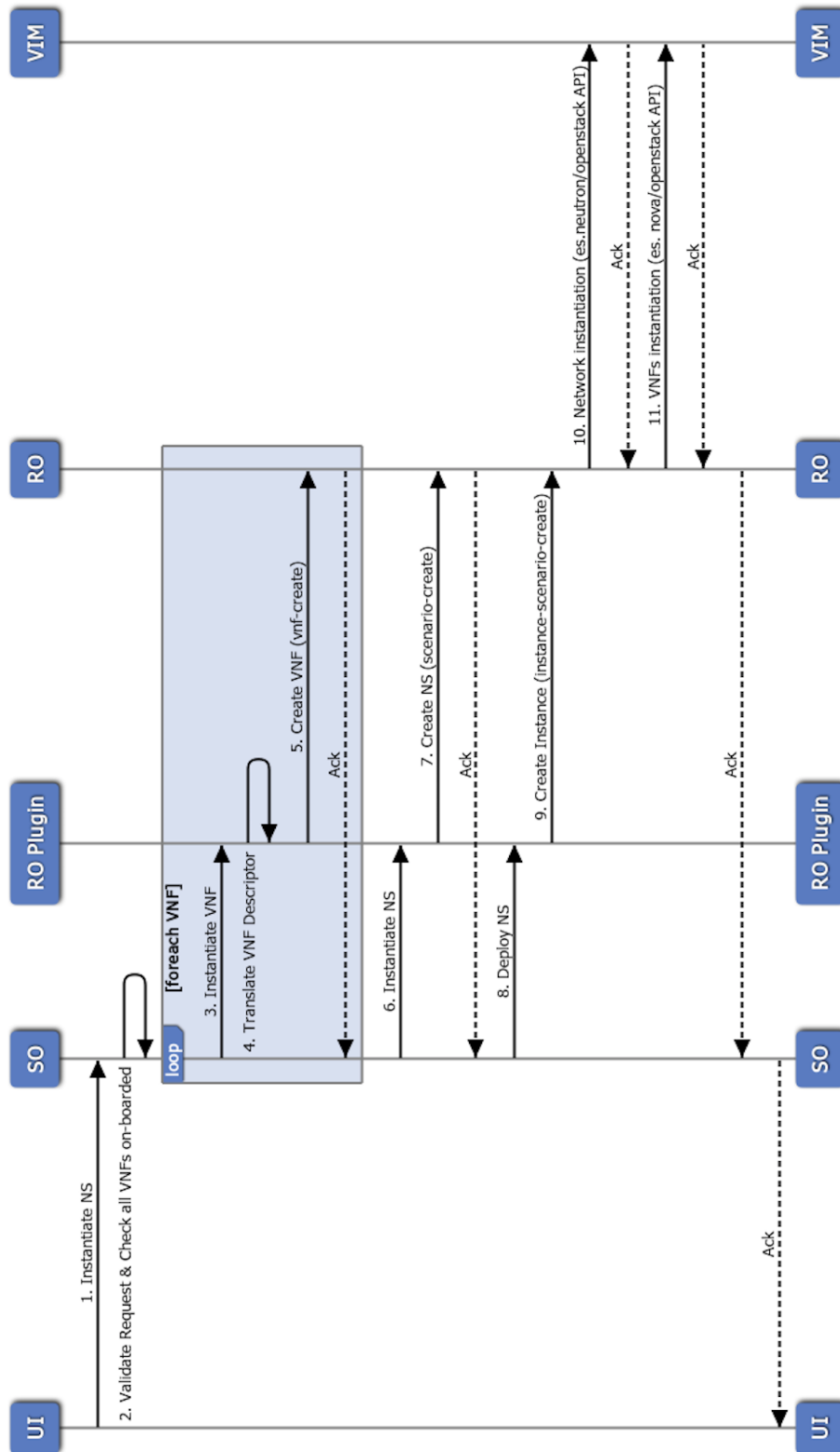


Figura 6.7: Architettura OSM (fase di *Instantiating*).

Figura 6.8: Workflow OSM (fase di *Instantiating*).

Le fasi di particolare interesse sono quelle di istanziazione e configurazione. Ripercorriamole, seguendo due approcci, uno architetturale e uno relativo al workflow. Partendo dalla prima delle due, osserviamo l'architettura presentata in figura 6.7. Tale architettura prevede un modulo contenente l'orchestratore OSM, con i relativi sottomoduli, e un secondo modulo che rappresenta il

VIM e l'infrastruttura NFVI. Sono stati accorpati in questo caso per ragioni di semplicità, più avanti vedremo più in dettaglio l'interazione tra VIM e NFVI. I due moduli presentati comunicano attraverso una rete esterna, questo permette concettualmente di avere, distribuiti su nodi diversi di rete, l'orchestratore e i vari VIM, concetto che rientra nella caratteristica di multi-PoP presentata nel capitolo 3.

Andando a dettagliare cosa accade in figura 6.7, possiamo notare, durante la fase rappresentata, come il modulo chiave sia in questo caso l'RO. Il Service Orchestrator, infatti, istruisce l'RO su quali sono le VNF, i link virtuali e gli NS desiderati, questo avviene attraverso la traduzione dei descrittori dal formato presentato ad OSM a quello adatto per l'RO. Esiste un modulo, come abbiamo visto nel capitolo 3, che permette tale traduzione, sfruttando il modello yang dei dati relativo ai descrittori. Una volta disponibile la nuova versione di descrittori per l'RO, quest'ultimo contatta il VIM, attraverso le API messe a disposizione, e gli specifica le operazioni che deve effettuare sull'infrastruttura.

Vengono così creati i link virtuali e le VNF desiderate, possiamo rivedere nel dettaglio la sequenza delle operazioni attraverso il workflow in figura 6.8.

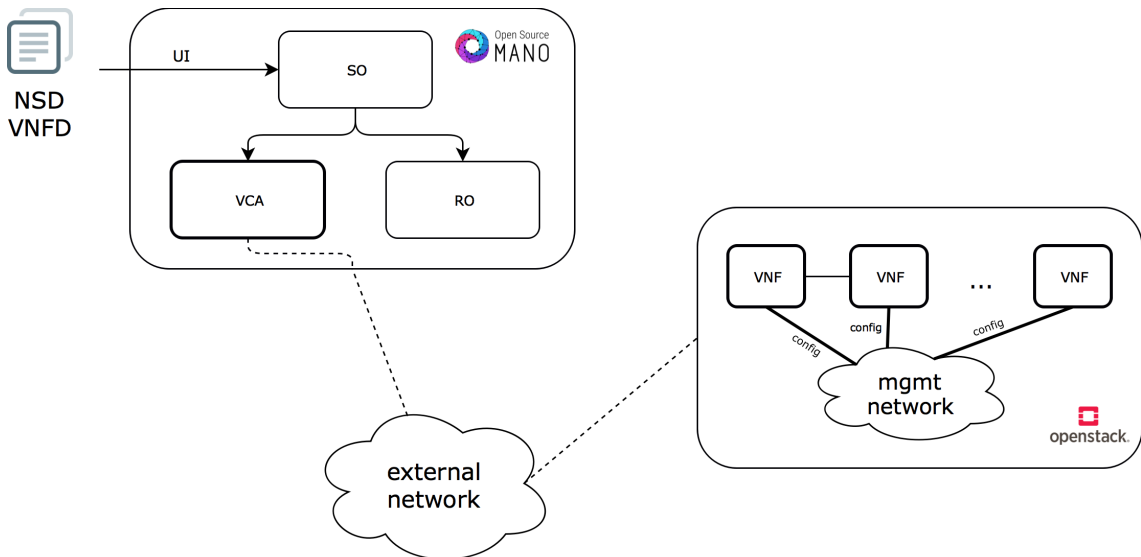
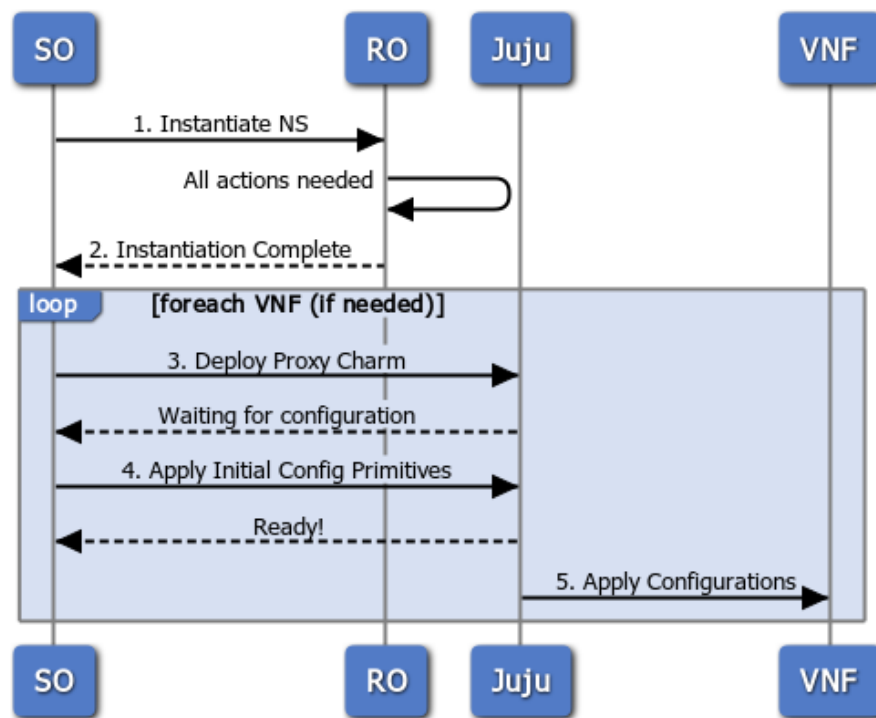


Figura 6.9: Architettura OSM (fase di *Configuring*).

Passiamo ora all'analisi della fase di *Configuring*, nella quale, sotto il profilo architetturale il modulo chiave è il VCA, nel quale ricordiamo vi sono gli strumenti di configurazione delle VNF (nel nostro caso Juju). Questo modulo comunica attraverso la rete esterna con il VIM, come si evince dalla figura 6.9, in particolare modo ha accesso alla *management network*, rete che gli permette di raggiungere su un'interfaccia nota (ad es. eth0) le VNF. Avendo la possibilità di raggiungere le VNF, è in grado di creare un canale di comunicazione sicuro con le stesse (SSH) e fare l'iniezione delle configurazioni, così da ottenere la personalizzazione desiderata per ogni singola VNF. Dato che il modulo VCA è in grado di collegarsi durante tutto il ciclo di vita delle VNF alle stesse, risulta possibile effettuare delle configurazioni anche successive a quella iniziale, permettendo una notevole flessibilità in termini di operazioni da poter effettuare sulla VNF (i dettagli in sezione 6.2.3).

La figura 6.10 mostra qual'è il workflow che porta alla configurazione delle VNF. In prima analisi sono riportate le operazioni per l'istanziamento dalla fase precedente (Instantiate NS). Subito dopo vengono ciclicamente effettuate una serie di operazioni, che coinvolgono in parte il VNF Manager Juju e il System Orchestrator. In ordine vengono istanziati degli oggetti chiamati proxy charm, che vedremo in dettaglio nella sezione 6.3.3, vengono applicate le configurazioni iniziali a questi ultimi ed infine, vengono applicate le vere e proprie configurazioni alla VNF. Rimandiamo alla sezione 6.2.3 maggiori dettagli.

Figura 6.10: Workflow OSM (fase di *Configuring*).

6.2.2 Descrittori

Vi sono due tipologie di descrittori supportati in Open Source MANO:

- *VNF Descriptor (VNFD)*: descrittore della funzione virtuale di rete;
- *NS Descriptor (NSD)*: descrittore del servizio di rete.

Entrambi sono tipicamente espressi in formato yaml e definiti attraverso un modello YANG, tale modello è reperibile attraverso i repository ufficiale OSM [37]. Per ognuno dei descrittori esistono poi diversi file di supporto che è possibile utilizzare, per questo motivo il descrittore finale viene distribuito sottoforma di pacchetto, contenente il file yaml di descrizione e tutti i file di supporto. Andiamo, per ognuna delle due tipologie di descrittore, a capire qual'è il modello di dati e come è possibile costruire il package finale.

VNFD

I descrittori della VNF, o VNFD, specificano tutte le caratteristiche desiderate per una particolare funzione di rete. Vi sono dati identificativi della funzione virtuale, prestazioni richieste, configurazioni e dati di supporto, il tutto all'interno del descrittore yaml che presenteremo a breve. Prima di descrivere i campi che lo compongono, valutiamo prima come viene modellata una funzione di rete in OSM.

Facendo riferimento alla figura 6.11, è possibile evincere che una singola VNF è composta a sua volta da tre elementi:

- Il descrittore vero e proprio modellato come VNF nella figura.
- La *Virtual Deployment Unit (VDU)* che corrisponde all'unità elementare che può essere istanziata. Questa coincide, nel caso in cui la virtualizzazione sia con hypervisor, alla singola macchina virtuale, a cui corrisponde una sua immagine software. Ovviamente, nel

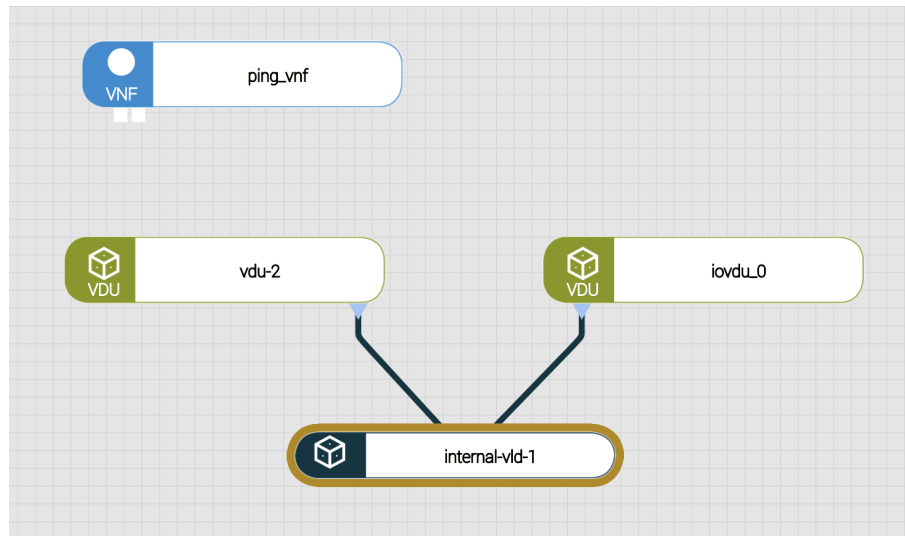


Figura 6.11: Modello di una VNF in Open Source MANO.

caso di altre tipologie di virtualizzazione, è possibile farla coincidere con la singola unità virtuale (sempre con la relativa immagine software di base), un esempio può essere il caso di light virtualization, in particolare un singolo docker e la sua immagine di base. Una singola VNF può essere dotata di più unità virtuali di base, che magari presentano ognuna la sua immagine software di base diversa.

- L' *Internal Virtual Link (IVL)* che corrisponde ad un collegamento interno tra le varie unità virtuali, costituenti la VNF.

I campi principali nel caso del descrittore generale della VNF sono i seguenti:

- **Dati Identificativi:** ID, Name, Short Name, Vendor, Vendor Logo, Version. Corrispondono alle caratteristiche che permettono di identificare univocamente una VNF.
- **Type of Node:** questo è un parametro che serve al Forwarding Graph per collocare la VNF all'interno della SFC, in base al suo ruolo. Viene utilizzato quando è integrato OpenDayLight per poter effettuare il mapping tra la funzione di rete e i tre ruoli supportati: Classifier, Service Function o Service Function Forwarder.
- **Meta-data:** dati che sono utili all'interfaccia grafica, per collocare gli elementi del modello o per mappare i campi con i file di supporto allegati al descrittore. Utili quando viene modellato il servizio attraverso l'interfaccia grafica.
- **VNF Configuration:** possono essere specificati in questo campo le informazioni riguardo alla configurazione delle VNF, che come vedremo poi in dettaglio possono essere attraverso: cloud-init, script o JUJU. Ovviamente vanno inseriti ulteriori dettagli negli appositi sottocampi, a seconda della configurazione scelta (es. charm, nel caso di Juju).
- **Config primitives:** sono delle funzioni, o per meglio dire delle azioni, che possono essere chiamate quando il servizio è online. Queste sono eseguite attraverso il metodo di configurazione specificato, tipicamente Juju e sono attivabili tramite l'interfaccia grafica di OSM. Ognuna di esse ha tutta una serie di parametri che possono essere passati durante l'attivazione.
- **Initial Config Primitives:** un sottoinsieme delle precedenti azioni che devono essere eseguite immediatamente dopo la fine della fase di istanziazione della VNF.
- **Mgmt Interface:** interfaccia dove vengono passate le configurazioni, tipicamente quella utilizzata da Juju.

- **Internal VLD:** indica la presenza o meno di uno o più link virtuali interni alla stessa VNF.
- **IP profiles:** permette di creare dei nodi di rete che verranno istanziati dal VIM. Possono essere configurati con indirizzi IP, utilizzo del DHCP, indirizzo del gateway e molto altro ancora.
- **Connection point:** i punti di connessione, rappresentano le interfacce ad alto livello che espone la VNF verso l'esterno.
- **VDU:** indica quali unità virtuali fanno parte della VNF.

Per quanto riguarda le unità virtuali abbiamo i seguenti campi fondamentali:

- **Dati identificativi:** ID, Name, Description. Come in precedenza rappresentano i dati identificativi.
- **Number of instances of VDU:** indica il numero di copie della stessa istanza in fase di deployment, per scalare eventualmente la VNF.
- **MGMT Virtual PCI Address:** l'indirizzo virtuale su cui viene mappata la porta di management.
- **Image Name:** il nome dell'immagine software.
- **Image checksum:** il checksum dell'immagine.
- **Cloud-init input:** il file di input della cloud-init.
- **VM Flavor:** numero di CPU, quantità di RAM, spazio su disco e altre caratteristiche della macchina virtuale che deve ospitare la VNF.
- **Interface:** le interfacce interne ed esterne della VDU, a seconda che siano collegate a link interni o esterni. Vengono mappate nel secondo caso con i connection point della VNF.

I link virtuali interni infine:

- **Dati identificativi:** sempre per l'identificazione univoca del link interno.
- **Type:** tipologia del link, ad esempio ELAN.
- **Internal connection point:** sono i punti, dichiarati in precedenza all'interno delle VDU, che vengono collegati da tale link.

Per quanto riguarda il formato dei descrittori vedremo alcuni esempi pratici nel capitolo 8, intanto ci interessa affrontare l'ultimo passaggio e capire come viene costruito il pacchetto del descrittore. Innanzitutto le informazioni contenute nel modello appena delineato vengono, come già accennato, incapsulate in un file yaml, successivamente viene creato un direttorio in cui è presente quest'ultimo insieme a tutti gli altri file di supporto.

Come si evince dalla figura 6.12, vi sono varie cartelle all'interno del package, tra cui quella per il charm, che vedremo a breve cos'è, gli script di configurazione e il file descrittore (`example_vnfd.yaml`). Il formato di compressione utilizzato per il pacchetto è solitamente `tar.gz`.

```

/example_vnfd
├── charms
│   ├── example_charm
│   │   └── charm_filesndirs
│   └── cloud_init
│       ├── cloud_init.cfg
│       └── icons
│           └── ubuntu-logo14.png
├── keys
│   ├── test4.pem
│   └── test4.pub
└── example_vnfd.yaml

```

Figura 6.12: Esempio package VNFD.

NSD

Come nel caso delle VNF, andiamo innanzitutto a vedere come viene modellato, in Open Source MANO, il Network Service.

A tale scopo ci serviamo della figura 6.13, dove si evince che il Network Service è modellato attraverso tre componenti:

- Il descrittore generale del servizio di rete come nel caso delle VNF dato da “NS”.
- Il *Virtual link (VL)* ovvero un’astrazione dei link tra le varie VNF, che rappresentano i collegamenti tra le interfacce esterne delle stesse. Questo link viene tipicamente mappato con una rete virtuale creata o messa a disposizione dal VIM.
- Il *Forwarding Graph* che costituisce l’astrazione del grafo e che quindi contiene le informazioni riguardo ai percorsi e ai classificatori di pacchetti. Tralascieremo questo elemento per adesso, per poi riprenderlo nel capitolo 9, capitolo riguardante gli sviluppi futuri.

Andiamo ad analizzare ora i campi presenti nel descrittore del servizio di rete (NS):

- **Dati identificativi:** ID, Name, Short Name, Vendor, Vendor logo, Description, Version. Ugualmente al caso VNF, servono per identificare in modo univoco il servizio di rete.
- **Meta-data:** utili all’interfaccia grafica per la disposizione dei componenti e i path dei file di supporto.
- **Connection Point:** punti di connessione tra più servizi di rete. In caso di necessità è possibile connettere in catena più NS, anche appartenenti a servizi di orchestrazione diversi. Qui ci si rifà al discorso dell’orchestrazione gerarchica e il concetto di Network Slice, che abbiamo accennato nel capitolo 3; argomento molto attuale nell’ambito delle reti 5G, ma che esula dagli scopi di questo lavoro.
- **Scaling Group Descriptor:** è possibile attraverso questo descrittore, più propriamente una serie di parametri, isolare un gruppo di VNF all’interno dello stesso che possono scalare, secondo certi parametri e all’occorrenza di alcuni eventi.
- **VNFFGD:** è il descrittore del *Forwarding Graph* presentato precedentemente. All’interno di un Network Service è possibile definire uno o più grafi di questo tipo.
- **IP profiles:** come nel caso delle VNF, anche se adesso più efficace ed utile, è possibile definire dei nodi virtuali di rete, con tutte le caratteristiche necessarie. Si possono assegnare un DHCP, un gateway o dare altri parametri di configurazione per tale nodo. Risulta interessante in questo caso, perché è possibile attraverso il VIM creare questi nodi di rete ed assegnarli direttamente ai link virtuali del servizio di rete, senza così aver bisogno di doverli definire in anticipo sull’infrastruttura.

- **Initial Service Primitive:** un insieme di funzioni che vanno eseguite all'inizio del deploy del servizio di rete.
- **Terminate Service Primitive:** un insieme di funzioni che vanno eseguite durante la terminazione dell'NS.
- **Key pairs:** si usa per configurare una lista di chiavi pubbliche da iniettare durante l'istanziamento del servizio di rete.
- **Users:** una lista di utenti da inserire nella cloud-init, vedremo poi cos'è quest'ultima.
- **VLD:** la lista dei link virtuali che fanno riferimento al servizio in oggetto.
- **Constituent VNFD:** la lista delle VNF che compongono il servizio, tramite ovviamente l'indicazione dei VNFD.
- **Monitoring Param:** parametri definiti per il monitoraggio di alcuni indicatori o metriche. Da utilizzare in relazione con gli strumenti di monitoraggio descritti in parte nel capitolo 3.
- **Service Primitive:** funzioni relative questa volta al servizio di rete, quindi la possibilità di effettuare operazioni su gruppi interi di VNF, opportunamente definite in precedenza sulle singole VNF o caricate a questo livello attraverso gli specifici parametri.

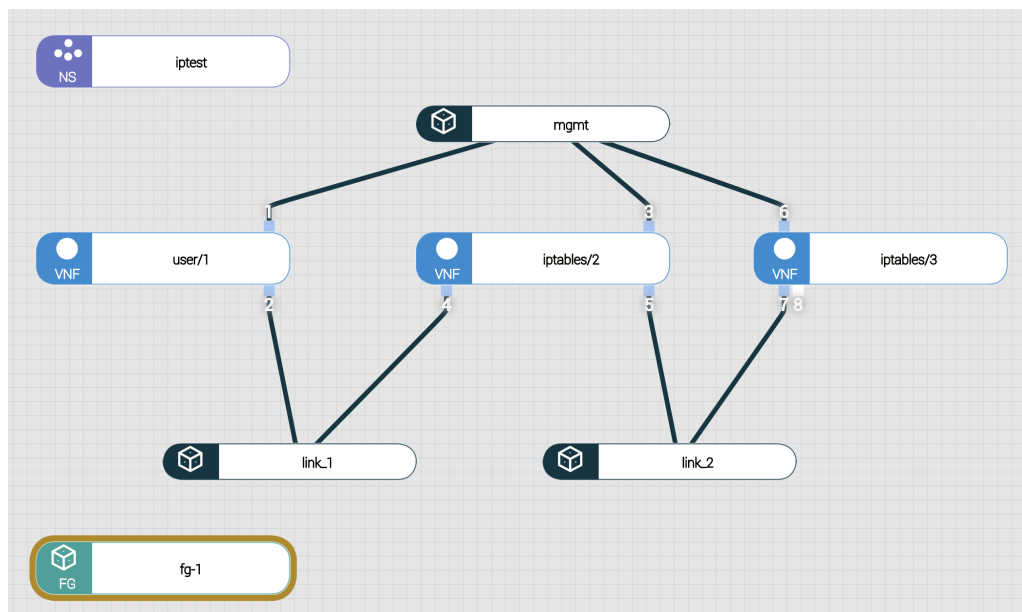


Figura 6.13: Modello di un Network Service in Open Source MANO.

Passiamo ora ai campi dei link virtuali esterni:

- **Dati identificativi:** ID, Name, Short Name, Vendor, Description, Version. Ugualmente al caso NS, servono per identificare in modo univoco il link virtuale.
- **Type:** tipo di link esterno (es. ELAN), vi sono alcuni parametri come la banda richiesta.
- **Flag VIM management:** indica se il link in questione è quello che corrisponde alla rete di gestione nel VIM.
- **Init Params:** una serie di parametri che consentono di identificare all'interno del VIM la rete con cui mappare il link in questione, oppure di identificare l'IP profile creato a tale scopo.
- **VNFD connection point:** rappresentano i punti di connessione (interfacce) delle VNF, punti che sono stati dichiarati precedentemente nel descrittore della VNF.

```

/example.nsd
├── config
│   └── init.cfg
├── icons
│   └── cirros_ns.png
└── example_nsd.yaml

```

Figura 6.14: Esempio package NSD.

Esempi pratici di tali descrittori verranno proposti nel capitolo 8 di implementazione della soluzione. Ora vediamo come viene costruito il pacchetto NSD.

Come si evince dalla figura 6.14, simmetricamente al caso VNF il descrittore sarà costituito da un file yaml che ospiterà questi parametri (e altri), più dei file di supporto che possono essere utili durante la configurazione o per motivi di identificazione del servizio di rete. Nello specifico caso in figura, sono presenti, a supporto del descrittore, il logo per identificare il servizio e un file di configurazione che può essere lanciato, ad esempio, durante l'istanziamento del servizio. Il tutto converge come in precedenza in un pacchetto `tar.gz`.

6.2.3 Configurazione delle VNF

Per quanto riguarda la configurazione delle VNF, Open Source MANO dispone di tre meccanismi per effettuarla, come riportato in figura 6.15:

- *Day-0 Configuration*: configurazione effettuata durante l'istanziamento delle VNF, tramite la sottomissione di script all'unità VDU o attraverso la cloud-init.
- *Day-1 Configuration*: configurazione che avviene dopo l'istanziamento delle VNF e può essere ripetuta o eseguita all'occorrenza di eventi specifici. Questa configurazione avviene attraverso il VNF Manager Juju della Canonical.
- *Day-2 Configuration*: una tipologia di configurazione sperimentale da parte di OSM, introdotta con la Release THREE del loro prodotto. Questa configurazione permette in linea teorica di effettuare operazioni che riguardano il livello del servizio di rete, questo significa che quando lo stesso è online è possibile effettuare operazioni non solo riguardanti singole o gruppi di VNF, ma di agire anche sulla configurazione del VIM e delle variabili che regolano l'NS. Un esempio potrebbe essere quello di cambiare dinamicamente il sito dove si è effettuato il deploy del servizio (da un VIM all'altro), senza dover manualmente intervenire per poterlo fare, ma a seconda di policy stabilite a priori in fase di definizione della configurazione. Per ora tale configurazione è del tutto sperimentale ed esula dai nostri scopi, quindi non ci soffermeremo a trattarla ulteriormente.

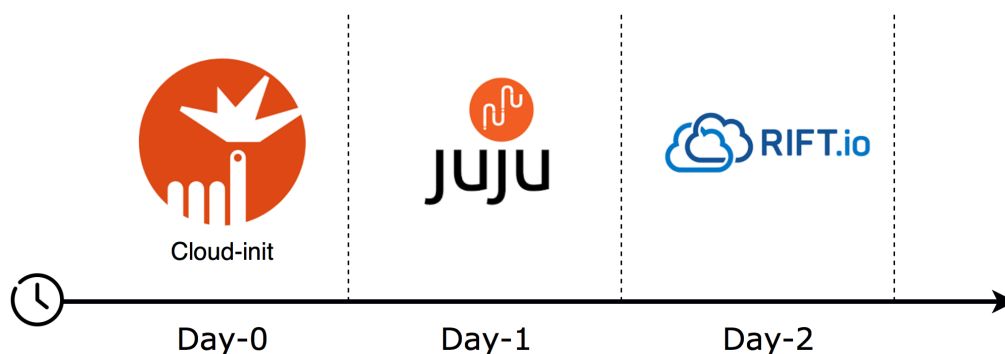


Figura 6.15: Meccanismi di configurazione di Open Source MANO.

Per quanto riguarda la Day-0 Configuration, noi utilizzeremo cloud-init, in quanto risulta molto più flessibile rispetto all'iniettare degli script direttamente nella VDU, capiremo poi i vantaggi a partire dalla sezione 6.2.4. Per la Day-1 Configuration, utilizzeremo come previsto Juju della Canonical, ma prima di parlare del modo specifico con cui viene utilizzato, faremo un'introduzione a questo strumento, di interesse per i nostri scopi anche in linea agli sviluppi futuri, in sezione 6.3. Per la parte specifica legata a OSM si rimanda alla sezione 6.3.3, dove sono presentati i proxy charm, strumento che permette al VCA di configurare le varie VNF.

6.2.4 Cloud-Init

Cloud-init nasce come un pacchetto di script che consente la gestione della fase di inizializzazione delle istanze cloud. Per comprendere al meglio l'utilità di cloud-init dobbiamo inserirla nel contesto cloud, definendo innanzitutto il concetto di *cloud image*.

Le cloud image sono dei *template* di sistemi operativi e come tali ogni singola istanza degli stessi viene lanciata come un clone identico di tale template. Quelli che permettono di personalizzare la singola istanza, sono i dati dell'utente. La cloud-init rappresenta lo strumento che consente di applicare questi dati utente all'istanza, in modo del tutto automatico.

Tipicamente le cloud image vi sono per ogni distribuzione di linux, di sicuro per le più diffuse, ne riportiamo di seguito alcune: Ubuntu, CentOS, Red Hat, Fedora, Arch Linux. La caratteristica fondamentale, oltre alla presenza del pacchetto precaricato cloud-init, è l'essenzialità delle stesse in termini di contenuti, infatti, un'immagine non supera in genere i 200 MByte. Il principio di base è quello di installare tutto e solo quello che è necessario al momento del boot ed evitare di dover precaricare ogni cosa su un'immagine, che diventerebbe di dimensioni considerevoli (tipicamente superiori al GByte). I vantaggi dell'utilizzo di questa tecnologia sono per l'appunto due: il primo relativo alle dimensioni esigue dell'immagine e il secondo il beneficio tratto dal non dover memorizzare tante immagini diverse per ogni esigenza, questo sarebbe disastroso in termini di spazio su disco. Ad esempio in una piattaforma cloud posso avere le immagini base dei diversi sistemi operativi e costruirmi da quelle le istanze personalizzate, modificandole con la tecnologia cloud-init.

Passiamo ora a vedere come personalizzare effettivamente l'istanza. Per fare questo si utilizzano dei file di configurazione nel formato yaml, di solito i file in questione sono i cloud-config. Tali file, secondo un modello di dati specifico, consentono all'utente di dichiarare tutta una serie di preferenze e di operazioni da effettuare sull'istanza nella fase di inizializzazione. Come fa poi la piattaforma cloud a passare i file creati dall'utente alla cloud-init dipende dalla tecnologia. *Red Hat Enterprise Virtualization (RHEV)* utilizza una tecnica per cui, durante la fase di inizializzazione, crea un floppy drive virtuale con il file ad esempio "user-data.txt". Cloud-init su RHEV cerca il floppy drive e legge il file contenuto al suo interno, da quest'ultimo viene a conoscenza delle preferenze dell'utente. Openstack e AWS, invece, mettono a disposizione due tipologie di file: i meta-data e gli user-data. Questi sono disponibili per l'istanza, e quindi alla cloud-init, all'URL "http://169.254.269.254/latest/*-data", sostituendo con l'asterisco la tipologia di dato desiderato. Una volta reperiti i file, la cloud-init sull'istanza può procedere alle configurazioni desiderate.

Vediamo adesso quali sono alcune delle operazioni che si possono effettuare con la cloud-init:

- creare gruppi ed utenti;
- iniettare chiavi SSH;
- scrivere file;
- caricare pacchetti;
- modificare i repository;
- configurare parametri di rete;
- installare ed eseguire ricette Chef o l'equivalente con Puppet (VNF Manager alternativi a quelli da noi utilizzati);

```
1 groups:
2   - ubuntu: [root,sys]
3   - cloud-users
4
5 users:
6   - default
7   - name: foobar
8     groups: cloud-users
9     selinux-user: staff_u
10    expiredate: 2012-09-01
11    lock_passwd: false
12    passwd: secret
13 write_files:
14   - encoding: b64
15     content: CiMgVGhpcyBmaWxlIGNvb...
16     owner: root:root
17     path: /etc/sysconfig/selinux
18     permissions: '0644'
19   - content: |
20       # My new /etc/sysconfig/samba file
21
22       SMBDOPTIONS="-D"
23     path: /etc/sysconfig/samba
24 packages:
25   - pwgen
26   - pastebinit
27   - [libpython2.7, 2.7.3-0ubuntu3.1]
```

Figura 6.16: Esempio di file di configurazione di cloud-init.

- eseguire comandi da terminale;
- cambiare punto di mount.

In sintesi, è possibile effettuare una grande quantità di operazioni attraverso questa tecnologia, per una documentazione completa rimandiamo al sito del progetto [39]. L'unica problematica da affrontare per i nostri scopi è il fatto che OSM supporta solo un sottoinsieme delle funzionalità di cloud-init, vedremo poi nello specifico come usare quelle supportate nel capitolo 8 di implementazione.

L'ultima cosa di interesse è valutare un esempio di configurazione e lo possiamo fare attraverso la figura 6.16, dove è mostrata una possibile configurazione di un'istanza. Iniziando dall'alto, sono stati definiti due gruppi (ubuntu e cloud-users) e un utente che appartiene ad uno dei due (foobar). Per l'utente in questione sono stati definiti dei parametri aggiuntivi come l'utente SELinux e la password, con la possibilità di autenticarsi tramite la stessa. Non ci interessa nello specifico le operazioni che vengono effettuate ma cosa è possibile fare. Andando avanti, oltre alla creazione di gruppi ed utente, è stato scritto un file precisamente al path indicato dal campo relativo. Tale file è codificato in base64 e il contenuto è riportato nello specifico campo, inoltre sono stati impostati i permessi ed il proprietario del file. Ancora un'ulteriore operazione è data dal caricamento dei pacchetti sotto il campo "packages", questo permette all'istanza di caricare in ordine i tre pacchetti per come sono stati elencati (potendo specificare anche la versione).

6.3 Juju

Juju è uno strumento di modellazione per il software operativo su piattaforme cloud. Juju permette di istanziare, configurare, mantenere e scalare applicazioni cloud in modo rapido ed efficiente. Permette inoltre di effettuare queste operazioni su differenti tipi di piattaforme: direttamente su macchine fisiche (MaaS Metal-as-a-Service), su piattaforme IaaS (Openstack, AWS, Google Platform) o su container (LXD hypervisor).

In ambienti moderni, i servizi vengono raramente istanziati in modo isolato, anche le applicazioni più semplici necessitano di almeno una base di dati o di un web server. Openstack, come abbiamo visto, ne è un esempio calzante, infatti, necessita di un insieme complesso di servizi per poter funzionare (nova, neutron, cinder, horizon, glance). Questo ci fa capire che il principio di base, con cui vengono sviluppate le applicazioni distribuite oggi, è quello dell’aggregazione di più servizi specifici e contenuti, i cosiddetti “microservizi”, che insieme collaborano nel fornire le funzionalità desiderate. In un panorama del genere, è utile avere a disposizione uno strumento che permetta di modellare tali servizi, la loro configurazione e le relazioni che intercorrono tra gli stessi. Juju è questo strumento, infatti, cerca mettere insieme tutte queste necessità e di presentare una soluzione flessibile e di semplice utilizzo.

Riportiamo di seguito quali sono le principali funzionalità messe a disposizione da questo strumento:

- *Application modelling*: permette la modellazione dei microservizi attraverso l’utilizzo di due strumenti, i *charm* e i *bundles*.
- *Provisioning*: fornisce le unità logiche e le macchine, per poter istanziare il software, consentendo di scalarle dinamicamente.
- *Deploying*: fornisce meccanismi per l’installazione dei servizi sulle unità e la loro configurazione.
- *Monitoring and management*: mette a disposizione degli strumenti per poter effettuare la gestione ed il monitoraggio delle macchine e delle unità logiche.

Nelle prossime sezioni andremo ad approfondire ognuno di questi aspetti.

6.3.1 Terminologia e definizioni

Presentiamo in questa sezione, tutta una serie di termini che vengono utilizzati nel contesto dello strumento Juju.

Cloud

Il cloud è una risorsa che fornisce le *machine* (istanze), lo storage e la connettività. Può essere assimilato, come già accennato, ad un servizio IaaS (Openstack, AWS), macchina fisica o un Hypervisor LXD.

Controller

Il *Juju Controller* si può identificare come la prima istanza cloud che viene creata. Tale istanza ha il compito di garantire l’accesso al cloud, permette quindi a Juju di contattare le API del cloud al fine di gestire le sue risorse. Il Juju Controller rappresenta anche il punto principale di controllo per quello specifico cloud e si preoccupa di servire tutte le richieste provenienti dal *Juju Client*.

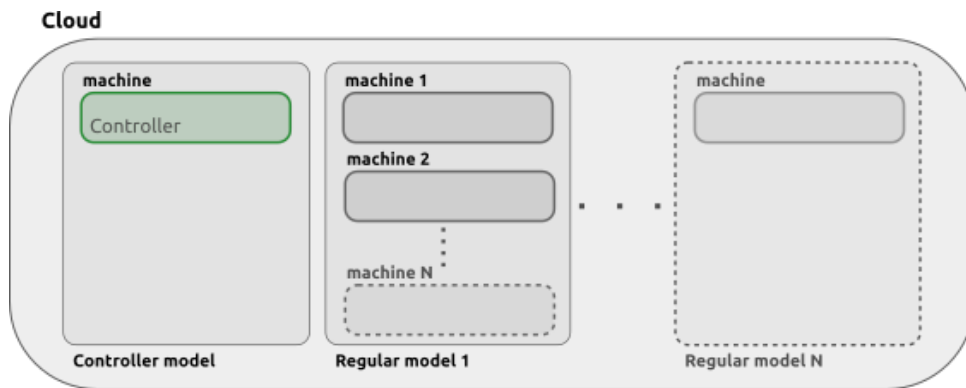


Figura 6.17: Architettura model e controller di Juju (fonte: [juju](#)).

Model

Un *Model* viene associato univocamente ad un controller e rappresenta lo spazio in cui vengono allocate le unità applicative per quel servizio complesso.

Come si evince dalla figura 6.17, sul cloud vengono allocati i vari modelli, il primo si riferisce al *controller model* che rappresenta il modello di gestione di tutte le operazioni su quel cloud, nello stesso viene allocata una singola macchina. Gli altri modelli corrispondono ad insiemi di più macchine, ognuno dei quali viene eseguito in modo isolato ed indipendente dagli altri.

Charm

Il charm corrisponde ad un insieme di script che contengono tutte le istruzioni necessarie per istanziare e configurare le unità applicative sulle macchine. I charm sono disponibili su uno store online “Charm Store”, dove si possono scaricare ed eseguire con estrema facilità. Il charm in altri termini rappresenta l’unità per la modellazione del servizio che si vuole offrire, rendendo semplice e rapida la distribuzione dello stesso e la sua istanziazione. Il modo più banale per capire come funziona il meccanismo di istanziazione dei charm è dato dalla figura 6.18, dove è presentato uno scenario in cui il juju client istanzia un’applicazione su una *juju machine* attraverso l’utilizzo di un charm. In seguito capiremo meglio come funziona tale meccanismo, per ora ci limiteremo a presentare la terminologia.

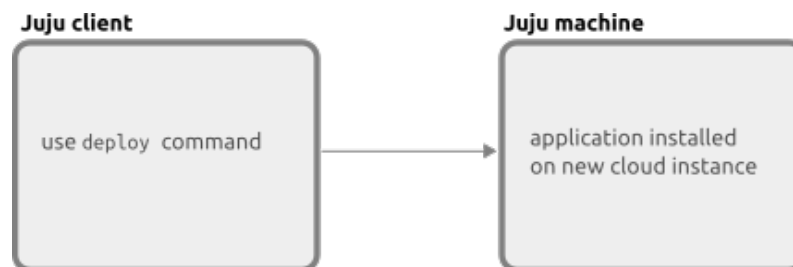


Figura 6.18: Deploy di un singolo charm (fonte: [juju](#)).

Bundle

Un bundle è un insieme di charm che vengono combinati tra loro al fine di ottenere un servizio più complesso. Questo strumento prevede la definizione di un insieme di charm e la definizione delle connessioni tra gli stessi, ovvero la creazione delle relazioni.

Machine

La *machine* descrive l'istanza base del cloud, nel caso di full-virtualization la VM, nel caso di container il singolo container LXD e nel caso MaaS la singola macchina fisica. In figura 6.19 è presentato un esempio di allocazione di più macchine in modo innestato, in particolare una creata con tecnologia LXD all'interno di un'altra creata con tecnologia VM o fisica.

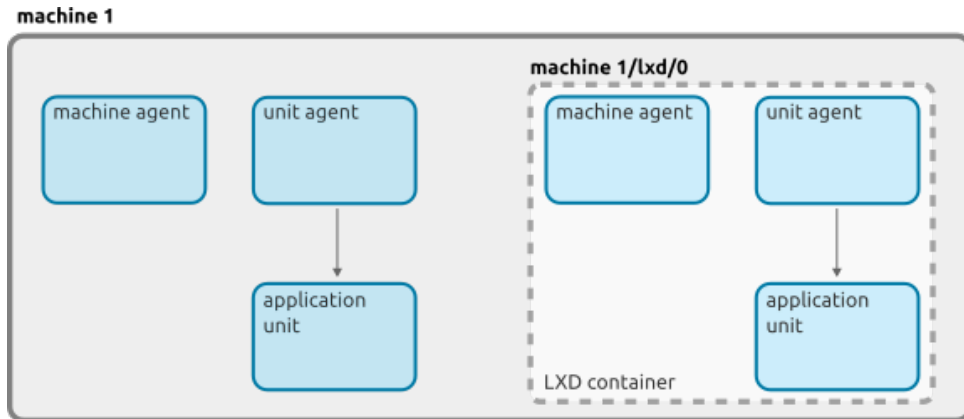


Figura 6.19: Esempio di *Juju machine* (fonte: [juju](#)).

Unit e application

Una *unit* o *application unit* è la singola unità software istanziata (attraverso il charm). È possibile avere applicazioni complesse formate da più unità, che possono risiedere su *machine* diverse.

Relation

La *relation* o relazione è una connessione tra le varie unità applicative, che consente alle stesse di poter comunicare attraverso delle interfacce definite. Tutti i dettagli su come viene impostata questa relazione, le interfacce e cosa può supportare un'applicazione sono dettagli contenuti all'interno del charm. Nell'esempio riportato in 6.20, si evince come differenti applicazioni e le rispettive unità possano esporre delle interfacce e collegarsi in modo automatico con delle altre. La definizione di una relazione all'interno di un *charm* non indica necessariamente che la stessa sia applicata, rende solo possibile poterlo fare. La relazione per essere applicata deve essere inizializzata dal juju controller, o attraverso il Juju client oppure attraverso un bundle.

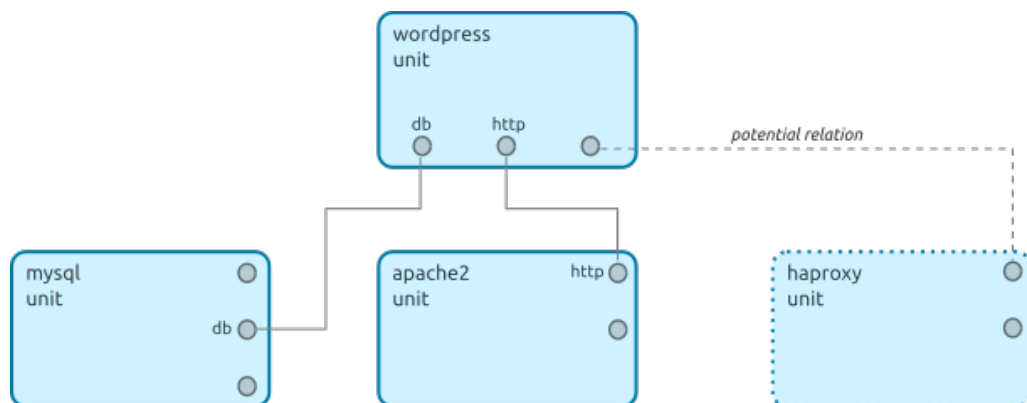


Figura 6.20: Esempio di relazione tra unità applicative (fonte: [juju](#)).

Client

Il Juju client è una *command line interface (CLI)* che viene utilizzata per gestire Juju, in particolar modo per gestire il controller. Da quest'ultima è possibile monitorare le macchine e le unità, avviare delle operazioni sulle stesse o instaurare relazioni.

Agent

L'*agent* è un software che viene eseguito su ogni machine. Vi sono un *machine agent* che lavora a livello della machine e un *unit agent* che lavora, invece, a livello della singola unit. Quest'ultimo è il responsabile dell'installazione del charm sull'unità e quindi di riflesso del deploy del singolo microservizio.

6.3.2 Charm

I charm e i bundle rappresentano gli strumenti di modellazione di un servizio nell'ecosistema Juju. In particolar modo il charm, contenendo tutte le informazioni relative all'istanziamento, configurazione e connessione, risulta essere il cuore pulsante di tutto ciò che Juju rappresenta. Andiamo in questa sezione a vedere come modellare un servizio attraverso i charm. Per ora non utilizzeremo gli strumenti grafici messi a disposizione da Juju, rimandiamo i dettagli di questi ultimi all'appendice [B](#).

Struttura di un charm

Il charm, come già detto, non è nient'altro che un pacchetto di script, composto da varie tipologie di file a seconda della funzione. Un charm di base è composto dalle seguenti componenti logiche:

- **Metadata:** sono dei dati che descrivono il charm e quali sono le relazioni che può accettare.
- **Configuration data:** definiscono come un servizio in esecuzione sul charm possa essere configurato dall'utente.
- **Hooks:** sono delle operazioni che possono essere invocate da Juju all'occorrenza di determinati eventi.
- **Action:** sono delle operazioni che l'utente può invocare attraverso Juju quando desidera.
- **Dati di supporto:** tutto un insieme di dati di supporto che occorrono per funzionalità aggiuntive a quelle elencate.

Un charm viene sviluppato tipicamente in Python, così da consentire una maggiore elasticità e modularità del codice, ma può contenere anche degli script di Bash.

In figura [6.21](#), viene mostrata la struttura effettiva di un charm. Come si evince dalla stessa, è costituito da una serie di descrittori (i file yml), una serie di cartelle con dentro degli eseguibili in Python o Bash (actions, hooks, lib, reactive, tests) e dei file di supporto (ad esempio il logo icon.svg).

Nello specifico il file principale di un charm è il metadata.yaml, quest'ultimo specifica le informazioni identificative dello stesso, le interfacce che espone e le relazioni che è disposto ad accogliere. Un esempio pratico è riportato in figura [6.22](#), dove è mostrato il metadata.yaml del charm di Docker. Quest'ultimo prevede all'interno vari campi identificativi, tra cui alcuni immediati come il nome, il sommario, lo sviluppatore, alcuni tag che sono utili all'identificazione dello stesso, e altri meno ovvi come subordinate, provides e requires. Subordinate, specifica se il servizio in oggetto può essere o meno istanziato nella stessa unit di un altro servizio. Questo è utile perché tipicamente ogni servizio viene istanziato in una unit diversa, che può essere vista come un contenitore isolato che non prevede l'interazione diretta tra i vari servizi, se non attraverso le interfacce esterne. Alcuni

```
/charm
├── actions
├── actions.yaml
├── config.yaml
├── hooks
├── icon.svg
├── layer.yaml
├── lib
├── metadata.yaml
├── reactive
└── tests
```

Figura 6.21: Struttura di un charm.

servizi, come quelli di logging, monitoring o backup, potrebbero avere l'esigenza di accedere al servizio target durante l'esecuzione e quindi poter sfruttare questa caratteristica. Provides e requires, servono per dichiarare le relazioni possibili tra i vari charm. Nello specifico provides, indica quali sono le relazioni e le interfacce dei servizi che mette a disposizione il charm, nel caso specifico di Docker sono due: docker-containers ed events, rispettivamente con le interfacce containers e docker-socket. Requires indica i servizi che accetta per usufruirne, nel nostro caso le relazioni sono network e logging, rispettivamente sulle interfacce overlay-network e docker-socket. Nel caso di charm subordinate, bisogna specificare almeno una relazione di tipo requires. Le interfacce, appartenenti ad entrambe le categorie di relazioni presentate, devono poi essere implementate, ad esempio è possibile esporre un servizio su una determinata porta ed implementarlo attraverso degli script ad hoc.

Per quanto riguarda le *action* è possibile specificarne una descrizione nel file actions.yaml, dopodiché si possono integrare gli script del caso nella cartella actions. Nella cartella hooks, vengono inseriti tutti gli script in risposta ad eventi scatenati da Juju e vi sono anche quelli per l'installazione e l'aggiornamento del charm. Nel file config.yaml vengono inseriti tutta una serie di parametri che vengono utilizzati solitamente dagli script di installazione e aggiornamento, la loro definizione è del tutto arbitraria e possono essere utilizzati anche da altri script definiti dall'utente. Nella cartella lib, sono definite alcune librerie di supporto agli script, mentre nella cartella tests vi sono degli script per il test della soluzione, che hanno l'obiettivo di valutare il corretto funzionamento del charm prima del deploy sul cloud.

Layered Charm

Un concetto importante nello sviluppo di un charm è quello di poter utilizzare una struttura organizzata su diversi livelli. Tipicamente tutta una serie di funzionalità di base possono essere riutilizzate in charm diversi, o ancora alcune parti del codice, per essere più modulari, possono essere incapsulate in un livello diverso. Tutto questo dà vita al concetto di *Layers* in Juju. Di questi ne esistono di tre tipologie con ruoli diversi:

- *Base/runtime*: questa tipologia di livello astrae il concetto di riutilizzo, infatti, molti charm utilizzano le stesse funzionalità di base. Per questa ragione sembra opportuno riutilizzare i livelli base di altri charm, onde evitare di riscrivere codice inutilmente. Degli esempi possono essere dati dal supporto per la gestione delle dipendenze di Python, dai decoratori per gli hook affinché il charm possa interagire con Juju e ancora dai decoratori logici per il codice bash e python. Un altro esempio interessante è quello dato da *layer-apache-php*, questo fornisce direttamente all'interno del charm Apache2 e l'interprete PHP mod-php.
- *Interface*: i livelli legati all'interfaccia sono responsabili della comunicazione attraverso le relazioni di due applicazioni. Nello specifico un livello di questo genere, incapsula un singolo protocollo di interfaccia e viene sviluppato e mantenuto tipicamente da chi scrive il primo charm che espone quella interfaccia. Ovviamente come abbiamo visto le relazioni sono delle

```
1 name: docker
2 summary: Deploys Docker Engine, a lightweight runtime and packaging tool.
3 maintainer: Charles Butler <charles.butler@ubuntu.com>
4 description: |
5   Deploys Docker Engine, a portable, lightweight runtime and packaging tool.
6 tags:
7   - ops
8   - application_development
9 subordinate: false
10 provides:
11   docker-containers:
12     interface: containers
13   events:
14     interface: docker-socket
15 requires:
16   network:
17     interface: overlay-network
18 # logging:
19 # interface: docker-socket
```

Figura 6.22: Esempio di un metadata.yaml.

coppie biunivoche, per questo il protocollo dev’essere implementato in ognuno dei charm della coppia.

- *Charm layers*: questo livello è quello che rappresenta il charm vero e proprio. Viene costruito sulla base degli altri due livelli ed è quello che incapsula la logica dell’applicativo. I principi di progettazione di questo livello sono quelli di mantenere saldo il focus sulle funzionalità che deve offrire ed in capsulare nei livelli precedenti tutte le funzionalità generiche.

Per includere e dichiarare la presenza di livelli si utilizza il file `layer.yaml`, presente nella struttura in figura 6.21.

Reactive Charm

Un’altra caratteristica interessante dei charm di Juju è la possibilità di utilizzo del paradigma di *reactive programming*. Tale paradigma permette di definire degli eventi all’interno del ciclo di vita del charm, che nell’ecosistema Juju sono dei flag chiamati *state*, i quali possono assumere valore true o false. È possibile poi creare degli *handler* per gestire uno o più di questi eventi.

Un esempio di gestione di eventi, mediante questo paradigma, è riportato in figura 6.23. Questo handler mostra una “reazione” a un duplice evento in un caso specifico: quando si è in attesa dei dettagli della connessione da parte di un database MySQL. Quando lo stesso è connesso e non ancora disponibile, viene settato a false l’evento che rappresenta l’inizializzazione del server Apache e viene aggiornato lo stato del charm, con le indicazioni di attesa. Il costrutto per l’utilizzo del paradigma *reactive*, come si evince dal codice, è dato dalla clausola “@when” che prende come argomenti il tipo di state o evento di interesse.

6.3.3 Architettura

Adesso andiamo a presentare l’architettura complessiva di Juju, che viene rappresentata in figura 6.24. In tale figura vi sono i componenti che costituiscono lo strumento Juju (in rosso) e tutto ciò che risiede sul cloud (delimitato dalla linea trattaeggiata). Iniziando con i moduli di Juju possiamo ritrovare:

```

1 @when('database.connected')
2 @when_not('database.available')
3 def waiting_mysql(mysql):
4     remove_state('apache.start')
5     status_set('waiting', 'Waiting_for_MySQL')

```

Figura 6.23: Esempio di reactive programming.

- **Juju CLI:** è la command line interface che permette di accedere al Juju Controller e di utilizzarne le funzionalità. Vedremo nell'appendice [B](#), quali sono i comandi a disposizione.
- **Juju Deployer:** è un modulo che mette a disposizione un linguaggio ad alto livello per la creazione di servizi complessi, costituiti da charm e relazioni, e di effettuarne il deploy attraverso lo state server.
- **Juju GUI:** è un'interfaccia grafica che permette l'interazione con il controller (state-server) e dispone di numerosi strumenti, tra questi alcuni per la modellazione dei servizi, altri per la gestione interattiva dei servizi istanziati.
- **State server:** è il modulo di controllo che viene caricato su ogni cloud per poter interagire con lo stesso. Si occupa tra le altre cose di gestire gli agent che vengono caricati su ogni istanza virtuale o machine e che permettono di gestirla.
- **Charm Store:** è lo store online di Juju, che permette l'accesso a charm e bundle preconfigurati e che possono essere istanziati in modo immediato sul proprio cloud.

Scopriamo ora cosa avviene sul cloud. Innanzitutto abbiamo la possibilità di creare un controller (state-server) su una qualsiasi delle piattaforme riportate in figura [6.24](#), da OpenStack fino alle macchine fisiche MAAS. Il controller, una volta installato, utilizza la piattaforma per allocare le istanze, che corrispondono banalmente alle machine. In figura è riportato l'esempio di macchine virtuali e all'interno delle stesse è rappresentato l'agent, il modulo che abbiamo detto essere il demone che consente la comunicazione con il juju controller e l'unico mezzo per eseguire operazioni sulla VM stessa. Le VM rappresentate in figura sono dotate di un sistema operativo ubuntu di base ed ospitano al loro interno delle unità sulle quali sono stati caricati dei charm ed istanziati i relativi servizi. Nella prima VM a partire da sinistra, ad esempio, è stato caricato il servizio Docker ed installato quindi Docker engine e tutti gli strumenti necessari per l'esecuzione di container Docker. Ricollegandoci alle definizioni date in precedenza, è utile ricordare che ogni unità dispone a sua volta di un agent che permette l'interazione e la configurazione dei servizi. Immaginiamo quindi di voler eseguire un'azione sul servizio Docker, possiamo attraverso la CLI eseguire il comando adatto e lo state server propagherà lo stesso all'agent relativo a Docker, facendo occorrere l'evento necessario per effettuare l'azione desiderata.

Questa è una panoramica di come lavora Juju e di come sia possibile gestire servizi attraverso tale strumento.

6.3.4 Proxy charm

Arrivati a questo punto, una volta capito come è strutturato Juju e cosa sono i charm, cerchiamo di capire come questi meccanismi si possono integrare in Open Source MANO.

Da quello che abbiamo visto finora, la scelta più naturale sembrerebbe quella di utilizzare Juju, sia come meccanismo per istanziare, sia come strumento per configurare le VNF. In particolare il Service Orchestrator potrebbe utilizzare il Juju controller per istanziare i servizi, che nel nostro caso sono le VNF, e intanto poter garantire la continua configurazione e manutenzione delle VNF attraverso i charm, il tutto come abbiamo visto negli esempi di funzionamento di Juju.

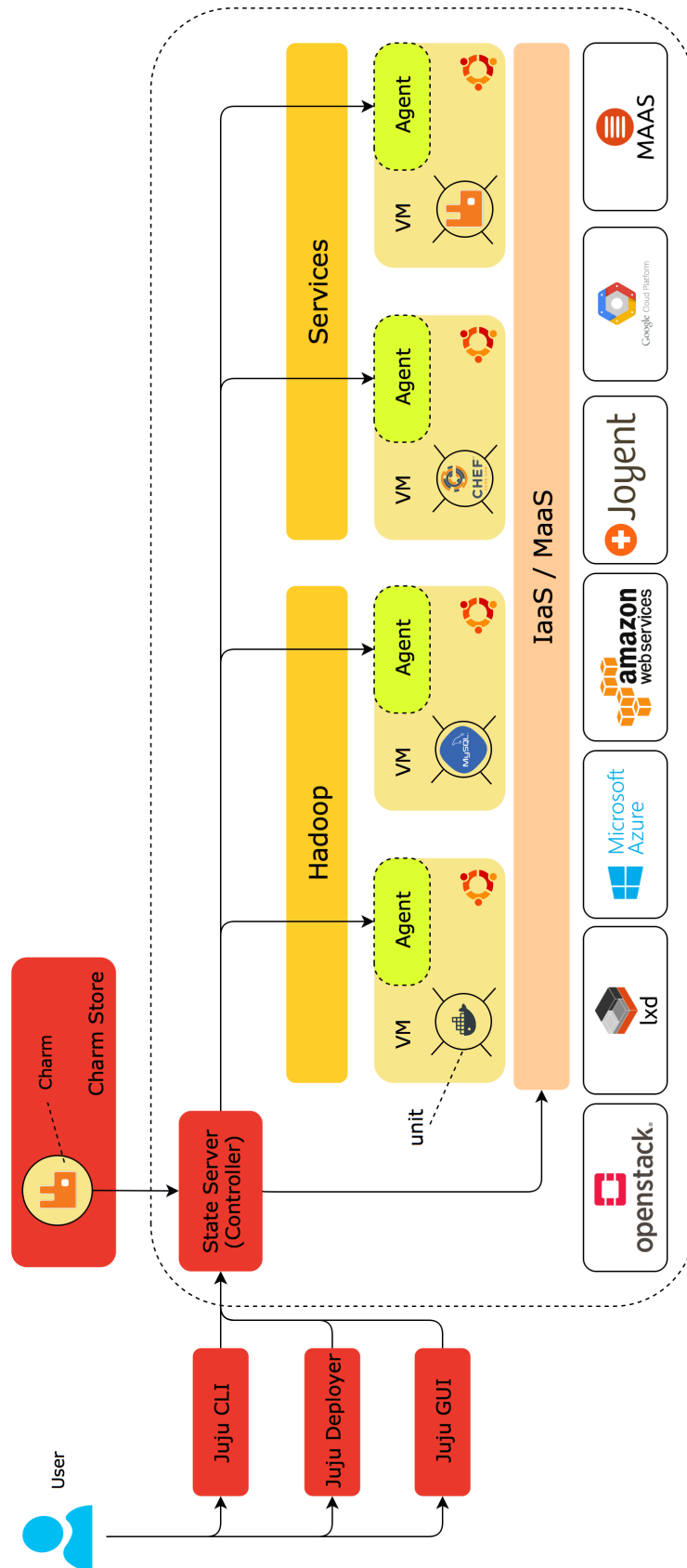


Figura 6.24: Architettura di Juju.

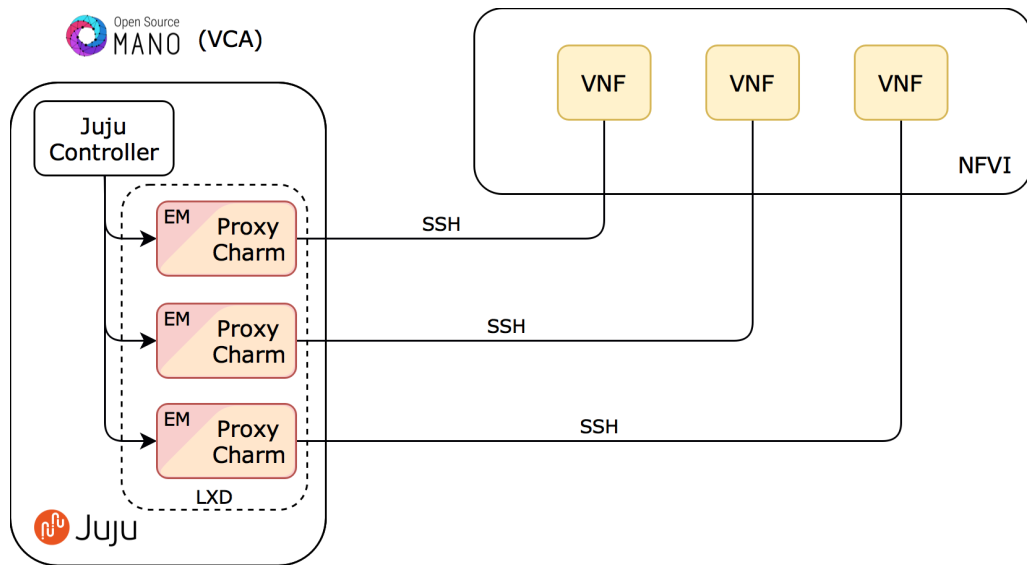


Figura 6.25: Juju in OSM.

In realtà OSM non integra Juju in questo modo, infatti, prevede che nel suo modulo VCA, modulo dedicato alla configurazione delle VNF, venga installato tutto l'ambiente Juju e che attraverso questo venga fatto, in locale o su un altro cloud, il deploy di servizi specializzati nella configurazione delle VNF. Questo porta all'esempio in figura 6.25, dove si vede che all'interno del modulo VCA, in questo caso con tecnologia LXD, sono stati istanziati tre servizi specializzati chiamati *proxy charm* che si occupano della configurazione delle VNF sull'infrastruttura. Non tutte le VNF potrebbero richiedere questa tipologia di configurazione, per questo motivo, soltanto quelle che definiranno nel proprio descrittore l'indicazione di Juju come VNF Manager, avranno a disposizione il proxy charm come meccanismo di configurazione. Adesso è chiaro perché nel workflow di OSM, era richiesto per alcune VNF un tempo di istanziazione dei *proxy charm*, per poi procedere all'effettiva configurazione. Tale tempo è dovuto all'istanziazione della macchina virtuale o del container per poter ospitare il charm, installarlo e configurarlo, solo allora si potranno effettuare le operazioni richieste.

In realtà il charm che istanzia il servizio *proxy charm* non è molto complesso, è composto da due livelli fondamentali: uno che si occupa della gestione della comunicazione attraverso un canale sicuro SSH e l'altro che contiene la logica applicativa che risponde alle esigenze di configurazione della VNF. Il primo è costituito da una libreria di funzioni che consentono le operazioni base su SSH e diversi meccanismi di autenticazione. Il secondo modulo dipende strettamente dal fornitore della VNF ed è tipicamente basato sul meccanismo delle action. Come abbiamo visto, attraverso l'utilizzo di queste ultime, è possibile definire delle operazioni personalizzate su un determinato servizio. Nel nostro caso il servizio è il proxy charm, che una volta istanziato, può inviare tramite il canale sicuro delle direttive alla VNF. Il processo in sintesi è questo: attraverso il Juju controller invio una action al servizio proxy charm, quest'ultimo eseguirà il codice corrispondente ed effettuerà delle operazioni tramite SSH sulla VNF relativa. In sostanza, sto andando a separare e decentralizzare l'elaborazione delle configurazioni su container virtuali isolati ed indipendenti dal VNF Manager. L'unica elaborazione che effettua il controller, è quella di dirigere la action da eseguire al proxy charm corrispondente alla VNF sulla quale voglio eseguire l'operazione. Alcune operazioni possono essere effettuate dal proxy charm in autonomia rispetto al VNF Manager, infatti, all'interno dello stesso servizio potrebbero esserci degli eventi che non dipendono dalle action ma da meccanismi relativi alla logica di gestione della VNF, prevista dal fornitore della stessa. In ultima analisi, potremmo vedere il proxy charm come l'Element Management dell'architettura NFV, in quanto si occupa della gestione delle operazioni FCAPS della VNF. Resta da vedere come si integra questo meccanismo nel Service Orchestrator.

Open source MANO prevede che nel package della VNF vi siano due elementi: un proxy charm e una serie di *service primitive* definite attraverso il descrittore, che corrispondono alle azioni da

eseguire su una VNF. Il Service Orchestrator quando istanzia un servizio, prima istanzia i proxy charm e successivamente effettua un mapping tra le action contenute nel charm e le service primitive dichiarate nel descrittore (figura 6.26). Queste primitive vengono visualizzate e richiamate nell'interfaccia grafica dell'orchestratore, così da poterle utilizzare in qualsiasi momento del ciclo di vita della VNF, in alternativa è possibile richiamarle da script attraverso il modulo VCA.

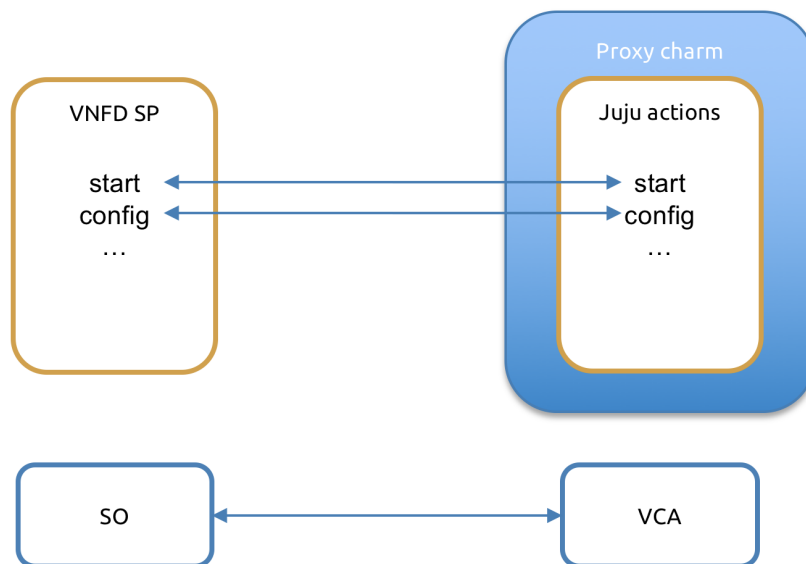


Figura 6.26: Mapping *action* su *service primitive*.

Il mapping illustrato è utile perché consente di definire in modo completo, attraverso il package del descrittore della VNF (VNFD), il comportamento della funzione di rete virtuale. Questo è possibile perché inseriamo nello stesso il proxy charm personalizzato, la cloud-init della VNF e le azioni che voglio eseguire sulla funzione (queste ultime nel descrittore yaml). Vedremo in pratica come è possibile sfruttare questi meccanismi in modo intelligente per creare le VNF nel capitolo 8.

Ora è utile fare delle riflessioni finali su questo meccanismo, per capire quali sono i vantaggi e svantaggi. In particolare, l'introduzione di un nuovo elemento che va istanziato nell'architettura NFV, sembra inizialmente un inutile overhead per il sistema. Quando istanzio una VNF, infatti, devo non solo preoccuparmi di allocare risorse per la stessa, ma anche trovarne per il proxy charm. In alternativa, poi, ho tanti altri strumenti per la configurazione delle VNF, ad esempio *Platform-as-a-Service* (PaaS) come Chef, Puppets e Ansible. Questi strumenti risultano anche molto più efficienti e di semplice utilizzo rispetto a Juju nella configurazione delle VNF e non prevedono come nel caso di Ansible nemmeno un agent sulla VNF per la configurazione. La domanda a questo punto è sul perché utilizzare Juju a fronte di strumenti così ben congegnati e caratterizzati da ottime prestazioni.

La risposta a questa domanda risiede nella scelta progettuale di OSM di dare molte alternative di personalizzazione alle VNF. In particolare Juju non è esattamente un servizio PaaS come gli altri citati, ma è uno strumento che lavora ad un livello superiore. Infatti, ho la possibilità di istanziare qualsiasi tipo di servizio anche gli stessi Chef, Puppets e Ansible per configurare una VNF. Un esempio potrebbe essere utilizzare un proxy charm che utilizza un layer Ansible (già sviluppato e utilizzato anche per il charm Docker), il quale mette a disposizione tutte le funzionalità per eseguire un playbook (un descrittore delle operazioni da effettuare su un'istanza) e sfruttarlo per configurare una VNF. Lo stesso discorso si può fare con ogni servizio PaaS che abbiamo citato. In più il focus di Juju, rispetto agli altri strumenti, è verso le relazioni con gli altri servizi proprio perché lavora ad un livello diverso. Questo mi dà la possibilità di caricare più servizi contemporaneamente e relazionarli tra loro per gestire una VNF o un insieme delle stesse. Insomma vi sono diversi scenari di personalizzazione, ma non è tutto a favore di Juju.

Un elemento sempre da considerare, è il vantaggio di allocare tutte queste risorse esclusivamente quando le VNF hanno bisogno di una configurazione sufficientemente complessa e necessitano di monitoraggio per tutto il ciclo di vita, altrimenti non ha senso sprecare tutte queste risorse. Fortunatamente Open Source MANO mette a disposizione anche il meccanismo di cloud-init per la configurazione, che può essere un'alternativa per evitare di sovraccaricare il sistema quando le VNF non necessitano di una configurazione troppo complessa. Infine, non è scontato l'utilizzo di Juju, si potrebbe anche pensare, come scelta progettuale, di utilizzare alternative come una delle piattaforme citate e semplicemente integrare la stessa nel modulo VCA, scelta non consigliata da OSM, ma una ragionevole alternativa per alcuni casi d'uso.

6.4 Light Virtualization

Presenteremo ora più in dettaglio il concetto di “virtualizzazione leggera”, questo ci permetterà di utilizzare le tecnologie legate ad essa, per alcuni aspetti della nostra implementazione.

La *light virtualization* rappresenta un insieme di tecnologie che consentono di eseguire in modo isolato su una stessa macchina più applicativi o istanze di sistemi operativi. La principale differenza rispetto alla full-virtualization sta nel fatto che tali applicativi o istanze, in gergo chiamate *container*, condividono il kernel del sistema operativo della macchina sulla quale vengono eseguiti. I container presentano numerosi vantaggi rispetto alle tradizionali VM:

- Non hanno bisogno dei tempi necessari al boot del kernel del sistema operativo.
- Non avendo un kernel, non hanno necessità di occupare memoria e CPU dedicata allo stesso, diminuendo considerevolmente l'overhead della virtualizzazione.
- L'I/O è più efficiente, poiché non c'è bisogno di attraversare il kernel del sistema operativo guest.
- Durante la creazione le macchine virtuali specificano una quantità fissa di RAM e le CPU dedicate, nel caso container invece le risorse vengono allocate con una granularità più fine.
- L'aggiornamento di un solo kernel per macchina fisica risulta più conveniente e semplice.

Tutti questi vantaggi hanno fatto sì che negli ultimi anni i container siano diventati una considerevole opportunità per molte compagnie, in termini di sviluppo delle loro risorse IT. Anche in ambito NFV stanno diventando sempre più un'interessante prospettiva per l'isolamento delle funzioni di rete virtuali.

Bisogna però considerare anche alcuni aspetti negativi nell'utilizzo dei container rispetto alle VM, ragion per cui alcuni sono ancora scettici riguardo alla loro adozione. Il primo è relativo alla capacità di virtualizzare un sistema operativo diverso rispetto a quello del kernel a disposizione; nel caso di container, infatti, c'è la possibilità di virtualizzare solo un sistema operativo che condivide il kernel con quello della macchina fisica sottostante (non posso eseguire windows su linux). Il secondo svantaggio è relativo al minore livello di isolamento rispetto alle VM, dovuto proprio all'assenza del kernel del sistema operativo per ogni container. Quest'ultimo problema sta diventando sempre meno rilevante negli ultimi anni, poiché i kernel supportano sempre meglio i container. L'ultimo svantaggio è relativo al fatto che le tecniche di migrazione dei container sono allo stato embrionale rispetto a quelle delle VM. In ogni caso, con le dovute configurazioni, è possibile utilizzare questa tecnologia in modo sicuro ed usufruire delle potenzialità intrinseche che mette a disposizione.

Rimandiamo alla letteratura [7] discussioni su come utilizzare in modo sicuro i container e ci concentriamo su un argomento più interessante dalla nostra prospettiva: presentare due tecnologie di virtualizzazione leggera che utilizzeremo più avanti, ovvero LXD e Docker.

Una prima differenza da fare tra queste è che LXD appartiene ai cosiddetti *machine container* e Docker invece ai *process container*, vedremo a breve le differenze.

6.4.1 LXD

LXD è un *container hypervisor*, che consente la gestione di linux container (LXC), attraverso l'utilizzo delle librerie *liblxc*. I container così creati, come già accennato, fanno parte della categoria machine container, questo perché si comportano come se fossero una VM a tutti gli effetti:

- hanno una versione ridotta del sistema operativo;
- ospitano varie applicazioni nello stesso container;
- hanno un loro processo `init()`;
- sono tipicamente stateful e utilizzano tecnologie come copy-on-write.

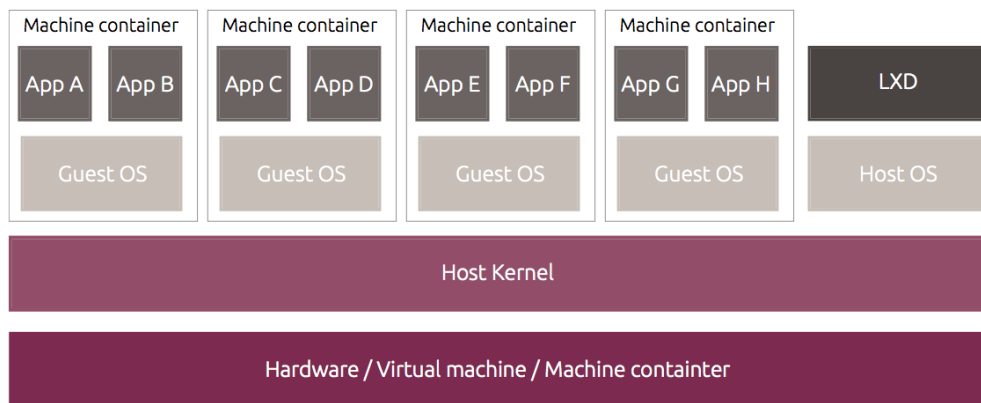


Figura 6.27: Esempio di *machine container* (fonte: [40]).

In figura 6.27, viene riportato uno scenario tipico di LXD e di virtualizzazione con machine container.

Per quanto riguarda le tecnologie alla base di LXC (figura 6.28), dobbiamo partire dal presupposto che a livello kernel non abbiamo il concetto di container. Per ottenere l'isolamento e la sicurezza richiesta dai container vengono utilizzate varie tecnologie messe a disposizione dal kernel linux: cgroup, namespace, Apparmor, SELinux, Seccomp, Chroot e le Kernel capability.

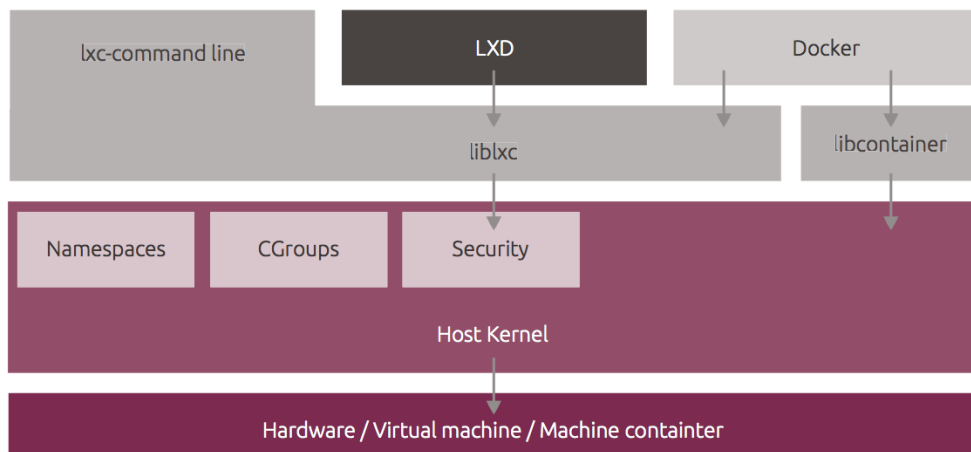


Figura 6.28: Container in Linux (fonte: [40]).

I cgroup (o control group) forniscono dei meccanismi per limitare l'utilizzo di vari tipi di risorse (CPU, memoria, I/O e così via). I namespace danno la possibilità di restringere la visibilità di alcune risorse unicamente ad un gruppo di processi. Un esempio è il namespace relativo alla rete che permette a gruppi differenti di processi di vedere in modo differente lo stack di rete. Un ragionamento analogo può essere effettuato per l'*IPC* (*Inter process communication*).

Questo primo livello di isolamento non risulta essere sufficiente, infatti, bisogna impedire ad esempio che un processo eseguito come root su di un container abbia accesso completo al sistema (anche l'hardware). Per problematiche del genere sono stati introdotti degli strumenti di sicurezza per i container attraverso Apparmor, SELinux, le kernel capability e seccomp. Due meccanismi offerti dalle stesse sono il DAC (Discretionary Access Control) e il MAC (Mandatory Access Control). Il primo fa da mediatore per l'accesso alle risorse attraverso delle policy definite dall'utente, così i vari container non possono interferire tra di loro. Il secondo si assicura che, né il codice del container stesso, né quello eseguito al suo interno abbiano dei privilegi più elevati di quelli richiesti dal processo che li esegue, questo minimizza i danni di processi compromessi.

LXD, inoltre, offre numerose funzionalità per la gestione dei container LXC ad esempio:

- una RESTful API e una CLI per la creazione e gestione dei container;
- servizio remoto per la gestione delle immagini software;
- una gestione flessibile dello storage e della rete;
- supporta tecnologie ARM, POWER, x86 e Z;
- possibilità di effettuare degli snapshot.

Vedremo in pratica queste caratteristiche e come utilizzare i container LXD nella documentazione contenuta nell'appendice C.

6.4.2 Docker

Docker è invece un esempio di process container, infatti, come è illustrato in figura 6.29, è progettato per contenere una singola applicazione (ovvero è single purpose).

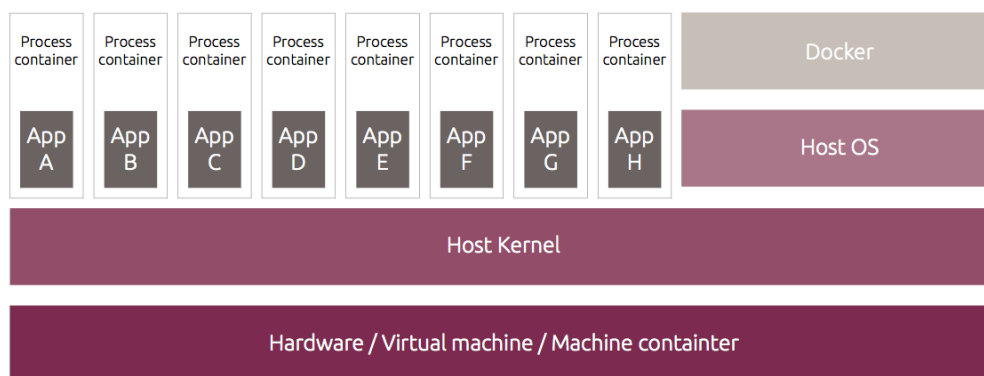


Figura 6.29: Esempio di *process container* (fonte: [40]).

I container Docker sono stati realizzati con l'idea di essere di dimensioni molto ridotte. Devono contenere pochi MB di dati, i binari dell'applicazione e un sottoinsieme di librerie che servono per la sua esecuzione. Questo è più un principio che un'applicazione pratica, infatti, è possibile anche con tecnologia Docker avere più applicativi nello stesso container.

La differenza sta nella gestione delle immagini, attraverso più layer diversi. come si vede in figura 6.30, infatti, Docker ha la possibilità di comporre le immagini attraverso diversi livelli, questi

livelli sono in sola lettura e vengono condivisi tra vari container. Al di sopra di tali livelli viene posto un'altro livello che rappresenta lo spazio in scrittura e lettura riservato al container. Questo permette di non avere un'immagine del sistema operativo per ogni container (ovviamente solo lo spazio utente, in quanto il kernel è sempre condiviso in ogni tecnologia di container). Di base lo spazio messo a disposizione per il container in scrittura è effimero, ovvero, viene distrutto alla distruzione del container stesso.

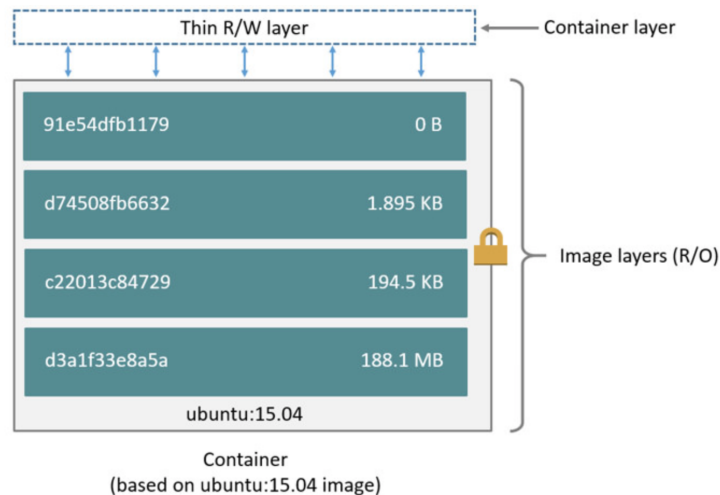


Figura 6.30: Immagini utilizzate da Docker (fonte: [docs docker](#)).

Per la gestione e la composizione dei livelli si utilizza il Dockerfile, quest'ultimo è un file di testo che contiene tutte le istruzioni necessarie per tale obiettivo.

A partire dalla figura 6.31, andiamo adesso a definire l'architettura di Docker attraverso la presentazione della terminologia e di alcune definizioni:

- **Docker Platform:** la piattaforma è l'insieme di strumenti che consentono la gestione del ciclo di vita dei container.
- **Docker Engine:** il Docker Engine è un'applicazione client-server che ha tre componenti fondamentali, ovvero un server che è un processo demone (**dockerd**), una REST API per accedere alle funzionalità del server e una CLI che consente di utilizzare le funzionalità di Docker da parte di un client.
- **Docker daemon:** il Docker daemon è il server che si occupa di gestire le chiamate API di Docker e gli oggetti Docker come le immagini, i container, le reti e i volumi. Un Docker Daemon può comunicare con altri processi demone per gestire i servizi Docker.
- **Docker Client:** è una CLI che consente l'interazione con l'ambiente Docker, attraverso il comando **docker**.
- **Docker registries:** sono i repository che memorizzano le immagini Docker. Gli esempi più diffusi sono Docker Hub e Docker Cloud, i registri pubblici che mette a disposizione Docker.
- **Docker objects:** gli oggetti Docker sono le immagini, i container e i servizi. Le immagini come abbiamo in parte visto sono dei template (read-only), che permettono di creare un container. Il container per l'appunto è un'istanza di un'immagine che viene eseguita e può essere creato, inizializzato, spostato e terminato, attraverso le Docker API o la CLI. Infine i servizi permettono di scalare i container su più Docker Daemon, questo consente di scalare orizzontalmente i container su più macchine. A questo punto i container possono lavorare insieme con diversi ruoli di *managers* e *workers*. Ogni attore in questo caso è un Docker Daemon e comunica con gli altri attraverso le Docker API.

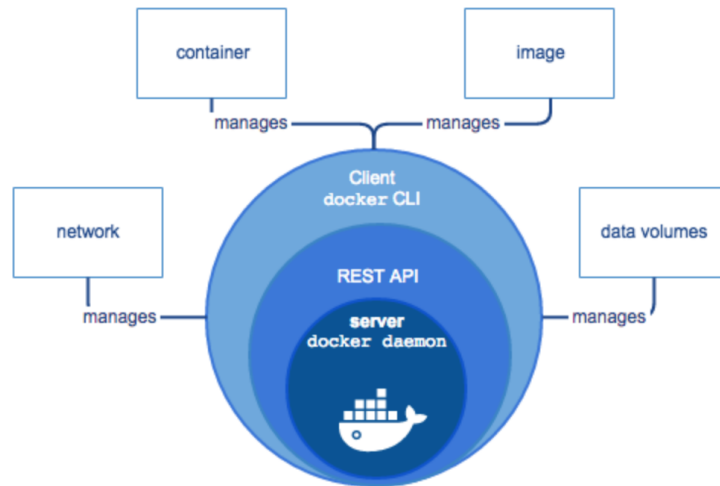


Figura 6.31: Docker Engine (fonte: [docs docker](#)).

Esistono altri strumenti a disposizione per creare applicazioni composte da più Docker, un esempio è Docker Compose. Ancora è possibile utilizzare degli strumenti di Orchestrazione come Docker Swarm e Kubernetes per gestire cluster di container.

Per quanto riguarda le tecnologie alla base di Docker, abbiamo le stesse che abbiamo presentato per LXD, ovvero cgroup, namespace e meccanismi di sicurezza del kernel linux. L'unica differenza è che, anche avendo compatibilità con le librerie liblxc, solitamente Docker utilizza le librerie libcontainer per interfacciarsi con il kernel.

Non siamo scesi troppo nei dettagli sulle tecnologie di virtualizzazione leggera, perché esula dagli scopi del lavoro, però è necessario presentare almeno le alternative per capire le motivazioni delle scelte nell'implementazione. Verranno dati dei dettagli ulteriori, di ordine pratico, nell'appendice [B](#).

Capitolo 7

Design della soluzione

7.1 Design di alto livello

7.1.1 Architettura complessiva

Lo scopo principale di questo capitolo è quello di presentare la progettazione del sistema di gestione ed orchestrazione di servizi di sicurezza, obiettivo cardine di questo lavoro di tesi. Abbiamo accennato, nei capitoli 2 e 5, cosa intendiamo per servizio di sicurezza ma adesso ne daremo una nostra precisa definizione. Il *Network Security Service* o servizio di sicurezza di rete è un caso particolare di un Network Service, laddove le VNF sono delle funzioni virtuali di sicurezza o vNSF. Tale servizio di sicurezza può essere completamente definito attraverso un forwarding graph come nel caso dell'NS, quindi può essere composto da una singola Service Function Chain (come in fig. 7.1) o da più SFC, tutto quello definito nei capitoli 2, 3 e 6 resta valido per un Network Security Service.

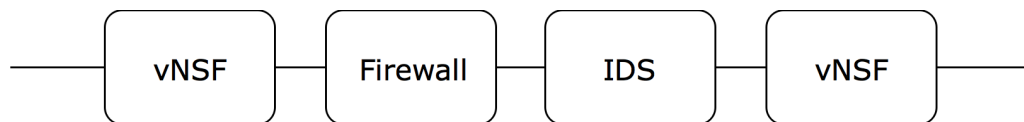


Figura 7.1: Network Security Service.

Una volta definito il servizio di sicurezza di rete, il passo successivo è quello di capire come applicare tutto quello che abbiamo visto nei capitoli precedenti per poter gestire ed orchestrare tali servizi.

Partiamo dall'architettura presentata in figura 7.2, e vediamo che questa si divide (linea verticale tratteggiata) in due sezioni principali: *Management* e *Infrastructure*.

La parte di Management è il cuore dell'architettura e comprende quello che chiamiamo *Security Service Manager*. Questo è l'elemento cardine che si occupa della gestione ed orchestrazione dei servizi di sicurezza. Esso è formato da tre moduli:

- **Open Source MANO:** è il sistema NFV MANO che abbiamo presentato nei capitoli 3 e 6, del quale sfrutteremo le funzionalità di gestione ed orchestrazione delle funzioni virtuali di rete.
- **Policy Services:** è il sistema di gestione delle policy, si occupa della memorizzazione, dei processi di refinement e analysis delle stesse.
- **Security Service Controller:** è il modulo che funge da intermediario tra i due moduli precedenti e lavora ad un livello di astrazione tale da permettere operazioni direttamente sui servizi di sicurezza (es. istanziazione, terminazione, gestione).

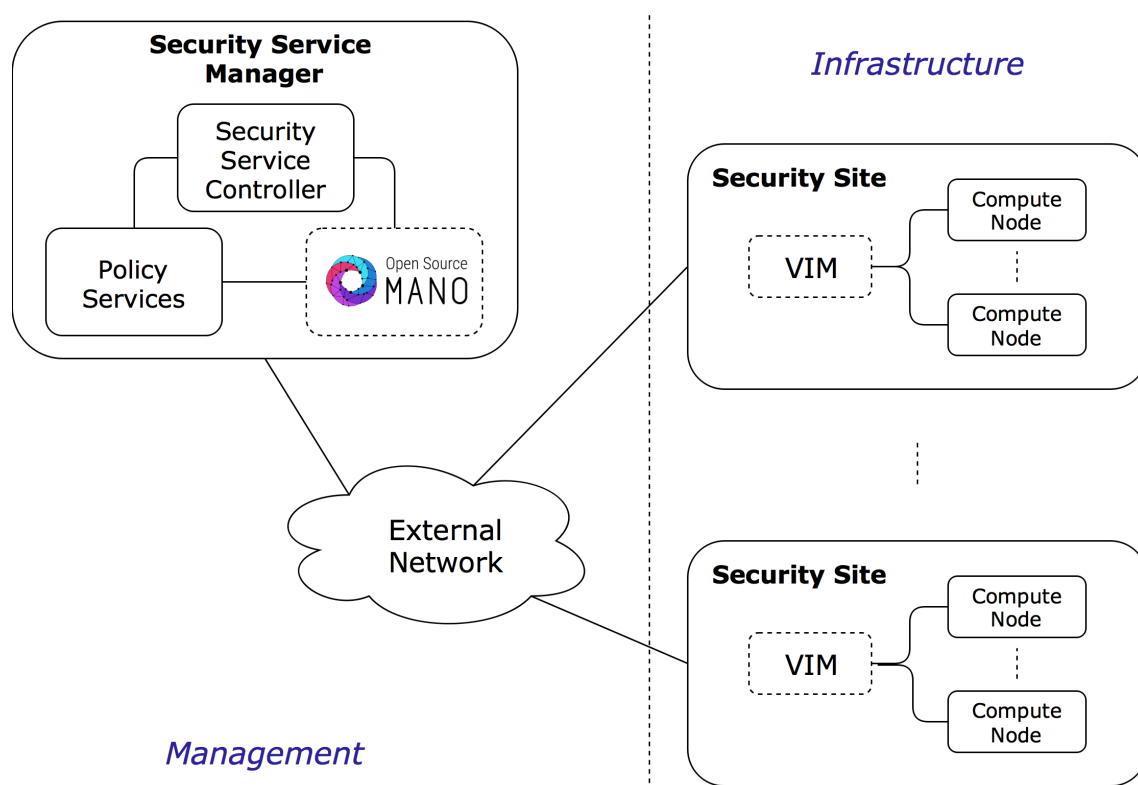


Figura 7.2: Architettura generale soluzione.

L'utilizzo di Open Source MANO non è sufficiente a gestire i servizi di sicurezza per come sono stati concepiti; infatti è necessario un sistema che sia in grado di automatizzare il deploy e la configurazione di tali servizi e di integrare le policy all'interno del framework; tutto questo giustifica la presenza degli altri due moduli.

L'Infrastructure contiene tutta la parte di NFVI, ovvero l'infrastruttura NFV che ospiterà le funzioni virtuali di sicurezza e il VIM, gestore di tale infrastruttura. Come vediamo dalla figura 7.2, sono previsti diversi nodi (Security Site); questo perché OSM è in grado di gestire diversi siti (o datacenter) e di conseguenza distribuire il carico delle vNSF tra questi. Una possibile motivazione nell'utilizzo di questa caratteristica, potrebbe essere la necessità di dover gestire i servizi di sicurezza in diversi punti distribuiti geograficamente, ad esempio i nodi finali di rete prima dei terminali dell'utente. Il concetto di security site è molto vicino a quello di datacenter; infatti esso, a prescindere dal numero di nodi fisici che lo compongono, astrae quella che è la gestione e allocazione di risorse virtuali. Il Security Site, in sostanza, può rappresentare sia un tipico datacenter con un VIM e più compute node (come nel caso di OpenStack), sia un sistema in cui gestione ed allocazione di risorse sono sullo stesso nodo (come nel caso tipico di Docker e dei container).

Il Security Service Manager comunica con i Security Site attraverso una rete esterna, questo proprio per permettere la distribuzione di tali nodi. Qui abbiamo il primo problema, in tale scenario Open Source MANO e i VIM (rappresentati con la linea tratteggiata) si scambiano informazioni di gestione e controllo attraverso una rete tipicamente pubblica. Per sopperire a tale problematica è possibile installare una VPN in modo tale che la comunicazione tra il Security Service Manager e i Security Site sia protetta e crittografata.

7.1.2 Security Service Manager

Il Security Service Manager è, come abbiamo visto, il modulo che ingloba architetturalmente i tre componenti che consentono l'orchestrazione del servizio di sicurezza. Andiamo adesso a vedere le sue funzionalità dal punto di vista logico.

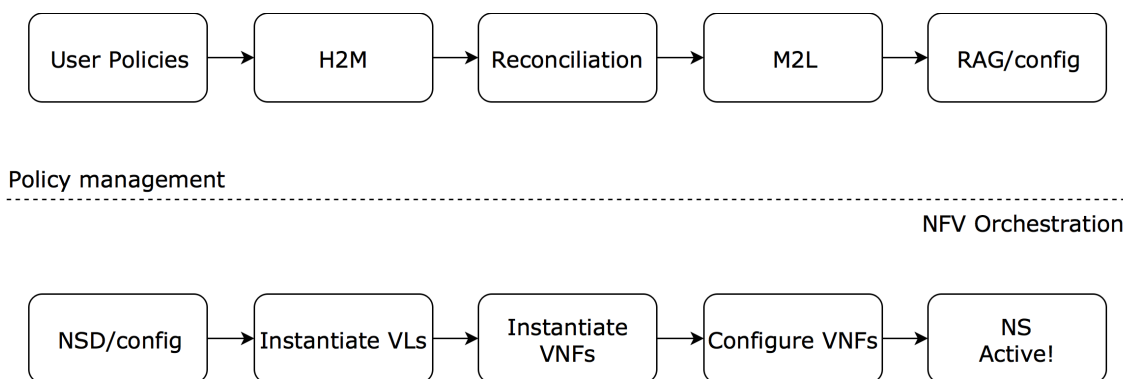


Figura 7.3: Funzionalità logiche del Security Service Manager.

Come si vede in figura 7.3, possiamo dividere le funzionalità di questo modulo in: *Policy Management* e *NFV Orchestration*.

Policy Management

Il policy management è la parte di gestione relativa in particolare al processo di trasformazione delle policy. Le policy vengono, infatti, definite dall'utente attraverso il linguaggio HSPL (capitolo 4) e successivamente trasformate in MSPL dal modulo H2M. Successivamente la reconciliation aggrega le policy provenienti da diversi attori, ed infine le stesse vengono tradotte in policy di basso livello. Queste corrispondono a configurazioni di basso livello che possono essere applicate direttamente alle vNSF. Un altro risultato del processo di reconciliation è il RAG (capitolo 4): questo è un grafo che contiene delle informazioni relative alle vNSF di cui l'utente ha esigenza nel suo servizio di sicurezza. Tutte queste operazioni sono svolte dal modulo Policy Services, attraverso il coordinamento del Security Service Controller.

Al termine di tutto il processo di trasformazione delle policy, quindi, ci si aspetta di avere due dati fondamentali: un Reconciled Application Graph e delle configurazioni di basso livello.

NFV Orchestration

L'NFV Orchestration comprende tutte le operazioni relative all'istanziamento e configurazione delle vNSF e del servizio di rete. Tutto questo processo riportato in figura 7.3 segue la stessa logica riportata nel capitolo 6, riguardo al ciclo di vita di un Network Service. Il Reconciled Application Graph restituito dal processo di Policy Management e le relative configurazioni di basso livello costituiscono gli input di questo ulteriore processo di trasformazione. Più in dettaglio, questo processo si occupa in prima istanza della trasformazione del RAG in NSD (secondo la sintassi OSM), dopodiché avvia tutto il processo di istanziazione e configurazione tipico di OSM. I moduli coinvolti sono Open Source MANO e il Security Service Controller.

7.1.3 Security Site

Il security Site costituisce l'elemento di base dell'Infrastructure della nostra architettura. La chiave, per poter rendere quanto più generale e modulare possibile la soluzione, è quella di rendere più indipendente possibile il Security Site dalla soluzione tecnologica adottata. Nella nostra implementazione, ad esempio, abbiamo utilizzato OpenStack sia come VIM che come infrastruttura e quindi compute node. A tutti gli effetti, però, un qualsiasi altro VIM può essere integrato in questa soluzione, a patto che lo stesso sia compatibile con il framework di OSM. Inoltre gli unici altri vincoli che legano la parte di Infrastructure a quella di Management sono la necessità di fornire un collegamento, attraverso una rete sicura, tra Open Source mano e i vari Security Site. Una volta soddisfatti tali requisiti, risultano del tutto indifferenti la collocazione geografica dei nodi dell'Infrastructure e i VIM utilizzati. Vedremo più in dettaglio questa parte nella sezione 7.3.

7.2 Policy Services

Il modulo *Policy Services*, appartenente al Security Service Manager, incapsula tutte le funzionalità e i servizi per la gestione delle policy. Nel caso del progetto SECURED tutte le funzionalità erano orchestrate da due manager (capitolo 4): l’offline workflow manager e l’online workflow manager. Nel nostro caso il framework di gestione delle policy non necessita di alcun modulo del genere, in quanto la gestione ad alto livello viene effettuata dal Security Service Controller.

Il nostro reale interesse è quello di esporre attraverso questo modulo tutti i servizi applicabili alle policy.

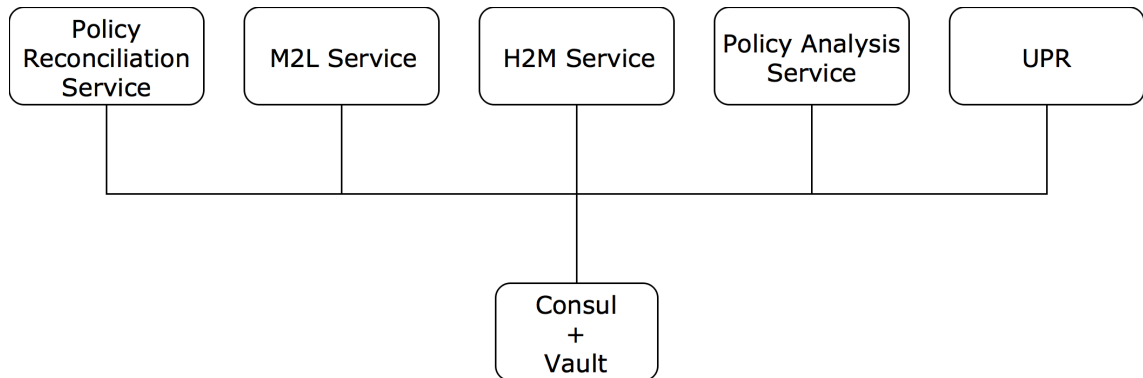


Figura 7.4: Architettura Policy Services.

Come si vede in figura 7.4, il modulo è composto da diversi servizi:

- **Policy Reconciliation Service:** questo servizio si occupa dell’aggregazione delle policy provenienti dai diversi attori.
- **H2M Service:** questo è il modulo che si occupa della traduzione da HSPL in MSPL.
- **M2L Service:** questo modulo si occupa della traduzione da MSPL in configurazioni di basso livello.
- **Policy Analysis Service:** questo modulo ha la possibilità di analizzare le policy definite in MSPL e riscontrare eventuali anomalie.
- **User Policy Repository (UPR):** questo è il servizio di gestione della base di dati dove vengono memorizzate le policy.

Ognuno di questi moduli espone una REST API che consente l’accesso al servizio corrispondente e vi è inoltre un modulo aggiuntivo, indicato come “Consul + Vault”, che funge da *service discovery* e *identity service* e che si occupa di fornire un end-point comune a tutti questi servizi e di gestire l’autenticazione ed autorizzazione al fine di permettere un controllo degli accessi ai medesimi.

7.2.1 Service discovery e authentication

Nella progettazione è stato previsto un modulo che permettesse a tutti i servizi di esporre un end-point comune e consentisse di gestire l’autenticazione dei processi e degli utenti che hanno necessità di accedere ai servizi del Policy Services. L’idea di base è quella di replicare qualcosa molto simile al Keystone di Openstack, che abbiamo utilizzato per il medesimo scopo nella gestione degli accessi ai servizi dell’Infrastructure (per ogni singolo security-site).

Keystone però è *special purpose*, infatti, si integra alla perfezione con i servizi di OpenStack, consentendone anche di estenderne alcuni, ma sempre con l’obiettivo di controllare gli accessi nell’ecosistema della piattaforma. A questo punto si è pensato all’utilizzo di un software della

Hashicorp, Consul che da solo, o in combinazione con *Vault*, fornisce un servizio di service discovery e di controllo degli accessi.

Consul presenta un'architettura distribuita, si adatta a sistemi di tipo cluster e prevede la presenza di un agent su ogni nodo (a prescindere dalla sua natura di VM o macchina fisica). Un agent può essere sia client che server, nel primo caso si occupa unicamente di esporre dei servizi specifici che sono a disposizione su un particolare nodo, nel secondo caso, in aggiunta alle funzioni di client, si occupa della gestione e coordinazione degli altri nodi, in particolare, nel caso in cui si abbiano diversi datacenter a disposizione, provvede allo scambio di messaggi con i server degli altri datacenter. Noi non ci troviamo in quel caso e proponiamo l'utilizzo di Consul su un unico host, con un unico server centrale e con dei client su ogni container LXD, ognuno dei quali espone un servizio diverso del modulo Policy Services. Il fatto che ci sia la possibilità di scalarlo orizzontalmente in modo semplice è in ogni caso di interesse per eventuali evoluzioni dell'architettura. Adesso andiamo a vedere nello specifico le funzioni che offre Consul:

- **Service Discovery:** un client può esporre un servizio (ad es. con delle API), registrandolo, e gli altri possono venire a conoscenza dei servizi esposti tramite il server di consul. In questo modo è possibile semplificare l'utilizzo dei servizi, anche in caso di modifica degli stessi, senza necessità di dover procedere a riconfigurazioni.
- **Health Checking:** è previsto un meccanismo di health check, che consente il monitoraggio dei servizi e la eventuale ridirezione di servizi su un altro servizio analogo (utile nel caso in cui i servizi debbano essere scalati e ci sia bisogno di fare load balancing in modo intelligente).

Consul offre anche funzionalità relative all'utilizzo su datacenter multipli, ma non sono al momento di nostro interesse. Una cosa che può tornarci utile è il controllo degli accessi, effettuato in modo opzionale da Consul attraverso una ACL (Access Control List). Il sistema è basato su token e può essere gestito sia attraverso le API di Consul sia attraverso l'integrazione di Vault. Quest'ultimo è ancora uno strumento della hashicorp che consente la memorizzazione di password, chiavi o più in generali segreti, in modo flessibile e sicuro. Ha dei meccanismi che consentono la generazione automatica di token, diversi meccanismi di autenticazione e permette anche la rotazione in modo automatico delle chiavi ssh, funzione che potrebbe tornarci utile, più avanti per quanto riguarda l'infrastruttura.

7.3 Security Site

La scelta tecnologica per quanto riguarda l'infrastruttura NFVI e il VIM è ricaduta su OpenStack. In particolare OpenStack con tecnologia di virtualizzazione KVM. Il motivo di tale scelta è data dal fatto che OpenStack è ormai uno standard de facto per quanto riguarda le soluzioni datacenter ed è inoltre la tecnologia principalmente consigliata dall'ETSI nello standard (Capitolo 2).

7.3.1 Architettura

L'architettura prevista per il Security Site è quella rappresentata in figura 7.5.

Abbiamo un Controller Node dove sono presenti i servizi minimi necessari ad OpenStack per operare, in accordo a quello visto nel capitolo 6, e diversi compute node dove eseguiremo le nostre vNSF.

Horizon permette ad un amministratore di accedere con una comoda interfaccia grafica alle funzionalità del VIM; questo ovviamente può essere utile per ragioni di manutenzione o di analisi dei log in caso di guasto. OSM si interfaccia direttamente con il Keystone di OpenStack, questo attraverso le API esposte, nel nostro caso versione 3. In particolare utilizza quelle di nova e di neutron per creare le reti (link virtuali) e le istanze delle funzioni di sicurezza.

Dal lato del compute node sono presenti nova compute, che si interfaccia con l'hypervisor KVM, e un neutron agent che permette attraverso Open vSwitch di creare la connettività tra le istanze e verso l'esterno.

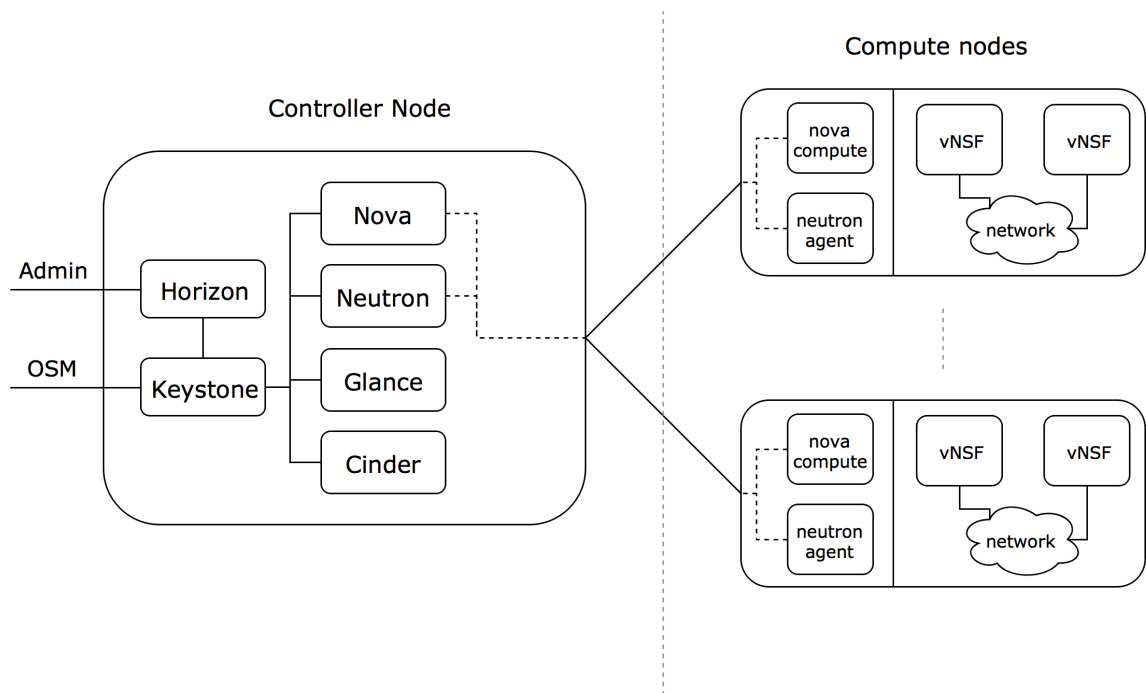


Figura 7.5: Architecture Security Site.

7.3.2 Progettazione della rete

Uno degli aspetti di rilievo della progettazione è la gestione delle connessioni di rete dell'infrastruttura NFVI. Il nodo che si occupa di tale gestione è quello di rete (nel nostro caso inglobato in quello dello controller) e in particolare viene guidato dal servizio neutron di OpenStack. Per quanto riguarda la connettività di livello 3, viene ricreato parte dello stack di rete e i relativi servizi offerti (come descritto nel capitolo 6); mentre a livello tecnologico, su ogni singolo nodo compute, è OVS che si occupa del dataplane e al neutron agent, installato su quel nodo, compete la gestione degli Open vSwitch. Tutto questo viene mascherato agli utenti di OpenStack che tipicamente ragionano in termini di reti, sottoreti e router virtuali.

Ora, a cominciare da un livello di astrazione più alto, forniamo i dettagli delle reti che sono state concepite nella nostra soluzione, fino a poi arrivare ad alcuni dettagli implementativi; questi ultimi necessari per comprendere il perché di alcune operazioni effettuate sul VIM.

Come rappresentato in figura 7.6, le reti a cui facciamo riferimento sono tre:

- management;
- virtual link;
- data-ext.

Il principio di base è quello di separare la rete che si occupa della gestione delle funzioni virtuali da quella dei dati. Per questo motivo, viene creata all'interno del VIM una rete di management che consente di connettersi ad ognuna delle interfacce di gestione delle vNSF. A questo punto, attraverso questa rete, il Resource Orchestrator di OSM e Juju hanno accesso alle vNSF.

Un'altra esigenza è data dalla connettività inter-vNSF, questa permette che vi sia il flusso di dati attraverso le funzioni virtuali e quindi di costruire la Service Function Chain. Questa tipologia di connessione è creata attraverso l'utilizzo dei virtual link. Questi sono, in OpenStack, delle vere e proprie sottoreti con tutti gli strumenti necessari per offrire connessione di livello 2 e 3, inoltre, c'è la possibilità di fornire strumenti come DHCP per ottenere l'assegnazione automatica degli indirizzi IP, senza dunque doverli specificare in modo statico nelle funzioni virtuali. Questi link non hanno

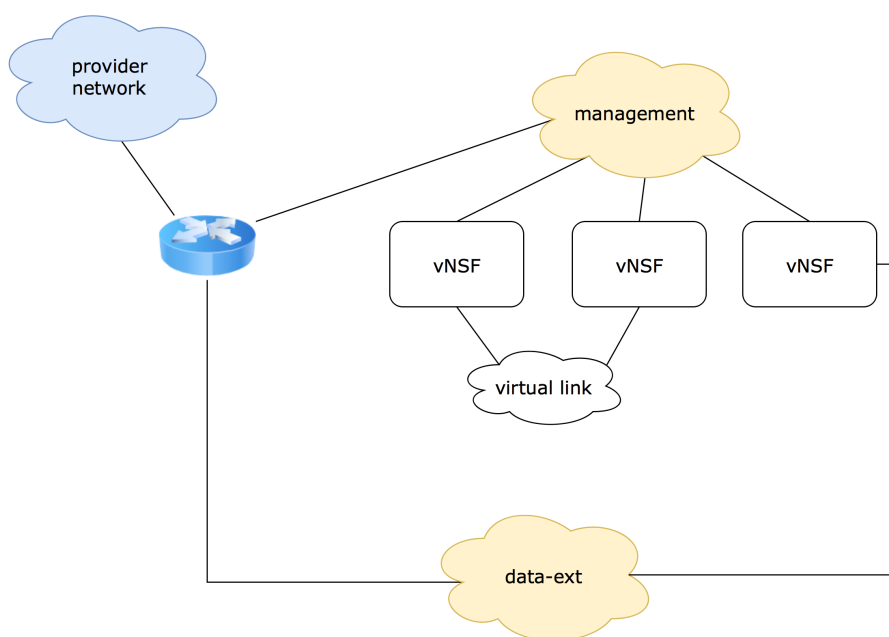


Figura 7.6: Gestione delle reti.

necessità di connessione verso l'esterno, quindi sono creati come delle reti locali vlan. L'unico tassello mancante è quello relativo alla configurazione delle interfacce delle vNSF, a tal proposito più avanti vedremo come sia stato messo a disposizione uno script ad hoc per la configurazione automatica delle stesse.

Ancora vi è una rete chiamata data-ext. Questa rete fornisce connettività esterna alle vNSF, nel caso in cui le stesse abbiano la necessità di inoltrare pacchetti su una rete esterna o debbano definire un end-point per l'utilizzo del servizio di sicurezza.

Un'ultima osservazione è relativa alla necessità, nella nostra architettura, di offrire connettività esterna anche alla rete di management, in quanto i moduli di OSM ed in particolare Juju hanno necessità di raggiungere le vNSF dall'esterno e quindi bisogno dell'accesso alla sottorete di management.

OpenStack crea le reti attraverso Open vSwitch e la modalità con cui offre la connettività esterna è l'utilizzo di un linux bridge, tipicamente chiamato br-ex, che permette di collegare la rete pubblica dell'host con una sottorete virtuale (tipicamente chiamata provider network). Una volta abilitato questo bridge (vedremo come fare nell'appendice D) è possibile raggiungere la sottorete virtuale che abbiamo introdotto. Per collegare la nostra rete di management e quella data-ext alla provider network, quello che si può fare, è creare un router in OpenStack che si occupi dell'inoltro dei pacchetti tra queste; collegando le interfacce dei gateway delle tre sottoreti (management, data-ext e provider network), infatti, si ottiene esattamente il risultato desiderato. Questo è il modo tipicamente utilizzato da OpenStack per assegnare i floating IP alle istanze, in questo caso però per ragioni di comodità abbiamo preferito definire una nostra sottorete di management e poi collegarla verso l'esterno.

7.4 Progettazione della vNSF

7.4.1 Architettura della vNSF

L'architettura della vNSF, ricalca per molti aspetti quella della VNF presentata nei precedenti capitoli (cap. 2).

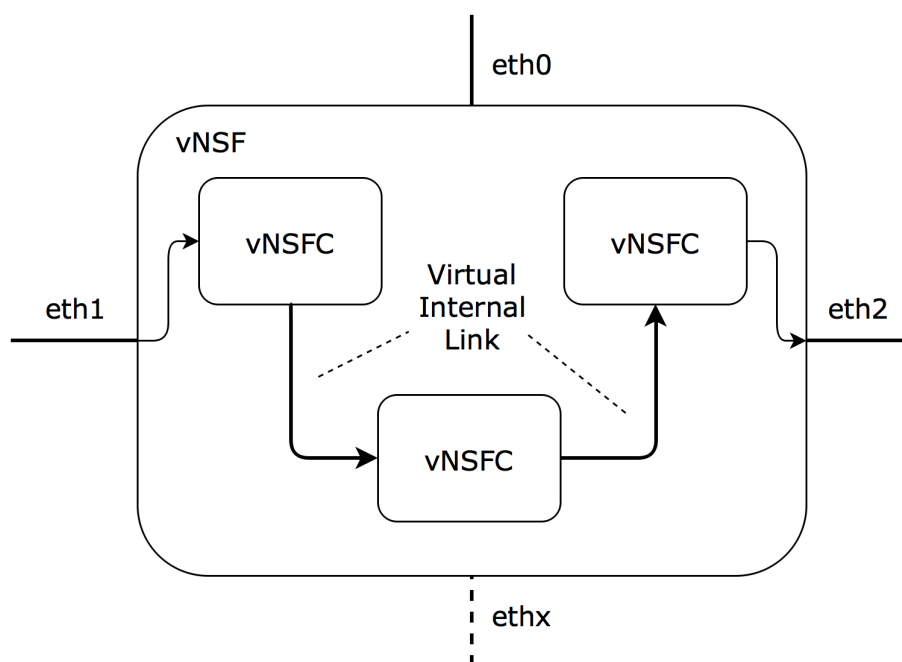


Figura 7.7: Modulo di una vNSF.

Come rappresentato in figura 7.7, la funzione virtuale di sicurezza è formata da uno o più moduli interni (vNSFC) che in Open Source MANO abbiamo chiamato VDU. Questi moduli hanno una loro immagine software e hanno la possibilità di essere configurati in modo separato ed indipendente l'uno dall'altro. Questi moduli, inoltre, hanno la possibilità di comunicare tra di loro definendo dei link virtuali interni (capitolo 6), che poi potranno essere mappati sulle reti fornite dal VIM o implementati con altri metodi di comunicazione. Tutta la progettazione, in questo caso, dipende dallo sviluppatore della vNSF, che deve scegliere in quanti moduli suddividere la stessa e come essi devono comunicare ed essere configurati.

Per quanto riguarda le connessioni esterne, invece, teoricamente è possibile definire un numero di interfacce arbitrarie in modo da poter implementare qualsiasi grafo, anche complesso. Nel caso della nostra soluzione, per ragioni di semplicità, trattiamo il caso semplice di singola catena, quindi definiamo le tre interfacce riportate in figura 7.7: **eth0**, **eth1** ed **eth2**. Rispettivamente le interfacce rappresentano quella di management (**eth0**), quella per i pacchetti in ingresso alla vNSF (**eth1**) e l'ultima quella per i pacchetti in uscita dalla vNSF (**eth2**). È stata prevista la possibilità, nel caso di una futura estensione, di aggiungere in modo arbitrario altre interfacce per la creazione di grafi complessi, questo si declina in una modifica non consistente della logica del controller, vedremo questo più avanti.

La cosa interessante, dopo tutte queste considerazioni, è che risulta possibile, attraverso l'utilizzo di OSM ed in particolare tramite il descrittore della funzione virtuale di sicurezza, definire in modo completo ognuna delle caratteristiche trattate (relative al software e alle interfacce). Questo come abbiamo anticipato nel capitolo 6 e come vedremo quando nel prossimo capitolo 8 mostreremo come implementare una vNSF.

7.4.2 Configurazione della vNSF

I meccanismi di configurazione sono stati descritti nel capitolo 6 e ne utilizzeremo nella nostra soluzione di due tipi:

- Day-0 configuration;
- Day-1 configuration.

La Day-0 configuration sarà eseguita attraverso l'utilizzo di cloud-init. In questo caso verranno fornite alla vNSF tutte le informazioni riguardo ai meccanismi di accesso (injection di chiavi, username/password), gli script di configurazione iniziale, eventuali pacchetti da installare o file da inserire al suo interno. Ovviamente avremo bisogno di questo tipo di configurazione quando utilizzeremo delle immagini software di tipo cloud image. Come abbiamo visto nel capitolo 6, infatti, tale meccanismo si applica nel caso in cui andiamo a personalizzare le immagini attraverso gli user-data, altrimenti non avrebbe senso fornire tale tipologia di configurazione.

A questo punto per ragioni di chiarezza dobbiamo fare una distinzione tra due approcci diversi in fase di progettazione della vNSF. È possibile, come nel caso del progetto SECURED, concepire delle immagini software che contengono (nell'immagine raw qcow2) tutto il necessario, inteso come insieme di software e dati, utili al funzionamento della vNSF. Questo significa che l'unica operazione da compiere in questo caso, in fase di istanziazione, è quella di effettuare il boot dell'immagine e attraverso un meccanismo di configurazione, diverso dalla Day-0 configuration, andare ad iniettare le sole configurazioni relative alle preferenze di sicurezza dell'utente (es. filtraggio di pacchetti provenienti da un particolare indirizzo sorgente). Nel caso, invece, di cloud image i software, gli script e anche i meccanismi di accesso devono essere iniettati nella vNSF prima di poter procedere alla vera e propria configurazione basata sulle preferenze dell'utente.

La Day-1 Configuration, viene effettuata attraverso l'utilizzo dello strumento Juju. Una volta istanziata la vNSF, sia nel caso dell'uso di cloud image o meno, posso procedere all'applicazione delle configurazioni personalizzate dell'utente. In questo caso Juju offre la possibilità di definire alcune action che vengano eseguite appena il charm viene istanziato e attraverso queste compiere due operazioni: trasferire il file di configurazione ed eseguire questi script sulla vNSF.

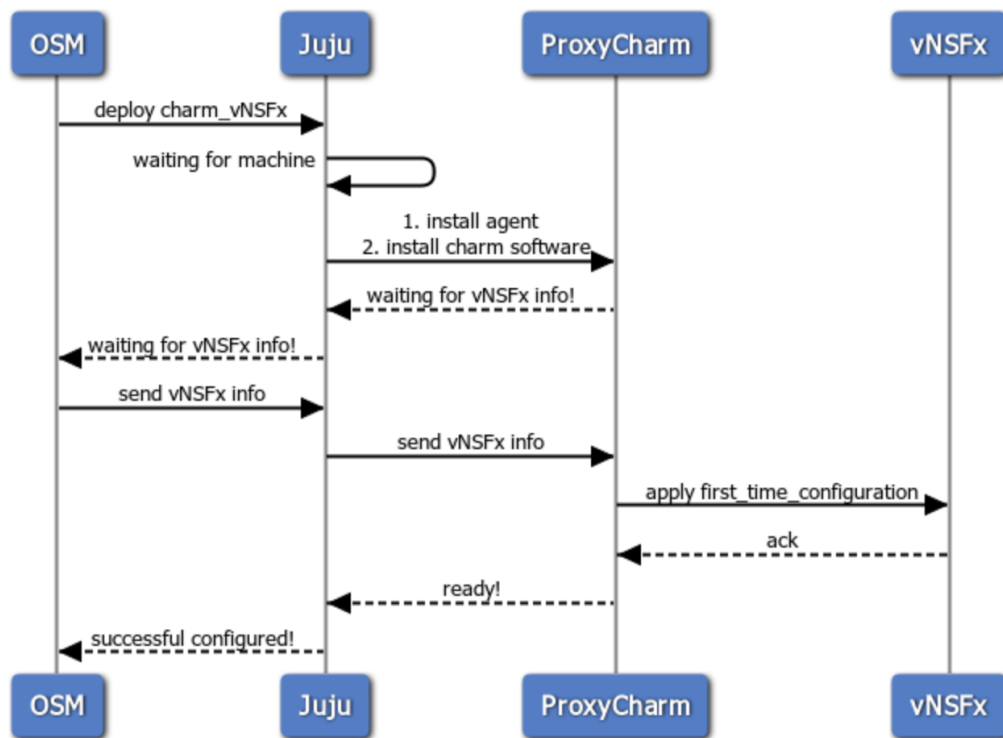


Figura 7.8: Workflow Day-1 configuration.

La figura 7.8, mostra quali sono i principali passaggi effettuati per iniettare le configurazioni dell'utente nella singola vNSF attraverso Juju e il relativo proxy charm.

Ripercorrendo brevemente i passaggi e facendo riferimento anche ad alcune descrizioni del capitolo 6, la prima operazione è sicuramente la definizione degli script cloud-init e del proxy charm all'interno del VNFD package. Questa operazione verrà descritta in pratica nel capitolo 8 di implementazione. Una volta definiti questi due oggetti e dichiarati nel descrittore yaml della vNSF, è possibile procedere all'onboarding della stessa nel repository di OSM. A questo punto

quando verrà istanziata la vNSF, prima sarà eseguita la day-0 configuration se presente, attraverso cloud-init, successivamente prenderanno corpo tutte le fasi descritte in figura 7.8. Una volta che tutto sarà stato portato a termine la vNSF sarà attiva e debitamente configurata.

La cosa interessante di questo ciclo di configurazioni, è la totale autonomia con cui le stesse vengono effettuate rispetto all'istanziamento del servizio di sicurezza. Più in dettaglio tutte le fasi che abbiamo descritto sono indipendenti da come il servizio di sicurezza viene concepito. Si ha la possibilità di definire le configurazioni per ogni singola vNSF, in modo comodo e veloce attraverso il package e i descrittori di questa; una volta caricato tale package in OSM le stesse non vengono più modificate. Il servizio di rete, a questo punto, ha bisogno di conoscere unicamente l'ID delle vNSF per poterle richiamare. Durante la fase di configurazione, poi, tutti gli attori in gioco dipendono unicamente dalle funzioni virtuali e dai package precaricati, quindi non c'è necessità di aggiungere informazioni nel descrittore del servizio di sicurezza; successivamente il processo di configurazione avviene parallelamente per tutte le vNSF. Al termine della configurazione verrà inviato un flag ad OSM che indicherà la corretta configurazione di tutte le vNSF in gioco.

7.5 Progettazione del Network Security Service

Abbiamo definito l'NSS in sezione 7.1.1 e con la figura 7.1. Per poter istanziare automaticamente, attraverso OSM, un servizio di sicurezza dobbiamo progettare un template di descrittore che abbia le seguenti informazioni:

- **Un insieme di constituent vNSF:** questi sono semplicemente i riferimenti alle vNSF (o meglio ai descrittori) che sono state create e delle quali è stato effettuato l'on-boarding.
- **Un insieme di Virtual Link:** sono un insieme di Virtual link e dei punti di connessione (interfacce esterne) che collegano le vNSF tra di loro. Ognuno dei Virtual Link contiene anche dei riferimenti per poter essere mappato correttamente sulle reti esistenti dal VIM, in alternativa è possibile definire delle reti ex novo attraverso gli IP profile.
- **Un insieme di IP profile:** questi rappresentano la definizione di nuove reti, che verranno istanziate dal VIM prima di poterci collegare le funzioni di sicurezza. È utile per la creazione automatica dei link tra le vNSF, senza doverli definire in anticipo sull'infrastruttura e senza utilizzare direttamente le API di OpenStack (neutron).

Una volta che sono stati definiti tutti questi elementi attraverso l'NSD di Open Source MANO (che per noi è Network Security Service Descriptor (NSSD) in figura 7.9), possiamo istanziare il nostro servizio di sicurezza senza difficoltà attraverso le fasi on-boarding e deploy.

7.6 Security Service Controller

Il Security Service Controller lavora ad un livello di astrazione più alto rispetto al Policy Services e ad OSM. Questo modulo, in accordo con la figura 7.10, fornisce i seguenti servizi:

- NSS Deployer;
- OSM CLI/UI;
- Policy GUI.

Il primo servizio ovvero il *Network Security Service Deployer (NSS Deployer)* si occupa di tutte le fasi necessarie al deploy del servizio di sicurezza. Con questo intendiamo che, a partire da uno specifico utente, è in grado di valutare quali sono le preferenze di sicurezza attraverso le policy e coordina OSM al fine di ottenere il deploy desiderato.

Tutto questo processo avviene in diverse fasi:

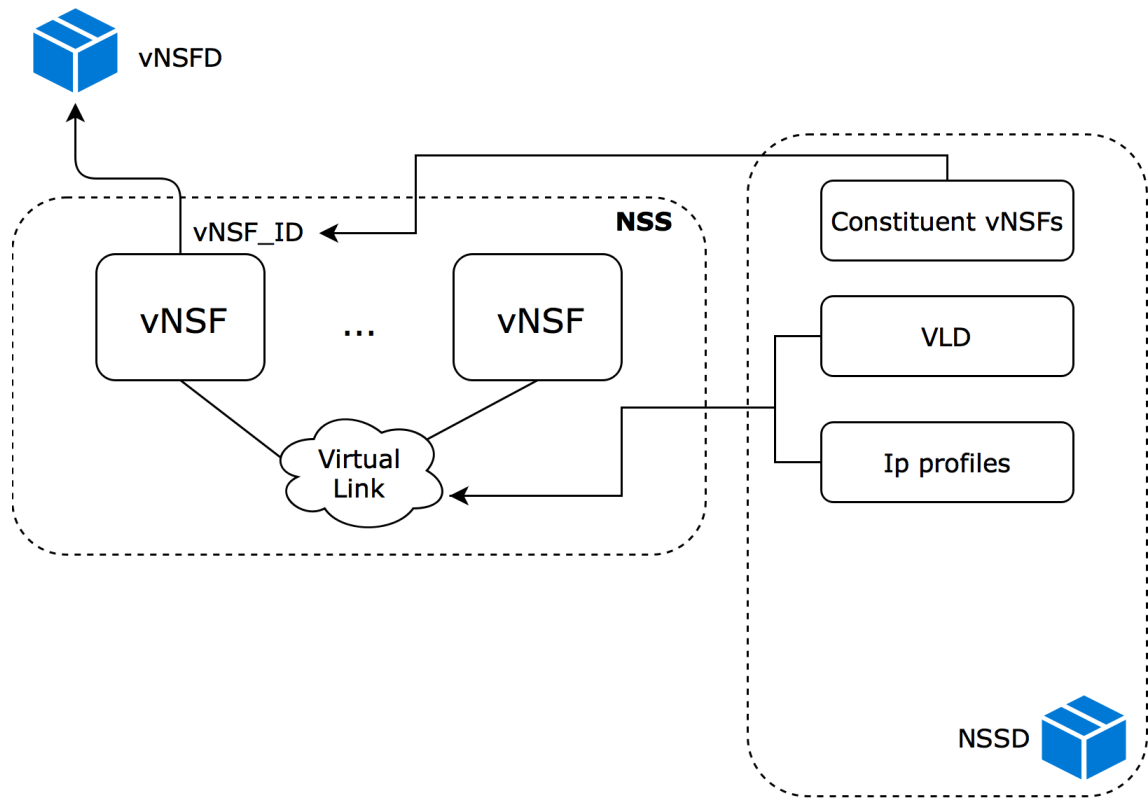


Figura 7.9: Progettazione di un template di NSSD.

- **RAG/configurations acquisition:** in questa fase viene contattato l'UPR attraverso i servizi esposti dal modulo Policy Services e vengono acquisiti il RAG dell'utente e le configurazioni di basso livello associate allo stesso.
- **NSD generation:** in questa fase il modulo, a partire dal RAG precedentemente acquisito, effettua la traduzione dello stesso in un NSD compatibile con OSM.
- **NSD packaging:** in questa fase, a partire dall'NSD generato in precedenza, viene costruito un package adatto per il successivo on-boarding nel catalogo di OSM.
- **NSD launching:** in questa fase viene effettuato l'on-boarding e il lancio del servizio di sicurezza attraverso la CLI di OSM.

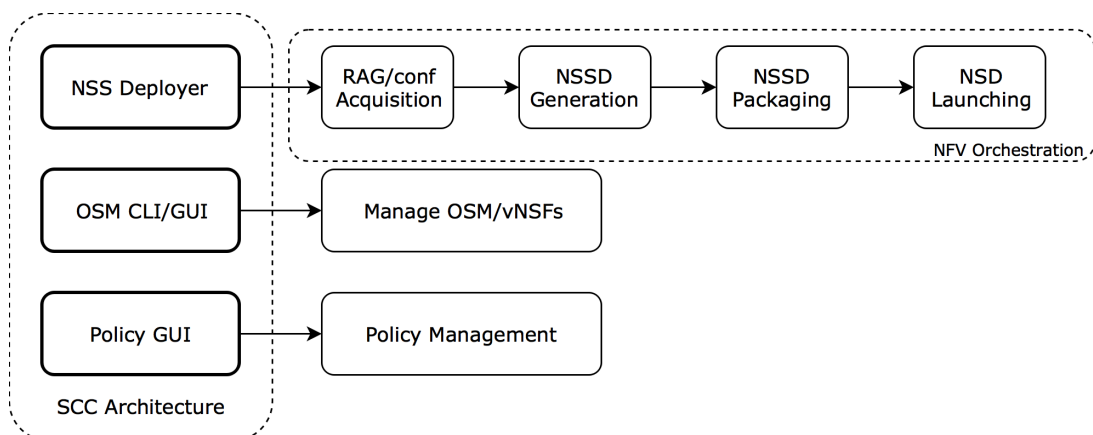


Figura 7.10: Moduli del Security Service Controller.

Il secondo servizio *OSM CLI/UI* mette a disposizione un client per accedere ad OSM tramite linea di comando ed ha la possibilità di connettersi all'interfaccia grafica di OSM. Questo è utile perché, come vedremo, da questa è possibile controllare in modo completo il servizio di sicurezza. Non è necessario quindi aggiungere ulteriori elementi grafici o CLI per effettuare operazioni sulle vNSF. Da questa interfaccia è possibile accedere alla console di ogni singola vNSF attraverso noVNC, applicare le azioni messe a disposizione da Juju, gestire la terminazione del servizio, scarlarlo e ancora osservare metriche eventualmente predefinite per monitorarlo.

Risulta quindi di interesse solo inglobare in un unico modulo tutti gli strumenti di accesso ad OSM per effettuare tali operazioni.

Il terzo ed ultimo servizio *Policy GUI* è una Graphic User Interface messa a disposizione dal Security Service Manager per gestire e richiamare alcuni dei servizi nel Policy Services e ancora per effettuare le operazioni di caricamento delle policy da parte dell'utente. L'utente, infatti, attraverso un modulo di autenticazione ha la possibilità di accedere a tale interfaccia grafica ed effettuare le configurazioni del caso. Quando modifica le policy di alto livello può lanciare un servizio di traduzione che, attraverso i servizi H2M, Reconciliation e M2L, generi il RAG e le configurazioni discusse, e che memorizzi questi dati nuovamente all'interno dell'UPR. Ancora è possibile effettuare dalla stessa la policy analysis, questo una volta eseguita la precedente traduzione. Dall'MSPL, infatti, risulta possibile analizzare eventuali anomalie e generare un report descrivente le stesse.

Come si vede la gestione delle policy e quella del servizio di sicurezza restano temporalmente separate. Si parte dal presupposto che l'utente abbia già caricato e modificato le policy nel momento in cui il Service Provider si appresta ad erogare il servizio di sicurezza.

7.7 Security Service Manager: controllo degli accessi

Un aspetto da affrontare, per ragioni di completezza, è quello del controllo degli accessi. Abbiamo visto nelle sezioni precedenti che sono stati proposti vari meccanismi per la risoluzione di tale necessità, soprattutto per quanto riguarda il modulo di Policy Services. Tale modulo, infatti, può essere dotato di un meccanismo di service discovery e controllo degli accessi tramite lo strumento Consul, utilizzato in combinazione con Vault.

Una questione da risolvere è quella di valutare come effettuare il controllo degli accessi per i moduli OSM e per il Security Service Controller. Per quanto riguarda quest'ultimo, dato che tutti i servizi sono disponibili sulla stessa sottorete a sua volta definita attraverso un linux bridge (sez. 8.2), si può pensare ad una registrazione del servizio SSC al consul-server e garantirne così la disponibilità e l'accesso controllato. In modo equivalente a come accade nel caso del Policy Services poi è possibile utilizzare Vault come backend.

Il modulo OSM, invece, a partire dalla release THREE mette a disposizione un servizio di autenticazione e autorizzazione basato su OpenID Connect e OAuth 2.0. Tra i vari servizi esposti dal container SO, vi è sulla porta 8009 un identity provider, che fornisce gli strumenti per autenticare e autorizzare gli utenti, tale servizio è legato al processo *rwmain* all'interno di *Rift.ware* (capitolo 3).

7.8 Workflow e logica della soluzione

Andiamo adesso a rivedere come funziona complessivamente la soluzione. A questo scopo ci viene in aiuto la figura 7.11, che mostra l'architettura della soluzione nelle sue componenti principali. Questa vista mette al centro l'obiettivo finale, ovvero la creazione del servizio di sicurezza attraverso il paradigma SECaaS, utilizzando tecnologie NFV. Vediamo che il servizio di sicurezza permette all'utente di accedere alla rete esterna passando attraverso la Service Function Chain costituita dalle nostre vNSF. Questo, in ultima analisi, permette di ottenere la sicurezza desiderata per il flusso di traffico dell'utente.

Ancora nella stessa figura è rappresentata la parte di gestione, ovvero il Security Service Manager, ma da un punto di vista diverso, mostrando di fatto tutti gli elementi cruciali per la realizzazione

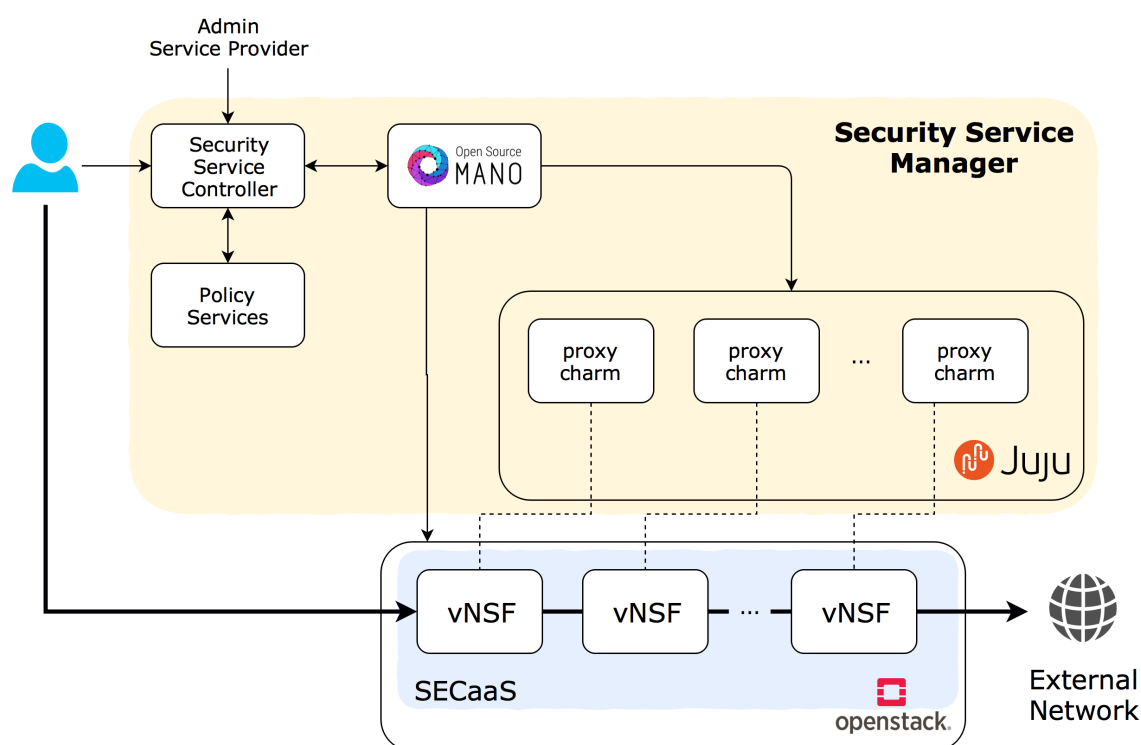


Figura 7.11: Vista architetturale complessiva con i moduli principali.

dell'obiettivo. Da una parte il Security Service Controller, che gestisce le richieste di modifica delle policy da parte dell'utente e le richieste di istanziazione del Service Provider. Dall'altra il supporto dei servizi legati alle policy (Policy Services) e Open Source MANO che coadiuvano il modulo precedentemente descritto. In ultima analisi, abbiamo Juju, modulo anch'esso cruciale in quanto artefice della configurazione automatica delle vNSF.

7.8.1 Lato utente

Andiamo ora a particularizzare il workflow visto dal lato dell'utente. L'unica operazione consentita allo stesso è quella, attraverso il Security Service Controller, di modificare o creare le policy di alto livello. Come si vede in figura 7.12, la prima operazione è proprio quella di collegarsi al Security Service Controller e, attraverso la policy GUI, sottoporre le nuove policy al sistema. Una volta che questa operazione è terminata, si avvia il processo di trasformazione, che in prima analisi prevede la memorizzazione delle policy di alto livello all'interno dell'UPR. A questo punto vengono richiamati in sequenza i servizi per la traduzione di queste policy, a partire dall'H2M fino al servizio di traduzione in configurazioni di basso livello.

Tutti questi servizi fanno uso del repository e nell'ultima fase memorizzano il RAG e le configurazioni di cui prima nello stesso. Una volta che tutto il processo è stato completato l'utente ottiene la notifica di corretta modifica/creazione delle policy.

7.8.2 Lato service provider

Per quanto riguarda la prospettiva del service provider, invece, lo scopo è dato dall'istanziazione e configurazione del servizio di sicurezza. La prima operazione è quella di contattare il Security Service Controller per far sì che esso controlli che le policy siano presenti nel repository UPR e dia inizio al processo di istanziazione del servizio di sicurezza.

Come rappresentato in figura 7.13, una volta effettuati tali controlli, l'SSC acquisisce il RAG e le configurazioni di basso livello, relative all'utente, dall'UPR e si occupa della traduzione del RAG

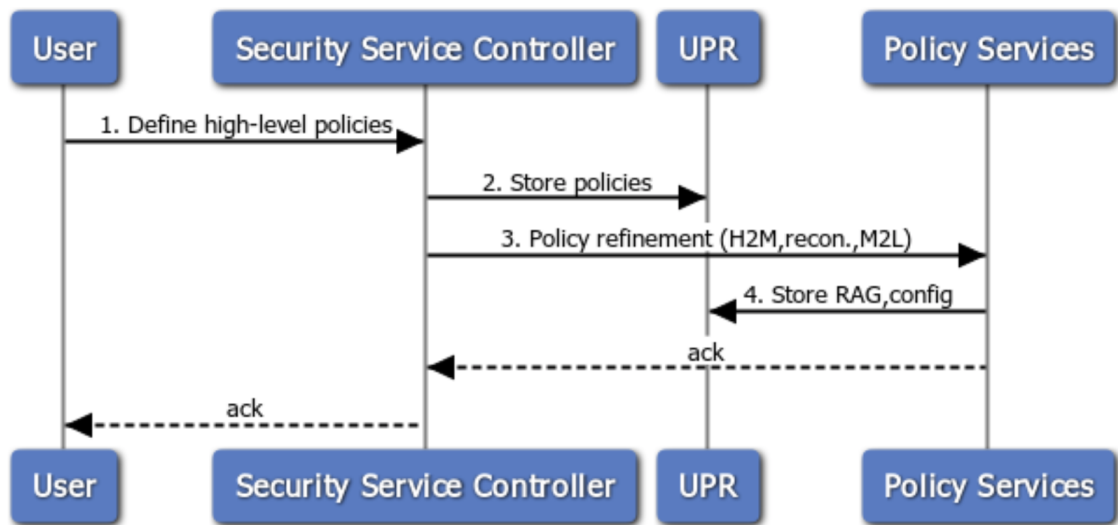


Figura 7.12: Workflow modifica delle policy da parte dell'utente.

in NSD (o per come lo vediamo noi NSSD). Una volta effettuata la traduzione in modo corretto, il Security Service Controller contatta OSM ed avvia la procedura di on-boarding dell'NSS, al termine della quale sarà possibile avviare il deploy dello stesso.

Come abbiamo visto in precedenza, Open Source MANO da questo punto in poi è in grado di istanziare e configurare il servizio di sicurezza in modo indipendente da tutta la parte di gestione delle policy. In sequenza quindi contatterà il VIM, istanziando le vNSF, e successivamente Juju per il deploy dei proxy charm e la loro configurazione.

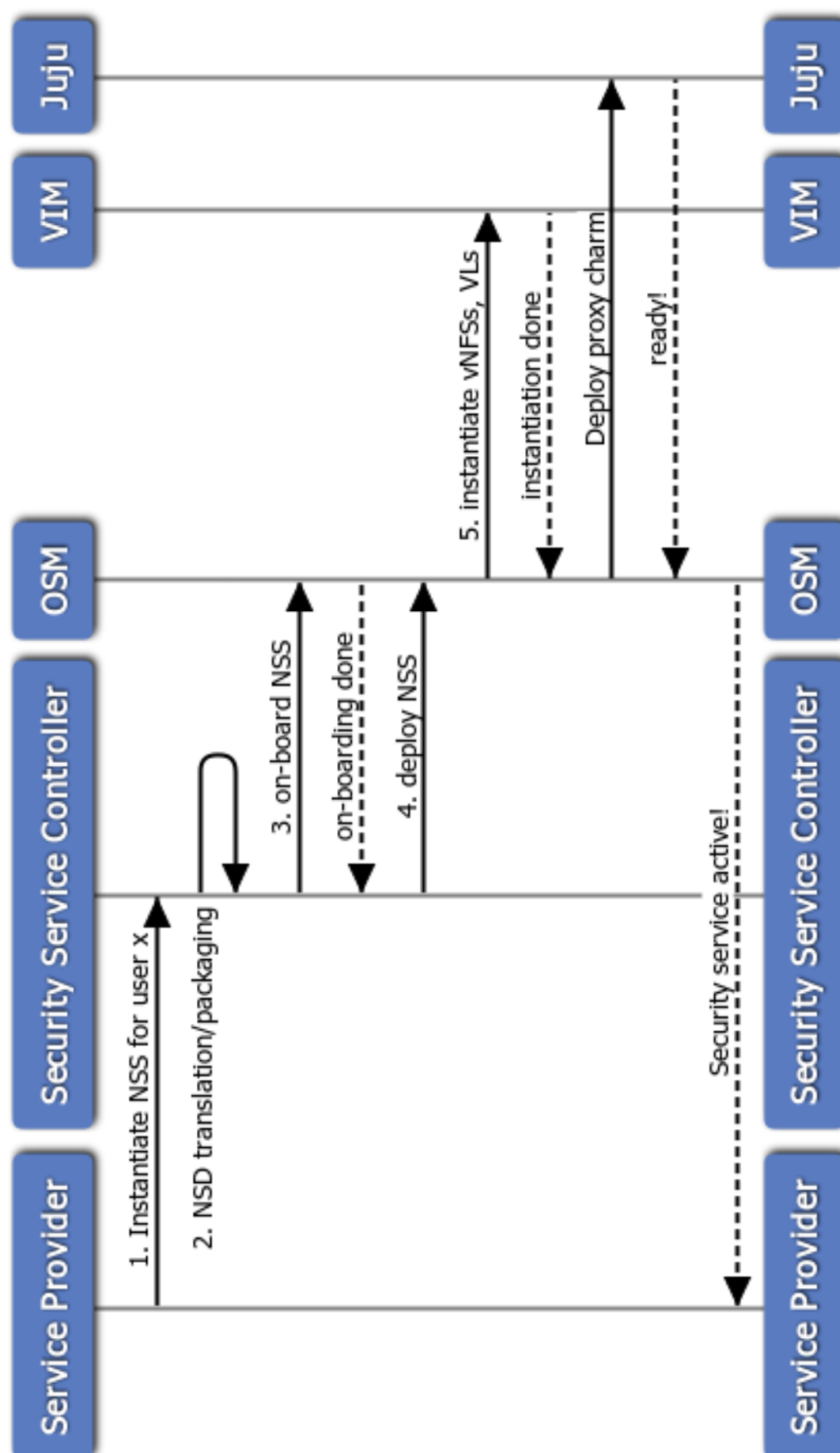


Figura 7.13: Workflow deploy del servizio da parte del Service Provider.

Capitolo 8

Implementazione e Testing

L'obiettivo di questo capitolo è quello di fornire un Proof-of-Concept che possa valutare la fattibilità e le prestazioni della soluzione progettata nel precedente capitolo.

8.1 Proof-of-Concept generale

Per la realizzazione del Proof-of-Concept sono state allestite due diverse macchine fisiche. La prima ospita tutta la parte di controllo: Security Service Manager. La seconda, invece, l'infrastruttura e il VIM: openstack, sia controller node che compute node.

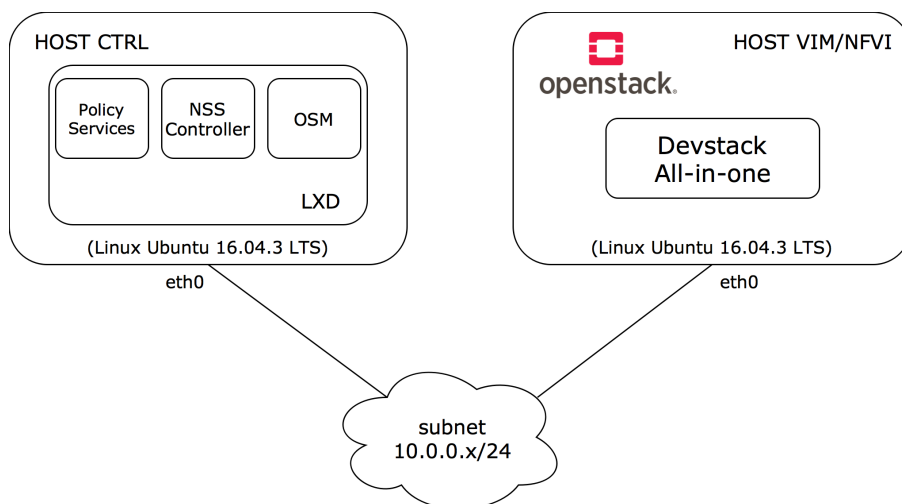


Figura 8.1: Architettura hardware Proof-of-Concept.

In figura 8.1 è rappresentato schematicamente l'allestimento di tale macchine. I due host in figura comunicano attraverso una sottorete ed espongono un'interfaccia fisica (**eth0**). Risulta utile testare la soluzione su due macchine diverse per varie ragioni. La prima è relativa al fatto che gli strumenti utilizzati necessitano di elevate prestazioni hardware, basti pensare che tali soluzioni tipicamente alloggiavano su datacenter e hanno a disposizione risorse consistenti. La seconda è che questo scenario ci permette di simulare il caso in cui la parte di controllo sia fisicamente dislocata rispetto all'infrastruttura NFVI.

Le macchine utilizzate come abbiamo detto sono due e nello specifico andiamo a definirne le caratteristiche tecniche di seguito:

HOST CTRL

- **CPU:** Intel Core i7 quad-core 2,6 GHz
- **RAM:** 16 GB
- **OS:** Linux Ubuntu 16.04.3 LTS (xenial)

HOST VIM/NFVI

- **CPU:** Intel Core i7 quad-core 2,6 GHz
- **RAM:** 16 GB
- **OS:** Linux Ubuntu 16.04.3 LTS (xenial)

8.2 Security Service Manager

La parte architetturale corrispondente al Security Service Manager, progettato nel capitolo 7, è stata implementata attraverso l'uso della tecnologia LXD.

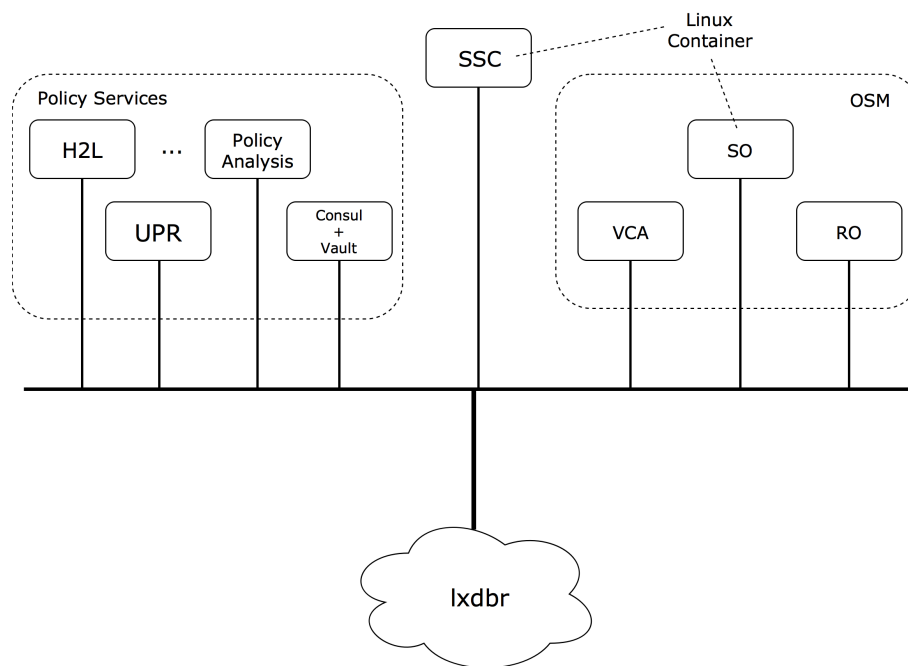


Figura 8.2: Implementazione del Security Service Manager in tecnologia LXD.

Come si vede in figura 8.2, per ognuno dei moduli presentati nel capitolo 7 ad alto livello, abbiamo un insieme di container in tecnologia LXD. A partire da sinistra troviamo i moduli dei servizi per la gestione delle policy, dei quali vedremo un esempio di reingegnerizzazione nell'appendice C (in particolare per il modulo UPR). Continuando abbiamo il Security Service Controller (SSC) che è il modulo main di tutta la soluzione ed espleta le funzioni che abbiamo visto nel capitolo 7. In ultima analisi, abbiamo Open Source MANO con i suoi tre moduli principali, anch'essi ampiamente dettagliati nei capitoli precedenti (cap. 3 e 6).

La scelta di creare un linux bridge (lxdbr) e collegare tutti i moduli LXD alla relativa sottorete è utile in quanto ogni sottomodulo è in grado di comunicare con gli altri, facilitando l'integrazione di futuri componenti aggiuntivi, composti da uno o più moduli. Tutto quello che sarà necessario fare, per integrare un nuovo componente, sarà lanciare un nuovo container LXD e installare tutti i

software e applicativi necessari, contando sul fatto che quest'ultimo sarà già pronto ad interagire con gli altri. Come dettagliato in precedenza è stata prevista anche l'eventuale aggiunta di un modulo di autorizzazione e autenticazione, questo potrebbe essere realizzato attraverso i medesimi strumenti utilizzati per il modulo Policy Services. In particolare si potrebbe pensare ad un'estensione di Consul e Vault a tutto il Sercurity Service Manager. Questo potrebbe fornire end-point comune a tutti i servizi giacenti sul linux bridge e la gestione centralizzata dell'autenticazione attraverso i token di Vault (maggiori dettagli nella precedente sez. 7.7).

8.3 Esempio reingegnerizzazione moduli SECURED: UPR

Ciascuno dei moduli software presenti nel progetto SECURED possono essere riutilizzati e riconvertiti in modo semplice in servizi esposti tramite LXD container. Questo processo di reingegnerizzazione permette di integrare ognuno di questi moduli nel nostro framework.

Sembra opportuno, come guida, fornire un esempio di come uno dei moduli sopra citati è stato reingegnerizzato al fine renderlo compatibile con la nostra soluzione. Il modulo scelto a tale scopo è l'User Policy Repository (UPR), modulo che fornisce tutti i servizi e le REST API per l'accesso alla base di dati dove vengono memorizzate le policy e i relativi attori che le hanno definite. La parte architetturale verrà fornita di seguito, mentre una guida pratica sarà presente nell'appendice C.

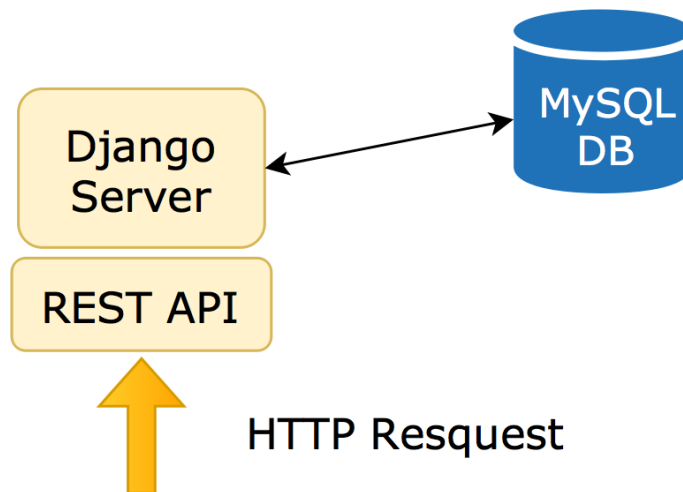


Figura 8.3: Architettura User Policy Repository.

Come si vede dalla figura 8.3, il modulo UPR è composto da un server Django, che contiene tutta la logica applicativa ed espone le REST API per l'interazione con la base di dati, e da un database MySQL, nel quale sono memorizzate informazioni relative a:

- Utenti del servizio di sicurezza (coloro i quali definiscono le policy);
- Policy di alto livello associate all'utente che le definisce;
- Policy di medio livello associate all'utente che le definisce;
- vNSF disponibili;
- Lista di configurazioni di basso livello generate ed associate all'utente e alle relative vNSF;
- Application Graph e Reconciled Application Graph associati all'utente;

Le REST API e l'utilizzo specifico di questo modulo verrà specificato in appendice C.

8.4 Infrastructure e VIM

Per quanto riguarda la parte infrastrutturale e di VIM abbiamo scelto, come già accennato, OpenStack. Esistono vari modi per installare OpenStack e ne abbiamo esplorati vari. La soluzione comune in genere è quella di utilizzare almeno due nodi, uno che funga da controller e l'altro che funga da compute. Ognuno dei servizi che lo compone (nova, neutron, cinder, glance, horizon) va installato separatamente e registrato al keystone, in modo che quest'ultimo possa esporre i suoi servizi previa autenticazione. Parte dei servizi va installata sul nodo di controllo, mentre altri (come nova-compute e il neutron-agent) vanno installati su quelli di compute. Una volta che i nodi sono stati creati ed inizializzati ogni nodo di computazione si autenticherà a quello di controllo e sarà così a disposizione per l'istanziamento di VM. Ci sono varie declinazioni architetturali per OpenStack, ad esempio è possibile separare, oltre che i nodi di controllo e di compute, anche i nodi di rete e quelli di storage persistente, ma questo non interessa per i nostri scopi.

Nell'implementazione del nostro PoC, non avendo a disposizione diversi nodi per poter testare una release distribuita di OpenStack, abbiamo esplorato varie alternative, altrettanto valide per effettuare dei test. Le alternative esplorate sono state tre:

- Devstack;
- OpenStack labs;
- Conjure-up.

La prima delle alternative è uno strumento utilizzato in particolar modo dagli sviluppatori di OpenStack. Questo consente di installare su un singolo nodo fisico (o in alternativa VM), tutti i servizi di OpenStack. In modo particolare è possibile avere le unità di controllo e le risorse allocate sulla stessa macchina. Questo è un notevole risparmio in termini di tempo ed è una scelta molto flessibile che nasconde problematiche che non sono di interesse per chi ha necessità di testing.

La seconda alternativa, ovvero OpenStack labs, è una soluzione che prevede l'utilizzo di due macchine virtuali in tecnologia VirtualBox o KVM, che facciano rispettivamente da compute node e da controller node. Queste due macchine vengono create e correlate da un insieme di script messi a disposizione direttamente dalla OpenStack Foundation. Il problema di questa soluzione è il fatto che risulta molto limitata in termini di prestazioni, perché lavorare con due macchine virtuali che richiedono notevoli prestazioni sullo stesso host fisico diventa già di per sé complesso. Poi si aggiunge la questione relativa alla virtualizzazione innestata, ovvero sul compute node già virtualizzato devono essere virtualizzate a loro volta le VM instance di OpenStack e questo in termini di prestazioni converge in un degrado non accettabile per i nostri scopi.

L'ultima alternativa è Conjure-up, ovvero uno strumento messo a disposizione dalla Canonical, per effettuare l'installazione di servizi complessi (come Openstack, Hadoop e Kubernetes) attraverso l'utilizzo di Juju. Il risultato di questa installazione automatica è che ognuno dei servizi di OpenStack, citati in precedenza, viene ospitato da un diverso LXD container e questi container vengono gestiti, istanziati e terminati con dei charm da Juju. I collegamenti tra gli stessi vengono definiti attraverso un bundle, il tutto come abbiamo visto per il caso tipico di Juju nel capitolo 6. Questa soluzione può consentire anche l'istanziamento, a dispetto delle classiche VM, di LXD container con immagini software adeguate (es. ubuntu-lxd-trusty), sfruttando l'integrazione di nova-lxd.

Quest'ultima versione risulta interessante, soprattutto nel caso in cui il deploy di tutto il servizio di orchestrazione venga effettuato da Juju, ma è adatta più in ambienti di produzione che di sviluppo, in quanto richiede anch'essa delle prestazioni ingenti per l'esecuzione e la disposizione di un certo numero di LXD container.

La scelta per la nostra soluzione è ricaduta su devstack, che si è dimostrato il tool più flessibile tra quelli presentati, mantenendo delle prestazioni accettabili anche in un ambiente di testing. Daremo tutti i dettagli sulla configurazione e messa in opera dell'ambiente devstack nell'appendice D.

Per quanto riguarda l'autenticazione e il service discovery è previsto, come da standard per OpenStack, l'utilizzo del servizio Keystone.

8.5 Network Security Service

Abbiamo visto nel capitolo 7 che il servizio di sicurezza risulta il punto di partenza cruciale per l'istanziamento delle vNSF. Abbiamo in oltre visto che le informazioni che deve contenere per adempiere al suo compito sono fondamentalmente tre: definire le vNSF che ne fanno parte, definire i link tra le stesse e istanziare eventualmente dei nuovi link.

L'operazione di definizione dell'NSS è stata automatizzata nel modulo del Security Service Controller dedito alla generazione dell'NSD. Questo significa che a partire dai dati del RAG (memorizzato sull'UPR) è possibile definire il codice di un servizio di sicurezza, ne vedremo un esempio nell'appendice B.

```

/example.nsd
├── checksum.txt
└── example_user_nsd.yaml

```

Figura 8.4: Esempio di package tar.gz di un NSSD.

Nel frattempo presentiamo la struttura del package in figura 8.4, descrittore del servizio di sicurezza. All'interno dello stesso troviamo il descrittore `example_user_nsd.yaml`, che riporta tutto quanto specificato nel capitolo 7 per quanto riguarda la sua progettazione, e un controllo di integrità tramite il file `checksum.txt`.

8.5.1 Mapping dei virtual link di OSM su OpenStack

Una problematica di interesse è legata a come OSM mappa i link virtuali sull'infrastruttura di OpenStack; questo avviene attraverso la definizione di nuove reti e sottoreti locali con neutron. Tipicamente un link virtuale è visto come una sottorete e può comprendere servizi aggiuntivi come un DHCP server o l'assegnazione di indirizzi IP statici predefiniti. In OSM è possibile settare questi parametri e altri ancora con estrema semplicità tramite gli strumenti definiti nel capitolo 6. Un campo specifico che permette di gestire OSM è il “port security”, questo è un parametro presente su ogni interfaccia di VM instance presente in OpenStack. Tale parametro rappresenta un controllo degli accessi sui vari link di OpenStack, infatti, quest'ultimo mantiene un'associazione tra gli indirizzi IP e il MAC address assegnato alle varie porte. Quando questo vincolo non viene soddisfatto (es. quando si utilizza SNAT), viene effettuato il drop dei pacchetti. Questa è una contromisura per evitare l'IP spoofing, ma in alcune situazioni può essere necessario disabilitare tale controllo.

8.6 Sviluppo di una vNSF

Un'interessante innovazione nella presente soluzione, è il disaccoppiamento della fase di progettazione e implementazione dei meccanismi di configurazione delle vNSF dalla definizione del servizio di sicurezza. Questo disaccoppiamento avviene grazie alla completa implementazione della struttura delle vNSF e delle relative configurazioni all'interno del package descrittore.

Come abbiamo visto nel capitolo 7 esistono diverse modalità per la creazione di una nuova vNSF. L'approccio che era legato all'utilizzo ad un'immagine monolitica (esempio progetto SECURED), con tutto il necessario già precaricato, l'abbiamo scartato, lasciandone una descrizione nei precedenti capitoli (cap. 7) in quanto potrebbe risultare utile come legacy rispetto al passato. In questa sezione ci concentreremo, invece, nel fornire una prospettiva nuova sul come progettare le vNSF con tecnologia cloud-init, soluzione che come visto permette di alleggerire il carico dovuto ad immagini di dimensioni considerevoli.

L'approccio che approfondiremo è quindi quello legato all'utilizzo in combinazione delle tecnologie cloud-init e Juju per la configurazione delle vNSF. Presenteremo, a scopo illustrativo, il caso di una semplice vNSF che funge da Packet Filter.

8.6.1 Packet Filter

Questa vNSF corrisponde alla capability `PacketFilteringCapability` e ricorre in questo caso semplicemente all'utilizzo di iptables; facendo l'injection delle preferenze dell'utente all'interno della vNSF come comandi per iptables, è possibile configurare la funzione di sicurezza per il suo corretto funzionamento.

Per la realizzazione di una vNSF generica occorre seguire i seguenti passaggi:

- **Individuazione del numero di unità VDU;**
- **Creazione dei link virtuali interni;**
- **Per ognuna delle VDU stabilire i meccanismi di configurazione;**
- **Individuare il numero di interfacce esterne e i loro identificativi;**
- **Definire gli script cloud-init e i Juju charm per ogni VDU interessata;**
- **Definire il descrittore yaml della vNSF, delineando le caratteristiche delle VDU e della vNSF complessiva;**
- **Procedere con la creazione del package e l'on-boarding della vNSF.**

Nel nostro caso la vNSF risulta molto semplice, quindi non conviene suddividerla in più unità, avremo quindi un'unica VDU. Quest'ultima non necessiterà di link interni, dobbiamo solo stabilire quali siano i meccanismi di configurazione della stessa. In questo caso utilizziamo una cloud image di base e carichiamo al boot tutto ciò che è necessario per la vNSF. Successivamente con Juju andremo a contattare il Security Service Controller e otterremo così le configurazioni per quello specifico utente. Una volta ottenute queste è possibile applicarle direttamente alla vNSF in questione. Questa logica viene espressa nella definizione dello script cloud-init e del Juju charm (appendice B). Abbiamo saltato il passaggio di scelta delle interfacce esterne perché abbiamo stabilito di lavorare, in questo caso per semplicità, con una singola catena, quindi avremo solo le tre interfacce definite in fase di progettazione (capitolo 7).

Una volta che i metodi di configurazione sono pronti, andiamo a descrivere il tutto nei file yaml (esempio in appendice B) e otterremo così tutta la logica della vNSF, in termini di risorse, istanziazione e configurazione.

L'ultimo passaggio è, attraverso degli script creati ad hoc, quello di creare il package ed effettuare l'on-boarding. A questo punto la vNSF potrà essere utilizzata in qualsiasi servizio di sicurezza.

Un esempio di package relativo al vNSF Descriptor è dato in figura 8.5, dove si evince come la struttura sia del tutto analoga a quella presentata nel capitolo 6 per un descrittore di VNFD, ovviamente con i charm e gli altri file di supporto, in questo caso, specializzati per funzioni di sicurezza. Per i dettagli sul codice rimandiamo all'appendice B.

8.6.2 Variante per ottimizzare le risorse

Come abbiamo visto nei precedenti capitoli 6 e 7, una vNSF che prevede l'utilizzo di juju charm introduce un overhead consistente (mostriamo i risultati con precisione nella sezione di testing 8.7.4) nell'istanziazione della stessa. Diventa quindi logico, capire se sia o meno il caso di introdurre un meccanismo in grado di ottimizzare tali tempi di istanziazione.

Una soluzione a questa problematica potrebbe essere innanzitutto quella di utilizzare i Juju proxy charm solo nel caso in cui effettivamente siano utili e quindi ci sia la volontà di effettuare delle configurazioni anche durante il ciclo di vita della vNSF, lasciando la configurazione di tutte le altre vNSF a carico della cloud-init.

```

/example_vnsfx_vnfd
├── charms
│   └── example_vnsfx_charm
│       └── charm_filesndirs
├── cloud_init
│   └── cloud_init.cfg
├── icons
│   └── vendor_logo.png
└── example_vnsfx_vnfd.yaml

```

Figura 8.5: Esempio package VNFD.

Quello che introdurrebbe un ulteriore grado di ottimizzazione sarebbe l'utilizzo di un approccio ibrido. Questo significa, che anche quando è necessario Juju e il charm relativo per un'eventuale configurazione Day-1, possiamo effettuare la prima configurazione (quella iniziale) con la cloud-init. Questo ci permetterebbe di sfruttare la velocità e l'efficienza di cloud-init per avere subito a disposizione il servizio e successivamente dopo un intervallo di tempo quantificabile, avere a disposizione i meccanismi di configurazione Day-1. Questo è fattibile perché tipicamente le configurazioni successive alla prima non vengono effettuate immediatamente dopo l'istanziamento, quindi questo permette di effettuare il deploy del charm senza problemi, ma nello stesso tempo la vNSF è già operativa.

Vedremo i test che giustificano l'utilizzo dell'approccio ibrido nella sezione test e i dettagli del codice nell'appendice [B](#).

8.6.3 Configurazioni con Ansible

Abbiamo definito Ansible nel capitolo [6](#), come una delle più utilizzate ed efficienti soluzioni per la configurazioni delle vNSF presenti nel panorama cloud. Questo è riscontrabile anche tra gli sviluppatori di OSM e dal loro forte interesse nell'integrazione di Ansible in OSM.

Dopo alcuni sforzi, è stato individuato un modo per integrare Ansible nelle possibili scelte di configurazione di OSM. Questo meccanismo prevede l'utilizzo sempre del proxy charm. Viene creato, infatti, per la vNSF un charm particolare, che tra i suoi livelli base ne ha uno sviluppato per fornire le primitive di esecuzione di un playbook Ansible. Questo strato software permette di eseguire una serie di playbook Ansible in risposta a determinate action. Un playbook Ansible è uno script, in formato yaml, che permette di eseguire tutte le operazioni che possono essere effettuate sia da cloud-init che da Juju su una vNSF, mettendo a disposizione una sintassi significativamente flessibile ed espressiva.

L'integrazione di questo strumento rappresenta un grande passo avanti per OSM, in quanto Ansible è considerato da molti esperti di soluzioni RHEL la soluzione migliore per il deploy e la configurazione di istanze virtuali. Presenteremo i dettagli di tale integrazione nell'appendice [B](#).

8.7 Testing

Per quanto riguarda la fase di testing della soluzione implementata, ci si è concentrati su un aspetto estremamente sensibile per un servizio di sicurezza e più in generale un servizio di rete: il tempo necessario per l'istanziamento e configurazione dello stesso.

Come prima cosa sono state aggiunte due VNF specializzate per consentire la valutazione della bontà della Service Function Chain derivante dal servizio di sicurezza ([8.6](#)). La prima (user) è una funzione virtuale che simula il comportamento dell'utente e permette di usufruire del servizio di sicurezza nel collegarsi alla rete esterna. L'ultima (gateway) è stata aggiunta per consentire l'accesso alla rete esterna, avendo capacità di NAT ed un collegamento all'interfaccia del router

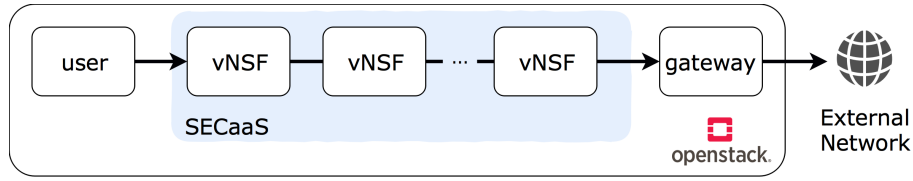


Figura 8.6: Modifiche architetturali per la fase di testing.

di OpenStack che consente l'accesso alla provider network. Successivamente sono stati creati in modo sequenziale servizi di sicurezza con un numero sempre crescente di vNSF per valutare quali fossero i tempi di inizializzazione e configurazione.

Sembra utile ricordare a questo punto che per inizializzazione intendiamo le tempistiche per effettuare il boot dell'istanza e renderla disponibile a tutte le configurazioni necessarie per predisporla alla sua funzione nell'NSS. Per configurazione, invece, intendiamo il tempo necessario al sistema di orchestrazione per applicare, da una parte le configurazioni menzionate, dall'altra per personalizzare le istanze nel caso delle cloud-image.

Una più formale definizione delle tempistiche necessarie alla completa istanziazione e configurazione di un servizio di sicurezza è dato dalla 8.1.

$$T_{tot} = t_{trans} + t_{init} + t_{conf} \quad (8.1)$$

Tale formula definisce come T_{tot} il tempo impiegato dal servizio di sicurezza per poter essere definito attivo. Gli altri membri dell'equazione, ovvero t_{trans} , t_{init} , t_{conf} , rappresentano rispettivamente il tempo di traduzione del RAG in NSD, il tempo di inizializzazione delle vNSF e il tempo di configurazione delle stesse e del servizio di rete. Le tempistiche sono calcolate in secondi, per questo motivo il tempo di traduzione può essere trascurato in quanto $t_{trans} \ll 1s$. Dedicheremo agli altri termini la nostra attenzione nelle successive sezioni.

8.7.1 Tecnologie utilizzate e scenari di test

Abbiamo discusso di varie tecnologie nel corso di questo capitolo e di quelli precedenti, soprattutto inerenti alla configurazione delle funzioni di rete virtuali. Sembra opportuno quindi testare la nostra soluzione in diversi scenari, utilizzando ognuna delle tecnologie presentate, così da apprezzare quali sono in termini quantitativi le differenze tra l'utilizzo di una rispetto ad un'altra.

Le tecnologie che varieremo nel corso dei test sono relative sia alla costruzione delle immagini software che alla configurazione e sono le seguenti:

- cloud-image con cloud-init;
- immagini software monolitiche precostituite;
- proxy charm e Juju.

Queste tecnologie, nello specifico, sono state utilizzate per la creazione di tre diversi scenari di test, che discuteremo in dettaglio in seguito:

- Confronto tra le prestazioni di cloud image con tecnologia cloud-init ed immagini monolitiche nelle fasi di istanziazione e configurazione;
- Confronto tra tempi di istanziazione e configurazione nel caso di utilizzo di uno o due VIM in parallelo;
- Calcolo dell'overhead nell'utilizzo di Juju per un'eventuale configurazione iniziale delle vNSF con tale strumento.

8.7.2 Cloud-image vs Monolithic Image

In questa sezione affronteremo il confronto nei tempi di istanziazione e configurazione tra due differenti tecnologie: cloud-image con cloud-init e immagini monolitiche (figura 8.7).

Una prima precisazione concerne la rivisitazione dei concetti di cloud-image e immagine monolitica. La tecnologia cloud-image è alla base delle immagini software utilizzate nella nostra soluzione e consente, a partire da un template di base, di istanziare diverse copie della stessa VM, utilizzando come strumento di personalizzazione la tecnologia cloud-init. Tutti i dettagli sono stati discussi nei capitoli 6 e 7.

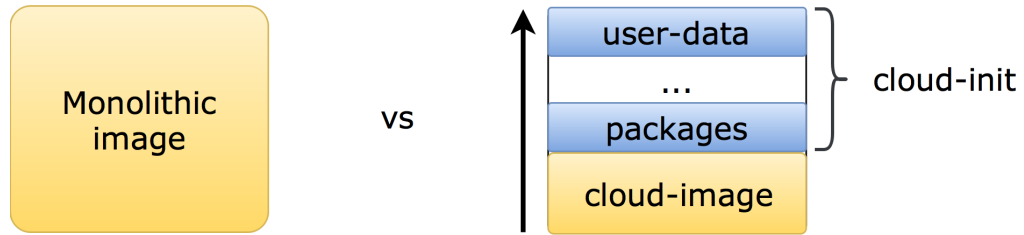


Figura 8.7: Monolithic image vs cloud-image/cloud-init.

Per quanto riguarda il concetto di immagine monolitica, intendiamo un'immagine che è stata preconstituita utilizzando strumenti come virt-edit o guestfish e che al boot contiene tutto ciò che è necessario per essere eseguita. Nei capitoli precedentemente citati, sono presenti confronti più dettagliati.

In sintesi la differenza è che la prima tecnologia (cloud-image) richiede un'immagine software molto leggera di base e costruisce su di essa tutto quello che serve all'istanza virtuale in fase di boot della stessa. La seconda tecnologia non richiede questo processo ma risulta essere molto pesante e talvolta poco ottimizzata.

Nr. istanze	$t_{init}(s)$	$t_{conf}(s)$	$T_{tot}(s)$
1	11	3	14
3	18	6	24
5	26	5	31
7	28	7	35
8	36	4	40
9	40	7	47
10	52	4	56
18	70	10	80
20	75	8	83
30	105	7	112

Tabella 8.1: Tempi di inizializzazione e configurazione vNFS con tecnologia cloud-image.

L'utilizzo della tecnologia cloud-image ovviamente ha enormi vantaggi in termini di flessibilità e consumo di spazio nei repository dei VIM; quello che vogliamo stimare è se i tempi di istanziazione risentono sensibilmente dei tempi di inizializzazione con cloud-init rispetto al caso base di immagine monolitica. In pratica la configurazione Day-0 con cloud-init (per installare package e file) comporta teoricamente un overhead in termini di inizializzazione della VM; questo potrebbe pesare sui tempi di istanziazione dell'NSS. Quello che vogliamo stimare è se l'immagine di base più piccola (cloud-image) e la tecnologia cloud-init sono ottimizzate a tal punto da non risentire in modo consistente di questo overhead. Per contro l'immagine pesante (monolitica) e la sua poca ottimizzazione (derivante ad esempio dalla presenza di software e file non necessari) potrebbe introdurre altrettanto. Dal bilanciamento di questi due aspetti, vogliamo valutare empiricamente se esiste e quanto è consistente un eventuale svantaggio nell'utilizzo di cloud-image. Per valutare le differenze di tempistiche sono state prese in considerazione delle vNFS, costitutive del servizio di

sicurezza, che rappresentano il carico medio di una tipica vNSF e un buon template per l'eventuale sviluppo di ulteriori vNSF.

In tabella 8.1 sono riportati i risultati dei test sulla tecnologia cloud-image. In particolare tempi di lancio ($t_{init}(s)$), di configurazione ($t_{conf}(s)$) e il totale di questi combinati ($T_{tot}(s)$). Questi dati da soli danno un'idea delle tempistiche di inizializzazione di un servizio di sicurezza, ma vanno combinati con quelli dell'altra tecnologia per avere un'interessante lettura, utile a cogliere informazioni di rilievo per il test.

Prima però andiamo a mostrare i dati relativi alla tecnologia con immagini monolitiche in figura 8.2. Tutto sempre facendo riferimento alle precedenti misurazioni.

Nr. istanze	$t_{init}(s)$	$t_{conf}(s)$	$T_{tot}(s)$
1	15	6	21
3	17	4	21
5	24	6	30
10	50	12	52
18	71	9	80
20	75	10	84
30	101	8	109

Tabella 8.2: Tempi di inizializzazione e configurazione vNSF con immagini monolitiche.

Ora andiamo a combinare i dati delle due precedenti tabelle e a vedere qual'è l'andamento dei due grafici corrispettivi, sovrapposti in figura 8.8. Come si evince dalla stessa al crescere del numero delle istanze l'andamento delle due curve risulta pressoché identico, questo porta ad un'importante considerazione: l'utilizzo di tecnologia cloud-image non introduce significativi ritardi nell'istanziamento delle vNSF. A questo punto dati i vantaggi della prima tecnologia, vediamo in questo test una conferma della scelta progettuale dell'utilizzare cloud-image anziché immagini precostituite; riusciamo ad utilizzarne i benefici senza perderne sensibilmente in prestazioni.

Un esempio dei benefici di cloud-image rispetto all'utilizzo di immagini monolitiche, già discusso in precedenza, è l'occupazione in termini di spazio su glance (repository immagini). I dati al variare delle della tipologia di vNSF sono riportati in figura 8.9. Dalla stessa si vede chiaramente come con l'unica immagine ad esempio di xenial-ubuntu è possibile servire tutta una serie di vNSF, mentre nel caso di immagini monolitiche, ne va aggiunta una nuova (oltretutto pesante) per ogni tipologia di vNSF nel repository.

A questo punto possiamo abbandonare, anche in fase di testing, l'utilizzo della tecnologia con immagini monolitiche e concentrarci sugli altri scenari di test, prendendo come riferimento di base sempre cloud-image.

Cloud-image in dettaglio

Lo scopo di questa sezione è in particolare del seguente scenario è quello di illustrare quali sono le tempistiche in dettaglio che caratterizzano questa tecnologia e qual'è il loro andamento.

In figura 8.10 sono presentati i tempi di inizializzazione e configurazione della sola tecnologia cloud-image. Lo scopo di questa finestra sui dati della cloud-image è quella di capire come evolvono separatamente i tempi di inizializzazione (t_{init}) e quelli di configurazione (t_{conf}) al variare del numero delle istanze.

Dai risultati capiamo che quelli che rappresentano la vera variabile sono quelli di inizializzazione, infatti, quelli di configurazione restano pressoché nello stesso intervallo e dipendono molto da variabili legate al sistema di orchestrazione e dal VIM, piuttosto che alla quantità di VM presenti nel servizio. Questo è dovuto al fatto che la loro configurazione è totalmente parallela, quindi a prescindere dal numero di istanze i loro tempi di configurazione saranno sempre appartenenti allo stesso intervallo (potremmo considerarli per semplicità costanti facendone una media). Questo è probabilmente uno dei fattori determinanti nel precedente confronto con immagini monolitiche, grazie a questo si riescono a mantenere tempi di inizializzazione pressoché invariati.

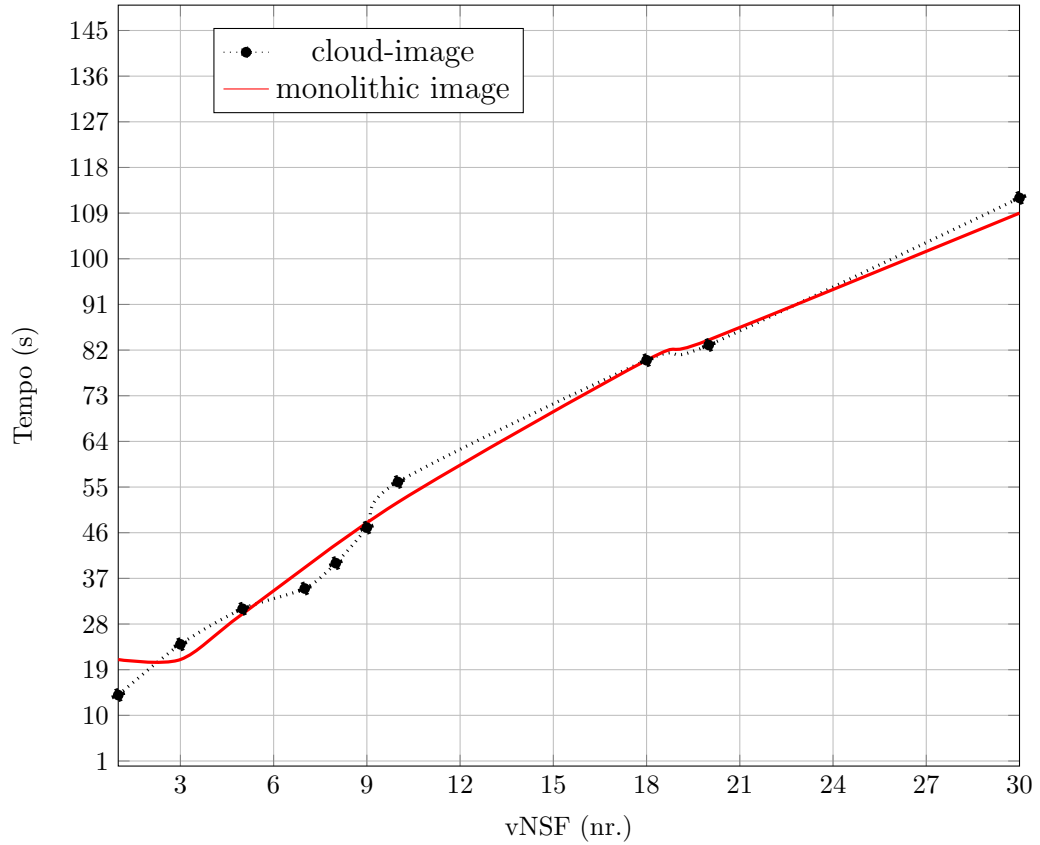


Figura 8.8: Confronto tra le tecnologie cloud-image e monolithic image.

8.7.3 Scenario cloud-image mono-site o multi-site

In questa sezione andremo a valutare quali sono i tempi di inizializzazione e configurazione al variare del numero di VIM collegati all'orchestratore, quindi in uno scenario multi-site (figura 8.11).

Per prima cosa mostriamo in figura 8.3 i dati rilevati nello scenario multi-site, dati che andranno incrociati con quelli dello scenario base riportato in sezione 8.7.2 e relativi alla cloud-image mono-site (su un unico VIM). Nel nostro caso particolare lo scenario multi-site è stato ricreato con l'utilizzo di due VIM ed ha solo lo scopo di verificare l'andamento al cospetto di tale modifica architetturale, non di dare un preciso riferimento quantitativo sulla capacità del sistema di scalare.

Nr. istanze	$t_{init}(s)$	$t_{conf}(s)$	$T_{tot}(s)$
2 (1/1)	15	3	18
4 (2/2)	19	6	24
6 (3/3)	23	5	31
12 (6/6)	33	7	35
20 (10/10)	59	8	67
30 (15/15)	86	11	97
40 (20/20)	133	15	148

Tabella 8.3: Tempi di inizializzazione e configurazione vNFS con tecnologia cloud-image (multi-site).

Una volta presentati i dati, possiamo valutare quali sono gli effetti della sovrapposizione dei due scenari che abbiamo discusso e lo facciamo con la figura 8.12. Inizialmente, con un piccolo numero di istanze il comportamento dei due scenari è identico; solo dopo aver superato le 7 istanze vediamo il sistema convergere a cosa ci aspettavamo, ovvero un miglioramento costante delle prestazioni. In

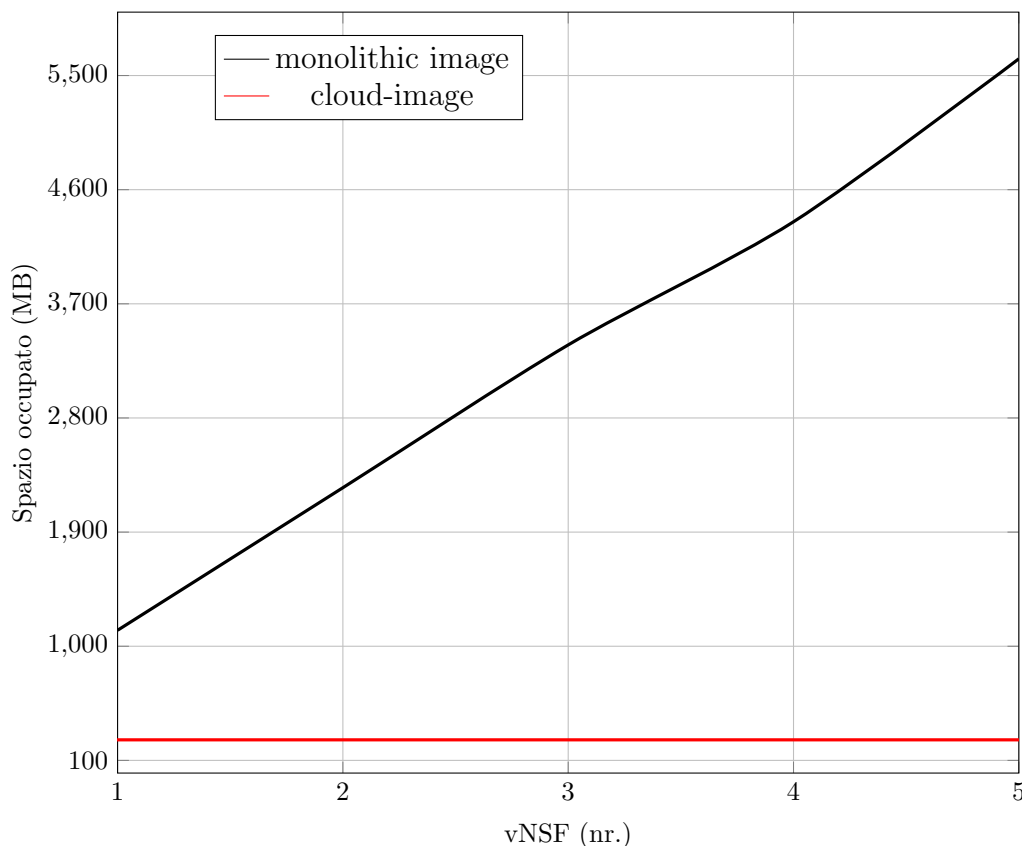


Figura 8.9: Differenza nell’occupazione di spazio su disco al crescere della tipologia di vNSF supportate.

particolare quello che vediamo è che, al crescere del numero delle istanze, le due soluzioni, mono-site e multi-site (di ordine 2), e le relative curve si distanziano di un rapporto pressoché costante. Ci aspettiamo che, al variare del numero di VIM, il comportamento porti ad un simile andamento, ma contestualmente ad un aumento del coefficiente “distanza” tra le due curve.

Questo comportamento dimostra che OSM è in grado di istanziare in modo dinamico lo stesso numero di vNSF su più nodi e questo, come ci si aspetta, si traduce in una diminuzione dei tempi di istanziazione. L’unico punto da affrontare, in questo caso, è come far comunicare le istanze su più nodi diversi. A questo però dà una risposta, se pur ancora teorica, l’utilizzo di un controller SDN in OSM e di tecnologie come SR-IOV, di cui discuteremo nel capitolo 9.

8.7.4 Scenario con l’introduzione di Juju e calcolo del suo overhead

Adesso andiamo a testare quale sia l’impatto di Juju sulle prestazioni del sistema complessivo. In particolare avevamo fatto nella parte teorica due considerazioni fondamentali:

- Ho la possibilità di utilizzare Juju anche come Day-0 configuration, ovvero aspettare l’istanziamento dei proxy charm per effettuare operazioni preliminari sulla vNSF (in pratica il servizio di sicurezza è online solo dopo tutto questo processo);
- Un approccio ibrido dove le Day-0 configuration sono fatte unicamente da cloud-init e le Day-1 configuration successive da Juju, rendendo di fatto il tempo di istanziazione dei proxy charm irrilevante (in pratica il servizio di sicurezza è online appena la cloud-init termina.)

Sulla base di questi due principi teorici, che abbiamo supposto nella parte teorica, andiamo a valutare quali sono i tempi di istanziazione dei charm di Juju e quale l’eventuale overhead, qualora

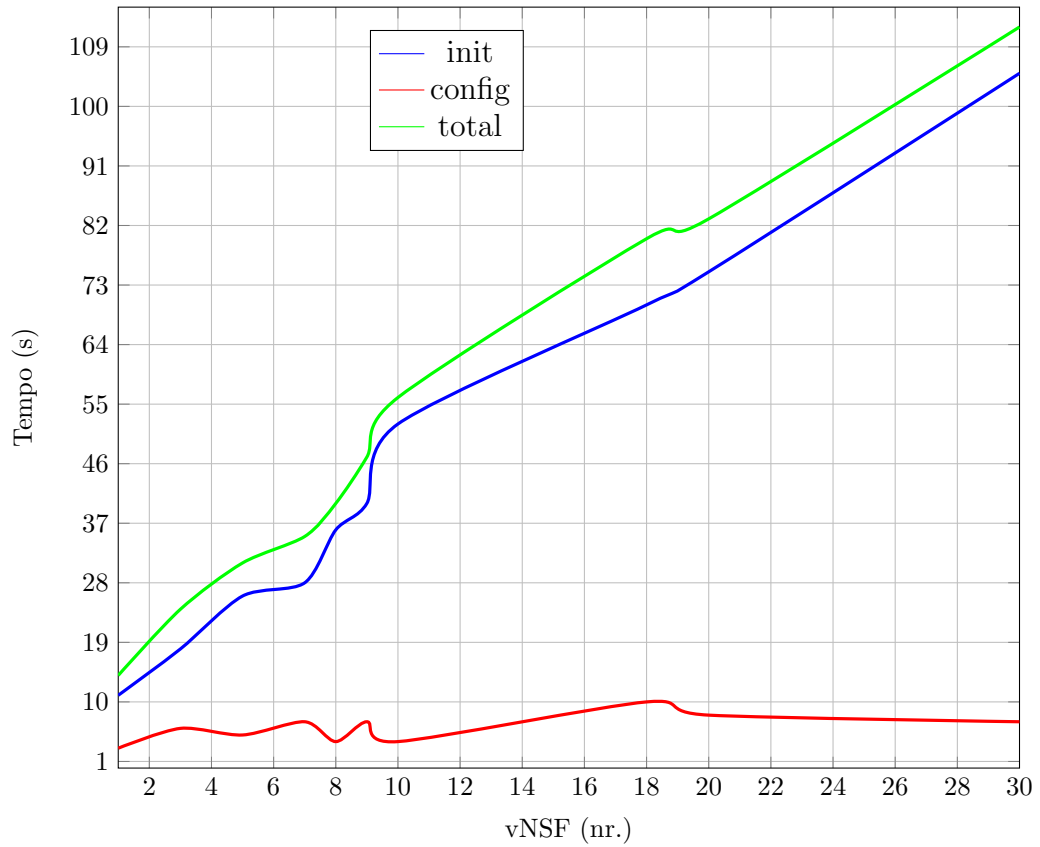


Figura 8.10: Tempo di inizializzazione e configurazione delle vNSF con tecnologia cloud-image.

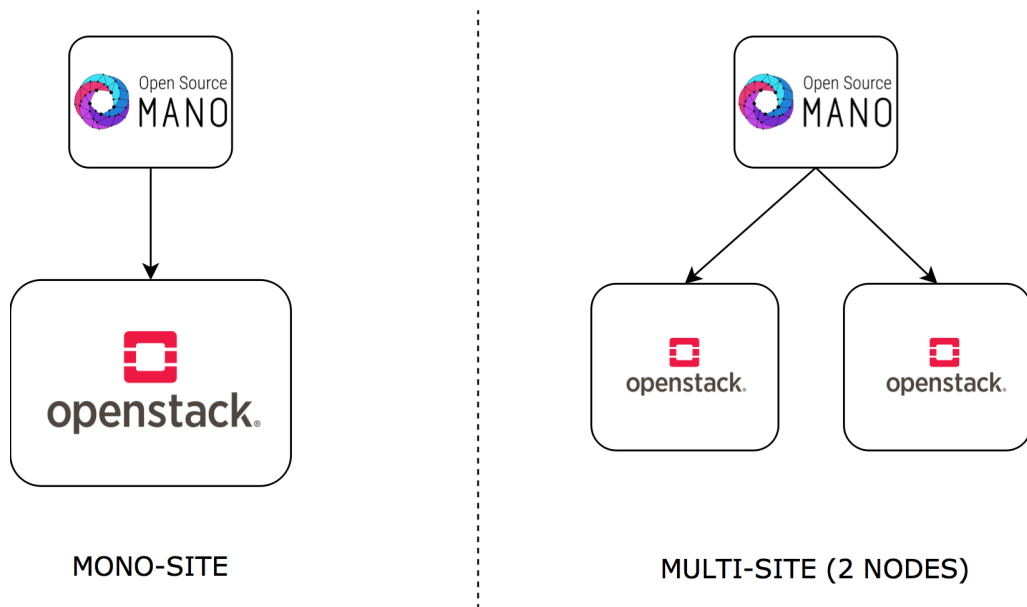


Figura 8.11: Mono-site vs multi-site.

fosse utilizzato anche come Day-0 configuration (figura 8.13). In tabella 8.4 sono riportati i tempi relativi ad alcune misurazioni campione per quanto riguarda l'istanziamento dei proxy charm. In questo caso non sono stati valutati i tempi di configurazione effettiva delle vNSF, perché risultano irrilevanti rispetto alle quantità che trattiamo nel caso di istanziazione di Juju e capiremo presto perché.

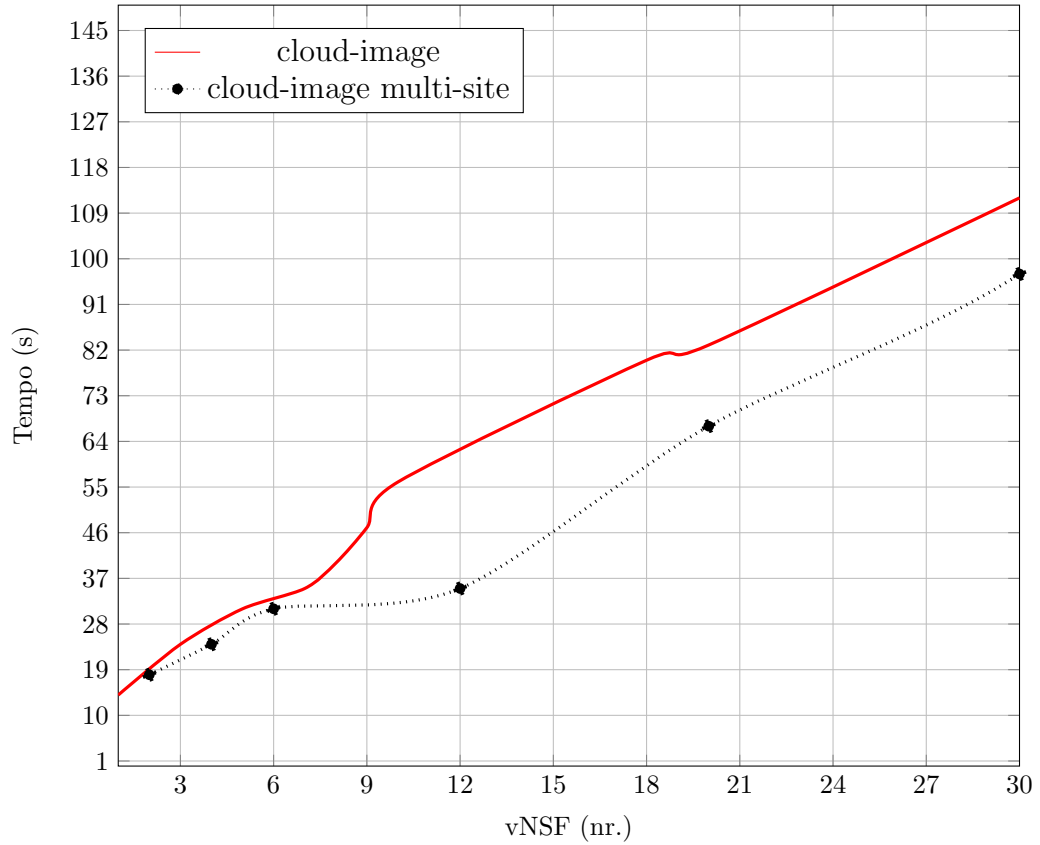


Figura 8.12: Tempi totali di inizializzazione dell’NSS con le tre tecnologie riportate, senza l’utilizzo di Juju.

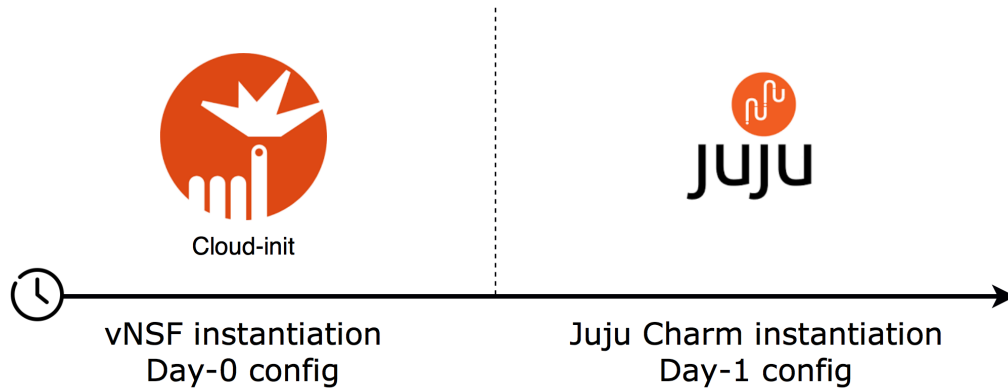


Figura 8.13: Utilizzo di juju ed impatto sul sistema.

Per valutare l’andamento e i numeri che sono riportati nella precedente tabella, guardiamo l’andamento della curva relativa a Juju rispetto alla cloud-image nell’immagine 8.14. Innanzitutto definiamo cosa sono le curve del grafico, iniziando con la blu tratteggiata che risulta essere la tempistica che impiega Juju a istanziare i proxy charm; abbiamo definito questa misura overhead in quanto è la crescita delle tempistiche che introdurrebbe Juju se fosse utilizzato anche per la Day-0 configuration. La curva rossa rappresenta l’insieme di tempistiche (T_{tot}) che serve per l’istanziamento con tecnologia cloud-image. Infine, la verde rappresenta la somma delle due tempistiche e quindi il tempo totale di istanziazione del servizio di sicurezza.

Ricordiamo che necessariamente c’è bisogno della fase relativa alla cloud-init quando utilizzo

Nr. istanze	$T_{tot}(s)$
1	125
3	152
7	248
12	446

Tabella 8.4: Tempi di istanziazione dei proxy charm di juju.

Juju con cloud-image, perché risulta necessario un minimo di configurazioni base per accedere l'istanza con ssh e una cloud-image necessita di tali informazioni (es. Key injection).

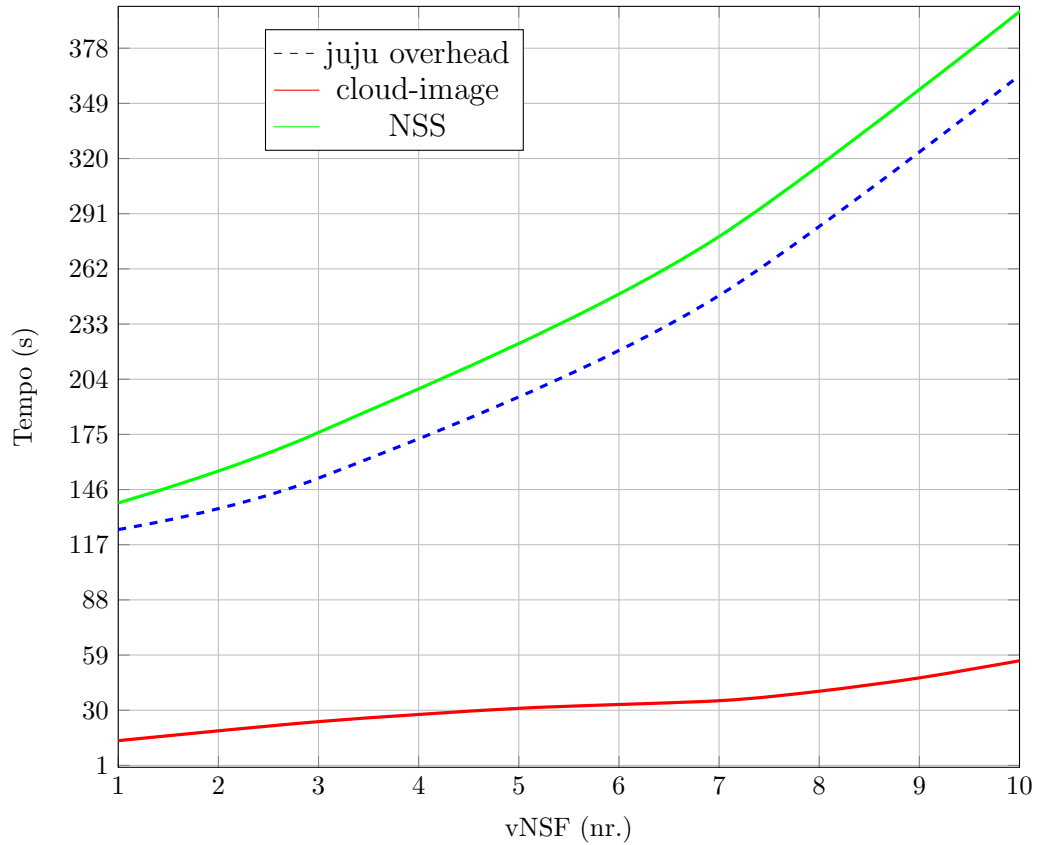


Figura 8.14: Overhead introdotto da Juju nell'istanziamento del servizio di rete.

Valutando ora il grafico a livello qualitativo quello che è evidente è il consistente overhead che Juju introduce e come esso diverga velocemente al crescere del numero delle istanze. Queste tempistiche sono dovute principalmente ai tempi di preparazione delle macchine LXD (ricordiamo che noi effettuiamo il deploy dei proxy charm su LXD container) e all'installazione degli agent, dei charm e dei software al loro interno. Diventa quindi davvero improponibile una soluzione in cui Juju sia utilizzato come Day-0 configuration; anzi l'unica cosa ragionevole sembra utilizzare Juju come Day-1 e oltretutto, per limitare lo spreco di risorse (istanziare un proxy charm/container LXD per ogni VM), utilizzarlo solo sulle vNSF che necessitano di essere riconfigurate con una certa frequenza.

Può essere utile fornire qualche ulteriore dettaglio sulle possibili ragioni del decadimento delle prestazioni di Juju. Innanzitutto le prestazioni a cui facciamo riferimento per Juju sono relative al framework di base di OSM. Questo significa che, per come previsto dall'installazione base di Open Source MANO, i proxy charm vengono istanziati in degli LXD container a loro volta istanziati nel container LXD VCA (il modulo di OSM che si occupa della configurazione). Questa virtualizzazione innestata porta ad un leggero overhead, ma non è la causa delle prestazioni insoddisfacenti, in termini di istanziazione, di Juju. Una soluzione a questa problematica sta nel distaccare Juju

in un nodo (macchina fisica) separato e collegarlo ad OSM, come è anche previsto dalle possibili configurazioni di Open Source MANO (ogni modulo volendo può essere separato in macchine diverse). In ogni caso però occorrerebbero prestazioni molto superiori in proporzione a quelle utilizzate dagli stessi compute node per le vNSF istanziate; questo è contro ogni logica, perché vorrebbe dire utilizzare più prestazioni per la configurazione che per le vNSF stesse.

La vera problematica risiede nello strumento Juju stesso e nell'ottimizzazione del processo di installazione e configurazione dei charm. Anche senza il meccanismo dei proxy charm, Juju tende ad avere tempistiche rilevanti nel deploy di bundles complessi come OpenStack o Kubernetes su tecnologia LXD. Una soluzione reale a questa problematica potrebbe essere cambiare tecnologia di virtualizzazione o cercare delle strade alternative per ridurre i tempi di configurazione (es. cambiare immagine software di base da ubuntu xenial a trusty). In ogni caso tale strumento è pensato per servizi complessi e durevoli, che difficilmente necessitano di tempi di istanziazione brevi, rendendo quindi poco appetibile un'ottimizzazione in tal senso.

Nonostante le due precedenti considerazioni esistono ancora due vantaggi nell'utilizzare Juju. Il primo è dato dall'irrelevanza dei tempi di istanziazione del proxy charm nel caso dell'approccio ibrido descritto in precedenza. In particolare, consentendo una rapida istanziazione del servizio di sicurezza attraverso cloud-init, è possibile utilizzare da subito tale servizio e nel frattempo procedere con il deploy dei proxy charm. A questo punto le tempistiche iniziali per l'NSS collaserebbero in figura 8.14 sulla linea rossa, risultando indistinguibili dal caso base di cloud-image. Allo stesso tempo configurando, in seguito, adeguatamente i proxy charm, non vi sarebbe necessità di istanziare nuovamente il servizio di sicurezza nel caso in cui alcune delle configurazioni dovessero cambiare nel tempo, andando a limitare così inutili riavvii del servizio. In quest'ottica risulta ancora interessante utilizzare Juju per tutta una serie di vNSF che necessitano spesso di riconfigurazioni (es. Firewall).

Un secondo vantaggio è sempre in linea con la capacità di Juju di effettuare configurazioni e task complessi con ogni tecnologia oggi a disposizione per tale fine. In pratica con Juju è vero che si introduce overhead (problema estinto in parte dalla precedente considerazione), ma ho in ogni caso la possibilità di configurare la vNSF con ogni meccanismo possibile: dallo sviluppo di script python e bash personalizzati, all'utilizzo di Chef, Puppets e Ansible con i relativi file di supporto. Questo è un vantaggio da non sottovalutare in un ecosistema così eterogeneo come quello dei fornitori di vNSF e degli strumenti utilizzati per la configurazione di istanze virtuali.

Capitolo 9

Sviluppi futuri

9.1 OSM vs container

Abbiamo visto nei capitoli precedenti (cap. 2 e 6) che la tecnologia utilizzata per la virtualizzazione è di fondamentale interesse per l'evoluzione futura di NFV. Abbiamo anche presentato alcune soluzioni VIM che prevedono l'integrazione di infrastrutture basate su container come OPNFV, OpenStack e il progetto Magnum, Kubernetes e altri ancora. Ognuno di questi strumenti è nato in modo specifico per gestire servizi nel mondo cloud, quindi applicarlo nel mondo NFV richiede ancora qualche accorgimento. Delle differenze sostanziali le presenteremo nella sezione 9.3, dove vedremo che NFV richiede alcune particolari tecnologie per poter effettivamente fornire le prestazioni desiderate.

Il focus attuale, però, è su come integrare una tecnologia di virtualizzazione, basata su container, in un ambiente per l'orchestrazione NFV. In modo particolare ci concentreremo su OSM e vedremo come sarebbe possibile effettuare l'istanziamento di funzioni di rete su container.

Una prima soluzione consiste nel lavorare a livello di Resource Orchestrator, infatti, è possibile gestire direttamente i container mappando le API di alcuni VIM specifici attraverso plug-in installati nell'RO. Nell'appendice B, si vede come poter integrare un plug-in per lavorare con VIM che non sono ancora compatibili con OSM. In modo open sono disponibili alcuni plug-in di prova che consentono ad esempio l'integrazione di Docker in OSM.

Una seconda soluzione per disporre di container, potrebbe essere quella di utilizzare in OpenStack lo strumento nova-lxd. Questo è un particolare plug-in per l'hypervisor LXD (nova-lxd), che consente di interfacciarsi con l'ambiente OpenStack ed istanziare container LXD, a dispetto delle tradizionali VM. Da quello che si evince dal capitolo 6, si vede come il passaggio da VM a container sia più morbido con tecnologia LXD rispetto a Docker. Il software conjure-up della Canonical, inoltre, potrebbe permettere il deploy di tutta una soluzione OpenStack (nova-lxd) in modo automatico, attraverso l'utilizzo di Juju. Questa soluzione data la capacità dei container LXD di esporre una porta SSH per connettersi da remoto in modo sicuro, consentirebbe anche una facile integrazione della nostra soluzione (capitoli 7 e 8); in particolare sarebbe compatibile la configurazione automatica attraverso Juju.

Una terza modalità potrebbe essere quella di integrare un sistema complesso di orchestrazione, ad esempio Apache Mesos/Marathon, in modo da poter gestire tutti gli aspetti dei container Docker. Questo si può ottenere effettuando sempre un mapping delle API di Marathon con le funzionalità dell'RO di OSM, o in alternativa, utilizzandolo direttamente come Resource Orchestrator in sostituzione del vecchio OpenMANO (questa modifica risulterebbe più invasiva).

L'ultima soluzione, soprattutto per sperimentare quali potrebbero essere le problematiche di configurazione automatica dei container rispetto alle VM, è data da VIM EMU. Questo è un emulatore che permette di simulare un'infrastruttura NFVI costituita da più PoP, con l'obiettivo di poter testare servizi di rete complessi in un ambiente controllato, prima della messa in opera in produzione o in scenari costosi di testing.

9.1.1 VIM EMU

Il progetto vim-emu di Open Source MANO nasce con l'obiettivo di supportare gli sviluppatori nel testare i propri servizi di rete in locale, prima di effettuare un deploy su larga scala. La tecnologia utilizzata per il suo sviluppo si basa su un progetto chiamato Sonata [19]. Quest'ultimo è un framework nato sulla base di containernet [41], un progetto che integrava le potenzialità di mininet e Docker. Containernet consente, attraverso delle API, di creare e gestire delle topologie di rete con mininet, predisponendo l'integrazione di un SDN controller per gestire il control plan. Ancora è possibile istanziare dei container Docker e collegarli alle topologie create. L'idea di containernet ha avuto un grande successo tra gli sviluppatori in ambito NFV, infatti, permette anche su piattaforme con prestazioni limitate (es. laptop) l'istituzione di servizi complessi. Sonata, a partire da questo framework, ha sviluppato un vero e proprio sistema per il testing di scenari complessi, integrando dapprima l'orchestratore Sonata del suo progetto e poi fornendo supporto ad Open Source MANO. Sonata ha riscosso altrettanto successo ed è citato in diversi recenti articoli, che presentano soluzioni per prototipizzare servizi di rete [42].

Prima di scendere nei dettagli su quali sono le funzionalità offerte da Sonata e vim-emu, andiamo a vedere come si colloca un emulatore nel framework dello standard ETSI. In figura 9.1, si vede come l'emulatore vim-emu si riferisce alla parte di NFVI e VIM, quindi simula sia l'infrastruttura che la parte di gestione della stessa. Questo comprende sia la parte di controllo delle istanze (Docker), che il controllo del data plane (con Pox o Ryu).

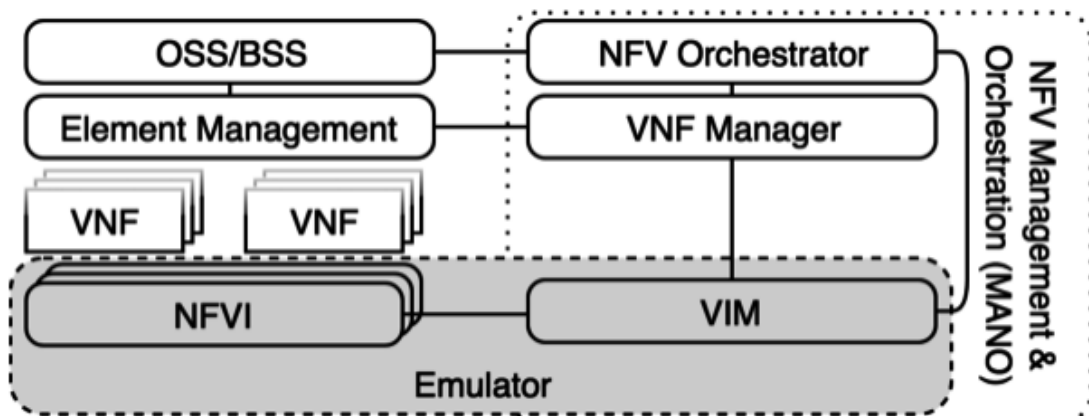


Figura 9.1: Mapping vim-emu sullo standard ETSI.(fonte: [osm wiki](#)).

Sonata e vim-emu sono praticamente due facce della stessa medaglia, infatti, offrono in pratica le stesse funzionalità. Si può dire, infatti, che vim-emu è semplicemente un “wrapper” di son-emu (emulatore del progetto sonata), con alcune modifiche per renderlo più facilmente integrabile con OSM. Andiamo ora a vedere quali sono le caratteristiche e funzionalità di vim-emu.

La figura 9.2 rappresenta l'architettura di vim-emu e due possibili sistemi di orchestrazione compatibili: Sonata e OSM. La prima cosa che si vede dalla figura è che la piattaforma emulata presenta diversi PoP, che simulano la distribuzione di diversi datacenter geograficamente sulla rete. Questi PoP vengono creati attraverso l'utilizzo di mininet, in particolare quest'ultima fornisce gli strumenti per la creazione di un insieme di vSwitch e di host virtuali; vim-emu ha esteso le sue API al fine di poter aggiungere e definire direttamente dei PoP. Questi PoP possono essere rappresentati nel caso più semplice da un singolo SDN vSwitch che fornisce connettività a tutti gli host (nel nostro caso container) del PoP. Le API messe a disposizione da vim-emu di OSM forniscono la capacità di creare automaticamente anche topologie più complesse del singolo vSwitch. Un'altra estensione alle API permette di gestire le istanze di computazione e quindi istanziare direttamente i container attraverso il Docker Engine e collegarli alle reti create.

Per collegare questa piattaforma al framework di orchestrazione (MANO) sono state definite tutta una serie di interfacce, che emulano il comportamento delle API di heat e degli altri servizi di OpenStack (nova, neutron, etc.) e consentono a tali sistemi di interfacciarsi con la piattaforma

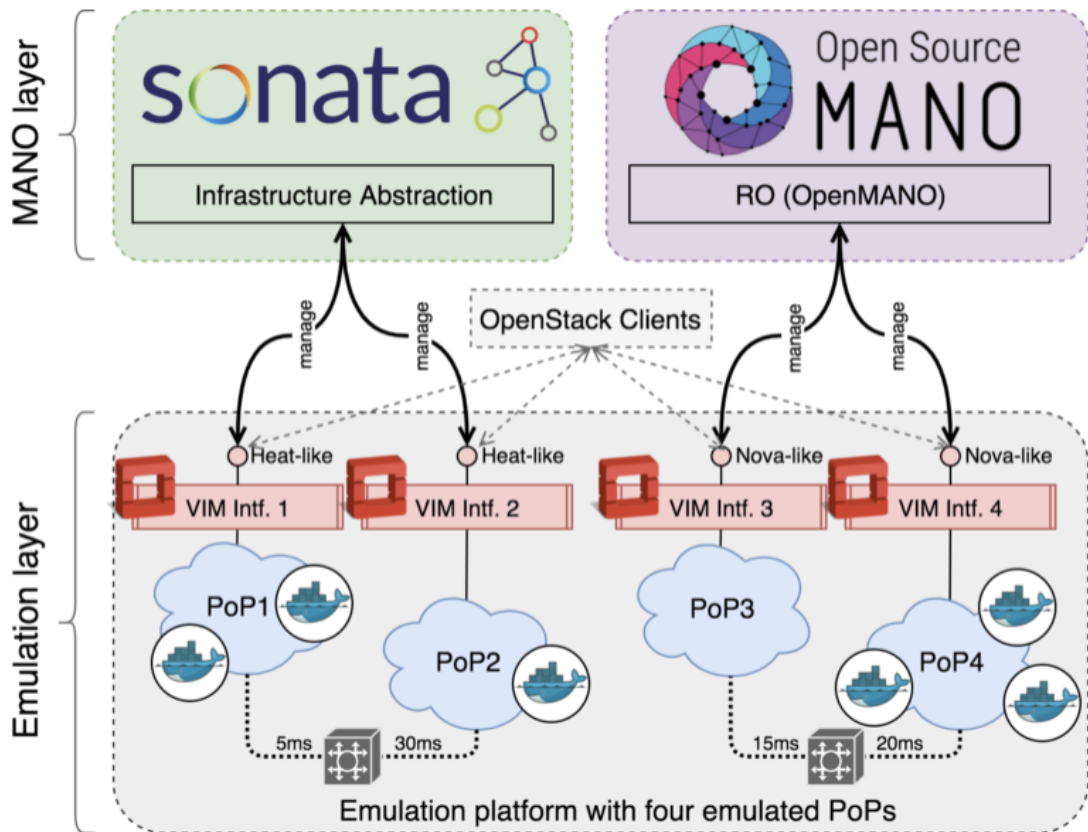


Figura 9.2: Architettura vim-emu.(fonte: [osm wiki](#)).

di emulazione. In particolare nel caso di OSM vedremo nell'appendice B, com'è possibile andare a collegare, come VIM, vim-emu e come è possibile orchestrare le risorse su questa piattaforma. Nel caso specifico vim-emu, al momento della sua installazione, crea un container Docker con la piattaforma di emulazione al suo interno e a partire da questa vengono istanziati gli altri container sullo stesso Docker Engine.

Il consentire, ai processi interni ad un container, di creare altri container dello stesso livello di quello dove vengono eseguiti è una scelta progettuale, in termini di sicurezza, poco accurata. Rientriamo nel caso di alcune vulnerabilità descritte nel capitolo 2, dove una volta fornito l'accesso alle API per la gestione dell'infrastruttura ad una delle istanze virtuali era possibile, in caso di compromissione della stessa, accedere direttamente all'infrastruttura di management e quindi potenzialmente di poter attaccare l'infrastruttura e le altre istanze. Nel caso di test del servizio e non di un ambiente di produzione, però, può essere una scelta che ha un senso; questo potrebbe servire a valutare solo le prestazioni di un sistema e la sua bontà progettuale. In ogni caso è possibile pensare a delle modifiche architetturali per ovviare a questa problematica e capire come poter sfruttare tale sistema in un ambiente diverso da quello di emulazione.

La motivazione, che ci spinge ad inserire questo framework negli sviluppi futuri, è la capacità di quest'ultimo di fornire un ambiente in grado di testare degli scenari interessanti, che rappresentano possibili evoluzioni della nostra soluzione. Ad esempio è possibile con vim-emu partire dagli stessi descrittori utilizzati per la gestione dei nostri servizi di sicurezza e, con delle opportune modifiche, valutare come uno stesso servizio di sicurezza può essere istanziato tramite Docker. Questo permetterebbe di esplorare, prima di concentrarsi su come progettare l'infrastruttura basata su container, la gestione del data plane, l'integrazione di Docker con OSM, tutti i possibili meccanismi di configurazione dei container e come sarebbe possibile concepire la nostra soluzione in tecnologia Docker. In più il poter integrare in modo "controllato", su un'infrastruttura limitata, la tecnologia SDN, potrebbe essere un ottimo punto di partenza per effettuare qualche test su grafi

complessi (composti da più Service Function Chain), che prevedono l'utilizzo di un SDN controller e hanno VNF composte da più di tre interfacce.

9.1.2 Prestazioni a confronto

Con l'emulatore descritto in precedenza abbiamo effettuato dei test sulle prestazioni, in termini di tempistiche di istanziazione del servizio di rete. Sebbene ci siano dei problemi di sicurezza nel framework e sia solo una piattaforma di emulazione, i container Docker vengono istanziati sul Docker Engine della macchina fisica. Questo ci dà un'idea realistica di quelli che possono essere i tempi di deploy di un servizio di rete basato su Docker.

Nr. istanze	$t_{init}(s)$	$t_{conf}(s)$	$T_{tot}(s)$
2	2	7	9
5	3	7	10
10	4	7	11
15	11	7	18
25	15	7	22
33	18	7	25

Tabella 9.1: Tempi di start-up di Docker.

Nella tabella 9.1, sono rappresentate le tempistiche di inizializzazione e configurazione attraverso OSM di servizi di rete composti dal numero di container indicato.

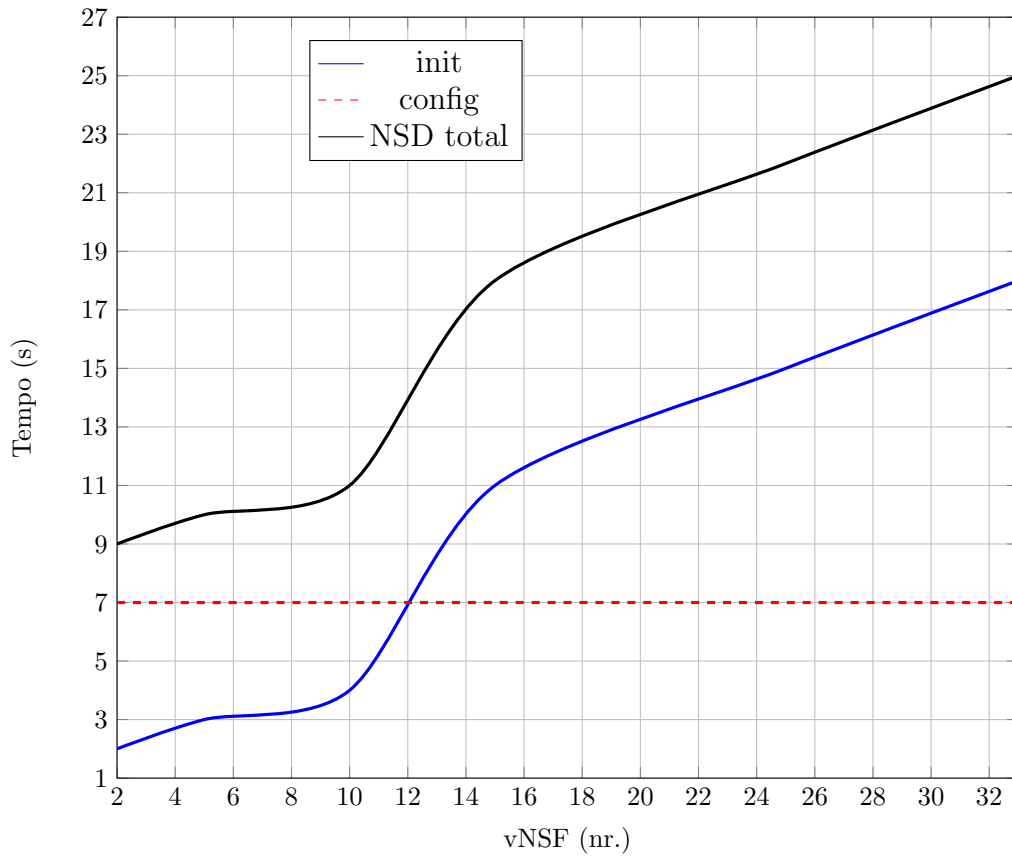


Figura 9.3: Tempo di inizializzazione e configurazione con tecnologia Docker.

In figura 9.3, è rappresentata una visione più chiara dei dati nella precedente tabella. Con la linea blu è rappresentato l'effettivo tempo di istanziazione dei container Docker a prescindere dalla configurazione di rete. La linea rossa tratteggiata rappresenta un tempo di start-up, pressoché

identico tra i vari test effettuati, che identifica il tempo di creazione dei Virtual Link necessari per la connettività dei container Docker. Questo tempo non varia perché la latenza è relativa soprattutto al tempo di traduzione delle chiamate API in stile OpenStack, dirette a vim-emu, in configurazioni per containernet e al relativo tempo di applicazione alla topologia. L'unica variabile del sistema dipende dalla quantità di container nel descrittore del servizio di rete. Abbiamo modellato direttamente con OSM diversi servizi di rete ed effettuato il deploy, ottenendo i dati in figura 9.3.

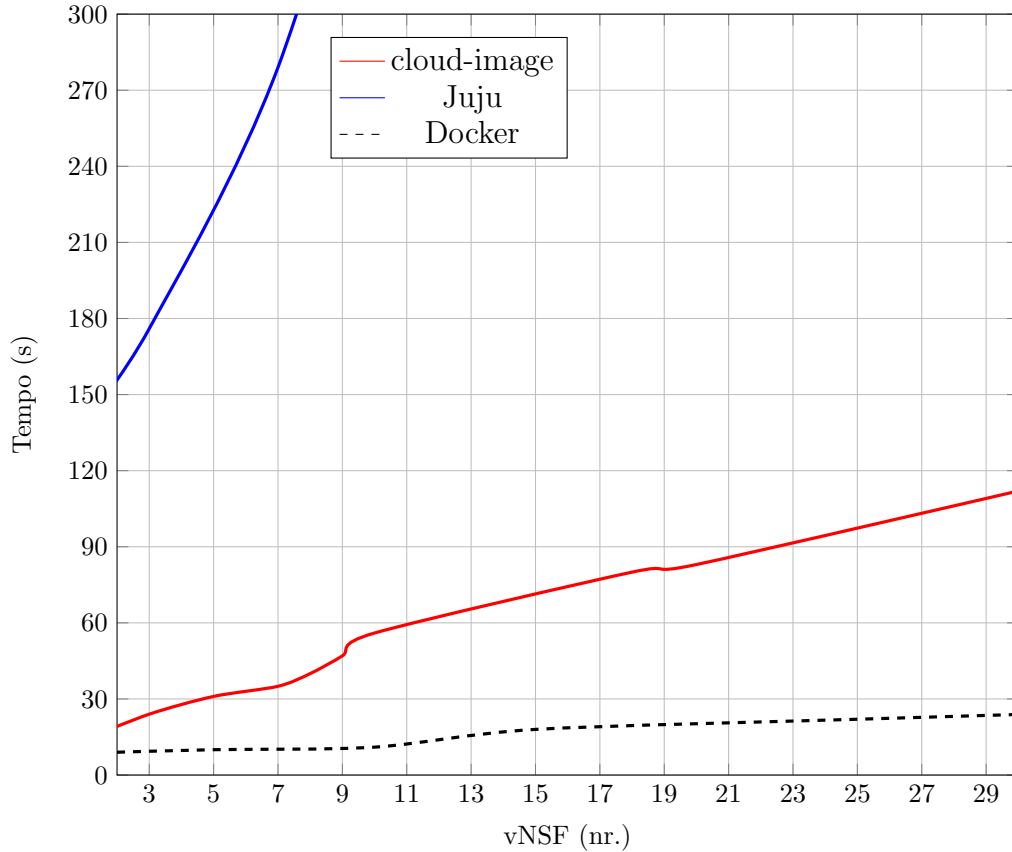


Figura 9.4: Confronto tra le tecnologie utilizzate per la soluzione implementativa e Docker.

Un'ultima analisi è relativa al confronto prestazionale tra le tecnologie utilizzate per la nostra soluzione e la soluzione basata su Docker. Come si vede dalla figura 9.4, a parità di numero di vNSF, l'utilizzo di Juju diverge rapidamente, ma questo non ci sorprende particolarmente; abbiamo compreso che il suo utilizzo si limita solo a configurazioni successive alla prima.

La cosa più interessante è il confronto prestazionale tra cloud-image/cloud-init e Docker: le soluzioni più performanti che abbiamo trovato. Vediamo, sempre in figura 9.4, che nonostante l'andamento al variare delle vNSF di cloud-image sia pressoché lineare, quello di Docker, tolto il periodo iniziale dovuto alle configurazioni di rete, risulta molto meno che lineare.

Questo risultato lascia spazio a numerose riflessioni sul futuro vantaggio nell'utilizzo di Docker rispetto alle VM e soprattutto sui meccanismi di configurazione per Docker. Avendo a disposizione delle prestazioni così elevate, nel caso Docker, sembra del tutto inutile l'utilizzo di meccanismi di Day-1 configuration o Day-2 configuration, rispettivamente inerenti alle vNSF e al servizio di sicurezza. Utilizzando Docker, infatti, la cosa più logica in caso di riconfigurazione risulta quella di terminare e ristabilire nuovamente il servizio, questo in termini di prestazioni risulta la scelta più intelligente. Inoltre, Docker con i Dockerfile mette a disposizione un meccanismo simile a quello di cloud-init per inizializzare le istanze e fornisce quindi la stessa flessibilità.

Il caso in cui potrebbe essere utile un approccio Day-1 o Day-2, nel caso di virtualizzazione su container, potrebbe essere nella prospettiva in cui VM e container vengano utilizzate in modo

ibrido. In particolare potrebbero essere gestiti vari livelli di virtualizzazione, per cui le tecnologie da noi descritte in fase implementativa (cloud image e Juju) forniscano un insieme di VM di base, in cui vengano poi eseguiti diversi sottoservizi di sicurezza attraverso la virtualizzazione tramite container. Come abbiamo visto, infatti, il concetto di rendere modulari anche le singole vNSF (o VNF) è alla base dello standard NFV di ETSI, questo mettendo in conto la presenza di diversi sottomoduli (VNFC) specializzati. In quest'ottica sarebbe possibile utilizzare cloud-init e Juju per la gestione dell'ambiente esterno (le vNSF su macchine virtuali) e Docker per i sottomoduli interni. Questo significherebbe una ancora maggiore flessibilità nella progettazione delle vNSF, perché, a quel punto, la riconfigurazione dei sottomoduli (VNFC) non andrebbe più fatta attraverso Juju o cloud-init per ognuno di questi, ma semplicemente attraverso Dockerfile in modo molto più rapido e dinamico (terminazione e reinizializzazione). Tutto questo mentre la vNSF esterna può essere gestita, invece, dai classici meccanismi (es. Juju) evitando quindi di essere nuovamente istanziata e fornendo al contempo più livelli di sicurezza.

9.2 Forwarding graph e SDN in OSM

Abbiamo discusso a più riprese della possibilità di integrare SDN in un ambiente NFV, sottolineando i relativi vantaggi. Un chiarimento sta nella considerazione che per implementare il concetto di Service Function Chaining, per come presentato nei capitoli precedenti (cap. 2 e 6), SDN è un tassello fondamentale. Nella nostra implementazione, infatti, abbiamo tenuto conto di un caso particolare di SFC, dove era presente una singola catena con un unico flusso di pacchetti. Questo ci ha permesso di implementare il tutto senza l'utilizzo di un SDN controller per gestire il dataplane; l'unica operazione necessaria è stata la configurazione automatica delle interfacce all'interno delle vNSF, al fine di raggiungere la successiva funzione di sicurezza.

Sembra chiaro che, in uno scenario più complesso, è possibile lavorare con più flussi di pacchetti e con grafi molto più complessi, costituiti magari da diverse catene. A questo punto andiamo a vedere quali sono i meccanismi che utilizza OSM per integrare SDN all'interno di NFV e come si possono sfruttare tali meccanismi per estendere la nostra soluzione al supporto di grafi più complessi.

9.2.1 Richiami al Virtual Network Functions Forwarding Graph in OSM

Abbiamo già accennato nei capitoli precedenti (cap. 2) al concetto di Virtual Network Functions Forwarding Graph in OSM, quello che facciamo ora è dare qualche dettaglio in più sugli strumenti di modellazione che OSM mette a disposizione per definirne uno.

Innanzitutto un VNFFG è completamente definito da:

- **Identification data:** VNFFGD ID, name, short name, vendor, description e version.
- **Rendered Service Paths (RSPs):** lista dei path che possono seguire i pacchetti attraverso le VNF.
- **Classifiers:** lista delle regole dei classificatori.

Gli RSP corrispondono al concetto di Service Function Chain e costituiscono l'insieme di VNF che definiscono uno specifico percorso. I Classifier permettono di definire delle regole di matching dei pacchetti che inoltrano il traffico attraverso uno di questi RSP.

Ogni singolo RSP in OSM è completamente definito da:

- **Identification Data:** ID, name.
- **VNFD Connection Point References:** una lista di riferimenti ai VNFD che costituiscono il path, specificando l'ordine di ognuna delle VNF nella catena.

Mentre gli elementi costitutivi di un Classifier sono:

- **Identification Data:** ID, name.
- **RSP ID:** riferimento al path associato al Classifier.
- **Member VNF Index Ref:** riferimento alla VNF specifica del path che costituisce il punto di ingresso nel path stesso.
- **VNFD connection point:** riferimento all'interfaccia specifica della VNF che costituisce il punto di ingresso.
- **Match attributes:** una lista di attributi che costituiscono le regole di inoltro dei pacchetti attraverso i diversi path. Nel caso specifico ad ogni RSP sono associati una serie di match attribute che indicano quali pacchetti devono passare attraverso quel dato path. Una serie di parametri vengono specificati per ognuno dei match attributes (protocollo IP, ip-porta sorgente, ip-porta destinazione).

9.2.2 Scenario di esempio multi PoP

L'idea finale che sta alla base del VNFFG è quella, in uno scenario come quello presentato in figura 9.5, di creare diversi flussi di pacchetti, a seconda di alcune regole definite dagli strumenti visti nella precedente sezione.

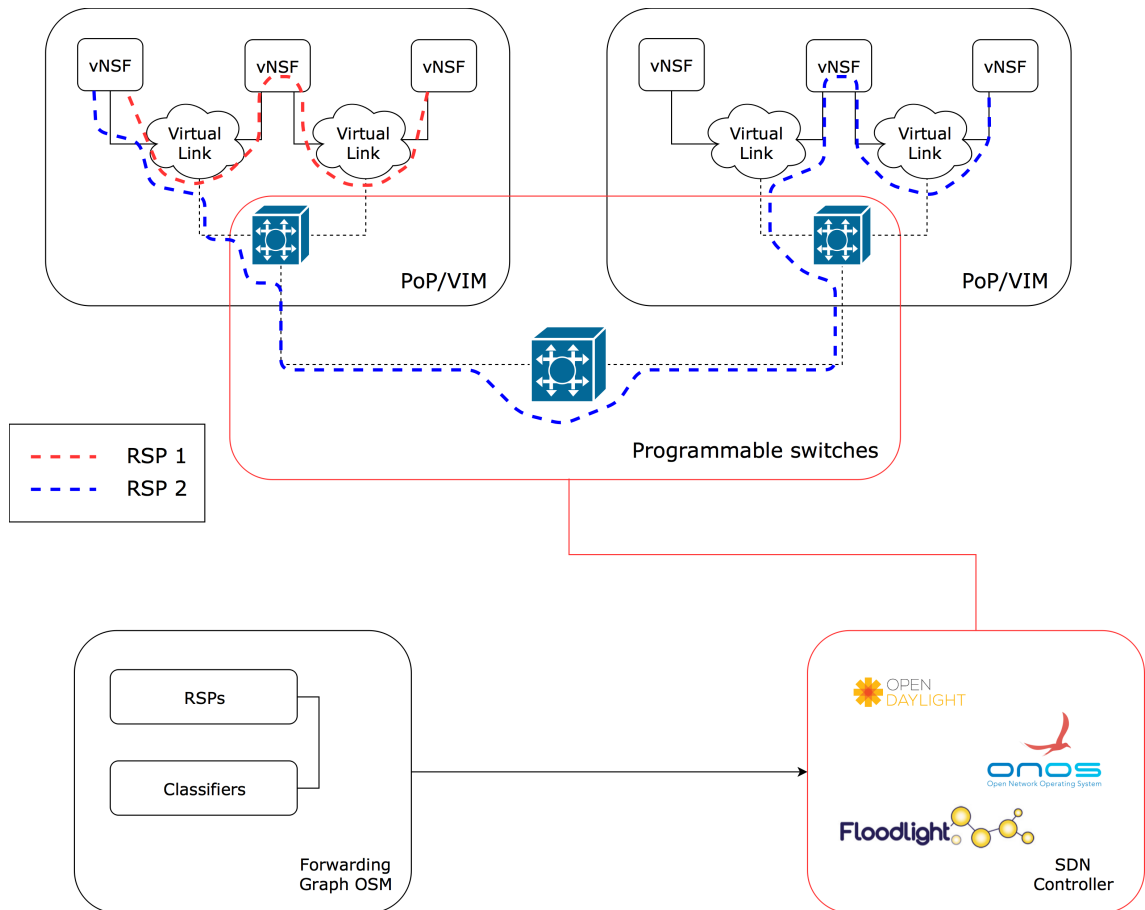


Figura 9.5: Esempio di forwarding graph applicato ad un scenario multi-PoP.

Tutto questo risulta possibile tramite l'integrazione di un controller SDN, all'interno dell'ambiente di Open Source MANO. Si parte dal presupposto che ogni PoP abbia la parte di networking implementata con degli switch programmabili e compatibili con uno dei protocolli SDN (es. OpenFlow); alcune tecnologie su singolo nodo le abbiamo citate in precedenza (es. Open vSwitch).

A partire dalle informazioni sulla collocazione delle VNF su uno specifico PoP (attraverso l'RO) e la definizione del grafo illustrato in precedenza, OSM è in grado di comunicare all'SDN controller le informazioni necessarie per modificare le Flow Table (nel caso di OpenFlow) degli switch e consentire ai pacchetti di essere inoltrati esattamente alle interfacce delle VNF desiderate. Questo, a tutti gli effetti, permette di disaccoppiare la parte di gestione del networking da quella di gestione delle singole VNF.

Per esemplificare meglio il concetto, si pensi ad un servizio di rete distribuito, come in figura 9.5, dove le VNF sono dislocate su più PoP. Il paradigma NFV viene utilizzato per istanziare e configurare le VNF su ogni singolo PoP, come abbiamo visto in precedenza. Questo include lo stabilire come trattare i pacchetti provenienti da un'interfaccia e su quali interfacce di uscita inoltrarli. Per quanto riguarda il resto, invece, ovvero tutta la parte di inoltro dei pacchetti da una funzione di rete ad un'altra, NFV abbandona quasi completamente la scena e si affida al controller SDN che, attraverso le informazioni presenti nel VNFFG, verifica il matching di alcuni pacchetti con le regole prestabilite e fa attraversare a questi ultimi le VNF interessate.

In pratica, sempre in figura 9.5, vediamo l'esempio di definizione di due diversi path percorribili dai pacchetti e corredati da alcune regole di matching per ogni singolo path. L'RSP 1 stabilisce che alcuni pacchetti devono attraversare le vNSF (nel caso di servizio di sicurezza) presenti sul PoP a sinistra internamente allo stesso PoP. L'RSP 2 invece definisce un percorso attraverso più PoP coordinato da un insieme più ampio di switch programmabili.

Il concetto finale è che attraverso l'integrazione dell'SDN controller e la sua configurazione opportuna, è possibile direttamente dal data model di OSM (descrittori), stabilire quali VNF e relative interfacce attraversare.

Risulta quindi di interesse provare ad integrare in modo concreto un SDN controller in OSM e valutare, come lavoro futuro, la possibilità di estendere le funzionalità della soluzione ai grafi complessi, ovvero costituiti da più SFC.

9.3 Ottimizzazione delle prestazioni: SR-IOV, DPDK ed EPA

L'ultima considerazione, per quanto riguarda gli sviluppi futuri, è relativa alle prestazioni di I/O e a delle nuove tecnologie che permettono di ottimizzare le attuali soluzioni NFV. Descriveremo di seguito brevemente tre tra queste soluzioni.

Interfacce SR-IOV

In ambito NFV, l'insieme delle caratteristiche più sensibili per quanto riguarda le prestazioni è costituito da quelle di I/O. Una delle problematiche, quindi, che trova molto riscontro nell'ambito della ricerca, anche a livello aziendale, è come ottimizzare tali prestazioni per avvicinarsi a quelle dei sistemi non virtualizzati.

Tipicamente, come abbiamo visto, sono a disposizione tante e svariate soluzioni di virtualizzazione, che permettono l'esecuzione di più istanze virtuali sulla medesima macchina fisica. Ognuna di queste istanze ha necessità di connettività anche verso l'esterno e questo implica il dover emulare via software l'adattatore di rete, condividendo di fatto il NIC (Network Interface Controller) fisico della macchina tra le varie istanze. Questo implica che il VMM o hypervisor, che gestisce la virtualizzazione, prenda tutte le decisioni relative alla destinazione ed inoltro dei pacchetti attraverso le macchine virtuali, creando un collo di bottiglia per le prestazioni e riducendo la densità della virtualizzazione su singolo server per poter bilanciare il carico delle prestazioni di I/O.

Quando arriva un pacchetto al NIC, tipicamente, viene generato un interrupt che viene a sua volta servito da un core della CPU. Nelle prime tecnologie Intel questo veniva servito sempre da un solo core dedicato e successivamente inoltrato all'apposita VM, rendendo di fatto impossibile raggiungere prestazioni vicine ai 10 Gbps di banda. Per ovviare a questo, la tecnologia Intel VMDq introdusse il concetto di code diversificate per ognuna delle VM istanziate su quel processore;

questo forniva un meccanismo di gran lunga più performante, che consentiva di generare interrupt differenti e specifici per ognuno dei core dove veniva eseguita la macchina virtuale.

Quello che restava un collo di bottiglia era ovviamente la gestione dell'inoltro del pacchetto, sempre a carico del VMM. Intel con un'altra tecnologia, la VT-d, ha creato la possibilità di bypassare il VMM e definire dei driver specifici per la singola VM che permettessero la comunicazione diretta con il NIC. Questo offriva la possibilità, qualora fossero presenti più NIC, di assegnare ogni NIC ad una specifica VM senza dover passare per l'hypervisor.

Ancora il problema era stato risolto parzialmente, in quanto ovviamente il numero delle VM supera sempre di gran lunga quello dei NIC fisici.

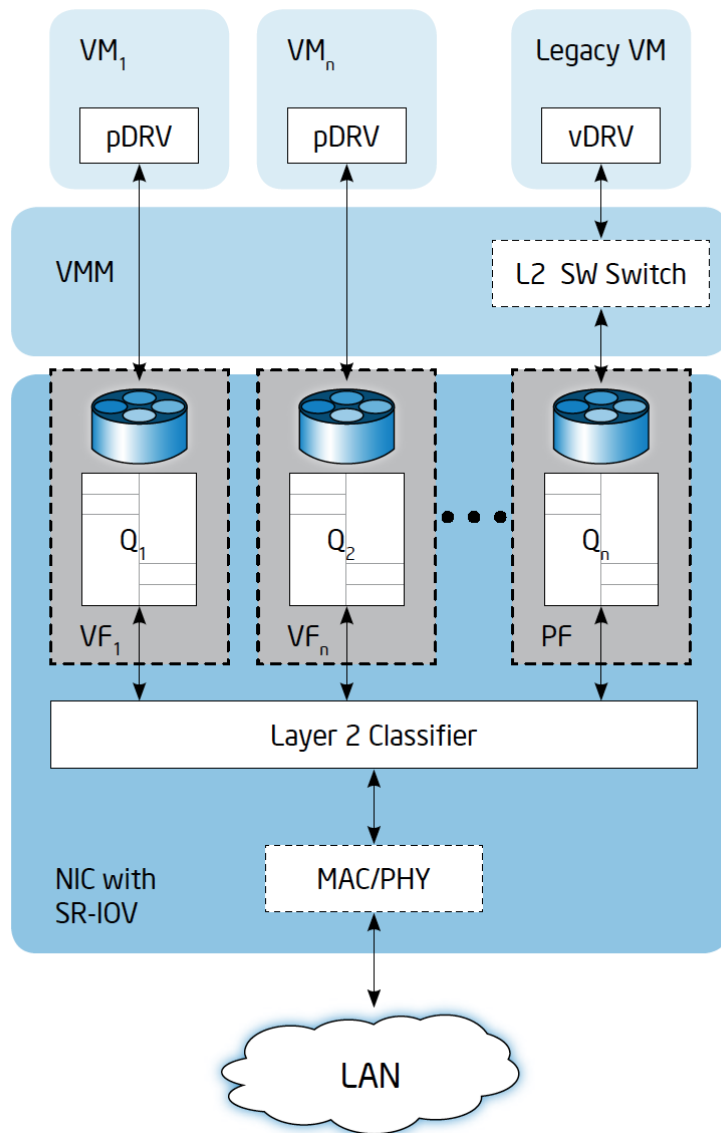


Figura 9.6: Scenario con utilizzo di SR-IOV.

A questo punto è stata introdotta la tecnologia Single Root I/O Virtualization (SR-IOV). Tale tecnologia permette ad un dispositivo, in questo caso un adattatore di rete, di condividere una risorsa senza l'emulazione software descritta in precedenza. In pratica un NIC che supporta tale tecnologia può, come descritto in figura 9.6, virtualizzare direttamente sul NIC a basso livello le funzioni necessarie per gestire l'I/O di una VNF. Ognuna delle funzioni virtuali (VF) si comporta come un fosse un vNIC con un MAC address assegnato direttamente dal NIC fisico. Ognuno di questi vNIC può, con la tecnologia Vd-T, descritta in precedenza, interfacciarsi attraverso degli

appositi driver con le singole VNF. Questo meccanismo permette di ottimizzare in modo sensibile le prestazioni di I/O delle VNF, rendendo possibile l'avvicinarsi di NFV a prestazioni simili a dispositivi hardware/software specifici e proprietari.

La Physical Function (PF), presente in figura 9.6, viene utilizzata per interfacciare il VMM con il NIC fisico e gestire le fasi iniziali di configurazione delle VM.

Non ci concentriamo su ogni dettaglio delle tecnologie presentate, perché esula dallo scopo di questo lavoro, ma identifichiamo future ricerche su queste tecnologie come un chiaro sviluppo futuro per rendere fattibile concretamente l'utilizzo delle tecnologie NFV, anche nel caso di funzioni di sicurezza su larga scala.

DPDK

Il Data Plane Development Kit (DPDK) [43] è un insieme di librerie e driver per l'elaborazione veloce dei pacchetti. Lo scopo principale è quello di spostare l'elaborazione dei pacchetti dallo spazio kernel a quello utente, bypassando di fatto lo stack di rete presente all'interno kernel linux. Questo introduce un'ottimizzazione in termini di prestazione soprattutto relative al tasso di trasferimento dei pacchetti. Alcuni test effettuati da Intel [44] mostrano che l'utilizzo di dpdk integrato nella tecnologia OVS, che alla base dell'implementazione del dataplane di OpenStack nel nostro caso, nel trasferimento di pacchetti tra due macchine virtuali che montano Ubuntu 17.04 migliora di quasi 2.5 volte.

Utilizzare questa tecnologia potrebbe migliorare di gran lunga le prestazioni nel trasferimento dati, anche quando si utilizzano tecnologie di virtualizzazione basate su container.

EPA

L'Enhanced Platform Awareness (EPA), rappresenta una nuova tecnologia promossa da Intel che consente alle piattaforme (intese come dispositivi hardware) di esporre alcune caratteristiche agli orchestratori e gestori di tale piattaforma. In sostanza i sistemi di orchestrazione in ambito NFV, tramite questa tecnologia, possono capire quali sono le caratteristiche hardware a disposizione su determinati nodi di rete, così da sfruttarle per gestire e distribuire in modo ottimale le VNF attraverso tali nodi. Alcuni dei VIM che supportano questo tipo di tecnologia sono OpenStack e Kubernetes, inoltre, anche sistemi NFV-MANO come Open Source MANO offrono tale supporto.

Tra le varie caratteristiche che i nodi fisici possono esporre ci sono:

- supporto alle Huge Pages;
- l'architettura NUMA a disposizione;
- CPU Pinning;
- integrazione di OVS con DPDK;
- supporto ad interfacce I/O Pass-through attraverso SR-IOV.

Anche questo concetto merita attenzione per gli sviluppi futuri, perché permette di gestire in modo efficiente l'allocazione delle risorse su nodi distribuiti ed eterogenei tra loro.

Capitolo 10

Conclusioni

Il lavoro svolto nell'ambito della presente tesi ha richiesto un rigoroso impegno, per poter perseguire gli obiettivi descritti all'inizio. Nonostante la complessità, però, è risultato un percorso molto fruttuoso da un punto di vista dei risultati.

Uno dei principali obiettivi era quello di investigare quali fossero le attuali tecnologie in ambito NFV per l'orchestrazione di funzioni di rete virtuali. Tale obiettivo è stato perseguito con successo e in merito sono stati presentati diversi strumenti, che rappresentano le ultime novità nell'ambito della gestione ed orchestrazione in NFV, uno tra tutti Open Source MANO.

La ricerca non si è fermata ad aspetti teorici, ma è stata resa concreta dalla progettazione di un sistema per la gestione di funzioni virtuali di sicurezza. In tale sistema sono state integrati tutti gli sforzi e le tecnologie studiate durante il corso della tesi. Per rendere ancora più tangibile il risultato, poi, sono state implementate le parti decisive di tale sistema per poter effettuare dei test che confermassero la bontà progettuale della soluzione.

Alla luce dei test eseguiti, possiamo affermare che la soluzione che abbiamo progettato è in linea con le nostre idee iniziali e si presta fattivamente ad un'implementazione reale. Ancora, i test ci hanno permesso di migliorare ed ottimizzare alcuni aspetti della soluzione che all'inizio non erano stati considerati. Un esempio tra tutti è l'utilizzo di Juju come prima scelta nella configurazione delle funzioni di rete. Tale scelta è stata parzialmente modificata e integrata con soluzioni che consentissero un notevole miglioramento delle prestazioni dell'implementazione finale (es. cloud-image/cloud-init).

Proprio questo aspetto ha dato spunto alla ricerca e allo studio di numerosi strumenti per la configurazione automatica delle funzioni virtuali di rete. Abbiamo con consapevolezza, quindi, delineato quali fossero le soluzioni attuali e quali le caratteristiche dominanti che facessero propendere per l'una o per l'altra scelta. Questo è stato un elemento di notevole interesse nell'ambito di questo lavoro.

Un altro focus di questa tesi è stato l'investigare quale potesse essere il seguito dell'orchestrazione in NFV. Ci si è dunque interrogati su quali fossero i plausibili scenari futuri per NFV-MANO e quali le tecnologie adottate nel prossimo futuro. Abbiamo delineato attraverso dei test, la possibilità di un notevole incremento di prestazioni attraverso l'utilizzo della tecnologia Docker come base per la virtualizzazione delle vNSF. Questo lascia numerosi spunti per il futuro e richiede ancora molte indagini per trovare soluzioni innovative nell'ambito della configurazione automatica di vNSF basate su container.

Un ultimo aspetto riguarda le prestazioni di NFV e il concetto di Service Function Chaining. Le prestazioni di NFV sono da sempre il punto centrale di ogni ricerca in quest'ambito e l'affrontare, anche se in modo parziale, quest'aspetto in questo lavoro costituisce un grande valore aggiunto. In questi termini si è presentato il concetto di dpdk e SR-IOV, soluzioni tecnologiche che diminuiscono notevolmente la latenza di I/O e consentono di avvicinarci alla prospettiva di NFV multi-site, con vNFS che comunicano ad elevate prestazioni (carrier-grade) tra di loro. Per quanto riguarda il Concetto di Service Function Chaining, invece, esso è stato presentato e discusso lungo tutto il

corso dell'elaborato. Senza di esso non esiste il concetto di servizio di rete e con esso tutte le basi su cui si poggia questo lavoro di tesi. In merito a quest'ultimo aspetto, sono state fatte numerose considerazioni su come integrare il paradigma SDN in NFV, al fine di consentire un più realistico scenario di comunicazione su più nodi distribuiti e dare supporto al deploy di servizi modellati con forwarding graph complessi. Il presente lavoro di tesi sottolinea che SDN non è un accessorio, ma è una vera e propria esigenza per ottenere la flessibilità delle reti necessaria ai futuri sistemi NFV-MANO.

Aspetti generali di NFV a parte, in conclusione questo lavoro ha cercato di dare delle linee guida su cui poggiare l'evoluzione dello stesso. Questo considerando degli aspetti, anche di ordine più pratico, su come poter semplificare l'integrazione dei servizi necessari per il funzionamento di NFV-MANO e delle sue estensioni. In ultima analisi, la struttura modulare e l'organizzazione su diversi livelli della soluzione proposta, coadiuvano questa volontà e aprono delle porte a numerose prospettive per lavori futuri.

Bibliografia

- [1] IBM Security, “IBM X-Force Threat Intelligence Index 2017”, March 2017
- [2] European Telecommunications Standards Institute, “Network Functions Virtualisation (NFV); Architectural Framework”, December 2014, http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf
- [3] European Telecommunications Standards Institute, “Network Functions Virtualisation (NFV); Virtual Network Functions Architecture”, December 2014, http://www.etsi.org/deliver/etsi_gs/NFV-SWA/001_099/001/01.01.01_60/gs_NFV-SWA001v010101p.pdf
- [4] J.Halpern, C.Pignataro, “Service Function Chaining (SFC) Architecture”, RFC-7665, October 2015, DOI [10.17487/RFC7665](https://doi.org/10.17487/RFC7665)
- [5] European Telecommunications Standards Institute, “Network Functions Virtualisation (NFV); Virtualisation Requirements”, October 2013, http://www.etsi.org/deliver/etsi_gs/NFV/001_099/004/01.01.01_60/gs_NFV004v010101p.pdf
- [6] European Telecommunications Standards Institute, “Network Functions Virtualisation (NFV); Infrastructure Overview”, January 2015, http://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_nfv-inf001v010101p.pdf
- [7] NCC Group, “Understanding and Hardening Linux Containers”, June 2016, <https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/>
- [8] National Institute of Standards and Technology, “The NIST Definition of Cloud Computing”, September 2011, <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>
- [9] S.Lal, T.Taleb, A.Dutta, “NFV: Security Threats and Best Practices”, IEEE Communications Magazine, Vol. 55, August 2017, pp. 211-217, DOI [10.1109/MCOM.2017.1600899](https://doi.org/10.1109/MCOM.2017.1600899)
- [10] Common Vulnerabilities and Exposures, CVE-2017-4934, December 2016, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-4934>
- [11] R.Mijumbi, J.Serrat, J.Gorricho, N.Bouten, F.De Turck, R.Boutaba, “Network Function Virtualization: State-of-the-Art and Research Challenges”, IEEE Communications Surveys & Tutorials, Vol. 18, No. 1, September 2015, pp. 236-262, DOI [10.1109/COMST.2015.2477041](https://doi.org/10.1109/COMST.2015.2477041)
- [12] European Telecommunications Standards Institute, “Network Functions Virtualisation (NFV); NFV Performance & Portability Best Practises”, June 2014, http://www.etsi.org/deliver/etsi_gs/NFV-PER/001_099/001/01.01.01_60/gs_nfv-per001v010101p.pdf
- [13] S.Mehraghdam, M.Keller, H.Karl, “Specifying and placing chains of virtual network functions”, 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet), Luxembourg (Luxembourg), Oct 8-10, 2014, pp. 7-13, DOI [10.1109/CloudNet.2014.6968961](https://doi.org/10.1109/CloudNet.2014.6968961)
- [14] A.Verma, “Large-scale cluster management at google with borg”, 10th EuroSys, Bordeaux (France), April 2015, pp. 1-17, DOI [10.1145/2741948.2741964](https://doi.org/10.1145/2741948.2741964)
- [15] European Telecommunications Standards Institute, “Network Functions Virtualisation (NFV); Management and Orchestration”, December 2014, http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf
- [16] R.Enns, M.Bjorklund, J.Schoenwaelder, A.Bierman, “Network Configuration Protocol (NETCONF)”, RFC-6241, June 2011, DOI [10.17487/RFC6241](https://doi.org/10.17487/RFC6241)
- [17] RO Northbound Interface, https://osm.etsi.org/wikipub/index.php/RO_Northbound_Interface
- [18] Apache Kafka, <https://kafka.apache.org>
- [19] Sonata project, <https://github.com/sonata-nfv/son-emu>
- [20] Open Baton, <https://openbaton.github.io/documentation/>

- [21] Industrial Technology Research Institute, Network Function Virtualization over Open DC/OS, https://www.communications.org.tw/news/item/download/191_c4756c757cc7c5eeba3a1b11dd241106.html
- [22] Apache Mesos, <http://mesos.apache.org/documentation/latest/architecture/>
- [23] Mesosphere Marathon, <https://github.com/mesosphere/marathon/tree/v1.4.0>
- [24] F.Valenza, "Modelling and Analysis of Network Security Policies", pp. 9-11, DOI [10.6092/polito/porto/2676486](https://doi.org/10.6092/polito/porto/2676486)
- [25] D.D.Clark, D.R.Wilson, "A Comparison of Commercial and Military Computer Security Policies", 1987 IEEE Symposium on Security and Privacy, Oakland (CA, USA), April 27-29, 1987, pp. 184-195, DOI [10.1109/SP.1987.10001](https://doi.org/10.1109/SP.1987.10001)
- [26] D.C.Robinson, M.S.Sloman, "Domains: a new approach to distributed system management", Workshop on the Future Trends of Distributed Computing Systems in the 1990s, Hong Kong (China), Sept. 14-16, 1988, pp. 154-163, DOI [10.1109/FTDCS.1988.26694](https://doi.org/10.1109/FTDCS.1988.26694)
- [27] A.Westerinen, J.Schnizlein, J.Strassner, M.Scherling, B.Quinn, S.Herzog, A.Huynh, M.Carlson, J.Perry, S.Waldbusser, "Terminology for Policy-Based Management", RFC-3198, November 2001, DOI [10.17487/RFC3198](https://doi.org/10.17487/RFC3198)
- [28] R.Yavatkar, D.Pendarakis, R.Guerin, "A Framework for Policy-based Admission Control", RFC-2753, January 2000, DOI [10.17487/RFC2753](https://doi.org/10.17487/RFC2753)
- [29] D.Steven, B.Jennings, J.Strassner, "The policy continuum: a formal model", Proc. of the 2nd IEEE International Workshop on Modelling Autonomic Communications Environments (MACE 2007), San José (CA, USA), October 2007, pp. 65-79
- [30] C.A.Astudillo, A.M.Gustin, O.J.Calderón, "Policy Creation Model for Policy-Based Management in Telecommunications Networks", 2010 IEEE Latin-American Conference on Communications (LATINCOM), Bogota (Colombia), September 15-17, 2010, <https://arxiv.org/abs/1108.0716>
- [31] D.C.Verma, "Simplifying network administration using policy-based management", IEEE Network, Vol. 16, No. 2, March-April 2002, pp. 20-26, DOI [10.1109/65.993219](https://doi.org/10.1109/65.993219)
- [32] SECURED consortium, "D4.1 - Policy specification", March 2015, https://www.secured-fp7.eu/files/secured_d41_policy_spec_v0100.pdf
- [33] SECURED consortium, "D4.2 - Policy transformation and optimization techniques", September 2015, https://www.secured-fp7.eu/files/secured_d42_policy_refinement_v0103.pdf
- [34] The TOR project, <https://www.torproject.org>
- [35] SECURED consortium, "D4.3 - Policy analysis and reconciliation", March 2016
- [36] Heat OpenStack Documentation, https://docs.openstack.org/heat/latest/template_guide/hot_spec.html
- [37] Modello YANG Open Source MANO, <https://osm.etsi.org/gitweb/?p=osm/IM.git;a=tree;f=models;h=ae728f6a12f850833dcf8d01b0192e843e48cdea;hb=HEAD>
- [38] OSM Information Model, https://osm.etsi.org/wikipub/index.php/OSM_Information_Model
- [39] Cloud-init documentation, <http://cloudinit.readthedocs.io/en/latest/topics/examples.html>
- [40] Canonical Ltd., "For CTO's: the no-nonsense way to accelerate your business with containers", February 2017, https://pages.ubuntu.com/rs/066-E0V-335/images/WP_The_no-nonsense-way-to-accelerate-your-business-with_containers.pdf
- [41] Containernet, <https://github.com/containernet/containernet>
- [42] M.Peuster, H.Karl, S.V.Rossem, "MeDICINE: Rapid Prototyping of Production-Ready Network Services in Multi-PoP Environments", IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Palo Alto (CA, USA), Nov 7-10, 2016, pp. 148-153, DOI [10.1109/NFV-SDN.2016.7919490](https://doi.org/10.1109/NFV-SDN.2016.7919490)
- [43] DPDK, <http://dpdk.org>
- [44] Intel, "Configure Open vSwitch with Data Plane Development Kit on Ubuntu Server 17.04", <https://software.intel.com/en-us/articles/set-up-open-vswitch-with-dpdk-on-ubuntu-server>

Elenco Appendici ed allegati

Tutti i dettagli tecnici, citati all'interno di questo lavoro di tesi, vengono documentati e dettagliati attraverso appendici specifiche e documentazione ulteriore allegata; il tutto secondo la logica riportata di seguito:

- **Appendice A:** manuale dell'utente: soluzione proposta (ref. [A](#)).
- **Appendice B:** manuale dello sviluppatore: Open Source MANO (ref. [B](#)).
- **Appendice C:** manuale dello sviluppatore: User Policy Repository (ref. [C](#)).
- **Appendice D:** manuale dello sviluppatore: OpenStack (ref. [D](#)).
- **Ulteriore materiale allegato:** codice e script utili all'installazione e configurazione della soluzione; una guida puntuale su come installare la soluzione e gli scenari presentati; ulteriori materiali di supporto ed approfondimento delle tematiche trattate.

Appendice A

Manuale utente: soluzione proposta

Abbiamo visto nei capitoli di teoria (cap. 7 e 8) com'è stata progettata la soluzione e come è stato predisposto il Proof-Of-Concept della stessa. Nelle appendici successive vedremo come singolarmente ogni componente è stato installato, configurato e quali sono le metodologie per lo sviluppo di alcune parti del codice. Scopo di questa sezione, invece, è quello di dare un'idea complessiva del funzionamento della soluzione e quali sono le funzionalità a disposizione dell'utente nell'attuale versione.

Per avere idea di come interagire con la soluzione descriviamo le tre fondamentali interfacce che sono a disposizione a tal fine:

- NSS Deployer (appartenente al Security Service Controller);
- Dashboard di Open Source MANO;
- Dashboard di OpenStack;
- Dashboard di Juju.

A.1 NSS Deployer

Questo risulta essere il gestore di alto livello del deploy del servizio di rete. In particolare questo modulo è concepito come una CLI, sotto forma di script bash, in grado di effettuare diverse operazioni (eventualmente estendibili con estrema facilità), attraverso i seguenti comandi:

- **deploy-sec-service**: questo comando permette di effettuare il deploy di un servizio di sicurezza a partire da uno specifico utente. I parametri che accetta sono `<user_name>` `<nss_name>`; il primo corrisponde al nome dell'utente, le cui policy vogliono essere utilizzate per creare il servizio di sicurezza; il secondo corrisponde al nome desiderato per il servizio di sicurezza (N.B. è previsto sempre un suffisso in `“_nsd”` da posporre al nome del servizio, ad es. `“Alice_nsd”`).
- **delete-sec-service**: elimina il servizio di sicurezza e tutte le relative istanze. Accetta come parametri `<nss_name>`.
- **sec-services-list**: restituisce la lista dei servizi di sicurezza attivi.
- **vnsf-list**: restituisce la lista delle vNSF attive.

Un esempio può essere il voler istanziare, da parte del service provider, un servizio di sicurezza per l'utente Alice. In questo caso la sintassi sarà:

```
$ ./nss_deployer deploy Alice Alice_nsd
```

Questo automaticamente darà inizio a tutte le operazioni necessarie per il recupero delle policy, la creazione del servizio di sicurezza, l'on-boarding e il successivo deploy. A questo punto, per controllare lo stato del servizio in modo più dettagliato, si possono utilizzare le due dashboard riportate di seguito, in quanto la loro accessibilità ed usabilità risulta tale da rendere superflua la creazione di ulteriori UI.

A.2 Dashboard Open Source MANO

Una volta che il servizio di sicurezza è stato lanciato, abbiamo nella dashboard di Open Source MANO (ricordiamo accessibile da https://OSM_HOST_IP_ADDRESS:8443) la situazione riportata in figura A.1. Si può vedere come il servizio sia attivo e sia possibile accedere dalla stessa ai relativi dettagli, questo cliccando sull'icona a forma di freccia adiacente al nome del servizio. A questo punto abbiamo la situazione in figura A.2, dove è presentato il menu di interazione con le istanze del servizio e delle vNSF.

Come è palese da tale immagine, per ogni singola vNSF risulta semplice selezionare ed eseguire le azioni a disposizione, semplicemente con un click sul relativo pulsante. Allora appariranno nella "Job list" le azioni in fase di scheduling per l'esecuzione. Un led verde segnerà l'azione andata a buon fine, uno giallo l'azione in lavorazione e uno rosso l'azione fallita. Ancora è possibile accedere alla console, per ogni singola vNSF, con la voce "VDU Console Links" e ottenere la schermata in figura A.3.

Come si vede chiaramente da questi semplici processi, avere un'interazione completa con le funzioni di sicurezza ed il servizio è molto comodo, questo a diversi livelli, dalle azioni da eseguire ad alto livello fino alla console, dov'è possibile accedere alla macchina virtuale per eventuali operazioni più specifiche.

The screenshot shows the OpenStack Dashboard (Launchpad) interface. The top navigation bar includes links for 'Launchpad', 'Istanze - OpenStack Dashboard', and 'Nuova scheda'. The main content area is titled 'LAUNCHPAD: DASHBOARD' and features a sidebar with navigation options: 'MANO', 'LAUNCHPAD', 'CATALOG', 'ACCOUNTS', and 'ADMINISTRATION'.

The 'NETWORK SERVICES' section displays a table of instantiated services:

+ Instantiate Service			
NS NAME	NSD	STATUS	UPTIME
test	Ignazio_nsd	Active	4m:16s

Below the table, the 'NETWORK SERVICE DETAILS' section provides information for the 'test' service:

- test** (link icon)
- NSD:** Ignazio_nsd
- MONITORING PARAMETERS:** NOT LOADED
- EPA-PARAMS:** NO EPA PARAMS CONFIGURED

The 'test' service is shown as 'Active' with an uptime of '4m:16s'.

Figura A.1: Dashboard di Open Source MANO a servizio di sicurezza istanziato.

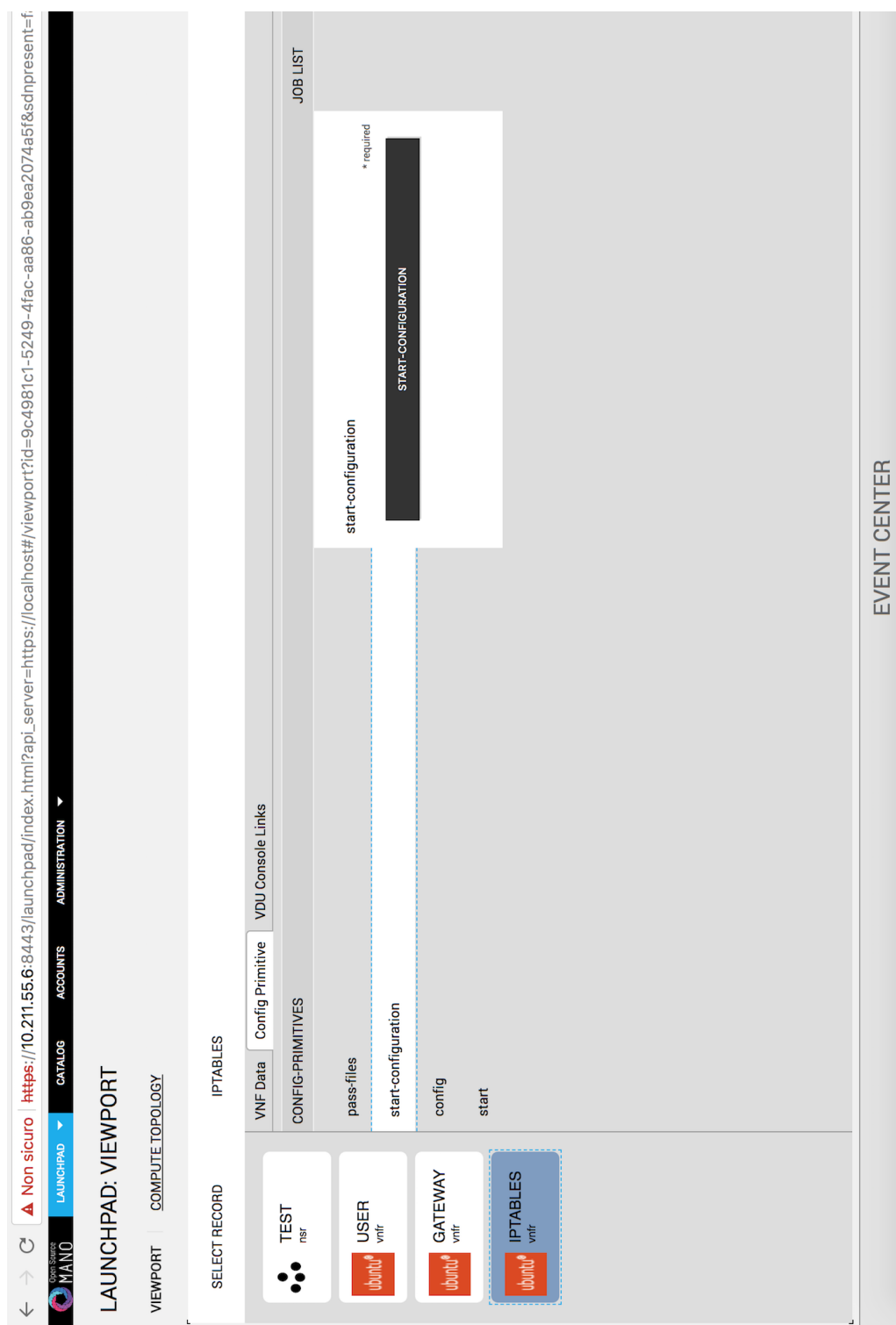


Figura A.2: Menu per eseguire le azioni sulle singole vNSF.

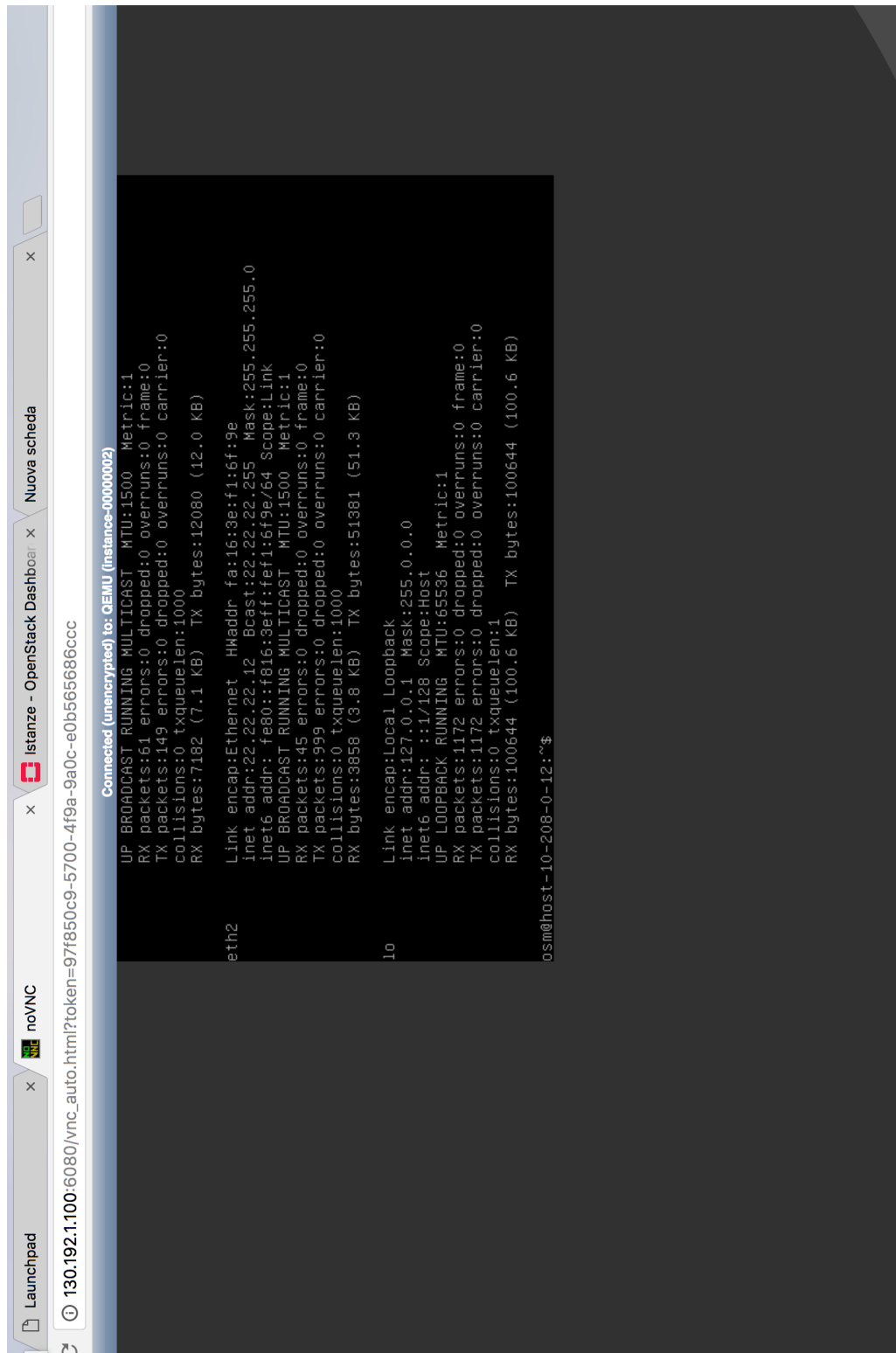


Figura A.3: Console noVNC per accedere alle singole vNSF.

A.3 Dashboard OpenStack

Una volta istanziato il servizio, se si vuole raggiungere dettagli ancora maggiori è possibile utilizzare la dashboard di OpenStack per gestire singolarmente le istanze e prevedere operazioni, come migrazione delle VM, snapshot o qualsiasi cosa inerente ai diversi domini dell'hypervisor.

Questo permette di avere un completo controllo sull'infrastruttura, mantenendo una notevole accessibilità e semplicità di utilizzo. Tutto questo grazie alla dashboard offerta da OpenStack, ovvero horizon. In figura [A.4](#) è riportata la situazione delle istanze, dopo il deploy di un semplice servizio di rete.



Figura A.4: Dashboard di OpenStack.

A.4 Dashboard Juju

Attraverso la dashboard di Juju è possibile controllare lo stato ed effettuare operazioni/configurazioni a livello di proxy charm. Questo significa che, indipendentemente dal servizio di rete, posso effettuare manutenzione sul proxy charm, attraverso l'interfaccia grafica (o la CLI di Juju), in modo rapido ed efficace. Potrebbe essere utile per un amministratore di sistema riavviare un charm o riconfigurarne in un servizio di lunga durata, senza intaccare tutte le funzionalità di orchestrazione di OSM.

Ricordiamo, infatti, che le configurazioni con Juju sono concettualmente disaccoppiate; quindi posso agire in modo indipendente sul proxy charm, senza intaccare il funzionamento del servizio di sicurezza online. In figura [A.5](#) è mostrata la tipica situazione di un proxy charm pronto all'uso e per questo valgono tutti i discorsi, che vedremo nell'appendice [B](#), su come gestire un charm da interfaccia grafica.

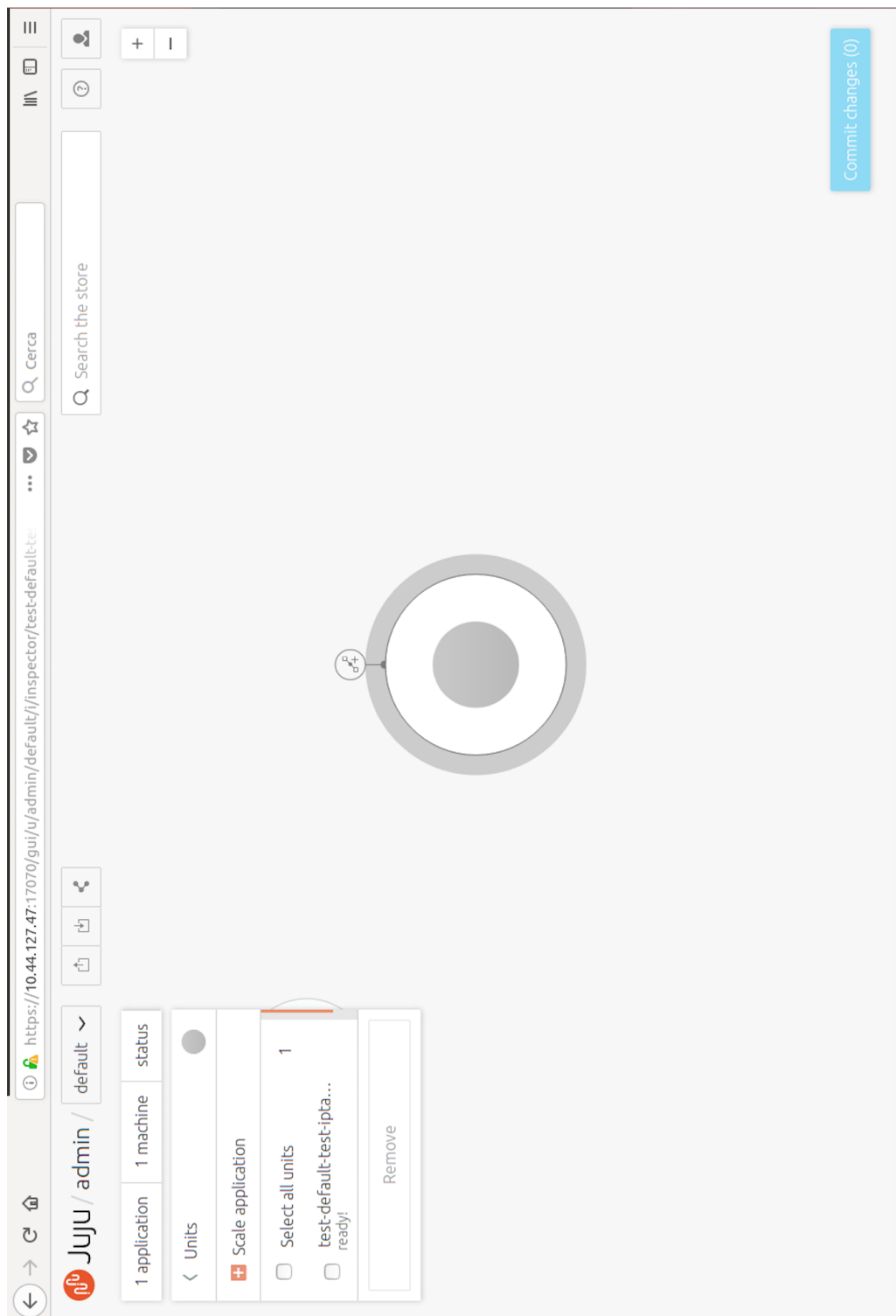


Figura A.5: Dashboard di Juju con proxy charm nello stato ready.

Appendice B

Manuale sviluppatore: Open Source MANO

B.1 Installazione e configurazione Open Source MANO

B.1.1 Installazione dipendenze OSM

La prima fase per l'installazione di OSM, per uno scenario di test, richiede la presenza di alcuni prerequisiti. Il primo è la presenza di una distribuzione di ubuntu linux; si consiglia l'utilizzo della versione 16.04.3 LTS reperibile al link:

<http://releases.ubuntu.com/16.04/>

È possibile effettuare il download sia della versione desktop che server, si consiglia la seconda per avere una versione più stabile e leggera, priva di interfaccia grafica. Una volta installata su una macchina fisica o VM tale distribuzione, bisogna procedere con l'integrazione dei pacchetti iniziali necessari:

```
$ sudo apt-get install curl
$ sudo apt-get install git
$ sudo apt-get install ssh #versione server per la connessione da remoto
```

I pacchetti in questo caso installati sono: curl, git ed ssh. Bisogna procedere poi all'installazione dell'hypervisor LXD per contenere i moduli di OSM:

```
# Installazione lxd
$ sudo apt-get update
$ sudo apt-get install -y lxd
$ newgrp lxd
```

Ora procediamo all’inizializzazione e configurazione di LXD:

```
# Configurazione LXD
$ sudo lxd init

Name of the storage backend to use (dir or zfs) [default=dir]:
Would you like LXD to be available over the network (yes/no) [default=no]?
Do you want to configure the LXD bridge (yes/no) [default=yes]?
Do you want to setup an IPv4 subnet? Yes
Default values apply for next questions
Do you want to setup an IPv6 subnet? No
LXD has been successfully configured.
```

Questa tipologia di installazione automaticamente crea un linux bridge `lxdbr0`, sul quale poi successivamente saranno disponibili i container LXC con i moduli OSM.

B.1.2 Installazione dipendenze OSM: opzionale

Nel caso in cui si voglia procedere ad una installazione innestata di OSM, ovvero installare i tre moduli RO, SO-ub, VCA (che sono contenuti in degli LXD container) in un ulteriore LXD container bisogna seguire questa procedura opzionale, da sostituire a parte di quella in sezione [B.1.1](#), in particolare a partire dalla configurazione LXD.

Innanzitutto creiamo un nuovo container LXD, quest’ultimo conterrà tutto l’ambiente OSM:

```
# Lanciare il container sull’host che deve ospitare OSM
$ lxc launch ubuntu:16.04 osmr3 -c security.privileged=true -c
  security.nesting=true
```

Ora bisogna limitare le risorse ed effettuare l’update del container:

```
# Limitare le risorse
$ lxc config set osmr3 limits.cpu 4
$ lxc config set osmr3 limits.memory 8GB

# Effettuare l’update del container
$ lxc exec osmr3 -- bash
$ sudo apt update
$ sudo apt upgrade
```

Ora è possibile effettuare la fase di inizializzazione e configurazione di LXD:

```
# inizializzare come segue lxd
$ sudo lxd init

# configuration to select
Do you want to configure a new storage pool (yes/no) [default=yes]?
Name of the new storage pool [default=default]:
Name of the storage backend to use (dir, btrfs, lvm, zfs) [default=zfs]: dir
Would you like LXD to be available over the network (yes/no) [default=no]?
Would you like stale cached images to be updated automatically (yes/no)
[default=yes]?
Would you like to create a new network bridge (yes/no) [default=yes]?
What should the new bridge be called [default=lxdbr0]?
What IPv4 address should be used (CIDR subnet notation, auto or none)
[default=auto]?
What IPv6 address should be used (CIDR subnet notation, auto or none)
[default=auto]? none
LXD has been successfully configured.
```

La situazione prodotta da questa diversa ed opzionale tipologia di installazione LXD, porta alla situazione in figura B.1. Potrebbe essere comodo nel caso di una sola macchina fisica a disposizione avere una tipologia di installazione del genere per ovviare ad un problema di accesso alla UI di OSM.

Dalla release THREE, infatti, il meccanismo di autenticazione avviene attraverso OAUTH 2 e viene ospitato nel container SO-ub un server di autenticazione che espone sulla porta 8009. Automaticamente OSM redirige tutte le connessioni, dell'host dove vengono ospitati i tre LXC container e relative alla porta 8443, al modulo SO-Ub, che espone un form di login che poi verrà reindirizzato al server di autenticazione, quest'ultimo rilascerà un token per accedere all'interfaccia se tale utente è autorizzato.

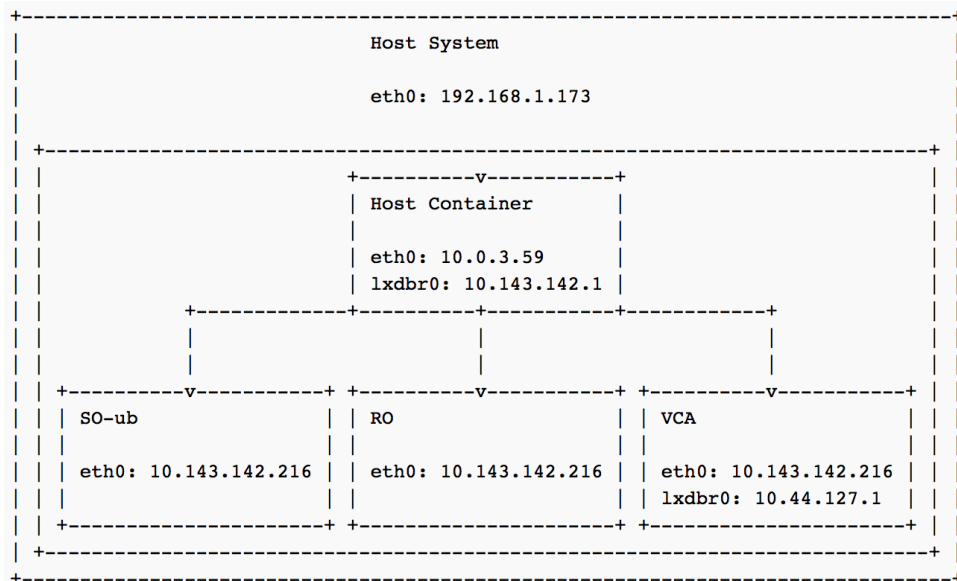


Figura B.1: Nested LXD.

Questo non permette all'utente di loggarsi direttamente dall'interno della macchina, per dei meccanismi di ridirezione impostati automaticamente da OSM durante l'installazione. Per ovviare

a questo problema si ci può connettere da un altro host (sulla stessa sottorete di quello di OSM), oppure nel caso in cui si è effettuata l'installazione con LXD innestato direttamente all'indirizzo esposto dal container più esterno.

B.1.3 Installazione OSM (Release THREE)

Una volta installate le dipendenze necessarie, come dalle precedenti sezioni, è possibile procedere all'installazione di OSM tramite i binari (consigliato) come riportato di seguito:

```
# effettuare il download tipicamente nella home
$ wget https://osm-download.etsi.org/ftp/osm-3.0-three/install_osm.sh
$ chmod +x install_osm.sh
$ ./install_osm.sh
```

Il codice appena presentato va eseguito sull'host nel caso di installazione bare-metal, invece nel container LXC esterno nel caso di nested LXD.

Vi è ancora la possibilità di ricompilare il codice e installarlo dai sorgenti questo effettuando il passaggio di seguito riportato, invece che quello precedente:

```
# effettuare il download tipicamente nella home
$ wget https://osm-download.etsi.org/ftp/osm-3.0-three/install_osm.sh
$ chmod +x install_osm.sh
$ ./install_osm.sh --source # aggiungere l'opzione \texttt{-b master} nel
    caso in cui si voglia installare dall'ultimo master
```

Una volta installate le dipendenze, tramite lo script presentato OSM, avremo collegati al linux bridge lxdbr0 tre container: SO-ub, RO e VCA; rispettivamente il Service Orchestrator, il Resource Orchestrator e il VNF Configuration & Abstraction. Juju sarà presente ed installato in quest'ultimo modulo. È necessario puntualizzare che facciamo riferimento alla release OSM THREE, di conseguenza vi è un meccanismo di autenticazione/autorizzazione attraverso Open ID connect e OAuth 2.0. Un Identity Provider viene esposto come servizio sulla porta 8009 del SO-Ub, quindi l'Identity Provider è parte del Service Orchestrator (in particolare del processo rwmmain). Risulta necessario che sia il browser che l'interfaccia grafica abbiano la possibilità di accedere al container SO-Ub e lo facciano utilizzando gli stessi URI. Tipicamente nell'installazione presentata non dovrebbero esserci problemi del genere, perché il tutto viene esposto sulla stessa sottorete (lxdbr0) e quindi i container riescono ad accedere mutuamente tra di loro.

Un'unica osservazione sta nel fatto che le ip route vengono automaticamente configurate dallo script di installazione e in particolare si noti che vi è una ridirezione del servizio dell'interfaccia grafica di OSM, dall'Host della macchina al container SO-Ub che la gestisce. In pratica nelle prime release di OSM per accedere alla UI bastava connettersi al container SO-Ub sulla porta 8443 (es. https://IP_SO-ub:8443), ora invece dato il meccanismo di ridirezione per accedere all'interfaccia grafica bisogna connettersi direttamente all'host che ospita OSM (o nel caso di LXD nested al container più esterno) (es. https://IP_HOST:8443). Tipicamente non è possibile accedere dall'Host stesso (dov'è installato OSM) all'interfaccia, a meno di alcune modifiche, ma questo spesso non risulta necessario in quanto è possibile accedervi tramite una macchina sulla stessa sottorete; tale problematica è relativa all'uso del NAT (di LXD bridge) configurato in precedenza. In ogni caso, qualora si abbia esigenza di configurazioni tutte sulla stessa macchina, si utilizzi una VM per OSM o una soluzione LXD nested, così diventa possibile accedere alla stessa dall'host fisico. L'ultima nota riguarda il certificato del Server OSM, è presente un certificato creato da Rift.io ovviamente non valido, per ragioni di test è possibile accettare la connessione SSL non sicura, installare il certificato non valido Rift.io nel browser o fornire un certificato valido in ambienti di produzione.

N.B. È possibile che ci siano dei problemi in alcune versioni dello script di installazione scaricato da OSM, in relazione all'introduzione della route di Juju tra quelle del kernel del sistema operativo host di OSM. In particolare l'host dove risiede OSM deve inoltrare i pacchetti diretti a Juju

attraverso il container VCA. Se non presente o non inserita automaticamente al reboot, si può inserire manualmente o in automatico con uno script, attraverso questo comando:

```
$ sudo ip route add $JUJU_IP_ADDRESS via $VCA_IP_ADDRESS
```

B.2 Descrizione UI e CLI di OSM

Adesso andiamo a descrivere quali sono i meccanismi per l'interazione con OSM, ovvero User Interface (UI) o CLI (Command Line Interface). Dalla Release THREE entrambe sono presenti e già configurate sulla macchina, quindi vi si può interagire appena dopo l'installazione.

B.2.1 User Interface (UI)

Per effettuare il login alla UI connettersi a https://IP_HOST_ADDRESS:8443, e si avrà la seguente schermata di login [B.2](#), alla quale accedere con user/pwd admin/admin.

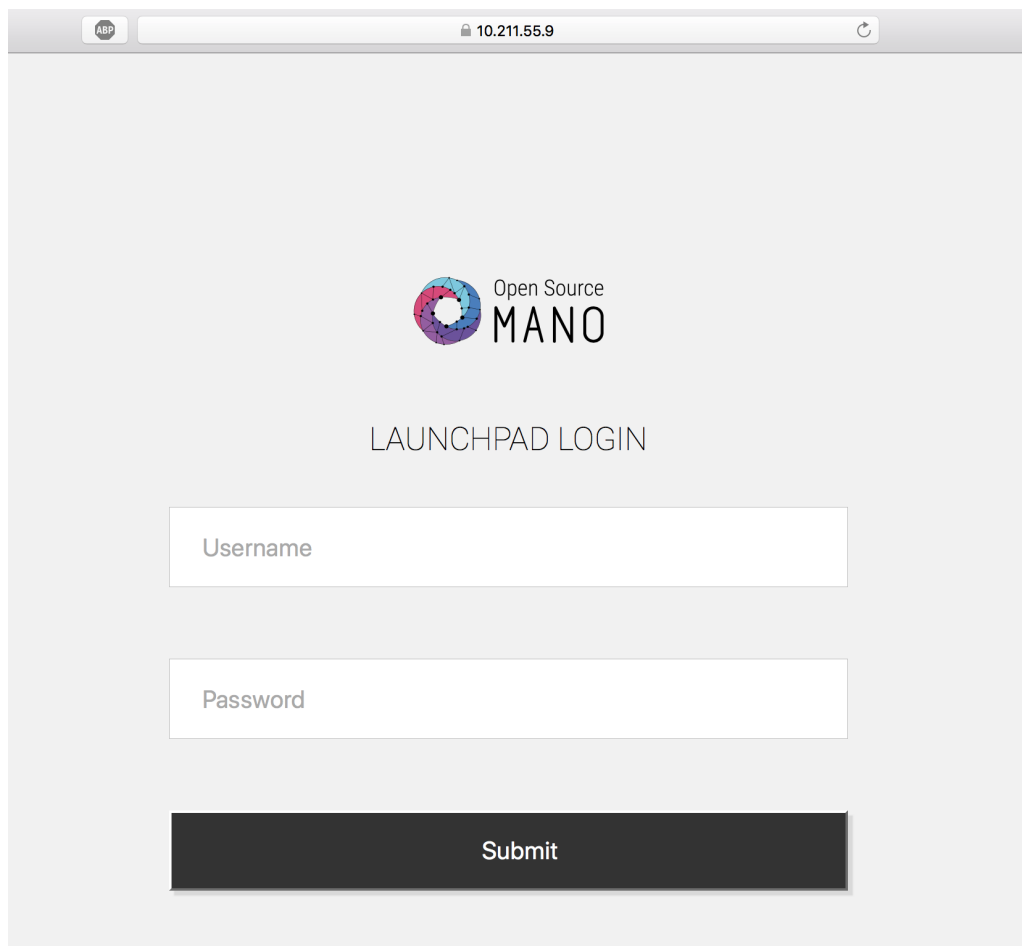


Figura B.2: Login UI.

Sezioni UI

- **Launchpad:** la sezione launchpad è quella relativa alla principale applicazione del Service Orchestrator, come specificato nei capitoli di teoria (cap. 3). Questa sezione prevede tre diverse sottosezioni: Dashboard, Instantiate e SSH keys. La prima aggrega tutti i servizi di rete in esecuzione e permette di interagire con essi e con le VNF. La seconda fornisce una

lista dei servizi di rete presenti nel catalogo e permette di istanziarli. L'ultima consente la definizione di chiavi SSH pubbliche da iniettare nelle istanze per connettersi successivamente, questo ovviamente dopo aver creato con tool esterni un keypair (Esempio di istanziazione nelle figure B.3 e B.4).

- **Catalog:** tale sezione gestisce i cataloghi sia relativi ai VNFD che agli NSD. È inoltre possibile modellare servizi di rete e VNF attraverso l'interfaccia grafica. L'applicazione che fa questo, integrata nell'UI, è il Composer. Vi è una sezione che permette di scegliere il catalogo e, una volta selezionato l'NS o la VNF desiderata, è possibile accedervi per modifiche. Risulta possibile anche crearne di nuove e gestire in modo automatico i package tar.gz, scaricando quelli modellati attraverso la UI per poi riutilizzarli. La modellazione presenta due sezioni: Descriptor e Assets. La prima sezione consente di modellare graficamente il servizio ad alto livello, la seconda consente di inserire dei file di supporto direttamente all'interno del descrittore, come se li avessimo inclusi nel package tar.gz. (fig. B.5)
- **Accounts:** tramite questa sezione, come riportato in figura B.6, è possibile aggiungere o modificare gli account relativi a tutte le piattaforme che ruotano intorno all'ecosistema OSM. È possibile ad esempio aggiungere e modificare VIM, SDN Controller, Resource Orchestrator e VNF Manager.
- **Administration:** In questo pannello è possibile modificare tutta una serie di informazioni relative alla gestione di OSM nel complesso: progetti, utenti, autenticazione, logging, ridondanza, access control role based. In sostanza dalla release THREE OSM ha consolidato le funzionalità relative alla ridondanza e alla resilienza della piattaforma, introducendo anche supporto ad un sistema di gestione degli accessi di tipo ACRB (Access Control Role Based); questo consente una migliore gestione nel complesso del concetto di multi-tenancy, ovvero la presenza di più utilizzatori che possono sfruttare un sottoinsieme di risorse di OSM. È possibile suddividere in diversi progetti e assegnare diversi ruoli agli utenti che vengono allocati ai diversi progetti, definendone le autorizzazioni (una gestione OpenStack-like).
- **Project:** da indicazione sul progetto corrente e con quali progetti ha la possibilità di interagire l'utente corrente.
- **Username:** indica i dettagli dell'utente corrente.

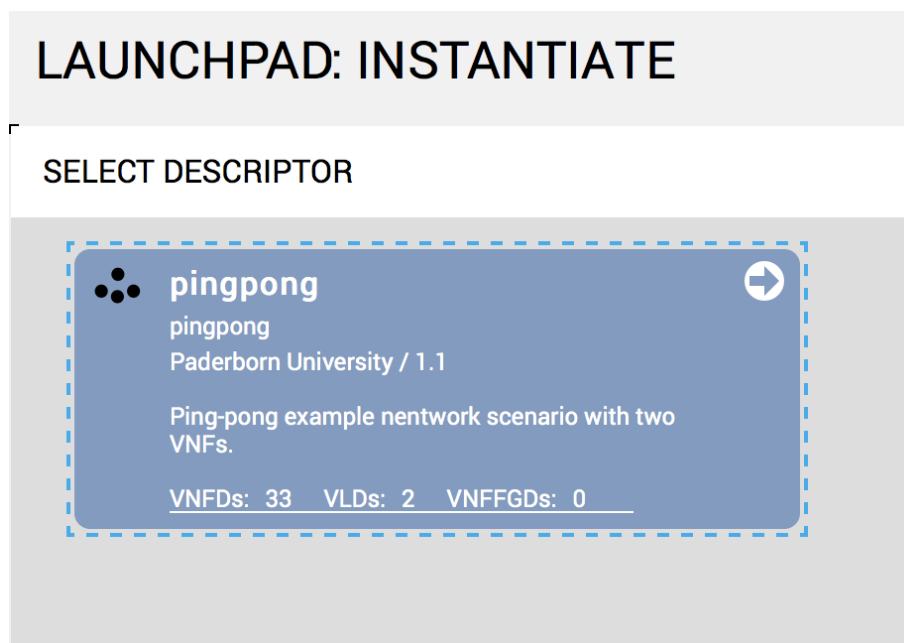


Figura B.3: Istanziamento tramite UI (fase 1/2).

LAUNCHPAD: INSTANTIATE

pingpong

Paderborn University / 1.1

Ping-pong example network scenario with two VNFs

VNFs: 33 VLDs: 2 VNFFGDs: 0

DESCRIPTOR

```
id:
  short name: "mgmt_vf"
  vnf network name: "default"
  description: "Management VLD"
  name: "mgmt_vf"
  mgmt network: "true"
  id: "mgmt_vf"
  vnfid connection point ref:
    member vnf index ref: 2
    vnfid connection point ref: "pong/cp0"
    vnfid id ref: "pong"
  member vnf index ref: 1
  vnfid connection point ref: "ping/cp1"
  vendor: "Paderborn University"
  type: "vnm-network-name"
  version: "1.0"
  id: "vld2"
  mgmt network: "false"
  vnf network name: "default"
  vnfid connection point ref:
    member vnf index ref: 2
    vnfid connection point ref: "pong/cp1"
    vnfid id ref: "pong"
  member vnf index ref: 3
  vnfid connection point ref: "ping/cp0"
  name: "vld2"
  type: "vnm-network-name"
  short name: "pingpong"
  description: "Ping-pong example network scenario with two VNFs"
  name: "pingpong"
  id: "pingpong"
  vendor: "Paderborn University"
  constituent vnfid:
    member vnf index: 1
    vnfid id ref: "ping"
```

INPUT PARAMETERS

INSTANCE NAME*

RESOURCE ORCHESTRATOR

osmopenmano

DATACENTER

emu-vimt

NS/VNF ACCOUNT PLACEMENTS

VNFD: PING

DATACENTER

CONFIG AGENT ACCOUNT

VNFD: PONG

DATACENTER

CONFIG AGENT ACCOUNT

VNFD: PING

DATACENTER

BACK

CANCEL

LAUNCH

Figura B.4: Istanziamento tramite UI (fase 2/2).

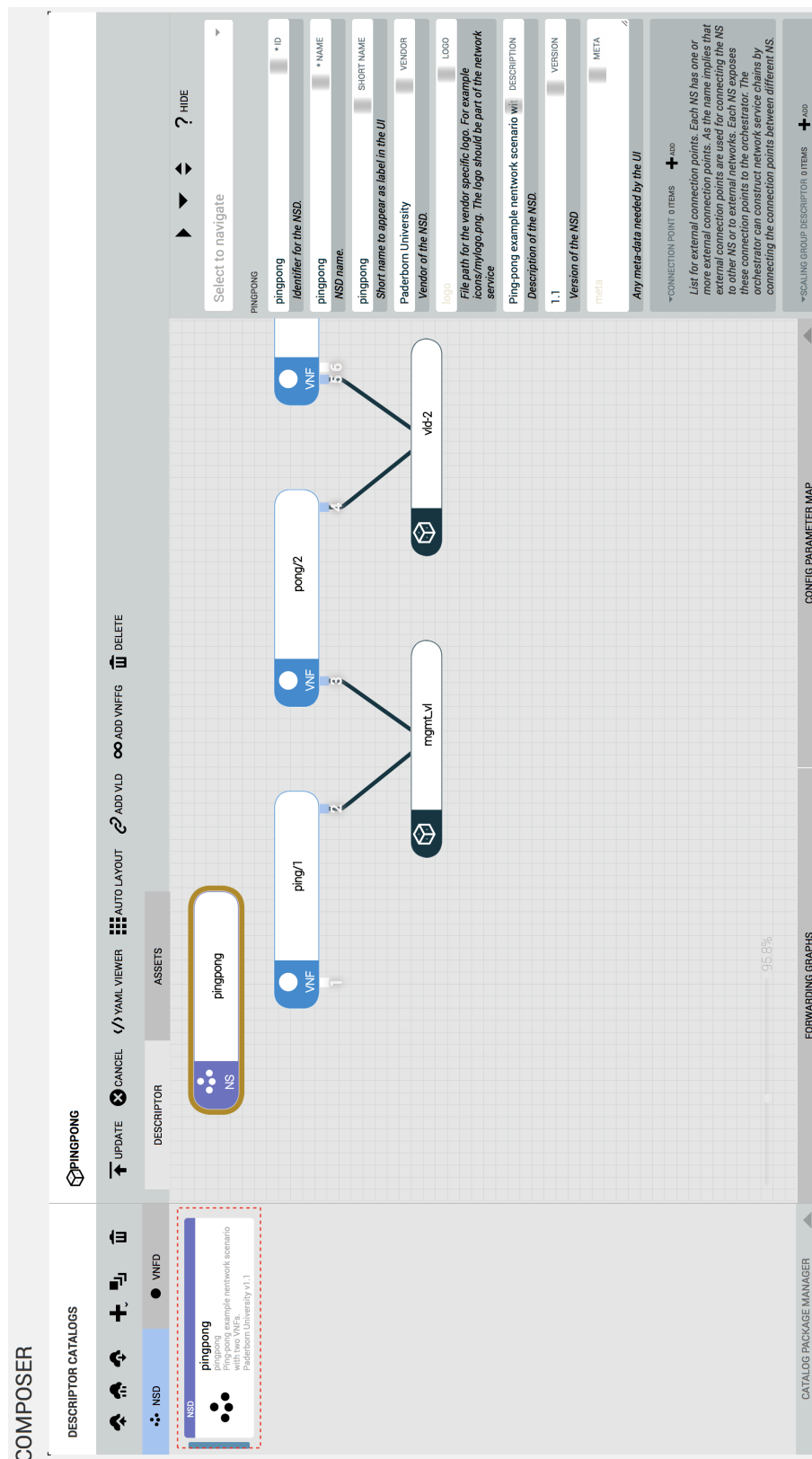


Figura B.5: Composer dell’UI per la modellazione dei servizi di rete e delle funzioni virtuali.

B.2.2 Command Line Interface (CLI)

La command Line Interface di OSM nasce con la release TWO in modo opzionale e viene completamente integrata all'interno delle Release THREE di OSM. Prima di utilizzare la stessa c'è

ACCOUNTS

CHECK ALL CONNECTIVITY STATUS

RO ACCOUNTS

OSMOPENMANO

ADD RO ACCOUNT

VIM ACCOUNTS

ADD VIM ACCOUNT

SDN ACCOUNTS

ADD SDN ACCOUNT

CONFIG AGENT ACCOUNTS

OSMJUJU

ADD CONFIG AGENT ACCOUNT

ACCOUNT

osmjuju

CONNECTION STATUS

SUCCESS

REFRESH STATUS

UPDATE ACCOUNT DETAILS

IP ADDRESS *

10.44.127.117

PORT

17070

USERNAME

admin

SECRET

.....

REMOVE ACCOUNT CANCEL UPDATE

Figura B.6: Gestione dei moduli esterni tramite UI (VIM, SDN Controller, RO, VNFN).

bisogno di definire due variabili d'ambiente (ovviamente è possibile definirle nel file `.bashrc` così da evitare di specificarle ogni volta):

```
export OSM_HOSTNAME=$(lxc list awk '($2=="SO-ub")print $6') —
export OSM_RO_HOSTNAME=$(lxc list awk '($2=="RO")print $6') —
```

Di seguito vedremo come è possibile operare con la CLI, attraverso l'analisi dei suoi comandi. Innanzitutto il comando base da anteporre agli altri è `osm`, questo sull'Host che virtualizza i container di OSM. I vari possibili comandi sono:

- **config-agent-add**: aggiunge un VNF Manager (es. Juju). I parametri sono (nella forma `--nome_parametro parametro`) `-name`, `-account_type` (es. Openstack), `-server` (indirizzo IP), `-user` (utente di osm) e `-secret` (password o chiave).
- **config-agent-delete**: elimina un VNF Manager, specificando il parametro `-name` (unicamente il nome senza l'opzione in questo caso).
- **config-agent-list**: restituisce la lista dei VNF Manager, utile in fase di sviluppo per ottenere le credenziali di Juju.
- **ns-create**: lancia un servizio di rete. I parametri sono `-ns_name`, `-nsd_name` (nome del relativo descrittore), `-vim_account` (su quale datacenter istanziare il servizio), `-admin_status`, `-ssh.keys` (lista ordinata delle chiavi da iniettare nelle VNF separate dalla virgola) e `-config` (configurazioni personalizzate aggiuntive in file yaml).
- **ns-delete**: elimina un servizio specificando il parametro `-ns_name`.
- **ns-list**: restituisce la lista di tutti i servizi di rete attivi o in fase di lancio/configurazione.

- **ns-monitoring-show:** mostra i parametri di configurazione di un servizio specificato con `-ns_name`.
- **ns-scale:** permette di scalare un servizio di rete specificando `-ns_name`, `-ns_scale_group` (il gruppo di VNF da scalare predefinito) e `-index` (l'indice di scalamento).
- **ns-scaling-show:** mostra i dettagli di scalamento di un servizio di rete specificando `-ns_name`.
- **ns-show:** mostra i dettagli di un servizio di rete, specificando il parametro `-ns_name`.
- **nsd-delete:** elimina un descrittore di un servizio di rete, specificando il parametro `-nsd_name`.
- **nsd-list:** restituisce la lista dei descrittori dei servizi di rete disponibili.
- **ro-dump:** salva lo stato del Resource Orchestrator.
- **upload-package:** permette di effettuare l'upload di ogni tipo di package sia di NSD che di VNFD, specificando il file comprensivo di path dove è situato il pacchetto tar.gz.
- **vcs-list:** restituisce una lista dei processi che costituiscono il Service Orchestrator e il loro stato.
- **vim-create:** permette di integrare un VIM in OSM, specificando i parametri `-name`, `-user` (del VIM), `-password`, `-auth_url` (url di autenticazione ad es. `http://controller:5000/v3` di solito per OpenStack), `-tenant`, `-config` (configurazioni aggiuntive ad esempio Domain in OpenStack nella forma `-config='availability_zone: one'`), `-account_type` (es. `openstack`) e `-description`.
- **vim-delete:** permette di eliminare un VIM, specificando il `-name`.
- **vim-list:** restituisce la lista dei VIM integrati in OSM.
- **vim-show:** restituisce le informazioni di un VIM, specificando il parametro `-name`.
- **vnf-list:** restituisce la lista delle VNF che sono in un servizio lanciato o in fase di lancio.
- **vnf-monitoring-show:** mostra i parametri predefiniti per il monitoraggio di un VNF, specificando il `-vnf_name`.
- **vnf-show:** mostra i dettagli di una VNF attiva, specificando il parametro `-vnf_name`.
- **vnfd-delete:** elimina un descrittore di una VNF, specificando il parametro `-vnfd_name`.
- **vnfd-list:** restituisce la lista dei descrittori dei VNFD presenti nel catalogo.

B.3 Operazioni di base con OSM

Andiamo a descrivere ora quali sono le operazioni tipiche di base da effettuare attraverso OSM: integrazione di uno o più VIM, onboarding dei descrittori ed infine il deploy del servizio di rete.

B.3.1 Integrazione di un VIM

Per integrare un VIM si possono scegliere più strade. La prima è quella di utilizzare la User Interface e il pannello account come riportato in figura B.6, altrimenti vi è la possibilità di integrarli attraverso la CLI o l'RO; queste ultime due strade rappresentano quelle più flessibili ed immediate.

Utilizzo della CLI

Come abbiamo visto il comando per l'integrazione attraverso la CLI è `osm vim-create`. A scopo di esempio illustreremo come integrare OpenStack che è anche il VIM utilizzato per la nostra soluzione. Il comando nel suo complesso è:

```
$ osm vim-create --name openstack-site --user admin --password userpwd
  --auth_url http://10.10.10.11:5000/v2.0 --tenant admin --account_type
  openstack --config='{security_groups: default, keypair: mykey}'
```

In tale comando specifichiamo i seguenti parametri il nome dell'utente OpenStack, la password, l'URL per autenticazione (come descritto in precedenza), il tenant e la tipologia di account. Le configurazioni sono facoltative e da utilizzarsi solo quando necessarie, ad esempio per istruire il controller di OpenStack su qual'è il dominio a cui si fa riferimento (tipicamente default), il keypair da utilizzare la comunicazione (chiavi SSH), la configurazione dei security group o la availability zone.

Utilizzo dell'RO

Attraverso l'RO, opzione retrocompatibile con le prime versioni di OSM, l'integrazione del VIM avviene attraverso due fasi: create ed attach. Vediamo di seguito i comandi per effettuare entrambe queste operazioni:

```
$ lxc exec R0 -- bash # accedo al container R0

#creazione del VIM
$ openmano datacenter-create openstack-site http://10.10.10.11:5000/v2.0
  --type openstack --description "OpenStack site"
  --config='{security_groups: default, keypair: mykey}'

#collegamento al VIM
$ openmano datacenter-attach openstack-site --user=admin --password=userpwd
  --vim-tenant-name=admin --config='{availability_zone: one}'
```

B.3.2 Onboarding dei descrittori

Per effettuare l'onboarding dei descrittori come sempre vi sono due strade o attraverso la UI, oppure tramite la CLI.

Con la prima soluzione possiamo effettuare il drag-and-drop dell'archivio tar.gz all'interno dell'UI, in particolare nel pannello catalog in figura [B.5](#).

Attraverso la CLI invece va specificato il comando `osm upload-package` con il nome dell'archivio ed eventualmente il path per identificarne la posizione. Un esempio è riportato di seguito:

```
$ osm upload-package cirros_vnf.tar.gz # upload VNFD
$ osm vnfd-list # controllo del corretto on-boarding

$ osm upload-package cirros_2vnf_ns.tar.gz # upload NSD
$ osm nsd-list # controllo
```

Nel primo caso abbiamo l'upload di un VNFD nel secondo un NSD, entrambi con il medesimo comando. Questo rappresenta un ottimo strumento per automatizzare questi processi via scripting. Nel processo, qualsiasi sia il metodo di onboarding, va tenuto conto dei vincoli relativi ai VNFD. In sostanza vanno caricati prima tutti i VNFD per poter istanziare l'NSD che li contiene, onde evitare errori; lo stesso ragionamento in fase di eliminazione.

B.3.3 Deploy di un servizio di rete

Una volta effettuata la fase di onboarding è possibile istanziare il servizio di rete. Questo come sempre si può fare attraverso UI o CLI. Nelle immagini [B.3](#) e [B.4](#), viene mostrato come effettuare il lancio tramite UI, in particolare attraverso la sezione instantiate.

Durante la seconda fase è possibile modificare tutta una serie di parametri di contesto, ad esempio per ogni VNF è possibile decidere su quale VIM essa venga allocata e quale dei VNFM è destinato alla sua configurazione. Nel caso di CLI, come abbiamo visto nel presentare i comandi, è sufficiente fornire il seguente comando (il resto dei parametri visti in precedenza (sez. [B.2.2](#)) è del tutto facoltativo e regolato da necessità specifiche):

```
$ osm ns-create --nsd_name nome_nsd --ns_name nome_ns vim_account nome_vim
```

B.4 Esempi di NSD e VNFD

Abbiamo visto nei capitoli del lavoro di tesi (cap. 6) quali sono i principali campi dei descrittori; adesso andiamo a vederne degli esempi nel caso specifico della funzione di sicurezza Packet Filter.

B.4.1 VNF Descriptor di una vNSF

Andiamo a valutare com'è stato definito il VNF Descriptor (per noi anche vNSFD) della vNSF Packet Filter. Per farlo ci rifacciamo alle figure B.7 e B.8. Innanzitutto ricordiamo che il modello di dati è stato definito attraverso YANG e il formato dei descrittori è tipicamente di tipo yaml. Nella prima parte notiamo come siano presenti i meta-data di cui abbiamo discusso nel capitolo 6.

Questi danno delle informazioni alla UI per quanto riguarda il posizionamento degli elementi nel composer, inoltre forniscono dei dettagli sui path dei file di supporto all'interno del descrittore.

Subito dopo vediamo come inizia la definizione delle interfacce esterne, che nel nostro caso (come da progetto nel capitolo 7) sono 3: eth0, eth1 ed eth2. Per comodità parte del codice è stato abbreviato per rendere solo i concetti fondamentali. Tornando alle interfacce esterne, queste possono essere di diversi tipi, in questo caso sono state definite come VPORT ed è stato disabilitato il port-security.

Abbiamo discusso di questa caratteristica in precedenza, ma sembra il caso di ricordare che questa è un'opzione di sicurezza di OpenStack contro l'IP spoofing, nel nostro caso viene disabilitata per permetterci di configurare le interfacce automaticamente senza l'utilizzo di SDN. Questo significa che per fare questo dobbiamo attraversare i nodi di rete creati da OpenStack (abbiamo visto come vengono mappati da OSM nel capitolo 8) e, senza l'ausilio di un SDN Controller, dobbiamo utilizzare funzionalità di DNAT ed SNAT; queste ultime vengono bloccate dal controllo di OpenStack contro l'IP spoofing.

Successivamente vi sono delle informazioni identificative della vNSF (nome, descrizione, logo). Ancora abbiamo un'indicazione su qual'è il ruolo della funzione virtuale all'interno della Service Function Chain, questo parametro viene settato come "UNAWARE" perché non è stato integrato l'SDN controller quindi non ha senso definire tale caratteristica.

Ora si passa alla VDU (Virtual Deployment Unit), a questa, come si vede in figura B.7, viene collegato un file di configurazione cloud_init.cfg, questo in accordo al discorso degli user-data fatto nel capitolo 6. Il parametro di scalabilità, ovvero quello che indica il numero di copie dell'istanza che vogliamo ed è settato ad 1, poiché non abbiamo ancora necessità di scalare orizzontalmente il servizio.

Viene poi definita la politica di cpu-pinning, ovvero a quali cpu assegnare il carico della CPU e in questo caso non ci interessa vincolare la funzione di sicurezza ad una CPU particolare, quindi la definiamo come "ANY", ovvero qualsiasi CPU può eseguirla. Definiamo il nome dell'immagine in "ubuntu1604", questo è il riferimento che utilizzerà OpenStack per scegliere l'immagine da utilizzare in Glance.

Ora c'è la fase di mapping delle interfacce esterne, definite a livello più alto dalla vNSF, sull'unità virtuale che esegue concretamente la computazione. Tra le varie possibilità c'è quella di assegnare dei floating-ip ad una delle interfacce così da rendere accessibile pubblicamente la funzione di sicurezza, in questo caso, poiché lavoriamo in un contesto controllato e la funzione è parte della catena non abbiamo di queste esigenze. Può essere utile eventualmente, in produzione, per definire gli end-point del servizio come pubblici.

```
vnfd:vnfd-catalog:
  vnfd:vnfd:
    - rw-vnfd:meta: '{"containerPositionMap": ...}' # UI meta-data
      vnfd:connection-point: # External interfaces
    - vnfd:name: eth0
      vnfd:port-security-enabled: 'false'
      vnfd:type: VPORT
    - vnfd:name: eth1
    ...
  vnfd:description: Generated by OSM package generator # General
    description
  vnfd:id: iptables
  vnfd:logo: ubuntu-logo14.png
  vnfd:mgmt-interface:
    vnfd:vdu-id: ubuntu_3iface_cloudinit
  vnfd:name: iptables
  vnfd:service-function-chain: UNAWARE
  vnfd:vdu: #VDU details
    - vnfd:cloud-init-file: cloud_init.cfg #config cloud-init file spec
      vnfd:count: '1' # Scalability parameter
      vnfd:description: ubuntu_3iface_cloudinit
      vnfd:guest-epa:
        vnfd:cpu-pinning-policy: ANY
      vnfd:id: ubuntu_3iface_cloudinit
      vnfd:image: ubuntu1604 # Software image used
      vnfd:interface: # Mapping of VNF connection points on VDU interfaces
    - rw-vnfd:floating-ip-needed: 'false'
      vnfd:external-connection-point-ref: eth0
      vnfd:name: eth0
      vnfd:position: '1'
      vnfd:type: EXTERNAL
      vnfd:virtual-interface:
        vnfd:bandwidth: '0'
        vnfd:type: OM-MGMT
    - rw-vnfd:floating-ip-needed: 'false' # OpenStack declaration for
      external reachability
      vnfd:external-connection-point-ref: eth1
    ...
  vnfd:name: ubuntu_3iface_cloudinit
  vnfd:supplemental-boot-data:
    vnfd:boot-data-drive: 'false'
  vnfd:vm-flavor: # VM flavor alias performances required
    vnfd:memory-mb: '256'
    vnfd:storage-gb: '4'
    vnfd:vcpu-count: '1'
  vnfd:vendor: 'Ignazio Pedone (Polito)'
  vnfd:version: '3.0'
```

Figura B.7: Esempio VNFD Packet Filter (Parte 1/2).

```
vnfd:vnf-configuration: # Configurations
  vnfd:config-primitive: # Defining configuration primitives for
    later mapping on actions
  - vnfd:name: pass-files
  - vnfd:name: start-configuration
  - vnfd:name: config
    vnfd:parameter: # Service Primitives parameters
    - rw-vnfd:out: 'false'
      vnfd:data-type: STRING
      vnfd:default-value: <rw_mgmt_ip>
    ...
  - vnfd:name: start
vnfd:initial-config-primitive:
- vnfd:name: config
  vnfd:parameter:
  - vnfd:name: ssh-hostname
    vnfd:value: <rw_mgmt_ip>
  ...
  vnfd:seq: '1'
vnfd:juju: # Configuration method via Proxy Charm
vnfd:charm: iptables
```

Figura B.8: Esempio VNFD Packet Filter (Parte 2/2).

A questo punto vi sono le informazioni sulle caratteristiche del flavour di OpenStack da utilizzare, in particolare quindi le prestazioni richieste dalla funzione di sicurezza.

Passando ora seconda parte in figura B.8, abbiamo i meccanismi di configurazione. Innanzitutto vengono definite le Service Primitives, in accordo con il capitolo 6, dove è stato definito come avveniva il mapping tra queste e le azioni di Juju. Infine, si notano i parametri di input che possono essere definiti per le stesse e la definizione del metodo di configurazione complessivo attraverso il Proxy Charm, in questo caso chiamato analogamente alla vNSF iptables.

Esempio di cloud-init

Risulta di interesse approfondire anche qual'è il ruolo della cloud-init nella fase iniziale della configurazione (Day-0 Configuration), questo analizzando il codice sempre nel caso specifico di Packet Filter (fig. B.9). Innanzitutto partendo dall'alto, vediamo che viene assegnata una password all'utente corrente, in questo caso per l'utente della cloud image "ubuntu", eventualmente è possibile cambiare tale utente attraverso la sintassi presentata nel capitolo 6. Ma ora concentriamoci sulla parte più interessante, ovvero sfruttare tutti i meccanismi a disposizione della cloud-init per configurare la vNSF.

Possiamo eliminare la necessità della password per il comando eseguito con i privilegi sudo, creare dei file in modo arbitrario, eseguire dei comandi e installare preventivamente alcuni pacchetti; tutto questo prima che l'istanza sia attiva e nei tempi testati nel capitolo 8.

Scendiamo più in dettaglio e vediamo che è possibile creare un file definendone direttamente il contenuto, oppure utilizzando una codifica base64 (binary), più idonea e priva di tutte le problematiche dei caratteri inseriti nel file yaml. Possiamo scegliere il percorso dove creare il file, il proprietario e regolare i permessi. Per quanto riguarda l'esecuzione dei comandi, "runcmd" è una delle modalità che permette di farlo in cloud-init. Nell'esempio creiamo una cartella di prova nella \$HOME, attiviamo la possibilità di inoltrare pacchetti (punto fondamentale per una vNSF e più in generale per una VNF) ed infine eseguiamo uno script.

In ultima analisi con "packages" andiamo a caricare tutti i pacchetti necessari per la vNSF.

Si noti come a questo punto con la capacità di definire script ed eseguirli, sia possibile configurare completamente una vNSF, senza l'ausilio di Juju; discorso che facevamo per seguire l'approccio ibrido indicato sopra. A questo punto potremmo settare tutto quello che serve per la vNSF ora e renderla già disponibile per il servizio di sicurezza. Nel caso di riconfigurazione poi entrerà in gioco Juju, evitando di incidere così sul tempo di istanziazione della vNSF.

```
#cloud-config
password: osm4u
chpasswd: { expire: False }
ssh_pwauth: True
sudo: ["ALL=(ALL) NOPASSWD:ALL"]

write_files:
- content: |
    # My new helloworld file

    owner: root:root
    permissions: '0644'
    path: /home/ubuntu/helloworld.txt
- content: !!binary |
    IyEvYmluL3NoCi...
    owner: root:root
    permissions: '0644'
    path: /home/ubuntu/script.sh
- content: !!binary |
    DQphdXRvIGVuczQNCmlmYWVWNl...#b64 encoding
    owner: root:root
    permissions: '0644'
    path: /home/ubuntu/append.sh
- content: !!binary |
    c3VkbYBzaCAtYyAnY2F0IGFwcG...
    owner: root:root
    permissions: '0644'
    path: /home/ubuntu/network.sh
- content: !!binary |
    IyEvYmluL3NoDQpzdWRvIHRyI...
    owner: root:root
    permissions: '0644'
    path: /home/ubuntu/conversion.sh

runcmd:
- 'mkdir test'
- 'sysctl -w net.ipv4.ip_forward=1'
- 'source /home/ubuntu/script.sh'

packages:
- nmap
```

Figura B.9: Esempio user-data cloud-init nel caso di Packet Filter.

B.4.2 NS Descriptor di un NSS

Vediamo adesso un esempio di come è definito, in modo del tutto automatico, il descrittore del servizio di sicurezza e quali sono gli elementi che lo compongono. Abbiamo visto nei capitoli precedenti (cap. 7 e 8) che servivano due elementi fondamentali per definire in modo completo un servizio di sicurezza (NSS): un insieme di vNSF costituenti e un insieme di collegamenti tra le stesse. Nella figura B.10 prende forma il primo dei due elementi, ovvero, oltre ai dati identificativi dell’NSS, vengono definite quali sono le vNSF che ne fanno parte. In aggiunta vi è un’opzione che inizializza immediatamente la vNSF e un indice per contraddistinguerla all’interno del servizio (utile soprattutto nell’integrazione con SDN e per i forwarding graph).

Nella seconda figura B.11, sono modellati i virtual link che collegano le varie vNSF che costituiscono il servizio di sicurezza. Analizziamo uno dei link, il primo ad esempio definisce una rete di management comune a tutte le vNSF e le stesse vengono tutte collegate attraverso l’interfaccia eth0. Il secondo invece costituisce la tipologia di link tra vNSF, infatti, vengono dichiarati i membri della connessione, nel nostro caso sono solo 2, e l’interfaccia di uscita della prima vNSF (eth2) viene collegata con quella di ingresso della seconda (eth1). In questo caso questi link sono stati mappati su delle reti precostituite sul VIM e quindi non è stato necessario creare degli IP profiles, ovviamente nel caso dell’implementazione finale anche le reti vengono create in modo automatico, questo per permettere di non dover precostituire le stesse attraverso il VIM. La sintassi non si complica, infatti, è sufficiente dichiarare degli IP profile e dettagli come la sottorete di base, l’utilizzo o meno di dhcp e un eventuale gateway (es. di creazione di IP profile nella documentazione ulteriore allegata).

```
nsd:nsd-catalog:
  nsd:
    - id: Ignazio_nsd
      name: Ignazio_nsd
      short-name: Ignazio_nsd
      description: Generated by OSM pacakage generator
      vendor: Ignazio Pedone
      version: '3.0'
      constituent-vnfd:
        - member-vnf-index: '1'
          start-by-default: 'true'
          vnfd-id-ref: user
        - member-vnf-index: '2'
          start-by-default: 'true'
          vnfd-id-ref: iptables
        - member-vnf-index: '3'
          start-by-default: 'true'
          vnfd-id-ref: gateway
```

Figura B.10: Esempio di NSD semplice relativo ad un Network Security Service (Parte 1/2).

```
vld: # Virtual Link definition
# Networks for the VNFs
- id: mgmt
  name: mgmt
  short-name: mgmt
  mgmt-network: 'true'
  vim-network-name: mgmt
  vnfd-connection-point-ref:
  - member-vnf-index-ref: 1
    vnfd-id-ref: user
    vnfd-connection-point-ref: eth0
  - member-vnf-index-ref: 2
    ...
- id: link_1
  mgmt-network: 'false'
  name: link_1
  vim-network-name: link_1
  vnfd-connection-point-ref:
  - member-vnf-index-ref: '1'
    vnfd-connection-point-ref: eth2
    vnfd-id-ref: user
  - member-vnf-index-ref: '2'
    vnfd-connection-point-ref: eth1
    vnfd-id-ref: iptables
- id: link_2
  mgmt-network: 'false'
  name: link_2
  vim-network-name: link_2
  vnfd-connection-point-ref:
  - member-vnf-index-ref: '2'
    vnfd-connection-point-ref: eth2
    vnfd-id-ref: iptables
  - member-vnf-index-ref: '3'
    vnfd-connection-point-ref: eth1
    vnfd-id-ref: gateway
- id: internet
  mgmt-network: 'false'
  ...
```

Figura B.11: Esempio di NSD semplice relativo ad un Network Security Service (Parte 2/2).

B.5 OSM vs Docker: vim-emu

La prima fase per l'installazione di OSM e vim-emu contemporaneamente, per uno scenario di test richiede la presenza di alcuni prerequisiti. Poiché tali prerequisiti sono in linea con quelli già discussi in precedenza rimandiamo alle sezioni [B.1.1](#) e [B.1.2](#) per la loro installazione.

Installazione di OSM e vim-emu

Una volta effettuate tutte le operazioni preliminari descritte si può procedere all'installazione congiunta di OSM e vim-emu attraverso i comandi:

```
# effettuare il download tipicamente nella home
$ wget -O install_osm.sh "https://osm.etsi.org/gitweb/?p=osm/devops.git;
  a=blob_plain;f=installers/install_osm.sh;hb=HEAD"
$ chmod +x install_osm.sh

# utilizzare il comando sudo per ottenere i privilegi per poter utilizzare
  Docker
$ sudo ./install_osm.sh --lxdimages --vimemu
```

Lo script appena utilizzato, sfrutta un playbook ansible per effettuare il deploy di vim-emu in un container Docker. Esiste una versione di tale playbook per CentOS, nel caso si fosse interessati ad un'alternativa a ubuntu e debian. In questo ultimo caso fare riferimento alla guida per l'installazione indipendente del solo vim-emu (bare-metal installation) al link https://osm.etsi.org/wikipub/index.php/VIM_emulator#Option_1:_Bare-metal_installation, avendo cura di sostituire il playbook Ansible con quello per CentOS, in questo caso si evita anche il problema di avere vim-emu in un docker in quanto viene installato direttamente sull'host.

Una volta installato OSM e vim-emu si può verificare la corretta installazione delle due componenti, prima utilizzando il seguente comando e avendo un risultato simile a quello riportato:

```
ignazio@ubuntu-linux:~$ lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
RO	RUNNING	10.212.6.4 (eth0)		PERSISTENT	0
SO-ub	RUNNING	10.212.6.116 (eth0)		PERSISTENT	0
VCA	RUNNING	10.44.127.1 (lxdbr0)		PERSISTENT	0
		10.212.6.112 (eth0)			

Successivamente attraverso il controllo della presenza del container vim-emu sul Docker Engine:

```
ignazio@ubuntu-linux:~$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND
1d1e4193dc3d	vim-emu-img	"/son-emu/utils/dock..."

B.5.1 Operazioni preliminari con vim-emu

Collegare vim-emu con OSM

Innanzitutto dopo la fase d'installazione, il container Docker con vim-emu viene lanciato direttamente. Per poterlo riavviare dopo un eventuale reboot occorre procedere con il seguente comando:

```
$ sudo docker run --name vim-emu -t -d --rm --privileged --pid='host' -v
/var/run/docker.sock:/var/run/docker.sock vim-emu-img python
examples/osm_default_daemon_topology_2_pop.py
```

Per configurare le variabili d'ambiente, invece, necessarie alle operazioni su vim-emu e sulla CLI di Open Source MANO bisogna eseguire i seguenti comandi:

```
$ export OSM_HOSTNAME=$(lxc list | awk '($2=="SO-ub"){print $6}')
$ export OSM_RO_HOSTNAME=$(lxc list | awk '($2=="RO"){print $6}')
$ export VIMEMU_HOSTNAME=$(sudo docker inspect -f '{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' vim-emu)
```

Una volta avviato vim-emu e settate le variabili d'ambiente passiamo al verificare che siano stati creati i due datacenter su vim-emu:

```
$ sudo docker exec vim-emu vim-emu datacenter list
```

Questo comando ci mostrerà i datacenter attivi. Ora possiamo procedere con il collegamento di vim-emu ad OSM, il tutto utilizzando la CLI come riportato di seguito:

```
$ osm vim-create --name emu-vim1 --user username --password password
--auth_url http://$VIMEMU_HOSTNAME:6001/v2.0 --tenant tenantName
--account_type openstack
```

On-boarding e istanziazione di un servizio di rete

Per effettuare l'on-boarding un servizio di rete, in questo scenario di test, occorre far riferimento ai descriptor presenti sotto la cartella “vim-emu” creata dallo script di installazione nella home dell'host. Prima si effettua quello delle VNF e successivamente quello del servizio di rete:

```
# onboarding VNFD
$ osm upload-package vim-emu/examples/vnfs/ping.tar.gz
$ osm upload-package vim-emu/examples/vnfs/pong.tar.gz

# onboarding NSD
$ osm upload-package vim-emu/examples/services/pingpong_nsd.tar.gz
```

Queste operazioni permettono di inserire i descriptor delle VNF e del servizio di rete nei cataloghi di OSM. Si può verificare la loro presenza con i comandi:

```
$ osm vnfd-list
$ osm nsd-list
```

A questo punto è possibile istanziare il servizio di rete con il comando:

```
$ osm ns-create --nsd_name pingpong --ns_name test --vim_account emu-vim1
```

Ora verranno creati i docker per ospitare le VNF desiderate e collegati alle interfacce definite nei descrittori. Tutte le operazioni possono essere eseguite in modo simile attraverso l'interfaccia grafica di OSM.

Possiamo ora accedere alle VNF e verificare che le stesse abbiano la connettività desiderata:

```
$ sudo docker exec -it mn.dc1_test.ping.1.ubuntu /bin/bash
$ ifconfig
```

Vedremo che esiste un'interfaccia "eth0" che corrisponde al collegamento con la rete di default di Docker, ovvero quella bridge. A tale rete viene collegato il container al momento della sua creazione. Una seconda rete è stata definita attraverso le API di vim-emu che consentono, come abbiamo visto in precedenza la creazione di PoP (datacenter). Abbiamo altresì visto che sono stati creati due datacenter appartenenti a vim-emu e i container definiti nel descrittore NSD appena lanciato afferiscono ad uno di questi due. L'interfaccia "ping0-0" rappresenta il collegamento a questa rete e permette ai due container lanciati nel PoP di comunicare.

Per verificare che la connessione sia stata instaurata correttamente, si può effettuare il ping all'altro host. Per terminare il servizio lo si può fare attraverso la UI di OSM o attraverso i comandi:

```
$ osm ns-delete test
$ sudo docker stop vim-emu
```

B.5.2 Possibili estensioni alla nostra soluzione

La nostra soluzione prevede l'utilizzo di servizi di sicurezza definiti con i descrittori di Open Source MANO. Con il framework appena presentato abbiamo effettuato dei test sulla fattibilità del porting dei descrittori nel caso di emulazione su container. Abbiamo creato un descrittore simile a quelli utilizzati per il nostro sistema, utilizzando le funzioni di rete ping e pong e provato se fosse possibile effettuare il deploy delle VNF in modo automatico su container Docker.

Il servizio di rete più semplice è quello costituito da una semplice catena con tre docker, rappresentato in figura [B.12](#). Questo è stato avviato con successo e i docker sono stati creati attraverso il Docker Engine presente sull'host.

Le problematiche relative al vim-emu rispetto alla nostra soluzione riguardano la gestione della rete. Le API di vim-emu creano attraverso containernet dei PoP che sono forniti di una sottorete a cui i Docker che afferiscono a quel PoP vengono collegati. Quindi esiste la possibilità utilizzando l'emulatore per com'è concepito ora di avere un'unica interfaccia docker, e tutti quelli appartenenti al PoP condividano la stessa sottorete.

Per ovviare a questo problema è necessario effettuare delle modifiche sulla topologia di rete che viene caricata al momento dell'inizializzazione del Docker contenente vim-emu. Questo però non rappresenta una soluzione dinamica, perché andrebbe ricaricato vim-emu ad ogni deploy di servizio di sicurezza. Una prospettiva per il futuro potrebbe essere quella di scendere nel dettaglio su come

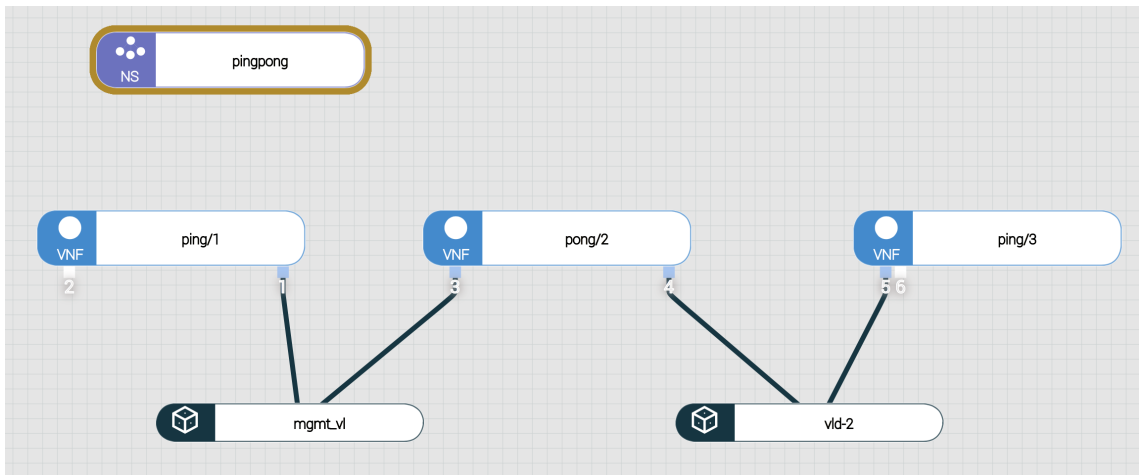


Figura B.12: Esempio di servizio di rete testato con vim-emu.

containernet gestisce le topologie di rete. Riuscendo a gestire attraverso le API di containernet le topologie create da mininet, è teoricamente possibile creare in modo automatico (con degli script in python) le reti necessarie al deploy del servizio di sicurezza, prima che vengano istanziati i Docker.

In particolare potresti creare tutti gli switch necessari per simulare i link tra le varie VNF e gestire il forwarding dei pacchetti attraverso l'SDN controller ryu (presente nella demo), questo sostituirebbe la fase che in OSM è delineata come creazione dei Virtual Link (VLs). Una volta fatta questa operazione, indipendentemente da OSM, si potrebbe procedere all'utilizzo di OSM per orchestrare le VNF e il servizio di sicurezza desiderato.

Il problema di strumenti come OSM è il loro stato embrionale, infatti, lo stesso si pone come una soluzione capace di eseguire qualsiasi servizio di rete a partire da un forwarding graph, gestendo le funzioni di rete e il dataplane con un SDN Controller, in modo del tutto automatico. Praticamente questo però non è ancora possibile senza effettuare una serie di assunzioni e configurazioni ad hoc per poter testare scenari di servizi complessi.

Questo ci suggerisce che per poter ottenere le funzionalità desiderate, in scenari di ricerca, occorrerebbe valutare l'ipotesi di estendere questo strumento con tecnologie affini, per poter controllare in modo più flessibile l'automatizzazione di alcuni task (es. il deploy di servizi di rete con più flussi e service function chain).

B.5.3 Strumenti di supporto: son-emu

Abbiamo visto come il framework di partenza di vim-emu sia quello di Sonata (son-emu). Andiamo ad illustrare uno scenario di esempio che mostra, da una parte, come sia possibile integrare altri strumenti a son-emu (e di conseguenza a vim-emu), dall'altra, come sia possibile emulare un servizio di rete concreto.

L'esempio parte dal sito del progetto [19] e prevede l'installazione di una macchina virtuale preconstituita (disponibile sul sito del progetto). Tale macchina virtuale ha a disposizione al suo interno OSM release TWO, containernet e degli strumenti che coadiuvano son-emu. Il servizio di rete che si intende lanciare è rappresentato in figura B.13, questo prevede l'utilizzo di un cache proxy, un L4 forwarder e un servizio web (in questo caso streaming video). Per simulare il servizio, per prima cosa bisogna avviare il terminale e creare la topologia di rete attraverso containernet. Per fare questo viene eseguito lo script:

```
$ sudo python demo/demo_topology.py
```

Successivamente va impostata la route per accedere alla UI di OSM 2 (installazione nested LXD):

```
$ source demo/osm/set_route.sh
```

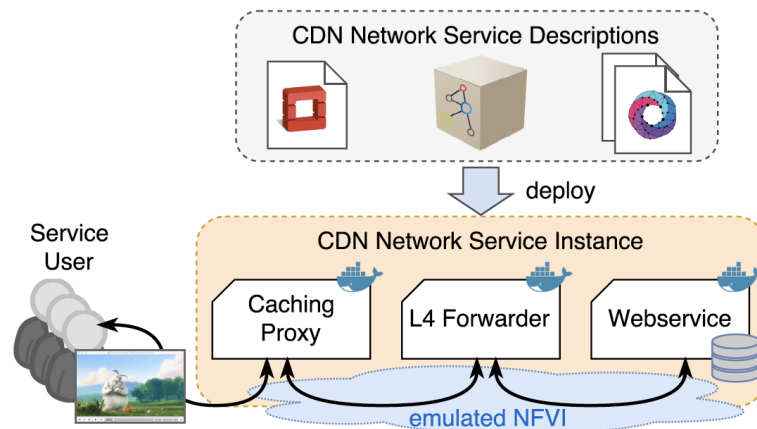


Figura B.13: Servizio di rete Sonata.

A questo punto si può avviare il browser e avviare l'interfaccia di OSM all'indirizzo:

<https://10.87.78.189:8443>

Ora procedere all'avvio del servizio di rete presente nel catalogo al nome “demo”, facendo attenzione a collocare la prima e la terza VNF nel datacenter “pop1” e la seconda nel “pop2”. È possibile dopo le precedenti operazioni, controllare la corretta inizializzazione delle VNF attraverso la dashboard come in figura B.14, all'indirizzo:

127.0.0.1:5001/dashboard

Come si evince dalla figura B.14, sono stati correttamente allocati i docker nei rispettivi datacenter emulati. Ora possiamo avviare lo strumento di monitoraggio con il comando:

```
$ sudo son-monitor msd -f /demo/osm/msd-osm-gui.yml
```

Emulated Datacenters 4					Latency: 0.53s
Label	Int. Name	Switch	Num. Containers	VNFs	
sonata-pop11	dc1	dc1.s1	0		
sonata-pop21	dc2	dc2.s1	0		
osm-pop21	dc4	dc4.s1	1	http	
osm-pop11	dc3	dc3.s1	2	l4fw,proxy	

Running Containers 3					Latency: 1.9s
Datacenter	Container	Image	docker0	Status	
osm-pop2	http	http-apache-img	172.17.0.4	running	
osm-pop1	l4fw	l4fw-socat-img	172.17.0.5	running	
osm-pop1	proxy	proxy-squid-img	172.17.0.6	running	

(c) 2017 by SONATA Consortium, Paderborn University and IMEC

Figura B.14: Dashboard Sonata Emulator.

e controllare il corretto avvio come in figura B.15 all'indirizzo:

127.0.0.1:3000/dashboard

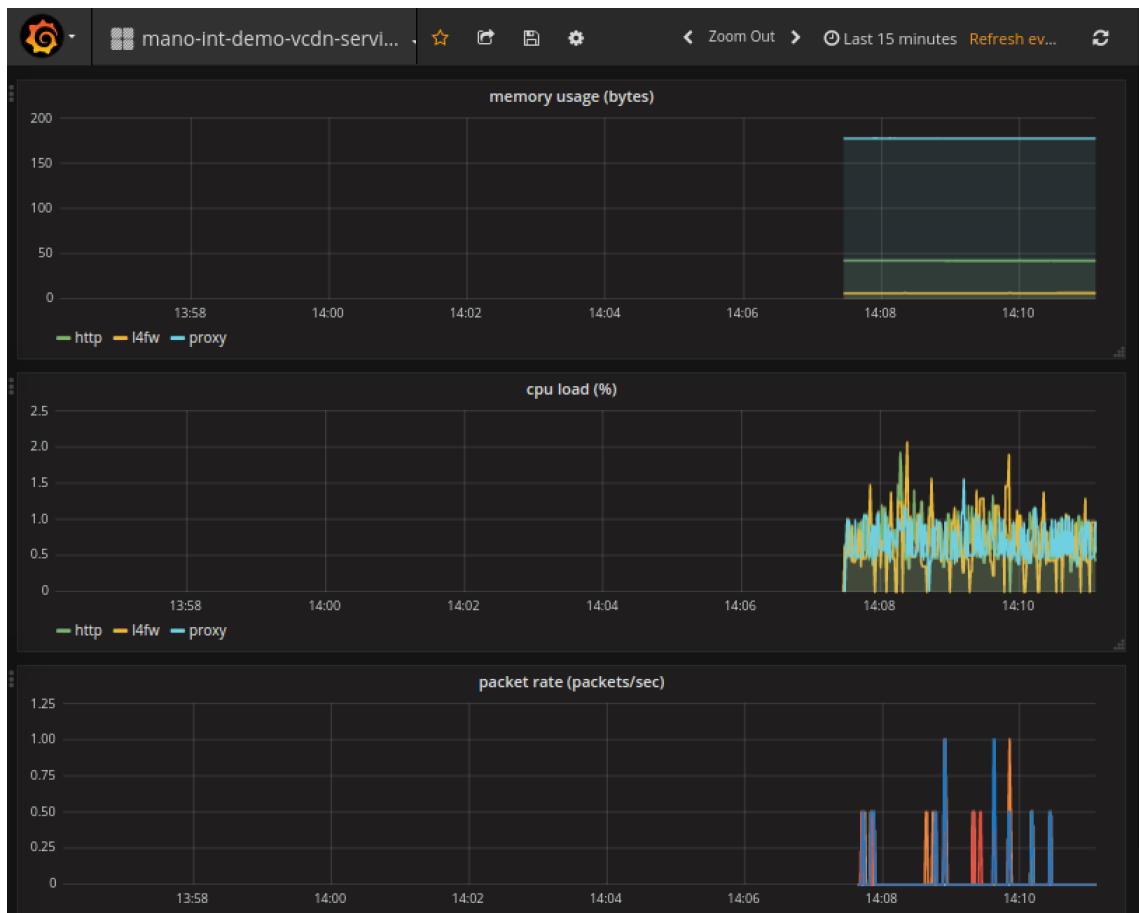


Figura B.15: Monitor Sonata Emulator prima dell'utilizzo del servizio.

Ora non ci resta che utilizzare il servizio e valutarne le prestazioni attraverso il monitor (basato su graphena e prometheus). Avviamo il browser all'indirizzo:

127.0.0.1:3000/dashboard

troviamo il servizio di web streaming e procediamo all'utilizzo caricando un video. La variazione di carico e di prestazioni è illustrata in figura B.16.

La ragione della presentazione di questa demo, del progetto sonata risiede nel fatto che un approccio basato su queste tecnologie può essere di interesse per lo sviluppo e il test di servizi di sicurezza basati su container. Questo esempio fornisce numerosi strumenti sia per il monitoraggio del servizio che per la creazione di topologie e scenari diversi. Si ritiene, quindi, che uno sviluppo futuro interessante potrebbe essere l'utilizzo di questo framework per sviluppare e testare in modo automatico servizi di sicurezza.

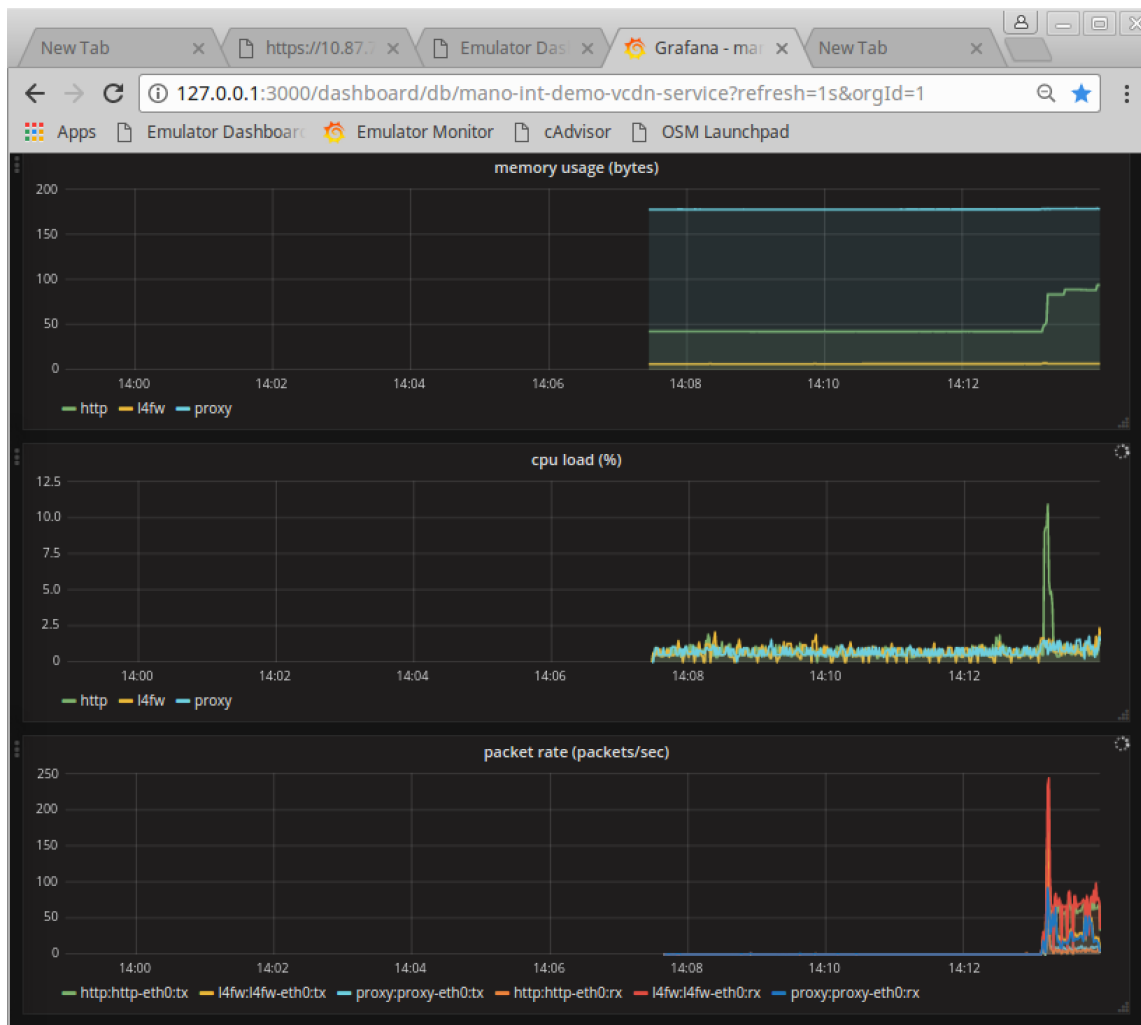


Figura B.16: Monitor Sonata Emulator durante l'utilizzo del servizio.

B.6 Juju

B.6.1 UI

Abbiamo parlato diffusamente di quali sono i meccanismi di funzionamento di Juju nei capitoli precedenti (cap. 6,7 e 8). Andiamo adesso, con un approccio più funzionale, a vedere com'è possibile utilizzarlo in modo scorrelato da OSM e poi ad integrarlo in quest'ultimo. Per descrivere le funzionalità dell'interfaccia grafica, faremo un esempio pratico di deploy di un bundle, in particolare Kubernetes. Kubernetes è un sistema di orchestrazione per container, che deriva da Google Borg, in questo caso ci interessa poco la sua funzione, ma vogliamo vedere com'è possibile istanziare velocemente il servizio e configurare ogni sua parte. Questo ci permetterà da una parte di esplorare la UI e dall'altra di vedere cosa effettivamente può fare uno strumento come Juju. Possiamo effettuare delle prove direttamente dalla versione di Juju installata nel container VCA:

```
$ lxc exec VCA -- juju gui
```

questo comando ci mostra come accedere all'interfaccia grafica di Juju, nel nostro caso siamo in un ambiente di sviluppo e non abbiamo sostituito il segreto per l'accesso a tale interfaccia. Lo si può reperire attraverso la CLI di OSM:

```
$ osm config-agent-list
```

Questo restituisce tutti i dettagli di Juju installata nel container VCA e gli strumenti per accedervi. Una volta ottenute tutte le informazioni necessarie per l'accesso, seguiamo con il login e esploriamo le funzionalità dell'interfaccia.

In figura [B.17](#), viene mostrata l'interfaccia grafica di Juju. Come si evince dall'immagine nella parte centrale vi è la possibilità di modellare graficamente un servizio; corrisponde alla possibilità di aggiungere e modificare manualmente le relazioni con charm e bundle scaricati dallo store di Juju. Per scaricare charm o bundle, è possibile effettuare una ricerca direttamente nello store con la barra in alto a destra. Una volta individuato il charm/bundle di interesse è possibile aggiungerlo al modello corrente. In figura [B.17](#) è stato scelto il bundle che permette il deploy del servizio Kubernetes, nella versione della Canonical. Questo prevede il deploy dei seguenti microservizi:

- **easysrsa:** funge da Certification Authority.
- **etcd:** è un key-value store distribuito e ne vengono eseguite 3 unità.
- **flannel:** fornisce una overlay network per la connessione dei container.
- **load-balancer:** fornisce funzionalità di load-balancing tramite nginx.
- **kubernetes-master:** nodo del cluster che coordina i workers.
- **kubernetes-worker:** nodo di elaborazione che contiene i pod (unità elementare che può essere gestite e contiene un certo numero di container) in esecuzione (3 unità).

Non siamo molto interessati alle funzionalità specifiche di Kubernetes, ci interessava solo contestualizzare il modello, del quale abbiamo fatto il deploy, per avere idea delle potenzialità di Juju. Come vediamo sempre in figura [B.17](#), il modello è costituito dai charm (nodi del grafo) corrispondenti ai microservizi citati sopra e delle relazioni che sono i link tra gli stessi. L'insieme di charm e link forma il bundle. Ora spostiamo la concentrazione sulla parte sinistra della stessa immagine, dove vediamo tre menu a disposizione:

- **Applications:** rappresenta tutte le applicazioni disponibili nel modello (in questo caso sono quelle del bundle) ed è possibile interagirvi e configurarle come vedremo più avanti.
- **Machines:** fornisce un resoconto delle macchine a disposizione per il deploy delle unità, da questo menu è possibile effettuare operazioni sulle stesse.
- **Status:** fornisce una visione di insieme su tutte le entità fondamentali di Juju (Applicazioni, Unità, Macchine e Relazioni) come riportato in fig. [B.19](#).

Ogni applicazione può essere esplorata e configurata attraverso il menu application. Selezionando una di quelle presenti in figura [B.17](#), è possibile ottenere il menu sulla sinistra in figura [B.18](#). Nel caso specifico si tratta dell'applicazione easysrsa ed è possibile visualizzarne le informazioni come riportato nel menu a destra della fig [B.18](#) o eseguire altre operazioni. Ad esempio è possibile configurare dei parametri della stessa, utili al ciclo di vita del charm, in figura [B.19](#) in basso a sinistra è riportato l'esempio di parte della configurazione del load-balancer nginx. Ancora si possono definire delle relazioni e vedere quali sono quelle disponibili, esporre la risorsa pubblicamente, visualizzare le risorse che utilizza o infine aggiornare la versione del charm all'ultima disponibile.

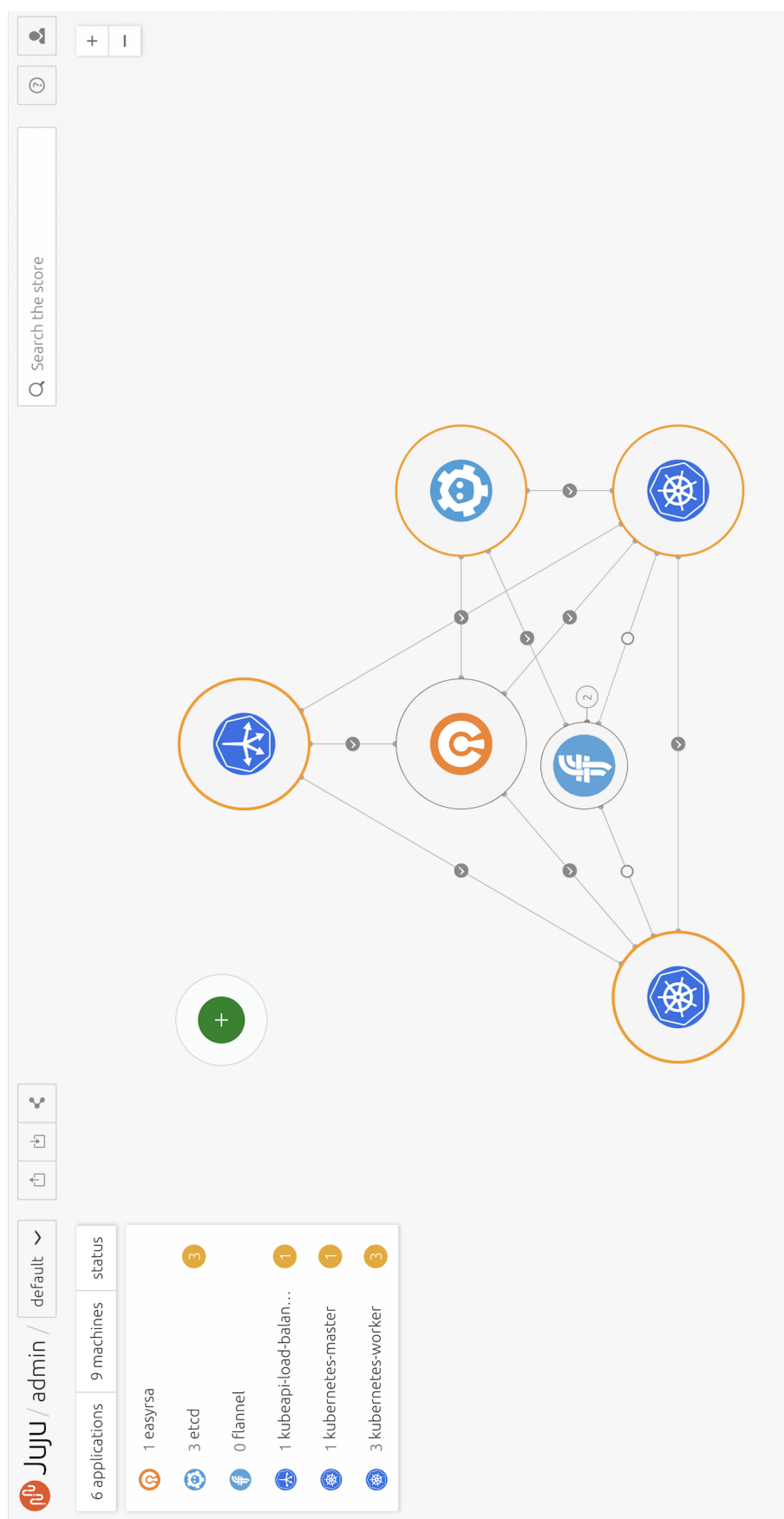


Figura B.17: Interfaccia grafica Juju ed esempio di deploy di Kubernetes.

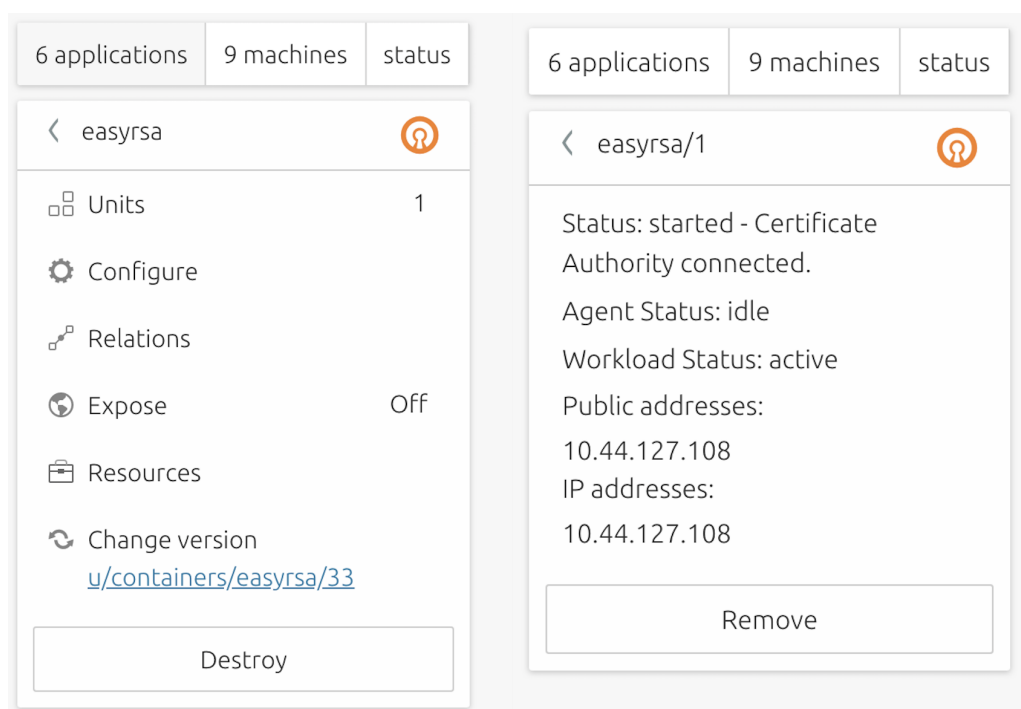


Figura B.18: Esempio di menu per informazioni e configurazioni di una singola app (easysrsa).

juju

admin / default

6 applications

9 machines

status

1 easyrsa

3 etcd

0 flannel

1 kubeapi-load-balan...

1 kubernetes-master

3 kubernetes-worker

default

Cloud/Region

Version

SLA

Applications

Remote applications

Units

Machines

Relations

localhost/localhost

2.2.6

6

0

9

13

Application

Version

Status

Scale

Charm

Store

Rev

easyrsa

active

1

u/containers/easyrsa

jujuharms

33

etcd

waiting

3

u/containers/etcd

jujuharms

74

flannel

0

u/containers/flannel

jujuharms

50

kubeapi-load-balancer

1

u/containers/kubeapi-load-balancer

jujuharms

55

kubernetes-master

maintenance

1

u/containers/kubernetes-master

jujuharms

97

kubernetes-worker

3

u/containers/kubernetes-worker

jujuharms

106

Unit

Workload

Agent

Machine

Public address

Ports

Message

easyrsa/1

active

idle

7

10.44.127.108

Certificate Authority ready.

etcd/1

maintenance

executing

4

10.44.127.38

installing charm software

etcd/2

waiting

allocating

6

waiting for machine

Configure

Space separated list of extra deb packages to install.
extra_sans (string)

Space separated list of extra SAN entries to add to the x509 certificate created for the load balancers.
host (string)

127.0.0.1

listen address

install_keys (string)

List of signing keys for install. sources package sources per charmhelpers standard format (a yaml list of strings encoded as a string). The keys should be the full ASCII armoured GPG public keys. While GPG key ids are also

Figura B.19: Menu che consente di visionare lo stato delle entità fondamentali di Juju (Applicazioni, Unità, Macchine e Relazioni).

B.6.2 CLI

Per quanto riguarda la CLI, ovviamente risulta possibile effettuare ogni operazione che abbiamo visto nella sezione UI attraverso quest'ultima. La CLI di Juju ha uno sconfinato numero di comandi che è possibile reperire (sempre nel contesto OSM) con:

```
$ lxc exec VCA -- juju help commands
```

Questo permette di avere una complessiva panoramica di tutto ciò che offre questa Command Line Interface. In ogni caso noi siamo interessati per i nostri scopi ad un sottoinsieme di tali comandi che elencheremo e definiremo di seguito. Innanzitutto è possibile eseguire i comandi direttamente all'interno del modulo VCA accedendovi e antepoendo ad ognuno dei comandi Juju (subito dopo vedremo i comandi di interesse):

```
$ lxc exec VCA -- bash
```

run-action

Permette di inserire un'azione di uno specifico charm nella coda di esecuzione di Juju. È necessario specificare l'unità sulla quale agire, in quanto potrebbero esserci più istanze dello stesso charm (per via della possibilità di scalare).

Esempio 1:

```
$ juju run-action mysql/3 backup --wait
```

In questo caso è possibile eseguire il backup del DB MySQL specificando l'unità e, con l'opzione `wait`, è possibile aspettare il risultato dell'azione. In questo caso otteniamo `action-id: ID`, `status: success`, `file_size: 873.2`, `units: GB` e `name: foo.sql`.

Esempio 2: `$ juju run-action mysql/3 backup --params parameters.yml`

Stessa funzione precedente, ma in questo caso è stato passato un file di parametri in formato `yaml`, ovviamente gli stessi parametri potrebbero essere inseriti da UI.

list-actions

Questo comando permette di ritornare la lista di azioni disponibili su una specifica applicazione. Va fornito il nome dell'applicazione come parametro. Questo è un alias del comando `actions` che svolge la funzione analoga.

Esempio:

```
$ juju actions git
```

show-action-output

Mostra l'output delle azioni nella coda, è possibile filtrare il risultato con l'ID dell'azione.

show-action-status

Mostra lo stato delle azioni, è possibile filtrare il risultato come nel caso precedente.

list-machines

Ritorna la lista di tutte le macchine fisiche o virtuali a disposizione.

list-controllers

Ritorna tutti i controller, che ricordiamo sono univocamente associati ad un cloud.

add-unit

Aggiunge una nuova unità ad una macchina.

add-machine

Aggiunge una nuova macchina.

remove-unit

Rimuove un'unità da una specifica macchina, va passato come parametro l'unità.

remove-application

Rimuove un'applicazione da un modello. Va passato come parametro l'applicazione.

remove-machine

Rimuove un'intera macchina, a prescindere dalla tecnologia utilizzata. Va passato come parametro la macchina.

scp

Permette di trasferire dei file provenienti o diretti verso una macchina Juju.

ssh

Permette di inizializzare una sessione ssh con una delle macchine istanziate da Juju.

ssh-keys

Restituisce tutte le chiavi ssh conosciute per il modello corrente.

subnets

Restituisce tutte le sottoreti conosciute da Juju.

deploy

Permette di effettuare il deploy di un'applicazione o di un bundle.

Esempi:

```
$ juju deploy mysql # 1
$ juju deploy mysql --to 23 # 2
$ juju deploy mysql --to lxd # 3
```

```
$ juju deploy mysql --to lxd:25 # 4
$ juju deploy mysql --to 24/lxd/3 # 5
$ juju deploy mysql -n 5 --constraints mem=8G # 6
```

Nel primo esempio viene effettuato semplicemente il deploy di un'applicazione su una nuova macchina. Nel secondo esempio viene effettuato il deploy dell'applicazione su una macchina preesistente (la 23). Nel terzo esempio viene effettuato il deploy su un nuovo container LXD su una nuova macchina. Il quarto esempio è uguale al terzo, a differenza dello specificare quale macchina utilizzare per istanziare il container LXD. L'esempio 5 permette di selezionare sia il container che la macchina. L'ultimo esempio (6) mostra com'è possibile inserire dei vincoli e scalare l'applicazione; in particolare viene effettuato il deploy di 5 istanze della stessa applicazione e creato un vincolo per la memoria ad un limite di 8 GB.

B.6.3 Juju & OSM: Proxy Charm

Abbiamo visto nei capitoli precedenti come si cala Juju nel contesto di OSM (cap. 6,7 e 8). Adesso vediamo praticamente come abbiamo sviluppato alcune parti fondamentali del Proxy Charm nel caso di Packet Filter. Questo ci darà un’idea di come sfruttare le caratteristiche dei charm presentate nel capitolo 6 come i layered charm, il reactive programming e la combinazione di entrambi per ottenere un charm flessibile che può essere potenzialmente esteso per ogni tipo di vNSF.

metadata

Iniziamo a guardare il cuore del proxy charm, ovvero il file metadata.yaml. Questo file, come si vede chiaramente in figura B.20, contiene tutti i dettagli identificativi del charm stesso. In particolare il nome, un eventuale sommario, lo sviluppatore e una descrizione.

Subito dopo sono presenti i tag che permettono di ritrovarlo all’interno dello store di Juju. Ancora sono presenti sotto la voce “series” le distribuzioni di linux compatibili come guest os e l’indicazione del subordinamento o meno del charm (capitolo 6).

```
"name": "iptables"
"summary": "<Fill in summary here>"
"maintainer": "ignazio.pedone@studenti.polito.it"
"description": |
  <Multi-line description here>
"tags":
  # Replace "misc" with one or more whitelisted tags from this list:
  # https://jujucharms.com/docs/stable/authors-charm-metadata
  - "misc"
  - "osm"
  - "vnf"
"series":
- "trusty"
- "xenial"
"subordinate": !!bool "false"
```

Figura B.20: File metadata.yaml del proxy charm Packet Filter.

layers

Il discorso relativo ai layer è stato affrontato nel capitolo 6. Ora vediamo concretamente com’è possibile riutilizzare delle librerie appartenenti a livelli precostituiti e a disposizione su un repository. In questo caso con il file layer.yaml in figura B.21, andiamo a definire i livelli di cui abbiamo bisogno per il nostro charm. In particolare è possibile specificare sotto la voce “includes” tutti i layer di interesse, nel nostro caso basic (quello che nel capitolo 6 abbiamo definito come infrastruttura di base per le operazioni di routine del charm), sshproxy (una libreria per il collegamento remoto ssh) e vnfproxy (che è un template per definire una libreria di supporto per il nostro proxy charm).

Una volta definiti i livelli è possibile inserire il repository di riferimento e la denominazione nello stesso del charm. Per ognuno dei livelli, incluso il layer aggiuntivo “iptables” che costituisce il nostro spazio di lavoro è possibile inserire delle opzioni. In questo caso vediamo che per il livello base ne sono state definite alcune, in particolare non sono stati aggiunti pacchetti all’unità di base, infatti, “packages” risulta vuota; non è stato previsto l’utilizzo di un virtual environment, in caso positivo sarebbe stato possibile dare all’ambiente virtuale accesso ai pacchetti del sistema tra cui python attraverso il successivo flag.

```
"options":
  "iptables": {}
  "sshproxy": {}
  "vnfproxy": {}
  "basic":
    "use_venv": !!bool "false"
    "packages": []
    "include_system_packages": !!bool "false"
  "includes":
    - "layer:basic"
    - "layer:sshproxy"
    - "layer:vnfproxy"
  "repo": "https://osm.etsi.org/gerrit/osm/juju-charms"
  "is": "iptables"
```

Figura B.21: File layers.yaml del proxy charm Packet Filter.

Sempre in merito ai layered charm, guardiamo quali sono alcune delle funzioni di interesse, offerte dal layer sshproxy e da estendere eventualmente per l'utilizzo nel nostro proxy charm, nella figura B.22. Vediamo due funzioni, in particolare una che ci consente di copiare un file dal charm stesso ad una locazione remota, un'altra, invece, che ci consente di eseguire un comando attraverso un canale sicuro ssh su una macchina remota. Queste due funzionalità sono di base ed estremamente utili per la comunicazione con le vNSF. In accordo con la teoria, infatti, abbiamo necessità di gestire le nostre funzioni virtuali di sicurezza attraverso un canale sicuro ssh e da remoto quindi c'è necessità di un insieme di funzioni che ne regoli il funzionamento. Queste sono date dal layer sshproxy e quelle presentate sono due delle fondamentali.

Entrambe le funzioni importano la libreria python "paramiko" che gestisce un canale SSHv2, esponendo un insieme di funzionalità. Nella prima funzione utilizziamo l'oggetto client e la relativa funzione `get_ssh_client` per ottenere tale oggetto. Questo modella l'entità client e permette di creare a partire dallo stesso un canale sicuro per lo scambio di file. Ad esempio sempre nella prima funzione, una volta ottenuto il canale sftp, tramite la funzione `put` possiamo inviare un file da locale (inteso come proxy charm) a remoto (inteso come vNFS).

La seconda funzione è molto simile, a differenza dello scopo, ovvero anziché del trasferimento del file si occupa dell'esecuzione di un comando. Tutto questo tramite la medesima libreria e la funzione `exec_command`.

```
def sftp(local_file, remote_file, host, user, password=None, key=None):
    """Copy a local file to a remote host."""
    client = get_ssh_client(host, user, password, key)

    # Create an sftp connection from the underlying transport
    sftp = paramiko.SFTPClient.from_transport(client.get_transport())
    sftp.put(local_file, remote_file)
    client.close()

def ssh(cmd, host, user, password=None, key=None):
    """Run an arbitrary command over SSH."""
    client = get_ssh_client(host, user, password, key)

    cmds = ' '.join(cmd)
    stdin, stdout, stderr = client.exec_command(cmds, get_pty=True)
    retcode = stdout.channel.recv_exit_status()
    client.close() # @TODO re-use connections
    if retcode > 0:
        output = stderr.read().strip()
        raise CalledProcessError(returncode=retcode, cmd=cmd,
                                output=output)

    return (
        stdout.read().decode('utf-8').strip(),
        stderr.read().decode('utf-8').strip()
    )
```

Figura B.22: Estratto dalle librerie del layer sshproxy.

actions

Ora passiamo all'elemento fondamentale, che utilizziamo per definire il comportamento delle nostre vNSF a seguito di alcuni eventi. Questo meccanismo viene modellato in parte con il meccanismo delle action e in altra parte con il reactive programming. Innanzitutto possiamo attraverso il file action.yaml, in figura B.23, andare a dichiarare le nostre azioni (in accordo con la definizione data nel capitolo 6). Definiamo tutta una serie di azioni e le relative descrizione, ma ci concentriamo su due in particolare: start e start-configuration.

Dichiarare le azioni mi permette di mapparle successivamente in OSM e utilizzarle attraverso la GUI di quest'ultimo. Ma quello che vogliamo è associare anche un comportamento all'evento scatenato dall'azione. In figura B.24, abbiamo un esempio di handler dell'evento relativo all'azione start. Una volta che l'evento viene scatenato, questo codice, presente nell'apposito file start.yaml sotto la cartella actions, viene eseguito. In questo caso come si evince dalla figura, l'unica azione, escludendo gli import del caso e la gestione delle eccezioni, è quella del settare il flag relativo allo stato dell'azione start a TRUE. Questo perché non stiamo utilizzando direttamente gli handler ma vogliamo avere la possibilità di combinare più eventi in modo flessibili, di conseguenza sfruttiamo il reactive programming. Per ora l'unica cosa che ci interessa è settare questo flag/evento a TRUE in risposta all'azione, nella sezione B.6.3 vedremo come gestire il tutto.

```
"start":
  "description": "Start configurations."
"stop":
  "description": "Stop the service on the VNF."
"restart":
  "description": "Stop the service on the VNF."
"reboot":
  "description": "Reboot the VNF virtual machine."
"upgrade":
  "description": "Upgrade the software on the VNF."
"pass-files":
  "description": "Pass configuration files through ssh"
"start-configuration":
  "description": "Start configuration from files passed before"
```

Figura B.23: Estratto dal file action.yaml del proxy charm Packet Filter.

```
#!/usr/bin/env python3
import sys
sys.path.append('lib')

from charms.reactive import main
from charms.reactive import set_state
from charmhelpers.core.hookenv import action_fail

"""
'set_state' only works here because it's flushed to disk inside the 'main()'
loop. remove_state will need to be called inside the action method.
"""

set_state('actions.start')

try:
    main()
except Exception as e:
    action_fail(repr(e))
```

Figura B.24: Estratto del file eseguibile relativo all'azione Start del proxy charm Packet Filter.

reactive code

Siamo arrivati a questo punto, a partire dalla definizione di azioni e relativi handler nelle sezioni precedenti. Una volta effettuato questo, per sfruttare il reactive programming, dobbiamo utilizzare i flag, attivati tramite gli handler delle azioni e costituenti i nostri eventi finali, per eseguire del codice in modo opportuno e flessibile, in funzione della combinazione di uno o di diversi eventi. In particolare, come visto nel capitolo 6, posso con la clausola @when controllare lo stato di un oggetto, nel nostro caso TRUE o FALSE, per verificare se quell'evento è scattato o meno.

Nell'esempio B.25 vediamo una dimostrazione pratica. Le azioni start e start-configuration sono state definite e collegate ai relativi handler che settano gli eventi `actions.start` e `actions.start-configuration` nella sezione precedente. Ora controlliamo delle combinazioni di tali eventi ed eseguiamo del codice in risposta.

In particolare la prima funzione, quando la configurazione del proxy charm iptables è andata a buon fine e viene richiesta l'azione start da OSM, esegue il codice relativo. In questo caso si tratta

```

@when('iptables.configured')
@when('actions.start')
def start():
    try:
        cmd = "source scripts/startConfig.sh"
        result, err = charms.sshproxy._run(cmd)
    except Exception as e:
        action_fail('command failed: {}'.format(e, e.output))
    else:
        action_set({'stdout': result,
                    'errors': err})
    finally:
        remove_flag('actions.start')

@when('iptables.configured')
@when('actions.start-configuration')
def start_configuration():
    try:
        cmd = "source iptables.sh"
        result, err = charms.sshproxy._run(cmd)
    except Exception as e:
        action_fail('command failed: {}'.format(e, e.output))
    else:
        action_set({'stdout': result,
                    'errors': err})
    finally:
        remove_flag('actions.start-configuration')

```

Figura B.25: Reative programming: un esempio nel proxy charm Packet Filter.

dell'esecuzione di un comando sulla vNSF che ad inizio alla sua configurazione.

Nel caso della seconda funzione, abbiamo un comportamento analogo. All'occorrenza degli eventi `iptables.configured` e `actions.start-configuration`, viene eseguito il relativo comando sulla vNSF, in questo caso inizializzare le configurazioni di secondo livello (non relative alla macchina virtuale ma alle funzione di sicurezza). Ovviamente in ambedue le funzioni vengono gestite le eccezioni; nel caso in cui il comando non vada a buon fine, infatti, viene notificato all'ambiente Juju e successivamente ad OSM. Ancora, si noti come sia stata utilizzata la libreria `sshproxy` e come vada ridefinito a `FALSE` il flag dell'azione appena eseguita (questo nel blocco `finally`).

Per eccesso di zelo, sottolineo che queste funzioni non intendono spiegare in ogni dettaglio qual'è il comportamento della funzione di sicurezza Packet Filter, ma dare un'idea di come si possa creare il meccanismo di azione/reazione alla base delle configurazioni Day-1 attraverso Juju. A questo punto dovrebbero esserci tutti gli elementi per poter sviluppare un charm, per la completa documentazione delle funzioni si rimanda al codice del progetto, compreso nell'ulteriore documentazione allegata.

B.7 Operazioni utili per sviluppi futuri

B.7.1 Integrazione SDN Controller (SDN assist/EPA)

Parliamo ora della possibile integrazione di un SDN controller nell'ambiente OSM. Abbiamo discusso, nei capitoli di teoria (cap. 6 e 9), delle soluzioni SDN compatibili con OSM: OpenDaylight, ONOS e floodlight. Questo ci dà un'informazione di massima su quali tecnologie utilizzare per il

control plane. Di particolare interesse risulta presentare come OSM sfrutta le tecnologie SR-IOV ed EPA al fine di integrare automaticamente un SDN Controller.

In sostanza dalla release TWO di OSM è stata introdotta una funzionalità chiamata *SDN assist*. In breve, quando OSM istanzia un servizio di rete su un compute node del VIM (ad. es OpenStack) e tale VIM supporta le tecnologie sopra citate, le interfacce dichiarate nel descrittore come SR-IOV/passthrough vengono assegnate a quelle “fisiche” del nodo (identificate con un PCI address, tutto in accordo con quanto visto per le tecnologie nel capitolo 9). Una volta assegnate tali interfacce, OSM contatta il VIM che gli fornisce informazioni sul nodo dove le VNF sono state istanziate e le relative interfacce. A questo punto OSM può mappare le interfacce con le porte di uno switch OpenFlow, mapping che va configurato in anticipo con OSM. A questo punto abbiamo uno switch che controlla le porte corrispondenti alle interfacce fisiche delle singole VNF. L’ultimo tassello è integrare e configurare un SDN controller (a scelta tra quelli compatibili) per controllare il flusso dei pacchetti attraverso la modifica delle flow table. Di questa ultima fase se ne occupa l’RO attraverso delle librerie ad hoc sviluppate per gli SDN controller.

I requisiti sono:

- uno switch che supporta OpenFlow e collegato alle interfacce fisiche dei nodi che ospitano le VM;
- un SDN controller esterno, che va poi integrato in OSM e che controlla questo dataplane;
- configurazione ad hoc per il VIM utilizzato, al fine di supportare SR-IOV o passthrough.

Per quanto riguarda la configurazione del VIM, ad esempio OpenStack, rimandiamo alla documentazione [https://osm.etsi.org/wikipub/index.php/Openstack_configuration_\(Release_THREE\)#Configure_Openstack_for_OSM_.28EPA.29](https://osm.etsi.org/wikipub/index.php/Openstack_configuration_(Release_THREE)#Configure_Openstack_for_OSM_.28EPA.29). Per quanto riguarda la configurazione dello switch programmabile e dell’SDN controller con il VIM rimandiamo alle guide per i casi specifici, tipicamente sono previsti dei plug-in come nel caso di OpenDaylight e ML2 di neutron OpenStack. Guardiamo soltanto la parte che concerne il Resource Orchestrator, in particolare vediamo come integrare l’SDN controller. Innanzitutto entriamo all’interno del container RO:

```
$ export OPENMANO_TENANT=osm # Indicate the RO tenant to use
$ lxc exec RO -- bash
```

Successivamente creiamo l’entità SDN controller nell’RO, l’esempio prevede l’utilizzo di ONOS:

```
$ openmano sdn-controller-create mySDN --ip=192.168.15.2 --port=8080
--dpid=56:55:12:12:12:12:12:12 --user sdnuser --passwd sdnpasswd --type
onos
$ openmano sdn-controller-list # controllare la lista degli SDN controller
```

A questo punto associare l’SDN controller al VIM:

```
$ openmano datacenter-edit mydc --sdn-controller mySDN
$ openmano datacenter-list mydc -vvv #controllare il successo dell’operazione
```

L’ultima operazione è effettuare il port mapping tra lo switch e le porte SR-IOV/passthrough. Un esempio è dato dal file riportato di seguito:

```
$ tail -n 24 R0/sdn/sdn_port_mapping.yaml
```

Modificando il file è possibile ottenere il mapping desiderato come riportano nella guida https://osm.etsi.org/wikipub/index.php/EPA_and_SDN_assist#Adding_a_port_mapping.

B.7.2 Ansible charm

L'integrazione di Ansible per la configurazione delle VNF non viene effettuato nel modo canonico. Ci si aspetterebbe, infatti, un'integrale sostituzione del controller di Juju nel VCA per fare posto alle primitive di esecuzione degli script Ansible; in modo da poter gestire in modo centralizzato la configurazione delle stesse.

A dispetto di tale metodologia, per l'integrazione dei playbook Ansible in OSM, viene invece effettuato un wrapping delle primitive Ansible in un charm di Juju. In pratica viene creato un layer di un particolare proxy charm che permette, una volta definiti dei playbook specifici, di associare delle azioni all'esecuzione di tali playbook sulle VNF. Questo metodo consente di lavorare con Ansible attraverso i proxy charm di Juju. Per ulteriori dettagli rimandiamo al codice e alla documentazione di tale soluzione, eventualmente da integrare per le vNSF da configurare con Ansible <https://github.com/5GinFIRE/mano/tree/master/charms/ansible-charm>.

B.7.3 RO extention

Per quanto riguarda l'estensione del Resource Orchestrator, in particolare per operazioni come la connessione di nuovi VIM non supportati, sono riportati di seguito dei riferimenti a guide su come procedere in tal senso.

Nello specifico vi è una parte di azzeramento sulla modalità sviluppatore di OSM in https://osm.etsi.org/wikipub/index.php/Developer_HowTo_for_RO_Module e in particolare per la connessione di nuovi VIM vi è una guida specifica all'indirizzo https://osm.etsi.org/wikipub/index.php/Developer_HowTo_for_RO_Module#Creating_a_new_VIM_plugin.

Appendice C

Manuale sviluppatore: User Policy Repository (UPR)

Lo User Policy Repository è il cuore della gestione del Policy Services, contenendo tutte le informazioni riguardo agli utenti e alle policy definite dagli stessi. Per questo motivo lo utilizziamo come esempio nel processo di reingegnerizzazione dei moduli del progetto di SECURED.

C.1 Installazione

C.1.1 Introduzione e creazione container LXD

Innanzitutto, come già visto nella parte teorica, abbiamo utilizzato la tecnologia di virtualizzazione LXD per esporre i servizi reingegnerizzati del progetto SECURED. Questo ci ha permesso di risparmiare considerevoli risorse di calcolo, rispetto alla precedente soluzione bare-metal, e risulta estremamente più flessibile e scalabile. Vediamo adesso come è possibile ricreare il modulo UPR con la tecnologia LXD.

Partiamo dal presupposto che OSM è già stato installato sulla macchina, questo ci dà il vantaggio di avere a disposizione un linux bridge su cui lavorare. In caso contrario si può provvedere diversamente alla creazione del bridge attraverso i comandi messi a disposizione da linux, ma non è di nostro interesse, in quanto vogliamo che le due soluzioni (OSM e Policy Services) siano sulla stessa sottorete. Ritornando quindi alla situazione di partenza con un linux bridge preinstallato sulla macchina (tipicamente con OSM `lxdbr0`), possiamo iniziare il processo creando un nuovo container LXD.

Per fare questo andiamo sulla macchina (dov'è installato OSM) ed eseguiamo il comando:

```
$ lxc launch ubuntu:16.04 upr -c security.privileged=true
```

Una volta che la creazione del container è stata effettuata con successo, possiamo controllare che tutto sia andato buon fine attraverso il comando:

```
$ lxc list
```

che ci dovrebbe restituire, tra tutti quelli già presenti, il container LXD appena creato, con assegnato un indirizzo IP ad una specifica interfaccia (tipicamente `eth0`). Questo è il punto di partenza ora, per iniziare il processo di installazione vero e proprio, accediamo al container con il comando:

```
$ lxc exec upr -- bash
```

C.1.2 Installazione UPR

La prima operazione, una volta entrati nel container upr è quella di scaricare ed installare le dipendenze. Iniziamo con:

```
$ sudo apt-get install mysql-server libmysqlclient-dev python-virtualenv libffi-dev
```

Nel caso in cui di dovessero essere problemi, effettuare un aggiornamento dei repository con:

```
$ sudo apt-get update
```

Successivamente riprovare con il precedente comando. Durante il processo di installazione va assegnata una password al database. Adesso clonare il repository dell'UPR di SECURED ed effettuare una serie di operazioni preliminari:

```
$ git clone https://github.com/SECURED-FP7/secured-upr.git
$ virtualenv .env
$ source .env/bin/activate
$ pip install django django-rest-framework MySQL-python django-rest-swagger
  wrapt bcrypt python-keystoneclient
$ cd secured-upr
```

Nel caso in cui si riscontrassero problemi al passo “pip install” procedere all'installazione dei seguenti pacchetti e ripartire dal medesimo passo:

```
$ sudo apt-get install gcc
$ sudo apt-get install python-dev
```

Adesso va inizializzato il DB, ricordando la password dei passi precedenti e avendo cura di dichiarare una nuova coppia utente e password all'interno dei campi 'DATABASE_USER' e 'DATABASE_PASS':

```
mysql -u root -p
mysql> CREATE DATABASE UPR;
mysql> GRANT ALL PRIVILEGES ON UPR.* TO
'DATABASE_USER'@'localhost' \
IDENTIFIED BY 'DATABASE_PASS';
mysql> GRANT ALL PRIVILEGES ON UPR.* TO 'DATABASE_USER'@'%' \
IDENTIFIED BY 'DATABASE_PASS';
mysql> exit
```

Una volta effettuato questo vanno effettuate delle modifiche al file upr/settings.py:

```
ALLOWED_HOSTS = ['IP_ADDRESS_LXD_HOST']
...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'UPR',
        'USER': os.getenv('DATABASE_USER', 'CHANGE_ONLY_HERE_USER'),
        'PASSWORD': os.getenv('DATABASE_PASS', 'CHANGE_ONLY_HERE_PASS'),
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

Le modifiche corrispondono all’inserimento degli host autorizzati l’ip del container LXD e utente e password scelti in precedenza nel campo “DATABASES”, avendo cura di sostituire solo dove richiesto nel codice utente e password. Prima dell’avvio del server Django dobbiamo effettuare altri due passaggi. Il primo è il downgrade della libreria django-rest-swagger:

```
$ pip uninstall django-rest-swagger
$ pip install django-rest-swagger==0.3.10
```

Il secondo consiste nella migrazione del database upr, cosa che consente di inizializzare il data model utilizzato:

```
$ python manage.py makemigrations upr
$ python manage.py migrate
```

Questo ci permetterebbe di avviare con successo il server, ma a causa di alcuni problemi di retrocompatibilità, siamo costretti ad effettuare altre operazioni per il successivo utilizzo del server. Creare le seguenti cartelle:

```
$ mkdir upr/scenarioOpenmano/NED
$ mkdir upr/scenarioOpenmano/TVD
```

Adesso modificare il pathScenario all’interno del file upr.conf come riportato:

```
[mano]
tenant = obtained uuid
ipMano = Openmano ip
portMano = Openmano port
pathScenario= /root/upr/scenarioOpenmano
```

Ancora procedere alla modifica del file upr/serializers.py:

```
class UserRAGAssociationSerializer (serializers.ModelSerializer):
    class Meta:
        model = UserRAGAssociation
        fields = '__all__'
```

Inserire il campo “ fields = '__all__' ” in ognuna delle funzioni per l’associazione alle chiamate REST, queste previene un bug durante l’utilizzo del server. A questo punto si può procedere con l’avvio del server, mostriamo la procedura da effettuare ad ogni riavvio:

```
$ source .env/bin/activate # non necessario se già nell’ambiente virtuale
$ cd secured-upr
$ python manage.py runserver IP_LXD_CONTAINER_UPR:8081
```

Un’ultima modifica è da farsi per ottenere la documentazione automatica dell’UPR sotto /docs. Questa va effettuata nel file upr/urls.py:

```
from rest_framework.documentation import include_docs_urls
...
url(r'^docs/', include_docs_urls(title='User Policy Repository')),
```

In particolare va importata la prima riga relativa alla documentazione e modificata l’ultima riga inerente all’url della stessa, il tutto per come è mostrato nel codice presentato.

C.1.3 Documentazione ed utilizzo

Tutta la documentazione relativa all’UPR si può trovare, una volta seguita pedissequamente la guida nella precedente sezione, al link:

http://INDIRIZZO_LXD_UPR:8081/docs

Tutte le REST API risultano documentate e attraverso la libreria `rest_framework_swagger` è possibile effettuare delle chiamate interattive direttamente da interfaccia grafica. Per agevolare l’utilizzo diretto dell’UPR sono forniti, allegati alla documentazione dei modelli per l’applicazione Restlet Client del browser Google Chrome. Questi modelli possono essere importati e contengono tutta una serie di chiamate REST precostituite per popolare o interrogare il repository.

Appendice D

Manuale sviluppatore: OpenStack

In quest'appendice vedremo come installare e configurare OpenStack.

D.1 Installazione con DevStack

Il nostro scopo è quello di testare OSM e integrarlo con un VIM, in particolare OpenStack. In generale non abbiamo necessità di avere ognuno dei principali servizi (nova, neutron, cinder) distribuiti. Questo ci ha portato all'utilizzo di una configurazione di OpenStack su singolo nodo (controller e compute), così da rendere contenuto l'ambiente di testing. Nel nostro caso specifico facciamo riferimento all'installazione messa a disposizione per gli sviluppatori di OpenStack, ovvero DevStack.

È possibile avere, con questa tipologia di installazione, un ambiente OpenStack pronto in pochi passaggi. Illustreremo come a breve e daremo delle note aggiuntive per dare spazio ad alcune personalizzazioni desiderate.

D.1.1 Installazione

Bisogna partire, come nel caso di OSM, con una distribuzione di linux Ubuntu preferibilmente la 16.04 LTS (che risulta la più stabile). Una volta soddisfatto questo prerequisito, si può procedere all'installazione vera e propria. Innanzitutto creiamo un nuovo utente per ospitare devstack e cloniamo il repository di devstack come riportato di seguito:

```
$ sudo useradd -s /bin/bash -d /opt/stack -m stack #aggiungo utente al
    sistema operativo
$ echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack # sudo
    senza password per stack
$ sudo su - stack
$ git clone https://git.openstack.org/openstack-dev/devstack # clono il
    repository
$ cd devstack
```

Una volta completati tutti questi passaggi possiamo pensare a come configurare il nostro ambiente DevStack, prima di procedere all'avvio dello script di installazione. All'interno della cartella `devstack`, si crei un file `local.conf`. il quale contiene le nostre preferenze durante l'installazione. Viene allegato alla documentazione un script di esempio da copiare in `devstack`, in linea con la nostra soluzione; questo per semplificare e velocizzare la prima installazione. Con il file `local.conf` è possibile in ogni caso aggiungere/modificare:

- credenziali dei servizi a disposizione (es. rabbit, database, etc.);
- aggiungere servizi (es. Heat) o modificarne la versione delle API utilizzate;
- precaricare delle immagini software in OpenStack;
- configurare un SDN controllo (ad es. OpenDaylight) e personalizzarne la configurazione;

tutto questo e molto altro riportato in dettaglio nella pagine del progetto <https://docs.openstack.org/devstack/latest/>. A noi basta, per i nostri scopi, la versione di `local.conf` allegata alla documentazione. Una volta copiato tale file in devstack, procediamo con l'installazione attraverso il comando:

```
$ ./stack.sh
```

D.1.2 Utilizzo e configurazione

Una volta completata l'esecuzione dello script di installazione (impiega in media 20 minuti) è possibile accedere ad horizon o alla CLI per poter interagire con OpenStack. Nel caso di horizon collegarsi a:

http://INDIRIZZO_IP_HOST

Automaticamente si viene ridiretti sulla dashboard, dove l'accesso è garantito attraverso le credenziali user/password inserite nel file `local.conf`, nel nostro caso admin/secret. Per quanto riguarda la CLI bisogna dichiarare una serie di variabili d'ambiente per avere accesso alla stessa. Per farlo si può accedere alla cartella devstack ed eseguire il comando:

```
$ source openrc admin
```

il quale garantirà l'accesso a OpenStack come utente amministratore. Una volta effettuati questi passi preliminari, si possono effettuare numerose operazioni sia da interfaccia grafica che da CLI. Un esempio è la creazione di sottoreti attraverso l'apposito menù o l'avvio di istanze a partire dalle immagini caricate. L'interfaccia grafica è molto intuitiva ed è superfluo concentrarci sulla stessa ora (rimandiamo alla documentazione <https://docs.openstack.org/horizon/queens/admin/>). Una cosa di particolare interesse, invece, è l'utilizzo della CLI, infatti, tramite essa è possibile utilizzare degli script per configurare o intraprendere operazioni in automatico in OpenStack. In particolare ci interessa la possibilità di creare sottoreti, modificare i parametri delle porte collegate a tali reti, modificare i security group (sono delle regole per il controllo degli accessi), creare router ed elementi di rete, effettuare l'upload di immagini ed avviare istanze. È possibile fare tutto questo attraverso l'utilizzo della CLI. In particolare illustreremo alcuni comandi di interesse:

- **openstack network create:** è possibile creare attraverso questo comando una rete, specificando la tipologia di rete e alcuni parametri aggiuntivi (di seguito spiegheremo dove trovarli).
- **openstack subnet create:** crea una sottorete associata ad una rete preesistente.
- **openstack network set:** modifica alcuni parametri della rete, ad esempio il port-security.
- **openstack security group rule create:** crea una regola per consentire o meno l'accesso ad una tipologia di traffico (ad. es ICMP) e/o ad uno specifico intervallo di indirizzi, consentendo un largo numero di personalizzazioni.
- **openstack router create router:** crea un router virtuale che può essere collegato ad interfacce (porte) di diverse sottoreti.
- **openstack router set router:** modifica i parametri del router appena creato.
- **openstack image create:** carica un'immagine nel repository di glance.

Tutti questi comandi sono utili per poter navigare la soluzione ed effettuare modifiche talvolta essenziali per il corretto funzionamento. Abbiamo sviluppato uno script, che attraverso i comandi della CLI, configurasse tutto ciò che era necessario per la nostra soluzione, evitando ulteriori processi manuali di configurazione. Da questo è possibile ricavare dettagli sui comandi appena citati e capire quali sono le logiche di funzionamento della CLI. Lo script a cui facciamo riferimento è `ConfigOpenstack.sh` e appena effettuata l'installazione è possibile eseguirlo per configurare il VIM per i nostri scopi:

```
$ source ConfigOpenstack.sh
```

Un'ultima osservazione riguarda la connettività esterna, che OpenStack consente attraverso l'uso del bridge `br-ex`, creato sull'`host`. A questo punto va abilitato e vanno configurate le interfacce per poter accedere alle macchine virtuali che verranno istanziate, questo è fondamentale per le configurazioni attraverso Juju. Abbiamo creato un script per effettuare anche questa configurazione, di seguito ripetiamo tutti i passi per avere a disposizione una macchina (fisica o virtuale) con il VIM OpenStack pronto per l'utilizzo con OSM:

```
$ source OpenstackConf.sh
```

Installiamo DevStack con la procedura riportata nella precedente sezione, sostituendo `local.conf` con l'omonimo file che abbiamo fornito in allegato. Una volta installato guadagniamo i privilegi della CLI (sezione precedente "openrc admin") ed eseguiamo lo script da noi creato "ConfigOpenstack.sh". Quest'ultimo script segnerà l'indirizzo IP dell'interfaccia del router collegata alla rete esterna. Questo è utile perché l'ultimo passaggio da effettuare è eseguire un altro script sviluppato da noi "OpenstackConf.sh", che inizializza il `br-ex` e configura le route di sistema. In questo script va sostituito l'indirizzo ip trovato al passo precedente prima di eseguirlo.

Quando sarà eseguito quest'ultimo sarà possibile utilizzare direttamente senza ulteriori configurazioni OpenStack e/o integrarlo con OSM (fino alla release THREE compresa).

D.2 Integrazione con OpenSource MANO

Come abbiamo visto nelle precedenti sezioni per integrare un VIM in OSM è sufficiente il comando da CLI:

```
$ osm vim-create --name openstack-site --user admin --password secret --auth_url http://IP_devstack_HOST/identity/v3 --tenant admin --account_type openstack
```

Tale comando permette la completa integrazione con DevStack. Ora l'ultimo passaggio sta nel configurare anche l'`host` di OSM con quello di OpenStack per raggiungere la rete di management delle VM. Questo lo facciamo con un altro script fornito in allegato "OsmConf.sh", che si occuperà della configurazione automatica dell'`host` OSM. Eventuali variazioni da effettuare saranno specificate direttamente all'interno dello script.