POLITECNICO DI TORINO
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Configurazione di funzioni di sicurezza virtualizzate basata su un approccio a intenti

Relatori
Prof. Antonio Lioy
Ing. Daniele Canavese
Ing. Ignazio Pedone

Candidato
Pierangelo Martellone

Anno accademico 2021-2022

Alla mia famiglia, ai miei amici, a M.
A R., un sentito e sincero grazie

Indice

1 Introduzione 7

2 Lavori Collegati 10

2.1 Intenti in stile discorsivo/strutturato e controllore 10

2.2 Estrazione di intenti 16

3 Approccio intent-based 19

3.1 Reti autonome 19

3.1.1 Definizione e principi 20

3.1.2 Obiettivi 21

3.2 Introduzione agli intenti 22

3.2.1 Definizioni 23

3.2.2 Principi e novità rispetto ai concetti noti 24

3.3 Intent-based networking 26

3.3.1 Evoluzione delle IBN e stato dell'arte 27

3.3.2 Funzionalità 28

3.4 Linguaggi di definizione degli intenti 30

3.4.1 Linguaggi strutturati e in stile discorsivo 30

3.4.2 NEtwork MOdeling 33

3.4.3 Analisi di OpenDaylight NEMO 37

4 Network Functions Virtualization 39

4.1 Introduzione ad NFV 39

4.1.1 Definizione e principi 40

4.1.2 Relazione con i concetti di SDN e Cloud Computing 41

4.1.3 Vantaggi 42

4.2 Standard ETSI NFV 43

4.2.1 Piattaforma NFV 44

4.2.2 Virtual Network Function 45

4.2.3 Network Functions Virtualization Infrastructure 46
4.3 Network Functions Virtualization MANO 48
4.3.1 Architettura 48
4.3.2 Open Source MANO 50
4.4 Virtual Network Security Functions 51

5 Virtual Private Network 52
5.1 Visione d'insieme 52
5.2 VPN tramite IPsec 53
5.3 Scenari 58
5.4 Soluzioni VPN di livello N 59
5.5 Strumenti 60

6 Design della soluzione 63
6.1 Progettazione di alto livello 63
6.1.1 Architettura 63
6.2 Piattaforma di gestione 65
6.2.1 NEMO 65
6.2.2 Confronto con l'architettura di OpenDaylight NEMO 68
6.2.3 Orchestratore 70
6.3 vNSF 72
6.3.1 Progettazione e configurazione della vNSF 72
6.4 Workflow 73

7 Implementazione 77
7.1 Realizzazione del framework NEMO 78
7.1.1 Estensione della sintassi 78
7.1.2 Modulo di traduzione 82
7.2 Orchestratore 84
7.3 Schema 85

8 Risultati sperimentali 87
8.1 Banco di prova 87
8.2 Analisi 89
8.2.1 Modulo di traduzione 89
8.2.2 Scenario completo 91

9 Conclusioni e sviluppi futuri 95

A Manuale utente 97

A.1 Requisiti 97

A.1.1 Codice Sorgente 97

A.1.2 Servizio Docker 99

A.2 Installazione 99

A.2.1 Codice Sorgente 99

A.2.2 Servizio Docker 100

A.2.3 Terminatori VPN 101

A.2.4 Creazione dell'ambiente di test 102

A.3 Utilizzo 105

A.3.1 Utilizzo di NEMO 105

A.3.2 Utilizzo del framework sviluppato 105

A.3.3 Modifiche allo scenario 106

B Manuale sviluppatore 108

B.1 NEMO 108

B.1.1 Installazione packages 108

B.1.2 Configurazioni IPsec 109

B.1.3 MD-SAL 112

B.1.4 Supporto a nuove vNSF 113

B.2 Orchestratore 114

Bibliografia 117

Capitolo 1

Introduzione

Dai primi mesi del 2020 l'intera popolazione mondiale è alle prese con la pandemia di COVID-19, che ha portato ad una percentuale di contagiati pari a circa il 3,58% con oltre 5,41 Mln di decessi[1]. Si tratta di una situazione estremamente complessa che ha cambiato radicalmente le abitudini sociali e lavorative in tutti i paesi del mondo, dovuto principalmente alle misure restrittive adottate, volte a contrastare la diffusione dei contagi. Dopo un'immobilità pressoché totale nel settore lavorativo, per indurre una ripresa e limitare i danni economici causati dalla pandemia, molte aziende hanno incentivato il lavoro da casa, ove possibile. Secondo BusinessWire[2], circa l'88% delle aziende nel campo IT hanno trasferito i propri dipendenti dagli uffici verso il lavoro da casa, adottando un modello destinato a durare per anni, anche nel periodo successivo alla pandemia. In aggiunta, la continua ricerca di fonti di informazione attendibili, al di fuori dei limiti imposti da alcuni paesi (es. Cina), o il tentativo di fruizione di contenuti multimediali non ammessi, utilizzati come passatempo nel periodo di lockdown, hanno portato ad un incremento significativo dell'utilizzo delle *Virtual Private Network* (VPN). Oltre a permettere, infatti, la connessione remota ad una rete aziendale, realizzando un canale sicuro attraverso una rete non fidata, le VPN permettono di accedere ad un server estero e mimetizzare la connessione tra quelle provenienti da quel paese, aggirando le restrizioni imposte da alcuni governi.

Sempre secondo l'articolo di BusinessWire, nel 2020 si è registrato un incremento nell'utilizzo pari a circa il 27,1% con una previsione di crescita, per il 2027, fino a 107,5 miliardi di dollari per il mercato delle VPN.

Uno studio realizzato dalla piattaforma Statista[3], sui dati raccolti tra l'8 e il 22 Marzo 2020, dimostra come l'incremento nell'utilizzo delle connessioni VPN sia particolarmente vertiginoso nei paesi fortemente colpiti dal COVID-19: si registra, in Italia, un aumento del 160% per settimana e del 124% negli Stati Uniti, a fronte di un numero di casi registrati pari a 51.768 e 33.005 rispettivamente, per settimana.

Ciò viene riscontrato anche dalle statistiche riportate da Google Trends[4], che evidenziano chiaramente un maggiore interesse, da parte delle persone, verso la tecnologia

[1] https://news.google.com/covid19/map?hl=it&gl=IT&ceid=IT%3Ait

[2] https://www.businesswire.com/news/home/20201127005318/en/Global-Virtual-Private-Network.com

[3] https://www.statista.com/statistics/1106137/vpn-usage-coronavirus/

[4] https://trends.google.it/trends/explore?date=2018-07-01%202022-01-29&geo=IT&q=VPN

VPN. Come si evince dal grafico riportato in Figura 1.1, le ricerche sul motore Google subiscono un picco evidente in corrispondenza dei primi mesi successivi alla diffusione del virus e in corrispondenza del secondo lockdown, nel 2020. In particolare, Stati Uniti e Cina risultano essere i paesi da cui provengono la maggior parte di queste ricerche, per i motivi descritti precedentemente.

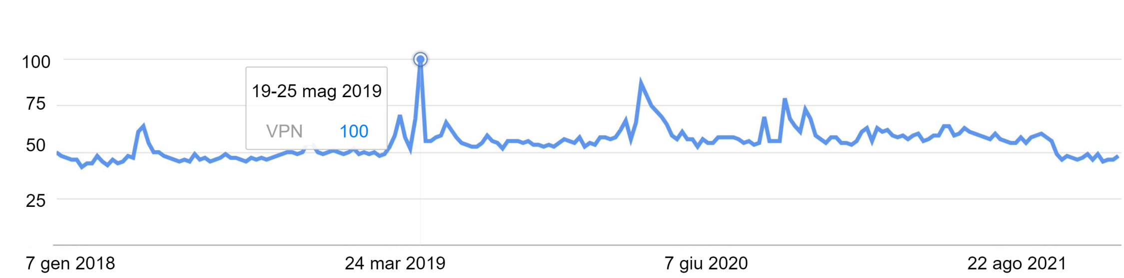


Figura 1.1. Ricerche su Google della parola VPN (fonte Google Trends)

Utilizzando una chiave di lettura diversa, questa ricerca massiva del termine e la raccolta di informazioni a riguardo, sono indice di una sostanziale inconsapevolezza nei confronti di questo strumento da parte dell'utente medio. In aggiunta, alla mancanza di competenze tecniche da parte dei piccoli utenti, si aggiunge, a livello aziendale, la difficoltà di gestione e l'importanza dell'impatto economico derivanti da un servizio di questo tipo.

Queste difficoltà hanno portato, gradualmente, alla nascita del paradigma SECaas (Security-as-a-Service) che permette di usufruire di un servizio di sicurezza completamente configurato e funzionante con tempi di realizzazione molto bassi. Appartenente alla classe SaaS (Software-as-a-Service), implementa le sue funzioni su server remoti sfruttando, molto spesso, la tecnologia NFV (*Network Function Virtualisation*). In questo modo è possibile sfruttare a pieno le risorse offerte dai server, realizzando svariate funzioni di rete chiamate virtual NSF (vNSF), offerte come servizio agli utenti. Questo si traduce, in termini pratici, nella realizzazione di servizi quali Firewall, Deep packet inspection, Virtual Private Network utilizzabili in modo trasparente e forniti da un *Provider*.

Sulla base di queste premesse, lo scopo di questa tesi è quello di realizzare una piattaforma di supporto agli utenti, al fine di semplificare la configurazione dei servizi di sicurezza realizzati come vNSF, sfruttando uno degli approcci emergenti più promettenti, basato sui principi dell'*Intent-based Networking*. Lo scopo di questo approccio è quello di eliminare la componente tecnica richiesta nell configurazione e nella personalizzazione di servizi di sicurezza come quelli precedentemente citati, a fronte di un utilizzo di un linguaggio di derivazione naturale definito, appunto, a intenti. In questo modo, è possibile esprimere una configurazione con un linguaggio "parlato" permettendo a ciascun utente di personalizzare un servizio, in quanto non sono richieste particolari competenze tecniche.

A tal proposito, il lavoro si inserisce nel contesto del progetto europeo *FISHY*[5] che si pone come obiettivo quello di realizzare una serie di sistemi completi di protezione semi-automatica da utilizzare come supporto alle catene di approvvigionamento in ambito

[5]https://fishy-project.eu/

Information and Communication Technologies[6] (ICT), garantendone sicurezza e affidabilità. Il tassello a cui si dedica questa tesi è rappresentato dalla realizzazione di una piattaforma portatile di gestione degli intenti in grado di configurare, nel contesto dei servizi vNSF, una connessione VPN. Il lavoro culmina con la realizzazione della piattaforma NEMO[7], di derivazione OpenDaylight[8], in grado di accettare in ingresso intenti e tradurli, successivamente, in configurazioni in modo del tutto trasparente verso l'utente.

Il progetto di OpenDaylight si inserisce nel contesto del *Software-defined Networking* (SDN)[9], di cui rappresenta una delle implementazioni di rilievo per quanto concerne il controllore. Sempre in un contesto di ricerca di agilità nella configurazione di una rete, NEMO propone un'interfaccia a intenti che mira a sottrarre l'utente dalla configurazione manuale di ciascun apparato a favore di una "descrizione" dello scenario desiderato.

Il *PoC* (*Proof of Concept*) fornito, include una piattaforma di orchestrazione in grado di iniettare le configurazioni generate da NEMO direttamente nelle vNSF, per configurare il software di realizzazione delle VPN. Inoltre, lo scenario testato permette la realizzazione di una connessione VPN di tipo *site-to-site*[10] tra due terminatori, configurati attraverso l'orchestratore e grazie al lavoro svolto dalla piattaforma NEMO. In conclusione, contestualizzando il lavoro in uno scenario reale, si può dire che queste piattaforme potrebbero trovare impiego, ad esempio, nella rete di un *Internet Service Provider* in cui sono implementate, nei nodi ai margini della rete verso l'utente finale, tali funzioni di rete (ad esempio un terminatore VPN) e i servizi offerti da NEMO, che permetterebbero all'utente di configurarle.

Questa trattazione si divide nei seguenti capitoli:

- Capitolo 2: presentazione ed analisi dei lavori collegati ai concetti espressi in questa tesi;
- Capitolo 3: presentazione dell'approccio *Intent-Based*;
- Capitolo 4: analisi degli strumenti e della tecnologia NFV secondo lo standard;
- Capitolo 5: presentazione dei concetti fondamentali della tecnologia VPN;
- Capitolo 6: progettazione della soluzione in vista degli obiettivi e delle criticità da risolvere;
- Capitolo 7: realizzazione della soluzione;
- Capitolo 8: analisi dei dati raccolti dai test effettuati;
- Capitolo 9: conclusioni e possibili sviluppi futuri;
- Appendice A: manuale utente;
- Appendice B: manuale sviluppatore.

[6] *Insieme delle metodologie e tecniche utilizzate nella trasmissione, ricezione ed elaborazione dei dati*

[7] https://test-odl-docs.readthedocs.io/en/latest/user-guide/nemo-user-guide.html

[8] https://www.opendaylight.org/

[9] Approccio che mira ad aumentare la dinamicità nella configurazione di una rete, separando la logica di controllo degli apparati dal piano di inoltro dei pacchetti.

[10] Con il termine *site-to-site* si indica una modalità di realizzazione di una connessione VPN nella quale vengono collegate due reti differenti, come ad esempio quelle di una filiale e della sede centrale. Si differenzia dalla modalità *access* in cui la connessione si instaura tra una singola macchina ed una rete remota.

Capitolo 2

Lavori Collegati

Questo capitolo ha lo scopo di analizzare alcuni lavori presenti in letteratura riguardanti la configurazione di reti e, più in generale, apparati e funzioni di rete secondo un approccio intent-based. Procedendo con la descrizione dei vari articoli, viene eseguito un confronto, sottolineando somiglianze e differenze tra i vari approcci.

Nonostante le strategie e le scelte implementative siano molto variegate, è possibile raggruppare alcuni lavori: l'approccio comune prevede l'uso di un'interfaccia, attraverso la quale definire gli intenti mediante un linguaggio ben definito, e di un controllore centralizzato che si occupa di eseguire, a basso livello, le azioni necessarie per far raggiungere alla rete lo stato desiderato. Questo tipo di architettura è utilizzata nei lavori approfonditi nella sezione 2.1. I lavori che si discostano da questo approccio, analizzati nella sezione 2.2, risultano più disomogenei e difficilmente catalogabili. Sono accomunati, tuttavia, dall'utilizzo di strategie per gestire gli intenti che non prevedono un linguaggio vero e proprio, ma tecniche per estrarli da entità già note.

2.1 Intenti in stile discorsivo/strutturato e controllore

L'articolo presentato da Huawei[1] [1] definisce per la prima volta un'architettura basata su *SDN* [2] integrata con una NBI[3] dichiarativa e basata sugli intenti. La riflessione da cui si parte mette in risalto un concetto largamente utilizzato nelle SDN, secondo cui la NBI è orientata verso le applicazioni e non verso l'utente vero e proprio. In questo tipo di rete, le informazioni riguardo le capacità degli apparati sottostanti sono messe a disposizione delle applicazioni con un approccio bottom-up, aumentando, di fatto, la programmabilità. Grazie a questo tipo di interfaccia, le applicazioni possono ad esempio scoprire la topologia, definire collegamenti e configurare interfacce in modo estremamente rapido. Huawei definisce un approccio diametralmente opposto, in cui la rete e i servizi che essa può offrire vengono astratti in modo che l'utente possa accedervi automaticamente, interagendo con l'applicazione. La complessità, dunque, che risiede tutta nell'interfaccia

[1] https://www.huawei.com/

[2] Software defined network, è un paradigma di rete che separa il piano di controllo, cioè la logica di un apparato di rete, dal piano di inoltro dei pacchetti. L'innovazione sta nel fatto di avere un controllore centralizzato (a livello logico e non fisico) e programmabile.

[3] Northbound interface, si tratta di un'interfaccia che permette ad un componente sottostante di dialogare con i livelli superiori.

del servizio di rete, viene risolta da un middleware di tipo dichiarativo che si occupa di comunicare al controllore quale stato abbia richiesto l'applicazione, convertendo questo tipo di richiesta in dettagli di configurazione implementati, poi, nei dispositivi, che, nel caso delle SDN in cui si utilizzano rappresentazioni software degli apparati di rete, prendono il nome di data path (Figura 2.1). A questo punto il controllore può adottare la strategia più opportuna per implementare a basso livello la configurazione richiesta.

Questo approccio ricalca parzialmente quello utilizzato in questo lavoro, integrato con un orchestratore NFV[4].

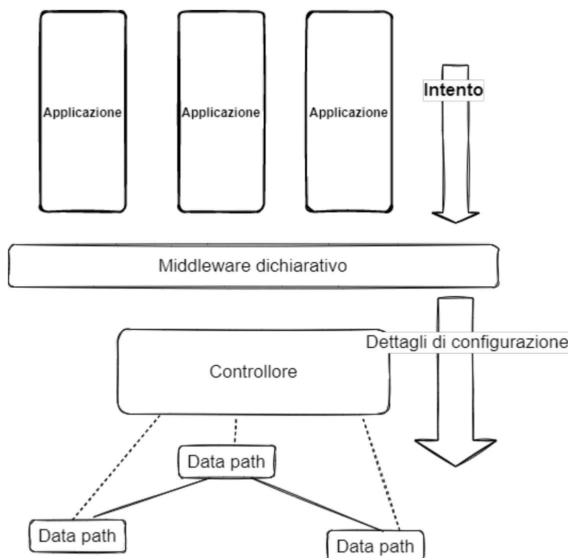


Figura 2.1. Middleware dichiarativo su un controllore SDN

Un articolo del 2018 [2], in analogia con il precedente, descrive un'architettura a livelli in cui l'orchestratore è esteso con un modulo in grado di compilare gli intenti. Lo scopo è quello di realizzare una piattaforma in grado di supportare la creazione automatica di servizi sicuri in una rete selezionando il livello OSI[5] migliore in cui eseguire la cifratura dei dati in base ai vincoli definiti dall'utente tramite intenti. Questo permette all'utente di richiedere un servizio dalla rete e specificare alcuni vincoli circa la cifratura dei dati. L'orchestratore ACINO[6] basato su ONOS[7] riceve gli intenti direttamente dalla NBI e li inoltra al modulo di compilazione, per selezionare la sequenza di azioni necessaria al soddisfacimento dell'intento. Il compilatore, infatti, è un modulo in grado di comprendere gli intenti definiti dall'utente e trasformarli in una forma di basso livello, adeguata per essere gestita dai protocolli sottostanti. Il caso d'uso riportato come esempio mostra la richiesta di un servizio cifrato attraverso un intento, in cui l'utente specifica dei vincoli sulla larghezza di banda desiderata. Contrariamente a questo lavoro, che utilizza intenti definiti in un linguaggio simile a quello umano, la soluzione proposta nell'articolo prevede l'utilizzo di intenti strutturati (definiti in JSON).

[4]Network functions virtualization, capacità di installare qualsiasi funzione di rete su un hardware standard, possibilmente sfruttando la virtualizzazione in modo da ottenere un uso efficiente delle risorse

[5]https://www.iso.org/standard/20269.html

[6]https://github.com/ACINO-H2020

[7]ONOS, controller per soluzioni che integrano SDN con funzioni di rete virtualizzate.

ONOS è anche il cuore dell'architettura descritta in [3]. In questa implementazione non è previsto l'utilizzo di orchestratori ad hoc ausiliari, ma è direttamente il controllore ad essere integrato con un motore linguistico. Questo, sviluppato come un applicativo direttamente sulla piattaforma ONOS, è in grado di analizzare gli intenti ricevuti e interagire con le API esportate dal core della piattaforma in modo da operare un controllo sulla rete sottostante. Di grande interesse è, sicuramente, il caso d'uso riportato che ricalca e descrive pienamente il tipo di comportamento ricercato in questo lavoro. La figura 2.3 rappresenta un potenziale script che il motore linguistico riceve, al fine di creare una DMZ[8] all'interno di un data center virtuale con la capacità di abilitare il BoD[9] tra il sito aziendale e quello interno; tale scenario è rappresentato graficamente in figura 2.2.

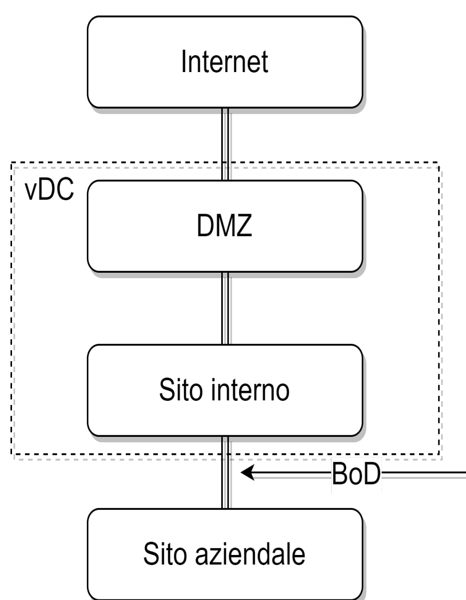


Figura 2.2. Scenario richiesto dagli intenti

```
IMPORT Node enterprise Type ext-group
    Property
    ac-info-network:"layer3",
    ac-info-protocol:"static",
    ip-prefix:"192.168.13.0/24";
IMPORT Node enterprise Type ext-group
    Property
    ac-info-network:"layer3",
    ac-info-protocol:"static",
    ip-prefix:"192.168.1.0/24";
CREATE Node dmz Type l2-group
    Property
    ip-prefix:"192.168.11.0/24",
    gateway-ip:"192.168.11.1";
CREATE Node interior Type l2-group
    Property
    ip-prefix:"192.168.12.0/24",
    gateway-ip:"192.168.12.1";
CREATE Connection c1 Type p2p
    Endnodes enterprise,interior;
CREATE Connection c2 Type p2p
    Endnodes interior,dmz;
CREATE Connection c3 Type p2p
    Endnodes dmz,internet;
CREATE Operation o1 Priority 2 Target
    c1 Condition time<9&&time>18
    Action qos-bandwith 2048 M;
CREATE Operation o2 Priority 1 Target
    c1 Action qos-bandwith 1024 M;
```

Figura 2.3. Esempio di un possibile script attraverso cui realizzare lo scenario descritto

In alcuni aspetti, il lavoro definito in [4] ricalca le scelte operate nel lavoro precedente. I casi d'uso utilizzano lo stesso tipo di linguaggio con una sintassi simile, ma, contrariamente, l'approccio utilizzato per configurare la rete è di tipo reattivo: lo scopo è quello di modificare la configurazione di rete analizzando continuamente lo stato attuale, reagendo al verificarsi di alcuni eventi, piuttosto che pianificare preventivamente le impostazioni della rete. La differenza sta nel definire delle operazioni da eseguire quando una particolare condizione si verifica nell'ambiente di rete. Un caso d'uso analizzato nell'articolo descrive due siti aziendali, uno centrale e uno distaccato, collegati da due linee a diversa

[8]De-Militarized Zone, sottorete che espone i servizi verso una rete ritenuta non sicura

[9]Bandwidth on Demand

larghezza di banda e diverso ritardo: quando la prima connessione sta per saturarsi, il traffico, etichettato con una certa priorità sarà rediretto verso l'altro collegamento. La figura 2.4 evidenzia le estensioni apportate al linguaggio in modo da definire particolari condizioni, come ad esempio la banda satura all'80%, e le corrispondenti contromisure. Questo tipo di approccio, stride leggermente con la definizione generale di intento che, dichiaratamente, mira a discostarsi dal concetto di politica.

```
CREATE flow f1 Match src-ip:10.0.0.0/24, dst-ip:10.0.1.0/24, port
    5004;
CREATE Operation o1 Priority 1 Target f1 Condition
    bandwitdth_target=c1, bandwidth_limit=80% Action go-through c2;
```

Figura 2.4. Estensione del linguaggio per definire condizioni e contromisure

Tuttavia, il lavoro che più si avvicina a quello proposto, è descritto in un articolo del 2019 [5]. L'architettura proposta (figura 2.5), unisce più moduli e tecnologie al fine di creare una *service function chain* (SFC)[10] su una piattaforma di orchestrazione NFV come Open Source MANO (OSM)[11]. Sfruttando direttamente le infrastrutture di CLoudLab[12], è stato realizzato un piccolo cluster composto da quattro nodi OpenStack[13], incluso uno che che agisce da controllore e un nodo addizionale che ospita la piattaforma OpenSource MANO e il modulo per gli intenti. Quest'ultimo prevede un'implementazione personalizzata che, esponendo un'interfaccia di tipo REST[14], è in grado di ricevere intenti in formato JSON, contenenti la definizione di ogni funzione di rete desiderata nella SFC. Il modulo di gestione, dopo aver filtrato gli intenti che contengono errori di sintassi, li invia al modulo di traduzione che li converte in dettagli di configurazione. I moduli di decisione e analisi, intervengono in un secondo momento, permettendo di verificare continuamente il corretto funzionamento della rete ed eventualmente correggerlo. L'implementazione descritta poc'anzi non è stata presa in considerazione in questo lavoro in quanto risulta limitata nella varietà di intenti che permette di esprimere e nella possibilità di essere estesa, al fine di prevedere altri scenari.

Partendo dalla configurazione desiderata dall'utente, espressa dal file JSON, il blocco di traduzione compone un file, chiamato *Network Service Descriptor* (NSD), contenente la descrizione di ogni servizio richiesto, nel formato accettato da OSM: analogamente alla strategia utilizzata in questo lavoro, l'articolo definisce un "catalogo" contenente svariati template, ognuno corrispondente ad una funzione di rete. A questo punto il modulo decisionale inietta il file NSD nella piattaforma OSM, la quale innesca la creazione delle istanze delle VNFs nel cluster.

Frutto di uno studio prettamente italiano, l'articolo in [6] fornisce un esempio di soluzione riguardo il tipo di richiesta di servizio che un'applicazione inoltra. Infatti ogni

[10]Architettura di rete che permette di collegare in cascata più funzioni di rete in un ordine ben preciso

[11]https://osm.etsi.org/

[12]https://www.cloudlab.us/

[13]Progetto open source che realizza una piattaforma per il cloud computing. È diventato il "sistema operativo" de-facto per il cloud.

[14]Architettura software per i sistemi distribuiti che tipicamente utilizza i metodi HTTP quali GET, PUT, POST, DELETE

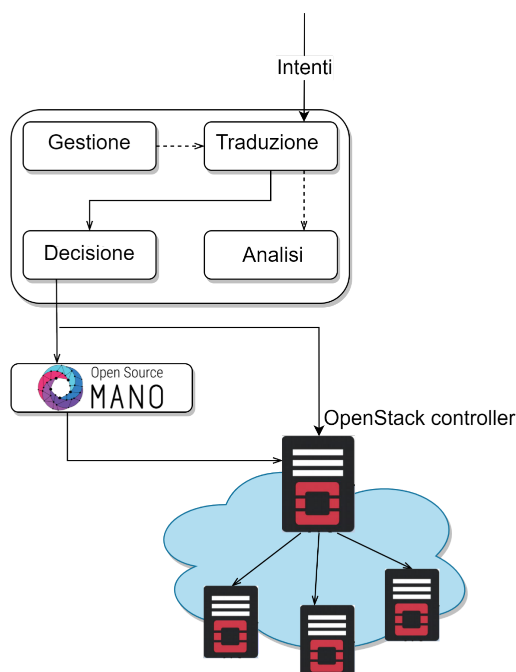


Figura 2.5. SFC su una piattaforma ETSI NFV

applicazione, definisce nei confronti della rete una serie di specifiche su vincoli quali banda, latenza e disponibilità del servizio; lo scopo del lavoro in esame è quello di fornire un'interfaccia a intenti in grado di definire queste richieste. Queste politiche, definite in linguaggio simile a quello umano, sono iniettate in un controllore SDN il quale permette di attivare un processo di negoziazione tra le richieste dell'applicazione e l'attuale disposizione della rete. Nel momento in cui l'intento viene espresso attraverso la NBI, il framework tenta di fornire una soluzione che possa rispettare tutti i requisiti e se questo risulta impossibile allo stato attuale della rete, il sistema attiva alternativamente due algoritmi: il primo, più semplice, fornisce strade alternative selezionando via via un requisito diverso che non si riesce a soddisfare, il secondo prevede una riallocazione di tutte le richieste di servizio in modo da soddisfare quella attuale.

Ancora un articolo italiano presentato in [7] discute di una piattaforma di gestione ed orchestrazione di una rete costituita da più domini tecnologici, con particolare attenzione al mondo dell'Internet of Things. Dopo aver definito una possibile topologia, in cui i paradigmi di SDN e NFV lavorano in sinergia, dotata di un orchestratore NFV e un gestore delle funzioni virtualizzate, gli autori passano in rassegna le caratteristiche ideali di riferimento di una North Bound Interface. Si tratta di una implementazione fatta ad hoc che, sfruttando la struttura dei file JSON, permette di descrivere la catena di servizio costituita da una sequenza di funzioni virtualizzate attraverso cui inoltrare il traffico. Questo approccio permette all'utente di definire, oltre alla sequenza delle funzioni che costituiscono la catena, i vincoli sulla QoS[15]. Inoltre è possibile descrivere ogni funzione virtualizzata: si può specificare se si tratta di una funzione di terminazione per il traffico oppure una che lo elabora e lo inoltra nuovamente; è possibile specificare se si tratta di una funzione simmetrica dal punto di vista delle porte, ovvero se riceve il traffico da una porta in input e lo inoltra verso un'altra; si può definire se un servizio è simmetrico

[15] Quality of service

dal punto di vista del traffico, ovvero se analizza il traffico in entrambe le direzioni. L'intento così composto viene poi tradotto e passato alla piattaforma di gestione ed orchestrazione denominata MANO, incaricata di realizzare e gestire il ciclo di vita delle funzioni virtualizzate in una piattaforma NFV.

Di grande rilievo per questo lavoro, l'articolo in [8] discute della possibilità di realizzare quella che è chiamata *VNF Placement solution*, ovvero l'implementazione di una VNF definita tramite un'interfaccia ed un linguaggio ad alto livello, in un'architettura multi-tenant[16]. L'intento viene definito tramite un template strutturato in XML che viene trasferito dall'interfaccia all'orchestratore e che contiene le caratteristiche e gli attributi necessari ad istanziarlo su una piattaforma NFV. In aggiunta, l'intento permette di definire dei livelli di sicurezza o di QoS discretizzati che saranno utilizzati come vincoli in fase di realizzazione del servizio. Una volta definito il file XML, l'orchestratore associa l'intento a una possibile implementazione definita a priori in base ai vincoli definiti e, successivamente, per gestire la *multi-tenancy*, va a cercare se esiste già un gruppo di utenti che hanno richiesto lo stesso tipo di servizio. Se esiste già un gruppo, allora gli intenti che hanno definito gli stessi vincoli vengono raggruppati e serviti allo stesso modo, altrimenti viene creata una richiesta per un nuovo tipo di servizio. Anche in questo caso, gli algoritmi che definiscono il nuovo servizio differiscono in base alla combinazione di vincoli definita con l'intento.

Il lavoro in [9], estrae le conoscenze e i metodi dal mondo delle reti per applicarli nel contesto delle blockchain (BC), ovvero implementazioni crittografiche fondamentali per l'utilizzo di valute virtuali. In particolare l'idea è quella di selezionare l'implementazione migliore di blockchain a fronte di alcune esigenze, espresse dall'utente con il linguaggio degli intenti. Il linguaggio proposto, che prende il nome di Controlled Natural Language (CNL), permette di comporre gli intenti usando parametri predefiniti quali condizioni, che permettono di selezionare politiche già attive, selettori che definiscono una blockchain dal set delle politiche attive, filtri che restringono il campo delle BC valide e modificatori che alterano il processo di selezione o filtraggio. A questo punto l'intento viene elaborato da una macchina a stati finiti che ne determina la correttezza della sintassi, completando la fase di validazione. Nella fase fase di traduzione avviene la raffinazione dell'intento, in cui vengono estratte molte altre informazioni non esplicitamente definite nell'intento che vengono poi inserite all'interno di un file JSON: questo file costituisce una rappresentazione di tutti i vincoli da rispettare nella scelta di una determinata BC, che può essere sottomesso ad una piattaforma per selezionare quella più adeguata.

Nell'articolo in [10] viene proposto un ulteriore linguaggio di definizione degli intenti e una nuova NBI. Il linguaggio utilizzato, Gherkin[17], permette di definire gli intenti in inglese e poi tradurli in un sistema pre-compilato di politiche che forniscono un livello medio di astrazione. Come ultimo passo del processo, le politiche vengono mappate su regole di basso livello dettate dall'architettura SDN sottostante. Tramite Gherkin, vengono descritte le caratteristiche del servizio virtuale che si vuole ottenere o del comportamento desiderato dalla rete in con uno stile poco strutturato e vincolante. Ogni dichiarazione inizia con una parola che verrà considerata la chiave da cui partire per definire il contesto di quella specifica riga. Una volta composto l'intento contenente svariate informazioni sul servizio richiesto, viene inoltrato al modulo di compilazione che tenta di effettuare un

[16]Si tratta di un'architettura in cui il software viene eseguito in un'unica istanza in grado, però, di servire più gruppi di utenti

[17]https://cucumber.io/docs/gherkin/reference/

mapping tra le parole chiave (la prima di ogni riga) e definizioni di più basso livello; se viene trovata una corrispondenza, l'interprete traduce gli intenti in politiche.

Gli autori del lavoro in [11] propongono un'interfaccia basata su intenti per una rete di backhaul[18], basata su tecnologia SDN. Lo scopo è quello di definire e gestire una rete virtualizzata che possa gestire il traffico prodotto da una rete wireless locale in uno stadio. Essendo una tecnologia broadcast, potenzialmente ogni dispositivo potrebbe accedere alla rete da più punti di accesso contemporaneamente, portando ad un traffico duplicato nella rete di backend degradando, così, le prestazioni. Il traffico generato deve essere etichettato tramite gli intenti in modo da dirigerlo verso le funzioni di rete corrette, così da filtrare il traffico duplicato oppure da effettuare un'analisi del traffico attraverso un Deep Packet Inspection. Tramite un linguaggio di tipo dichiarativo, è possibile definire per ogni link una destinazione ed eventualmente una sorgente del traffico. È infine necessario definire la classe di traffico cui applicare gli intenti: il traffico selezionato, passerà per la rete di backhaul per raggiungere la destinazione, dopo averlo analizzato come definito dagli intenti.

L'ultimo lavoro analizzato introduce una soluzione per IBN chiamata INSpIRE che implementa una tecnica basata sulla raffinazione degli intenti per tradurli in configurazioni di basso livello [12]. Le configurazioni prese in considerazione includono sia quelle da applicare su dispositivi fisici, sia su VNF. Per assicurare il corretto funzionamento della piattaforma, è necessario che alcune informazioni siano fornite da un operatore. Infatti, il framework deve conoscere a priori gli apparati e le VNFs presenti nell'infrastruttura di rete, oltre alle informazioni che li descrivono. Il processo di raffinazione procede alternativamente in due fasi, una costituita dai casi di utilizzo degli apparati fisici e una in cui si configurano funzioni virtualizzate. Nel primo caso, gli intenti passano per un classificatore di traffico che li traduce in "oggetti di traffico", ovvero strutture dati contenti informazioni come sorgente, destinazione e tipo di traffico, che poi saranno utilizzati per indirizzare il traffico. Nel caso in cui le configurazioni abbiano come target le VNF, gli intenti vengono elaborati da un identificatore di funzioni che ne estrapola il contesto di utilizzo e lo inoltra al componente incaricato ad istanziare la funzione corretta, effettuando un mapping in un database con implementazioni pre-compilate.

2.2 Estrazione di intenti

Cercando di generalizzare il più possibile gli approcci alternativi a quelli definiti precedentemente (sezione 2.1) trovati in letteratura, si potrebbe definire un filo conduttore che percorre i lavori descritti in questa sezione, ovverosia l'utilizzo degli intenti come punto di arrivo e non come informazione di input su cui basare alcune azioni future.

In un lavoro del 2018 [13], viene discussa una strategia per estrarre gli intenti da frasi definite in linguaggio naturale, ricorrendo in modo massiccio alle tecniche e architetture definite dal campo del data processing. Grazie al framework DialogFlow[19], è possibile sfruttare un modello di machine learning[20] in grado di riconoscere, dopo averlo "allenato" con dei casi di esempio, le entità all'interno di una frase scritta o pronunciata. Gli

[18] Indica la porzione di una rete gerarchica che contiene i collegamenti tra la rete centrale, estesa su scala geografica, e le sottoreti ai margini della rete gerarchica

[19] https://dialogflow.cloud.google.com/

[20] Tecnica appartenente alla branca dell'intelligenza artificiale che permette ad una macchina o ad un algoritmo di apprendere informazioni autonomamente dall'analisi dei dati

autori evidenziano, infatti, la possibilità di interagire attraverso una chat o utilizzando gli assistenti vocali. Il processo di estrazione prevede una fase ulteriore di raffinazione, nella quale entra in gioco un modello di apprendimento basato su due reti neurali[21] che permette di tradurre le entità precedentemente estratte in intenti strutturati espressi con un linguaggio personalizzato chiamato *Nile*. L'articolo definisce poi un possibile caso d'uso, virando verso l'approccio di questo lavoro, in cui si cerca di applicare il file scritto in Nile su una piattaforma di orchestrazione NFV; non sono forniti ulteriori dettagli implementativi, al contrario si dichiara l'incompletezza della soluzione nei riguardi delle reti SDN. L'utilizzo dell'intelligenza artificiale è apparso come una strategia molto interessante, ma l'elevata complessità nei modelli di addestramento rendono questo approccio poco flessibile. Infatti, l'accuratezza della traduzione delle entità dipende fortemente dalla bontà dei modelli utilizzati in fase di addestramento; a questo va aggiunto il periodo di tempo, non trascurabile, richiesto dal processo di addestramento e la necessità di aggiornarlo qualora si volesse estendere il modello con nuove entità.

Il lavoro in [14] sposa un approccio ibrido in cui il modulo basato sugli intenti ha lo scopo di mostrare lo stato attuale delle reti virtualizzate implementate, ma anche di fornire uno strumento di management. L'idea di fondo rispecchia quelli che sono i punti cardine della tecnologia SDN, che nasce dalla necessità di implementare reti virtualizzate slegandole, di fatto, all'hardware sottostante. Tramite un'architettura creata ad hoc che prende il nome di DOVE (Distributed Overlay Virtual Ethernet network), si cerca di realizzare quella sovrapposizione tra livello fisico e virtual appliance che è stata l'obiettivo di partenza di SDN. Ciò comporta, ovviamente, numerosi vantaggi, di cui il più importante ai fini dell'articolo è sicuramente la possibilità di isolare le reti virtuali dagli apparati fisici, non curandosi della tecnologia usata e abilitando ciascuna rete a definire le proprie caratteristiche. Gli intenti entrano in gioco dal momento in cui si ha la necessità di riflettere in modo inequivocabile quelle che sono le intenzioni espresse dal design della rete: i network blueprint definiscono la topologia della rete, gli intenti definiscono le motivazioni di quel design. È citato inoltre l'utilizzo del modulo a intenti per definire le topologie stesse, dichiarando le funzioni implementate, le connessioni e tutti i parametri necessari a definire una rete virtuale.

L'ultimo lavoro che andremo ad analizzare è certamente il più interessante in questa sezione; si tratta di un articolo del 2020 [15] e sposa in pieno l'approccio comune definito in principio. Lo scopo degli autori è quello di colmare la carenza di soluzioni riguardo le interpretazioni delle direttive di basso livello, rappresentandole sotto forma di intenti. Un possibile utilizzo, sicuramente di grande interesse, è quello di rappresentare ad alto livello le regole di un firewall, definendone così il comportamento, che risulterebbe altrimenti di difficile comprensione. Partendo, dunque, da un approccio bottom-up, un programma sviluppato dagli autori (SCRIBE, SeCuRity Intent-Based Extractor) è incaricato di estrarre le configurazioni dai componenti di rete, operando un reverse-engineering in base al fornitore; queste vengono sintetizzate e raccolte in un database centralizzato, sfruttando una rappresentazione intermedia. Nella fase di consolidamento, l'obiettivo è quello di innalzare ulteriormente il livello di astrazione del modello di rete precedentemente realizzato. Questo viene ottenuto analizzando il modello intermedio al fine di estrarre le parole chiave che costituiranno un modello indipendente dal fornitore. In questo passaggio, è fondamentale raggruppare entità che abbiano caratteristiche simili, in modo da formare

[21]Modello computazionale in grado di rappresentare relazioni complesse tra i dati in ingresso e quelli in uscita che altre funzioni lineari non sono in grado di modellare. Ogni stato intermedio tra l'ingresso e l'uscita è rappresentato da un nodo con connessioni multiple

poi un unico intento (es. stesse interfacce di origine e destinazione). Il penultimo passaggio consiste nell'arricchimento di tale struttura dati con informazioni complementari raccolte da fonti esterne, creando, così, un meta-intento. Topologia, nomi degli host, porte, interfacce e altri parametri di questo tipo sono utilizzati per arricchire un meta-intento. Al fine di fornire una rappresentazione di più alto livello, nell'ultimo passaggio il meta-intento è ulteriormente processato e tradotto utilizzando un linguaggio già citato precedentemente, Nile. L'estensione di tale linguaggio permette di definire numerosi obiettivi, in modo da ottenere un mapping preciso tra ciascuna delle entità definite nel meta-intento e le parole chiave accettate da Nile (es. "endpoint", "protocol", "address"). Tutto il processo di estrazione è raffigurato nella figura 2.6.

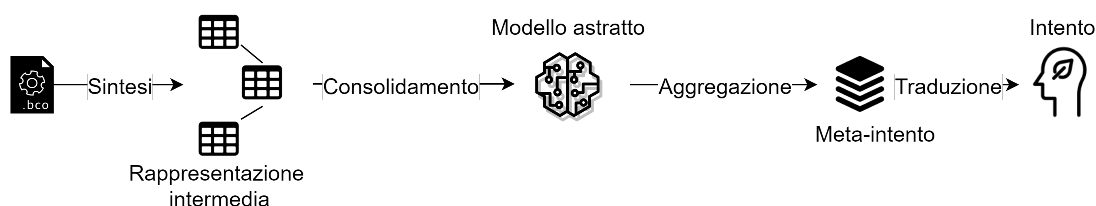


Figura 2.6. Processo di estrazione degli intenti

Capitolo 3

Approccio intent-based

In questo capitolo verranno illustrati il concetto di *intento*, che è il cuore di questo lavoro, insieme alle nozioni ad esso correlate e i contesti in cui si inserisce.

Negli anni il mondo delle telecomunicazioni ha cercato di migliorare il proprio approccio prettamente “statico” verso le novità tecnologiche apparse sul mercato: basti pensare agli scenari che hanno dominato per anni il design delle reti, ai protocolli e agli standard che, immutati, sono applicati da molto tempo. Tuttavia, negli ultimi anni, si sono verificati grandi cambiamenti non solo architetturali ma anche di visione. Grazie anche all'influenza del mondo del cloud computing, che ha introdotto concetti nuovi e in rapida espansione, è cambiata l'identità degli operatori e delle aziende che hanno maturato una nuova idea di gestione delle reti. Un ulteriore cambiamento nella gestione delle reti potrebbe scaturire dalla tecnologia ad intenti, con lo scopo di ottenere un grado di automazione superiore.

3.1 Reti autonome

L'automazione nelle reti è un concetto verso cui i provider nutrono ancora incertezza rispetto al suo valore[1], seppur ad oggi largamente utilizzato. Infatti, il mercato del mondo telco è molto differente da quello che si può trovare in altri settori, seppur molto vicini. Il mercato mobile, ad esempio, è estremamente stratificato, in cui ogni componente è realizzato da un provider specializzato in quel settore, a partire dal fornitore di silicio a quello del sistema operativo. Di contro, il mercato del networking si è sviluppato come un monolite, per semplicità e per esigenza delle aziende che, in presenza di eventuali guasti o manutenzione, potevano evitare di intaccare i loro profitti in quanto il provider si sarebbe adoperato a rimettere in piedi il sistema [16]. Questo sistema monolitico ha fatto nascere colossi come Cisco, Juniper, Huawei, DELL, VMware che per anni sono stati abituati a fornire, in un unico blocco, sia l'hardware che l'appliance di rete. Di contro, la separazione dei vari livelli, avrebbe portato le aziende ad avere più fornitori per lo stesso apparato, significando quindi una maggior difficoltà nel gestire imprevisti con un conseguente calo dei ricavi[2]. È questo, insieme all'opposizione dei fornitori maggiori, il problema di fondo che ha rallentato il processo di innovazione dei concetti di rete; a fare da spartiacque però sono state alcune aziende minori e startup, che hanno sperimentato per

[1] https://www.mef.net/edge-view-blog/service-providers-move-to-network-automation/

[2] https://www.nibusinessinfo.co.uk/content/choose-between-single-or-multiple-supplier-strategy

prime un approccio orientato alla separazione dei fornitori. Gli hardware acquistati erano costituiti da "semplici" schede di silicio (i cosiddetti COTS) su cui installare qualsiasi tipo di applicazione, come ad esempio router, firewall e così via[3]. Questo approccio ha portato numerosi vantaggi, che di fatto hanno superato le difficoltà delle aziende, rendendolo una strategia molto diffusa. Inoltre gli ultimi anni sono stati caratterizzati da numerose innovazioni tecnologiche che hanno facilitato l'utilizzo delle reti da parte di molti dispositivi, anche apparentemente fuori da questo contesto, scatenando una serie di azioni che hanno contribuito all'espansione di Internet e alla diffusione delle reti[4]. Oltretutto, ciò, unito agli standard sempre crescenti riguardo i parametri fondamentali quali latenza, affidabilità e continuità di servizio, ha portato all'esplosione della complessità stessa delle reti, rendendo la gestione sempre più complessa e costosa[5]. La spinta decisiva arriva dalla proliferazione delle tecnologie di softwerizzazione della rete, che gravano in modo importante, con il loro quantitativo di dati necessari, sulla gestione di rete. Lo scenario raffigurato rende necessario lo sviluppo di un nuovo protocollo di gestione.

3.1.1 Definizione e principi

Le reti autonome, definite per la prima volta da IBM[6] nel 2001[7], si basano sul concetto fondamentale di eliminare dal ciclo di controllo della rete qualsiasi sistema esterno e renderlo di fatto un sistema chiuso, mirando ad ottenere le capacità di ottimizzarsi, di auto-configurazione e che sia in grado di reagire autonomamente ai guasti (RFC-7575 [17]). Per auto-configurazione si intende la capacità della rete di di configurare e riconfigurare i propri apparati, basandosi su conoscenze estratte, metadati già in possesso del sistema o su intenti. Il processo basilare dell'automazione prevede, infatti, l'estrazione e la collezione di dati a partire da informazioni di rete, di gestione e da conoscenze derivate dai metadati inseriti dall'utente, al fine di presentare una risposta adattiva e dinamica. Fondamentale in questo contesto, la capacità di ottimizzarsi permette alla rete di gestire la mole di dati trattata nella fase di estrazione e migliorare, se possibile, le performance. Questo comportamento è ottenuto per mezzo di misure ripetitive catalogate dalla rete che, misurando la deviazione dal comportamento atteso e ideale, definisce delle strategie per perfezionare l'utilizzo delle risorse e mitigare il calo di prestazioni. Con l'automazione delle attività e delle funzioni di rete e con il controllo e la gestione automatica dei processi ripetitivi, la disponibilità del servizio di rete migliora. Un'altra proprietà sicuramente centrale è l'auto-guarigione che permette alla rete di reagire in caso di errori inattesi, possibilmente nel minor tempo possibile, così da garantire una continuità al servizio. Nel set degli errori rientrano anche potenziali attacchi, quali ad esempio gli attacchi di DoS o di DDoS[8].

Analizziamo ora quelli che sono i principi cardine della *network automation*:

[3] https://www.networkworld.com/article/2161400/why-sdn-will-accelerate-cots-adoption.html

[4] https://www.zrix.com/blog/impacts-of-internet-of-things-on-society

[5] https://www.oracle.com/corporate/pressrelease/enterprise-complexity-092518.html

[6] https://www.ibm.com/

[7] https://www.ietfjournal.org/autonomic-networking/

[8] Denial of Service, attacco informatico ad un sistema perpetrato da una fonte che mira a saturare le risorse di un server rendendolo, quindi, non disponibile; si parla di Distributed Denial of Service se l'attacco non proviene da un'unica fonte, ma da un sistema distribuito che prende spesso il nome di botnet

- *API unificate*: indipendentemente dal fornitore, è necessario disporre di metodi universalmente validi da interrogare per ottenere informazioni o iniettare configurazioni. I protocolli di comunicazioni fra le parti, infatti, devono essere definiti in modo da essere supportati da tutti i dispositivi;

- *configurazione unificata*: come nel caso dei protocolli di comunicazione, è necessario definire modelli di dati validi per tutti i fornitori;

- *archiviazione dei dati di configurazione*: un insieme di possibili configurazioni pronte all'uso secondo casi ben noti, come ad esempio la configurazione iniziale ma anche quella corrente;

- *test e validazione delle configurazioni*: se il processo di estrazione di informazioni conduce alla generazione di una determinata configurazione, è necessario, poi, verificare che effettivamente la configurazione ottenuta sia quella che conduce al comportamento atteso.

3.1.2 Obiettivi

Gli obiettivi di design ad alto livello delle reti autonome sono molteplici e prescindono, dichiaratamente, dal tipo di soluzione adottata [18]. Uno dei primi obiettivi che bisogna considerare nella progettazione di una rete autonoma è sicuramente l'integrazione e la coesistenza con altri paradigmi di gestione della rete. Essendo l'automazione un processo continuo di raccolta di informazioni e di apprendimento della rete e, essendo i nodi di rete con comportamento autonomo ancora un eccezione rispetto a soluzioni più semplici, è necessario supportare la coesistenza di questo paradigma con altri. Necessario, ad esempio, lo sviluppo di un rapporto di non conflittualità con altri protocolli di gestione di rete, quali ad esempio NETCONF, SNMP (RFC-1157 [19]), SDN o riga di comando. Tale rapporto è costruito su un processo di gestione delle priorità: come accade nelle politiche di routing, una rotta fondamentale definita manualmente da un amministratore di rete ha la massima priorità, viene cioè designata anche se i protocolli di routing ne individuano altre. Analogamente, il comportamento autonomo non può essere settato alla massima priorità, altrimenti nessun altro protocollo potrebbe intervenire, qualora fosse necessario, per mitigare eventuali errori nella configurazione. Al contrario, generalmente, più il processo richiede un grado di astrazione elevato tanto più sarà impostato su una priorità bassa. Per questo, la linea generalmente seguita nel campo delle reti autonome prevede una priorità minima per il comportamento di default, una priorità media per gli intenti, e la massima priorità per protocolli specifici per ogni nodo, come quelli menzionati precedentemente. Tuttavia anche selezionando diversi gradi di priorità, è fondamentale attuare dei processi al fine di mantenere un controllo sottotraccia da parte delle funzioni autonome che possano entrare in gioco qualora, a causa di una mal configurazione causata dai classici metodi di gestione, la rete raggiungesse uno stato indefinito.

Secure by Default è un obiettivo fondamentale nella progettazione di reti autonome che mira a rendere sicure le interazioni fra processi e dispositivi. È necessario infatti regolare l'autenticazione di ogni dispositivo che deve asserire la sua appartenenza a quel dominio di rete, tipicamente ricorrendo all'utilizzo certificati rilasciati da un'autorità di certificazione fidata, che dimostrino la corrispondente identità. Ciò permette ad ogni nodo di identificare i limiti di quel dominio, dovuto alla conoscenza dei dispositivi con un certificato valido per quel dominio, e di rendere la rete sicura senza dover configurare altri parametri per le interazioni, come ad esempio le chiavi simmetriche. Nonostante la separazione netta dei domini di rete per mezzo di domini di identità, ogni nodo può

comunque interfacciarsi con nodi di diversa appartenenza in quanto condividono un punto nella catena delle autorità di certificazione, fidata per entrambi i domini.

Un altro concetto che possiamo annoverare tra gli obiettivi di un rete autonoma è sicuramente l'alto livello di astrazione. Per ottenere un comportamento di questo tipo, è necessario che l'amministratore di rete abbia la possibilità di interfacciarsi con protocolli di alto livello, in cui è possibile dichiarare le configurazioni senza scendere nei dettagli (concetto di "intento" analizzata nel dettaglio nella sezione 3.2). Questo grado di astrazione permette non soltanto di interagire con le configurazioni senza curarsi dei dettagli implementativi, ma anche di ottenere segnalazioni e informazioni ad alto livello dalla rete stessa.

La definizione di un'infrastruttura comune per tutte le funzioni di rete autonome è un argomento topico per questa tecnologia. Le funzioni di rete autonome attualmente sviluppate hanno raggiunto un numero elevato e sempre crescente, tuttavia necessitano ciascuna dei propri protocolli e architetture per poter comunicare, definire e distribuire politiche etc. Lo scopo che, a tal proposito, si è prefissato il gruppo dell'IETF[9] è quello di definire un insieme di servizi comuni di controllo e gestione per quanto concerne la negoziazione, diagnostica, monitoraggio e distribuzione di intenti.

Riguardo le funzioni autonome implementate nelle reti, è necessario definire un piano di controllo per i nodi per rendere possibile la comunicazione. L'implementazione di questo componente deve prescindere dal tipo di nodo che lo ospita; gli switch di rete, i router, un server o qualsiasi altro apparato devono supportare questo tipo di funzioni.

3.2 Introduzione agli intenti

In questa sezione verranno presentati i concetti base riguardo la tecnologia basata su intenti e gli eventi che hanno permesso il suo sviluppo e la sua evoluzione. Infine saranno analizzate le differenze con alcuni concetti già noti evidenziandone le innovazioni.

La volontà di semplificare la gestione di rete negli aspetti delle mansioni ripetitive e la ricerca di un supporto ausiliario nella gestione di reti sempre più complesse potrebbero trovare una soluzione adottando una visione d'insieme, in cui non ci si concentra sulla configurazione del singolo dispositivo. Questo rappresenta, infatti, uno degli aspetti topici delle reti autonome. Tuttavia, nonostante la caratteristica di questo tipo di reti preveda l'esposizione di proprietà di "auto-gestione" reagendo continuamente ai cambiamenti raccolti e analizzati dalla rete stessa, è necessario, in ogni caso, l'intervento di un operatore di rete che fornisca le informazioni iniziali. Questo input, che proviene dall'esterno del sistema stesso, prevede l'iniezione di informazioni cruciali, quali ad esempio obiettivi e servizi che la rete deve offrire, ogni volta che si abbia il desiderio di modificare il comportamento del sistema. Ci si riferisce a questo concetto con il nome di "intento", alle reti che necessitano di un input esterno per definire il proprio comportamento con il nome di "Intent-Based Networks" (IBN) e ai sistemi che aiutano ad implementare gli intenti con il nome di "Intent-Based Systems" (IBS). Tuttavia, il concetto di intento ha una valenza molto più ampia rispetto al campo delle reti autonome; non riguarda, infatti, solamente la capacità di definire un tipo di interazione tra operatore e rete, ma riguarda anche la capacità di fornire all'operatore la possibilità di concentrarsi solamente su quale dovrebbe essere il risultato della rete lasciando che i dettagli implementativi siano risolti dal sistema sottostante.

[9] Internet Engineering Task Force

3.2.1 Definizioni

Il termine intento viene introdotto per la prima volta nel 2015 all'interno del RFC-7575 [17] riguardo l'ambito delle reti autonome. Tale documento dell'IETF si riferisce al concetto in esame come una politica astratta di alto livello utilizzata per gestire una rete". Per completezza, ai fini della comprensione dell'argomento, si definisce il termine di politica di rete come un insieme di regole, vincoli e impostazioni che permettono di definire alcuni aspetti comportamentali di rete: si può, ad esempio, definire chi e sotto quali circostanze è autorizzato a connettersi alla rete ed eventualmente effettuare alcune operazioni. Sintetizzando, secondo questa definizione, la definizione di intento riguarda un particolare tipo di politica inserita dall'amministratore di rete al fine di fornire una direzione operativa alle reti autonome. Chiaramente, ad oggi la comunità si riferisce al concetto di intento con un'accezione molto più ampia, non limitata, cioè, alla semplice sinonimia con una nozione già presente in letteratura. Infatti, pur riferendosi agli intenti come un'indicazione fondamentale da parte di un'entità esterna per il corretto funzionamento delle reti autonome che, altrimenti, non avrebbero modo di ottenere, essi rappresentano, comunque, una modellazione di quelli che sono i comportamenti desiderati. Delineano le "intenzioni" di un operatore di rete riguardo il funzionamento della rete stessa e, quindi, si tratta di un concetto molto più ampio che non può essere ridotto alle sole reti autonome, intese come reti che dispongono di un piano di controllo autonomo (che implementano cioè un ciclo di controllo continuo di verifica e azione). Più nello specifico, l'intento rappresenta una dichiarazione di obiettivi operazionali che la rete deve raggiungere e i risultati che la rete dovrebbe fornire, senza specificare come ottenerli [20]. Essendo una definizione dichiarativa, essi applicano alcuni concetti fondamentali:

- astrazione dei dati: permette una separazione dei compiti che permette agli utenti di non dover amministrare configurazioni di basso livello e di non dover conoscere i dettagli tecnici implementativi;
- astrazione dalla logica di controllo: permette all'utente di disinteressarsi dal procedimento necessario ad ottenere un determinato scenario che, limitandosi a definire il risultato desiderato, cede il compito all'IBS che traduce in una serie di azioni.

Un'altra possibile definizione è fornita da Open Networking Foundation (ONF) [21]. Il concetto di intento, secondo l'accezione di ONF, definisce semplicemente un'interfaccia dichiarativa fornita da un controllore. Questo prevede chiaramente la presenza di funzioni centralizzate che siano in grado di interagire con l'utente, ricevere ed elaborare l'intento. Successivamente il processo prevede che gli intenti vengano trasformati in regole di basso livello e orchestrate lungo la rete. Quella fornita da ONF è certamente una definizione diversa, che poggia su basi più pratiche e meno teoriche, fornendo, però, una visione più ristretta di quella definita in precedenza. La sola interazione con una rete basata su intenti non definisce completamente il comportamento di questa tecnologia.

Per chiarezza, in conclusione, sono riportati alcuni esempi "letterali" di ciò che potrebbe rappresentare un intento e ciò che invece si discosta dalla definizione, seppur spesso confusi.

Esempi di intenti espressi in linguaggio naturale:

- "Il servizio VPN nella rete deve sempre fornire protezione per tutti i percorsi";
- "Devia il traffico generato da una sorgente verso una destinazione evitando il passaggio attraverso una determinata area geografica";

- "Massimizza l'utilizzo della rete anche a costo di gravare sulla qualità del servizio fin quando non abbia raggiunto un deterioramento del 20% rispetto ai valori standard".

Esempi di dichiarazioni che concettualmente non definiscono un intento, espressi in linguaggio naturale:

- "Configura una determinata interfaccia di rete con un indirizzo IP";
- "Quando un collegamento raggiunge una saturazione superiore al 30% emetti una segnalazione";
- "Configura un tunnel VPN tra la sorgente A e la destinazione B attraverso il percorso P".

L'ultimo esempio riportato nel secondo elenco corrisponde, a grandi linee, con l'approccio utilizzato in questo lavoro nella gestione degli intenti. Anche se riportato nell'elenco degli utilizzi impropri del linguaggio dichiarativo, l'utilizzo ideale, caratterizzato da una gestione autonoma della rete, era fuori dagli scopi del progetto. Piuttosto, la dichiarazione di questo tipo di intenti, in continuità con l'utilizzo degli intenti da parte del framework sfruttato che permette di costruire una topologia virtuale utilizzando un linguaggio dichiarativo, ha perfettamente coinciso con le necessità del lavoro stesso. Con questo tipo di espressioni ad alto livello è possibile definire il tipo di servizio desiderato senza specificare le azioni da attuare per realizzarlo. Per chiarezza è bene precisare nuovamente che l'obiettivo non è quello di gestire una rete autonoma e tutti i suoi servizi, bensì quello di istanziare servizi di sicurezza dichiarandoli solamente ad alto livello.

3.2.2 Principi e novità rispetto ai concetti noti

Prima di procedere nella descrizione delle novità apportate dall'approccio intent-based è opportuno passare in rassegna, molto brevemente, i concetti collegati con cui effettuare un confronto:

- un *Service Model* costituisce un modello che rappresenta un servizio fornito dalla rete, estrapolandone i dettagli senza dipendere dal tipo di implementazioni e descritti in un schema portatile che può essere utilizzato indipendentemente dall'ambiente di utilizzo. Un esempio di servizio potrebbe essere una Virtual Private Network (VPN) di livello 3, in cui vengono definiti i nodi terminali. È necessario, tuttavia, mantenere traccia di come mappare il servizio rispetto alle risorse di rete e salvare il legame tra gli oggetti di alto livello con quelli di più basso livello. Infine, un processo di orchestrazione provvede a iniettare le configurazioni corrette nei corrispondenti dispositivi, controllando se queste vengono accettate e se conducono il sistema verso lo stato desiderato.
- una *politica* è un insieme di regole che governano le scelte nel comportamento di un sistema, andando a modificare o mantenere lo stato di uno o più servizi gestiti. Tali regole consistono in una serie di eventi, la cui occorrenza attiva una regola, una serie di condizioni che devono essere verificate ed eventualmente una serie di azioni in risposta.

I concetti delineati precedentemente hanno in comune con la definizione di intento sicuramente il fatto di coinvolgere un alto livello di astrazione della rete e di non dipendere

da ogni singolo dispositivo. Tutti i concetti tentano di fornire una gestione della rete più snella e dedicata ad una visione d'insieme della rete, piuttosto che alla configurazione manuale di ogni dispositivo.

Rispetto ai service models, gli intenti non hanno dipendenze dai modelli di basso livello, necessarie nella corrispondenza tra il servizio e le risorse dell'infrastruttura di rete. Inoltre, non hanno bisogno (solo a livello concettuale, nella pratica esiste un componente fisicamente distribuito per questo scopo) di un orchestratore per gestire il servizio in ciascun apparato.

Rispetto alle politiche, come già accennato precedentemente, non prevedono l'utilizzo di catene operative che, una volta scatenate, attuano una serie di processi per cambiare lo stato della rete. Le politiche, infatti, lasciano all'utente il compito di definire solamente cosa fare sotto alcune circostanze.

Procediamo ad analizzare i principi fondamentali su cui sono fondati i sistemi che operano secondo un paradigma a intenti, enunciati in [20]:

- *Single Source of Truth*: SSoT è la pratica di modellazione dati che prevede l'aggregazione di informazioni estratte da modelli di dati in un'unica locazione. In un sistema di gestione basata su intenti la SSoT è costituita dall'insieme degli intenti già validati. Questo permette di operare un confronto tra lo stato desiderato e quello attuale, calcolando eventuali deviazioni e mettendo in atto le azioni correttive;

- *One-touch but not one-shot*: questo principio si riferisce alle modalità in cui un sistema accetta in input alcune forme dichiarative di intenzioni e in che modo queste vengono tradotte in operazioni. Infatti, l'utente definisce gli intenti secondo una forma prestabilita e accettata dal sistema, al contrario, un sistema "zero-touch" sarebbe in grado di riconoscere tali espressioni all'interno di qualsiasi tipo di dato. Il processo di traduzione e implementazione, che a partire da una serie di intenti porta la rete in un nuovo stato, non è un processo istantaneo ma incentrato su un sistema iterativo che, con il susseguirsi di azioni designate, permette di mettere in atto quelle che erano le intenzioni dell'operatore di rete;

- *autonomia e supervisione*: questo principio, già incontrato in precedenza, permette al sistema di raggiungere il giusto grado di autonomia per rispondere alle esigenze dell'utente e di condurre i propri compiti senza un intervento esterno e, inoltre, garantisce il livello corretto di supervisione, provvedendo a fornire all'utente le corrette informazioni;

- *apprendimento*: un sistema intent-based attua un processo continuo di apprendimento, in modo da poter soddisfare sempre le volontà dell'utente. L'apprendimento avviene grazie all'interazione con l'operatore che, a fronte di un intento non corrisposto dalla rete, provvede ad attuare le modifiche "insegnando" di fatto alla rete il giusto stato corrispondente a quell'intento;

- *esposizione delle capacità*: definisce la capacità di comporre e scomporre gli intenti e mapparli con le corrispondenti capacità e servizi della rete;

- *astrazione*: fa riferimento ad un concetto già ampiamente analizzato, secondo cui l'utente non deve preoccuparsi di come le sue intenzioni vengano messe in atto nel sistema, ma solo di definirle in modo dichiarativo.

3.3 Intent-based networking

In questa sezione verranno presentati gli aspetti fondamentali delle reti e dei sistemi basati su intenti per poi fornire una modellazione architetturale e un excursus sull'evoluzione storica di questa tecnologia.

Per chiarezza, è bene sottolineare brevemente una piccola differenza tra due concetti incontrati spesso e a volte utilizzati come sinonimi: Intent-Based Networks (IBN) e Intent-Based System (IBS). Ci si riferisce al concetto di IBN delineando una rete che può essere gestita utilizzando gli intenti; è sottintesa la capacità di riconoscere ed elaborare un intento definito da un operatore, adattando il proprio comportamento in modo da modificare lo stato della rete, senza richiedere all'utente la definizione dei dettagli implementativi. Il concetto di IBS si riferisce, in senso più stretto, ad un sistema che permette di amministrare la rete tramite intenti, mettendo a disposizione un'interfaccia interattiva per l'utente e relazionandosi con la rete per ottenere lo stato corretto.

Al fine di inquadrare al meglio il concetto di IBN, sono state raccolte varie definizioni che con il tempo sono state delineate da varie istituzioni:

- *Gartner*: definisce questo tipo di reti in base alle capacità che possono fornire [22]. In particolare:
  - Traduzione e Validazione
  - Implementazione automatizzata
  - Consapevolezza dello stato di rete
  - Garanzia del risultato e ottimizzazione dinamica
- *Cisco*: una rete intent-based mira a colmare il gap tra l'azienda e il settore IT[10], andando a catturare gli intenti e cercando di allineare, con un processo continua, l'infrastruttura alle necessità aziendali riguardo la sicurezza, servizi, processi produttivi etc. [23]
- *Huawei*: una rete intent-driven è in grado di operare basandosi sugli intenti di una strategia aziendale. Il cuore di questa rete è costituito da un motore proprietario (Network Cloud Engine) che è in grado di eseguire i task necessari a realizzare gli intenti [24].

Le sfide da affrontare, per una rete di questo tipo, sono ancora molteplici e devono far fronte spesso a problemi tecnici. Uno dei possibili problemi riguarda sicuramente le informazioni di log per quanto concerne il funzionamento, costringendo l'intera piattaforma a gestire una grande quantità di dati, come ad esempio l'utilizzo di memoria, CPU e traffico di rete, e renderli disponibili in tempo reale all'utente. Il ciclo continuo che mette in atto la rete potrebbe diventare un altro potenziale problema per la rete: durante la fase di verifica potrebbero sollevarsi numerose eccezioni dovute a ridondanza o correlazione dei dati e causare quindi ritardi aggiuntivi al processo. L'ultimo grande problema riguarda uno scenario già raffigurato nella prefazione di questo capitolo, in cui viene descritto il conflitto di interesse di ciascun produttore tra la fornitura di una soluzione personalizzata e l'utilizzo di un'implementazione definita da uno standard. Ciò ha rallentato il processo di definizione di un'interfaccia comune, quindi ogni implementazione dipende dal tipo di sviluppo messo in atto dal fornitore.

[10] Information Technology

3.3.1 Evoluzione delle IBN e stato dell'arte

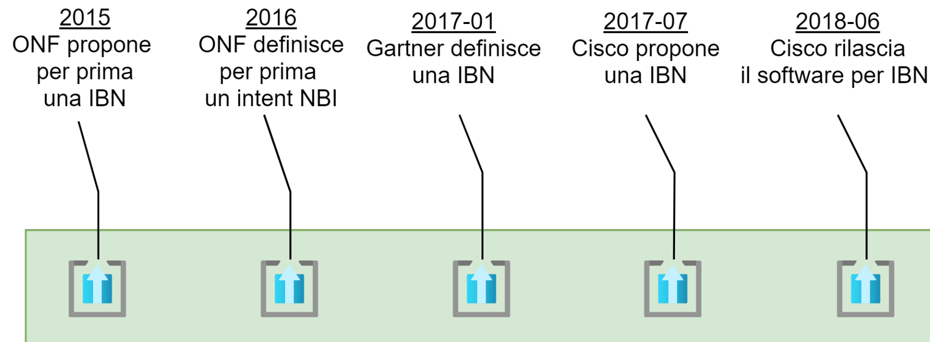


Figura 3.1. Evoluzione delle IBN

Come evidenziato dalla figura 3.1 l'evoluzione delle IBN può essere sintetizzato in alcune tappe fondamentali. La prima volta che il concetto di Intent-Based Network viene menzionato in letteratura è ad opera della fondazione ONF nel 2015; nell'occasione viene rilasciato un documento in cui vengono descritte le idee di base. Nel 2016, la stessa fondazione rilascia un nuovo documento in cui sintetizza, questa volta, i concetti per creare un'interfaccia (NBI) in grado di interagire con gli intenti. ONF definisce il funzionamento base dell'interfaccia, oltre che una possibile architettura e le sue proprietà. Il 2017 è la volta dell'istituto di ricerca Gartner che pubblica una sua definizione di IBN, come precedentemente descritto. Nel corso della metà del 2017 Cisco mostra i frutti dei suoi studi introducendo il concetto di "Automated Driving Network" e nel 2018 rilascia la prima versione del suo prodotto.

Gli istituti di standardizzazione attualmente stanno conducendo ricerche a riguardo, principalmente l'Internet Engineering Task Force (IETF) e l'istituto European Telecommunications Standards Institute (ETSI). In particolare il primo ha definito un progetto molto interessante, nel tentativo di sviluppare un modello di rete autonoma e una gestione automatizzata di rete, sotto il nome di ANIMA (RFC-8994 [25]). Ricorrendo all'utilizzo di YANG come linguaggio di data modeling, lo scopo è quello di migliorare l'efficienza nella gestione; per integrare il processo di configurazione si utilizza il protocollo NETCONF.

Naturalmente, l'argomento in esame non è trattato solamente da istituti ufficiali, ma anche aziende private, comunità open-source e ricercatori accademici. In particolare, i gruppi open-source stanno analizzando le relazioni tra il concetto di Software Defined Network (SDN) e IBN: in questo ambito, la prima viene considerata una possibile implementazione della seconda. I principali attori nel design di controllori SDN guidati da intenti e di interfacce intent-based sono sicuramente Open Networking System (ONOS) e OpenDaylight. Quest'ultima ha già sviluppato un modulo di astrazione, chiamato *Intent Component*, che astrae gli oggetti e le capacità della rete. Altri due progetti fondamentali sono sicuramente NIC[11] e NEMO[12] (descritto in dettaglio nella sezione 3.4.2): il primo, acronimo di Network Intent Composition, mira a fornire, come suggerisce il nome stesso, una grammatica completa e una sintassi ben definita per descrivere gli intenti, mentre il secondo sfrutta il linguaggio introdotto da Huawei per strutturare un framework in grado

[11] https://wiki.opendaylight.org/display/ODL/Network+Intent+Composition

[12] https://test-odl-docs.readthedocs.io/en/latest/user-guide/nemo-user-guide.html

di creare una rete virtuale gestita da intenti. Sicuramente degni di menzione il componente sviluppato da ONOS "Intent Framework" e i componenti orientati alle North Bound Interface quali "Congress" e "GBP" sviluppati da OpenStack.

Anche la Commissione Europea si è mossa in questa direzione, finanziando un progetto di sicurezza in grado di abbracciare i temi appena descritti. In particolare, il progetto FISHY[13] si basa sulla progettazione di uno strato di sicurezza finalizzato a rendere resiliente dal punto di vista della sicurezza l'intero sistema della catena di fornitura *Information and Communications Technology* (ICT), ma anche a misurare correttamente la completa conformità della sicurezza e di conseguenza innescare le azioni richieste. A questo scopo integra nella sua architettura diversi domini tra cui SDN, NFV e IBN. Per fornire un livello di resilienza adeguato in ambito di sicurezza ad una qualsiasi infrastruttura ICT considerata intrinsecamente insicura adotta una strategia basata, fra tutti, su funzioni di sicurezza di rete virtualizzate, per fornire sicurezza on-demand in base alle necessità, e su interfacce intent-based per la configurazione delle caratteristiche. Uno degli obiettivi di questo lavoro di tesi è quello di fornire un contributo al progetto FISHY rispetto ai componenti strategici appena citati.

3.3.2 Funzionalità

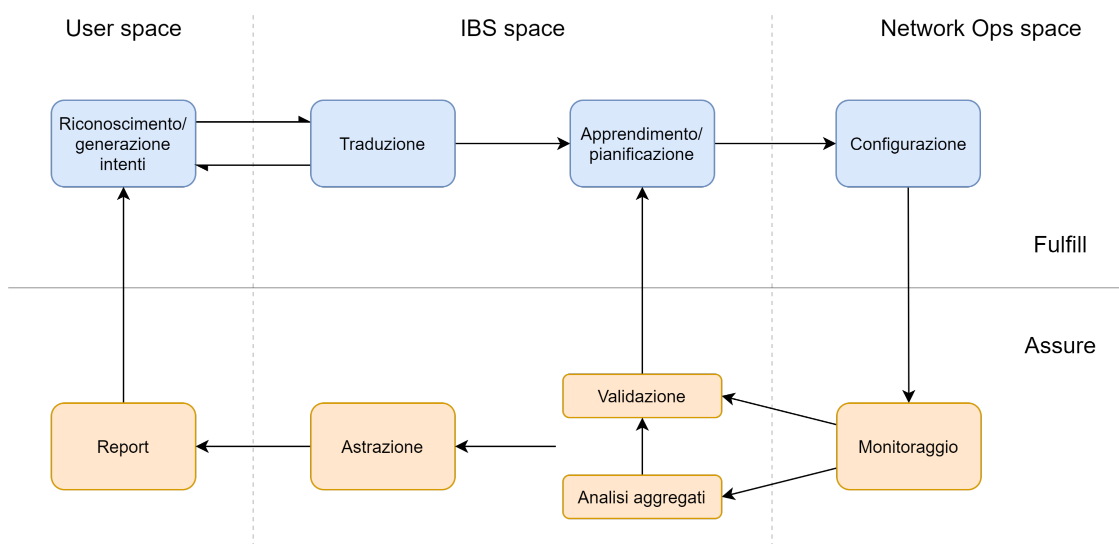


Figura 3.2. Ciclo di vita degli intenti

Utilizzando la figura 3.2 come riferimento, analizziamo le funzioni implicate nelle reti basate sugli intenti, descritte in [20]. Per semplicità nell'analisi si possono suddividere in due macro-categorie e per ognuna verranno descritte le funzioni principali:

- *realizzazione dell'intento*: fornisce le funzioni fondamentali per interfacciarsi con l'utente in modo da generare nuovi intenti e svolge tutti i task necessari per garantire che l'intento sia trasformato in uno stato della rete. Include, inoltre, tutte le funzioni per ottimizzare la resa del risultato e per gestire e collegare tutte le astrazioni di alto livello verso i dispositivi di più basso livello. Le tipologie di funzioni che operano in questo campo sono quattro:

[13] https://fishy-project.eu/

  - *riconoscimento e generazione di intenti*: questo gruppo di funzioni si riferisce specificamente all'interazione tra il sistema e l'utente. Fornisce, tipicamente, gli strumenti per definire un intento mediante un'interfaccia e possibilmente un processo di raffinamento che permette all'utente di specificare lo stato desiderato senza dover necessariamente utilizzare una sintassi precisa. L'interazione permette al sistema di guidare l'utente verso una dichiarazione sempre più precisa che a quel punto è possibile processare. Molto più interessante questo approccio lato utente rispetto al semplice utilizzo di API, ma richiede una capacità non ancora del tutto raggiunta della rete di apprendere il modo di scrivere di un umano;
  - *traduzione*: nel cuore del sistema IBS, il modulo di traduzione costituisce l'argomento più studiato in questo ambito in quanto responsabile di colmare il gap tra l'astrazione ad alto livello e il management della rete tradizionale. Si occupa di definire un insieme di azioni da intraprendere sulla rete a fronte di un particolare tipo di intento, che possano modificare le configurazioni degli apparati a basso livello;
  - *apprendimento*: modulo che spesso troviamo indissolubilmente legato a quello di traduzione a causa del loro lavoro sinergico, che dovrebbe portare ad ottener il miglior risultato possibile nel caso in cui più vie fossero percorribili. Tale modulo, infatti, permette all'intero sistema di evolvere continuamente, apprendendo da situazioni note e migliorando la resa dell'intento. In aggiunta, in alcuni casi, permette di prevedere lo stato futuro della rete, anticipando la risposta del sistema;
  - *configurazione*: si occupa di orchestrare e attuare le modifiche decise dai moduli precedenti lungo la rete.

- *verifica dell'intento*: raccoglie tutte le funzioni che entrano in gioco quando la configurazione è già stata modificata e occorre verificare che il comportamento ottenuto sia quello desiderato. Provvedono a monitorare la rete, misurare le eventuali deviazioni rispetto al comportamento atteso e forniscono una visualizzazione adeguata all'utente. I gruppi basilari sono:

  - *monitoraggio*: include le funzioni che lavorano sull'infrastruttura di rete al fine di raccogliere informazioni riguardo eventi e efficienza del servizio in modo da collezionarli e successivamente elaborarli;
  - *analisi e validazione*: ha il compito di analizzare i dati aggregati di report forniti dal reparto di monitoraggio in modo da valutare il lavoro delle funzioni di realizzazione dell'intento. Sfruttando anche il concetto di SSoT, si effettua un confronto tra lo stato attuale della rete e quello che ci si aspettava corrispondesse all'intento ricevuto in input. Una volta effettuati questi confronti, i dati elaborati possono essere inviati al modulo di apprendimento che, così, potrà basarsi su questo evento per prendere in futuro scelte potenzialmente differenti. Inoltre, tale modulo provvede a modificare l'attuale stato della rete qualora non venisse considerato corretto;
  - *astrazione e report*: necessarie per fornire all'utente un risultato ad alto livello di come sia cambiato lo stato della rete, collegato al suo intento. È necessario perciò fornire una visualizzazione astratta dei dati raccolti a basso livello affinché possano essere consegnati all'operatore in linguaggio di alto livello.

3.4 Linguaggi di definizione degli intenti

In questa sezione ci occuperemo di elencare i più importanti linguaggi di definizione degli intenti e analizzare le scelte compiute a riguardo in questo lavoro.

Prima di procedere è necessario fare un breve richiamo alle North Bound Interfaces che, ovviamente, saranno trattate nell'ambito degli intenti. Come definito in [26], una intent-based NBI è un modulo sottostante il livello applicativo che permette di collegarlo al modulo centrale di gestione della rete. La funzione fondamentale che la NBI svolge è sicuramente quella di traduzione degli intenti che permette di colmare il distacco tra l'alto livello con cui gli intenti vengono definiti e il livello di configurazione, in cui sono necessari comandi e azioni concrete per portare la rete in un nuovo stato. Grazie a due sotto-moduli, *Intent Compiler* e *Intent Solver*, la NBI è in grado di ricevere delle espressioni in linguaggio dichiarativo, compilarle in modo che siano meno orientate al linguaggio naturale e più strutturate in modo da poter essere elaborate, e poi risolverle, cioè trasformarle nel corretto flusso di esecuzione a basso livello. In particolare il primo sotto-modulo realizza le funzionalità di verifica della sintassi, assicurandosi che l'intento sottomesso sia adeguato alle regole di forma imposte dalla NBI, mentre il secondo processa gli intenti al fine di estrarre informazioni da utilizzare in fase di traduzione. In questo modo il controllore centralizzato può essere facilmente integrato con una tecnologia di questo tipo essendo, tuttavia, ignaro della trasformazione che le informazioni in ingresso hanno subito. L'introduzione di una NBI intent-based permette di gestire una piattaforme attraverso l'utilizzo di linguaggi dichiarativi che possono essere definiti in vari modi.

3.4.1 Linguaggi strutturati e in stile discorsivo

Nell'analisi dei linguaggi di definizione degli intenti è fondamentale partire da una premessa per chiarire lo scopo di utilizzarne uno piuttosto che un altro; il concetto di linguaggio per intenti non definisce di per sé una caratteristica specifica nella sintassi e molteplici stili possono essere utilizzati allo scopo. Si tratta, in effetti, di un linguaggio dichiarativo - questa, al contrario, deve essere una caratteristica - che è strutturato in modo da permettere di definire alcune azioni basilari senza scendere nei dettagli implementativi. Ciò significa che molti linguaggi possono essere utilizzati per definire gli intenti; a tal proposito meritano una menzione speciale i linguaggi Extended Markup Language (XML), Domain Specific Language (DSL) e JSON che, di frequente, trovano impiego nel campo degli intenti. Chiaramente, essendo un linguaggio destinato alla relazione con l'utente finale che, per dichiarazione di scopi delle piattaforme intent-based, potrebbe essere rappresentato da un utente medio (e non un operatore con competenze), l'utilizzo di linguaggi generici potrebbe portare complicazioni nell'utilizzo della sintassi e nell'eventuale estensione del linguaggio da parte degli sviluppatori. Per questo motivo, trattandosi di una scelta di intuitività per l'utente, pur potendo contare su svariati linguaggi purché supportati dai livelli inferiori, si ricorre spesso a linguaggi ad hoc, pensati per essere facili da utilizzare e assimilare. Tipicamente i linguaggi trovati in letteratura possono dividersi in due categorie principali, in base allo stile di scrittura che adottano [26]:

- *stile strutturato*: questa categoria racchiude una tipologia di linguaggi meno sofisticati, molto simili per impostazione e sintassi, per esempio, ad un file JSON. L'intento risulterà quindi definito in modo schematico in cui per ogni riga viene definita una parola chiave e il corrispondente valore;

- *stile discorsivo*: più semplicemente, questa categoria rappresenta una tipologia di intenti più sofisticati e più intuitivi in quanto espressi in un linguaggio quasi naturale. La dichiarazione è definita come una frase costituita da un soggetto, un'azione, complemento ed eventualmente dei vincoli.

In riferimento alla prima categoria, è possibile trovare molti esempi e casi in cui il linguaggio *Nile* [13] viene utilizzato. Il linguaggio, il cui acronimo sta per *Network Intent LanguagE*, costituisce una rappresentazione intermedia con il linguaggio naturale, garantendo, tuttavia, una struttura abbastanza definita da poter essere facilmente tradotto in piattaforme diverse ed essere utile per il processo di apprendimento della rete. Infatti, un linguaggio di questo tipo si presta a numerosi utilizzi in quanto agisce da livello di astrazione rispetto ai meccanismi delle policy, risparmiando agli operatori lo sforzo di apprendere sempre nuovi linguaggi. Nile mira ad ottenere un'elevata leggibilità, rendendo di fatto ogni parola chiave auto esplicativa, e un'elevata scrivibilità, permettendo di estendere la sua sintassi in modo semplice.

```
define intent qos Intent:
    from endpoint('gateway')
    to endpoint('database')
    for client('B')
    add middlebox('firewall'), middlebox('ids')
    with latency('less','10s'),
          throughput('more or equal','100mbps')
    allow traffic('https')
    start hour('09:00')
    end hour('18:00')
```

Figura 3.3. Intento definito con Nile

Partendo dall'esempio mostrato in figura 3.3, descriviamo brevemente la sintassi: inizialmente è necessario definire l'intento specificandone il nome, per poi passare al corpo vero e proprio. Si parte, chiaramente, dalla definizione dei target, in questo caso due, definiti con le parole chiave "from" e "to" seguiti entrambi dal nome del nodo; proseguendo poi con gli altri vincoli, è necessario definire l'azione desiderata, in questo caso la parola "add" definisce l'intento vero e proprio, seguita dalla keyword "middlebox" che indica il servizio da aggiungere. Infine l'esempio determina dei vincoli sul tipo di traffico e sulla fascia oraria.

La costruzione dell'intento procede in maniera simile nel caso in cui il linguaggio utilizzato sia JSON. Nonostante si tratti di un linguaggio di serializzazione dati, si presta molto bene allo stile strutturato di questo tipo di intenti. Naturalmente, al contrario degli altri tipi di linguaggi nati allo scopo, JSON dipende fortemente dall'implementazione della piattaforma su cui vengono utilizzati; si definisce a priori una forma accettata che poi dovrà essere utilizzata in tutte le dichiarazioni, risultando così poco flessibile. Un esempio è fornito dall'articolo in [5] e riportato in figura 3.4. In questo caso le necessità dell'applicazione richiedevano un linguaggio che permettesse di definire le caratteristiche di una Service Function Chain, per ogni funzione da creare. Nell'esempio si può notare chiaramente l'utilizzo del linguaggio strutturato in cui alla keyword "service_blocks" corrisponde una serie di definizioni, che rappresentano ciascuna una funzione. La parola chiave "block" definisce il tipo di funzione, alla chiave "managed" corrisponde un valore

booleano che intende la volontà di collegare quella funzione alla sottorete di management, la chiave "order" permette di definire l'ordine di occorenza nella service chain e il valore corrispondente alla parola "symmetric" determina se il componente può essere attraversato dal traffico in entrambe le direzioni.

```
{ "name": "serviceA",
  "service_blocks": [
    {
        "block": "dpi",
        "managed": true,
        "order": 1,
        "symmetric": true
    },
    {
        "block": "firewall",
        "managed": false,
        "order": 0,
        "symmetric": true } ]
```

Figura 3.4. Esempio di intento JSON

Il linguaggio Merlin [27] merita sicuramente, insieme a quelli già trattati, un'analisi sintattica per la sua diffusione soprattutto nel mondo delle Software Defined Network. Si tratta di un linguaggio di definizione delle policy di alto livello che, allineandosi con la caratteristica principale di questa categoria, fornisce un'elevata espressività seppur trattando parametri più a basso livello. Basandosi sulla sintassi definita in figura 3.5, Merlin costruisce le policy su una collezione di *"statements"*, che definiscono le modalità di gestione di un sottoinsieme del traffico di rete, formati da una variabile, da un predicato logico e da un'espressione regolare. I predicati logici includono le espressioni atomiche, nella forma $f = v$, come è possibile vedere nell'esempio 3.1 in cui l'espressione confronta l'indirizzo ip sorgente con un valore dato, ma anche le espressioni booleane di *and*, *or* e negazione (*!*). Le espressioni regolari seguono il simbolo "→" e corrispondono ad una serie di funzioni o locazioni nella rete; il simbolo "**.**" definisce il match con una locazione arbitraria della funzione della rete, scelta dal compilatore. È possibile, inoltre, definire dei vincoli sulla banda utilizzata utilizzando le parole chiave "*max*" e "*min*"; potenzialmente, il vincolo può essere applicato anche sul traffico aggregato utilizzando il simbolo di addizione. Sulla base di queste premesse, risulta di facile comprensione l'esempio 3.1, in cui viene indicato un vincolo nel percorso del traffico *FTP* (porta 20) costringendolo verso una funzione di deep packet inspection e viene identificato il traffico di controllo di *FTP*, senza aggiungere altri vincoli. A fronte di queste identificazioni, viene aggiunto un vincolo sul traffico *HTTP* tra i due nodi (dpi e NAT[14]) e infine un vincolo sui due tipi di traffico: il traffico aggregato con porte destinazione 20 e 21 viene limitato a 50MB/s mentre al traffico *HTTP* viene garantita una banda minima di 100MB/s.

Per quanto riguarda invece la seconda categoria, citiamo Flowlog e NEMO, a cui dedicheremo una sezione a parte. Il primo racchiude un linguaggio dichiarativo contenente una serie di regole che manipolano il comportamento della rete e una serie di vincoli che regolano le tabelle di stato dello script stesso definito in Flowlog. L'utilizzo di tabelle

[14]Network Address Translation

Figura 3.5. Sintassi di Merlin [27]

$loc \in$ *Locations*		
$t \in$ *Transformation Functions*		
$pol ::= (s_1; \ldots; s_n)$	Policy	
$s ::= q\ p \rightarrow e$ at r	Statement	
$q ::=$ forall | exists	Quantifier	
$p ::= m \mid p_1$ and $p_2 \mid p_1$ or $p_2 \mid !\ p_1$	Predicate	
$m ::= f = n$	Match	
$e ::= . \mid c \mid e\,e \mid e \mid e \mid e^* \mid !\ e$	Path Expression	
$c ::= loc \mid t$	Path Character	
$r ::= \max(n)^?\ \min(n)^?$	Rate	

Listing 3.1. Esempio di policy Merlin [27]

```
[ x : (ip.src = 192.168.1.1 and
ip.dst = 192.168.1.2 and
tcp.dport = 20) -> .* dpi .* ;
y : (ip.src = 192.168.1.1 and
ip.dst = 192.168.1.2 and
tcp.dport = 21) -> .* ;
z : (ip.src = 192.168.1.1 and
ip.dst = 192.168.1.2 and
tcp.dport = 80) -> .* dpi *. nat .*
    ],
max(x + y,50MB/s) and min(z,100MB/s)
```

informative rende questo linguaggio estremamente complesso e articolato e l'analisi dettagliata sulla sintassi è fuori dagli obiettivi di questo lavoro, tuttavia è possibile, ricorrendo all'utilizzo di un semplice esempio, definirne a grandi linee il funzionamento:

Listing 3.2. Processo di verifica della sintassi

```
ON packet_in(p) WHERE p.dlTyp = 0x1001: INSERT (p.dlSrc,
    p.dlDst, p.locSw, p.locPt) INTO ucST;
```

Quello appena presentato è un esempio di ciò che viene chiamato "ruleset", contiene, cioè, una serie di regole da applicare su una determinata tabella che contiene le informazioni necessarie da elaborare. Queste si dividono in **INCOMING** e **OUTCOMING**, in cui si processa un evento per volta, selezionandolo dalla tabella di ingresso e inserendo il risultato in quella in uscita. Ogni regola inizia con il comando **ON** e viene scatenata quando quel tipo di condizione si verifica: nell'esempio, **packet_in**, denota la tabella contenente tutti i pacchetti ricevuti e "p" il singolo pacchetto. Le azioni possibili da eseguire sono **INSERT** o **DELETE**, mentre per azioni basilari è possibile anche l'opzione **DO** (es. forward). Se il verbo selezionato è **DO** allora verrà popolata direttamente, altrimenti verrà utilizzata una tabella ausiliaria di appoggio prima di raccogliere tutte le informazioni in quella di uscita.

È bene citare, in conclusione, altri linguaggi utilizzati nel contesto degli intenti come Frenetic[15], Maple[16], Cisco ACI language[17], Apstra AOS language[18] e Gherkin, strutturato similmente a Nile.

3.4.2 NEtwork MOdeling

Prima di descrivere le caratteristiche del linguaggio NEMO è bene fare delle distinzioni nell'utilizzo del termine. Il termine NEMO è utilizzato in letteratura con accezioni

[15] http://www.frenetic-lang.org/

[16] https://www.maplesoft.com/

[17] lhttps://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html

[18] https://www.juniper.net/us/en.html

diverse e in contesti diversi riguardo, tuttavia, lo stesso argomento principale, quello dell'Intent Based Networking. Il termine NEMO viene introdotto per la prima volta da Huawei, riferendosi ad un linguaggio dichiarativo supportato da una NBI installata su un controllore SDN. Successivamente, il termine viene utilizzato per riferirsi al progetto che Huawei stessa ha lanciato, mirando allo sviluppo di una NBI completa basata sull'omonimo linguaggio. Successivamente, l'istituto IETF, ritenendo il momento opportuno per un tentativo di standardizzare la tecnologia, rilascia una bozza [28] in cui spiega i dettagli di un linguaggio intent-based "semplice", descrivendone le peculiarità e definito IB-NEMO (Intent-Based NEMO). Nello stesso anno in cui viene rilasciata la bozza, la comunità open source OpenDaylight rilascia la prima versione del codice che includeva una piattaforma di gestione di intenti, basata sul linguaggio NEMO e chiamata, appunto, OpenDaylight NEMO [19].

Partendo da questi chiarimenti, procediamo con la spiegazione del linguaggio. Il linguaggio che Huawei mirava a creare doveva basarsi sulla visione *"application-centric"* della rete, modello secondo il quale l'applicazione non vede i dettagli che non sono strettamente necessari. Le necessità erano quelle di creare un linguaggio comune di definizione degli intenti che potesse condurre verso un'architettura di NBI applicabile su tutti i sistemi. In questo modo, diveniva concreta la possibilità di comunicare con il sistema di gestione della rete, in modo da amministrarla, e di non far esplodere la complessità che, altrimenti, sarebbe gravata interamente sull'intent engine. Inoltre, era fondamentale che il linguaggio fosse intuitivo, in modo che qualsiasi utente potesse utilizzarlo sfruttando le sue caratteristiche di auto esplicazione. Ogni parola chiave doveva essere chiara e il suo comportamento intrinseco nella parola stessa e nel complesso l'intento doveva risultare facilmente comprensibile. Questo richiedeva certamente una sintassi snella e una scelta mirata nelle keyword da implementare. Quando Huawei definisce il proprio linguaggio, intuisce la soluzione della complessità sintattica in un semplice concetto definito "80/20 rule". Questa regola parte dall'assunzione secondo la quale l'80% delle applicazioni, utilizzano solamente il 20% dei comandi di cui una API è provvista. Lo scopo, quindi, di un protocollo, che poggia le proprie basi su questa teoria, è quello di individuare quella piccola parte di comandi utilizzati dall'applicazione per interagire con il sistema e implementarli. Il concetto è, quindi, quello di ottenere il massimo risultato evitando di creare un linguaggio con una sintassi importante e di difficile assimilazione. Per ovviare alla mancanza di supporto per il restante 20% delle applicazioni, NEMO è progettato in modo da semplificare il più possibile l'estensione del linguaggio. Chiaramente, la sfida e la complessità sta nel definire quali comandi andranno a comporre il linguaggio. Huawei risolve questa problematica utilizzando un approccio inverso: partendo dai casi d'uso ritenuti più importanti e rappresentativi, viene fornito un possibile prototipo di codice che dovrebbe permettere di ottenere quel determinato scenario. A quel punto, viene sviluppata una possibile soluzione di test, utilizzando il codice ottenuto precedentemente e, se il risultato è congruente con lo stato atteso, si inizia la fase di raffinazione provando a snellire al massimo i comandi necessari. A questo punto si passa al caso d'uso successivo; la catena di casi d'uso più importanti, generalmente, viene discussa con i fornitori più importanti, che dovrebbero conoscere le statistiche degli scenari più comuni.

Come raffigurato nello schema 3.6, la sintassi del linguaggio NEMO può essere modellata con la tripletta Oggetto, Operazione, Risultato.

Le possibilità di espressione di un intento prevedono due strade, la prima in cui si definiscono un oggetto e un'operazione, la seconda costituita da un oggetto e un risultato.

[19] https://test-odl-docs.readthedocs.io/en/latest/user-guide/nemo-user-guide.html

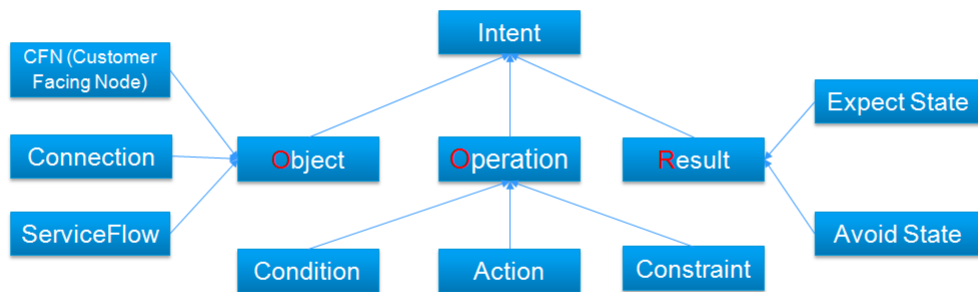


Figura 3.6. Modellazione sintassi NEMO (fonte: [1])

Come si evince dallo schema, ogni entità descritta precedentemente non costituisce un elemento singolo (ad esempio l'operazione non è intesa in senso letterale), ma ciascuna è caratterizzata da svariati elementi.

Nel dettaglio:

- Object:
  - Customer Facing Node: rappresenta il nodo con cui l'utente vuole interagire. Ad esempio "`CREATE a Network for HR;`" definisce l'intenzione dell'utente di creare una sottorete separata;
  - Connection: rappresenta un link fra due nodi, con tutte le caratteristiche che ne conseguono. Un esempio è dato dalla seguente espressione "`ADJUST the bandwidth to 10G;`";
  - ServiceFlow: identifica un traffico dati di un particolare tipo all'interno della connessione. Ad esempio "`BLOCK the http flow;`".
- Operation:
  - Condition: rappresenta la condizione al cui verificarsi viene scatenata una corrispondente azione. Ad esempio "`Condition time<9 && time>18;`";
  - Action: esegue un'azione al verificarsi di una condizione. Per esempio "`Action qos-bandwidth:1024M;`";
  - Constraint: aggiunge un vincolo all'azione specificata. Ad esempio si può garantire una determinata banda solo se l'utilizzo è inferiore ad una determinata soglia "`Constraint bandwidth-usage<60%;`".
- Result:
  - ExpectState: definisce lo stato che ci si aspetta la rete possa raggiungere.
  - AvoidState: definisce lo stato che la rete non deve raggiungere. Esempio "`AVOID ftp-traffic-rate>20%`".

Il listato 3.3 rappresenta una sintassi parziale del linguaggio NEMO, in formato BNF. Come si può vedere, sono riportate le quattro tipologie di comandi accettati da NEMO e ne sono stati approfonditi quelli più importanti, utili per realizzare uno scenario semplice. Inoltre l'elenco delle parole chiave fornisce nel dettaglio i token accettati nella sintassi degli intenti e i tipi di dato che caratterizzano i parametri, ciascuno definito da una espressione regolare. In generale, si può dire che la maggior parte dei comandi realizzabili con NEMO iniziano con uno dei verbi fondamentali quali `CREATE`, `IMPORT`, `UPDATE`

e DELETE, seguito dal complemento oggetto, che indica il target dell'azione, e da svariate informazioni peculiari per ogni tipologia di oggetto trattata. Ogni parola è separata da uno spazio o da un segno di punteggiatura, se si tratta di un elenco di informazioni riferite alla stessa tipologia, e ogni comando termina con il punto e virgola: il caso più chiaro è quello delle proprietà, definite come coppie chiave-valore intervallata dal segno di due punti e separate dalla virgola. Inoltre, il linguaggio è *case sensitive* per cui è fondamentale, qualora fosse richiesto dalla sintassi, dichiarare nomi o altri parametri in minuscolo differenziandole dalle parole chiave che, come riportato nel listato 3.3, prevedono la prima lettera maiuscola.

In particolare, nella categoria dei comandi di accesso alle risorse, viene analizzata la sequenza di creazione di una connessione: in riferimento a questo comando, è vincolante definire il nome, in formato <ID>, oltre al tipo, anch'esso in formato <ID>, che deve trovare una corrispondenza tra quelli definiti nella piattaforma, e all'informazione sui nodi di terminazione, identificati da una stringa univoca. Analogamente al tipo, anche le proprietà devono trovare riscontro in quelle definite in NEMO.

Listing 3.3. Nuova sintassi parziale di NEMO

```
<NEMO_cmd> ::= <model_definition_cmd> | <resource_access_cmd> |
               <behavior_cmd> | <transaction cmd>

<model_definition_cmd> ::= <node_definition> | <connection_definition>
    |
                           <action_definition> | <model_description>

<resource_access_cmd> ::= <node_create> | <node_import> |<node_update>|
                          <node_del> | <connection_create> |
                          <connection_update> | <connection_del> |
                          <serviceflow_create> | <serviceflow_update> |
                          <serviceflow_del>

<behavior_cmd> ::= <query_cmd> | <operation_create> |
                   <operation_update> | <operation_del> |
                   <notification_create> | <notification_update> |
                   <notification_del>

<transaction_cmd> ::= <transaction_begin> | <transaction _end>

<key_word> ::= Range | Integer | String | <UNUMBER> | <ID> | <TEMPID> |
               <ETHPREF> | <IPV4PREF> | <DATEVAL> | <TIMEVAL> |
               <FULLTIME> | <ETHADDR> | <IPV4ADDR> | <URI> | <UBYTE> |
               <HEXDIGIT> | <YEAR> | <SMONDAY> | <LMONDAY> | <HOUR> |
               <MINUTE> | <SECOND> | <DIGIT> | NodeModel |
               ConnectionModel | ServiceFlowModel | ActionModel |
               Description | Porperty | Node | Connection|
               ServiceFlow | ConnectionPoint | EndNodes | Type |
               Contain | Match | List | Flow | End |
               nodes | connections | flows | operations | at | http |
               https | file | Range | Query | From | Notification |
               Listener | Operation | Target | Priority | Condition |
```

```
              Action | Transaction | Commit | CREATE | IMPORT |
              UPDATE | DELETE



<connection_create> ::= CREATE Connection <connection_id>
                        Type <connection_type>
                        EndNodes <node_id>, <node_id>
                        [Property {<property_name>: <value>}]
                        [ACL {<src_ip_space> to <dst_ip_space>}];

<connection_del> ::= DELETE Connection <connection_id>;

<node_create> ::= CREATE Node <node_id> Type <node_type>
                  [Contain {<node_id>}]
                  [Property {<property_name>: <value>}];
```

3.4.3 Analisi di OpenDaylight NEMO

OpenDaylight rappresenta una comunità open source affermata nel mondo SDN. Tuttavia, negli ultimi anni, si è impegnata molto sul tema dei sistemi intent-based dando alla luce tre progetti: OpenDaylight Group Based Policy (GBP)[20], OpenDaylight Network Intent Composition (ODL-NIC)[21], Open Daylight Network Modeling: NEMO. Il primo si concentra principalmente sulle politiche, ottenendo un grado di automazione grazie al loro utilizzo simile agli intenti, il secondo, invece, mira a creare un'interfaccia NBI completa, venendo meno al concetto caratterizzante della regola 80/20 di NEMO.

Come anticipato brevemente nell'introduzione di questa sezione, ODL NEMO costituisce una release open source di IB-NEMO e ricalca perfettamente le scelte sintattiche delineate precedentemente. L'entità Object conta di tre possibilità Node, Connection, e Flow; l'entità Operation permette di definire condizioni e azioni, entrambi opzionali e l'entità Result resta invariata. Ad ogni oggetto corrisponde un tipo, che definisce la categoria di appartenenza dell'entità dichiarata e che possiede svariate proprietà che lo caratterizzano. Tutte questi parametri sono configurati dall'utente e definiti a priori.

Quello che segue è un possibile intento definito sul framework che permette di creare un router virtuale e di definire il suo indirizzo IP pubblico:

Listing 3.4. Esempio di intento

```
CREATE Node r1 Type router Property public-ip-address: 192.168.58.101;
```

La caratteristica principale del framework sta nella sovrapposizione tra rete virtuale e rete fisica. Infatti, per incrementare al massimo la flessibilità di NEMO, OpenDaylight ha

[20] https://docs.opendaylight.org/en/stable-fluorine/user-guide/group-based-policy-user-guide.html

[21] https://docs.opendaylight.org/en/stable-fluorine/user-guide/network-intent-composition-(nic)-user-guide.html

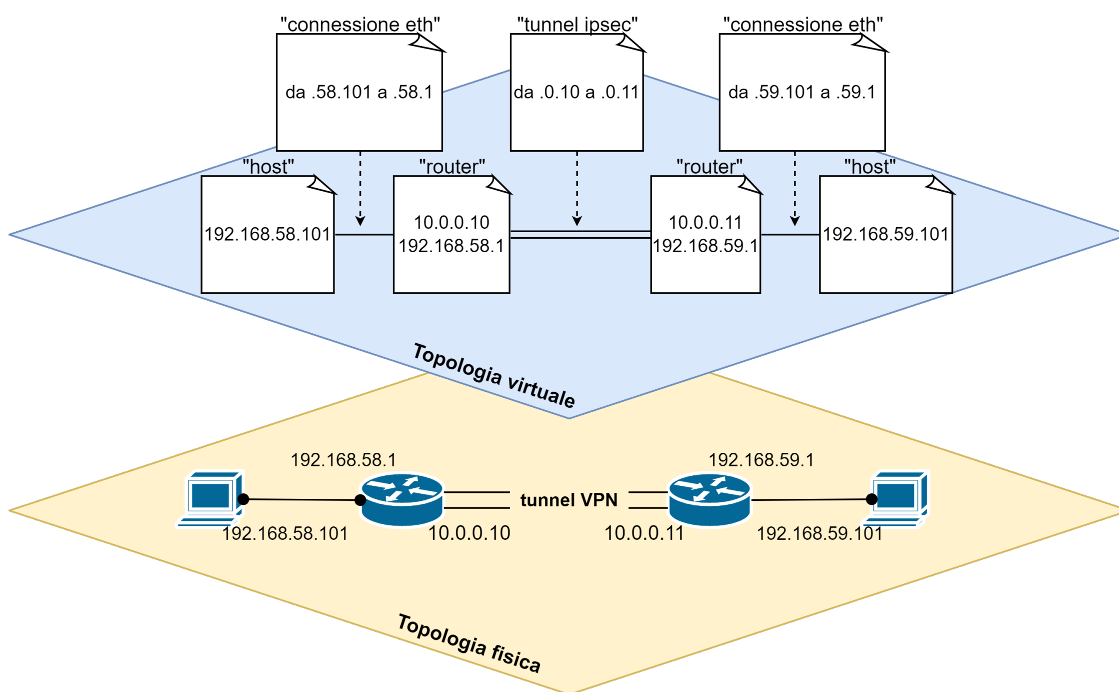


Figura 3.7. Sovrapposizione dei livelli fisico e virtuale

organizzato la rete su due livelli: il primo, quello virtuale, viene creato immediatamente a fronte dell'inserimento di una serie di intenti da parte dell'utente, con la possibilità di definire nodi, connessioni e flussi virtuali, il secondo, invece, rispecchia la topologia fisica sottostante. Questo modello è rappresentato graficamente in figura 3.7 .In seguito alla creazione di una topologia virtuale, NEMO esegue un mapping per realizzare la configurazione delineata sugli apparati fisici. Per semplificare la gestione dei dati, NEMO modella i suoi dati sulla figura del ruolo dell'utente. La struttura interna permette di raccogliere i dati della rete e associarli al singolo utente, permettendo di fatto più strutture virtuali sulla stessa topologia fisica. Inoltre, basando l'accesso sul ruolo dell'utente, è possibile definire alcune informazioni legate al ruolo stesso: ad esempio, se gli utenti sono utilizzatori di una rete locale, le informazioni che potrebbero aver in comune riguarderebbero lo spazio di indirizzamento, le connessioni e le policy.

Capitolo 4

Network Functions Virtualization

4.1 Introduzione ad NFV

In questo capitolo verrà presentato il paradigma NFV insieme ad alcuni concetti strettamente correlati; infine sarà analizzata la specifica funzione di (sicurezza) rete utilizzata in questo lavoro.

È tuttavia necessario, prima di iniziare con la nostra analisi, fornire una brevissima panoramica sul concetto di virtualizzazione e come questo si sia diffuso in molti settori, specialmente nel mondo IT[1].

Scatenata da una serie di eventi, la nascita della virtualizzazione vede il suo principio nel momento in cui le grandi aziende del mondo tech hanno mostrato la necessità di migrare i propri server locali in strutture dedicate, al fine di semplificare la loro gestione sia dal punto di vista di sicurezza, sia dal punto di vista economico. È noto, infatti, che la protezione dei server in termini di accessi all'infrastruttura sia un tema molto sensibile. Trasferendo tutte le risorse in strutture dedicate, chiamate appunto datacenter, si semplifica significativamente il problema della sicurezza fisica, controllando, di fatto, gli accessi alla struttura, e in più si semplifica la gestione manuale e si migliora nettamente lo sforzo economico che deriva dal raffreddamento dell'ambiente, che così si concentra su un'unica struttura. Tuttavia le migliorie erano ancora limitate dal punto di vista dello sfruttamento delle risorse computazionali: vigeva la regola, all'interno dei datacenter, denominata "one application per server" che rendeva, di fatto, la maggior parte delle risorse dei server completamente inutilizzate. Questo era necessario per assicurare una gestione più agile dell'intero sistema; in questo modo si evitavano conflitti con applicazioni che richiedevano dipendenze o versioni differenti e, inoltre, era più facile fare previsioni sui comportamenti di ciascun server. Se un server consumava troppa banda oppure registrava un malfunzionamento, era facile stabilire quali applicazioni avrebbe danneggiato. Naturalmente, si trattava di una gestione poco sostenibile dal punto di vista aziendale, che si sarebbe ritrovata di lì a poco ad avere enormi strutture piene di server largamente sottoutilizzati. È così che nasce il concetto di virtualizzazione, che avrebbe permesso di sfruttare tutte le risorse fornite, senza venir meno alle esigenze garantite dalla regola "one application per server". Essenzialmente, rappresenta un modo per astrarre le capacità fornite dall'hardware ed ottenere risorse virtuali mantenendo una corrispondenza con quelle fisiche. Questo concetto, ben presto, si è diffuso in molti settori trovando molteplici applicazioni. In particolare, un suo utilizzo molto diffuso permette la realizzazione

[1]Information Technology

di sistemi completi in cui vengono virtualizzate tutte le risorse necessarie per eseguire un sistema operativo: questo è reso possibile dal lavoro svolto dall'*hypervisor*, incaricato di gestire completamente il ciclo di vita delle macchine virtuali. In particolare, gestisce l'allocazione di risorse virtuali, realizza l'isolamento di ciascuna macchina virtuale e si assicura che l'ambiente di esecuzione sia una copia di quello fisico

4.1.1 Definizione e principi

L'European Telecommunications Standards Institute (ETSI) introduce per la prima volta la tecnologia NFV nel 2012 [29] dichiarando quelli che saranno gli scopi di questo paradigma. Secondo ETSI, NFV mira a cambiare radicalmente le tecniche architetturali delle reti, trasportando la tecnologia della virtualizzazione nelle reti e consolidando le funzioni all'interno di server standard. Tale approccio, espresso molto chiaramente dalla figura 4.1, permette il passaggio da un utilizzo frammentato su hardware dedicati delle funzioni di rete verso un utilizzo e uno sviluppo più agile, eliminando la necessità di installare fisicamente gli apparati su una rete. La sfida consiste nel replicare tutte le funzioni fondamentali su server standard che sfruttano capacità di storage standard e gestiti a loro volta da switch standard

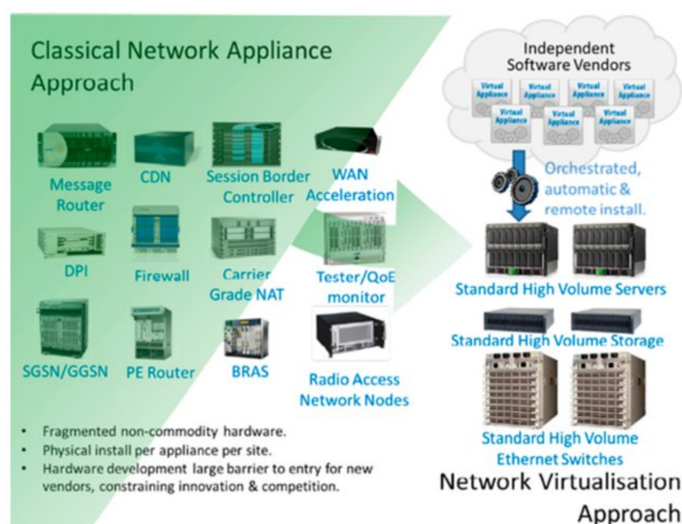


Figura 4.1. ETSI NFV (fonte: [29])

Sulla base del contesto fornito precedentemente, si può definire la Network Functions Virtualization come la tendenza a realizzare qualsiasi funzione di rete (es. router, firewall etc.) in un contesto in cui si fa ricorso alla virtualizzazione: questo significa che la funzione, sfruttando le risorse fornite da un hardware standard (e non ad hoc come ad esempio un router fisico) e sfruttando le caratteristiche della "computing virtualization" introdotte poc'anzi, viene realizzata via software così da ottenere un efficiente utilizzo delle risorse. I pilastri principali di questa tecnologia sono quindi:

- Hardware standard: un comune server piuttosto che hardware con memorie dedicate e bus dati dedicati;
- Software: le funzioni di reti, che per anni sono state ospitate da un box dedicato, diventano parti di codice implementate sui server;

- Virtualizzazione delle risorse: per implementare più funzioni sullo stesso hardware si ricorre alla virtualizzazione, con tutti i vantaggi che ne conseguono;
- Standard API: l'interfaccia e l'infrastruttura diventa comune per tutte le soluzioni, grazie allo standard fornito da ETSI (sezione 4.2)

Attualmente il concetto ha subito un'evoluzione ulteriore, permettendo di fatto due possibili scenari: il primo, in cui le funzioni di rete vengono fornite come moduli completamente realizzati in software, e un secondo scenario prevede l'installazione di funzioni software in hardware generici ma in box dedicati. In questo modo, si ottiene un apparato fisico (es. router) con hardware generico che esegue del codice.

4.1.2 Relazione con i concetti di SDN e Cloud Computing

La definizione fornita da ETSI su NFV si articola su campi differenti, andando ad "invadere" altri settori ed in particolare quelli di Cloud Computing e Software Defined Network. Definiamo quindi i punti in comune e quali sono invece le differenze (sostanziali) tra le tecnologie.

Rispetto alla prima tecnologia, il punto di partenza migliore per questa analisi è la definizione di Cloud Computing. Secondo IBM, il Cloud Computing riguarda l'accesso, tramite una rete, a risorse computazionali remote, server, archivi di dati, capacità di rete e tutte le altre tecnologie ospitate in un data center ed esposte come servizi [30]. Analogamente, è possibile inquadrare NFV da un'angolazione simile, definendola come una tecnologia che sposta le funzioni di rete in una struttura centralizzata, i cui servizi vengono esposti attraverso la rete internet. Tuttavia è chiaro che NFV si concentra in modo specifico sulle funzioni di rete, che richiedono determinati requisiti e, pertanto, sono ottimizzate allo scopo al contrario della tecnologia cloud-based che fornisce servizi generici. La differenza sostanziale risiede tutta nel tipo di servizio esposto: i servizi di rete sono, infatti, orientati allo sfruttamento massiccio delle risorse di input e output, di contro le tecnologie cloud richiedono uno sforzo maggiore sulla CPU per gestire numerosi task contemporaneamente.

La tecnologia SDN invece, già brevemente citata nei capitoli precedenti, costituisce una tecnologia di rete che prevede il disaccoppiamento della logica di controllo dal data plane; questo si riflette sulla programmabilità del control plane che permette ad esempio, in modo semplice e veloce, di dirigere il traffico secondo nuove regole. In questo caso, infatti, la relazione tra NFV e SDN riguarda più un certo grado di interoperabilità rispetto alla somiglianza tra i concetti. Di fatto, la prima si occupa di fornire servizi virtualizzati, mentre la seconda riguarda nello specifico i percorsi di rete, guidati dal controllore centralizzato. Tuttavia, proprio quest'ultima conduce, sovente, ad un utilizzo sinergico delle due tecnologie, con NFV che sfrutta le capacità di gestione del traffico di SDN inoltrando il traffico attraverso più funzioni. Entrambe le tecnologie rivoluzionano l'ambito in cui sono introdotte, riflettendo i cambiamenti sulle scelte architetturali. Entrambe richiedono un cambiamento radicale nello sviluppo delle applicazioni, che devono essere riscritte nel caso delle SDN per essere eseguite da un control plane diverso o, nel caso delle NFV devono essere implementate in toto. Entrambe traggono vantaggio, seppure in maniera e in misura differente, dalla velocità dell'hardware utilizzato: SDN sfrutta gli hardware accelerati dal punto di vista del data plane, riuscendo ad eseguire l'inoltro dei pacchetti con una frequenza più alta e quindi a smaltire più traffico, NFV, invece, può ottenere vantaggi da questo tipo di hardware, ottenendo migliori prestazioni dalle sue funzioni. La differenza, seppur sottile, è sostanziale; SDN si basa sull'utilizzo di hardware ottimizzati,

NFV non si occupa di questo aspetto, ma implementandone i moduli su queste tipologie di hardware, si otterrebbero prestazioni migliori. Per questo la tecnologia SDN risulta più efficiente ma risulta sicuramente meno flessibile dal punto di vista delle applicazioni supportate, che necessitano una riconversione e un adattamento alla nuova architettura.

Infine è bene sottolineare ancora una volta che si tratta di due concetti distinti che possono indistintamente operare in ambiti separati o coesistere nello stesso scenario: NFV si occupa di trasportare le funzioni di rete o di sicurezza in un ambiente virtualizzato, mentre SDN si occupa della gestione del traffico che può essere programmata. Questo aspetto permette di sfruttare la tecnologia SDN per determinare la sequenzialità delle funzioni attraverso cui dirigere il traffico, realizzando quelle che vengono chiamate *Service Function Chain* (SFC).

4.1.3 Vantaggi

NFV permette al mondo del networking di effettuare un grosso balzo in avanti dal punto di vista innovativo, fornendo una grande flessibilità che permette agli operatori di far fronte alle nuove esigenze del mercato. I vantaggi, chiaramente, sono molteplici e non si identificano solamente con quello appena descritto. Proviamo, di seguito, ad analizzare gli aspetti vantaggiosi di NFV:

- Economia: grazie all'economia di scala, che garantisce in previsione la diminuzione del costo di un'apparecchiatura proporzionalmente all'aumento della sua produzione, NFV permette di registrare la diminuzione delle spese dovuta agli apparati fisici, ma soprattutto la diminuzione delle spese di gestione in materia di consumo elettrico. Partendo dalla considerazione secondo cui non esiste una relazione di linearità tra l'utilizzo della CPU e il consumo elettrico (è quindi conveniente avere un server che lavori al massimo delle sue potenzialità rispetto a due sottoutilizzati) e considerando che le spese di gestione e raffreddamento di un datacenter sono nettamente inferiori ai costi necessari per più server room dislocate geograficamente, si può facilmente dedurre il conseguente abbattimento delle spese. Questi vantaggi sono in parte ereditati dal concetto di virtualizzazione;

- Virtualizzazione: rappresenta essa stessa un vantaggio, permettendo un uso più sofisticato delle risorse senza preoccuparsi di dove siano fisicamente localizzate e quanto si possano sfruttare;

- Orchestrazione: la tecnologia NFV, permette, implicitamente, la gestione di centinaia di dispositivi in modo del tutto trasparente all'utente;

- Programmabilità e testing: è possibile in modo estremamente rapido cambiare il comportamento di una fuzione semplicemente scrivendo nuove parti di codice e implementarle su una nuova piattaforma virtuale, differentemente a quanto accadeva agli apparati fisici che necessitavano una sostituzione fisica dell'hardware stesso con un firmware aggiornato. Questo comportamento è fondamentale per stimolare la velocità di crescita del mercato che, nel caso della tecnologia NFV, deve concentrarsi unicamente sul software da implementare, portando soluzioni sempre migliori. Inoltre è possibile eseguire produzione e test direttamente sulla stessa piattaforma riducendo notevolmente i tempi di sviluppo;

- Scalabilità dinamica: scompare in questo modo l'attenzione alla gestione del workload del singolo apparato che può contare su un quantitativo di risorse importante.

Il comportamento dinamico è garantito dalla virtualizzazione, che è in grado gestire le risorse a proprio piacimento. Trattandosi di un hardware standard e utilizzando immagini software delle funzioni di rete, a fronte di un carico di lavoro massiccio, è possibile creare più repliche della stessa funzione che gestiranno così il carico di lavoro oppure si può decidere di assicurare un maggior numero di risorse alla funzione stressata;

- Visibilità: si ottiene una visibilità semplificata sulle risorse utilizzate e quelle ancora disponibili, facilitando la fase di monitoraggio;
- Performance: nel caso delle performance il discorso è leggermente più articolato. Chiaramente si perdono i vantaggi dell'hardware dedicato che permettevano un forwarding rapido, ma si ottengono performance migliori dal punto di vista dell'utilizzo delle risorse. A riguardo si è fatta strada una tecnologia basata su driver accelerati, chiamata DPDK[2], che permette di ottimizzare il data plane in modo da accelerare il processing dei pacchetti. In questo modo, si perde in flessibilità in quanto le modifiche al data plane rendono, poi, i server dedicati alle sole funzioni di rete;
- Multi-tenancy: il sistema è in grado di gestire più tenant contemporaneamente, dando la possibilità agli operatori di fornire servizi mirati al singolo utente, tutti implementati sullo stesso hardware ottenendo comunque la sicurezza e l'isolamento garantiti dalla virtualizzazione;
- Apertura: sviluppa un'apertura sia dal punto di vista della scelta dei servizi che sono richiesti, sia dal punto di vista della fornitura, incoraggiando piccole imprese, ricercatori e altri operatori nel mercato a fornire il proprio codice o a sviluppare nuove applicazioni riducendo al minimo il rischio di esposizione economica.

4.2 Standard ETSI NFV

In questa sezione andremo ad analizzare più nel dettaglio lo standard ETSI citato nella sezione precedente, descrivendo l'architettura proposta e la struttura di una funzione.

Gli studi sono stati compiuti dall'Industry Specification Group (ISG) dell'istituto ETSI che ha definito, a riguardo, altri gruppi di esperti per lavorare su ciascun componente. In particolare:

- Architettura per l'infrastruttura di virtualizzazione
- Gestione e orchestrazione
- Architettura software
- Affidabilità e disponibilità del servizio, resilienza e resistenza ai guasti
- Dimostrazioni pubbliche e Proof of Concept
- Performance
- Sicurezza

[2] https://www.dpdk.org/

4.2.1 Piattaforma NFV

Prima di proseguire con la descrizione dell'architettura, seguendo l'approccio dei paper di ETSI, è necessario fornire la terminologia adatta per comprendere tutti gli acronimi:

- *Network Function (NF)*: definisce un blocco funzionale con interfacce ben definite e un comportamento noto a priori;
- *Virtualized Network Function (VNF)*: implementazione software di una NF che può essere realizzata in un'infrastruttura virtualizzata. Il concetto verrà ripreso nel dettaglio più avanti;
- *NFV Infrastructure (NFVI)*: definisce l'insieme dell'hardware e software necessari a implementare, gestire ed eseguire le VNF. Questo comprende anche la gestione delle risorse computazionali, di rete e anche di storage dei dati. Questo componente sarà analizzato più nel dettaglio nella sottosezione 4.2.3;
- *NFV Management and Orchestration*: componente fondamentale che automatizza l'implementazione e la gestione delle VNFs.

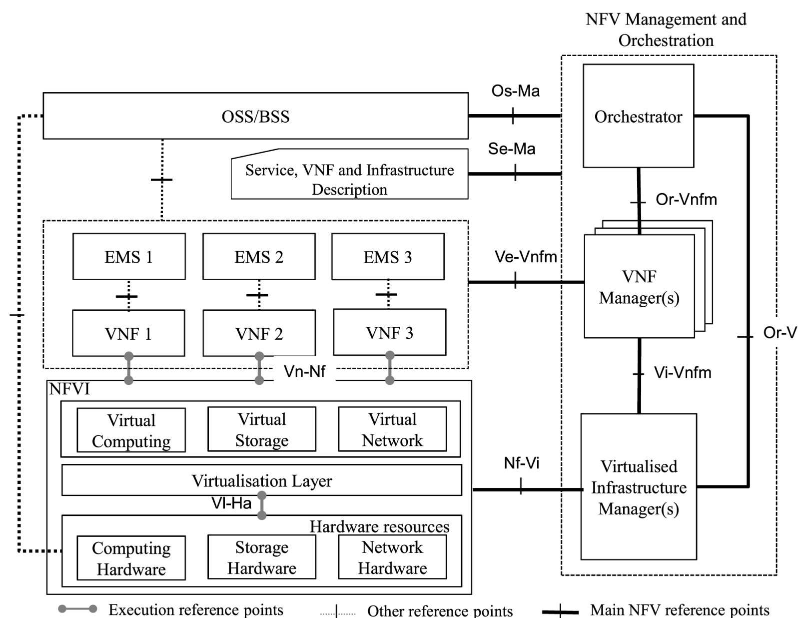


Figura 4.2. ETSI Framework (fonte: [31])

Come illustrato dalla figura 4.2, l'architettura proposta da ETSI consta ad alto livello di tre moduli separati che garantiscono il ciclo di vita completo delle funzioni: il modulo contenente le VNFs, il modulo contentente l'infrastruttura vera e propria (NFVI) e il modulo di orchestrazione e gestione (NFV MANO). Partendo dal grado più alto di astrazione e seguendo l'ordine definito dalla figura, procediamo con la descrizione dei moduli.

Il modulo *OSS/BSS*, di derivazione dell'ambito delle telecomunicazioni, fornisce supporto al sistema. Il primo sistema, l'Operations Support System (OSS), interviene nella configurazione della rete, nella gestione dei casi di malfunzionamento e nella fornitura del servizio. Il secondo sistema, invece, chiamato Business Support System (BSS) è più orientato a supporto dell'azienda, in particolare gestisce le operazioni commerciali nei

confronti dei clienti. Tuttavia la separazione dei due concetti non fornisce una visione corretta del modulo che viene inteso come congiunto e svolge azioni di monitoraggio e gestione.

Direttamente collegato al modulo precedente, il modulo *Element Management System (EMS)* si occupa di gestire una funzione di rete secondo le indicazioni del modulo OSS/BSS. Ad esempio, il modulo EMS è responsabile per il soddisfacimento di un determinato grado di *Quality of Service* (QoS).

Il modulo *Service, VNF and Infrastructure Description* definisce un aspetto fondamentale della tecnologia NFV. Fornisce, infatti, il data model che permette di descrivere le caratteristiche del servizio di rete da implementare e quindi della VNF. Tramite tale descrizione è possibile, poi, eseguire il deploy della funzione.

Il *Virtualization Layer* è incaricato di astrarre le risorse e realizza la separazione tra la funzione software e l'hardware fisico sottostante. Tale modulo è incaricato, inoltre, di partizionare logicamente le risorse utilizzabili e, fornendole in modo adeguato alle VNF, permette la loro esecuzione. Il comportamento di questo comportamento ricalca perfettamente quello delineato dal concetto di hypervisor nell'architettura di virtualizzazione di sistemi operativi (virtual machines).

Ai moduli non passati in rassegna in questa descrizione, verranno dedicate le prossime sezioni al fine di illustrare al meglio quelli che sono considerati i componenti centrali dell'architettura considerata.

4.2.2 Virtual Network Function

Procediamo ora con l'analisi del modello vero e proprio della VNF descritto da ETSI in [32]. In continuità con lo stile seguito in precedenza per cercare di fornire una visione della tecnologia nel modo più semplice e comprensibile, procederemo alla spiegazione partendo da una rappresentazione grafica funzionale dei componenti fondamentali (figura 4.3).

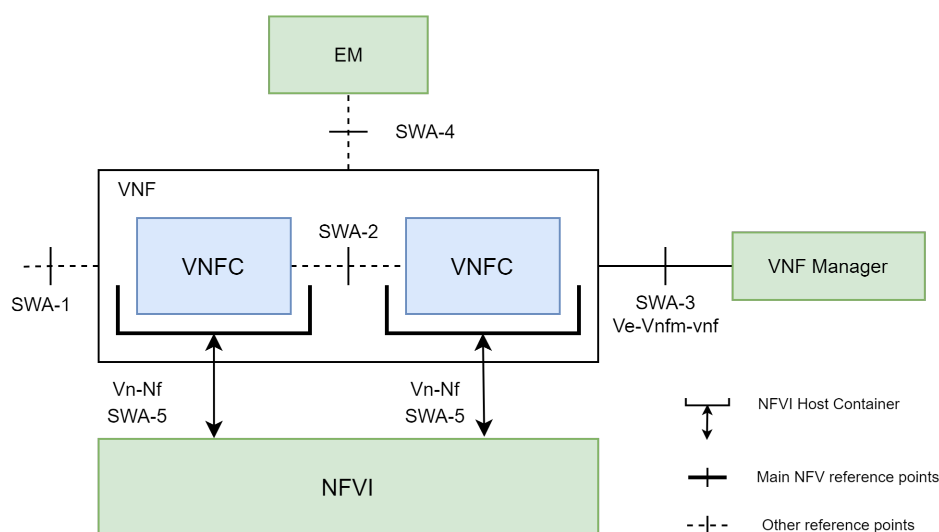


Figura 4.3. Rappresentazione di una VNF

Come definito in precedenza, con il termine VNF si definisce un blocco funzionale dell'architettura ETSI che rappresenta una realizzazione software di una funzione di rete.

Essa necessita di un'infrastruttura per essere eseguita (NFVI) e affida la sua gestione ad un terzo componente, il modulo NFV MANO (sez. 4.3).

Ciascuna VNF, è costituita da uno o più moduli software denominati *VNF Component (VNFC)* definiti dal fornitore del software. È possibile, infatti, che lo sviluppatore strutturi il proprio codice in componenti separati; tale suddivisione si riflette anche sulle immagini software che verranno prodotte richiedendo, così, l'esecuzione di tutti i componenti al fine di un corretto funzionamento della funzione. Per eseguire una funzione modulare, l'infrastruttura mette a disposizione più container, ciascuno responsabile dell'esecuzione di un modulo. Si tratta, infatti, di contenitori forniti dal virtualization layer in grado di ospitare una funzione ed eseguirla. L'esposizione di tali contenitori avviene grazie all'interfaccia *Vn-Nf*. A tal proposito la figura 4.3 chiarisce molto bene l'utilizzo che ETSI propone di queste interfacce. Ogni "collegamento" tra moduli funzionali è caratterizzato, infatti, dalla presenza di *Reference point*, secondo il lessico dell'istituto; in totale possiamo contarne cinque:

- *SWA-1*: interfaccia che permette la comunicazione con altre funzioni di rete all'interno dello stesso servizio di rete o in servizi differenti. Per completezza, si può definire un servizio di rete come un insieme di VNF che mirano a creare una funzione, appunto un servizio, molto più complessa;

- *SWA-2*: si riferisce alle interfacce interne al modulo VNF, predisposte, cioè, alla comunicazione tra differenti VNFC. Realizzate e definite dal fornitore del software che implementa la funzione, queste interfacce sono trasparenti dal punto di vista dell'utente, pur essendo specifiche per ogni vendor. I meccanismi di comunicazione tra i moduli sono definiti dall'infrastruttura sottostante, tuttavia, sovente si ricorre all'utilizzo di tecniche diverse, in particolare per il supporto ai vincoli di latenza. Tali tecniche includono, ad esempio, comunicazioni basate su canale o su memoria condivisa; da quest'ultima ne conseguono potenziali rischi di sicurezza, venendo meno il concetto di isolamento;

- *SWA-3*: corrispondente al reference point *Ve-Vnfm-vnf*, l'interfaccia SWA-3 permette la comunicazione tra la VNF e il modulo di gestione, contenuto nel componente NFV MANO;

- *SWA-4*: è utilizzata dal modulo Element Management (EM) per comunicare con una VNF e per gestire la stessa durante l'esecuzione;

- *SWA-5*: corrisponde al reference point *Vn-Nf* e costituisce una rappresentazione logica di più interfacce. In generale, sono incaricate della comunicazione tra la VNF e l'infrastruttura NFVI e coprono le differenti tipologie di risorse che l'infrastruttura può virtualizzare. Queste tipologie comprendono risorse di rete, risorse di calcolo o di storage.

4.2.3 Network Functions Virtualization Infrastructure

Proseguiamo con la descrizione dell'architettura NFV delineata da ETSI con un altro componente chiave, l'infrastruttura NFV. Denominata NFVI (sez. 4.2.1), costituisce il motore della tecnologia analizzata, comprendendo la totalità dell'hardware e software che permette di eseguire e gestire le VNF. La figura 4.4 riprende nel dettaglio l'architettura illustrata nelle sezioni precedenti. Nonostante la NFVI sia composta da più blocchi logici fondamentali, è essenziale chiarire che questa visione non è esportata anche nei livelli

superiori. Dalla prospettiva delle VNF, infatti, l'infrastruttura costituisce una singola entità designata a fornire una struttura concreta su cui eseguire il codice della funzione, interfacciandosi semplicemente con il reference point corrispondente.

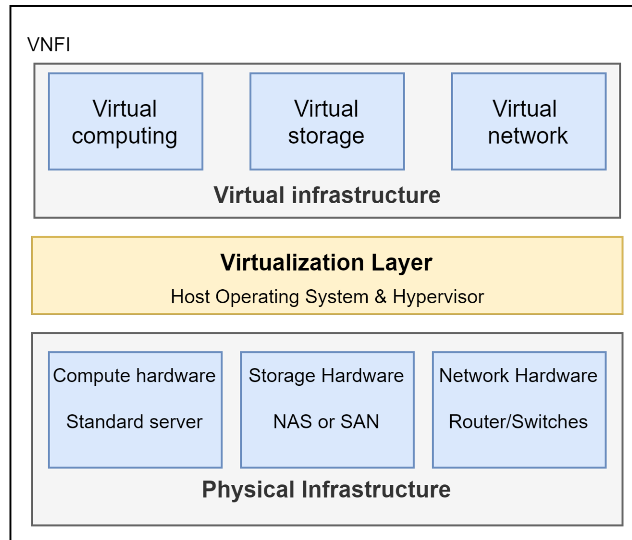


Figura 4.4. Network Function Virtualization Infrastructure

Scendendo, invece, nell'analisi dei singoli moduli costituenti, si può notare chiaramente la presenza di una struttura fisica, che fornisce le risorse concrete a tutta l'impalcatura virtualizzata. In questo livello si possono differenziare, a grandi linee, tre macro-categorie di risorse hardware: le risorse computazionali, le risorse di memoria, e le risorse di rete. Per quanto concerne le prime, si tratta di hardware COTS (Commercial Off-the-Shelf) non ottimizzato per un particolare uso ma *general purpose*. Tipicamente, in questo contesto, si fa riferimento a server standard che utilizzano hardware commerciale (es. CPU Intel, GPU Nvidia etc.). I dispositivi che forniscono le risorse di memoria tipicamente sono tipicamente dei dischi all'interno di una rete di storage (SAN) che sfruttano la memoria di massa contenuta in dispositivi condivisi all'interno della rete stessa (NAS). Infine, le risorse di rete includono tutti i dispositivi basilari di cui una rete necessita quali, ad esempio, switch di livello 2 e livello 3, ma anche link di tipo diretto o wireless.

Lo strato intermedio, rappresentato dal virtualization layer, è costituito da un hypervisor che, astraendo le risorse sottostanti, permette di disaccoppiare lo strato software delle VNF dallo strato hardware. Questo permette di eseguire funzioni su qualsiasi tipo di hardware e utilizzando tecnologie di virtualizzazione differenti.

Senza scendere nel dettaglio, si può analizzare questo concetto fornendo una visione dello stato dell'arte riguardo le tecnologie di implementazione di questo strato. Tipicamente, anche con un'accezione impropria, si utilizzano i termini “Virtualization Layer” e “Hypervisor” riferendosi allo stesso componente; questo è dovuto principalmente al fatto che delineava la tecnologia implementata nella maggior parte dei casi, quando ETSI ha definito lo strato [33]. In questo caso, la tecnologia utilizzata per implementare il virtualization layer prevede la presenza di un componente installato al di sopra di un sistema operativo ospitante, chiamato appunto Hypervisor, che permette di astrarre le risorse del sistema al fine di crearne uno nuovo, virtualizzato, che possa eseguire un sistema operativo (guest). Questo sistema ospitato definisce nell'insieme una *Virtual Machine* sulle quali eseguire le VNF. In parte questo approccio, che sarà ripreso nel capitolo 6, è stato seguito

anche in questo lavoro, in cui, sfruttando un Hypervisor open source, è stato possibile creare due macchine virtuali contenenti una VNF.

La principale alternativa a questo approccio, anch'essa sfruttata ai fini della progettazione della soluzione, è costituita da una tecnologia che fa ricorso ad una declinazione del concetto di virtualizzazione definita "leggera". Contrariamente alle VM, questa tecnologia non prevede l'esecuzione di un sistema operativo guest completo, ma riesce comunque, grazie a svariate soluzioni e meccanismi, a creare un ambiente virtualizzato e isolato all'interno del sistema operativo ospitante. In questo modo, ogni VNF può essere eseguita su un "container" diverso senza chiedere aiuto ad un componente esterno (hypervisor) e soprattutto senza ricorrere all'esecuzione di un sistema operativo completo per eseguire una sola funzione.

Nell'ultimo strato troviamo invece il livello virtualizzato, contenente le rappresentazioni virtuali delle risorse che l'hypervisor mette a disposizione delle VNF. In riferimento alla figura 4.4, è possibile analizzare le differenti categorie di risorse virtuali: per quanto concerne il modulo di virtual computing, le rappresentazioni corrispondono a macchine virtuali o container; nel virtual storage, invece, possiamo trovare file, volumi o altri oggetti di storage; nella categoria virtual network citiamo sicuramente router, switch o link virtuali.

Seguendo nuovamente le linee guida di ETSI in [33], proponiamo un ulteriore raggruppamento riguardo la NFVI. È possibile, infatti, delineare tre domini di lavoro nell'infrastruttura:

- Compute Domain: a questo dominio appartengono le risorse hardware COTS. A riguardo ETSI cita la CPU, la scheda di rete, lo storage e il server. Per trasposizione, si può identificare questo dominio con parte dello strato fisico, escludendo le risorse di rete che compongono un dominio diverso;
- Hypervisor Domain: effettua una mediazione tra le risorse del *Compute Domain* e le macchine virtuali delle funzioni software. Grazie alla virtualizzazione delle risorse, è in grado di emulare qualsiasi tipo di hardware sottostante, al punto che una macchina virtuale non è in grado di identificare "l'hardware virtuale" da quello fisico;
- Network Domain: dominio incaricato di fornire una strumentazione completa dal punto di vista della connettività della rete. Include sia le risorse fisiche, sia quelle virtualizzate.

4.3 Network Functions Virtualization MANO

In questa sezione, andremo a completare l'analisi sulla tecnologia proposta da ETSI, delineando le caratteristiche principali del modulo di gestione e orchestrazione, proponendo infine un esempio di implementazione.

4.3.1 Architettura

Procedendo con la descrizione degli standard ETSI, giungiamo alla presentazione dell'ultimo componente fondamentale della tecnologia NFV descritto in [34]. Proponendo la figura 4.5 come riferimento, individuiamo nell'architettura tre blocchi funzionali principali:

- *NFV Orchestrator (NFVO)*;
- *VNF Manager (VNFM)*;
- *Virtualized Infrastructure Manager (VIM)*;

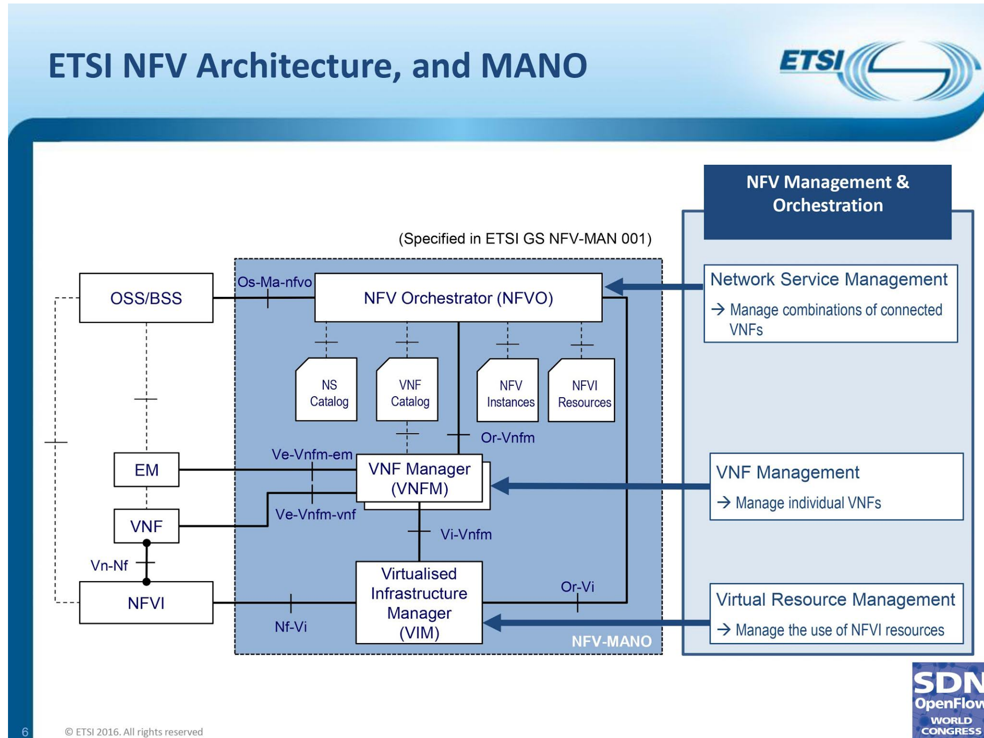


Figura 4.5. Architettura di ETSI MANO (fonte: OPNFV Summit 2016)

Prima di procedere con la descrizione dei singoli blocchi, è bene precisare che la descrizione degli elementi e delle interfacce principali è stata presentata nelle sezioni precedenti; gli altri componenti elementari presenti in figura non sono funzionali allo scopo di questo lavoro, pertanto si rimanda, per la loro descrizione, allo standard [34].

Il primo blocco preso in analisi in questa descrizione è il modulo di orchestrazione NFV. Posto logicamente nel livello più alto dell'architettura, il modulo NFVO è responsabile principalmente di due operazioni:

- l'orchestrazione delle risorse fornite dall'infrastruttura;
- la gestione del ciclo di vita completo dei servizi di rete (citati brevemente nella sezione 4.2.2).

Queste operazioni sono realizzate, rispettivamente dal *Resource Orchestration* e dal *Network Service Orchestration*. Il primo offre all'NFVO le funzionalità per permettere l'accesso alle risorse virtualizzate fornite dall'infrastruttura, fornendo un'astrazione che lo rende indipendente dal singolo gestore dell'infrastruttura (vedi *VIM*). Queste riguardano l'autorizzazione necessaria per richiedere le risorse, sia da parte dei singoli nodi che dai Network Service, andando così a definire la suddivisione delle stesse, ma anche l'enforcement delle policy per entrambe le entità. Il secondo, invece, permette di coordinare

tutte le funzioni fondamentali che compongono un servizio di rete, riuscendo così a configurarlo ed eseguirlo. Inoltre è responsabile per la definizione delle policy, ma anche della topologia e della sua gestione.

Il secondo blocco fondamentale, è rappresentato dal VNF Manager. Esso è responsabile per la gestione completa del ciclo di vita delle istanze VNF implementate e, come si evince dalla figura 4.5, è eseguito in più repliche, ciascuna dedicata ad una VNF in esecuzione. In alcuni casi, è possibile raggruppare più funzioni dello stesso tipo sotto la gestione di un'unica istanza di VNFM. La grande maggioranza dei task che questo modulo deve svolgere, riguarda azioni ordinarie che possono essere generalizzate nella gestione di un tipo generico di VNF; questo si riflette sull'implementazione stessa del blocco, che risulta in gran parte simile in tutte le sue declinazioni. Tuttavia, potrebbe essere necessario, in alcuni casi, gestire comportamenti e richieste tipici di una particolare funzione di rete. A tal proposito, è possibile specificare pacchetti aggiuntivi contenenti la descrizione dei task necessari al management della VNF in esame. L'implementazione e la definizione del funzionamento di una determinata VNF passano attraverso la scrittura di un template chiamato *Virtualized Network Function Descriptor*, che, una volta servito al modulo NFV-MANO, viene utilizzato per scatenare l'istanziazione di una nuova VNF, seguendo i dettagli presenti nel file descrittore. Grazie alle indicazioni fornite, il modulo di gestione sarà in grado di richiedere le risorse richieste dalla VNF, seppur senza violare i vincoli definiti più ad alto livello, che sovrascrivono di fatto quelli indicati nel VNFD. Questo meccanismo, basato sulla descrizione dei requisiti all'interno di un descrittore, permette di istanziare una VNF su qualsiasi infrastruttura, indipendentemente dal tipo di vendor.

L'ultimo blocco dell'architettura NFV-MANO che separa il componente stesso dall'infrastruttura sottostante è il *Virtualized Infrastructure Manager (VIM)*. Esso si occupa di controllare e gestire tutte le risorse hardware che fanno parte del modulo NFVI e, come nel caso del VNF Manager, è possibile che sia dedicato ad un solo tipo di risorse oppure che sia in grado di gestire sia risorse computazionali, sia risorse di memoria, sia risorse di rete. Il modulo VIM collabora con gli altri mediante l'esposizione di due interfacce, una northbound interface ed una southbound interface, in direzione, rispettivamente, dei moduli di più alto livello e di più basso livello. La prima fornisce ai livelli superiori le API necessarie per interagire con le risorse hardware, la seconda, invece, legata direttamente all'infrastruttura fisica, permette di riflettere queste azioni direttamente sulle risorse.

4.3.2 Open Source MANO

Open Source MANO è una piattaforma di gestione e orchestrazione NFV che nasce sotto la guida diretta di ETSI. Lanciato nel Giugno 2016 in occasione del *Mobile World Congress*, attualmente si trova alla release numero nove. Si tratta quindi di un esempio di implementazione dell'infrastruttura di gestione descritta precedentemente (MANO). Per tutti i dettagli implementativi rimandiamo alla documentazione ufficiale [35] [36].

La caratteristica principale di OSM coincide con l'utilizzo di file descrittori per scatenare l'istanziazione di funzioni di rete. In un file *VNFD Descriptor*, infatti, vengono descritte tutte le proprietà desiderate da una particolare funzione di rete, la cui creazione viene affidata successivamente direttamente dall'infrastruttura. Tutto il ciclo di vita del servizio è, tuttavia, amministrato dal modulo chiamato *Service Orchestrator*.

Una caratteristica sicuramente di rilievo è il supporto al software Juju[3], il quale, definendo degli oggetti chiamati "charms", permette la creazione di VNFM, per gestire ogni singola funzione implementata. Infine da sottolineare il ruolo del *Resource Orchestrator* nella gestione e nella coordinazione delle risorse gestite dall'NFVI.

4.4 Virtual Network Security Functions

In questa sezione verranno presentati i concetti base di una vNSF, con particolare riferimento a quella prescelta per questo lavoro; affronteremo la discussione teorica sulle Virtual Private Network nel corso del capitolo 5.

Il concetto di virtual Network Security Function (vNSF) si inserisce in un contesto intermedio tra la gestione e prevenzione dei rischi in una rete generica e la tecnologia di virtualizzazione di funzioni di rete NFV descritta nelle sezioni precedenti. Si tratta infatti di implementare, su un'architettura come quella descritta nella sezione 4.2, funzioni di sicurezza per le reti. In un contesto antecedente a quello anticipato precedentemente, il design della rete prevedeva l'inclusione di apparati dedicati a svolgere funzioni come firewall, intrusion detection/prevention systems o deep packet inspection, la cui implementazione è specifica per ogni fornitore. Questo influenzava pesantemente il design della rete, la cui struttura, era fornita di sottoreti ad hoc per ospitare questo tipo di servizi; inoltre un aspetto complesso da gestire riguardava le informazioni di routing e le modalità per dirigere il traffico attraverso i middleboxes per "attivare" le funzioni.

Lo step successivo, introdotto da SDN, ha permesso una gestione del "traffic steering" molto più agile, permettendo di deviare il traffico semplicemente modificando le tabelle di inoltro degli switch virtuali. Il passo definitivo verso questo approccio è stato determinato dalla diffusione della tecnologia NFV che elimina, di fatto, ogni apparato fisico e molti vincoli sul design della rete, favorendo la nascita del paradigma *Security as a Service* (SECaaS): questo delinea la capacità di eseguire funzioni di sicurezza (software) in un ambiente remoto, come ad esempio un datacenter, i cui servizi vengono ottenuti tramite l'accesso alla rete Internet, o su un semplice nodo della rete dell'*Internet Service Provider* (ISP). In questo modo è facile implementare delle funzioni di sicurezza, molto meno dipendenti dal fornitore, e includerle all'interno di una virtual machine, eseguita in un'infrastruttura NFV. Questo è stato l'approccio di base seguito in questo lavoro i cui dettagli implementativi sono definiti nel capitolo 6.

Il servizio di sicurezza che si vuole implementare in questo lavoro è catalogabile tra le funzioni di *Attack Prevention* ed è rappresentato da una Virtual Private Network. Nel prossimo capitolo, andremo a descrivere i concetti fondamentali di questa funzione, discutendone il tipo di implementazione e lo strumento utilizzato per realizzarla, motivandone la scelta.

[3]https://juju.is/

Capitolo 5

Virtual Private Network

Lo scopo di questo capitolo è quello di delineare alcuni concetti fondamentali su cui poggia parte della nostra soluzione, riguardo la tecnologia delle VPN e in particolare sul protocollo IPsec.

5.1 Visione d'insieme

Per *Virtual Private Network* (VPN) si intende una tecnica, che indistintamente, può fare ricorso a risorse hardware o software, per realizzare una rete privata, attraverso canali di comunicazione o reti considerati non fidati. Ciò permette alle aziende di estendere la propria rete locale, con un conseguente taglio nei costi inerenti al collegamento dei siti interni dislocati geograficamente, ma anche di utenti remoti, sostituendo le linee private a noleggio con reti pubbliche condivise. L'utilizzo di un canale di comunicazione pubblico implica alcuni accorgimenti dal punto di vista della sicurezza dei dati, particolarmente importanti nelle reti basate su protocollo IP (RFC-791 [37]), che non garantisce, da solo, i requisiti fondamentali di sicurezza. Infatti, gli indirizzi coinvolti nella comunicazione non sono autenticati, per cui non si ha la certezza di comunicare con l'interlocutore atteso, e non è fornito alcun tipo di protezione sui pacchetti riguardo l'integrità degli stessi, l'autenticazione del mittente, la riservatezza e contro i duplicati. L'assenza di queste proprietà si traduce un elenco di attacchi conducibili contro i canali di comunicazione ampio, che possono essere generalizzati e raggruppati in quattro categorie:

- attacchi contro l'integrità: in qualsiasi tipo di comunicazione, anche se sicura, è possibile manipolare e modificare le informazioni contenute nei pacchetti. È possibile, però, introdurre delle tecniche per mitigare questo attacco, che consistono nel rilevamento di tali manipolazioni;

- attacchi contro l'identità: riguarda una tipologia di attacchi che colpiscono una delle caratteristiche fondamentali di un tipo qualsiasi di comunicazione, vale a dire l'identità degli interlocutori. Questa tipologia di attacchi è finalizzata ad assumere l'identità di un soggetto per ottenerne gli eventuali privilegi. L'autenticazione delle parti mitiga questo tipo di attacco;

- attacchi contro la riservatezza: raggruppa tutti gli attacchi che mirano ad ottenere informazioni riservate, sfruttando la visibilità delle informazioni in transito su una rete. La cifratura del traffico può sventare questa tipologia di attacchi;

- attacchi di tipo replay: identifica gli attacchi che consistono nel replicare l'invio di uno o più pacchetti all'interno della comunicazione per trarne un vantaggio. Tipicamente vengono realizzati da un attaccante che intercetta la comunicazione, ponendosi tra i due lati, e che replica l'invio di una credenziale di autenticazione comunicata da un host all'altro, simulandone l'identità.

Lo scopo di una VPN, è quello di fornire protezione da questi attacchi, proteggendo, in toto o in parte, il traffico di rete, in base alla sua implementazione. A queste proprietà, ottenibili anche per mezzo di altri protocollo, le VPN aggiungono altri vantaggi unici di questa tecnologia:

- anonimato: si tratta di una caratteristica di alcune realizzazioni di livello 3 che, grazie alla cifratura del pacchetto, nascondono l'indirizzo mittente effettivo;
- trasparenza della protezione: in alcune realizzazioni, il traffico è inoltrato verso i terminatori che realizzano la VPN, obbligando il traffico a transitare all'interno del canale virtuale sicuro in modo del tutto trasparente all'utente.

Le tecniche implementative sono molteplici e conducono a gradi di sicurezza differenti; in particolare, la realizzazione di una VPN mediante l'utilizzo di indirizzi IP nascosti o tramite un tunnel IP non costituiscono versioni sicure di questa tecnologia. Al contrario, una delle versioni più utilizzate, è realizzata attraverso un tunnel IP arricchito con alcune funzioni complementari, che ne garantiscono la sicurezza. Questa tipologia, conosciuta anche come *Secure VPN*, può fornire protezione con funzioni di integrità, autenticazione, riservatezza e numerazione dei pacchetti, a seconda del grado di protezione voluto. L'integrità e l'autenticazione di ciascun pacchetto sono garantite da un valore calcolato sull'intero pacchetto (*keyed digest*), che include al suo interno una chiave e che viene inviato insieme al pacchetto stesso; una volta arrivati al destinatario, è possibile effettuare di nuovo il calcolo per confrontare il valore ottenuto con quello ricevuto, asserendo, così, che il mittente è in possesso della chiave. Questo procedimento è necessario al fine di autenticare ogni singolo pacchetto, evitando così che un attaccante possa identificare una parte, dopo che questa abbia dimostrato la propria identità. La riservatezza, banalmente, è ottenuta tramite cifratura dei pacchetti inviati in rete con una chiave simmetrica; il destinatario in possesso della chiave può decifrare e leggere il contenuto del pacchetto. La numerazione dei pacchetti, invece, garantisce la ricezione degli stessi una volta soltanto: questo evita che un attaccante possa intercettare i pacchetti con cui una parte effettua l'autenticazione e riproporli al destinatario, fingendo di essere in possesso della chiave (attacco replay).

5.2 VPN tramite IPsec

Andiamo adesso a definire lo standard utilizzato come linea guida per la realizzazione della VPN in questo lavoro.

Nell'RFC-4301 [38], IETF definisce l'architettura standard per implementare funzioni di sicurezza a livello rete per creare VPN su reti non fidate o per fare sicurezza end-to-end. In questa discussione, ci concentreremo, tuttavia, sulla definizione dei concetti chiave riguardanti la costruzione di un tunnel VPN e si rimanda alla documentazione ufficiale per tutti gli altri dettagli.

La logica alla base del funzionamento di IPsec prevede la negoziazione di alcuni parametri, inerenti la protezione del traffico, tra i sistemi coinvolti nella comunicazione. Il

salvataggio di questi attributi in uno stato, crea, di fatto, una connessione logica tra le entità coinvolte nella comunicazione che prende il nome di *Security Association* (SA). Queste permettono di mantenere, in una base dati chiamata *SA Database* (SAD), le caratteristiche di sicurezza da applicare su una singola direzione del canale di comunicazione. Ne consegue che per avere una protezione completa di un canale bidirezionale è necessaria la negoziazione di due SA differenti che, potenzialmente, potrebbero contenere parametri differenti. La struttura di una SA prevede un campo per indicare gli indirizzi dei sistemi coinvolti nella comunicazione, un campo per indicare il protocollo utilizzato da IPsec, uno che indichi gli algoritmi utilizzati per garantire le proprietà di sicurezza richieste con le eventuali chiavi, e un valore chiamato *Security Parameter Index* (SPI). L'esempio mostrato in figura 5.1 riporta la struttura appena delineata e sottolinea la presenza di due differenti SA, con caratteristiche differenti, delegate alla protezione di un canale bidirezionale. La figura mostra come sia possibile utilizzare algoritmi e chiavi differenti in base alla direzione del traffico per cui la SA offre il servizio di protezione.

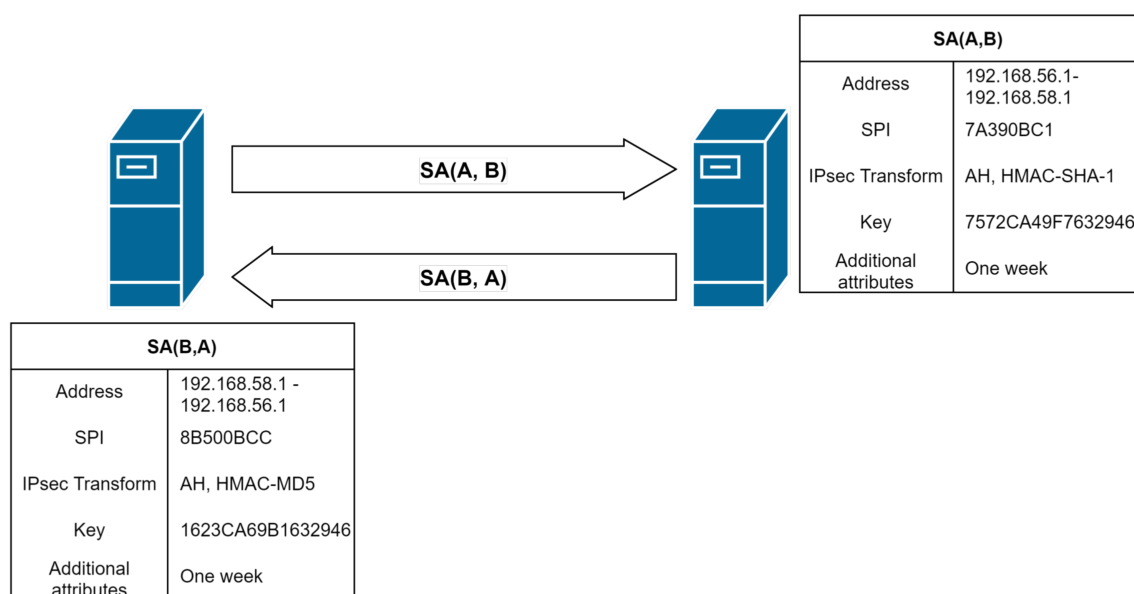
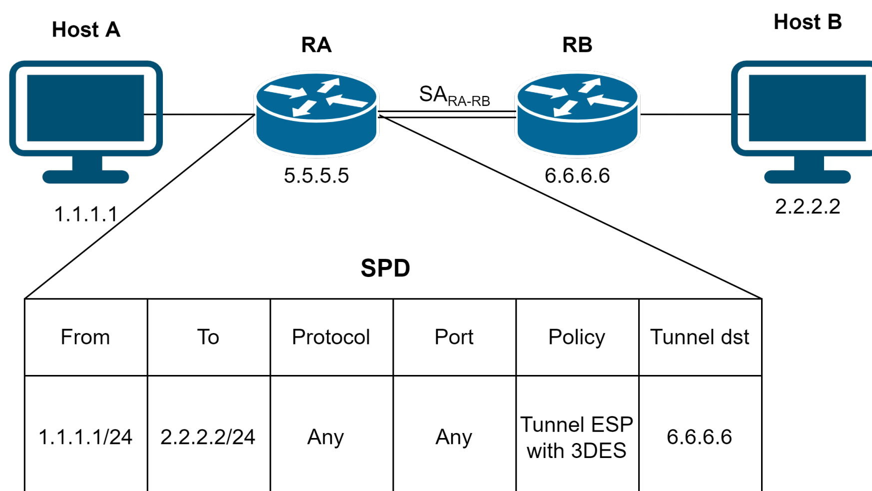


Figura 5.1. SA tra due sistemi

Il valore corrispondente allo SPI in figura definisce un indice per effettuare l'accesso nel SAD ed individuare la SA corrispondente. In aggiunta, un ulteriore database, chiamato *Security Policy Database*, è utilizzato da IPsec per tenere traccia delle policy da applicare sul traffico: banalmente seleziona quale tipologia di traffico, basandosi tipicamente sugli indirizzi IP, deve essere processato da IPsec e quale invece deve essere, invece, gestito come traffico normale. Come mostrato dall'esempio in figura 5.2, il traffico è selezionato dalla combinazione dei valori indicati dai campi degli indirizzi sorgente e destinazione, del protocollo e della porta utilizzata. Secondo il contenuto del database raffigurato, tutto il traffico diretto verso l'host B proveniente dall'host A viene protetto attraverso un tunnel VPN che utilizza il protocollo ESP e cifra i dati con 3DES (RFC-1851 [39]).

Il flusso completo di IPsec prevede che, quando un pacchetto deve essere inviato, l'host esegua un lookup nel SPD e, qualora il traffico in esame dovesse essere gestito da IPsec, esegua un secondo lookup nel SAD, nel quale leggere i parametri fondamentali di sicurezza per costruire il pacchetto.

L'architettura delineata da IETF definisce due protocolli in grado di fornire servizi di sicurezza al traffico dati, *Authentication Header* (AH) [40] e *Encapsulating Security*



From	To	Protocol	Port	Policy	Tunnel dst
1.1.1.1/24	2.2.2.2/24	Any	Any	Tunnel ESP with 3DES	6.6.6.6

Figura 5.2. Esempio di SPD in un terminatore VPN

Payload (ESP) [41], utilizzabili per modificare i pacchetti IP, e un protocollo di scambio chiavi denominato *Internet Key Exchange* (IKE) [42]. Il primo formato, divenuto opzionale nella versione v3 di IPsec, fornisce integrità, autenticazione e protezione da attacchi replay, per ciascun pacchetto. La protezione fornita da AH copre quasi tutto il pacchetto IP, tuttavia alcuni campi (es. TTL) vengono modificati durante la trasmissione per cui se venissero inclusi nel calcolo dell'integrità, questo causerebbe malfunzionamenti. I servizi offerti dal protocollo AH vengono garantiti tramite la modifica di ogni singolo pacchetto IP, cui viene aggiunto un header strutturato secondo lo schema fornito in figura 5.3.

Next Header	Payload Length	Riservato
Security Parameter Index (SPI)		
Sequence Number		
dati di autenticazione Integrity Check Value (ICV)		

Figura 5.3. Formato dell'header AH

In particolare, il campo dedicato all'header successivo contiene il protocollo del livello successivo utilizzato e imbustato all'interno del pacchetto; il campo contenente il *Sequence Number* è utilizzato per proteggere la comunicazione da attacchi di tipo replay, numerando progressivamente i pacchetti; il campo dedicato all'autenticazione, contiene un valore (ICV) che rappresenta un digest contenente una chiave condivisa e calcolato su tutto il pacchetto. L'utilizzo di una chiave garantisce l'identità dell'interlocutore: in particolare, quando viene ricevuto un pacchetto IPsec protetto con AH il sistema estrae l'ICV ed utilizza lo SPI per accedere alla SA corretta e prelevarne i parametri e gli algoritmi; a partire da questi dati, viene ricalcolato il valore di autenticazione sull'intero pacchetto e viene confrontato con quello estratto dal pacchetto. Se i valori differiscono necessariamente si è verificata una manomissione sul pacchetto oppure la chiave utilizzata nel calcolo non è corretta.

Il secondo formato definito da IPsec nella terza versione fornisce i servizi di riservatezza, autenticazione del mittente, integrità e protezione da attacchi replay. Tuttavia ESP, al contrario di AH, non copre tutto il pacchetto, ma esclude l'header IP. La protezione messa in atto dal protocollo dipende da quali dei servizi vengono negoziati nella SA e quindi abilitati: a tal proposito nell'RFC-4303 [41] le possibili combinazioni fornite indicano che è opzionale il supporto alla sola riservatezza, utilizzando eventualmente AH per le altre proprietà, ma è un requisito fondamentale il supporto all'utilizzo della proprietà di integrità, sia in combinazione con la riservatezza, sia come unico servizio. In analogia con il primo formato, anche ESP integra ciascun pacchetto con dati aggiuntivi che, in questo caso, sono divisi in *header* e *trailer* e delimitano i dati da cifrare. Nella struttura riportata in figura 5.4, lo SPI e il *Sequence Number* sono inseriti nell'header, mentre i campi *Padding*, *Padding Length* e *Next Header* sono inseriti nel trailer.

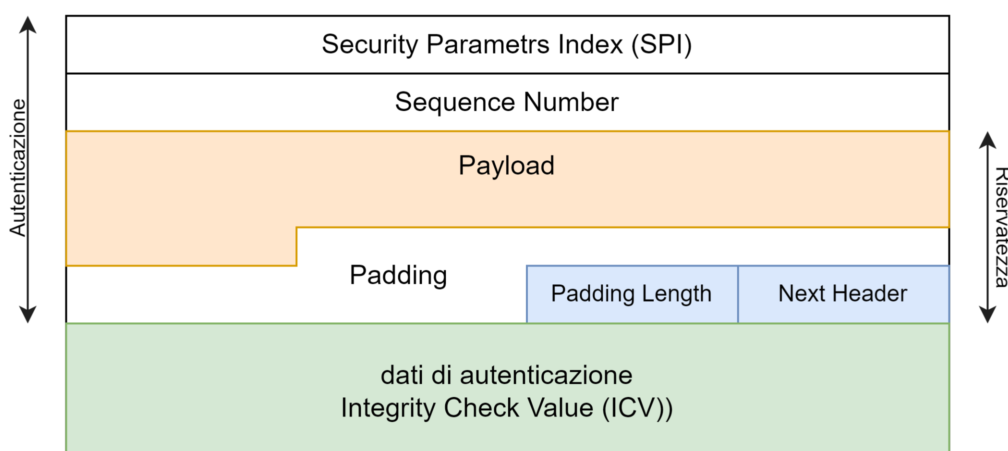


Figura 5.4. Struttura di un pacchetto protetto con ESP

La maggior parte dei campi sono analoghi a quelli presentati per AH, ad esclusione di quelli contenenti il Padding e la sua dimensione, indispensabili per il servizio di riservatezza. Infatti, alcuni algoritmi di cifratura lavorano su una quantità di dati multipla di determinati valori fissi per cui può risultare necessario ampliare il Payload e ottenere la dimensione corretta. Infine, per poter ricostruire il pacchetto originario è fondamentale conoscere la dimensione dei dati aggiunti.

I due formati, che definiscono informazioni aggiuntive trasportate nei pacchetti, caratterizzano, in base alle modalità di costruzione del pacchetto IP, modalità diverse di servizi di sicurezza. La prima modalità, denominata *Transport Mode*, è utilizzata per fare sicurezza end-to-end, in cui IPsec è disponibile direttamente sugli host. La costruzione del pacchetto IP prevede un passo aggiuntivo in cui una parte contenente informazioni riguardo IPsec è aggiunta prima dell'header di livello 4, come mostrato in figura 5.5. In questa modalità, in caso di cifratura, vengono protetti solo i protocolli di livello superiore, ottenendo un costo computazionale inferiore a quello richiesto dal secondo scenario, ma una protezione minore sull'header, dalla quale vengono esclusi i campi variabili. La seconda modalità, chiamata *Tunnel Mode*, è utilizzata per la realizzazione di VPN e prevede l'utilizzo IPsec direttamente sui gateway. In questo caso, il tunnel si costruisce inserendo i dati relativi ad IPsec tra due header IP, di cui uno responsabile dell'inoltro del pacchetto tra i due gateway e l'altro dell'inoltro all'interno della rete di destinazione. Giunto al gateway destinazione, l'header più esterno viene rimosso e, dopo aver estratto il pacchetto dalla protezione di IPsec, questo viene inoltrato seguendo le indicazioni dell'header più interno. Da questa descrizione si evince che in tunnel mode vengono protetti anche

i campi variabili, ma in fase di inoltro e ricezione è necessario eseguire più operazioni (bisogna rimuovere più header per accedere ai dati ed eventualmente decifrare una parte più grande) per questo risulta computazionalmente più pesante.

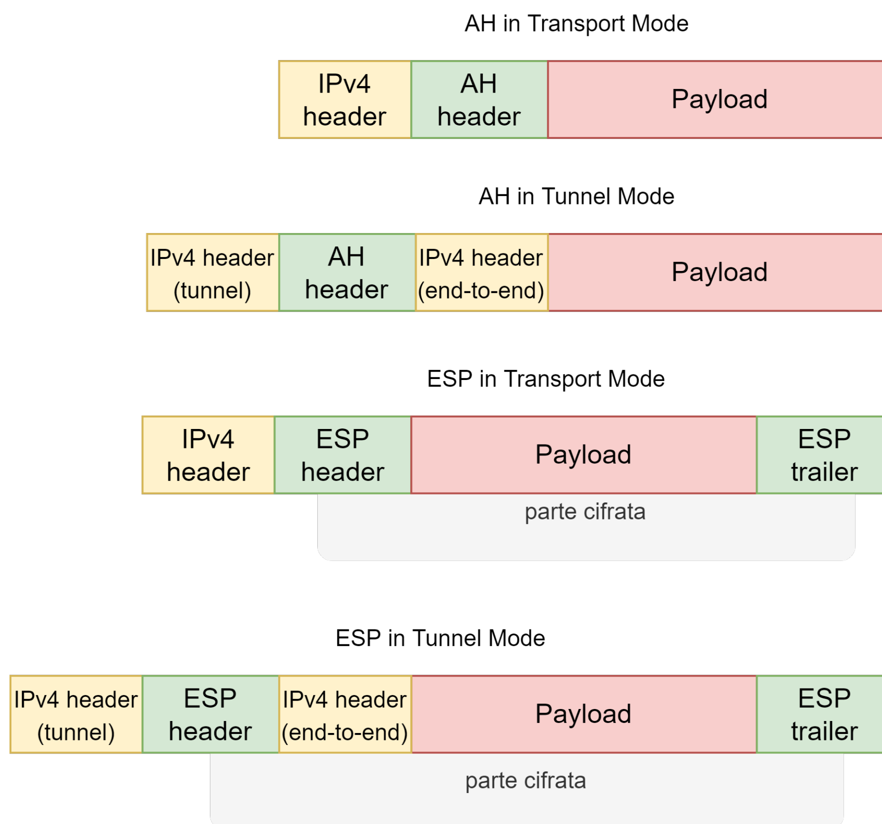


Figura 5.5. Formati dei pacchetti IPsec

IPsec definisce un protocollo di gestione delle chiavi simmetriche, fondamentale per semplificare la configurazione di ogni dispositivo. Infatti, per supportare i servizi di sicurezza di autenticazione e cifratura dei pacchetti, è necessario installare le chiavi su ogni endpoint della VPN; per non configurare manualmente tutti i segreti condivisi, IKE (RFC-2409 [42]) corre in soccorso del protocollo IPsec, automatizzando la gestione. IKE si appoggia su due protocolli distinti, uno denominato *ISAKMP* (RFC-2408 [43]), indica le modalità per negoziare stabilire e cancellare le SA, l'altro denominato *OAKLEY* (RFC-2412 [44]), che realizza a livello pratico lo scambio autenticato di chiavi simmetriche. Come rappresentato in figura 5.6, la negoziazione e lo scambio delle chiavi avviene attraverso due fasi e attraverso la creazione di due SA diverse: la prima fase prevede la creazione di una SA per protegge la negoziazione ISAKMP che avviene durante la seconda fase, durante la quale avviene la negoziazione della SA vera e propria richiesta da IPsec. La prima SA creata, quella che fa da involucro a quella effettiva, può essere utilizzata per negoziare più SA diverse. Naturalmente, prima di iniziare tutte le fasi di negoziazione, è necessario che ogni parte fornisca un prova della propria identità autenticandosi. Discutiamo di seguito i possibili metodi supportati da IKE:

- *Public Key Encryption*: grazie all'installazione di certificati X.509 (RFC-5280 [45]), che contengono informazioni sull'identità del soggetto a cui appartiene, e, facendo ricorso alla cifratura asimmetrica, è possibile autenticare le parti. La cifratura

asimmetrica si basa sull'utilizzo di due chiavi distinte, una privata e una pubblica, e sul concetto fondamentale secondo cui se la cifratura avviene mediante una chiave, è necessaria l'altra per decifrare. In questo meccanismo di autenticazione, un nodo invia un valore casuale all'altra parte coinvolta nella negoziazione, cifrato con la chiave pubblica del nodo ricevente (contenuta nel suo certificato scambiato nella prima fase); se il nodo ricevente è in grado di decifrare ed inviare il valore corretto all'altro nodo, allora dimostra di essere in possesso della chiave privata. Il dettaglio fondamentale di questa procedura risiede nell'esclusività della conoscenza della chiave privata: solo all'endpoint è nota la chiave privata per cui è l'unico in grado di decifrare;

- *Revised Public Key Encryption*: si tratta di una variante dell'autenticazione classica realizzata tramite cifratura a chiave pubblica in cui vengono eseguite meno operazioni crittografiche;
- *Pre-shared Key*: in questa modalità, le chiavi vengono pre-installate manualmente sugli endpoint che possono essere identificati esclusivamente dal rispettivo indirizzo IP. Questo aspetto risulta particolarmente problematico per gli utenti mobili, il cui indirizzo cambia;
- *Digital Signature*: anche in questo caso si fa ricorso ai meccanismi forniti dalla cifratura asimmetrica, invertendo, però, l'utilizzo delle due chiavi. Infatti, l'endpoint che vuole autenticarsi sceglie un messaggio da inviare e ne calcola un digest che poi cifra con la sua chiave privata. Il valore ottenuto viene concatenato al messaggio e inviato all'altro interlocutore che, decifrando con la chiave pubblica, ricava il digest. A questo punto il ricevente può facilmente confrontare il digest ricavato con quello calcolato sul messaggio ricevuto, verificando, così, l'identità del mittente. Questa modalità garantisce il non ripudio della negoziazione IKE, per cui l'utente non potrà negarne la validità.

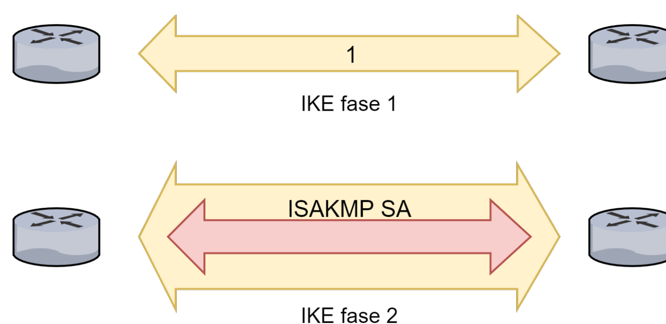


Figura 5.6. Negoziazione IKE

5.3 Scenari

Per completezza illustriamo brevemente due possibili architetture di esempio di VPN, che delineano due scenari differenti in cui richiedere specifici servizi di sicurezza. Al fine di semplificare la comprensione dei concetti illustrati successivamente nel capitolo 6, è necessario citare lo scenario *site-to-site*. In questa particolare architettura riassunta in figura 5.7, il tunnel VPN permette di collegare due reti locali, estendendo di fatto il loro dominio; nel caso delineato in figura, il tunnel VPN attraverso la rete Internet, permette di

collegare un sito aziendale esterno alla sede centrale, permettendo ai dispositivi connessi remotamente di avere gli stessi privilegi di una connessione diretta locale. In effetti, gli utenti collegati dal sito esterno riceveranno indirizzi IP privati appartenenti alla rete centrale aziendale. In questo lavoro, come sarà descritto nel dettaglio nel capitolo 6, l'architettura proposta ricalca lo scenario analizzato.

La seconda architettura, che prende il nome di *access VPN*, rappresenta un approccio misto a quello definito precedentemente, in cui il software IPsec è installato da un lato su un host, dall'altro su un gateway. Questo scenario, rappresentato in figura 5.8, permette ad un singolo host remoto di collegarsi alla rete centrale aziendale: questo approccio è diventato più diffuso nell'ultimo periodo, in cui sempre più persone, costrette a lavorare da remoto, necessitano un accesso diretto alle risorse aziendali.

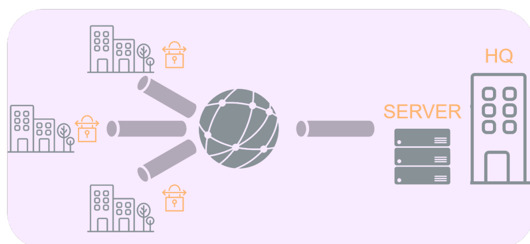


Figura 5.7. Site-to-site VPN

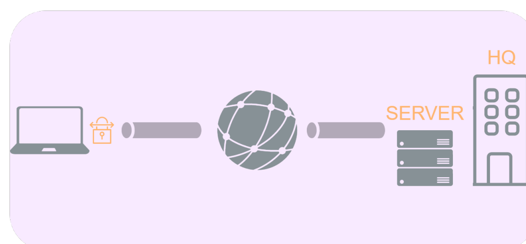


Figura 5.8. Access VPN

5.4 Soluzioni VPN di livello N

La descrizione di IPsec, trattata nella sezione 5.2, coincide soltanto con una delle modalità di realizzazione di una Virtual Private Network. Infatti, da un punto di vista puramente implementativo nell'architettura della rete, le VPN possono essere realizzate a diversi livelli della pila ISO/OSI, in base ai quali sono suddivise.

Il concetto che è alla base della suddivisione consiste nell'analizzare il livello in cui il trasporto dei pacchetti è gestito: in particolare, una VPN è categorizzata in base al livello del protocollo che realizza il tunnel o in base al livello in cui questo servizio è fornito.

La soluzione basata su MPLS (RFC-3031 [46]) è annoverata tra le VPN di livello 2 in quanto il protocollo è utilizzato per imbustare pacchetti IP. Considerando la caratteristica principale di MPLS, che effettua l'inoltro dei pacchetti in base alle direttive contenute all'interno di un'etichetta inserita nei pacchetti, è possibile far arrivare un pacchetto ad un nodo destinazione, facendo percepire il percorso come un collegamento punto-punto. Infatti, l'header IP contenente la destinazione non è mai considerato e l'inoltro avviene grazie all'etichetta che indica la prossima destinazione; per questo, quando arriva a destinazione, a livello 3 risulta come se il collegamento fosse avvenuto attraverso un cavo diretto. In modo analogo ad IPsec in tunnel mode, vengono utilizzate due etichette, di cui una esterna che indica il percorso tra i due punti di accesso della rete e una più interna a selezionare il traffico corretto e separarlo dagli altri nello stesso terminatore.

Nelle soluzioni di livello 3 citiamo *L2TP* (RFC-2661 [47]) e *PPTP* (RFC-2637 [48]). In questi scenari, le VPN sono di livello 3 in quanto si utilizza una busta IP nel pacchetto, ma potrebbero appartenere al livello 2 a causa dei protocolli utilizzati per realizzare il tunnel. Nel primo caso viene fornito un servizio VPN di livello rete grazie all'utilizzo di un tunnel attraverso un access point, che possa supportare il protocollo chiamato

L2TP Access Concentrator (LAC), terminato in un server chiamato *L2TP Network Server* (LNS). Grazie all'aggiunta di un header nel pacchetto, tra quello di livello 4 e i dati, il LAC saprà come gestire l'inoltro: come prima cosa stabilisce una connessione di controllo con il LNS di destinazione che crea il tunnel, all'interno del quale, successivamente, è possibile istanziare più sessioni diverse. Prima di stabilire la connessione è prevista l'autenticazione delle parti basato sul protocollo CHAP (RFC-2759 [49]), ma non sono garantite l'autenticazione e la cifratura del traffico, per cui si delega ad un protocollo come IPsec. Inoltre non viene fornito nessun controllo sulla numerazione dei pacchetti, per cui è facilmente possibile inserire pacchetti all'interno del traffico alterando la connessione. La seconda soluzione, che prevede concetti simili alla precedente, fa ricorso ad un *PPTP Access Concentrator (PAC)* e un *PPTP Network Server (PNS)* come terminatori del tunnel. Il funzionamento prevede l'incapsulamento di pacchetti *PPP* (RFC-1661 [50]), utilizzati nel trasporto dati nelle connessioni punto-punto, all'interno di pacchetti IP, usando il protocollo aggiuntivo GRE. Senza scendere nei dettagli, è sufficiente dire che il protocollo GRE (RFC-2784 [51]) permette di incapsulare dei pacchetti all'interno di IP. In questo modo, nuovamente, viene simulata una connessione diretta tra le parti connesse dalla VPN. Come la precedente soluzione, utilizza una connessione di controllo come gestione, ma, al contrario, utilizza TCP come protocollo di trasporto.

Una VPN di livello 4 utilizza, tipicamente, il protocollo TCP per realizzare il tunnel; l'header IP più esterno consente l'instradamento tra i gateway e il pacchetto di livello 4 contiene le informazioni per inoltrare il pacchetto verso la destinazione. Dal punto di vista della sicurezza, il protocollo TLS (RFC-8446 [52]) permette di integrare i servizi del livello di trasporto, fornendo le proprietà di autenticazione delle parti oltre alla cifratura e all'autenticazione dell'intera sessione. Dal punto di vista della copertura fornita sui pacchetti TLS offre lo stesso tipo di protezione offerta da IPsec, escludendo completamente l'header IP. Operando al di sopra del livello 4, TLS è in grado di rendere sicura una sessione applicativa instaurata da un utente verso un determinato servizio. Questo permette di realizzare versioni sicure di tutti i protocolli di livello applicativo, fra cui è possibile citare POPS e Secure IMAP (RFC-2595 [53]), HTTPS (RFC-2818 [54]), SFTP [55]. Tutte le realizzazioni di questo protocollo operano sulle sessioni instaurate da un Web Browser: negli scenari tipici, il browser viene utilizzato per salvare i dati di autenticazione del server cui è collegato. Infatti, l'autenticazione tipica in TLS è unilaterale, consentendo al browser di verificare l'identità del server applicativo pur mantenendo l'anonimato nei confronti del server.

5.5 Strumenti

Con l'analisi degli strumenti che forniscono le funzionalità delle Virtual Private Network si chiude questa discussione riguardo la tecnologia VPN.

Partendo dagli strumenti presi in considerazione durante la realizzazione della VPN, ne delineiamo brevemente le caratteristiche e motiviamo le scelte effettuate. A tale scopo sono stati analizzati i seguenti software:

- *OpenVPN* [1]: si tratta di una soluzione open source che permette di creare VPN basate sui protocolli SSLv3 (RFC-6101 [56]) e TLSv1 (RFC-2246 [57]), realizzando il software sia lato client che server. Tutto il funzionamento è basato sulla libreria

[1] https://openvpn.net/

OpenSSL [2], che fornisce supporto per quanto riguarda la cifratura e l'autenticazione delle parti. Quest'ultimo servizio è supportato mediante meccanismi basati su chiave condivisa, certificati digitali e, a partire dalle ultime versioni, anche su credenziali di accesso quali nome utente e password. Il supporto a TCP ed UDP rendono questa soluzione una valida alternativa ad IPsec, in particolare nei casi in cui il fornitore di servizi Internet arrivi a bloccare alcune tipologie di VPN, tipicamente ad uso aziendale, costringendo gli utenti a virare verso un servizio apposito, con costi maggiori. Questo tipo di servizio VPN si concentra principalmente su soluzioni fornite già pre-costruite come "VPN as a service" fornendo una documentazione molto ricca verso questo aspetto, ma meno completa riguardo l'installazione manuale sui dispositivi;

- *xfrm* [3]: si tratta di un modulo presente nel kernel Linux [4] per trasformare i pacchetti IP e che viene utilizzato per implementare l'insieme di protocolli definiti in IPsec. Alla base del funzionamento di questa piattaforma ci sono i concetti di "stato" e "policy": nel primo caso definisce un oggetto in grado di operare sul Security Association Database, mentre nel secondo caso si tratta di una struttura dati operante sul Security Policy Database. Questo si traduce in una serie di comandi che permettono di definire le SA e le regole per indicare il traffico da instradare nella VPN. La configurazione passa, quindi, attraverso questi comandi che richiedono dettagli di basso livello come l'indice da utilizzare come SPI o le chiavi di cifratura e autenticazione;

- *strongSwan* [5]: si tratta di una soluzione VPN open source incentrata sulla suite di IPsec e basata sull'utilizzo dei protocolli di scambio e gestione chiave IKEv1 e IKEv2 (RFC-5996 [58]). Tutto il funzionamento di strongSwan è basato su un file system ben definito e su processi demone in grado di realizzare le operazioni richieste. Come per OpenVPN, vengono utilizzati dei file configurativi in cui esprimere le caratteristiche desiderate: il file `ipsec.conf` contiene le caratteristiche desiderate da ciascuna VPN, riportate in modo schematico e ciascuna individuata da un nome univoco; il file `ipsec.secrets` contiene tutte le chiavi necessarie ad IPsec contenute in una cartella apposita, come anche i certificati. Il demone di strongSwan è in grado di realizzare una connessione VPN estraendo le informazioni da questi file e scatenando una negoziazione IKE tra le due parti.

  Il listato 5.1 rappresenta il possibile contenuto del file di configurazione di strongSwan relativo ad una sola VPN. In particolare, tutti i valori riportati tra parentesi devono essere caratterizzati al fine di definire le peculiarità della connessione.

Listing 5.1. Template di configurazione per StrongSwan

```
1 conn {{name}}
2             type={{type}}
3             auto=start
4             keyexchange=ikev2
5             authby={{authmethod}}
6             left={{localPublicIp}}
7             leftsubnet={{lPrivateIp}}
8             right={{remotePublicIp}}
```

[2] https://www.openssl.org/

[3] https://man7.org/linux/man-pages/man8/ip-xfrm.8.html#DESCRIPTION

[4] https://www.linux.it/

[5] https://wiki.strongswan.org/

```
9               rightsubnet={{rPrivateIp}}
10              ike={{enc1}}-{{hash1}}-modp4096!
11              esp={{enc2}}-{{hash2}}!
12              aggressive=no
13              keyingtries=0
14              ikelifetime=1h
15              lifetime=8h
16              dpddelay=30s
17              dpdtimeout=120s
18              dpdaction=restart
19              leftcert={{myCertName}}
20              leftid="C={{C1}}, O={{O1}}, CN={{peer1}}"
21              rightid="C={{C2}}, O={{O2}}, CN={{peer2}}"
```

Analizzandoli in ordine, è possibile definire il tipo di connessione, tipicamente scegliendo tra*tunnel* o *transport* (altri valori sono supportati), il tipo di autenticazione utilizzata e i dati relativi agli indirizzi necessari alla realizzazione del tunnel. A tal proposito *strongSwan* supporta tre tipi di autenticazione durante la negoziazione IKEv2: *Public Key Authentication*, *Pres-Shared-Key* e, attraverso il protocollo EAP (RFC-3748 [59]), svariati metodi di autenticazione supportati da quest'ultimo. I valori `left` e `right` indicano gli indirizzi dei gateway, mentre i parametri `leftsubnet` e `rightsubnet` indicano gli indirizzi degli endpoint o di tutta la rete in cui si trovano, se si vuole proteggere tutto il traffico scambiato dalle due. È possibile definire gli algoritmi di cifratura e hashing utilizzati sia durante la negoziazione IKE sia dal protocollo IPsec per la protezione dei dati. Infine, se il tipo di autenticazione indicata richiede l'uso di certificati a chiave pubblica, è necessario specificare il nome del file che lo contiene e alcune informazioni riguardo l'emittente dello stesso. Oltre a questi parametri, è necessario stabilire informazioni secondarie, impostate ad un valore fisso: `aggressive` permette di caratterizzare la quantità di pacchetti scambiati durante la negoziazione IKE, scegliendo tra *aggressive mode* e *main mode*; `keyingtries` stabilisce quanti ulteriori tentativi devono essere fatti se la negoziazione restituisce esito negativo; `ikelifetime` e `lifetime` stabiliscono la durata, rispettivamente, della validità dei parametri IKE e dell'intera connessione; i paramteri `dpddelay`, `dpdtimeout` e `dpdaction` riguardano la gestione del protocollo *Dead Peer Detection* (RFC-3706 [60]) e permettono di definire l'azione da intraprendere quando viene rilevata l'inattività di una parte, di definire l'intervallo temporale entro cui inviare un messaggio di verifica e l'intervallo di timeout dopo il quale scatenare l'azione definita.

La scelta finale per questo lavoro è ricaduta su questo strumento che è apparso molto completo dal punto di vista delle configurazioni possibili, ma anche per quanto riguarda la documentazione. La configurazione della VPN risulta molto agile e inoltre si tratta di un software molto utilizzato quindi molto affidabile e sicuro. Questo permette di avere un bacino di utenza molto ampio per cui risulta agevole consultare le casistiche evidenziate da altri utilizzatori.

Capitolo 6

Design della soluzione

6.1 Progettazione di alto livello

In questo capitolo verranno discussi i dettagli progettuali della soluzione di questo lavoro, che ha come scopo la realizzazione di un sistema di configurazione basato su intenti per funzioni di sicurezza virtualizzate. In particolare, lo scenario di riferimento è costituito da un tunnel VPN realizzato secondo lo schema site-to-site, i cui terminatori sono realizzati da funzioni virtualizzate.

6.1.1 Architettura

L'architettura di alto livello riportata in figura 6.1 definisce lo schema risultante dall'integrazione dei servizi basati su intenti, utilizzati per veicolare la configurazione, in una piattaforma NFV. A tal proposito, nel capitolo 4 abbiamo discusso i principi e l'architettura di una piattaforma di virtualizzazione per funzioni di rete, per cui nel corso di questa sezione facciamo riferimento a quei concetti nella descrizione dei moduli aggiuntivi e delle scelte progettuali.

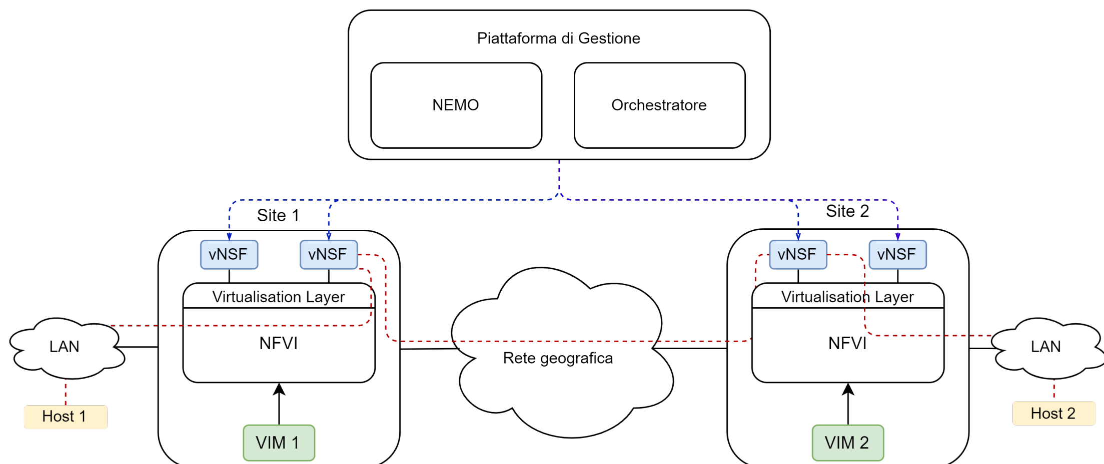


Figura 6.1. Visione di alto livello

Nello scenario delineato un controllore si occupa di gestire e configurare vNSF dislocate in due ambienti separati, ciascuno dotato della propria NFVI e il proprio VIM.

Come si evince chiaramente anche dalla figura 4.2, questo componente riceve istruzioni dai componenti di gestione ed orchestrazione per gestire le risorse hardware di ciascun sito. A partire dalle risorse hardware messe a disposizione, lo strato di virtualizzazione le trasforma in risorse virtuali, permettendo di sfruttare al massimo le capacità dell'infrastruttura. Interagendo con le API messe a disposizione dall'orchestratore, è possibile configurare un servizio realizzato sulla corrispondente infrastruttura fisica; in questo caso il *VNF Manager* è incaricato della gestione completa del ciclo di vita della macchina virtuale mentre il modulo EM è in grado di comunicare direttamente con l'immagine eseguita nella vNSF. In un'infrastruttura di questo tipo si è deciso, in fase di progettazione, di integrare il modulo a intenti direttamente nella piattaforma di gestione e orchestrazione delle vNSF; comunicando con il VNFM, permette di realizzare un'interfaccia utente basata su intenti attraverso la quale interagire con l'infrastruttura sottostante. Questo modulo si occupa di generare, di fatto, le configurazioni *Day-1* da iniettare nelle vNSF.

Per quanto riguarda i metodi di configurazione, infatti, si ricorre ad almeno due metodologie, una chiamata *Day-0 Configuration*, effettuata quando il servizio non è ancora attivo, e una chiamata *Day-1 Configuration*, realizzata durante l'esecuzione del servizio. Riprenderemo in seguito questi concetti nella sezione 6.3.1, in cui descriveremo come sono state trattate nella nostra soluzione. Generalmente, la configurazione *Day-0* viene gestita attraverso *cloud-init* mentre la *Day-1 Configuration* è affidata a *Juju*, e quindi gestita attraverso SSH. Il primo approccio, prevede la definizione di uno script, che verrà incluso nel pacchetto di cui è necessario effettuare l'onboarding, mediante il quale definire alcune caratteristiche di quello che possiamo considerare l'immagine base del sistema operativo che realizza la vNSF. L'idea è quella di definire un file yaml contenente una serie di azioni da intraprendere al momento dell'avvio, a partire da un'immagine software di base, che quindi può essere utilizzata per numerosi servizi con configurazioni diverse, evitando di creare immagini sempre diverse; attraverso i file *cloud-config*, è possibile definire parametri di rete, ma soprattutto installare pacchetti software in grado di realizzare il comportamento della vNSF (es. software che implementa IPsec, configurare SSH). La configurazione *Day-1*, invece, sfrutta tipicamente le potenzialità del protocollo SSH, ad esempio attraverso il software Juju. Questo permette di collegarsi direttamente con il servizio ed iniettare le configurazioni corrette.

La figura 6.1, in aggiunta, mostra ad alto livello il traffico scambiato dai componenti distinguendo il traffico dati, in rosso, da quello di configurazione riportato in blu. Ipotizzando che la vNSF realizzi un terminatore VPN, il traffico dati ha origine in uno dei due host, ciascuno collegato alla propria LAN. Una volta uscito dalla rete locale il traffico viene instradato verso il sito che, in uno scenario reale, potrebbe trovarsi all'interno della rete dell'operatore dell'host. Grazie alle risorse hardware di rete di cui dispone ciascuna NFVI, il traffico viene processato e inoltrato internamente verso il servizio opportuno, per poi essere inviato verso il terminatore dall'altro capo del tunnel, transitando nuovamente attraverso l'infrastruttura fisica. Il traffico relativo alla configurazione, invece, ha origine dal modulo di gestione delle vNSF e diretto verso le funzioni; in aggiunta, come anticipato precedentemente, anche il VIM riceve istruzione dal modulo centrale. Anche questo tipo di traffico è processato dall'infrastruttura fisica, passando attraverso un'interfaccia differente rispetto a quella per il flusso dati.

In riferimento all'architettura, i blocchi funzionali di base da analizzare sono:

- *Piattaforma di Gestione*: utilizzando questo nome generico ci riferiamo ad un componente logico incaricato di tutta la gestione e configurazione dei servizi di sicurezza aggiunti all'architettura;
- *vNSF*: identifica un servizio di sicurezza generico.

La Piattaforma di Gestione, come dimostrato dalla figura 6.1, è composta da due moduli fondamentali, ciascuno con caratteristiche e compiti ben delineati: il framework NEMO, al fine di generare file di configurazione, ed un modulo di orchestrazione per la relativa gestione.

Il primo modulo realizza una piattaforma di configurazione per i servizi di sicurezza, richiesti tramite l'interfaccia REST che esso espone verso l'utente. Questa piattaforma permette, tramite un'interfaccia Northbound intent-based, di richiedere con un linguaggio quasi naturale come IB-NEMO, un servizio di sicurezza arricchito delle configurazioni espresse dall'utente. La piattaforma NEMO si limita, dopo aver tradotto gli intenti verso alcuni file di configurazione d, a passarli al modulo di orchestrazione che provvederà ad installarli nella vNSF corrispondente. Questo garantisce la possibilità di poter gestire la configurazione delle vNSF in modo centralizzato e del tutto automatizzato. In questo caso NEMO è stato adattato per gestire le configurazioni riguardanti le VPN, ma con modalità analoghe può essere esteso al fine si supportare un servizio generico.

IL secondo modulo, come anticipato, realizza il comportamento di un orchestratore per cui è incaricato di prelevare il file di configurazione prodotto dal traduttore e successivamente iniettarli all'interno del servizio corrispondente. In questa fase, si è deciso di utilizzare un orchestratore capace di integrare i servizi della piattaforma di configurazione; da ciò ne consegue che il modulo non gestisce il ciclo di vita completo delle vNSF.

6.2 Piattaforma di gestione

Come descritto precedentemente, il blocco funzionale di gestione dei servizi di sicurezza è composto a livello logico da due moduli, uno contenente la piattaforma NEMO e uno l'orchestratore. Andiamo, quindi, ad analizzarne la struttura logica.

6.2.1 NEMO

Si tratta del framework di gestione del ciclo di vita degli intenti, realizzato a partire da una versione di OpenDaylight NEMO. La scelta che ha condotto a questa piattaforma è stata effettuata analizzando e comparando i vari linguaggi di definizione degli intenti i cui usi erano ampiamente documentati. Il requisito che ha veicolato la ricerca coincide con la capacità del linguaggio esaminato di essere esteso con nuove parole chiave e nuovi valori in modo semplice ed auto-esplicativo. In secondo luogo, la scelta di un linguaggio con stile naturale avrebbe incrementato l'usabilità dell'interfaccia utente. A fronte di queste considerazioni, il linguaggio che più di tutti ha aderito ai nostri requisiti è NEMO. La sua realizzazione da parte di OpenDaylight figura tra i paper di IETF e risulta già in corso di standardizzazione.

Internamente, il processo di elaborazione degli intenti attraversa più fasi e più moduli logici, opportunamente estesi o integrati con altri. La scelta, in prima fase, di delegare l'installazione della configurazione ad un componente esterno ha permesso di semplificare la struttura di questo elemento e minimizzare le integrazioni. L'analisi delle funzionalità desiderate ha permesso di separare logicamente due macro-gruppi di moduli da realizzare, in base al servizio che intendono servire: il primo ha come obiettivo il primo processamento degli intenti e il successivo salvataggio dei risultati in una base dati; il secondo si occupa di realizzare la traduzione delle entità salvate in file consistenti che possano essere utilizzati per configurare un servizio di sicurezza. Questa suddivisione costituisce

la base da cui partire per estendere l'architettura di NEMO che, insieme allo studio dell'architettura originaria, ha permesso di giungere all'architettura mostrata in figura 6.2. Il risultato mostrato rende evidente due ulteriori scelte progettuali operate rispetto al gruppo di funzioni incaricate di realizzare le configurazioni, a partire dalle entità messe a disposizione dalla base dati. Un importante guadagno in termini di flessibilità è garantito, infatti, dalla separazione in due blocchi del macro-modulo, di cui uno incaricato di gestire e richiedere le entità opportune e uno deputato alla traduzione. In prospettiva futura, l'utilizzo di due moduli separati permette di trasformare l'architettura minimizzando gli interventi. La seconda scelta operata in questa direzione riguarda la dotazione del modulo di traduzione di un motore in grado di caratterizzare dinamicamente tipi di file differenti; in questo modo, il campo di azione della soluzione non è ristretto al solo caso d'uso, ma può essere adattato per qualsiasi tipo di vNSF, adattando il traduttore con l'opportuno scheletro da utilizzare come base per la configurazione.

Successivamente discuteremo, nel corso della sezione 6.2.2, le differenze tra il progetto iniziale e quello ottenuto come risultato delle scelte operate; di seguito, invece, riportiamo una descrizione dell'architettura della piattaforma NEMO estesa al fine di supportare la configurazione di vNSF.

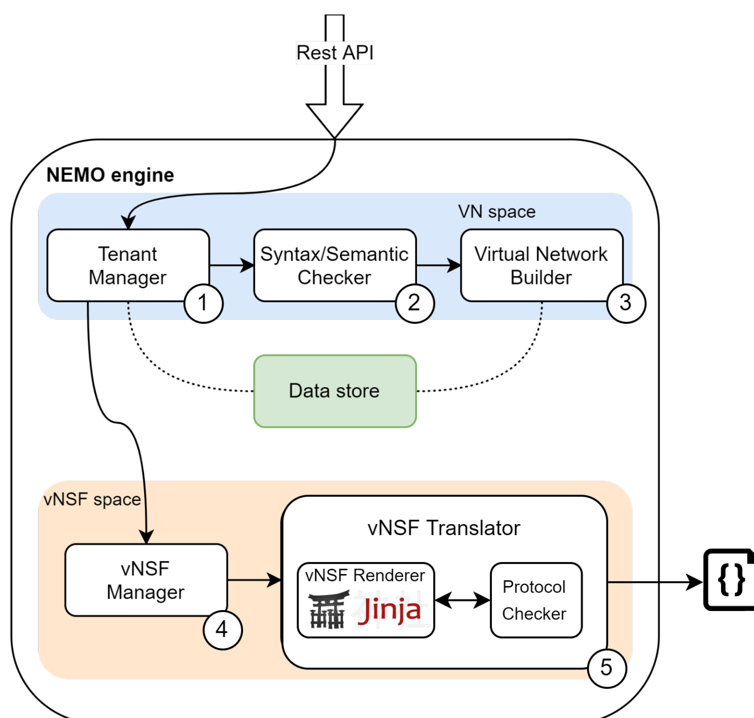


Figura 6.2. Architettura logica di NEMO

Tramite una REST API, che espone tutti i metodi di alto livello per interfacciarsi con la piattaforma e le sue strutture dati, è possibile definire gli intenti in linguaggio *language-style*. I metodi dell'interfaccia vengono gestiti dal modulo *Tenant Manager*, che implementa due componenti. Il primo, si occupa della gestione di una declinazione del protocollo di *Autenticazione, Autorizzazione, Accounting* (AAA) [61], mentre il secondo gestisce, di fatto, le richieste provenienti dall'interfaccia, chiamando il metodo di gestione opportuno in base al contenuto dell'intento. Il protocollo, supportato da un *data store*, modella tutte le informazioni derivate dagli intenti e dal loro processamento sull'entità *utente*. Questo garantisce la proprietà di *multi-tenancy* a livello di intenti, associando

qualsiasi informazione inserita attraverso l'interfaccia all'utente che le ha definite; in particolare, la piattaforma è in grado di gestire più scenari contemporaneamente restituendo configurazioni potenzialmente differenti in base all'utente che le ha richieste.

Il modulo *Syntax/Semantic Checker* fornisce supporto al precedente, processando le richieste autorizzate dal protocollo AAA. A tal fine, vengono integrati due handler in risposta alla tipologia di linguaggio utilizzato, a seconda che si tratti di *language-style* o di *structure-style*. Dopo essere state prese in carico dal gestore corrispondente, questo modulo realizza un controllo sulla correttezza della sintassi e della semantica degli intenti inseriti dall'utente. Utilizzando un parser, NEMO verifica la correttezza di ogni singola parola e la corretta struttura dell'intento in termini di entità definite. In particolare i limiti semantici sono imposti dal linguaggio stesso, che prevede delle precedenze nell'utilizzo di particolari termini, e dalle strutture dati modellate da NEMO, che definiscono le entità e le loro proprietà, la cui caratterizzazione può essere obbligatoria o opzionale. Vedremo, in seguito, un esempio pratico a tal proposito.

Il *Virtual Network Builder* è l'ultimo modulo del cosiddetto *VN space*, dedicato alla gestione della rete virtuale. Infatti, come ampiamente spiegato nella sezione 3.4.3, il framework realizza una rete virtuale modellata in base agli intenti, per poi riportarla in termini di configurazione sulla rete fisica. Questo modulo realizza le entità corrispondenti a quelle definite dagli intenti validati dal blocco precedente, salvandole nel *data store*. Tali entità, tipicamente, sono oggetti già definiti in NEMO con proprietà già esistenti (es. un host ha come proprietà un indirizzo ip), ma è possibile estendere la lista di elementi supportati.

Il modulo *vNSF Manager* viene contattato direttamente attraverso dall'interfaccia REST, dopo che il *Tenant Manager* abbia verificato la legittimità della richiesta. Tramite l'interfaccia, infatti, l'utente può richiedere al framework di configurare un servizio già presente nella topologia della rete virtuale, specificandone il nome. Il *vNSF Manager*, a fronte di tale richiesta, si occupa di creare una struttura dati consistente, raccogliendo tutte le proprietà definite per quel particolare servizio di sicurezza, dopo averne controllata l'effettiva esistenza nello spazio virtuale della rete.

L'ultimo blocco che definiamo in questa visione logica di NEMO coincide con il *vNSF Translator*. Come suggerito dal nome che abbiamo attribuito al componente, si tratta di un modulo di traduzione per file di configurazione basati su template. Questo modulo, che va a colmare le lacune lasciate dall'integrazione incompleta di OpenDaylight NEMO con i protocolli di basso livello, permette di realizzare una configurazione generica, a seconda del template che gli viene fornito, opportunamente completato con le informazioni suggerite dall'entità selezionata. Questo è reso possibile grazie all'integrazione con un modulo di renderizzazione, il *vNSF Renderer*, basato sul motore fornito da *Jinja*[1]. Infine, il modulo di traduzione prevede l'integrazione con un ulteriore servizio di supporto verso l'utente in materia di protocolli supportati. Il modulo che chiamiamo *Protocol Checker* permette di effettuare una ricerca in un file di configurazione, al fine di verificare che i protocolli di sicurezza (es. cifratura o hashing) richiesti non siano fra quelli deprecati. Il file che ci aspettiamo possa uscire dal modulo di traduzione rappresenta una configurazione completa, pronta per essere installata sulla vNSF.

Per quanto riguarda il *data store*, per il momento ci limitiamo a dire che rappresenta la base dati del sistema, organizzata secondo una struttura di tipo *data-tree*, che rende

[1] https://jinja.palletsprojects.com/en/3.0.x/

persistenti le informazioni sulla rete. La sua implementazione, basata su *MD-SAL*[2], sarà discussa brevemente nel capitolo 7.

Ciascuno dei moduli descritti precedentemente sfrutta le caratteristiche di un linguaggio *Object-Oriented* come Java per interfacciarsi con gli altri componenti. Infatti, grazie all'utilizzo di metodi pubblici, ogni modulo è in grado di sfruttare i servizi forniti dagli altri componenti, realizzando la catena di connessione logica illustrata in figura 6.2. Il modulo *Tenant Manager*, inoltre, espone un'interfaccia verso il punto di ingresso delle REST API basata su *remote procedure calls*, che permettono la comunicazione asincrona tra le due parti. Il data store, infine, espone un'interfaccia logica collegata internamente alla struttura dati vera e propria. È possibile così richiedere oggetti salvati nel data-tree senza conoscerne l'architettura e il funzionamento.

Il servizio in sé, prevede quindi un *ingress point* ed un *egress point*: il punto logico di input di NEMO permette di richiederne le funzionalità, attraverso dei metodi specifici esposti, e di scatenare una sequenza di interazioni che portano alla creazione di una configurazione. L'architettura della piattaforma include un *web server*, esposto sulla porta `8181`, abilitando, di fatto, l'interazione attraverso richieste specifiche HTTP. Il secondo punto di interazione con il servizio permette, invece, di instaurare un canale sicuro (es. SSH) ed implementare un server SFTP[3]; in questo modo, l'orchestratore è in grado di prelevare il file di configurazione corretto, mantenendo ben separati i processi, a fronte di una semplificazione in una possibile estensione della soluzione.

6.2.2 Confronto con l'architettura di OpenDaylight NEMO

Come già ampiamente anticipato nelle sezioni 3.4.3 e 6.1.1, la piattaforma OpenDaylight NEMO è stata rivisitata al fine di creare il modulo NEMO nella nostra architettura. Lo scopo di questa sottosezione è quello di mostrarne la struttura per evidenziare le differenze con la versione realizzata in questo lavoro.

A tal proposito la figura 6.3 rappresenta l'architettura definita da OpenDaylight nella documentazione per l'utente [62]. Con particolare riferimento ad un'altra figura riportata in 6.2, definiamo le principali differenze nella progettazione logica della piattaforma.

Dal confronto delle due rappresentazioni, si evince in modo molto chiaro che le modifiche riguardano la parte di interazione e gestione rispetto ai protocolli di basso livello. La sezione logica di OpenDaylight NEMO chiamata *Vn space*, infatti, è stata riutilizzata senza modifiche strutturali consistenti. Questo ci ha permesso di sfruttare la rappresentazione topologica virtuale che questo componente realizza, dopo aver risolto gli intenti che riceve tramite l'interfaccia. Si tratta di una sezione di primaria importanza che già nella sua versione originale dimostra di non avere difetti strutturali che possano influire sul nostro lavoro. Per questo, ad esclusione delle estensioni definite nel dettaglio nella sezione 7.1, è stato riutilizzato secondo la sua struttura originaria.

In aggiunta, è fondamentale, nella piattaforma sviluppata all'interno di questa soluzione, l'utilizzo dell'interfaccia *Northbound* fornita da OpenDaylight. In questo modo è possibile sfruttare le API di tipo *REST* per comunicare direttamente con NEMO: tramite queste richieste è possibile definire la topologia virtuale e richiedere eventuali configurazioni, oltre che configurare i modelli, al fine di estendere la sintassi o aggiungere proprietà alle entità già definite.

[2]Progetto di OpenDaylight che fornisce funzionalità a NEMO

[3]SSH File Transfer Protocol

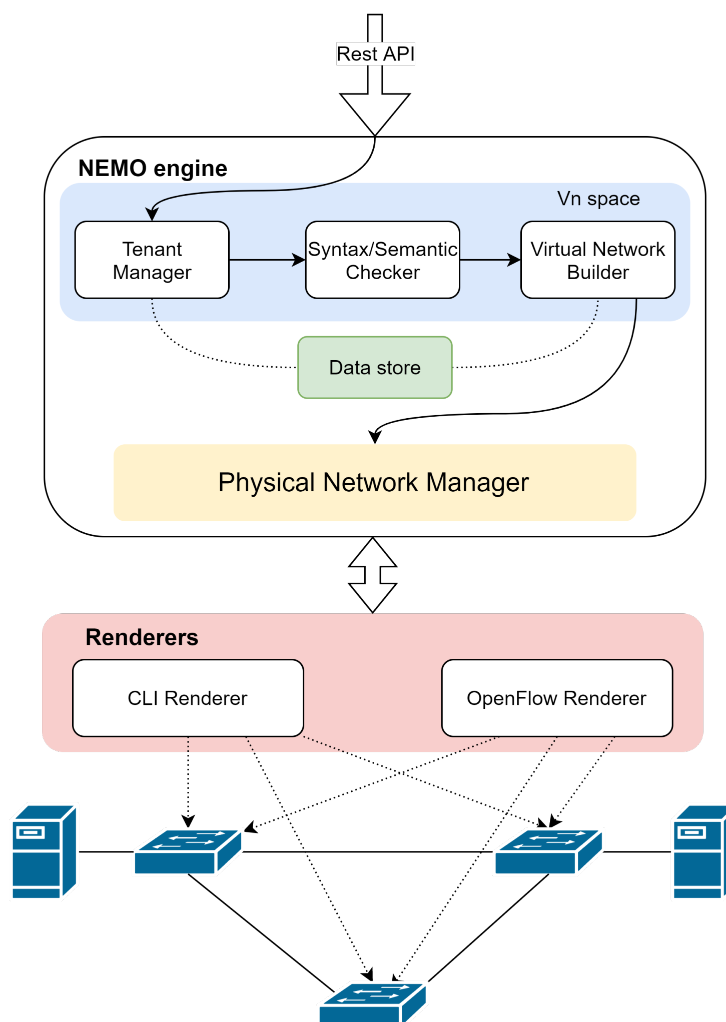


Figura 6.3. Architettura del framework OpenDaylight NEMO

Già nella descrizione del *Physical Network Manager*, riportato in figura 6.3, trattiamo di un componente che non viene integrato nella soluzione proposta. Il compito è quello di gestire la comunicazione con gli apparati fisici e delegare la configurazione al componente corretto. Infatti, uno dei processi più importanti eseguiti dal componente di gestione per la rete fisica prevede la rilevazione della topologia reale sottostante, al fine di configurarla correttamente secondo le indicazioni contenute nel *VN space*. A tal proposito, si occupa di realizzare e mantenere la corrispondenza tra ciascun apparato virtuale con l'analogo fisico.

Il *Physical Network Manager* introduce già i primi problemi, in particolare riguardo la rilevazione della topologia fisica, per i problemi già definiti nella sezione 3.4.3. Nella nostra soluzione, i processi svolti da questo componente sono eseguiti, almeno nella progettazione logica dell'apparato, dall'orchestratore. In particolare, si occupa di mantenere il mapping tra lo spazio virtuale e quello fisica, per facilitare la successiva configurazione, grazie alla sua conoscenza della topologia. Di fatto, questa integrazione nelle funzionalità dell'orchestratore, nonché la progettazione stessa del componente, è influenzata in maniera importante dalle incongruenze introdotte dal modulo originale e dalla necessità di risolverle.

Si ottiene, quindi, dall'orchestratore un duplice funzionamento: è possibile che l'utente

definisca la sua rete virtuale mappata successivamente nella corrispondente topologia fisica, oppure si può ottenere un processo inverso. L'orchestratore stesso si occupa di mantenere nello spazio virtuale una rappresentazione della topologia fisica, in modo da avere un base dati già consistente qualora si volesse configurare uno degli apparati fisici rilevati dall'orchestratore.

Secondo l'architettura originaria, i moduli che si occupano effettivamente di realizzare le configurazioni sugli apparati fisici sono chiamati *Renderers*. In particolare, NEMO ne realizza due a seconda del protocollo *Southbound* utilizzato: il *CLI Renderer*, che esegue una serie di comandi da terminale per configurare i dispositivi, e *OpenFlow Renderer*, per configurare gli switch basati su questa tecnologia SDN.

Anche in questo caso, le funzionalità risultano molto limitate, in particolare nel caso della riga di comando, ma, in generale, queste caratteristiche fornite da ODL-NEMO non sono allineate con lo scopo di questo lavoro. Per questo, ancora una volta, le funzionalità di configurazione dei dispositivi fisici sono state delegate all'orchestratore. Infatti, non rientra tra gli scopi del lavoro quello di configurare dispositivi operanti nel dominio delle reti SDN e l'approccio basato su CLI risulta in disaccordo con il progetto della vNSF.

Per quanto riguarda la prima casistica, l'obiettivo di voler configurare funzioni di sicurezza virtualizzate non impone alcun limite nel caso in cui la rete trattata fosse di tipo SDN; non se ne supporta, tuttavia, la configurazione degli apparati al di fuori di quelli previsti in questa soluzione. Il secondo caso, che prevede l'utilizzo di comandi da terminale, si discosta dal concetto di configurazione di vNSF, che dovrebbe essere quanto più indipendente possibile dalla realizzazione, e conduce, inoltre, ad una gestione dei comandi più complessa, in quanto specifici per ciascuna piattaforma. L'utilizzo di un VNFC univoco permette la migrazione del servizio su qualsiasi piattaforma e rende possibile in qualsiasi caso la sua configurazione.

Chiaramente, i cambiamenti operati sulla versione originale non si limitano a questi, ma per quanto riguarda il design architetturale è sufficiente sottolineare la sostituzione dei moduli *Renderers* e *Physical Network Manager* con un modulo esterno di orchestrazione ed alcuni interni, confinati nel *vNSF space*. Le motivazioni di tali scelte sono state spiegate nella sezione 3.4.3, mentre i dettagli sugli interventi in ciascun modulo sono forniti nel corso del capitolo 7.

6.2.3 Orchestratore

Al fine di fornire un'idea completa dell'architettura realizzata, introduciamo il modulo di orchestrazione, di cui analizzeremo l'implementazione nel capitolo 7. Come abbiamo già precedentemente introdotto, NEMO da solo non è in grado di gestire correttamente una vasta gamma di casi di configurazione, per questo si è reso necessario lo sviluppo di un piccolo componente esterno che aiutasse a calare le configurazioni più a basso livello. In aggiunta l'utilizzo di un orchestratore allinea la soluzione all'architettura standard e ai concetti definiti nel corso del capitolo 4 che rappresentano, in ultima analisi, lo scenario generico su cui realizzare tutti i componenti definiti in questo lavoro.

Il modulo di orchestrazione permette, di installare una configurazione su un'istanza di VNF. Come descritto nella figura 6.4, ciò è permesso da due moduli logici fondamentali: i moduli *Client SSH* e il *Controllore di Configurazione*.

Il modulo *Client SSH* fornisce gli strumenti necessari ad eseguire tutte le funzioni basilari dell'orchestratore. Sfruttando le capacità del protocollo SSH, questo componente si interfaccia con le vNSF e con il framework NEMO, da cui estrae i file di configurazione.

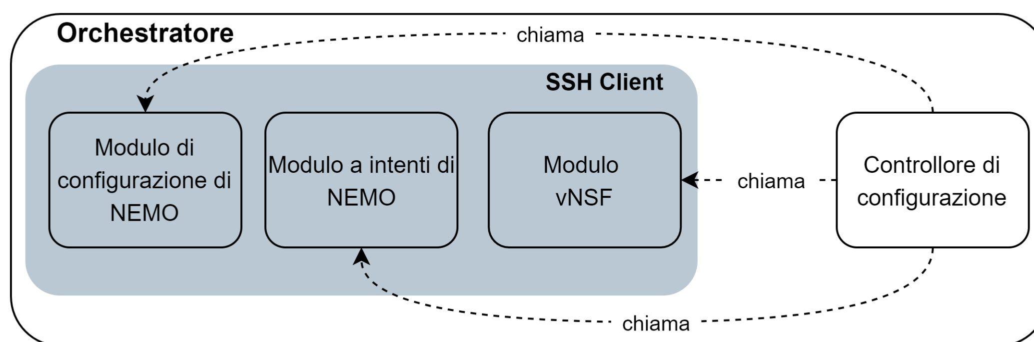


Figura 6.4. Architettura del modulo di orchestrazione

Questo componente realizza le sue funzionalità con l'ausilio di moduli dedicati. Il modulo che chiamiamo *Modulo di configurazione di NEMO* permette la configurazione preliminare delle entità di NEMO, in modo che possa mappare nel suo spazio virtuale tutti gli elementi della topologia desiderata. Il design di questo modulo è stato realizzato in modo da rendere agevole qualsiasi tipo di estensione in termini di topologie supportate; ciò è ottenuto mediante una lista di entità, con le rispettive caratteristiche, che questo modulo inietta nella piattaforma a intenti e che risulta facilmente integrabile.

Il secondo blocco funzionale chiamato *Modulo a intenti di NEMO*, è responsabile di interfacciarsi con la REST API di NEMO. La sua realizzazione rappresenta un punto fondamentale dal momento che realizza un livello di automazione tale per cui è possibile testare l'architettura complessiva, comunicando una serie di intenti alla piattaforma.

Il completamento del ciclo di vita delle configurazioni viene assicurato, infine, dal modulo chiamato *Modulo vNSF* che si relaziona con i servizi di sicurezza. È incaricato della gestione del ciclo di vita del canale SSH attraverso cui veicola comandi sia al fine di ottenere informazioni sullo stato attuale del servizio, sia per modificare il comportamento delle vNSF. La funzionalità principale di questo modulo risiede nella possibilità di prelevare il file di configurazione prodotto da NEMO e installarlo direttamente sulla vNSF indicata. In aggiunta, in questa particolare realizzazione, il Modulo vNSF interagisce con i terminatori incaricati di realizzare il tunnel VPN, offrendo la possibilità di gestire interamente il ciclo di vita dello stesso. Le proprietà di questo modulo si sovrappongono parzialmente a quelle del VNF Manager, in termini di interazione con lo stato della macchina virtuale, ma ricalcano in modo più preciso le funzionalità dell'*Element Manager*, in grado di interagire direttamente con l'applicazione in esecuzione all'interno della VM.

Il componente chiamato *Controllore di Configurazione* realizza il cuore vero e proprio dell'orchestratore. Basandosi sui servizi offerti dal modulo *Client SSH*, esegue una pipeline completa per eseguire la configurazione sul servizio di sicurezza. Anche in questo caso il componente è influenzato dall'implementazione finalizzata a realizzare un flusso di lavoro completo di simulazione, ma le sue funzionalità possono essere astratte e generalizzate. L'idea di base dell'orchestratore è fondata sulla visione completa della rete che immaginiamo un componente di questo tipo possa avere. Partendo da questo concetto, il comportamento che ci si aspetta dall'orchestratore prevede due fasi: la prima in cui il controllore preleva la configurazione corretta, la seconda in cui, dopo aver individuato il target nella rete, la applica.

In conclusione, possiamo dire che i due moduli fondamentali dell'orchestratore, il *Controllore di Configurazione* e il blocco *Client SSH*, espongono metodi che si interfacciano fra di loro. In particolare, il primo modulo utilizza i metodi del secondo come libreria,

per eseguire le operazioni necessarie al completamento della configurazione. Da ciò ne consegue che i moduli del blocco che realizza la libreria non interagiscono fra di loro, ma forniscono le funzionalità che il *Controllore di Configurazione* è incaricato di mettere insieme.

6.3 vNSF

Procediamo con la descrizione della nostra architettura analizzando il modulo *vNSF*. Nel design della soluzione non è stata presa in considerazione la possibilità di sviluppare servizi di sicurezza complessi, basati, ad esempio, sull'utilizzo di più VNF a creare una "Service Function Chain". Per questo motivo, il modulo in questione coincide, di fatto, con la vNSF vera e propria, realizzata su una macchina virtuale. Grazie ad un ipervisore, abbiamo realizzato due macchine virtuali gemelle e installato un software già predisposto che fornisse il servizio di sicurezza. Questa scelta ci ha permesso di poter operare su un box già completo dal punto di vista delle funzionalità, senza occuparci dell'implementazione vera e propria della vNSF, e di concentrarci, quindi, solo sull'automazione della configurazione.

6.3.1 Progettazione e configurazione della vNSF

Come già anticipato, le considerazioni che andremo a fare in questa sezione sono finalizzate a dare una visione più vicina allo standard ETSI della nostra soluzione.

Dal punto di vista progettuale, la vNSF risulta, a livello schematico, essenziale. L'architettura è costituita da un unico modulo VNFC, contenente tutta la parte software utilizzata per realizzare la macchina virtuale e il servizio di sicurezza stesso. Questo include, in particolare, l'immagine del sistema operativo che funge da base per la macchina virtuale, ma soprattutto il software StrongSwan, che si occupa di realizzare il tunnel VPN e il software che gestisce il protocollo di comunicazione sicura.

Questo si traduce, nella nostra soluzione, in vNSFs che realizzano, grazie a questo strumento, dei terminatori VPN. A questo proposito, la scelta di un'architettura "prefabbricata" come quella fornita da una macchina virtuale di questo tipo costituisce un limite, in particolare sulla flessibilità dei moduli software VNFC. Risulta, infatti, molto complesso realizzare una vNSF più elaborata, in cui più moduli elementari si interfacciano tra di loro, dovuto principalmente al fatto che il tutto poggia su un'infrastruttura inadeguata come quella proposta in questa soluzione. Tuttavia, come già anticipato, la scelta effettuata è assolutamente in linea con gli obiettivi del lavoro.

A completare il design architetturale, analizziamo le interfacce realizzate sul servizio di sicurezza. La flessibilità su questo aspetto è decisamente un aspetto meno marginale rispetto ai moduli VNFC, in quanto il tipo di macchina utilizzata permette di definire facilmente interfacce di rete virtualizzate: è possibile, infatti, definire fino ad un massimo di otto interfacce di rete esterne. Quelle realizzate sono tre e vengono utilizzate per gestire la configurazione e permettere il flusso dati: in particolare, l'interfaccia `interface0` è utilizzata come interfaccia di gestione, utilizzata per configurare la vNSF mentre le interfacce `interface1` e `interface2` sono utilizzate come punto di ingresso e punto di uscita del traffico dati. Infatti, come vedremo nel corso del capitolo 7, l'interfaccia di gestione è utilizzata per configurare direttamente i meccanismi di StrongSwan che, coadiuvato dalle informazioni di instradamento, configura la VNF per selezionare il traffico in ingresso e incapsularlo all'interno del tunnel.

Internamente, la semplicità dell'architettura ci suggerisce una corrispondenza univoca tra le interfacce interne, riferite al modulo VNFC, e le interfacce standard della vNSF.

Utilizzando la nomenclatura introdotta nella sezione 4.2.2 l'interfaccia di gestione si associa alla tipologia descritta nello standard ETSI [32] come *SWA-3*, per quanto riguarda le qualità di gestione del ciclo di vita della funzione stessa e, più in generale, di controllo della funzione. Al contrario, le interfacce `interface1` e `interface2`, aderiscono alla definizione di interfaccia *SWA-1*: entrambe si occupano della comunicazione con un servizio di rete differente ed in particolare, in un caso comunicano con una VNF "gemella", in cui è implementata un'istanza dello stesso servizio, in un altro costituiscono il punto di ingresso o il punto di uscita dei dati.

Il software installato nella VNF espone, attraverso l'interfaccia `interface0`, un servizio di gestione di un canale sicuro con cui ricevere i comandi. É possibile, infatti, sfruttare un server SSH cui collegarsi in modo sicuro per iniettare comandi adeguati per gestire la configurazione del tunnel VPN, o il ciclo di vita della funzione stessa. Sfruttando la flessibilità della tecnologia utilizzata, è possibile installare il software di gestione del protocollo TLS/SSL e, esponendo una porta diversa, iniettare comandi tramite *telnet* realizzato su un canale sicuro.

Un altro tema su cui sarebbe bene discutere è rappresentato dalla realizzazione della configurazione del servizio. In continuità con l'approccio utilizzato in questa sezione è interessante portare la nostra soluzione verso quelle soluzioni che, ad oggi, rappresentano lo standard de facto. A riguardo le tecnologie del mondo cloud, ma anche ETSI con OSM, definiscono le configurazioni etichettandole con il termine "*Day-x*" dove la lettera x indica il momento in cui è realizzata. Per rendere più chiara questa terminologia, OSM ad esempio etichetta con *Day-0* la configurazione realizzata al momento dell'istanziazione della VNF e con *Day-1* la configurazione eseguita quando la VNF è già attiva [63]. Nell'esempio della nostra architettura, la trasposizione assume un valore nel caso della configurazione *Day-1*; di fatto il caso della configurazione *Day-0* non è gestito in modo rilevante, lasciando al software di realizzazione del servizio di sicurezza, l'onere di eseguire l'istanza con le configurazioni previste di default. La configurazione *Day-1*, al contrario, è gestita adeguatamente e rappresenta una parte rilevante della soluzione. Una volta che la vNSF è stata istanziata ed è in funzione, gli obiettivi del lavoro dettano la necessità di predisporre un sistema dedicato alla gestione della configurazione, in modo da cambiarne il comportamento mentre questa è in esecuzione. In questo caso, come spiegheremo nel capitolo 7, è stato adottato un meccanismo basato sul protocollo SSH, che provvede a iniettare nuove configurazioni nella vNSF ogni volta che ne viene data indicazione dall'utente.

6.4 Workflow

Nelle precedenti sezioni abbiamo presentato l'architettura di ciascun componente e le scelte progettuali che l'hanno caratterizzata. Nella discussione abbiamo evidenziato l'interazione tra i moduli logici di ciascun blocco; in questa sezione concludiamo con una descrizione del flusso di lavoro completo operato dalla soluzione, dalla definizione dell'intento all'installazione della configurazione.

L'intero processo viene scatenato dall'interazione dell'utente con l'interfaccia che NEMO mette a disposizione, attraverso la descrizione di un intento. Discuteremo in seguito, nel capitolo 7, in che modo viene realizzata di fatto questa interazione. Nella figura 6.5, sono rappresentati due tipi di interazioni possibili con la piattaforma: la prima permette

di creare entità da realizzare nello spazio virtuale di NEMO, la seconda, invece, permette di ottenere il file di configurazione destinato ad una delle entità create precedentemente.

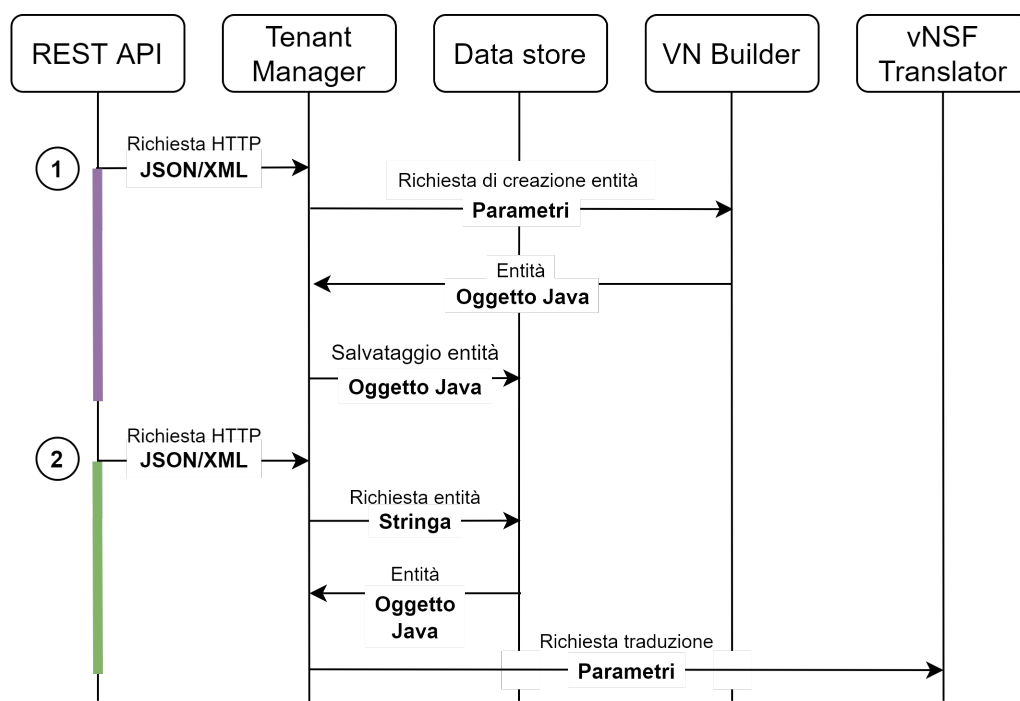


Figura 6.5. Diagramma di sequenza della piattaforma NEMO

In entrambi i casi, l'interfaccia riceve dall'utente una richiesta in formato *XML* o *JSON*, il cui corpo contiene l'intento vero e proprio. In fase di ricezione, il *Tenant Manager* la scompone nelle sue entità fondamentali, in base ai modelli che la descrivono. Per fornire un esempio concreto ci riferiamo ai gruppi di flussi descritti in figura 6.5: nella prima richiesta, un modello YANG suggerisce al modulo di gestione che è composta dall'identificativo dell'utente e dall'intento scritto in linguaggio NEMO, secondo lo stile naturale; nella seconda, un altro modello struttura la richiesta in due parti, contenenti rispettivamente l'identificativo dell'utente che la effettua e il nome dell'entità da tradurre.

In questo passaggio, quindi, il modulo *Tenant Manager* si trova a gestire esclusivamente stringhe o composizioni di esse (es. liste di stringhe), estrapolate dalla richiesta HTTP. Si può notare, inoltre, in modo chiaro il ruolo che svolge questo modulo nel collegamento del *data store* con gli altri componenti.

Nel primo scenario rappresentato nel diagramma di sequenza 6.5, i parametri estrapolati dalla richiesta HTTP vengono inoltrati al modulo *VN Builder* il quale modella un oggetto Java corrispondente all'entità definita nell'intento e lo restituisce al modulo chiamante. Anche in questo caso, il lavoro del *VN Builder* è guidato da un file che definisce la struttura del tipo di entità richiesta, descrivendo ciascuna proprietà cui saranno associati i parametri ricevuti. A questo punto, nell'ultimo passaggio, il *Tenant Manager* invia l'oggetto rappresentante l'entità verso il *data store*, per salvarlo nella base dati.

Nel secondo scenario riportato, l'interazione con la base dati avviene sempre ad opera del modulo *Tenant Manager*, ma la richiesta è differente: è rappresentato, infatti, il contesto in cui l'utente richiede la traduzione di un'entità già presente nello spazio virtuale, per ottenere il corrispondente file di configurazione. In questo caso, è sufficiente richiedere al *data store* l'oggetto specificato dall'utente in base al nome univoco; dopo averlo

ottenuto, sarà compito del *vNSF Translator* generare il file corretto e salvarlo nel file system della piattaforma. Per mantenere la suddivisione logica tra i moduli e lasciare al *Tenant Manager* l'esclusività nel relazionarsi con il *data store*, non viene inviato il singolo oggetto al traduttore, bensì svariati parametri nei tipi primari (es. stringhe), costruendo un processo inverso a quello precedente. Nel primo scenario, infatti, dai parametri estratti dalla richiesta si cercava di costrurire un oggetto strutturato, mentre in questo caso, a partire dall'istanza della classe Java corrispondente, si estrapolano gli oggetti che essa contiene. A questo punto, partendo da tali parametri e seguendo un processo che sarà definito nel corso del capitolo 7, viene generato un file di configurazione per l'entità richiesta.

Non discutiamo un possibile flusso nel caso in cui la richiesta in ingresso richiedesse delle modifiche a entità già precedentemente definite; l'interazione tra i moduli,infatti, non subisce variazioni in quanto è il *data store* a farsi carico della gestione delle modifiche alle entità, in modo del tutto trasparente.

A questo punto del flusso il modulo di orchestrazione interviene per completare la configurazione della vNSF e realizzare il comportamento espresso dall'utente tramite intenti. La figura 6.6 mostra la sequenza di interazioni scatenata al completamento della traduzione. In questo passaggio, si evince il lavoro prezioso dell'orchestratore, che mette insieme la logica di gestione degli intenti con il servizio da configurare. A seguito della richiesta inoltrata dal *Controllore di Configurazione*, che invia l'identificativo corrispondente al servizio da configurare, interviene il *Modulo vNSF*. Tramite l'identificativo univoco, questo modulo è in grado di individuare il file di configurazione corrispondente a quell'entità e prelevarlo direttamente dal file system della piattaforma NEMO. A questo punto, collegandosi all'interfaccia di gestione della vNSF presentata nella sezione 6.3.1, l'orchestratore invia la configurazione nella macchina virtuale.

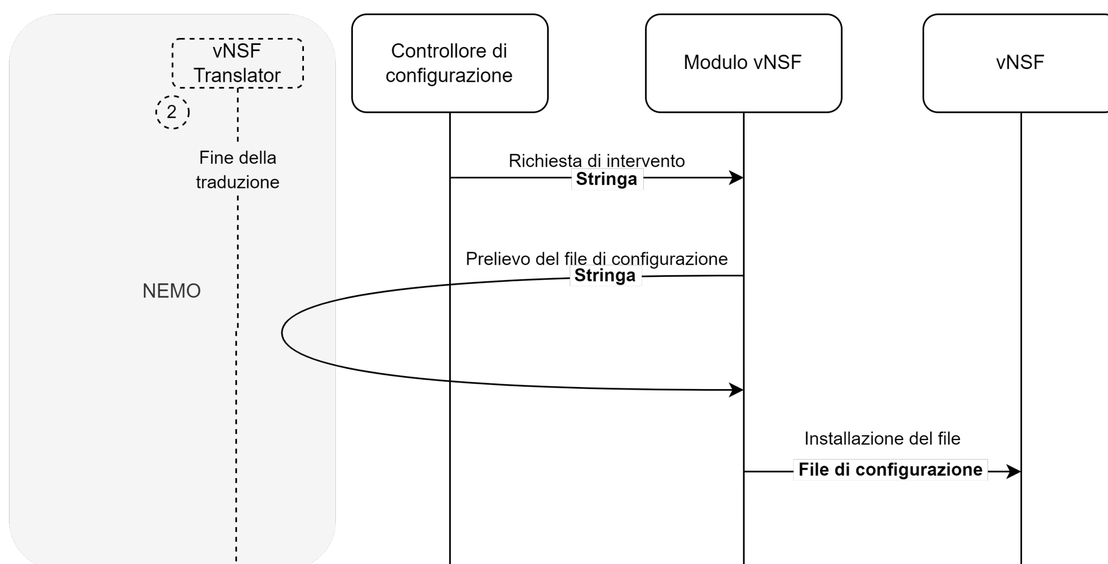


Figura 6.6. Diagramma di sequenza per l'installazione del file di configurazione

L'immagine eseguita nella macchina virtuale, che implementa le funzionalità della vNSF, in modo completamente automatico attinge dalle informazioni appena caricate dall'orchestratore per cambiare il proprio comportamento. Infatti, il software legge il file di configurazione, che contiene un elenco di caratteristiche desiderate dal servizio, e provvede ad auto-configurarsi per soddisfare quei requisiti. L'iniezione di file di configurazione corretti ed adeguati è sufficiente ad ottenere un servizio funzionante. È bene precisare,

inoltre, che tale meccanismo è reso possibile dal software utilizzato per implementare la vNSF, ma è possibile estendere il supporto a molti altri software che utilizzano, come approccio comune di configurazione, file di configurazione esterni. La semplicità di riconfigurazione ottenuta si riflette sulle caratteristiche dell'orchestratore: infatti, pur svolgendo un lavoro fondamentale, è stato possibile mantenere la sua architettura essenziale.

Capitolo 7

Implementazione

Lo scopo di questo capitolo è quello di caratterizzare dal punto di vista implementativo le architetture delineate fino ad adesso. La figura 7.1 ricalca ad alto livello l'architettura definita nella sezione 6.1.1 e ne evidenzia gli aspetti che sono stati oggetto di rivisitazione o completamente realizzati.

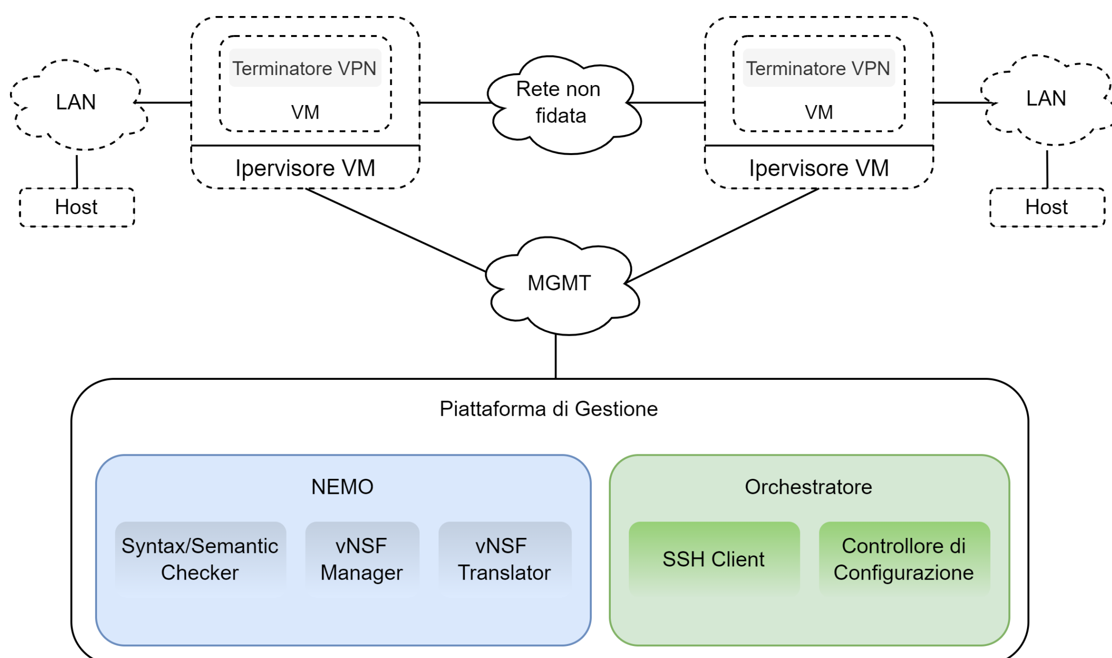


Figura 7.1. Evidenza dei moduli realizzati

Si evince, quindi, che la realizzazione dei terminatori VPN, attraverso lo sviluppo di macchine virtuali, non ha richiesto interventi consistenti. Al contrario, la piattaforma NEMO e l'Orchestratore costituiscono il cuore del processo implementativo. Il primo è stato sviluppato, in questo lavoro, come servizio all'interno di un container Docker[1]. Come abbiamo spiegato brevemente nella sezione 4.2.3, il concetto di "virtualizzazione leggera" ci ha permesso di alleggerire notevolmente il carico di lavoro, ma, cosa molto importante, ci permette di ottenere una piattaforma estremamente portatile. Grazie alla piattaforma fornita da Docker è possibile ricreare lo stesso servizio in qualsiasi ambiente

[1] https://www.docker.com/

di esecuzione. Infatti, al contrario di una semplice macchina virtuale, che ha bisogno di un ipervisore che esegua un sistema operativo completo, un container permette di emulare esclusivamente l'ambiente di esecuzione del singolo servizio, grazie all'utilizzo di alcune librerie. Semplificando, otteniamo una versione estremamente ridotta di una macchina virtuale, in cui è eseguita al sola applicazione designata, risparmiando così molte risorse. Il container è realizzato a partire da un file accettato in ingresso da Docker contenente la descrizione delle caratteristiche desiderate; questo viene utilizzato per definire le interfacce da esporre, ma anche per arricchire la versione base di NEMO con alcune librerie aggiuntive.

L'Orchestratore, invece, viene eseguito direttamente nell'ambiente di esecuzione della Piattaforma di Gestione, senza alcun tipo di virtualizzazione.

Nelle prossime sezioni illustreremo gli interventi che si sono resi necessari sulla piattaforma NEMO e per la realizzazione in toto dell'Orchestratore.

7.1 Realizzazione del framework NEMO

Nel corso del capitolo 6 abbiamo definito l'architettura complessiva e le interazioni tra i vari moduli. In particolare è stato illustrato il ruolo della piattaforma NEMO e cosa si vuole ottenere dal suo processo di esecuzione. Definiamo, a questo punto, gli obiettivi del processo implementativo al fine di strutturare una serie di interventi, necessari per ottenere il comportamento desiderato. Fornendo un'interfaccia utente in grado di definire gli intenti, uno dei primi obiettivi coincide con l'adattamento del linguaggio utilizzato in modo da supportare le proprietà relative ad una vNSF. In particolare il linguaggio deve fornire una sintassi in grado di definire un tunnel VPN. Il secondo scopo coincide con l'integrazione delle funzionalità fornite da NEMO con le proprietà definite nella sezione 6.2.1 in corrispondenza dei moduli *vNSF Manager* e *vNSF Translator*. L'obiettivo è quello di modificare l'output della piattaforma.

Gli interventi principali che hanno permesso di ottenere le caratteristiche desiderate e raggiungere gli obiettivi prestabiliti possono essere suddivisi in due gruppi principali:

- estensione della sintassi di IB-NEMO;
- realizzazione del modulo di traduzione.

7.1.1 Estensione della sintassi

L'estensione della sintassi di NEMO costituisce un passo fondamentale nella costruzione della soluzione, al fine di definire intenti più completi. L'idea di base si sviluppa a partire dai concetti e dalle metodologie enunciati nella sezione 3.4.3, in cui abbiamo descritto le proprietà di NEMO cui facciamo riferimento in questa discussione. Le integrazioni con cui ampliare il linguaggio IB-NEMO possono essere raggruppate in due tipologie:

- estensione in termini di oggetti supportati;
- estensione delle parole chiave utilizzabili negli intenti.

In questa discussione approfondiremo entrambe le tipologie, riferendoci in particolare allo scenario desiderato; tuttavia, è bene precisare che questi procedimenti possono essere generalizzati a qualsiasi topologia.

Il primo passo è finalizzato all'estensione delle tipologie di connessioni virtuali rappresentabili in NEMO, per mapparle con un tunnel VPN. Per permettere questo tipo di estensioni, NEMO espone un metodo attraverso cui accetta in ingresso un file; in questo modo, si fornisce una lista di caratteristiche dell'oggetto che verrà successivamente caratterizzata con i dati specificati dall'utente. Come mostrato in figura 3.7, questo sarà utilizzato come modello per una struttura dati in cui salvare le proprietà definite dall'utente. Partendo dalla struttura già nota di uno dei tipi di *Object* definiti in figura 3.6, NEMO crea una nuova classe Java a cui aggiunge le proprietà definite nel file. Questo è un aspetto chiave per quanto concerne l'estensione della sintassi: come si può osservare nelle figure 7.2 e 7.3, in questa soluzione si è aggiunto un nuovo *Object* di tipo "connessione" e uno di tipo "nodo" di cui NEMO è in grado di creare una classe. In particolare sono state specificate le proprietà di un router, come ad esempio un indirizzo ip privato ed uno pubblico e la descrizione di una nuova connessione chiamata *tunnel_ipsec*. Risulta molto più complessa la definizione di un oggetto di un tipo sconosciuto, come vedremo in seguito; questo significa che non è possibile, ad esempio, definire semplicemente nello stesso file JSON la struttura e le proprietà di un oggetto di tipo "security service" e utilizzarlo nella sintassi.

In riferimento alla sintassi definita nel corso della sezione 3.4.2, mostriamo di seguito un possibile intento realizzabile con le modifiche introdotte finora.

Listing 7.1. Creazione di un oggetto virtuale di tipo router

```
CREATE Node r1 Type router Property public-ip-address:10.0.0.1;
```

Questa integrazione è sufficiente a permettere la definizione di due router collegati con un tunnel IPsec, con intenti simili a quello mostrato nel listato 7.1. Tuttavia per definire completamente un tunnel VPN di tipo *site-to-site* è necessario definire un'*Access Control List* che determini quale traffico vada protetto. In questo caso il processo di estensione della sintassi è più complesso: mentre nel primo caso si tratta di definire le caratteristiche di un oggetto già noto nella struttura, in questo caso è richiesta la gestione di una nuova *keyword*. In questo senso abbiamo definito due parole chiave, *"ACL"* e *"to"* da poter inserire all'interno di un intento, per permettere di definire una lista di indirizzi che identifichino le tipologie di traffico da instradare nella VPN.

L'integrazione in termini di parole chiave supportate viene realizzata nel *parser* (realizzato a partire da *JavaCC* [64]), inserito nel modulo *Syntax/Semantic Checker* come descritto nella sezione 6.2.1. Come primo passo, è necessario inserire le nuove parole nella lista utilizzata dal parser, associando a ciascuna un identificativo. Questo valore sarà utilizzato, successivamente, in fase di esaminazione della parola: se viene incontrato un valore noto allora ne consegue che si tratta di una parola conosciuta dal parser e che quindi si può procedere alla sua gestione. Infatti, seguendo una sequenza finita di confronti in cui si determina lo stato lessicale corrente, il parser cerca una corrispondenza tra il token incontrato nella sequenza di input e i valori definiti dalla sua grammatica, estesa precedentemente con delle nuove parole chiave. Il confronto avviene carattere per carattere: se viene trovata una corrispondenza, viene analizzata la lettera successiva, tenendo comunque traccia dei caratteri analizzati fino a quel momento. In questo modo, per ogni lettera letta si aprono potenzialmente due strade: la sequenza già letta in aggiunta alla lettera corrente individua una parola, interrompendo l'analisi, oppure la sequenza letta individua parte di una parola e per questo si continua con il processo. Per questo motivo,

```
1  data={
2    "node-definitions": {
3      "node-definition": [
4      {
5        "node-type": "router",
6        "property-definition": [
7          {
8          "property-name":
               "ip-address",
9          "property-value-type":
               "string"
10         },
11         {
12         "property-name":
               "public-ip-address",
13         "property-value-type":
               "string"
14         }
15       ]
16     }
17     ]
18 }
```

Figura 7.2. JSON che definisce il nuovo nodo

```
1  data={
2    "connection-definitions": {
3      "connection-definition": [
4      {
5       "connection-type": "tunnel_ipsec",
6         "property-definition": [
7         {
8             "property-name": "auth-method",
9             "property-value-type": "string"
10        },
11        {
12            "property-name":
                  "ike-hash-protocol",
13            "property-value-type": "string"
14        },
15        {
16            "property-name":
                  "ike-enc-protocol",
17            "property-value-type": "string"
18        },
19        {
20            "property-name":
                  "esp-hash-protocol",
21            "property-value-type": "string"
22        },
23        {
24            "property-name":
                  "esp-enc-protocol",
25            "property-value-type": "string"
26        },
27        {
28            "property-name": "ipsec-mode",
29            "property-value-type": "string"
30        },
31        {
32            "property-name":
                  "certificate-name1",
33            "property-value-type": "string"
34        },
35        {
36            "property-name":
                  "certificate-name2",
37            "property-value-type": "string"
38        }
39        ]
40     }
41     ]
42 }
43 }
```

Figura 7.3. JSON che definisce la nuova connessione

dopo aver definito le parole aggiuntive è fondamentale integrare il parser per gestire opportunamente la loro ricorrenza e verificare che rispettino l'ordine prestabilito all'interno dell'intento.

Il nuovo processo iterativo analizza tutte le parole a partire dal momento in cui incontra nell'intento la definizione della connessione di tipo *tunnel_ipsec*. La nuova sintassi desiderata prevede che la parola chiave "*ACL*" sia seguita da un indirizzo di rete sorgente,

a cui fanno seguito la keyword "*to*" e l'indirizzo destinazione. Inoltre si vuole consentire l'inserimento di più reti sorgente e destinazione, ciascuna separata da una virgola.

Descriviamo il processo definito poc'anzi nel listato 7.2, in cui viene riportata l'integrazione al modulo *Syntax/Semantic Checker* in pseudocodice. Una funzione apposita che restituisce la parola successiva letta in input dà il via al processo iterativo che analizza le possibilità, secondo la sintassi che è stata stabilita. La prima parola richiesta per definire gli indirizzi delle reti private è "ACL", come spiegato precedentemente; in qualsiasi passaggio, se non viene rispettata la successione delle parole stabilita, il processo viene interrotto con un errore. A questo punto viene letto l'IP sorgente atteso dopo il primo token e viene letta la parola successiva. Lo stesso procedimento viene seguito per leggere il secondo indirizzo. Se gli indirizzi inseriti sono molteplici, è necessario che ciascuna sequenza sia separata dalla virgola e che gli indirizzi sorgente e destinazione siano sempre separati dal token `TO`. La lista di indirizzi si blocca quando viene trovato il punto e virgola; a quel punto viene invocato il modulo *VN Builder* che crea la connessione virtuale definita tramite l'intento, a cui viene passato tutto il risultato del processo.

Listing 7.2. Processo di verifica della sintassi

```
1  leggi la prossima parola {
2     IF parola == ACL AND connessione == tunnel ipsec THEN
3
4         leggi IP sorgente
5         leggi la prossima parola
6         IF parola == TO THEN
7             leggi IP destinazione
8
9         leggi la prossima parola
10        IF parola == ; THEN
11            termina il processo
12         ELSE IF parola == , THEN
13            ricomincia dall'IP sorgente
14
15 }
```

Al fine di riassumere le modifiche descritte finora e per chiarire la sintassi ottenuta, riportiamo di seguito l'esempio di un possibile intento per creare una VPN site-to-site con autenticazione basata su certificati, i cui nodi terminali sono chiamati *r1* ed *r2* e le reti da collegare sono esplicitamente dichiarate grazie alle nuove parole chiave:

Listing 7.3. Nuova sintassi

```
CREATE Connection vpn1 Type tunnel_ipsec Endnodes r1,r2 Property auth-method:
certificate, ipsec-mode: tunnel ACL 192.168.56.0/24 to 192.168.57.0/24;
```

Le modifiche introdotte finora consentono di estendere la sintassi di NEMO al fine di supportare le entità che permettono di realizzare la nostra soluzione. Il parser, a questo punto, è in grado di processare correttamente un intento definito con la semantica appena descritta e mostrata nell'esempio precedente. Tuttavia, come abbiamo già discusso

nel precedente capitolo, NEMO realizza una topologia di rete virtualizzata, in cui ogni apparato contiene una descrizione dettagliata delle proprie caratteristiche, che viene salvata nel data store. A tal proposito, l'intervento finale che ci ha permesso di ottenere un processo funzionate, dall'inserimento dell'intento fino al salvataggio delle entità estratte, consiste nell'introdurre nuove classi tramite il componente MD-SAL [65].

Questo componente permette a NEMO di definire lo scheletro di una classe o di un'interfaccia che viene successivamente utilizzato, al tempo di compilazione, come base per creare in modo automatico una nuova classe. Per chiarire meglio il prossimo passo, facciamo riferimento al processo di estensione delle entità supportate, descritto all'inizio di questa sezione. Nella prima fase, infatti, abbiamo specificato come la struttura di un oggetto di tipo "connessione" è già nota alla piattaforma; risulta quindi facile definirne varie realizzazioni, basate sullo scheletro principale. In questo caso, l'oggetto ACL non è associabile a nessuno degli *Object* definiti dal linguaggio IB-NEMO. A tal fine, grazie ad MD-SAL, è possibile definire un modello YANG che rappresenti l'oggetto *ACL* che costituirà lo schema della nuova classe. In questo caso, gli unici campi che saranno rappresentati nella classe coincidono con gli indirizzi IP sorgente e destinazione.

7.1.2 Modulo di traduzione

Procediamo ora con la descrizione dei passaggi che hanno reso possibile la realizzazione completa del modulo di traduzione. Secondo il design della piattaforma NEMO (sezione 6.2.1), questo componente entra in gioco quando l'utente esprime la volontà di ottenere i file di configurazione e, pertanto, la base dati è già in uno stato consistente e contiene tutte le entità definite. Il modulo *vNSF Translator* si occupa della traduzione della topologia virtuale di NEMO in file di configurazione da applicare alle vNSF, coadiuvato dal *vNSF Manager*.

In figura 7.4 riportiamo il flusso completo di scambio dati tra i vari moduli impegnati nel processo di traduzione; il processo inizia nel momento in cui l'utente esprime la volontà di ottenere i file di configurazione e finisce quando il file di configurazione viene generato. Non potendo comunicare direttamente con la base dati, il *vNSF Manager* invoca i metodi del *Tenant Manager*, specificando il nome della connessione che si vuole realizzare e i terminatori del tunnel al fine di ottenere le strutture dati necessarie per costruire la configurazione. Questo modulo inoltra la richiesta verso la base dati e appena ricevuto il dato, lo invia al modulo chiamante. A questo punto il *vNSF Manager* avvia un processo iterativo in cui, ad ogni passo, si verifica l'effettiva presenza delle informazioni nel dato strutturato ricevuto. Dopo aver controllato l'identità dell'utente, si verifica la presenza della connessione, che rappresenta il tunnel vero e proprio, che i nodi tra i quali è definita siano presenti e che siano configurati correttamente, che sia stato definito il traffico da indirizzare nel tunnel e a quel punto si raggruppano le caratteristiche di ciascuna entità. Vengono estratti gli indirizzi IP assegnati sui terminatori VPN e tutte le proprietà che sono state definite riguardo il tunnel stesso, comprese quelle sull'ACL. Queste informazioni ricevute dal *vNSF Manager* vengono inviate al modulo *vNSF Translator* che ne eseguirà l'effettiva traduzione. Come sovente si usa nel linguaggio Java, le informazioni vengono inserite all'interno del modulo destinazione tramite i cosiddetti metodi *setters*, che permettono di copiare oggetti nello spazio dedicato ad una classe differente.

In aggiunta, durante l'esecuzione dei metodi di copia delle informazioni riguardanti i protocolli di hashing e cifratura, nel traduttore viene invocato il modulo *Protocol Checker*: questo permette di verificare se sono validi, leggendo da file la lista di protocolli deprecati

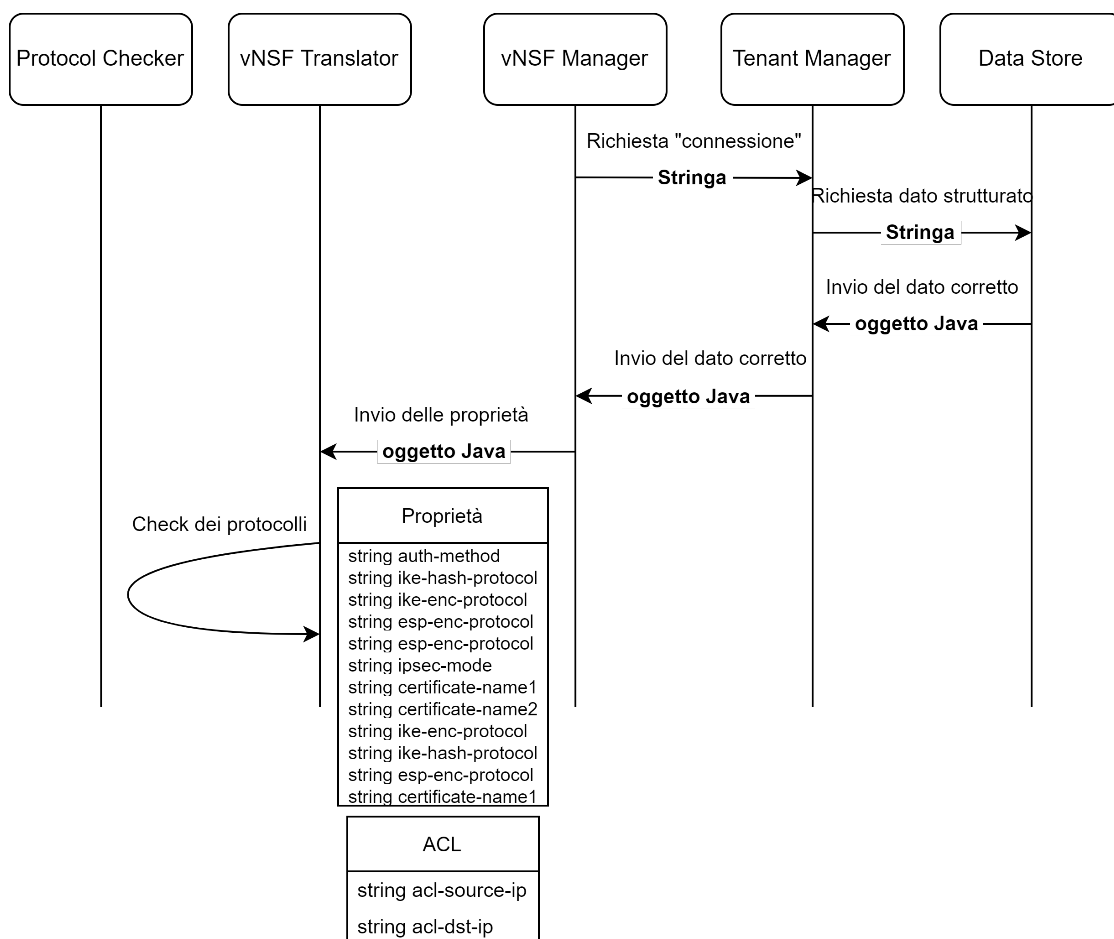


Figura 7.4. Gestione delle informazioni tra i moduli

indicati da StrongSwan,. In questo modo è possibile estendere facilmente la lista, facendo in ogni caso riferimento alle direttive di strongSwan [2].

Una volta ottenute e settate tutte le impostazioni nel modulo *vNSF Translator*, inizia il processo vero e proprio che vede come protagonista il tool Jinja. Il file di configurazione da cui StrongSwan attinge le informazioni per auto-configurarsi segue una struttura ben precisa e sempre uguale nello scheletro, per cui abbiamo optato per l'integrazione con un motore per template come Jinja. A tal proposito è bene specificare che i relativi concetti teorici sono stati introdotti nella sezione 3.1.1, cui rimandiamo per ogni approfondimento.

Il primo passo che abbiamo quindi affrontato è stata la definizione dello scheletro da utilizzare, in modo che possa avere validità in qualsiasi situazione che la piattaforma intenda supportare. Questo approccio garantisce un vantaggio in termini di flessibilità qualora si decidesse di estendere il modulo verso il supporto di una gamma di funzioni di rete più ampia, che richiederebbe la definizione di un nuovo scheletro. Il traduttore prepara un contesto, all'interno del quale inserisce delle coppie chiave-valore, in cui la chiave indica il punto all'interno del template in cui deve essere inserito il valore corrispondente. La struttura è definita da strongSwan, in base ai parametri di cui necessita per definire una connessione VPN, già discussi nel corso della sezione 5.5.

[2] https://wiki.strongswan.org/projects/strongswan/wiki/IKEv2CipherSuites

A questo punto, il modulo *vNSF Renderer*, che esegue una trasposizione di Jinja su ambiente Java chiamata appunto Jinjava, legge da file il template andandone a sostituire ogni chiave con cui è stata trovata una corrispondenza nel contesto, con l'analogo valore. Al termine di questo processo, si avrà in uscita un file completo in cui, al posto delle stringhe circoscritte dalle parentesi, si leggeranno i valori definiti dall'utente. Trattandosi di un tunnel VPN, è necessario iterare una seconda volta per ottenere anche la configurazione del secondo terminatore. Se il modulo *vNSF Renderer* percepisce l'assenza di uno dei parametri all'interno del contesto, ove possibile inserisce i valori predefiniti.

7.2 Orchestratore

Il modulo *Orchestratore* funge da collegamento tra NEMO e le vNSF, installando le configurazioni generate dalla piattaforma direttamente su strongSwan. Inoltre il comportamento desiderato e realizzato non coincide con quello di un orchestratore tradizionale. Tuttavia i metodi implementati sono riconducibili a molti scenari diversi, guadagnando in modo consistente in generalità. In riferimento alla figura 7.1 procediamo con la descrizione dell'implementazione di questo componente che consta di due blocchi, uno centrale che realizza un possibile funzionamento nello scenario creato, e uno secondario che agisce da libreria, contenente metodi generici per operare con target come quelli utilizzati nella soluzione. Quest'ultimo implementa i metodi sfruttando le funzionalità della libreria esterna *Paramiko*, che permette di gestire completamente il protocollo SSH [66]. Si tratta, infatti, di una libreria di basso livello che offre metodi per gestire tutti gli aspetti del protocollo lato client: si parte dai metodi più ovvi attraverso i quali aprire una connessione SSH per poi passare a classi più raffinate, che permettono ad esempio di gestire il canale a livello di trasporto dati, ma anche lo scambio e la gestione delle chiavi di autenticazione. D'altro canto, se da un lato identifica una delle librerie SSH più conosciute e quindi più supportate dalla comunità (documentazione aggiornata, test di sicurezza ripetuti, esempi d'uso), dall'altro bisogna considerare il suo lavoro a basso livello, che in alcuni utilizzi costringe l'utente a maneggiare molti dettagli.

Come descritto nella sezione 6.2.3 la libreria si può suddividere in tre moduli logici, a seconda dei target dei metodi che implementa (Figura 6.4):

- *Modulo di configurazione di NEMO*: nel primo modulo abbiamo realizzato tutti i metodi di configurazione di NEMO. Sfruttando l'interfaccia RESTCONF esposta sulla porta 8181, questi metodi permettono di modellare, attraverso file JSON, le entità che la piattaforma utilizza per realizzare la topologia virtuale. Completando l'argomentazione della sezione 7.1.1, questi file contengono la descrizione dettagliata delle caratteristiche degli oggetti: ad esempio definiscono quali sono le tipologie di connessione(figura 7.3) o di nodi che sono consentiti con le annesse proprietà. Questo si traduce, nella nostra soluzione, in un'importante flessibilità dal punto di vista della sintassi, in quanto è sufficiente modificare il file JSON fornito a NEMO per aggiungere caratteristiche alla VPN.

- *Modulo a intenti di NEMO*: il secondo modulo della libreria interagisce ancora con l'interfaccia RESTCONF, ma sfruttando i metodi della sezione "*operations*" che definiscono gli strumenti per influenzare il meccanismo di NEMO. A tal proposito, sono implementati tutti i metodi che permettono di registrare un utente, definire un nuovo intento e richiedere una configurazione. In aggiunta, rispettando le peculiarità di MD-SAL, sono aggiunti due metodi di apertura e chiusura di una

transazione: tutte le operazioni definite all'interno della stessa transazione vengono gestite contemporaneamente alla sua chiusura, pertanto non è possibile far riferimento ad un oggetto nella base dati definito all'interno della transazione stessa. Tutti questi metodi richiedono, come parametri, l'identificativo dell'utente e l'indirizzo dell'interfaccia cui collegarsi per eseguire una *post* ed inviare i dati.

- *Modulo vNSF*: L'ultimo modulo logico della libreria, invece, rappresenta il servizio che si interfaccia con le vNSF. Tale interazione è basata interamente sulle potenzialità del protocollo SSH e pertanto viene invocata la dipendenza esterna di *Paramiko*. I metodi realizzati sono molto generici e permettono di realizzare svariati scenari, compreso quello delineato nella nostra soluzione. Banalmente è possibile invocare le funzioni di apertura e chiusura di una connessione SSH, prelevare un oggetto dal file system della macchina o eseguire comandi su un host remoto. Quest'ultimo in particolare è sfruttato al fine di interagire direttamente con il demone di strongSwan, per avviare o sospendere il processo di IPsec, oppure per iniziare o terminare una determinata VPN, oltre che ottenere informazioni sullo status attuale. I comandi più specifici relativi alla nostra soluzione riguardano la gestione dei file di configurazione di StrongSwan: a tal proposito si suppone che l'orchestratore sappia la locazione del file system in cui prelevare le configurazioni, sulla piattaforma NEMO, e sappia inoltre dove depositarle. A riguardo ci viene in aiuto il design di StrongSwan che definisce locazioni ben definite da cui il demone attinge per configurare il processo IPsec.

Per quanto riguarda NEMO sono state effettuate delle scelte riguardo la locazione dei file, riportata anche nei metodi dell'orchestratore, mentre, come si può vedere dal precedente esempio, la parte di file system dedicata a StrongSwan è strutturata in maniera statica. I metodi implementati permettono, ad esempio, di leggere il file `ipsec.conf` o sostituirlo con quello prodotto da NEMO, creando di fatto una nuova VPN o modificando il comportamento di quella attualmente in uso. Inoltre le rispettive cartelle contengono, eventualmente, il certificato che il terminatore VPN vuole utilizzare, la sua chiave privata, e il certificato di una serie di autorità ritenute fidate, per legittimare il certificato dell'altra parte. Sempre relativo alla gestione dell'autenticazione a chiave pubblica, abbiamo implementato un metodo per inserire il nome del file contenente la chiave privata all'interno di `ipsec.secrets`, come richiesto da StrongSwan. Chiariamo fin dal principio, che la gestione dei certificati ha un fine puramente illustrativo; risulta infatti complesso gestire certificati reali in uno scenario simile. Chiariremo questo concetto nel corso del capitolo successivo.

Il secondo blocco, che costituisce il controllore del processo di orchestrazione, provvede a realizzare, tramite una sequenza di funzioni fornite dalla libreria, una simulazione completa. Discuteremo della simulazione e dei risultati da essa ottenuti nel capitolo 8.

7.3 Schema

In conclusione, può essere utile analizzare lo schema architetturale finale della soluzione. A tal proposito, la figura 7.5 mostra chiaramente lo scenario ottenuto: in questo modo è possibile chiarire ulteriormente il grado di flessibilità ottenuto nel lavoro.

Partendo dal basso, è possibile trovare la piattaforma incaricata di svolgere le funzioni di controllo e configurazione, per mezzo dell'orchestratore e del framework NEMO. L'orchestratore si occupa della mediazione tra i servizi offerti da NEMO stesso e le vNSFs;

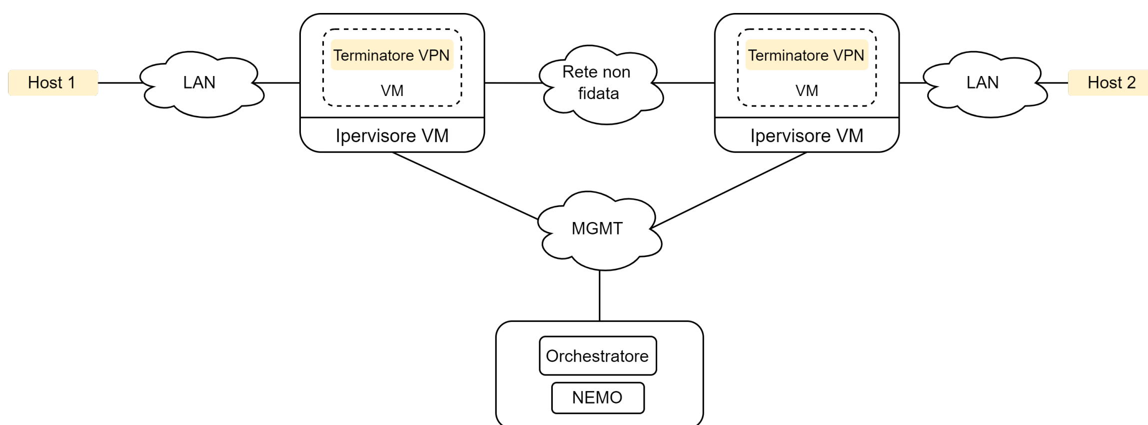


Figura 7.5. Schema della soluzione

esso coincide con il mittente e il destinatario dell'intero traffico scambiato sulla rete di gestione (*MGMT*). Già questa suddivisione ci permette di effettuare un confronto con l'architettura descritta nella sezione 6.1.1: ricordando che NEMO è realizzato come un servizio esterno, risulta immediata la sostituzione del nostro orchestratore con una piattaforma più completa come OSM, integrato, eventualmente, con la piattaforma NEMO eseguita, ad esempio, in un container. OSM svolgerebbe le sue funzionalità di orchestrazione attraverso la rete di gestione e comunicherebbe attraverso le API con il servizio di NEMO, rendendo, di fatto, l'intervento poco invasivo.

Questo è coadiuvato, certamente, dalla realizzazione delle vNSFs come macchine virtuali ben strutturate. Mantenendo invariato il software interno e, soprattutto, la gestione delle interfacce di rete, ci si slega da qualsiasi vincolo sulla collocazione e sul dispiegamento dei servizi stessi. Anche in un'architettura più complessa, l'utilizzo di due interfacce dati e una di gestione permette di installare la macchina virtuale in modo agevole. Questo perché il metodo di configurazione adottato, basato sul protocollo SSH, necessita solamente di un'interfaccia di gestione attraverso la quale iniettare le configurazioni. In aggiunta, le interfacce che gestiscono il traffico dati, come mostrato in figura 7.5, costituiscono, in qualsiasi macchina virtuale realizzata, il punto di ingresso e il punto di uscita dei pacchetti.

In aggiunta, le peculiarità appena descritte, sviluppate nelle macchine virtuali, rendono il servizio avulso dalla tecnologia implementativa utilizzata: in pratica, non c'è alcuna dipendenza con l'ipervisore che virtualizza le risorse e può essere utilizzato, quindi, in moltissimi contesti. Per questo è possibile delegare l'istanziazione delle macchine virtuali ad un software come OpenStack: come già anticipato, il gestore dell'infrastruttura non definisce un particolare tipo di hypervisor da utilizzare nella virtualizzazione delle risorse e risulta compatibile con molteplici tecnologie. Inoltre, l'utilizzo di OpenStack è reso possibile dalle scelte progettuali adottate riguardo le macchine virtuali, come spiegato precedentemente: infatti, OpenStack istanzia un servizio in modo non predicibile all'interno dell'infrastruttura fisica, ma, nella nostra soluzione, non costituisce un vincolo significativo in quanto ogni macchina virtuale mantiene la sua struttura e i suoi metodi di configurazione che ne permettono il funzionamento indipendentemente dalla locazione in cui viene installata.

Capitolo 8

Risultati sperimentali

In questo capitolo andremo ad analizzare i risultati ottenuti da questo lavoro, partendo dalla descrizione dello scenario allestito per fornire un Proof-of-Concept. Premettendo che la soluzione proposta nei precedenti capitoli è orientata verso la semplificazione della gestione dei servizi di sicurezza per la rete lato utente e che quindi risulta arduo il compito di quantificare i risultati, è fondamentale raccogliere alcuni dati per valutare la fattibilità della tecnologia presentata e valutarne le prestazioni. A tal proposito, utilizziamo come parametri fondamentali di riferimento il tempo necessario per completare il ciclo delle configurazioni e la larghezza di banda ottenuta, per eseguire confronti opportuni.

8.1 Banco di prova

Prima di procedere con l'analisi dei dati, è fondamentale descrivere lo scenario allestito per dare significato ai risultati ottenuti e valutarne, in prospettiva di una reale applicazione, la bontà.

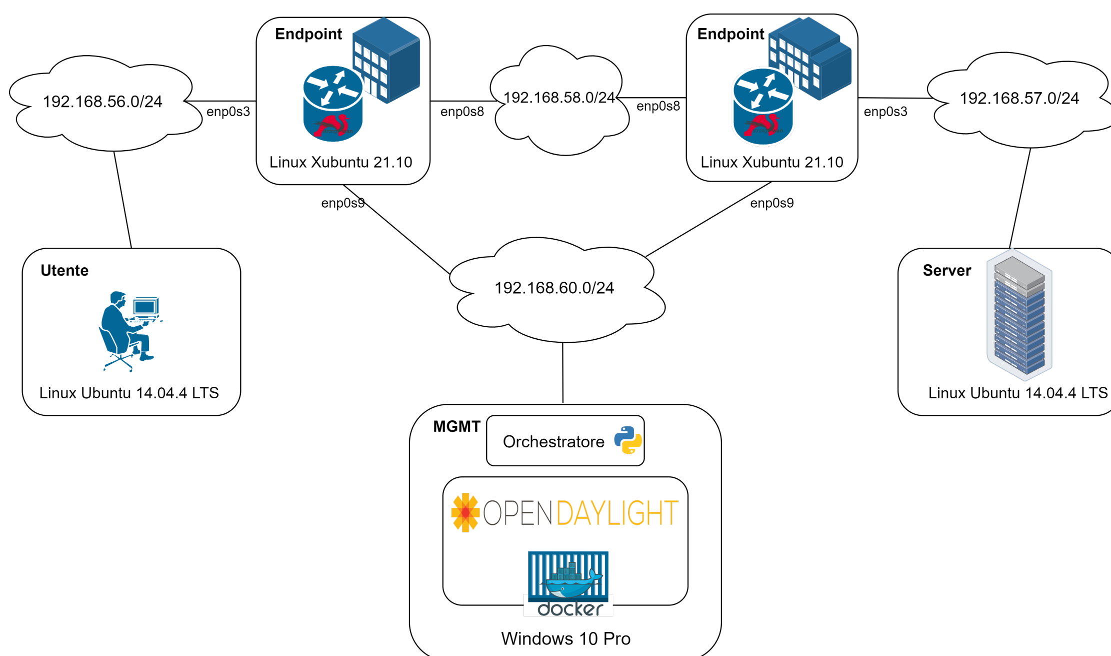


Figura 8.1. Scenario

Per realizzare lo scenario rappresentato in figura 8.1 è stata utilizzata una sola macchina fisica (MGMT), in cui vengono eseguiti l'orchestratore, la piattaforma NEMO e il software di virtualizzazione; tutti gli strumenti sono eseguiti in macchine virtuali diverse, ciascuna con caratteristiche differenti. Chiaramente, quello realizzato non rappresenta il banco di prova ideale per la nostra soluzione, in quanto siamo costretti a sfruttare pesantemente il software di virtualizzazione, inficiando in modo consistente le prestazioni. Uno scenario più adatto potrebbe includere altre due macchine fisiche a realizzare i terminatori VPN, in quanto il carico di lavoro si concentra particolarmente i quei nodi. Lasciamo tuttavia questa possibilità come un possibile sviluppo futuro e ci concentriamo sugli strumenti a disposizione che, in ogni caso, ci permettono di valutare la piattaforma realizzata; in fondo, l'interesse di studio principale riguarda le prestazioni della piattaforma di gestione a intenti e non l'ottimizzazione della VPN vera e propria.

Di seguito riportiamo le caratteristiche di ciascun nodo:

- **MGMT**:
  - CPU: Intel Core i7-9750H CPU @ 2.60GHz, 6 core, 12 processori logici
  - RAM: 16GB
  - Sistema Operativo: Windows 10 Pro
- **Utente, Server**:
  - CPU: Virtualizzata basata su ipervisore VirtualBox, 1 core virtuale
  - RAM: 1GB
  - Sistema Operativo: Linux Ubuntu 14.04.4 LTS sprovvisto di interfaccia grafica
- **Endpoint**:
  - CPU: Virtualizzata basata su ipervisore VirtualBox, 2 core virtuali
  - RAM: 2GB
  - Sistema Operativo: Linux Xubuntu 21.10

Le configurazioni testate su questa architettura riguardano la realizzazione di scenari con VPN basate su autenticazione con certificati X.509.

Prima di iniziare con la descrizione dei risultati, è fondamentale fare una precisazione: l'autenticazione basata su certificati X.509, ha richiesto un piccolo intervento sulla piattaforma NEMO e sull'orchestratore per far in modo che la simulazione funzioni completamente. La realizzazione di uno scenario di prova, con l'utilizzo di certificati validi risulta molto complesso, per questo motivo abbiamo integrato nel modulo di gestione delle configurazioni, un piccolo modulo che fosse in grado di operare con il formato *PKCS #12*; sfruttando le capacità di strongSwan, abbiamo generato due certificati per i due terminatori VPN firmati da una *Autorità di Certificazione* (CA) creata da noi. Ciò garantisce che le due parti, che si "fidano" della CA, possano accettare ciascuna il certificato dell'altra e ritenerlo valido. A seguito della generazione dei certificati, il modulo creato permette di definire una struttura dati ad hoc per il trasporto di materiale crittografico: sfruttando ancora una volta la libreria OpenSSL, vengono creati due involucri in formato *PKCS #12*, contenenti ciascuno la chiave privata e il certificato dell'endpoint, oltre al certificato della CA radice, da utilizzare in fase di verifica. Entrambi gli involucri, protetti da una password, vengono inviati ad entrambi i terminatori per essere utilizzati dal software di realizzazione della VPN. In conclusione, abbiamo stabilito che, per semplicità di sintassi,

se un utente specifica l'autenticazione basata su chiave pubblica, ma non indica il nome del certificato, allora questo viene generato, al fine di rendere flessibile la simulazione.

Naturalmente, questa soluzione rappresenta solo un modo di gestire i certificati funzionale per la nostra soluzione, ma ignora i protocolli di scambio sicuri comunemente utilizzati e non tiene conto di una possibile lista di certificati revocati (*CRL*).

8.2 Analisi

Procediamo ora con la descrizione delle analisi e delle simulazioni effettuate, volte a provare la fattibilità di questa soluzione. L'approccio comune a tutti i test effettuati prevede la misurazione del parametro di riferimento in casi simili, in cui di volta in volta si modifica la configurazione e si confrontano i risultati.

8.2.1 Modulo di traduzione

La prima simulazione eseguita ha come target il modulo di traduzione che, come descritto nel capitolo 7, abbiamo realizzato in toto; lo scopo è quello di stressarlo con richieste multiple di traduzioni per analizzare la sua tenuta. In particolare tramite l'interfaccia di NEMO, abbiamo definito una topologia elementare costituita da due nodi da utilizzare come terminatori per le VPN. Per automatizzare la simulazione, abbiamo inserito parte del processo eseguito dall'orchestratore in un ciclo in cui, semplicemente definendo il numero di iterazioni, si ottiene un numero di connessioni diversamente ampio. Tale processo iterativo include la definizione dell'intento per richiedere la VPN e la richiesta di creazione dei file di configurazione, mentre esclude la registrazione dell'utente e la richiesta dei nodi tramite intenti. In aggiunta, in questo scenario non sono presi in considerazione i contributi aggiuntivi del processo di orchestrazione per cui tutti i tempi si riferiscono al solo tempo intercorso tra la sottomissione dell'intento e la generazione del file di configurazione.

L'intento che specifica le caratteristiche della connessione è il seguente, dove la lettera `X` rappresenta un numero progressivo:

```
CREATE Connection vpnX Type tunnel_ipsec Endnodes r1,r2 Property auth-method:
certificate, ipsec-mode:  tunnel, esp-enc-protocol:  aes256, esp-hash-protocol:
sha256 ACL 192.168.56.0/24 to 192.168.57.0/24;
```

Router	VPN	Tempo (s)
2	50	13,131
2	100	26,749
2	150	38,866
2	200	53,046
2	250	67,973

Tabella 8.1. Dettagli sulla topologia creata con relativi tempi di esecuzione

I dettagli sulle iterazioni effettuate sul processo sono riportati nella tabella 8.1: mantenendo costante il numero di router, tra i quali sono state istanziate inizialmente cinquanta VPN, si è arrivati ad una topologia più complessa con molte VPN. Non è stata presa in considerazione una topologia più vasta per non aggiungere carico computazionale necessario per definire tutti i nodi; questo avrebbe inciso sui risultati riportando un contributo

strettamente legato alla piattaforma NEMO e meno legato al processo esaminato. L'unità di campionamento utilizzata nella valutazione di questo scenario è di cinquanta VPN; questo implica che ad ogni campionamento vengano generati cinquanta file aggiuntivi rispetto alla precedente iterazione. La formula 8.1 definisce il contributo di ogni operazione al tempo finale ottenuto:

$$t_{tot}(i) = i * (t_{transazione}(i) + t_{intento}(i) + t_{generazione}(i)) \tag{8.1}$$

In particolare, i dati riportati in tabella risentono del tempo necessario ad aprire e chiudere la transazione entro la quale definire l'intento per realizzare una connessione virtuale; l'ultimo contributo è dato dal processo di generazione del file che inizia con la richiesta allo specifico metodo dell'interfaccia e termina con la creazione del file. Il peso di ciascun contributo dipende dal numero di iterazioni.

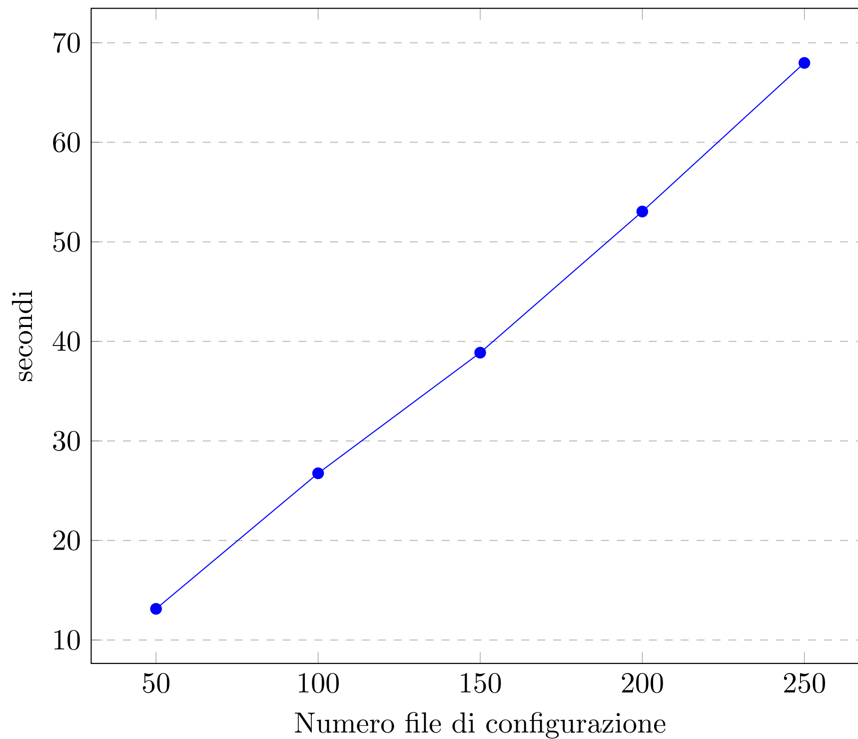


Figura 8.2. Andamento temporale in relazione al numero di configurazioni generate

Dalla figura 8.2 risulta evidente l'andamento lineare della curva caratteristica dei tempi di generazione dei file di configurazione rispetto al numero di file creati. Questo si traduce in un andamento costante delle prestazioni nel processo di traduzione delle componenti realizzate in questa soluzione, che non tendono a deteriorarsi con l'ingrandimento della base dati né con il numero sempre crescente di iterazioni. Questo è dimostrato sperimentalmente dal calcolo del tempo medio unitario di configurazione. La formula in 8.2 permette di calcolare il tempo medio impiegato per realizzare un file di configurazione:

$$t_{avgperunit} = \frac{\sum_{i=1}^{n} \frac{t(x_i)}{x_i}}{n} \tag{8.2}$$

In questo scenario il valore ottenuto è $t_{avgperunit} = 0{,}265s$ che, come avremo modo di verificare nella sezione 8.2.2, è assolutamente in linea con il tempo richiesto dalla

piattaforma NEMO per eseguire un ciclo completo e generare una configurazione. Nello scenario completo, il tempo attribuito alla piattaforma NEMO comprende anche la definizione della topologia mentre in questo test, come abbiamo precedentemente spiegato, il tempo considerato è relativo alla sola traduzione; questo giustifica la piccola differenza fra i valori con quelli riportati in tabella 8.2.

D'altro canto analizzando le iterazioni più dispendiose da un punto di vista temporale è possibile fare alcune considerazioni in vista di potenziali ottimizzazioni. Infatti, il processo di traduzione prende il via con l'apposita richiesta tramite l'interfaccia dopo aver definito, tramite un intento, la corrispondente connessione virtuale nello spazio di NEMO; in relazione ad uno scenario in cui vengono generate numerose configurazioni, questo si traduce in molte richieste e molti intenti sottomessi attraverso l'interfaccia apportando un carico aggiuntivo. In vista di una possibile implementazione reale, in cui è frequente la manipolazione di un elevato numero di configurazioni, sarebbe opportuno implementare un metodo in grado di ricevere l'istruzione di generare configurazioni per più dispositivi evitando la sottomissione di un elevato numero di richieste. Questo si traduce in un potenziale guadagno significativo per la configurazione di topologie vaste, dal punto di vista temporale.

L'ultimo test eseguito per valutare le prestazioni dell'intero processo di traduzione è strettamente correlato al precedente: si tratta dello stesso test, eseguito nelle stesse condizioni, in cui lo scopo è quello di valutare l'impatto registrato sulla piattaforma dovuto alla gestione di un intento più corposo. Sono state replicate le stesse condizioni topologiche del precedente test ed è stato richiesto un file di configurazione per ciascuna VPN, definita dal seguente intento:

```
CREATE Connection vpnX Type tunnel_ipsec Endnodes r1,r2 Property auth-method:
certificate, ipsec-mode:  tunnel, esp-enc-protocol:  aes256, esp-hash-protocol:
sha256, ike-enc-protocol:  aes128, ike-hash-protocol:  sha256 ACL
192.168.56.0/24 to 192.168.57.0/24;
```

In figura 8.3 è riportato il confronto fra gli andamenti dei due test eseguiti: il grafico denota un andamento molto simile per tutti i campionamenti eseguiti evidenziando, tuttavia, un lieve discostamento tra i due scenari in corrispondenza di topologie più popolose. Questo è dovuto alla gestione di un intento più dettagliato, utilizzato nel secondo scenario, e quindi di un quantitativo superiore di informazioni che assume un peso maggiore all'aumentare delle iterazioni. Infatti, se da un punto di vista di creazione del file i parametri da gestire sono costanti in numero, è da sottolineare come la definizione di più dettagli relativi ad una connessione si traduca nella creazione di entità virtuali più dettagliate e, quindi, più pesanti per la piattaforma. Ad ogni modo, il discostamento registrato è dell'ordine del secondo e quindi accettabile dal un punto di vista implementativo.

8.2.2 Scenario completo

Dopo aver discusso le prestazioni del processo di traduzione in una topologia complessa, valutiamo le prestazioni generali della soluzione in uno scenario classico, considerando il ciclo completo della configurazione di due terminatori VPN.

A tal proposito, abbiamo realizzato scenari differenti per testare le funzionalità implementate; in ciascuna simulazione abbiamo calcolato i contributi di ogni componente al tempo di configurazione totale. In particolare, considerando il processo nella sua interezza, partendo dalla registrazione dell'utente fino all'installazione del file di configurazione di strongSwan, abbiamo separato e confrontato i tempi necessari all'Orchestratore e alla

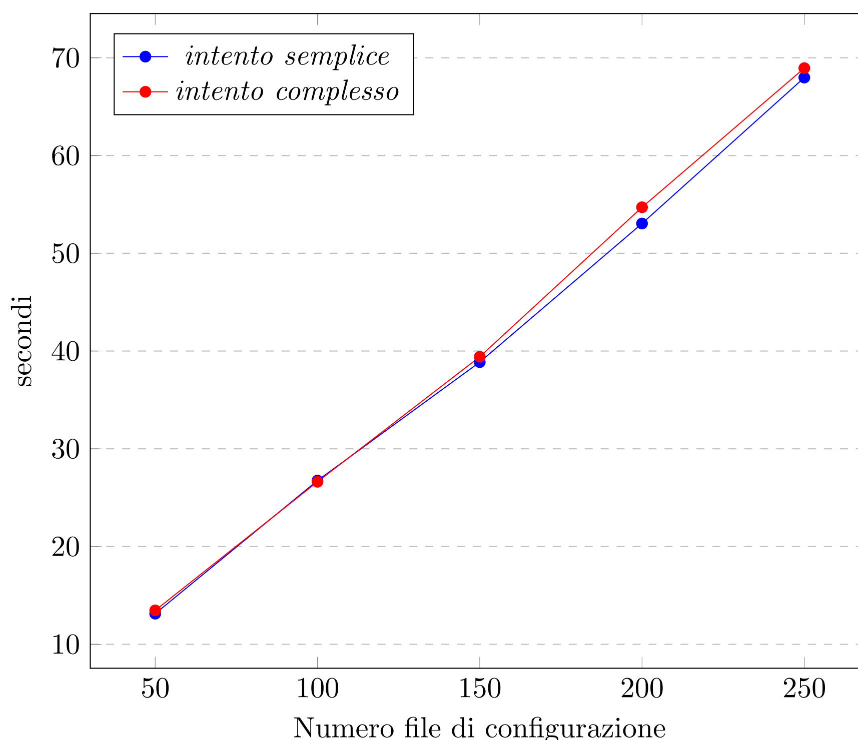


Figura 8.3. Andamento temporale in relazione al numero di configurazioni generate

piattaforma NEMO per avere una VPN funzionante secondo i parametri definiti dall'intento. Per ciascun calcolo le misurazioni sono state ripetute al fine di calcolare il valore medio di ogni contributo, riportati nella tabella 8.2.

Test	**NEMO(s)**	**Orchestratore (s)**
Riconfigurazione	0.388	1.344
Restart	0.349	5.617
VPN spenta	0.350	1.247

Tabella 8.2. Dettagli sui contributi nei test

Come riportato in figura 8.4, abbiamo simulato tre scenari diversi in cui si ottengono configurazioni e caratteristiche differenti. La topologia comune, rappresentata nella sezione 8.1, prevede la configurazione di un tunnel IPsec tra due macchine virtuali, che eseguono ciascuna un'istanza del software strongSwan.

Il primo test è realizzato a partire da uno scenario in cui nessun tipo di informazione preliminare è fornita a strongSwan e il nessuna connessione è attiva. In questo caso, la piattaforma realizza una connessione VPN basata su autenticazione a chiave pubblica, per questo nel contributo totale di NEMO figura anche il tempo di gestione dei certificati. In questo scenario il tempo assegnato al flusso di NEMO è ottenuto tenendo in considerazione tutte le interazioni con l'interfaccia di cui l'ultima permette la generazione del file; il timer per l'Orchestratore inizia con l'apertura dei canali SSH e termina quando la VPN è attiva e funzionante.

A partire da questo scenario, in cui il demone di strongSwan è attivo e la VPN è costruita e funzionante, abbiamo realizzato altre due simulazioni. Lo scopo è quello di testare uno scenario in cui è necessario riconfigurare una connessione già instaurata; il

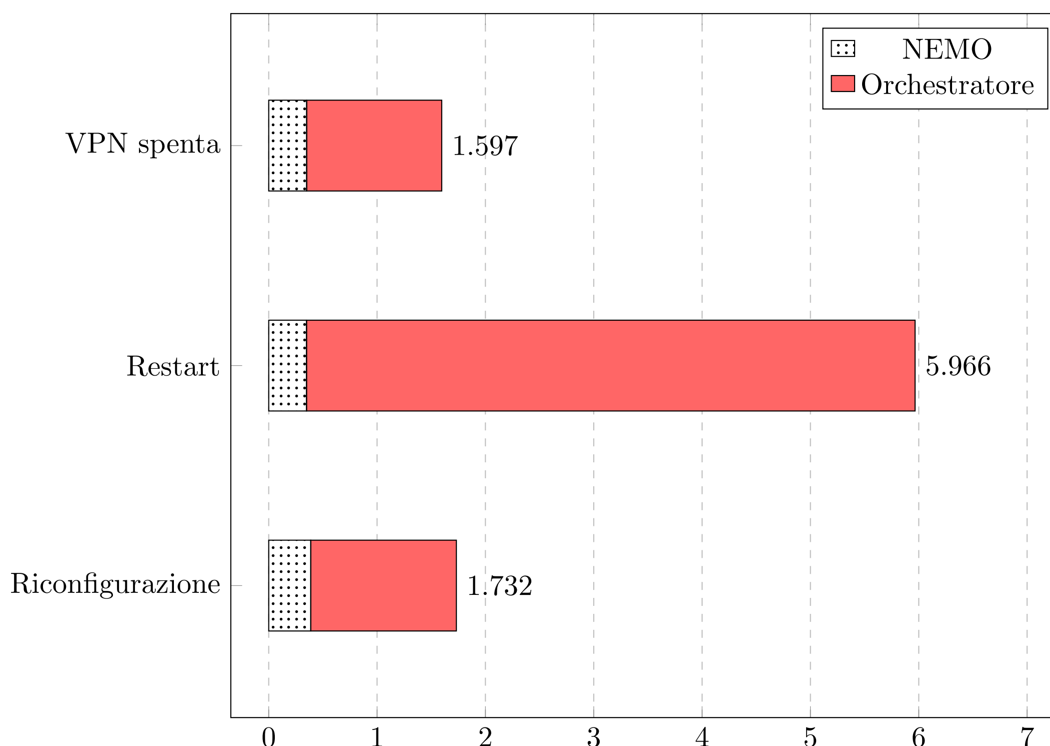


Figura 8.4. Configurazione dello scenario completo

tipo di autenticazione si basa, come per il caso precedente, su chiave pubblica. Come per il caso precedente nel processo è necessario definire le proprietà desiderate, tramite intenti, e successivamente chiederne la configurazione. Tuttavia, per rendere effettive le modifiche e permettere al demone di realizzare il comportamento desiderato, il processo di orchestrazione esegue sequenzialmente il comando di riavvio per entrambi i software di realizzazione VPN. Questo implica, per ciascun comando, l'inattività del processo per due secondi tra lo spegnimento e l'avvio impattando pesantemente sulle prestazioni. In figura 8.4 è evidente il peso del processo di orchestrazione rispetto al contributo di NEMO, nel caso in cui si esegua un riavvio del demone.

L'ultimo caso analizzato ci permette di valutare una soluzione alternativa che possa permettere di aggiornare la configurazione di una connessione già avviata, senza impattare in consistentemente le prestazioni. Una possibile soluzione che permette di ottenere risultati confrontabili con il primo scenario analizzato consiste nel creare una connessione gemella rispetta a quella correntemente eseguita. Considerando che risulta estremamente complesso forzare strongSwan ad aggiornare una connessione instaurata, se non riavviando il demone, la creazione di una nuova connessione permette di aggirare il problema: semplicemente definendo la connessione con un nuovo nome e con le nuove proprietà, l'Orchestratore permette di spegnere la precedente connessione ed attivare quella nuova con i parametri aggiornati. Il grafico in 8.4 dimostra la validità della soluzione che richiede un tempo di configurazione paragonabile allo scenario migliore.

Una considerazione finale molto importante va fatta sul contributo principale in ciascuno scenario, rappresentato dal lavoro di orchestrazione delle configurazioni. Come si può notare dai valori riportati in figura 8.4, i contributi sono confrontabili fra di loro dal punto di vista dell'ordine di grandezza, considerando che il processo è gestito allo stesso modo a prescindere dalla configurazione, ad esclusione dello scenario in cui il riavvio di strongSwan influisce pesantemente sulle prestazioni. Su entrambi gli scenari grava un

carico aggiuntivo che l'orchestratore deve gestire, rappresentato dallo scambio del materiale crittografico, a causa del processo descritto in 8.1, che, tuttavia, in uno scenario implementativo reale sarebbe da escludere, non causando quindi rallentamenti nella configurazione. Risultano invece sempre confrontabili le prestazioni della piattaforma NEMO che in tutti gli scenari descritti realizza le stesse operazioni.

Capitolo 9

Conclusioni e sviluppi futuri

Nel conseguimento degli obiettivi, il primo passo è stata l'analisi e la ricerca degli strumenti più adatti allo scopo. Tale investigazione era rivolta, in primo luogo, alla scelta di un linguaggio di alto livello adatto congiuntamente ad una piattaforma in grado di gestirli ed elaborarli. La selezione, appunto, ha portato verso l'adozione del linguaggio ODL-NEMO e della piattaforma NEMO sviluppata da OpenDaylight.

In secondo luogo la ricerca si è concentrata sulla realizzazione della VPN, identificata con lo studio di numerosi software e relativi metodi di configurazione. Anche questa ricerca ha condotto verso risultati positivi, presentando il software StrongSwan come il più funzionale. A questa scelta si è aggiunta quella riguardante il metodo di configurazione delle vNSFs, ricaduta, poi, sul protocollo SSH. La complessità nel processo di ricerca è acuita dalla necessità di conciliare la bontà delle prestazioni, in vista di un futuro utilizzo in uno scenario concreto, ma soprattutto la flessibilità e la possibilità di estendere la piattaforma.

L'analisi progettuale termina con lo studio della piattaforma NFV secondo lo standard ETSI e della possibilità di effettuare una trasposizione della soluzione in questo scenario. Trattandosi di uno scenario reale standard, la trattazione contenuta nella sezione 6.1.1 evidenzia la possibilità di realizzare questa soluzione in un contesto di applicazione reale.

Per valutare le scelte progettuali effettuate, è stato realizzato un sistema in grado di raccogliere tutti i frutti della ricerca e delle analisi effettuate sugli strumenti scelti. Non essendo ciò sufficiente ad ottenere una simulazione completamente funzionante, ai fini dei test, è stato realizzato un software esterno di orchestrazione, riprendendo il concetto di automazione nelle reti.

I test mirati realizzati sullo scenario finale ottenuto hanno evidenziato dei risultati esattamente in linea con le aspettative, in riferimento ai parametri in termini prestazioni, usabilità e flessibilità. Tra le simulazioni effettuate, anche quelle vicine a scenari reali hanno restituito risultati positivi, sottolineando la bontà delle scelte progettuali effettuate. Eseguendo dei test ripetuti abbiamo verificato come siano sufficienti 1,59s per la configurazione completa di una VPN partendo da zero e solo 0,26s in media al processo di traduzione per generare un file pronto per essere installato. Anche i test su scala più ampia hanno restituito buoni risultati, permettendo di creare 250 file di configurazione in 67,9s, pur individuando possibili ottimizzazioni.

In previsione di uno scenario più ampio, il primo intervento da annoverare tra i possibili lavori futuri coinvolge tutti i moduli impegnati nella realizzazione della configurazione, al fine ottenere un grado di automazione maggiore che possa incrementare ulteriormente l'usabilità. Infatti, il sistema realizzato è in grado di configurare le funzioni di sicurezza,

ma non si occupa, in nessun caso, di raccogliere le informazioni di routing, al fine di consentire il corretto scambio di pacchetti tra i terminatori. Se così fosse, la piattaforma sarebbe di realizzare uno scenario completamente funzionante senza richiedere alcun intervento dell'utente, ma richiede la conoscenza della topologia in esame. Per abilitare la piattaforma NEMO a fornire le informazioni sulla raggiungibilità di alcune destinazioni, è necessario che tutta la topologia sotto il controllo della piattaforma sia registrata nello spazio virtuale.

Questo conduce direttamente ad un altro possibile sviluppo futuro, che permetterebbe di scoprire la struttura della topologia in modo automatico. In questo modo, lo spazio virtuale di NEMO conterrebbe, in ogni momento, una fotografia esatta della topologia di rete. Ricollegandoci all'analisi precedente, le informazioni sulla topologia permetterebbero a NEMO di generare informazioni di routing da iniettare, poi, in ciascun nodo da configurare. Secondo questa strategia, NEMO sarebbe in grado di vedere dall'alto la topologia fisica e in grado di creare strutture dati corrispondenti ad ogni nodo fisico, per poi trarne informazioni per una configurazione completa ed automatica.

Dal punto di vista della sicurezza, in previsione di una possibile applicazione in uno scenario reale, è sicuramente necessaria una trasposizione di tutti i metodi offerti dall'interfaccia di NEMO su un protocollo sicuro come TLS. Il server web di Karaf permette, operando delle modifiche direttamente sui suoi file di configurazione, di predisporre la piattaforma all'utilizzo di HTTPS.

Appendice A

Manuale utente

Lo scopo di questa appendice è quello di definire il manuale per un utente generico, per fornire le linee guida per l'installazione e l'utilizzo generico della piattaforma. Il capitolo sarà organizzato come segue: procederemo, prima di tutto, all'analisi dei requisiti necessari all'installazione della piattaforma e, a seguire, l'installazione stessa del software; infine, illustreremo l'utilizzo del programma.

A.1 Requisiti

Per eseguire la versione di NEMO estesa è necessario partire dal codice sorgente, ma per procedere alla sua compilazione è necessario prima eseguire alcune operazioni preliminari. In aggiunta, distingueremo il caso in cui si voglia eseguire il codice direttamente nella macchina in cui viene compilato dal caso in cui si voglia realizzare un servizio in esecuzione su un container Docker.

Certamente, in entrambi i casi è essenziale disporre di un interprete Python per eseguire le operazioni disposte dall'orchestratore. In aggiunta, tramite il gestore di pacchetti *pip* è necessario installare le librerie esterne sfruttate dal programma:

```
$ sudo apt install python3-pip #installazione di pip
$ pip install paramiko
$ pip install requests
$ pip install rstr
```

A.1.1 Codice Sorgente

Al fine di preparare la macchina alla compilazione del programma è necessario installare programmi aggiuntivi e aggiornarne alcuni già presenti.

Come prima operazione, è necessario eseguire l'aggiornamento dei pacchetti del sistema operativo e, in seguito, si procede con l'installazione di pacchetti aggiuntivi per facilitare il processo:

```
$ sudo apt-get -y update
$ sudo apt-get -y install wget
```

Successivamente, per eseguire la compilazione del programma è necessario installare il *Java Development Kit* in versione 8, così da accertare il supporto con tutte le altre dipendenze, e creare la variabile d'ambiente per poterlo utilizzare da qualsiasi cartella:

```
$ sudo apt-get -y install openjdk-8-jdk
$ export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Per automatizzare la costruzione delle dipendenze, NEMO sfrutta il progetto *Apache Maven*, che permette di definire dei file in cui includere l'elenco di dipendenze esterne da scaricare a tempo di compilazione. Per questo è fondamentale assicurarsi della presenza di Maven all'interno della macchina:

```
$ wget https://dlcdn.apache.org/maven/maven-3/3.8.4/binaries/
  apache-maven-3.8.4-bin.tar.gz
$ sudo mkdir /usr/local/apache-maven
$ sudo mv apache-maven-3.5.4-bin.tar.gz /usr/local/apache-maven
$ sudo tar -xzvf /usr/local/apache-maven/apache-maven-3.5.4-bin.tar.gz
  -C /usr/local/apache-maven/
$ sudo update-alternatives --install /usr/bin/mvn mvn
    /usr/local/apache-maven/apache-maven-3.5.4/bin/mvn 1
$ sudo update-alternatives --config mvn
```

Con questi comandi si ottiene la versione `3.8.4` di Maven e si preparano le cartelle destinatarie del file binario. Con gli ultimi due si esegue l'installazione, assicurandosi venga utilizzata questa versione.

Infine è necessario, nuovamente, esporre una variabile d'ambiente contenente il percorso verso il file binario di Maven:

```
$ export M2_HOME=/usr/local/apache-maven/apache-maven-3.5.4
#impostazioni di memoria per JVM che esegue Maven
$ export MAVEN_OPTS="-Xms256m -Xmx512m"
```

A questo punto è necessario preparare l'ambiente per l'esecuzione indicando le dipendenze richieste dal programma, che saranno risolte da Maven stesso. Il file è fornito da OpenDaylight e le dipendenze vengono scaricate in un repository locale definito da Maven:

```
$ rm -rf ~/.m2
$ curl https://raw.githubusercontent.com/opendaylight/odlparent/master/
  settings.xml --create-dirs -o ~/.m2/settings.xml
```

A.1.2 Servizio Docker

L'installazione come servizio della piattaforma NEMO costituisce la soluzione più semplice dal punto di vista dell'utente, poiché le operazioni descritte nella sezione A.1.1 sono realizzate in modo automatico da Docker, a fronte di un file che le descriva.

In questo caso, l'unico requisito aggiuntivo, rispetto a quelli in comune con le altre tipologie di installazione, riguarda il motore di realizzazione del container chiamato appunto *Docker Engine*:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Questi comandi sono sufficienti a garantire un'installazione corretta del conatiner in ambiente Ubuntu. In ambito Microsoft Windows, invece, l'installazione di *Docker Desktop* permette di ottenere un'interfaccia grafica in cima al motore di esecuzione vero e proprio.

A.2 Installazione

Anche in questo caso è necessario separare i contesti di utilizzo testati in questa soluzione. Tuttavia, al contrario dei requisiti, l'installazione come servizio richiede comandi più complessi dovuto al fatto che eseguono operazioni più complesse.

A.2.1 Codice Sorgente

A partire dal codice sorgente, le uniche operazioni necessarie coincidono con la compilazione e la configurazione del programma. Per il primo passo, si sfruttano le caratteristiche di Maven:

```
# Comando da lanciare nella cartella contenente il codice sorgente
$ mvn clean install -DskipTests
```

Dopo aver atteso la creazione di tutti i file eseguibili operata da Maven, è possibile lanciare il programma per configurarlo. L'ipotesi è quella di lavorare nella cartella contenente il codice sorgente:

```
# Lancia la console di NEMO
$ ./nemo-karaf/target/assembly/bin/karaf
# Per selezionare i pacchetti da installare in NEMO
  feature:install odl-nemo-engine-ui odl-mdsal-all
# Chiude la console e conclude la configurazione
  logout
```

Dopo aver avviato la console, lo scenario che si presenta coincide con quello riportato in figura A.1, in cui si apre il pannello di controllo di NEMO. Dalla nuova riga di comando si possono eseguire i comandi per installare nuovi pacchetti, fondamentali per personalizzare l'installazione ed eseguire una versione funzionante del programma.

Descriveremo la sintassi e i comandi di questa nuova console nell'appendice B.

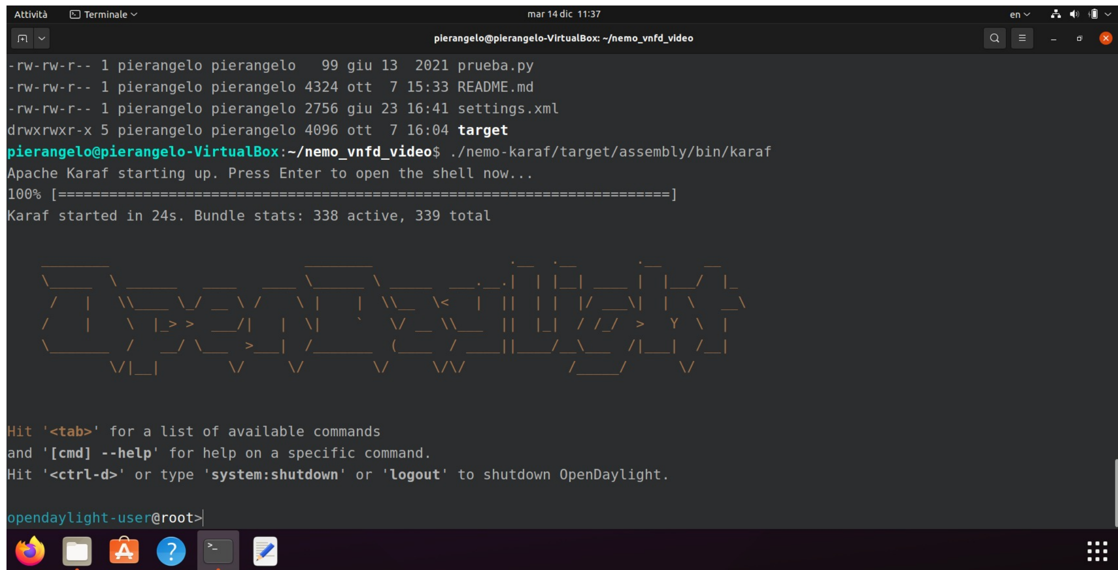


Figura A.1. Console di NEMO pronto per essere configurato

A.2.2 Servizio Docker

Nell'esposizione del servizio all'interno di un container, è fondamentale sfruttare le caratteristiche del motore di esecuzione di Docker per compilare il codice ed "impacchettare" tutti i file necessari all'esecuzione. Inseriti, poi, nell'ambiente d'esecuzione del singolo container, permettono di avviare il programma in modo agile, ma soprattutto di installarlo o migrarlo su qualsiasi piattaforma.

Il punto di partenza è il Dockerfile, da copiare nella cartella contenente il codice sorgente. A partire da questo file, si crea l'immagine software da utilizzare come base per il container; tale immagine, conterrà tutti i file compilati di NEMO e i servizi accessori per garantire il completo funzionamento. Creiamo l'immagine a partire dal codice e dal Dockerfile:

```
$ docker build -t nemo --progress plain .
```

In questo modo avremo la possibilità di osservare i progressi del processo, al termine del quale avremo un'immagine software chiamata `nemo` grazie alla quale potremo creare dei container in grado di eseguire il software di NEMO.

Passiamo quindi alla creazione vera e propria del container, nella quale bisogna specificare la corrispondenza tra le porte utilizzate dal programma e quelle esposte dal container, di cui forniremo una descrizione nella sezione A.2.4:

```
$ docker run --name nemo -p 8181:8181 -p 8101:8101 -p 44444:44444
  -p 1099:1099 -p 2222:2222 nemo
```

In un ambiente di esecuzione in cui è installato Docker Desktop è possibile vedere nella dashboard il risultato di questi comandi, che mostra il container creato ed in esecuzione (figura A.2).

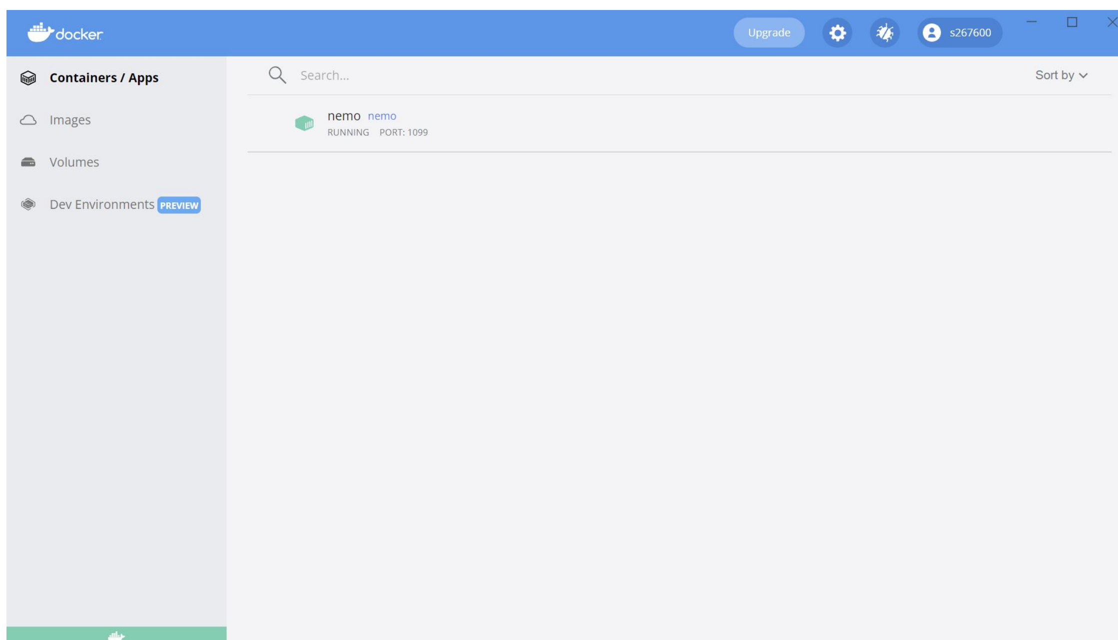


Figura A.2. Dashboard Docker Desktop

Come ultimo passaggio, è necessario avviare il server *SSH* installato all'interno del container:

```
$ docker exec -it nemo bash
  service ssh start
```

A.2.3 Terminatori VPN

Partendo da una distribuzione completa di Ubuntu, per configurare le vNSF che realizzano il terminatore VPN è sufficiente installare il software di StrongSwan ed aggiornare le regole di inoltro dei pacchetti. Questo è necessario per cambiare il comportamento della scheda di rete in modo da inoltrare un pacchetto con indirizzo di destinazione diverso dal proprio. Procediamo quindi con la configurazione:

```
$ sudo apt-get install strongswan strongswan-pki -y
$ sudo nano /etc/sysctl.conf
# modificare le righe corrispondenti all'interno del file
  net.ipv4.ip_forward = 1
```

```
  net.ipv6.conf.all.forwarding = 1
  net.ipv4.conf.all.accept_redirects = 0
  net.ipv4.conf.all.send_redirects = 0
# per applicare le modifiche
$ sudo sysctl -p
```

A.2.4 Creazione dell'ambiente di test

Per completare la preparazione dell'ambiente di esecuzione dell'intera soluzione, vediamo come emulare uno scenario completo in cui è possibile realizzare una VPN site-to-site grazie a VirtualBox. A livello logico, il design della rete individua due spazi separati, lo spazio virtuale creato da VirtualBox e lo spazio delineato nel lato utente. L'ambiente virtualizzato contiene tutta l'infrastruttura di rete che permette il transito dei pacchetti tra gli ipotetici host presenti nelle rispettive LAN e tra le vNSFs, mentre lato utente è contenuto quello che abbiamo definito come *Management Framework*.

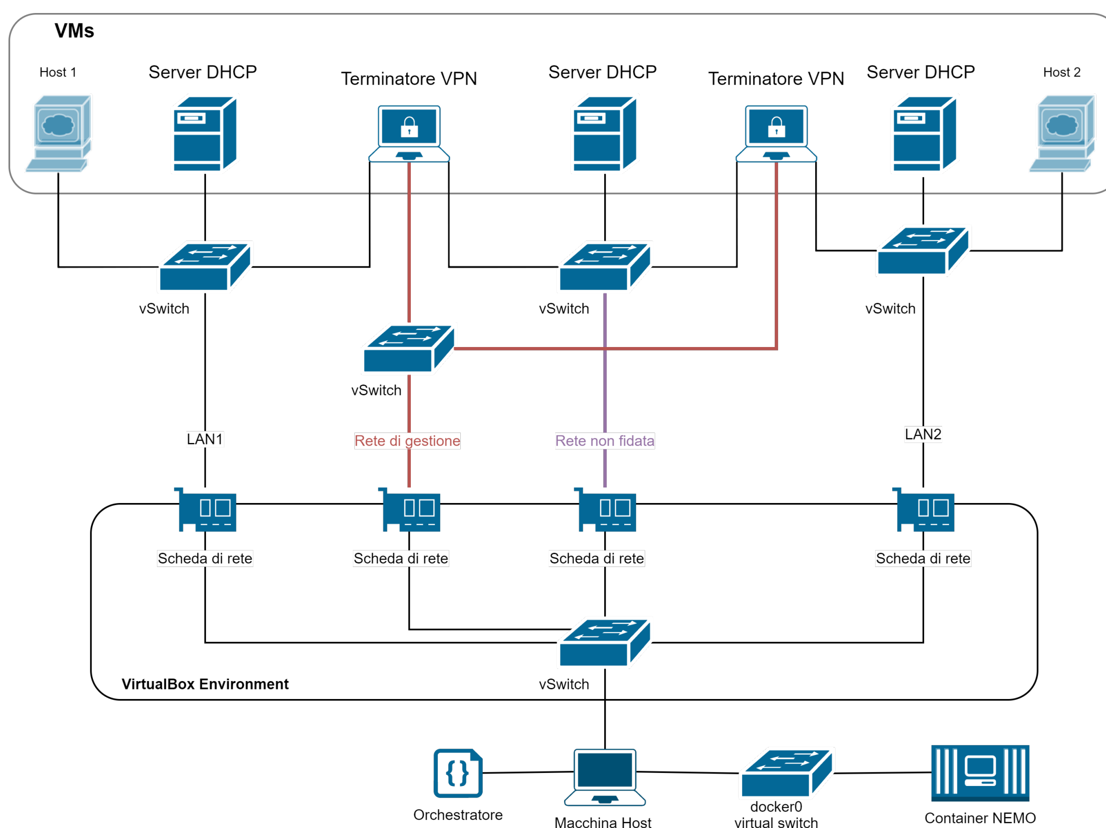


Figura A.3. Topologia di rete

L'infrastruttura realizzata lato VirtualBox fa ricorso pesantemente allo strumento di rete di cui il software di virtualizzazione è fornito. Questo permette di collegare le macchine virtuali secondo scenari preconfigurati: nella gestione del networking, infatti, è possibile selezionare il tipo di rete cui si vuole collegare la macchina virtuale (es. NAT network), sfruttandone le caratteristiche che conseguono. Nella nostra soluzione si è sfruttata la tipologia di rete definita *Host-only network*, che simula una rete Ethernet di livello 2, le cui caratteristiche permettono di soddisfare perfettamente i requisiti di

design che abbiamo precedentemente illustrato. Specificando il collegamento di una macchina virtuale ad una rete di questo tipo, il tool di gestione di VirtualBox provvede ad istanziare una scheda di rete virtualizzata, come illustrato in figura A.3, definendo, così, un nuovo spazio di indirizzamento. Questo nuovo set di indirizzi circoscrive la rete di livello 2 desiderata. L'utilizzo di un dispositivo di questo tipo è, chiaramente, una scelta fondamentale: bisogna considerare che la macchina ospitante ha una scheda fisica alla quale è assegnato un indirizzo definito dalla rete locale cui è connesso, quindi, per simulare una rete Ethernet separata, è necessario costruire un collegamento tra i due spazi di indirizzamento. In questo modo, oltre a creare uno spazio di indirizzamento separato, che avrebbe richiesto altrimenti un apparato di livello 3, si permette la comunicazione tra macchina fisica e macchina virtuale. Da questo processo ne risulta che la macchina su cui è eseguito il software di virtualizzazione avrà una scheda collegata alla rete fisica e tante schede virtuali quante sono le reti *Host-only* create: in breve, si avranno tanti indirizzi ip quante sono le interfacce, ciascuno appartenente a uno spazio di indirizzamento diverso, in modo da mettere in comunicazione l'host con tutte le reti virtualizzate. Come già anticipato, si è reso necessario l'utilizzo di quattro reti diverse, quindi quattro schede di rete virtualizzate differenti.

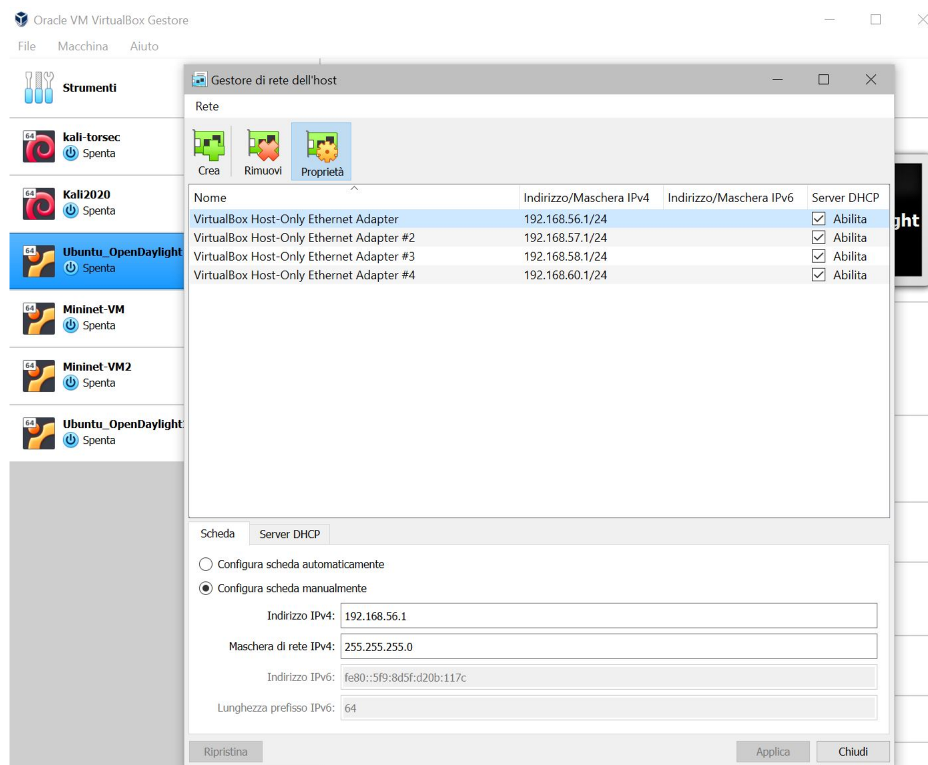


Figura A.4. Gestore di rete dell'host

A completare la gestione di ciascuna rete, si è implementato un server DHCP (Dynamic Host Configuration Protocol) sfruttando, ancora una volta, il tool di VirtualBox. Il server è incaricato, all'accensione di ciascuna macchina virtuale, di assegnare un indirizzo valido per lo spazio di indirizzamento cui appartiene: tipicamente, dopo aver selezionato un indirizzo alla prima accensione, verrà mantenuto in memoria il mapping al fine di assegnare lo stesso anche nelle volte successive. Infine, ciascuna rete è dotata di un *vSwitch* per consentire il collegamento di più macchine virtuali.

Per realizzare questa configurazione, come primo passo è necessario creare le schede di

rete virtualizzate e quindi definire un nuovo spazio di indirizzamento utilizzando il *Gestore di rete dell'host* fornito da VirtualBox. Utilizzando la scorciatoia da tastiera `Ctrl + h`, si ottiene il *tool* mostrato in figura A.4. Una volta creata la scheda virtualizzata, con l'apposito bottone in alto a sinistra, è possibile personalizzare la rete definendo lo spazio di indirizzamento e il numero di indirizzi da riservare. Successivamente, attraverso la scheda *DHCP*, è possibile attivare la spunta del server per facilitare l'assegnazione degli indirizzi a ciascuna macchina virtuale collegata alla rete.

A questo punto, dopo aver creato le macchine virtuali corrispondenti, è possibile utilizzare la scheda *Impostazioni* relativa al terminatore da configurare: come mostrato in figura A.5, dal tab *Rete* è possibile personalizzare quattro schede di rete virtuali, scegliendone la tipologia e la rete cui collegarle. Sarà sufficiente, quindi, per ciascuna macchina virtuale collegarla alle reti corrette, definite in fase di progettazione. Ad esempio, ciascun terminatore dovrà essere collegato a tre reti distinte e quindi sarà necessario configurare tre schede di rete differenti.

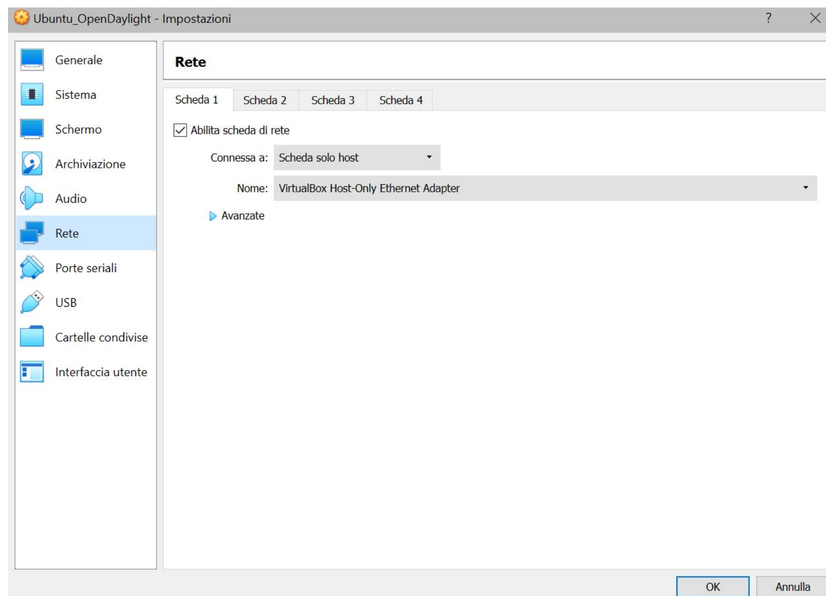


Figura A.5. Configurazione scheda di rete

Lato utente, invece, il networking è estremamente più semplice: l'orchestratore, se eseguito sulla macchina host, comunica con le macchine virtuali attraverso lo switch virtuale, mentre il framework NEMO, eseguito su un container, comunica attraverso il virtual switch fornito da Docker, chiamato *docker0*.

Una volta ottenuta la topologia completa, è necessario configurare, per ogni macchina virtuale, le regole di routing. In riferimento alla figura A.3, le configurazioni da realizzare sono le seguenti:

1. *Host 1*: è necessario configurare una regola per cui il traffico diretto verso *Host 2* viene inoltrato, come *next hop*, verso il *Terminatore VPN* collegato alla rete;
2. *Terminatore VPN*: in entrambi gli endpoint, è necessario definire due regole di routing, in modo da definire la direzione verso cui raggiungere la LAN di destinazione, ma anche la direzione per raggiungere l'host direttamente connesso;
3. *Host 2*: come per *Host 1*, è necessario specificare il terminatore direttamente collegato come rotta per raggiungere l'altro host.

Il comando generico, per inserire nuove regole di routing, permette di definire sia un indirizzo di rete che un indirizzo singolo:

```
$ sudo ip route add $IP_DESTINAZIONE via $IP_NEXT_HOP
```

In conclusione analizziamo brevemente le porte su cui è possibile contattare i servizi. Nel caso delle macchine virtuali, utilizzando il protocollo SSH, viene esposta la porta 22 TCP e UDP ed è utilizzata dai moduli di gestione per configurare le vNSFs. Lato host invece le interfacce sono molteplici: come detto l'orchestratore è eseguito sull'host e non espone interfacce mentre il container che esegue NEMO effettua il binding con cinque porte diverse. In particolare sulla porta 8181 esponiamo il servizio dell'interfaccia Web e i metodi MD-SAL RESTCONF, sulla porta 2222 esponiamo il servizio SSH mentre sulle porte 1099 e 44444 replichiamo servizi già esposti in automatico da NEMO riguardo la connessione remota al servizio (*Java RMI*). Tutte le porte appartengono al dominio del protocollo TCP.

A.3 Utilizzo

Concludiamo questa appendice con le indicazioni per un utilizzo generico delle piattaforme sviluppate all'interno di questa soluzione, oltre che ad un utilizzo completo al fine di avviare una simulazione. Ad ogni modo, per ciascun caso che verrà trattato, il punto di partenza è costituito dall'insieme di indicazioni fornite in questo capitolo, per cui si considerano i servizi già installati e correttamente configurati.

A.3.1 Utilizzo di NEMO

Considerando di aver già compilato il codice sorgente o di aver creato in precedenza il container, è necessario avviare il servizio per poter sfruttare le sue potenzialità:

```
# Per avviare NEMO dai file binari a partire dalla cartella principale
$ ./nemo-karaf/target/assembly/bin/karaf
# Per avviare il container NEMO
$ docker start nemo
```

A questo punto è possibile contattare la piattaforma attraverso le API esposte. A tal proposito è utile elencare i metodi più frequentemente utilizzati in fase di test della soluzione:

A.3.2 Utilizzo del framework sviluppato

In questa sezione vedremo i comandi necessari ad avviare una simulazione completa, sfruttando il flusso predefinito di lavoro implementato nell'orchestratore, di cui discuteremo i metodi nell'appendice B al fine di fornire le basi per realizzare scenari differenti.

Per questo, dopo aver installato e configurato correttamente tutti i componenti e, dopo aver ricreato la topologia definita nella sezione A.2.4, seguendo i passaggi descritti

nel corso di questa sezione si ottiene un tunnel VPN correttamente configurato, in cui le parti si autenticano sfruttando un certificato a chiave pubblica.

Come primo passo è fondamentale avviare, se non è già avviato, il servizio IPsec implementato da StrongSwan sui terminatori:

```
# Se si vuole inserire il comando tramite SSH, altrimenti si passa
  direttamente al secondo comando
$ ssh root@interface0_IP
$ sudo ipsec start
```

Per avviare il servizio NEMO ci rifacciamo, invece, alle operazioni descritte nella sezione A.3.1.

A questo punto, l'unico passaggio mancante risulta l'avvio dell'orchestratore che realizzi la simulazione vera e propria. Per completezza, ricordiamo che il programma principale dell'orchestratore permette di definire tramite intenti una topologia virtuale che ricalchi quella realizzata tramite VirtualBox, descritta nella sezione A.3.1, e successivamente, grazie ai file di configurazione prodotti da NEMO, permette la configurazione di IPsec, eseguito in ciascuno dei terminatori.

Tutto ciò è già costruito all'interno del programma principale, per cui è sufficiente avviarlo:

```
# Da lanciare all'interno della cartella principale dell'orchestratore
$ python3 demo.py
```

La demo permette di registrare un nuovo utente sulla piattaforma e, dopo aver aperto una nuova transazione, sottomette due intenti alla piattaforma richiedendo la realizzazione nello spazio virtuale di NEMO di due router che fungeranno, in ultima battuta, da terminatori per la VPN. A quel punto, all'interno di una nuova transazione, viene definito un intento per caratterizzare la VPN nello spazio virtuale di NEMO. L'ultima operazione prevede il prelievo dei file di configurazione dal file system di NEMO per installarli nei rispettivi terminatori.

A.3.3 Modifiche allo scenario

Un aspetto da chiarire rispetto ad un potenziale sviluppatore coincide con la creazione di uno scenario differente da quello presentato in questa soluzione. Come mostrato più volte, nel corso dei capitoli 6, 7, 8, lo scenario desiderato è formato da due terminatori VPN tra i quali realizzare un tunnel VPN di tipo site-to-site.

Un esempio di scenario realizzabile, senza intervenire sulla struttura della soluzione, è rappresentato da un contesto in cui tra i due terminatori vengono realizzati molteplici tunnel, ciascuno responsabile della protezione di un tipo di traffico diverso; ricordiamo che, in questo senso, il traffico viene individuato dalla coppia delle reti sorgente e destinazione.

La gestione di tunnel multipli è attuata da StrongSwan sfruttando gli identificativi definiti durante la configurazione di una nuova connessione. In particolare, il demone che

realizza la connessione legge tutte le informazioni contenute nel file *ipsec.conf*, in cui ogni configurazione corrisponde ad un nome, e realizza tutte le VPN richieste.

L'obiettivo da raggiungere, quindi, è quello di riempire il file con tanti blocchi di configurazione quante sono le VPN che si vogliono realizzare, ciascuna con un identificativo univoco. In tal senso, il modulo di traduzione è stato già progettato al fine di caratterizzare ciascuna configurazione con un identificativo, corrispondente al nome definito tramite intenti, assegnato alla connessione virtuale salvata nella base dati di NEMO.

Il secondo tassello verso una realizzazione di questo tipo di scenario è fornito dal processo di gestione delle nuove configurazioni realizzato in NEMO. In previsione di scenari arbitrariamente complessi, si è progettato un processo che operasse per singola connessione in base alle richieste inoltrate tramite intenti, per cui è sufficiente specificare il nome della connessione da configurare. Infatti, considerando che NEMO non pone alcun limite sullo scenario virtuale, la scelta più flessibile risulta quella di processare una connessione per ogni richiesta. All'utente è permesso di definire due endpoint collegati da molteplici connessioni di tipo VPN nello spazio virtuale e, successivamente, richiederne, per ciascuna, i file di configurazione. Tutto ciò è assolutamente in linea con gli obiettivi dichiarati in questa sezione.

Grazie ad un processo iterativo, dopo aver definito i due terminatori, si definiscono N intenti in cui vengono dichiarate N connessioni VPN con identificativi univoci (es. *vpn1...N*); in seguito, iterando nuovamente sui nomi delle VPN, si richiedono i file di configurazione da iniettare nei terminatori per configurare la connessione.

Quest'ultimo concetto conduce l'argomentazione verso un ultimo passaggio, necessario ad ottenere uno scenario completamente funzionante secondo gli obiettivi iniziali. Infatti, l'unico limite in questo senso è rappresentato dall'orchestratore che, ricordiamo, è incaricato di iniettare, tramite il protocollo SSH, i nuovi file di configurazione all'interno dei file *ipsec.conf* contenuti in ciascuno dei terminatori. In questa operazione, l'orchestratore è stato progettato, per scopi puramente implementativi, per sovrascrivere interamente il file con la nuova connessione richiesta, così da ottenere in qualsiasi scenario un aggiornamento delle caratteristiche della VPN.

A tal proposito, occorre effettuare una modifica all'interno del metodo di libreria presente nel file `orchLib.py` sotto il nome di `set_ipsec_conf`. La modifica consiste proprio nel rimuovere il trasferimento, con la seguente sovrascrittura, del file di configurazione di IPsec e realizzare una funzione di *append* nel file stesso. In questo modo, ogni nuovo file creato da NEMO non viene sostituito a quella già presente nel terminatore ma viene integrato al fondo, realizzando così esattamente il comportamento voluto.

Al termine di questo processo si otterranno due file di configurazione, ciascuno in un terminatore diverso, in cui è presente una configurazione di default e tante connessioni quante sono quelle definite nello spazio virtuale di NEMO. A questo punto, se si è utilizzato un metodo di autenticazione di cui è implementato lo scambio in maniera automatica (certificati X.509), il demone di StrongSwan sarà in grado di configurare ed avviare correttamente le connessioni; altrimenti, è fondamentale installare, in ciascun terminatore, tante *pre-shared-key* quante sono le connessioni realizzate.

Appendice B

Manuale sviluppatore

Lo scopo di questa appendice è quello di descrivere i componenti sviluppati in questa soluzione in termini di estensione delle funzionalità, in vista di un possibile utilizzo che si discosti da quello presentato nel corso di questa trattazione.

B.1 NEMO

B.1.1 Installazione packages

La prima possibilità di estensione in termini di funzionalità, che tratteremo brevemente, coincide con l'installazione di pacchetti aggiuntivi forniti da NEMO stesso. Sfruttando le potenzialità di *Apache Karaf*, che OpenDaylight usa come contenitore per le sue applicazioni, è infatti possibile integrare funzionalità al software installando parti aggiuntive, esportate appunto come pacchetti. OpenDaylight mette a disposizione numerosi *repository* da cui prelevare parti di codice da installare al momento, inseriti, tipicamente, all'interno dei file di configurazione di NEMO. In realtà, la console di *Karaf* supporta molti comandi, ma in questa sezione saranno esplorati quelli dedicati ai pacchetti.

Nella documentazione[1] sono riportati sotto la cateogria `features`, sottolineando la parola chiave con cui iniziano i comandi appartenenti a questa tipologia. La sintassi generica è la seguente:

```
features:<comando> [<opzioni>+] [<parametro>]
```

Per installare nuove funzionalità direttamente dalla console di NEMO, si parte sicuramente dai seguenti comandi che, rispettivamente, mostrano una lista di possibili opzioni oppure forniscono informazioni su una pacchetto in particolare:

```
> features:list [<opzioni>]
# -o visualizza i pacchetti in ordine alfabetico
# -i visualizza i pacchetti attualmente installati
```

[1] https://karaf.apache.org/manual/latest-2.x/commands/commands.html

```
> features:info [<opzioni>] nome
# -c mostra informazioni sulla configurazione
# -d mostra informazioni sulle dipendenze
```

Una volta individuato il pacchetto di interesse è possibile installarne il codice e renderlo operativo tramite il comando:

```
> features:install [<opzioni>] nome
# per disinstallarla
> features:uninstall [<opzioni>] nome
```

Tra i più utili e diffusi, alcuni pacchetti sono in grado di fornire interfacce grafiche per interagire con la piattaforma o per visualizzare le strutture dati secondo uno schema logico di alto livello. In particolare i pacchetti `odl-dluxapps-topology` e `odl-dluxapps-yangui`, forniscono una visualizzazione di quella che è la topologia rilevata dalla piattaforma (come se operasse da controllore SDN) e dei metodi esposti dall'interfaccia. La schermata mostrata in figura B.1, si ottiene collegandosi alla URL `http://IP:8181/dlux/index.html#/topology/index`, dopo aver installato il primo dei pacchetti citati ed essersi autenticati sulla piattaforma. Infine citiamo il pacchetto `odl-nemo-engine-ui` che fornisce supporto grafico agli strumenti di interazione con le operazioni tipiche di NEMO e del linguaggio a intenti.

In conclusione è importante sottolineare che la disponibilità di tali pacchetti dipende, sovente, dalla versione di ODL-NEMO eseguita a causa delle difficoltà di manutenzione del codice.

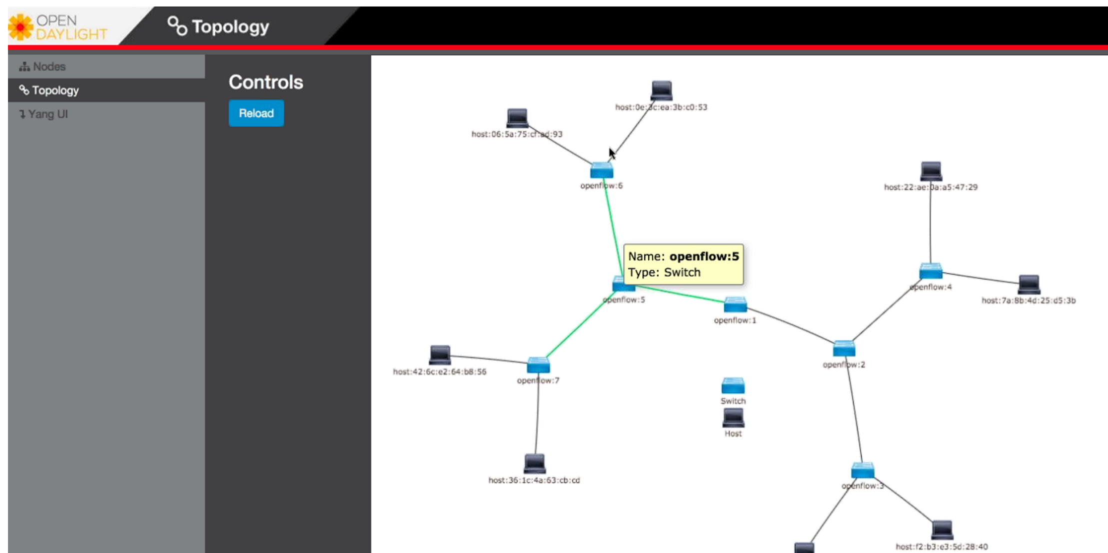


Figura B.1. Dlux Topology

B.1.2 Configurazioni IPsec

Descriviamo, a questo punto, come si pone NEMO rispetto all'estensione delle configurazioni di IPsec supportate. Premettendo che si è cercato di supportare tutti i parametri

e le parole chiave definite da StrongSwan, anche in questo aspetto la piattaforma è stata progettata per facilitare future integrazioni dovute, magari, ad un utilizzo di una versione aggiornata del software di realizzazione per connessioni VPN. L'obiettivo di questa sezione è quello di mostrare la procedura da seguire per estendere le proprietà che NEMO permette di configurare rispetto alla VPN.

Sfruttando la caratteristica di configurazione di StrongSwan, che basa il suo funzionamento sulle informazioni lette da un demone su un file di configurazione predefinito, è stato costruito uno scheletro che rispecchiasse esattamente la struttura di quel file, come ampiamente spiegato nella sezione 7.1.2. Partendo da questo file, è sufficiente riempirlo con i valori corrispondenti in modo da ottenere la configurazione desiderata; il resto del lavoro, come abbiamo visto, è realizzato dall'orchestratore.

Con questa premessa, appare evidente il primo target su cui concentrarsi al fine di estendere il supporto ad IPsec. Il file individuato dal seguente percorso relativo `nemo-impl/src/main/resources/site-to-site` è estendibile a piacere; seguendo le direttive di StrongSwan si può definire una parola chiave, all'interno del blocco di configurazione della connessione, seguita da un "segnaposto" individuato dai simboli {{}} che sarà riempito successivamente da un valore significativo. Per questo qualsiasi tipo di proprietà non ancora supportata può essere inserita in questo file. Ad esempio:

```
    ⋮
rightfirewall={{condition}} # yes|no
    ⋮
```

Un altro file da estendere è certamente `demo/connectiontypes.json` utilizzato, nel contesto della simulazione, per configurare le caratteristiche di tutte le connessione che si vogliono supportare nella piattaforma; in alternativa, si possono seguire le direttive fornite nella sezione 7.1.1 per contattare il metodo offerto da NEMO per configurare questo tipo di strutture dati. In ogni caso, questa struttura dati costituisce il parametro corrispondente alla nuova proprietà, definita seguendo le precedenti indicazioni, nel contesto degli intenti. In particolare, definendo in NEMO una nuova proprietà, è possibile inserirla all'interno di un intento, salvando così nella base dati della piattaforma un valore che trova corrispondenza con la caratteristica desiderata. In riferimento al precedente esempio è necessario effettuare la seguente modifica al file:

```
    ⋮
{
     "connection-type": "tunnel_ipsec",
     "property-definition": [
        {
         "property-name": "rightfirewall",
         "property-value-type": "string"
        },
         ⋮
    ]
    ⋮
}
```

Ciò rende possibile la definizione di un intento simile al seguente, grazie al quale si specifica un valore consistente da sostituire al segnaposto:

```
CREATE Connection vpn1 Type tunnel_ipsec Endnodes r1,r2 Property
rightfirewall:yes;
```

A questo punto del processo, si ottiene un file di configurazione "pronto" per essere completato con i valori corretti, prima di essere iniettato nei terminatori VPN, e si è guadagnata la possibilità di definire un valore, per ciascuna delle proprietà che compongono il file, tramite intenti.

Il collegamento mancante consiste nell'inserire ciascun valore dichiarato nell'intento nel corrispondente segnaposto, all'interno del file `site-to-site`. Anche a fronte di questa integrazione, il codice è stato progettato per poter minimizzare l'impatto dell'intervento. Infatti, tutto ciò si ottiene aggiungendo, nel modulo *vNSF Manager*, un nuovo caso nello *switch* che possa gestire la nuova proprietà, contenuta nella base dati, e nel modulo di traduzione, aggiungendo un oggetto che possa contenere il valore della proprietà stessa. Nella prima classe, infatti, è presente un costrutto che itera sulle proprietà trovate e ne verifica la corrispondenza con quelle supportate: aggiungendo un nuovo caso, si può gestire l'occorrenza della nuova proprietà invocando un metodo *setter*, per trasferire il valore dalla base dati alla classe `nemo-impl\src\main\java\org\opendaylight\nemo\user\climanager\vNSFGenerator.java`

In questa classe, è necessario implementare il metodo per salvare il valore della proprietà nel nuovo oggetto.

Una volta ottenuto il passaggio del valore della proprietà da una classe all'altra, nel modulo di traduzione è fondamentale aggiungere al contesto la nuova proprietà, associata al valore del segnaposto che si vuole sostituire. A quel punto, sarà il motore di Jinja a rimuovere il valore fittizio e a sostituirlo con quello corretto, definito tramite intento. Un esempio, riferito al caso precedente, potrebbe essere il seguente:

```
context.put("condition", firewallCondition);
# sostituisce la stringa "condition" con il valore contenuto
nell'oggetto Java "firewallCondition", corrispondente a quello definito
nell'intento
```

Questo metodo consente, quindi, di supportare qualsiasi tipo di configurazione IPsec permessa da StrongSwan che potenzialmente costituisce, di fatto, l'unico limite. Tuttavia, come già analizzato nella sezione 7.1.1, il template 5.1 permette di caratterizzare fortemente la connessione VPN; rimandiamo alla sezione citata poc'anzi per i dettagli sullo scheletro.

In conclusione, è possibile trasportare questi concetti verso un tipo di VPN diverso da quello realizzato in questo lavoro, corrispondente ad uno scenario di connessione *site-to-site*. Anche in questa integrazione, apparentemente più invasiva, è possibile seguire esattamente i passi descritti precedenti. Anche nella configurazione di una VPN di tipo *access*, tutto il processo viene delegato al demone di StrongSwan, per cui, ancora una

volta, il target dell'intervento è il template da caratterizzare secondo le aspettative del software per una connessione di questo tipo.

B.1.3 MD-SAL

Lo scopo di questa sezione è quello di introdurre, attraverso le scelte operate in questo lavoro, le modalità di estensione del modulo MD-SAL al fine di definire nuove strutture dati e nuovi metodi nell'interfaccia.

```
list access-control-list {
     key "access-control-list-id";
     leaf access-control-list-id {
         type access-control-list-id;
     }
     leaf acl-source-ip {
         type traffic-ip-address;
     }
     leaf acl-destination-ip {
         type traffic-ip-address;
     }
     uses access-control-list-instance;
  }
```

Figura B.2. Schema YANG

La figura B.2 mostra il modello YANG che abbiamo aggiunto nella soluzione, al fine di rappresentare l'entità ACL nella base dati. Il modello rappresenta una struttura dati di tipo "*list*" contenente una serie di oggetti strutturati allo stesso modo: ciascuno contiene una chiave (identificatore) univoca di tipo "access-control-list-id" e due indirizzi ip, sorgente e destinazione. Ogni oggetto nella struttura dati è definito da un tipo che abbiamo introdotto, come ad esempio "*traffic-ip-address*" che definisce il formato dell'indirizzo accettato, alla stregua del modello YANG. Al tempo di compilazione, MD-SAL crea una serie di classi Java per rappresentare perfettamente lo schema riportato, che saranno utilizzate successivamente per lavorare con il data store. Nel momento in cui il parser rileva l'entità *ACL*, invoca i metodi di questo modulo che ne costruiranno l'istanza da salvare in memoria: grazie alla creazione automatica di una classe *Builder* specifica per questa entità, basata sullo schema definito, viene popolato lo scheletro YANG con i dettagli inseriti che costituiscono l'istanza dell'oggetto `access-control-list`.

Come già anticipato, questo modulo implementa gran parte dell'interfaccia API, anch'essa guidata da modelli YANG. Ciascun metodo offerto dall'interfaccia è creato a partire dal corrispondente schema che implementa la *remote-procedure-call*. Nella nostra soluzione, è stato introdotto uno schema definito con il linguaggio YANG, che fornisse ad MD-SAL una descrizione del metodo con cui l'utente può richiedere l'intervento del traduttore, per permetterne la realizzazione automatica. Questo modello specifica il nome del nuovo metodo, i dati che richiede in input e quelli che fornisce come output: nella soluzione il metodo richiede in input l'identificativo univoco dell'utente e il nome (univoco) dell'oggetto creato nella rete virtuale per cui si richiede il file di configurazione e restituisce un messaggio riguardante l'esito della richiesta. Dopo aver definito la nuova *rpc* tramite MD-SAL, è possibile effettuare una `POST` verso l'indirizzo RESTCONF (http:

//IP:8181/restconf/operations/nemo-intent:output-cli-sequence); inoltre, è necessario creare un metodo che possa gestire la richiesta. In tal senso abbiamo integrato il modulo *Tenant Manager* con un'ulteriore funzione, al fine di realizzare le azioni che l'utente si aspetta possano scaturire dalla sua richiesta. Nel metodo introdotto, infatti, viene invocato il modulo *vNSF Manager* per gestire la traduzione, che descriveremo nella sezione 7.1.2. Inoltre, ricordiamo che il comportamento previsto previsto per il *Teanant Manager* consiste nel gestire le richieste provenienti dall'interfaccia API: è l'interfaccia stessa di NEMO che, a fronte di una richiesta in ingresso, provvede ad invocare il metodo di gestione corrispondente.

In conclusione, può risultare utile sottolineare la flessibilità che un framework come MD-SAL fornisce alla soluzione: in particolare, in ottica futura, appare molto semplice estendere le funzionalità offerte dall'interfaccia di NEMO, ma, ancor più importante, risulta immediata la definizione di nuove classi Java di cui è necessario fornire solamente lo schema in formato YANG. Naturalmente, tutte le classi accessorie che vengono realizzate automaticamente, costituiscono un overhead importante e conducono all'esplosione del quantitativo di codice da analizzare per un potenziale futuro sviluppatore.

B.1.4 Supporto a nuove vNSF

L'ultima analisi da affrontare rispetto processo di estensione delle funzionalità della piattaforma realizzata riguarda uno dei punti fondamentali che hanno guidato la realizzazione della soluzione: l'idea di base è quella di fornire un framework modulare, che possa garantire il supporto ad un portafoglio ampio di vNSF. Pertanto, le fasi di design sono state influenzate dalla scelta di seguire questa linea guida.

In primis, l'utilizzo del protocollo SSH apre, potenzialmente, alla possibilità di configurare qualsiasi applicazione di rete, che disponga di una riga di comando per influenzarne il comportamento. In secondo luogo, la struttura della piattaforma permette di implementare metodi di gestione e configurazione rivolti a molteplici funzioni.

Tuttavia, il primo problema da risolvere, affrontato originariamente durante la progettazione della soluzione e che si presenta per ogni integrazione all'interno della piattaforma NEMO, riguarda la realizzazione di una corrispondenza tra il linguaggio ad intenti e le entità da realizzare. In altri termini, bisogna identificare o estendere la sintassi, in modo da poter esprimere dei concetti rivolti a vNSF diverse. In questa soluzione, ad esempio, si è resa necessaria l'aggiunta di alcuni termini per definire gli indirizzi che identificassero il traffico da proteggere. In aggiunta, oltre alla sintassi, è fondamentale categorizzare la nuova funzione di rete con uno tra i modelli degli operatori di rete supportati da NEMO. In questo lavoro, il tunnel VPN è stato mappato sulla struttura dati della connessione. Tale processo è stato già descritto nella sezione 7.1.1, ma ne analizzeremo gli aspetti più tecnici.

Come mostrato in figura 7.3, dopo aver individuato, in fase di progettazione, la categoria di appartenenza della funzione di rete da integrare (es. firewall come nodo), è necessario aggiungere la descrizione della stessa nel file JSON corrispondente alla categoria e inviarlo, con il metodo responsabile per la categoria, alla piattaforma NEMO. I metodi incaricati, sono parte dell'interfaccia RESTCONF e sono strutturati, tipicamente, secondo uno schema fisso: http://NEMO_IP:8181/restconf/config/nemo-object:CATEGORY-definitions. Con una richiesta di tipo `POST` all'indirizzo corrispondente, in cui la parole `CATEGORY` è sostituita, appunto, dal modello (es. node, connection) e nel cui corpo è presente, in formato JSON, la descrizione della funzione, si consente a NEMO di

creare le strutture dati adatte per trattare questo nuovo tipo di dato. Questo processo è descritto nella sezione 7.1.1.

Dal punto di vista della sintassi, invece, la cartella presente all'interno del repository della soluzione, al percorso relativo `nemo-impl\src\main\java\org\opendaylight\nemo\user\vnspacemanager\languagestyle\NEMOParse`, contiene tutti i file responsabili della sua gestione e, quindi, i target di questo intervento. Il primo di questi è il file `LanguageStyle.jj`, che realizza, letteralmente, un dizionario per il parser JavaCC in cui sono presenti tutti i token, segni di punteggiatura e espressioni regolari accettati dalla sintassi. Dopo aver aggiunto in questo file la nuova parola, bisogna creare una corrispondenza nella classe `nemo-impl\src\main\java\org\opendaylight\nemo\user\vnspacemanager\languagestyle\NEMOParse\NEMOparserConstants.java`, assegnando al nuovo token un valore numerico da utilizzare come identificativo in fase di processamento.

Questa rappresenta certamente la fase più complessa e più variabile per quanto concerne l'estensione della sintassi. Un metodo specifico per ogni azione e per ciascuna categoria, permette di processare l'intento parola per parola secondo le regole definite. Nella classe `nemo-impl\src\main\java\org\opendaylight\nemo\user\vnspacemanager\languagestyle\NEMOParse\NEMOparser.java`, dopo aver individuato il metodo corretto, è necessario gestire la nuova parola all'interno del costrutto *switch*. Il frammento di codice `switch((jj_ntk==-1)?jj_ntk():jj_ntk)` permette di avanzare nella stringa di input processando la prossima parola, per cui, a seconda della sintassi ricercata, per ogni ripetizione dello switch si aprono una serie di casi in cui bisogna inserire le azioni da scatenare, qualora il parser verificasse l'occorrenza della nuova parola. Tale processo iterativo è già stato analizzato nella sezione 7.1.1.

Dopo aver adeguato la sintassi è necessario implementare la gestione e la configurazione della nuova vNSF, intervenendo sulle classi Java `vNSFManager.java` e `vNSFGenerator.java`. Entrambe sono state progettate e realizzate tenendo conto di possibili evoluzione, per cui risulta agile un intervento volto ad estendere il supporto ad altre vNSF.

B.2 Orchestratore

In questa sezione conclusiva, verrà fornita la documentazione della libreria realizzata per l'orchestratore, al fine di fornire supporto per eventuali integrazioni e per la realizzazione di scenari differenti. Verranno analizzate, a tal fine, le funzioni principali.

1. **Funzioni di configurazione di NEMO:**

   - `config(NEMO_IP)`: si tratta di un metodo di alto livello che contiene numerose chiamate a funzione che permettono di configurare tutte le strutture dati necessarie al funzionamento di NEMO. Tra tutti, vengono configurate le tipologie di nodi supportati, con le corrispettive proprietà, le tipologie di connessioni, i ruoli accettati per un nuovo utente e molte altre strutture dati. Tali informazioni vengono lette in input a partire da file JSON cercati nello stesso percorso della libreria: i nomi sono costruiti a partire dalla parte finale della URL corrispondente fornita da NEMO, escludendo i segni di punteggiatura e sostituendo la parola "*definitions*" con "*types*" (es. `nodetypes.json`). Vengono lanciate due eccezioni, a seconda che l'errore si sia verificato durante la richiesta `HTTP` oppure durante l'apertura del file;

2. **Funzioni di interfaccia di NEMO:**

    - `create_user_id()`: restituisce un identificativo univoco nella forma corretta richiesta da NEMO;
    - `register_user(uuid, username, password, role, NEMO_IP)`: esegue una `POST` verso l'interfaccia RESTCONF di NEMO e registra l'utente definito dai parametri richiesti in ingresso, quali identificativo univoco, nome utente, password, ruolo e indirizzo cui contattare la piattaforma;
    - `transaction_begin(uuid, NEMO_IP)`: tramite una `POST` comunica a NEMO l'inizio di una transazione richiesto dall'utente `uuid`;
    - `transaction_end(uuid, NEMO_IP)`: chiude una transazione precedentemente aperta dall'utente identificato dalla stringa `uuid`;
    - `language_style_nemo_request(uuid, NEMO_IP, intent)`: invia a NEMO l'intento `intent`, che verrà salvato all'interno di una transazione. È necessario definire l'identificativo dell'utente e l'indirizzo per contattare NEMO;
    - `generate_conf(uuid, NEMO_IP, name)`: esegue una `POST` per richiedere la configurazione corretta per l'oggetto `name`, salvato nella partizione della base dati corrispondente all'utente `uuid`. Il file di configurazione viene salvato nel percorso `/root/nemo/name.conf`, dove il nome del file è, di default, `device1` o `device2`;
    - `language_style_file_nemo_request(uuid, NEMO_IP, intent_file)`: metodo di alto livello che permette di comunicare a NEMO una serie di intenti definiti nel file `intent_file`.

3. **Funzioni di interazione con le vNSF:**

    - `open_ssh(ip_address, username, password, port)`: apre una connessione SSH verso l'endpoint all'indirizzo `ip_address`. È necessario specificare la porta, nel caso in cui non venisse utilizzata quella di default (porta 22 TCP/UDP), oltre al nome utente e password per effettuare l'accesso. Restituisce un oggetto di tipo `SSHClient` dalla libreria *Paramiko*;
    - `strongswan_restart(client)`: sfruttando il client SSH realizzato dopo l'apertura della connessione, invia i comandi per riavviare il software di StrongSwan;
    - `strongswan_status(client)`: mostra lo stato delle VPN instaurate da un terminatore, utilizzando il client SSH per collegarsi allo stesso;
    - `strongswan_start_connection(client, connection_name)`: si collega ad un terminatore tramite il client SSH ed avvia una connessione VPN, già configurata, di cui è necessario fornire il nome;
    - `set_ipsec_conf(client, file_path)`: utilizzando l'oggetto `client`, ottenuto dopo aver aperto una connessione, permette di modificare il file di configurazione della connessione di StrongSwan. In particolare, sostituisce l'intero file con quello trovato nel percorso locale indicato da `file_path`;
    - `upload_certificate(client, certificate_path, pri_key_path)`: permette di caricare, in qualsiasi terminatore, il certificato a chiave pubblica e la corrispondente chiave privata, di cui bisogna fornire il percorso locale. Il trasferimento dei file è permesso dal client SSH;

- `set_tunnel_rsa_key_auth(client, pri_key_name)`: consente di specificare il nome del file contenente la chiave privata RSA utilizzata nell'autenticazione a chiave pubblica per una specifica connessione. Il file viene ricercato all'interno della cartella predefinita `/etc/ipsec.d/private`;

Bibliografia

[1] Huawei, “Intent NBI for software defined networking.” http://www-file.huawei.com/~/media/CNBG/Downloads/Technical%20Topics/Fixed%20Network/Intent%20NBI%20for%20Software%20Defined%20Networking-whitepaper

[2] T. Szyrkowiec, M. Santuari, M. Chamania, D. Siracusa, A. Autenrieth, V. Lopez, J. Cho, and W. Kellerer, “Automatic intent-based secure service creation through a multilayer SDN network orchestration”, J. Opt. Commun. Netw., vol. 10, Apr 2018, pp. 289–297, DOI 10.1364/JOCN.10.000289

[3] T. Zhou, Z. Ji, Y. Zhang, S. Liu and Y. Xia, “NEMO language.” https://wiki.onosproject.org/display/ONOS/NEMO+Language

[4] Y. Tsuzaki and Y. Okabe, “Reactive configuration updating for intent-based networking”, 2017 International Conference on Information Networking (ICOIN), Da Nang (Vietnam), 2017, pp. 97–102, DOI 10.1109/ICOIN.2017.7899484

[5] D. Borsatti, W. Cerroni, G. Davoli, and F. Callegati, “Intent-based service function chaining on ETSI NFV platforms”, 2019 10th International Conference on Networks of the Future (NoF), Rome (Italy), 2019, pp. 144–146, DOI 10.1109/NoF47743.2019.9015069

[6] A. Marsico, M. Santuari, M. Savi, D. Siracusa, A. Ghafoor, S. Junique, and P. Skoldstrom, “An interactive intent-based negotiation scheme for application-centric networks”, 2017 IEEE Conference on Network Softwarization (NetSoft), Bologna (Italy), 2017, pp. 1–2, DOI 10.1109/NETSOFT.2017.8004251

[7] W. Cerroni, C. Buratti, S. Cerboni, G. Davoli, C. Contoli, F. Foresta, F. Callegati, and R. Verdone, “Intent-based management and orchestration of heterogeneous openflow/IoT SDN domains”, 2017 IEEE Conference on Network Softwarization (NetSoft), Bologna (Italy), 2017, pp. 1–9, DOI 10.1109/NETSOFT.2017.8004109

[8] A. Leivadeas and M. Falkner, “VNF placement problem: A multi-tenant intent-based networking approach”, 2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), Paris (France), 2021, pp. 143–150, DOI 10.1109/ICIN51074.2021.9385553

[9] E. J. Scheid, P. Widmer, B. B. Rodrigues, M. F. Franco, and B. Stiller, “A controlled natural language to support intent-based blockchain selection”, 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Toronto (Canada), 2020, pp. 1–9, DOI 10.1109/ICBC48266.2020.9169473

[10] F. Esposito, J. Wang, C. Contoli, G. Davoli, W. Cerroni, and F. Callegati, “A behavior-driven approach to intent specification for software-defined infrastructure management”, 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Verona (Italy), 2018, pp. 1–6, DOI 10.1109/NFV-SDN.2018.8725754

[11] T. Subramanya, R. Riggio, and T. Rasheed, “Intent-based mobile backhauling for 5G networks”, 2016 12th International Conference on Network and Service Management (CNSM), Montreal (Canada), 2016, pp. 348–352, DOI 10.1109/CNSM.2016.7818445

[12] E. J. Scheid, C. C. Machado, M. F. Franco, R. L. dos Santos, R. P. Pfitscher, A. E. Schaeffer-Filho, and L. Z. Granville, "INSpIRE: Integrated NFV-based intent refinement environment", 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Lisbon (Portugal), 2017, pp. 186–194, DOI 10.23919/INM.2017.7987279

[13] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, "Refining network intents for self-driving networks", Proceedings of the Afternoon Workshop on Self-Driving Networks, New York, NY, USA, 2018, p. 15–21, DOI 10.1145/3229584.3229590

[14] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain, "An intent-based approach for network virtualization", 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent (Belgium), 2013, pp. 42–50

[15] R. Ribeiro, A. Jacobs, R. Parizotto, L. Zembruzki, A. Schaeffer-Filho, and L. Granville, "A bottom-up approach for extracting network intents", Advanced Information Networking and Applications, pp. 858–870, Alberto Egon Schaeffer-Filho's Lab, March 2020, DOI 10.1007/978-3-030-44041-1_75

[16] W. Lehr, "Vertical integration and internet industry structure: An application of the pricing taxonomy", MIT Sociotechnical Systems Research Center (SSRC), July 2002

[17] M. H. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. E. Carpenter, S. Jiang, and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals." RFC 7575, June 2015, DOI 10.17487/RFC7575

[18] S. Jiang, B. E. Carpenter, and M. H. Behringer, "General Gap Analysis for Autonomic Networking." RFC 7576, June 2015, DOI 10.17487/RFC7576

[19] M. Fedor, M. L. Schoffstall, J. R. Davin, and D. J. D. Case, "Simple Network Management Protocol (SNMP)." RFC 1157, May 1990, DOI 10.17487/RFC1157

[20] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Intent-Based Networking - Concepts and Definitions", Internet-Draft draft-irtf-nmrg-ibn-concepts-definitions-06, Internet Engineering Task Force, December 2021. https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-concepts-definitions-06

[21] C. Janz, N. Davis, D. Hood, M. Lemay, D. Lenrow, L. Fengkai, F. Schneider, J. Strassner, and A. Veitch, "Intent NBI – definition and principles", ONF foundation, October 2016. https://opennetworking.org/wp-content/uploads/2014/10/TR-523_Intent_Definition_Principles.pdf

[22] A. Lerner, "Intent-Based Networking", Gartner, February 2017. https://blogs.gartner.com/andrew-lerner/2017/02/07/intent-based-networking/

[23] "Intent-Based Networking", Cisco, December 2018. https://www.cisco.com/c/en/us/solutions/intent-based-networking.html

[24] "IDN Maximize Your Business Value", Huawei, 2018. https://developer.huawei.com/ict/en/site-idn

[25] T. Eckert, M. H. Behringer, and S. Bjarnason, "An autonomic control plane." RFC 8994, May 2021, DOI 10.17487/RFC8994

[26] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani, "A survey on intent-driven networks", IEEE Access, vol. 8, 2020, pp. 22862–22873, DOI 10.1109/ACCESS.2020.2969208

[27] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for managing network resources", IEEE/ACM Transactions on Networking, vol. 26, no. 5, 2018, pp. 2188–2201, DOI 10.1109/TNET.2018.2867239

[28] S. Hares, "Intent-Based Nemo Overview", Internet-Draft draft-hares-ibnemo-overview-01, Internet Engineering Task Force, October 2015. https://datatracker.ietf.org/doc/html/draft-hares-ibnemo-overview-01

[29] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, D. C. Cui, D. H. Deng, J. Benitez, *et al.*, "Network functions virtualisation an introduction, benefits, enablers, challenges & call for action", SDN and OpenFlow World Congress, October 22-24, 2012

[30] S. Vennam, "IBM on Cloud Computing", standard, IBM, August 2020. https://datatracker.ietf.org/doc/html/draft-hares-ibnemo-overview-01

[31] "Network Functions Virtualisation (NFV); Architectural Framework", standard, European Telecommunications Standards Institute, October 2013. https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf

[32] "Network Functions Virtualisation (NFV); Virtual Network Functions Architecture", standard, European Telecommunications Standards Institute, January 2014. https://datatracker.ietf.org/doc/html/draft-hares-ibnemo-overview-01

[33] "Network Functions Virtualisation (NFV); Infrastructure Overview", standard, European Telecommunications Standards Institute, January 2015. https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_nfv-inf001v010101p.pdf

[34] "Network Functions Virtualisation (NFV); Management and Orchestration", standard, European Telecommunications Standards Institute, December 2014. https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf

[35] V. Little, "TSC perspective OSM architecture", Zero Touch Congress, October 2017

[36] A. Israel, A. Hoban, A. T. Sepúlveda, F. J. R. Salguero, and G. G. de Blas, "OSM release three, a technical overview", OSM Community White Paper, October 2017

[37] "Internet Protocol." RFC 791, September 1981, DOI 10.17487/RFC0791

[38] K. Seo and S. Kent, "Security Architecture for the Internet Protocol." RFC 4301, December 2005, DOI 10.17487/RFC4301

[39] P. R. Karn, W. A. Simpson, and P. E. Metzger, "The ESP Triple DES Transform." RFC 1851, September 1995, DOI 10.17487/RFC1851

[40] S. Kent, "IP Authentication Header." RFC 4302, December 2005, DOI 10.17487/RFC4302

[41] S. Kent, "IP Encapsulating Security Payload (ESP)." RFC 4303, December 2005, DOI 10.17487/RFC4303

[42] D. Carrel and D. Harkins, "The Internet Key Exchange (IKE)." RFC 2409, November 1998, DOI 10.17487/RFC2409

[43] J. Turner, M. J. Schertler, M. S. Schneider, and D. Maughan, "Internet Security Association and Key Management Protocol (ISAKMP)." RFC 2408, November 1998, DOI 10.17487/RFC2408

[44] H. Orman, "The OAKLEY Key Determination Protocol." RFC 2412, November 1998, DOI 10.17487/RFC2412

[45] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC 5280, May 2008, DOI 10.17487/RFC5280

[46] A. Viswanathan, E. C. Rosen, and R. Callon, "Multiprotocol Label Switching Architecture." RFC 3031, January 2001, DOI 10.17487/RFC3031

[47] A. J. Valencia, G. Zorn, W. Palter, G.-S. Pall, M. Townsley, and A. Rubens, "Layer Two Tunneling Protocol (L2TP)." RFC 2661, August 1999, DOI 10.17487/RFC2661

[48] G. Zorn, G.-S. Pall, and K. Hamzeh, "Point-to-Point Tunneling Protocol (PPTP)." RFC 2637, July 1999, DOI 10.17487/RFC2637

[49] W. A. Simpson, “PPP Challenge Handshake Authentication Protocol (CHAP).” RFC 1994, August 1996, DOI 10.17487/RFC1994
[50] W. A. Simpson, “The Point-to-Point Protocol (PPP).” RFC 1661, July 1994, DOI 10.17487/RFC1661
[51] T. Li, D. Farinacci, S. P. Hanks, D. Meyer, and P. S. Traina, “Generic Routing Encapsulation (GRE).” RFC 2784, March 2000, DOI 10.17487/RFC2784
[52] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3.” RFC 8446, August 2018, DOI 10.17487/RFC8446
[53] C. Newman, “Using TLS with IMAP, POP3 and ACAP.” RFC 2595, June 1999, DOI 10.17487/RFC2595
[54] E. Rescorla, “HTTP Over TLS.” RFC 2818, May 2000, DOI 10.17487/RFC2818
[55] J. Galbraith and O. Saarenmaa, “SSH File Transfer Protocol”, Internet-Draft draft-ietf-secsh-filexfer-13, Internet Engineering Task Force, July 2006. Work in Progress
[56] A. O. Freier, P. Karlton, and P. C. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0.” RFC 6101, August 2011, DOI 10.17487/RFC6101
[57] C. Allen and T. Dierks, “The TLS Protocol Version 1.0.” RFC 2246, January 1999, DOI 10.17487/RFC2246
[58] P. Eronen, Y. Nir, P. E. Hoffman, and C. Kaufman, “Internet Key Exchange Protocol Version 2 (IKEv2).” RFC 5996, September 2010, DOI 10.17487/RFC5996
[59] J. Vollbrecht, J. D. Carlson, L. Blunk, D. B. D. Aboba, and H. Levkowetz, “Extensible Authentication Protocol (EAP).” RFC 3748, June 2004, DOI 10.17487/RFC3748
[60] S. Beaulieu, G. Huang, and D. Rochefort, “A Traffic-Based Method of Detecting Dead Internet Key Exchange (IKE) Peers.” RFC 3706, February 2004, DOI 10.17487/RFC3706
[61] D. B. D. Aboba and J. Wood, “Authentication, Authorization and Accounting (AAA) Transport Profile.” RFC 3539, June 2003, DOI 10.17487/RFC3539
[62] OpenDaylight, “OpenDaylight Project: NEMO, User Manual.” https://wiki-archive.opendaylight.org/view/NEMO:User_Manual
[63] G. García, “Day-0, day-1 and day-2 configuration in OSM”, Zero Touch Congress, April 2018
[64] JavaCC Community, JavaCC Parser, https://javacc.github.io/javacc/#community
[65] “OpenDaylight Model-Driven Service Adaptation Layer (MD-SAL), Release 8.0.6”, release, OpenDaylight, September 2021. https://docs.opendaylight.org/_/downloads/mdsal/en/latest/pdf/
[66] JParamiko, A Python implementation of SSHv2, https://www.paramiko.org/index.html