POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Configurazione dinamica di Network Functions basata su modelli

Relatore
prof. Fulvio Risso

Tutore:
dott. Ivano Cerrato

Candidato
Giuseppe Piscopo

15 Dicembre 2017

Alla mia famiglia.

Indice

Elenco delle figure

Capitolo 1

Introduzione

La virtualizzazione delle funzioni di rete, nota come Network Functions Virtualization (NFV), ha introdotto un sostanziale cambio di paradigma nel modo in cui vengono realizzate le reti di telecomunicazioni, spezzando il legame tra hardware e software. Infatti, grazie ad NFV, le funzioni di rete (NAT, firewall, ecc) non sono più eseguite all'interno di dispositivi hardware proprietari, ma diventano applicazioni software istanziate all'interno di server; tali applicazioni prendono il nome di Virtual Network Functions (VNFs). Negli ultimi anni, l'interesse verso NFV si è diffuso sempre più tra i fornitori dei servizi di rete; l'idea è quella di sfruttare la versatilità offerta dal software per introdurre dinamicità nel dislocamento e nella gestione delle proprie funzioni di rete e dei servizi offerti agli utenti. Inoltre, l'utilizzo di tale tecnologia risulta vantaggiosa anche in termini economici, in quanto il consolidamento di più VNFs in un singolo server consente di ridurre sia i costi fissi (CAPEX) che quelli di gestione (OPEX).

In questo nuovo scenario prendono presto forma svariate soluzioni architetturali e nuovi componenti software necessari per poter sfruttare al meglio i vantaggi offerti da queste tecnologie; particolare enfasi viene data ad un architettura che prevede un componente denominato *orchestratore*, il cui compito è quello di coordinare l'allocazione di servizi generici su un'infrastruttura che si compone di nodi sempre più eterogenei, i quali mettono a disposizione risorse di rete, computing e potenzialmente di svariati altri tipi andando ad inglobare i settori della domotica e dell'*Internet of Things*.

Questa tesi si inserisce nell'ambito dell'orchestrazione di VNFs e si focalizza sulla parte relativa alla configurazione. In particolare viene proposto un framework che permette la configurazione e l'esportazione dinamica dello stato delle VNFs. Tale soluzione, da un lato offre al provider la possibilità di configurare a runtime i servizi istanziati utilizzando un linguaggio dichiarativo, dall'altro, fornisce degli strumenti che possono essere utilizzati per creare dei servizi che automaticamente siano in grado di monitorare e configurare le VNFs. Tale framework verrà validato all'interno di un architettura di orchestrazione già esistente, l'architettura FROG, definita dal

Netgroup del Politecnico di Torino, che consente di istanziare servizi di rete costituiti da VNFs. L'integrazione sarà possibile solo dopo avere esteso l'architettura aggiungendogli il supporto alla configurazione. Infine verranno sviluppati dei servizi che sfruttano le potenzialità del framework e che consentiranno di validarlo in use cases reali.

L'elaborato è strutturato come segue:

- **Capitolo 2**: espone il problema della configurazione e del monitoring dello stato e presenta la soluzione proposta da questa tesi.
- **Capitolo 3**: descrive lo stato dell'arte delle soluzione e le architetture utilizzate nel contesto della configurazione ed esportazione dinamica dello stato di VNFs.
- **Capitolo 4**: presenta una panoramica delle architetture sulle quali questa tesi si basa e dei componenti utilizzati.
- **Capitolo 5**: descrive l'architettura del FROG, ovvero l'architettura in cui è stato integrato il framework proposto.
- **Capitolo 6**: descrive le caratteristiche e il funzionamento del framework sviluppato.
- **Capitolo 7**: descrive gli uses cases utilizzati per validare il framework.
- **Capitolo 8**: descrive l'implementazione del framework e degli use-cases presentati.
- **Capitolo 9**: descrive i test effettuati e fornisce una descrizione dettagliata dei risultati ottenuti.
- **Capitolo 10**: espone le conclusioni del lavoro svolto i possibili lavori futuri.

Capitolo 2

Scenario

2.1 Il problema della configurazione e del monitoring dello stato

Per definire un servizio di rete in un ambiente virtualizzato, occorre creare un *service graph*; come mostrato in Figura 2.1 (in alto), il service graph è un grafo in cui i nodi rappresentano le VNFs, mentre gli archi rappresentano i link virtuali. Tali servizi, dopo essere stati definiti, necessitano di essere allocati all'interno dell'infrastruttura di rete, composta da nodi eterogenei, i quali mettono a disposizione risorse di rete e computing; in particolare le VNFs devono essere istanziate su risorse computazionali, mentre i virtual link devono essere mappati/implementati su link esistenti. A tale scopo, si sfrutta un componente chiamato *orchestratore*, il cui compito è quello di coordinare l'allocazione dei servizi sull'infrastruttura e di gestirne il ciclo di vita. Inoltre, analizzando i parametri di infrastruttura come ad esempio il consumo di CPU e memoria, l'orchestratore si occupa di effettuare scale in/out delle istanze di VNFs quando necessario.

Tuttavia, dopo che un servizio viene istanziato, le varie VNFs che lo compongono necessitano di essere configurate e monitorate dall'operatore di rete; in particolare occorre monitorare lo stato interno delle VNF in quanto è possibile che una VNF debba essere configurata dinamicamente a fronte di un cambiamento di stato o di un evento generato da un'altra VNF (si pensi ad esempio ad un firewall che debba essere configurato a seguito di un attacco rilevato dall'IDS). Inoltre, si tenga presente che ogni funzione di rete può essere implementata utilizzando metodiche e tecnologie differenti, quindi potenzialmente l'accesso ad ogni VNF per la configurazione ed il monitoring richiederebbe diverse API dipendenti dall'implementazione. Come conseguenza, il componente che si occupa della configurazione e del monitoring del servizio, necessiterebbe di conoscere tutti i comandi/formalismi di tutte le VNFs e dovrebbe essere aggiornato tutte le volte che una VNF venga modificata.

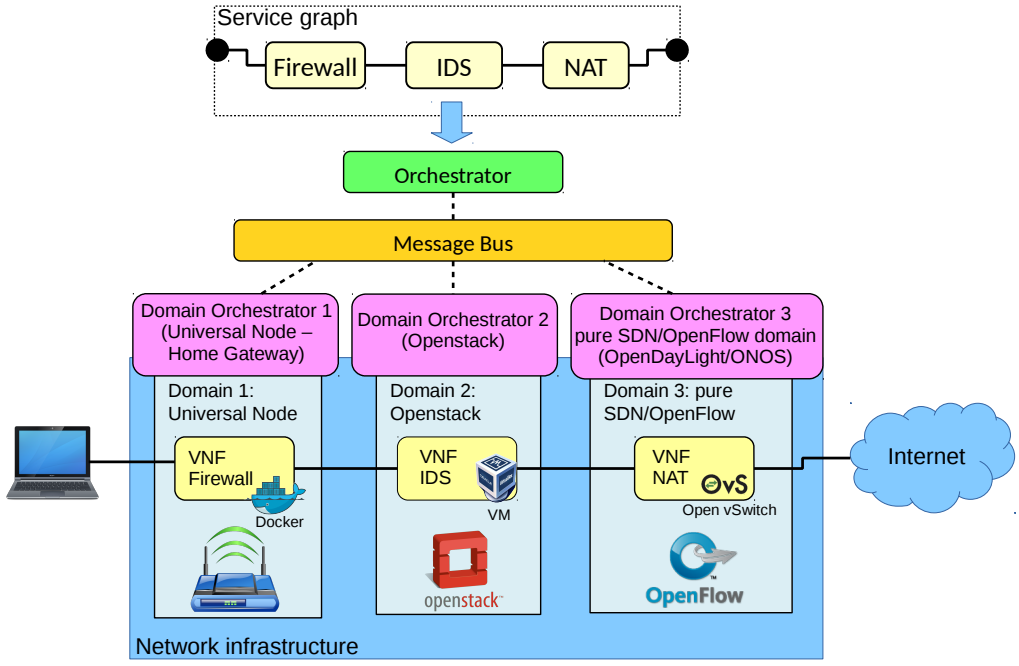


Figura 2.1. Panoramica dell'architettura di orchestrazione.

2.2 La soluzione proposta: un framework di configurazione dinamica

Per risolvere questo problema, in questa tesi si propone un framework che permette la configurazione ed il monitoring dello stato delle VNFs in maniera agnostica rispetto alle loro implementazioni e che fornisce una soluzione al problema che sussiste quando occorre configurare una VNF in base ai cambiamenti di stato di un'altra.

Facendo riferimento all'esempio di prima in cui il firewall deve essere configurato a fronte di un attaccco rilevato dall'IDS, occorrerebbe inserire all'interno della VNF dell'IDS la logica che permetta di configurare il firewall; l'IDS dovrebbe inoltre conoscere le API esposte dal firewall e ne risulterebbe strettamente dipendente. Utilizzando il framework proposto, è possibile utilizzare degli *orchestratori di servizio* che gestiscono la configurazione e il monitoring, facendo così in modo che le VNFs non debbano più doversi configurare a vicenda. Il sistema comprendente l'orchestratore di servizio e le VNFs utilizzate prende il nome di *servizio autoconfigurante*. L'obiettivo degli orchestratori di servizio è quello di monitorare lo stato di una o più VNFs e, a fronte di specifici eventi, in base alla logica di servizio che implementano, generare una configurazione da inviare a chi di dovere, eliminando il problema di 'sporcare' le VNF con funzionalità non strettamente legate alla loro funzione.

Inoltre tali orchestratori risultano agnostici rispetto all'implementazione delle VNFs interessate, per cui un orchestratore scritto per funzionare tra due tipologie di VNFs (come i già citati firewall e IDS), funzionerà indipendentemente da come queste siano implementate.

Questo è stato possibile definendo, per ogni tipologia di VNF (firewall, NAT, ecc), un *data-model* indipendente dall'implementazione che ne descriva i parametri di configurazione e di stato e che indichi come un'entità esterna (ad esempio l'orchestratore di servizio menzionato in precedenza) possa accedere a tali informazioni. Per la definizione dei modelli si utilizza il linguaggio YANG [1]. Il framework inoltre definisce un *piano di configurazione/monitoring* che abilità le VNFs e le entità esterne (ad esempio gli orchestratori di servizio) ad interagire tra di loro basandosi sul data-model associato alle VNFs. Per rendere compatibili le VNFs con il framework, ognuna di esse viene equipaggiata con un *agent* che permette di disaccoppiare la VNF e il data-model. Tale agent, infatti, da un lato espone una northbound interface che fornisce le API definite nel data-model, dall'altro implementa la logica che si occupa di tradurle nei corrispettivi comandi specifici dell'implementazione. L'agent si aspetta le configurazioni ed espone lo stato tramite il piano di configurazione/monitoring.

Capitolo 3

Related Work

Ci sono una serie di lavori in IETF che hanno influenzato/ispirato il lavoro realizzato in questa tesi. Ci sono anche numerosi lavori in letteratura che considerano il problema della gestione dello stato delle VNFs. In particolare, proposte come Split/merge[2], OpenNF[3], [4] e SliM[5] si concentrano nello scale in/out del numero di istanze di VNFs, un compito che richiede di muovere lo stato interno (o una parte) da un'istanza ad un'altra quando si aumenta (scale out) o riduce (scale in) il numero di istanze. Questi lavori propongono degli algoritmi efficienti che consentono di spostare informazioni basandosi sul flusso da un un'istanza ad un'altra, senza preoccuparsi ne di cosa ha richiesto lo scale in/out, ne di come questi dati vengano ricevuti. Inoltre, lo stato associato al flusso è sempre migrato tra due istanze identiche della stessa VNF. Considerazioni simili sono valide anche per Swing State[6], sebbene sia stato fatto su misura per funzionare su VNFs implementate attraverso il P4 language[7]. Il lavoro proposto invece, tramite l'associazione delle VNFs con un data-model indipendente dall'implementazione, definisce un framework per accedere in lettura e scrittura alla configurazione e allo stato interno della VNF. Esso potrebbe essere sfruttato dalle proposte di sopra, in modo da capire se ci sia bisogno di fare scale in/out basandosi su informazioni specifiche della VNF e non sono monitorano i parametri di infrastruttura quali RAM e CPU. In aggiunta il framework proposto, definisce i servizi autoconfiguranti in cui la lettura dello stato di una VNF può essere usato per creare una nuova configurazione per un'altra VNF.

Altri lavori che si possono menzionare sono Pico Replication[8] e the Stateless Network Function[9], che si occupano di condividere le informazioni di stato tra istanze multiple di VNFs allo stesso tempo. In particolare, mentre i primi propongono un framework che replica lo stato associato al flusso a diverse istanze di VNFs in modo da poter ripristinare lo stato in una nuova VNF in caso di crash, quest'ultimi disaccoppiano il processamento dello stato di una VNF, piazzandolo all'interno di un datastore centralizzato che può essere acceduto da tutte le istanze di VNFs. In questo modo, si aumenta la flessibilità e si semplifica lo scale in/out, dal momento che l'avvio di una nuova istanza di VNF richiede solamente lo spostamento di flussi,

essendo lo stato accessibile attraverso il datastore centralizzato.

Capitolo 4

Background

In questo capitolo vengono introdotte le principali tecnologie che sono state utilizzate per realizzare il framework proposto in questa tesi. Le prime due tecnologie presentate sono il *Software Defined Networking* e la *Network Functions Virtualization*, paradigmi che negli ultimi anni hanno introdotto una grande rivoluzione nel mondo delle reti; seppur nei capitoli precedenti si è più volte fatto riferimento ad esse, in questo capitolo sono state riservate due sezioni apposite in modo da chiarirne concetti fondamentali, motivazioni e ambiti applicativi.

4.1 Software Defined Networking

Il **Software-defined networking** (SDN) è un approccio moderno al problema del *Computer Networking*. Esso ha lo scopo di affrontare e colmare le carenze da sempre presenti nell'architettura statica delle reti tradizionali, ossia l'assenza di dinamicità, scalabilità e computing richiesti in ambienti informatici attuali quali ad esempio i data center.

Il concetto fondamentale che sta alla base di questo paradigma è il disaccoppiamento tra quelli che, in ambito networking, vengono comunemente chiamati *"control plane"* e *"data plane"*. Il *control plane* è il sistema che prende decisioni riguardanti dove il traffico deve essere inviato e solitamente tale processo si completa con la popolazione di una struttura contenente le decisioni prese (come ad esempio la tabella di routing); il *data plane* è il componente sottostante il cui compito è quello di instadare il traffico verso il nodo successivo utilizzando le informazioni contenute nella struttura generata dal control plane.

Questo disaccoppiamento permette allo strato di controllo di diventare direttamente programmabile e all'infrastruttura sottostante di essere astratta verso le applicazioni ed i servizi di rete [10].

Essenzialmente ciò si traduce in un controller centralizzato che gestisce una moltitudine di dispositivi di rete molto semplici, sui quali viene implementata solamente

la logica di data plane; il comportamento di questi dispositivi risulta facilmente configurabile via software tramite il controller, il quale si interfaccia con essi attraverso una interfaccia di southbound e al contempo ne esporta una di northbound, tramite la quale espone delle API che consentono ad eventuali applicazioni di manipolare agevolmente lo stato ed il comportamento dell'infrastruttura fisica sottostante.

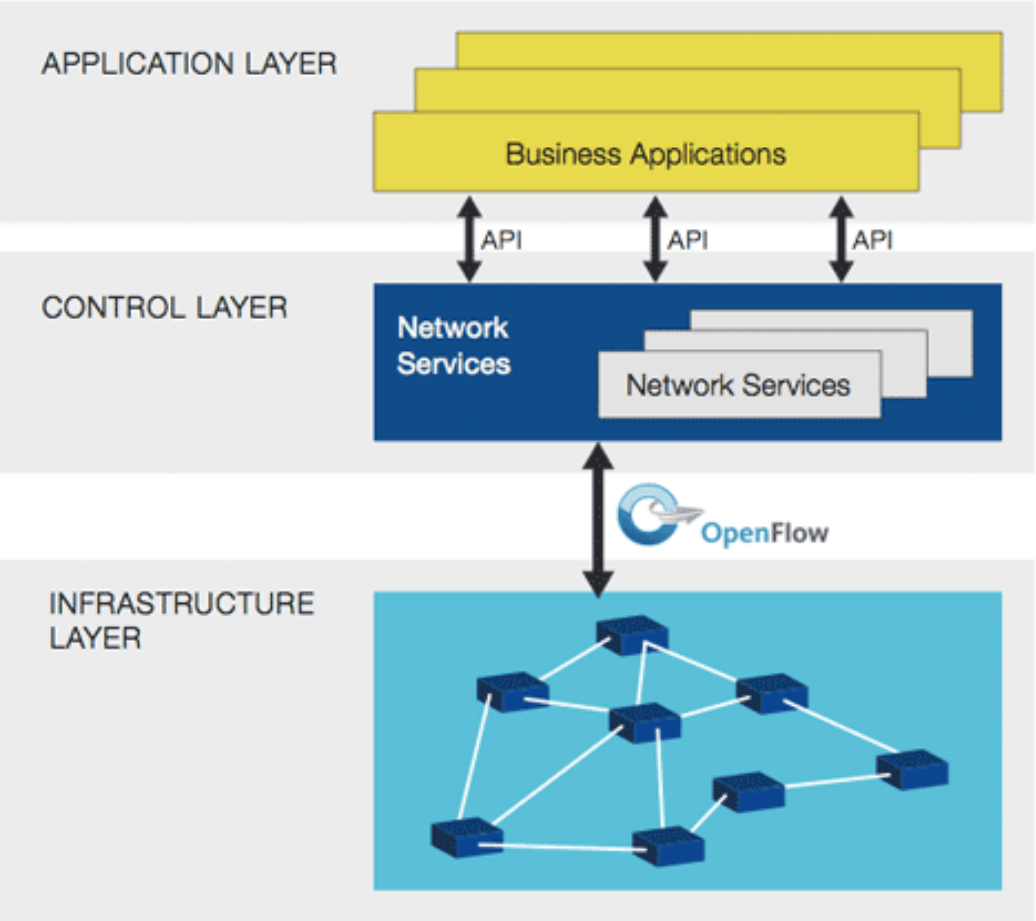


Figura 4.1. Architettura generale SDN (Fonte: Open Networking Foundation [10]).

I benefici introdotti da questa architettura sono molteplici e di impatto sostanziale. Il controller risulta direttamente programmabile, in quanto completamente disaccoppiato dalle funzioni di forwarding; la rete può essere configurata, gestita, resa sicura, ottimizzata e personalizzata tramite una serie di procedure automatizzate semplici da preparare sotto forma di applicazioni. L'infrastruttura, il controllore e le applicazioni, diventando dei componenti completamente separati, possono evolvere rapidamente ed in maniera indipendente; di fatto questo è ciò che avviene da decenni in ambito IT ed è ciò che ha favorito una veloce evoluzione dei moduli hardware e software dei computer rispetto al mondo delle reti. La gestione della

rete diventa centralizzata, grazie ai controller i quali mantengono una visione globale della stessa che appare dall'esterno come un singolo logico grande switch. Le applicazioni, non dovendo avere a che fare con la complessità dell'infrastruttura fisica, possono definire ad alto livello le politiche comportamentali che il traffico deve seguire. In più l'architettura SDN viene implementata attraverso standard aperti, cosa che semplifica il design della rete a causa del fatto che le istruzioni vengono definite e fornite dai controllori SDN piuttosto che da una moltitudine di dispositivi e protocolli dipendenti dal particolare venditore.

Lo strato di controllo si comporta dunque come una sorta di sistema operativo di rete, che astrae il livello fisico e consente a delle semplici applicazioni, che chiunque può scrivere quando se ne ha la necessità, di utilizzarlo in maniera opportuna; esso si occupa di trasformare le politiche di alto livello così definite in comandi di basso livello verso i dispositivi fisici. L'implementazione attualmente più diffusa dell'interfaccia che permette al controller di comunicare con il livello fisico è *Openflow* [11]. Lo strato di controllo può essere monolitico o costituito da un insieme di componenti ciascuno dei quali adibito a gestire funzionalità precise; in questo secondo caso i vari componenti possono essere dislocati in più nodi, realizzando di fatto un controller distribuito.

In una architettura SDN, l'interfaccia di southbound può generalmente essere utilizzata seguendo due differenti paradigmi per istanziare i flussi sui dispositivi di rete:

- *paradigma reattivo* – all'arrivo di un pacchetto su un dispositivo, se la tabella di quest'ultimo non contiene nessuna regola che combacia col pacchetto in ingresso, questo viene mandato al controller, che lo elabora (a seconda di una qualche logica applicativa) e genera le regole corrette istanziandole in tutta la rete; in questo modo da quel momento in poi tutti i pacchetti di quel tipo verranno trattati tramite quelle regole di flusso, senza avere nuovamente il bisogno di recapitare ciascuno di essi al controllore;

- *paradigma proattivo* – invece di "reagire" all'arrivo di un pacchetto, le tabelle dei flussi vengono popolate in anticipo con delle regole capaci di gestire tutti i match di traffico che si prevede possano attraversare la rete. Questo approccio è simile a quello utilizzato fino ad ora nelle reti tradizionali (si pensi alla tipica tabella di routing). Il risultato è che tutti i pacchetti vengono trasmessi alla massima velocità consentita dall'apparato di rete, ma diventa scomodo introdurre logiche particolarmente complesse e che richiedono una certa dinamicità.

Questa nuova tecnologia introduce un sostanziale cambiamento nel modo di concepire le reti di calcolatori, e apre le porte a nuove interessanti architetture di networking ed a nuove sfide riguardanti la soluzione dei problemi che ovviamente si presentano in uno scenario ancora acerbo, ma promettente e stimolante.

4.2 Network Functions Virtualization

La **Network Functions Virtualization** è una architettura di networking che utilizza le tecnologie di virtualizzazione IT per realizzare tramite macchine virtuali intere classi di servizi e funzionalità di rete fino ad ora implementate attraverso hardware dedicato. Le macchine virtuali possono così essere organizzate e consolidate all'interno di server situati in data center, nodi di rete o nei dispositivi dell'utente, in quanto la maggior parte dei dispositivi con capacità di computing è capace di eseguirle.

Le funzionalità convertite in questo modo vanno dalle tipiche funzioni delle reti informatiche (routing, traduzione degli indirizzi, ecc...) fino ai servizi collaterali (sicurezza, acceleratori web, ecc...).

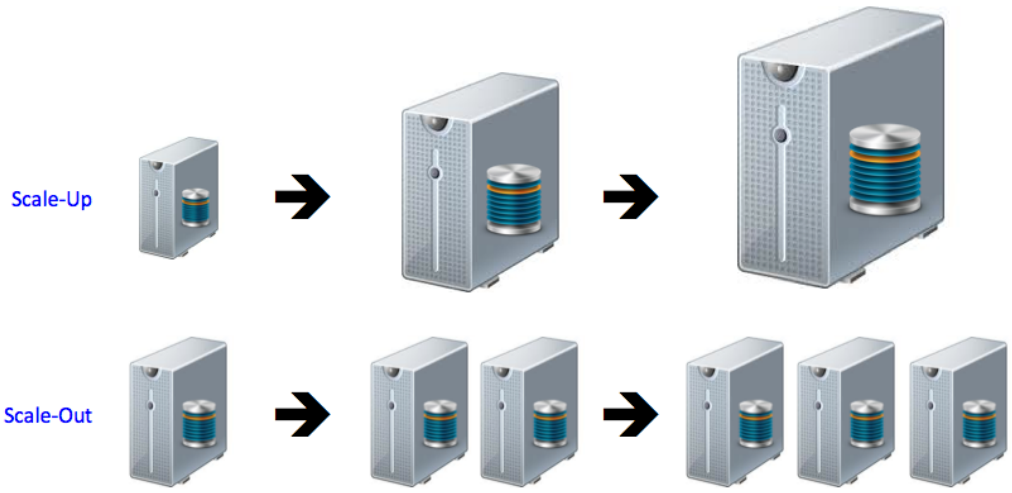


Figura 4.2. Metodi di scalabilità offerti dalla virtualizzazione: verticale (scale up) e orizzontale (scale out).

I vantaggi derivanti da questo approccio sono innumerevoli, dalla flessibilità e riduzione dei costi fino ad una migliore integrazione tra il mondo delle telecomunicazioni e quello delle tecnologie informatiche. Il principale beneficio è che molteplici funzioni di rete, che prima avevano bisogno di altrettanti dispositivi fisici, possono adesso essere accorpate sullo stesso server pur mantenendo un totale isolamento tra esse grazie alla tecnica della virtualizzazione. In questo modo si ha la possibilità di ridurre i costi legati all'infrastruttura e al contempo di massimizzarne l'utilizzo, viste le particolari proprietà intrinseche di scalabilità di cui godono le macchine virtuali. Esse possono infatti essere scalate in due modi diversi: *verticalmente*, cioè aumentando le risorse allocate alla singola macchina virtuale sul server fisico, oppure *orizzontalmente*, attivando multiple istanze su più server fisici in maniera distribuita (Figura 4.2).

Un altro grande vantaggio introdotto dall'adozione di questa tecnologia è la possibilità di istanziare dinamicamente i servizi senza particolari vincoli geografici, e di de-istanziare le VNF relative quando il servizio non è più richiesto, liberando le risorse che gli erano state allocate.

4.3 DoubleDecker, un framework di message brokering distribuito

DoubleDecker [12] è un sistema di messaggistica gerarchico distribuito basato su *ZeroMQ* che può essere utilizzato per consentire lo scambio di informazioni tra processi situati sulla stessa macchina o su macchine multiple. È gerarchico nel senso che i message broker (che si occupano dello smistamento dei messaggi) sono connessi tra loro in una topologia ad albero ed inviano i messaggi verso l'alto nel caso in cui il client destinazione non si trovi sotto essi (Figura 4.3).

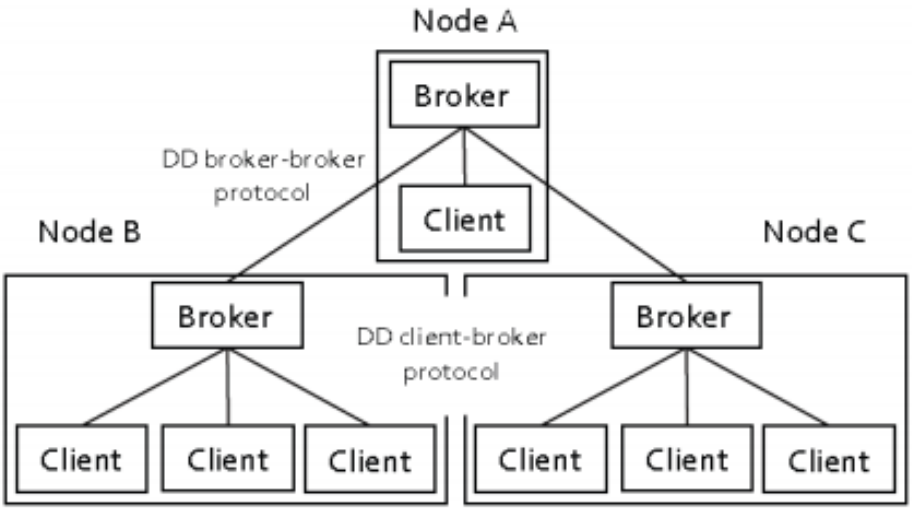


Figura 4.3. L'architettura gerarchica di DoubleDecker (Fonte: Unify [13]).

Attualmente DoubleDecker supporta due tipi di messaggistica, *Notifications*, cioè messaggi punto punto da un client ad un altro, e *Publish/Subscribe* su un topic. Il meccanismo di Pub/Sub per di più permette di restringere lo scope della sottoscrizione. Questo significa che un client può decidere di ricevere solo i messaggi di un determinato topic che rientrano in un determinato scope, ad esempio inviati da client collegati direttamente allo stesso broker, ad uno specifico o a gruppi di essi.

Sono supportati gruppi multipli di client, detti "tenant", l'appartenenza ai quali è verificata tramite una autenticazione a chiave asimmetrica. I messaggi appartenenti a client di un determinato tenant sono cifrati e non possono passare da un tenant all'altro. Esiste un tenant speciale detto "pubblico" che può oltrepassare questo limite, e può essere utilizzato per connettere nodi il cui scopo è quello di fornire/utilizzare un servizio pubblico.

In questa tesi DoubleDecker è stato utilizzato come message broker, in modo da rendere possibile la comunicazione dei vari componenti dell'architettura che sarà presentata in seguito. Il paradigma publish/subscribe si presenta particolarmente scalabile ed assume un valore chiave in un contesto altamente dinamico come quello presentato, in quanto evita che un attore con l'intenzione di comunicare una informazione debba preoccuparsi di chi effettivamente necessita di ricevere quei dati e delle sue coordinate. Dunque è il consumatore del dato che notifica il suo interesse verso un particolare argomento, mentre il produttore deve solo preoccuparsi di etichettarlo con il topic corretto e di *pubblicarlo* sul bus, lasciando che il broker si occupi di smistarlo a tutti coloro che ne hanno bisogno.

4.4 YANG data modeling

YANG [1] è un linguaggio di modellazione dati nato per la definizione del formato dei dati inviati attraverso il protocollo NETCONF (network configuration protocol [14]). Può essere utilizzato sia per modellare dati di configurazione sia dati di stato dei componenti di rete. Le istanze di dati che devono seguire i modelli definiti tramite questo linguaggio possono essere codificate in qualunque modo, ad esempio XML o JSON.

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

I componenti di un modello YANG sono elencati di seguito.

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

Un modulo può contenere più **submodules**. La direttiva *include* permette di referenziare elementi appartenenti ad un altro submodule, mentre *import* consente di farlo con elementi appartenenti ad un altro modulo.

- I **nodi** sono gli elementi che contengono i dati, e possono essere di quattro tipi diversi:
  - **leaf**, contiene un dato primitivo, come un intero o una stringa, e non ha nodi figli;
  - **leaf-list**, consiste in una lista di nodi leaf (tutti dello stesso tipo);
  - **container**, è un nodo che non può assumere un valore, ma può contenere altri nodi, siano essi leaf o altri container;
  - **list**, consiste in una lista di nodi container (tutti dello stesso tipo).
- I **grouping** sono utilizzati per definire strutture dati riutilizzabili più volte, sia all'interno del modulo stesso sia in moduli esterni, tramite la direttiva **uses**. Appena si utilizza un grouping, è possibile modificarne alcuni elementi tramite la direttiva **refine**.
- Una **choice** permette di definire che in un determinato punto del modello si possano trovare, in maniera esclusiva, nodi di diverso tipo.
- Gli elementi **augment** definiscono l'estensione di moduli esterni da parte del modulo corrente. Bisogna specificare il path del modulo da estendere, i nodi da aggiungere e dei vincoli che permettono di limitare l'estensione solo a particolari casi.

Nel corso del lavoro di tesi il linguaggio YANG è stato utilizzato per descrivere i data-model delle VNFs.

Capitolo 5

Il sistema di orchestrazione FROG

In questo capitolo viene presentata l'architettura del Frog, un sistema di orchestrazione distribuito sviluppato presso il Politecnico di Torino che costituisce lo scenario all'interno del quale è stato sviluppato il lavoro di tesi. Il framework presentato e sviluppato in questo elaborato sarà utilizzato all'interno di questa architettura, dopo averla opportunamente estesa.

5.1 Una implementazione dell'architettura ETSI MANO

Recentemente, il progetto europeo Unify [13] ha proposto una architettura multi livello che, sfruttando diversi livelli di astrazione, è capace di distribuire ed orchestrare servizi generici di rete sull'infrastruttura fisica degli operatori telefonici. Questa architettura segue molto da vicino quella proposta dal workgroup ETSI NFV ed è stata lavoro del gruppo di ricerca *Netgroup* del Politecnico di Torino. L'architettura generale, come mostrato in Figura 5.1, è stratificata in tre componenti principali, cioè il *service layer*, l'*orchestration layer* e l'*infrastructure layer*.

5.2 Il Service Layer

Il *service layer* rappresenta il componente di più alto livello del Frog, ed è pensato per consentire a molteplici utenti di definire i propri servizi di rete. Per farlo utilizza una descrizione complessiva del servizio espressa in un formalismo di alto livello chiamato *service graph*, il quale consente la definizione di servizi generici e il modo in cui essi interagiscono tra loro [15]. Il service graph supporta la definizione di alcuni parametri non funzionali; in particolare è possibile specificare alcuni requisiti qualitativi che il servizio deve soddisfare, come la massima latenza tra due VNF o la

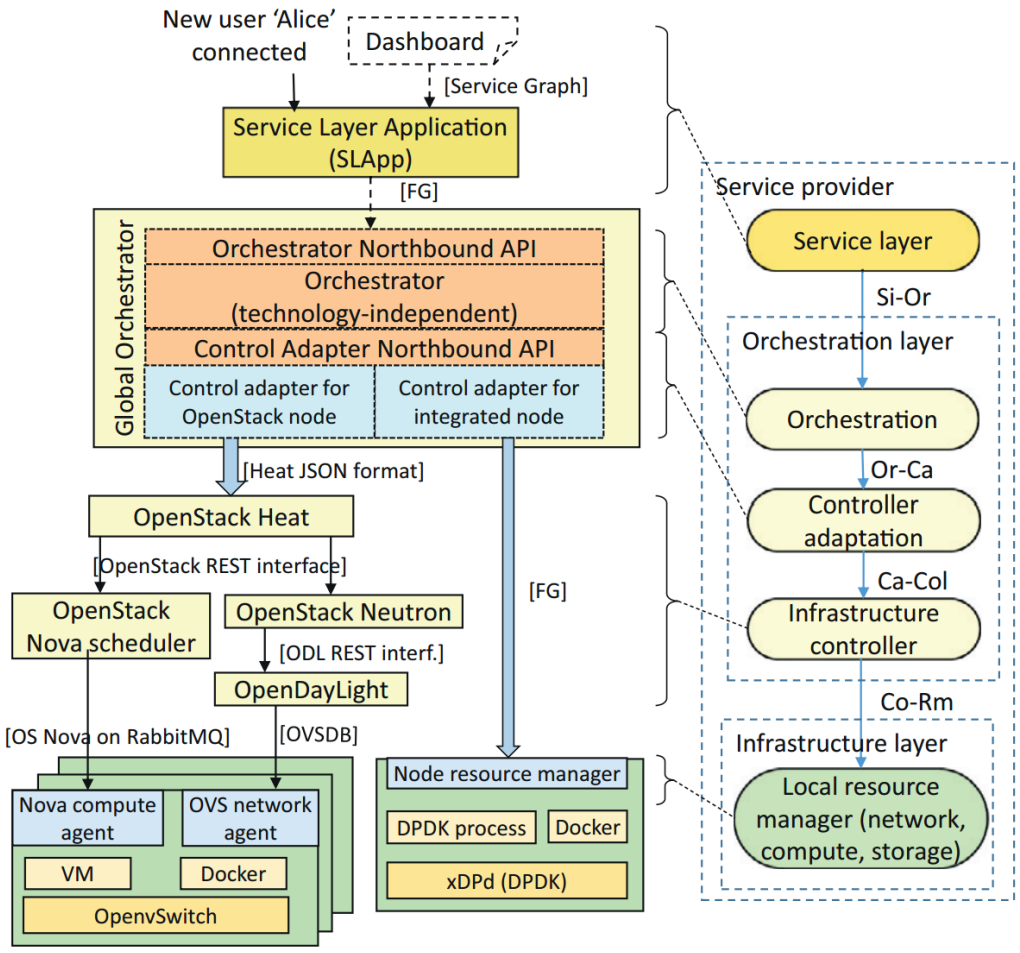


Figura 5.1. Visione d'insieme del sistema, incluse due diverse implementazioni di infrastructure layer.

latenza massima dell'intero servizio. In più è prevista la definizione di una lista di politiche di alto livello da prendere in considerazione durante la fase di istanziazione del servizio (ad esempio il requisito di allocare il servizio all'interno di una particolare area geografica).

Il service layer ha anche il compito di tradurre il service graph in un formalismo orientato all'orchestrazione, chiamato *forwarding graph*. Questa nuova rappresentazione fornisce una più precisa vista del servizio che deve essere allocato, sia in termini di computing che di risorse di rete, vale a dire VNF e interconnessioni tra loro, mantenendo gli indicatori di qualità e le politiche specificate dall'utente che ha definito il servizio.

Come mostrato in Figura 5.1, questo strato include un componente che implementa la logica di servizio, identificato con il blocco denominato service layer

application (SLApp) nello schema. In più il service layer esporta delle API che consentono agli altri componenti di notificare il verificarsi di alcuni eventi specifici nell'infrastruttura di basso livello. In questo modo il modulo SLApp può reagire a questi eventi in maniera tale da implementare la logica di servizio richiesta dallo specifico caso d'uso. Ad esempio uno di questi eventi può consistere in un dispositivo utente (come uno smartphone) che si connette alla rete, evento che potrebbe scatenare l'istanziazione di un grafo di servizio rappresentante le funzionalità in precedenza richieste da quell'utente, o l'aggiornamento di un eventuale servizio già istanziato (Figura 5.2).

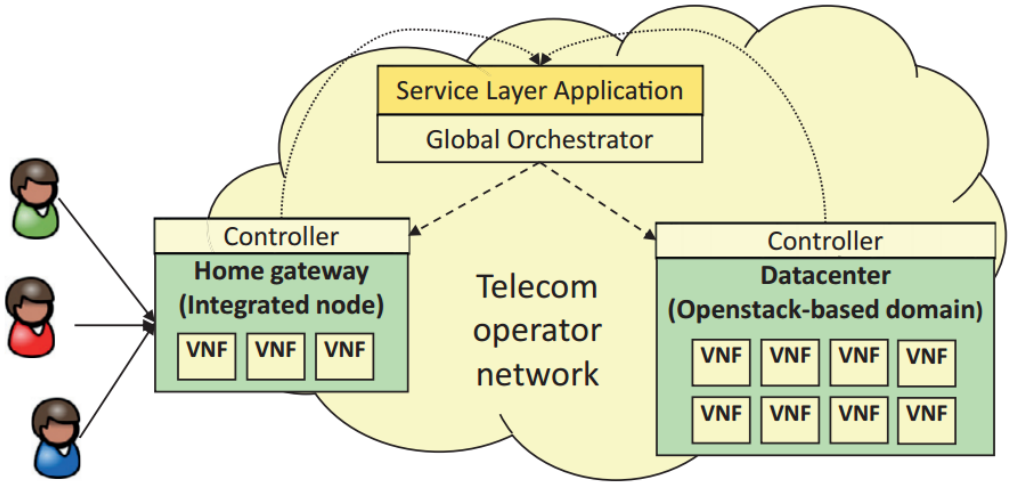


Figura 5.2. Rappresentazione del caso d'uso in cui l'operatore di rete fornisce i servizi richiesti ad utenti ubicati a valle di speciali home-gateway detti "integrated nodes". La presenza di un nuovo utente viene catturata da questi dispositivi come *evento* e notificata al service layer, che provvede a recuperare il relativo servizio ed a richiederne l'istanziazione all'orchestratore.

5.3 Orchestration Layer

L'*orchestration layer* si colloca immediatamente sotto il service layer ed è responsabile di due importanti fasi riguardanti il deployment del servizio [15].

Per prima cosa, manipola il forwarding graph in modo da permettere che esso venga istanziato sull'infrastruttura, adattando la definizione del servizio alle capabilities del livello infrastrutturale. In secondo luogo, questo strato implementa lo scheduler che è responsabile delle decisioni riguardanti dove istanziare le componenti che implementano il servizio richiesto. Il processo di scheduling può basarsi su differenti classi di parametri: *(i)* informazioni riguardanti le VNF, come CPU e memoria richieste; *(ii)* politiche di alto livello e requisiti sulla qualità del servizio

forniti con il forwarding graph; *(iii)* risorse disponibili sull'infrastruttura fisica, come la presenza di un determinato acceleratore hardware in un certo nodo, così come il carico attuale dei nodi stessi.

In accordo con la Figura 5.1, l'orchestration layer si compone di tre differenti sottolivelli logici.

- Il sub-strato detto *di orchestrazione* implementa la trasformazione del forwarding graph e lo scheduling tramite un approccio indipendente dalla tecnologia fisica, senza quindi preoccuparsi dei dettagli relativi alla particolare infrastruttura.
- il sub-strato di *controller adaptation* è diverso per ogni tipo di infrastruttura ed implementa una logica ad hoc per ciascuna di esse. Si occupa di tradurre il forwarding graph nell'appropriato set di chiamate verso le API di northbound del relativo *infrastructure controller*, che costituisce la parte inferiore dell'orchestration layer.
- l'*infrastructure controller* ha il compito di applicare i comandi di cui sopra sui nodi costituenti l'infrastruttura fisica; tale set di comandi viene chiamato *infrastructure graph* e cambia in base al nodo/dominio fisico che ospiterà il servizio che si sta per istanziare. Questo sub-strato ha anche il compito di catturare gli eventi che si verificano nell'infrastruttura fisica contollata e di notificarli ai livelli superiori.

Come mostrato in Figura 5.1, ciascun dominio ha bisogno di un suo controller adaptation e del relativo infrastructure controller. In questo testo si farà spesso riferimento al sub-strato di orchestrazione indipendente dalla tecnologia col nome di *Global Orchestrator*, in quanto di fatto è l'unica parte dell'orchestratore che sovraintende più domini differenti. Allo stesso modo ci si riferirà ai due sub-strati inferiori come *Domain Orchestrator*, in quanto ciascuno di essi si colloca logicamente su un singolo dominio.

Questa architettura di orchestrazione consente di istanziare un forwarding graph su infrastrutture multiple, attraverso una opportuna procedura di splitting del grafo stesso attuata dall'orchestratore globale.

5.4 Infrastructure Layer

L'*infrastructure layer* è situato sotto lo strato di orchestrazione e contiene tutte le risorse fisiche che ospiteranno effettivamente il servizio una volta istanziato [15]. Include differenti nodi (o domini), ciascuno dei quali è gestito da un proprio orchestratore di dominio (o locale) come spiegato alla fine del paragrafo precedente.

Vista l'eterogeneità delle reti moderne viene prevista la possibilità di avere molteplici nodi implementati con tecnologie differenti.

- la prima classe consiste in domini di cloud-computing come quello *Open-stack based* presente in Figura 5.1; ciascuno di essi consiste in un cluster di macchine fisiche gestite da un singolo controllore centralizzato.

- la seconda classe di risorse è invece completamente distaccata dai tradizionali scenari di cloud-computing e rappresenta nodi come i futuri home-gateway situati nelle case degli utenti. Lo schema di un nodo di questo tipo, detto *integrated node* o *universal node* [16], è mostrato in basso a sinistra in Figura 5.2 e consiste in una singola macchina fisica fornita principalmente con software sviluppato all'interno del progetto Frog stesso e che integra nello stesso host anche lo strato di orchestrazione locale.

5.5 Il grafo delle funzioni di rete

5.5.1 Service Graph

Il **Service Graph (SG)** è una rappresentazione ad alto livello del servizio che include sia aspetti relativi alle network functions che implementano il servizio (come ad esempio quali sono e come devono essere interconnesse tra loro) sia aspetti legati alla configurazione di queste funzioni [15]. Gli elementi base che costituiscono il SG (mostrati in Figura 5.3) sono descritti di seguito.

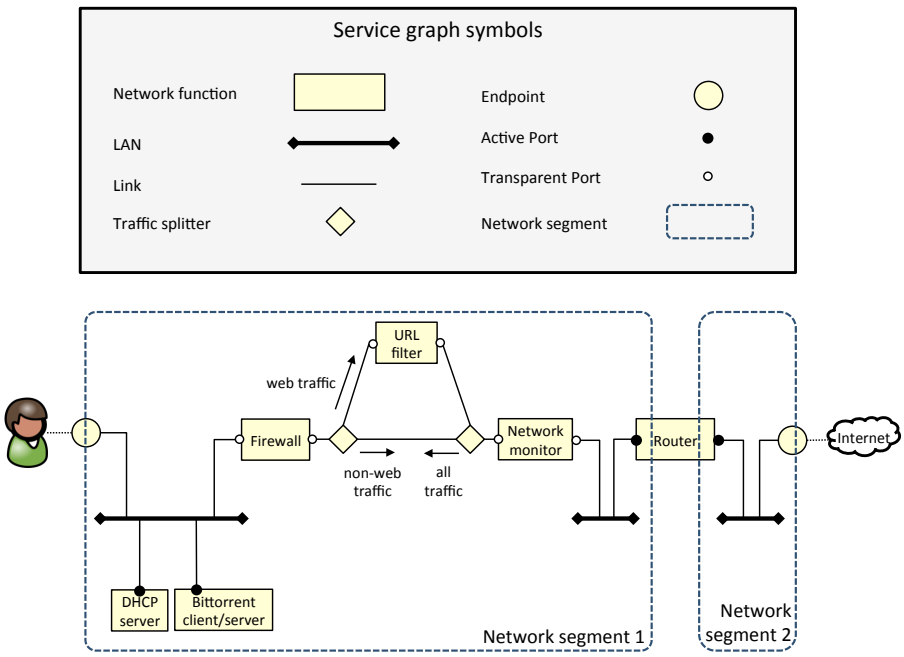


Figura 5.3. Esempio di service graph e insieme dei componenti base.

Quello detto **network function** (NF) è un blocco funzionale che verrà in seguito tradotto in una (o più) VNF (o anche in un componente hardware dedicato o in una applicazione SDN, in seguito al lavoro di questa tesi). Ciascuna NF è associata ad un template che la descrive in termini di interfacce e risorse richieste.

Le **porte** costituiscono le interfacce delle NF, e possono essere **attive** (se richiedono un indirizzo di rete) o **trasparenti**.

La **LAN** rappresenta il mezzo logico di comunicazione di tipo broadcast. La disponibilità di questa primitiva facilita la creazione di servizi complessi che includono non solo NF trasparenti, ma anche tradizionali servizi host-based.

Il **link** (point-to-point) definisce il collegamento logico tra differenti componenti e può essere utilizzato per connettere tra loro porte, endpoint, LAN, eccetera.

Il **traffic splitter/merger** è un blocco funzionale che permette di suddividere il traffico in una serie di flussi in base a delle regole specifiche, o di unificare quello proveniente da link differenti.

Infine, l'**endpoint** rappresenta un punto esterno a cui è agganciato il SG. Può trattarsi sia di una entità logica, sia di una porta specifica (ad esempio una NIC fisica/virtuale, un tunnel di rete, etc.), purché sia attiva su un dato nodo dell'infrastruttura fisica. Un endpoint può essere utilizzato per collegare il SG ad internet, ad un dispositivo dell'utente finale, ma anche all'endpoint di un altro service graph qualora occorresse collegarne più di uno in cascata per creare un servizio più complesso.

Come accennato in precedenza, il SG include anche aspetti relativi alla configurazione delle network functions, i quali rappresentano importanti parametri dello strato di servizio che vanno definiti assieme alla topologia e che possono essere utilizzati dal piano di controllo dell'infrastruttura di rete per configurare in maniera opportuna il servizio. Questi parametri riguardano aspetti di rete come gli indirizzi da assegnare alle porte delle NF, così come configurazioni specifiche delle funzioni di rete stesse.

5.5.2 Forwarding Graph

Il formalismo definito dal service graph è di alto livello e non adatto alla fase di allocazione del servizio sull'infrastruttura fisica, in quanto non include tutti i dettagli di cui questo ha bisogno per operare [15]. Dunque si ha la necessità di trasformarlo in una rappresentazione più orientata alle risorse, cioè nel **Forwarding Graph (FG)**, attraverso un processo detto di *"lowering"* (o di espansione) le cui fasi sono riportate in Figura 5.4.

Nel primo step, detto di **espansione di controllo e di gestione**, viene aggiunta una LAN di configurazione a cui vengono collegate tutte le NF che presentano una vNIC dedicata identificata come *interfaccia di controllo*; questa rete ha il compito di garantire la possibilità di monitorare e configurare opportunamente le NF e può

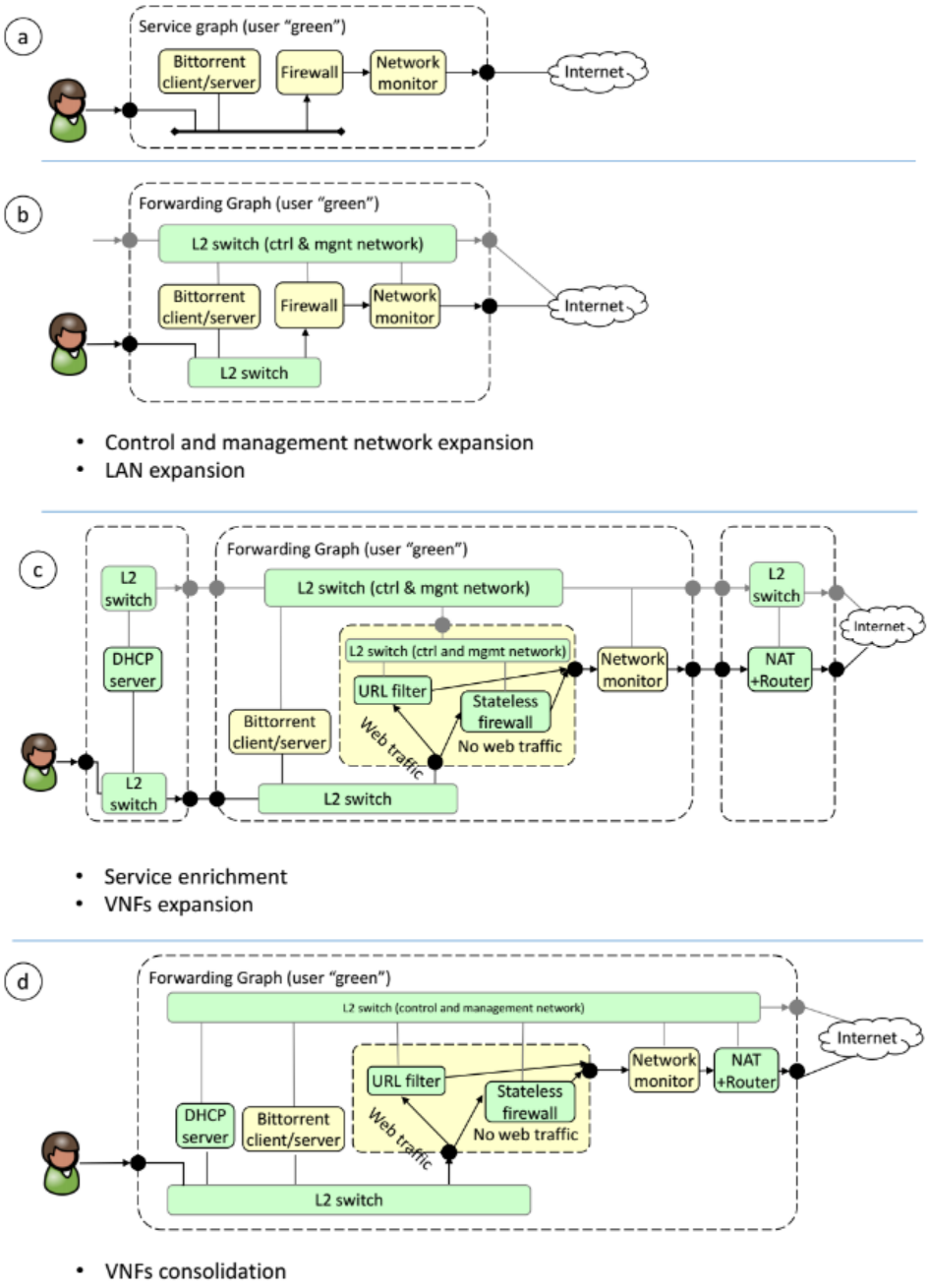


Figura 5.4. Dal service graph al forwarding graph: il *processo di espansione*.

essere un dettaglio non importante per l'utente che richiede il servizio (e quindi non presente nell'astrazione del SG), ma richiesto affinché il servizio stesso sia in grado di operare.

Nel secondo step avviene la **LAN expansion**, la quale traduce l'elemento LAN astratto definito nel SG in un appropriato set di VNF che emulano il mezzo di comunicazione broadcast. I primi due step sono visibili nel passaggio da Figura 5.4(a) a Figura 5.4(b).

La fase di **service enrichment** analizza ed arricchisce il grafo con quelle funzioni che non sono state inserite nel SG dall'utente, ma sono comunque richieste per la corretta implementazione del servizio. In Figura 5.4(c) l'analisi del grafo determina che il segmento di rete che connette il grafo utente ad internet non presenta alcun servizio di NAT e routing, e che non è presente nessun server DHCP all'interno della rete locale; il set appropriato di NF viene automaticamente aggiunto.

La **NFs expansion** può sostituire una network function con altre equivalenti, o con un intero sottografo equivalente, in modo da implementare il servizio richiesto. In Figura 5.4(c) il firewall è stato sostituito con un sottografo i cui endpoint coincidono con le porte della vecchia NF.

La fase di **NFs consolitation** analizza il FG cercando eventuali funzioni ridondanti ed ottimizzando il grafo dove possibile. Per esempio, la Figura 5.4(d) mostra come dome due switch software connessi insieme siano stati rimpiazzati da un unica NF, riducendo le risorse richieste per implementare il grafo sull'infrastruttura fisica.

La **endpoint translation** converte gli endpoint del grafo nelle appropriate porte fisiche del nodo in cui il servizio deve essere istanziato, nei tunnel di rete (ad esempio GRE) che connettono il grafo ad un altro istanziato in un dominio esterno, e così via.

Lo step finale è quello della **flow-rules definition**, in cui le connessioni tra NF, le regole definite negli splitter/merger del SG e le regole di ingresso associate ad ogni endpoint vengono tradotte in una serie di coppie match-azione, seguendo un paradigma SDN-based.

5.5.3 Infrastructure Graph

La rappresentazione finale del servizio da istanziare è detta **Infrastructure Graph (IG)**; esso è semanticamente equivalente al FG, ma sintatticamente diverso [15]. L'IG è ottenuto tramite il processo di **riconciliazione**, il quale mappa il FG sulle risorse disponibili nel dominio, e consiste nella sequenza di comandi che devono essere eseguiti sull'infrastruttura fisica in modo da istanziare, configurare e connettere correttamente tra loro tutte le VNF richieste.

Capitolo 6

Framework di configurazione dinamica

In questo capitolo verranno spiegati singolarmente tutti i componenti del framework.

6.1 Network Function Data Model

Ogni VNF è associata ad un data-model, scritto utilizzando il linguaggio YANG [1]. Il data-model è indipendente dall'implementazione della VNF. Esso definisce la sintassi dei parametri di configurazione e di stato che la VNF espone all'esterno e così come le API che permettono di interagire con la VNF per accedere a tali parametri. Un esempio di data-model che descrive il NAT è mostrato nella Figura 6.1, mentre nella Figura 6.2 sono presenti un esempio di stato esportato (a sinistra) e di configurazione (a destra) della VNF, entrambi definiti nel data-model. Dall'immagine si può notare che i dati che derivano dal modello sono codificati in JSON.

Come mostrato nella Figura 6.1, oltre agli elementi base di un modello YANG (container, leaf, leaflist, list), sono stati definiti dei nuovi statement per aumentare le funzionalità del modello. Questo è stato possibile in quanto YANG permette di creare delle *extension* che consentono di aggiungere dei nuovi attributi. E'stato definito l'attributo `atomic` che indica se l'elemento deve essere acceduto atomicamente `(atomic true)` o no `(atomic false)`. Per esempio il nodo *interface* in figura è atomico, per cui un modulo esterno non può accedere ai singoli elementi separatamente (es. *netmask*) ma solo all'oggetto *interface* intero. Ogni nodo è inoltre associato allo statement `config`, che indica se i parametri del nodo sono read/write oppure solo read-only. In Figura 6.1 si nota che la lista delle nat-session essendo un elemento di stato non è configurabile ma è solo leggibile *(config false)*, al contrario l'elemento *interface* essendo configurabile può essere sia letta che scritta `(config true)`.

Inspirandoci ad alcuni recenti lavori di IETF [17], ad ogni nodo atomico è stato associato un nuovo statement, `advertise`, che indica come la VNF espone il valore

```
module nat{
 ...
  list nat-session {
    atomic true;
    config false;
    advertise periodic;

    advertise-period 1;
    advertise-unit s;
    leaf protocol {
      type proto-type
    }
    leaf src_address {
      type inet:ip-address;
    }
    leaf src_port {
      type port-type;
    }
    leaf translated_address {
      type inet:ip-address;
    }
    leaf translated_port {
      type port-type;
    }
  }
...
```

```
...
container interface{
    key "name";
    config true;
    advertise onchange;
    atomic true;
    leaf name {
      type string;
    }
    container ipv4-config{
      leaf address{
        type inet:ipv4-address;
      }
      leaf netmask{
        type inet:netmask;
      }
      leaf mtu{
        type integer;
      }
    }
    container ipv6_config{
      ...
    }
  }
}
```

Figura 6.1. Esempio di YANG Data-Model che descrive un NAT.

del nodo all'esterno. Sono stati definiti quattro tipi di advertisement: *onchange, onthreshold, periodic* e *ondemand*. Un nodo **onchange** rappresenta un informazione che viene esportata non appena viene creata, modificata o cancellata. E'usata principalmente nei parametri di stato i cui valori cambiano raramente, tipo la configurazione di un'interfaccia di rete, come mostrato nell'esempio in Figura 6.1. Per gli elementi che invece vengono esportati solo quando il valore raggiunge una determinata soglia si utilizza il tipo **onthreshold**; per questo tipo il data model definisce anche il valore minimo e/o massimo che causa l'esportazione dell'elemento. **Onthreshold** risulta utile nei data-model che descrivono funzioni relative al mondo *Internet of Things (IoT)*, per esempio nel caso di un nodo che descrive la temperatura misurata da un sensore è utile esportare il valore solo quando questo eccede una determinata soglia oppure si trova all'interno di un intervallo. E' definito `periodic` un nodo il cui valore è esportato periodicamente, indipendentemente dal fatto che sia cambiato o no; in questo caso il data-model definisce anche il periodo `(period)`. Questo advertisement è utilizzato soprattutto nei nodi il cui valore cambia molto

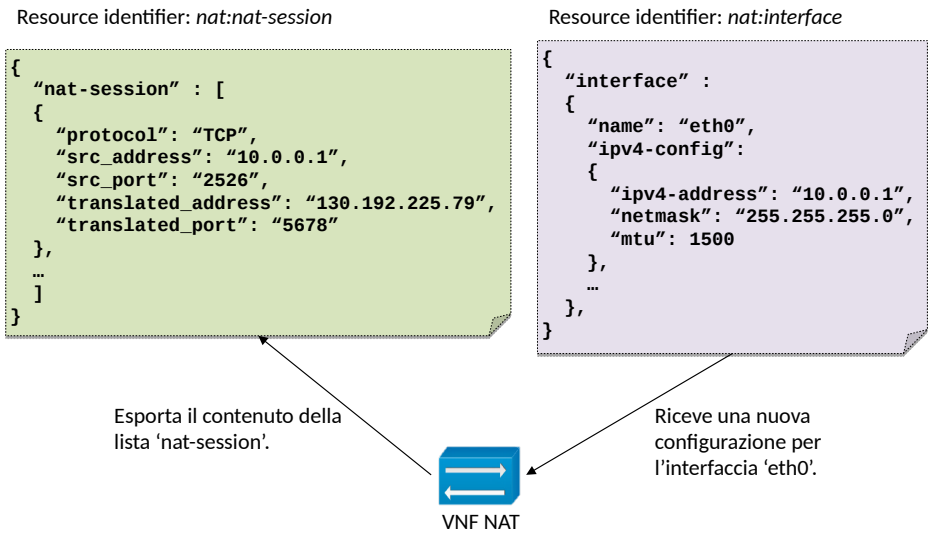


Figura 6.2. Esempio di stato esportato (a sinistra) e di configurazione (a destra) del NAT, definiti nel data-model in Figura 1. In alto l'immagine riporta il *resource identifier*.

frequentemente e di conseguenza risulterebbe impossibile esportarlo ad ogni cambiamento. Dei tipici esempi di nodi `periodic` sono un contatore pacchetti associato ad un'interfaccia di rete e il contenuto di una tabella di NAT, come mostrato in Figura 6.1. Infine è stato definito il tipo `ondemand`, che rappresenta un informazione che deve essere richiesta esplicitamente dall'entità interessata e che non è mai esportata automaticamente dalla VNF; questo è il valore di default per lo statement `adverstise`, anche in caso non sia esplicitamente scritto nel data-model.

Come specificato in precedenza, il data-model definisce anche le API che permettono ad un'entità esterna (es. l'orchestratore) di accedere ai parametri di stato e di configurazione. Tale API sono ricavabili dal modello stesso e sono costruite estendendo le regole già definite in [14]. Partendo dal data-model in Figura 6.1, nella Figura 6.3 è mostrato il relativo set di regole. Come si può notare ogni elemento YANG del modello è associato ad un *resource identifier*. Tale identificatore è costruito partendo dal "module-name:" del modello, seguito dalla concatenazione, dall'esterno all'interno, di tutti gli elementi che vengono attraversati per raggiungere il nodo di interesse, ognuno separato dal carattere '/'. Se il nodo di interesse è un nodo di lista, il segmento è costruito come "list-name=key-value" in modo da selezionare un elemento specifico (linea 8).

#	YANG element	Resource identifier
1	Entire data-model	nat
2	Container	nat:generic-data
3	Leaf/Anydata	nat:name
4	Leaf in a generic position	nat:generic-data/exec-env
5	Entire leaf-list	nat:authors
6	Element in a leaf-list	nat:authors=Piscopo
7	Entire list	nat:nat-session
8	*Element in a list	nat:nat-session=0xa26
9	*Element inside *atomic* node	No resource identifier

Figura 6.3. Derivazione del resource identifier a partire dal modello YANG. Le nuove regole sono identificate con '*'.

6.2 Piano di configurazione/monitoring

Per abilitare le VNFs a ricevere le configurazioni e/o esportare lo stato come descritto nel data-model, è stato definito un piano di configurazione/monitoring che prevede l'utilizzo di due connessioni: un canale REST e un message bus. Il primo è utilizzato per mandare le configurazioni alla VNF e per richiedere lo stato degli elementi di tipo **ondemand**, il secondo è utilizzato per l'esportazione. Quest'ultimo fornisce una comunicazione basata sul paradigma publisher/subscriber (pub/sub). Tale paradigma consente alle entità connesse al bus possono pubblicare dei messaggi associandoli a specifici topic, dopodichè il bus si preoccuperà di mandarli a tutte le entità che precedentemente hanno fatto subscribe a quei topic; questo consente ad una VNF di esportare lo stato in maniera del tutto agnostica rispetto a chi saranno i consumatori. Il message bus supporta inoltre i *topic gerarchici* e il *multi-tenancy*. La prima proprietà consente alle varie entità di fare publish/subscribe sia ad un topic che ad un subtopic. Successivamente il messaggio pubblicato in un subtopic verrà automaticamente consegnato sia all'entità che hanno fatto subscribe al subtopic, sia al supertopic. Un esempio è mostrato in Figura 6.4 (a), dove il messaggio pubblicato nel subtopic *T1/A* è ricevuto sia dal *subscriber 3* che dal *subscriber 1*, che rispettivamente hanno fatto subscribe al subtopic *T1/A* e al supertopic *T1*. La seconda proprietà, il *multi-tenancy*, assicura invece che i messaggi siano ricevuti solamente all'entità che appartengono allo stesso tenant del publisher, come mostrato in Figura 6.4 (b).

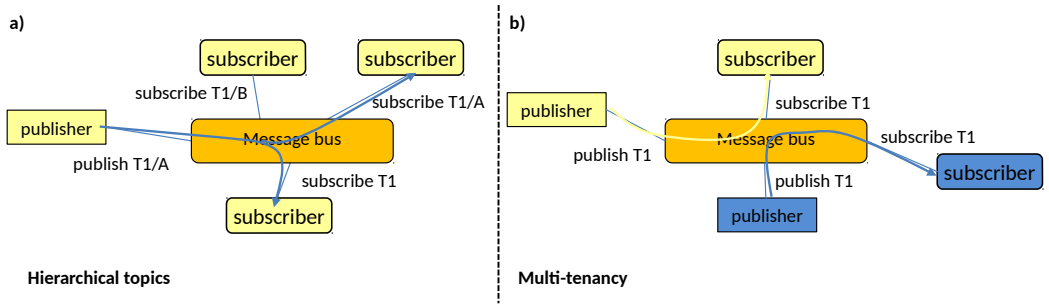


Figura 6.4. Proprietà del message bus.

6.3 VNF Agent

Tutte le VNFs contengono un agent, un modulo che ha il compito configurare le VNF e di esporre lo stato all'esterno, come definito nel data-model, nascondendo i dettagli specifici dell'implementazione e rendendo le unità esterne agnostiche rispetto l'implementazione della VNF. La sua northbound interface deriva dal data-model della VNF, infatti l'agent si aspetta i dati di configurazione strutturati come definito nel modello e indirizzati attraverso i resource identifier come descritto nella sezione precedente. Quando si riceve una configurazione, l'agent converte la rappresentazione ad alto livello (es. JSON) in una serie di comandi specifici dell'implementazione che configurano la VNF. Per esempio, a seguito della ricezione della configurazione JSON mostrata in Figura 6.2, l'agent del NAT esegue il comando *'ifconfig eth0 10.0.0.0/24 mtu 1500'* per configurare l'interfaccia di rete. Diversamente, un agent di una diversa implementazione del NAT potrebbe eseguire comandi diversi per soddisfare la stessa richiesta. Tuttavia dall'esterno questa traduzione sarà nascosta, quindi la configurazione da mandare sarà sempre la stessa indipendentemente dall'implementazione. In maniera analoga, per leggere lo stato della VNF, l'agent utilizzerà dei comandi che dipendono dall'implementazione, tuttavia esporterà utilizzando le regole definite nel data-model (definite dallo statement *advertisement*), che risultano uguali per tutte le implementazioni, in quanto il data-model è lo stesso. L'agent si aspetta le configurazioni ed esporta lo stato attraverso il piano di configurazione/monitoring utilizzando sia il canale REST che quello basato su pub/sub.

6.3.1 Rest API

L'agent espone delle API REST che consentono a moduli esterni sia di cambiare la configurazione della VNF, sia di richiedere esplicitamente lo stato. L'URL da utilizzare per indirizzare una specifica risorsa è ricavabile dal resource identifier e dal data-model come descritto precedentemente nella Sezione 6.1. Le REST API

forniscono l'accesso all'intero data-model associato alla VNF, ovvero ad ogni elemento. In particolare un comando REST può riferirsi ad uno specifico nodo *atomic* del data-model, oppure ad elementi gerarchicamente superiori, ad esempio il *container generic-data* in Figura 6.1. Infine, le REST API rappresentano l'unico modo per poter leggere il valore corrente dei nodi di tipo *ondemand*, che sono forniti dalla VNF solo quando esplicitamente richiesti da un modulo esterno.

6.3.2 Pub/Sub API

L'agent si connette al message bus al boot della VNF, e agisce sia come publisher e potenzialmente come subscriber. In particolare, l'agent periodicamente pubblica un *hello* message nel topic hello, in modo da notificare tutti i moduli interessati che la VNF (identificata tramite il VNF instance ID) è attiva ed è raggiungibile (attraverso le API REST) a un specifico endpoint (Indirizzo IP e porta), così da semplificare il VNF discovering. Un nuovo modulo interessato alla VNF, deve semplicemente attaccarsi al bus e fare un subscribe al topic 'vnf-hello'; in questo modo, automaticamente scoprirà l'endpoint REST da utilizzare per interagire con la VNF. Il message bus è poi usato dall'agent per esportare le informazioni corrispondenti ai nodi *atomic* il cui advertise è di tipo *onchange, onthreshold o periodic*, in quanto i dati vengono esportati pubblicando su un topic specifico derivato dal resource identifier dello specifico nodo YANG. La scelta di utilizzare un message bus per esportare automaticamente le informazioni invece che utilizzare altre tecniche basate su HTTP è la seguente. Il message bus consente all'agent di esportare le informazioni senza conoscere i consumatori dei dati e senza il bisogno di inviare la stessa informazione più volte nel caso ci siano più entità interessate agli stessi parametri (nodi YANG). In particolare, grazie al bus, l'agent gestisce una singola connessione verso il bus, senza preoccuparsi delle entità che consumeranno i dati richiesti; il bus infatti, garantisce di consegnare una copia del messaggio pubblicato a tutti moduli interessati (che hanno fatto subscribe al topic) e che ne hanno il diritto di riceverlo (appartengono allo stesso tenant del publisher).

Capitolo 7

Use Cases

In questo capitolo verranno presentati gli use cases utilizzati per validare il framework.

7.0.1 Servizi autoconfiguranti

Un primo use cases che sfrutta le potenzialità del framework sono i servizi autoconfiguranti. Tali servizi, come è possibile vedere in Figura 7.1, comprendono un orchestratore di servizio che opera su diverse VNFs, il cui compito è quello di monitorare lo stato delle VNFs e che al verificarsi di determinate condizioni, reagisce cambiando la configurazione di altre VNFs, oppure scalando il numero di istanze di VNF, senza richiedere alcun intervento esterno. Tale funzionalità è possibile grazie al framework in quanto: (i) il data-model della VNF descrive in maniera indipendente dall'implementazione i parametri di configurazione e di stato della VNF, così come interagire con essa; (ii) l'agent della VNF si preoccupa di convertire le richieste ad alto livello in comandi specifici dell'implementazione e di esportare lo stato come definito nel data-model; (iii) il piano di configurazione/monitoring consente di spostare le informazioni da un componente all'altro.

Gli orchestratori di servizio sono componenti che non operano sul data-plane della rete (non processano il traffico di rete), ma solo sul piano di configurazione/-monitoring. Ognuno di essi è progettato per interagire con specifici data-models di VNF (per esempio IDS e Firewall, come si può vedere nel servizio in Figura 7.2), ma resta agnostico rispetto l'implementazione di tali VNFs, infatti utilizza solo le informazioni descritte nei data-models. Tali servizi disaccoppiano due VNFs nel piano di configurazione/monitoring, infatti, ogni VNF riceve una nuova configurazione solo in base al suo data-model, è agnostica rispetto chi la configura. Per questo, gli orchestratori di servizio sono utilizzati per evitare di aggiungere qualsiasi tipo di logica di orchestrazione all'interno della VNF, in maniera da lasciare le VNF il più generiche possibile così da poter essere usate all'interno di servizi differenti. Per

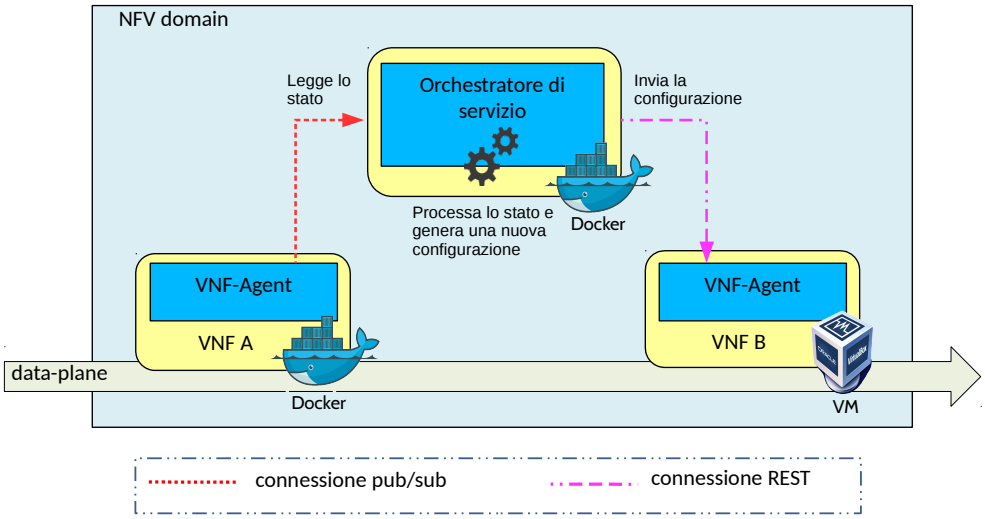


Figura 7.1. Servizio autoconfigurante.

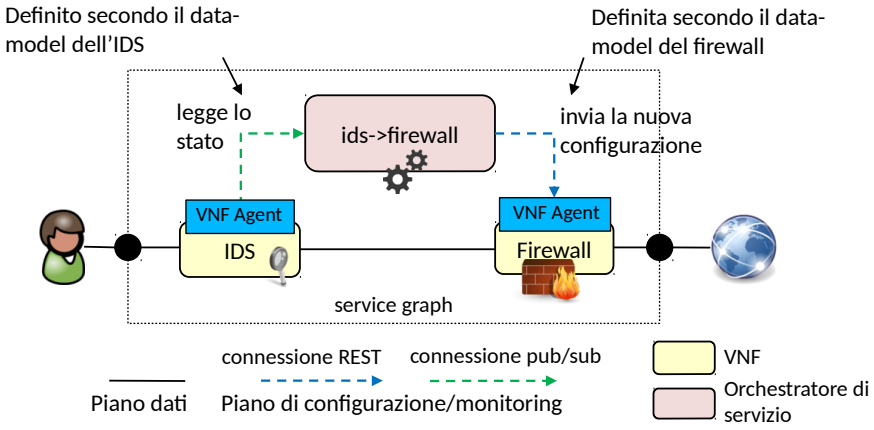


Figura 7.2. Esempio di servizio autoconfigurante.

esempio, lo stesso firewall può essere usato sia in un servizio come quello in Figura 7.2, dove la sua configurazione cambia in funzione dello stato esportato dall'IDS, sia in un altro servizio in cui la sua configurazione è influenzata dallo stato esportato dal server DHCP (es. il firewall adotta un politica di *whitelist*, abilitando il traffico solo da/verso gli indirizzi IP assegnati nella LAN dal server DHCP). Inoltre, come mostrato nell'esempio in Figura 7.3, la configurazione di una VNF può dipendere

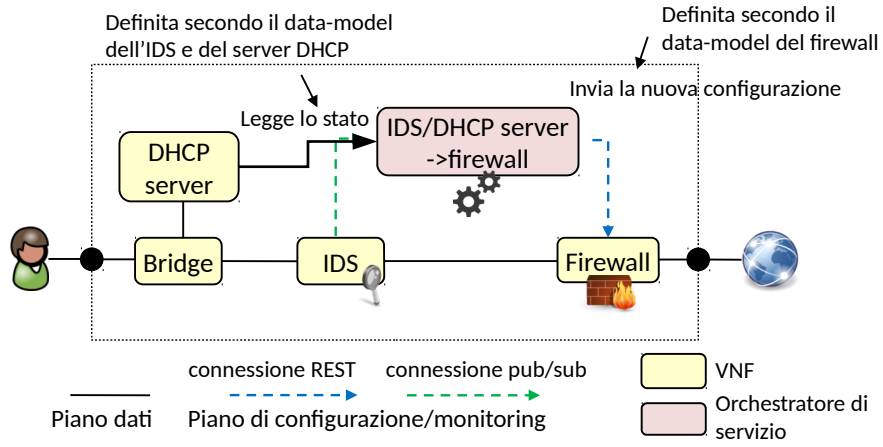


Figura 7.3. Esempio di servizio autoconfigurante dove due VNF contribuiscono alla configurazione del firewall.

dallo stato di molteplici VNFs; infatti il servizio in figura aggiunge una nuova regola nel firewall ogni volta che il server DHCP assegna un nuovo indirizzo IP, ma anche quando l'IDS rileva un attacco che deve essere bloccato. Si noti che, grazie alla presenza dell'orchestratore di servizio, il firewall non conosce chi gli cambia la configurazione, tantomeno come questa configurazione viene creata (in questo caso processando lo stato esportato dal server DHCP e dall'IDS), ma si preoccupa solo di ricevere i nuovi parametri secondo il proprio data-model e di applicare la nuova configurazione (grazie all'utilizzo dell'agent). Similmente, una VNF espone lo stato secondo il proprio data-model, senza preoccuparsi del consumatore di tale informazione (non sa che il proprio stato verrà usato per creare una nuova configurazione per il firewall).

Di seguito sono riportati degli esempi di servizi autoconfiguranti:

- IDS-Firewall Service: il servizio si occupa di configurare automaticamente il firewall non appena l'IDS rileva un attacco verso la rete.

- DHCP-Firewall Service: il servizio si occupa di configurare automaticamente il firewall non appena il server DHCP fornisce un indirizzo IP ad nuovo client per permettergli di inviare e ricevere il traffico sulla rete.

- NAT Improvement Service: il servizio si occupa di monitorare un nat; quando il numero di entry della nat table raggiunge un certa soglia, reagisce effettuando lo scale in/out delle istanze di nat.

7.0.2 Migrazione di VNF basata sullo stato

Un altro use cases in cui il framework può essere utilizzato è quello di una *live migration* di VNFs basata sulla migrazione dello stato. Questa tipologia di migrazione è diversa rispetto la migrazione tradizionale di Virtual Machine. Nelle migrazioni tradizionali di VM, vengono copiate interamente le immagini e la memoria da un'istanza all'altra, generando un'elevata quantità di traffico di rete; inoltre è possibile migrare i dati solo tra istanze identiche di VM. La migrazione basata sullo stato risolve questi limiti. I pacchetti generati sulla rete diminuiscono sensibilmente in quanto si migra soltanto lo stato e non tutta l'immagine; inoltre è possibile effettuare la migrazione anche tra VNF implementate con tecnologie completamente diverse, in quanto non è necessario che le istanze siano uguali, l'importante è che facciano riferimento allo stesso data-model.

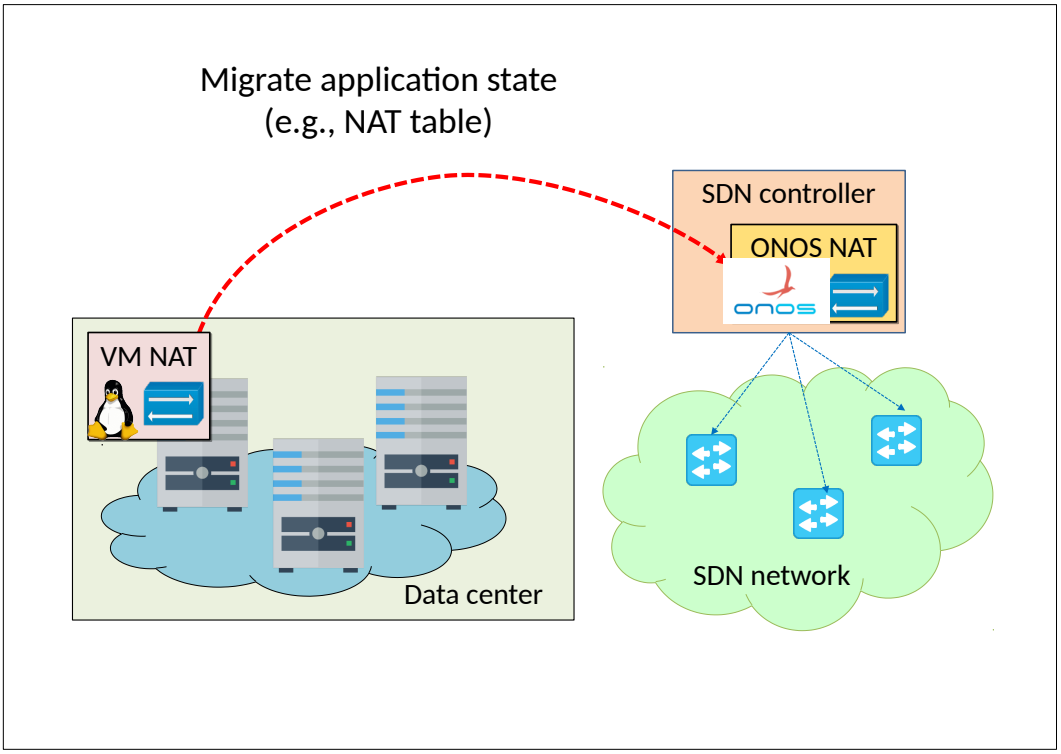


Figura 7.4. Esempio di live-migration.

Per effettuare la migrazione, occorre che ci sia un componente che coordini le operazioni e che sia in grado di lavorare con i dati del data-model associato alle VNFs da migrare. Inoltre, deve comunicare anche con l'orchestratore che gestisce il ciclo di vita delle VNFs del servizio in quanto, oltre alla migrazione dello stato

occorre aggiornare il grafo per spostare i collegamenti dalla VNF di partenza a quella di arrivo.

Supponiamo di voler migrare una VNF-A di un dominio 1 ad una VNF-B su un dominio 2 in cui entrambe le VNFs condividono lo stesso data-model ma sono implementate con due tecnologie differenti. Le fasi necessarie per eseguire la migrazione sono visibili in Figura 7.5 e descritte qui di seguito:

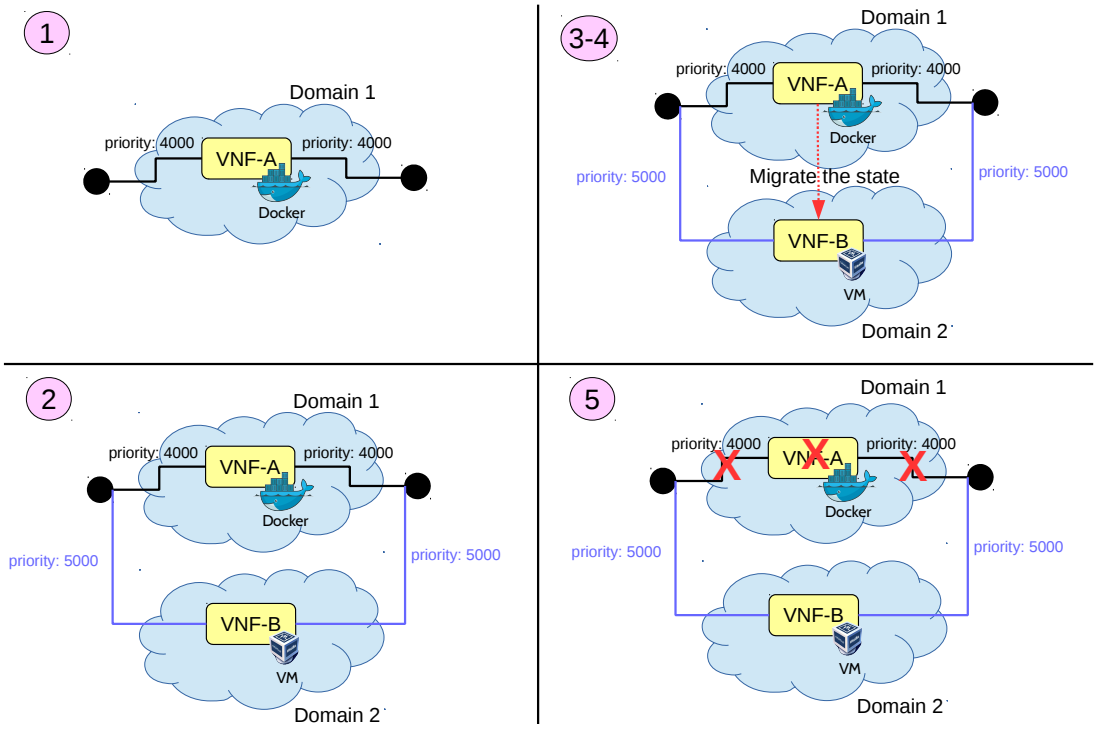


Figura 7.5. Fasi necessarie ad eseguire la migrazione tra due VNF.

1. Deploy del grafo di servizio contenente la VNF-A.
2. Update del grafo aggiungendo la VNF-B; i link che convergono alla VNF-A vengono copiati sulla VNF-B e gli viene assegnata una priorità maggiore in modo tale che i nuovi flussi inizino a passare dalla la VNF-B.
3. Viene prelevato esplicitamente lo stato dalla VNF-A (richiesta di tipo *on-demand*)
4. Lo stato prelevato viene iniettato nella VNF-B.
5. Update del grafo eliminando la VNF-A e i suoi links.

Capitolo 8

Implementazione

In questo capitolo verranno spiegate le varie parti dell'implementazione, fornendo una descrizione per ogni componente realizzato. La fase implementativa si è svolta nel seguente modo: dopo aver definito il framework, sono state realizzate le VNFs e i relativi agent; in particolare sono state sviluppate le seguenti VNF configurabili: NAT, server DHCP, Intrusion Detection System (IDS) e Firewall. Sucessivamente è stata estesa l'architettura di orchestrazione FROG per fornirgli il supporto alla configurazione. Infine sono stati sviluppati i servizi definiti negli use cases. Tutti i componenti sono stati realizzati in Python, tranne alcune estensioni dell'architettura che ovviamente sono state eseguite utilizzando il linguaggio già presente (C++).

8.1 Estensione dell'architettura FROG

L'architettura di orchestrazione FROG da cui si è partiti, come descritto nel Capitolo 5, permetteva di istanziare i servizi di rete, ma non offriva nessun supporto alla configurazione delle VNFs. Una volta istanziato il grafo di servizio, per configurare le VNFs che ne facevano parte, era necessario entrare all'interno di ognuna e inserire i comandi di configurazione a mano. L'architettura dunque necessitava di un unità centralizzata in grado di gestire le configurazioni delle varie VNFs; inoltre occorreva un meccanismo per poter istanziare un grafo di servizio e associargli una configurazione iniziale di partenza da inserire nelle varie VNFs. A tale scopo è stato sviluppato un nuovo componente, il *configuration-orchestrator*, a cui è stato affidato il compito di gestire le configurazioni in maniera centralizzata. L'integrazione di tale componente nell'architettura ha richiesto l'estensione degli orchestratori di dominio in quanto si è dovuta aggiungere la logica che permettesse la comunicazione con esso. Infine, per poter inserire al boot delle VNFs la configurazione iniziale, è stato esteso il plugin dell'orchestratore che si occupa di avviare le VNFs, aggiungendogli la possibilità di montare un disco dinamico al boot (*Bootstrapping Disk*).

L'architettura FROG estesa, comprendente il supporto alla configurazione, è diventata quella mostrata in Figura 8.1.

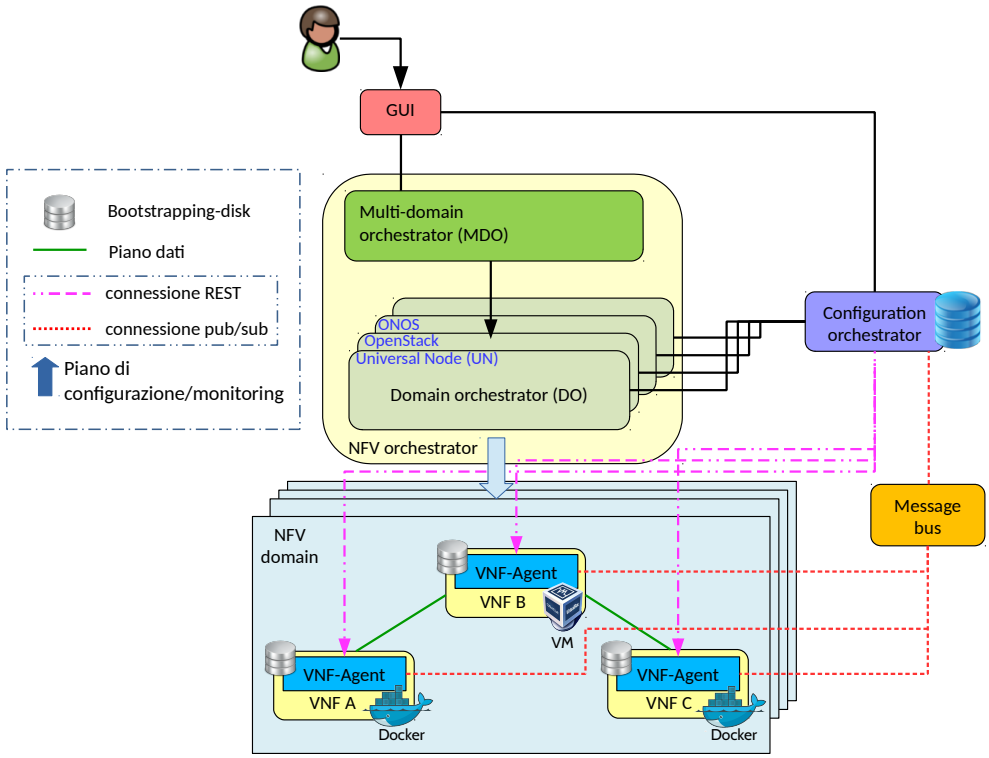


Figura 8.1. Schema dell'architettura FROG estesa.

- **NFV Orchestrator**: tale area comprende il sistema di orchestrazione descritto nel Capitolo 5, che utilizzando i vari orchestratori di dominio, permette di istanziare il grafo di servizio nei vari domini dell'infrastruttura.
- **NFV Domain**: include tutte le VNFs del servizio di rete allocate sull'infrastruttura dall'NFV Orchestrator. Come si può notare in Figura 8.1, tutte le VNF sono equipaggiate con un agent.
- **Message Bus**: Come descritto precedentemente nel Capitolo 6, il framework prevede l'utilizzo di un message bus.
- **Configuration-Orchestrator**: è il componente che permette di gestire in maniera centralizzata le configurazioni delle VNFs. La sua northbound interface fornisce delle API REST per gli orchestratori di dominio e per gli operatori di rete: quelle verso l'orchestratore consentono a quest'ultimi di prelevare (se presente) una configurazione iniziale da inserire al boot delle varie VNFs del

grafo più altri file richiesti necessari all'avvio dall'agent. Le altre, quelle rivolte verso l'operatore di rete, consentono di leggere e modificare la configurazione delle varie VNFs a runtime, offrendo all'operatore un unico punto di contatto, evitando di dover fargli contattare la varie VNFs singolarmente. Come si vede dall'architettura, l'operatore può sfruttare una GUI per leggere e modificare la configurazione delle VNFs.

- **GUI**: l'interfaccia grafica permette ad un'utente/operatore di rete di visualizzare il grafo che è stato istanziato, mostrando tutte le VNFs coinvolte e i loro collegamenti; inoltre essendo collegata anche al configuration-orchestrator sfrutta le API esposte da quest'ultimo, per permettere di visualizzare e modificare la configurazione delle VNFs istanziate.

8.1.1 Un nuovo componente: il configuration orchestrator

Il *configuration orchestrator* (C.O.) ha introdotto nel sistema FROG il supporto alla configurazione delle VNFs. In particolare le funzionalità introdotte sono:

- **Possibilità di associare ad un grafo una configurazione di partenza**: Questa funzionalità è molto utile in quanto permette di associare una configurazione di partenza ad un grafo, in modo tale che durante il deploying del servizio di rete, le varie VNFs vengano configurate automaticamente al boot con una specifica configurazione, evitando di dover essere configurate singolarmente ogni volta che vengono istanziate. Per memorizzare le varie informazioni, il C.O. si serve di un database in cui vengono salvate tutte le configurazioni e le relative associazioni con le VNF e i grafi; infatti ogni configurazione è associata ad una VNF e ad un grafo. Volendo è anche possibile memorizzare una configurazione di default per ogni tipologia di VNF (es. NAT, Firewall) da utilizzare qualora non fosse trovata una configurazione specifica associata al grafo.

- **Gestione centralizzata delle configurazioni**: Questa funzionalità risolve il problema di dover entrare manualmente in ogni VNF da configurare per inserire la configurazione. Il C.O. diventa infatti l'unità centralizzata che gestisce le configurazioni. Di conseguenza, per configurare una VNF, è sufficiente inviare la configurazione (scritta secondo il data-model YANG associato alla VNF) al C.O. e questo si occuperà di inoltrarla alla VNF specificata. Questo è possibile in quanto ogni VNF, una volta avviata, si registra sul bus pubblicando sul topic 'vnf-hello' un messaggio contenente il suo identificatore (VNF instance ID) e l'indirizzo IP del server REST per contattarla. Il C.O. quando viene avviato, fa una subscribe al topic 'vnf-hello', così ogni qualvolta che una

VNF viene istanziata, il C.O. riceve il messaggio di hello e memorizza la coppia identificatore e indirizzo IP della VNF. In questa maniera il C.O. riesce a mantenere una mappa di tutte le VNF attive e dei relativi indirizzi IP.

Per queste funzionalità vengono esposte le seguenti REST API:

- **GET /started-vnf**: restituisce la lista di tutte le VNF attive che si sono registrate al C.O., comprendenti il VNF instance ID e l'indirizzo dell'endpoint REST.
- **GET, PUT /<tenant-id>/<graph-id>/<vnf-id>/<url>**: consente di prelevare (GET) o configurare (PUT) lo stato di una qualsiasi VNF attiva. I primi tre parametri identificano la VNF, l'ultimo indirizza la risorsa della VNF a cui si vuole accedere e deve essere scritto secondo le regole esposte precedentemente nella Figura 6.3).

8.1.2 Interazione tra orchestratore e configuration-orchestrator

Nell'orchestratore di dominio, che nel nostro caso sarà lo Universal Node (UN), è stata aggiunta la logica per comunicare con il configuration-orchestrator, in modo tale da poter richiedere gli eventuali file associati alle VNF che dovranno essere inseriti nel disco di bootstrapping. L'interazione tra i due componenti avviene tramite chiamate REST, il C.O. infatti, espone due API, una per richiedere la lista dei file associata ad una VNF, e l'altra per richiedere l'invio di un singolo file specificato come parametro.

Di seguito è spiegato come interagiscono i due componenti.

Per ogni VNF del grafo:

1. L'orchestratore contatta il C.O. chiedendo se ci sono dei file associati alla VNF passandogli come parametro l'identificativo della VNF.
2. Il C.O. interagisce con il suo database e verifica se per quella VNF sono presenti dei file. In caso affermativo, restituisce una lista contenente l'elenco dei file, in caso contrario, restituisce una lista vuota.
3. L'orchestratore, a questo punto, scandisce la lista ricevuta e se questa non è vuota, richiede ciascun file al C.O. memorizzandolo in una cartella temporanea associata alla VNF.
4. L'orchestratore infine è pronto ad avviare la VNF. Se sono stati ricevuti dei file associati ad essa, la VNF viene avviata e gli viene passato anche il datadisk contenente i file, altrimenti la VNF viene avviata normalmente senza datadisk.

Come si può notare l'orchestratore rimane agnostico rispetto ai file ricevuti; non è compito suo processarli, esso si preoccupa solo di riceverli e inserirli dentro la VNF.

8.1.3 Rete di management

Tutte le VNF configurabili necessitano di un'interfaccia di management per poter essere raggiunte dall'esterno, infatti l'agent utilizza l'indirizzo IP di tale interfaccia come endpoint in cui avviare il server REST per esporre la northbound interface. Per questo motivo, tutte le VNFs configurabili devono essere collegate ad una rete di management che mette in comunicazione le VNFs con il configuration-orchestrator e il broker del message bus. In fase di configurazione, è importante tenere a mente che tale rete non deve avere un indirizzamento sovrapposto alla rete di servizio. Per poter assegnare degli indirizzi IP in maniera dinamica alle interfacce di management di tutte le VNFs configurabili, per comodità, nella rete di management si inserisce un Server DHCP in maniera da automatizzare la procedura. Per ovvie ragioni la VNF di questo Server DHCP è l'unica VNF che non può essere configurata dall'esterno.

8.2 VNF-Agent

Dal momento che si è dovuto creare un agent per ogni VNF, il software è stato progettato in maniera tale da avere una parte comune a tutte le VNFs, più una parte specifica relativa alla VNF a cui è associato. Infatti, per ogni VNF, la fase di bootstrap è uguale per tutte, mentre cambiano i metodi per la configurazione ed esportazione dei parametri in quanto dipendono dal data-model associato alla VNF. Utilizzando un approccio del genere, realizzata la parte comune, non è difficile scrivere degli agent per altre VNF in quanto occorre implementare solo i metodi specifici di configurazione ed esportazione. Inoltre, per fare in modo che l'agent fosse indipendente dall'implementazione della VNF, i metodi di esportazione e configurazione sono stati definiti come interfacce; è la VNF che li implementa utilizzando la tecnologia necessaria.

8.2.1 Bootstrapping-disk

Come preannunciato precedentemente, l'orchestratore monta un disco di bootstrapping ad ogni VNF, che contiene informazioni necessarie all'agent per il boot della VNF stessa. In particolare l'agent si aspetta di trovare nel disco le seguenti informazioni. (i) La configurazione iniziale da applicare alla VNF appena viene avviata, in particolare la configurazione dell'interfaccia di management, scritta secondo il data-model associato alla VNF stessa. (ii) L'identificatore delle VNF (VNF instance ID) da pubblicare nel message bus nell'hello message, e che sarà anche utilizzato sia per creare le URL esposte nelle REST API, sia nei topic utilizzati per pubblicare lo stato. (iii) I parametri da utilizzare per connettersi al message bus tra i quali l'endopoint del message broker. (iv) Le credenziali del tenant a cui la VNF appartiene; queste, devono essere usate sia per registrarsi sul bus, sia per autenticare gli altri

moduli che contattano la VNF tramite le REST API. (v) Le chiavi da utilizzare per cifrare/decifrare le informazioni pubblicate/ricevute attraverso il bus.

8.2.2 Monitoring dello stato

Il monitoring dello stato è una delle funzionalità fondamentali dell'agent così come una delle più critiche quando è associato all'esportazione di tipo *onchange*. Monitorare lo stato continuamente, può essere molto costoso in termini di risorse, per cui si utilizzano tecniche diverse atte a minimizzare il consumo di CPU. Le soluzioni adottate sono sostanzialmente due e si distinguono per la quantità di risorse utilizzate. La prima prevede il polling sull'elemento da monitorare in attesa di cambiamenti, l'altra permette di essere notificati non appena si verifica un cambiamento.

- **Soluzione basata sul polling:** tale tecnica è assolutamente la più costosa in termini di CPU, in quanto si utilizza un thread per ogni risorsa da monitorare, ma è anche quella applicabile per ogni elemento. Il monitoring consiste in un loop che prevede la scansione dell'elemento e il confronto con l'ultima versione esportata, in modo tale da rilevare i nuovi cambiamenti, che in tal caso scateneranno l'esportazione. L'intervallo di tempo tra una scansione e l'altra, può essere deciso dall'esterno, tramite uno dei parametri passati all'avvio dell'agent.

- **Soluzione basata su notifica:** questa alternativa è molto più efficiente della precedente in quanto non consuma costantemente CPU, tuttavia è possibile applicarla solo in alcuni casi, ovvero agli elementi il cui monitoring è basato su file. Infatti, la soluzione consiste nel selezionare un file da monitorare, e non appena questo verrà modificato, si riceverà una notifica evitando di dover ciclare su di esso per rilevare i cambiamenti. Un esempio in cui viene adottata questa soluzione è nel monitoring degli attacchi dell'IDS: quando viene rilevato un attacco, questo viene scritto dentro un file di log; la modifica del file scatenerà una notifica verso l'agent che provvederà ad esportare l'attacco.

Per quanto riguarda il monitoring delle risorse il cui advertisement è di tipo *periodic* si ricorre all'utilizzo dei timer. In particolare, ad ogni nodo periodico si associa un timer con una funzione di callback. Quando questa viene attivata, si preoccupa di prelevare ed esportare lo stato del nodo associato. Questo tipo di advertisement è meno complicato da gestire rispetto a quello di tipo *onchange*, in quanto lo stato non deve essere monitorato continuamente; in questo caso essendo che viene esportato periodicamente indipendentemente dal fatto che sia cambiato o meno rispetto l'ultima volta, non occorre mantenere in memoria lo stato precedente ed effettuare il confronto prima dell'esportazione.

Infine, per quanto riguarda l'advertisement di tipo *ondemand*, non occorre il monitoring della risorsa, ad ogni nodo di quel tipo è associata una funzione che viene chiamata esplicitamente dall'utente per richiedere lo stato.

8.2.3 DoubleDecker Python API

Per quanto riguarda il message bus, come descritto precedentemente, si è scelto di utilizzare DoubleDecker[12]. L'implementazione all'interno dei vari componenti non è stata difficile in quanto gli sviluppatori offrono una classe Python che fornisce i metodi necessari per il funzionamento.

Di seguito sono elencati i principali metodi utilizzati:

- **def register-to-bus(self, name, dealer-url, customer, keyfile)**:
  Permette ad un entità di registrarsi al bus, è il primo metodo da chiamare prima di poter utilizzare gli altri. I parametri richiesti sono:
  - name: nome che identifica univocamente il client appartenenti ad un tenant.
  - dealer-url: corrisponde all'endpoint del broker, deve essere scritto nella forma: tcp://ip-address:port.
  - customer: è l'identificativo del tenant.
  - keyfile: richiede la chiave del tenant per la registrazione.
- **def subscribe(self, topic, scope)**:
  Permette di sottoscriversi ad uno specifico topic. Come parametri richiede il nome del topic e lo scope.
- **def unsubscribe(self, topic)**:
  Permette di annullare l'iscrizione ad un topic passato come argomento.
- **publish-topic(self, topic, msg)**:
  Permette di pubblicare un messaggio in uno topic specifico. Accetta come parametri il messaggio da pubblicare e il nome del topic.
- **send-message(self, dst, msg)**:
  Permette di inviare un messaggio privato ad un altro client registrato. Il destinatario è identificato dal nome e deve appartenere allo stesso tenant del mittente.
- **on-reg(self)**:
  Callback chiamata non appena la registrazione sul broker va a buon fine. Serve a verificare che la registrazione si è conclusa correttamente.

- **on-pub(self, src, topic, msg)**:
  Callback richiamata non appena qualcuno pubblica in un topic in cui si aveva fatto subscribe precedentemente. I parametri identificato il mittente, il topic stesso, e il messaggio inviato.

- **on-data(self, src, msg)**:
  Callback chiamata non appena viene ricevuto un messaggio privato. I parametri identificano il mittente del messaggio e il messaggio stesso.

- **on-error(self, code, msg)**:
  Callback chiamata non appena si verifica un errore sul bus. Fornisce un codice identificativo dell'errore e un messaggio esplicativo.

8.2.4 Flusso di esecuzione

Il flusso di esecuzione del programma è suddiviso nelle seguenti fasi:

1. **Applicazione della configurazione iniziale**: l'agent legge la configurazione iniziale ricevuta nel datadisk e la applica. Non è necessario fornire una configurazione completa in quanto può essere mandata anche successivamente, tuttavia è fondamentale fornire una configurazione per l'interfaccia di management, in quanto senza un indirizzo IP valido è impossibile comunicare con la VNF dall'esterno. Tale configurazione può essere ricavata anche tramite DHCP, purchè sia specificato nel file di configurazione stesso.

2. **Avvio del REST Server**: l'agent avvia il REST Server utilizzando come endpoint, l'indirizzo IP appena configurato, tuttavia ancora non può essere raggiunto dalle unità esterne in quanto ancora non conoscono l'indirizzo dell'endpoint.

3. **Registrazione sul message bus**: l'agent utilizza le credenziali del tenant fornite nel datadisk per registrarsi sul message bus.

4. **Registrazione sul configuration-orchestrator**: l'agent pubblica sul topic 'vnf-hello' in cui specifica il proprio identificatore di VNF (passatogli nel datadisk) e l'indirizzo dell'endpoint REST appena configurato. In questo modo tutte le entità esterne che hanno fatto subscribe al topic, possono conoscere quali sono le varie VNF che si attivano e come raggiungerle. In realtà questo messaggio non viene pubblicato solo all'inizio, ma periodicamente, in modo tale che se si attivano successivamente delle nuova entità, possono comunque conoscere le VNF attive.

5. **Monitoring/esportazione dello stato**: a questo punto l'agent inizia a monitorare lo stato della VNF e ad esportarlo secondo le regole definite nel data-model della VNF. Inoltre resta sempre attivo sul REST Server, in attesa di nuove richieste.

8.3 VNF realizzate

Di seguito sono descritte le VNF che sono state realizzate, ciascuna delle quali ha previsto la scrittura di un agent con la struttura e le caratteristiche descritte sopra, ma con le funzionalità specifiche della VNF associata.

8.3.1 NAT

La VNF implementa le funzionalità di un NAT aggiungendo la possibilità di poter esportare la NAT table. Come si vede nella Figura 8.2 la VNF possiede tre porte. La porta 0 è la porta di management, utilizzata per interagire con la VNF dall'esterno; la porta 1 è quella collegata verso la rete interna; la porta 2 è quella di NAT.

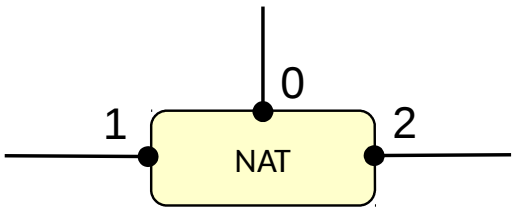


Figura 8.2. Schema della VNF NAT.

Per abilitare la macchina virtuale a fare da NAT occorre dare i seguenti comandi che abilitano l'ip forwarding:

- *echo 1 > /proc/sys/net/ipv4/ip-forward*
- *iptables -t nat -A POSTROUTING -o public-interface-name -j MASQUERADE*

Per quanto riguarda la lettura della NAT table, l'agent si serve di un tool, *conntrack* [18] che fornisce una serie di funzionalità tra cui la lettura delle sessioni di NAT. In particolare, per poter utilizzare tale tool all'interno del codice Python, si utilizza un pacchetto (*pynetfilter-conntrack*) che implementa un wrapper Python di Conntrack. Una limitazione di questo tool è che non permette di scrivere la NAT table ma solo di leggerla.

8.3.2 DHCP Server

La VNF implementa un server DHCP configurabile. In Figura 8.3 si vede che la VNF è costituita da due porte: la porta 0 che è quella di management e la porta 1 che va verso la rete. Tramite la porta di management è possibile modificare la configurazione del server DHCP, secondo quanto definito nel data-model della VNF. Al contrario in uscita, ogni volta che il server assegna un nuovo indirizzo IP ad un client, viene esportata tale informazione (come visibile nell'esempio in Figura 8.8 a sinistra).

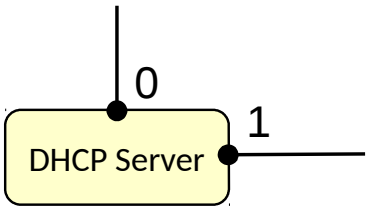


Figura 8.3. Schema della VNF DHCP Server.

Per poter abilitare la macchina virtuale a fare da DHCP, è stato installato il pacchetto *isc-dhcp-server* che comprende un DHCP Server. Internamente l'agent, per poter configurare e leggere la configurazione del server si serve del file di configurazione */etc/dhcp/dhcpd.conf* il quale viene letto o scritto in base alla richiesta. Inoltre tale file viene anche monitorato dall'agent, una modifica causa l'esportazione sul bus dell'informazione. Invece, per ottenere la lista dei client che hanno ricevuto un indirizzo IP, l'agent utilizza un altro file: */var/lib/dhcp/dhcpd.leases*. La lettura di questo file non avviene direttamente, ma si utilizza un pacchetto Python (*isc-dhcp-leases*) che permette di ricavarne solo le informazioni necessarie. Facendo riferimento al data-model del Server DHCP, i parametri che vengono letti per ogni client sono: mac-address, hostname, ip-address, lease-time. Anche in questo caso, il file viene monitorato, ad ogni cambiamento, l'informazione del client viene esportata sul bus.

8.3.3 Firewall

La VNF implementa un firewall trasparente. Come mostrato in Figura 8.4 la VNF comprende 3 porte, di cui due sono trasparenti. La porta 0 è quella di management, le altre due sono quelle trasparenti e sono collegate tra di loro attraverso un bridge. Le funzionalità fornite dal firewall comprendono la configurazione di policies, e la gestione di blacklist e whitelist per le url. La prima consente di specificare delle regole per il traffico, le altre due permettono di bloccare o meno determinate url,

in particolare quelle specificate nella blacklist saranno sempre bloccate, quelle nella whitelist saranno sempre consentite. Internamente tutte queste funzionalità offerte saranno tradotte in regole *iptables* ed *ebtables*[19]. Quest'ultimo è un tool simile ad iptables che consente di specificare le interfacce nelle regole anche quando queste fanno parte di un bridge, per cui risulta necessario quando il firewall è trasparente. Per questo, in base alla policy che viene inserita, l'agent riconosce automaticamente in quale comando in quale comando tradurla.

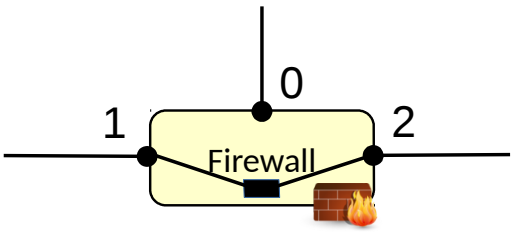


Figura 8.4. Schema della VNF Firewall.

8.3.4 NIDS

La VNF implementa un Network Intrusion Detection System (NIDS). Anche in questo caso, come nel firewall, la VNF ha due porte trasparenti collegate tramite un bridge e una porta di livello 3 di management (Figura 8.5). L'agent utilizza il tool snort [20] per rilevare gli attacchi e utilizza il message bus per esportarli. La versione attuale dell'agent, essendo stata scritta solo per determinati use cases, è in grado di rilevare solamente due attacchi, *ping flood* e *port scan*, tuttavia non è difficile estenderla aggiungendone altri. Per rilevare gli attacchi, occorre inserire delle specifiche regole dentro snort, che successivamente verranno utilizzate per analizzare il traffico passante. Il primo attacco, il ping flood, consiste nel mandare un'enorme quantità di pacchetti ICMP Echo Request (ping) verso il sistema vittima, provocando nel caso peggiore un denial of service. Per proteggersi da questo attacco è stata inserita una regola che fa scattare l'allarme quando vengono ricevuti più di 100 pacchetti ICMP Echo Request dallo stesso indirizzo sorgente nell'arco di 5 secondi. I valori sono stati tarati in base agli use cases che siamo andati a testare. Per proteggersi dal secondo tipo di attacco, il port scan, è stata inserita una regola che scatta non appena vengono rilevate più di 30 connessioni TCP dallo stesso indirizzo IP nell'arco di 60 secondi. Anche questa è stata tarata in base ai nostri use cases. Per quanto riguarda l'esportazione degli attacchi, l'agent utilizza la soluzione basata su notifica, che prevede di essere notificato non appena viene rilevato e inserito un nuovo attacco all'interno del file di logging. Questo fa si che l'agent riesca ad implementare al meglio l'esportazione di tipo *onchange* degli attacchi rilevati.

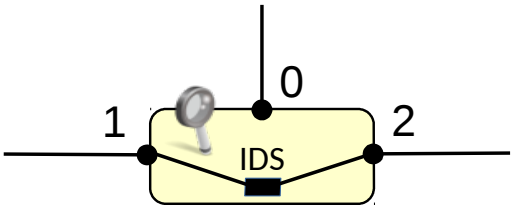


Figura 8.5. Schema della VNF IDS.

8.4 Servizi autoconfiguranti

I servizi autoconfiguranti sono delle VNF a tutti gli effetti, infatti per essere istanziati basta semplicemente aggiungerli nel grafo di servizio e verranno trattati allo stesso modo delle altre VNFs. Inoltre, anche essi possono essere eseguiti sia dentro container docker che dentro Virtual Machine (VM). Per quanto riguarda la loro implementazione si è scelto di utilizzare il linguaggio Python, in quanto si è potuto riciclare alcune parti di software già scritte durante l'implementazione degli agent delle VNFs.

Dal punto di vista implementativo il software che realizza un servizio autoconfigurante è formato dai seguenti componenti:

- **Service logic**:
  La logica di servizio è il core del servizio stesso in quanto implementa la funzionalità specifica che è stata definita e che sarà diversa in ogni servizio autoconfigurante. Tale logica si pone logicamente al centro del modulo in quanto utilizza come input lo stato esportato dalla prima VNF e fornisce in output una configurazione da applicare ad un'altra VNF. In realtà, ad essere più precisi, alcuni servizi invece di cambiare la configurazione ad un'altra VNF, potrebbero scatenare un evento ad un altro componente; si pensi allo scale in/out delle istanze di VNFs in base al monitoring di parametri di stato di una VNFs, in quel caso la logica di servizio produrrà un evento da inviare ad esempio all'orchestratore.

- **REST Server**:
  Il Server REST è necessario in quanto occorre offrire delle API che consentono di specificare a runtime dall'esterno quali sono le VNFs da monitorare/configurare. Infatti, sebbene la logica di servizio e i data-models sono componenti cablati nel programma, le istanze delle VNFs da utilizzare devono essere specificate dinamicamente, in base al grafo che viene istanziato. Inoltre in questo modo è anche possibile modificare le istanze da utilizzare nel mentre che il

servizio risulta attivo. In particolare le API esposte richiedono l'inserimento del VNF instance ID delle VNF da utilizzare.

- **Message Bus client**:
  L'utilizzo del message bus è fondamentale in quanto permette al servizio di ricevere lo stato esportato dalla/e VNF che si stanno monitorando. Inoltre è utilizzato anche per annunciarsi sulla rete agli altri componenti, in quanto come le altre VNFs, l'indirizzo IP del server REST viene configurato al boot e deve essere pubblicato all'esterno. Anche in questo caso, si utilizza il client doubledecker utilizzando i metodi dell'interfaccia Python fornita dagli sviluppatori e già discussa precedentemente nella Sezione 8.2.3.

- **REST Client**:
  Il client REST è utilizzato dal programma dopo che la logica di servizio ha prodotto la configurazione e deve essere inviata alla specifica VNFs. Infatti, per inviarla, si utilizzano le API REST esposte dall VNF; in particolare l'url della risorsa da scrivere è già cablato nel codice (è stato derivato dal datamodel) tuttavia per costruire l'endpoint intero si utilizzano le informazioni ricavate a runtime quali l'indirizzo ip della VNFs , il VNF instance ID e l'url della risorsa.

8.4.1 Flusso di esecuzione

Il flusso di esecuzione si compone nelle seguenti fasi:

1. **Configurazione dell'interfaccia di management**: l'interfaccia di controllo, quella collegata alla rete di management, necessità di un indirizzo IP per poter essere contatta dall'esterno. Il programma legge il file presente nel datadisk contenente la configurazione iniziale e assegna un indirizzo IP all'interfaccia in maniera statica o utilizzando il DHCP secondo quanto specificato nel file.

2. **Avvio del REST Server**: avendo assegnato un'indirizzo IP all'interfaccia di management, è possibile avviare il server REST utilizzando come endpoint l'indirizzo appena configurato, tuttavia ancora non può essere contattato dall'esterno in quanto l'indirizzo non è stato ancora annunciato all'esterno.

3. **Registrazione sul message bus**: l'agent utilizza le credenziali del tenant fornite nel datadisk per registrarsi sul message bus. Una volta registrato, si annuncia alla rete il proprio endpoint REST pubblicandolo sul topic 'cf-hello' e si fa subscribe al topic 'vnf-hello' per scoprire le VNFs attive. Tuttavia fin quando non vengono specificate quali sono le istanze da utilizzare, gli 'hello' delle varie VNFs vengono scartati.

4. **Configurazione delle istanze di VNFs** a questo punto il programma è ancora inattivo, occorre che qualcuno dall'esterno gli configuri l'identificativo delle VNFs da utilizzare (VNF instance ID). Non appena l'operatore gliele configura (utilizzando le REST API fornite), il programma può iniziare a funzionare. Come prima cosa, conoscendo l'identificatore della VNF da monitorare fa il subscribe al topic di interesse per ricevere lo stato, successivamente inizia a leggere i vari messaggi di 'hello' delle varie VNFs confrontando per ognuno il VNF instance ID fino a quando non trova quello corrispondente alla VNF da configurare e in quel caso si memorizza l'indirizzo IP dell'endpoint della VNF.

5. **Monitoring e configurazione**: il programma risulta configurato, deve solo attendere che gli venga esportato qualcosa per poter compiere il suo lavoro.

Nel corso del lavoro di tesi sono stati implementati due use cases di servizi autoconfiguranti:

8.4.2 Use case: IDS-Firewall Service (IFS)

Tale servizio permette l'interazione tra un IDS e il firewall, in particolare si occupa di configurare automaticamente il firewall non appena l'IDS rileva un attacco verso la rete. Può essere inserito dinamicamente all'interno del grafo, dal momento che sia il firewall che l'IDS risultano agnostici rispetto ad esso. Inoltre, in base alle necessità è possibile fargli monitorare più IDS (purchè tutti utilizzino lo stesso data-model), tuttavia l'implementazione fornita ne utilizza solo uno.

Il funzionamento è il seguente:

1. Non appena l'IDS rileva un attacco pubblica un messaggio sul topic 'ids/attack-detected'; il messaggio esportato è scritto utilizzando la sintassi definita nel data-model dell'IDS (Figura 8.7 a sinistra) e contiene il nome dell'attacco (es. port-scan), l'indirizzo ip dell'attaccante e l'indirizzo ip dell'host attaccato.

2. L'IFS che aveva precedentemente fatto subscribe al topic da monitorare, riceve il messaggio e costruisce una regola utilizzando la sintassi definita nel data-model del firewall (Figura 8.7 a destra).

3. L'IFS configura il firewall, inviando la regola appena costruita.

In Figura 8.6 viene mostrato un esempio di dati ricevuti e inviati dall'IFS.

8.4.3 Use case: Dhcp-Firewall Service (DFS)

Questo servizio permette l'interazione tra un server DHCP e il firewall, in particolare si occupa di configurare automaticamente il firewall non appena il server DHCP

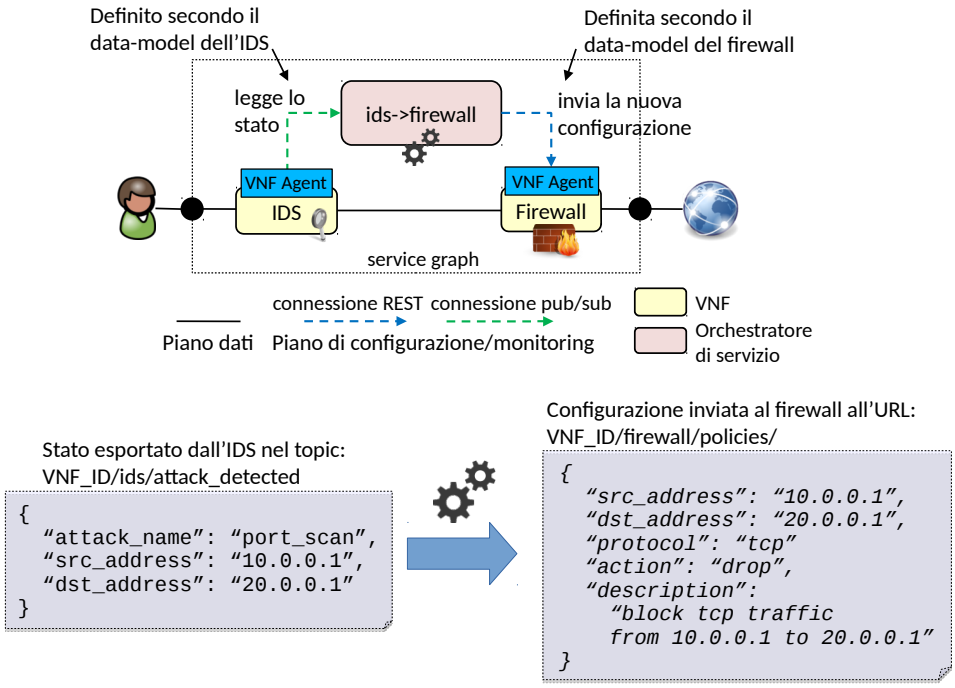


Figura 8.6. Esempio di input ed output dell'IFS.

fornisce un indirizzo IP ad nuovo client per permettergli di inviare e ricevere il traffico sulla rete. Il funzionamento è basato sul fatto che il firewall adotti una politica di tipo *whitelist*, ovvero che di default blocca tutto a meno di eccezioni specificate. Il DFS si occupa proprio di inserire queste eccezioni, che riguardano proprio i client che man mano si collegano alla rete.

Il funzionamento è il seguente:

1. Non appena il server DHCP assegna un nuovo indirizzo esporta tale informazione pubblicando un messaggio sul topic 'dhcp-server/clients'; tale messaggio è scritto utilizzando la sintassi definita nel data model del server DHCP (Figura 8.9 a sinistra) e contiene le informazioni relative al nuovo client quali: mac-address, ip-address, hostname e lease-time.

2. Tale messaggio arriva al DFS, dal momento che questo si è preoccupato di fare subscribe al topic specifico non appena è stato configurato. Il DFS processa il messaggio e costruisce due regole utilizzando la sintassi definita nel data-model del firewall (Figura 8.9 a destra); una abilita il traffico in ingresso dalla rete verso il client e l'altra abilita il traffico in uscita.

```
module ids {
 ...
  container attack_detected{
    config false;
    advertise onchange;
    atomic true;
    leaf attack_name{
      type enumeration{
        enum port_scan;
        enum ping_flood;
      }
    }
    leaf src_address{
      type inet:ip-address;
    }
    leaf dst_address{
      type inet:ip-address;
    }
  }
 ...
```

```
module firewall {
 ...
   list policies {
     atomic true;
     advertise onchange;
     key "id";
     leaf id {
       type string;
     }
     leaf description{
       type string;
       mandatory false;
     }
     leaf action {
       type enumeration {
         enum drop;
         enum reject;
         enum accept;
       }
     }
     leaf protocol {
       type enumeration {
         enum tcp;
         enum udp;
         enum icmp;
         enum all;
        }
     }
     leaf in-interface {
       type leafref {
         path "/interfaces/ifEntry/name";
       }
       mandatory false;
     }
     leaf out-interface {
       type leafref {
         path "/interfaces/ifEntry/name";
       }
       mandatory false;
     }
     leaf src-address {
       type inet:ip-address;
       mandatory false;
     }
     leaf dst-address {
       type inet:ip-address;
       mandatory false;
     }
     leaf src-port {
       type inet:port-number;
       mandatory false;
     }
     leaf dst-port {
       type inet:port-number;
       mandatory false;
     }
   ...
   }
```

Figura 8.7. Estratti di data-models dell'ids e del firewall.

3. il DFS configura il firewall inviando sequenzialmente le due regole appena costruite.

In Figura 8.8 viene mostrato un esempio di dati ricevuti e inviati dal DFS.

8.5 Prototipo di live-migration

Il prototipo di live-migration realizzato permette la migrazione di un NAT tra due VNFs implementate in maniera differente. Nella prima, il NAT è implementato con la tecnologia del Docker container, nell'altra il NAT è realizzato come applicazione ONOS implementato utilizzando la tecnologia Open vSwitch [21]. Ovviamente le due implementazioni fanno riferimento allo stesso un data-model.

Nello scenario utilizzato, agli estremi del NAT sono collegati da un lato un client e dall'altro un server. La demo prevede che una volta istanziato il grafo con il primo NAT, il client esegua una 'wget' sul server e inizi a scaricare un file. Successivamente mentre il download è in corso, si effettua la migrazione del NAT. La demo vuole dimostrare che il download non viene annullato dopo la migrazione ma che continua senza perdere la connessione.

Per realizzare la migrazione è stato sviluppato un componente, il *Migration-Orchestrator* (M.O.), che comunica con tutti i componenti necessari e coordina le

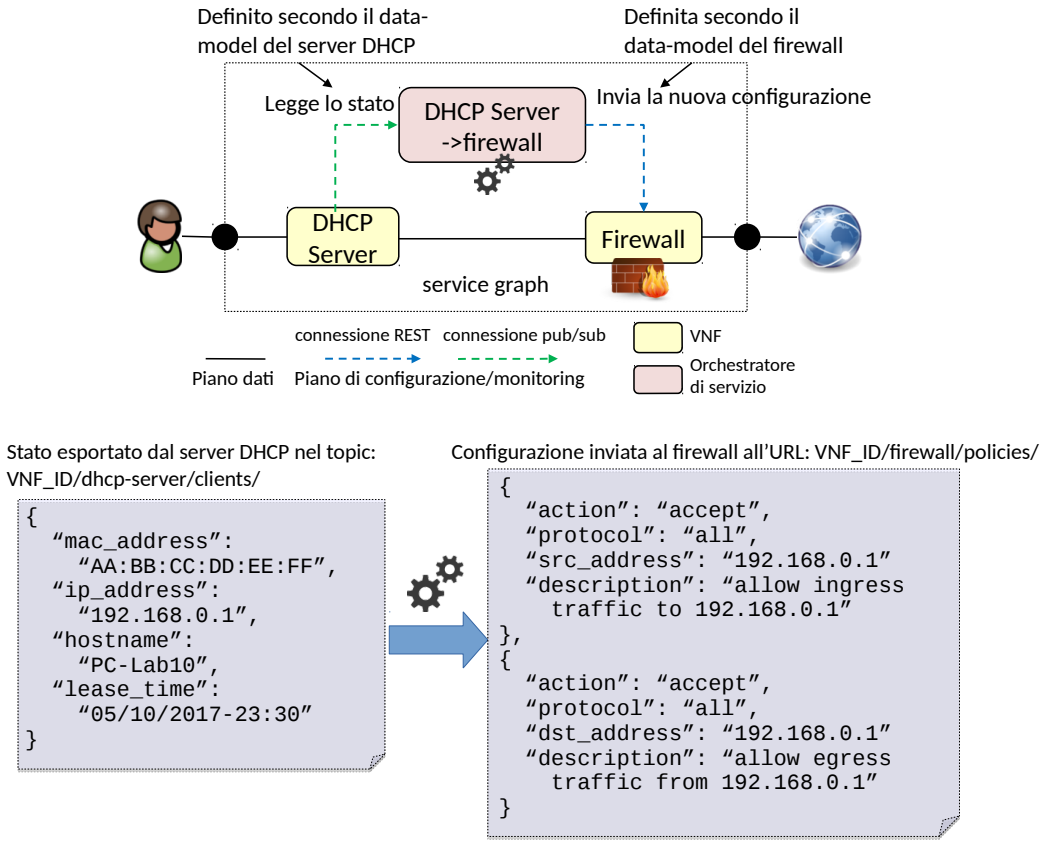


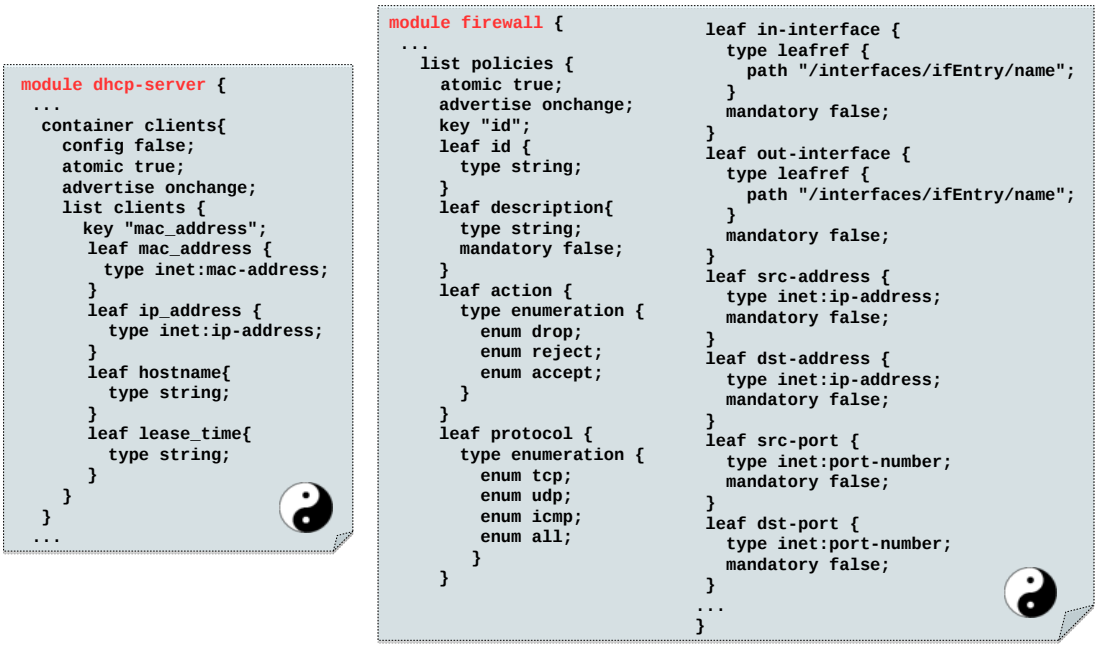
Figura 8.8. Esempio di input ed output dell'DHCP-FW Service.

varie fasi. In particolare interagisce con l'orchestratore globale per permettere l'aggiornamento del grafo e con il configuration-orchestrator (C.O.) per parlare con le VNFs. Infatti il M.O. non interagisce direttamente con le VNFs da migrare ma si appoggia al C.O. Tutta la parte di comunicazione verso gli altri componenti avviene attraverso l'utilizzo di REST API. In particolare, l'orchestratore espone i metodi per istanziare, modificare, cancellare un grafo, mentre il C.O. fornisce quelli per configurare o prelevare lo stato di una VNF.

Di seguito è riportata la sequenza di operazioni svolte dal migration-orchestrator per eseguire la live-migration nello scenario descritto precedentemente.

1. **POST /login**: si esegue il login verso l'orchestratore globale passandogli username e password di un utente valido; se l'autenticazione va a buon fine viene restituito un token da utilizzare in tutte le chiamate successive.

2. **POST /graphs/**: si manda all'orchestratore globale il primo grafo da istanziare. Nella demo realizzata il grafo di servizio comprende un client, un server e il primo NAT che si sceglie di istanziarlo nel dominio Universal Node (UN).



```
module dhcp-server {
 ...
  container clients{
    config false;
    atomic true;
    advertise onchange;
    list clients {
      key "mac_address";
      leaf mac_address {
        type inet:mac-address;
      }
      leaf ip_address {
        type inet:ip-address;
      }
      leaf hostname{
        type string;
      }
      leaf lease_time{
        type string;
      }
    }
  }
 ...
```

```
module firewall {
 ...
   list policies {
     atomic true;
     advertise onchange;
     key "id";
     leaf id {
       type string;
     }
     leaf description{
       type string;
       mandatory false;
     }
     leaf action {
       type enumeration {
         enum drop;
         enum reject;
         enum accept;
       }
     }
     leaf protocol {
       type enumeration {
         enum tcp;
         enum udp;
         enum icmp;
         enum all;
        }
     }
     leaf in-interface {
       type leafref {
         path "/interfaces/ifEntry/name";
       }
       mandatory false;
     }
     leaf out-interface {
       type leafref {
         path "/interfaces/ifEntry/name";
       }
       mandatory false;
     }
     leaf src-address {
       type inet:ip-address;
       mandatory false;
     }
     leaf dst-address {
       type inet:ip-address;
       mandatory false;
     }
     leaf src-port {
       type inet:port-number;
       mandatory false;
     }
     leaf dst-port {
       type inet:port-number;
       mandatory false;
     }
   ...
   }
```

Figura 8.9. Estratti dei data-models del Server DHCP e del firewall.

3. **PUT /graphs/<id>**: si aggiorna il grafo inserendo anche il secondo NAT nel dominio ONOS e i relativi links verso il client e il server; tali links verranno settati con una priorità maggiore in modo che le nuove connessioni tra client e server verranno aperte utilizzando questo NAT.

4. **GET /<tenant-id>/<graph-id>/<vnf-id>/**: si chiede al C.O. di prelevare lo stato corrente dal NAT da migrare (quello istanziato sul dominio UN). I parametri richiesti dal metodo corrispondono ad una tripla che identifica una specifica VNF nel grafo. Il C.O. ha bisogno di sapere qual'è la VNF da cui deve prelevare lo stato; fornita la tripla, il C.O. la utilizza come chiave nella sua tabella delle VNF attive per ricavare l'indirizzo dell'endpoint presso cui deve fare richiesta dello stato.

5. **PUT /status/<tenant-id>/<graph-id>/<vnf-id>/**: si chiede al configuration-orchestrator di iniettare lo stato appena prelevato nel NAT ONOS. I parametri richiesti dal metodo sono gli stessi di quelli di prima; in questo caso occorre cambiare il <vnf-id> per identificare l'altro NAT. A questo punto, dopo aver iniettato lo stato, le connessioni in corso nel NAT 'vecchio' continueranno a persistere anche in questo NAT.

6. **UPDATE /graphs/<id>**: Infine si aggiorna il grafo eliminando il NAT 'vecchio' e i relativi links. La migrazione può considerarsi conclusa.

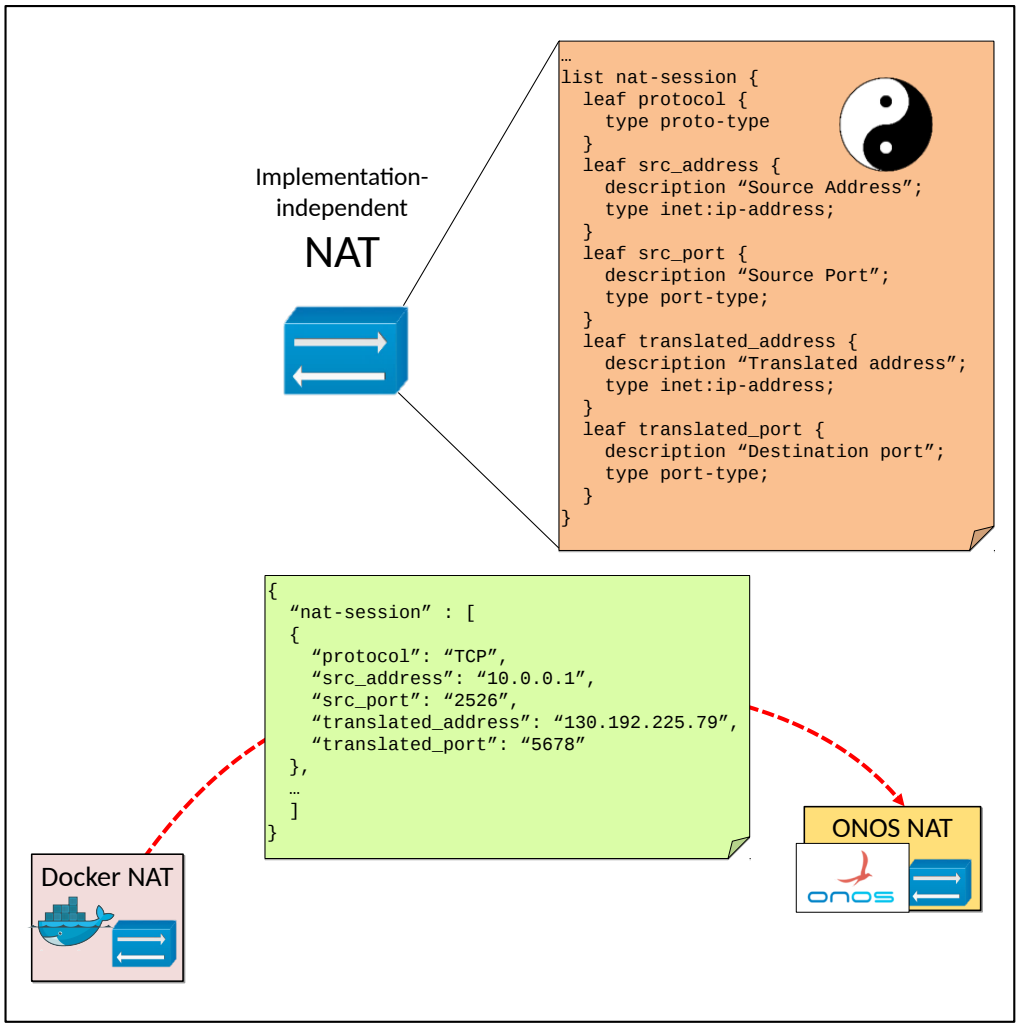


Figura 8.10. Esempio di NAT migration. In alto è presente un frammento di data-model che condividono i due NAT, mentre in basso si può vedere un’istanza di nat-session che viene migrata.

Si noti che evitando l’interazione diretta tra migration-orchestrator e le VNFs, ha permesso di mantenere il M.O. scollegato dal message-bus. Infatti i messaggi che pubblicano sul bus le VNFs per annunciarsi, vengono ricevuti dal configuration-orchestrator ed è lui che si preoccupa di tenere la lista delle VNFs attive e i loro estremi. Il M.O. deve solo fornire gli identificatori delle VNFs al C.O. e scegliere se vuole prelevare o iniettare lo stato, ma non sarà lui ad occuparsene.

Per evitare di eseguire manualmente tutte le operazioni riportate sopra ogni volta che si vuole riprodurre la demo della migrazione, è stata realizzata un interfaccia

grafica che grazie all'ausilio di semplici permette di riprodurre la demo step-by-step nascondendo la chiamate REST. Di seguito è mostrata un'immagine della GUI:

Figura 8.11. Interfaccia grafica del migration-orchestrator.

Capitolo 9

Validazione dei risultati

Al fine di valutare le prestazioni del framework, sono stati eseguiti dei test sull'IDS-Firewall Service (IFS), uno dei due servizi autoconfiguranti realizzati. In particolare si è misurato il tempo medio impiegato dal servizio a processare un evento (cambiamento di stato) al crescere del numero di eventi contemporanei da gestire. Il tempo di processamento di un evento è un tempo significativo in quanto è dato dal contributo di tutti i componenti del servizio, a partire dall'agent della prima VNF che esporta lo stato, fino a quando la configurazione generata viene settata nella seconda VNF.

Per poter simulare più cambiamenti di stato contemporaneamente, è stata creata una *botnet* virtuale in grado di simulare un attacco distribuito contemporaneo. L'esecuzione di tale attacco genera sull'IDS un numero di cambiamenti di stato pari al numero di attaccanti, in quanto ogni attacco viene rilevato singolarmente essendo generato da un indirizzo IP diverso. Questo permette di verificare come reagisce il servizio quando deve processare più cambiamenti di stato contemporaneamente.

9.1 La Botnet

La *botnet* utilizzata è stata creata istanziando, in una macchina fisica a parte, un numero variabile di *bot* all'interno di container docker. Tra gli N bot presenti, N-1 costituiscono gli *slave*, mentre il rimanente costituisce il *master*, il cui compito è quello di coordinare gli slave nell'attacco. Per poter controllare la botnet infatti, è stato realizzato un applicativo in Python composto da una parte client utilizzata dagli slave e una parte server utilizzata dal master. Utilizzando tale programma, gli slave, non appena vengono istanziati, si connettono al master che diventa così in grado di controllarli. In particolare, usando l'applicativo, il master è in grado di inviare dei comandi a tutti gli slave (o anche solo ad alcuni); inoltre, sfruttando il fatto che tutti i bot condividono lo stesso clock (essendo containers docker che girano sullo stesso host), nell'applicazione è stata implementata una funzionalità

che permette di associare un orario al comando da inviare, per fare in modo che tutti gli slave eseguano il comando nello stesso istante invece che eseguirlo non appena viene ricevuto. Inoltre, quando termina l'esecuzione del comando sugli slave, essi invieranno automaticamente al master un file contenente il log dell'output del comando eseguito, in modo tale che il master possa conoscerne l'esito.

9.2 Test

Lo scenario in cui sono stati eseguiti i test è quello mostrato in Figura 9.2.

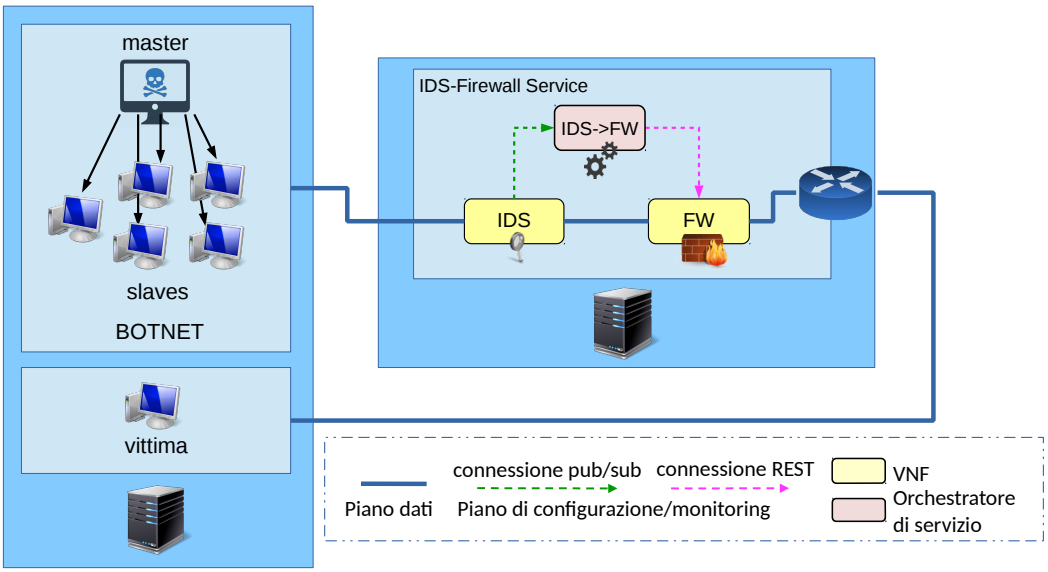


Figura 9.1. Scenario utilizzato per i test.

Come si può notare, sia botnet che la vittima (anche essa è costituita da una container docker) sono istanziati sulla stessa macchina fisica, mentre l'IFS si trova in un'altra macchina. Tutta via il data-plane prevede che pacchetti generati dalla botnet raggiungeranno l'altra macchina, saranno processati dall'IFS e infine arriveranno alla vittima. In Figura 9.2 sono riportate le specifiche tecniche delle due macchine fisiche. L'attacco utilizzato per il test è stato un *ping flood*; utilizzando il master della botnet, si è dato l'ordine agli slave di eseguire 100 ping al secondo verso la vittima. Il test è stato ripetuto più volte aumentando di volta in volta il numero di attaccanti, in modo tale che sull'IDS venisse esportato un numero sempre più grande di cambiamenti di stato.

PC 1: botnet and server	
O.S	Ubuntu 14 64 bit
CPU	Intel Core i7-3770 @ 3.40Ghz*8
RAM	32 Gb
HD	1 Tb
Eth	1000 Mbit/s

PC 2: Universal Node	
O.S	Ubuntu 14 64 bit
CPU	Intel Core i7-6700 @ 3.40Ghz*8
RAM	32 Gb
HD	1 Tb
Eth	1000 Mbit/s

Figura 9.2. Specifiche tecniche delle macchine utilizzate nei test.

9.3 Risultati

In Figura 9.3 è riportato il grafico che mostra i risultati del test. Come previsto, all'aumentare del numero di eventi contemporanei da gestire, diminuiscono le prestazioni del framework. Nonostante il maggiore carico di lavoro sull'agent dell'IDS, maggiore traffico sul message bus, un maggior carico di lavoro sull'orchestratore di servizio (per ogni aggiornamento di stato ricevuto deve potenzialmente elaborare una nuova configurazione per il firewall) ed un maggiore carico di lavoro all'agent del firewall, la diminuizione della prestazioni è da attribuire al message bus. Infatti, come si nota dal grafico in Figura 9.4, all'aumentare dei cambiamenti di stato contemporanei, la maggior parte del tempo di processamento è impiegato dal message bus per esportare lo stato costituendo dunque un collo di bottiglia del sistema.

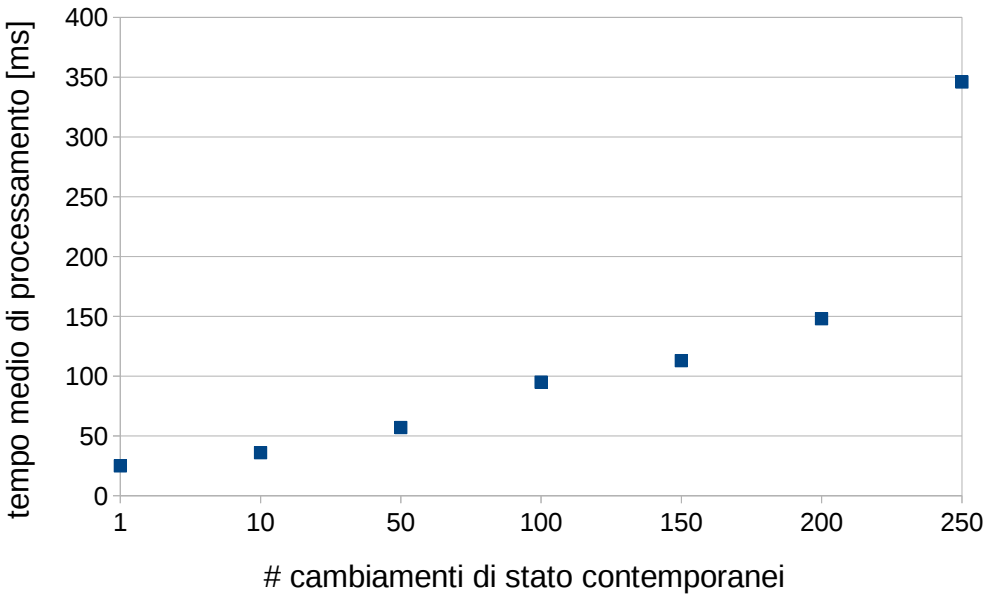


Figura 9.3. Risultati.

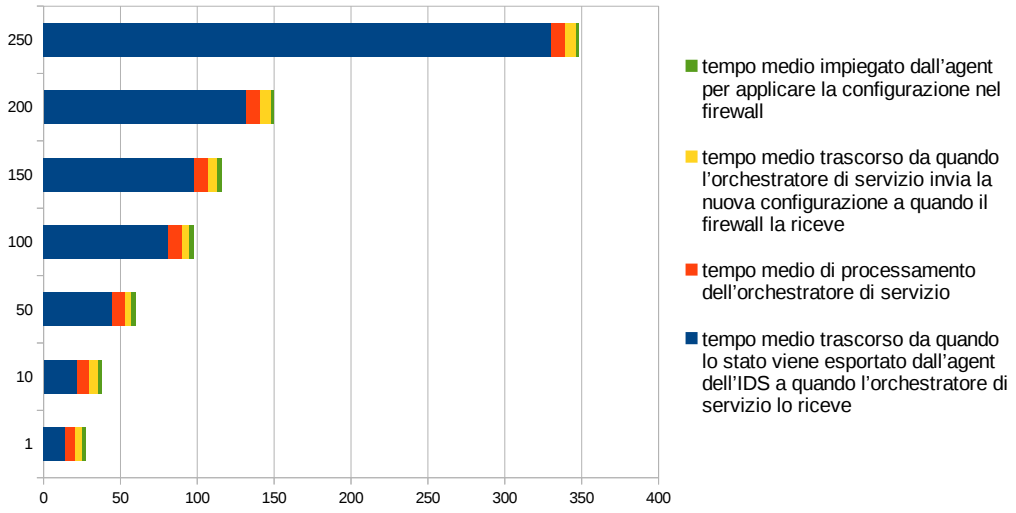


Figura 9.4. Tempo di processamento dei vari moduli coinvolti nel servizio autoconfigurante.

Capitolo 10

Conclusioni e sviluppi futuri

Per concludere, il framework proposto ha rispettato le aspettative in quanto ha permesso di risolvere i problemi per i quali è stato definito; i prototipi dei servizi realizzati ne sono stati la prova. L'obiettivo del framework infatti, è stato quello di permettere di configurare dinamicamente le VNFs in maniera indipendente dall'implementazione, fornendo degli strumenti in grado di configurare anche le VNFs la cui configurazione dipende dal cambiamento di stato di altre.

Grazie alla sua architettura, il framework può essere integrato all'interno di architetture di orchestrazioni già esistenti senza troppe modifiche; infatti, per rendere compatibili delle VNFs già esistenti con il framework, basta sviluppare per ognuna di esse un agent, che permette di disaccoppiare la VNF e il data-model, senza doverne cambiare l'implementazione.

Le funzionalità offerte dagli orchestratori di servizio possono essere utilizzate laddove si abbia la necessità di configurare una VNF in base ai cambiamenti di stato che si verificano in altre il cui stato necessita di essere monitorato. Gli scenari presentati in questa tesi sono soltanto alcuni di tutti quelli in cui è possibile usare il framework; ad esempio un ulteriore scenario potrebbe essere quello in cui è presente un orchestratore di servizio che monitora i parametri di stato di una VNF e, al variare di questi, decide se è necessario effettuare scale in/out delle istanze di VNFs. Inoltre, come si è visto dal prototipo di migrazione, il framework può essere utilizzato nei casi in cui occorra effettuare una migrazione tra due VNF implementate in maniera differente, dove non è possibile migrare l'intera istanza da una VNF all'altra.

Dal punto di vista prestazionale invece, come hanno dimostrato i test, è possibile migliorare le prestazioni del framework, migliorando il message bus in quanto si è dimostrato essere un collo di bottiglia. Si potrebbe provare ad utilizzare message bus più efficiente, tuttavia questo è lasciato come lavoro futuro, in quanto questa tesi si è focalizzata più sulle funzionalità del framework che non sulle effettive prestazioni.

Appendice A

YANG data-models

In questa appendice vengono riportati i data-models associati alle VNF sviluppate.

A.1 DHCP

```
module config-dhcp-server {
    namespace "http://netgroup.ipv6.polito.it/dhcp";
    prefix "dhcp";
    import ietf-inet-types {
        prefix inet;
    }
    container server{
        description "name='DHCP Server configuration'";
        atomic true;
        config true;
        advertise onchange;
        leaf defaultLeaseTime {
            type string;
        }
        leaf maxLeaseTime {
            type string;
        }
      leaf subnet {
         type inet:ipv4-address;
      }
      leaf subnetMask {
            type inet:ipv4-address;
      }
      leaf defaultGw {
```

```
25            type inet:ipv4-address;
26        }
27        leaf dnsPrimaryServer {
28            type inet:ip-address;
29        }
30        leaf dnsSecondaryServer {
31            type inet:ip-address;
32            mandatory "false";
33        }
34        leaf dnsDomainName {
35            type inet:domain-name;
36        }
37        list sections {
38              key "sectionStartIp";
39              leaf sectionStartIp {
40                  type inet:ipv4-address;
41              }
42              leaf sectionEndIp {
43                  type inet:ipv4-address;
44              }
45          }
46      }
47      container clients{
48      description "name='DHCP Clients'";
49          atomic true;
50           config false;
51          advertise onchange;
52          list clients {
53              key "mac_address";
54              leaf mac_address {
55                  type inet:mac-address;
56              }
57              leaf ip_address {
58                  type inet:ipv4-address;
59              }
60              leaf hostname{
61                  type string;
62              }
63              leaf lease_time{
64                  type string;
65              }
```

```
66          }
67      }
68  }
```

A.2 NAT

```
1   module config-nat {
2       namespace "http://netgroup.ipv6.polito.it/vnf";
3       prefix "nat";
4       import ietf-inet-types {
5           prefix inet;
6       }
7       container nat{
8           description "name='Nat";
9           leaf private-interface {
10              advertise ondemand;
11             type leafref {
12                  path "/interfaces/ifEntry/id";
13             }
14          }
15          leaf public-interface {
16              advertise ondemand;
17             type leafref {
18                  path "/interfaces/ifEntry/id";
19             }
20          }
21      container nat-table{
22          list nat-session {
23              key "hash";
24              atomic true;
25              config false;
26              advertise periodic;
27              advertise-period 1;
28              leaf hash {
29                  type hex-string;
30              }
31           leaf protocol {
32              type proto-type;
33           }
34           leaf src_address {
```

```
35              type inet:ip-address;
36          }
37          leaf src_port {
38              type inet:port-number;
39          }
40          leaf translated_address {
41              type inet:ip-address;
42          }
43          leaf translated_port {
44              type inet:port-number;
45          }
46        }
47    }
48 }
```

A.3 Firewall

```
1  module config-firewall {
2      namespace "http://netgroup.ipv6.polito.it/vnf";
3      prefix "firewall";
4      import ietf-inet-types {
5          prefix inet;
6      }
7      container firewall {
8          list policies {
9              atomic true;
10             config true;
11             advertise onchange;
12             key "id";
13             leaf id {
14                 type string;
15             }
16             leaf description{
17                 type string;
18                 mandatory false;
19             }
20             leaf action {
21                 type enumeration {
22                     enum drop;
23                     enum reject;
```

```
24                      enum accept;
25                  }
26              }
27              leaf protocol {
28                  type enumeration {
29                      enum tcp;
30                      enum udp;
31                      enum icmp;
32                      enum all;
33                  }
34              }
35              leaf in-interface {
36                  type leafref {
37                      path "/interfaces/ifEntry/name";
38                  }
39                  mandatory false;
40              }
41              leaf out-interface {
42                  type leafref {
43                      path "/interfaces/ifEntry/name";
44                  }
45                  mandatory false;
46              }
47              leaf src-address {
48                  type inet:ip-address;
49              }
50              leaf dst-address {
51                  type inet:ip-address;
52                  mandatory false;
53              }
54              leaf src-port {
55                  type inet:port-number;
56                  mandatory false;
57              }
58              leaf dst-port {
59                  type inet:port-number;
60                  mandatory false;
61              }
62          }
63          list blacklist {
64              key "url";
```

```
65              description "name='URL', tooltip='eg:
                   www.youtube.com'";
66              advertise onchange;
67              leaf url{
68                  type string;
69              }
70          }
71          list whitelist {
72              key "url";
73              description "name='URL', tooltip='eg:
                   www.youtube.com'";
74              advertise onchange;
75              leaf url{
76                  type string;
77              }
78          }
79      }
80  }
```

A.4 IDS

```
1  module config-ids {
2      namespace "http://netgroup.ipv6.polito.it/vnf";
3      prefix "ids";
4      import ietf-inet-types {
5          prefix inet;
6      }
7      container ids {
8          container configuration{
9              atomic true;
10             config true;
11            advertise onchange;
12            leaf network_to_defend{
13             description "name='Network to defend
                  (IP/Netmask)";
14               type inet:ipv4-address;
15            }
16            list attack_to_monitor{
17               key "name";
18             leaf name{
```

```
19                  type enumeration {
20                          enum port_scan;
21                          enum ping_flood;
22                      }
23                }
24             }
25         }
26         container attack_detected{
27             atomic true;
28             config false;
29             advertise onchange;
30             leaf attack_name{
31                 type enumeration {
32                 enum port_scan;
33                     enum ping_flood;
34                 }
35             }
36             leaf src_address{
37                 type inet:ip-address;
38             }
39             leaf dst_address{
40                type inet:ip-address;
41             }
42         }
43     }
44 }
```

Bibliografia

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

[2] H. Jamjoom S. Rajagopalan D. Williams e A. Warfield. «Split/merge: System support for elastic execution in virtual middle- boxes.» In: *NSDI* (2013), pp. 227–240.

[3] C. Prakash R. Grandl J. Khalid S. Das A. Gember-Jacobson R. Viswanathan e A. Akella. «Opennf: Enabling innovation in network function control». In: *ACM SIGCOMM Computer Communication Review* (2014), pp. 163–174.

[4] A. Gember-Jacobson e A. Akella. «Improving the safety, scalability, and efficiency of network function state transfers». In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (2015), pp. 43–48.

[5] V. Hilt L. Nobach I. Rimac e D. Hausheer. «Slim: Enabling efficient, seamless nfv state migration». In: *2016 IEEE 24th International Conference on Network Protocols (ICNP)* (nov. 2016).

[6] H. Yu S. Luo e L. Vanbever. «Swing state: Consistent updates for stateful and programmable data planes». In: *Proceedings of the Symposium on SDN Research* (2017), pp. 115–121.

[7] G. Gibb M. Izzard N. McKeown J. Rexford C. Schlesinger D. Talayco A. Vahdat G. Varghese P. Bosshart D. Daly e D. Walker. «P4: Programming protocol-independent packet processors». In: *SIGCOMM Comput. Commun. Rev* (giu. 2014), pp. 87–95.

[8] D. Williams S. Rajagopalan e H. Jamjoom. «Pico replication: A high availability framework for middleboxes». In: *Proceedings of the 4th Annual Symposium on Cloud Computing* (2013), pp. 1–15.

[9] A. Alsudais M. Kablan e F. Le. «Stateless network functions: Breaking the tight coupling of state and processing». In: *NSDI* (2017), pp. 97–112.

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

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

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

[13] *UNIFY website*. URL: https://www.fp7-unify.eu.

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

[15] Ivano Cerrato et al. «Towards Dynamic Virtualized Network Services in Telecom Operator Networks». In: *Computer Networks* 92 (2015), pp. 380–395.

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

[17] A. Prieto A. Tripathy E. Nilsen-Nygaard A. Bierman A. Clemm E. Voit e B. Lengyel. *Subscribing to yang datastore push updates*. Rapp. tecn. Giu. 2017. URL: http://www.ietf.org/internet-drafts/draft-ietf-netconf-yang-push-07.txt.

[18] *The conntrack-tools user manual*. URL: http://conntrack-tools.netfilter.org/manual.html.

[19] *What is ebtables?* URL: http://ebtables.netfilter.org/.

[20] *Snort*. URL: https://www.snort.org/.

[21] *Open vSwitch*. URL: http://vswitch.org.