POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Migrazione di funzioni di rete virtualizzate con stato tra domini tecnologici eterogenei

Relatore
prof. Fulvio Risso

Supervisore:
dott. Ivano Cerrato

Candidato
Nicola Chiarappa

A.A. 2017/2018

Alla mia famiglia

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

La virtualizzazione delle funzioni di rete, nota come Network Function Virtualization (NFV) [1], ha introdotto un sostanziale cambio di paradigma nel modo in cui vengono realizzate le reti, operando un disaccoppiamento tra hardware e software. Grazie ad NFV, infatti, le funzioni di rete (ad esempio router, NAT, firewall, ecc.) non sono più eseguite all'interno di dispositivi hardware di tipo specializzato e proprietario, ma diventano applicazioni software istanziate all'interno di server commodity general-purpose [2]; tali applicazioni prendono il nome di Virtual Network Functions (VNF). Negli ultimi anni l'interesse verso NFV è cresciuto sempre più tra i fornitori dei servizi di rete [3, 4]: l'idea è quella di sfruttare la grande flessibilità offerta dal software per introdurre dinamicità nel dislocamento e nella gestione delle proprie funzioni di rete.

Questa tesi si inserisce nell'ambito dell'orchestrazione di VNF e si focalizza in particolare sulla migrazione delle VNF con stato. Per far sì che le VNF possano essere facilmente manutenute e aggiornate, infatti, si pone il problema di trovare una maniera per poter migrare senza soluzione di continuità la VNF stessa. Ciò si traduce non solo nel problema di dover migrare il traffico che la attraversa, ma anche dello stato al suo interno. A tal fine, il presente lavoro di tesi propone un algoritmo per la migrazione delle VNF con stato. Tale algoritmo verrà validato mediante un prototipo sviluppato nell'ambito di un framework già esistente, IOVnet, curato dal Netgroup [5] del Politecnico di Torino.

L'elaborato è strutturato come segue:

- **Capitolo 2:** Presenta il problema della migrazione di una VNF, le motivazioni alla sua base e infine viene spiegato l'obiettivo della tesi;
- **Capitolo 3:** Presenta le soluzioni attualmente disponibili allo stato dell'arte per gestire una migrazione di una VNF e ne elenca i principali vantaggi e svantaggi;
- **Capitolo 4:** Illustra la soluzione proposta da questa tesi, spiegando l'algoritmo per la migrazione di una VNF con stato;
- **Capitolo 5:** Analizza le parti più importanti del prototipo sviluppato per effettuare la validazione dell'algoritmo per la migrazione di una VNF con stato;

- **Capitolo 6:** Mostra i test effettuati per validare la soluzione proposta e riporta i risultati ottenuti;
- **Capitolo 7:** Espone le conclusioni e i possibili sviluppi futuri.

Capitolo 2

Migrazione di una VNF

Le funzioni di rete (successivamente abbreviate in NF, ovvero network functions) sono sistemi che esaminano e spesso modificano pacchetti in transito sulla rete in maniera sofisticata: alcuni esempi possono essere intrusion detection systems (IDS), load balancers, caching proxies, ecc. Recentemente si è vista sempre più una tendenza e un crescente interesse nel sostituire hardware dedicato (molto spesso di tipo proprietario), preposto alle funzioni di una NF, con software che viene eseguito su hardware commodity. Questo trend ha preso il nome di NFV, o Network Function Virtualization [1] mentre le funzioni di rete virtualizzate prendono il nome di VNF (Virtual Network Functions).

Parallelamente a quest'ultimo si è affiancato il paradigma SDN (Software Defined Networking). Esso attua la separazione del data plane dal control plane il quale è centralizzato in un elemento chiamato controller. Quest'ultimo, tramite un protocollo SDN (ad esempio Openflow [6]) permette di "programmare" il dataplane. Nel control plane centralizzato, perciò, attraverso applicazioni dedicate eseguite sul controller, si possono decidere politiche di steering sul traffico di rete per molteplici scopi. Uno di questi ultimi può essere, ad esempio, il bilancio del carico su diverse NF disponibili [7–11] attraverso la ridistribuzione dinamica del processamento dei pacchetti su istanze multiple di una stessa funzione di rete.

Insieme, NFV e SDN hanno aiutato a raggiungere importanti obiettivi nel mondo delle reti, tra cui:

1. la possibilità di creare servizi di rete complessi, costituiti da diverse funzioni di rete;
2. la capacità di manipolare il traffico di rete e quindi di creare percorsi per il traffico stesso;
3. la minimizzazione dei costi fissi e dei costi operativi (CAPEX e OPEX) di una NF, grazie alla possibilità di consolidare più VNF su un singolo server, escludendo la necessità di avere più macchine dedicate separate per ogni funzione di rete;
4. la riduzione del tempo di *deploy* di un servizio.

Una delle problematiche nel contesto appena presentato e argomento principale di questo lavoro di tesi è la migrazione di una VNF, che si traduce nella migrazione del traffico che

l'attraversa e dello stato necessario per poterlo processare. L'istanza di VNF migrante verrà chiamata d'ora in poi istanza sorgente della VNF (istanza Src) e l'istanza su cui verrà migrata la VNF verrà chiamata istanza destinazione della VNF (istanza Dst).

Diverse sono le motivazioni per cui si può decidere di migrare una VNF. Di seguito vengono illustrate quelle principali di cui si è tenuto conto in questa tesi.

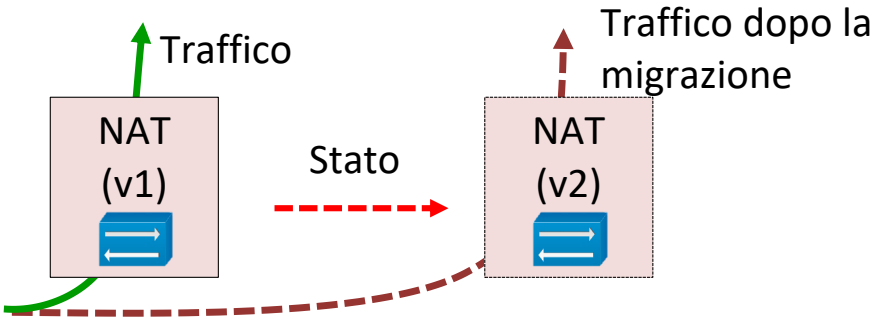


Figura 2.1. Esecuzione di una versione più aggiornata della VNF.

Esecuzione della versione più aggiornata della network function (Fig. 2.1): per motivi di sicurezza o nell'ottica dell'introduzione programmata di nuove funzionalità nel corso del tempo, si può decidere di far processare il traffico sempre dall'ultima versione software disponibile della NF.

Monitoraggio del carico di rete e ottimizzazione dei costi: si può decidere, infatti, di dover effettuare delle operazioni di scale-in/scale-out. Nel primo caso, essa è necessaria quando una istanza di NF è in sovraccarico; nel secondo caso, viceversa, quando una istanza di NF sta processando poco traffico e consumando risorse che possono essere liberate ai fini della ottimizzazione dei costi di gestione.

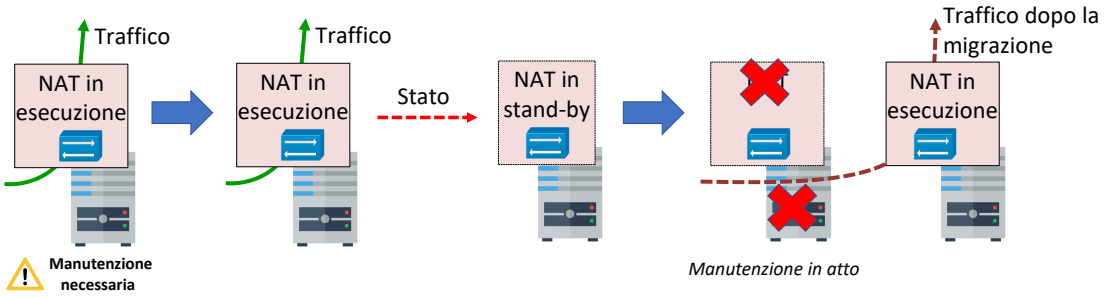


Figura 2.2. Migrazione di una VNF dovuta ad esigenze di manutenzione.

Esigenze di manutenzione (Fig. 2.2): all'interno di un datacenter, per esempio, può verificarsi la necessità di dover effettuare una manutenzione sull'hardware su cui

è stata istanziata la funzione di rete. In questo caso, una migrazione di quest'ultima permette al traffico che la attraversa di non essere interrotto in alcun modo.

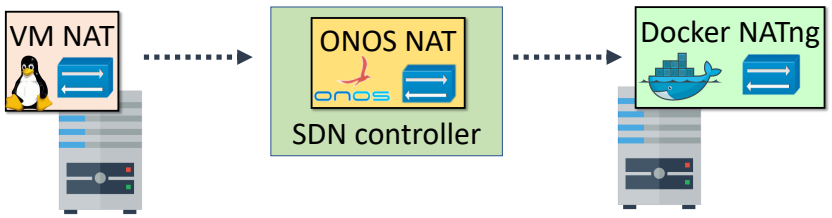


Figura 2.3. Cambio dell'ambiente di esecuzione di una VNF.

Cambio dell'ambiente di esecuzione (Fig. 2.3): si può pensare, ad esempio, di migrare una istanza di funzione di rete sottoforma di una applicazione SDN ad una nuova istanza della stessa sottoforma di virtual machine.

Come detto precedentemente, la migrazione di una VNF presuppone la migrazione del suo stato. Definiamo stato di una funzione di rete quell'insieme di strutture dati che sono lette e a volte modificate o cancellate per processare pacchetti di rete (ad esempio la tabella di routing per un router). I pacchetti di rete, a loro volta, possono appartenere ad uno o più "flussi". La definizione di flusso di rete può variare in base alla funzione di rete in esame (ad esempio una connessione TCP nel caso di un NAT). Lo stato può essere creato, in base alla funzione di rete in esame, in diverse occasioni (ad esempio quando una nuova connessione TCP, o flusso, è stabilita nel caso di un NAT).

Diverse possono essere le problematiche da affrontare quando si vuole effettuare la migrazione di una VNF. Tra di esse annoveriamo:

- **Garantire l'accuratezza della funzione di rete:** quando si vuole migrare lo stato di una VNF da un'istanza sorgente ad una destinazione può capitare che arrivino dei pacchetti all'istanza sorgente prima che il trasferimento di stato sia stato completato. Se non si adottano le dovute precauzioni, i cambiamenti allo stato della NF sorgente dovuti a questi pacchetti potrebbero essere persi o avvenire in maniera fuori ordine, causando un funzionamento errato della VNF stessa. Ad esempio, un firewall potrebbe non rilevare un attacco in corso e non scartare pacchetti relativi all'attacco stesso. In maniera simile, quando lo stato è copiato tra diverse istanze di NF, se si permette che i cambiamenti di stato possano avvenire contemporaneamente sia sull'istanza sorgente che su quella destinazione, potrebbero verificarsi dei problemi di inconsistenza.

- **Gestire le problematiche di overhead:** la migrazione deve avvenire in maniera quanto più possibile efficiente. Il trasferimento e la condivisione di stato tra due istanze di NF consuma risorse computazionali e di rete. In più, se si desiderano

garanzie quali mancanza di packet-loss, di reordering e di inconsistenza tra diverse versioni dello stesso stato si incorre nella necessità di introdurre il buffering dei pacchetti, che provoca latenza e problemi di overhead di memoria. Per gestire questa problematica si può pensare a controllare la migrazione di traffico e di stato con una granularità molto fine (ad esempio migrando un po' alla volta porzioni di stato che appartengono ad un insieme di flussi) e a specificare quali garanzie si desidera di volta in volta (ad esempio mancanza di packet-loss, non riordino dei pacchetti).

- **Gestire le diverse tipologie di stato:** quando si vuole effettuare una operazione di scaling per ribilanciare il carico di diverse istanze della stessa NF bisogna tener conto del fatto che esse operano grazie a dello stato che potrebbe appartenere anche a più di un flusso (ad esempio un IDS che mantiene dei contatori di pacchetti per ogni end-host). Questo aspetto richiede che, in caso di condivisione di stato tra più istanze di NF, tutte debbano garantire consistenza in ogni momento. Questo significa che ad esempio, nel caso in cui una istanza venga spenta, sia necessario effettuare delle operazioni di merging tra lo stato dell'istanza appena spenta e le altre istanze di NF.

- **Gestire correttamente la creazione di nuovo stato:** durante la migrazione di una funzione di rete bisogna garantire che si possa creare eventualmente nuovo stato causato da traffico che prima non attraversava la funzione di rete.

- **Garantire service level agreements (SLAs)**: ad esempio, si possono avere dei vincoli sul downtime della funzione di rete, sul massimo tempo di migrazione accettabile o si può desiderare di non avere perdita di pacchetti o reordering degli stessi.

Per risolvere le problematiche appena introdotte la tesi presenta un algoritmo per la migrazione di una VNF tra domini tecnologici eterogenei.

Capitolo 3

Stato dell'arte

Mentre in letteratura vi sono diversi metodi [7–9] per gestire lo stato della rete e lo stato delle funzioni di rete virtualizzate in maniera separata, vi sono poche soluzioni proposte per la loro gestione coordinata, la quale è necessaria per poter effettuare una corretta migrazione di una VNF, oggetto della presente tesi. Si distinguono in particolare, tre diversi tipi di soluzioni:

1. la migrazione VM-like;
2. la migrazione di tipo stateful;
3. la migrazione di tipo stateless.

In questo capitolo, esse verranno illustrate e ne verranno evidenziate vantaggi e svantaggi.

3.1 Le soluzioni di migrazione VM-like

Alcuni lavori [12–16] propongono la migrazione di una NF virtualizzata in una virtual machine (VM) e quindi adottano come soluzione la migrazione di tutta la VM stessa. Questa soluzione nasce dalla considerazione del fatto che molte funzioni di rete mantengono la maggior parte del loro stato interno in memoria per ridurre i ritardi sul singolo pacchetto processato. Alla base, quindi, vi è la necessità di migrare la memoria da una VM ad un'altra. Questa operazione può essere fatta mediante la combinazione delle seguenti metodologie:

1. *Pre-copy*: il processing è affidato alla macchina virtuale sorgente e le pagine di memoria sono copiate alla macchina virtuale destinazione durante round successivi:
   - il primo round copia tutte le pagine di memoria attive;
   - i round successivi copiano solo le pagine di memoria modificate (le cosiddette pagine *dirty*) nell'intervallo di tempo trascorso tra due round consecutivi;
2. *Stop-and-copy*: la macchina virtuale sorgente viene sospesa del tutto e le pagine di memoria modificate rimanenti vengono copiate sulla macchina virtuale destinazione; una volta che il trasferimento delle pagine di memoria viene completato, il processing viene ripreso sulla macchina virtuale di destinazione;

3. *Demand-migration*: la macchina virtuale destinazione inizia il processing e all'occorrenza richiede alla macchina virtuale sorgente le pagine di memoria non disponibili, quando vi si cerca di accedere.

Gli algoritmi che migrano le VM nella maggior parte dei casi eseguono molteplici round di pre-copy e un round finale di stop-and-copy una volta che la percentuale di pagine di memoria modificate tra un round e il successivo scende al di sotto di una certa soglia prestabilita. Questo approccio presenta diversi svantaggi, considerando il caso in esame della migrazione di una VNF, infatti:

- la gestione della migrazione del traffico e dello stato della VNF viene eseguita in maniera congiunta ma in blocco e non ad una granularità fine (ad esempio migrando un po' alla volta porzioni di traffico e del relativo stato);
- la fase di *pre-copy* potrebbe risultare inefficiente, dal momento in cui potrebbe crearsi continuamente nuovo stato sull'istanza sorgente e/o modificarsi frequentemente stato preesistente, impedendo di fatto la partenza della fase di stop-and-copy;
- la fase di *stop-and-copy* provoca un periodo di indisponibilità (il cosiddetto *downtime*) nel quale i pacchetti vengono fondamentalmente scartati; questo provoca ovviamente anche la perdita di eventuali cambiamenti di stato sulla NF, dovuti per l'appunto ai pacchetti scartati.

Questo approccio, in definitiva:

- permette unicamente la clonazione dell'istanza della NF nella sua interezza (ovvero la clonazione di tutta la memoria necessaria alla macchina virtuale);
- richiede che la virtual machine, e quindi il processing dei pacchetti che si effettua su di essa, siano sospesi durante la migrazione;
- implica, a causa del punto precedente, che tutto il traffico - incluso il traffico altamente sensibile al ritardo - subisca latenze notevoli durante la migrazione;
- non permette di effettuare il merging di stato tra istanze diverse della stessa NF, non distinguendo di fatto tra stato condiviso e stato non condiviso tra due istanze;
- migrando tutta la memoria non permette di effettuare la migrazione di stato (e conseguentemente di traffico) ad una granularità fine (ad esempio migrando una porzione di traffico e del relativo stato);
- obbliga a dover copiare un'ingente quantità di memoria (di cui una minima parte è rappresentata dallo stato della NF) da una macchina virtuale sorgente ad una destinazione, il che implica overhead in termini di tempo e risorse.

Per cercare di mitigare gli effetti negativi descritti, [15, 16] propongono un'ottimizzazione chiamata *daisy chaining* mostrata in Fig. 3.1 che consiste nell'eseguire parallelamente due istanze della VNF e nel far in modo che il potenziale nuovo stato che potrebbe crearsi sull'istanza sorgente venga a crearsi invece direttamente sull'istanza destinazione,

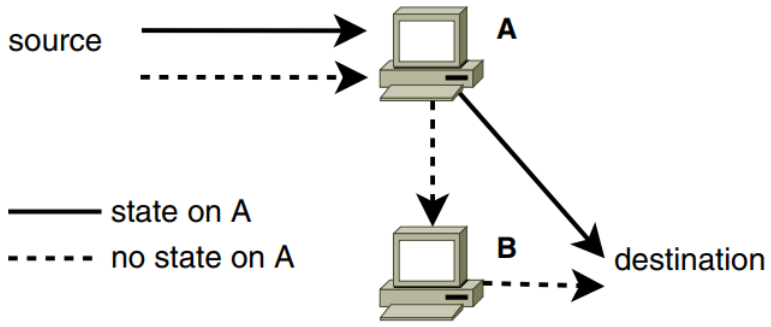


Figura 3.1. L'architettura daisy-chain (proveniente da [15]).

mediante il redirezionamento (direttamente da istanza sorgente a istanza destinazione) dei pacchetti che potrebbero far realizzare questa situazione. I pacchetti inoltrati relativi a nuovi flussi, perciò, verranno processati solo dall'istanza destinazione, come si vede in Fig. 3.1.

3.2 Le soluzioni stateful

Le soluzioni che verranno presentate in questa sezione presuppongono che lo stato della NF sia residente all'interno della NF stessa[1] e non hanno come vincolo il fatto che la NF sia eseguita all'interno di una VM.

Verranno illustrate in maniera più dettagliata OpenNF [17] e DiST [18], poichè principalmente su di esse è stato basato il lavoro della presente tesi.

3.2.1 OpenNF

Analizzando le diverse funzioni di rete OpenNF [17] classifica il loro stato basandosi su quanti flussi di rete leggono e/o modificano il suddetto stato per essere processati. In particolare, lo stato può essere classificato come di tipo:

1. *Per-flow*: quando il processing di uno e un solo flusso richiede la lettura/modifica dello stato in esame (ad esempio mapping per la traduzione in un NAT inerente ad una sola connessione TCP);
2. *Multi-flow*: quando il processing di due o più flussi (ma non tutti) richiede la lettura/modifica dello stato in esame (ad esempio in un firewall, un contatore di pacchetti ricevuti aventi stesso indirizzo IP sorgente);
3. *All-flow*: quando il processing di tutti i flussi richiede la lettura/modifica dello stato in esame (ad esempio in un load balancer, l'algoritmo di load balancing per l'assegnazione di un back-end server e i dati sul carico sui diversi back-end server).

[1] in contrapposizione alle soluzioni stateless che presuppongono che lo stato sia centralizzato in un datastore remoto, condiviso tra le diverse istanze

OpenNF [17] propone una architettura che ha come scopo principale il controllo efficiente e coordinato durante la migrazione sia dello stato interno delle funzioni di rete sia dei flussi di rete che le attraversano. L'idea principale di OpenNF è quella di usare una applicazione di controllo centrale per gestire la migrazione di stato e flussi da una istanza di NF ad un'altra. Un apposito orchestratore, chiamato *OpenNF controller*, si occupa di recuperare lo stato dall'istanza sorgente e di copiarlo sull'istanza destinazione e di migrare il traffico tra le due in maniera opportuna. Durante il processo i pacchetti sono accodati in un buffer nel controller fino al completo trasferimento di stato nell'istanza destinazione. Una volta completato il trasferimento di stato, i pacchetti precedentemente accodati vengono poi mandati all'istanza destinazione. Requisito fondamentale del lavoro di OpenNF è quello di lavorare all'interno di una rete SDN.

L'architettura di OpenNF

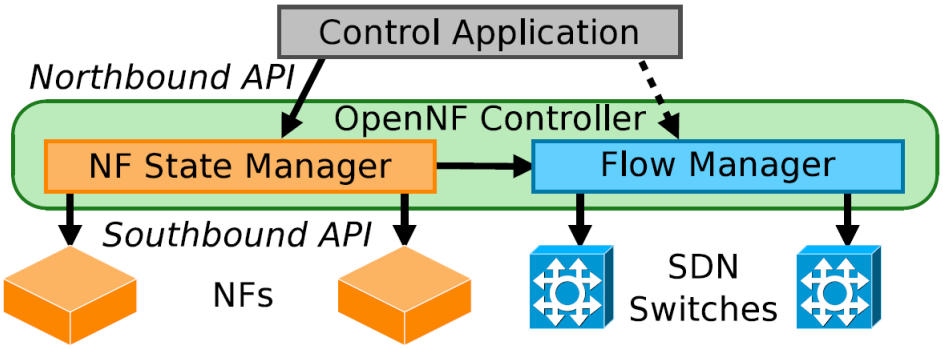


Figura 3.2. L'architettura di OpenNF (proveniente da [17]).

Come mostrato in Figura 3.2, l'architettura di OpenNF è costituita da diverse componenti.

Le applicazioni di controllo e il controller OpenNF

Partendo dall'alto, in primis, ci sono le applicazioni di controllo che hanno i seguenti compiti principali:

- determinare il preciso insieme di flussi che le varie istanze di NF devono processare;
- istruire il controller affinchè fornisca ad ogni istanza destinazione di NF in esame lo stato da migrare;
- chiedere al controller di garantire determinate proprietà di migrazione (ad esempio evitare il reordering dei pacchetti, evitare packet-loss).
- decidere quando lo stato interno e il traffico debbano essere migrati da una istanza di VNF sorgente a una destinazione (ad esempio per motivi di sovraccarico su una o più istanze);

- decidere quale sottoinsieme di tutto lo stato interno di una istanza di VNF debba essere migrato;
- stabilire quali siano le istanze sorgente e destinazione durante la migrazione.

Le applicazioni di controllo, attraverso le northbound API, comunicano con il controller OpenNF che ha il compito, quando richiesto, di effettuare la migrazione di una NF garantendo determinate proprietà quali, per esempio, evitare il reordering dei pacchetti o evitare il packet loss.

Un'altra componente dell'architettura OpenNF è il controller che è composto da un gestore dello stato delle funzioni di rete (*NF State Manager* in Figura 3.2) e da un gestore del traffico di rete (*Flow Manager* in Figura 3.2). Il primo gestore si occupa, attraverso le southbound API, di importare e/o esportare stato da/nella network function mentre il secondo gestore si occupa di reinstradare il traffico attraverso il paradigma SDN sugli switch SDN interessati durante la migrazione.

Northbound API

```
move(srcInst, dstInst, filter, scope, properties)
copy(srcInst, dstInst, filter, scope)
share(list<inst>, filter, scope, consistency)
```

Listing 3.1. Pseudo codice delle northbound API di OpenNF

Come mostrato nel listato 3.1 le northbound API esportate dal controller verso le applicazioni sono rappresentate da tre funzioni principali: *move*, *copy* e *share*. Le northbound API permettono alle applicazioni di controllo di migrare, copiare o condividere stato tra diverse istanze della stessa NF mediante comunicazione al controller OpenNF. Nel primo caso, le applicazioni di controllo hanno la possibilità di specificare e richiedere al controller il rispetto di determinate proprietà durante il processo di migrazione (ad esempio nessuna perdita di pacchetti e nessun reordering dei pacchetti). Sarà compito del controller OpenNF tradurre questi comandi provenienti dalle applicazioni di controllo in operazioni da effettuare attraverso il gestore dello stato delle istanze di funzioni di rete e il gestore del traffico di rete.

Nella presente tesi, visti gli obiettivi dichiarati in precedenza, verrà presa in esame esclusivamente la primitiva *move*, che permette di migrare lo stato tra diverse istanze della NF. I parametri della primitiva *move* sono:

- **srcInst**: specifica l'istanza sorgente della NF che si vuole migrare;
- **dstInst**: specifica l'istanza destinazione della NF su cui si vuole migrare la NF;
- **filter**: specifica un filtro per definire quale porzione di traffico si vuole migrare (ovvero quali flussi);
- **scope**: specifica quali tipologie di stato si vuole migrare (ovvero per-flow e/o multi-flow);

- **properties**: specifica quali proprietà debba garantire la migrazione (ovvero mancanza di packet-loss, di reordering);

Southbound API

```
multimap<flowid,chunk> getPerflow (filter)
void putPerflow(multimap<flowid,chunk>)
void delPerflow(list<flowid>)
multimap<flowid,chunk> getMultiflow(filter)
void putMultiflow(multimap<flowid, chunk>)
void delMultiflow(list<flowid>)
list<chunk> getAllflows()
void putAllflows(list<chunk>)
void enableEvents(filter, action)
void disableEvents(filter)
```

Listing 3.2. Pseudo codice delle southbound API di OpenNF

Come mostrato nel listato 3.2 le southbound API sono rappresentate da tre semplici funzioni principali declinate in più modi (*get*, *put* e *delete*) e da due funzioni *enableEvents* e *disableEvents*. Esse sono adoperate sulle istanze delle funzioni di rete dal gestore dello stato delle istanze presente nel controller OpenNF.

Le funzioni con prefisso *get* sono usate per esportare dello stato, quelle con prefisso *put* per importarlo, mentre quelle con prefisso *del* per cancellarlo.

Per comprendere meglio il loro ruolo, è dapprima necessario chiarire il significato di *filtro*, *flowid* e *chunk*. Un *filtro* è un insieme di coppie chiave-valore che specificano i valori che determinati campi dell'header del pacchetto in esame devono avere (ad esempio indirizzo IP sorgente/destinazione) affinchè esso venga selezionato dal filtro. I suddetti filtri sono simili ai criteri di matching in Openflow [6]. Un *flowid* è un identificativo univoco del flusso in esame. Un *chunk*, invece, consiste di una o più strutture dati interne alla NF che rappresentano lo stato associato allo stesso flusso, o allo stesso insieme di flussi. Grazie ai filtri è possibile definire il set di flussi il cui stato è da recuperare (*get*), creare/aggiornare (*put*) o cancellare (*delete*). Si osservi che per le funzioni *getAllflows* e *putAllFlows* non vi è il bisogno di specificare un filtro, dal momento che esse si riferiscono a stato che afferisce a tutti i flussi. In maniera simile, non vi è una funzione *delAllflows*.

Quando le funzioni *getPerflow* o *getMultiflow* vengono chiamate, la NF ha il compito di identificare e fornire al controller tutto lo stato per-flow o multi-flow che afferisce a quei flussi che corrispondono al filtro selezionato. Ad esempio, in un firewall, potrebbe esserci un contatore di pacchetti che si riferiscono ad un determinato flusso TCP identificato dalla quintupla TCP (stato *per-flow*) e un contatore di pacchetti che si riferisce a tutti quei flussi TCP con stesso indirizzo IP sorgente (stato *multi-flow*). Lo stato viene rappresentato come una mappa che ha come chiave il *flowid* del flusso e come valore il *chunk* relativo al *flowid*. Analogamente, quando sono chiamate le funzioni *putPerflow* o *putMultiflow* la NF è responsabile di sovrascrivere o combinare lo stato eventualmente

preesistente per un determinato flusso (o insieme di flussi) con lo stato fornito sottoforma di mappa (identica per forma alla suddetta) attraverso l'invocazione della funzione stessa. Le funzioni *delPerflow* o *delMultiflow*, invece, hanno il compito di rimuovere lo stato eventualmente preesistente nella NF, per un determinato flusso (o insieme di flussi) specificato dai *flowid* forniti.

Bisogna sottolineare che il design delle API appena esaminate è stato pensato in modo tale da evitare il bisogno, da parte dell'applicazione di controllo, di essere consapevole della maniera con la quale la NF organizza al suo interno lo stato di cui necessita.

Infine, OpenNF usa un meccanismo denominato *evento* per osservare dall'esterno eventuali aggiornamenti allo stato operati dalla NF.

Un *evento* è un pacchetto creato ad-hoc e mandato da una istanza di NF al controller OpenNF che contiene:

1. il pacchetto che ha provocato la generazione dell'evento stesso;
2. alcuni metadati (ad esempio il filtro, la NF che l'ha generato).

Nello specifico, il controller ha a disposizione le funzioni *enableEvents* e *disableEvents*:

- *enableEvents* indica alla istanza di NF su cui è chiamata di mandare degli *eventi* al controller nel caso in cui si ricevano pacchetti che soddisfano il filtro specificato (*filter*) e indica quale comportamento (*action*) essa debba adottare in questo caso;
- *disableEvents* permette di disabilitare i comportamenti specificati in precedenza grazie alla funzione *enableEvents*.

I tipi di *action* disponibili sono:

- *process*: indica alla istanza di processare i pacchetti;
- *buffer*: indica alla istanza di accodare i pacchetti e non processarli fino a nuovo ordine;
- *drop*: indica alla istanza di scartare i pacchetti.

Queste ultime due primitive, come vedremo successivamente, saranno fondamentali nel garantire le proprietà di mancanza di packet-loss e di mancanza di reordering durante la migrazione della NF.

L'operazione *move*

L'operazione move consente di migrare sia lo stato sia il traffico per un insieme di flussi da una istanza sorgente di NF a una istanza destinazione, garantendo eventualmente determinate proprietà di migrazione spiegate in precedenza.

La topologia di riferimento è mostrata in Figura 3.3: con il termine *sw* si intende l'ultimo switch della rete SDN in comune tra l'istanza sorgente e l'istanza destinazione della NF che si vuole migrare (denominate *srcInst* e *dstInst* in Figura 3.3). Si sottolinea che la topologia ipotizzata da OpenNF presume una rete SDN a supporto e presume anche che il controller OpenNF tenga traccia di quale sia l'ultimo switch in comune tra le due

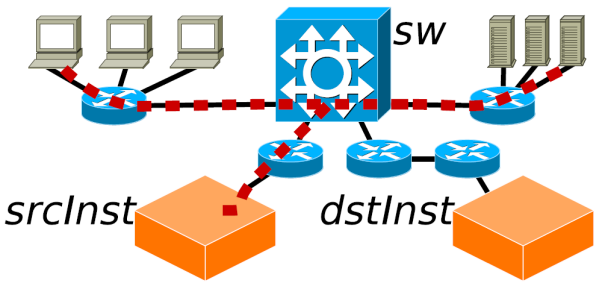


Figura 3.3. La topologia ipotizzata da OpenNF durante la migrazione (proveniente da [17]).

istanze. Il lavoro di OpenNF, inoltre, assume che la perdita di pacchetti e il reordering degli stessi non avvenga nel percorso tra *sw* e *srcInst*.

OpenNF prevede la possibilità di avere tre tipologie di *move* in base alle proprietà di migrazione che si vuole garantire (ovvero evitare perdita di pacchetti e/o evitare reordering dei pacchetti). Dati gli obiettivi di questa tesi, prenderemo in esame solamente la tipologia di *move* che permette di effettuare una migrazione evitando sia la perdita di pacchetti che il reordering.

```
1  eventReceivedFromSrcInst (event)
2      if shouldBufferEvents then
3          eventQueue.enqueue (event.packet)
4      else
5          sw.forward (event.packet, dstInst)
6  packetReceivedFromSw (packet)
7      if lastPacketFromSw== null then
8          signal (GOT_FIRST_PKT_FROM_SW)   // wait @ 24
9      lastPacketFromSw ← packet
10 eventReceivedFromDstInst (event)
11     if event.packet == lastPacketFromSw then
12         signal (DST_PROCESSED_LAST_PKT)  // wait @ 26
13 moveLossfreeOrderpreserve (srcInst, dstInst, filter)
14     shouldBufferEvents ← true
15     srcInst.enableEvents (filter, DROP)
16     chunks ← srcInst.getPerflow (filter)
17     srcInst.delPerflow (chunks.keys)
18     dstInst.putPerflow (chunks)
19     foreach event in eventQueue do
20         sw.forward (event.packet, dstInst)
21     shouldBufferEvents ← false
22     dstInst.enableEvents (filter, BUFFER)
23     sw.install (filter, {srcInst, ctrl}, LOW_PRIORITY)
24     wait (GOT_FIRST_PKT_FROM_SW)
25     sw.install (filter, dstInst, HIGH_PRIORITY)
26     wait (DST_PROCESSED_LAST_PKT)
27     dstInst.disableEvents (filter)
```

Figura 3.4. Pseudocodice dell'operazione di *move* (proveniente da [17]).

Le azioni che il controller OpenNF effettua quando una applicazione di controllo lo istruisce affinchè effettui una migrazione con le suddette proprietà sono riassunte in Fig. 3.4 e sono le seguenti:

1. **enableEvents**(*filter*,drop) su *SrcInst*: abilita l'invio di eventi al controller da parte di *SrcInst* che scarterà i pacchetti selezionati grazie al filtro specificato;

2. **getPerflow**(*filter*) su *SrcInst*: il controller recupererà da *SrcInst* qual è lo stato relativo al traffico di rete che si vuol migrare;

3. **putPerflow** su *DstInst*: il controller copierà su *DstInst* lo stato relativo al traffico di rete che si vuol migrare, ottenuto col passo precedente;

4. Il controller accoderà i pacchetti ricevuti grazie agli eventi abilitati al passo 1 finchè la *putPerflow* del passo 3 non terminerà;

5. quando la *putPerflow* del passo 3 terminerà, i pacchetti accodati al punto precedente sul controller verranno inoltrati a dstInst mediante lo switch con un particolare flag denominato *do-not-buffer* che indicherà alla istanza destinazione di non accodarli;

6. **enableEvents**(*filter*,buffer) su *DstInst*: abilita l'invio di eventi al controller da parte di *DstInst* che accoderà i pacchetti selezionati grazie al filtro specificato (tranne quelli ricevuti col particolare flag *do-not-buffer*);

7. ha luogo l'aggiornamento delle flow rules su *sw*:

   (a) si cala una prima regola in modo tale da far sì che sw invii i pacchetti relativi al filtro specificato sia a *SrcInst* sia al controller: il controller in questo modo conoscerà qual è l'ultimo pacchetto relativo al filtro specificato ricevuto da *SrcInst*;

   (b) si cala una seconda regola con priorità più alta rispetto alla suddetta in modo tale da inviare i pacchetti relativi al filtro specificato solo a *DstInst*

8. dato che i pacchetti arrivati a *SrcInst* devono essere processati da *DstInst* prima di quelli arrivati a DstInst stesso (per garantire il non riordino dei pacchetti), il controller aspetta di ricevere l'evento da *SrcInst* relativo all'ultimo pacchetto ricevuto da *SrcInst* stesso;

9. una volta ricevuto l'evento relativo all'ultimo pacchetto ricevuto da *SrcInst*, il controller invia il pacchetto a *DstInst* e aspetta di ricevere l'evento da *DstInst* relativo a questo stesso pacchetto;

10. quando ha ricevuto l'evento di cui al punto 9 da *DstInst*, il controller disabilita gli eventi su *DstInst* mediante la primitiva **disableEvents**(*filter*) in modo tale da rilasciare i pacchetti mandati in precedenza dallo switch a *dstInst* e accodati da quest'ultimo.

Il bisogno di garantire la proprietà di preservare l'ordine dei pacchetti in arrivo implica un costo a livello di performance: viene infatti aggiunta latenza ai pacchetti accodati dovuta al fatto che per processare i pacchetti in arrivo su *DstInst* bisogna aspettare che vengano processati prima i pacchetti in arrivo su *SrcInst*. Per cercare di attenuare questo problema, OpenNF opera due ottimizzazioni fondamentali:

- permette l'esecuzione delle primitive *getPerflow* e *putPerflow* in parallelo;
- adopera una strategia combinata di *early-release* (o rilascio anticipato dei buffer) e di *late-locking* (redirezione ritardata dei pacchetti): la prima consiste nel non aspettare che il trasferimento dell'intero stato in esame sia completato e nel mandare i pacchetti a *DstInst* solo se lo stato corrispondente a quei pacchetti è stato già mandato e ricevuto da *DstInst*, mentre la seconda consiste nel redirezionare i pacchetti che arrivano a *SrcInst* solo se lo stato associato è già stato mandato al controller.

Si noti che queste due ottimizzazioni garantiscono però l'ordine dei pacchetti solo all'interno dei flussi e non tra i flussi stessi.

Vantaggi e svantaggi

I vantaggi principali di OpenNF sono i seguenti:

- effettua il trasferimento delle sole strutture dati relative allo stato, in contrasto con l'approccio di migrazione VM-like: questo apporta vantaggi in termini di risparmio di tempo di migrazione e di risorse di rete (cioè implica un ridotto consumo della banda disponibile);
- opera una distinzione netta tra stato *per-flow*, *multi-flow* e *all-flow*, permettendo di avere un controllo sulla migrazione ad un livello di granularità fine;
- permette di effettuare la migrazione di stato specificando le proprietà che si vuole garantire (mancanza di packet loss, mancanza di reordering, entrambe o nessuna).

Gli svantaggi, invece, sono:

- nonostante la distinzione tra stato per-flow, multi-flow e all-flow, gli autori di OpenNF affermano di voler proporre una soluzione per migrare correttamente lo stato condiviso tra più flussi ma non dettagliano le operazioni necessarie;
- l'introduzione di latenze aggiuntive sul traffico in migrazione;
- la scarsa scalabilità e il possibile collo di bottiglia in termini di risorse computazionali rappresentati dal controller e un sovraccarico della rete di controllo, dovuti al fatto che sia lo stato sia il traffico passano dal controller per poter migrare (essi vengono di fatto triangolati dall'istanza sorgente al controller e dal controller all'istanza destinazione);
- la potenziale problematica del buffer overflow relativa al buffering di pacchetti sul controller, che potrebbe provocare conseguenti problemi di perdita di pacchetti, di perdita di cambiamenti e di inconsistenze di stato sulla NF;
- l'esigenza di dover modificare la NF, seppur in maniera marginale, per poter permettere l'importazione e l'esportazione di stato e l'abilitazione e disabilitazione di eventi;

- la potenziale creazione di nuovo stato sull'istanza sorgente durante la migrazione, il che implica un prolungamento potenzialmente infinito del tempo di migrazione.

A fronte di questi limiti, la soluzione proposta nella tesi si propone di superarli, in particolare:

- chiarendo in dettaglio come gestire la migrazione non solo dello stato di tipo *per-flow*, ma anche di quello *multi-flow*e *all-flow*;
- riducendo la complessità e il carico di lavoro del controller centrale durante la migrazione, in particolare:
  - evitando la triangolazione sia dei pacchetti sia dello stato tra le istanze di VNF e il controller, riducendo il più possibile le latenze aggiuntive che questa comporta sul traffico migrante;
  - evitando il buffering dei pacchetti sul controller stesso, che ora non rappresenta più un possibile collo di bottiglia;
- gestendo la potenziale creazione di nuovo stato sull'istanza Src durante la migrazione, garantendo in questo modo un tempo di migrazione non infinito.

3.2.2 DiST

DiST (abbreviazione di Distributed State Transfer) [18] si propone come estensione del precedente lavoro di OpenNF e si prefigge come obiettivo quello di migliorare i punti deboli della sua architettura. Per raggiungere questo scopo DiST adopera i seguenti accorgimenti:

- usa la rete di controllo solo per esigenze di segnalazione: il trasferimento di stato e la redirezione dei pacchetti è eseguita direttamente tra le due istanze sorgente e destinazione della VNF in maniera P2P-like, sfruttando i link del dataplane;
- il controller viene escluso dal percorso critico durante la migrazione: esso gestisce esclusivamente i messaggi di controllo da/per le istanze sorgente e destinazione della VNF e per gli switch interessati durante la migrazione.

DiST, però, a differenza di OpenNF, considera solo il caso in cui si voglia migrare stato di tipo per-flow.

La *move* di OpenNF modificata da DiST

Come in OpenNF, l'operazione di migrazione di DiST può garantire sia che i pacchetti redirezionati dalla istanza sorgente della VNF siano processati prima di quelli inoltrati dallo switch all'istanza destinazione (proprietà di non riordino dei pacchetti), sia che non vi sia perdita di pacchetti durante il processo.

La sincronizzazione tra i due flussi di pacchetti provenienti dallo switch e diretti alle istanze sorgente e destinazione avviene usando un pacchetto speciale chiamato "Inband

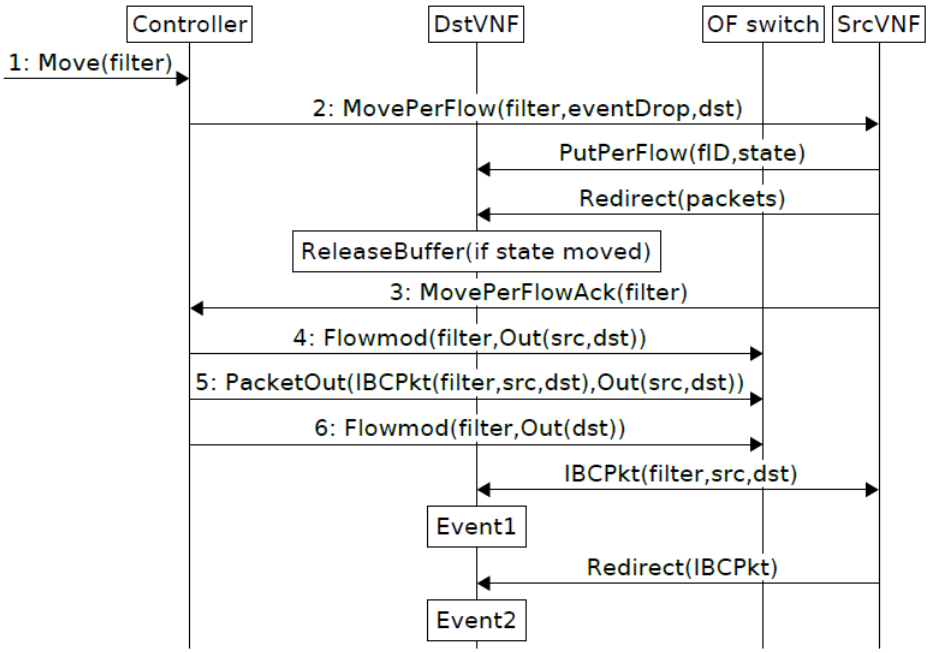


Figura 3.5. Il diagramma di sequenza della migrazione DiST (proveniente da [18]).

control packet" (abbreviato in *IBCPkt*) che deve essere creato per ogni filtro specificato per la migrazione. Il filtro di questo pacchetto, inoltre, deve corrispondere alla definizione di filtro in entrambe le istanze e deve essere distinguibile dai normali pacchetti di dati.

Le azioni eseguite per effettuare una migrazione di una porzione di traffico e del relativo stato con le proprietà di mancanza di packet-loss e mancanza di reordering sono le seguenti (riassunte nel diagramma di sequenza in Fig. 3.5):

1. l'istanza sorgente trasferisce in maniera P2P-like alla istanza destinazione lo stato relativo a quei flussi che sono selezionati grazie al filtro specificato (ovvero flussi migranti);

2. durante il trasferimento dello stato l'istanza sorgente redireziona verso l'istanza destinazione i pacchetti relativi ai flussi migranti;

3. l'istanza destinazione accoda tutti i pacchetti ricevuti dall'istanza sorgente;

4. quando il trasferimento di cui al punto 1 termina, l'istanza destinazione rilascia i pacchetti accodati in precedenza e inizia a processarli;

5. viene duplicato il traffico relativo ai flussi migranti, mediante una `flowmod` sull'ultimo switch in comune tra le due istanze;

6. viene mandato in contemporanea, mediante una operazione di *packet-out* [6], un pacchetto *IBCPkt* ad entrambe le istanze sorgente e destinazione;

7. viene inoltrato il traffico relativo ai flussi migranti solo ed esclusivamente alla istanza destinazione;
8. a questo punto:
   (a) l'istanza sorgente continua a redirezionare verso l'istanza destinazione i pacchetti relativi ai flussi migranti fino a che essa non riceve il pacchetto *IBCPkt* dallo switch (una volta ricevuto essa scarterà i pacchetti dei flussi migranti);
   (b) l'istanza destinazione processa tutti i pacchetti provenienti dall'istanza sorgente fino a che non riceve il pacchetto *IBCPkt* dall'istanza sorgente stessa e
      i. scarta tutti i pacchetti che provengono dallo switch **prima** di ricevere il pacchetto *IBCPkt* proveniente dallo switch stesso;
      ii. accoda tutti i pacchetti che provengono dallo switch ricevuti **dopo** il pacchetto *IBCPkt* proveniente dallo switch stesso.
9. la ricezione da parte dell'istanza destinazione del pacchetto *IBCPkt* dall'istanza sorgente rappresenta la fine del processamento di tutti i pacchetti provenienti da quell'istanza e ciò implica che l'istanza destinazione può iniziare a processare i pacchetti accodati al punto 8b (in questo modo, grazie anche alla duplicazione del traffico di cui al punto 5, si garantiscono le proprietà di non riordino dei pacchetti e di mancanza di perdita di pacchetti).

Vantaggi e svantaggi

I vantaggi principali di DiST sono i seguenti:

- in maniera simile a OpenNF:
  - effettua il trasferimento delle sole strutture dati relative allo stato, in contrasto con l'approccio di migrazione VM-like: questo apporta vantaggi in termini di risparmio di tempo di migrazione e di risorse di rete (ovvero implica un ridotto consumo della banda disponibile);
  - permette di effettuare la migrazione di stato specificando le proprietà che si vuole garantire (mancanza di packet loss, mancanza di reordering, entrambe o nessuna).
- la rete di controllo non è più sovraccaricata durante la migrazione: non vi è più triangolazione per migrare stato e pacchetti, infatti il controller è ora responsabile solo delle operazioni di controllo durante la migrazione (*signalling*); in questo modo il costo delle operazioni è spostato dal controller alle VNF.

Gli svantaggi, invece, sono:

- in maniera simile a OpenNF:
  - l'introduzione di latenze aggiuntive sul traffico in migrazione;

  - la potenziale problematica del buffer overflow relativa al buffering di pacchetti sull'istanza destinazione, che potrebbe provocare conseguenti problemi di perdita di pacchetti, di perdita di cambiamenti e di inconsistenze di stato sulla NF;
  - l'esigenza di dover modificare la NF, seppur in maniera marginale, per poter permettere l'importazione e l'esportazione di stato e il riconoscimento dei pacchetti IBCPkt con relativo cambiamento di comportamento;
  - la potenziale creazione di nuovo stato sull'istanza sorgente durante la migrazione, il che implica un prolungamento potenzialmente infinito del tempo di migrazione;

- permette la migrazione di solo stato per-flow;
- la scarsa scalabilità e il possibile collo di bottiglia in termini di risorse computazionali rappresentati dall'istanza destinazione che deve accodare determinati pacchetti.

A fronte di questi limiti, la soluzione proposta nella tesi si propone di superarli, in particolare:

- chiarendo in dettaglio come gestire la migrazione non solo dello stato per-flow, ma anche multi-flow e all-flow;
- cercando di ridurre il più possibile le problematiche relative al buffer overflow e alle latenze aggiuntive sul traffico migrante, mediante operazioni di buffering più intelligenti che hanno una durata il più breve possibile;
- gestendo la potenziale creazione di nuovo stato sull'istanza Src durante la migrazione, garantendo in questo modo un tempo di migrazione non infinito.

3.2.3 Ulteriori soluzioni di tipo stateful

In [19] si suggerisce un miglioramento per le soluzioni di migrazione presentate pocanzi: in sostanza si permette all'istanza sorgente della NF di continuare a processare pacchetti di flussi migranti mentre il trasferimento di stato relativo è in corso. Nel caso in cui un pacchetto ricevuto dall'istanza sorgente provochi una modifica dello stato in corso di migrazione, una copia del pacchetto in esame viene mandata alla istanza destinazione dove viene riprocessato (avendo cura però di impedire qualsiasi output dovuto di fatto a questo reprocessing(ovvero logs, pacchetti)) per far sì che lo stato si aggiorni anche sull'istanza destinazione e rimanga consistente con la versione dell'istanza sorgente. Questa tecnica permette inoltre di ridurre l'overhead di latenza sui pacchetti. Il problema in [19] è che non viene considerata la proprietà di mancanza di riordino dei pacchetti, ma viene garantita solo la proprietà di mancanza di perdita di pacchetti.

[20], basandosi su OpenNF, mira a ridurre l'overhead di latenza e il tempo di migrazione. L'idea è quella di distinguere tra i cosiddetti flussi *mice* e i cosiddetti flussi *elephant*. I primi sono quei flussi che, secondo [20], sfruttano una porzione di stato con un'alta probabilità di scadere prima del termine della migrazione, mentre i secondi sono

quei flussi che hanno una probabilità più alta di perdurare nel tempo e il cui stato dovrebbe pertanto essere migrato sull'istanza destinazione e mantenuto nel tempo. La porzione di stato dei flussi *mice* viene mantenuta solo ed esclusivamente sull'istanza sorgente della VNF, riducendo in questo modo la porzione di stato da migrare e di conseguenza i tempi di migrazione e migliorando complessivamente le performance. Il problema di questo approccio è principalmente la difficoltà nel distinguere tra flussi *elephant* e flussi *mice*, la notevole complessità introdotta nell'architettura per raggiungere questo scopo e le possibili problematiche dovute a un potenziale errore nella distinzione tra i due tipi di flussi.

[21] pone invece l'attenzione sulla questione dell'alta disponibilità che la NF deve garantire nel tempo e sulla sua tolleranza ai guasti. Esso adopera una politica di tipo hot-standby suddividendo l'intero stato della NF in piccoli sottoinsiemi chiamati *replication groups*, ognuno dei quali ha una sua frequenza di checkpoint (ovvero la frequenza con la quale verrà effettuato il backup del suddetto sottoinsieme) e una istanza di destinazione sulla quale verrà effettuato il backup. Il punto di forza di questo approccio è la possibilità di avere un overhead di latenza più basso dovuto al fatto che le frequenze di checkpoint possono essere anche molto elevate (finanche 1000 Hz) poichè lo stato considerato è solo una minima parte di tutto lo stato della NF.

3.3 Le soluzioni stateless

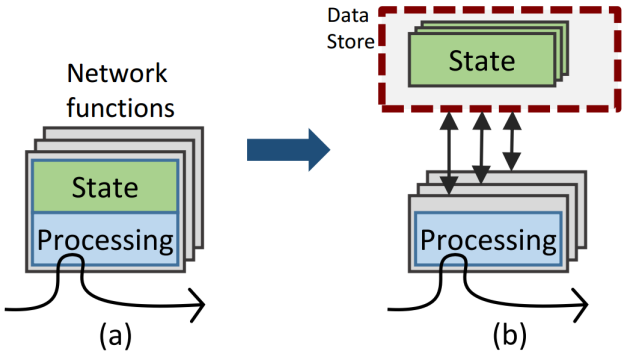


Figura 3.6. La transizione da soluzione stateful(a) a soluzione stateless(b) (proveniente da [22]).

[22–24] propongono una prospettiva diversa ovvero quella del disaccoppiamento tra la NF e il relativo stato di cui essa ha bisogno: quest'ultimo non è più residente all'interno della NF stessa bensì è su un datastore resiliente e che garantisce accessi ai dati a bassa latenza (ad esempio RAMCloud [25] col supporto di una rete Infiniband [26]). L'attenzione è posta stavolta però sullo stato condiviso tra diverse istanze della NF, ma non tra flussi di rete diversi. Questi approcci si basano sul fatto di dover interpellare il datastore remoto per creare, modificare e/o accedere allo stato necessario alla NF per processare ogni pacchetto: questo implica maggiore latenza dovuta ai tempi di accodamento dei pacchetti stessi in attesa del recupero dello stato dal datastore remoto e problemi

di scalabilità del datastore stesso che deve essere potenzialmente in grado di gestire un numero molto alto di richieste provenienti da molteplici funzioni di rete. Per mitigare queste problematiche, alle funzioni di rete può essere aggiunto un caching layer nei casi in cui ci si può permettere la violazione della consistenza sullo stato. Per quanto riguarda i vantaggi, si annovera la possibilità di garantire alta disponibilità delle funzioni di rete, grazie a politiche di tipo hot-standby, anche di carattere speculativo, che permettono l'avvio di una nuova istanza della NF in tempi minimi.

Capitolo 4

L'algoritmo per la migrazione di una VNF

In questo capitolo viene presentato il nuovo algoritmo per la migrazione di una funzione di rete virtualizzata, che rappresenta il contributo principale di questo lavoro di tesi.

4.1 L'algoritmo

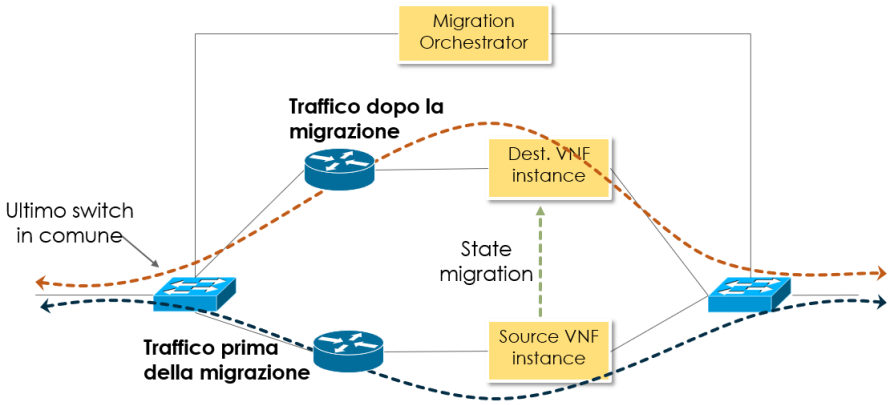


Figura 4.1. Lo scenario di migrazione di una VNF in una rete SDN.

Gli obiettivi principali dell'algoritmo che si sta per presentare sono:

- la migrazione di una VNF, che richiede lo spostamento tutto lo stato e tutto il traffico da una istanza di VNF sorgente (Src) ad una di destinazione (Dst) di cui si esegue dapprima il *deploy* appositamente e la cui implementazione potrebbe essere completamente diversa (ad esempio da una VNF Src in una VM ad una VNF Dst in Docker [27]);
- la corretta gestione di porzioni di stato che potrebbero cambiare sulla istanza sorgente durante la migrazione;

- la gestione della creazione di nuovo stato su Src: durante la migrazione di una VNF bisogna garantire che si possa creare eventualmente nuovo stato causato da traffico che prima non attraversava la VNF; ci sono diverse strategie per risolvere questo problema:
  - si può decidere di permettere la creazione di nuovo stato sulla istanza sorgente della VNF;
  - si può cercare di reinstradare la nuova porzione di traffico sulla nuova istanza di VNF.

  Nel primo caso potrebbe configurarsi un problema di tempistiche: infatti nel caso in cui si venga a creare nuovo stato in maniera continuativa sull'istanza migrante della VNF, si potrebbe incorrere in tempi di migrazione potenzialmente infiniti, dovuti al fatto che si deve effettuare la migrazione sia di nuovo stato sia di nuovo traffico. Nel secondo caso, invece, adotta verificandosi la creazione di nuovo stato direttamente sulla istanza destinazione della VNF, i problemi suddetti non si pongono. Nell'algoritmo che si sta per presentare è stata scelta quest'ultima soluzione.
- evitare l'inoltro di traffico tra le due istanze, escludendo, per le motivazioni esposte nel Capitolo 2, anche la possibilità di triangolazione mediante una entità terza;
- gestire la migrazione di tutti i tipi di stato esposti nel Capitolo 3 (per-flow, multi-flow, all-flow);
- la possibilità di migrare anche solo porzioni di stato per far sì che si possa migrare una porzione di traffico sull'istanza destinazione della NF che potrà subito processarlo;
- cercare di minimizzare il più possibile:
  - il downtime, inteso come lasso di tempo in cui il processing dei pacchetti viene sospeso;
  - il tempo di migrazione, inteso come tempo necessario per migrare tutto lo stato e tutto il traffico tra le due istanze affinché l'istanza destinazione possa processare tutto il traffico che prima attraversava l'istanza sorgente;
  - il tempo di accodamento dei pacchetti e quindi l'overhead di latenza che ne deriva;
- garantire la mancanza di packet-loss;
- garantire il non riordino dei pacchetti;
- garantire che il tempo di migrazione sia finito.

Inoltre, l'algoritmo presentato in questo capitolo ha il vincolo che le due istanze della VNF (Src e Dst) siano istanziate in una rete SDN.

La Figura 4.1 mostra lo scenario di migrazione di una VNF da una istanza che denominiamo istanza Src ad una istanza Dst. Come si può notare in Figura 4.1, l'algoritmo

presuppone l'esistenza di un migration orchestrator che ha il compito di far partire la migrazione, di gestirla e di assicurarsi la sua corretta esecuzione fino a che tutte le operazioni necessarie siano completate. Per svolgere le sue funzioni, l'orchestratore deve sapersi interfacciare non solo con le istanze della NF ma anche con gli switch della rete SDN. La descrizione dell'algoritmo che segue è stata fatta assumendo che la rete SDN sia controllata mediante il protocollo OpenFlow [6].

Partiremo con il considerare, per semplicità, la migrazione di flussi che afferiscono a stato di tipo *per-flow*. Successivamente considereremo anche gli altri tipi di stato (si veda appendice A).

4.1.1 Migrazione di flussi con solo stato per-flow

In questa sezione verrà illustrato l'algoritmo per la migrazione di flussi con solo stato di tipo per-flow. Prendendo spunto dagli approcci VM-like presentati nel Capitolo 3, l'algoritmo si compone fondamentalmente di due fasi:

1. fase di *pre-copy*: durante questa fase si effettua una copia di *tutto lo stato* da Src a Dest. Durante questa fase, inoltre, si cerca di redirezionare il traffico relativo a nuovi flussi direttamente sull'istanza Dst in modo tale da rendere finito il tempo di migrazione;

2. fase di *stop-and-copy*: durante questa fase si migrano i flussi e il relativo stato da Src a Dst ad una granularità fine (quindi per piccole porzioni di stato e di traffico).

Insieme, le due fasi permettono:

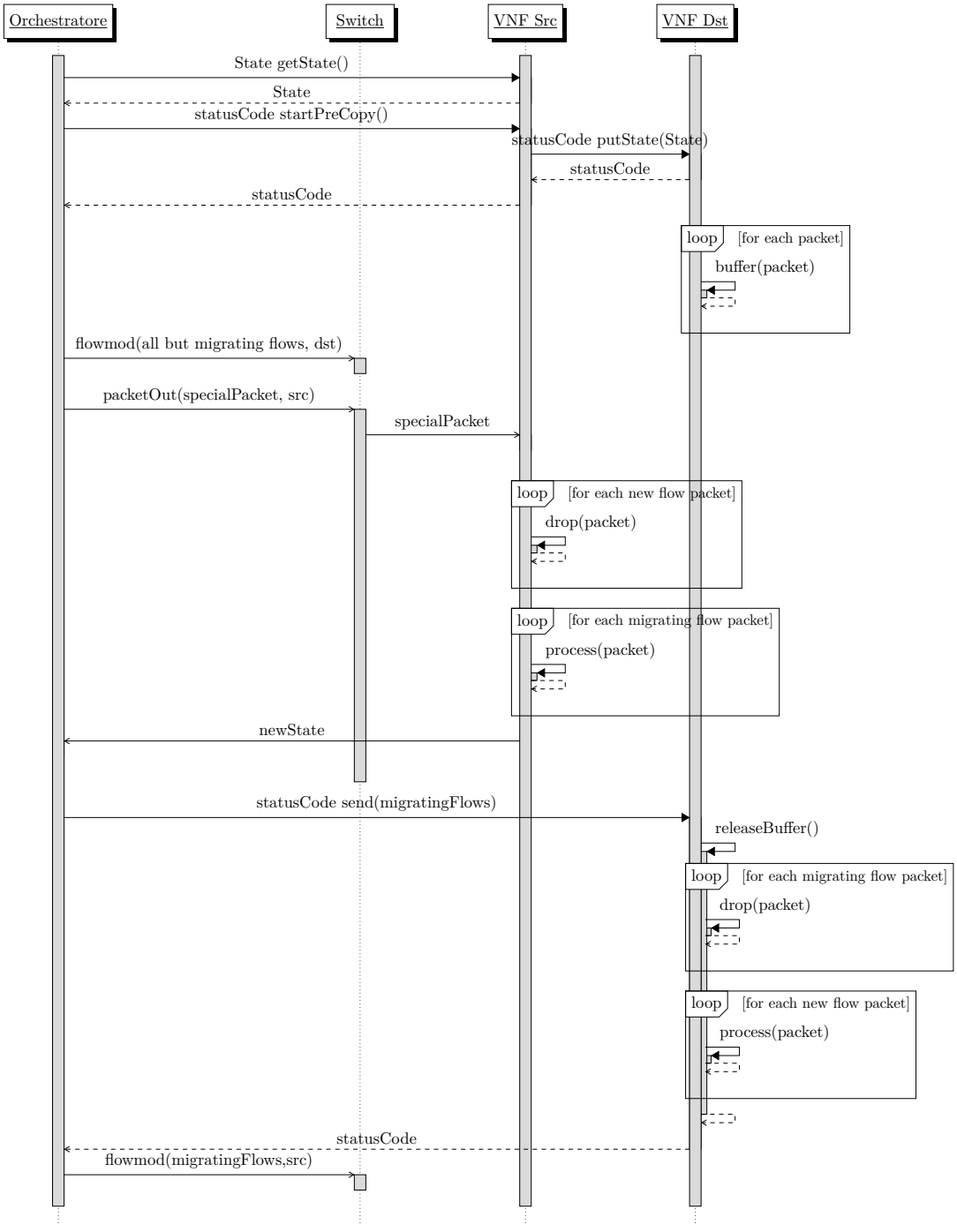
- di diminuire notevolmente il tempo di migrazione, dato che potrebbe esserci una porzione di stato già copiata grazie alla pre-copy che non è cambiata fino alla fase di stop-and-copy. Questo permetterebbe di poter anche anticipare il rilascio dei pacchetti accodati sull'istanza Dst in attesa dello stato necessario per il processing;

- di non interrompere il processing di *tutto* il traffico che attraversa Src ma di farlo a livello di piccole porzioni di traffico;

- di diminuire il *downtime* e di conseguenza le latenze introdotte sul traffico.

L'algoritmo e le sue principali funzioni sono riassunte nel diagramma di sequenza in Fig. 4.2 e nella Tabella 4.1 che seguono.

Figura 4.2. Diagramma di sequenza dell'algoritmo.



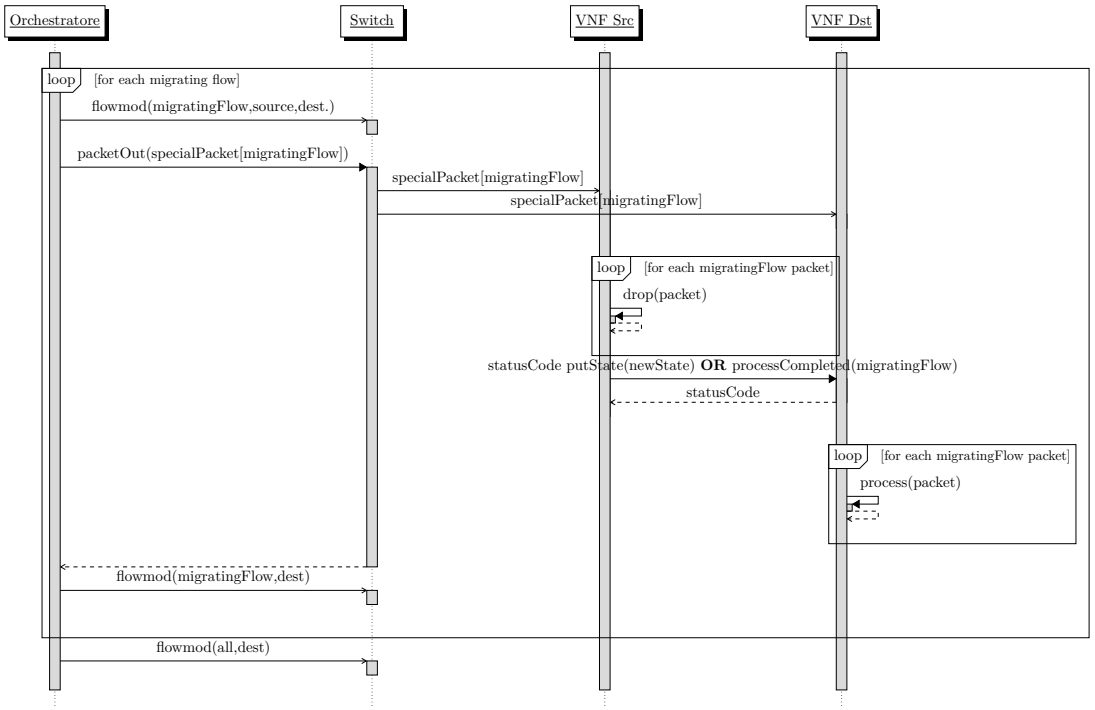


Tabella 4.1. Le principali chiamate dell'algoritmo.

Funzione	**Descrizione**
State getState()	Recupera lo stato presente sull'istanza su cui viene chiamata
statusCode startPreCopy(ist)	Provoca l'avvio della pre-copy dall'istanza su cui viene chiamata verso l'istanza ist
statusCode putState(State)	Copia lo stato State sull'istanza su cui viene chiamata
void buffer(packet)	Accoda packet
void process(packet)	Processa packet
void flowmod(filter,port1,[port2])	Cala le flow rules opportune per redirezionare il traffico individuato con filter verso le porte specificate
void packetOut(packet, port)	Provoca una operazione di packet-out di packet sulla porta port
statusCode send(migratingFlows)	Invia indicazioni circa i flussi migranti
statusCode processCompleted(migratingFlow)	Informa l'istanza su cui viene chiamata che il processing dei pacchetti relativi a migratingFlow è terminato

Per rendere l'algoritmo di più facile comprensione, prenderemo in considerazione solo le azioni eseguite dall'orchestratore sul primo switch da sinistra della Figura 4.1. Si tenga

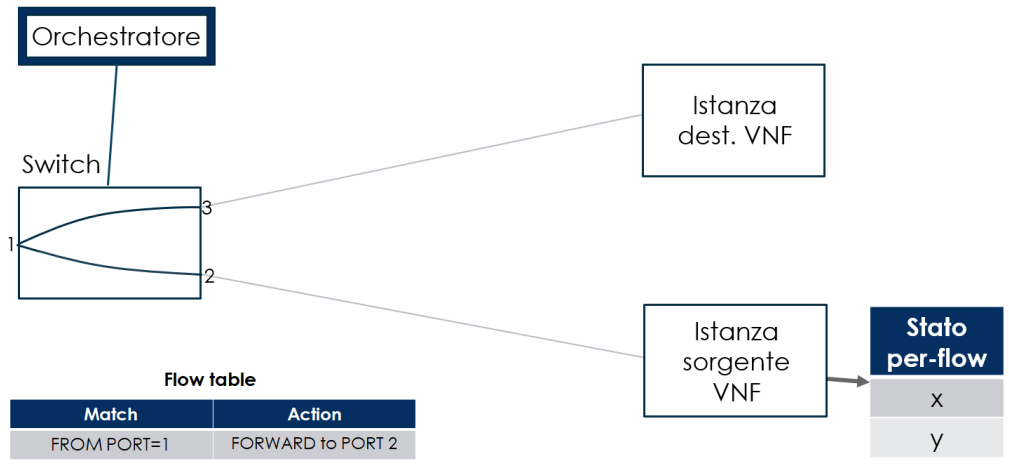


Figura 4.3. Un esempio di stato iniziale della rete prima della migrazione.

in considerazione che i messaggi `flowmod`[1] mandati dall'orchestratore sul primo switch verranno eseguite in maniera duale sul secondo switch per aver cura di gestire anche il traffico di ritorno.

Si consideri lo scenario in Fig. 4.3 in cui lo switch stia inviando il traffico verso l'istanza sorgente di VNF (d'ora in poi denominata Src) e che su di essa sia stato creato in precedenza lo stato per il processing di due flussi diversi, che per semplicità denomineremo con due lettere diverse (x e y). Ad esempio x ed y potrebbero essere due connessioni TCP identificate mediante la relativa quintupla TCP. Lo stato di x e y è di tipo dinamico e per-flow.

Le azioni eseguite per effettuare la migrazione di Src (ovvero di tutto il traffico gestito da essa e del relativo stato) verso l'istanza VNF di destinazione (d'ora in poi denominata Dst) con le proprietà di mancanza di packet-loss e mancanza di reordering sono le seguenti:

1. L'orchestratore (Fig. 4.4) chiede a Src quali sono i flussi per i quali la VNF Src ha dello stato, riconoscendo in questo modo quali sono i flussi che l'istanza stessa sta gestendo correntemente;

[1] per flowmod si intendono i messaggi del protocollo Openflow che permettono di modificare le regole nei dispositivi controllati mediante il suddetto protocollo

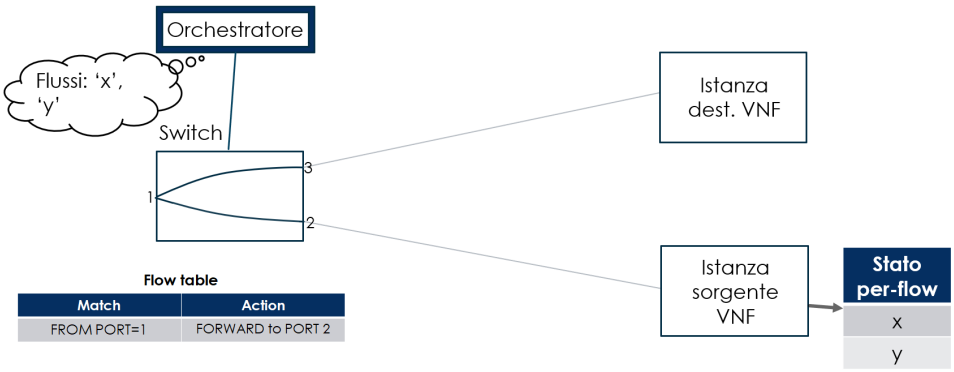


Figura 4.4. L'orchestratore acquisisce la conoscenza dei flussi in gestione sull'istanza sorgente.

2. L'orchestratore (Fig. 4.5) ordina a Src di effettuare una operazione di pre-copy (ovvero di copiare tutto il suo stato sull'istanza Dst);

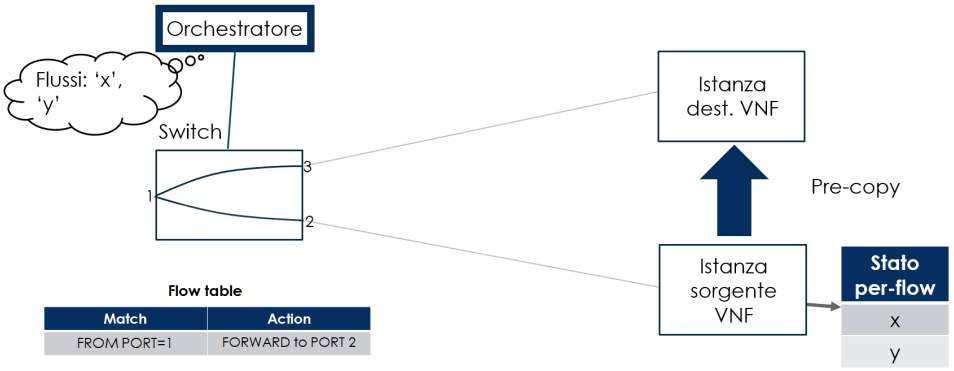


Figura 4.5. L'istanza sorgente inizia l'operazione di pre-copy.

3. Al termine della fase di pre-copy(Fig. 4.6), l'istanza Dst avrà una copia di tutto lo stato dell'istanza Src. La copia sull'istanza Dst potrebbe essere in futuro disallineata rispetto alla versione dello stato sull'istanza Src, dato che quest'ultima può modificare ancora lo stato a causa del processamento di ulteriori pacchetti;

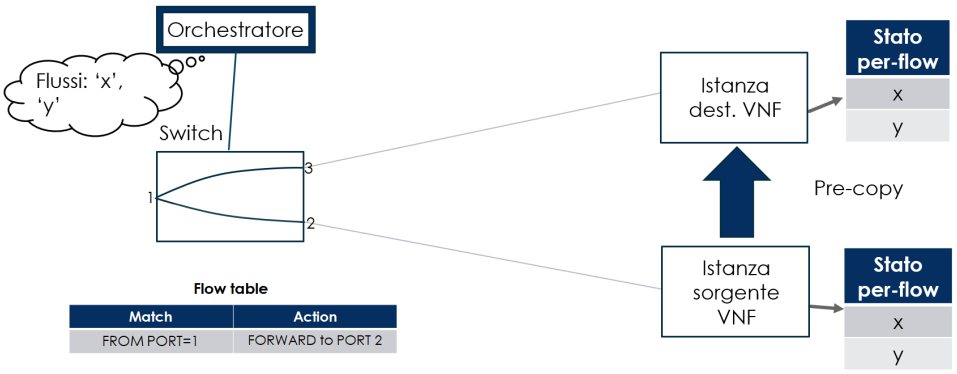


Figura 4.6. L'istanza destinazione al termine della fase di pre-copy.

4. L'istanza Dst, d'ora in poi, accoderà tutto il traffico ricevuto (Fig. 4.7). Si supponga ora che ci sia un pacchetto in volo verso l'istanza Src relativo ad un nuovo flusso che l'istanza Src non aveva mai gestito prima d'ora (ad esempio pacchetto z_0 del flusso z in Fig. 4.7). Se non gestiti in maniera adeguata, questi pacchetti relativi a nuovi flussi potrebbero causare il prolungamento indefinito del tempo di migrazione, dato che si dovrebbe in continuazione migrare i nuovi flussi e il nuovo stato da Src a Dst;

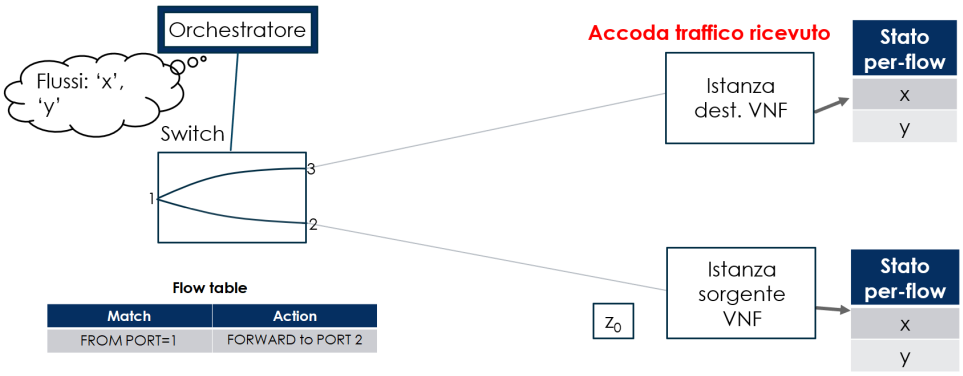


Figura 4.7. L'istanza destinazione accoda il traffico ricevuto.

5. L'orchestratore effettua le dovute `flowmod` sullo switch, in modo tale da inoltrare verso l'istanza Dst tutto il traffico, escluso quello relativo ai flussi appresi al punto 1 che sono già gestiti dall'istanza Src (ad esempio x,y in Fig. 4.8);

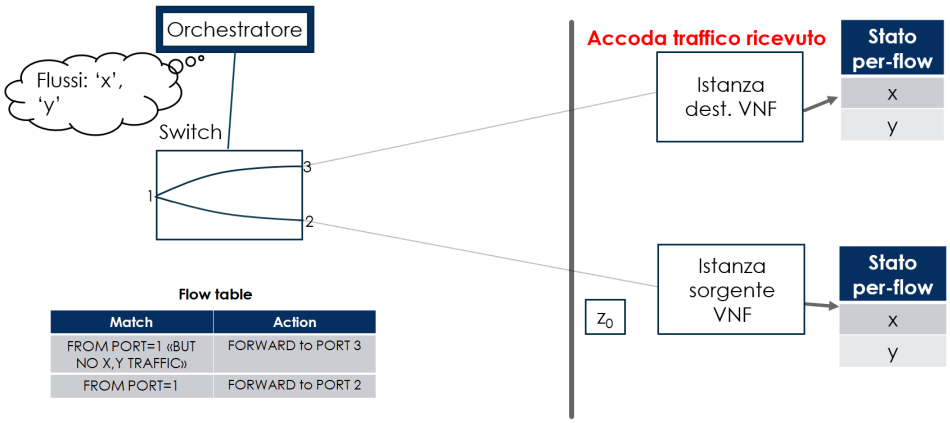


Figura 4.8. L'orchestratore manda i dovuti messaggi di `flowmod` sullo switch per modificare la flow table.

6. il traffico relativo ai nuovi flussi (ovvero quelli di cui l'orchestratore non è venuto ancora a conoscenza al passo 1, ad esempio flusso z in Fig. 4.9) viene ora inoltrato dallo switch sia all'istanza Src che all'istanza Dst;

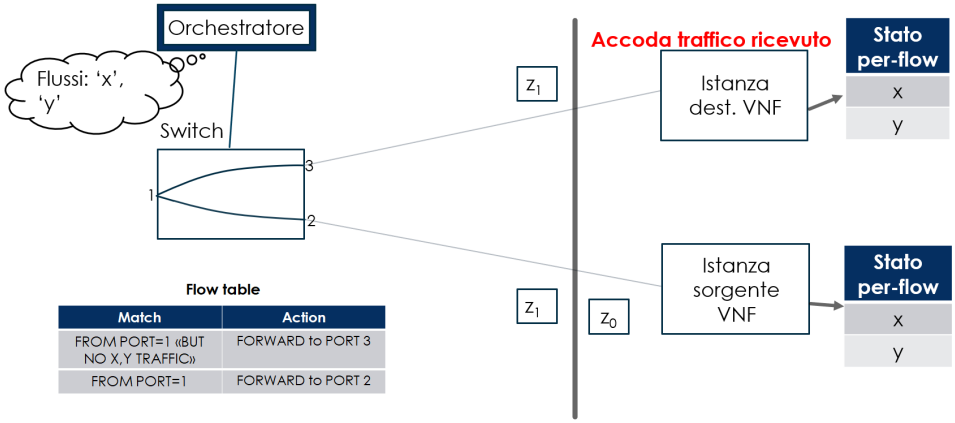


Figura 4.9. Il traffico dei nuovi flussi viene inoltrato alle due istanze di VNF.

7. l'orchestratore invia un pacchetto speciale (che chiameremo pacchetto *blocker*) all'istanza Src, mediante un'operazione di *packet-out* sullo switch (pacchetto blu in Fig. 4.10); questo pacchetto deve essere identificato in maniera univoca dall'istanza Src (ad esempio da uno specifico valore del campo Protocol Type dell'header IP [28]). Le sue funzioni verranno chiarite nei prossimi passi;

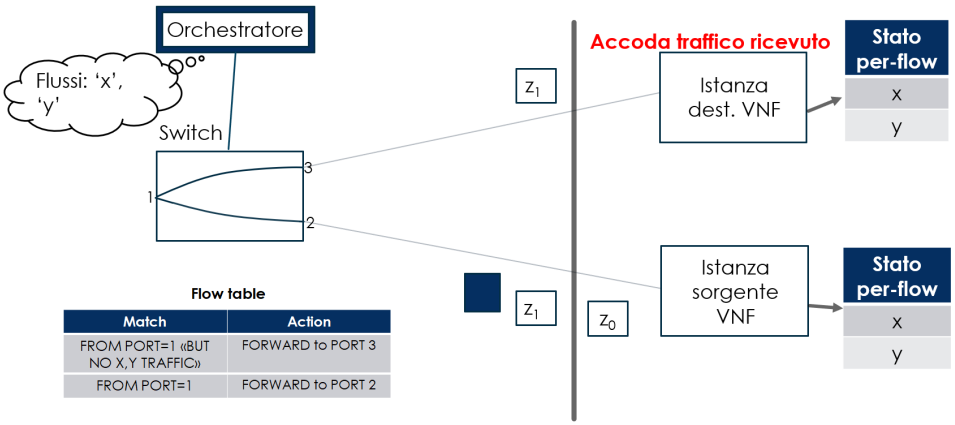


Figura 4.10. Stato della rete dopo l'operazione di packet-out del pacchetto *blocker*.

8. in Fig. 4.11 viene mostrata l'istanza sorgente che processerà tutti i pacchetti fino alla ricezione del suddetto pacchetto *blocker*: questo potrebbe causare la creazione di nuovo stato su di essa (ad esempio stato relativo al flusso z in Figura 4.11) e anche la modifica dello stato preesistente su di essa (ad esempio quello relativo al flusso x); si noti che pacchetti relativi ad un flusso di cui nessuna delle due istanze ha lo stato necessario per poter processare (ad esempio pacchetto w_0 del flusso w in Figura 4.11) sono ora in volo verso le due istanze di VNF;

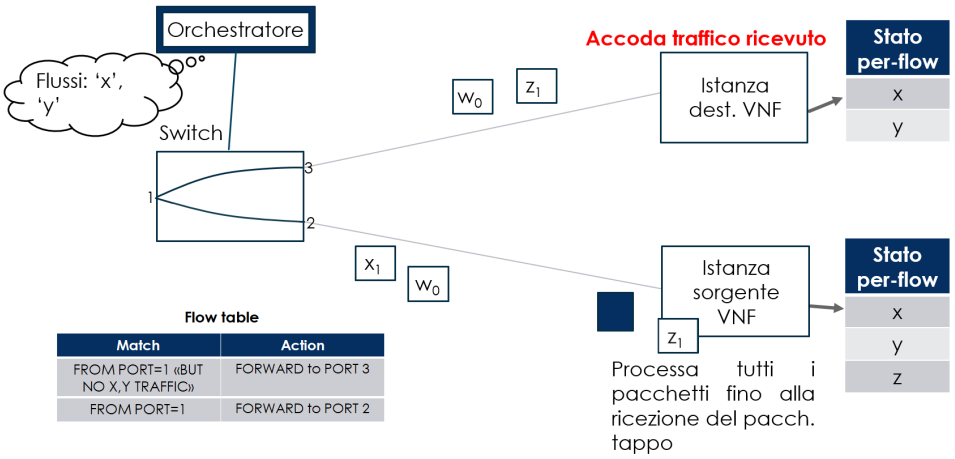


Figura 4.11. Possibile creazione di nuovo stato sull'istanza sorgente, prima della ricezione del pacchetto *blocker*.

9. l'istanza Src riceve il pacchetto *blocker* di cui al punto 7 (Fig. 4.12); esso ha le seguenti funzioni:

   - cambiare il comportamento dell'istanza Src. Essa infatti ora:

  - scarterà tutti i pacchetti che provocherebbero creazione di nuovo stato su di essa (ad esempio tutti i pacchetti che non appartengono ai flussi x, y e z in Fig. 4.12);
  - processerà tutti i pacchetti di cui ha già il relativo stato (ad esempio tutti i pacchetti che appartengono ai flussi x, y e z in Fig. 4.12);
- rendere noto all'orchestratore quali sono i flussi di cui l'istanza Src ha il relativo stato; quest'ultima ora informerà l'orchestratore circa i flussi di cui ha lo stato necessario per il processing (ad esempio x, y e z in Fig. 4.12). Si noti che l'orchestratore ora è a conoscenza del potenziale nuovo stato creatosi a causa dei pacchetti ricevuti dall'istanza Src dopo che l'orchestratore ha effettuato le operazioni di cui al punto 1 e prima della ricezione del pacchetto *blocker* in esame (ad esempio stato relativo al flusso z in Fig. 4.12);

Alla luce di quanto detto, è evidente come il pacchetto *blocker* permetta di garantire che la migrazione abbia un tempo finito: la creazione di nuovo stato dovuto al processing di traffico relativo a nuovi flussi, infatti, viene permessa ora solo sull'istanza Dst e non più sull'istanza Src.

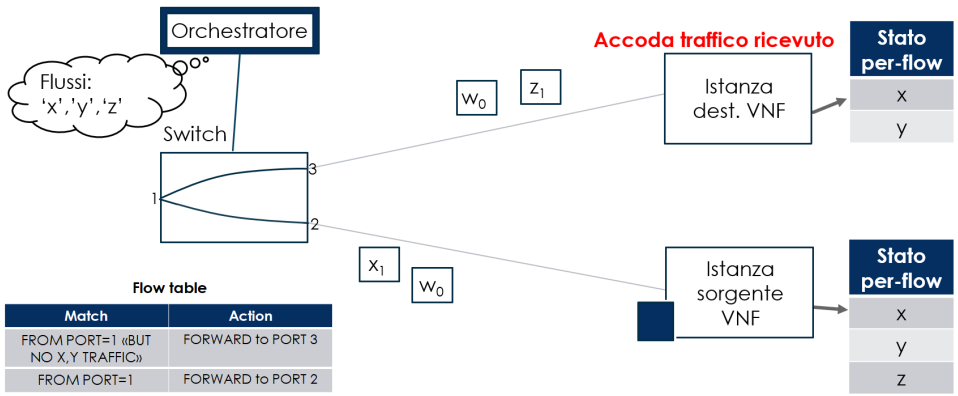


Figura 4.12. Ricezione del pacchetto *blocker* da parte dell'istanza sorgente della VNF.

10. l'orchestratore ora informerà l'istanza Dst riguardo i flussi i cui pacchetti sono processati dall'istanza Src (ovvero quei flussi di cui l'istanza Src ha lo stato necessario per il processing, ad esempio x, y e z); in questo modo l'istanza Dst potrà rilasciare i pacchetti accodati in precedenza e:

    - scarterà tutti i pacchetti relativi ai flussi di cui l'istanza Src ha già lo stato necessario per il processing (ad esempio pacchetti dei flussi x,y,z; z_1 in Fig. 4.13)
    - processerà tutti gli altri pacchetti (ad esempio tutti i pacchetti non appartenenti ai flussi x,y,z; ad es. pacchetto w_0 in Fig. 4.13)

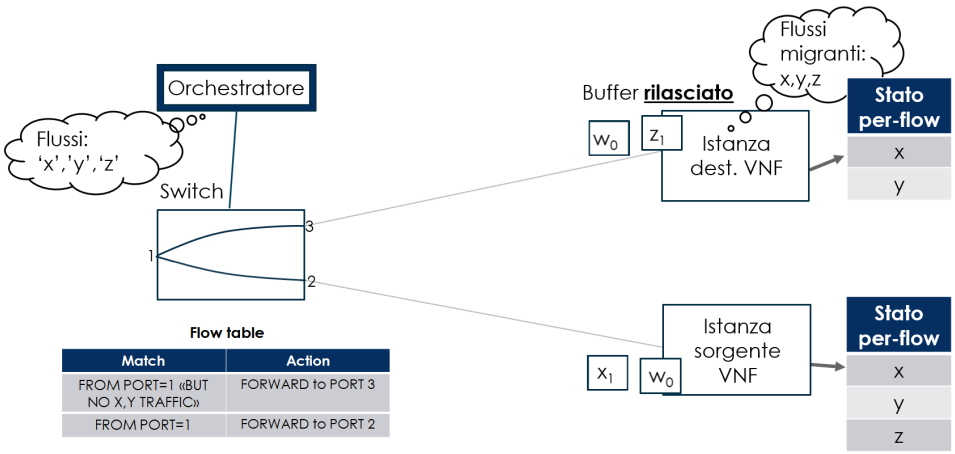


Figura 4.13. L'istanza destinazione della VNF apprende quali sono i flussi in gestione sull'istanza sorgente e rilascia i pacchetti accodati in precedenza.

11. può ora iniziare la migrazione dei flussi i cui pacchetti sono al momento ancora processati dall'istanza Src, ovvero quelli di cui quest'ultima ha il relativo stato (ad esempio flussi x, y, z in Fig. 4.14);

12. l'orchestratore manda i dovuti messaggi `flowmod` sullo switch affinchè:

    - il traffico ricevuto dall'istanza Src sia composto solo da pacchetti relativi a flussi che la stessa istanza ha già processato in passato, ovvero di cui ha il relativo stato (ad esempio flussi x, y, z in Fig. 4.14). In questo modo ora abbiamo isolato il traffico dei flussi migranti solo ed esclusivamente sull'istanza Src mentre Dst riceve il restante traffico. Si ricordi che Src sta ancora ricevendo pacchetti relativi a traffico di cui non ha lo stato necessario per il processing e che continuerà a scartarli come detto in precedenza;

    - il traffico di un flusso da migrare sia **duplicato** e ricevuto da entrambe le istanze; nell'esempio, per semplicità, migreremo e quindi duplicheremo il traffico di un flusso alla volta (ad esempio in Fig. 4.14 flusso x i cui pacchetti verranno ora ricevuti da entrambe le istanze);

    Si osservi in Fig. 4.14 che prima dei messaggi `flowmod` in considerazione in questo momento, sono stati inoltrato dallo switch all'istanza Src due pacchetti, che ora sono pertanto in volo, relativi ad un flusso di cui l'istanza sorgente ha lo stato e che potrebbe in futuro provocare il cambiamento del suddetto stato (ad esempio pacchetti x_1 e x_2 in Fig. 4.14).

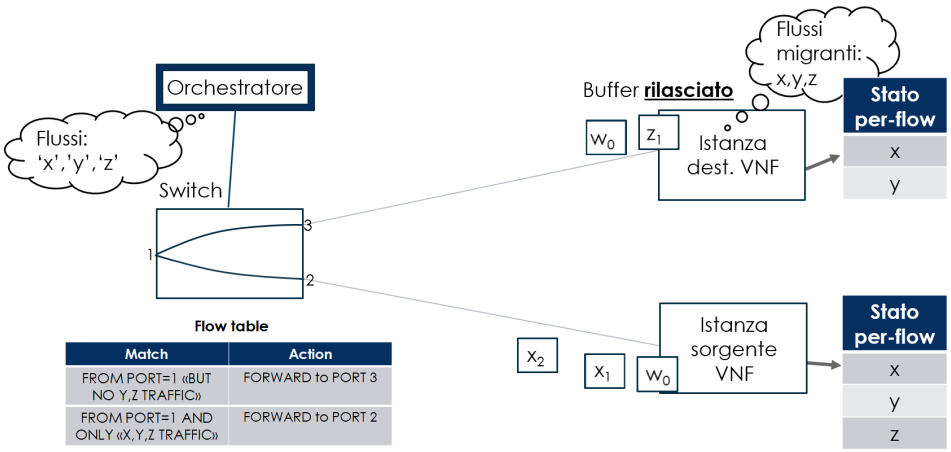


Figura 4.14. Lo switch con le flow-rules aggiornate dopo il passo 11(a).

13. l'orchestratore invia contemporaneamente due pacchetti speciali identici ad entrambe le istanze di VNF, mediante un'operazione di *packet-out* sullo switch (pacchetti blu in Fig. 4.15); anche questi pacchetti devono essere identificati in maniera univoca dalle due istanze (e.g da uno specifico valore del campo Protocol Type dell'header IP [28]). I due pacchetti speciali in esame specificheranno all'interno del payload qual è il flusso che si è deciso di migrare dall'istanza Src all'istanza Dst. Le loro funzioni sono diverse a seconda dell'istanza a cui arrivano. Chiameremo il pacchetto speciale che arriva a Src pacchetto "*stopper*" mentre quello che arriva a Dst pacchetto "*starter*". Le loro funzioni verranno chiarite in un passo successivo;

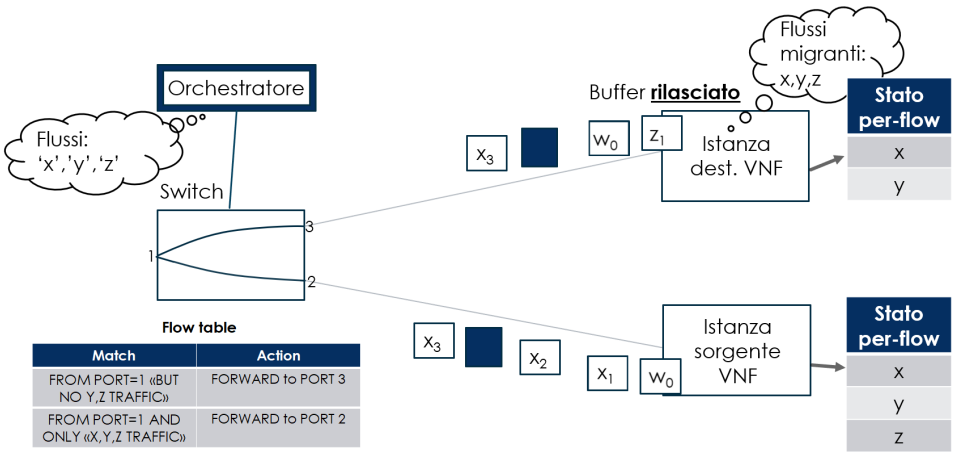


Figura 4.15. I due pacchetti speciali (*stopper/starter*) inviati contemporaneamente alle due istanze mediante l'operazione di packet-out da parte dell'orchestratore.

14. l'orchestratore manda i dovuti messaggi `flowmod` sullo switch affinchè il traffico relativo al flusso in corso di migrazione sia ricevuto ora soltanto dall'istanza Dst (ad esempio flusso x in Fig. 4.16);

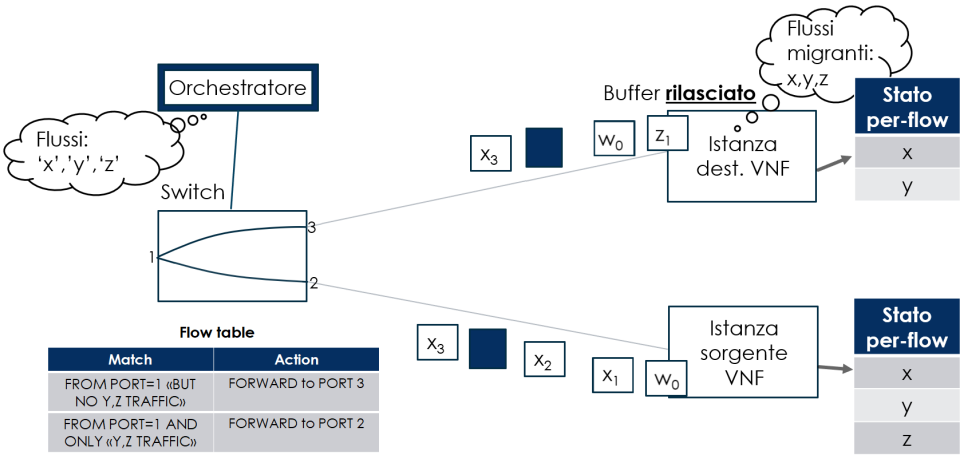


Figura 4.16. Lo switch con le flow-rules aggiornate dopo il passo 11(c).

15. a questo punto i comportamenti delle due istanze saranno i seguenti:

- l'istanza Src:
  - scarterà tutti i pacchetti relativi a nuovi flussi ricevuti dopo il pacchetto *blocker*, come descritto al punto 9 (ad esempio w_0 in Fig. 4.17);
  - processerà un pacchetto relativo ad un flusso da migrare se ricevuto **prima** del pacchetto *stopper* (di cui al punto 11(b)) riferito allo stesso flusso a cui esso appartiene (ad esempio x_1, x_2 in Fig. 4.17);
- l'istanza Dst, invece:
  - processerà tutti i pacchetti relativi a nuovi flussi, creandosi, se necessario, il relativo stato (ad esempio w_0 in Fig. 4.17);
  - scarterà un pacchetto relativo ad un flusso da migrare se ricevuto **prima** del pacchetto *starter* (di cui al punto 11(b)) riferito allo stesso flusso a cui esso appartiene (ad esempio z_1 in Fig. 4.17);

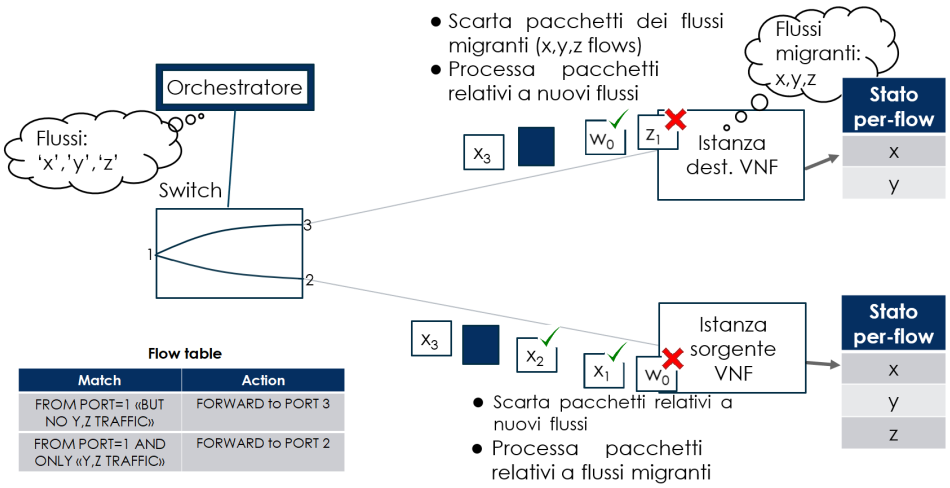


Figura 4.17. I comportamenti da parte delle due istanze prima della ricezione dei pacchetti *stopper/starter* riferiti a un flusso migrante.

16. i pacchetti speciali di cui al punto 11(b) vengono ricevuti dalle due istanze; esse si comporteranno nella maniera seguente:

    - l'istanza Src:
        - continuerà a scartare tutti i pacchetti relativi a nuovi flussi ricevuti dopo il pacchetto *blocker*, come descritto al punto 9;
        - inizierà la cosiddetta fase di stop-and-copy per il flusso specificato nel pacchetto *stopper* appena ricevuto. Durante questa fase l'istanza:
            * controllerà se lo stato relativo al flusso specificato nel pacchetto *stopper* appena ricevuto sia cambiato dopo la ricezione del pacchetto *blocker* di cui al punto 7. In caso affermativo, Src invierà copia dello stato aggiornato a Dst; viceversa significherà che lo stato che Dst ha ottenuto grazie alla pre-copy è tuttora consistente e valido. In quest'ultimo caso Src invierà pertanto solo una segnalazione a Dst con il solo fine di avvertirla dell'avvenuta fine del processing dei pacchetti relativi al flusso specificato nel pacchetto *stopper* ricevuto (questo meccanismo garantisce la mancanza di reordering dei pacchetti dello stesso flusso migrante);
            * scarterà tutti i pacchetti relativi al flusso specificato nel pacchetto *stopper* appena ricevuto (ad esempio x_3 in Fig. 4.18);
    - l'istanza Dst, invece:
        - continuerà a processare tutti i pacchetti relativi a nuovi flussi, creandosi, se necessario, il relativo stato (come specificato al punto 10);
        - se ha ricevuto da parte dell'istanza Src la versione aggiornata dello stato o la segnalazione di avvenuta fine del processing relativa al flusso specificato nel pacchetto *starter* appena ricevuto, processerà tutti i pacchetti relativi

al flusso specificato nel pacchetto *starter* (ad esempio x_3 in Fig. 4.18). In caso contrario i pacchetti verranno accodati fino a che una delle due condizioni non sia soddisfatta. Si noti che se l'istanza Dst riceve la segnalazione di avvenuta fine del processing da parte di Src significa che lo stato copiato su Dst durante la pre-copy è ancora valido e consistente. Questo permette il cosiddetto *rilascio anticipato* dei pacchetti accodati in precedenza, comportando un notevole risparmio in termini di latenze introdotte sui pacchetti stessi e di tempi di accodamento;

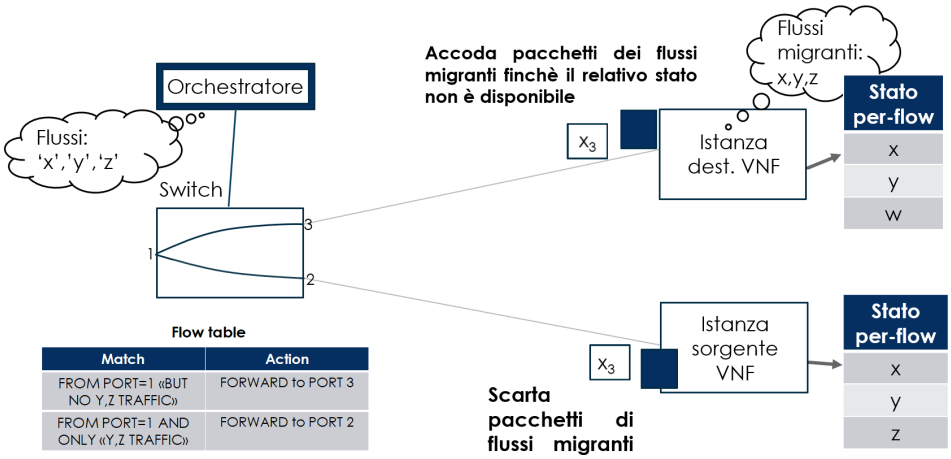


Figura 4.18. I comportamenti da parte delle due istanze dopo la ricezione dei pacchetti speciali riferiti a un flusso migrante.

17. si ripetono i passi 12, 13 e 14 fino a che tutti i flussi e il relativo stato sono stati migrati dall'istanza Src all'istanza Dst. Si osservi che, modificando adeguatamente i pacchetti speciali e i messaggi di `flowmod` dei passi 12, 13 e 14 è possibile effettuare la migrazione anche di più flussi in parallelo, riducendo la durata della migrazione. Una volta che tutti i flussi e tutto lo stato sono stati migrati da Src a Dst, vengono mandati i dovuti messaggi `flowmod` sullo switch in modo tale da inoltrare tutto il traffico solo ed esclusivamente all'istanza Dst (Fig. 4.19) e la migrazione può considerarsi a tutti gli effetti conclusa;

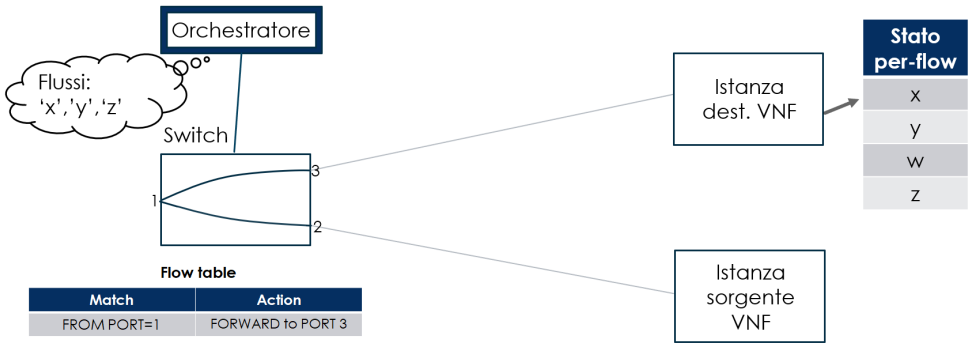


Figura 4.19. Migrazione conclusa: tutto il traffico viene inoltrato solo all'istanza destinazione.

4.1.2 Migrazione di flussi con stato multi-flow e di stato all-flow/not-flow

Finora l'algoritmo ha considerato la migrazione di flusso con solo stato di tipo per-flow. Si deve considerare, però, che le funzioni di rete possono avere anche stato di tipo multi-flow e all-flow. Nella Tabella 4.2 sono riportate, in base al tipo di stato in esame, le politiche adottate dall'algoritmo per la migrazione di stato non attinente a flussi (ovvero *not-flow*), di stato condiviso tra alcuni o tutti i flussi (ovvero *multi-flow* e *all-flow*) e dei flussi stessi che condividono queste tipologie di stato. La classificazione fatta è la seguente:

- **configurazione statica**: stato che rappresenta la configurazione della NF in esame (ad esempio la configurazione delle sue interfacce di rete); essendo questo tipo di stato necessario alla NF per poter processare tutti i flussi, l'algoritmo lo migrerà prima di tutti gli altri;

- stato **necessario** per poter processare nuovi flussi; le politiche di migrazione che si possono adottare per questo tipo di stato sono due, a seconda delle necessità della NF:

  - **consistenza soft**: se la NF può processare nuovi flussi senza avere necessariamente una versione consistente dello stato in esame durante la migrazione, si può decidere di migrare la versione attuale di quest'ultimo insieme alla configurazione statica. In questo modo si permette alla NF di poter processare subito i nuovi flussi. Successivamente, a migrazione completata, si effettua il *merging* delle due versioni di stato su Src e Dst, che possono essere pertanto anche temporaneamente inconsistenti tra loro durante la migrazione. Un esempio di questo tipo di stato può essere il pool di indirizzi IP e porte disponibili per la traduzione in un NAT;
  - **consistenza hard**: se la NF *non* può processare nuovi flussi senza avere una versione consistente dello stato in esame durante la migrazione. Un esempio può essere rappresentato da un ipotetico contatore di pacchetti di un IDS, condiviso tra più flussi, che permette alla NF di decidere se scartare o meno

determinati pacchetti (ad esempio, per impedire o mitigare un attacco). In questo caso, come si può facilmente dedurre, non si può permettere a Dst di far processare nuovi flussi con una versione inconsistente dello stato. Pertanto sarà necessario fermare su Src il processing dei flussi che afferiscono allo stato condiviso in esame, se ne effettuerà la migrazione (insieme al relativo traffico) applicando l'algoritmo spiegato nella Sezione 4.1.1 e si farà ripartire il processing sull'istanza Dst;

- stato **potenzialmente necessario** per poter processare nuovi flussi: un esempio di questo tipo di stato può essere rappresentato dai dati relativi al carico sui diversi server di back-end in un load balancer. Qui vi è un grado di libertà in più, infatti si può decidere tra due alternative:

  1. lasciar processare a Dst i nuovi flussi, permettendo così a Src e Dst di avere due versioni inconsistenti tra loro dello stato in esame. Una volta terminata la migrazione, si effettua il *merge* dello stato per rendere consistente lo stato su Dst;
  2. non permettere a Dst di processare i nuovi flussi senza lo stato in esame: in questo caso, si procederà alla migrazione dello stato e del traffico relativo subito dopo la migrazione della configurazione statica;

- stato **non necessario** per poter processare nuovi flussi: un esempio di questo tipo di stato può essere rappresentato dai log di una NF generica; in questo caso si decide di migrare ed effettuare il *merge* dello stato in esame quando la migrazione di tutte le altre tipologie di stato sarà completata.

Tabella 4.2. Politiche adottate dall'algoritmo per la migrazione di stato condiviso tra flussi

Tipo di stato	**Politica adottata per la migrazione**	**Esempi**
Configurazione statica	Migrare *prima* di tutti gli altri stati	1) Configurazione delle interfacce (e.g. indirizzo IP) 2) Politiche di sicurezza (Firewall + NIDS) 3) Indirizzi IP dei server di backend (Load Balancer) 4) Configurazione DHCP (Range di indirizzi IP + def. Gw + DNS servers) 5) Limiti di banda (Traffic Shaper)
Stato **necessario** (*Dinamico*) per poter processare nuovi flussi	**CONSISTENZA SOFT**: Migrare *dopo/insieme* alla configurazione statica e, se necessario, effettuare il merging a migrazione completata	1) Pool di indirizzi IP e porte disponibili per la traduzione (NAT) 2) Pool di indirizzi IP disponibili per l'assegnazione (DHCP Server) 3) Dati relativi al carico sui diversi server di backend (necessari all'algoritmo di backend del Load Balancer) [potenzialmente necessario]
	CONSISTENZA HARD: *Stop* del processing dei flussi che afferiscono a questo stato condiviso sulla istanza sorgente, *migrare* stato/traffico applicando l'algoritmo e rifar partire il processing sulla istanza destinazione	1) IDS registri di connessione (e.g. contatori di pacchetti)

Stato **potenzialmente necessario** (*Dinamico*) per poter processare nuovi flussi	**Due opzioni**: 1) *Migrare e fare il merging* a migrazione effettuata (l'istanza destinazione può e prende decisioni senza questo stato) 2) Migrare *dopo/insieme* alla configurazione statica	1) Dati relativi al carico sui diversi server di backend (necessari all'algoritmo di backend del Load Balancer)
Stato **non necessario** (*Dinamico*) per poter processare nuovi flussi	*Migrare ed effettuare merging* a migrazione completata	1) Logs (e.g. FW/IDS Log)

Capitolo 5

Implementazione

In questo capitolo viene illustrato il prototipo realizzato per validare l'algoritmo descritto nel Capitolo 4.

5.1 Tecnologie utilizzate

Si è scelto di validare l'algoritmo prendendo il NAT come VNF d'esempio; in particolare, è stato usato e modificato il servizio di rete NAT modellato attraverso il linguaggio YANG [29] e disponibile nel framework IOVnet, basato sul progetto IO Visor [30] e su eBPF. Per quanto concerne la rete SDN e la sua gestione, invece, si è fatto ricorso ad un bundle ONOS creato appositamente che ha svolto le funzioni di orchestratore della migrazione usando il protocollo OpenFlow per pilotare switch virtuali Open vSwitch e comunicando con i NAT mediante REST API.

5.1.1 BPF e l'evoluzione eBPF

BPF [31], acronimo di Berkeley Packet Filter, è il primo packet filter in-kernel proposto nel 1992 e introdotto nel kernel Linux a partire dalla versione 2.1.75. L'obiettivo principale di BPF è quello di effettuare il filtraggio dei pacchetti già all'interno del kernel in maniera veloce ed efficiente minimizzando così la copia di pacchetti indesiderati da kernel verso processi in userspace. Tutto questo è possibile grazie ai programmi BPF che sono scritti in *bytecode* e il cui codice viene iniettato nell'omonima virtual machine per l'esecuzione. Una volta che il programma è iniettato ed è in esecuzione nel kernel, esso può leggere i pacchetti e filtrarli per decidere se è necessario scartarli o passarli in userspace.

Negli ultimi anni BPF è evoluto profondamente e in particolare oggi con **eBPF** (extended BPF), introdotto a partire dalla versione 3.15 del kernel Linux, è possibile eseguire codice generico nel kernel con meno limitazioni rispetto a prima, attraverso una virtual machine general purpose (*eBPF engine*) in un ambiente sicuro, flessibile e performante. eBPF ha introdotto inoltre la possibilità di eseguire programmi per il tracing e il processamento dei pacchetti (non solo per il filtraggio). eBPF ha introdotto anche le cosiddette mappe eBPF che sono delle strutture dati di tipo mappa chiave/valore (persistenti tra diversi eventi) che possono essere condivise tra userspace e kernel space. Un aspetto

importante per il presente lavoro di tesi è la necessità di dover accodare pacchetti per le esigenze illustrate nel Capitolo 4. A tal riguardo, si sottolinea che eBPF non permette di "salvare" pacchetti nel kernel. Questa limitazione ha influenzato in maniera importante l'implementazione poiché è stato necessario accodare i pacchetti in buffer residenti in userspace.

5.1.2 IO Visor

IO Visor [30] è un progetto open-source sostenuto da una comunità di sviluppatori che ha come obiettivo quello di promuovere l'innovazione e lo sviluppo di servizi IO virtualizzati per diverse esigenze, tra cui vi è quella di svolgere funzioni di rete.

Il progetto IO Visor fornisce un dataplane programmabile e un insieme di strumenti a supporto dello sviluppo (chiamati IO Visor Dev tools) per semplificare la creazione dei cosiddetti *IO modules*. Questi ultimi sono oggetti IO general purpose ad alte performance che possono essere iniettati dinamicamente nel kernel a runtime per effettuare operazioni di diverso genere (ad esempio networking, tracing, security). All'interno del kernel gli IO modules possono essere eseguiti, collegati tra di loro e attaccati a diversi hooks del kernel (ad esempio TC, XDP). L'idea principale è quella di suddividere le componenti dell'IO module tra userspace e kernel space. Nel control plane, in userspace, un controller controlla l'IO Module iniettato nel kernel. Gli IO modules in kernel space sono implementati attraverso codice eBPF eseguito mediante l'eBPF engine, ovvero la macchina virtuale descritta in precedenza. È possibile far comunicare le diverse componenti in kernel e userspace, mediante le mappe eBPF.

IOVnet

IOVnet è un framework creato dal Netgroup del Politecnico di Torino [5] per la creazione e il deploy di funzioni di rete virtualizzate basate su IO Visor, come bridge, router, NAT e firewall. Ogni VNF può essere considerata un IOModule e questo permette di interconnettere diverse funzioni per creare una catena di VNF personalizzata in base alle esigenze dell'utente. Le funzionalità principali del framework sono:

- il supporto per la creazione di funzioni di rete attraverso la implementazione di un fast path (ovvero codice eBPF che verrà iniettato in kernel) e di uno slow path (ovvero codice che verrà eseguito in user space per poter gestire pacchetti che non possono essere gestiti in kernel space) ;
- la possibilità di configurare le funzioni di rete attraverso REST API e CLI mediante la definizione di primitive che rappresentano il control plane del servizio;
- la possibilità di interconnettere le funzioni di rete in maniera personalizzata, semplice e veloce.

IOVnet si rivolge sia agli utenti finali, che possono sfruttare i servizi già presenti in esso, sia a sviluppatori che vogliono creare per le proprie esigenze un servizio di rete personalizzato o non ancora presente. A tal fine, gli sviluppatori sono agevolati dal fatto che è prevista la generazione automatica di parte del codice necessario per poter implementare, a partire

dal modello YANG del servizio stesso, sia le REST API sia i metodi per l'interfacciamento attraverso CLI.

Nella presente tesi, è stato modificato un servizio di rete preesistente nel framework IOVnet, ovvero il NAT, per permetterne la migrazione mediante l'algoritmo presentato nel Capitolo 4.

Architettura

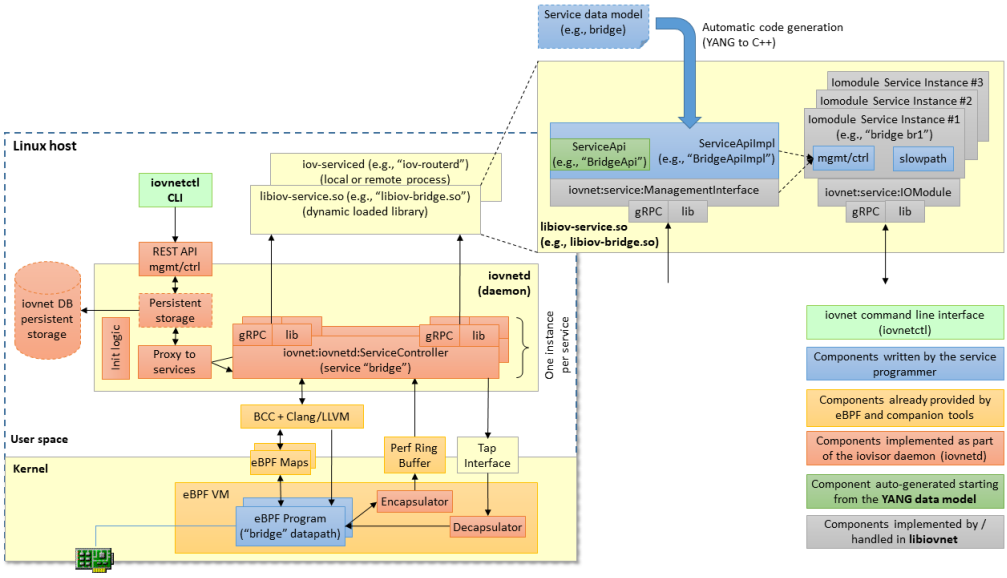


Figura 5.1. L'architettura di IOVnet.

L'architettura di IOVnet mostrata in Fig. 5.1 può essere riassunta in quattro componenti principali:

- **iovnetd**: è il componente principale dell'architettura, un demone che controlla il servizio IOVnet e che pertanto si occupa di far partire il servizio principale per poter permettere il deploy, la configurazione e l'arresto dei servizi di rete che sono richiesti dall'utente. Le richieste di quest'ultimo sono ricevute da iovnetd mediante la sua interfaccia REST di northbound;

- **servizi IOVnet**: rappresentano fondamentalmente le funzioni di rete di cui si è parlato in precedenza. Essi possono essere considerati simili a dei plug-in che possono essere installati e lanciati a run-time;

- **iovnetctl**: modulo che implementa la gestione della CLI che permette il controllo di iovnetd (ad esempio per la creazione, configurazione, interrogazione e arresto di servizi);

- **libiovnetd**: libreria che facilita lo sviluppo di servizi di rete. Essa permette facilmente, ad esempio, la creazione di collegamenti tra servizi e l'accesso alle mappe eBPF.

5.1.3 YANG

Il linguaggio di modellazione YANG [29] è stato sviluppato nell'ambito della definizione del protocollo NETCONF (network configuration protocol) [32]. Esso è stato ideato con l'obiettivo di descrivere la configurazione e lo stato di componenti di rete ma può essere usato anche per altri scopi, grazie alla sua elevata flessibilità nel definire nuovi tipi di dato oltre a quelli predefiniti disponibili. Attraverso un modello YANG si possono rappresentare dati tramite una struttura ad albero che consente di codificarli in diversi modi (ad es. JSON o XML).

Nel prototipo realizzato in questa tesi, è stato necessario modellare le strutture dati necessarie al NAT per poter non solo eseguire le sue funzioni di base ma per poter supportare la migrazione. Inoltre, grazie al modello YANG del NAT, è stato possibile generare automaticamente le classi da dover implementare successivamente per poter controllare il NAT mediante chiamate REST. Per poter comprendere al meglio il modello YANG che la tesi proporrà successivamente, si elencano di seguito alcuni dei componenti di base di un modello YANG che sono stati usati:

- **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 chiamati **submodules**;
- **nodi**: sono gli elementi che contengono i dati veri e proprio e possono essere di diversi tipi:
  - **leaf**: contiene un dato primitivo, come un intero o una stringa; non ha nodi figli;
  - **leaf-list**: nodo composto da nodi leaf dello stesso tipo;
  - **container**: nodo che non assume un valore ma può contenere altri nodi (leaf o container);
  - **list**: lista di nodi container dello stesso tipo

5.1.4 Openflow, Open vSwitch, ONOS

Openflow

In una rete SDN, requisito fondamentale per l'algoritmo della presente tesi, vi è la necessità di controllare il data plane mediante un control plane, data la separazione tra di essi. Uno dei meccanismi che permette di fare ciò è il protocollo *OpenFlow* proposto dalla Open Networking Foundation (ONF) [33]. Il suddetto protocollo rappresenta l'interfaccia di southbound con cui i controller OpenFlow di una rete SDN possono stabilire i comportamento dei dispositivi compatibili con OpenFlow (ad esempio switch) ad esempio per determinare, mediante opportuno filtraggio, il percorso dei pacchetti di rete che li attraversano (ovverosia il forwarding plane). Attraverso il controller si mettono in atto,

quindi, le politiche di rete prescelte applicandole sui dispositivi di rete usando le API di OpenFlow. Uno dei punti di forza di OpenFlow è quello di permettere una gestione del traffico più sofisticata grazie al fatto di poter filtrare i pacchetti basandosi su una molteplicità di campi diversi che vanno dal livello 2 al livello 4 del modello ISO/OSI.

Il paradigma di programmazione di OpenFlow si basa su tabelle chiamate *flow tables* (presenti sui dispositivi di rete OpenFlow) e sulle cosiddette *actions*. Le flow tables contengono le *flow entries*, composte da vari campi tra cui i *match fields* le suddette actions. Quando un dispositivo di rete riceve un pacchetto esso ricerca, attraverso le flow entries, una corrispondenza (matching) su determinati criteri (i match fields) quali ad esempio valori degli header di un pacchetto o della porta d'ingresso da cui proviene il pacchetto, selezionando così le azioni (actions) da eseguire sul pacchetto in esame (ad esempio modifica del pacchetto, selezione della porta di uscita sul dispositivo). Per poter creare, modificare, cancellare le flow entries il controller invia dei messaggi chiamati messaggi `flowmod` nei quali si specificano per l'appunto i criteri di matching e le azioni corrispondenti da effettuare.

Oltre alla possibilità di decidere il percorso dei pacchetti di rete dei dispositivi pilotati, OpenFlow permette al controller anche di poter iniettare pacchetti nel data plane. Questo è possibile grazie a un messaggio chiamato *packet-out* che il controller può inviare al dispositivo che dovrà immettere i pacchetti nel data-plane.

Come detto pocanzi, il controller OpenFlow è l'elemento responsabile della gestione dei dispositivi OpenFlow. Esso può agire in due modalità: proattiva e reattiva. Nella prima, il controller aggiunge e rimuove le flow entries in maniera statica e predefinita mentre nella seconda aggiunge e rimuove dinamicamente le flow entries a seconda di condizioni prestabilite, dopo aver ricevuto pacchetti dal dispositivo di rete il quale al momento non sa come trattare.

La molteplicità dei controller esistenti rappresenta per certi versi il grande interesse da parte della comunità verso il protocollo OpenFlow. Tra le tante alternative a disposizione si è scelto di usare un controller *ONOS* [34] per la qualità della documentazione disponibile, per il supporto in rete e per il linguaggio di programmazione usato (Java). Il lavoro di questa tesi sfrutta Openflow in modalità proattiva.

Open vSwitch

Per la presente tesi si è scelto di pilotare mediante il protocollo appena presentato software switch virtuali. La scelta è ricaduta su **Open vSwitch** (OvS) [35], una implementazione open-source di uno switch software virtuale. Attraverso il protocollo Openflow presentato pocanzi, sono state effettuate la creazione, modifica e cancellazione di regole OpenFlow sui suddetti switch OvS.

ONOS

ONOS (Open Network Operating System) [34] è un progetto open-source che si prefigge l'obiettivo di creare un sistema operativo di rete basato sulla tecnologia SDN. Esso fornisce gli strumenti per gestire i componenti di rete SDN. Il core di ONOS, come le applicazioni che vengono eseguite su di esso, sono scritte in linguaggio Java sotto forma di bundle

che vengono caricati in un container Karaf OSGi [36], componente di sistema per Java che permette l'installazione e l'attivazione di moduli nella Java Virtual Machine. Le applicazioni che si adoperano in ONOS possono essere gestite tramite CLI, REST API o GUI. Ciascuna delle applicazioni può fare ricorso a servizi messi a disposizione da ONOS per poter, ad esempio, conoscere lo stato della rete e controllare il traffico che l'attraversa. In particolare, uno di questi servizi è stato usato nella presente tesi per far sì che l'orchestratore della migrazione possa interfacciarsi con switch OVS appena presentati mediante il protocollo Openflow.

5.2 La funzione di rete NAT in IOVnet

Passiamo adesso a illustrare l'implementazione della VNF NAT in IOVnet e in particolare il suo modello YANG, il codice eBPF e il codice in userspace con l'aggiunta delle modifiche apportate per supportare la migrazione.

5.2.1 Il modello YANG

In questa sezione verranno illustrate le porzioni salienti del data model YANG del NAT e le modifiche ad esso apportate per poter supportare la migrazione.

- **Interfacce pubblica e privata**: il servizio NAT in IOVnet è stato pensato per essere trasparente e richiede la presenza di un router. Esso ha due interfacce, una privata e una pubblica modellate attraverso *container* come mostrato nel listing 5.1. La prima non ha un indirizzo IP e solo attraverso essa è possibile instaurare una connessione mediante protocollo TCP o UDP. L'interfaccia pubblica, invece, a differenza della prima ha un indirizzo IP e non può essere instaurata una connessione mediante essa. Ricevendo pacchetti relativi a una determinata connessione TCP/UDP dall'interfaccia privata il NAT crea un mapping, effettua la traduzione indirizzo-porta modificando i valori negli header IP/TCP/UDP e invia sull'interfaccia pubblica i pacchetti così tradotti. I pacchetti ARP vengono inoltrati normalmente tra le due interfacce, senza apportar loro modifiche. La configurazione dei valori di questi parametri è obbligatoria. Si precisa che non è stato necessario estendere questa porzione del data model per supportare la migrazione.

```
container publicinterface {
    leaf ip {
        type inet:ipv4-address;
        description
        "Public IP address of the NAT.";
    }

    leaf port {
        type yang:uuid;
        description
        "Port the public interface is connected to.";
```

```
        }
    }

    container privateinterface {
        leaf port {
            type yang:uuid;
            description
            "Port the private interface is connected
               to.";
        }
    }
```

Listing 5.1. Porzione del modello YANG che descrive la VNF NAT.

- **Mappings**: quando il servizio di rete NAT processa per la prima volta i pacchetti di una determinata connessione TCP/UDP crea un mapping modellato attraverso una *list* come mostrato nel listing 5.2 che mette in relazione:
  - una *chiave* che identifica il flusso TCP/UDP (ovvero indirizzo IP sorgente, porta TCP/UDP sorgente, protocollo UDP/TCP);
  - un *valore* ovvero la traduzione da effettuare sugli header IP e TCP o UDP del pacchetto (ovvero nuovi indirizzo IP sorgente e nuova porta TCP/UDP sorgente).

  Si precisa che non è stato necessario estendere questa porzione del data model per supportare la migrazione.

```
    list mappingentry {
        key "intsaddr intsport proto";
        description
        "NAT mapping entry.";

        leaf intsaddr {
            type inet:ipv4-address;
            description
            "Corresponds to the source IPv4 address
            of the IPv4 packet.";
        }

        leaf intsport {
            type inet:port-number;
            description
            "Corresponds to the source port of the
            IPv4 packet.";
        }
```

```
        leaf extsaddr {
            type inet:ipv4-address;
            description
            "External IPv4 address assigned by NAT.";
        }

        leaf extsport {
            type inet:port-number;
            description
            "External source port number assigned by
               NAT.";
        }

        leaf proto {
            type enumeration {
            enum ICMP;
            enum TCP;
            enum UDP;
            }
            description
            "Upper-layer protocol associated with this
               mapping.";
        }
    }
```

Listing 5.2. Porzione del modello YANG che descrive la VNF NAT.

- **Ruolo nella migrazione**: la *leaf* mostrata nel listing 5.3 è stata aggiunta per supportare la migrazione della VNF. Essa rappresenta il ruolo che l'istanza in esame deve avere durante la migrazione, ovvero se è l'istanza sorgente o destinazione. Questa informazione servirà all'istanza per scegliere quale comportamenti adottare durante la migrazione.

```
    leaf natrole {
        type enumeration {
            enum SOURCE;
            enum DESTINATION;
        }
        default SOURCE;
        description "Role of the NAT in a NAT migration
           process.";
    }
```

Listing 5.3. Porzione del modello YANG che descrive la VNF NAT.

- **Flussi migranti**: la *list* mostrata nel listing 5.4 è stata aggiunta per supportare la migrazione. Essa rappresenta la lista dei flussi TCP/UDP da migrare dall'istanza sorgente all'istanza destinazione del NAT. Essa identifica il flusso TCP/UDP migrante mediante la *chiave* formata dalle informazioni indirizzo IP sorgente, porta TCP/UDP sorgente, protocollo UDP/TCP presenti negli header dei pacchetti in transito. Attraverso questa *list* è stato possibile comunicare all'istanza destinazione del NAT mediante chiamata REST quali fossero i flussi migranti, in modo tale da potersi comportare di conseguenza, come descritto in 4.1.1.

```
list migratingflows {
    key "intsaddr intsport proto";
    description
    "TCP/UDP flows that are migrating from Source to
       Destination NAT";

    leaf intsaddr {
        type inet:ipv4-address;
        description
        "Corresponds to the source IPv4 address
        of the IPv4 packet.";
    }

    leaf intsport {
        type inet:port-number;
        description
        "Corresponds to the source port of the
        IPv4 packet.";
    }

    leaf extsport {
        type inet:port-number;
        description
        "External source port number assigned by
           NAT.";
    }

    leaf proto {
        type enumeration {
            enum ICMP;
            enum TCP;
            enum UDP;
        }
        description
        "Upper-layer protocol associated with this
           mapping.";
```

```
        }
    }
```

Listing 5.4. Porzione del modello YANG che descrive la VNF NAT.

Il data model completo del NAT è presente nell'appendice B.

5.2.2 Il codice eBPF - datapath

Nel datapath il codice eBPF si occupa della funzione principale del NAT, ovvero la traduzione dei pacchetti TCP/UDP in transito. In questa sezione verranno illustrate le modifiche fatte al codice eBPF in datapath per poter permettere la migrazione della VNF NAT.

Le strutture dati

Per identificare il ruolo dell'istanza NAT durante la migrazione (ovvero sorgente/destinazione), è stata aggiunta una nuova mappa chiamata *natrole_map* (5.5):

```
BPF_ARRAY(natrole_map, uint8_t, 1);
```

Listing 5.5. Parte del codice sorgente eBPF modificato del NAT in IOVnet.

Questa informazione è fondamentale in quanto servirà all'istanza per scegliere quale comportamenti adottare durante la migrazione, come descritto in 4.1.1.

Per poter effettuare la traduzione il NAT usa due mappe (v. 5.6) chiamate rispettivamente *egress_nat_map* e *reverse_nat_map*: si fa ricorso alla prima quando si vuole tradurre un pacchetto proveniente dalla interfaccia privata e alla seconda quando si vuole tradurre un pacchetto proveniente dalla interfaccia pubblica. La prima, pertanto, ha come chiave l'indirizzo IP sorgente, la porta sorgente TCP/UDP e il protocollo della quintupla TCP/UDP. Come valore ha, invece, la traduzione corrispondente (ovvero i nuovi indirizzo IP sorgente e porta sorgente TCP/UDP) con un timestamp. Quest'ultimo ha lo scopo di far scadere la entry entro un certo timeout prestabilito per poter riassegnare la nuova porta sorgente TCP/UDP ad un nuovo flusso. La seconda, invece, ha come chiave l'indirizzo IP destinazione, la porta destinazione TCP/UDP e il protocollo della quintupla TCP/UDP. Si ricordi che il NAT crea un nuovo mapping per i pacchetti relativi ad un nuovo flusso provenienti dalla sola interfaccia privata e non da quella pubblica.

Per le esigenze di migrazione obiettivo di questa tesi, è stato aggiunto ai valori delle due mappe dell'istanza NAT sorgente un nuovo campo chiamato *must_drop*(v. 5.6). Quest'ultimo può essere considerato un flag che indica se i pacchetti del flusso in esame devono essere scartati o meno secondo le direttive descritte in 4.1.1. Il valore del campo *must_drop* è settato a *false* fino a che non viene ricevuto il pacchetto *stopper* descritto in 4.1.1, momento in cui assumerà il valore *true*. Attraverso i valori degli header dei pacchetti ricevuti mappati sulle strutture *egress_nat_key* e *reverse_nat_key* sono stati calcolati degli hash per identificare il flusso TCP/UDP di appartenenza di ogni pacchetto.

È stato inoltre aggiunto un flag chiamato *drop_new_flows* (v. 5.6) che indica all'istanza sorgente di NAT se deve scartare i pacchetti che, se processati, provocherebbero la creazione di un nuovo mapping nelle due mappe appena presentate (ovvero pacchetti di un nuovo flusso TCP/UDP non ancora gestito dal NAT). Esso assume il valore *false* fino a che non viene ricevuto il pacchetto *blocker* descritto in 4.1.1, momento in cui assumerà il valore *true*. Ogni volta che viene processato un pacchetto ricevuto da una delle due interfacce, attraverso una ricerca nelle mappe, viene quindi adesso controllato se il pacchetto deve essere scartato o meno controllando il valore dei due flag appena presentati, secondo le direttive descritte in 4.1.1.

```
BPF_HASH(egress_nat_map, struct egress_nat_key, struct
   egress_nat_value, NAT_MAP_DIM);
BPF_HASH(reverse_nat_map, struct reverse_nat_key, struct
   reverse_nat_value, NAT_MAP_DIM);
BPF_TABLE("array", u32, u32, drop_new_flows, 1);
struct egress_nat_key {
  __be32 src_ip;
  __be16 src_port;
  __be16 proto;
};

struct egress_nat_value {
  __be32 ip_new;
  __be16 port_new;
  u64 timestamp;
  __be8 must_drop;
};

struct reverse_nat_key {
  __be32 dst_ip;
  __be16 dst_port;
  __be16 proto;
};

struct reverse_nat_value {
  __be32 ip_new;
  __be16 port_new;
  __be8 must_drop;
};
```

Listing 5.6. Parte del codice sorgente eBPF modificato del NAT in IOVnet.

Per implementare i pacchetti *blocker*, *stopper* e *starter* si è deciso di usare tre particolari valori del campo "Protocol type" del protocollo IP [28] non ancora assegnati (si è scelto rispettivamente il valore 143, 144 e 145). Il codice eBPF è stato modificato,

pertanto, per poter riconoscere questi tre pacchetti speciali e gestirli in maniera consona a quanto descritto in 4.1.1. In particolare:

- l'istanza NAT sorgente:
  - se riceve il pacchetto *blocker* aggiornerà il valore del flag *drop_new_flows* a *true* e manderà il pacchetto stesso con i metadati opportuni allo slowpath che provvederà a informare l'orchestratore dei flussi correntemente gestiti, come descritto in 4.1.1;
  - se riceve un pacchetto *stopper* relativo a un flusso TCP/UDP aggiornerà il valore del flag *must_drop* a *true* e manderà il pacchetto stesso con i metadati opportuni (in questo caso l'hash che identifica il flusso TCP/UDP) allo slowpath che provvederà a mandare il mapping all'istanza destinazione del NAT (in caso il mapping sia cambiato dopo la pre-copy) e ad informarla del fatto che il processing dei pacchetti per il flusso in esame è terminato;
- l'istanza NAT destinazione:
  - se riceve un pacchetto *starter* relativo a un flusso TCP/UDP manderà il pacchetto stesso con i metadati opportuni (in questo caso l'hash che identifica il flusso TCP/UDP) allo slowpath che provvederà a accodare il pacchetto nel relativo buffer in attesa del mapping corrispondente come descritto in 4.1.1;

Infine è stato modificato il comportamento dell'istanza destinazione del NAT, in modo tale che mandasse tutti i pacchetti relativi ai flussi TCP/UDP con relativo hash identificativo del flusso stesso (ad eccezione dei pacchetti provenienti direttamente dallo slopwpath e dei pacchetti *starter*) nello slowpath per le esigenze di buffering descritte in 4.1.1.

5.2.3 Slowpath

Il NAT senza supporto per la migrazione non gestisce alcun pacchetto attraverso lo slowpath, processando tutti i pacchetti in datapath. Per le esigenze di migrazione e per le limitazioni di eBPF precedentemente esposte, nello slowpath vi è la necessità di accodare pacchetti, rilasciare pacchetti e gestire i pacchetti speciali *starter*, *stopper* e *blocker* per cui:

- sono state implementate nuove REST API per poter:
  - assegnare il ruolo all'istanza NAT per la migrazione (su entrambe le istanze sorgente e destinazione);
  - comunicare all'istanza NAT destinazione i flussi migranti, come descritto in 4.1.1;
- sono state implementate le logiche per:
  - poter identificare ogni flusso TCP/UDP mediante un hash come descritto precedentemente (su entrambe le istanze sorgente e destinazione);

  - poter accodare i pacchetti di ogni flusso TCP/UDP in un buffer distinto per ogni flusso (solo sull'istanza NAT destinazione) dato che eBPF non permette di l'accodamento degli stessi in kernel;
  - poter gestire i comportamenti da tenere quando sono ricevuti i pacchetti speciali *blocker*, *starter* e *stopper*;
  - poter rilasciare i pacchetti precedentemente accodati controllando i requisiti descritti in 4.1.1 mediante un nuovo thread lanciato appositamente (solo sull'istanza NAT destinazione).

Le strutture dati più importanti aggiunte in slowpath solo sull'istanza destinazione del NAT sono le seguenti:

- una struttura *pckt* (v. 5.7) che rappresenta il pacchetto accodato che contiene:
  - **data**: il pacchetto vero e proprio che viene accodato;
  - **egress**: un flag che indica se l'interfaccia di provenienza del pacchetto è quella pubblica o privata;
  - **is_starter**: un flag che indica se il pacchetto accodato è un pacchetto di tipo *starter*;

```
typedef struct {
        std::vector<uint8_t> data;
        bool egress;
        bool is_starter;
} pckt;
```

Listing 5.7. Parte del codice sorgente aggiunto nel NAT di IOVnet.

- una mappa *packetBuffers* (v. 5.8) che ha come chiave un hash che identifica il flusso TCP/UDP e come valore una coda di strutture *pckt* appena descritte (ovvero il buffer dei pacchetti del flusso TCP/UDP):

```
  std::unordered_map<uint32_t, std::queue<pckt>>
     packetBuffers;
```

Listing 5.8. Parte del codice sorgente aggiunto nel NAT di IOVnet.

- un vettore *migrating_flows* che contiene strutture di tipo *migrating_flow* (v. 5.9) che identificano ogni flusso migrante e che contengono informazioni importanti per poter garantire il rilascio dei buffer di ogni flusso TCP/UDP al momento opportuno. Essa contiene:
  - **hash** e **hash_reverse**: i due hash del flusso TCP/UDP in considerazione, calcolati sui valori descritti nella Sezione 5.2.2;

- il flag **starter_received**: specifica se il NAT destinazione ha ricevuto il pacchetto *starter* relativo al flusso TCP/UDP considerato;
- il flag **mappingentry_received**: specifica se il NAT destinazione ha ricevuto il mapping relativo al flusso TCP/UDP considerato;
- i flag **starter_found** e **starter_found_reverse**: specificano se il pacchetto *starter* è stato già trovato e scartato nei buffer relativi al flusso TCP/UDP considerato.

```
typedef struct {
        uint32_t hash;
        uint32_t hash_reverse;
        bool starter_received;
        bool mappingentry_received;
        bool starter_found;
        bool starter_found_reverse;
} migrating_flow;
std::vector<migrating_flow> migrating_flows;
```

Listing 5.9. Parte del codice sorgente aggiunto nel NAT di IOVnet.

5.3 L'orchestratore della migrazione

È stato creato un bundle in ONOS per gestire la migrazione e in particolare esso:

- ha permesso di pilotare gli switch Openflow al fine di:
  - gestire il percorso del traffico di rete con una granularità fine (flusso TCP/UDP);
  - effettuare le operazioni di creazione e *packet-out* relative ai pacchetti speciali *blocker*, *stopper*, *starter*;
- ha permesso la comunicazione tra le istanze di NAT e l'orchestratore (anche per operazioni di segnalazione) mediante REST API appositamente create.

La migrazione può essere effettuata, inoltre, migrando un singolo flusso TCP/UDP alla volta oppure migrando molteplici flussi TCP/UDP in parallelo (ad esempio più flussi aventi lo stesso indirizzo IP sorgente). Sono state gestite inoltre problematiche quali la possibile perdita dei pacchetti speciali *blocker*, *stopper* e *starter* mediante timeout e reinvio dei suddetti pacchetti. In Fig. 5.2 è riassunto il comportamento dell'orchestratore sotto forma di flowchart.

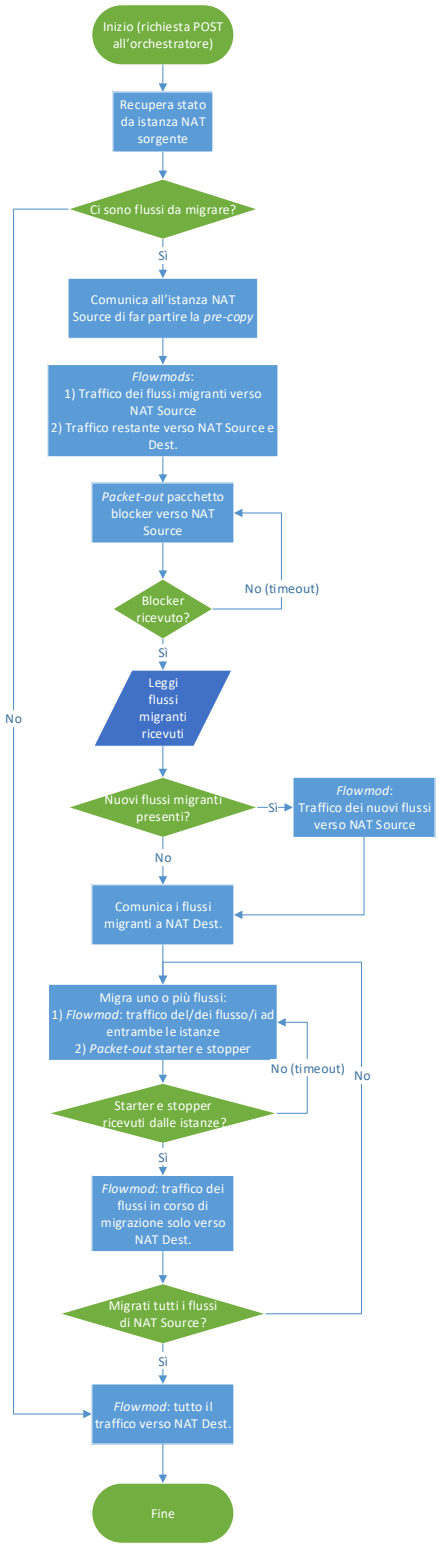


Figura 5.2. Flowchart dell'orchestratore della migrazione.

Capitolo 6

Validazione

Il presente capitolo presenta i test eseguiti al fine di validare l'algoritmo definito nel corso di questo lavoro di tesi (Sez. 4.1.1) ed il relativo prototipo (Sez. 5.2).

6.1 Banco di prova

I test sono stati condotti in un ambiente configurato come illustrato in Figura 6.1, composto da tre macchine:

1. macchina 1 che ha ospitato l'orchestratore, i clients, i due virtual switch OvS e il server:
   - CPU Intel i7-4770@3.4 GHz (8 core);
   - RAM 16 GB
   - due NIC 10Gbit a due porte
   - sistema operativo Ubuntu Linux 14.04 LTS
2. macchina 2 che ha ospitato il router e l'istanza NAT sorgente (entrambi in IOVnet):
   - CPU Intel i7-3770@3.4 GHz (8 core);
   - RAM 32 GB
   - una NIC 10Gbit a due porte
   - sistema operativo Ubuntu Linux 16.04 LTS
3. macchina 3 che ha ospitato il router e l'istanza NAT destinazione (entrambi in IOVnet):
   - CPU Intel i7-4770@3.4 GHz (8 core);
   - RAM 32 GB
   - una NIC 10Gbit a due porte
   - sistema operativo Ubuntu Linux 16.04 LTS

I collegamenti indicati in rosso nella Figura 6.1 sono relativi alla rete di controllo la quale ha permesso all'orchestratore e alle due istanze di NAT di poter dialogare; i collegamenti della rete di controllo sono stati fatti mediante cavi Ethernet categoria 5 (100 Mbps). I collegamenti della rete dati, raffigurati in nero, sono stati fatti mediante cavi Ethernet categoria 6A (10 Gbps). Si precisa, inoltre, che è stata sfruttata l'implementazione della VNF router già presente in IOVnet per i router illustrati nella Fig. 6.1.

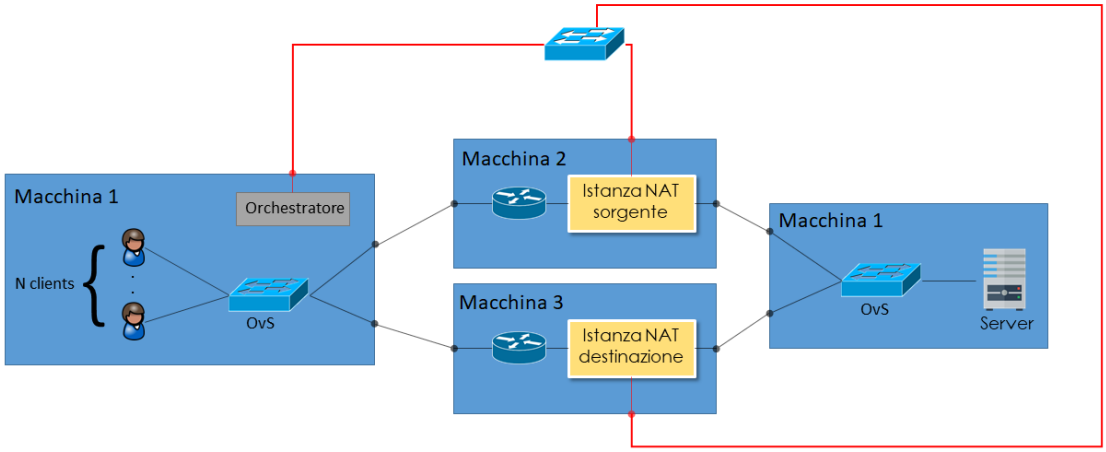


Figura 6.1. Configurazione dell'ambiente di validazione.

6.2 Test e risultati

In questa sezione verranno illustrati i test che sono stati eseguiti e verranno commentati i risultati ottenuti. Mediante il tool *Iperf* [37] si è simulato traffico TCP sulla istanza NAT sorgente proveniente da un numero prefissato di client situati in namespace diversi sulla macchina 1 (Fig. 6.1) e diretto verso il server. In particolare, i client sono stati ripartiti in maniera uguale tra i diversi namespace: ad esempio, nel caso di 5 clients e 3 namespace, sono stati eseguiti 2 client *Iperf* sul primo e secondo namespace mentre un client *Iperf* sul terzo namespace. Al variare di alcune condizioni che saranno illustrate nelle prossime sezioni sono stati misurati in particolare:

- il tempo di migrazione (Sez. 6.2.1);
- il throughput medio di ogni flusso migrante (Sez. 6.2.2);
- il packet-loss rate e i pacchetti out-of-order: uno degli obiettivi dell'algoritmo oggetto del presente lavoro di tesi, infatti, è stato quello di effettuare la migrazione con le proprietà di mancanza di packet-loss e di reordering dei pacchetti. Attraverso il tool *Iperf* [37] è stato verificato sperimentalmente che questo obiettivo è stato raggiunto.

Si precisa che i test sono stati effettuati 20 volte e ne verranno rappresentati ora i valori medi rilevati.

6.2.1 Il tempo di migrazione

Una delle variabili da tenere in considerazione quando si compie una operazione di migrazione è il tempo impiegato per la migrazione stessa. Test atti a misurarlo sono di notevole importanza nella presente tesi poiché uno degli obiettivi principali dell'algoritmo è stato proprio quello di impedire il prolungamento indefinito del tempo di migrazione dovuto alle problematiche affrontate nel Capitolo 3. Il tempo di migrazione è inteso come tempo necessario per migrare tutto lo stato e tutto il traffico tra le due istanze affinchè l'istanza destinazione possa processare tutto il traffico che prima attraversava l'istanza sorgente e anche il nuovo traffico. È stato misurato il tempo che è intercorso tra l'avvio dell'operazione di migrazione stessa (mediante richiesta POST all'orchestratore stesso) e gli ultimi messaggi di acknowledgment ricevuti dalle istanze Src e Dst, atte a confermare che la migrazione di tutti i flussi fosse stata completata.

Le misure sono state effettuate variando alcune condizioni, in particolare:

- abilitando/disabilitando l'operazione di pre-copy;
- migrando un flusso alla volta o più flussi in parallelo (in particolare tutti i flussi con uno stesso indirizzo IP sorgente);
- variando il numero di flussi migranti dall'istanza sorgente a quella destinazione da un minimo di un flusso fino ad un massimo di cento flussi.

Risultati

In Fig. 6.2 sono mostrati i tempi di migrazione al variare del numero di flussi TCP migranti, ottenuti migrando un flusso TCP alla volta, con e senza fase di pre-copy. Come prevedibile, si osserva una crescita del tempo di migrazione al crescere del numero di flussi TCP migranti: si passa da un minimo di 77ms per la migrazione di un solo flusso TCP sino ad un massimo di 852ms per la migrazione di 100 flussi TCP. Si noti inoltre che la migrazione di due flussi TCP dura meno del doppio della migrazione di un singolo flusso TCP (rispettivamente 93ms e 77ms). Questo è dovuto al fatto che la migrazione è composta dapprima di una fase di *pre-copy* e da una successiva fase di *stop-and-copy*. Il tempo della fase di pre-copy è fondamentalmente influenzato dalle condizioni attuali della sola rete di controllo. Essa è infatti composta:

- dai tempi della richiesta e della risposta, rispettivamente del migration orchestrator e di Src, circa i flussi gestiti correntemente da Src;
- dal tempo necessario per effettuare la copia dello stato da Src a Dst.

Il tempo della fase di *stop-and-copy*, invece, dipende fortemente dalle condizioni attuali della rete dati, essendo regolata dai pacchetti *starter* e *stopper* che viaggiano insieme ai normali pacchetti dati. Si può pensare, quindi, che la migrazione sia composta da due tempi: uno, pressochè costante, rappresentato dalla fase di *pre-copy* e uno, variabile in

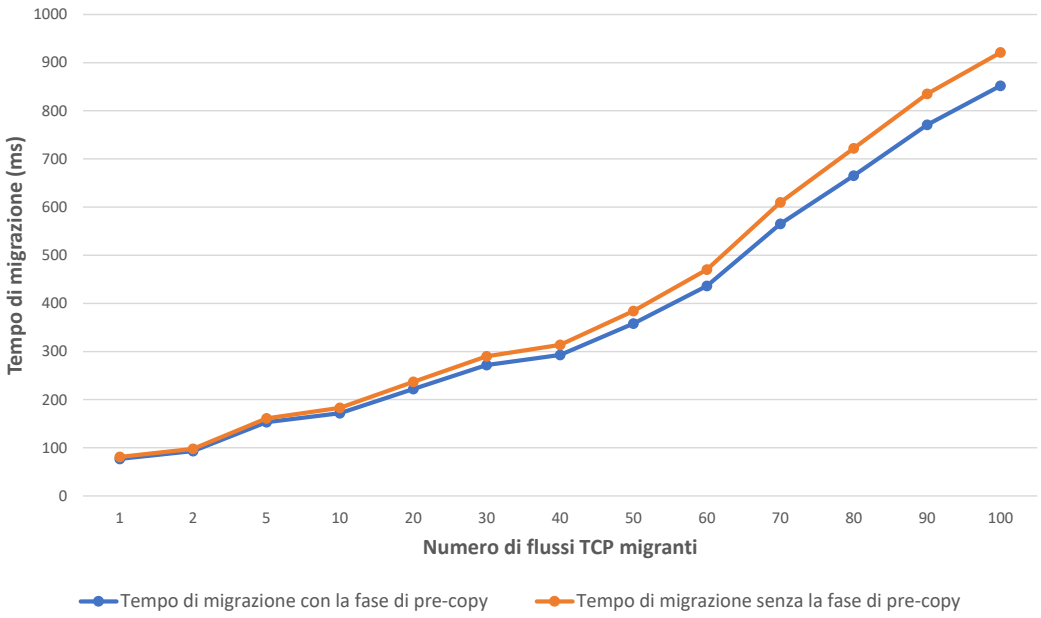


Figura 6.2. Confronto tra tempi di migrazione al variare del numero di flussi TCP migranti con e senza fase di pre-copy.

base al numero di flussi TCP migranti e alle condizioni della rete dati, rappresentato dalla fase di *stop-and-copy*. Al crescere dei flussi TCP migranti aumenta anche la differenza tra i tempi di migrazione con e senza fase di pre-copy: mentre migrando un solo flusso TCP la differenza percentuale è quasi il 5%, migrando 100 flussi TCP la differenza percentuale tra i due tempi è di circa il 7,5%.

In Fig. 6.3 sono mostrati, invece, i tempi di migrazione al variare del numero di flussi TCP migranti, ottenuti migrando un flusso TCP alla volta o multipli flussi TCP contemporaneamente (tutti i flussi con stesso indirizzo IP sorgente). In particolare, nel secondo caso, sono stati migrati contemporaneamente tutti i flussi con lo stesso indirizzo IP sorgente. Si osserva, analogamente a quanto detto in precedenza per gli andamenti in Fig. 6.2, anche nel caso di migrazione contemporanea di multipli flussi TCP, un aumento del tempo di migrazione al crescere del numero di flussi TCP migranti. Confrontando i due andamenti in Fig. 6.3 si nota inoltre come la migrazione di più flussi in parallelo comporta un notevole risparmio in termini di tempo: in media, infatti, si risparmia circa il 30% di tempo rispetto alla migrazione di un singolo flusso per volta. Si potrebbe dunque pensare alla migrazione parallela di tutti i flussi, avvicinandosi alla soluzione VM-like presentata nel Capitolo 3. Ricordiamo però che queste ultime soluzioni provocano un periodo di indisponibilità (*downtime*) molto elevato nel quale tutti i pacchetti vengono fondamentalmente scartati, provocando anche la perdita di eventuali cambiamenti di stato sulla NF. Nella soluzione proposta, invece, sono garantite la proprietà di nessun packet-loss e la consistenza dello stato della NF Src durante tutta la migrazione.

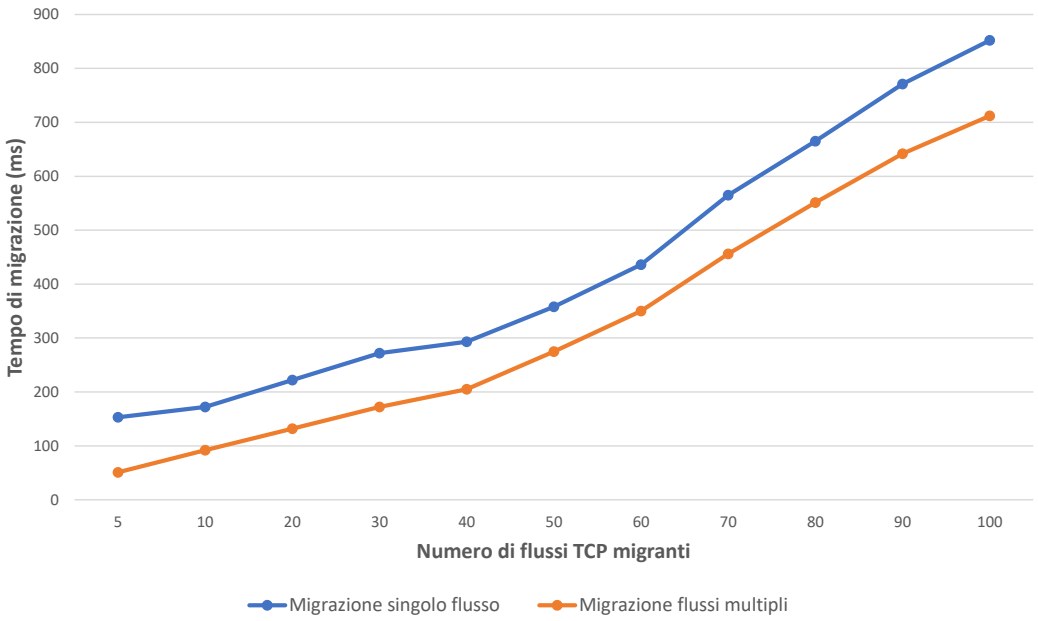


Figura 6.3. Confronto tra tempi di migrazione al variare del numero di flussi TCP migranti (migrazione singolo flusso e multipli flussi).

6.2.2 Throughput

La misurazione del throughput è stata effettuata mediante il tool *Iperf* [37] e l'intervallo di osservazione è stato di venti secondi. A metà dell'intervallo di osservazione, ovvero al secondo dieci, è stata fatta partire la migrazione.

Le misure sono state effettuate sia migrando un flusso alla volta sia più flussi in parallelo (in particolare tutti i flussi con uno stesso indirizzo IP sorgente) e variando il numero di flussi migranti.

Risultati

In Fig. 6.4 sono mostrati i valori del throughput medio per flusso in Mbps al variare del numero di flussi TCP migranti, ottenuti migrando un flusso TCP alla volta, anche con l'ausilio della fase di pre-copy. Si osserva una decrescita del throughput medio per flusso al crescere del numero di flussi TCP migranti: si passa da un massimo di 4380 Mbps per la migrazione di un solo flusso ad un minimo di 75Mbps per la migrazione di cento flussi TCP.

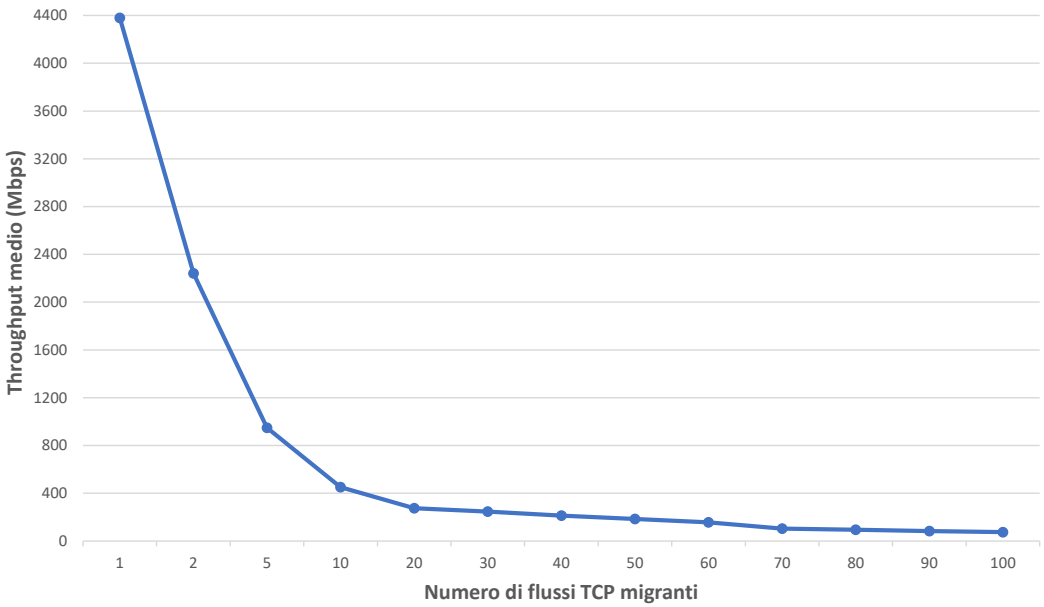


Figura 6.4. Throughput medio per flusso al variare del numero di flussi TCP migranti (migrazione singolo flusso)

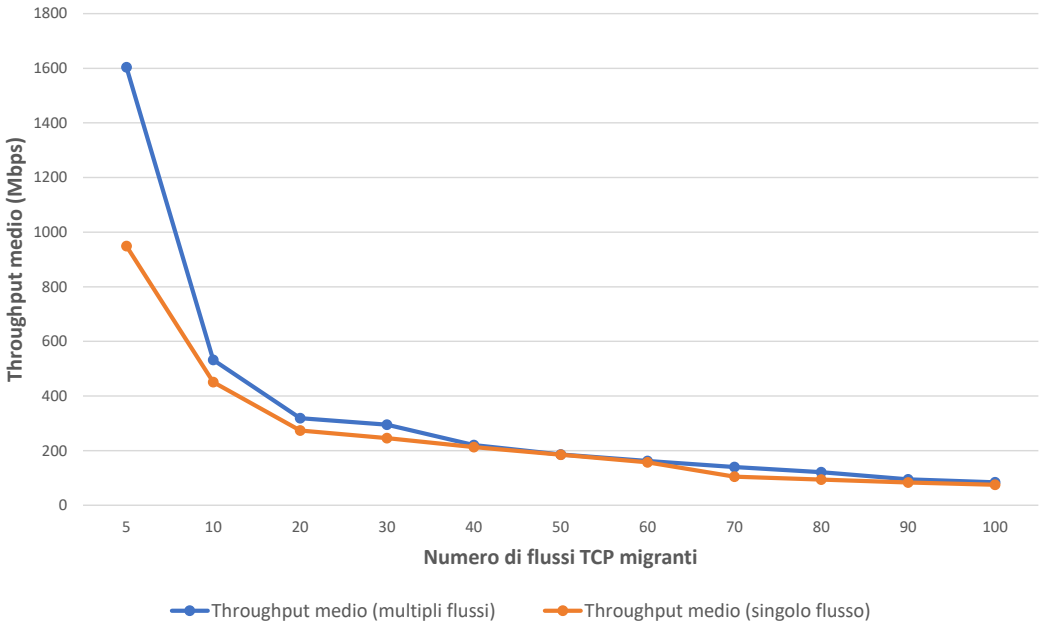


Figura 6.5. Confronto tra i valori di throughput medio al variare del numero di flussi TCP migranti (migrazione singolo flusso e multipli flussi)

In Fig. 6.5 sono messi a confronto i valori del throughput medio in Mbps al variare del numero di flussi TCP migranti, ottenuti migrando un flusso TCP alla volta o multipli flussi TCP contemporaneamente (tutti i flussi con stesso indirizzo IP sorgente). Si può osservare la permanenza dell'andamento decrescente della curva anche nel caso di migrazione di multipli flussi TCP. La migrazione di multipli flussi TCP in contemporanea, inoltre, migliora notevolmente il throughput medio: in media il throughput cresce, infatti, di circa il 15% rispetto alla migrazione singolo flusso.

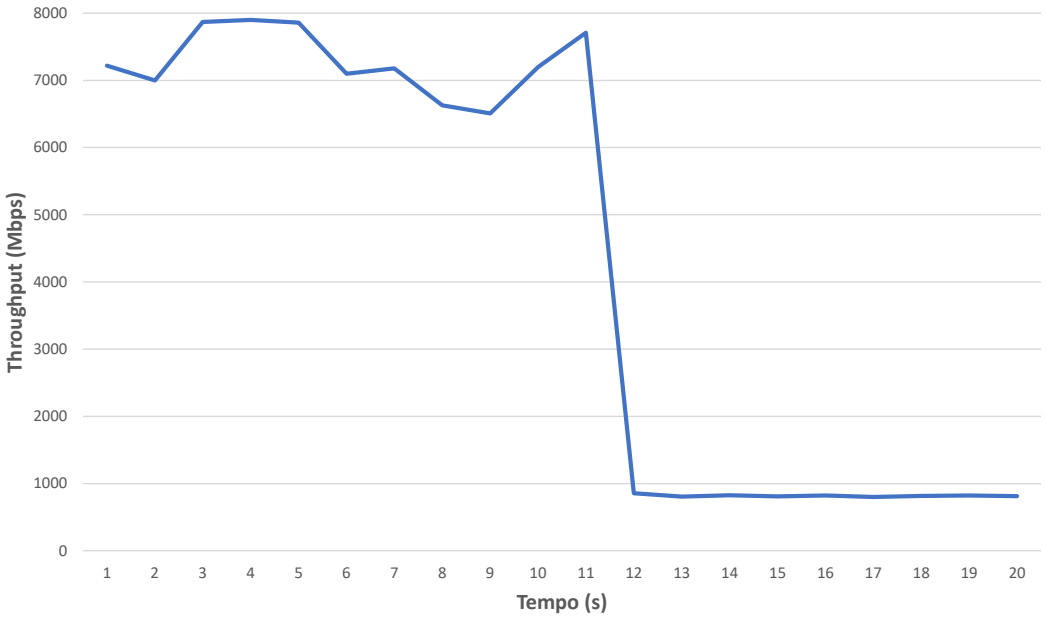


Figura 6.6. Andamento del throughput nel tempo durante la migrazione di un flusso singolo.

In Fig. 6.6 sono mostrati i valori del throughput istantaneo in Mbps durante la migrazione di un singolo flusso TCP. Si osservi come al partire della migrazione (a 10s) i valori di throughput decrescano in maniera considerevole ma rimangano pressochè costanti negli istanti di tempo successivi. È doveroso precisare e tener conto a tal proposito che le prestazioni della VNF NAT in relazione al throughput sono peggiorate dopo la migrazione a causa del fatto che il processing dei pacchetti avviene ancora nel kernel (datapath) ma solo dopo aver effettuato l'accodamento e il rilascio dei pacchetti stessi nello slowpath. Dati i risultati mostrati e le osservazioni fatte si può affermare inoltre che se si implementasse un buffer nel kernel in futuro, l'andamento del throughput nel tempo potrebbe essere quasi costante nel tempo durante la migrazione.

Capitolo 7

Conclusioni e sviluppi futuri

Nel corso di questa tesi è stato proposto un nuovo algoritmo per la migrazione di funzioni di rete virtualizzate con stato tra domini tecnologici eterogenei.

È stato sottolineato come nonostante ci siano numerose soluzioni proposte dallo stato dell'arte per la migrazione di VNF con stato, ognuna di esse presenta delle limitazioni, tra cui la più importante è il tempo di migrazione potenzialmente infinito. Le strategie proposte dall'algoritmo di questa tesi mirano a risolvere le suddette limitazioni, senza dover per questo sacrificare le proprietà fondamentali che esse garantiscono (tra cui la mancanza di perdita di pacchetti e di reordering).

È stato sviluppato inoltre un prototipo che ha avuto lo scopo principale di verificare se l'algoritmo proposto avesse raggiunto gli obiettivi prefissati. I risultati ottenuti nella validazione sono sicuramente positivi e incoraggiano l'approfondimento delle potenzialità dell'algoritmo. Il prototipo sviluppato, infatti, rappresenta solo una semplificazione di quello che si potrebbe considerare una implementazione definitiva.

A partire dal lavoro svolto, un importante punto da cui partire per gli sviluppi futuri è sicuramente l'implementazione di buffer residenti in kernel. Attraverso quest'ultimo, infatti, si potranno intercettare (ad esempio, attraverso i diversi punti delle catene di Netfilter [38]) i pacchetti ed accodarli ancor prima del loro arrivo alle VNF. Così facendo la migrazione sarà più efficiente. Essa permetterà, infatti, throughput più elevati dovuti al fatto che non si accoderanno più pacchetti in userspace e si potranno processare direttamente nel datapath. Un altro vantaggio che si otterrà da questa miglioria sarà la riduzione del carico e delle modifiche da effettuare sulle istanze di VNF, che non dovranno più preoccuparsi di accodare pacchetti e che quindi non rappresenteranno più un potenziale collo di bottiglia.

Altri aspetti da considerare potrebbero essere, inoltre: un metodo diverso per la gestione della perdita dei pacchetti *blocker*, *starter* e *stopper*; la scelta della strategia più efficiente per parallelizzare la migrazione dei flussi; il miglioramento della scalabilità dell'orchestratore, in relazione al numero di migrazioni parallele in atto da parte di diverse VNF.

Appendice A

Lo studio dei diversi tipi di stato

Dovendo migrare lo stato delle funzioni di rete, ne è stato studiato quello relativo alle funzioni di rete più comuni al fine di cercare la strategia più adatta da adottare per la migrazione. Lo stato è stato classificato suddividendolo in diverse categorie:

- stato di tipo *statico* o *dinamico*: specifica se lo stato cambia nel corso del tempo;
- stato che viene *letto/modificato* da flussi in corso di migrazione e/o da nuovi flussi;
- stato di tipo *per-flow*, *multi-flow*, *all-flow* rispettivamente stato relativo ad un singolo flusso, a più flussi e a tutti i flussi;
- stato che necessità di essere migrato sull'istanza destinazione prima di poter effettuare il processing di nuovi flussi;
- stato che può cambiare mentre si sta processando flussi migranti;
- stato che necessita o non necessita di operazioni di merging dopo la migrazione.

Si noti che le ultime tre classificazioni possono essere derivate dall'informazione circa la possibile lettura/modifica dello stato in esame da parte di flussi in corso di migrazione e/o da nuovi flussi. Nella Tabella A.1 sono riassunte le informazioni appena descritte.

Tabella A.1. Classificazione dello stato delle funzioni di rete più comuni.

Funzione di rete	Stato	Statico/Dinamico	Flussi da migrare		Nuovi flussi		Per-flow/multi-flow/not-flow	Necessità di migrare questo stato prima di poter processare nuovi flussi	Lo stato può cambiare mentre si sta processando flussi da migrare	Stato necessita di merging dopo la migrazione
			Read	Write	Read	Write				
NAT	Mappings (i.e. traduzioni IP/porte)	Dinamico[3]	x	x	x	x	Per-flow[2,3]	No	Sì	No[2]
	Configurazione (interfacce pubbliche/private)	Statico[1]	x		x		Not-flow	Sì	No	/
	Pool di indirizzi IP e porte disponibili per la traduzione	Dinamico			x	x	Multi-flow	Sì	Sì	Sì[5]
Firewall	Configurazione (politiche di sicurezza)	Statico	x		x		Multi-flow	Sì	No	/
	Records di connessione (e.g. ricostruzione di messaggi L7)	Dinamico		x		x	Per-flow	No	Sì	No
	Configurazione (interfacce)	Statico	x		x		Not-flow	Sì	No	/
Load balancer	Backend servers IPs	Statico	x		x		Not-flow	Sì	No	/
	Mappings (i.e. server assegnato al flusso in esame)	Dinamico	x	x[4]	x	x	Per-flow	No	Yes	No
	Configurazione (interfacce e algoritmo di load balancing)	Statico	x		x		Not-flow	Sì	No	/
	Dati relativi al carico sui diversi server di backend (necessari all'algoritmo di backend)	Dinamico	x	x	x	x	Not-flow	In alcuni casi sì	Sì	Sì
NIDS (transparent)	Configurazione (politiche di sicurezza)	Statico	x		x		Multi-flow	Sì	No	/
	"Livello di allerta" (fino a quale livello di profondità la funzione di rete deve analizzare i pacchetti in questo momento)	Dinamico	x	x	x	x	Multi-flow & Per-flow	Sì/No (dipende dalla tupla IP)	Sì	Sì/No (dipende dalla tupla IP)
	Logs	Dinamico		x		x	Multi-flow/Per-flow	No	Sì	Sì (append)
	Records di connessione (e.g. (tupla IP; Stato (**anche contatori**); Seq#; Payloads))	Dinamico	x	x	x	x	Multi-flow & Per-flow	Sì/No (dipende dalla tupla IP)	Sì	Sì/No (dipende dalla tupla IP)
Traffic Shaper	Configurazione (interfacce)	Statico	x		x		Not-flow	Sì	No	/
	Limiti di banda (i.e. (Indirizzo IP Source; Limite)	Statico	x		x		Per-flow/Multi-flow se l'indirizzo IP è un indirizzo di una rete	No/Si se l'indirizzo IP è un indirizzo di una rete	No	No

Funzione di rete	Stato	Statico/Dinamico	Flussi da migrare		Nuovi flussi		Per-flow/multi-flow/not-flow	Necessità di migrare questo stato prima di poter processare nuovi flussi	Lo stato può cambiare mentre si sta processando flussi da migrare	Stato necessita di merging dopo la migrazione
			Read	Write	Read	Write				
DHCP Server	Pool di indirizzi IP disponibili per l'assegnazione	Dinamico			x	x	Not-flow	Sì	Sì (entry timeout)	/
	Mappings (i.e. configurazione assegnata ad un client {MAC addr.; IP addr. assegnato; Timer})	Dinamico	x	x	x	x	Per-flow	No	Sì (entry timeout)	No
	Configurazione (Interfacce + range di indirizzi IP + Def. Gateway + DNS Servers)	Statico	x		x		Not-flow	Sì	No	/
VPN Gateway	Configurazione (Interfacce + range di indirizzi IP)	Statico	x		x		Not-flow	Sì	No	/
	Pool di indirizzi IP disponibili per l'assegnazione	Dinamico			x	x	Not-flow	Sì	Sì (rinegoziazione parametri di sicurezza)	/
	Mappings (i.e. configurazione assegnata + config. di sicurezza opzionale)	Dinamico	x	x	x	x	Per-flow	No	Sì	No

Osservazioni inerenti alla tabella:

1. Stiamo ipotizzando che la configurazione statica non cambi durante la migrazione;
2. si noti che di solito se uno stato è classificato come *per-flow* non serve l'operazione di merge dopo il completamento della migration;
3. la gestione della migrazione degli stati di tipo *per-flow* e *dinamico* sarà discussa nella sezione 4.1.1;
4. se cambia il carico sui server di backend, il load balancer potrebbe spostare il flusso da un backend server ad un altro;
5. questo tipo di stato è particolare: potremmo scegliere una politica di soft-consistency. Potremmo infatti migrare il valore attuale di questo stato, continuare ad aggiornarlo sia sull'istanza sorgente che su quella destinazione e poi fare il merge alla fine della migration senza avere problemi di consistenza. Per maggiori dettagli, si rimanda alla sezione 4.1.2.

Appendice B

NAT YANG data model

```
module nat {
    yang-version 1.1;
    namespace "http://iovisor.org/iovnet/nat";
    prefix "nat";

    import base-iovnet-service-model { prefix "basemodel"; }

    import ietf-inet-types { prefix "inet"; }

    import ietf-yang-types { prefix "yang"; }

    organization "Politecnico di Torino";
    description "Data model for the IOVisor NAT service";

    uses "basemodel:base-yang-module";

    leaf natrole {
        type enumeration {
            enum SOURCE;
            enum DESTINATION;
        }
        default SOURCE;
        description "Role of the NAT in a NAT migration
           process.";
    }

    list mappingentry {
        key "intsaddr intsport proto";
        description
        "NAT mapping entry.";
```

```
        leaf intsaddr {
            type inet:ipv4-address;
            description
            "Corresponds to the source IPv4 address
            of the IPv4 packet.";
        }

        leaf intsport {
            type inet:port-number;
            description
            "Corresponds to the source port of the
            IPv4 packet.";
        }

        leaf extsaddr {
            type inet:ipv4-address;
            description
            "External IPv4 address assigned by NAT.";
        }

        leaf extsport {
            type inet:port-number;
            description
            "External source port number assigned by NAT.";
        }

        leaf proto {
            type enumeration {
            enum ICMP;
            enum TCP;
            enum UDP;
            }
            description
            "Upper-layer protocol associated with this
               mapping.";
        }
    }

    list migratingflows {
        key "intsaddr intsport proto";
        description
        "TCP/UDP flows that are migrating from Source to
           Destination NAT";
```

```
        leaf intsaddr {
            type inet:ipv4-address;
            description
            "Corresponds to the source IPv4 address
            of the IPv4 packet.";
        }

        leaf intsport {
            type inet:port-number;
            description
            "Corresponds to the source port of the
            IPv4 packet.";
        }

        leaf extsport {
            type inet:port-number;
            description
            "External source port number assigned by NAT.";
        }

        leaf proto {
            type enumeration {
                enum ICMP;
                enum TCP;
                enum UDP;
            }
            description
            "Upper-layer protocol associated with this
               mapping.";
        }
    }

    container publicinterface {
        leaf ip {
            type inet:ipv4-address;
            description
            "Public IP address of the NAT.";
        }

        leaf port {
            type yang:uuid;
            description
            "Port the public interface is connected to.";
        }
```

```
    }

    container privateinterface {
        leaf port {
            type yang:uuid;
            description
            "Port the private interface is connected to.";
        }
    }
}
```

Bibliografia

[1] *Network functions virtualization: introductory white paper*. URL: https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[2] Vyas Sekar et al. "Design and Implementation of a Consolidated Middlebox Architecture". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 24–24. URL: http://dl.acm.org/citation.cfm?id=2228298.2228331.

[3] Michio Honda et al. "Is It Still Possible to Extend TCP?" In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*. IMC '11. Berlin, Germany: ACM, 2011, pp. 181–194. ISBN: 978-1-4503-1013-0. DOI: 10.1145/2068816.2068834. URL: http://doi.acm.org/10.1145/2068816.2068834.

[4] Justine Sherry et al. "Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service". In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '12. Helsinki, Finland: ACM, 2012, pp. 13–24. ISBN: 978-1-4503-1419-0. DOI: 10.1145/2342356.2342359. URL: http://doi.acm.org/10.1145/2342356.2342359.

[5] *The NetGroup at Politecnico di Torino*. URL: http://netgroup.polito.it/.

[6] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74. DOI: 10.1145/1355734.1355746.

[7] Zafar Ayyub Qazi et al. "SIMPLE-fying middlebox policy enforcement using SDN". In: *ACM SIGCOMM computer communication review*. Vol. 43. 4. ACM. 2013, pp. 27–38. DOI: 10.1145/2486001.2486022.

[8] Dilip A Joseph, Arsalan Tavakoli e Ion Stoica. "A policy-aware switching layer for data centers". In: *ACM SIGCOMM Computer Communication Review*. Vol. 38. 4. ACM. 2008, pp. 51–62. DOI: 10.1145/1402958.1402966.

[9] Aaron Gember et al. *Stratos: A network-aware orchestration layer for middleboxes in the cloud*. Rapp. tecn. Technical Report, 2013.

[10] Seyed Kaveh Fayazbakhsh et al. "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags." In: *NSDI*. Vol. 14. 2014, pp. 533–546. DOI: 10.1145/2491185.2491203.

[11] Bilal Anwer et al. “A slick control plane for network middleboxes”. In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 147–148. DOI: 10.1145/2491185.2491223.

[12] Paul Barham et al. “Xen and the art of virtualization”. In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 164–177. DOI: 10.1145/1165389.945462.

[13] Umar Farooq Minhas et al. “RemusDB: Transparent High Availability for Database Systems”. In: *The VLDB Journal* 22.1 (feb. 2013), pp. 29–45. ISSN: 1066-8888. DOI: 10.1007/s00778-012-0294-6. URL: http://dx.doi.org/10.1007/s00778-012-0294-6.

[14] Brendan Cully et al. “Remus: High Availability via Asynchronous Virtual Machine Replication”. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI’08. San Francisco, California: USENIX Association, 2008, pp. 161–174. ISBN: 111-999-5555-22-1. URL: http://dl.acm.org/citation.cfm?id=1387589.1387601.

[15] Vladimir Andrei Olteanu e Costin Raiciu. “Efficiently Migrating Stateful Middleboxes”. In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’12. Helsinki, Finland: ACM, 2012, pp. 93–94. ISBN: 978-1-4503-1419-0. DOI: 10.1145/2342356.2342376. URL: http://doi.acm.org/10.1145/2342356.2342376.

[16] Vladimir Andrei Olteanu, Felipe Huici e Costin Raiciu. “Lost in Network Address Translation: Lessons from Scaling the World’s Simplest Middlebox”. In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox ’15. London, United Kingdom: ACM, 2015, pp. 19–24. ISBN: 978-1-4503-3540-9. DOI: 10.1145/2785989.2785994. URL: http://doi.acm.org/10.1145/2785989.2785994.

[17] Aaron Gember-Jacobson et al. “OpenNF: Enabling innovation in network function control”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 163–174. DOI: 10.1145/1165389.945462.

[18] Babu Kothandaraman, Manxing Du e Pontus Sköldström. “Centrally Controlled Distributed VNF State Management”. In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox ’15. London, United Kingdom: ACM, 2015, pp. 37–42. ISBN: 978-1-4503-3540-9. DOI: 10.1145/2785989.2785996. URL: http://doi.acm.org/10.1145/2785989.2785996.

[19] Aaron Gember-Jacobson e Aditya 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*. HotMiddlebox ’15. London, United Kingdom: ACM, 2015, pp. 43–48. ISBN: 978-1-4503-3540-9. DOI: 10.1145/2785989.2785997. URL: http://doi.acm.org/10.1145/2785989.2785997.

[20] Libin Liu et al. “U-HAUL: Efficient State Migration in NFV”. In: *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys ’16. Hong Kong, Hong Kong: ACM, 2016, 2:1–2:8. ISBN: 978-1-4503-4265-0. DOI: 10.1145/2967360.2967363. URL: http://doi.acm.org/10.1145/2967360.2967363.

[21] Shriram Rajagopalan, Dan Williams e Hani Jamjoom. “Pico Replication: A High Availability Framework for Middleboxes”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 2013, 1:1–1:15. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523635. URL: http://doi.acm.org/10.1145/2523616.2523635.

[22] Murad Kablan et al. “Stateless Network Functions: Breaking the Tight Coupling of State and Processing”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 97–112. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan.

[23] M. Peuster e H. Karl. “E-State: Distributed state management in elastic network function deployments”. In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. 2016, pp. 6–10. DOI: 10.1109/NETSOFT.2016.7502432.

[24] Murad Kablan et al. “Stateless Network Functions”. In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox ’15. London, United Kingdom: ACM, 2015, pp. 49–54. ISBN: 978-1-4503-3540-9. DOI: 10.1145/2785989.2785993. URL: http://doi.acm.org/10.1145/2785989.2785993.

[25] John Ousterhout et al. “The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM”. In: *SIGOPS Oper. Syst. Rev.* 43.4 (gen. 2010), pp. 92–105. ISSN: 0163-5980. DOI: 10.1145/1713254.1713276. URL: http://doi.acm.org/10.1145/1713254.1713276.

[26] *Introduction to Infiniband*. URL: https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

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

[28] IETF. *RFC 791: Internet Protocol Specification*. URL: https://tools.ietf.org/html/rfc791.

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

[30] *The IOVisor project*. URL: https://www.iovisor.org.

[31] Steven McCanne e Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture”. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX’93. San Diego, California: USENIX Association, 1993, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=1267303.1267305.

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

[33] *The Open Networking Foundation*. URL: https://www.opennetworking.org.

[34] *The ONOS Project*. URL: https://onosproject.org/.

[35] *Open vSwitch*. URL: https://www.openvswitch.org/.

[36] *Apache Karaf*. URL: https://karaf.apache.org/.

[37] Jon Dugan et al. *iPerf : the ultimate speed test tool for TCP, UDP and SCTP*. URL: https://iperf.fr.

[38] *The netfilter.org project*. URL: https://www.netfilter.org/.