ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA
in
Mobile Systems M

Multi-layer Routing in Reti Spontanee basato su Software-Defined Networking

Relatore:
Chiar.mo Prof. Ing.
Paolo Bellavista

Presentata da:
Dmitrij David Padalino Montenero

Correlatore:
Dott. Ing.
Carlo Giannelli

Sessione I
Anno Accademico 2018/2019

Parole chiave: *Mobile Systems*
Spontaneous Networking
Software-defined Networking
Routing
Traffic Engineering

Indice

Introduzione

Negli ultimi anni si è assistito ad un incremento vertiginoso della diffusione del numero di dispositivi mobili in tutti i settori dell'industria e del mercato consumer. Il conseguente alto tasso di adozione di tali dispositivi, caratterizzati dalla presenza di multiple interfacce di comunicazione basate su tecnologie wireless eterogenee, ha creato un forte interesse nella comunità scientifica soprattutto per quanto riguarda l'utilizzo efficiente di queste interfacce per consentire loro di comunicare direttamente senza la presenza di una infrastruttura di supporto. Queste motivazioni hanno portato alla nascita di un campo di ricerca noto al giorno d'oggi come Mobile Ad-Hoc Networking (MANET). Tra le varie tipologie di rete coperte da questo campo esistono le cosiddette reti spontanee, caratterizzate da un numero limitato di dispositivi presenti nella stessa località atti a interagire socialmente per consentire il rapido scambio di qualunque tipo di informazione.

In un ambiente come questo, dove non è prevista la presenza di un'infrastruttura a supporto della comunicazione, sono i dispositivi stessi che tramite l'utilizzo di protocolli distribuiti, basati sullo scambio di informazioni locali, devono essere in grado di coordinarsi al fine di comunicare gli uni con gli altri. L'impiego di tali protocolli può diventare un limite in caso si vogliano adottare delle tecniche di routing più sofisticate, rispettare dei requisiti di Quality of Service oppure introdurre delle politiche di Traffic Engineering. A tal scopo può essere utile prendere ispirazione da quanto fatto nello sviluppo di altre tecnologie per introdurre un nuovo paradigma di gestione delle reti spontanee. Una possibile soluzione è rappresentata dall'utilizzo dei principi alla base del Software-Defined Networking (SDN), utilizzato ampiamente nella gestione dei data center, il quale grazie alla presenza di un'intelligenza centralizzata può ampliare considerevolmente le capacità che una rete spontanea è in grado di supportare.

Lo scopo di questo lavoro consiste nell'impiegare il paradigma SDN nell'ambito delle reti spontanee per realizzare una funzionalità di basso livello per consentire l'instradamento dei messaggi scambiati tra i dispositivi mobili attraverso il policy-based routing e un'altra di alto livello per permettere l'attivazione e l'applicazione di regole di Traffic Engineering per una manipolazione avanzata in tempo reale dei pacchetti in transito in fase di comunicazione.

La trattazione che segue si articola in quattro parti fondamentali. Nel Capitolo 1 viene fatta un'introduzione sullo stato dell'arte delle tecniche di routing, descrivendone il funzionamento e gli scenari applicativi in cui vengono utilizzate. Il Capitolo 2 è dedicato alla descrizione dei principi alla base del paradigma SDN, mostrandone le motivazioni che hanno portato al suo sviluppo. Verrà fornita inoltre una descrizione generale del protocollo OpenFlow, considerato l'implementazione più famosa di SDN. Nel Capitolo 3 si passerà alla descrizione del middleware RAMP, che ha rappresentato il punto di partenza per dell'attività progettuale, illustrandone l'architettura generale, gli scopi e le estensioni che sono state aggiunte nel corso del suo sviluppo. Infine nel Capitolo 4 verrà presentata con un approccio top-down l'attività progettuale svolta, si illustrerà dapprima in modo generale lo schema che ha portato alla realizzazione delle funzionalità citate, si passerà poi alla descrizione dettagliata dell'implementazione per poi mostrare i risultati sperimentali ottenuti in fase di testing.

Capitolo 1
Introduzione al Routing

Nel campo delle reti di calcolatori il routing, o instradamento, è quel processo che si occupa di trasferire un messaggio da un nodo sorgente ad uno di destinazione appartenenti nel caso più semplice alla stessa rete o in quello più comune a due reti distinte per località. Questo processo è di solito implementato, eseguito e gestito da un dispositivo dedicato chiamato router. Il routing rappresenta un fattore chiave nel mondo di Internet perché consente a due nodi A e B, non collegati direttamente, di comunicare tra loro mediante la collaborazione di altri nodi posti su un cammino nella rete che connette A e B [1].

Ogni nodo tra sorgente e destinazione, chiamato intermediario, attua l'instradamento inoltrando il messaggio al nodo successivo. Parte di questo processo implica la consultazione di una struttura dati locale al nodo, chiamata tabella di routing, per determinare il percorso migliore.

Avendo come riferimento il modello ISO/OSI, il routing è un concetto affrontato dal Livello 3 detto Livello di Rete. In tale ambito, quando ci si riferisce ai messaggi scambiati, si utilizza il termine pacchetto.

Essendo l'instradamento una problematica orizzontale nel campo delle reti questo capitolo introdurrà come è stato affrontato in diversi scenari applicativi. In particolare ci si soffermerà innanzitutto su alcuni modelli logici introduttivi per poi trattare i protocolli e i paradigmi adottati nelle reti IP tradizionali. Infine verranno descritte brevemente alcune soluzioni di routing adottate in reti di MANET (Mobile Ad-Hoc Networks), composte da nodi mobili che comunicano tra loro utilizzando interfacce wireless.

1.1 Piani di networking

Per comprendere i meccanismi utilizzati per consentire lo scambio di informazioni tra nodi attraverso la rete è utile introdurre un modello di valenza generale.

Un dispositivo di rete, come un router, svolge diverse mansioni quando è in funzione. È possibile separare logicamente queste mansioni in tre aree operative, chiamate piani di networking. Come illustrato in Figura 1, essi sono:

- Il management plane o piano di gestione.
- Il control plane o piano di controllo.
- Il data plane o piano dei dati.

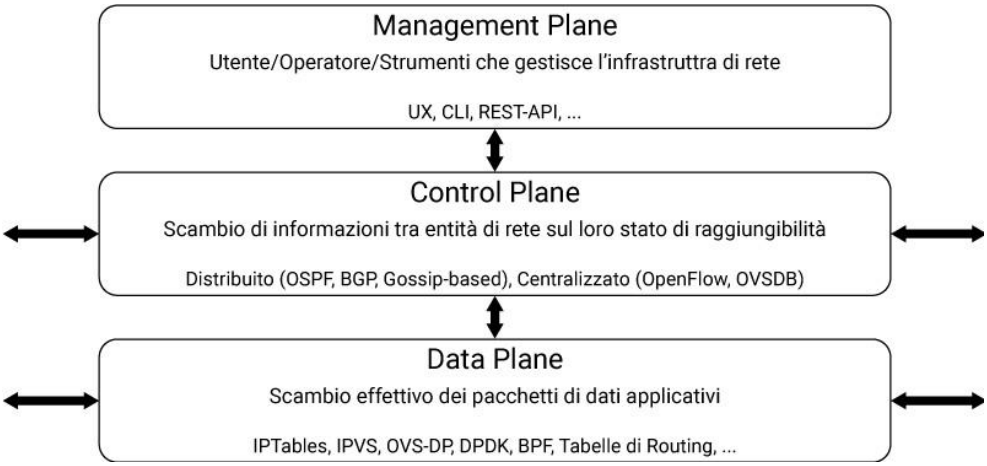

Figura 1 - Piani di networking.

Il management plane di un dispositivo di rete si occupa dell'aspetto amministrativo dell'infrastruttura, fornendo i servizi di gestione, monitoraggio e configurazione a tutti i livelli dello stack di rete e ad altre parti del sistema.

Il control plane si occupa di gestire il traffico dei pacchetti di controllo tra i dispositivi di rete ed è il livello responsabile delle funzioni di routing. I protocolli del control plane permettono ad un dispositivo di rete di crearsi una vista della topologia di rete e di popolare le tabelle di routing per instradare correttamente i pacchetti dati in entrata al nodo successivo.

Il data plane inoltra il traffico al nodo successivo selezionando il percorso in accordo a quanto definito dal control plane.

Solitamente nelle reti IP tradizionali control e data plane sono implementati nel firmware di router e switch. In altre tipologie di rete non è detto che siano implementati nello stesso dispositivo di rete; nelle software-defined network trattate nel Capitolo 2, ad esempio, risultano disaccoppiati.

1.2 Routing nelle reti IP

Nelle reti IP esistono diversi paradigmi per attuare l'instradamento dei pacchetti. Di seguito si descriverà lo stato dell'arte sui principali approcci utilizzati partendo da soluzioni tradizionali come il routing basato sulla

destinazione e il source routing, per poi passare a soluzioni più recenti come il policy-based routing, MPLS e il segment routing.

1.2.1 Destination-based routing

Il destination-based routing è la tipologia di routing storicamente più utilizzata. L'idea di base è molto semplice: ogni volta che si intende inviare un messaggio ad un nodo destinazione, il nodo sorgente inserisce nell'header del pacchetto, contenente il messaggio, l'indirizzo IP di destinazione. Le decisioni per il corretto instradamento prese dai router intermedi saranno basate esclusivamente sull'indirizzo di destinazione.

In questo tipo routing un insieme di protocolli determinano il percorso che un pacchetto dovrà seguire pe raggiungere la destinazione, percorso che comporterà l'attraversamento di un certo numero di router appartenenti a reti diverse.

Alcuni di questi protocolli che operano a livello di control plane sono:

- Border Gateway Protocol (BGP) [2].
- Intermediate System - Intermediate System (IS-IS) [3].
- Open Shortest Path First (OSPF) [4].
- Routing Information Protocol (RIP) [5].

Nell'esempio seguente, illustrato in Figura 2, viene illustrato l'invio di un pacchetto da un host sorgente S ad un server destinazione D.

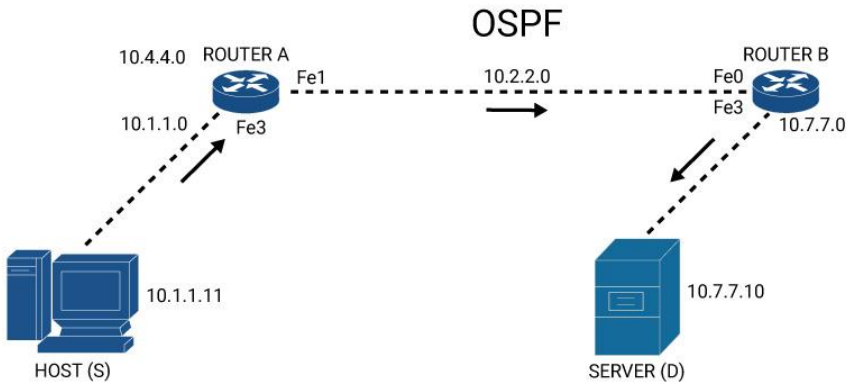

Figura 2 - Destination-based routing.

S e D appartengono a reti differenti e non sono connessi direttamente l'uno con l'altro. Il router A deve inoltrare il pacchetto al router B e il router B deve inoltrare il pacchetto al server D.

Come primo passo si assegnano degli indirizzi a tutte le interfacce presenti nei router e successivamente i router popolano una struttura dati chiamata routing table con queste informazioni. I router non conoscono tutte le informazioni riguardo agli indirizzi delle interfacce degli altri router, in questo esempio A non è a conoscenza dell'esistenza dell'indirizzo 10.7.7.0 assegnato all'interfaccia Fe3 del router B. A verrà a conoscenza degli indirizzi di B solo quando esso li dichiarerà; per farlo bisogna stabilire una comunicazione tra i due router. Questa comunicazione è facilitata dai protocolli di routing come RIP, OSPF, IS-IS. Per esempio il protocollo OSPF permette di far capire a due reti adiacenti di essere direttamente connesse e di scambiarsi il contenuto delle rispettive tabelle di routing. In questo protocollo i router A e B dichiarano gli indirizzi delle proprie interfacce e popolano le tabelle di routing. Ora A e B hanno conoscenza completa della topologia di rete ed S può inviare un messaggio al D.

Immaginando che l'indirizzo 10.7.7.10 di D sia noto, S invia il pacchetto al router A, il quale ispezionando il pacchetto scopre che l'indirizzo di destinazione è 10.7.7.0. Il router A quindi controlla la sua tabella di routing e capisce che la destinazione può essere raggiunta sfruttando l'interfaccia Fe1. A utilizza quell'interfaccia per inviare il pacchetto B. Anche il router B eseguirà le stesse operazioni condotte da A e scopre che il server può essere raggiunto utilizzando l'interfaccia Fe3, invia infine il pacchetto a destinazione utilizzando l'interfaccia trovata. Questo processo di identificazione dell'interfaccia che il pacchetto dovrà attraversare per raggiungere la destinazione è noto come *IP lookup*.

1.2.1.1 La tabella di routing

Come riportato nell'esempio precedente, lo strumento utilizzato dai router per capire a quale nodo successivo inoltrare un pacchetto in transito è la *tabella di routing*, nota anche come routing table.

La tabella di routing è una struttura dati presente in un router o in un computer connesso ad una rete che riporta la lista dei percorsi verso reti di destinazione, e in alcuni casi anche metriche associate a questi percorsi. Il popolamento delle tabelle di routing è l'obiettivo principale dei protocolli di routing. Le voci riportate all'interno della tabella sono pertanto il risultato di una qualche procedura di discovery da parte dei protocolli di routing. È possibile inoltre che la tabella contenga ulteriori voci aggiunte tramite inserimento manuale da parte dell'amministratore di rete, tali voci si dicono percorsi statici.

Network Destination	Netmask	Gateway	Interface
192.168.0.0	255.255.255.0	0.0.0.0	eth0

192.24.0.0	255.255.192.0	192.24.0.1	eth1
192.24.12.0	255.255.252.0	192.24.12.1	wlan0

Tabella 1 - Esempio di tabella di routing.

Una tabella di routing contiene almeno i seguenti campi:

- Le colonne **Network Destination** e **Netmask** (o prefisso) descrivono la sottorete di destinazione o un host. Queste due informazioni sono sempre lette insieme nel processo di IP lookup.
- La colonna **Gateway** indica l'indirizzo del nodo successivo per giungere alla rete di destinazione.
- La colonna **Interface** riporta quale interfaccia, disponibile localmente, deve essere impiegata per raggiungere il Gateway.

Per esempio considerando la prima riga della Tabella 1, 192.168.0.0 con Netmask 255.255.255.0 indica tutti gli indirizzi che iniziano con 192.168.0.X, cioè quelli appartenenti alla rete locale. Questa riga dice dunque che tutti i pacchetti aventi un indirizzo di destinazione appartenente alla rete locale non devono essere instradati (questo è il significato di 0.0.0.0 nel campo Gateway).

Quando un router IP riceve un pacchetto deve scoprire a quale rete il pacchetto deve essere inoltrato. Il router scansiona la tabella sequenzialmente, partendo dalla prima voce e procedendo verso il basso, per verificare che ci sia una corrispondenza prima di tutto con la maschera di rete più lunga e in caso di mancato riscontro cerca una nuova corrispondenza con una seconda maschera di rete più lunga. Il router può inoltrare il pacchetto solo se questa ricerca ha esito positivo.

Se l'indirizzo di destinazione coincide ai campi Network Destination e Netmask, il pacchetto viene inoltrato al Gateway usufruendo dell'interfaccia corrispettiva.

1.2.2 Source-based routing

Il *source-based routing*, noto anche come path addressing, è un processo specifico di instradamento dove il nodo sorgente può specificare, parzialmente o completamente, il percorso che un pacchetto di dati deve intraprendere attraverso la rete. Come approccio è un'alternativa al tradizionale instradamento basato sulla destinazione.

Esistono due diverse tipologie di source routing, chiamate rispettivamente *loose* and *strict*. Nel loose source routing il pacchetto deve passare attraverso una lista specifica di nodi intermediari sottoinsieme del percorso che il pacchetto seguirà, mentre nello strict source routing il nodo sorgente specifica ogni nodo intermediario del percorso.

Prendendo come riferimento IPv4, per utilizzare il source routing è necessario configurare opportunamente alcuni campi, detti opzioni, nell'header di un datagramma predisposti a tale funzione. In particolare per il loose source routing è necessario configurare le opzioni Loose Source e Record Route (LSRR) mentre per lo strict source routing le opzioni da configurare sono Strict Source e Record Route (SSRR).

1.2.2.1 Record Route

L'opzione Record Route consente di memorizzare il percorso di un datagramma IP. La struttura dell'opzione è la seguente:

00000111	length	pointer	route data
8 bit	8 bit	8 bit	lista di lunghezza variabile

L'opzione inizia con un campo di 8 bit che specifica il **codice identificativo** dell'opzione Record Route avente sempre valore binario 00000111, 7 in decimale. Il secondo ottetto è il campo **length** che riporta la lunghezza totale dell'opzione, l'unità di misura da utilizzare per questo valore è il byte. Il terzo ottetto chiamato **pointer** è il puntatore corrente al campo route data da cui iniziare a scrivere l'indirizzo successivo del percorso da memorizzare. Il puntatore è relativo a questa opzione e il minimo valore consentito per questo campo è 4.

Il campo **route data** contiene gli indirizzi IP del percorso, è composto da un vettore di lunghezza variabile contenente indirizzi IP. Ogni indirizzo IP è composto da 32 bit. Se il valore del campo pointer è maggiore del valore del campo length, vuol dire che l'area di memoria predisposta alla memorizzazione del percorso è piena. Di conseguenza è necessario che il nodo sorgente imposti opportunamente questa opzione in modo che l'area di memoria riesca a contenere tutti gli indirizzi IP previsti. La dimensione dell'opzione, indicata dal valore del campo length, non cambia dinamicamente a causa dell'inserimento di nuovi indirizzi IP durante l'instradamento ma deve essere impostata staticamente prima di inviare il datagramma IP. Il contenuto inziale del campo route data deve essere zero, infatti un datagramma prima di essere inviato non ha attraversato nessun nodo.

Quando un router instrada un datagramma controlla se l'opzione record route è presente. In caso affermativo, il router inserisce il proprio indirizzo IP nel campo route data nella porzione di memoria indicata dal valore del campo pointer e incrementa il pointer di 4 per il router successivo.

In caso la sorgente non abbia dimensionato opportunamente l'area di memoria per questa opzione e di conseguenza l'area di memoria del campo route data è già piena (il valore di pointer eccede il valore di length) il datagramma sarà inoltrato senza inserire l'indirizzo del router corrente nel campo route data. Se invece l'area di memoria per il campo route data contiene ancora un po' di spazio ma non a sufficienza per contenere l'indirizzo IP completo del router corrente, il datagramma viene considerato errato e pertanto scartato. In entrambi i casi potrebbe essere inviato un messaggio di errore alla sorgente del datagramma.

In caso di frammentazione del datagramma questa opzione non viene riportata in ogni frammento ma solo nel primo. Questa opzione può apparire al massimo una volta nel datagramma [6].

1.2.2.2 Loose Source e Record Route

Le opzioni Loose Source e Record Route (LSRR) consentono al nodo sorgente di un datagramma IP di specificare informazioni parziali sull'instradamento che saranno utilizzate dai router intermedi nell'inoltro del datagramma verso il nodo destinazione, e di registrare le informazioni del percorso in termini di nodi attraversati per giungere a destinazione.

La struttura dell'opzione Loose Source è la seguente

10000011	length	pointer	route data
8 bit	8 bit	8 bit	lista di lunghezza variabile

L'opzione inizia con un campo di 8 bit che specifica il **codice identificativo** dell'opzione Loose Source avente sempre valore binario 10000011, 131 in decimale. Il secondo ottetto è il campo **length** che riporta la lunghezza totale dell'opzione, l'unità di misura da utilizzare per questo valore è il byte. Il terzo ottetto chiamato **pointer** è il puntatore corrente al campo route data da cui iniziare a leggere il successivo indirizzo da processare. Il puntatore è relativo a questa opzione, e il minimo valore consentito per questo campo è 4.

Il campo **route data** è composto da un vettore di lunghezza variabile contenente indirizzi IP. Ogni indirizzo IP è composto da 32 bit. Se il valore del

campo pointer è maggiore del valore del campo length, vuol dire che gli indirizzi IP contenuti nel campo route data sono stati tutti processati e l'area di memoria per il campo route data nell'opzione Record Route piena. In caso non si sia ancora raggiunta la destinazione da ora in avanti il routing sarà basato sull'indirizzo IP di destinazione specificato nel campo apposito del datagramma IP.

Se l'indirizzo contenuto nel campo indirizzo di destinazione è stato raggiunto tuttavia il valore del campo pointer non eccede il valore del campo length, l'indirizzo successivo nel campo route diventa il nuovo indirizzo IP di destinazione, e l'indirizzo IP contenuto nel campo route data dell'opzione Record Route sostituisce quello specificato dal nodo sorgente appena impiegato, e il valore nel campo pointer incrementato di 4.

Questa procedura di sostituzione dell'indirizzo specificato dalla sorgente con l'indirizzo memorizzato nell'opzione Record Route implica che la dimensione di questa opzione, e di conseguenza dell'header IP, rimane invariata durante l'avanzamento del datagramma nel suo percorso attraverso la rete. Questa opzione è considerata di tipo loose perché il router può utilizzare qualsiasi percorso composto da un numero qualsiasi di router intermediari per raggiungere l'indirizzo successivo nel percorso.

In caso di frammentazione del datagramma, questa opzione deve essere copiata in ogni frammento. Questa opzione può apparire al massimo una volta nel datagramma [6].

1.2.2.3 Strict Route e Record Route

Le opzioni Strict source e Record Route (SSRR) consentono al nodo sorgente di un datagramma IP di specificare informazioni complete sull'instradamento che saranno utilizzate dai router intermedi nell'inoltro del datagramma verso il nodo destinazione, e di registrare le informazioni del percorso in termini di nodi attraversati per giungere a destinazione.

La struttura dell'opzione è la seguente

10001001	length	pointer	route data
8 bit	8 bit	8 bit	lista di lunghezza variabile

L'opzione inizia con un campo di 8 bit che specifica il **codice identificativo** dell'opzione Strict Route avente sempre valore binario 10001001, 137 in decimale. Il secondo ottetto è il campo **length** che riporta la lunghezza totale dell'opzione, l'unità di misura da utilizzare per questo valore è il byte. Il terzo

ottetto chiamato pointer è il puntatore corrente al campo route data da cui iniziare a leggere il successivo indirizzo specificato dalla sorgente da processare. Il puntatore è relativo a questa opzione, e il minimo valore consentito per questo campo è 4.

Il campo **route data** è composto da un vettore di lunghezza variabile contenente indirizzi IP. Ogni indirizzo IP è composto da 32 bit. Se il valore del campo pointer è maggiore del valore del campo length, vuol dire che gli indirizzi IP contenuti nel campo route data sono stati tutti processati e l'area di memoria per il campo route data nell'opzione Record Route piena e in caso non si sia ancora raggiunta la destinazione da ora in avanti il routing sarà basato sull'indirizzo IP di destinazione specificato nel campo apposito del datagramma IP.

Se l'indirizzo contenuto nel campo indirizzo di destinazione è stato raggiunto tuttavia il valore del campo pointer non eccede il valore del campo length, l'indirizzo successivo nel campo route diventa il nuovo indirizzo IP di destinazione, e l'indirizzo IP contenuto nel campo route data dell'opzione Record Route sostituisce quello specificato dal nodo sorgente appena impiegato, e il valore nel campo pointer incrementato di 4.

Questa procedura di sostituzione dell'indirizzo specificato dalla sorgente con l'indirizzo memorizzato nell'opzione Record Route implica che la dimensione di questa opzione, e di conseguenza dell'header IP, rimane invariata durante l'avanzamento del datagramma nel suo percorso attraverso la rete.

Questa opzione è considerata di tipo strict perché il router per raggiungere la destinazione deve inviare il datagramma direttamente all'indirizzo IP successivo indicato nel campo route data che deve appartenere ad una rete direttamente connessa al router.

In caso di frammentazione del datagramma, questa opzione deve essere copiata in ogni frammento. Questa opzione può apparire al massimo una volta nel datagramma [6].

1.2.2.4 Problemi di sicurezza

La tecnica di source routing è raramente impiegata. Uno dei casi in cui può essere utilizzata è quando l'amministratore di rete per motivi specifici decide di forzare un percorso per determinati pacchetti.

Ad ogni modo in caso di IP source routing un intruso potrebbe tentare di comunicare con uno degli host appartenenti ad una rete inserendo il proprio

indirizzo IP come nodo intermediario nel percorso intrapreso da un messaggio nella comunicazione tra due host legittimi.

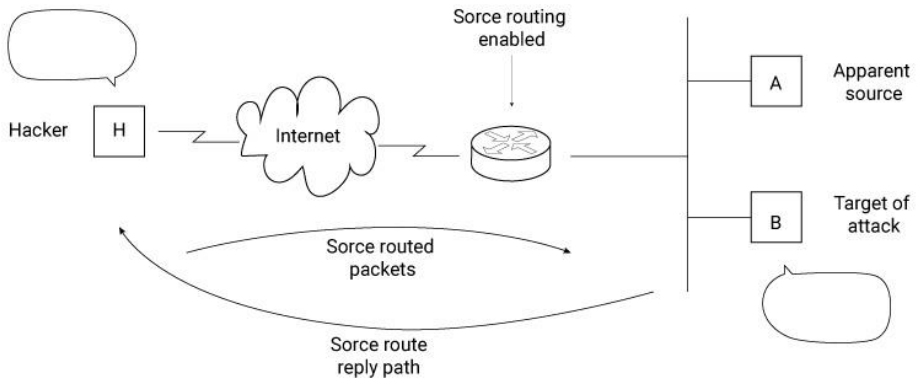

Figura 3 - Schema di attacco basato su Source Routing.

In figura è mostrato lo schema di attacco. L'intruso H vuole far credere di essere un hop intermediario in una comunicazione di tipo IP source routing da host B a host A. H fabbrica una richiesta inserendo un percorso fittizio avente indirizzo sorgente B e hop intermediario H. H invia il messaggio ad A il quale, alla ricezione del messaggio, recupera l'indirizzo della sorgente che scopre essere host B. A considera quindi B legittimo perché appartiene alla sua stessa sottorete e quindi gli invia una risposta utilizzando lo stesso percorso della richiesta passante per H. H è ora in grado di comunicare con A.

Un intruso può effettuare con successo questo tipo di attacco nel caso in cui sia A che il router hanno abilitato la possibilità di scambiare pacchetti utilizzando IP source routing. Tutti gli host che utilizzano TCP/IP solitamente hanno questa opzione abilitata di default [7].

1.2.3 Policy-based routing

Il policy-based routing è una tecnica utilizzata per intraprendere decisioni di instradamento basate su politiche definite dall'amministratore di rete.

I router generalmente utilizzano il destination-based routing ma in alcuni casi può essere utile inoltrare i pacchetti secondo criteri differenti. Per esempio, un amministratore di rete potrebbe specificare che il routing sia effettuato sulla base dell'indirizzo mittente anziché di destinazione. Questo consente l'instradamento dei pacchetti aventi sorgenti diverse verso reti differenti anche nel caso in cui la destinazione sia la stessa, utile in caso di interconnessione fra reti private.

Oltre all'indirizzo mittente, altri criteri possono essere la dimensione del pacchetto, il protocollo di trasporto utilizzato o ulteriori informazioni presenti nell'header o nel payload del pacchetto.

Utilizzare il policy-based routing non equivale a modificare la tecnica di instradamento utilizzata dai router. I router continueranno a instradare i pacchetti secondo il paradigma destination-based. La differenza consiste nell'aggiunta di uno strato software che sostituisce la procedura di IP lookup con una basata su regole più avanzate definite dall'amministratore di rete, senza tuttavia modificare le tecnologie sottostanti.

1.2.3.1 Triade del Policy-based Routing

L'idea chiave del policy-based routing si fonda sull'utilizzo di tre elementi.

- *Indirizzo* o *Address.*
- *Percorso* o *Route.*
- *Regola.*

I primi due elementi sono i concetti tradizionali di indirizzo e percorso presenti nelle comuni tabelle di routing impiegate nel destination-based routing. La regola invece rappresenta la componente di innovazione. Questa triade costituisce la struttura intorno alla quale si basa l'implementazione del policy-based routing. Pur essendo autonomi nel loro funzionamento, il loro impiego combinato restituisce un risultato di gran lunga più efficace e flessibile rispetto al routing basato sulla sola destinazione.

L'ordine che si applica nel trattare questi elementi determina il risultato derivato dal sistema. Essendo ogni elemento indipendente dagli altri, è fondamentale capire gli effetti che ognuno di essi produce:

- Per **indirizzo** si intende il normale IP che indica la località di una sottorete o di un host che fornisce uno o più servizi. L'indirizzo specifica l'oggetto che sta operando o su cui si opera [8].
- Il **percorso** codifica il metodo di inoltro per raggiungere un indirizzo destinazione. Quando si parla di policy-based routing, come detto in precedenza, si parla di routing basato su altri criteri oltre la destinazione. In questo approccio la novità è rappresentata dalla modalità di selezione del percorso che risulta più versatile e flessibile. Per l'invio di un pacchetto lungo il percorso selezionato si continua a utilizzare la tecnica destination-based [8].

- L’ultimo elemento è la **regola**. È qui che risiede la potenza del policy-based routing, che offre modalità più avanzate di selezione del percorso [8].

È per la definizione di politiche raffinate che entra in gioco tale componente. Risulta utile alla sua comprensione pensare una regola come un metodo per introdurre una ACL (Access Control List) per i router. La regola consente di specificare i filtri per identificare un pacchetto e quale percorso scegliere in caso quest’ultimo corrisponda ai criteri imposti dal filtro. Poiché il filtro è parte del meccanismo di selezione della regola, si possono definire regole per specificare altre opzioni avanzate che includono tra quelle disponibili la sorgente, il protocollo e la funzionalità NAT.

Pensare di utilizzare il policy-based routing in un sistema dove è presente solamente una tabella di routing è limitante.

1.2.3.2 Tabelle di routing multiple

In un sistema che prevede l’esistenza di una singola routing table, come i router o la maggior parte dei sistemi operativi, tutti i percorsi disponibili hanno una voce associata riportata in una singola struttura dati che è la tabella di routing. Questa tabella viene consultata sequenzialmente e una volta trovata la corrispondenza si utilizza l’oggetto percorso trovato.

Viene illustrato in seguito un semplice esempio per evidenziare le carenze e le problematiche nella quali si rischia di incorrere impiegando una sola tabella di routing.

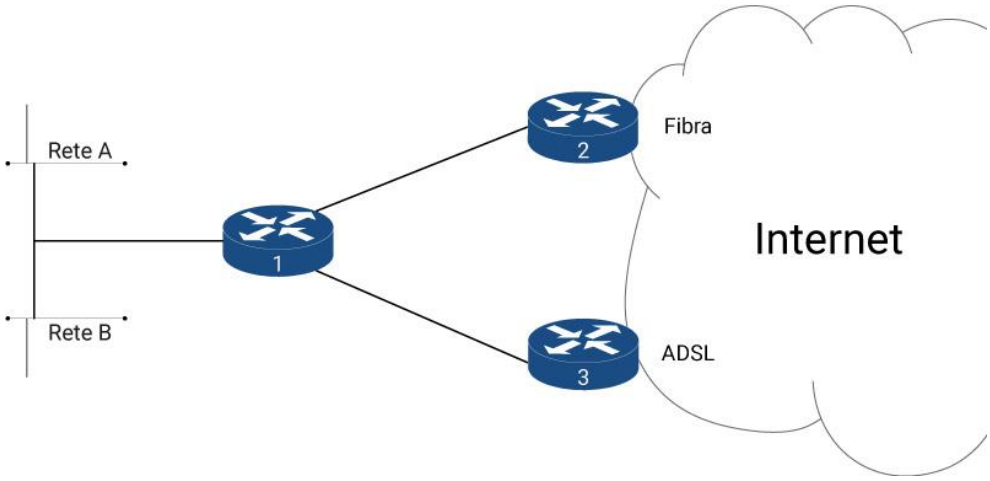

Figura 4 - Esempio Policy Based Routing.

Si immagini di avere tre router in una ipotetica rete aziendale. L’organizzazione ha sottoscritto due abbonamenti presso lo stesso Internet Service Provider, uno con connessione fibra e l’altro con ADSL. Ad usufruire del servizio

vi sono i dipendenti dell'azienda in questione, collegati alla rete A e con la necessità di una connessione veloce, e eventuali utenti ospiti, per i quali è a disposizione la rete B nella quale la velocità della connessione non è un requisito necessario. Esiste inoltre un terzo router che ha la responsabilità di smistare il traffico di A e B verso il router corretto. Supponendo che il router 1 possa fare affidamento su una sola tabella di routing, nel momento in cui arriveranno dei pacchetti da A o da B non sarà in grado di inoltrarli in modo differenziato, poiché l'IP lookup restituisce il primo percorso che rispetta i requisiti richiesti e non è detto che il router successivo, corrispondente al percorso, sia quello corretto. Se ad esempio il percorso di default segue la connessione lenta, anche gli utenti di A si ritroveranno a utilizzarla a causa dell'assenza di un meccanismo di filtraggio del traffico in base alla sorgente.

Per questo motivo l'implementazione della componente regola nel policy-based routing prevede che per il suo corretto funzionamento ci siano tabelle di routing multiple. Se prima si scansionava una tabella contenente percorsi, ora si scansiona una lista di regole ciascuna delle quali ha una tabella associata. Riprendendo l'esempio proposto, se l'amministratore di rete definisse le seguenti due regole:

from rete A lookup tabella 1

from rete B lookup tabella 2

Il contenuto della tabella 1 consentirebbe agli utenti di A di utilizzare il router 2, mentre la tabella 2 indirizzerebbe il traffico di B verso il router 3, ottenendo così il comportamento richiesto.

1.2.3.3 Implementazione in Linux

In Linux, l'implementazione del policy-based routing si basa sul meccanismo del Routing Policy DataBase (RPDB). Il RPDB è un insieme organizzato di route, tabelle di routing e rule. La funzionalità primaria del RPDB è quella di fornire la struttura e il meccanismo per implementare l'elemento rule della triade policy routing. Fornisce anche la possibilità di avere tabelle di routing multiple in Linux.

Il RPDB di Linux consente l'esistenza di 255 tabelle di routing e di 2^32 rule. Questo vuol dire che è possibile definire una regola per qualsiasi indirizzo IPv4. Il RPDB opera sulle rule e le route della triade. Quando è in funzione, l'RPDB considera come primo elemento la rule. La rule, come detto in precedenza, può essere considerata come il filtro di selezione per applicare la politica di routing.

1.2.4 MPLS

Per meglio comprendere il segment routing, descritto nella sezione successiva, è utile introdurre la tecnologia di routing da cui prende ispirazione chiamata MPLS [9].

MPLS è una tecnologia molto diffusa oggi malgrado la sua definizione sia tutt'altro che recente. La sua prima introduzione è del 1998 per opera del consorzio IETF (Internet Engineering Task Force). MPLS è l'acronimo di Multi-Protocol Label Switching. Al momento è utilizzata solamente in reti IP ma in teoria MPLS può lavorare con altri protocolli di rete e consente di ottenere performance elevate nell'inoltro del traffico e nel traffic engineering dove si vuole garantire qualità del servizio.

Degli scenari in cui MPLS è adottato sono per esempio il settore retail dove il costante uso dei dispositivi Point-of-Sale (POS) richiede che i dati di una transazione siano trasportati velocemente al data center per la validazione le transazioni. Oggi molte applicazioni che utilizzano MPLS operano nel campo delle VPN (Virtual Private Network).

Il nome MPLS è composto da due concetti fondamentali Multi-Protocol e Label Switching. MPLS è Multi-Protocol perché agnostico rispetto al protocollo di Livello 2 sottostante. MPLS risiede sopra il Livello 2 e fornisce un metodo di trasporto efficiente e veloce sulla rete a pacchetti. Nel modello ISO/OSI MPLS si colloca tra il Livello 2 e il Livello 3, una sorta di Livello 2.5

Figura 5 - MPLS nel modello ISO/OSI.

MPLS svolge fondamentalmente due compiti:

- Crea un mapping su qualsiasi protocollo di Livello 2 sottostante.
- Controlla il pacchetto IP proveniente dal Livello 3 e lo instrada lungo il percorso *predeterminato* più efficiente.

Per farlo MPLS introduce il concetto di etichetta, o label. Queste etichette sono manipolate con un'operazione di commutazione, da qui la seconda parte del nome Label Switching. Prima di vedere un esempio concreto del suo funzionamento è necessario fare una distinzione tra Routing e Switching.

Il router IP quando scansiona la tabella di routing utilizza la regola del prefisso più lungo per trovare una corrispondenza. Questa operazione crea una congestione in quanto dispendiosa in termini di tempo e può ritardare di molto il processo di inoltro dei pacchetti.

L'idea di switching è diversa. Anziché avere una tabella di routing si ha una struttura dati chiamata **Switch Info Table**.

INPUT		**OUTPUT**	
in interface id	in label	out label	out interface id
2	4	9	0
3	8	10	0

Tabella 2 - Switch Info Table.

Questa tabella contiene due sezioni, una di input e una di output. Nella sezione di input c'è una lista di identificatori di interfacce di ingresso e di referenze di pacchetto in entrata mentre in quella di output vi sono le nuove referenze da applicare al pacchetto in uscita e il corrispondente identificativo dell'interfaccia di uscita. Ogni riga della tabella si riferisce a pacchetti dello stesso tipo e con la stessa importanza. Per esempio la riga uno dice che in caso di arrivo di un pacchetto dall'interfaccia 2 con referenza 4 deve essere inoltrato con referenza 9 all'interfaccia 0. Questo processo consente un inoltro dei pacchetti di gran lunga più veloce rispetto alla scansione della tabella di routing.

MPLS usa il meglio del routing e dello switching. Vedremo a breve che le decisioni di routing prese per un pacchetto quando entra nel dominio MPLS possono essere utilizzate per tutta la classe di pacchetti simili. Questi pacchetti ricevono un'etichetta di classe che è semplicemente letta e sostituita ad ogni router MPLS. Il dispositivo che rende il protocollo MPLS così efficiente è chiamato Label Switch Router (LSR) ed è una combinazione di uno switch e di un router. Nel concreto gli LSR non sono altro che router IP classici in grado di utilizzare MPLS grazie ad un aggiornamento software.

Se nella tradizionale rete IP si hanno i router IP, in una rete MPLS esistono diversi LSR. Il compito di un LSR è quello di accettare un pacchetto in entrata e di inoltrarlo verso la corretta interfaccia di uscita il più velocemente possibile.

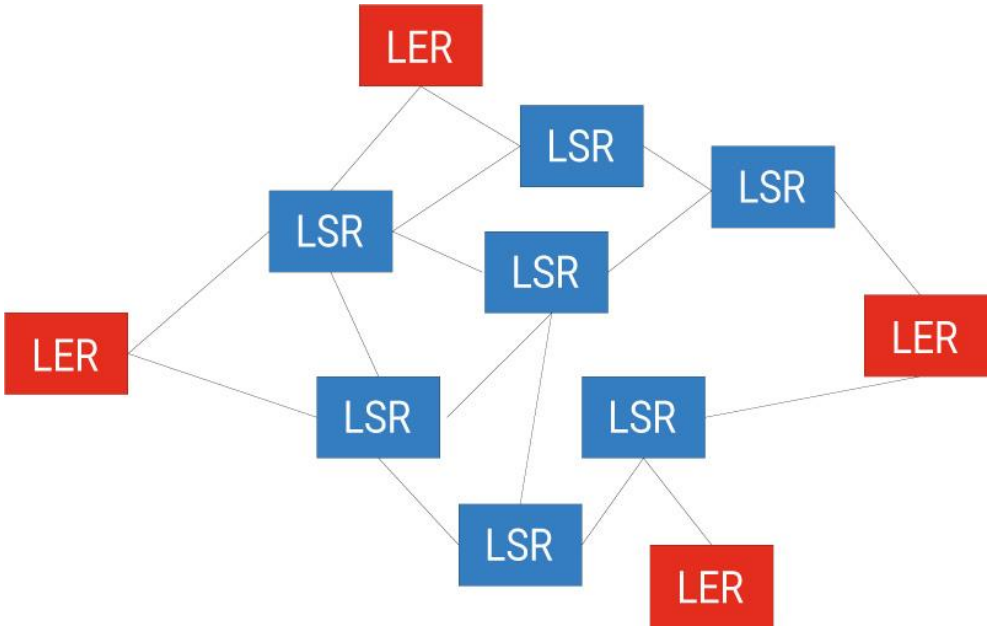

Figura 6 - Topologia MPLS.

Un LSR che si trova ai confini del dominio LSR deve inoltre svolgere una funzionalità aggiuntiva. Questo tipo di LSR si chiama Edge LSR o più comunemente Label Edge Router (LER).

Di seguito verrà mostrato cosa succede quando un pacchetto IP entra in un dominio MPLS, lo attraversa e poi esce ritornando nel dominio IP.

Il pacchetto IP entra nel dominio MPLS attraverso un LSR di entrata chiamato ingress LER. Il router LER controlla le informazioni di Livello 3 nell'header del pacchetto, controlla la sua tabella di Lookup e trova un riferimento chiamato Forwarding Equivalent Class per quel particolare tipo di pacchetto. Un'etichetta per quella classe, rappresentata da un numero, è semplicemente affissa al pacchetto e il pacchetto viene inoltrato nel dominio MPLS. Questa etichetta è contenuta in un header speciale chiamato **Shim Header** posto tra gli header di Livello 2 e Livello 3.

Figura 7 - MPLS header.

All’interno del dominio MPLS quando il pacchetto arriva ad un LSR il quale controlla l’etichetta del pacchetto e consultando la sua Label Information Base commuta l’etichetta con un nuovo valore e continua l’inoltro nel dominio MPLS.

Il pacchetto finalmente arriva al confine di uscita del dominio MPLS ad un dispositivo chiamato egress LER il quale rimuove l’etichetta e inoltra nuovamente il pacchetto verso il dominio IP utilizzando le normali regole di instradamento.

Una differenza fondamentale di MPLS rispetto al tradizionale instradamento IP è che il percorso attraverso la rete MPLS è stabilito prima che il pacchetto intraprenda il proprio viaggio per la destinazione. Per tutte le Forwarding Equivalent Class, menzionate precedentemente, esistono dei corrispondenti percorsi chiamati Label Switch Path. Questi percorsi sono instaurati attraverso uno dei seguenti protocolli che per il loro corretto funzionamento devono essere impiegati da tutti i nodi della rete MPLS:

- Label Distribution Protocol (LDP) [10].
- Resource Reservation Protocol - Traffic Engineering (RSVP-TE) [11].

Gli LSP una volta creati hanno la caratteristica di essere unidirezionali, infatti in caso di comunicazione bidirezionale il pacchetto di ritorno potrebbe intraprendere un percorso differente.

In Figura 7 si presenta un esempio completo. In questa rete MPLS vi sono quattro LSR, di cui due sono LER: un ingress LER a sinistra e un egress LER a destra.

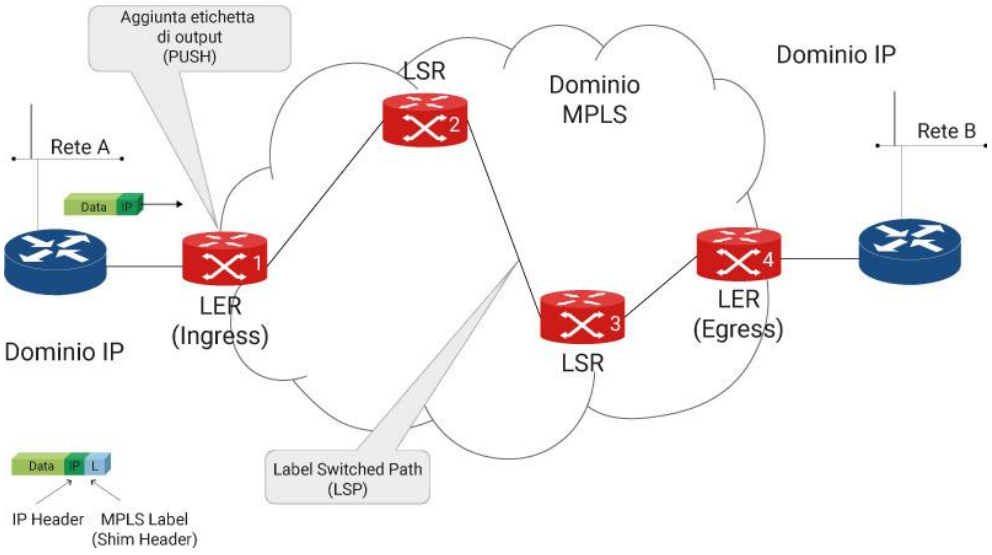

Figura 8 - Comunicazione MPLS.

Come fase preliminare si costruisce un LSP utilizzando come protocollo LDP o RSVP-TE. Un pacchetto IP standard arriva dalla Rete A, posta a sinistra, all'ingress LER. Il dispositivo LER assegna al pacchetto una FEC e assegna al pacchetto l'etichetta 16. Il pacchetto viene inoltrato al LSR successivo, il quale controlla per quella etichetta la voce corrispondente nella sua Label Information Base, in corrispondenza di 16 trova la nuova etichetta 27 e commuta l'etichetta da 16 a 27 e lo inoltra al successivo LSR. Il pacchetto arriva all'LSR numero 3, anche qui si esegue la stessa operazione l'etichetta viene commutata in 33. È importante specificare che il valore FEC non cambia lungo il percorso, solo l'etichetta viene modificata. Finalmente il pacchetto arriva all'egress LER, il quale rimuove l'etichetta e tutte le informazioni relative a MPLS e inoltra il pacchetto IP fuori dal dominio MPLS alla rete B.

1.2.5 Segment Routing

Come ultima tecnica di routing verrà illustrato il Segment Routing. L'instradamento IP tradizionale pecca in flessibilità, controllo dei percorsi e in performance. Alcune delle problematiche del routing IP possono essere risolte utilizzando un'infrastruttura di rete MPLS. MPLS è più flessibile, il Traffic Engineering è più semplice da implementare e i pacchetti sono inoltrati utilizzando un percorso predeterminato la cui selezione è basata sull'etichetta assegnata al pacchetto.

Tuttavia il costo da pagare con MPLS è molto elevato a causa dell'introduzione di un control plane più complesso. Per far fronte a queste problematiche CISCO ha proposto come soluzione quella di connettere tutti i nodi ad un controller e farli comunicare attraverso un protocollo comune. In questo modo l'unica cosa da fare è controllare il router sorgente o controller, questo rende l'infrastruttura più programmabile e scalabile. La risposta che CISCO ha dato a questa problematica è l'architettura basata sul Segment Routing (SR) che risulta molto più flessibile, scalabile, meno complessa e facile da eseguire rispetto al routing IP e MPLS. SR rappresenta di fatto un'evoluzione del tradizionale source-based routing. La sorgente seleziona il percorso e lo codifica nell'header del pacchetto come una lista ordinata di segmenti [12].

Grazie al segment routing la rete non ha più bisogno di mantenere uno stato per-applicazione o per-flow. Invece, segue le istruzioni di instradamento fornite dal pacchetto.

Il segment routing si affida ad un piccolo numero di estensioni dei protocolli IS-IS e OSPF. Può operare con un data plane MPLS o IPv6, e si integra

con le ricche funzionalità multi-servizio di MPLS. Questa tecnica di routing può essere direttamente applicata ad un'architettura MPLS senza alcuna modifica del data plane.

I segmenti sono degli identificatori per ogni tipo di istruzione. Ogni segmento è identificato da un "segment ID" (SID) rappresentato da un intero senza segno a 32 bit. Le istruzioni che un segmento può esprimere possono essere:

- Vai al nodo N utilizzando il percorso più breve.
- Vai al nodo N utilizzando il percorso più breve verso M e successivamente segui i collegamenti imposti dai protocolli di Livello 1, Livello 2 e Livello 3.

Il segment routing utilizza la larghezza di banda a disposizione più efficacemente rispetto alle tradizionali reti MPLS e offre anche latenza minore. Un segmento viene codificato come un'etichetta MPLS. Una lista ordinata di segmenti è ordinata come uno stack di etichette gestito in modalità LIFO. Il segmento da processare risiede in alto allo stack. La relativa etichetta è rimossa nello stack dopo il corretto processamento del segmento.

Il segment routing fornisce una protezione automatica del traffico senza alcuna restrizione dal punto di vista della topologia. La rete protegge il traffico in caso di fallimento di un collegamento o di un nodo senza introdurre messaggi addizionali di segnalazione nella rete. Le tecnologie esistente di IP Fast Re-route (FRR), in combinazione con la funzionalità di routing esplicito di SR garantiscono una protezione totale grazie al backup dei percorsi ottimi.

1.2.5.1 Funzionamento

In Figura 9 è mostrato un esempio di dominio segment routing.

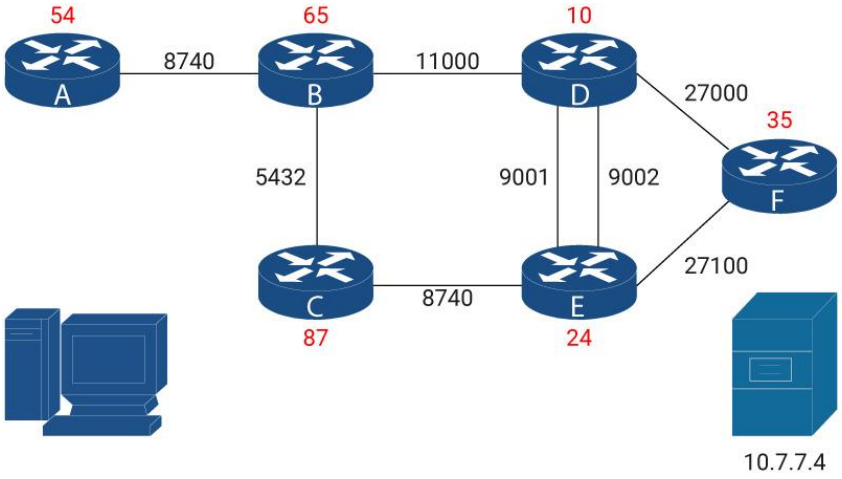

Figura 9 - Dominio SR.

In SR il routing è attuato attraverso l'utilizzo di segmenti. I segmenti indicano fondamentalmente il percorso esatto che il pacchetto deve seguire quando attraversa la rete. I segmenti sono le istruzioni che il pacchetto deve rispettare per raggiungere la sua destinazione.

In questo esempio ci sono sei router A, B, C, D, E ed F. Ad ogni router nella rete viene assegnato un identificatore univoco chiamato Node ID in rosso in Figura 9: per esempio il router A ha Node ID 54 e il router E ha Node ID 24. Tra questi router si creano dei collegamenti individuati da identificatori di adiacenza chiamati Adjacency ID: per esempio l'Adjacency tra il router A e il router B è 8740. Questi segment ID (SID) sono utilizzati dalla tabella di inoltro del router ricevente per determinare l'interfaccia di uscita per raggiungere l'hop successivo. Una volta che questi SID sono assegnati vengono distribuiti tra tutti i router della rete per mezzo del protocollo IGP.

Come detto nella parte introduttiva il segment routing utilizza anche uno dei protocolli IS-IS o OSPF. In questo particolare esempio è utilizzato OSPF. OSPF invia dei pacchetti di segnalazione a tutti i router presenti nella rete. Attraverso l'invio e la ricezione di questi pacchetti di controllo ogni nodo viene a conoscenza dell'esistenza di tutti gli altri nodi. Fondamentalmente OSPF invia questi pacchetti di segnalazione tramite flooding, così da creare una visione completa della topologia di rete. In caso di caduta di un router, tutti i router della rete possono venirne a conoscenza e aiutano il router sorgente nella determinazione di un nuovo percorso migliore. Inoltre OSPF invia le informazioni riguardo alle etichette per popolare le tabelle di inoltro di ogni router. Per percorso migliore non si intende il percorso con il minor numero di nodi intermedi, ma il percorso migliore rispetto ad una metrica scelta, ad esempio il numero di hop, la larghezza di banda o load balancing.

In base ai parametri scelti, OSPF fornisce il percorso migliore. Inoltre IGP esegue dei compiti importanti come advertising dei router, distribuzione dei SID di tutti i router nella rete, calcola il percorso migliore per raggiungere la destinazione e passa le informazioni del percorso migliore alla tabella di routing. IGP utilizza il Path Computation Engine (PCE) per calcolare il percorso più breve. Al completamento delle operazioni di questi protocolli, l'infrastruttura di rete è pronta per supportare la comunicazione vera e propria tra due nodi.

Di seguito si mostrerà come i pacchetti si muovono all'interno della rete.

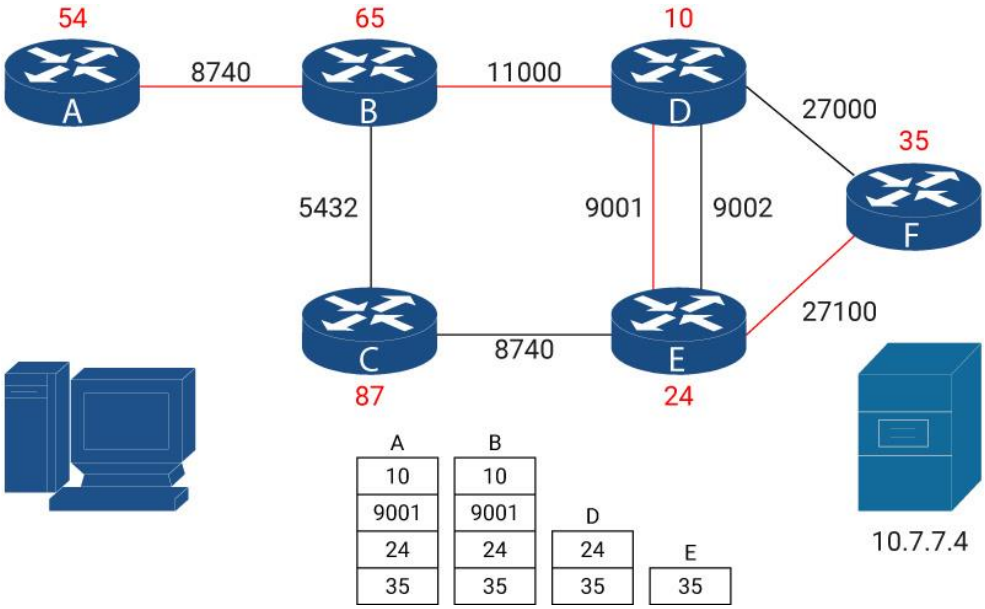

Figura 10 - Esempio comunicazione Segment Routing.

Un pacchetto viene ricevuto dal nodo di ingresso A del dominio SR, il nodo A aggiunge al pacchetto un header SR che consiste in uno stack di etichette. Questa lo stack fornisce istruzioni su come il pacchetto deve essere inoltrato per raggiungere la sua destinazione. Per il router A la lista di segmenti è [10, 9001, 24, 35], ovvero il pacchetto per giungere ad F deve passare per il router D e per il router E attraverso il collegamento 9001.

Per prima cosa il router A controlla la sua tabella di routing, le cui voci contengono il nodo destinazione e il nodo successivo. In questo caso il pacchetto ha destination ID 10.7.7.4. Per raggiungere D (10) il pacchetto deve passare per B (65). Quando arriva in B il router non esegue alcuna operazione sullo stack, l'unica cosa che fa è leggere l'elemento in cima allo stack, 10. Il pacchetto quindi passando per B arriva a D (10). Gli altri router che saranno attraversati dal pacchetto hanno a disposizione una tabella chiamata Label Information Base (LIB). Per esempio la tabella per il router D è

Incoming SegID	Outgoing SegID	Interface
10	9001	0.3

Tabella 3 - LIB router D.

Una LIB contiene informazioni riguardanti l'identificativo del segmento in entrata, l'identificativo del segmento in uscita e l'interfaccia da utilizzare in uscita. 9001 è un adjacencyID tra i router D ed E, da notare che tra D ed E non c'è solo 9001 ma anche 9002, tuttavia il pacchetto deve passare per 9001 e non per 9002 perché queste sono le istruzioni fornite dallo stack di etichette. Il nodo B prima di inviare il pacchetto rimuove dalla pila 2 etichette 10 e 9001 e lo stack è inoltrato al router successivo D (24). Quando il pacchetto arriva

in E, il router legge solo 24 e prima di inoltrare il pacchetto rimuove 24 dallo stack inoltra il pacchetto che conterrà solo l'etichetta 35 ad F (35). Da F il pacchetto viene inoltrato alla sua destinazione 10.7.7.4. Una cosa importante da notare è che lo stato completo del flusso del traffico risiede nel pacchetto e non nella rete.

1.3 Routing in reti MANET

Il Dynamic Source Routing (DSR) è una tecnica di routing utilizzata in scenari MANET (Mobile Ad-Hoc Networks). Prima di parlare di questa tecnica è utile fare una breve introduzione su cos'è una MANET.

Si immagini di avere una rete composta di dispositivi wireless in uno scenario sfidante dove non è presente infrastruttura, dove quindi tanti problemi che nelle reti tradizionali sono risolti qui non lo sono, tra cui ovviamente il routing.

Una rete si dice ad-hoc quando viene creata dinamicamente senza il bisogno di elementi di infrastruttura, se tutti i suoi nodi sono mobili si chiama Mobile Ad-Hoc. L'idea nasce da scenari di deployment che non consentono di avere un'infrastruttura o perché si desiderano collegamenti ad alta volatilità o perché in ambienti troppo sfidanti per avere un'infrastruttura come alcune zone degli oceani. I nodi di queste reti hanno sicuramente funzionalità e risorse energetiche differenziate. Una rete ad hoc è una rete wireless dove i nodi possono comunicare anche se non sono nello stesso raggio di copertura e che sono quindi collegati con percorsi multi-hop.

Fare routing nelle MANET comporta delle difficoltà rispetto alle reti tradizionali. Un approccio può essere quello di utilizzare i protocolli di routing già ampiamente collaudati nelle reti tradizionali, tuttavia il risultato non sarebbe ragionevole. Il motivo risiede nelle assunzioni: nelle reti tradizionali si parte dal presupposto che la topologia di rete cambi molto lentamente nel tempo rispetto alle necessità di comunicazione e qui purtroppo questa assunzione non vale.

Negli anni sono stati proposti centinaia di protocolli per il routing in MANET, questo evidenzia il fatto che è impossibile trovare un protocollo general purpose ottimale. In generale il routing cerca di ottimizzare la latenza e la banda, nelle MANET potrebbe aver senso considerare, oltre ai parametri classici, anche requisiti come l'ottimizzazione della stabilità del percorso o l'ottimizzazione della capacità residua della batteria di un nodo mobile perché si rischierebbe il partizionamento della rete.

I protocolli di routing in MANET possono essere classificati in 4 macro-categorie:

1) Proattivi: cercano di ottenere informazioni sui percorsi validi prima che ci sia la necessità di comunicazione tra i nodi appartenenti a quel percorso. Questo è quello che fanno i protocolli di routing tradizionali. I percorsi sono preparati prima. Questa categoria risulta perdente nelle MANET.
2) Reattivi: data la dinamicità delle MANET si decide di non calcolare i percorsi in anticipo ma aspettare che ci sia bisogno di comunicazione. Magari si tiene qualche informazione ma il percorso è calcolato solo on-demand.
3) Geografici: a volte è così difficile fare routing che è possibile farlo solo conoscendo il posizionamento fisico dei nodi.
4) Ibridi: prende parti da differenti famiglie di protocolli e li mescola avendo un comportamento adattivo a seconda delle condizioni circostanti.

1.3.1 Flooding-based routing

Per capire come funziona il Dynamic Source Routing (DSR) è necessario immaginare la soluzione più semplice possibile, quella basata sul flooding. Si potrebbe pensare ad un algoritmo di routing inefficiente e facile da implementare che ogni volta che c'è bisogno di comunicare fa il broadcast di tutto quello che riceve verso tutti i nodi nel suo raggio di copertura fino a quando la destinazione non viene raggiunta. Solo il destinatario può accorgersi che il messaggio è per lui perché nel messaggio è contenuto l'ID del destinatario.

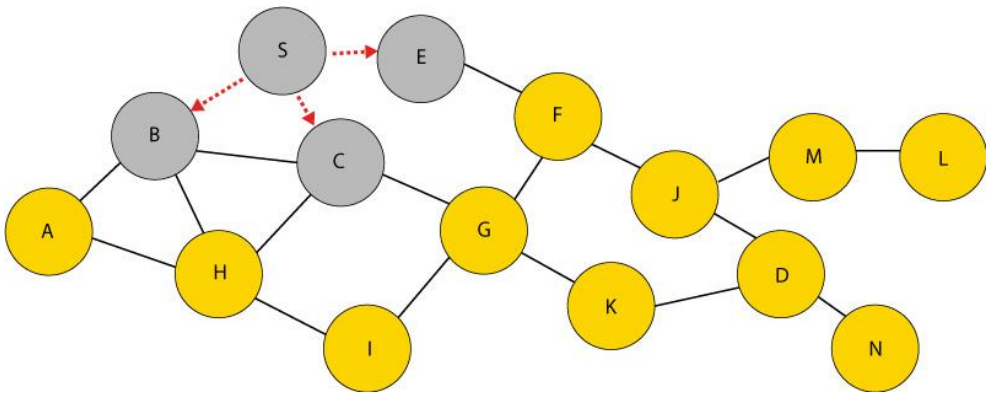

Figura 11 - Flooding da S a D.

Questa soluzione data la sua semplicità è caratterizzata da alcuni punti di inefficienza.

Un rimedio per migliorare le prestazioni di questo algoritmo è l'introduzione di un sistema di anti-replay. Considerando la Figura 11, se il nodo S invia un messaggio a tutti i nodi vicini, in questo caso B, C ed E, questi nodi quando inoltreranno il messaggio ai loro vicini eviteranno di inoltrare il messaggio nuovamente ad S. Questo miglioramento è possibile solo se non si usa un broadcast di basso livello. Un ulteriore miglioramento del meccanismo di anti-replay è il seguente: i nodi possono tenere traccia in una tabella dei messaggi già inoltrati ai vicini, per cui se il messaggio ripassa non viene ulteriormente duplicato. Per esempio il nodo C e il nodo B alla ricezione di un messaggio da S lo inoltrano ai loro vicini. In questo caso C e B sono anche vicini, C invia il messaggio a B e viceversa generando un possibile loop.

In un protocollo come questo si ha una propagazione di messaggi infinita nello spazio e nel tempo, per evitare questo si introduce un Time-To-Live (TTL) dei messaggi in termini di numero di nodi attraversati.

Un ultimo aspetto da considerare è una problematica dovuta alla natura della connettività senza fili esistente tra i nodi: le collisioni. Avendo sempre riferimento la Figura 11 si immagini che S invii un messaggio a B e a C, questi due nodi non appena ricevono il messaggio lo inoltrano e questo genera un'infinità di collisioni. Un modo per risolvere il problema è che l'inoltro non venga eseguito subito ma lo si faccia dopo un tempo casuale. Questa problematica viene definita in letteratura *broadcast storming* ed è la cosa peggiore che può accadere in comunicazioni wireless.

In generale il flooding ha un overhead elevato, tuttavia ha anche i suoi vantaggi. Per esempio non richiede alcun coordinamento tra i nodi, inoltre è reattivo, è molto robusto alla mobilità dei nodi perché basta che esista un possibile cammino e il messaggio riesce ad arrivare a destinazione, garantendo quindi alta affidabilità che tuttavia si riduce a causa del grande numero di collisioni. Il flooding è una soluzione inadatta per scambiare pacchetti dati, tuttavia è utile impiegarlo per i pacchetti di controllo, il DSR lo utilizza in questo modo.

1.3.2 Dynamic Source Routing

È ora possibile introdurre il Dynamic Source Routing. Esso è reattivo: quando il mittente vuole trasmettere ad un destinatario viene usato un pacchetto di controllo distribuito in flooding per determinare il percorso. Il

pacchetto arriva a destinazione secondo un determinato percorso, in seguito la destinazione risponde al mittente utilizzando un secondo pacchetto di controllo che adotterà il percorso del primo e i pacchetti dati useranno sempre quella strada. A tal fine nell'header dei pacchetti dati sono specificati mittente, destinatario e percorso.

Per scoprire il percorso il nodo sorgente avvia una procedura chiamata Route Discovery, la quale consiste nel flooding di un pacchetto di controllo denominato Route Request (RREQ). Ogni volta che un nodo riceve un RREQ deve aggiungere in coda al campo percorso del pacchetto il suo ID e propagarlo. Ad un certo punto il messaggio RREQ arriva a destinazione, il nodo destinazione capisce che il pacchetto è indirizzato a lui perché nel campo destinatario dell'header è riportato il suo ID.

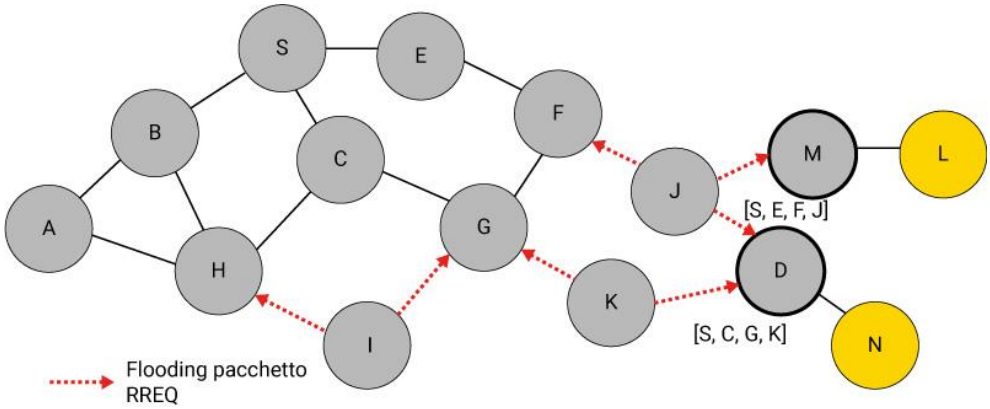

Figura 12 - Propagazione RREQ.

Inoltre, a causa del flooding, il nodo destinazione riceverà più copie di RREQ da cammini diversi. Di solito il primo RREQ che arriva vince e il percorso che questo RREQ ha utilizzato per arrivare a destinazione sarà quello sfruttato per lo scambio dei dati. Per non inficiare le prestazioni, il pacchetto RREQ ha un TTL massimo e i nodi intermedi utilizzano meccanismi di anti-replay per evitare il *broadcast storming*. Nell'esempio mostrato in Figura 12 il messaggio RREQ vincente è quello proveniente dal nodo J.

La propagazione del messaggio RREQ per il nodo D si interrompe quando lo raggiunge, per tutti gli altri nodi ciò avviene all'esaurimento del TTL come nel nodo M in figura.

Quando RREQ raggiunge il nodo destinazione D, esso prepara un altro pacchetto di controllo, che in DSR è detto Route Reply (RREP), rimandandolo alla sorgente S esattamente sul percorso seguito dal messaggio RREQ vincente. Se RREP non giunge alla sorgente, per esempio a causa della caduta di un nodo, dopo un timeout predefinito opportunamente dimensionato, S inizia il protocollo da capo.

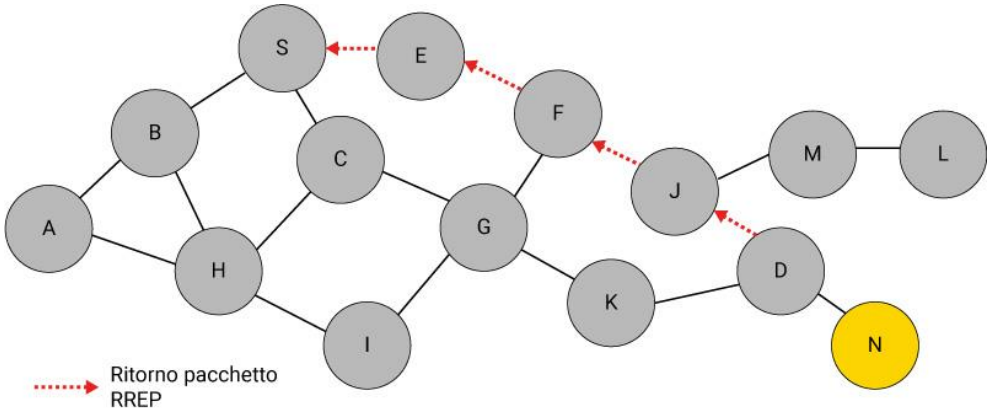

Figura 13 - Percorso del pacchetto RREP.

Supponendo che la comunicazione vada a buon fine e che RREP riesca ad arrivare a S. Di conseguenza il nodo sorgente ha in mano un percorso che almeno per lo scambio dei due pacchetti di controllo è risultato valido, tale percorso sarà utilizzato per lo scambio dei dati. Affinché esso sia impiegato nello scambio dei messaggi dati, la sorgente lo inserisce nell'apposito campo dell'header. A questo punto non si fa più il flooding e le scelte di routing sono comandate dal percorso deciso dalla sorgente descritto nel pacchetto. Per questo motivo si parla di *source routing*. Il termine *dynamic* è attribuito al calcolo on-demand del percorso.

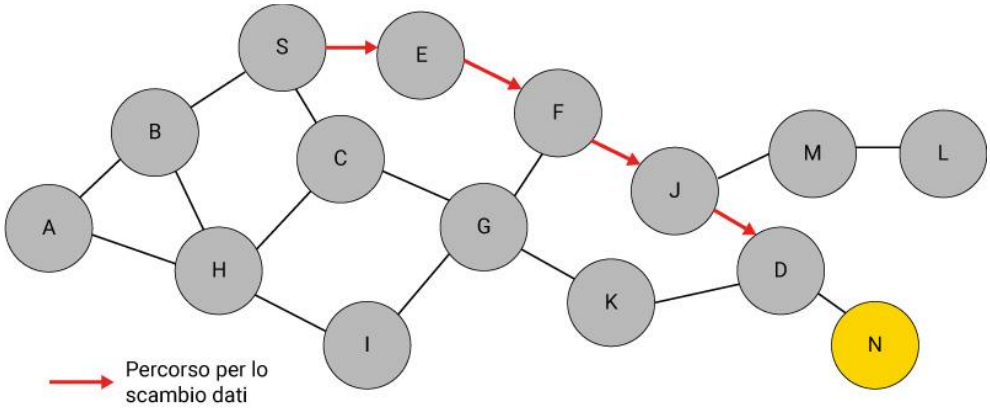

Figura 14 - Scambio dati tra S e D.

In DSR esiste un altro pacchetto di controllo che è in grado di gestire le cadute dinamiche dei nodi.

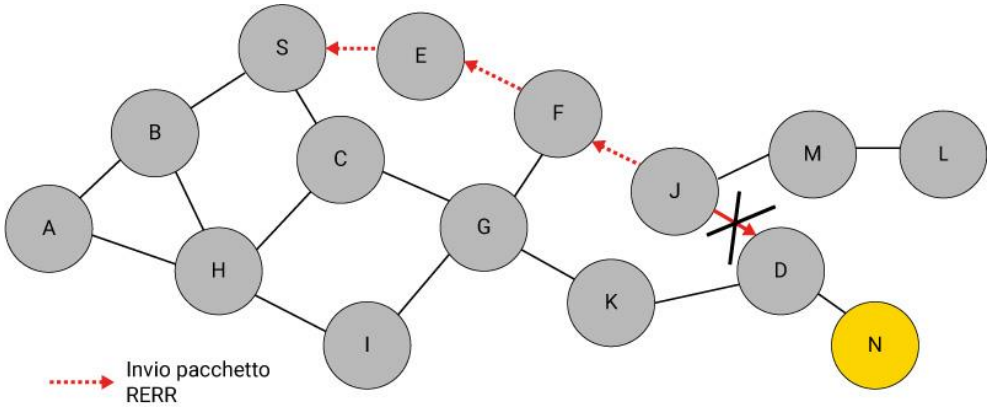

Figura 15 - Invio pacchetto RERR.

Ad esempio se J durante il percorso non è in grado di inviare un pacchetto a D, J è obbligato ad inviare indietro ad S un pacchetto di controllo chiamato Route Error (REER). Quando REER arriva ad S, esso può iniziare il protocollo per individuare un nuovo percorso.

In DSR di base non è possibile fare Path Caching, meccanismo per il quale i nodi intermedi memorizzano informazioni di stato sui percorsi ottenute dalle interazioni precedenti. I dati in cache hanno una scadenza, motivo per cui è necessario introdurre un sistema di invalidazione per garantire la coerenza dei dati, per esempio una routine periodica di ping sui vicini. Tale operazione naturalmente comporta un costo computazionale che è ragionevole in reti tradizionali, ma che può non essere sostenibile in reti MANET, nelle quali si deve tenere in conto l'alta dinamicità della topologia di rete e la sua frequenza di utilizzo.

Utilizzare il caching accelera il processo di ricerca dei percorsi e riduce il tempo di propagazione dei RREQ, tuttavia richiede una partecipazione dei nodi intermedi impattando il consumo di batteria, di memoria e la complessità dell'algoritmo di routing. Queste modifiche quindi introducono un overhead che può essere più o meno trascurabile a seconda del dominio applicativo [13].

Capitolo 2
Software-defined Networking

Grazie all'avvento del Cloud Computing sono stati proposti nuovi concetti relativi al networking con lo scopo di semplificare la gestione delle reti rendendole programmabili. L'introduzione del paradigma Software-defined Networking (SDN) è uno di questi concetti adottati nelle moderne soluzioni cloud, che ha come obiettivo quello di eliminare i processi di mantenimento dell'infrastruttura di rete e di garantire una gestione facilitata delle sue risorse. In quest'ottica SDN si prefigge l'obiettivo di ottenere alte performance in tempo reale e di rispondere ai requisiti di high-availability. Storicamente il concetto legato a questo approccio è stato introdotto a metà degli anni 90, tuttavia solo all'inizio del 2011 con il primo rilascio della specifica OpenFlow si sono visti i primi prototipi.

SDN è un approccio innovativo per la progettazione, l'implementazione e la gestione di reti che ha tra i suoi punti salienti la separazione del control plane dal data plane. Tale caratteristica è una differenza sostanziale rispetto alle reti IP tradizionali dove solitamente il control plane è distribuito tra i vari router della rete. Questa separazione offre numerosi benefici in termini di flessibilità e controllabilità. Da una parte infatti permette di combinare i vantaggi della virtualizzazione di sistema e del cloud computing e dall'altra grazie alla presenza di un'intelligenza centralizzata di ottenere una chiara visibilità della rete al fine di facilitare la sua gestione e manutenzione che si traduce in un miglior controllo e reattività della rete stessa.

Nelle convenzionali infrastrutture di rete, l'implementazione, la configurazione e la risoluzione di problemi richiede solitamente l'intervento di ingegneri altamente qualificati che comporta un alto costo di gestione associato. Infatti la varietà e la complessità degli apparati di rete rendono la loro manutenzione molto costosa e l'infrastruttura sottostante meno affidabile in caso di fallimenti di rete frequenti.

Dato che SDN separa le decisioni di routing del piano di controllo dalle decisioni di inoltro dei pacchetti del piano dei dati, l'amministrazione e la gestione della rete diventa più semplice perché il control plane si occupa solamente delle informazioni relative alla topologia di rete logica, il routing del traffico e così via. Dall'altra parte il data plane orchestra il traffico di rete in accordo alle istruzioni stabilite dal control plane.

In SDN, le operazioni di controllo sono centralizzate in un componente chiamato *controller* che impone le politiche di rete da utilizzare. Esistono diverse piattaforme open-source basate sulla presenza di un controller come ad esempio Floodlight [14], OpenDayLight [15] e Beacon [16]. La gestione della rete può essere ottenuta operando sui tre livelli di networking, per esempio i fornitori di servizi possono allocare risorse ai clienti per mezzo del management plane, configurare e modificare le politiche di rete e le entità logiche lavorando sul control plane e installare elementi di rete fisici agendo sul data plane [17].

L'obiettivo principale di SDN è quello di rendere la progettazione delle reti aperta e programmabile. Nel caso in cui un'organizzazione richiedesse un comportamento specifico di rete, gli sviluppatori potrebbero utilizzando questo paradigma per implementare e rilasciare un'applicazione per realizzare tale comportamento. Queste applicazioni possono fornire funzioni di rete comuni come ad esempio traffic engineering e sicurezza. Di conseguenza SDN crea un'ambiente di forte flessibilità dove la rete può evolvere alla velocità del software.

2.1 Architettura

Per comprendere al meglio il concetto di SDN può essere utile riferirsi a un esempio pratico ben conosciuto come il modello di un sistema operativo. L'esempio in questione verrà preso come analogia per capire come funzionano le reti SDN.

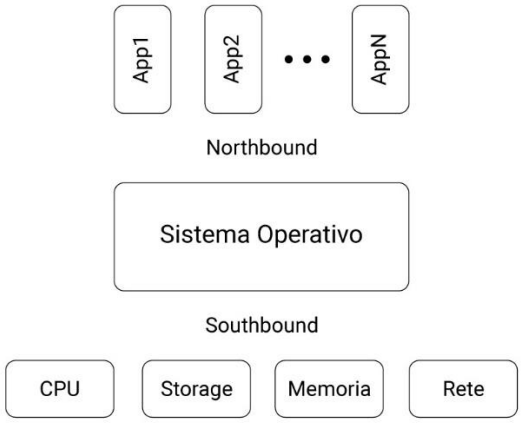

Figura 16 - Modello del sistema operativo.

Come illustrato in Figura 16 è possibile separare logicamente un sistema operativo in tre livelli di base. Il sistema operativo si pone da intermediario tra le applicazioni e l'hardware sottostante; per svolgere questa funzione di

mediazione è dotato di numerosi servizi di base. Il sistema operativo è responsabile della gestione dell'hardware di sistema presente nel livello più basso come CPU, spazio di archiviazione, memoria centrale e interfacce di rete.

Il sistema operativo per comunicare con il livello inferiore dell'hardware e con il livello superiore delle applicazioni è dotato di due interfacce che in questo esempio si definiscono rispettivamente di lato sud o southbound e di lato nord o northbound. La possibilità di sviluppare, di aggiungere e di rimuovere applicazioni all'interno di un sistema operativo rende il sistema flessibile, personalizzabile e general purpose.

Il modello SDN, illustrato in Figura 17, opera in modo analogo al modello appena descritto.

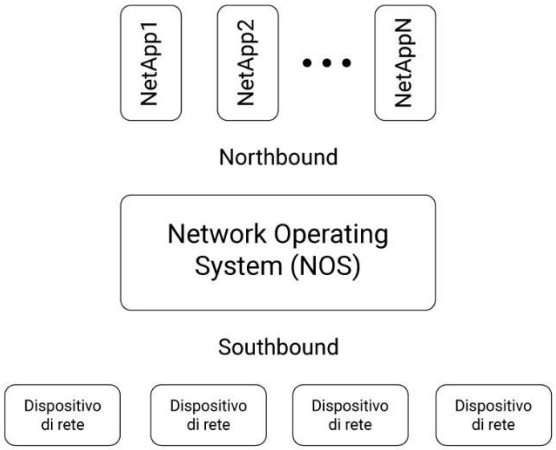

Figura 17 - Modello SDN.

Al posto del sistema operativo il livello intermedio del modello SDN è occupato dal Network Operating System, o NOS, chiamato più comunemente *controller SDN*. Il NOS è composto da dei servizi di base per supportare la comunicazione con i nodi della rete e per fornire un'interfaccia programmabile verso le applicazioni. Nel lato sud, al posto delle risorse computazionali, di archiviazione, di memoria ci sono i dispositivi di rete per l'inoltro del traffico. Questi dispositivi alla ricezione dei pacchetti possono agire su di essi e aggiornare le informazioni una volta inoltrati. Le azioni intraprese possono essere di qualunque tipologia: tra le più comuni figurano quelle di scarto di un pacchetto, di modifica dell'header o di invio di un pacchetto utilizzando una o più porte di uscita. Le istruzioni che indicano quali azioni i dispositivi di rete possono compiere su determinati pacchetti sono demandate dal controller SDN.

Come nel modello del sistema operativo, nel livello superiore ci sono le applicazioni, chiamate in questo dominio *applicazioni di rete* o networking

application. Proprio come le applicazioni che eseguono all'interno di un sistema operativo, le applicazioni di rete possono adempiere a numerosi scopi. Tuttavia parlando di SDN il loro scopo è focalizzato su un contesto applicativo di networking. Quando un pacchetto arriva ad un dispositivo di rete gestito da un controller SDN, il dispositivo legge le informazioni contenute nell'header e può fare due cose: o sa esattamente come trattare quel pacchetto oppure se non ha informazioni per quel tipo di pacchetto chiede al NOS quali azioni devono essere eseguite.

Le applicazioni di rete che eseguono al di sopra del Controller SDN determinano, grazie all'interfaccia di northbound, quali azioni devono essere effettuate sul pacchetto e queste informazioni vengono inviate al dispositivo di rete utilizzando il Controller SDN come intermediario.

I dispositivi di rete solitamente effettuano il caching di tali istruzioni in modo da non richiedere nuovamente l'intervento del controller per pacchetti futuri della stessa tipologia. Questo processo continua da dispositivo a dispositivo finché il pacchetto non lascia la rete SDN verso la sua destinazione che può essere una diversa regione SDN o una rete tradizionale non SDN. I pacchetti futuri della stessa tipologia attraverseranno dei percorsi più veloci in quanto la consultazione del controller SDN non sarà necessaria grazie al caching delle informazioni ottenute durante le interazioni precedenti. Avendo introdotto il concetto di caching, questi percorsi veloci dureranno chiaramente fino alla scadenza di queste informazioni dovuta alla presenza di un timer di controllo della freschezza di tali informazioni.

Dalla descrizione di questo modello si evince la presenza di un sistema operativo di rete logicamente centralizzato. Il controller SDN ha una visione globale di tutti i dispositivi di rete ed è in grado di comunicare a questi dispositivi le istruzioni su come inoltrare i pacchetti. Il controller crea quindi un'astrazione o una vista semplificata della rete alle applicazioni di rete soprastanti, le quali utilizzano queste informazioni per prendere decisioni chiave su come implementare le politiche di rete.

2.1.1 Componenti principali del controller SDN

In questa sezione si forniscono dettagli aggiuntivi sul modello SDN e in particolare sul controller. Come introdotto nella sezione precedente, nel livello più basso ci sono i dispositivi di rete, noti anche come switch, che hanno come scopo principale l'inoltro del traffico. Questi apparati possono essere dei tradizionali router realizzati in hardware oppure possono essere dei dispositivi realizzati in software utilizzando ad esempio Open vSwitch. La

condizione necessaria e sufficiente affinché questi dispositivi possano essere impiegati nel modello SDN è che supportino un'interfaccia programmabile southbound come OpenFlow. Gli switch in hardware solitamente hanno performance più elevate mentre quelli software forniscono maggiore flessibilità in caso di nuove funzionalità.

Il controller SDN necessita di un meccanismo per comunicare con i dispositivi di rete sottostanti. Le informazioni che possono essere comunicate includono:

- Istruzioni sulla gestione dei pacchetti.
- Messaggi di alert dovuti all'arrivo di pacchetti ai nodi di rete.
- Notifiche di cambiamento di stato a seguito per esempio della creazione o caduta di un link.
- Informazioni statistiche sullo stato e il comportamento degli switch.

Tutto questo scambio di informazioni avviene per mezzo dell'interfaccia di southbound. Il protocollo più famoso che realizza l'interfaccia di southbound è OpenFlow che sarà illustrato a breve. Complementare ad OpenFlow è presente il protocollo OVSDB, protocollo di gestione della rete utilizzato per gestire la configurazione dei dispositivi di rete.

Il Controller SDN esegue dei servizi di base come:

- Servizio di topologia.
- Servizio di inventario.
- Servizio di raccolta di informazioni statistiche.
- Servizio di host tracking.

Il **servizio di topologia** si occupa di determinare come i vari dispositivi di rete sono connessi gli uni con gli altri e costruisce quello che è chiamato il grafo della topologia di rete. Questo servizio può essere realizzato per esempio imponendo agli switch di inviare al Controller SDN pacchetti LLDP o altri pacchetti specializzati.

Il **servizio di inventario** è utilizzato per tracciare tutti i dispositivi che sono abilitati alle funzionalità SDN e di raccogliere informazioni di base sul loro conto come ad esempio la versione di OpenFlow utilizzata e quali funzionalità supportano.

Il **servizio di statistica** può essere utilizzato per leggere le informazioni dei counter come i contatori del traffico sulle interfacce di flow e le tabelle di flow.

Il **servizio di host tracking** si occupa di scoprire dove si trovano gli indirizzi IP e gli indirizzi MAC all'interno della rete tipicamente attraverso l'intercettazione dei pacchetti di rete.

Il controller SDN, tramite l'interfaccia northbound, fornisce al livello applicativo un'astrazione semplificata dell'infrastruttura di rete sottostante. Molto spesso è sufficiente che l'intera rete sia rappresentata come un singolo switch logico per un'applicazione. Tipicamente sono API REST le quali grazie a chiamate dirette HTTP al controller SDN consentono di modificare il comportamento di rete ed usufruire delle informazioni raccolte dai servizi di base. Nella comunità di sviluppatori rimane ancora aperta la discussione su come standardizzare l'interfaccia northbound utile per dare una spinta innovativa allo sviluppo delle applicazioni di rete e garantire portabilità tra diversi controller. Recentemente l'organizzazione Open Networking Foundation (ONF) si è focalizzata sulle SDN northbound API dando vita ad un gruppo di lavoro chiamato Northbound Working Group il cui scopo è quello di sviluppare prototipi e valutare se creare o meno una specifica per le interfacce northbound [18]. Ad oggi purtroppo questo tipo di standardizzazione non esiste.

La presenza di un controller SDN consente di organizzare opportunamente la rete a seconda del carico di lavoro rendendola altamente reattiva. Questa organizzazione può interessare livelli diversi. Per esempio a livello southbound le classificazioni del traffico possono interessare controller SDN separati mentre a livello northbound una o più applicazioni di rete possono gestire tipologie di traffico in modo diverso. L'ambiente SDN permette quindi di trattare la rete come una risorsa da gestire opportunamente a seconda dell'utilizzatore o del carico di lavoro. Questo approccio elastico è molto simile a come le risorse computazionali sono gestite nelle attuali infrastrutture cloud.

2.1.2 Fault tolerance e scalabilità

Comunemente ci si riferisce al controller SDN come un'entità logicamente centralizzata. Inoltre è importante sottolineare che questo concetto è diverso dal dire che tale entità è fisicamente centralizzata. Quando una rete SDN viene rilasciata in produzione questa non dipende unicamente da un singolo controller SDN fisico che comporterebbe la presenza di un singolo punto di vulnerabilità per l'intera rete e porterebbe a evidenti limiti di scalabilità. Esistono diverse metodologie per far in modo che una rete SDN soddisfi i requisiti di high availability e scalabilità.

Un modo consiste nell'utilizzare una tecnica nota come *clustering*. L'idea è quella di organizzare i controller SDN in un cluster in grado di bilanciare il carico di lavoro invece di delegare ad un unico controller la gestione dell'intero sistema. Questa soluzione offre high availability perché in caso di caduta di uno o più controller le funzionalità del controller rimangono comunque attive e scalabilità in quanto diversi controller possono gestire le richieste multiple in parallelo.

Inoltre a livello organizzativo la rete potrebbe essere separata in regioni differenti, ciascuna della quale gestita da un controller SDN regionale. Regioni diverse possono all'occorrenza coordinarsi scambiandosi informazioni attraverso interfacce orizzontali chiamate eastbound e westbound dando vita ad una federazione di controller.

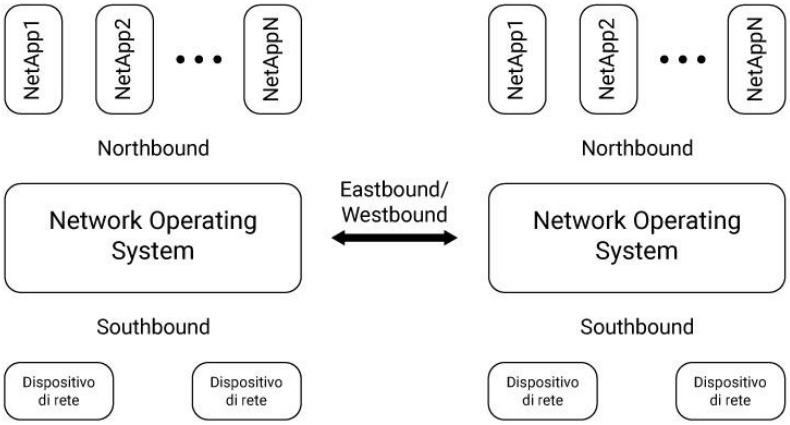

Figura 18 - Interfacce eastbound e westbound.

Infine i controller potrebbero essere progettati come una gerarchia. In questa configurazione esistono controller di alto livello con un alto livello di astrazione e controller sottostanti di più basso livello più vicini ai dispositivi di rete.

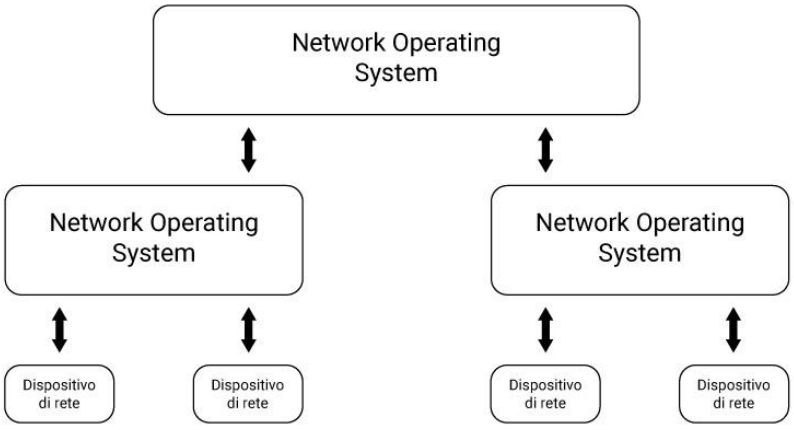

Figura 19 - Organizzazione gerarchica di controller SDN.

2.1.3 Differenze tra reti SDN e reti tradizionali

I router di rete tradizionali hanno la caratteristica di avere al loro interno i moduli responsabili alla gestione sia del control plane che del data plane. Come detto in precedenza il compito del data plane è quello di gestire e inoltrare i pacchetti seguendo le istruzioni contenute in strutture dati locali al dispositivo come la routing table. Queste istruzioni sono il risultato delle decisioni prese dal control plane. Il dispositivo di rete utilizza il control plane per comunicare con gli altri nodi della rete eseguendo dei protocolli distribuiti come BGP, MPLS e OSPF. È il control plane quindi che si occupa di gestire la complessità della rete e dove si definiscono le politiche di rete. Il control plane determina come i singoli tipi di pacchetto dovrebbero essere gestiti ed invia questa informazione al data plane.

Esistono quindi delle differenze tra il modello di rete tradizionale e il modello SDN. Prima di tutto i tradizionali nodi di rete sono caratterizzati da hardware e software proprietari e in quanto tali vanno trattati come delle scatole chiuse. A livello implementativo il control plane è fortemente accoppiato con il data plane; non è possibile accedere direttamente al comportamento del data plane e di conseguenza in caso di definizione ed attuazione di una nuova politica da parte dell'amministratore di rete, quest'ultimo deve pensare a seconda del tipo di dispositivo quali funzionalità il control plane rende disponibili. Generalmente gli amministratori di rete configurano questi dispositivi utilizzando un'interfaccia a linea di comando e quando bisogna aggiungere una nuova politica di gestione del traffico le opzioni a disposizione potrebbero essere limitate.

Un altro svantaggio delle reti tradizionali se comparate alle SDN consiste nel fatto che ogni nodo della rete deve essere configurato singolarmente. Pertanto l'amministratore di rete non ha mai a che fare con un unico punto di configurazione. Per esempio in un data center potrebbero esserci decine o centinaia di nodi da riconfigurare in caso si voglia implementare una nuova politica. Ognuno di questi nodi è dotato di un proprio control plane e tutti questi control plane devono comunicare tra loro utilizzando un protocollo distribuito. L'utilizzo di questo paradigma tipico delle reti tradizionali risulta complesso e porta facilmente ad errori di configurazione. Al contrario in una rete SDN esiste un controller logicamente centralizzato, il quale ha una visione globale dell'intera rete. La presenza di un'entità centralizzata nasconde le complessità legate alla gestione di protocolli distribuiti e alla configurazione di nuove politiche su un numero di nodi elevato. È opportuno sottolineare che l'impiego di una rete SDN comporta la presenza di complessità non

indifferenti derivate dal corretto funzionamento dei servizi presenti all'interno del controller.

2.1.4 Problematiche legate all'utilizzo di SDN

L'introduzione del modello SDN porta chiaramente dei vantaggi in termini di gestione, performance e reattività delle risorse di rete. D'altra parte però il debutto di una nuova tecnologia se da un lato risolve delle problematiche dall'altro può generarne di nuove. In particolare le sfide di primaria importanza che ogni organizzazione deve affrontare in caso di adozione di una rete SDN sono [17]:

- **Scalability**: definisce la capacità di una rete SDN di gestire ed elaborare il traffico a fronte di un crescente carico di lavoro. Soddisfare questo requisito comporta l'implementazione di tecniche come decentralizzazione delle responsabilità, clustering e funzionalità di processamento elevato per far fronte ad alti carichi.
- **Reliability**: il controller SDN viene considerato affidabile quando è in grado di notificare in tempo reale eventuali fallimenti nell'invio dei dati. In questo tipo di reti deve essere garantita una soglia minima di affidabilità nella consegna di dati critici. Nelle implementazioni attuali, i controller SDN devono essere capaci di soddisfare requisiti minimi di consegna delle informazioni affidabile e tempestiva.
- **High Availability**: il controller deve essere sempre disponibile ogni qualvolta un cliente, applicazione, dispositivo o amministratore di rete che sia, richiede le sue funzionalità.
- **Elasticity**: è la capacità di un controller SDN di adattare dinamicamente la sua capacità scalando superiormente o inferiormente le risorse disponibili in modo da far fronte alle continue variazioni del carico di lavoro.
- **Security**: la sicurezza in SDN consiste nella protezione delle informazioni da furto o danneggiamento delle risorse hardware o software. Una rete SDN si dice sicura quando l'hardware è messo fisicamente in sicurezza e quando il software è logicamente dotato di meccanismi atti a prevenire minacce provenienti dalla rete. Le vulnerabilità di SDN sono delle porte di accesso ad eventuali attacchi di sicurezza di natura intenzionale o accidentale.

- **Resilience**: la resilienza in SDN è la capacità di mantenere uno standard minimo accettabile del livello generale di servizio anche in caso di fallimento di un singolo servizio, della rete o di un nodo. Quando un elemento di SDN va incontro ad un fallimento, la rete dovrebbe continuare a fornire il servizio con le stesse prestazioni.
- **Dependability**: il termine dependability è strettamente connesso ai concetti di availability e reliability. Il soddisfacimento della dependability si riferisce alla prevenzione dei guasti e all'implementazione di meccanismi di fault tolerance per garantire il livello operativo delle funzionalità SDN anche a fronte di un degrado delle risorse.

2.2 OpenFlow

Dopo aver introdotto i principi alla base di SDN, in questa sezione si illustrerà OpenFlow, il protocollo standard più famoso operante al livello southbound che consente ad un controller SDN di interagire con dispositivi di rete come router e switch di produttori diversi. Questo protocollo, largamente utilizzato nelle attuali reti SDN, fornisce una soluzione per controllore il comportamento degli apparati di rete rendendoli dinamici e programmabili.

Le origini di OpenFlow risalgono al 2006 quando Martin Casado, durante il suo PhD all'Università di Stanford, sviluppò il progetto Ethane. L'obiettivo di Ethane era quello di creare un'architettura di rete di livello enterprise in grado di centralizzare la gestione delle risorse di rete in modo da facilitare la definizione e l'attivazione di politiche raffinate per tutti i nodi della rete [19]. L'intuizione di Casado fu sviluppata e migliorata da un lavoro di ricerca congiunto delle Università di Stanford e Berkeley il quale diede vita ad OpenFlow. Infine il progetto passò sotto la supervisione della Open Networking Foundation la quale iniziò il processo di standardizzazione che culminò con il rilascio della prima versione di OpenFlow nel Febbraio 2011 [20]. Di seguito si farà riferimento alla versione 1.5.1, ultima disponibile. [21]

Prima di procedere con la descrizione del protocollo è importante sottolineare due punti. Il primo riguarda l'utilizzo del termine OpenFlow in letteratura; molto spesso infatti ci si riferisce a questa soluzione come sinonimo di SDN. Questa associazione è errata in quanto SDN rappresenta un modello di architettura di rete mentre OpenFlow è solo una delle possibili soluzioni utilizzabili per la realizzazione dell'interfaccia southbound. Da questa osservazione deriva il secondo punto: OpenFlow non è l'unica soluzione utilizzabile

per l'interfaccia southbound; ne esistono altre che svolgono funzionalità simili e che per completezza sono citate di seguito:

- **Forwarding and Control Element Separation** (ForCes) standardizzato da IETF e definito nel documento RFC 5810 [22].
- **Path Computation Element** definito nel documento RFC 5440 [23].
- **Locator/ID Separation Protocol** (LISP) di Cisco definito nel documento RFC 6830 [24].
- L'architettura **SoftRouter** di Microsoft [25].

La definizione di OpenFlow data dall'organo ONF è la seguente:

"…la prima interfaccia di comunicazione standard tra il piano di controllo e il piano dati di una architettura SDN. Permette l'accesso diretto e la manipolazione del piano di dati di un dispositivo di rete…"

Nel protocollo OpenFlow ci sono due attori principali:

- Il *controller OpenFlow*.
- Lo *switch OpenFlow*.

Il controller è un modulo software in grado di interfacciarsi con gli switch OpenFlow usando il protocollo. Solitamente è capace di controllare più dispositivi contemporaneamente. Questa interazione avviene attraverso una interfaccia southbound logicamente identificata come OpenFlow Channel.

Il secondo è uno switch logico, costituito delle risorse gestite da OpenFlow. Come mostrato in Figura 20 le risorse a disposizione di uno switch sono:

- Uno o più *canali* per la comunicazione tra switch e controller.
- Una o più *porte* attraverso le quali i pacchetti attraversano lo switch in ingresso e in uscita.
- Una o più *flow table* e una *group table* per consentire il riconoscimento e di conseguenza l'elaborazione dei pacchetti in transito.

È possibile distinguere lo switch in due tipologie:

- *OpenFlow-only*.
- *OpenFlow-hybrid*.

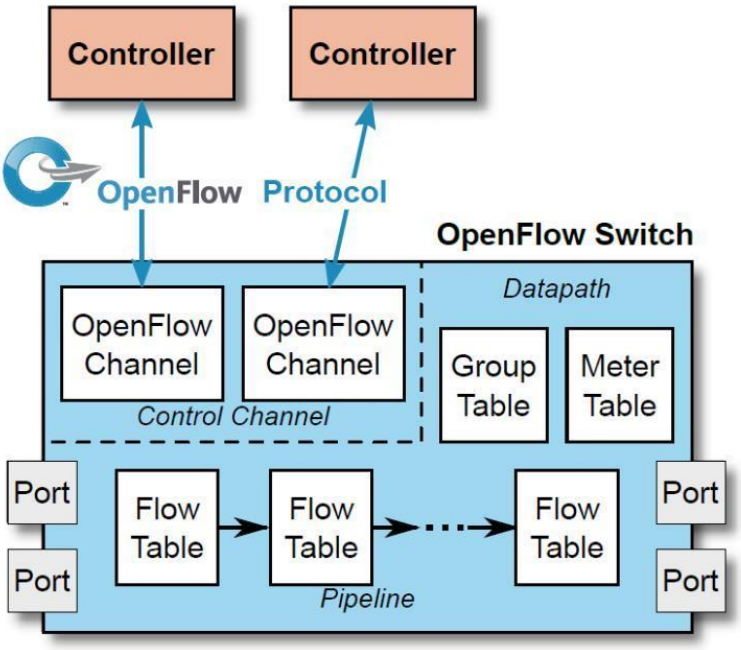

Figura 20 - Switch OpenFlow.

La differenza tra i due consiste nella capacità di interagire con più domini di rete: le risorse del primo sono completamente gestite dal protocollo OpenFlow mentre il secondo oltre alle funzionalità OpenFlow può agire autonomamente come un apparato di rete comune e può quindi operare come ponte tra una rete SDN e una rete tradizionale.

In prima analisi il nome switch OpenFlow può risultare fuorviante. Il termine switch usato nella specifica non va inteso come sinonimo di Livello 2 ma in senso più generale ovvero come dispositivo che si occupa dello smistamento del traffico. Facendo riferimento alle reti tradizionali uno switch OpenFlow può essere un router o uno switch.

2.2.1 Canali OpenFlow

Il canale OpenFlow è il mezzo attraverso il quale il controller e lo switch possono scambiarsi informazioni. L'associazione tra un controller e uno switch non è necessariamente di tipo uno-a-uno. È infatti possibile che uno switch possa essere gestito contemporaneamente da più controller in caso di clustering per soddisfare i requisiti di high availability e scalability. In caso di controller multipli i canali di comunicazione sono esclusivi, ogni controller ha un canale di comunicazione dedicato con ciascun switch.

L'interazione tra un controller e uno switch può essere di tre tipologie:

- *Controller-to-switch.*

- *Asynchronous*.
- *Symmetric*.

La comunicazione *Controller-to-switch* parte dal controller e può essere sincrona o asincrona. Di questa tipologia fanno parte quei messaggi che sondano le funzionalità, le configurazioni o lo stato di uno switch. Il messaggio più importante impiegato in questo tipo di comunicazione è sicuramente il messaggio di inoltro di un pacchetto che consente l'instradamento. In dettaglio i messaggi scambiati utilizzando questo tipo di comunicazione sono:

- **Features**: con questo messaggio il controller può richiedere allo switch logico di ottenere informazioni basilari come la sua identità e quali funzionalità è in grado di supportare. Questo tipo di richiesta è comunemente effettuata in fase di inizializzazione del canale OpenFlow.
- **Configuration**: il controller può leggere e scrivere parametri di configurazione presenti all'interno dello switch.
- **Modify-State**: messaggio inviato dal controller per la gestione dello stato corrente degli switch. Lo scopo principale di questo messaggio è l'aggiunta, rimozione o modifica di Flow Entry e Group Entry, l'aggiunta o rimozione di action bucket nelle Group Entry e la manipolazione delle proprietà associate alle porte dello switch.
- **Read-State**: messaggio impiegato dal controller per ottenere informazioni dallo switch quali configurazione corrente, dati statistici e funzionalità supportate.
- **Packet-Out**: tramite questo messaggio il controller ha l'abilità di iniettare pacchetti nel data plane di uno switch e di specificare quale porta di uscita dovrà utilizzare per l'inoltro di tali pacchetti.
- **Barrier**: utilizzato dal controller per assicurarsi che le dipendenze richieste da alcuni messaggi siano rispettate e per ricevere da parte dello switch notifiche di completamento delle operazioni di interesse.
- **Role-Request**: utilizzato dal controller per impostare o richiedere il ruolo del suo canale OpenFlow o il proprio Controller ID. Utile nell'eventualità in cui uno switch sia associato a più di un controller.

La comunicazione di tipo *Asynchronous* parte su iniziativa dello switch, il quale può inviare al controller aggiornamenti sullo stato corrente delle sue risorse. Messaggi relativi allo stato possono riguardare ad esempio le porte e i flussi, fa parte di questa categoria anche l'invio di un pacchetto al controller. I messaggi appartenenti a questa tipologia sono:

- **Packet-in**: il messaggio di tipo *Packet-In* è il modo che uno switch ha a disposizione per inviare un pacchetto in attraversamento al controller. Ci sono due motivazioni che portano all'utilizzo di questo messaggio: potrebbe esserci un'azione esplicita di invio del pacchetto al controller risultante da un match nella Flow Table oppure nel caso in cui lo switch non ha informazioni a sufficienza su come trattare il pacchetto in questione e demanda il suo processamento al controller.
- **Flow-removed**: notifica il controller dell'avvenuta rimozione di una Flow Entry in una Flow Table. Questo messaggio viene inviato come risposta in caso di richiesta esplicita di rimozione della Flow Entry da parte del controller oppure a seguito della scadenza di uno dei time-out associati al flow.
- **Port-status**: il comportamento che si aspetta da uno switch è quello di notificare il controller ogni qual volta la configurazione o lo stato di una porta cambia, ad esempio a causa di un intervento manuale dell'utente o di una caduta improvvisa di un link.
- **Role-Staus**: informa il controller sulla modifica del suo ruolo. Quando un nuovo controller si elegge master autonomamente, lo switch deve inviare questo messaggio al controller precedente.
- **Controller-status**: comunica al controller un cambiamento nello stato di un canale OpenFlow. Questo messaggio risulta particolarmente utile per supportare la procedura di failover nel caso i cui i controller non fossero più in grado di comunicare tra loro.
- **Flow-monitor**: messaggio inviato al controller in caso di modifica di una Flow Table.

L'ultima tipologia di comunicazione, *Symmetric*, può essere iniziata dal controller o da uno switch ed è utile per lo scambio di informazioni di controllo. I messaggi appartenenti a tale categoria sono:

- **Hello**: messaggio scambiato tra switch e controller durante il protocollo di inizializzazione della connessione.

- **Echo**: messaggio inviato dal controller o dallo switch a cui segue obbligatoriamente un messaggio dello stesso tipo in risposta, utile per assicurarsi la presenza della connessione tra controller e switch o per misurare la latenza o la larghezza di banda della connessione.
- **Error**: consente ad uno dei due end-point della connessione di notificare all'altro il verificarsi di un errore. Spesso impiegato da uno switch per notificare il controller del fallimento di una sua richiesta.
- **Experimenter**: fornisce una modalità standard per consentire agli switch OpenFlow di scambiare con il controller messaggi appartenenti a funzionalità non ancora parte della specifica.

Per la comunicazione tra controller e switch, OpenFlow garantisce la consegna di tutti i messaggi inviati, che possono essere raggruppati in messaggi *Bundle*. Per quanto invece riguarda l'ordinamento dei messaggi è possibile ottenerlo utilizzando messaggi *Barrier*.

Uno switch riconosce una connessione con un controller attraverso un identificatore univoco chiamato *Connection URI*. Questo URI deve essere conforme alla sintassi specificata dal documento RFC 3986 e può avere una delle seguenti forme:

- *protocol:name-or-address:port*
- *protocol:name-or-address*

I valori ammisibili per *protocol* possono essere *tcp* o *udp* per le connessioni principali o *tcp*, *udp*, *tls* o *dtls* per connessioni ausiliarie. Per quanto riguarda il campo *name-or-address* i valori ammissibili sono l'hostname o l'indirizzo IP del controller. Il campo *port* indica la porta della connessione al controller.

Una volta stabilita la connessione tra switch e controller, le due parti iniziano il protocollo di negoziazione scambiandosi messaggi di tipo *Hello* contenente la più alta versione del protocollo supportata; in caso di successo è possibile procedere alla comunicazione di tipo operativo. In caso di caduta della connessione tra controller e switch, quest'ultimo tramite il messaggio *Controller-status* deve notificare tale evento a tutti gli altri controller. Nel caso critico di interruzione della comunicazione con tutti i controller lo switch deve entrare immediatamente, a seconda della sua configurazione e implementazione, o in *fail secure mode* o in *fail standalone mode*. Nella prima modalità l'unica modifica al suo comportamento consiste nello scartare tutti i pacchetti e i messaggi destinati al controller; nella seconda modalità lo switch

si comporta come uno switch o un router Ethernet di tipo legacy, modalità disponibile solo in caso di *switch OpenFlow-hybrid*.

2.2.2 Porte standard

Le porte di uno switch OpenFlow sono le interfacce di rete che consentono il passaggio dei pacchetti dall'esterno all'interno del dominio OpenFlow dove verranno elaborati nella pipeline. Gli switch si connettono logicamente gli uni con gli altri mediante l'utilizzo di queste porte. Un pacchetto può essere inoltrato da uno switch ad un altro solo attraverso una porta di uscita nel primo switch e una porta di ingresso nel secondo. La specifica suddivide le porte standard in tre tipologie: porte fisiche, porte logiche e porte riservate.

Le *porte fisiche* sono interfacce dello switch caratterizzate da un mapping uno-a-uno con un'interfaccia fisica del dispositivo. Questa corrispondenza consente, ad esempio, di mettere in comunicazione diretta lo switch logico con un'interfaccia verso il mondo esterno. È tuttavia possibile virtualizzare un'interfaccia fisica in modo da dedicare una parte della stessa interfaccia a più porte fisiche dello switch.

Le *porte logiche* sono al contrario delle interfacce completamente astratte non avendo alcuna controparte fisica. Una porta di questo tipo può realizzare una funzionalità di rete come un'interfaccia di loopback, un tunnel oppure fungere da multiplexer per diverse porte fisiche.

Le *porte riservate* si occupano invece di nascondere una specifica funzionalità di instradamento, specificano azioni generiche di inoltro come inviare un pacchetto al controller, effettuare il flooding o inoltrare il traffico utilizzando metodi al di fuori del dominio OpenFlow come in caso di *fail standalone mode* dove lo switch si può comportare come un apparato di rete tradizionale. Lo standard definisce alcune porte come *Required*, da implementare obbligatoriamente, e altre come *Optional*, di natura opzionale. A seconda del tipo di switch OpenFlow che si vuole realizzare dipenderà il tipo di porte che si dovranno implementare. Per esempio, implementando o meno le porte Optional si andrà a creare uno switch di tipo OpenFlow-hybrid. Esistono nove tipologie di instradamento previste da queste porte logiche di cui sei *Required* e tre *Optional*.

Le porte di tipo *Required* sono:

- **ALL**: rappresenta la strada verso tutte le interfacce disponibili. Utilizzabile solo in uscita.

- **CONTROLLER**: rappresenta il canale di comunicazione con il controller. Può essere utilizzata come porta di ingresso o di uscita. Quando questa porta viene specificata come di uscita, il pacchetto viene incapsulato in un messaggio di tipo *Packet-in* e inviato al controller. In caso questa porta venga specificata come di ingresso, identifica un pacchetto proveniente dal controller.
- **TABLE**: rappresenta l'inizio della pipeline di OpenFlow. Risulta valida solo all'interno di una lista di azioni di un messaggio di tipo *Packet-out*, al fine di indirizzare un pacchetto verso la prima tabella della pipeline la quale darà inizio alla sua elaborazione.
- **IN_PORT**: indica la porta di ingresso di un pacchetto. Può essere utilizzata solo come porta di uscita per inviare il pacchetto verso la sua porta di ingresso.
- **ANY**: valore speciale utilizzato per alcune richieste OpenFlow quando nessuna porta è stata specificata.
- **UNSET**: valore speciale per indicare che nessuna porta di uscita è stata impostata nell'*Action-Set*.

Infine le porte di tipologia *Optional* sono:

- **LOCAL**: consentono ad entità remote di interagire con lo switch e i suoi servizi di rete utilizzando la rete OpenFlow anziché comunicare per mezzo di una rete di controllo separata.
- **NORMAL**: rappresenta il canale verso le tabelle di inoltro legacy, da utilizzare in caso si voglia gestire il traffico come un dispositivo di rete tradizionale.
- **FLOOD**: consente di utilizzare il flooding utilizzando le tabelle di inoltro legacy.

Le operazioni effettuate sulle varie porte, da specifica, devono essere sempre monitorate. Ogni aggiunta, modifica o rimozione di una porta, che sia stata effettuata per ordine del controller o su iniziativa dello switch deve sempre essere notificata al controller.

2.2.3 Tabelle e Pipeline di elaborazione dei pacchetti

Le pipeline sono gli strumenti messi a disposizione dallo standard che consentono di agire sul traffico. Il traffico passante per uno switch OpenFlow viene trasportato attraverso l'impiego di due canali, o pipeline, che si

occupano di elaborarlo. Le pipeline sono chiamate rispettivamente *Ingress processing* e *Egress processing*, la prima rappresenta un canale per il traffico in entrata mentre la seconda si occupa del canale in uscita, la seconda pipeline, introdotta nell'ultima versione del protocollo, è opzionale.

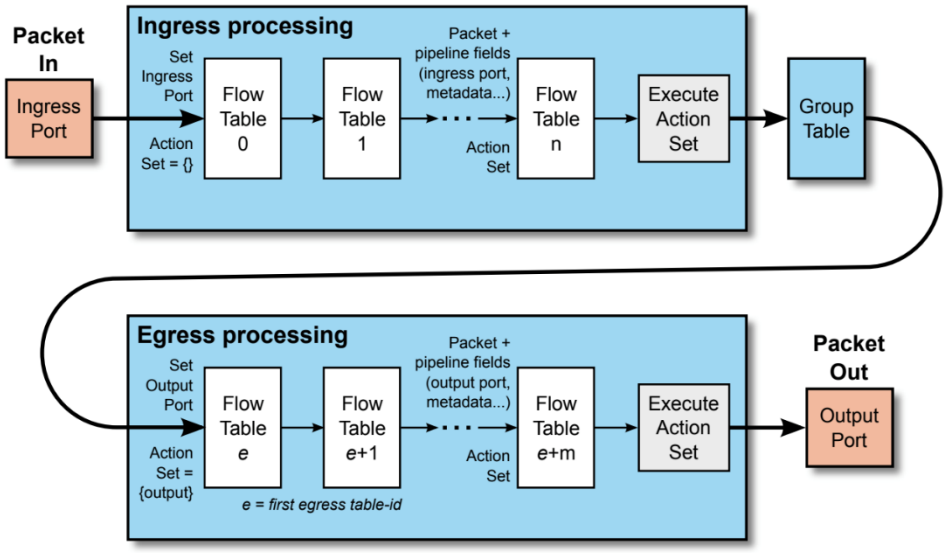

Figura 21 - Pipeline di elaborazione dei pacchetti.

Come illustrato in Figura 21 il processamento del traffico viene gestito da una serie di tabelle chiamate *Flow Table*. Le voci in questa tabella, chiamate *Flow Entry*, sono delle regole a cui è associato un *Instruction-Set* che corrisponde ad un insieme di azioni che dovranno essere eseguite in caso di corrispondenza con la specifica Flow Entry. Queste istruzioni possono andare a costruire e manipolare un insieme di azioni, denominato *Action-Set*, la cui applicazione al termine della pipeline realizza l'elaborazione effettiva del traffico.

Quando un pacchetto giunge ad una qualsiasi porta di ingresso, viene inserito nella pipeline di ingresso, in questa pipeline deve esserci sempre almeno una *Flow Table* di ingresso denominata "0" a cui è possibile concatenare una o ulteriori tabelle, da questa configurazione deriva il termine pipeline. La *Flow Table* non è altro che un contenitore logico in cui organizzare diverse regole, o *Flow Entry*, le quali svolgono l'effettiva elaborazione dei pacchetti. Alla fine di ogni *Flow Table*, è possibile definire anche una *Flow Entry* speciale chiamata *Table-Miss* che entrerà in gioco quando non è stata trovata alcuna corrispondenza per quel particolare pacchetto. La struttura di una *Flow Entry* è costituita dai seguenti elementi:

- **Match fields**: si specificano le caratteristiche che il pacchetto deve avere affinché possa essere trovata una corrispondenza con la *Flow Entry* in questione.
- **Priority**: stabilisce una relazione d'ordine tra le diverse *Flow Entry*.
- **Counters**: contatore aggiornato ogni volta che viene trovata una corrispondenza tra un pacchetto e la *Flow Entry* corrente.
- **Instructions**: istruzioni da eseguire in caso di match, possono avere effetto immediato o essere inserite nell'*Action-Set*.
- **Timeouts**: tempo massimo di validità o di inattività dopo il quale il flow viene considerato non più valido dallo switch.
- **Cookie**: informazione non utilizzata per l'elaborazione dei pacchetti. È un valore opaco scelto dal controller, può essere utilizzato da quest'ultimo per filtrare le *Flow Entry* che presentano determinate statistiche o che sono interessate da richieste di modifica o di eliminazione.
- **Flags**: alterano le modalità con le quali le *Flow Entry* sono gestite.

È possibile identificare univocamente una Flow Entry attraverso i campi *Match Fields* e *Priority* in particolare nel primo sono definiti quei parametri utilizzati per discriminare l'appartenenza di un pacchetto ad una determinata *Flow Entry* e quindi di capire cosa filtrare, come e con quale granularità. I parametri configurabili sono numerosi; la specifica infatti ne definisce 44, non è necessario definirli tutti ma solo il sottoinsieme utile a riconoscere il pacchetto che varia da caso a caso. Un pacchetto viene associato ad una determinata Flow Entry se e solo se soddisfa tutte le condizioni specificate nel campo *Match Fields*. Alcuni di questi parametri sono:

- Porta di ingresso dello switch.
- Indirizzo Ethernet di destinazione.
- Indirizzo Ethernet sorgente.
- VLAN ID.
- Indirizzo IPv4/IPv6 sorgente/destinazione.
- Porta UDP sorgente/destinazione.
- ARP sorgente/destinazione IPv4, IPv6, MAC.

- Flag TCP.
- Codice ICMP.
- Tipo ICMPv6.

Il campo *Priority* entra in gioco in caso di corrispondenza sovrapposta su più Flow Entry e specifica quale Flow Entry abbia precedenza sull'altra. A questo campo è associato un valore di 2 byte, pertanto è possibile definire fino a 65535 possibili priorità. Al valore maggiore corrisponde una priorità maggiore. Una volta determinata la Flow Entry di appartenenza per un pacchetto, viene incrementato il valore del campo counter della Flow Entry. Successivamente si eseguono le *Instruction* contenute nel rispettivo campo. Le azioni possibili sono diverse, si possono ad esempio effettuare modifiche al pacchetto, inoltrarlo verso un'altra *Flow Table*, modificare il suo *Action-Set* o operare su altri metadati. Per quanto riguarda l'invio del pacchetto ad altre *Flow Table*, è importante specificare che non è mai possibile far tornare un pacchetto in una tabella precedente e logicamente le *Flow Entry* dell'ultima tabella non potranno mai inviare il pacchetto ad un'altra tabella. La gestione dell'*Action-Set* è fondamentale in questa fase, in quanto all'interno di questo insieme vengono accumulate tutte le elaborazioni da effettuare sul pacchetto, prima di concludere il suo percorso nella pipeline. Una volta che il pacchetto ha terminato il suo percorso all'interno della pipeline è pronto per essere elaborato dallo switch. L'*Action-Set* durante l'attraversamento è stato incrementato, decrementato e ripulito nel numero di volte stabilito dalle *Instruction* presenti nelle *Flow Entry*. Le azioni da svolgere sul pacchetto devono essere eseguite nell'ordine stabilito a monte. Alcune di queste azioni sono:

- **Copy TTL inwards**: copia il valore Time To Live dall'header più esterno all'header successivo più vicino; ad esempio da un header IP ad un altro, da un header MPLS ad un altro oppure da un header MPLS ad uno IP.
- **Copy TTL outwards**: copia il valore Time To Live dall'header di riferimento all'header più esterno; ad esempio da un header IP ad un altro, da un header MPLS ad un altro oppure da un header IP ad uno MPLS.
- **Push/Pop MPLS**: rimuove l'header MPLS dal pacchetto corrente.
- **Push MPLS**: inserisce un header MPLS nel pacchetto corrente.
- **Decrement IPv4 TTL**: decrementa il valore di Time To Live dell'header più esterno del pacchetto.

- **Set IPv4 Source Address, Set IPv4 Destination Address, Set Ipv4 ToS bits, Set IPv4 ECN bits**: sostituiscono i campi appropriati nell'header IP più esterno e aggiorna il valore di checksum.
- **Output to Switch Port**: porta da utilizzare per l'inoltro in uscita del pacchetto.

Tra queste azioni eseguibili ci sono ad esempio la manipolazione dei TTL, il pop e il push di alcune tipologie di tag come VLAN e MPLS e soprattutto l'inoltro del pacchetto in uscita per mezzo delle azioni Group e Output.

2.3 Segment Routing in contesto SDN

Nel primo capitolo è stato introdotto il funzionamento di base del Segment Routing mostrando un esempio pratico in uno scenario di rete tradizionale. La particolarità di questa soluzione è quella di essere stata progettata per essere compatibile con il modello SDN e rappresenta il meccanismo di base per le applicazioni cosiddette AER, Application Egineered Routing, caratterizzate dalla capacità di comunicare alla rete i requisiti che devono essere soddisfatti affinchè possano inoltrare il traffico [26]. SR a differenza degli altri protocolli di routing rappresenta un ponte tra le reti tradizionali dove l'intelligenza della rete è distribuita e il modello SDN caratterizzato dalla centralizzazione dell'intelligenza di rete. L'immagine seguente illustra un semplice scenario SDN nel quale il controller centralizzato consente la comunicazione tra un router A e un router B.

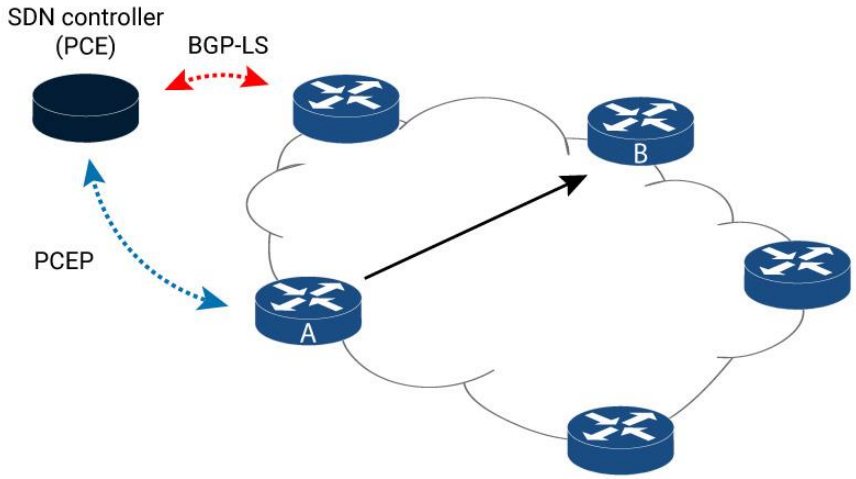

Figura 22 - Segment Routing in un dominio SDN.

In questo breve esempio, il controller ha una visione globale della topologia di rete e dei flow. Un router A può quindi richiedere un percorso verso

una destinazione B con certe caratteristiche, per esempio il rispetto di requisiti come latenza o banda. Il controller calcola un percorso ottimale e restituisce al router la corrispondente lista di segmenti in termini di stack di etichette MPLS. A questo punto il nodo A può iniettare il traffico verso B utilizzando la lista di segmenti senza inviare ulteriori messaggi di segnalazione nella rete. Queste liste di segmenti permettono una virtualizzazione completa della rete senza appesantire la rete con informazioni di stato dell'applicazione, il quale è codificato nel pacchetto grazie a tali liste. Questa informazione essendo molto leggera consente alla rete di supportare un numero enorme di richieste per tutte le applicazioni senza aggiungere un overhead significativo nello scambio effettivo dei dati tra i diversi nodi della rete [12].

Capitolo 3
Middleware RAMP

Prima di procedere alla descrizione del lavoro svolto, inerente agli argomenti trattati nei capitoli precedenti, è necessario descrivere il sistema da cui si è partiti per realizzare la soluzione proposta. Tale sistema è Real Ad-Hoc Multi-hop Peer-to-peer (RAMP), un middleware per la gestione dinamica e flessibile di reti spontanee e opportunistiche contraddistinte da eterogeneità di vario tipo. [27]

Lo scenario considerato da questo middleware è quello di permettere l'interazione tra un numero non troppo elevato di dispositivi mobili anche a fronte di alta mobilità utilizzando il paradigma di Store, Carry e Forward. RAMP svolge le sue funzionalità al livello applicativo e sfruttando le tecnologie di connettività fisica disponibili localmente ai nodi e i protocolli di Data Link maggiormente diffusi effettua un routing a livello di middleware, tale soluzione consente di essere indipendenti dal sistema operativo sottostante garantendo interoperabilità tra diversi sistemi e configurazioni. Inoltre tale instradamento cerca di soddisfare al meglio le caratteristiche della comunicazione alle necessità richieste dal livello applicativo, supportando casi d'uso e situazioni di diverso tipo.

Gli obiettivi che RAMP si pone di raggiungere per risolvere le problematiche derivanti da questo scenario sono:

- Sfruttare al massimo le tecnologie wireless comunemente diffuse con una gestione effettuata a livello applicativo in modo che il sistema possa essere adottato velocemente e facilmente.
- Supportare l'utilizzo concorrente di interfacce di rete wireless eterogenee presenti all'interno di uno stesso nodo come ad esempio l'impiego simultaneo di connessioni IEEE 802.11 e Bluetooth.
- Data la mobilità dei nodi e la dinamicità dei loro collegamenti, il sistema deve utilizzare un meccanismo di discovery in grado di rilevare prontamente e con un overhead limitato i nodi e i servizi che diventano disponibili a run-time.
- Offrire delle funzionalità che consentano il supporto di una vasta gamma di applicazioni peer-to-peer come messaggistica, file sharing e streaming audio-video.

In questo capitolo si procederà con una descrizione generale del contesto in cui RAMP opera ovvero le reti spontanee e opportunistiche, si passerà poi alla descrizione della sua architettura di base per poi concludere con la descrizione di una sua estensione in grado di organizzare e gestire una rete di dispositivi secondo un paradigma SDN-like, la quale rappresenta il punto di partenza per lo sviluppo di questo lavoro.

3.2 Reti spontanee e opportunistiche

Nel capitolo relativo al routing si è introdotto il concetto di MANET, per loro natura etereogenee e comprensive di numerose sottotipologie dedicate al soddisfacimento di requisiti differenti e in generale caratterizzate da un'assenza di infrastruttura che gestisca le connessioni e le comunicazioni tra i nodi della rete. Come già evidenziato, la soluzione a tale mancanza è l'impiego di protocolli di routing distribuiti, come ad esempio il DSR. Tra le sovracitate tipologie di MANET, il middleware RAMP si occupa in particolare di gestire reti spontanee e opportunistiche.

Le reti spontanee possono essere definite in contesti con un numero limitato di nodi presenti nella stessa località allo scopo di interagire socialmente. Tale tipologia di rete è utile nel caso in cui un utente abbia necessità di ottenere determinate informazioni o servizi disponibili solo in una certa area. Quando il dispositivo dell'utente accede a tale zona, entra automaticamente a far parte della rete e vi permane per il tempo necessario al reperimento dell'informazione richiesta. I requisiti di comunicazione non sono particolarmente vincolanti in termini di affidabilità. Inoltre, si assume che durante la comunicazione e per il tipo di informazioni trasmesse, la mobilità dei nodi sia necessariamente limitata. Dispositivi tipicamente associati a questo tipo di rete sono quelli dell'elettronica di consumo come smartphone, tablet e laptot, dotati di più interfacce di comunicazione wireless. Per tale motivo è necessario coordinarne l'utilizzo quanto più efficientemente possibile tramite adeguate modalità di gestione, per massimizzare le opportunità di comunicazione.

Se le reti spontanee si basano sull'assunzione che i nodi siano caratterizzati da mobilità ridotta, tale assunzione cade nel caso delle reti opportunistiche, nelle quali non è detto che fin da subito ci siano dei percorsi che interconnettono i nodi che intendono comunicare. Nelle reti opportunistiche fattori come mobilità e dinamicità influenzano in modo significativo le soluzioni da adottare per consentire lo scambio di informazioni. Lo scopo principale di tale tipologia di rete è quello di aumentare al massimo le possibilità di inoltrare i

pacchetti quanto più in prossimità ai nodi di destinazione, considerando che durante la comunicazione può verificarsi un'assenza improvvisa di connessione o abbandono da parte di un nodo dalla rete, o altri eventi dipendenti dalla volatilità della rete che possono causare l'interruzione delle comunicazioni. Per far fronte a tali problematiche si fa ricorso al paradigma di Store, Carry e Forward e di replicazione dei pacchetti al fine di avvicinarli il più possibile alla destinazione, ad esempio inviandoli a dei nodi caratterizzati da forte mobilità e con un'alta probabilità di raggiungerla.

3.3 Architettura

RAMP è caratterizzato da un'architettura sviluppata su due livelli:

- il *Service Layer* si occupa della gestione dei servizi peer-to-peer tramite funzionalità di registrazione, di advertising e di discovery.
- il *Core Layer* fornisce le astrazioni necessarie alla comunicazione end-to-end di tipo unicast e broadcast.

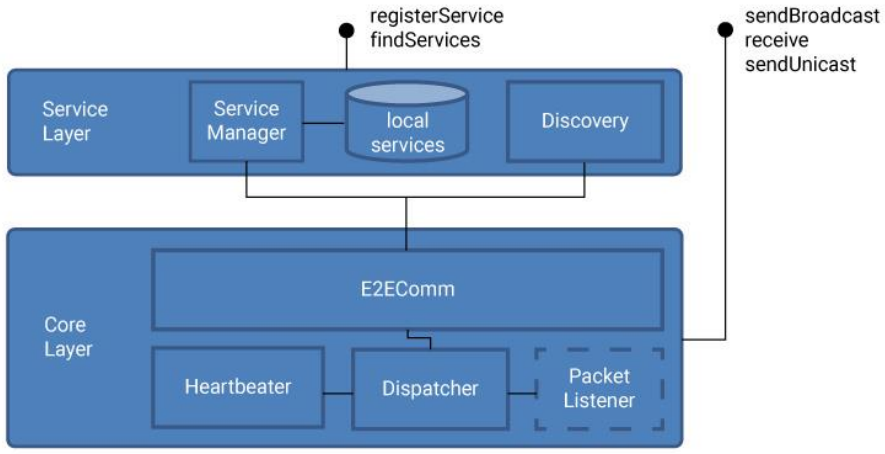

Figura 23 - Architettura RAMP.

3.3.1 Service Layer

Il Service Layer supporta la registrazione e la scoperta dinamica di servizi peer-to-peer che possono sfruttare le astrazioni messe a disposizione del Core Layer per comunicare. Questo livello è composto da due elementi principali, il *Service Manager* e il *Service Discovery*.

Il Service Manager permette alle applicazioni che utilizzano RAMP di pubblicare, tramite registrazione, un servizio offerto di qualunque tipologia. In fase di registrazione l'applicazione deve fornire alcune informazioni per identificare e localizzare il servizio, in particolare:

- Nome del servizio.
- Indirizzo IP e porta a cui è possibile accedere al servizio.

Tali informazioni sono salvate in un database locale a questo componente, il quale rimane in ascolto per eventuali richieste broadcast di servizi da parte di tutti gli altri nodi della rete. Ogni volta che arriva una richiesta per un determinato servizio, il Service Manager consulta il suo database locale e in caso venga trovata una corrispondenza invia in risposta al nodo richiedente un pacchetto unicast contenente tutte le informazioni utili all'invocazione del servizio.

Il Service Discovery è il componente duale che consente di scoprire un servizio remoto disponibile nella rete. Il ritrovamento di un servizio si basa su un protocollo distribuito basato su pacchetti di tipo broadcast, i quali attraversano la rete fino a quando il servizio richiesto non viene trovato. In caso di mancata corrispondenza per evitare una propagazione infinita dei pacchetti all'interno della rete si utilizza un meccanismo di Time-to-Live.

3.3.2 Core Layer

Il *Core Layer* si occupa di offrire i meccanismi a supporto della comunicazione tra i vari nodi della rete. I componenti che costituiscono questo livello sono *E2EComm*, *Dispatcher* e *Heartbeater*.

E2EComm fornisce le primitive per l'invio e la ricezione di messaggi sia di tipo unicast multi-hop che di tipo broadcast. In caso di messaggi broadcast è previsto l'utilizzo di un Time-to-Live.

Il Dispatcher si occupa di realizzare le operazioni di comunicazioni vere e proprie tra nodi a distanza single-hop. Ogni volta che il Dispatcher invia un pacchetto ad un nodo remoto a seconda del protocollo di trasporto selezionato crea una nuova socket UDP o TCP sul link single-hop. Questo componente sfrutta socket diverse per inviare pacchetti diversi in modo da migliorare le prestazioni in caso di trasmissione concorrente di pacchetti multipli. In fase di ricezione il Dispatcher gestisce i messaggi a seconda della loro tipologia, in caso di messaggio unicast:

- Se il nodo locale è la destinazione specificata dal pacchetto, lo inoltra all'indirizzo localhost alla porta di destinazione.
- Altrimenti, inoltra il pacchetto al Dispatcher del nodo successivo.

In caso di messaggio broadcast:

- Se il nodo locale non è la sorgente, decrementa il Time-to-Live del pacchetto e lo inoltra all'indirizzo localhost alla porta di destinazione.
- Se il Time-to-Live è maggiore di zero, recupera dal componente Heartbeater le informazioni relative ai nodi vicini e inoltra il pacchetto a tutti i vicini assicurandosi di escludere quei vicini che si trovano nella stessa sotto-rete della sorgente.

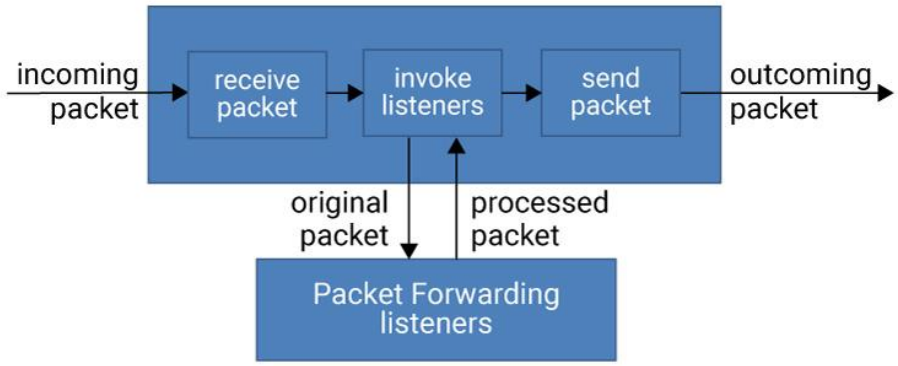

Figura 24 - Meccanismo a plug-in del Dispatcher.

Il Dispatcher inoltre offre un meccanismo a plug-in per inserire comportamento aggiuntivo al suo normale processamento dei pacchetti. Per utilizzare questa funzionalità il Dispatcher consente la registrazione di componenti che implementano un'opportuna interfaccia chiamata *PacketForwardingListener*. In questo modo ogni volta che arriva un nuovo pacchetto, questo viene inoltrato ai componenti registrati i quali una volta applicata la logica relativa restituiscono il controllo al Dispatcher che può così proseguire il normale flusso di processamento. Questa funzionalità è molto utile, come verrà mostrato nel capitolo finale, perché rende estendibile la gestione di un pacchetto senza necessità di apportare modifiche profonde al sistema.

Il componente Heartbeater ha la responsabilità di tenere traccia dell'insieme dei vicini single-hop raggiungibili dal nodo locale. Al fine di scoprire questi nodi, l'Heartbeater invia periodicamente una richiesta UDP all'indirizzo 255.255.255.255 alla quale i nodi RAMP presenti nella stessa sottorete sono tenuti a rispondere.

3.3.3 Strategie di invio e ricezione dei pacchetti

Come accennato in precedenza RAMP adatta le sue strategie di gestione dei pacchetti per soddisfare al meglio i requisiti di comunicazione specificati da una applicazione. Per farlo il sistema effettua una pacchettizzazione a livello di middleware in modo da evitare l'invio e la ricezione di un intero messaggio in una singola trasmissione single-hop, in particolare un'applicazione che utilizza RAMP per comunicare può scegliere tre strategie diverse di invio e ricezione dei pacchetti le quali possono impiegate simultaneamente all'interno della stessa rete spontanea. Queste strategie non sono le uniche possibili, è possibile infatti aggiungerne di ulteriori grazie all'architettura a plug-in del middleware che grazie alla sua estendibilità consente facilmente l'aggiunta di nuovi comportamenti. Le strategie di default che RAMP mette a disposizione sono:

- *Nonreliable Bulk Transfer* (NBRT).
- *Reliable Packet Streaming* (RPS).
- *High Reliable Message Delivery* (HRMD).

La strategia NRBT è pensata per comunicazioni ad alte prestazioni e con basso overhead dove l'affidabilità non è un requisito stringente. È ideale in caso di trasferimento di file bulk in un canale di comunicazione a bassa latenza e comporta un overhead limitato per i nodi partecipanti. Non considerando l'affidabilità come obiettivo principale, in caso di fallimento dovuto ad esempio alla mobilità dei nodi intermedi l'utente è tenuto a iniziare esplicitamente una nuova comunicazione dato che non è previsto nessun intervento da parte del middleware per far fronte a tale fallimento.

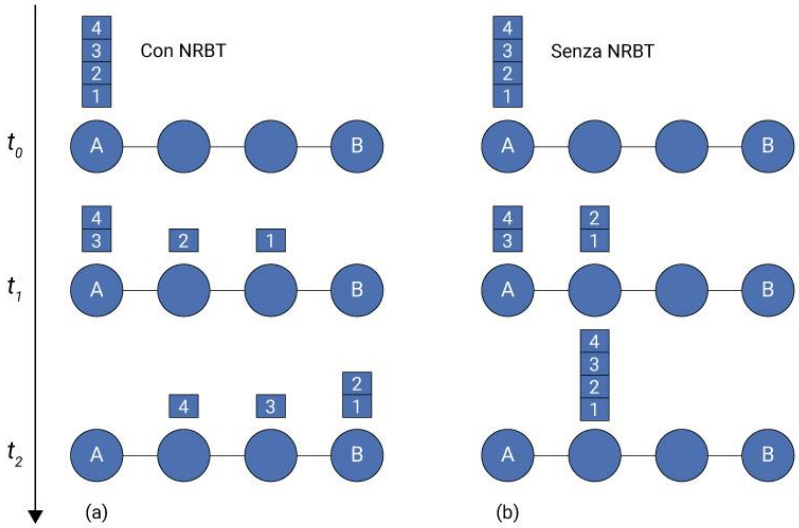

Figura 25 - Strategia Non Reliable Bulk Transfer (NRBT).

Quando un'applicazione decide di utilizzare questa strategia, RAMP trasparentemente esegue una segmentazione dei pacchetti in parti più piccole, dette chunk, le quali sono inviate da sorgente a destinazione in modo ordinato e concorrente. Normalmente quello che accadrebbe senza l'utilizzo di questa tecnica è un trasferimento hop-by-hop dell'intero messaggio. Questa pacchettizzazione che di fatto organizza i chunk in pipeline consente ad un nodo intermedio di inviare al nodo successivo solo una sequenza ordinata di parti più piccole senza dover attendere l'arrivo di tutti i chunk rimanenti ottenendo performance in termini di latenza più elevate. Inoltre tale strategia si traduce in un'occupazione delle risorse di memoria limitata in quanto ogni nodo intermedio non deve allocare un buffer di dimensioni significative per ospitare l'intero messaggio ma solo quello necessario a contenere un chunk.

La strategia RPS cerca di far fronte ad eventuali disconnessioni per mezzo di risorse addizionali, seppur limitate, dei nodi partecipanti. Risulta utile in scenari di streaming multimediale, infatti rispetto a NRBT in caso di improvvisa interruzione del percorso corrente è in grado di scoprirne uno nuovo in modo del tutto trasparente. Questa modalità permette di conseguenza di rispettare il requisito di affidabilità in caso di variazione limitata della topologia di rete attraverso una gestione automatizzata dei percorsi effettuata a livello di middleware.

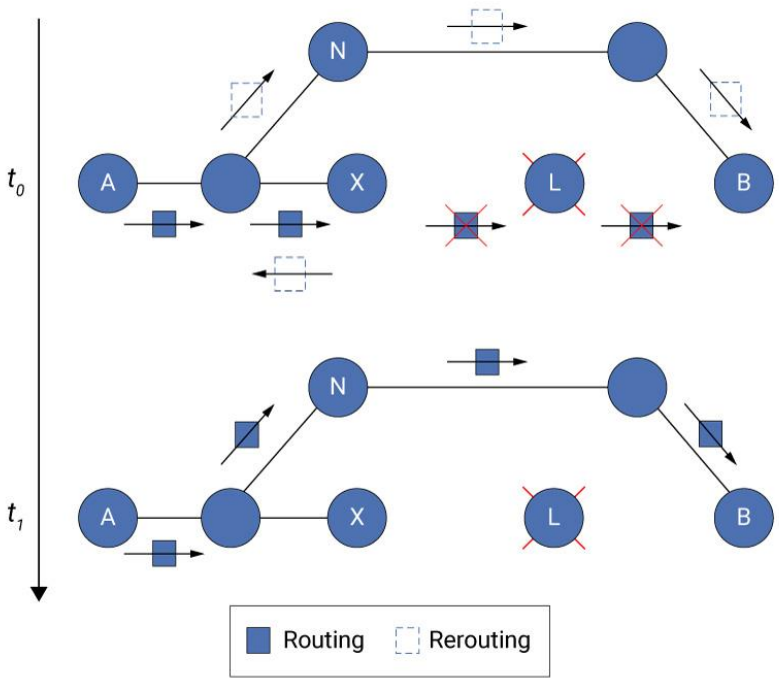

Figura 26 - Strategia Reliable Packet Streaming (RPS).

Quando un'applicazione utilizza questa strategia, RAMP adotta un approccio distribuito e leggero per inoltrare i pacchetti lungo un nuovo percorso in caso di fallimento di un nodo intermedio. In particolare quando un nodo scopre che la strada utilizzata fino a quel momento non è più percorribile, in completa autonomia e in modo reattivo cerca un nuovo segmento verso la destinazione e una volta trovato inizia a inoltrare i pacchetti in arrivo verso il nuovo percorso. In parallelo, nodo in questione che al momento è l'unico a conoscenza di tale interruzione notifica la sorgente inviandogli il nuovo percorso da utilizzare in modo che possa utilizzarlo per continuare il trasferimento di dati. In caso non venga trovata un'alternativa valida per raggiungere la destinazione, i pacchetti in arrivo verranno scartati. Se comparata con NRBT, RPS introduce un overhead addizionale localizzato ai nodi vicini al fallimento del link solo quando tale fallimento si verifica. L'overhead consiste nelle operazioni di scoperta di un nuovo percorso, della sua notifica alla sorgente e di modifica dell'header dei pacchetti attualmente disponibili localmente affinchè il loro rerouting segua la strada appena trovata. Queste operazioni come detto in precedenza vengono effettuate una sola volta solo in caso di caduta di un link, in particolare per quanto riguarda l'ultima sarà temporanea fino a quando la sorgente non utilizzerà il nuovo percorso valido.

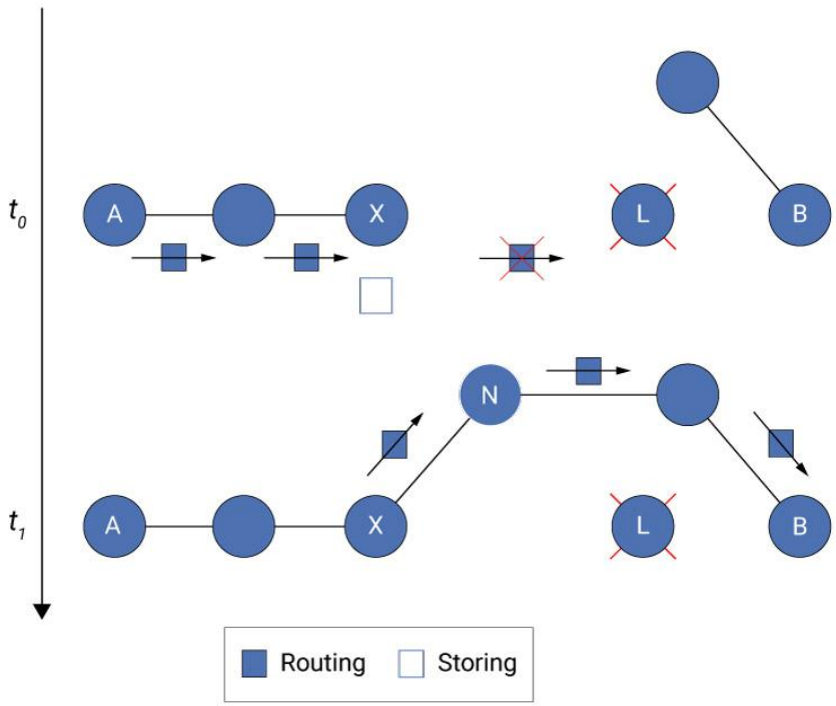

Figura 27 - Strategia High Reliable Message Delivery (HRMD).

La terza strategia, HRMD, ha come obiettivo quello di massimizzare le probabilità di invio di un messaggio verso la sua destinazione anche quando un percorso valido per raggiungerla non è al momento disponibile. Per attuare questa strategia RAMP utilizza il paradigma di Store, Carry e Forward, in particolare quando attiva e non è presente nessuna strada verso la destinazione il nodo intermediario salva temporaneamente in memoria il pacchetto in modo che possa riprovare a inoltrarlo successivamente. A fronte di una modifica della topologia dovuta alla mobilità del nodo intermedio che adesso è in visibilità della destinazione oppure alla presenza di un nuovo nodo che ha eliminato la partizione della rete, RAMP può procedere alla consegna trasparente del pacchetto attraverso una ritrasmissione effettuata a livello di middleware. Queste ritrasmissioni sono effettuate un certo numero di volte al termine delle quali il pacchetto viene scartato.

Se confrontata con le due strategie precedenti, HRMD comporta l'introduzione di un overhead aggiuntivo dovuto alla capacità di mantenere in memoria centrale i pacchetti da inviare e al calcolo del percorso verso la destinazione una volta disponibile. Pertanto come modalità di comunicazione risulta ideale per il trasferimento di messaggi di dimensione contenuta e in scenari di emergenza dove i nodi partecipanti sono disposti a collaborare nella consegna del messaggio mettendo a disposizione una parte anche significativa delle loro risorse.

3.3.4 Supporto a reti opportunistiche

RAMP nella sua versione iniziale si poneva come obiettivo quello di fornire una soluzione dinamica e flessibile per le reti spontanee. Successivamente il progetto è cresciuto introducendo anche il supporto al networking di tipo opportunistico con l'aggiunta di un nuovo componente nell'architettura chiamato RAMP Opportunistic Networking (RON) [28].

Il middleware come visto in precedenza con la strategia HRMD possiede già un meccanismo per la consegna di messaggi in caso di comunicazione tollerante al ritardo. Il componente che realizza il paradigma di Store, Carry e Forward è il *Continuity Manager* (CM), il quale si occupa di rilevare le interruzioni delle connessioni e i fallimenti in fase di invio notificati dal Dispatcher e di consultare periodicamente il Resolver per verificare la presenza o meno di percorsi alternativi per ritentare la consegna dei pacchetti. Il limite di questo componente è il salvataggio temporaneo dei pacchetti in memoria centrale i quali in assenza di strade valide dopo un certo numero di tentativi saranno scartati.

L'introduzione di RON consente di dare una possibilità in più a tutti quei pacchetti che verrebbero scartati dal Continuity Manager adottando delle tecniche di networking opportunistico per massimizzare le loro probabilità di successo nella consegna. Quando il CM decide di scartare un pacchetto, questo viene preso in gestione da RON il quale controlla che sia ancora valido e che non sia già presente una copia al proprio interno. Se il controllo dà esito positivo, il pacchetto viene salvato in memoria persistente. Periodicamente ogni pacchetto gestito viene deserializzato e per ciascuno di essi viene invocato il Resolver per la scoperta di un percorso valido verso la destinazione designata, in caso affermativo si procede con l'invio. Se un percorso valido non fosse ancora disponibile, RON per aumentare le probabilità di consegna inoltra i pacchetti ai nodi vicini assicurandosi di non replicarli ai nodi che li hanno già ricevuti in interazioni precedenti.

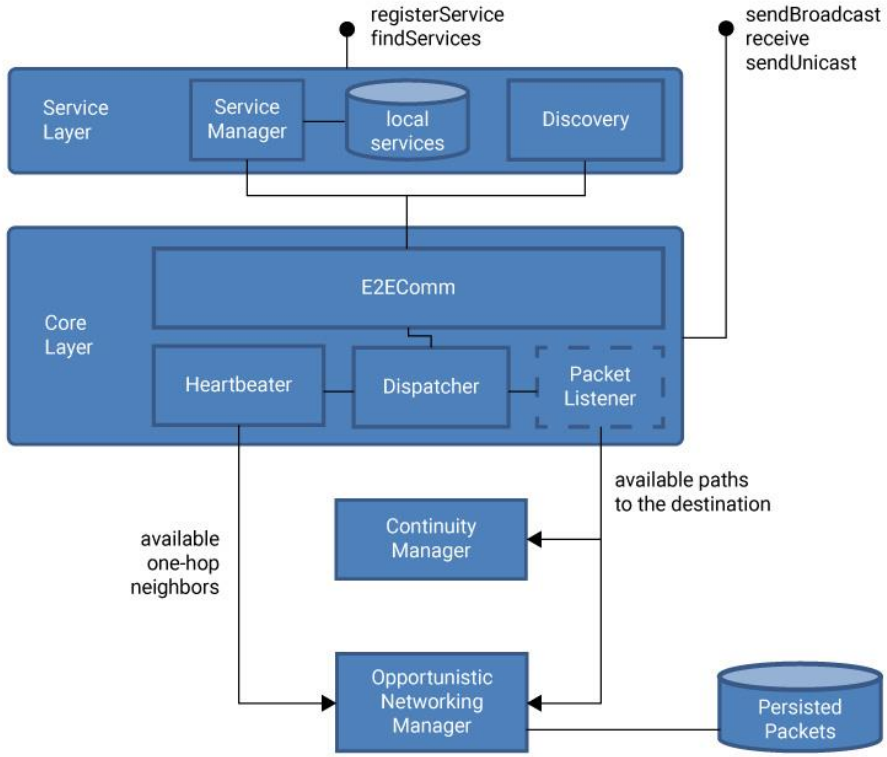

Figura 28 - Estensione RAMP per supporto alle reti opportunistiche.

3.4 Logica di controllo basata su SDN

Per quanto descritto fino ad ora, il middleware RAMP fornisce una soluzione per garantire la connettività tra nodi eterogenei e per gestire la loro mobilità all'interno della rete. Per quanto riguarda la comunicazione adotta per lo più delle strategie best-effort senza offrire un meccanismo per controllare esplicitamente il livello di Quality of Service. In reti di questo tipo può essere utile introdurre la possibilità dii gestire opportunamente i flussi di dati scambiati a seconda del tipo di traffico, dei requisiti specificati a livello applicativo e dell'attuale consumo delle risorse dei nodi partecipanti.

Per queste motivazioni come parte conclusiva del capitolo sarà descritta l'ultima estensione di RAMP attualmente disponibile che introduce la possibilità di utilizzare una logica di controllo della rete che prende ispirazione dal paradigma SDN. L'obiettivo di questa estensione è quello di rendere disponibile anche nell'ambito delle reti spontanee una modalità per consentire la regolazione del livello di qualità del servizio e differenziare così il traffico sulla base delle condizioni attuali dei nodi della rete. L'estensione che sarà brevemente illustrata di seguito rappresenta la base di partenza per il lavoro che sarà ampiamente descritto nel capitolo successivo.

3.3.1 Separazione del control plane dal data plane

Come trattato precedentemente l'adozione di una soluzione SDN comporta la presenza di due attori principali, da una parte il Controller SDN che incorpora le funzionalità del control plane e dall'altra il Control Agent che svolge le funzionalità del data plane rispettando le decisioni prese dal controller. Per riflettere questa separazione anche in RAMP sono stati introdotti due componenti denominati *ControllerService* e *ControllerClient* che svolgono rispettivamente il ruolo di Controller SDN e di Control Agent, in particolare in caso si decida di utilizzare questa funzionalità data una rete RAMP ci sarà un nodo su cui sarà in esecuzione il componente ControllerService predisposto alla messa in atto dei ruoli decisionali e i nodi restanti che eseguiranno il componente ControllerClient che si occupa di interagire con il controller per fornirgli informazioni utili alla gestione della comunicazione tra nodi e per ricevere le indicazioni relative alle azioni da compiere a seconda del tipo di traffico da inoltrare.

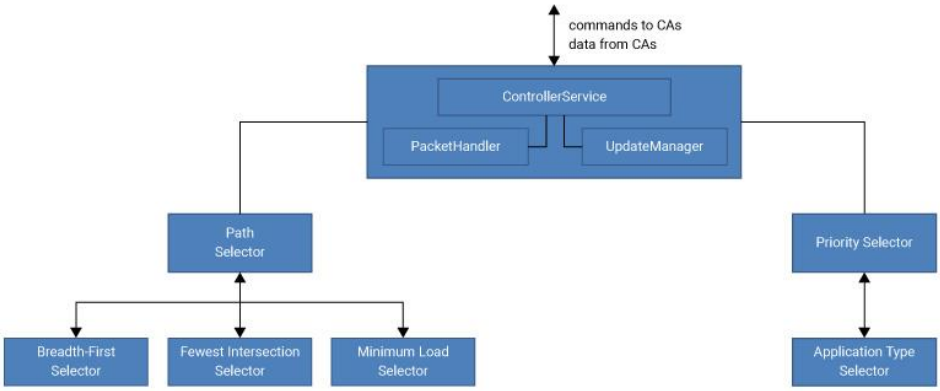

Figura 29 - Architettura Controller SDN.

Il Controller SDN, la cui architettura è mostrata in figura, per prendere delle decisioni in relazione alla miglior gestione possibile delle risorse di rete sfrutta le seguenti informazioni:

- Conoscenza completa della topologia di rete e del suo stato attuale ottenuta grazie allo scambio di messaggi di controllo con i Control Agent attualmente presenti.
- Conoscenza dei requisiti applicativi ricevuti dai Control Agent prima della generazione del traffico da utilizzare per ridurre al minimo i conflitti che possono occorrere in fase di comunicazione.

Per quanto riguarda la gestione del traffico e delle politiche di Quality of Service, in base alle conoscenze a disposizione il controller supporta:

- Politiche di routing, per modificare dinamicamente il percorso che i pacchetti attraversano da sorgente a destinazione per evitare la congestione dei link.
- Politiche di traffic engineering, per ottimizzare il quantitativo delle risorse che i flussi di traffico possono utilizzare, per esempio assicurando per la comunicazione una certa larghezza di banda per servizi ad alta priorità.

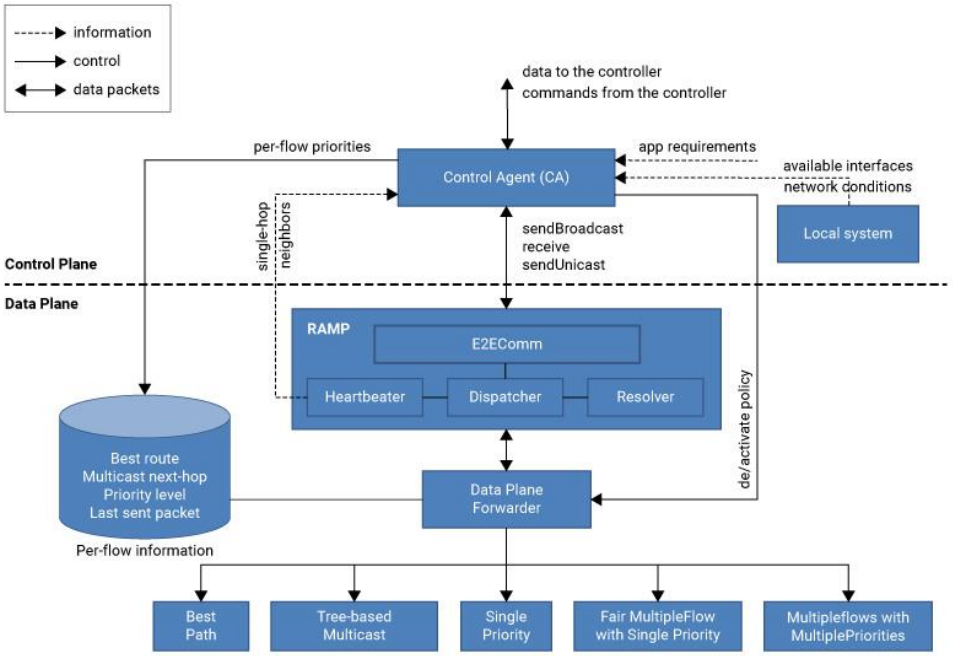

Figura 30 - Architettura Control Agent.

Il Control Agent, una volta scoperto il controller ed entrato a far parte della sua sfera di influenza, svolge le seguenti funzionalità:

- Invio periodico di informazioni al controller inerenti allo stato attuale delle proprie risorse computazionali, di memoria e di rete complessivamente disponibili e attualmente in uso.
- Invio periodico di informazioni al controller riguardanti i nodi vicini per fornirgli supporto alla costruzione della topologia di rete.
- Inoltro dei pacchetti in accordo alle decisioni di routing e di traffic engineering specificate dal controller.

Infine il Control Agent offre al livello applicativo la possibilità di ottenere un percorso dal controller fornendo oltre alla destinazione delle informazioni aggiuntive che saranno prese in considerazione per garantire la qualità del servizio richiesta. In particolare queste informazioni addizionali sono i requisiti applicativi in termini di tipo di dato da inviare, durata della

comunicazione, banda richiesta e la metrica da impiegare per la selezione del percorso. Per quanto riguarda quest'ultima il framework permette di specificare i seguenti selettori che saranno impiegati dal controller in fase di computazione di un percorso:

- *Breadth-First*, procede attuando una ricerca di tipo breadth-first nel grafo rappresentante la topologia di rete, partendo dal nodo mittente e fermandosi quando incontra il destinatario, restituendo così il primo percorso trovato.
- *Fewest Intersections*, seleziona il percorso con il minor numero di intersezioni con i percorsi già attivi all'interno della rete. Il caso di parità di nodi attraversati si passa al controllo dei link utilizzati in modo da ottenere il massimo rendimento dalla presenza di link multipli.
- *Minimum Network Load*, seleziona il percorso per cui il carico di rete complessivo dei nodi attraversati risulta minima. Per carico si intende la somma della quantità di byte inviati e ricevuti.

3.4.2 Organizzazione del traffico in flow

Dopo aver descritto sommariamente quali sono le funzionalità dei due componenti e quali sono le informazioni di controllo scambiate per garantire il funzionamento corretto della rete, per quanto riguarda la comunicazione di livello applicativo tra due nodi in questo contesto RAMP introduce il concetto di *flow*.

Per flow si intende una sequenza ordinata di pacchetti relativi ad una particolare applicazione all'interno della quale viene instaurata una comunicazione tra due nodi. Ad ogni flow viene associato un'etichetta identificativa chiamata *flowId*, la quale una volta applicata al pacchetto consente di controllare il traffico corrente, distinguendo le singole comunicazioni attive e gestendone il routing secondo quanto stabilito dal controller. Non è detto che per tutte le comunicazioni questo valore deve essere specificato, in particolare sono presenti due valori riservati -1 e 0 per indicare rispettivamente l'assenza di flowID, per i casi in cui non si voglia sfruttare la comunicazione SDN-based e l'assegnamento ad un flow di tipologia default, regolato tramite strategie differenti a quelle utilizzate per i flow con un valore definito.

3.4.3 Politiche di routing

Le politiche di routing hanno come obiettivo la selezione del percorso migliore da una sorgente a una o più destinazioni. Le politiche di routing messe a disposizione da questa estensione sono:

- *Best Path (BP)*, quando un'applicazione intende iniziare una comunicazione unicast con un determinato nodo, per mezzo del Control Agent locale, richiede al Controller SDN il percorso migliore fornendo come detto in precedenza i requisiti applicativi. Il controller ricevuta la richiesta sfruttando la sua visione globale della topologia e dello stato della rete calcola il percorso migliore da sorgente a destinazione. Il Control Agent locale riceve dal controller il percorso calcolato congiuntamente al flowId, configura il proprio data plane in modo da utilizzare il percorso associato al flowId appena ricevuto e infine restituisce quest'ultimo all'applicazione in modo che possa applicarlo ai pacchetti in fase di invio. Infine si inoltra il traffico identificato dal flowId lasciando al data plane sottostante il compito di modificare il percorso in relazione alle informazioni ricevute dal controller. Questa politica coinvolge solamente il control plane e il data plane del nodo sorgente.
- *Tree-based Multicast (TM)*, quando un'applicazione intende inoltrare del traffico verso nodi multipli, per mezzo del suo Control Agent locale, richiede al Controller il calcolo di un percorso in accordo ai requisiti applicativi specificati. Il controller alla ricezione della richiesta calcola il miglior spanning tree dalla sorgente ai nodi di destinazione e per ogni nodo dello spanning tree fornisce le informazioni relative al nodo successivo a cui i pacchetti in arrivo etichettati con il flowId corrente dovranno essere inoltrati. Infine il controller restituisce in risposta al Control Agent il flowId che il livello applicativo dovrà utilizzare per sfruttare questa politica di routing. A differenza della politica BP, TM coinvolge il control plane e il data plane di tutti i nodi parte dello spanning tree.

3.4.4 Politiche di traffic engineering

Le politiche di traffic engineering hanno l'obiettivo di gestire i pacchetti in transito nei nodi della rete al fine di raggiungere determinati obiettivi di Quality of Service. Rispetto alle politiche di routing che si occupano di individuare i percorsi da sorgente a una o più destinazioni, le politiche di traffic

engineering regolano i flussi di traffico associati ai diversi flow che competono per le stesse risorse di rete favorendone alcuni e interrompendone altri utilizzando un meccanismo basato su diversi livelli di priorità. In particolare, le applicazioni comunicano al Controller SDN i requisiti di comunicazione e come risultato ottengono un flowId da applicare al traffico da inoltrare, inoltre il controller a seconda del tipo di dato specificato nei requisiti applicativi assegna un valore di priorità e notifica il flowId corrente e la relativa priorità a tutti i Control Agent presenti. Per garantire una gestione consistente dei livelli di priorità è prevista l'attivazione di una politica di traffic engineering alla volta. Le politiche messe a disposizione dal framework sono:

- *Single Flow with Single Priority (SF-SP),* considera un solo livello di priorità pertanto un flow può essere prioritario o meno. I pacchetti appartenenti a flow prioritari sono sempre inoltrati il più velocemente possibile mentre quelli appartenenti a flow non contraddistinti da un livello di priorità sono inoltrato solo in mancata presenza di flow prioritari in competizione per lo stesso link di uscita. Nell'eventualità in cui il link di uscita sia occupato da flow prioritari, l'invio dei pacchetti ordinari viene ritardato fino al completamento dell'inoltro di quelli prioritari. In caso di presenza di flow diversi a medesima priorità i loro pacchetti sono inoltrati il più veloce possibile con conseguente competizione per le risorse di rete.
- *Multiple Flows with Single Priority (MF-SP),* come nel caso SP questa politica contempla la presenza di un solo livello di priorità. La differenza consiste nella gestione delle risorse di rete a fronte di competizione dovuta alla presenza di più di un flow ad alta priorità, in questo caso infatti la banda disponibile viene divisa equamente tra tutti i flow presenti ad alta priorità.
- *Multiple Flows with Multiple Priorities (MF-MP),* analogamente a FMFSP la seguente politica gestisce i flow con lo stesso livello di priorità in modo equo. In aggiunta, in caso di più livelli di priorità, i flow con priorità più basse possono utilizzare una quantità di risorse di rete ridotta senza dover attendere il completamento dell'invio dei pacchetti associati a priorità più elevate. In pratica, le risorse di rete sono equamente condivise tra i flow di pari livello tuttavia le risorse di rete associate sono inversamente proporzionali al livello di priorità in modo da favorire i flow più elevati.

Capitolo 4
Multi-layer routing in SDN-oriented MANET

Nei capitoli precedenti sono stati introdotti i principi e le tecnologie alla base di questo lavoro, riportando definizioni, descrizioni delle loro caratteristiche ed esempi di utilizzo. Di seguito verrà mostrata l'attività progettuale svolta con un approccio top-down, fornendo prima una panoramica generale della soluzione per poi descrivere in dettaglio l'implementazione.

Gli obiettivi principali di questo lavoro consistono nella progettazione e nella realizzazione di:

- Un meccanismo di basso livello per consentire il routing a livello di sistema operativo in un contesto di reti spontanee i cui i nodi appartengono a livello logico ad una stessa overlay network, ma a livello fisico a reti diverse e adiacenti.
- Una soluzione di alto livello per permettere al data plane dei nodi di una rete spontanea di applicare regole di elaborazione dei messaggi inoltrati in base al loro tipo e al loro contenuto informativo in caso di routing applicativo attuato in una overlay network.

L'assunzione fatta al fine di raggiungere questi obiettivi è che i nodi appartenenti alla rete spontanea siano gestiti con una logica di controllo basata sul paradigma SDN, che prevede la presenza di una intelligenza centralizzata con conoscenza completa dei partecipanti attivi nella rete.

L'implementazione di quanto detto si è basata sull'estensione del middleware RAMP, in particolare della sua variante SDN-based. Il risultato consiste nella possibilità da parte dei nodi di comunicare direttamente gli uni con gli altri attraverso l'impiego di comuni socket a livello di sistema operativo in alternativa al routing di default di RAMP, effettuato a livello applicativo. Vi è inoltre la possibilità da parte del controller SDN di comunicare ai nodi quali regole devono applicare in base al tipo di dato inoltrato e di fornire dinamicamente a tempo di esecuzione nuovi tipi di dato e nuove regole.

In questo capitolo finale si procederà dunque a mostrare le particolarità dell'attività svolta, iniziando dalle motivazioni che hanno spinto

all'introduzione delle nuove funzionalità per poi passare alla loro descrizione di alto livello e infine in termini di realizzazione software. La parte conclusiva riguarderà la fase di testing, allo scopo di illustrare i risultati ottenuti in fase sperimentale sia dal punto di vista della correttezza che delle performance.

4.1 Scenario attuale e motivazioni

Nella precedente estensione del middleware RAMP che ha introdotto la possibilità di gestire i nodi secondo il paradigma SDN, l'obiettivo è stato quello di soddisfare i requisiti legati alla qualità delle comunicazioni in quanto, in precedenza, le funzionalità presenti nel middleware si affidavano solamente a meccanismi best-effort. Pertanto è stato necessario realizzare un Controller SDN che, avendo a disposizione tutte le informazioni dei nodi attivi nella rete, è in grado di prendere decisioni di routing in accordo ai requisiti di Quality of Service specificati dalle applicazioni in esecuzione nei Control Agent. Tali decisioni di instradamento non si basano solamente sui requisiti specificati a livello applicativo ma anche sulla capacità dei nodi e sulle condizioni attuali della rete. Le comunicazioni tra i diversi Control Agent, che si affidano al Controller SDN per la scelta del percorso, sono contraddistinte da identificatori univoci chiamati *flowId*, che consentono ad un nodo intermedio di individuare il flusso di informazioni inoltrato e di rispettare a livello di data plane le decisioni prese dal Controller.

Sempre in accordo al soddisfacimento dei requisiti di qualità del servizio, questa estensione ha introdotto delle politiche di Traffic Engineering inserendo un meccanismo basato su priorità per differenziare flussi diversi a seconda del tipo di informazione trasmessa. In questo modo, nell'eventualità in cui un nodo intermedio sia attraversato da molteplici flussi che competono per le stesse risorse di rete, è in grado di favorirne alcuni e penalizzarne altri a seconda del livello di priorità di appartenenza.

Tale estensione, per quanto ricca di novità per il middleware e dalla portata innovativa nell'ambito delle reti spontanee, presenta alcune limitazioni. È innanzitutto compatibile solo con i dispositivi che eseguono RAMP, non consentendo quindi la comunicazione con altri nodi che per varie motivazioni non sono in grado utilizzarlo, come ad esempio i dispositivi legacy. In secondo luogo le politiche di Traffic Engineering presenti, basate esclusivamente sulla priorità, non sono le uniche possibili. A seconda del contesto applicativo potrebbero essere necessarie nuove regole da applicare a tipi di dato diversi da quelli che il middleware mette a disposizione di default, e inoltre

per messaggi dello stesso tipo la regola potrebbe comportarsi diversamente a seconda dell'informazione trasportata. A tale scopo sarebbe pertanto utile introdurre un grado di dinamicità più elevato e cercare di offrire un meccanismo per il Traffic Engineering più flessibile e soprattutto più personalizzabile, per soddisfare i requisiti strettamente dipendenti da tutti i possibili scenari applicativi non noti a priori.

4.2 RAMP Multi-LANE

Le considerazioni appena riportate hanno rappresentato le linee guida alla base della soluzione proposta, che ha portato ad una nuova estensione di RAMP chiamata Multi-Layer Advanced Networking Environment (Multi-LANE) operante su due piani differenti. Il primo è quello relativo ad una gestione avanzata delle tabelle di routing del sistema operativo e il secondo è quello inerente al potenziamento del data plane dei Control Agent SDN.

Per quanto tale estensione sia stata specificatamente pensata per integrarsi con il middleware di riferimento di questo lavoro, gli spunti che offre sono di valenza generale e possono essere fonte di ispirazione per lo sviluppo di nuove funzionalità riguardanti il routing e il Traffic Engineering nel campo delle reti spontanee.

4.2.1 Routing a livello di sistema operativo

Per affrontare il problema dell'instradamento, RAMP si affida ad una soluzione di livello applicativo astraendo dal sistema operativo sottostante, così da garantire interoperabilità tra sistemi e configurazioni software differenti. Per farlo crea una overlay network composta da tutti i dispositivi che eseguono il middleware, i quali vengono distinti gli uni dagli altri per mezzo di un identificativo applicativo chiamato *RampId*. Nell'accezione iniziale di RAMP la conoscenza della topologia di rete è divisa tra i vari nodi che, attraverso l'attuazione di un protocollo distribuito, sono in grado di venire a conoscenza di tutti quei dispositivi che non sono in diretta visibilità. Questo protocollo di discovery consente ad un nodo di ottenere le informazioni riguardo il suo vicinato utili a consentire successivamente la comunicazione tra i vari nodi RAMP. I messaggi scambiati contengono due informazioni principali:

- Il RampId del nodo.

- Le interfacce di rete a disposizione e l'indirizzo IP assegnato a tali interfacce.

In base a queste informazioni e al lavoro congiunto dei vari dispositivi presenti nella rete, il middleware è in grado di instradare un messaggio da sorgente a destinazione, tramite comunicazioni point-to-point.

Tale soluzione utilizza un approccio best-effort senza tenere in considerazione né tantomeno rispettare alcun requisito di Quality of Service della comunicazione. Per superare questa mancanza è stata introdotta una gestione centralizzata ispirata al paradigma SDN che prevede la presenza di un Controller. Sfruttando le informazioni ottenute dal protocollo distribuito attuato tra i vari Control Agent, il Controller è in grado di costruirsi una visione completa della topologia di rete contenente non solo i dati relativi ai collegamenti presenti tra i vari nodi, ma anche informazioni più ricche riguardanti il loro stato attuale delle risorse computazionali e di rete. In accordo con tali informazioni e con i requisiti specificati dall'applicazione in esecuzione, il Controller è in grado di effettuare il routing nel modo più appropriato in quanto unico responsabile della funzionalità del control plane.

Questo approccio al momento non prevede la possibilità per quei nodi non equipaggiati del software RAMP di interagire con la rete spontanea gestita dal middleware. Per consentire questo tipo di funzionalità è necessario abbandonare l'idea di effettuare il routing a livello applicativo, in quanto i nodi non-RAMP, non essendo a conoscenza del protocollo di discovery, non potrebbero mai scoprire i servizi offerti all'interno dell'overlay network. L'unica alternativa percorribile per rendere possibile l'interazione tra nodi RAMP e nodi non-RAMP è trovare una soluzione che permetta di interfacciarsi con le tradizionali tabelle di routing.

Tale approccio di basso livello risulta sicuramente utile in caso di comunicazione con dispositivi legacy e può rappresentare una valida alternativa al routing effettuato a livello di middleware caratterizzato da un inoltro point-to-point, consentendo di ottenere migliori prestazioni.

Essendo in un contesto di reti spontanee dove sono i protocolli distribuiti a prevalere, la presenza di un Controller SDN risulta fondamentale. Un'opzione potrebbe consistere nel cercare di effettuare il routing a livello di sistema operativo in modo distribuito, sfruttando le informazioni locali a ciascun nodo e raccoglierle per creare una visione consistente della topologia di rete. Tuttavia tale approccio porterebbe da un lato ad una complessità non trascurabile, dovuta alla natura dinamica delle reti spontanee, dall'altro ad un overhead di comunicazione elevato non sempre sostenibile. Avendo a

disposizione un Controller con conoscenza completa della topologia e dello stato attuale delle risorse dei nodi, il problema si focalizza su quale tecnica di routing tradizionale si presta alla risoluzione del problema. Tra tutte le varianti disponibili la scelta è ricaduta sull'impiego del **Policy-based Routing**, che grazie alla triade di concetti *Address*, *Route* e *Rule* consente una gestione di alto livello delle tabelle di routing del sistema operativo.

Come introdotto nella parte relativa a questa tecnica, lo scopo è quello di fornire agli amministratori di rete un pannello di controllo avanzato delle tabelle di routing. In questo caso il ruolo di amministratore è ricoperto dal Controller SDN, le cui decisioni di instradamento ricadranno sui Control Agent che si occuperanno a loro volta della modifica delle tabelle locali al sistema operativo in cui eseguono. Per rendere effettiva tale modifica, è necessario interfacciarsi con il sistema operativo con un conseguente cambio di contesto dall'ambiente in cui RAMP è in esecuzione.

Una volta decisa la tecnica di routing da impiegare per la realizzazione di questa funzionalità è stato necessario capire quale fosse la scelta migliore in termini di grado di direzionalità del percorso trovato. Il Control Agent può infatti decidere se sfruttare una connessione basata sui protocolli UDP e TCP per effettuare la comunicazione. Al di là del soddisfacimento o meno del requisito di affidabilità che contraddistingue i due protocolli, una differenza tra TCP e UDP è la natura della connessione stabilita, rispettivamente bidirezionale e unidirezionale. Questa differenza è stata rilevante per il tipo di soluzione realizzata. Dovendo modificare le tabelle IP dei singoli Control Agent, connessi tra loro attraverso reti locali adiacenti, è necessario tenere in considerazione il caso in cui questi decidano di utilizzare UDP o TCP e riflettere tale scelta in termini di percorsi configurati. In fase realizzativa si è deciso di creare percorsi bidirezionali sia in caso UDP che TCP per due principali motivi:

- Il primo è dato dalla volontà di fornire al Control Agent un percorso che possa utilizzare sia in UDP che TCP. Per garantire questo requisito bisogna sempre considerare il caso peggiore, ovvero la costruzione di un percorso bidirezionale.
- Il secondo deriva dal tipo di comunicazione che due Control Agent possono intraprendere. Nel caso UDP un nodo sorgente A invia un messaggio ad una destinazione B sfruttando un percorso unidirezionale creato dal Controller su richiesta di A. Nell'eventualità in cui B intenda rispondere ad A questo dovrebbe chiedere nuovamente al Controller la creazione di un nuovo percorso unidirezionale da B ad A, con un ulteriore overhead nella comunicazione.

Per ovviare a questo inconveniente si è deciso di creare sin da subito un percorso bidirezionale che una volta instaurato, oltre ad essere inviato al nodo che ha richiesto la sua creazione, viene anche notificato al nodo destinazione. In questo modo si dà anche la possibilità ad un nodo che non ha richiesto esplicitamente il calcolo di un percorso valido di iniziare la comunicazione per primo.

Prima di procedere alla descrizione di alto livello del protocollo che consente a due Control Agent di comunicare attraverso la modifica delle tabelle di routing, è necessario precisare quattro ulteriori aspetti:

- **Validità temporale del percorso**: non deve essere indefinita ma bensì rispecchiare le scelte di durata fornite dall'applicazione che esegue al di sopra del Control Agent. Per questo motivo è necessaria la presenza di un componente locale a tutti i CA in grado di controllare la validità del percorso e in caso non lo fosse più di eliminare la sua voce nella tabella di routing.
- **Quality of Service**: il secondo aspetto valuta la possibilità di soddisfare dei requisiti di Quality of Service nella costruzione di un percorso. Data la presenza del Controller SDN, avente a disposizione ricche informazioni sullo stato della rete, è possibile sfruttarle in modo da dare la possibilità ai CA di richiedere, ad esempio, il percorso verso una destinazione con il carico di rete minimo o il carico computazionale minimo.
- **Identificazione dei percorsi**: la terza considerazione riguarda l'identificazione dei percorsi creati all'interno di RAMP. Analogamente a quanto fatto nell'estensione SDN-based che ha introdotto il concetto di *flowId*, si introduce il concetto di *routeId* che identifica in modo univoco un percorso bidirezionale tra due Control Agent indipendentemente dal protocollo di trasporto utilizzato.
- **Cardinalità dei percorsi**: l'ultima analisi prende in esame la cardinalità dei percorsi possibili a parità di indirizzo IP sorgente e destinazione. Per il momento si è deciso di associare un solo percorso alla coppia univoca IP sorgente e IP destinazione. Di conseguenza, se due Control Agent sfruttano una sola interfaccia per comunicare, tra i due esisterà solo un percorso valido identificato dal relativo *routeId*. In caso uno dei due Control Agent richieda il calcolo di un nuovo percorso in presenza di uno già attivo, il Controller restituisce un messaggio di fallimento. Sebbene possa sembrare una limitazione, questa soluzione non preclude la possibilità che due Control Agent possano avere più percorsi attivi contemporaneamente

per comunicare. In caso ad esempio di multiple interfacce queste possono essere impiegate per identificare una nuova coppia univoca IP sorgente e IP destinazione.

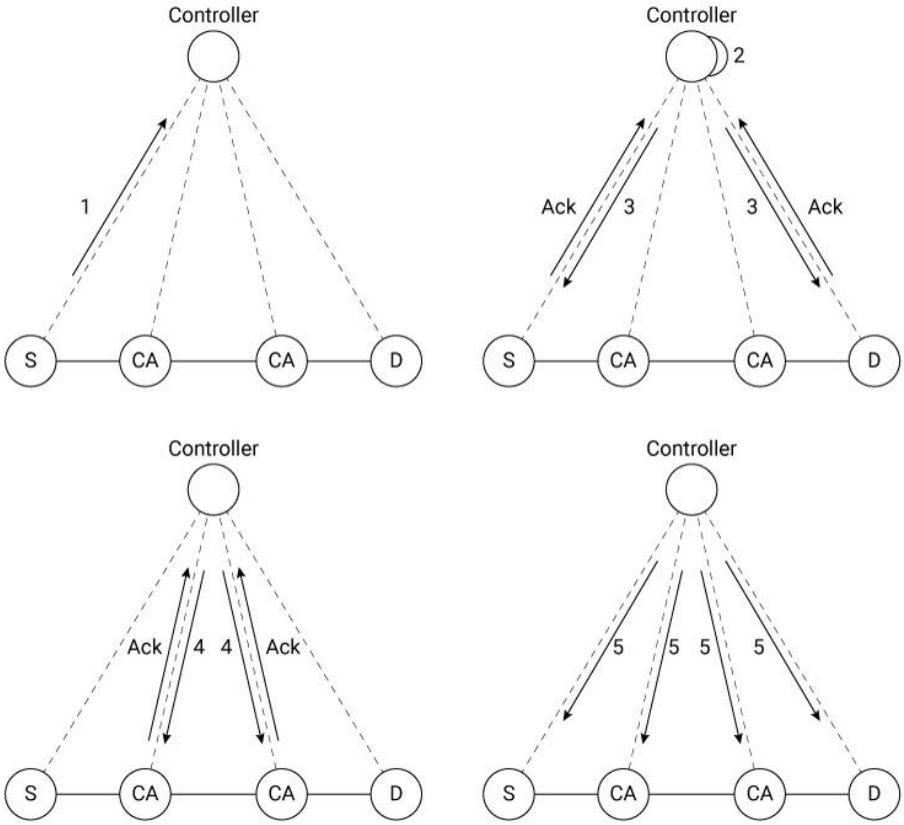

Figura 31 - Protocollo di creazione di un percorso.

Il protocollo di creazione di un nuovo percorso è sempre iniziato dal Control Agent che intende comunicare con un altro all'interno della rete spontanea gestita da RAMP e si configura nei seguenti passi:

1. Il Control Agent chiede al Controller SDN il calcolo di un percorso valido specificando: il RampID di destinazione del nodo con cui si intende comunicare, la durata del percorso e il tipo di selettore che il Controller deve utilizzare per la computazione della strada richiesta.
2. In accordo a quanto specificato dal Control Agent il Controller genera un percorso consultando il grafo della topologia di rete che ha a disposizione. Se non è possibile trovare dei percorsi validi il protocollo fallisce e viene inviata una notifica di errore al Control Agent che ha effettuato la richiesta.

3. Una volta individuati i percorsi di andata e di ritorno, il Controller inizia la procedura di scambio di messaggi di controllo contenenti le istruzioni per la modifica delle tabelle di routing con tutti i nodi dei percorsi trovati, inclusi la sorgente e la destinazione. In una prima fase il Controller invia un messaggio di controllo ai nodi sorgente e destinazione con le istruzioni da utilizzare per supportare la presenza di un nuovo percorso bidirezionale; in particolare al CA sorgente sono inviate le informazioni per la configurazione del primo segmento del percorso verso la destinazione e al CA destinatario sono inviate le informazioni per creare il primo segmento della strada per raggiungere la sorgente. In caso la modifica venga attuata con successo i due CA restituiscono un messaggio di Ack e il protocollo può continuare. In caso di messaggio di Abort da parte di uno dei due CA il protocollo fallisce e viene inviata una notifica di errore al Control Agent che ha effettuato la richiesta.

4. La seconda fase della procedura si occupa della costruzione dei successivi segmenti costituenti il percorso bidirezionale. A seconda del selettore di percorso scelto può non interessare gli stessi hop intermedi in entrambe le direzioni. Pertanto il Controller invia un messaggio di controllo a tutti i nodi intermedi del percorso di andata e di ritorno in modo da modificare opportunamente le loro tabelle di routing per supportare la presenza del nuovo percorso. Ogni nodo intermedio restituisce un messaggio di Ack o di Abort per notificare l'esito dell'operazione. In caso di ricezione anche di un solo Abort il protocollo fallisce e viene inviata una notifica di errore al Control Agent che ha effettuato la richiesta.

5. Una volta che le decisioni di routing prese dal Controller si riflettono nelle tabelle di routing di ogni singolo Control Agent, il percorso è pronto per essere utilizzato. A tal fine il Controller invia ai nodi sorgente e destinazione rispettivamente: il percorso di andata e di ritorno, ai quali è associato lo stesso *RouteId*, e la durata che questo percorso deve avere. Inoltre il Controller notifica a tutti i nodi intermedi il *RouteId* del percorso appena calcolato e la sua durata.

Al completamento del protocollo sia il Control Agent sorgente che quello di destinazione possono iniziare la comunicazione.

L'ultima peculiarità di questa soluzione riguarda il Quality of Service. Come accennato nel punto 1 del protocollo, il Control Agent richiedente la computazione di un nuovo percorso specifica il selettore che il Controller

deve utilizzare per il recupero di una strada valida. Questi selettori, già presenti nell'estensione SDN-based di RAMP e opportunamente adattati per lavorare con questa funzionalità, sono:

- **Selettore *Breadth-First***: si tratta dell'alternativa più semplice tra quelle messe a disposizione. Il selettore considera il grafo che rappresenta la topologia della rete e procede attuando una ricerca di tipo breadth-first, partendo dal nodo mittente e fermandosi quando incontra il destinatario, restituendo così il primo percorso trovato. Con questa opzione i percorsi di andata e di ritorno interessano gli stessi nodi, e in caso esista già un percorso di questa tipologia tra sorgente e destinazione il selettore restituisce, se possibile, un percorso che utilizza interfacce diverse, così da preservare l'unicità IP sorgente e IP destinazione e da sfruttare al massimo le risorse di rete a disposizione dei Control Agent.
- **Selettore *Fewest Intersection***: rappresenta un caso più evoluto rispetto al precedente. Il funzionamento di questo selettore prevede in un primo momento l'ottenimento dell'insieme di tutti i percorsi che collegano il nodo mittente al destinatario mediante l'utilizzo dell'algoritmo di Dijkstra. Successivamente, tra i percorsi restituiti viene selezionato quello che presenta il minor numero di intersezioni con i percorsi giù attivi all'interno della rete. Il confronto è realizzato sulla base dei nodi attraversati e, in caso di parità, dei link utilizzati, potendo in questo modo trarre vantaggio anche dalla presenza di eventuali collegamenti multipli tra una data coppia di nodi. Con questa opzione i percorsi di andata e di ritorno sono generalmente diversi e solo in caso in cui esista una sola strada possibile tra i due CA essi coincidono.
- **Selettore *Minimum Network Load***: similmente a quanto visto per il selettore precedente, anche in questo caso si procede applicando l'algoritmo di Dijkstra per calcolare l'insieme di percorsi tra i due end-point della comunicazione. In seguito, il componente seleziona il percorso per cui la somma del carico relativo al traffico di rete di ciascun nodo attraversato risulta minima, con riferimento alla quantità di byte ricevuti e inviati. Poiché tale selettore considera le statistiche assolute di ciascun nodo, il percorso di andata e di ritorno coincidono in termini di nodi intermedi interessati.

Nella seguente figura è illustrata l'architettura di alto livello di questa funzionalità.

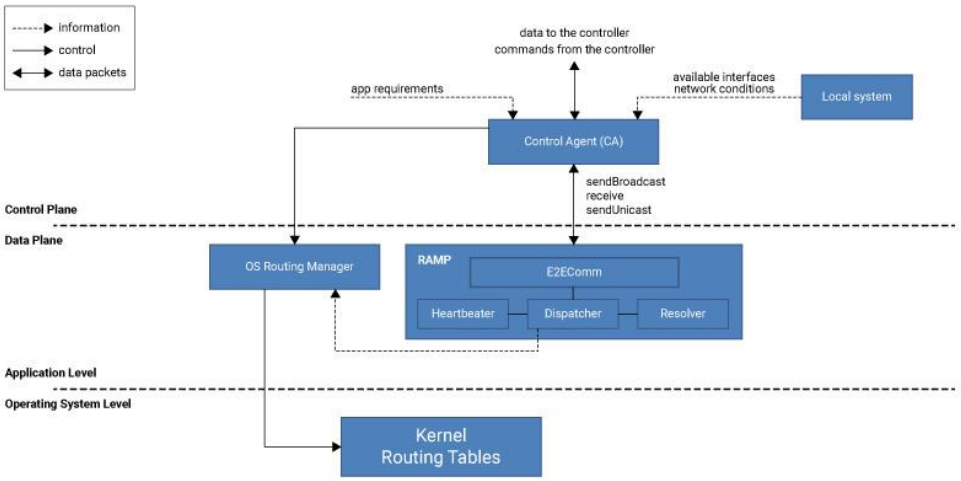

Figura 32- Architettura di alto livello del Control Agent.

4.2.2 Gestione avanzata del data plane basato su regole

In questa sezione si mostrerà la soluzione proposta relativa alla seconda parte dell'attività progettuale. Se prima si è scesi fino al livello del sistema operativo per la gestione delle tabelle di routing, qui ci si concentra sul livello applicativo, in particolare sulle funzionalità di data plane messe a disposizione dall'estensione SDN-based di RAMP. Come fatto nella sezione precedente prima di soffermarsi sulla descrizione dettagliata dell'implementazione si procederà con una panoramica di alto livello della soluzione proposta.

Nel capitolo 3 dedicato al middleware RAMP è stata introdotta l'estensione relativa all'applicazione del paradigma SDN il cui obiettivo è fornire, anche nell'ambito delle reti spontanee, un meccanismo per consentire ad un nodo di specificare i requisiti di Quality of Service da rispettare durante la comunicazione con gli altri dispositivi appartenenti alla stessa overlay network. A tal fine è stata effettuata una separazione netta del control plane dal data plane che ha portato all'introduzione dei concetti di Controller SDN e Control Agent. In seguito, come possibile politica di Traffic Engineering offerta direttamente da RAMP, è stata proposta una soluzione in grado di differenziare il traffico in diversi livelli di priorità. Per quanto utile, questa soluzione non rappresenta l'unica politica di Traffic Engineering possibile e chiaramente il middleware non può incorporare tutte quelle possibili, essendo fortemente legate allo scenario applicativo di interesse. Inoltre la differenziazione basata su diversi livelli di priorità si focalizza sul concetto di *FlowId*, che si traduce in una gestione uniforme di tutti i pacchetti appartenenti ad un determinato flow senza tenere in considerazione il contenuto informativo e il tipo di dato di ciascun pacchetto.

Prendendo come fonte di ispirazione l'approccio utilizzato nella realizzazione di tale politica di Traffic Engineering basata sul concetto di priorità, si è deciso di creare un nuovo componente in grado di applicare delle regole di elaborazione per i pacchetti inoltrati non basandosi unicamente sul flow ma anche sul tipo di pacchetto inoltrato e sulle informazioni trasportate nel payload. Inoltre, per far fronte al numero imprecisato di possibili scenari in cui è possibile utilizzare il middleware, tale funzionalità deve essere fortemente dinamica: deve essere in grado di applicare delle regole di elaborazione dei pacchetti non note a priori da RAMP, dando la possibilità di iniettarle a tempo di esecuzione e distribuirle a tutti o ad una parte dei Control Agent attivi e di applicarle quando necessario, secondo i criteri decisi dall'amministratore del Controller SDN.

Un'altra considerazione importante sull'attuale funzionamento della gestione differenziata del traffico risiede nel criterio di attribuzione del livello di livello di priorità. RAMP associa staticamente dei valori di priorità a delle categorie di dato anch'esse definite staticamente. Di conseguenza, quando il Control Agent vuole utilizzare tale politica è costretto a indicare nei requisiti applicativi una di queste categorie, che non sempre riflettono il reale contesto applicativo. Un altro limite da superare dunque è la staticità delle categorie possibili in favore di un altro componente, in grado di definire dinamicamente i tipi di dato a cui applicare le regole di data plane e di iniettarli a run-time all'interno della rete distribuendoli a tutti i Control Agent attualmente attivi. Questa funzionalità porta ad un'altra questione da risolvere, ovvero come riconoscere a tempo di comunicazione il tipo di dato trasportato dal pacchetto in modo da applicare la regola assegnatagli. Al momento l'unica politica di Traffic Engineering presente in RAMP prevede l'applicazione di una regola a livello di flow e la logica di riconoscimento è abbastanza semplice: quando un Control Agent ottiene un *flowId* dal Controller SDN e nella rete è attiva la differenziazione del traffico su diversi livelli di priorità, il Controller dissemina le informazioni sull'associazione flow-priorità a tutti i nodi della rete in modo che la politica sia applicata point-to-point. Quando il Control Agent invia dei pacchetti ad una destinazione è tenuto ad inserire il *flowId* nell'header del pacchetto, che durante l'attraversamento dei nodi intermedi viene ispezionato da ciascuno. I nodi recuperano il flowId e in base alle informazioni ottenute dal Controller sono in grado di individuare il livello di priorità e di applicare correttamente la differenziazione del traffico. Per identificare il tipo di dato si è deciso di utilizzare il suo *serialVersionUID* che consente di individuare univocamente il tipo di dato. Prendendo spunto dall'attuale soluzione, si è deciso di aggiungere un nuovo campo all'header dei pacchetti RAMP destinato a contenere tale informazione. In questo modo quando un

pacchetto contenente un determinato tipo di dato, a cui è stata associata una regola di elaborazione, attraversa un nodo questo è in grado di riconoscerlo e di applicare la regola senza difficoltà.

Una volta appurato come riconoscere un tipo di dato quando un pacchetto attraversa un Control Agent, è necessario individuare il momento in cui applicare la regola durante l'instradamento. RAMP, con la sua architettura a plug-in, consente facilmente l'inserimento di comportamento aggiuntivo nel processo di elaborazione dei pacchetti. Di conseguenza è bastato aggiungere alla catena di processamento un nuovo elemento responsabile del riconoscimento del tipo di dato e della conseguente applicazione delle regole associategli.

Ricapitolando, la funzionalità proposta si occupa di potenziare le capacità del data plane attraverso l'applicazione di regole e prevede la presenza di due nuovi componenti:

- **Data Type Manager**, che si occupa di gestire i tipi di dato che possono essere definiti a run-time all'interno della rete.
- **Data Plane Rule Manager**, responsabile di gestire l'associazione tra tipo di dato e regole, di applicare tali regole, e di fornire la possibilità di aggiungere a run-time nuove regole di data plane non note a priori.

Questi moduli, che verranno descritti accuratamente nella sezione relativa all'implementazione, sono messi a disposizione sia del Controller SDN che del Control Agent, che li impiegano in modo differente. In particolare il Controller:

- Sfrutta i due manager per fornire all'amministratore di rete le informazioni relative ai tipi di dato e alle regole di data plane attualmente disponibili e pronti per essere utilizzati.
- Si occupa di distribuire a tutti i nodi della rete la definizione di un nuovo tipo di dato fornita dall'amministratore di rete.
- Si occupa di distribuire a tutti i nodi della rete una regola fornita dall'amministratore di rete.
- Informa tutti i Control Agent, o solo un loro sottoinsieme, quali regole devono essere applicate ai pacchetti contenenti un determinato tipo di dato in accordo a quanto deciso dall'amministratore di rete. Analogamente è possibile successivamente rimuovere tali associazioni.

Il Control Agent utilizza questi due moduli per:

- Ricevere la definizione di un nuovo tipo di dato in modo da poterlo riconoscere a tempo di esecuzione e poterlo utilizzare per comunicare con altri Control Agent.
- Ricevere una nuova regola in modo che possa essere applicata quando il Controller lo riterrà opportuno.
- Applicare le regole di data plane ai pacchetti in attraversamento.

In figura si riporta un'architettura di alto livello della gestione avanzata del data plane.

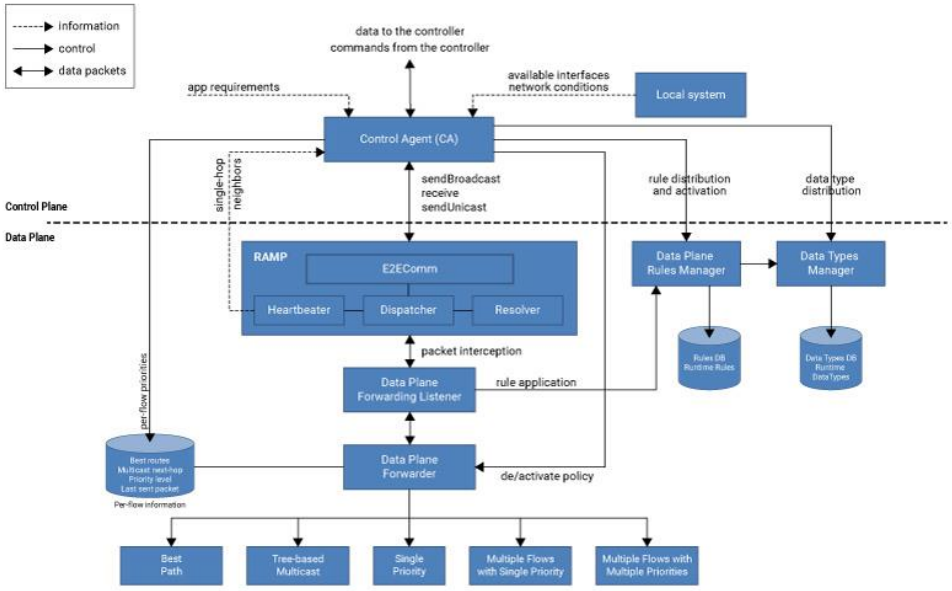

Figura 33- Architettura di alto livello della funzionalità.

4.3 Implementazione dell'estensione

Dopo avere presentato in modo generale tutti gli aspetti principali della soluzione proposta, si passerà alla descrizione della sua implementazione, scendendo nei particolari dei componenti più rilevanti.

L'attività progettuale ha previsto l'estensione del middleware RAMP per introdurre il supporto alle nuove funzionalità desiderate, che ha comportato l'introduzione di nuovi moduli software e la modifica di alcuni già esistenti così da integrare al meglio le novità al codice presente. Il codice sorgente di RAMP, contenente l'estensione frutto di questo lavoro, è consultabile al seguente indirizzo: https://github.com/DSG-UniFe/ramp.

4.3.1 Architettura generale

Come appena riportato, la realizzazione dei componenti software che soddisfano i requisiti della soluzione definiti in fase di progettazione ha assunto come punto di partenza il middleware RAMP, le cui caratteristiche principali sono già state presentate nel capitolo ad esso dedicato. L'estensione SDN-based di RAMP è stata opportunamente modificata per consentire l'introduzione delle funzionalità aggiuntive operanti su piani differenti. Da una parte la possibilità per i Control Agent di sfruttare una comunicazione basata sul routing a livello di sistema operativo, dall'altra l'introduzione di una gestione avanzata dell'elaborazione dei pacchetti a livello di data plane attraverso un sistema basato su regole.

Di seguito sono riportati i diagrammi delle architetture del Controller SDN e del Control Agent presenti in RAMP, contenenti le estensioni realizzate in questo lavoro.

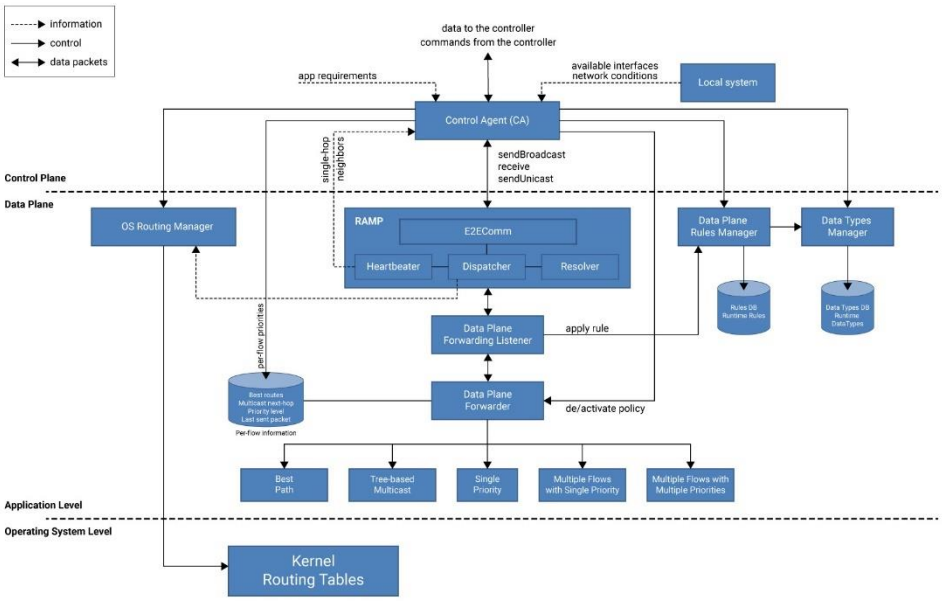

Figura 34 - Architetture ControllerClient con Multi-LANE.

Nella terminologia di RAMP il componente che si occupa di svolgere le funzionalità di Controller SDN è chiamato *ControllerService*, mentre quello preposto alle funzionalità di data plane è denominato *ControllerClient*. Da ora in avanti ci si riferirà agli attori principali del paradigma SDN con questi nomi.

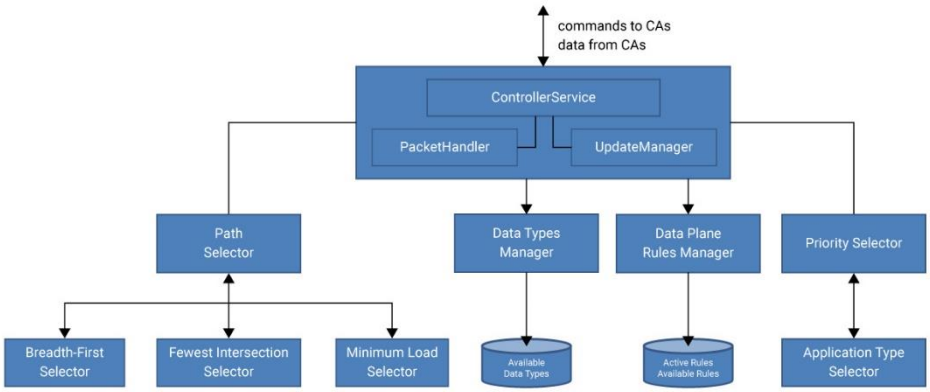

Figura 35 - Architettura ControllerService con Multi-LANE.

In seguito saranno mostrati i componenti principali che costituiscono la soluzione proposta, discutendone il funzionamento e riportandone dove presenti le problematiche affrontate a livello implementativo.

4.3.2 Operating System Routing Manager

Il primo componente che si andrà ad analizzare è il gestore delle tabelle di routing chiamato *Operating System Routing Manager* o *OSRoutingManager*. Le sue funzionalità sono di uso esclusivo del *ControllerClient* e il suo compito è quello di interfacciarsi con il sistema operativo per modificare le tabelle di routing, in accordo con le istruzioni fornite dal *ControllerService*.

Per rendere effettive tali modifiche si è deciso di utilizzare **iproute2**, una collezione di tool a riga di comando preinstallato in tutte le versioni di Linux per il controllo del networking e del traffico di rete, sia IPv4 che IPv6. Lo scopo di questa collezione di strumenti è quello di rimpiazzare il famoso set di software Unix chiamato *net-tools*, precedentemente utilizzato per la configurazione delle interfacce di rete, delle tabelle di routing e per la gestione delle tabelle ARP. Tale tool non era più aggiornato dal 2001, mentre l'ultima versione disponibile di iproute2 è la 5.1.0 rilasciata il 10 maggio 2019 [29]. Nel numeroso insieme di strumenti che iproute2 mette a disposizione sono stati impiegati quelli che consentono la gestione policy-based delle tabelle di routing: il comando *ip route*, che consente di inserire dei percorsi statici all'interno delle tabelle di routing, e il comando *ip rule*, che permette di specificare i filtri per identificare un pacchetto e quale percorso scegliere in caso quest'ultimo corrisponda ai criteri imposti dal filtro. Per meglio comprendere come iproute2 viene utilizzato dall'*OSRoutingManager* è utile riportare un semplice esempio.

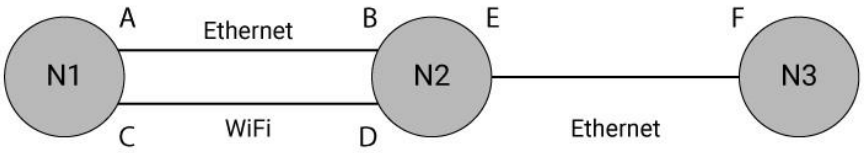

Figura 36 - Esempio d'uso della suite di strumenti iproute2.

Si immagini di avere una topologia come quella mostrata in figura: i nodi 1 e 2 possono comunicare tra loro per mezzo di due link separati (per esempio Ethernet e IEEE 802.11n) mentre il nodo 2 e il nodo 3 sono collegati tra loro attraverso una connessione Ethernet. Gli indirizzi IP del collegamento Ethernet tra 1 e 2 sono rispettivamente A e B, quelli del collegamento WiFi sono C e D. Infine gli indirizzi IP del collegamento Ethernet tra 2 e 3 sono rispettivamente E ed F. L'obiettivo di questo esempio è mostrare come è possibile far comunicare i nodi 1 e 3 che non sono in diretta visibilità e le cui reti locali di appartenenza sono adiacenti. In questa topologia l'unica strada disponibile per raggiungere il nodo 3 è passare per 2. Inoltre il nodo 1 può raggiungere il nodo 3 sfruttando una delle due interfacce di rete a disposizione. Decidendo per esempio di utilizzare l'interfaccia Ethernet, il percorso per raggiungere il nodo 3 sarà [B, F] con indirizzo mittente A. Nel caso in cui si voglia sfruttare il canale WiFi il percorso sarà di conseguenza [D, F] con indirizzo IP sorgente C. Per instaurare questi percorsi è necessario impiegare la triade del policy-based routing. Prendendo ad esempio il primo percorso dei due elencati, i passi da compiere per il nodo sorgente 1 sono i seguenti:

1. Creazione di una nuova tabella di routing personalizzata chiamata ad esempio *table1* da inserire nel file */etc/iproute2/rt_tables* con il comando *echo 1 table1 >> /etc/iproute2/rt_tables*.
2. Creazione del filtro utilizzando il meccanismo di regole messo a disposizione da iproute2 con il comando *ip rule add from A lookup table1*.
3. Inserimento di un percorso statico nella tabella di routing appena creata con il comando *ip route add F via B table table1*.

Con questa serie di comandi è stato creato il primo segmento per connettere il nodo 1 al nodo 3. Ogni volta che il nodo 1 utilizzerà l'indirizzo A come sorgente per comunicare con il nodo 3, il sistema operativo, prima di interrogare le tabelle di routing di default, consulterà la tabella *table1* associata alla regola definita al passo 2. In caso in cui il nodo 1 voglia comunicare con il nodo 2 avente indirizzo F, la voce presente all'interno della tabella indica che il prossimo hop a cui inoltrare il pacchetto è B. La serie di comandi eseguita configura solamente il segmento che va dal nodo 1 al nodo 2, in quanto una

volta che il pacchetto arriva al nodo 2 questo non saprebbe come trattarlo. È pertanto necessario eseguire i comandi riportati sopra anche per il nodo 2 in modo da configurare l'ultimo segmento del percorso. I passi sono i seguenti:

1. Creazione di una nuova tabella di routing personalizzata chiamata ad esempio table2 da inserire nel file /etc/iproute2/rt_tables con il comando echo 1 table2 >> /etc/iproute2/rt_tables.

2. Creazione di una nuova regola con il comando *ip rule from A lookup table2*.

3. Inserimento di un percorso statico nella tabella di routing appena creata con il comando *ip route add F via F table table2*.

In questo modo quando il nodo 1 invia un pacchetto con indirizzo sorgente A e indirizzo destinazione F, questo verrà inoltrato a 2 previa consultazione della tabella *table1* e una volta arrivato a 2 verrà inoltrato al nodo 3 grazie alle informazioni contenute nella tabella *table2*. Analogamente a quanto appena mostrato è possibile creare un percorso statico utilizzando il secondo canale WiFi che collega i nodi 1 e 2. I passi da compiere sono gli stessi e saranno omessi per brevità.

È il componente *OSRoutingManager* ad eseguire questo tipo di comandi. Il ruolo di amministratore di rete viene svolto dal ControllerService il quale, una volta calcolato un percorso, si occupa di inviare a tutti i nodi, esclusa la destinazione, i parametri elencati di seguito:

- Indirizzo IP sorgente.
- Indirizzo IP dell'hop successivo.
- Indirizzo IP di destinazione.

I valori di IP sorgente e destinazione sono gli stessi per tutti i nodi, mentre l'indirizzo dell'hop successivo cambia da nodo a nodo e consente di configurare ogni segmento del percorso trovato. Ricordando inoltre che si è deciso di creare un percorso bidirezionale da sorgente a destinazione, tale operazione di configurazione verrà effettuata due volte.

Prima di mostrare la logica del manager è opportuno menzionare alcuni ostacoli incontrati in fase di implementazione. iproute2 è una suite di strumenti a riga di comando che per consentire la creazione di tabelle e di regole e la modifica delle tabelle, necessita dei privilegi di amministratore. Allo stesso tempo RAMP è un middleware sviluppato in Java ed è stato quindi necessario trovare una soluzione per effettuare il cambio di contesto dall'ambiente JVM a quello del sistema operativo. Per quest'ultima criticità è stata

impiegata la libreria **Apache Commons Exec** nella sua ultima versione disponibile 1.3, la quale garantisce affidabilità nell'esecuzione di processi esterni lanciati all'interno della Java Virtual Machine [30]. Il problema dell'ottenimento delle credenziali di amministratore non è stato completamente risolto: all'avvio il ControllerClient richiede esplicitamente la password dell'account amministratore che verrà mantenuta in memoria per tutto il ciclo di vita dell'applicazione. Dal punto di vista della sicurezza questo approccio è carente. Una possibile soluzione potrebbe essere quella di aggiungere RAMP ad un gruppo autorizzato dall'account amministratore all'utilizzo di iproute2, così che in fase di startup del ControllerClient non sia più necessario il recupero delle credenziali.

Nell'Elenco 1 di seguito è riportato il codice relativo alla funzionalità di aggiunta di un segmento di un nuovo percorso secondo le istruzioni fornite dal ControllerService.

```
1.   public boolean addRoute(String sourceIP, String destinationIP, String
2.       viaIP, int routeId) {
3.     if (RampEntryPoint.os.startsWith("windows")) {
          System.out.println("OSRoutingManager: Unsupported Operating
4.           System: " + System.getProperty("os.name"));
5.        return false;
6.     } else if (RampEntryPoint.os.startsWith("linux")) {
7.        try {
8.            /*
9.             * Check if this node is the sender node.
10.            */
11.            updateLocalIpAddresses();
12.
13.            boolean isSender = this.localIpAddresses.contains(sourceIP);
14.
15.            /*
16.             * iproute2 has reserved values for these tables
17.             * 255     local
18.             * 254     main
19.             * 253     default
20.             * 0       unspec
21.             * so the routingLocalTablesIndex must start from 1 and be less
22.             * than 253.
23.             * If this range is exceeded it is not possible to add a new
24.             * local routing table.
25.             */
26.            if (this.routingLocalTablesIndex >= 253) {
27.               return false;
28.            }
29.
30.            String sudoCommandResult;
31.            String ipRouteCommand = "ip route";
32.            System.out.println("OSRoutingManager " + ipRouteCommand);
33.            sudoCommandResult = sudoCommand(ipRouteCommand);
34.            System.out.println("OSRoutingManager " + ipRouteCommand +
35.               " result: " + sudoCommandResult);
36.
37.            /*
38.             * Check if a table name for the current sourceIP exists
39.             * otherwise create it.
40.             */
41.            if (!this.routingLocalTables.containsKey(sourceIP)) {
42.               int localIpTableIndex = this.routingLocalTablesIndex;
43.               String localIpTableName = "sdnOsRouting" +
44.                  localIpTableIndex;
45.
```

```
46.                String addRoutingTableShellCommand = "echo " +
47.                   localIpTableIndex + " " + localIpTableName + " >>
48.                   /etc/iproute2/rt_tables";
49.                System.out.println("OSRoutingManager " + addRoutingTable
50.                   ShellCommand);
51.                sudoShellCommand(addRoutingTableShellCommand);
52.                this.routingLocalTables.put(sourceIP, new LocalRouting
53.                Table(localIpTableIndex, localIpTableName));
54.                /*
55.                 * For this source we have to look up the table just
56.                 * created that will contain all the routes, so we add
57.                 * a rule for all the packets coming from this sourceIP.
58.                 */
59.                String addRuleCommand = "ip rule add from " + sourceIP +
60.                   " lookup " + localIpTableName;
61.                System.out.println("OSRoutingManager " + addRuleCommand);
62.                sudoCommand(addRuleCommand);
63.
64.                /*
65.                 * Increment the index for the next local routing table to
66.                 * create.
67.                 */
68.                this.routingLocalTablesIndex++;
69.             }
70.
71.             String addRouteCommand;
72.             IpRouteRule addRouteRule;
73.             String localTableName = this.routingLocalTables.get(sourceIP)
74.                .getTableName();
75.
76.             if (isSender) {
77.                addRouteCommand = "ip route add " + destinationIP + " via "
78.                  + viaIP + " src " + sourceIP + " table " + localTableName;
79.
80.                addRouteRule = new IpRouteRule(sourceIP, viaIP,
81.                  destinationIP);
82.             } else {
83.                addRouteCommand = "ip route add " + destinationIP + " via "
84.                   + viaIP + " table " + localTableName;
85.
86.                addRouteRule = new IpRouteRule(viaIP, destinationIP);
87.             }
88.
89.             System.out.println("OSRoutingManager " + addRouteCommand);
90.             sudoCommandResult = sudoCommand(addRouteCommand);
91.
92.             /*
93.              * It is not possible to add a route for this routeId
94.              * because a route for the same destination already exists.
95.              */
96.             if (sudoCommandResult.contains("File exists")) {
97.                return false;
98.             } else {
99.                /*
100.                * Everything is fine so we can keep track of the rule for
101.                * this routeId.
102.                */
103.               if(!this.currentRules.containsKey(routeId)) {
104.                  List<String> sourceList = new ArrayList<>();
105.                  sourceList.add(sourceIP);
106.                  this.currentRules.put(routeId, sourceList);
107.               } else {
108.                  this.currentRules.get(routeId).add(sourceIP);
109.               }
110.            }
111.
112.            /*
113.             * Flush the cache
114.             */
115.            ipRouteFlushCache();
116.
117.            if (!this.currentRoutes.containsKey(routeId)) {
118.               List<IpRouteRule> rulesList = new ArrayList<>();
119.               rulesList.add(addRouteRule);
```

```
120.                this.currentRoutes.put(routeId, rulesList);
121.             } else {
122.                this.currentRoutes.get(routeId).add(addRouteRule);
123.             }
124.
125.             return true;
126.          } catch (Exception e) {
127.             return false;
128.          }
129.       } else {
130.           System.out.println("OSRoutingManager: Unsupported Operating
131.              System: " + System.getProperty("os.name"));
132.           return false;
133.       }
```

Elenco 1 - Codice della funzione addRoute dell'OSRoutingManger.

Oltre alla funzionalità di configurazione di un segmento di un percorso per mezzo della funzione *addRoute*, l'*OSRoutingManager* offre la possibilità di eliminare una voce all'interno delle tabelle di routing personalizzate con la funzione *deleteRoute*, che può essere invocata in due casi:

- Quando il ControllerService, durante configurazione di un nuovo percorso, incontra degli errori nella configurazione di un segmento e invia un messaggio di controllo a tutti quei nodi appartenenti ai segmenti precedenti correttamente configurati.
- Quando un percorso non è più valido in quanto la sua durata si è esaurita.

Non verrà riportato il codice di tale funzionalità perché poco rilevante ai fini della trattazione.

4.3.2.1 Protocollo di control plane

Dopo aver mostrato il funzionamento dell'*OSRoutingManager*, in questa sezione si presenterà il protocollo di comunicazione tra ControllerClient e ControllerService che porta al suo utilizzo. Rispetto alla descrizione qualitativa data nella presentazione generale della soluzione, si riporteranno i passi specifici che portano alla configurazione di un percorso caratterizzato da routing a livello di sistema operativo.

Il protocollo è sempre iniziato dal ControllerClient che intende comunicare con un altro nodo della rete, che tramite l'invio di un messaggio di tipo *OS_ROUTING_REQUEST* contenente:

- RampId del nodo destinazione.
- La durata desiderata del percorso.
- Selettore di percorso.

richiede al ControllerService di scoprire un nuovo percorso valido che rispetti i requisiti specificati. Il risultato finale del protocollo consiste nell'ottenimento di un *routeId* e del percorso associato per raggiungere la destinazione.

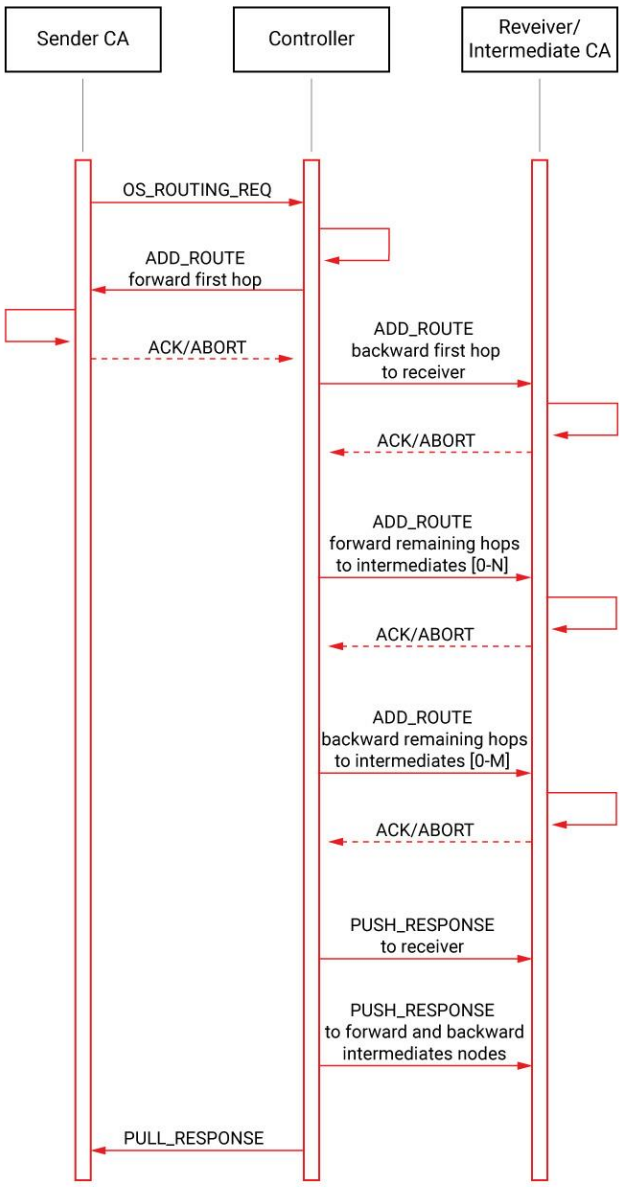

Figura 37 - Diagramma temporale del protocollo di configurazione del percorso.

Alla ricezione della richiesta il ControllerService estrae dal messaggio le informazioni sopra riportate, genera un nuovo *routeId* che identificherà il percorso bidirezionale e quindi calcola le strade di andata e di ritorno in accordo al selettore di percorso indicato. L'algoritmo relativo a tale operazione è mostrato nell'elenco seguente:

```
1.  ApplicationRequirements applicationRequirements =
2.     requestMessage.getApplicationRequirements();
3.  OsRoutingTopologyGraphSelector pathSelector = osRoutingPathSelector;
4.  PathSelectionMetric pathSelectionMetric =
```

```
5.     requestMessage.getPathSelectionMetric();
6.
7.  if (pathSelectionMetric != null) {
8.    if (pathSelectionMetric == PathSelectionMetric.BREADTH_FIRST)
9.      pathSelector = new BreadthFirstOsRoutingPathSelector(topologyGraph);
10.   else if (pathSelectionMetric == PathSelectionMetric.FEWEST_INTERSECTIONS)
11.     pathSelector =
12.       new FewestIntersectionsOsRoutingPathSelector(topologyGraph);
13.   else if (pathSelectionMetric == PathSelectionMetric.MINIMUM_NETWORK_LOAD)
14.     pathSelector =
15.       new MinimumNetworkLoadOsRoutingPathSelector(topologyGraph);
16. }
17. System.out.println("ControllerService: first OS routing path request for
18.   node ID" + clientNodeId + ", selecting a path");
19.
20. /*
21.  * Generating the routeID
22.  */
23. int routeId = ThreadLocalRandom.current().nextInt();
24. while (routeId == GenericPacket.UNUSED_FIELD ||
25.         forwardOsRoutingPaths.containsKey(routeId)) {
26.   routeId = ThreadLocalRandom.current().nextInt();
27. }
28.
29. boolean aborted = false;
30.
31. OsRoutingPathDescriptor oSRoutingForwardPath =
32.   pathSelector.selectPath(clientNodeId, destNodeId,
33.     applicationRequirements, forwardOsRoutingPaths);
34.
35. OsRoutingPathDescriptor oSRoutingBackwardPath = null;
36. if (pathSelectionMetric == PathSelectionMetric.FEWEST_INTERSECTIONS) {
37.   OsRoutingPathDescriptor reversePath =
38.     pathSelector.reversePath(oSRoutingForwardPath);
39.
40.   Map<Integer, OsRoutingPathDescriptor> candidateBackwardOsRoutingPaths =
41.     new ConcurrentHashMap<>(backwardOsRoutingPaths);
42.
43.   candidateBackwardOsRoutingPaths.put(routeId, reversePath);
44.
45.   oSRoutingBackwardPath =
46.     pathSelector.selectPath(destNodeId, clientNodeId,
47.     applicationRequirements, candidateBackwardOsRoutingPaths);
48. } else if(oSRoutingForwardPath != null) {
49.     oSRoutingBackwardPath = pathSelector.reversePath(oSRoutingForwardPath);
50. }
```

Elenco 2 - Algoritmo di calcolo del percorso effettuato dal ControllerService.

I percorsi di andata e di ritorno sono contenuti in un due oggetti distinti appartenenti alla classe *OsRoutingPathDescriptor* riportati in linea 21 e in linea 35, contenenti:

- La lista completa di rampId dei nodi che compongono il percorso, inclusa la sorgente.
- La lista completa degli indirizzi IP degli hop che serviranno per configurare i segmenti intermedi.
- L'indirizzo IP sorgente del percorso.
- L'indirizzo IP destinazione del percorso.

Ora che i percorsi sono stati correttamente individuati, il Controller Service si occupa di inviare un messaggio di tipo *OS_ROUTING_ADD_ROUTE* sia

al nodo sorgente del percorso di andata che a al nodo sorgente di quello di ritorno, finalizzati all'impostazione del primo segmento nelle due direzioni. Tale messaggio contiene:

- Indirizzo IP sorgente.
- Indirizzo IP del primo hop.
- Indirizzo IP destinazione.
- RouteId.

Si è deciso di contattare per primi i nodi mittente e ricevente per valutare sin da subito la fattibilità del percorso. Le richieste di tipo *OS_ROUTING_ADD_ROUTE* sono infatti di tipo sincrono e prevedono in risposta un messaggio di tipo *OS_ROUTING_ACK* in caso di operazione completata con successo o di tipo *OS_ROUTING_ABORT* in caso di errore. Se già in questa fase si ricevesse un messaggio di errore non avrebbe senso scomodare i nodi intermedi data l'impossibilità di creazione del percorso. Una volta ricevuta la richiesta, il Controller Client invoca la funzione *addRoute* dell'*OsRoutingManager* come mostrato nella sezione precedente e a seconda dell'esito dell'operazione invia in messaggi di risposta appena menzionati.

Assumendo che tutto sia andato a buon fine e che il primo segmento del percorso di andata e di quello di ritorno siano stati correttamente configurati, il Controller Service procede all'invio di un messaggio di tipo *OS_ROUTING_ADD_ROUTE* per tutti i nodi intermedi del percorso di andata e del percorso di ritorno. Come già detto, tutte queste richieste sono sincrone e in caso di messaggio *OS_ROUTING_ABORT* l'intero protocollo viene interrotto. Per i nodi intermedi il messaggio di richiesta contiene le seguenti informazioni:

- Indirizzo IP sorgente.
- Indirizzo IP dell'hop successivo.
- Indirizzo IP destinazione.
- RouteId.

Una volta terminata la distribuzione di tutte le istruzioni necessarie alla configurazione dell'intero percorso in ambo le direzioni, il ControllerService attua la fase conclusiva del protocollo notificando a tutti i nodi l'avvenuta creazione inviando:

- Al nodo richiedente il calcolo del percorso un messaggio di tipo *OS_ROUTING_PULL_RESPONSE* contenente l'oggetto *OsRoutingPathDescriptor* del percorso di andata e il *routeId*.
- Al nodo destinazione un messaggio di tipo *OS_ROUTING_PUSH_RESPONSE* contenente l'oggetto *OsRoutingPathDescriptor* del percorso di ritorno e il *routeId.*
- A tutti i nodi intermedi del percorso di andata e di ritorno un messaggio di tipo *OS_ROUTING_PUSH_RESPONSE* contenente la durata del percorso e il *routeId*.

Durante tutto il protocollo in caso di errore dovuto al ControllerService o alla ricezione di un messaggio *OS_ROUTING_ABORT*, il Controller invia un messaggio di controllo di tipo *OS_ROUTING_DELETE_ROUTE* a tutti quei nodi che durante il protocollo hanno correttamente configurato un segmento in modo da rimuovere tutte le informazioni relative ad esso e invia un messaggio al nodo richiedente un messaggio di controllo *OS_ROUTING_PULL_RESPONSE* con *routeId* con valore -1.

4.3.2.2 Gestione della durata di un percorso

Uno dei requisiti definiti in fase di progettazione riguarda la durata del percorso che si è detto deve essere finita. A tal fine il ControllerClient specifica la validità temporale della strada richiesta. Quando il ControllerService configura il percorso comunica a tutti i nodi con i messaggi di controllo *OS_ROUTING_PULL_RESPONSE* e *OS_ROUTING_PUSH_RESPONSE* la durata temporale dei percorsi associati ad un determinato *routeId*. Queste informazioni sono salvate localmente ai ControllerClient, che utilizzando il componente *UpdateManager* controllano periodicamente quali percorsi risultano ancora attivi e in caso si accorgano della loro scadenza invocano l'*OSRoutingManager* per rimuovere le informazioni dalle tabelle di routing. Nel seguente elenco è mostrata la routine di controllo attuata dal ControllerClient.

```
1.  private void updateOsRoutes() {
2.    for (Integer routeId : osRoutesStartTimes.keySet()) {
3.      long osRouteStartTime = osRoutesStartTimes.get(routeId);
4.        int duration = osRoutesDurations.get(routeId);
5.        long elapsed = System.currentTimeMillis() - osRouteStartTime;
6.        if (elapsed > (duration + (duration / 4)) * 1000) {
7.          osRoutingManager.deleteRoute(routeId);
8.          osRoutesStartTimes.remove(routeId);
9.          osRoutesDurations.remove(routeId);
10.         osRoutesPaths.remove(routeId);
11.       }
12.     }
13. }
```

Elenco 3 - Routine di aggiornamento dei percorsi.

In maniera del tutto analoga anche nel ControllerService è presente un componente responsabile del controllo della freschezza dei percorsi attualmente attivi.

4.3.3 Data Types Manager

Il Data Types Manager in congiunzione al Data Plane Rules Manager realizza la funzionalità di gestione avanzata del data plane basata su regole. Il suo scopo è quello di gestire tutti i tipi di dato che due ControllerClient possono scambiarsi quando intendono sfruttare il sistema di manipolazione dei pacchetti basato su regole.

Come detto in precedenza la particolarità più interessante di questo componente è la sua capacità di gestire dei tipi di dato non noti a priori e iniettati dall'amministratore di rete all'interno della MANET. Dal punto di vista implementativo si traduce nella possibilità da parte dell'amministratore di rete di iniettare il codice di una classe di interesse, la quale viene distribuita a tutti i ControllerClient. Tramite questo componente sono in grado di caricare la classe e utilizzarla a run-time sia per riconoscere eventuali pacchetti a cui applicare la regola sia in caso decidano di utilizzare un'istanza della stessa per incapsularla in un messaggio da inviare ad un ControllerClient remoto. La sfida implementativa è stata trovare un modo efficace per distribuire delle classi a tempo di esecuzione e consentirne il corretto caricamento e utilizzo. I punti salienti della soluzione derivano da due considerazioni:

- Non deve essere possibile iniettare una classe con una struttura arbitraria ma deve ereditare da una classe astratta presente all'interno di RAMP.
- La classe distribuita deve essere caricata correttamente dalla JVM in modo che possa essere incapsulata nel payload di un pacchetto durante la comunicazione tra due ControllerClient e quindi serializzata e deserializzata.

Il primo punto ha portato alla definizione di una classe astratta chiamata *AbstractDataType* che deve essere estesa da tutti i tipi di dato che si intende utilizzare. Inoltre per consentire ai clienti del middleware di poter definire dei tipi di dato personalizzati da utilizzare all'interno della rete è stato sviluppato un Software Development Kit chiamato RAMP SDK, che permette la

creazione di tipi di dato personalizzati compatibili con il middleware. Il rilascio pubblico di questo SDK è imminente.

Per quanto riguarda il secondo punto si è deciso di creare un nuovo URL Class Loader chiamato *RampClassLoader* che eredita dal Class Loader di sistema e permette di caricare dei file .class esterni all'applicazione. Per garantire la corretta deserializzazione delle istanze delle classi personalizzate in fase di ricezione, è stato modificato il metodo *deserialize* del modulo software *E2EComm* di RAMP che adesso prevede l'utilizzo del *RampClassLoader*. Se questa modifica non fosse stata apportata il metodo *deserialize* avrebbe utilizzato il Class Loader di sistema il quale, essendo agnostico sulle operazioni effettuate a run-time, avrebbe restituito un errore.

In ultima analisi fino ad ora si è fatto riferimento solo a tipi di dato definiti dinamicamente. Tuttavia si è ritenuto opportuno inserire direttamente all'interno del middleware dei tipi di dato di default pronti per essere utilizzati.

Dopo queste premesse frutto del lavoro di ricerca effettuato per realizzare la soluzione in questione, è possibile illustrare le funzionalità principali del Data Types Manager. Le sue responsabilità sono:

- Caricare a tempo di startup tutti i tipi di dato disponibili localmente.
- Caricare a tempo di esecuzione i tipi di dato personalizzati forniti come file .class dal ControllerService.
- Istanziare sotto richiesta del ControllerClient nuovi oggetti delle classi dei tipi di dato che gestisce.

Nel listato seguente è mostrata la sequenza di operazioni eseguite in fase di inizializzazione del manager.

```
1.  private void initialise() {
2.    /*
3.     * Add the dataTypesManager manged directory to classpath so that
4.     * the user defined DataTypes located in this directory can be
5.     * founded at runtime.
6.     */
7.     rampClassLoader.addPath(userDefinedDataTypesDirectoryName);
8.
9.     /*
10.     * Initialise default DataTypes, the ones currently available
11.     * at development time.
12.     */
13.    String defaultDataTypesPackage = "it.unibo.deis.lia.ramp.core.inter
14.       node.sdn.advancedDataPlane.dataTypesManager.defaultDataTypes";
15.
16.    List<Class<?>> defaultDataTypeClasses =
17.       GeneralUtils.getClassesInPackage(defaultDataTypesPackage);
18.
19.    for (Class<?> dataTypeClass : defaultDataTypeClasses) {
20.      addDataTypeToDataBase(dataTypeClass);
```

```
21.     }
22.
23.     /*
24.      * Initialise user defined DataTypes if available,
25.      * the ones currently available from previous
26.      * ControllerClient sessions.
27.      */
28.      Set<String> userDefinedDataTypes =
29.         Stream.of(new File(userDefinedDataTypesDirectoryName).listFiles())
                .filter(file -> !file.isDirectory())
                .map(File::getName)
                .collect(Collectors.toSet());
30.
31.      for (String dataTypeFileName : userDefinedDataTypes) {
32.        String dataTypeClassName =
33.           dataTypeFileName.replaceFirst("[.][^.]+$", "");
34.        try {
35.          Class cls = rampClassLoader.loadClass(dataTypeClassName);
36.          addDataTypeToDataBase(cls);
37.        } catch (ClassNotFoundException e) {
38.          e.printStackTrace();
39.        }
40.      }
41. }
```

Elenco 4 - Operazioni di inizializzazione del Data Types Manager.

Questo metodo si occupa di inizializzare il database locale salvato in memoria centrale contenente tutti i riferimenti delle classi gestite dal manager. In particolare tali riferimenti sono salvati in strutture di tipo *ConcurrentHashMap* chiamate:

- *dataTypeClassMapping* contenente coppie (chiave, valore), la chiave è una stringa indicante il nome della classe mentre il valore l'oggetto *Class* della classe stessa.
- *dataTypeMappingByName* che mantiene la corrispondenza tra nome della classe e il suo *serialVersionUID*, che come riportato è l'identificatore impiegato per individuare un tipo di dato per l'applicazione delle regole.

Un'altra funzione degna di nota del Data Types Manager è quella che si occupa di caricare dinamicamente un file .class contenente un tipo di dato personalizzato fornito dal ControllerService. Le operazioni eseguite dal metodo che la realizza sono riportate di seguito:

```
1.  public boolean addUserDefinedDataType(DataPlaneMessage dataPlaneMessage) {
2.    String dataTypeFileName = dataPlaneMessage.getFileName();
3.    String dataTypeClassName = dataPlaneMessage.getClassName();
4.
5.    if (containsDataType(dataTypeClassName)) {
6.      System.out.println("DataTypeManager: user defined DataType: " +
7.         dataTypeClassName + "already exists.");
8.      return true;
9.    }
10.
11.   File dataTypeClassFile = new File(userDefinedDataTypesDirectoryName + "/"
12.      + dataTypeFileName);
13.   byte[] bytes = dataPlaneMessage.getClassFile();
```

```
14.
15.   try {
16.     OutputStream os = new FileOutputStream(dataTypeClassFile);
17.     os.write(bytes);
18.     os.close();
19.     System.out.println("DataTypeManager: user defined DataType: " +
20.        dataTypeFileName + "successfully stored.");
21.   } catch (Exception e) {
22.     System.out.println("DataTypeManager: user defined DataType: " +
23.        dataTypeFileName + "received but not stored.");
24.     return false;
25.   }
26.
27.   try {
28.     Class dataTypeClass = rampClassLoader.loadClass(dataTypeClassName);
29.     addDataTypeToDataBase(dataTypeClass);
30.   } catch (ClassNotFoundException e) {
31.     System.out.println("DataTypeManager: user defined DataType: " +
32.        dataTypeClassName + "not loaded by RampClassLoader.");
33.     return false;
34.   }
35.
36.   return true;
37. }
```

Elenco 5 - Inserimento dinamico di un nuovo tipo di dato.

La funzione *addUserDefinedDataType* riceve in ingresso un'oggetto della classe *DataPlaneMessage* che contiene al suo interno il file .class serializzato di un tipo di dato definito dinamicamente. Una volta deserializzato, il file viene salvato in memoria persistente in modo da essere disponibile nuovamente in caso di caduta del ControllerClient. Utilizzando il *RampClassLoader* si ottiene l'oggetto *Class* del nuovo tipo di dato e lo si aggiunge al database del manager.

4.3.3.1 Protocollo di distribuzione di un nuovo tipo di dato

Essendo in un contesto ispirato ai principi SDN, è il ControllerService ad avere la responsabilità di distribuire a tutti i ControllerClient un nuovo tipo di dato fornito dall'amministratore della rete. In particolare il Controller espone un metodo che prende come parametri di ingresso il nome della classe e il relativo oggetto *File* del file .class.

Il protocollo di distribuzione è molto semplice e consiste nell'invio a tutti i ControllerClient attualmente presenti nella rete di un messaggio di controllo chiamato *DATA_PLANE_ADD_DATA_TYPE* contenente:

- Il nome del file.
- Il nome della classe.
- Il file .class serializzato.

Tale richiesta di aggiunta di un nuovo tipo di dato è sincrona e le risposte possibili sono un messaggio di tipo:

- *DATA_PLANE_DATA_TYPE_ACK*, se il nuovo tipo di dato è stato aggiunto correttamente.
- *DATA_PLANE_DATA_TYPE_ABORT* in caso contrario.

L'esito dell'operazione dipende dal risultato restituito dalla funzione *addUserDefinedDataType* del Data Types Manager locale ad ogni ControllerClient. In caso di errore il ControllerService invia a tutti i Control Agent che durante l'esecuzione del protocollo hanno correttamente caricato il nuovo tipo di dato un messaggio *DATA_PLANE_REMOVE_DATA_TYPE* il cui effetto è la rimozione del tipo di dato dal database del Data Types Manager, ma non dal file system.

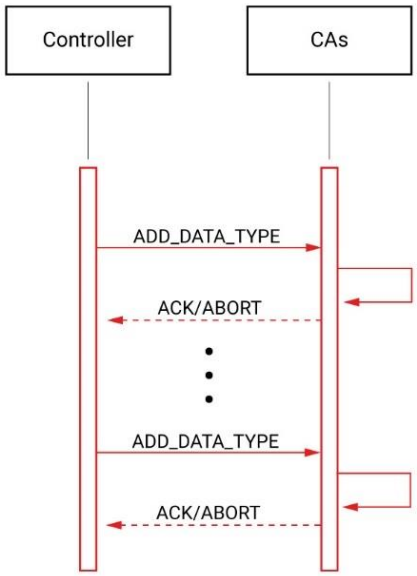

Figura 38 - Diagramma temporale del protocollo di distribuzione del tipo di dato.

4.3.4 Data Plane Rules Manager

Il componente che invece si occupa della gestione delle regole è il Data Plane Rules Manager, che rappresenta il cuore di questa funzionalità di alto livello. Grazie ad esso è possibile manipolare i pacchetti in transito a tempo di esecuzione attraverso l'applicazione di regole (che possono essere definite dinamicamente su tipi di dato eventualmente iniettati a run-time all'interno della rete) rendendo tale funzionalità non solo efficace ma soprattutto altamente flessibile in modo da far fronte a tutti i tipi di scenari applicativi.

Avendo alcune funzionalità in comune con il Data Types Manager le loro strutture sono molto simili, in particolare nella parte relativa al caricamento dinamico di nuove regole che è pressoché identica. Risolto il problema di

come aggiungere una nuova regola a run-time, le questioni da risolvere sono state due:

- Non deve essere possibile iniettare una regola definita da una classe con una struttura arbitraria ma deve ereditare da una classe astratta presente all'interno di RAMP.
- Dove inserire l'intercettore del pacchetto in modo tale da poter applicare le regole ai pacchetti in transito.

Il primo problema è stato risolto analogamente a quanto fatto con i tipi di dato, è stata definita una classe astratta chiamata *AbstractDataPlaneRule* il cui codice è riportato di seguito:

```
1.  public abstract class AbstractDataPlaneRule implements Serializable {
2.    private static final long serialVersionUID = 2551324504447890609L;
3.
4.    public AbstractDataPlaneRule() {
5.
6.    }
7.
8.    /**
9.     * This rule is used when a fragmented packet arrives, in particular
10.    * the rule is called as soon as the UnicastHeader is received.
11.    * @param uh incoming UnicastHeader
12.    */
13.   public void applyRuleToUnicastHeader(UnicastHeader uh) { }
14.
15.   /**
16.    * This rule is called when a UnicastPacket arrives.
17.    * @param up incoming UnicastPacket
18.    */
19.   public void applyRuleToUnicastPacket(UnicastPacket up) { }
20.
21.   /**
22.    * This rule is called when a BroadcastPacket arrives.
23.    * @param bp incoming BroadcastPacket
24.    */
25.   public void applyRuleToBroadcastPacket(BroadcastPacket bp) { }
26. }
```

Elenco 6 - Classe astratta AbstractDataPlaneRule.

il significato dei metodi verrà descritto a breve.

Il secondo problema è stato risolto facilmente grazie all'architettura a plug-in di RAMP la quale consente di inserire un intercettore del pacchetto registrandolo presso il Dispatcher e tale oggetto deve implementare l'interfaccia *PacketForwardingListener*. Il Data Plane Rules Manager, quando viene inizializzato, registra il suo intercettore chiamato *DataPlaneForwardingListener*, la cui implementazione è riportata di seguito.

```
1.  public class DataPlaneForwardingListener implements
2.     PacketForwardingListener {
3.
4.    private DataPlaneRulesManager dataPlaneRulesManager =
5.       DataPlaneRulesManager.getInstance();
```

```
6.
7.    @Override
8.    public void receivedUdpUnicastPacket(UnicastPacket up) {
9.      receivedTcpUnicastPacket(up);
10.   }
11.
12.   @Override
13.   public void receivedUdpBroadcastPacket(BroadcastPacket bp) {
14.     receivedTcpBroadcastPacket(bp);
15.   }
16.
17.   @Override
18.   public void receivedTcpUnicastPacket(UnicastPacket up) {
19.     long dataTypeId = up.getHeader().getDataType();
20.     if (dataTypeId != GenericPacket.UNUSED_FIELD &&
21.       dataPlaneRulesManager.containsDataPlaneRuleForDataType(dataTypeId)) {
22.       dataPlaneRulesManager.executeUnicastPacketDataPlaneRule(dataTypeId,
23.          up);
24.     }
25.   }
26.
27.   @Override
28.   public void receivedTcpUnicastHeader(UnicastHeader uh) {
29.     long dataTypeId = uh.getDataType();
30.     if (dataTypeId != GenericPacket.UNUSED_FIELD &&
31.       dataPlaneRulesManager.containsDataPlaneRuleForDataType(dataTypeId)) {
32.       dataPlaneRulesManager.executeUnicastHeaderDataPlaneRule(dataTypeId,
33.          uh);
34.     }
35.   }
36.
37.   @Override
38.   public void receivedTcpPartialPayload(UnicastHeader uh, byte[] payload,
39.       int off, int len, boolean lastChunk) {
40.   }
41.
42.   @Override
43.   public void receivedTcpBroadcastPacket(BroadcastPacket bp) {
44.     long dataTypeId = bp.getDataType();
45.     if (dataTypeId != GenericPacket.UNUSED_FIELD &&
46.       dataPlaneRulesManager.containsDataPlaneRuleForDataType(dataTypeId)) {
47.       dataPlaneRulesManager.executeBroadcastPacketDataPlaneRule(dataTypeId,
48.          bp);
49.     }
50.   }
51.
52.   @Override
53.   public void sendingTcpUnicastPacketException(UnicastPacket up,
54.      Exception e) {}
55.
56.   @Override
57.   public void sendingTcpUnicastHeaderException(UnicastHeader uh,
58.      Exception e) {}
59. }
```

Elenco 7 - Implementazione DataPlaneForwardingListener.

È proprio questo il componente responsabile del riconoscimento del tipo di dato e in caso vi fosse associata una regola di data plane alla sua applicazione. I tipi di pacchetti a cui è possibile applicare la regola sono quelli che RAMP definisce al livello applicativo:

- Pacchetti di tipo broadcast.
- Pacchetti di tipo unicast.
- Header di pacchetti di tipo unicast.

indipendentemente dal fatto che il protocollo di trasporto utilizzato sia UDP o TCP.

Da come si evince dal codice riportato le regole sono applicate solo in caso di ricezione di uno dei tipi di pacchetto appena menzionati. In una estensione futura si prevede di inserire la possibilità di applicare delle regole anche in fase di invio. Prendendo come esempio la funzione *receivedTCPUnicastPacket*, il riconoscimento di un pacchetto si basa sul recupero del suo *dataTypeId* presente nell'header, dopodiché si consulta il Data Plane Rule Manager per controllare l'esistenza di una regola e in caso affermativo si procede alla sua applicazione dando così il via alla manipolazione dei pacchetti. Dal codice appena presentato si può comprendere il significato dei metodi definiti nella classe astratta *AbstractDataPlaneRule*:

- *applyRuleToUnicastHeader* è la regola da applicare in caso di ricezione di un header di pacchetto di tipo unicast.
- *applyRuleToUnicastPacket* è la regola da applicare in caso di ricezione di un pacchetto di tipo unicast.
- *applyRuleToBroadcastPacket* è la regola da applicare in caso di ricezione di un pacchetto di tipo broadcast.

Mostrati gli elementi fondamentali atti al corretto funzionamento del Data Plane Rule Manager è arrivato il momento di presentarlo. Le sue responsabilità sono:

- Caricare a tempo di startup tutte le regole disponibili localmente.
- Caricare a tempo di esecuzione le regole personalizzate fornite come file .class dal ControllerService.
- Attivare o disattivare una regola per un determinato tipo di dato secondo quanto demandato dal ControllerService.

Il codice relativo all'aggiunta a run-time di una nuova regola fornita dal ControllerService è molto simile a quanto già visto per il Data Types Manager e pertanto non verrà mostrato.

Di seguito è invece illustrata la sequenza di operazioni inerenti all'attivazione di una regola per un determinato tipo di dato.

```
1.  public boolean addDataPlaneRule(long dataTypeId, String dataPlaneRuleName)
2.  {
3.    /*
4.     * First check if the dataType reference exists in the
5.     * DataTypesManager.
6.     */
7.     if (dataTypesManager.containsDataType(dataTypeId)) {
8.       if (!activeDataPlaneRules.containsKey(dataTypeId)) {
```

```
9.          List<String> rules = new ArrayList<>();
10.         rules.add(dataPlaneRuleName);
11.         activeDataPlaneRules.put(dataTypeId, rules);
12.       } else {
13.         if(!activeDataPlaneRules.
14.             get(dataTypeId).contains(dataPlaneRuleName)) {
15.
16.           activeDataPlaneRules.get(dataTypeId).add(dataPlaneRuleName);
17.         }
18.       }
19.       return true;
20.     }
21.     return false;
22. }
```

Elenco 8 - Attivazione di una regola di data plane.

Quando il ControllerClient riceve dal ControllerService un messaggio di controllo per l'attivazione di una nuova regola, questo invoca il metodo *addDataPlaneRule* e il risultato di tale operazione è l'inserimento in un database locale al manager, salvato in memoria centrale, della regola da applicare ad un determinato tipo di dato. Il *DataPlaneForwardingListener* consulta questo database per scoprire se al tipo di dato corrente è associata una regola e in caso affermativo la applica.

Di seguito è riportato l'algoritmo di applicazione di una regola nel caso di un pacchetto di tipo unicast.

```
1.  public void executeUnicastPacketDataPlaneRule(long dataTypeId,
2.      UnicastPacket up) {
3.
4.    String dataTypeName = dataTypesManager.getDataTypeName(dataTypeId);
5.    List<String> activeDataTypeRules = activeDataPlaneRules.get(dataTypeId);
6.    Object dataPlaneRule = null;
7.    Method method;
8.    for (String dataPlaneRuleName : activeDataTypeRules) {
9.      Class dataPlaneRuleClass =
10.         dataPlaneRulesClassMapping.get(dataPlaneRuleName);
11.     try {
12.       dataPlaneRule =
13.         dataPlaneRuleClass.getDeclaredConstructor().newInstance();
14.     } catch (InstantiationException e) {
15.       e.printStackTrace();
16.     } catch (IllegalAccessException e) {
17.       e.printStackTrace();
18.     } catch (InvocationTargetException e) {
19.       e.printStackTrace();
20.     } catch (NoSuchMethodException e) {
21.       e.printStackTrace();
22.     }
23.
24.     try {
25.       method = dataPlaneRuleClass.
26.         getDeclaredMethod("applyRuleToUnicastPacket", UnicastPacket.class);
27.       method.invoke(dataPlaneRule, up);
28.     } catch (NoSuchMethodException e) {
29.       /*
30.        * This means that the empty method of AbstractDataPlaneRule
31.        * has not been overriden.
32.        */
33.       continue;
34.     } catch (IllegalAccessException e) {
35.       e.printStackTrace();
36.     } catch (InvocationTargetException e) {
```

```
37.       e.printStackTrace();
38.     }
39.
40.     System.out.println("DataPlaneRulesManager: unicastPacketDataPlaneRule:
41.         " + dataPlaneRuleName + " for DataType: " + dataTypeName +
42.         " successfully applied.");
43.   }
44. }
```

Elenco 9 - Applicazione di una regola per un pacchetto di tipo unicast.

Il funzionamento di questo metodo è piuttosto semplice: si effettua un ciclo per individuare tutte le regole da applicare al pacchetto e ad ogni iterazione si recupera dal database delle regole attive l'oggetto *Class* associato alla regola e si invoca il metodo *applyRuleToUnicastPacket* fornendogli come parametro di ingresso il pacchetto stesso. In merito a questa funzione è utile riportare due dettagli. Il primo è che ad ogni tipo di dato è possibile associare più di una regola, il secondo invece è un aspetto implementativo dovuto alla gestione di classi iniettate a runtime: si utilizza Java Reflection per l'invocazione dei metodi che applicano le regole. Ciò non implica che esistano solamente regole definite a run-time e sono infatti state inserite all'interno del middleware due regole di default:

- *PrintDataTypeIdDataPlaneRule*, che stampa a console il *serialVersionUID* del tipo di dato correntemente inoltrato.
- *DeserializationDataPlaneRule,* che deserializza il contenuto del pacchetto in transito.

La seconda regola tra quelle fornite è sicuramente la più interessante poiché fornisce un esempio concreto di come sia possibile in base al contenuto del tipo di dato prendere delle decisioni di routing o di Traffic Engineering, che possono andare al di là di quelle decise dal ControllerService a livello di flow. Deserializzando un pacchetto, ad esempio, è possibile raccogliere statistiche precise del contenuto del tipo di informazioni scambiate oppure, in base al contenuto, decidere di inoltrare il pacchetto ad un altro nodo oltre a quello di destinazione. I possibili comportamenti realizzabili sono infiniti.

Come ultimo appunto è utile sottolineare come il RAMP SDK sia stato progettato per offrire agli sviluppatori anche la possibilità di definire una regola di data plane personalizzata e compatibile con il middleware.

4.3.4.1 Protocollo di attivazione di una regola

Prima di procedere alla descrizione del protocollo di attivazione di una regola, il Data Plane Rules Manager prevede anche un protocollo di distribuzione di una regola che non verrà descritto, in quanto è del tutto simile a

quello utilizzato dal Data Types Manager per la distribuzione di un nuovo tipo di dato.

Tornando al protocollo di attivazione, questo viene iniziato dal Controller-Service sotto comando dell'amministratore di rete. In tale protocollo vi è la possibilità di attivare una regola globalmente a tutti i nodi o solo ad un loro sottoinsieme.

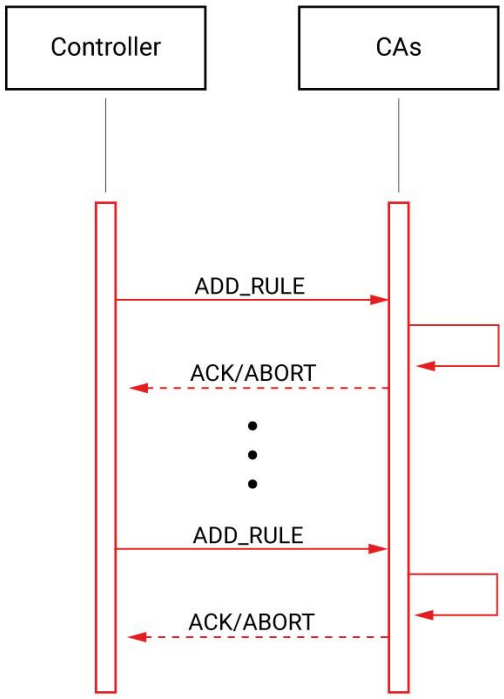

Figura 39 - Diagramma temporale del protocollo di attivazione di una regola.

Il protocollo consiste nell'invio a tutti i ControllerClient attivi nella rete, o ad una loro parte, a seconda di quanto specificato di un messaggio di controllo chiamato *DATA_PLANE_ADD_RULE* contenente:

- Il nome del tipo di dato.
- Il nome della regola.

Tale richiesta di attivazione è sincrona e le risposte possibili sono un messaggio di tipo:

- *DATA_PLANE_RULE_ACK*, se la regola è stata attivata correttamente.
- *DATA_PLANE_RULE_ABORT* in caso contrario.

L'esito dell'operazione dipende dal risultato restituito dalla funzione *addDataPlaneRule* del Data Plane Rules Manager locale ad ogni ControllerClient.

In caso di errore il ControllerService invia a tutti i Control Agent che hanno correttamente attivato la regola un messaggio *DATA_PLANE_REMOVE* il cui effetto è la disattivazione della regola.

4.4 Risultati sperimentali

Al fine di mostrare la correttezza e le prestazioni delle nuove funzionalità del middleware realizzate in fase progettuale, sono stati condotti alcuni test prendendo in considerazione diversa topologie di rete, ottenute collegando tra loro un certo numero di dispositivi eterogenei per risorse hardware e configurazioni software. Le interconnessioni tra i dispositivi sono state realizzate utilizzando protocolli di Livello 2 diversi, in particolare sono stati sfruttati Ethernet (IEEE 802.3) e WiFi (IEEE 802.11n).

Su ogni dispositivo è stato eseguito RAMP dotato delle nuove funzionalità proposte e per lo svolgimento dei test sono state realizzate ed impiegate due applicazioni ad interfaccia grafica per il ControllerService e per il ControllerClient per supportare l'esecuzione del piano di test e sono state apportate delle modifiche ad alcuni componenti del middleware per facilitare la raccolta dei dati.

L'applicazione sviluppata per fornire una vista alle funzionalità del ControllerService, denominata SDNControllerService, offre le seguenti funzionalità:

- Visualizzazione grafica in tempo reale della topologia di rete.
- Pannello di controllo per selezionare la politica di routing che la rete deve utilizzare.
- Pannello di controllo per selezionare la politica di Traffic Engineering basata su priorità che la rete deve utilizzare.
- Pannello di controllo del Data Types Manager per la distribuzione a tempo di esecuzione di un nuovo tipo di dato.
- Pannello di controllo del Data Plane Rules Manager per la distribuzione a tempo di esecuzione di una nuova regola di data plane e per l'attivazione o disattivazione di una regola per un determinato tipo di dato.

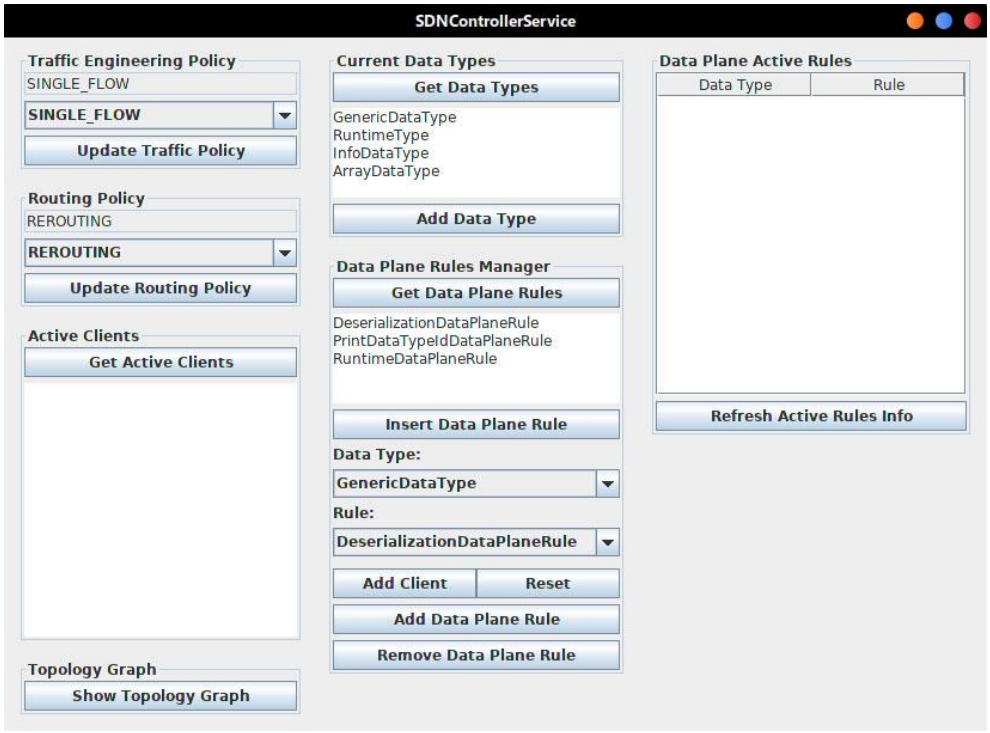

Figura 40 - Screenshot Applicazione SDNControllerService.

L'applicazione sviluppata per fornire una vista alle funzionalità del ControllerClient, denominata SDNControllerClient, offre le seguenti funzionalità:

- Monitoraggio della politica di routing e di Traffic Engineering al momento attive.
- Visualizzazione tramite il componente *ServiceDiscovery* della lista di nodi raggiungibili per comunicare.
- Pannello di controllo per effettuare la comunicazione unicast e multicast basata su *flowId*, con la possibilità di specificare i requisiti applicativi.
- Pannello di controllo per effettuare la comunicazione basata su *routeId* sfruttando il routing a livello di sistema operativo, con la possibilità di specificare i requisiti applicativi.
- Pannello di controllo per inviare messaggi e visualizzare quelli ricevuti, con possibilità di generare del traffico specificando il numero di pacchetti al secondo.

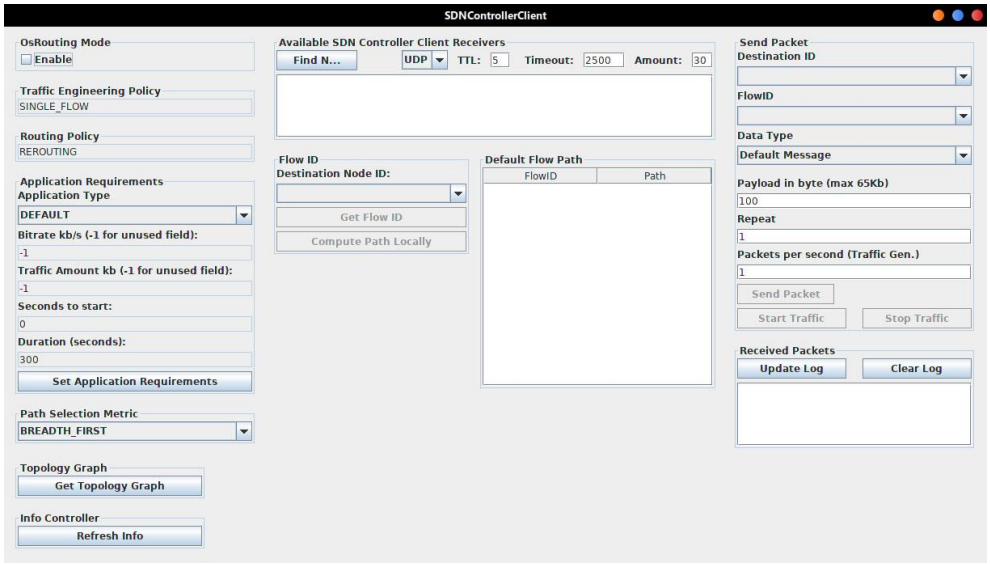

Figura 41 - Screenshot Applicazione SDNControllerClient.

L'obiettivo principale del piano di test è stato quello di analizzare il comportamento della rete in termini di quantità di informazioni trasmesse e tempo di esecuzione dei protocolli e di consumo delle risorse computazionali e di memoria dovuti all'utilizzo delle soluzioni proposte.

Per il routing a livello di sistema operativo, sono state condotte delle prove per misurare la latenza e la quantità di byte trasmessi per l'attuazione del protocollo di configurazione di un percorso.

Per la gestione avanzata del data plane basato su regole, sono stati eseguiti dei test finalizzati alla verifica della correttezza e alla misurazione dei tempi necessari all'attivazione di una regola.

Successivamente sono state effettuate delle prove per mettere a confronto i diversi meccanismi che un Control Agent ha a disposizione per comunicare con altri nodi della rete. Le alternative possibili sono:

- Il Control Agent ottiene dal Controller la topologia della rete e autonomamente si prende la responsabilità di calcolare il percorso verso la destinazione desiderata.
- Il Control Agent richiede al Controller il calcolo di un percorso finalizzato alla comunicazione basata sul routing a livello di sistema operativo.
- Il Control Agent richiede al Controller il calcolo di un percorso finalizzato alla comunicazione basata sul routing a livello di middleware.

Nella fase finale del piano di test sono stati messi a confronto i tempi e le risorse impiegate necessari alla trasmissione di una stessa informazione in caso di:

- Routing a livello di sistema operativo.
- Routing a livello applicativo nella versione RAMP di base.
- Routing a livello applicativo nella versione SDN-based di RAMP.
- Routing a livello applicativo nella versione SDN-based di RAMP con applicazione di regole di data plane a complessità crescente.

Prima di procedere alla descrizione dei risultati sperimentali di seguito sono riportate le configurazioni hardware e software dei dispositivi utilizzati nell'attuazione del piano di test:

- Raspberry Pi 3 Model B+, CPU: Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz, RAM: 1GB LPDDR2 SDRAM, Interfaccia WiFi: 2.4GHz and 5GHz IEEE 802.11b/g/n/ac wireless LAN, Interfaccia Ethernet: Gigabit Ethernet over USB 2.0 (throughput massimo 300 Mbs), Sistema Operativo: Raspbian Jessie, Versione di Java: 8.
- Raspberry Pi 3 Model B, CPU: Quad Core @ 1.2GHz Broadcom BCM2837 64-bit, RAM: 1GB, Interfaccia WiFi: BCM43438 wireless LAN, Interfaccia Ethernet: 100 Base Ethernet, Sistema Operativo: Raspbian Jessie, Versione di Java: 8.
- Acer V3-571G, CPU: Quad Core Intel Core i7-3610QM @ 2.30 GHz con Hyperthreading, RAM: 8GB, Interfaccia WiFi: Qualcomm Atheros AR5BWB222 Wireless Network Adapter, Interfaccia Ethernet: Broadcom NetLink Gigabit Ethernet, Sistema Operativo: Ubuntu 18.04.2 LTS Bionic Beaver, Versione di Java: 11.

Per consentire una raccolta accurata dei dati temporali all'inizio di ogni prova è stata effettuata una sincronizzazione degli orologi dei diversi dispositivi utilizzando il Network Time Protocol (NTP). In particolare è stato messo in esecuzione un server locale NTP nel dispositivo Acer V3-571G e tutti gli altri dispositivi sono stati configurati per sincronizzarsi con tale server.

4.4.1 Routing a livello di sistema operativo

Le prime prove effettuate sono state rivolte alla verifica della correttezza del protocollo che un ControllerClient deve intraprendere in collaborazione con il ControllerService per la creazione di un percorso al fine di comunicare con un altro ControllerClient. Come descritto in dettaglio precedentemente questa funzionalità comporta una configurazione delle tabelle di routing di ciascun nodo appartenente al percorso calcolato dal Controller. La topologia di rete presa in esame è la seguente:

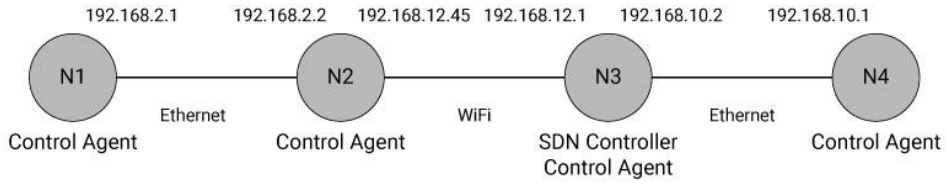

Figura 42 - Topologia di test, routing a livello di sistema operativo.

Di seguito sarà mostrato lo scambio dei pacchetti necessari all'esecuzione del protocollo per il calcolo di un percorso da N1 a N4 con selettore *Breadth-First*, in questa descrizione ci si concentrerà sui dati quantitativi e si farà riferimento a quanto descritto nella Sezione 4.3.2.1.

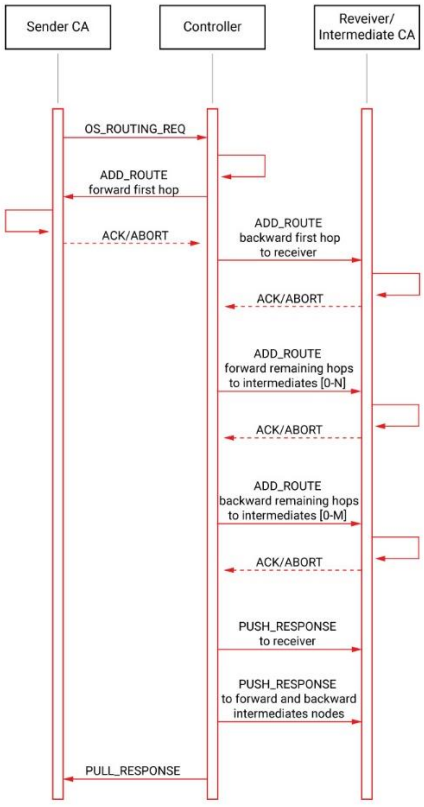

Figura 43 - Protocollo di configurazione di un nuovo percorso.

I passi per completare la procedura sono i seguenti:

1. La sorgente N1, per iniziare il protocollo, invia a N3 una richiesta di tipo *OS_ROUING_REQUEST* della dimensione di 1438 byte.
2. N3 alla ricezione della richiesta, calcola un percorso bidirezionale da N1 a N3.
3. N3 invia un messaggio di controllo di tipo *OS_ROUTING_ADD_ROUTE* a N1 della dimensione di 1448 byte per la configurazione del primo segmento del percorso di andata.
4. N1, alla ricezione del messaggio, recupera le istruzioni ottenute, configura le tabelle di routing e al completamento dell'operazione risponde a N3 con un messaggio di tipo *OS_ROUTING_ACK* della dimensione di 479 byte.
5. N3 invia un messaggio di controllo di tipo *OS_ROUTING_ADD_ROUTE* a N4 della dimensione di 1448 byte per la configurazione del primo segmento del percorso di ritorno.
6. N4, alla ricezione del messaggio, recupera le istruzioni ottenute, modifica le tabelle di routing e al completamento dell'operazione risponde a N3 con un messaggio di tipo *OS_ROUTING_ACK* della dimensione di 479 byte.
7. Per la configurazione dei restanti segmenti del percorso di andata, N3 invia un messaggio di controllo di tipo *OS_ROUTING_ADD_ROUTE* della dimensione di 1448 byte prima a N2 e successivamente al ControllerClient presente in N3.
8. N2 e N3 alla ricezione del messaggio, eseguono le istruzioni indicate dal Controller N3 e gli restituiscono un messaggio di tipo *OS_ROUTING_ACK* della dimensione di 479 byte.
9. Per il setup dei restanti segmenti del percorso di ritorno, N3 invia un messaggio di controllo di tipo *OS_ROUTING_ADD_ROUTE* della dimensione di 1448 byte prima al ControllerClient presente in N3 e successivamente a N2.
10. N3 e N2 alla ricezione del messaggio, eseguono le istruzioni indicate dal Controller N3 e gli restituiscono un messaggio di tipo *OS_ROUTING_ACK* della dimensione di 479 byte.
11. N3 invia a N4 un messaggio di tipo *OS_ROUTING_PUSH_RESPONSE* della dimensione di 1425 byte.

12. N3 invia a N2 e al ControllerClient un messaggio di tipo *OS_ROUTING_PUSH_RESPONSE* della dimensione di 800 byte. Dato che i percorsi di andata e di ritorno interessano gli stessi nodi intermedi si evita di inviare questo messaggio due volte.
13. N3 invia a N1 un messaggio di tipo *OS_ROUTING_PULL_RESPONSE* della dimensione di 1446 byte.

In Tabella 4 sono riportati i dati raccolti, i quali fanno riferimento a medie relative di 5 ripetizioni della stessa prova.

Parametri	**Media**	**Deviazione standard**
Completamento protocollo	1842 ms	525 ms
Tempo complessivo dedicato alla modifica delle tabelle di routing	1407 ms	249 ms
Tempo di modifica di una tabella di routing	235 ms	41 ms
Tempo totale scambio messaggi di controllo	409 ms	480 ms
Quantità di informazioni scambiate	16399 byte	9 byte

Tabella 4 - Risultati test del calcolo del percorso.

Dai risultati dei test si è potuto evincere che il tempo necessario al completamento del protocollo è stato molto elevato, ciò è dovuto a due ragioni fondamentali, la prima è dovuta al modo con cui vengono scambiati i messaggi di controllo: le richieste di tipo *OS_ROUTING_ADD_ROUTE* sono sincrone e non si sovrappongono mai nel tempo, questo porta ad un allungamento considerevole dei tempi di completamento del protocollo; la seconda riguarda i tempi di modifica delle tabelle di routing il cui tempo di completamento è fortemente dipendente dalle risorse computazionali del nodo: il ControllerClient di N1 è stato eseguito sul dispositivo Acer V3-571G con un tempo medio di modifica di 39 millisecondi con deviazione standard di 9 millisecondi mentre il ControllerClient di N4 è stato eseguito sul dispositivo Raspberry Pi 3 Model B con un tempo medio di modifica di 265 millisecondi con deviazione standard di 83 millisecondi. Da ciò consegue che il cambio di contesto dalla JVM al sistema operativo incide sulla latenza ma non quanto la natura sincrona del protocollo. Pertanto, esaminando i dati raccolti, è stato possibile stimare il tempo necessario all'esecuzione del protocollo in versione asincrona:

- il tempo medio necessario al trasferimento del messaggio di *OS_ROUING_REQUEST* è di 23 millisecondi con deviazione standard di 12 millisecondi.

- Il tempo medio che il Controller impiega per calcolare i percorsi di andata e di ritorno è di 20 millisecondi con deviazione standard di 17 millisecondi.
- Il tempo medio del trasferimento del messaggio *OS_ROUTING_ADD_ROUTE* è di 20 millisecondi con deviazione standard di 17 millisecondi.
- il tempo medio impiegato dai nodi per modificare le tabelle di routing è di 235 millisecondi con deviazione standard di 41 millisecondi.
- si stima di 100 millisecondi il tempo necessario alla raccolta asincrona degli Ack da parte del Controller.
- si stima di 50 millisecondi il tempo per l'invio dei messaggi di *OS_ROUTING_PUSH_RESPONSE* e di OS_ROUTING_PULL_RESPONSE.
- infine si aggiunge una componente variabile addizionale di 40 millisecondi per il completamento di tutte le operazioni.

Questa stima porta ad una durata del protocollo in versione asincrona di 488 millisecondi.

Per quanto invece riguarda la correttezza delle configurazioni delle tabelle di routing è stato possibile verificarla utilizzando il tool a riga di comando *iPerf* [31].

È stato poi effettuato un secondo test prendendo in considerazione una topologia più complessa mostrata in figura:

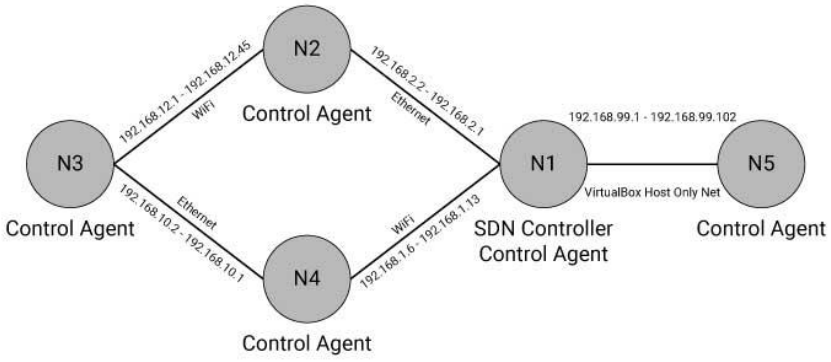

Figura 44 - Topologia ad "aquilone" per selettore Fewest Intersection.

L'obiettivo di questo test è dimostrare la flessibilità della soluzione proposta nel calcolare un percorso bidirezionale, che in contrapposizione a quanto mostrato nella prova precedente, potrebbe non interessare lo stesso insieme di

nodi intermedi per la strada di andata e quella di ritorno. Tale carattere di elasticità risulta utile in contesti applicativi in cui si vogliono sfruttare al massimo i canali di interconnessione tra nodi.

Nel caso proposto, per raggiungere N3, la sorgente N5 ha a disposizione due possibili percorsi: il primo passante per N2 e il secondo passante per N4. Le prove del tutto qualitative sono state condotte utilizzando le due applicazioni ad interfaccia grafica descritte nella sezione precedente. Specificando la strategia *Fewest Intersection* si è potuto verificare che il percorso per raggiungere N3 è quello passante per N2 mentre quello per andare N3 a N5 è quello passante per N4. In fase di calcolo della strada di ritorno, il Controller si assicura di non selezionare lo stesso percorso scelto per l'andata. L'algoritmo ha restituito per il percorso di andata la seguente lista di hop [192.168.99.1, 192.168.2.2, 192.168.12.1] mentre per quello di ritorno la lista [192.168.10.1, 192.168.1.13, 192.168.99.102].

4.4.2 Protocolli di control plane per la gestione delle regole

In questa fase del piano di test, si verificherà la correttezza dei protocolli di control plane introdotti per quanto riguarda:

- La distribuzione di un nuovo tipo di dato e il suo corretto utilizzo a tempo di esecuzione.
- La distribuzione di una nuova regola e la sua successiva applicazione a tempo di esecuzione per un determinato tipo di dato.

Per tutte le prove riportate è stata considerata la topologia mostrata in figura:

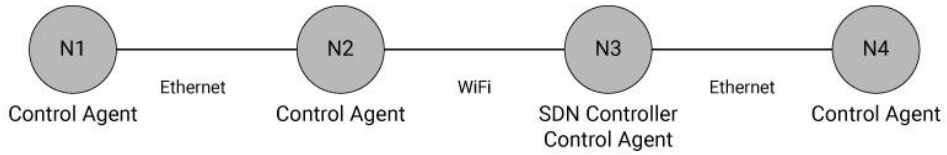

Figura 45 - Topologia di test dei nuovi protocolli di control plane.

La prima parte di questo test consiste nel verificare che un determinato tipo di dato, definito per mezzo del RAMP SDK, sia distribuito correttamente dal Controller N3 a tutti i nodi della rete e nel misurare la latenza e la quantità di byte trasmessi necessari al completamento del protocollo. Per condurre questa prova è stato utilizzato il pannello di controllo del Data Types Manager presente nell'applicazione SDNControllerService.

In questa prova, il tipo di dato utilizzato è quello definito dalla seguente classe Java:

```
1.  public class RuntimeType extends AbstractDataType implements Serializable {
2.    private static final long serialVersionUID = 1611977844899648823L;
3.
4.    public RuntimeType() {
5.      super();
6.    }

7.    public RuntimeType(int seqNumber, int payloadSize) {
8.      this.seqNumber = seqNumber;
9.      this.payload = new byte[payloadSize];
10.   }
11. }
```

Elenco 10 - Classe RuntimeType definita dinamicamente.

La dimensione del relativo file .class è di 479 byte. Di seguito si riporta la descrizione quantitativa del protocollo presentato in dettaglio nella Sezione 4.3.3.1:

1. Il Controller N3, una volta ricevuto il file RuntimeType.class, invia a tutti i nodi un messaggio di controllo di tipo *DATA_PLANE_ADD_DATA_TYPE* della dimensione di 1986 byte.
2. Alla ricezione del messaggio i nodi recuperano le informazioni contenute al suo interno, caricano la classe ricevuta e restituiscono a N3 un messaggio di controllo di tipo *DATA_PLANE_DATA_TYPE_ACK* della dimensione di 657 byte.

In Tabella 5 si mostrano i risultati delle prove effettuate.

Parametri	Media	Deviazione standard
Completamento protocollo	242 ms	24 ms
Tempo di caricamento della classe	28 ms	6 ms
Quantità di informazioni scambiate	10570 byte	13 byte

Tabella 5- Risultati prove di inserimento dinamico di un nuovo tipo di dato.

Il tempo impiegato dal protocollo ha una latenza accettabile. Esiste tuttavia un margine di miglioramento in caso di implementazione asincrona della procedura di invio dei messaggi di controllo. Infine, utilizzando il pannello di controllo dell'applicazione SDNControllerClient per l'invio e la ricezione di messaggi, è stato possibile verificare il corretto utilizzo a tempo di esecuzione del nuovo tipo di dato sia in fase di inoltro che di ricezione.

Utilizzando la stessa topologia, si è passati alla verifica della corretta distribuzione, attivazione e applicazione di una regola di data plane iniettata nella rete a run-time.

Sfruttando il pannello di controllo del Data Plane Rules Manager dell'applicazione SDNControllerService è stato possibile effettuare il deployment di una nuova regola definita per mezzo del RAMP SDK. La classe della regola oggetto di questo test è riportata nel seguente elenco:

```
1.  public class RuntimeDataPlaneRule extends AbstractDataPlaneRule {
2.
3.    public RuntimeDataPlaneRule() {}
4.
5.    @Override
6.    public void applyRuleToUnicastPacket(UnicastPacket up) {
7.      UnicastHeader uh = up.getHeader();
8.      System.out.println("RuntimeDataPlaneRule: DataTypeId: " +
9.         uh.getDataType());
10.   }
11. }
```

Elenco 11 - Classe RuntimeDataPlaneRule definita dinamicamente.

Lo scopo della regola RuntimeDataPlaneRule è quello di stampare a console il *serialVersionUID* del pacchetto di tipo unicast a cui viene applicata.

Il protocollo del tutto simile a quello del test precedente si sviluppa nei seguenti passi:

1. Il Controller N3, una volta ricevuto il file RuntimeDataPlaneRule.class, invia a tutti i nodi un messaggio di controllo di tipo *DATA_PLANE_ADD_RULE_FILE* della dimensione di 2528 byte.
2. Alla ricezione del messaggio i nodi recuperano le informazioni contenute al suo interno, caricano la classe ricevuta e restituiscono a N3 un messaggio di controllo di tipo *DATA_PLANE_RULE_ACK* della dimensione di 653 byte.

Di seguito i dati sperimentali raccolti.

Parametri	Media	Deviazione standard
Completamento protocollo	317 ms	178 ms
Tempo di caricamento della classe	26 ms	5 ms
Quantità di informazioni scambiate	12721 byte	9 byte

Tabella 6 - Risultati test di distribuzione di una nuova regola.

I dati riportati fanno riferimento a 5 ripetizioni della stessa prova. Dalle prove sperimentali si osserva che la latenza dipende linearmente dalla dimensione del file inviato.

La prova successiva si è focalizzata sul protocollo di attivazione di una regola per un determinato tipo di dato. Anche per questa prova è stato impiegato il pannello di controllo del Data Plane Rules Manager dell'applicazione SDNControllerService e si è deciso di mostrare l'attivazione della regola

RuntimeDataPlaneRule per il tipo di dato RuntimeType per tutti i nodi della rete. Di seguito si riporta la descrizione quantitativa del protocollo presentato dettagliatamente nella Sezione 4.3.4.1:

1. N4 invia a tutti i nodi della rete un pacchetto di tipo *DATA_PLANE_ADD_RULE* della dimensione di 1191 byte.

2. Alla ricezione del messaggio di controllo i nodi recuperano le informazioni contenute al suo interno, attivano la regola per il tipo di dato indicato e restituiscono a N3 un messaggio di controllo di tipo *DATA_PLANE_RULE_ACK* della dimensione di 653 byte.

I dati riportati in tabella fanno riferimento a 5 ripetizioni della stessa prova.

Parametri	Media	Deviazione standard
Completamento protocollo	173 ms	98 ms
Tempo di attivazione della regola	8 ms	5 ms
Quantità di informazioni scambiate	7373 byte	6 byte

Tabella 7- Risultati test di attivazione di una regola di data plane.

In conclusione, per constatare l'effettiva applicazione della regola da parte di tutti i nodi, è stato inviato un pacchetto di tipo RuntimeDataType dal N1 a N4 e si è verificato che ogni nodo ha stampato a console il seguente messaggio: "RuntimeDataPlaneRule: DataTypeId 16119778448996488 23" indicante il *serialVersionUID* della classe RuntimeDataType.

4.4.3 Confronto tra i meccanismi di calcolo di un percorso

In questa sezione si inizia la presentazione dei risultati sperimentali dei test più significativi di questo lavoro di tesi. Se fino ad ora sono state condotte delle prove incentrate principalmente sulla correttezza della soluzione proposta, in questa fase si analizzeranno le performance delle alternative che un Control Agent ha a disposizione per ottenere un percorso verso un altro Control Agent. Come anticipato in precedenza le opzioni possibili sono:

A. Il Control Agent autonomamente calcola il percorso verso la destinazione dopo aver ottenuto la topologia dal Controller al fine di sfruttare il routing a livello di middleware.
B. Il Control Agent delega al Controller il calcolo del percorso al fine di sfruttare il routing a livello di sistema operativo.
C. Il Control Agent delega al Controller il calcolo del percorso al fine di sfruttare il routing a livello di overlay network.

Il meccanismo associato all'opzione A non è una feature al momento disponibile in RAMP pertanto è stato aggiunto un metodo ad-hoc nel codice del ControllerClient per simulare tale funzionalità, di seguito l'implementazione della funzione.

```
1.  /**
2.  * This functionality is not currently working, it has been added
3.  * only for testing purposes to simulate
4.  * the scenario of a fat ControllerClient able to compute a new path
5.  * by itself without the ControllerService
6.  * intervention. The only interaction that the ControllerClient
7.  * has with the ControllerService is about
8.  * the retrieval of the most updated topology graph.
9.  */
10. public PathDescriptor computeUnicastPathLocally(int clientNodeId, int des
11.     tinationNodeId, PathSelectionMetric pathSelectionMetric) {
12.
13.   getTopologyGraph();
14.   if (this.topologyGraph == null) {
15.     return null;
16.   }
17.
18.   PathDescriptor path = null;
19.   TopologyGraphSelector pathSelector = null;
20.   if (pathSelectionMetric != null) {
21.     if (pathSelectionMetric == PathSelectionMetric.BREADTH_FIRST)
22.      pathSelector = new BreadthFirstFlowPathSelector(topologyGraph);
23.     else if (pathSelectionMetric ==
24.         PathSelectionMetric.FEWEST_INTERSECTIONS)
25.      pathSelector = new FewestIntersectionsFlowPathSelector(topologyGraph);
26.     else if (pathSelectionMetric ==
27.         PathSelectionMetric.MINIMUM_NETWORK_LOAD)
28.      pathSelector = new MinimumNetworkLoadFlowPathSelector(topologyGraph);
29.
30.      path = pathSelector.selectPath(clientNodeId, destinationNodeId, null,
31.          flowPaths);
32.   }
33.
34.   if (path == null) {
35.     return null;
36.   }
37.
38.   return path;
39. }
```

Elenco 12 - Funzione di calcolo del percorso locale al Control Agent.

Il test si concentrerà sul confronto in termini di latenza e di quantità di informazioni scambiate per l'ottenimento del percorso per ciascuno dei meccanismi menzionati. La topologia che si prenderà in considerazione è la stessa utilizzata per le prove condotte per le funzionalità avanzate di data plane.

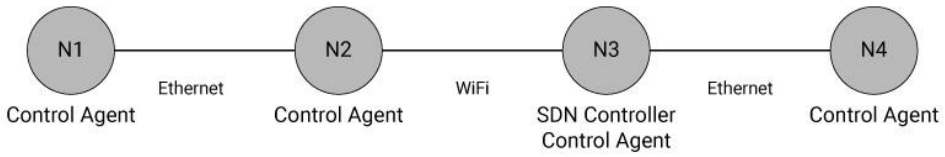

Figura 46 - Topologia di test per i meccanismi di calcolo di un percorso.

Il percorso che si andrà a calcolare è quello che va da N1 a N4, il ruolo di Controller è ricoperto dal nodo N3.

Per tutte le prove si è considerato il ritrovamento del percorso per andare da N1 a N4 specificando come strategia di calcolo il selettore *BreadthFirst*. I valori medi riportati di seguito fanno riferimento a 5 ripetizioni del test.

Per il caso A i dati sperimentali raccolti sono stati:

- Tempo medio di 106 millisecondi con deviazione standard di 22 millisecondi.
- Quantità totale media di dati trasmessi di 3516 byte con deviazione standard di 3 byte.

Per il caso B i dati sperimentali raccolti sono stati:

- Tempo medio di 1842 millisecondi con deviazione standard di 525 millisecondi.
- Quantità totale media di dati trasmessi di 16399 byte con deviazione standard di 9 byte.

Per il caso C i dati sperimentali raccolti sono stati:

- Tempo medio di 48 millisecondi con deviazione standard di 19 millisecondi.
- Quantità totale media di dati trasmessi di 2659 byte con deviazione standard di 5 byte.

È utile evidenziare che il numero ridotto di byte trasmessi per l'opzione A dipende strettamente dal numero di nodi presenti nella topologia, pertanto in caso di un numero maggiore di nodi attivi tale dato è destinato a salire. Stesso dicasi per il caso B, dove la quantità totale di byte trasmessi aumenta drasticamente in caso di percorsi lunghi in quanto bisogna pagare il prezzo per la configurazione dei percorsi di andata e di ritorno. Per quanto invece riguarda la latenza, il risultato ottenuto dal caso B è pessimo se confrontato con gli altri due. C'è da però da dire che a differenza delle altre due alternative i cui protocolli interessano solo il nodo sorgente e il Controller, il caso B comporta la partecipazione di tutti i nodi del percorso da calcolare. In accordo all'analisi effettuata nella sezione precedente, si stima che il tempo di completamento di una versione asincrona del protocollo di routing di basso livello, a parità di informazioni scambiate, sia di 488 millisecondi.

Di seguito si riportano i grafici riassuntivi dei dati sperimentali ottenuti:

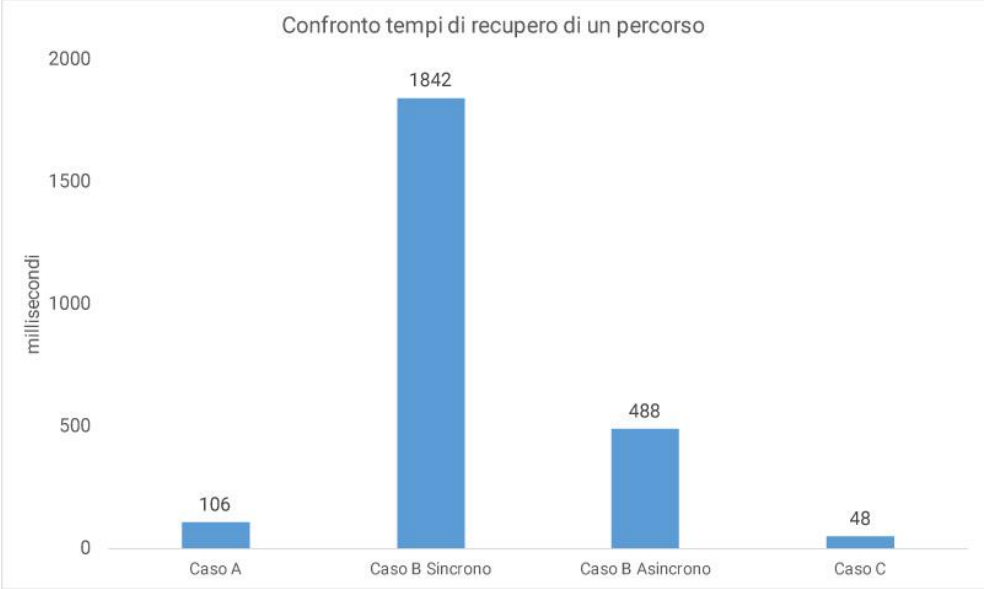

Figura 47 - Diagramma di confronto dei tempi.

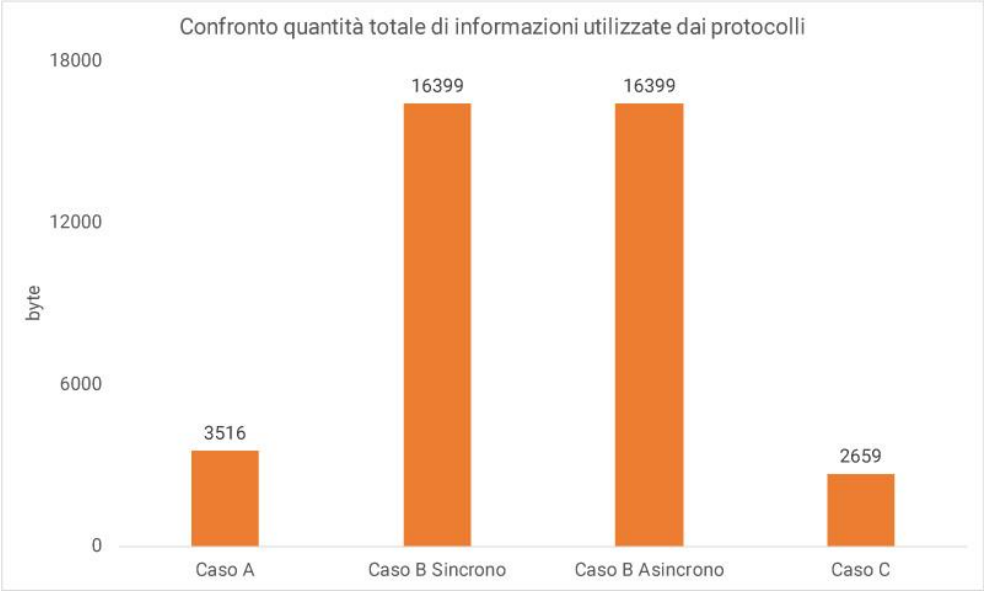

Figura 48 - Diagramma di confronto della quantità di informazioni scambiate.

4.4.4 Confronto prestazionale delle soluzioni di data plane

Nella parte conclusiva del piano di test è stato analizzato l'aspetto più interessante, ovvero le prestazioni delle nuove funzionalità di data plane a tempo di comunicazione.

La topologia utilizzata è stata la seguente.

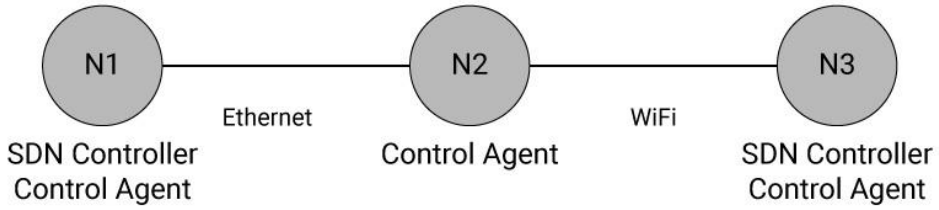

Figura 49 - Topologia per i test prestazionali del data plane.

Lo scenario è stato quello di comunicazione dal nodo N1 al nodo N3, in particolare N1 ha svolto i ruoli sia di Control Agent che di Controller. Il dispositivo utilizzato per N1 è stato l'Acer V3-571G mentre per i nodi N2 e N3 sono stati impiegati dei Raspberry Pi 3 Model B+.

Le prove svolte non si sono focalizzate esclusivamente sulle nuove alternative di routing che questa estensione offre, ma sono stati effettuati anche dei confronti con le soluzioni già presenti nel middleware così da comprendere le caratteristiche di ciascuna. I meccanismi confrontati sono stati:

- Comunicazione con routing a livello di sistema operativo.
- Comunicazione con routing distribuito nell'accezione base di RAMP.
- Comunicazione con instradamento basato su flow con strategia di Rerouting.
- Comunicazione con instradamento basato su flow con strategia di Rerouting e applicazione di regole. In particolare si sono state analizzate le prestazioni in caso di applicazione di due tipi di regole a costo computazionale crescente: una regola che si occupa di ispezionare l'header di un pacchetto per riconoscerne il tipo di dato e una regola che invece effettua la deserializzazione del pacchetto per consultarne il suo contenuto.

Le due regole di data plane sopra citate sono state prese in considerazione in quanto particolarmente significative: la prima consente di approssimare una regola in grado di prendere decisioni di routing in base al tipo, utile ad esempio in uno scenario di sensoristica dove si vogliono raccogliere statistiche sul tipo di payload in transito e prendere decisioni di routing in base ai dati raccolti; la seconda invece consente, tramite "deep packet inspection", di prendere decisioni di routing non solo in base al tipo ma anche al suo contenuto corrente. Una regola di questo tipo permette ad esempio di prendere decisioni di instradamento se un valore di interesse del pacchetto trasportato

ricade all'interno di un intervallo specificato all'interno della regola. Quest'ultima alternativa mette quindi a disposizione un meccanismo di altissimo livello, che da un lato permette di raggiungere un grado di flessibilità elevato e dall'altro introduce un overhead dovuto al costo di deserializzazione.

Lo scenario considerato è stato l'invio di una raffica di pacchetti della durata media di 15 secondi con le seguenti frequenze:

- 1 pacchetto al secondo.
- 10 pacchetti al secondo.
- 50 pacchetti al secondo.
- 100 pacchetti al secondo.
- 250 pacchetti al secondo.

e considerando pacchetti delle dimensioni seguenti:

- 100 byte.
- 1024 byte.
- 5120 byte.
- 10240 byte.

I parametri prestazionali presi in esame sono stati:

- Tempo totale medio di completamento dell'operazione, considerando il tempo di invio del primo pacchetto e il tempo di ricezione dell'ultimo.
- Consumo medio di CPU della Java Virtual Machine sul nodo intermedio N2.
- Consumo medio di memoria della Java Virtual Machine su nodo intermedio N2.
- Nel caso di applicazione di regole di data plane il tempo medio necessario alla loro applicazione.

Considerando la natura eterogenea delle connessioni della topologia presa in esame è stato utilizzato il protocollo TCP e a riguardo è stato valutato, sul nodo destinazione N3, il rispetto dell'ordine di invio dei pacchetti che sono stati opportunamente identificati da un numero di sequenza. Lo scopo di quest'ultimo aspetto è legato alla natura distribuita di RAMP: ogni pacchetto inviato viene considerato dal middleware come un messaggio singolo e non

come parte di un flusso, pertanto ad ogni inoltro viene appositamente creata una socket. I messaggi di ogni socket sono inoltrati in un thread dedicato per migliorare il parallelismo e a riguardo, nell'implementazione di RAMP, vi è un componente che gestisce i thread responsabili della trasmissione dei pacchetti, in un poll. Tale modulo, chiamato *ThreadPool*, di base ha una configurazione per cui la dimensione del pool è limitata al fine di non saturare le risorse hardware del dispositivo in cui il middleware è in esecuzione. Alla luce di questa osservazione, per lo svolgimento dei test, il componente è stato opportunamente modificato per supportare l'invio ad alta frequenza dei pacchetti evitando di introdurre un ritardo significativo che avrebbe falsato le rilevazioni.

Infine, dato il numero elevato di combinazioni considerate nello svolgimento delle prove, si mostreranno dei risultati qualitativi allo scopo di fornire una visione generale delle prestazioni del sistema. Un'analisi più accurata verrà condotta successivamente nella fase di approfondimento del lavoro di tesi.

Il primo aspetto preso in esame è stato il confronto dei tempi necessari al completamento del trasferimento di una raffica variabile di pacchetti della dimensione di 5 KB nell'arco di 15 secondi. Come anticipato, le alternative considerate sono state:

- Comunicazione con routing a livello di sistema operativo.
- Comunicazione con routing distribuito nell'accezione base di RAMP.
- Comunicazione con instradamento basato su flow con strategia di Rerouting.
- Comunicazione con instradamento basato su flow con strategia di Rerouting e applicazione di una regola di ispezione dell'header per recuperare il tipo di dato trasportato.
- Comunicazione con instradamento basato su flow con strategia di Rerouting e applicazione di una regola di deserializzazione del pacchetto.

Per quanto concerne il routing a livello di sistema operativo, tramite lo strumento di analisi *iPerf*, la banda media disponibile è stata di:

- 19.8 Mbit/s con deviazione standard di 1.43 Mbit/s per il percorso da N1 a N3.

- 17,42 Mbit/s con deviazione standard di 1,97 Mbit/s per il percorso da N3 a N1.

Nei test è stato utilizzato solamente la direzione da N1 a N3.

Nella seguente tabella si mostra il risultato delle rilevazioni.

Pacchetti al secondo	OS Routing	RAMP di base	Rerouting	Ispezione Header	Deserializ-zazione
1	15.060 s	14.037 s	15.045 s	15.055 s	15.068 s
10	17.383 s	14.344 s	15.151 s	15.854 s	15.850 s
50	17.038 s	15.804 s	15.560 s	15.565 s	16.179 s
100	17.600 s	15.660 s	15.741 s	17.601 s	17.812 s
250	18.704 s	24.447 s	21.040 s	34.967 s	36.209 s

Tabella 8 - Dati dei tempi rilevati nel test di latenza per pacchetti di 5 KB.

In Figura 46 si riporta la visualizzazione grafica dei dati riportati in tabella.

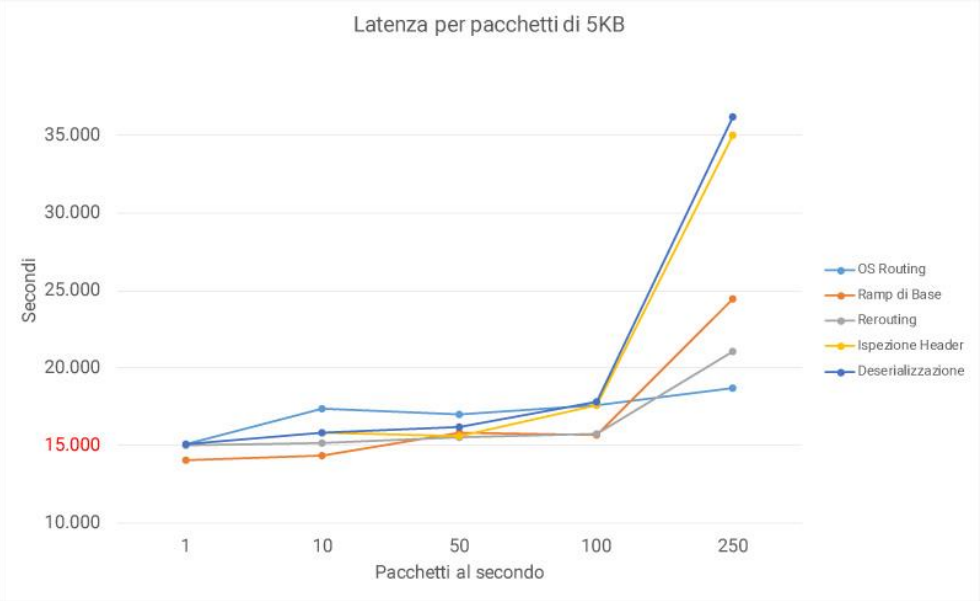

Figura 50 - Grafico di comparazione della latenza.

Dal grafico si evince che fino ad una frequenza di 100 pacchetti al secondo tutte le alternative che il middleware mette a disposizione completano l'invio dei pacchetti nell'intorno dei 15 secondi. Le prestazioni degradano sensibilmente in caso di 250 pacchetti al secondo, ad eccezion fatta per il routing a livello di sistema operativo (curva celeste) che subisce un lieve aumento della latenza. È possibile dedurre un primo risultato significativo: un incremento notevole della latenza in caso di utilizzo di instradamento con applicazione di regole a fronte di una frequenza crescente nell'invio dei pacchetti. In particolare, tra le soluzioni di routing applicativo, la comunicazione con applicazione della regola di deserializzazione (curva blu) è quella che in tutti i casi necessita di maggior tempo per essere completata, seguita immediatamente

dalla regola di ispezione dell'header (curva gialla). Il secondo risultato degno di nota è l'ottimo comportamento del routing di basso livello, che ha dimostrato garantire prestazioni pressoché invariate in tutte le frequenze di invio considerate; questo risultato, se confrontato con il routing applicativo di RAMP, è dovuto all'inoltro effettuato a livello di kernel che, essendo meno sofisticato del routing applicativo, comporta un overhead limitato sui nodi intermedi. Al contrario, le comunicazioni che sfruttano il routing di RAMP sono caratterizzate da inoltri point-to-point che da un lato consentono un instradamento di alto livello, dall'altro introducono un overhead che, a fronte di alte frequenze di invio, peggiora le prestazioni in modo considerevole. Il caso estremo è rappresentato dalla comunicazione che utilizza l'applicazione di regole dove il prezzo da pagare per l'utilizzo di politiche più raffinate, rispetto alla strategia di Rerouting (curva grigia), è un maggiore ritardo nell'inoltro dei pacchetti. In conclusione si può constatare che sebbene il routing a livello di sistema operativo sia stato introdotto per garantire retrocompatibilità con i dispositivi legacy che vogliono accedere ai servizi offerti dalla rete spontanea, può essere anche una valida alternativa in caso si voglia intraprendere comunicazione ad alte prestazioni tra nodi RAMP all'interno della rete. Questa soluzione presenta tuttavia due limiti:

- Il numero di canali dedicati a disposizione tra due Control Agent è solitamente limitato.
- È fortemente suscettibile alla mobilità dei nodi, infatti ogni nodo intermedio rappresenta un singolo punto di fallimento nella comunicazione.

In questo scenario di test, oltre alla latenza, sono state misurate anche le risorse computazionali e di memoria utilizzate da N2 così da valutare il carico a cui sono sottoposti i nodi intermedi quando attuano il routing a livello applicativo. Le rilevazioni riportate nelle due seguenti tabelle fanno riferimento all'utilizzo di CPU e memoria della Java Virtual Machine in esecuzione sul nodo N2 e sono state ottenute utilizzando lo strumento di profilazione VisualVM nella versione 1.3.9 [32].

Pacchetti al secondo	RAMP di base	Rerouting	Ispezione Header	Deserializzazione
1	13 %	13.9 %	14.1 %	13.8 %
10	14.45 %	15.9 %	18.35 %	17.7 %
50	24.2 %	25.2 %	27.7 %	29.4 %
100	28 %	29 %	36.4 %	38.7 %
250	34.46 %	32.68 %	37.5 %	40.7 %

Tabella 9 - Carico medio CPU del nodo intermedio N2.

Pacchetti al secondo	RAMP di base	Rerouting	Ispezione Header	Deserializza-zione
1	14.94 MB	12.52 MB	15.75 MB	14.98 MB
10	15.82 MB	13.44 MB	14.84 MB	14.15 MB
50	15.08 MB	15.22 MB	16.45 MB	16.06 MB
100	16.93 MB	14.74 MB	17.2 MB	17.56 MB
250	17.31 MB	17.1 MB	18.02 MB	18.86 MB

Tabella 10 - Carico medio di memoria del nodo intermedio N2.

Di seguito si riportano il grafico riassuntivo della Tabelle 5:

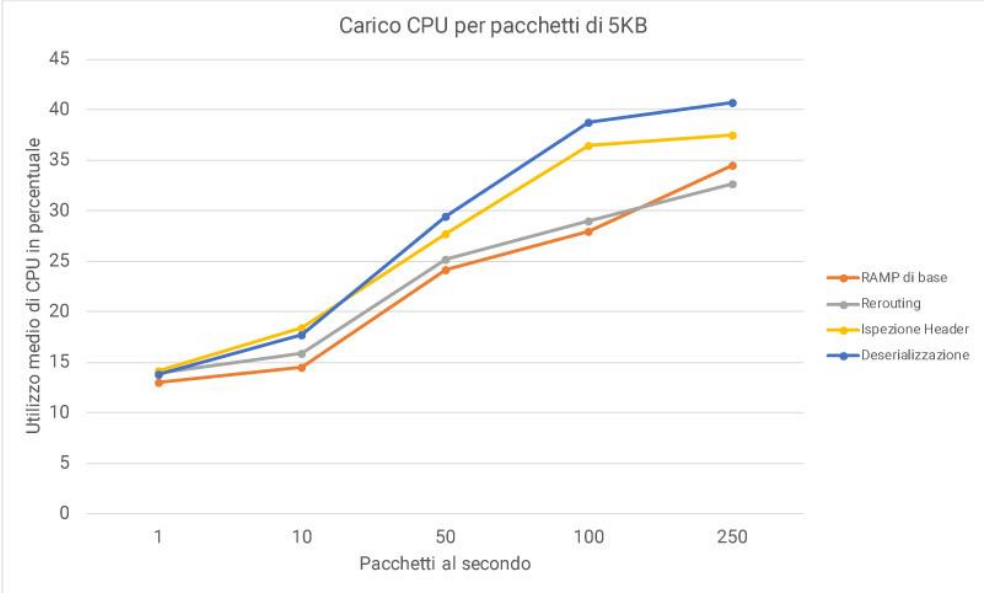

Figura 51 - Consumo medio di CPU sul nodo N2.

Dal grafico in Figura 49 si può notare come l'utilizzo di CPU rifletta l'ordine di complessità crescente delle alternative possibili. La soluzione che richiede meno risorse computazionali è infatti la comunicazione che utilizza RAMP di base (curva arancione), seguita dal Rerouting (curva grigia) e dalla regola di ispezione dell'header (curva gialla). Infine vi è la regola di deserializzazione (curva blu) che richiede maggiori risorse computazionali. In generale il carico medio di CPU per tutti i tipi di instradamento ha un andamento quasi lineare per frequenze che vanno da 1 a 100 pacchetti al secondo. A 250 pacchetti al secondo, per l'applicazione delle regole, si va incontro al collo di bottiglia introdotto dal *ThreadPoll* che, per garantire un consumo contenuto delle risorse, ritarda l'invio dei pacchetti con conseguente incremento della latenza. Dall'andamento delle curve è possibile infine dedurre che per frequenze da 50 a 250 pacchetti al secondo rispetto al caso di Rerouting vi è un incremento medio del:

- 4.9 % in caso di utilizzo di regola di ispezione dell'header.
- 7.3 % in caso di utilizzo di regola di deserializzazione.

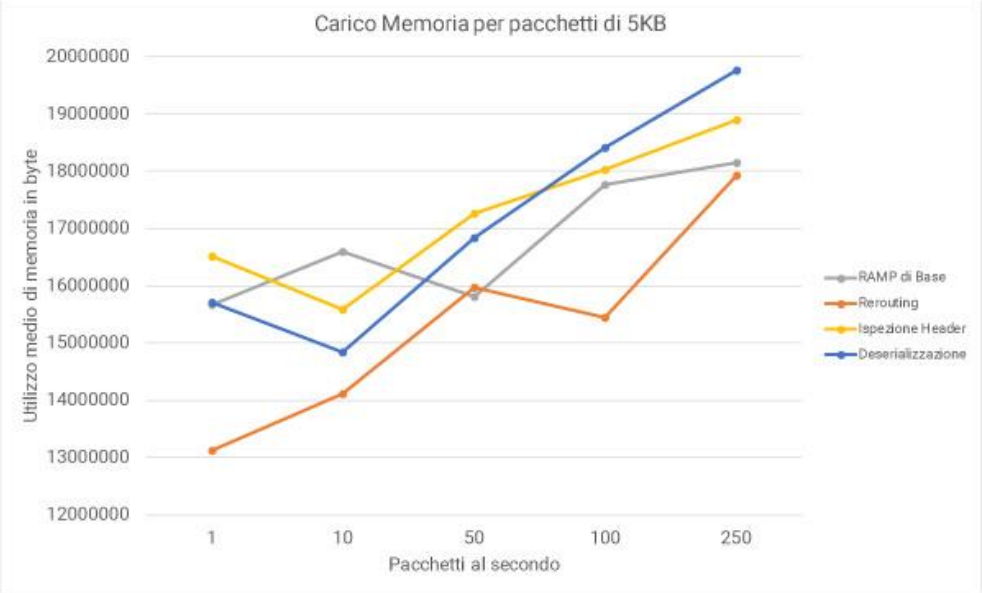

Figura 52 - Carico medio di memoria sul nodo N2.

Se l'andamento del carico di CPU è per lo più lineare, non si può dire la stessa cosa per quanto riguarda l'utilizzo di memoria. A livello asintotico tutte le soluzioni tendono ad avere un incremento lineare a fronte di un aumento della frequenza di invio tuttavia da 1 a 50 pacchetti al secondo vi sono delle fluttuazioni che non consentono di dedurre uno schema significativo legato alla complessità delle tecniche di routing utilizzate. L'aumento della memoria per un alto numero di pacchetti al secondo è dovuto probabilmente al numero considerevole di thread responsabili dell'inoltro dei pacchetti istanziati dal *ThreadPool* per far fronte all'elevato traffico in transito.

Per valutare l'impatto comportato dall'utilizzo delle regole sono state effettuate ulteriori misurazioni sui tempi necessari alla loro applicazione. Si riportano di seguito i dati rilevati sul nodo N2 al variare della dimensione del payload in caso di frequenza di invio a 250 pacchetti al secondo.

Regola	**100 byte**	**1 KB**	**5 KB**	**10 KB**
Ispezione Header	26 ms	33 ms	45 ms	80 ms
Deserializzazione	54 ms	142 ms	167 ms	255 ms

Tabella 11 - Latenza dovuta all'applicazione di regole di data plane.

Nella seguente figura si illustra la presentazione grafica dei dati riportati in tabella.

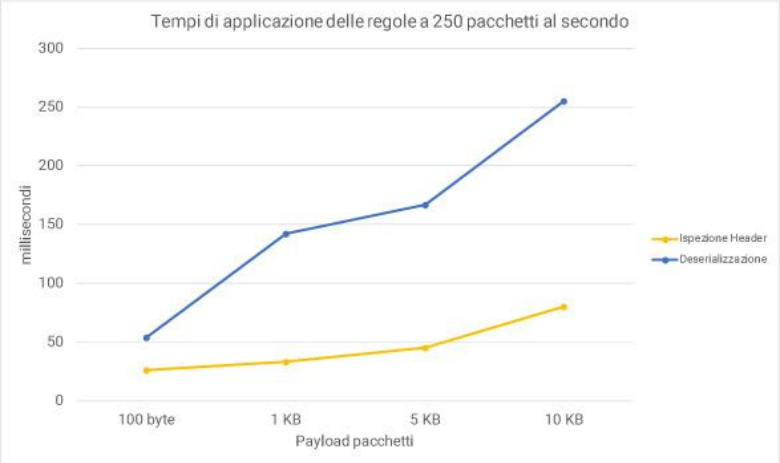

Figura 53 - Grafico di comparazione dei tempi di applicazione delle regole.

Si è deciso di mostrare il comportamento a fronte di 250 pacchetti al secondo in quanto è quello che mette in luce tutte le caratteristiche prestazionali delle funzionalità di routing avanzato introdotte da RAMP Multi-LANE. In questo scenario si mostra uno stato di congestione del nodo intermedio N2, il quale deve applicare le regole a fronte di un elevato traffico in entrata. I valori medi dei dati riportati in Tabella 6 sono molto elevati, ma è tuttavia importante dire che tali valori sono caratterizzati da una varianza altissima dovuta proprio alla congestione del nodo. Normalmente i tempi di applicazione in caso di flusso regolare rientrano nell'ordine di 1-3 millisecondi per l'applicazione della regola di ispezione del tipo di dato e di 3-10 millisecondi per l'applicazione della regola di deserializzazione in base alla dimensione del payload. In generale si può affermare che la regola di ispezione (curva gialla) scala meglio a fronte di un incremento del payload del tipo trasportato e in caso di congestione dovuto al traffico elevato in entrata e ad un consumo elevato di CPU. Per quanto invece riguarda la regola di deserializzazione (curva blu), le sue prestazioni sono fortemente influenzate dalla dimensione del payload dei pacchetti. Questa impiega più tempo a deserializzare pacchetti di dimensioni elevate, introducendo una latenza che in questo caso risulta significativa, provocata anche dal collo di bottiglia introdotto dal *ThreadPool*.

L'ultimo aspetto preso in esame è un parametro puramente qualitativo frutto delle osservazioni effettuate durante la raccolta dei dati che si è ritenuto opportuno riportare. RAMP non garantisce in fase di ricezione l'ordine di invio dei pacchetti, poiché effettua l'inoltro dei pacchetti point-to-point a livello applicativo. Si è deciso quindi di confrontare il grado di disordine introdotto da tutte le soluzioni di routing applicativo più avanzate, escludendo RAMP

di base, e dall'instradamento attuato a livello di sistema operativo. Al fine di misurare il livello di disordine è stata utilizzata la seguente funzione di ranking.

$$\frac{|ordineDiArrivoPacchetto - ordineDiPartenzaPacchetto|}{numeroTotaleDiPacchettiTrasmessi}$$

L'intervallo di valori del codominio della funzione è [0, ∞) dove:

- 0 indica il rispetto totale dell'ordine di invio dei pacchetti in fase di ricezione.
- ∞ (infinito) indica il non rispetto totale dell'ordine di invio dei pacchetti in fase di ricezione.

Il valore assoluto al numeratore è stato utilizzato per coprire il caso di arrivo in anticipo di un pacchetto.

In Tabella 8 sono riportati i valori della funzione di ranking in caso di invio di un pacchetto della dimensione di 5 KB al variare della frequenza di invio dei pacchetti.

Pacchetti al secondo	OS Routing	Rerouting	Ispezione Header	Deserializzazione
1	0	0	0	0
10	0	0.78	0.42	0.44
50	0	0.36	1.12	1.65
100	0	9.25	98.27	101
250	0	146	172	163

Tabella 12 - Valori della funzione di ranking per pacchetti di 5 KB.

In Figura 52 è mostrato il grafico comparativo dei dati presenti in tabella.

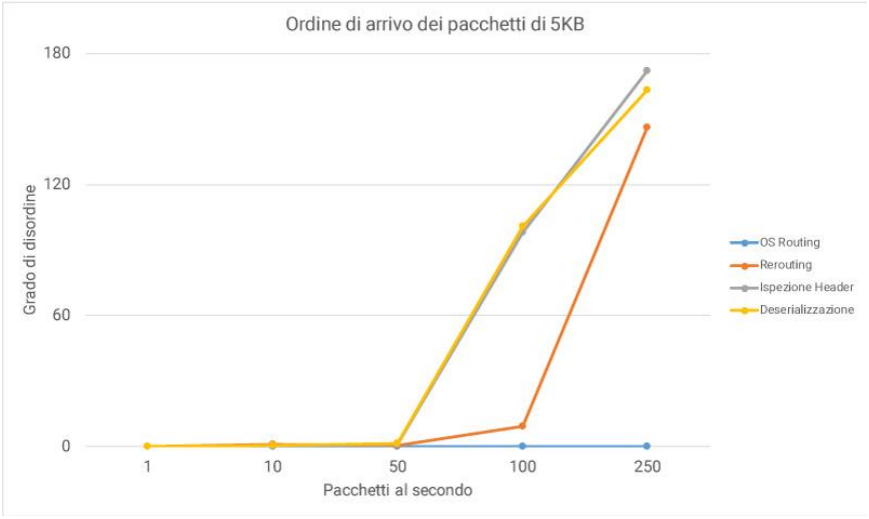

Figura 54 - Grafico di confronto ordine pacchetti.

L’esito della prova per quanto riguarda il routing a livello di sistema operativo è alquanto scontato: l’ordine viene sempre rispettato in ogni circostanza. L’aspetto più interessante è invece legato alle tecniche di routing applicativo. Fino a 50 pacchetti al secondo tutte le alternative considerate si comportano in maniera accettabile introducendo un livello di disordine minimo. Se fino a 100 pacchetti al secondo la strategia di Rerouting continua a reggere, la stessa cosa non si può dire in caso di applicazione delle regole che a parità di frequenza presentano un’impennata notevole. A 250 pacchetti al secondo anche la strategia di Rerouting non è più in grado di sostenere l’ordine di partenza dei pacchetti. In conclusione si può affermare che in caso di applicazione di regole vi è un degrado anticipato di questa proprietà.

Conclusioni

Nell'attività progettuale che ha portato allo sviluppo di questa tesi, si è preso in considerazione l'ambito delle MANET ed in particolare quello delle reti spontanee, sperimentando l'introduzione di nuove funzionalità finalizzate ad aumentarne le capacità. Sfruttando una soluzione di gestione di reti spontanee ispirata al paradigma SDN, fortemente diffuso nell'industria del cloud, si è indagato sulla possibilità di impiegare le tecniche di routing tradizionale in un contesto altamente dinamico e di introdurre un sistema per la manipolazione programmabile dei pacchetti inoltrati durante la comunicazione tra dispositivi mobili.

Il lavoro ha comportato l'estensione di RAMP, un middleware per la gestione dinamica e flessibile di reti spontanee e opportunistiche contraddistinte da eterogeneità di vario tipo, al fine di introdurre le nuove feature frutto del lavoro di ricerca. L'estensione sviluppata, chiamata RAMP Multi-LANE, ha introdotto con successo la possibilità di sfruttare il policy-based routing a livello di sistema operativo per offrire un'alternativa al classico instradamento effettuato a livello applicativo; in secondo luogo ha messo a disposizione un sistema di gestione avanzato per la definizione dinamica, applicazione e attivazione di regole a livello di data plane per l'elaborazione dei pacchetti in transito. Al fine di supportare quest'ultima funzionalità è stato sviluppato anche un Software Development Kit per fornire uno strumento utile alla comunità di sviluppatori che intendono utilizzare il middleware nelle loro sperimentazioni

Attraverso l'esecuzione di un corposo piano di testing, in cui sono stati presi in considerazione numerosi scenari e parametri di valutazione, si è potuto dimostrare il raggiungimento degli obiettivi prefissati e la validazione delle funzionalità sviluppate. Le prestazioni dei protocolli di control plane introdotti dall'estensione si sono rivelati efficienti, con l'unica eccezione del protocollo per il routing di basso livello, la cui latenza del protocollo di control plane è fortemente influenzata dal cambio di contesto dovuto all'interfacciamento con il kernel. Il comportamento dell'estensione a livello di data plane ha evidenziato un buon compromesso tra prestazioni in caso di instradamento di basso livello e latenza dovuta all'impiego di politiche di routing dinamiche a granularità fine che introducono un livello di espressività inedito nel campo delle reti spontanee, flessibilità che si paga anche in termini di consumo aggiuntivo di risorse computazionali e di memoria.

Il lavoro svolto può essere la base per la realizzazione di ulteriori estensioni del middleware. Un primo esempio può riguardare il routing a livello di sistema operativo. Dal momento in cui è stato introdotto un componente in grado di modificare in modo efficiente le tabelle di routing, è possibile concentrarsi sullo studio e lo sviluppo di nuovi algoritmi che cerchino di soddisfare al meglio nuovi requisiti applicativi. Per quanto riguarda invece il sistema di gestione avanzato delle regole, essa si configura come una soluzione che di fatto generalizza tutte le regole di data plane già presenti in RAMP, quelle di routing e quelle di Traffic Engineering basate sulla priorità. Potrebbe quindi essere rilevante riorganizzarle in modo da inglobarle nel nuovo sistema di regole. In ultima analisi, si può prevedere di estendere le capacità della rete contemplando la presenza di più Controller SDN che, tramite delle nuove interfacce eastbound e westbound, siano in grado di coordinarsi dando vita ad un'organizzazione federata della rete.

Riferimenti

[1] J. Elias, «Jocelyne Elias,» [Online]. Available: https://cs.unibg.it/elias/documenti/architetture/6-Routing.pdf. [Consultato il giorno 24 Marzo 2019].

[2] Y. Rekhter, T. Li e S. Hares, «A Border Gateway Protocol 4 (BGP-4),» Gennaio 2006. [Online]. Available: https://tools.ietf.org/html/rfc4271. [Consultato il giorno 12 Marzo 2019].

[3] Cisco Systems, «IS-IS Overview and Basic Configuration,» [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_isis/configuration/15-mt/irs-15-mt-book/irs-ovrw-cf.pdf. [Consultato il giorno 8 Marzo 2019].

[4] J. Moy, «OSPF Version 2,» Aprile 1998. [Online]. Available: https://tools.ietf.org/html/rfc2328. [Consultato il giorno 20 Marzo 2019].

[5] C. Hedrick, «Routing Information Protocol,» Giugno 1988. [Online]. Available: https://tools.ietf.org/html/rfc1058. [Consultato il giorno 17 Marzo 2019].

[6] Information Sciences Institute, University of Southern California, “RFC 791 - Internet Protocol,” 1981. [Online]. Available: https://tools.ietf.org/html/rfc791. [Accessed 01 04 2019].

[7] S. Richardson, «Disable IP Source Routing,» 25 Marzo 2019. [Online]. Available: https://www.ccexpert.us/basic-security-services/disable-ip-source-routing.html. [Consultato il giorno 01 Aprile 2019].

[8] M. G. Marsh, «IPROUTE2 Utility Suite Documentation,» [Online]. Available: http://www.policyrouting.org/iproute2.doc.html#ss9.2. [Consultato il giorno Febbraio 2019].

[9] E. C. S. I. Rosen, A. F. N. I. Viswanathan e R. J. N. I. Callon, «RFC 3031 - Multiprotocol Label Switching Architecture,» Gennaio 2001. [Online]. Available: https://tools.ietf.org/html/rfc3031. [Consultato il giorno 20 Marzo 2019].

[10] L. Andersson, I. Minei e B. Thomas, «LDP Specification,» Ottobre 2007. [Online]. Available: https://tools.ietf.org/html/rfc5036. [Consultato il giorno 11 Maggio 2019].

[11] A. Farrel, A. Ayyangar e J. Vasseur, «Inter-Domain MPLS and GMPLS Traffic Engineering -- Resource Reservation Protocol-Traffic Engineering (RSVP-TE) Extensions,» Febbraio 2008. [Online]. Available: https://tools.ietf.org/html/rfc5151. [Consultato il giorno 8 Marzo 2019].

[12] Cisco Systems, «Introduction to Segment Routing,» [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/seg_routing/configuration/xe-3s/segrt-xe-3s-book/intro-seg-routing.pdf. [Consultato il giorno 5 Aprile 2019].

[13] P. Bellavista, «Mobile Systems M,» Gennaio 2019. [Online]. Available: http://lia.disi.unibo.it/Courses/sm1819-info/. [Consultato il giorno 13 Febbraio 2019].

[14] Project Floodlight, «Project Floodlight,» [Online]. Available: http://www.projectfloodlight.org/. [Consultato il giorno 12 Maggio 2019].

[15] Linux Foundation, «OpenDayLight,» [Online]. Available: https://www.opendaylight.org/. [Consultato il giorno 12 Maggio 2019].

[16] D. Erickson, «Beacon,» [Online]. Available: https://www.sdxcentral.com/projects/beacon/. [Consultato il giorno 12 Maggio 2019].

[17] K. Benzekki, A. E. Fergougui e A. E. Elalaoui, «Software-defined networking (SDN): a survey,» 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/sec.1737. [Consultato il giorno 15 Aprile 2019].

[18] O. N. Foundation, «https://www.opennetworking.org/news-and-events/press-releases/open-networking-foundation-introduces-northbound-interface-working-group/,» 16 Ottobre 2013. [Online]. Available: https://www.opennetworking.org/news-and-events/press-releases/open-networking-foundation-introduces-northbound-interface-working-group/. [Consultato il giorno 2019 Aprile 20].

[19] M. Casado, «Ethane: taking control of the enterprise,» 2007. [Online]. Available: https://dl.acm.org/citation.cfm?id=1282382. [Consultato il giorno 21 Aprile 2019].

[20] S. Evans, «The history of OpenFlow,» [Online]. Available: https://www.computerweekly.com/feature/The-history-of-OpenFlow. [Consultato il giorno 20 Aprile 2019].

[21] O. N. Foundation, «OpenFlow Switch Specification,» 16 Marzo 2015. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf. [Consultato il giorno 15 Dicembre 2018].

[22] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal e J. Halpern, «Forwarding and Control Element Separation (ForCES),» Marzo 2010. [Online]. Available: https://tools.ietf.org/html/rfc5810. [Consultato il giorno 24 Aprile 2019].

[23] J. Vasseur e J. L. Roux, «Path Computation Element (PCE) Communication Protocol (PCEP),» Marzo 2009. [Online]. Available: https://tools.ietf.org/html/rfc5440. [Consultato il giorno 15 Aprile 2019].

[24] D. Farinacci, V. Fuller, D. Meyer e D. Lewis, «The Locator/ID Separation Protocol (LISP),» Gennaio 2013. [Online]. Available: https://tools.ietf.org/html/rfc6830. [Consultato il giorno 20 Aprile 2019].

[25] T. V. Lakshman, T. Nandagopal, R. R. K e S. T. Woo, «The SoftRouter Architecture,» Gennaio 2004. [Online]. Available:

https://www.microsoft.com/en-us/research/publication/the-softrouter-architecture/. [Consultato il giorno 21 Aprile 2019].

[26] Cisco Systems, «Application-Engineered Routing,» 2016. [Online]. Available: https://www.segment-routing.net/conferences/2016-application-engineered-routing/. [Consultato il giorno 13 Aprile 2019].

[27] P. Bellavista, A. Corradi e C. Giannelli, «Middleware for Differentiated Quality in Spontaneous Networks,» Marzo 2012. [Online]. Available: https://ieeexplore.ieee.org/document/5975133. [Consultato il giorno 25 Ottobre 2018].

[28] P. Bellavista, C. Giannelli, S. Lanzone, G. Riberto, C. Stefanelli e M. Tortonesi, «A Middleware Solution for Wireless IoT Applications,» 3 Novembre 2017. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5713098/. [Consultato il giorno Dicembre 2018].

[29] M. G. Marsh, Policy Routing Using Linux (Professional), SAMS, 2001.

[30] Apache Commons, «Apache Commons Exec,» 2 Novembre 2014. [Online]. Available: https://commons.apache.org/proper/commons-exec/. [Consultato il giorno 15 Marzo 2019].

[31] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer e K. Prabhu, «iPerf - The ultimate speed test tool for TCP, UDP and SCTP,» [Online]. Available: https://iperf.fr/. [Consultato il giorno 30 Maggio 2019].

[32] J. Sedlacek e T. Hurka, «VisualVM,» 2019. [Online]. Available: https://visualvm.github.io/. [Consultato il giorno 13 Maggio 2019].