Corso di Laurea in Informatica

TESI DI LAUREA

Studio della gestione degli intenti nel controller SDN Teraflow

Relatore:
Federica Paganelli

Correlatore:
Stefano Forti

Candidato:
Carolina Ferrari

ANNO ACCADEMICO 2023/2024

Indice

Elenco delle figure

1. Introduzione

L'evoluzione delle reti moderne, accompagnata dall'incremento delle esigenze di prestazioni, affidabilità e dall'aumento della quantità del traffico, ha portato alla necessità di sviluppare nuovi paradigmi di gestione e controllo delle reti.
In questo contesto, il Software Defined Networking (SDN) ha rivoluzionato la gestione delle reti attraverso un approccio logicamente centralizzato e programmabile separando il piano di controllo dal piano dati. Questa separazione permette agli amministratori di rete di semplificare configurazioni complesse e di introdurre politiche di rete dinamiche e affidabili. Inoltre, SDN fornisce il supporto per interfacce nordbound e southbound che facilitano rispettivamente l'integrazione con varie applicazioni di terze parti e con i dispositivi nella rete, contribuendo a un ecosistema più flessibile e reattivo [11].
La tecnologia SDN è oggi largamente impiegata in diversi contesti. Nei cloud data center, ad esempio, consente una gestione semplificata e ottimizzata dell'infrastruttura di rete, migliorando la distribuzione delle risorse e l'efficienza delle attività amministrative riducendo allo stesso tempo i costi [12]. Nelle reti di trasporto, SDN viene utilizzato per gestire il traffico in modo più dinamico e flessibile, permettendo un'ottimizzazione della larghezza di banda su lunghe distanze. Allo stesso modo, nelle reti mobili, SDN offre un controllo più efficiente dell'infrastruttura di comunicazione tra le applicazioni, i servizi in cloud e l'utente finale. In particolare, con l'avvento del 5G, l'SDN assume un ruolo fondamentale nel regolare dinamicamente la larghezza di banda per ciascun punto di accesso radio fornendo una gestione flessibile dei router e delle risorse di rete[13] [14].
Tuttavia, nonostante l'SDN astragga il controllo di gestione dai dispositivi, le sue funzioni rimangono principalmente focalizzate sulla configurazione dei singoli componenti della rete. Le attuali interfacce SDN, infatti, si concentrano sulla definizione dettagliata dei percorsi per la trasmissione dei dati piuttosto che su un modello più astratto che permetta agli amministratori di esprimere in modo semplice cosa desiderano ottenere dalla rete [11]. Questo approccio, benché efficace nella gestione delle risorse di rete, limita parzialmente la flessibilità e l'automazione in contesti fortemente dinamici, come quelli caratterizzati

da architetture di Edge Computing o reti altamente virtualizzate.

Per questo motivo è stato introdotto l'Intent-Based Networking (IBN), o Networking basato su intenti, un paradigma che mira a separare ulteriormente la complessità di implementazione dal livello di gestione. IBN affronta le sfide emergenti fornendo un'astrazione di alto livello che consente di esprimere gli obiettivi di rete in modo chiaro e intuitivo. Ad esempio, un intento potrebbe essere: *"consentire alle applicazioni contabili di accedere al server XYZ, ma non consentire l'accesso alle applicazioni di produzione"* oppure *"mantenere sempre un'elevata qualità del servizio e un'elevata larghezza di banda per gli utenti di un certo livello"* [15].

Il framework IBN si occupa di tradurre gli intenti in configurazioni di rete specifiche adattando dinamicamente la rete per rispettarli. Grazie al paradigma SDN, IBN non richiede l'inserimento manuale dei comandi di policy: una volta ricevute le richieste di servizio, le converte in Key Performance Indicators (KPI), che rappresentano le metriche rilevanti da monitorare. Ciò permette una verifica continua dello stato della rete, assicurandosi che quest'ultima soddisfi le richieste attraverso un monitoraggio in tempo reale delle KPI, garantendo così un'elevata qualità dell'esperienza (QoE) [16]. Questo approccio consente quindi alla rete di adattarsi dinamicamente ai cambiamenti, gestendo automaticamente le complessità sottostanti, migliorando l'efficienza e la reattività.

Alcuni controller SDN implementano già il concetto di Intent-Based Networking.

ONOS, ad esempio, integra un componente chiamato Intent Framework, che consente alle applicazioni di esprimere le loro esigenze tramite direttive basate su policy. In modo diverso, OpenDayLight ha sviluppato l'Intent Northbound Interface (NBI), un progetto attivo fino alla release "Oxygen" del 2018 ma successivamente abbandonato per favorire approcci più pratici e scalabili. Cisco, con il suo DNA Center, ha introdotto soluzioni IBN che automatizzano la configurazione e l'implementazione della rete grazie all'intelligenza artificiale e all'apprendimento automatico concentrandosi principalmente sulla sicurezza. Ad esempio, quando un nuovo dispositivo viene aggiunto all'infrastruttura, il sistema assegna automaticamente policy basate sulla sua identità, proponendosi di risolvere i problemi di funzionamento della rete [16].

Recentemente nell'ambito del progetto europeo TeraFlow H2020, è stato sviluppato un

controller SDN omonimo. L'obiettivo di questa tesi è lo studio del controller open source Teraflow, sviluppato da ETSI [17] a partire dal 2020. In particolare il lavoro si è concentrato nello studio di come in TeraFlow vengano modellati e gestiti gli "intenti" attraverso le componenti di policy e service. Nella prima fase, è stato necessario comprendere l'architettura del controller per capire come le diverse componenti interagiscano tra loro e come queste supportino la gestione dei servizi e delle politiche di rete. A causa della scarsità di documentazione, l'analisi dell'architettura e del funzionamento del controller è stata accompagnata dallo studio del codice e da test basati su scenari forniti dagli sviluppatori del progetto. Questi test sono stati cruciali per comprendere il comportamento del sistema in condizioni applicative specifiche vista la scarsità della documentazione aggiornata. Successivamente, si è passati alla fase di sperimentazione. E' stata condotta una fase di sperimentazione volta a verificare la reazione del controller a guasti o a cambiamenti nelle risorse e nei dispositivi disponibili. L'obbiettivo era di riuscire a implementare delle politiche di gestione della rete per far cambiare automaticamente il percorso tra due endpoints quando le metriche richieste non erano più rispettate. Per garantire questo funzionamento si è implementato un sistema di monitoraggio continuo delle metriche specificate nella politica. Queste informazioni venivano inviate al controller che le utilizzava per prendere le decisioni in tempo reale riducendo al minimo l'impatto negativo sull'esperienza degli utenti.

Il resto del documento sarà organizzato come segue.

Nel Capitolo 2 verrà esposto il contesto. All'interno ci sarà un'analisi approfondita del Software Defined Networking (SDN), dell'Intent-Based Networking (IBN) e verranno illustrati alcuni dei principali controller SDN presenti sul mercato che implementano soluzioni differenti per l'IBN. Alla fine del capitolo saranno introdotti Kubernetes e i microservizi che stanno alla base del design di TeraFlow.

Nel Capitolo 3 verrà descritto in modo approfondito il controller Teraflow su cui si incentra la tesi. Verrà delineata l'architettura e l'interazione necessaria tra le componenti con attenzione particolare alle componenti coinvolte nella gestione dei servizi e delle policy. Sarà inoltre introdotto gRPC, un framework grazie al quale le componenti comunicano tra di loro.

Nel Capitolo 4 vengono presentati gli esperimenti svolti che costituiscono la parte pratica del lavoro. Dopo aver introdotto la teoria e gli strumenti necessari, viene definita la configurazione di un servizio di rete tra due end-point. Gli esperimenti includono la gestione automatica delle politiche di rete e il monitoraggio dei parametri con l'adozione di misure correttive nel caso essi non siano più rispettati.
Infine, nell'ultimo Capitolo, saranno presentate le conclusioni.

2. Background

Nel seguente Capitolo verranno presentate le principali tecnologie alla base di questo lavoro, ovvero il Software Defined Networking (SDN) e l'Intent-Based Networking (IBN). Verranno descritti due dei controller SDN più rilevanti allo stato dell'arte, ONOS [18] e OpenDaylight (ODL) [19] e infine Kubernetes e i microservizi, tecniche alla base del controller TeraFlow.
Questa panoramica fornirà il contesto necessario per comprendere meglio i successivi sviluppi trattati nel documento.

2.1 Software Defined Networking

L'architettura tradizionale di rete si basa su dispositivi fisici di interconnessione che facilitano la comunicazione tra più host a livello locale e consentono lo scambio di informazioni. Nell'architettura tradizionale ciascun dispositivo integra al suo interno sia le funzioni del piano dati (data plane) che del piano di controllo (control plane).
Il piano dati è responsabile della ricezione, del processamento e dell'inoltro dei pacchetti in base alle tabelle di routing che associano un indirizzo a una data porta d'uscita [20]. Queste tabelle vengono gestite dal piano di controllo che calcola i percorsi per l'instradamento in base alla destinazione dei paccchetti e aggiorna le tabelle dei dispositivi. Nei protocolli di rete tradizionali, questi due piani operano separatamente all'interno dei dispositivi svolgendo i loro compiti in maniera indipendente.
Per determinare i percorsi di rete esistono diversi protocolli di routing che adottano un approccio decentralizzato.
RIP (Routing Information Protocol), ad esempio, utilizza un algoritmo di distance-vector in cui ogni nodo conosce solo le informazioni dai suoi vicini, quindi non la conformazione globale dell'infrastruttura, e aggiorna la propria tabella sulla base dei messaggi di routing scambiati [21]. Il protocollo OSPF (Open Shortest Path First) adotta invece un approccio globale. In questo caso, ogni router invia e riceve dei messaggi *"link state"*,

attraverso i quali acquisisce informazioni sullo stato dei collegamenti all'interno dell'area, conoscendo così la topologia completa della rete. Infine i router calcolano i percorsi in modo indipendente utilizzando l'algoritmo di Dijkstra [22].
Questo approccio richiede in ogni caso l'esecuzione di un algoritmo di routing che, tramite un protocollo dedicato, scambia messaggi con altri dispositivi per prendere decisioni. Tuttavia, ciò introduce notevoli ritardi [23], rendendo così la rete meno adatta alle nuove esigenze delle applicazioni moderne che necessitano di un'elevata dinamicità.
La complessità e la staticità dell'architettura di rete tradizionale, progettata intorno a una serie di protocolli indipendenti, ciascuno focalizzato su una parte specifica delle esigenze di rete, aumentano le difficoltà nel rispondere alle nuove sfide poste da tecnologie emergenti come il cloud computing, i big data, lo streaming in tempo reale e l'Internet of Things (IoT).
Aggiungere o spostare dispositivi nella rete diventa particolarmente complicato: ogni volta che avviene una modifica, gli operatori devono aggiornare manualmente le configurazioni di numerosi dispositivi introducendo un significativo problema di scalabilità. Per far fronte a limitazioni di capacità e ai picchi di traffico imprevisti, invece di aggiungere collegamenti, molte aziende sovradimensionano quelli già presenti nella rete in base alle previsioni di traffico che però risultano spesso inadeguate.
Un ulteriore ostacolo è rappresentato dalla mancanza di interfacce aperte e standardizzate per le funzioni di rete. Questa dipendenza dai fornitori di apparecchiature con protocolli proprietari riduce la flessibilità e rallenta l'introduzione di nuove funzionalità [24].
Le reti tradizionali, con il loro approccio distribuito e decentralizzato al controllo e all'instradamento, si sono dimostrate inefficaci nel rispondere rapidamente a cambiamenti dinamici.
Il Software Defined Networking (SDN) nasce per rimediare ai suddetti limiti [25]. Proposto negli ultimi anni dalla Open Networking Foundation (ONF) [26], SDN introduce un'architettura che separa il piano di controllo dal piano dati, rendendo quest'ultimo programmabile e semplificando la gestione della rete. A differenza delle reti tradizionali, dove il controllo è distribuito su ogni dispositivo, SDN associa le funzioni di controllo a un dispositivo dedicato, chiamato controller, che gestisce e coordina le politiche di rete,

la configurazione e l'instradamento. Questa separazione permette ai dispositivi di rete di operare come semplici apparati di inoltro, mentre il controller centrale gestisce le decisioni più complesse.

Per poter funzionare, i dispositivi devono essere in grado di comunicare con il controller e riconoscere cambiamenti significativi degni di notifica per una gestione ottimizzata della rete. Questo è possibile tramite l'installazione al loro interno di componenti software con le caratteristiche necessarie detti agent [23].

La base di questo paradigma è un controller remoto che, interagendo con gli agent locali, riceve informazioni sui collegamenti e sul traffico in tempo reale ed è in grado di configurare autonomamente i dispositivi collegati sulla base degli eventi notificati o delle richieste da parte degli utenti. Lo scopo principale è quindi ridurre e semplificare il carico di amministrazione per i singoli dispositivi.

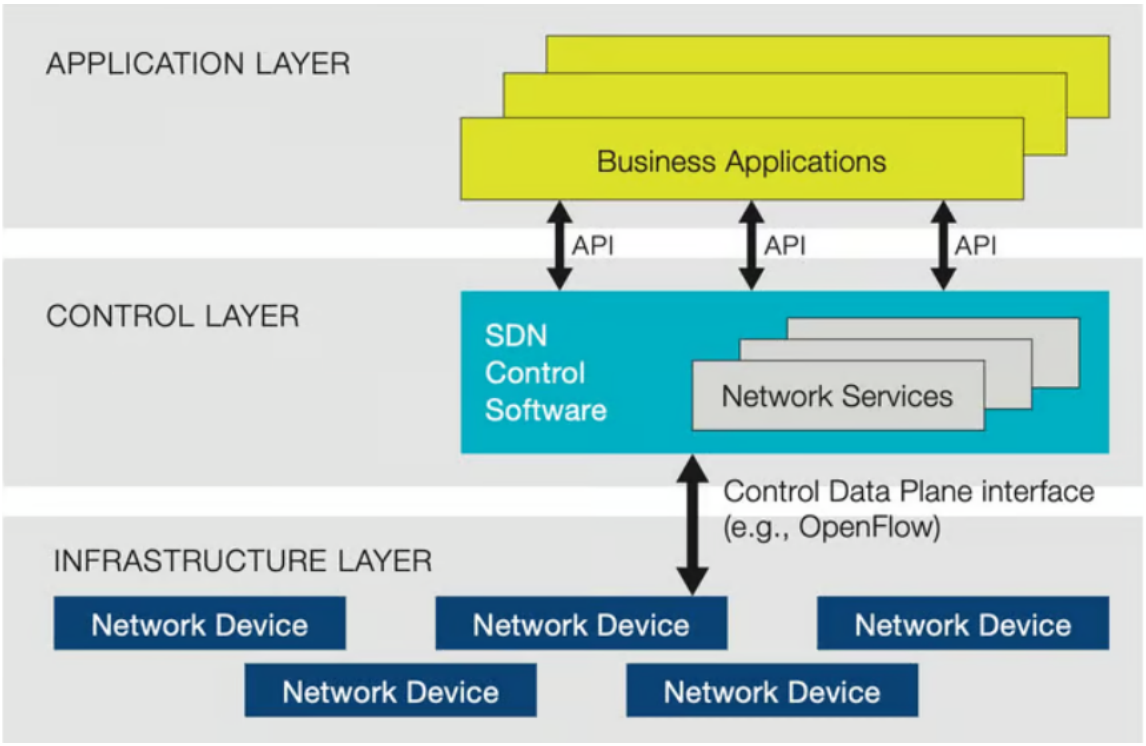


Figura 2.1: Struttura di una rete SDN [1]

Come si può notare dalla Figura 2.1 la rete è stato suddivisa in tre livelli: Infrastructure layer, Control layer e Application layer [27].

Partendo dal livello più basso troviamo l'infrastruttura di rete il cui unico compito è implementare il piano dati, ossia la parte che supporta un protocollo condiviso per comunicare con il controller e installare le regole di inoltro per i paccchetti sulla base delle configurazioni imposte da quest'ultimo. Questa divisione consente di evitare algoritmi di routing all'interno dei dispositivi di rete visto che il routing sarà gestito direttamente dai livelli sovrastanti [23].

Nel control layer si trova il controller SDN che, tramite API northbound (NBI) e southbound (SBI), comunica con gli altri due livelli. Le API NBI consentono al controller di interfacciarsi con le applicazioni e i servizi situati nel livello superiore, mentre le API SBI, tipicamente implementate tramite OpenFLow, permettono al controller di comunicare con i dispositivi di rete nel livello inferiore.

Il control layer consente l'implementazione del piano di controllo, che gestisce la configurazione e l'ottimizzazione delle risorse. Attraverso programmi dinamici e automatizzati, il controller calcola i percorsi ottimali per il traffico di rete non più sulla base della destinazione ma in modo generalizzato sui vari header del pacchetto e, tramite le API SBI, impone regole di inoltro ai dispositivi sottostanti. Questo processo include la manipolazione delle tabelle di routing e l'aggiornamento delle stesse in risposta a eventi in tempo reale.

Il controller è logicamente centralizzato, in questo modo il piano di gestione (management plane) situato sopra di esso interagisce con un unico punto di accesso [20]. Fisicamente può anche essere distribuito, soprattutto nei processi di produzione, per garantire scalabilità e affidabilità [27].

L'application layer comprende le applicazioni e i servizi che sfruttano le capacità della rete SDN per la realizzazione del piano di gestione. Grazie a questo livello si possono definire politiche o intenti da implementare all'interno della rete tramite interfacce grafiche e strumenti dedicati all'utente finale. Queste regole, tramite le API NBI, sono poi comunicate al controller che si occuperà di farle rispettare mediante il costante monitoraggio delle risorse del piano dati. Per esempio, nel contesto dell'application layer, le applicazioni possono includere strumenti tradizionali come firewall, che definiscono politiche di sicurezza per bloccare il traffico da indirizzi IP sospetti [28] o bilanciatori di carico, che distribuiscono

il traffico tra diversi server per evitare sovraccarichi e ottimizzare l'uso delle risorse per migliorare le prestazioni.
Il disaccoppiamento dei vari livelli consente alla rete di diventare direttamente programmabile da un'unica unità centralizzata riuscendo a mantenere una visione globale e permettendo l'astrazione dell'infrastruttura sottostante per affrontare le sfide di gestione incontrate nelle reti moderne.

2.2 Intent-based Networking

L'Intent-Based Networking (IBN) è un paradigma innovativo per la gestione delle reti che permette di separare la complessità di implementazione dal livello di gestione. Esso è nato per rispondere alla crescente ampiezza e dinamicità delle reti moderne, dove la gestione tradizionale basata su comandi manuali e configurazioni dettagliate non è più sostenibile. Negli ultimi anni, grazie a nuove tecnologie come il 5G o l'Internet of Things (IoT), applicazioni innovative stanno emergendo in differenti campi industriali.
In questo contesto, le implementazioni cloud si sono estese ed è diventato essenziale aumentare le capacità di elaborazione, eseguire servizi distribuiti e garantire il comportamento interattivo che queste nuove applicazioni richiedono.
Sono state concepite diverse tecnologie emergenti, oltre all'IBN, per far fronte a queste necessità. Ognuna ha differenti obiettivi e spesso si integrano tra loro [29].
Il Multi-Access Edge Computing[30] (MEC) fornisce funzionalità cloud alla rete per migliorare la qualità dei servizi offerti in tempo reale portando della capacità di calcolo ai punti di accesso.
Il Network Function Virtualization[31] (NFV) permette di distribuire le funzioni di rete (firewall, NAT, DPI) come apparecchi virtuali. Questi vengono forniti in modo flessibile al cloud, consentendo così modelli innovativi di fornitura di servizi che migliorano la flessibilità e l'agilità della rete.
L'IBN nasce come un approccio nuovo, concepito dall'IETF [32] che si occupa della gestione della rete per astrarne la complessità permettendo agli utenti finali di concentrarsi sugli obiettivi di performance senza preoccuparsi dei dettagli tecnici.

L'IBN può essere visto come un'evoluzione dell'SDN, poiché incorpora le sue principali caratteristiche superandone alcuni limiti.

Mentre l'SDN fornisce delle northbound APIs che solitamente sono complesse e richiedono la conoscenza di dettagli tecninci di rete [33], IBN, invece, adotta un approccio più astratto in cui gli utenti possono esprimere le proprie esigenze definendo degli intenti, ovvero una serie di obiettivi di alto livello.

Gli intenti sono espressi in un linguaggio naturale che descrive i risultati desiderati lasciando al sistema IBN il compito di tradurli nei dettagli di configurazione. Un intento di rete si riferisce infatti a un livello di astrazione in cui la logica dell'applicazione è espressa in termini di cosa deve essere fatto, utilizzando regole di semantica, piuttosto che di come deve essere implementato [34]. L'idea centrale dell'IBN è di non specificare i dettagli di implementazione della rete; piuttosto, è la rete stessa che deve eseguire le azioni necessarie per soddisfare gli intenti espressi. In questo modo le applicazioni non devono gestire le direttive di rete di basso livello specifiche della tecnologia. Infatti i livelli applicativi possono interagire con l'Intent Layer evitando di apprendere il linguaggio tecnico-specifico del sistema sottostante.

L'approccio IBN è reso possibile grazie alla mediazione di un Intent Orchestration Layer, che gestisce e regola il ciclo di vita delle richieste di intenti provenienti dalle applicazioni attraverso operazioni di adempimento e garanzia in un flusso di lavoro a ciclo chiuso. [29]. Queste operazioni, oltre a includere la traduzione e l'eventuale orchestrazione di configurazione per la realizzazione dei singoli intenti, mirano a garantire che la rete rispetti effettivamente l'intento desiderato sulla base della raccolta, aggregazione e valutazione in tempo reale dei dati di monitoraggio.

L'IBN fa uso di un Intent Repository, un database in grado di interagire con i moduli di gestione e traduzione degli intenti per fornire la mappatura tra l'intento e la sua configurazione [35]. Questo paradigma offre vantaggi anche ai fornitori di rete, infatti permette di migliorare l'agilità, la disponibilità e la gestione delle reti a un livello di astrazione più elevato e verificare continuamente che gli obiettivi siano raggiunti.

2.3 Controller allo stato dell'arte

Prima di analizzare nel dettaglio TeraFlow di seguito si introducono due controller allo stato dell'arte: ONOS e OpenDayLight. Questi controller rappresentano soluzioni già consolidate nel campo del Software Defined Networking e sono ampiamente utilizzati e studiati sia in ambito accademico che industriale. La loro descrizione ci consentirà di mettere in evidenza le differenti caratteristiche per poter analizzare meglio le innovazioni introdotte da TeraFlow.

2.3.1 ONOS

Open Network Operating System (ONOS) [18] è uno dei controller SDN più noti. E' un progetto nato dalla Open Networking Foundation (ONF) [26] al fine di soddisfare le esigenze degli operatori per poter costruire reali soluzioni SDN/NFV. I principali obiettivi sono quelli di introdurre modularità del codice, configurabilità, separazione di interessi e indipendenza dai protocolli.

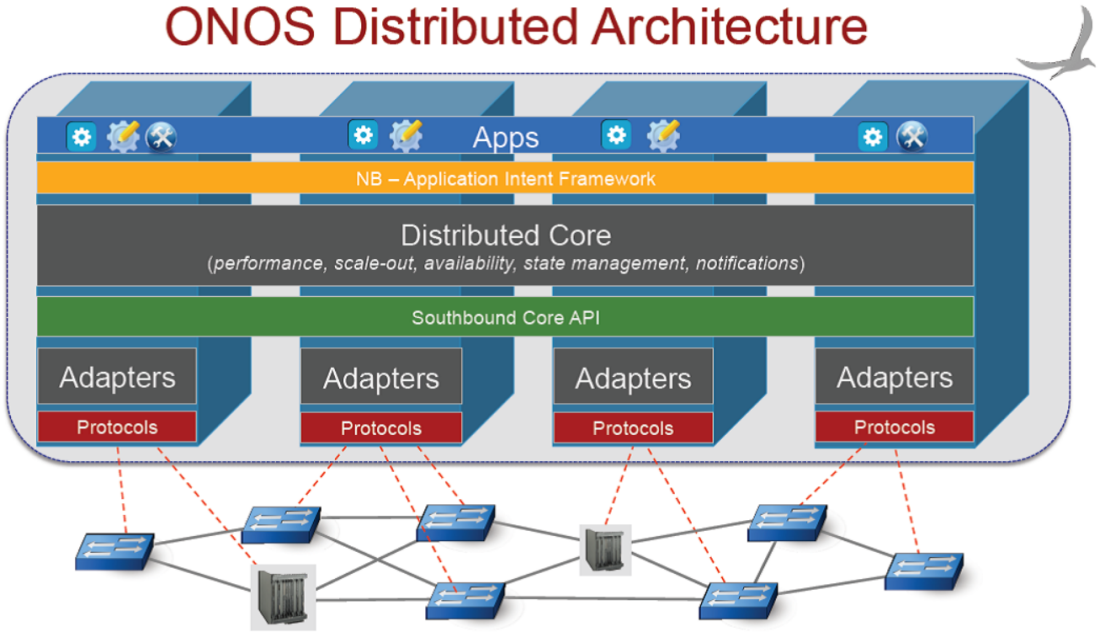


Figura 2.2: Architettura di ONOS [2]

La base dell'architettura di ONOS, come si può vedere dalla Figura 2.2, è costituita da una piattaforma di applicazioni distributite collocata sopra OSGi [36] e Apache Karaf [37]. Queste applicazioni utilizzano Java come linguaggio di programmazione e offrono delle funzionalità di base per il sostegno del livello superiore. Quest'ultimo fornisce i controlli di rete e le astrazioni di configurazione necessarie per il corretto funzionamento del controller.

ONOS supporta quindi un'architettura modulare che permette agli operatori di adattare facilmente la rete alle loro esigenze garantendo flessibilità e reattività alle variazioni delle condizioni operative in modo dinamico.

Per estendere le funzionalità iniziali sono necessarie delle applicazioni ONOS aggiuntive che si integrano con quelle già presenti comportandosi come una loro estensione. Ognuna di esse è gestita da un singolo sottosistema che, all'interno del controller, è rappresentato da un modulo. I moduli attualmente installabili sono più di 100.

ONOS è stato progettato come un sistema distribuito in cui tutti i nodi del cluster sono equivalenti in termini di funzionalità e capacità software. Ogni nodo può quindi svolgere le stesse operazioni e contribuire in maniera simmetrica al funzionamento del sistema, cosicchè in caso di guasto di una componente le altre sono in grado di sostituirla per permettere la continuità del servizio.

Pur essendo fisicamente disaggregato offre comunque una visione logicamente centralizzata al fine di fornire l'accesso di ogni informazione alle applicazioni in maniera uniforme.

ONOS supporta diverse API northbound tra cui:

- **GUI**: offre un'interfaccia grafica per interagire con l'utente
- **REST API**: facilita l'integrazione con sistemi di orchestrazione e altri controller
- **gRPC**: per un'interazione ad alte prestazioni tra applicazioni e altre entità o protocolli della piattaforma

Per quanto riguarda le API southbound supportate fornisce diversi adattatori che rendono il sistema indipendente dai vari protocolli.

Abilitando il Transport Layer Security (TLS) per l'interfaccia SBI e l'Hypertext Transfer Protocol Secure (HTTPS) per l'interfaccia NBI, ONOS garantisce una buona sicurezza

monitorando e bloccando l'accesso non autorizzato alle risorse in fase di esecuzione [38].

ONOS per la gestione degli intenti utilizza il framework Intent Monitor and Reroute (IMR) al fine di monitorarli e ottimizzare l'uso delle risorse in base agli obiettivi definiti dagli utenti. Ciò è reso possibile grazie all'interazione con l'Intent Manager, che si occupa di gestire il ciclo di vita degli intenti, e il Flow Rule Manager per installare le regole di flusso sui dispositivi di rete [39].
Una caratteristica chiave è la possibilità di raccogliere statistiche aggiornate, per il flusso di rete e gli intenti monitorati, consentendo la riconfigurazione dinamica dei dispositivi in caso di animalie e massimizzando l'uso di ciascun collegamento durante la trasmissione dei dati. Queste statistiche possono essere filtrate e rese accessibili agli utenti tramite API come CLI o REST, permettendo loro di monitorare la rete e le politiche richieste [40].
Un intento è considerato dal sistema un sottoinsieme del traffico con una specifica astratta dei requisiti che devono essere soddisfatti. Gli utenti possono quindi definire percorsi ottimali che soddisfino determinati requisiti, come la larghezza di banda o il passaggio attraverso nodi specifici. Un intento può trovare in tre differenti stati: Not Monitored (non monitorato), To Be Monitored (da monitorare) e Monitored (monitorato). Di default, l'intento è in stato Not Monitored. Quando un'applicazione o un utente richiede il monitoraggio, IMR aggiorna lo stato a To Be Monitored o, se l'intento è già attivo, a Monitored, avviando così la fase di tracciamento delle statistiche. Nel caso in cui l'intento non sia ancora installato, IMR lo inserisce in una lista di attesa fino a quando l'evento di tipo INSTALLED non viene ricevuto dall'Intent Manager, attivando il monitoraggio. In caso di eventi di tipo WITHDRAWN, l'intento è stato disattivato [39].
IMR supporta due tipologie di obiettivi: *"Point-to-point"*, che stabiliscono una connessione diretta tra nodi, e *"link collection"*, che si riferiscono a un insieme di collegamenti monitorati per ottimizzare il traffico e la distribuzione delle risorse. Grazie alla combinazione di capacità di monitoraggio e allocazione dinamica delle risorse, l'IMR migliora la capacità di ONOS di ottimizzare il traffico di rete in tempo reale. Consente un controllo granulare sul ciclo di vita degli intenti di rete e facilita la riconfigurazione dinamica in risposta a eventuali anomalie. Questo lo rende un componente chiave per ottenere

l'adattabilità necessaria nelle moderne architetture basate su SDN.

2.3.2 ODL

OpenDaylight[19] è un progetto open source che utilizza protocolli aperti al fine di fornire controlli centralizzati e gestire il monitoraggio della rete.
Fa parte della fondazione LF Networking [41] che si occupa di fornire supporto a progetti open source volti a migliorare la comunicazione e la gestione dei dati su una rete.
ODL è un framework scritto in Java progettato per soddisfare esigenze specifiche dell'utente e offrire alta flessibilità.

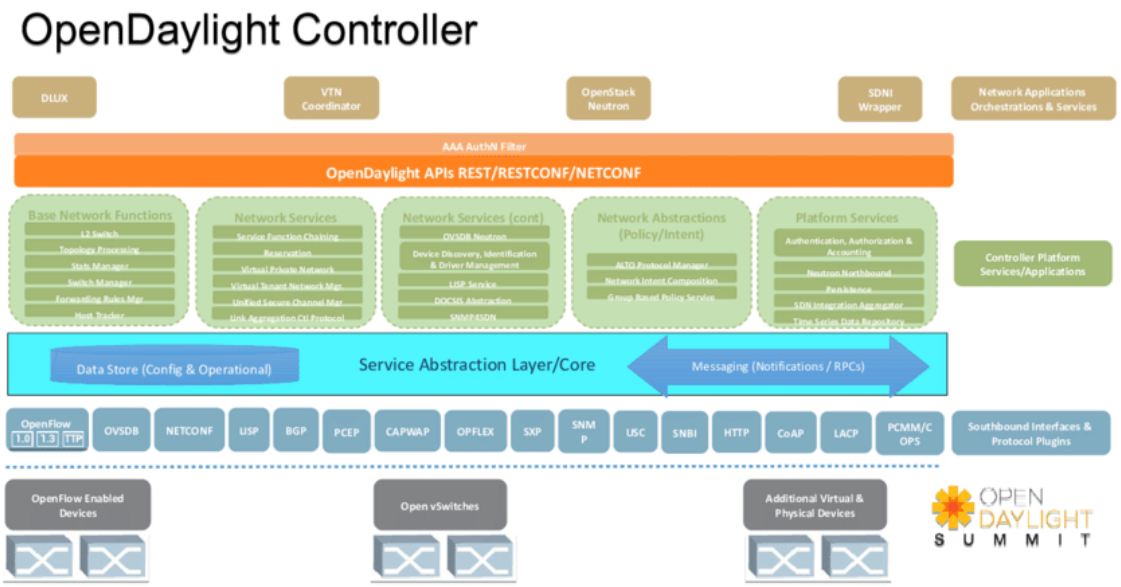


Figura 2.3: Architettura del controller OpenDayLight [3]

L'architettura di OpenDayLight, come mostrato in Figura 2.3, è su più livelli [42]. Il livello principale è costituito dal Controller Platform in quanto al suo interno risiede il controller stesso. Si occupa di gestire il flusso andando a modificare le tabelle di inoltro dei dispositivi fisici o virtuali.
Un aspetto rilevante dell'architettura è la presenza di servizi che l'utente può attivare o disattivare in base alle proprie esigenze. Di default sono tutti disabilitati offrendo un alto

livello di personalizzazione.
I servizi sono implementati come moduli all'interno del controller, e possono essere collegati tra loro per eseguire attività di rete complesse.
Il Service Abstraction Layer (SAL) è il livello inferiore che si occupa del supporto ai vari protocolli SBI, come OpenFlow o NETCONF, offrendo un'interfaccia tra il piano dati e il piano di controllo. All'interno di questo livello, il collegamento dei moduli tra il controller e i dispositivi avviene dinamicamente al fine di soddisfare il servizio richiesto indipendentemente dal protocollo utilizzato.
Per la gestione dei moduli a runtime e l'installazione di nuove funzionalità da implementare ODL utilizza Apache Karaf [37].
Karaf fornisce un ambiente modulare in cui è possibile creare e aggiornare i moduli senza interrompere il funzionamento del controller.
Grazie al framework Model-Driven Service Abstraction Layer (MD-SAL), gli sviluppatori possono implementare nuove funzionalità sotto forma di servizi e protocolli interconnessi.
Il controller espone delle API NBI di supporto tra cui il framework OSGi [36], per gestire le applicazioni che girano all'interno del controller, e REST per la comunicazione con le applicazioni esterne [43].
Per risolvere i problemi legati alla scalabilità, disponibilità e persistenza dei dati, ODL può essere distribuito in più istanze su macchine diverse, le quali cooperano tra loro tramite un meccanismo di clustering. Questo approccio non solo garantisce una gestione efficiente della rete in ambienti complessi, ma offre anche flessibilità nel rispondere a richieste dinamiche di rete.

ODL per gestire gli intenti aveva messo a disposizione il Network Intent Composition (NIC), una NorthBound Interface che è stata abbandonata a partire dalle release successive a Oxygen nel 2018.
Network Intent Composition (NIC) [44] è un'interfaccia progettata per consentire agli utenti di esprimere uno stato desiderato in modo indipendente dall'implementazione sottostante, detto intento. Ciò è reso possibile grazie all'Intent Compilation Engine che ha il compito di convertire e tradurre gli intenti in regole di configurazione concrete per lo

specifico protocollo di controllo, tra cui OpenFLow, SNMP e NETCONF, garantendo che le richieste siano trasformate in comandi eseguibili sui dispositivi di rete sia fisici che virtuali.
Una delle caratteristiche centrali di NIC è l'utilizzo di un linguaggio di programmazione degli intenti che permette di utilizzare una varietà di linguaggi di policy e di programmazione SDN per consentire di descrivere in modo flessibile il comportamento desiderato della rete.
Grazie alla funzione di composizione degli intenti, NIC consente di combinare più richieste di politiche provenienti da varie applicazioni SDN in un insieme coerente di azioni, gestendo i conflitti tra politiche diverse e garantendo coerenza nelle operazioni di rete.
Per la gestione delle politiche NIC impiega diversi database logici di informazioni tra cui: il Network Service Intent DB per le politiche relative ai servizi di rete, l'End Point Intent DB per politiche sugli endpoint, il Network Security Intent DB per le politiche riguardandi la sicurezza della rete. Oltre a questi, altri database descritti in [45], supportano differenti tipologie di politiche, garantendo una gestione completa delle esigenze.
Dal punto di vista dell'interoperabilità NIC è progettato per essere indipendente dal controller e dal protocollo di rete permettendo la portabilità degli intenti tra diverse implementazioni di controller. Gli utenti possono interagire con NIC attraverso l'interfaccia RESTful utilizzando operazioni standard RESTCONF oppure tramite la Karaf console CLI.
Le operazioni REST supportate includono [44]:

- POST: per creare un nuovo intento, specificandone l'ID come attributo.
- GET: per recuperare la lista degli intenti configurati o un intento specifico.
- DELETE: per rimuovere un intento configurato dalla rete.

Attualmente, ODL gestisce gli intenti utilizzando il framework MD-SAL e applicazioni di rete che interagiscono tramite API come REST o NETCONF.
Senza una specifica interfaccia dedicata, gli intenti vengono espressi e gestiti attraverso una combinazione di applicazioni e protocolli che traducono i requisiti dell'utente in configurazioni di rete concrete.

2.4 Kubernetes

Kubernetes [46], noto anche come K8s, è una piattaforma open-source per l'orchestrazione dei container, unità software eseguibili che contengono tutti gli elementi necessari per l'esecuzione in qualsiasi ambiente [47]. Creato originariamente da Google nel 2014, è stato poi donato alla Cloud Native Computing Foundation (CNCF [48]).

E' progettato per automatizzare la gestione, lo scaling e il deployment di applicazioni containerizzate. Quest'ultime vengono eseguite all'interno dei cluster Kubernetes costituiti da

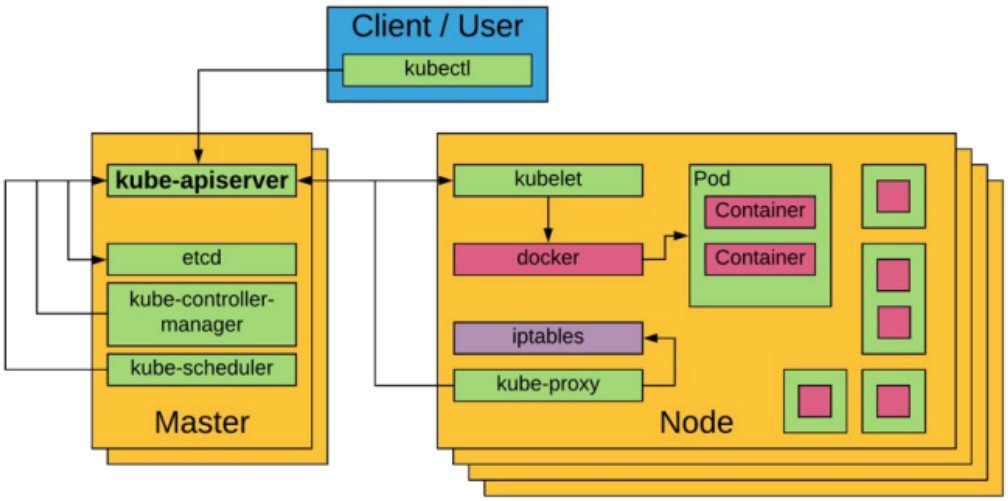


Figura 2.4: Architettura di Kubernetes [4]

macchine fisiche o virtuali chiamate nodi. I cluster comprendono due tipologie principali di macchine: il Master, che gestisce l'intera orchestrazione, e i Nodi, che eseguono i container all'interno di pod che condividono risorse come CPU e memoria. Le risorse dei cluster vengono specificate attraverso file di configurazione, tipicamente utilizzando YAML (Yet Another Markup Language), consentendo agli utenti di specificare deployment, servizi e configurazioni necessarie all'esecuzione delle applicazioni.

Kubernetes segue un'architettura basata sul paradigma client-server (raffigurata in Figura 2.4), con i pod come unità di base, il Master che agisce da server centrale e i Nodi che rappresentano i client. Ogni Nodo esegue due servizi principali: il kubelet, che riceve

e gestisce i comandi per l'esecuzione dei container, e il kube-proxy, che si occupa della configurazione delle regole di rete garantendo che le connessioni siano correttamente instradate verso i pod. Il Master è responsabile del coordinamento del cluster, e ciò avviene tramite alcuni componenti essenziali [4]:

- **etcd**: un database distribuito che memorizza lo stato del cluster.
- **kube-apiserver**: fornisce l'interfaccia di comunicazione tra i componenti interni al sistema e gli utenti esterni.
- **kube-controller-manager**: monitora lo stato delle risorse del cluster e applica le modifiche necessarie.
- **kube-scheduler**: decide su quali nodi eseguire i pod in base alle risorse disponibili.

In Kubernetes è possibile avere più di un Master per migliorare la disponibilità del sistema utilizzando un Master principale e dei nodi replica che garantiscano la continuità del servizio in caso di guasto.
Un aspetto fondamentale di Kubernetes è la caratteristica di self-healing; se un processo si arresta o un pod fallisce, Kubernetes è in grado di riavviarlo automaticamente. Ciò permette alle applicazioni di rimanere sempre in uno stato ottimale. Inoltre, durante gli aggiornamenti, Kubernetes utilizza il meccanismo di rolling update (aggiornamenti in sequenza): i pod vengono aggiornati uno alla volta, garantendo che il numero richiesto rimanga sempre attivo e in esecuzione, senza provocare interruzioni del servizio.
Oltre alle funzionalità di base, Kubernetes offre concreti vantaggi operativi, soprattutto in termini di ottimizzazione delle risorse. Grazie alla capacità di distribuire le applicazioni containerizzate in modo efficiente su un numero limitato di macchine è possibile ridurre i costi legati all'infrastruttura. Questo porta a un utilizzo più vantaggioso delle risorse hardware, limitando il tempo in cui le macchine restano inattive e riducendo il consumo energetico [49]. Anche TeraFlow adotta Kubernetes per orchestrare le sue componenti. Ciò permette a TeraFlow di gestire dinamicamente i suoi servizi e adattarsi rapidamente alle esigenze di rete, garantendo funzionalità come autoriparazione, integrità e bilanciamento del carico [50]

2.5 Miscroservizi

I microservizi rappresentano un nuovo paradigma di progettazione software che si basa sulla scomposizione di applicazioni in una serie di servizi autonomi, chiamati microservizi. Ogni microservizio ha un obiettivo specifico e può essere gestito in modo indipendente dagli altri mantenendo comunque la capacità di interagire con gli altri tramite protocolli standardizzati.

La loro introduzione è stata motivata dalle limitazioni delle architetture monolitiche che sono diventate più evidenti con l'aumento della complessità delle applicazioni. Nelle architetture monolitiche, l'intera applicazione è costituita da un unico blocco di codice in cui tutte le funzionalità, indipendentemente dal loro grado di correlazione, sono strettamente interconnesse. Ciò rende complessa la gestione e l'evoluzione nel tempo [51]. Ad esempio, l'aggiunta di nuove funzionalità o la risoluzione di errori può richiedere modifiche di un'ampia porzione di codice aumentando il rischio di introdurre nuovi errori in altre parti dell'applicazione. Un ulteriore svantaggio è dato dal fatto che il lavoro degli sviluppatori è spesso vincolato a un unico repository e a un linguaggio di programmazione scelto inizialmente, limitando il processo di svulippo e l'autonomia. Questo introduce un rallentamento del ciclo di aggiornamento, insieme con il fatto che anche piccole modifiche richiedono spesso la ricompilazione dell'intero sistema.

Dal punto di vista della scalabilità, le applicazioni monolitiche mostrano ulteriori limiti. Per gestire l'incremento delle richieste l'intero sistema deve essere replicato anche quando l'aumento del carico riguarda solo componenti specifici. Ciò porta a un uso inefficiente delle risorse e a un incremento dei costi di gestione.

L'architettura a microservizi risolve molte di queste problematiche; il sistema è stato scomposto in moduli più piccoli e indipendenti, facilitando la manutenibilità e permettendo una gestione più agile delle risorse.

Gli aspetti chiave si possono riassumere in [51]:

- **Flessibilità**: il sistema può evolvere in modo continuo, facilitando l'adozione di nuove funzionalità.

- **Modularità**: il sistema è composto da servizi indipendenti, ognuno dei quali contribuisce al comportamento generale.
- **Evoluzione**: l'introduzione di nuove funzionalità avviene in modo graduale, riducendo il rischio di errori e semplificando la manutenzione.

In particolare, i microservizi influenzano attributi fondamentali per la qualità del software [51]:

- **Disponibilità**: essendo ogni servizio autonomo, la disponibilità del sistema dipende da quella dei singoli servizi. E' tuttavia necessario mantenere un bilanciamento nella complessità di integrazione che può portare a una riduzione dell'affidabilità del sistema.
- **Affidabilità**: l'affidabilità complessiva è strettamente collegata ai meccanismi di comunicazione tra i servizi che devono essere quindi progettatati con particolare attenzione introducendo interfacce semplici.
- **Manutenibilità**: l'indipendenza dei servizi riduce i costi di modifica e facilità l'aggiunta di nuove funzionalità.
- **Prestazioni**: la comunicazione tra microservizi avviene tramite la rete e questo può comportare un degrado delle prestazioni rispetto alle chiamate interne delle architetture monolitiche. Tuttavia, un sistema ben progettato con contesti definiti e una giusta interconnessione può minimizzare questo impatto.
- **Sicurezza**: l'utilizzo di formati di scambio dati introduce la necessità di meccanismi di crittografia e autentificazione, specialmente quando si integrano servizi di terze parti.
- **Testabilità**: uno dei principali vantaggi dei microservizi è la possibilità di testare ogni componente in isolamento, facilitando la rilevazione degli errori. E' necessaria una particolare attenzione all'integrazione tra più microservizi che può risultare complessa e creare dei anomalie non presenti nei singoli componenti.

Nonostante alcune sfide, i microservizi offrono vantaggi significativi alle aziende e, pur essendo un paradigma relativamente nuovo, è già ampiamente diffuso. Un'indagine del 2021 ha infatti rivelato che il 71% delle imprese adottava almeno parzialmente i microservizi [52].

Inoltre, sistemi monolitici preesistenti possono essere gradualmente trasformati in un'architettura di microservizi. Un esempio noto è quello di Amazon che ha abbandonato un database monolitico a favore una struttura basata sui microservizi. Altre aziende come Netflix, Uber e LinkedIn li hanno adottati riscontrando un miglioramento nei loro tempi di rilascio.

Nel contesto delle applicazioni cloud-native, i microservizi svolgono un ruolo centrale, in particolare si possono orchestrare tramite Kubernetes. In un'architettura basata su microservizi e orchestrata da Kubernetes, come TeraFlow, ogni microservizio è isolato in un cointainer che può avere più istanze, ognuna rappresentata da un pod. Questo consente una gestione delle risorse più efficiente rispetto alle macchine virtuali (VM) tradizionali, dove ogni istanza necessita di un sistema operativo completo, e facilita operazioni di aggiornamento o ripristino [53], oltre a ridurre il consumo di risorse condividendo librerie e componenti.

3. TeraFlow: architettura e gestione degli intenti

Il controller SDN su cui ci focalizzeremo è TeraFlow [54], una piattaforma recentemente proposta, sviluppata da un'ampia comunità open source nell'ambito di un progetto europeo.

Infatti, è stato finanziato dall'unione europea per il programma di ricerca e innovazione Horizon 2020 [55] e supportato dal 5G PPP [56], un'iniziativa congiunta tra la Commissione Europea e l'industria europea delle telecomunicazioni.

Nonostante la quantità di controller SDN i fondatori di TeraFlow hanno riscontrato come problema comune il calo delle contribuzioni ai progetti open source rilevanti negli ultimi anni. Questo declino mette a rischio il continuo sviluppo e il supporto per i nuovi bisogni delle reti moderne lasciando molte soluzioni esistenti inadeguate per affrontare le sfide emergenti.

L'obiettivo di TeraFlow è implementare un controller SDN Carrier Grade (reti o infrastrutture ben testate con livelli estremamente elevati di affidabilità, ridondanza e sicurezza) che soddisfi i requisiti attuali ed eventualmente futuri per le reti, sia architetturali che infrastrutturali, e che sia in grado di gestire miliardi di dispositivi. Il controller mira a migliorare le capacità di elaborazione dei flussi permettendo di gestire un volume di traffico equivalente a un terabit al secondo. Questa capacità è cruciale per supportare le elevate esigenze di connettività delle reti B5G (Beyond Fifth Generation).

Un ulteriore obiettivo è ridurre il divario tra le necessità delle industrie e ciò che offrono gli standard SDN.

Questo controller, attualmente in fase di sviluppo, sarà progettato per integrarsi con gli attuali framework NFV e MEC. Inoltre, si prevede che supporti l'integrazione delle apparecchiature di rete ottica e a microonde e che sarà compatibile con altri controller come ONOS, ma anche con istanze multiple di TeraFlow che gestiscono diversi domini, al fine di sfruttare funzionalità avanzate e facilitare l'interoperabilità con altre reti.

A differenza dei controller presentati precedentemente TeraFlow è stato progettato con un'architettura cloud-native, pensata per sfruttare appieno le caratteristiche degli ambienti

cloud. Questo approccio, basato su microservizi containerizzati, garantisce una maggiore flessibilità e capacità di adattamento rispetto ai sistemi tradizionali.
L'uso di container, gestiti tramite Kubernetes, permette di isolare ogni microservizio utilizzando una minore quantità di risorse rispetto alle macchine virtuali (VM).
TeraFlow infatti mira a ottimizzare l'uso delle risorse di rete per migliorare l'efficienza energetica e ridurre i costi operativi. Ciascun microservizio rappresenta una componente che interagisce attraverso la connessione di rete rendendo il controller disaggregato. Le componenti principali del sistema sono implementate in Java (solo quelle di Automation e Policy) e Python e l'ambiente è sviluppato presso la sede del CTTC a Barcellona.
Alcuni dei requisiti funzionali del controller sono rappresentati da [57]:

- **usabilità**: realizzata grazie a un'interfaccia utente web (web UI) che consente la configurazione di servizi predefiniti e visualizzazione personalizzabile delle metriche.
- **scalabilità**: è intrinseca nel design del controller con la replicazione automatica dei miscroservizi per gestire elevati volumi di richieste in ingresso.
- **affidabilità**: garantita attraverso robusti meccanismi di monitoraggio che supervisionano lo stato dei miscroservizi e dei flussi, attivando automaticamente dei processi di ripristino se necessari.

Dal punto di vista della sicurezza il sistema utilizza un approccio basato sul Machine Learning per la prevenzione e la mitigazione degli attacchi [58].
Questo progetto riveste un ruolo chiave nel panorama delle tecnologie 5G, contribuendo a unire diverse università e istituti di ricerca per sviluppare soluzioni all'avanguardia lavorando con organismi di standardizzazione per garantire l'adozione su scala globale.

3.1 Componenti architetturali

Le componenti di TeraFlow sono classificate in due categorie; le componenti principali del sistema operativo di rete e le netapp sovrapposte [59]. Le componenti del sistema operativo di rete (Network Operating System - NOS) formano la base dell'infrastruttura

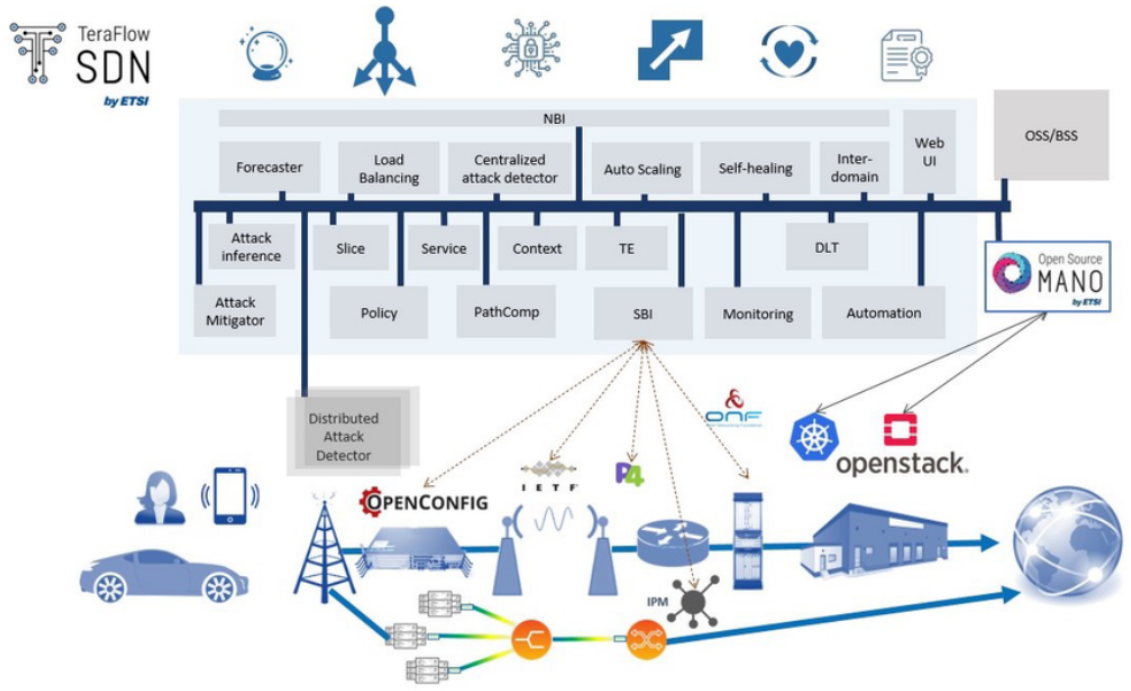


Figura 3.1: Architettura di TeraFlow [5]

di TeraFlow, illustrata in Figura 3.1, offrendo servizi di connettività per infrastrutture di rete programmabili avanzate [59] basate su tecnologie come P4 (Programming Protocol-independent Packet Processors), OpenConfig e TAPI (Transport API), che forniscono modalità flessibili per gestire il traffico e le configurazioni di rete. Una delle componenti principali per questa gestione è quella di Context. Ha il compito di memorizzare topologie, dispositivi, collegamenti e servizi in un database No-SQL[60]. Questa componente garantisce la coerenza dei dati gestiti, controllando gli accessi concorrenti dei vari componenti del controller SDN [61], grazie a una Database API.

Le netapp sono applicazioni che operano sopra il livello del NOS, sfruttando le interfacce di comunicazione, indipendentemente dal linguaggio di programmazione, per interagire con le infrastrutture di rete. Queste applicazioni possono includere strumenti per il monitoraggio, la gestione della qualità del servizio, e altre funzionalità.

TeraFlow fornisce, inoltre, soluzioni di automazione SDN e IBN basate su policy. Nello specifico, TeraFlow Automation sfrutta importanti eventi di sistema per realizzare la

(ri)configurazione di servizi e dispositivi in modo zero-touch, ossia minimizza l'intervento umano attraverso l'automazione. Riducendo la necessità di interventi manuali da parte degli operatori vengono ridotte significativamente anche le spese operative [59].
Alcune componenti di TeraFlow sono suddivise in livelli di astrazione per facilitare la gestione della rete:

- Device Level Abstraction: fornisce un'astrazione dei dispositivi fisici presenti, consentendo al controller di interagire con diverse tipologie di hardware.
- Service Level Abstraction: focalizzato sulla gestione e configurazione dei servizi, questo livello gestisce le interazioni con l'SBI (Southbound Interface) per fornire un'interfaccia unificata.
- Management Level Abstraction: realizza la gestione complessiva della rete, che include il monitoraggio, il controllo e la manutenzione dei servizi e dei dispositivi gestiti tramite SBI, garantendo un controllo centralizzato dell'infrastruttura.

Nelle sezioni successive saranno descritte le componenti maggiormente coinvolte nella gestione degli intenti. La definizione delle rimanenti si può trovare nella documentazione ufficiale [6].

3.2 Gestione degli intenti

La gestione degli intenti all'interno di un sistema SDN è un processo complesso che si avvale di diverse componenti fondamentali. Nella seguente sezione verranno esaminati nel dettaglio i principali moduli responsabili di questa gestione evidenziando le modalità di comunicazione tra loro. Per garantire un'analisi più chiara l'approfondimento è stato suddiviso in vari livelli di astrazione, partendo dalla gestione dei dispositivi fino alla descrizione delle politiche finali.

3.2.1 Device Level Abstraction

Il Device Level Abstraction permette l'interazione con i dispositivi presenti all'interno della rete.

Una componente fondamentale per questo livello di astrazione è la componente SBI (Southbound Interface), che permette al controller di gestire diversi tipi di dispositivi, garantendo eterogeneità attraverso il supporto a vari protocolli. Il ruolo principale dell'SBI è stabilire e mantenere la connessione con i dispositivi, consentendo così la loro integrazione nell'ecosistema del controller [6]. Dopo aver stabilito la connessione con un determinato dispositivo e aver verificato la disponibilità, la componente mette a disposizione una API che consente di inviare dinamicamente le configurazioni dal controller al dispositivo in formato JSON. Queste configurazioni permettono di aggiornare i parametri operativi del dispositivo in base alle politiche di rete definite.

Un'altra componente che fa parte di questo livello è quella di Monitoring il cui compito è gestire le diverse metriche configurate per i dispositivi e i servizi di rete. A tale scopo memorizza i dati relativi alle Key Performance Indicators (KPI) selezionate e persiste le informazioni all'interno di un database (Metrics Database). Questo permette di fornire dati dimensionali con serie temporali visualizzabili su Grafana [62]. Per il corretto funzionamento la componente deve essere in grado di recuperare le metriche da tutti i diversi dispositivi monitorati che, spesso, implementano protocolli diversi. Per questo motivo sono stati inclusi una serie di sottomoduli che si connettono agli elementi monitorati utilizzando i protocolli SBI necessari.

Quando un valore KPI registrato supera determinate soglie predefinite, la componente di Monitoring invia una notifica dell'evento alla componente che ha originariamente richiesto il monitoraggio della metrica all'interno del controller.

Ogni evento è composto da una KPI che identifica la regola a cui si riferisce, un timestamp e un KPIValue che rappresenta il valore della specifica metrica monitorata in tempo reale dalla KPI al momento indicato dal timestamp. Supponendo che la componente di Monitoring stia monitorando la latenza di un collegamento di rete tramite una KPI denominata "Latency" e che la latenza massima accettabile sia 100ms, se la latenza supera questa soglia, ad esempio 120ms, si genera un evento così composto:

- KPI: "Latency"
- Timestamp: "06-09-2024 14:35:20" (la data e l'ora a cui si è verificato l'evento)

- KPIValue: 120 ms

che sarà poi gestito dai livelli sovrastanti. Questa funzione, e quindi la componente stessa, è di supporto al Management-level per garantire l'automazione dei servizi. Quanto si tratta di gestire topologie più complesse che coinvolgono molti dispositivi è necessario un livello di astrazione superiore per specificare la connessione tra vari end-points senza la necessità di configurare singolarmente ogni dispositivo. A tal fine si introduce il Service Level.

3.2.2 Service Level Abstraction

Il Service Level è responsabile della creazione e dell'aggiornamento dei servizi di rete. Questo livello di astrazione permette agli utenti di definire astrazioni specifiche per la connessione tra gli end-points attraverso la componente di Service.
Per svolgere questo compito, la componente si avvale di diverse funzionalità offerte dalle altre parti del sistema. In particolare, per il calcolo dei percorsi di rete, si affida alla componente PathComp (Path Computation). Ad esempio, se un intento richiede la creazione di una connessione con una latenza minima tra due nodi specifici, la componente invia una richiesta alla PathComp, che calcola il percorso ottimale in vase ai valori della latenza. Una volta ricevuta la risposta, la componente di Service utilizza uno scheduler per configurare i dispositivi di rete lungo il percorso selezionato, utilizzando le connessioni restituite [6]. Le specifiche configurazioni vengono propagate all'interfaccia Southbound (SBI) attraverso dei file JSON, consentendo un'automatizzazione della rete e permettendo di astrarre la complessità del livello sottostante all'utente.
La componente di Service supporta diversi tipi di servizi ed è in grado di utilizzare vari protocolli. Implementa a tale scopo una Service Handler API che consente agli operatori di rete di definire i comportamenti necessari per ciascun tipo di servizio [6].

- **L2-VPN**: servizio per dispositivi OpenConfig
- **L3-VPN**: servizio per dispositivi emulati o OpenConfig con supporto per ACLs
- **Connectivity**: servizio per dispositivi TAPI

- **L2 service Handler**: servizio per dispositivi P4
- **Microwave service Handler**

Per la sperimentazione è stato usato il servizio *L2 Service Handler* sviluppato per topologie basate su P4. Questo utilizza lo stesso servizio di *L2-VPN* ma usa il dispositivo per tradurre le richieste ci connessione in regole di forwarding per la pipeline. Questa interfaccia implementa solamente due funzioni; *SetEndpoint* per creare o aggiornare un insieme di dispositivi utilizzati dal servizio, e *DeleteEndpoint* per eliminare le regole di configurazione quando il servizio non è più attivo [6]. Un'altra componente, già introdotta precedentemente, che fa parte di questo livello è la PathComp. Si occupa di gestire la selezione del percorso tra gli end-points per i servizi di connettività di rete. Riceve richieste dalla componente di Service e, interagendo con la componente di Context, recupera le informazioni sulle topologie sottostanti al fine di creare percorsi che soddisfino i requisiti richiesti.
La PathComp rappresenta un'entità singola e specializzata dove possono essere ospitati diversi algoritmi. Questo permette che qualsiasi nuovo algoritmo utilizzato non impatti su altre componenti del controller. Per confrontare i percorsi viene utilizzato un algoritmo K-SP dove i k percorsi sono ordinati per numero di passi (hop), ritardo end-to-end e larghezza di banda disponibile sul link più congestionato [63].
Il service layer permette quindi al controller di tradurre gli intenti in regole concrete che vengono poi propagate al livello sottostante. Tuttavia, da solo non ha la capacità di rispondere dinamicamente agli eventi che si verificano nella rete in tempo reale, come il cambiamento dello stato di un collegamento o di una risorsa di rete. Per essere in grado di creare, aggiornare o cancellare i servizi di rete in base a tali eventi è necessario introdurre un ulteriore livello di astrazione che automatizzi queste operazioni: il Management Level Abstraction.

3.2.3 Management Level Abstraction

Il Management Level Abstraction è stato introdotto per consentire l'interazione dinamica con la componente di Service permettendo l'automazione di un servizio in risposta agli

eventi provenienti dalla rete. In altre parole, il sistema non si limita a configurare i dispositivi solo su input manuali o su richiesta esplicita, ma è in grado di adattarsi dinamicamente ai cambiamenti in tempo reale. Questo livello consente dunque al controller di reagire rapidamente a eventi come congestioni, guasti o modifiche nella topologia, garantendo che la rete mantenga un alto livello di efficienza e l'intento specificato come servizio continuo da essere realizzato nella rete.

Una componente fondamentale è la componente di Policy che interagisce strettamente con la componente di Monitoring descritta precedentemente.

Si occupa di definire condizioni di politica, associate a un determinato servizio, che possono essere applicate sia a livello di singoli dispositivi che a livello di dominio di rete. Le politiche possono includere più regole collegate tra loro tramite condizioni logiche di AND/OR [6], ognuna delle quali genera una KPI specifica da monitorare.

Ogni regola è composta, oltre che dalla KPI che la identifica, da un operatore numerico di confronto (maggiore, minore o uguale), e da un KPIValue. Quest'ultimo, in combinazione con l'operatore numerico, definisce l'intervallo di valori accettabili o non accettabili per quella specifica metrica.

Se le regole definite nella politica vengono soddisfatte la componente di Monitoring genera un allarme che viene inviato, in questo caso, alla componente di Policy, che reagisce attivando le azioni predefinite. Ad esempio, se si verifica un eccesso di latenza, la componente di Policy può attivare il ricalcolo del percorso, o altre azioni correttive, tramite la componente di Service consentendo di ripristinare lo stato desiderato.

La componente Policy utilizza il Context Database per identificare quali dispositivi o servizi sono coinvolti nella politica. Se è stato fornito l'ID del servizio, la componente recupera i dispositivi associati a esso, in caso contrario scansiona una lista di dispositivi per individuare quelli che devono rispettare le regole impostate.

Una regola di politica può avere vari stati che possono essere rappresentati tramite la macchina a stati in Figura 3.2:

- **inserted** (inserita)
- **validated** (convalidata)

- **provisioned** (provvista)
- **actively enforced** (attivamente applicata)
- **failed** (fallita)
- **updated** (aggiornata)
- **removed** (rimossa)

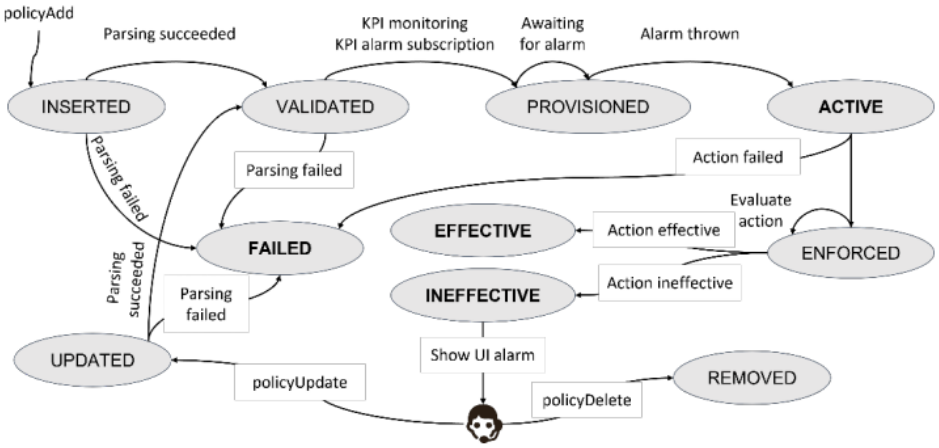


Figura 3.2: Macchina a stati interna alla componente di policy [6]

Grazie all'integrazione tra le componenti di Policy e di Monitoring, TeraFlow ha reso possibile l'associazione delle condizioni di politica con gli allarmi generati dal sistema. In questo modo può implementare una gestione dinamica della rete permettendo di mantenere gli SLA (Service Level Agreements) senza necessitare di interventi manuali continui da parte degli operatori riducendo anche la complessità operativa. In ambienti su larga scala, come le reti distribuite composte da microservizi, questa capacità di adattamento consente anche di prevenire potenziali criticità, ottimizzando continuamente le risorse di rete.

3.3 gRPC

Per garantire l'interoperabilità tra le diverse componenti, TeraFlow utilizza un bus gRPC (Google Remote Procedure Call [64]) come protocollo interno, un framework che facilita

la comunicazione tra servizi in modo efficiente e preciso.

Google Remote Procedure Calls (gRPC[64]) è un framework OpenSource sviluppato da Google che si ispira al paradigma RPC (Remote Procedure Call). Permette di connettere, invocare, operare e fare debug di programmi eterogenei in modo semplice.

E' basato sul protocollo di trasporto HTTP/2 che supporta la comunicazione bidirezionale e il multiplexing delle connessioni.

gRPC include supporto per il bilanciamento del carico, il tracciamento, il controllo dello stato e l'autentificazione[65] [66], inoltre consente di definire i servizi, i loro metodi di comunicazione e trasportare messaggi attraverso dei file di descrizione dell'interfaccia detti Protocol Buffer, o più semplicemente file proto.

I file proto sono un meccanismo indipendente dal linguaggio e dalla piattaforma e permettono di serializzare le strutture dati in modo più efficiente rispetto a formati come JSON o XML, sia in termini di dimensione dei messaggi che di velocità di elaborazione. In Figura

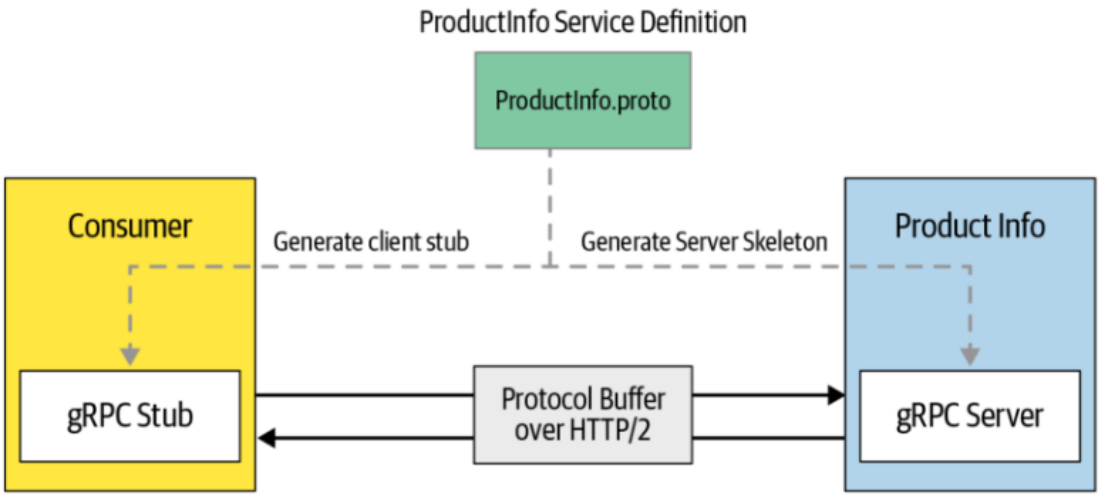


Figura 3.3: Funzionamento del protocollo gRPC [7]

3.3 viene illustrata l'interazione tra un microservizio e un client tramite gRPC.

Per sviluppare un'applicazione gRPC è necessario definire un'interfaccia dei servizi all'interno di un file proto (in Figura 3.3 ProductInfo.proto). Anche se a prima vista può sembrare un file di testo ordinario, come si può notare in Figura 3.4, in realtà specifica i metodi con i parametri di input e i valori di ritorno.

```
service ForecasterService {
  rpc GetForecastOfTopology (context.TopologyId) returns (Forecast) {}
  rpc GetForecastOfLink(context.LinkId) returns (Forecast) {}
  rpc CheckService (context.ServiceId) returns (ForecastPrediction) {}
}

message SingleForecast {
  context.Timestamp timestamp= 1;
  double value = 2;
}

message Forecast {
    oneof uuid {
      context.TopologyId topologyId= 1;
      context.LinkId linkId = 2;
    }
    repeated SingleForecast forecast = 3;
}

enum AvailabilityPredictionEnum {
    FORECASTED_AVAILABILITY = 0;
    FORECASTED_UNAVAILABILITY = 1;
}
```

Figura 3.4: Esempio di Protocol Buffer file in TeraFlow [8]

Una volta definita l'interfaccia è possibile generare il codice per il server (server skeleton) e per il client (client stub) nel linguaggio desiderato usando il compilatore Protobuf *protoc*[7]. Ciò permette al client di invocare i metodi definiti da remoto.

Quando il client richiede un servizio gRPC, utilizza i protocol buffer per serializzare la chiamata in un formato compatibile con il protocollo. Successivamente, la richiesta è stata inviata tramite HTTP/2 e, una volta ricevuta dal server, viene deserializzata, con il supporto dei Protocol Buffers, per invocare il metodo appropriato. La risposta segue il flusso inverso da server a client[7].

Il framework gRPC semplifica quindi la gestione di aspetti complessi come vincoli di servizio, serializzazione dei dati, comunicazioni di rete e autenticazione. E' progettato per trasportare messaggi peer-to-peer in modo distribuito e non durevole, consentendo a più servizi di scambiarsi informazioni attraverso un bus condiviso. Questo lo rende particolarmente adatto per architetture basate su microservizi dove scalabilità e prestazioni sono

fondamentali. Teraflow usa gRPC per la comunicazione tra le componenti e i Protocol Buffers utilizzati sono descritti alla pagina del sito [8].

Negli esperimenti svolti in [67] si è dimostrato che gRPC, grazie ai file proto, in scenari che coinvolgono ambienti più complessi con molte componenti (come server, DNS, firewall..) può portare a una riduzione di quasi il 27% nel tempo di creazione del server rispetto a REST (Representational State Transfer), uno tra i più moderni e utilizzati sistemi di trasmissione dati, riducendo il tempo di esecuzione complessivo. Infine, gRPC supporta nativamente TLS (Transport Layer Security), che garantisce la sicurezza delle comunicazioni cifrando i messaggi scambiati tra servizi. Questo aspetto è cruciale per assicurare la protezione dei dati in sistemi distribuiti e critici.

4. Studio e sperimentazione della gestione di policy in Teraflow

In questo Capitolo verranno esposti i vari esperimenti eseguiti utilizzando il controller SDN TeraFlow per quanto riguarda la gestione degli intenti. Nella fase iniziale è stato replicato l'esperimento seguito nell'Hackfest 3 [68] organizzato da ETSI presso la sede del CTTC a Barcellona. L'Hackfest è un evento con scopi formativi e di sperimentazione in cui viene fornito un ambiente di test pre-configurato, in questa edizione per il controller TeraFlow, con l'obiettivo di eseguire esperimenti e dimostrazioni pratiche delle sue funzionalità. Il lavoro è iniziato con l'installazione della macchina virtuale [69] su VirtualBox 7.0.
Di seguito vengono riportate le specifiche della VM utilizzata:

- IP Address: 10.0.2.X/24 (DHCP)
- Gateway: 10.0.2.1
- DNS: 8.8.8.8, 8.8.4.4
- Requisiti minimi: 4 vCPU, 8 GB di RAM, 60 GB di spazio sul disco, Virtual Disk Image (VDI)
- Connessione di rete: NAT Network con porte esposte 22 per SSH e 80 per HTTP
- Sistema operativo Ubuntu Server 22.04.4 LST (compatibile anche con la versione successiva 22.04.6 LTS), senza interfaccia grafica per ridurre il consumo di risorse.

Per configurare l'ambiente necessario viene installato Docker CE (Community Edition) insieme al plugin Docker BuildX. Succcessivamente viene installato MicroK8 con Kubernetes v1.24.17 assicurandosi di disabilitare il firewall ufw e abilitare tutti i plugin necessari. A questo punto viene installata la versione di TeraFlow 3.0 in una nuova cartella tfs-ctrl. Successivamente si eseguono i comandi *source my_deploy.sh* e *./deploy/all.sh* per

distribuire il controller su MicroK8. TeraFlow ha preinstallato al suo interno Mininet containerizzato. Per avere a disposizione più topologie e poter riuscire a connetterle correttamente alle componenti del controller SDN è stata integrata la cartella ngsdn-tutorial [70]. Al suo interno contiene esercizi pratici e topologie per l'utilizzo di P4 e ONOS, e in questo caso è stata adattata per funzionare con TeraFlow, con l'obiettivo di agevolare le sperimentazioni e testare le capacità del controller SDN. Infine viene configurata la versione di Python 3.9.18.
La scelta di configurare la VM è il frutto di varie prove, tra cui l'installazione della macchina virtuale creata appositamente per l'Hackfest 3 su VirtualBox 6.1 (con 6 GB di RAM e 50 GB di spazio sul disco) con la versione di TeraFlow 2.1 con degli adattamenti specifici per i test. In questa prima VM è stata riscontrata un'incompleta implementazione della componente di Policy, che non era in grado di riconoscere le KPI e di conseguenza di gestire le politiche di rete richieste.
I vari esperimenti si sono svolti seguendo il seguente schema: attraverso il Service Level è stato introdotto un servizio nella rete sotto forma di intento per stabilire la connessione, e quindi il percorso, tra due endpoint creando delle KPI specifiche per latenza, packet loss e capacità. Successivamente, tramite il Management Level, è stata inserita una politica basata sugli eventi che ha consentito di associare un Service Level Agreement (SLA) al servizio, specificando le condizioni da monitorate e i requisiti che da rispettare durante l'esecuzione.

4.1 Strumenti per la sperimentazione

Prima di passare alla sperimentazione verranno descritti due strumenti che sono stati fondamentali per questa fase. L'uso di Mininet è stato necessario per emulare una rete reale costituita da switch P4. Questi hanno permesso, attraverso la loro programmazione, di portare a termine i vari esperimenti sui servizi e sulle politiche.

4.1.1 Mininet

Mininet [71] è un sistema open source per l'emulazione di reti, su un unico ambiente Linux, che permette di emulare un'intera rete su un singolo computer, come rappresentato in Figura 4.1. E' ampiamente utilizzato in ambiti di ricerca e sviluppo per creare reti virtuali

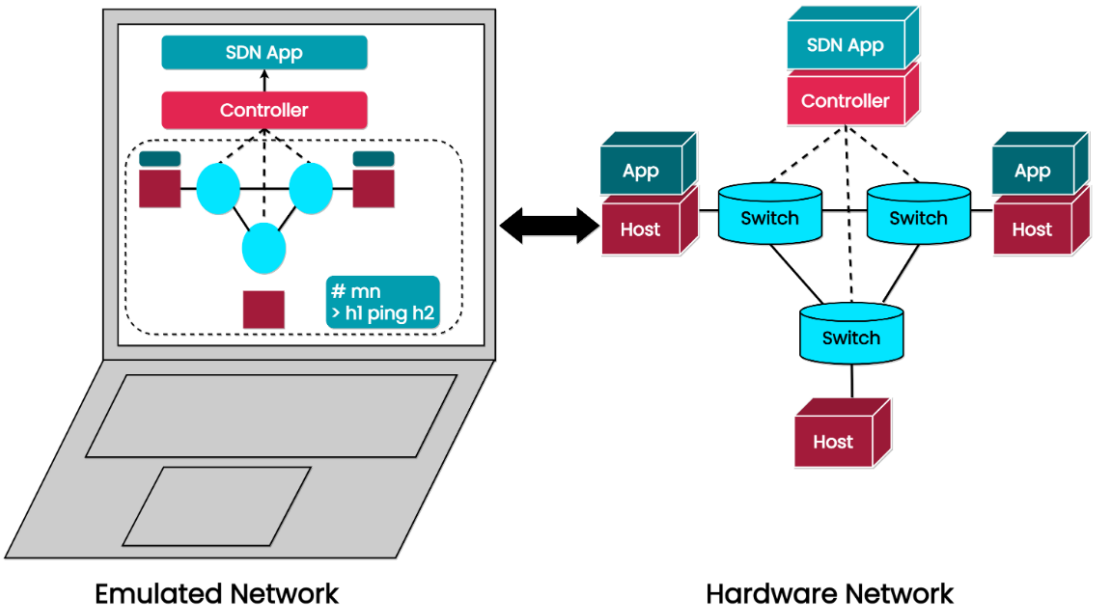


Figura 4.1: Rete mininet

in modo realistico e testare nuove applicazioni, o protocolli, in un ambiente controllato prima di implementarli su reti reali.

Mininet, a differenza di altri emulatori che utilizzano macchine virtuali per ogni dispositivo, è in grado di gestire un insieme di terminali di rete utilizzando la virtualizzazione leggera e consentendo di avviare numerosi host e switch (fino a 4096) [72]. Questo è reso possibile da tecnologie implementate nel kernel, come i network namespaces, che permettono di creare istanze separate di interfacce di rete, tabelle di routing e tabelle ARP che operano in modo indipendente [73].

Mininet offre delle API e un interprete Python, o un interfaccia a riga di comando (CLI), che consentono di definire e gestire facilmente le topologie di rete. In entrambi i casi, si possono scegliere topologie predefinite o personalizzarne di nuove aggiungendo e rimuo-

vendo switch, router, host, controller e link. Mininet consente anche di configurare regole per l'inoltro dei pacchetti per testare diverse funzionalità, come NAT (Network Address Translation) o ECMP (Equal-Cost Multi-Path), facilitando la replica degli scenari di test in ambienti con caratteristiche differenti.
Mininet mette a disposizione tre livelli differenti di API [74]:

- **Low-level**: consiste nelle classi dei nodi e dei link istanziati individualmente e usati per creare una rete.
- **Mid-level**: aggiunge l'oggetto Mininet, un contenitore di nodi e link e fornisce metodi per la configurazione di rete.
- **High-level**: aggiunge l'astrazione della topologia di rete, la classe Topo. Offre la possibilità di creare modelli di topologia riusabili passandoli al comando mn da linea di comando.

Si possono anche impostare i link come up o down e inserire metriche specifiche come quelle di banda, di ritardo, di perdita o di massima lunghezza della coda di recezione per rendere la rete più realistica e adatta a esperimenti.
Mininet è stato progettato per essere facilmente integrabile con altri software e strumenti di rete, infatti anche se è fornito un controller di default, consente di connetterne uno remoto agli switch, indipendentemente dal PC su cui è installato. Questa funzionalià ci ha permesso di poter effettuare la simulazione.

Alcuni comandi fondamentali

Di seguito sono stati elencati i comandi fondamentali di Mininet usati per la sperimentazione tramite linea di comando. La documentazione completa si può trovare [75].
Innanzitutto è fondamentale creare una topologia di rete con il seguente comando[75]:
$ sudo mn
Di default è stata inizializzata la topologia minimale (–topo=minimal) che consiste in uno switch connesso a due host e un controller OpenFlow. All'interno di Mininet si possono trovare altre topologie disponibili e visualizzabili con il comando

$sudo mn -h

che si possono specificare tramite l'opzione $--topo$.

Per avviare la topologia esistono diverse opzioni da poter applicare.Ad esempio, l'opzione $--controller$ seguito dall'indirizzo IP specifica il controller al quale gli switch dovranno collegarsi al posto del predefinito offerto da Mininet.

Una volta creata la topologia per avere informazioni su di essa esistono diversi comandi:

- *nodes*: per visualizzare i nodi presenti.
- *net*: per visualizzare i nodi e i link presenti.
- *dump*: per visualizzare tutte le informazioni di dump dei nodi.
- *h1 ifconfig*: per visualizzare le interfacce del nodo h1.

Alcuni comandi per interagire con la rete e fare dei test minimali sono:

- *h1 ping -c 1 h2* : verifica il corretto funzionamento del percorso tra h1 e h2.
- *pingall*: esegue il ping tra tutti gli host connessi alla rete.
- *iperf*: esegue un test di banda tra 2 degli host della rete.
- *exit*: esce dalla rete.

Per manipolare le metriche relative ai link invece vengono messi a disposizione i seguenti comandi:

- *link s1 h1 down*: disabilita un link, in questo caso quello tra lo switch s1 e l'host h1.
- *link s1 h1 up*: attiva un link, in questo caso quello tra lo switch s1 e l'host h1.
- *s2 tc qdisc add dev s2-eth2 root netem loss 50%* : aggiunge una packet loss del 50% sulla porta eth2 dello switch s2.
- *s2 tc qdisc add dev s2-eth2 root netem delay 200ms*: aggiunge un ritardo di 200ms sulla porta eth2 dello switch s2.

- *s2 tc qdisc del dev s2-eth2 root netem loss 50%* : elimina una packet loss del 50% sulla porta eth2 dello switch s2.
- *s2 tc qdisc del dev s2-eth2 root netem delay 200ms*: elimina un ritardo di 200ms sulla porta eth2 dello switch s2.

API Python

Le API Python di Mininet permettono di creare e gestire topologie di rete in modo più flessibile e programmabile. Di seguito esponiamo alcune classi e comandi della Mid-level API utilizzate negli script:

- *Mininet*: classe per creare e gestire la rete. Il costruttore prende in input diversi parametri la topologia, gli host, gli switch,i controller, i link e ritorna un oggetto di rete.
- *addSwitch()*: aggiunge uno switch alla topologia.
- *addHost()*: aggiunge un host alla topologia.
- *addLink()*: aggiunge un link alla topologia. Si possono specificare paramentri come la banda espressa in Mbit (bw=10), il ritardo (delay='5ms'), massima dimensione della coda espressa in numero di pacchetti (max_queue_size=1000), la loss espressa in percentuale (loss=10)
- *start*: avvia la rete
- *stop*: esce dalla rete
- *pingall*: esegue il ping tra tutti gli host connessi alla rete
- *h1.cmd('comando da eseguire')*: esegue un comando su h1 da CLI e prende l'output

Con le API in Python si può anche estendere il comando *mn* usando l'opzione *–custom* per invocare la topologia ricreata nello script.
sudo mn –your_script.py –topo your_topo

4.1.2 P4

Programming Protocol-indipendent Packet Processor (P4 [76]), è un linguaggio di programmazione flessibile che permette di descrivere il comportamento degli elementi di rete, consentendo di personalizzare come i dispositivi elaborano i pacchetti. P4 è stato introdotto per superare le limitazioni di OpenFlow al fine di fornire una soluzione più flessibile.
OpenFlow, pur separando il piano di controllo e il piano dati, si basa su regole di elaborazione dei pacchetti attraverso tabelle che mappano i campi degli header (indirizzi IP, MAC, porte...) in un insieme fisso di funzionalità.
Negli anni le specifiche dei pacchetti sono diventate sempre più complesse, nonchè dipendenti dalle singole aziende che hanno iniziato a sviluppare l'hardware indipendentemente, rendendo necessari continui aggiornamenti per supportare le varie esigenze[77]. D'altra parte, gli switch non aggiornati, o prodotti da aziende diverse, non riescono a supportare tutte le nuove caratteristiche a causa delle limitazioni hardware.
Inoltre, OpenFLow non fornisce delle interfacce operative o amministrative standard, quindi rende complicato aggiungere supporto per nuovi protocolli. Tutte queste problematiche hanno portato all'introduzione di dispositivi programmabili che utilizzano P4.
I principali obiettivi sono:

- **Indipendenza dal protocollo**: P4 non è vincolato a nessun protocollo specifico consentendo anche di definirne di nuovi o modificare quelli esistenti
- **Indipendenza dal target**: Il codice può essere compilato per funzionare su diversi dispositivi, sia hardware che software, rendendolo versatile
- **Riprogrammabilità**: Il comportamento del piano dati può essere aggiornato dinamicamente consentendo di rispondere rapidamente ai cambiamenti delle esigenze di rete.

P4 consente di definire intestazioni e tabelle personalizzate, e programmare esplicitamente il flusso di controllo dello switch, permettendo di adattarsi rapidamente ai cambiamenti e alle innovazioni.
Nella Figura 4.2 è illustrato il Workflow del modello di P4.

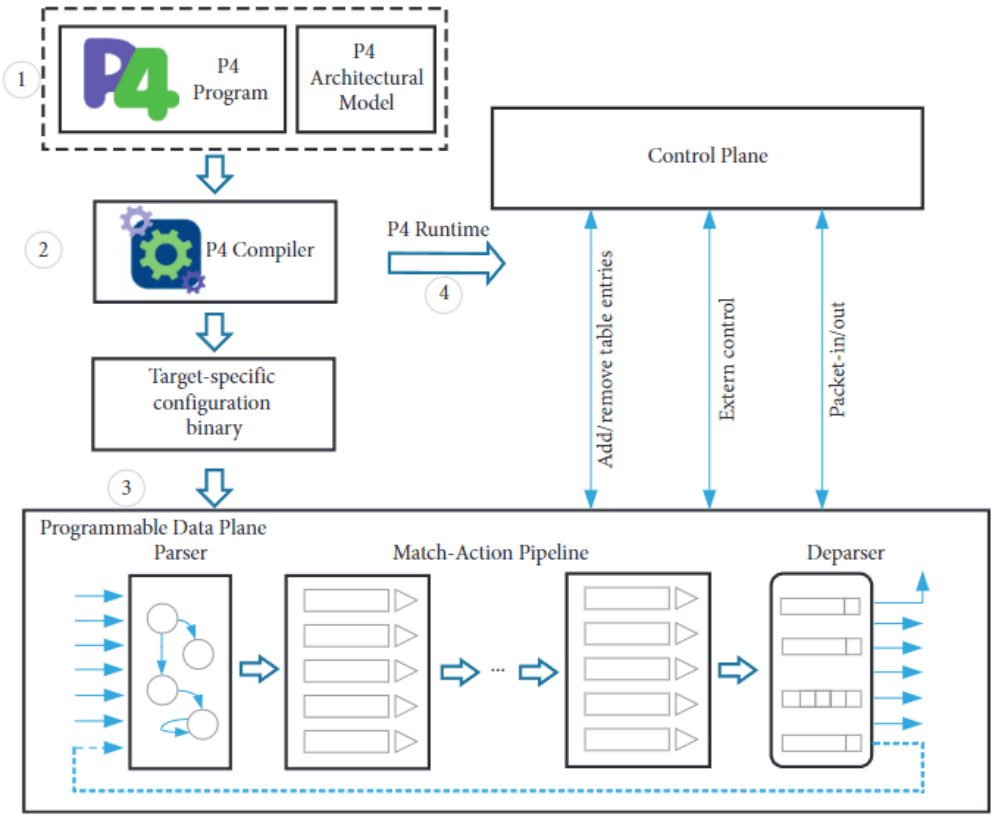


Figura 4.2: Workflow di P4 sul piano dati [9]

Un programma P4 (P4 Program) definisce il comportamento del piano dati desiderato ed è suddiviso in varie sezioni, ognuna delle quali descrive un aspetto specifico del trattamento dei pacchetti. Il processo inizia con la dichiarazione degli header del pacchetto da analizzare, successivamente si definisce il comportamento del parser, le tabelle, le azioni e il flusso di controllo.

Il programma è stato poi inviato al compilatore P4 (P4 Compiler) che genera due tipi di output. Il primo è un file eseguibile che descrive le varie operazioni da implementare all'interno del dispositivo target. I target P4 sono dispositivi programmabili ottenibili tramite hardware, come ASICs o FPGA, o software, che realizzano il comportamento desiderato mettendo in atto le specifiche descritte nel programma grazie al file eseguibile. Il parser all'interno dei dispositivi specifica come estrarre e interpretare i vari header dei pacchetti seguendo uno schema prestabilito.

La pipeline di elaborazione (Match-Action Pipeline) include tabelle e azioni che determinano come processare i pacchetti. La struttura della pipeline è distinta per ciascun dispositivo ed è descritta da un determinato modello di architettura. Il flusso di controllo invece coordona parser e pipeline per garantire il corretto funzionamento dell'intero processo. Infine, il deparser ricompone i pacchetti con le eventuali modifiche degli header e successivamente li reintroduce nella rete.

Il secondo file generato dal compilatore è indipendente dal target e contiene le informazioni necessarie per far comunicare il piano di controllo e il piano dati tramite l'API P4Runtime.

P4Runtime permette al controller di connettersi ai dispositivi e interagire con la pipeline per poter inviare le configurazioni nella relativa tabella [9]. I dettagli hardware del piano dati sono nascosti al piano di controllo rendendolo indipendente dalle funzionalità e dai protocolli supportati. P4 rappresenta un passo significativo verso reti più flessibili e programmabili, consentendo agli sviluppatori di adattarsi rapidamente ai cambiamenti dei requisiti di rete. Si propone come una soluzione innovativa e versatile per superare le limitazioni degli attuali protocolli e dispositivi di rete fornendo un linguaggio dinamico e indipendente dall'hardware.

4.2 Flusso di lavoro dell'esperimento

L'ambiente di sperimentazione utilizzato è costituito da due componenti principali eseguite all'interno della macchina virtuale: il controller e la rete emulata P4.

Il controller rappresenta l'elemento centrale che gestisce e orchestra la rete consentendo di monitorare lo stato dei collegamenti e di applicare le politiche.

La rete è costituita da dispositivi P4 emulati su Mininet che consentono la programmazione dinamica del piano dati attraverso il linguaggio P4. Quesi due elementi riescono a comunicare tramite P4 Runtime e la componente SBI del controller.

Nei paragrafi successivi verrà descritto il flusso di lavoro della sperimentazione e l'interazione tra le componenti che ne fanno parte. La Figura 4.3 rappresenta una visione complessiva della configurazione. In seguito verranno forniti i dettagli più pratici relativi

a un'analisi approfondita del codice e della documentazione delle funzioni del controller utilizzate relative ai servizi e alle politiche.

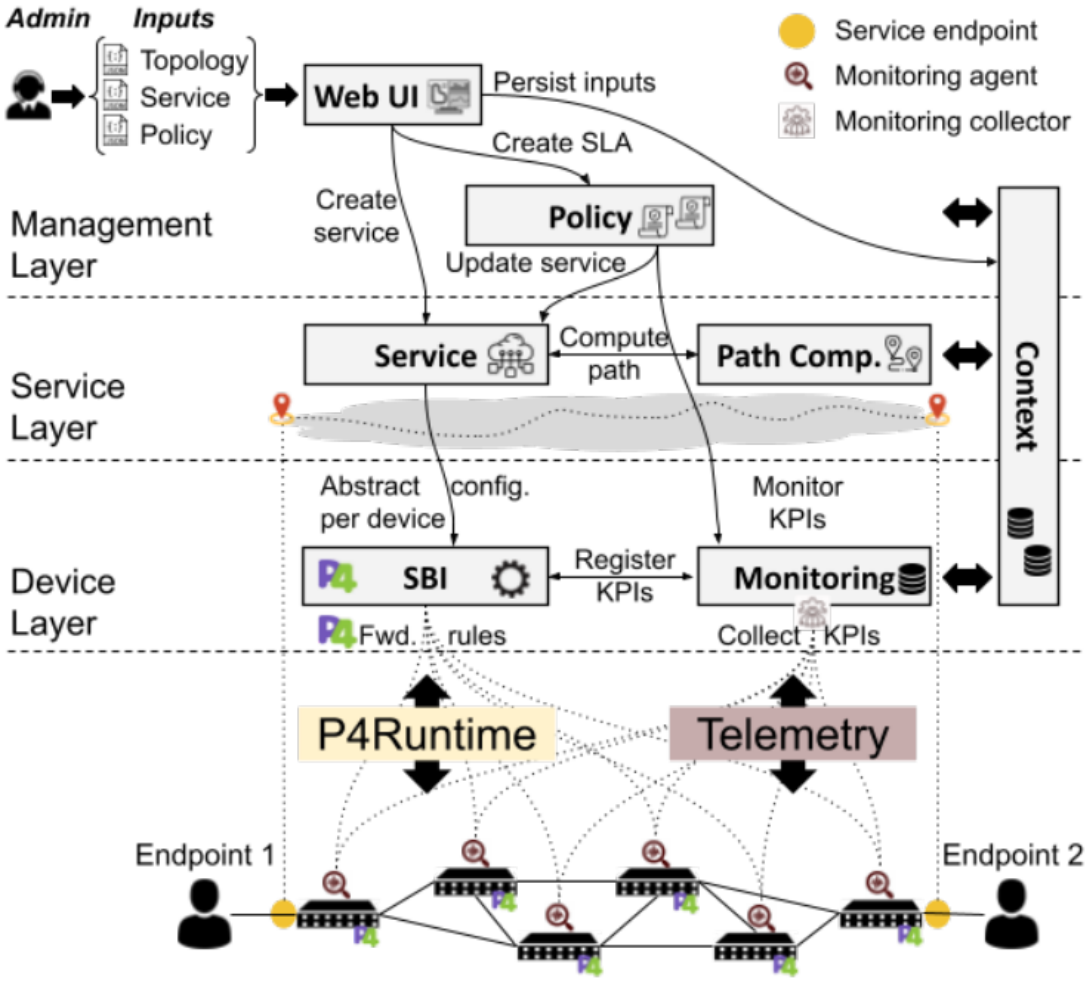


Figura 4.3: Interazione tra le componenti in TeraFlow per la creazione di un intento [10]

4.2.1 Servizio end-to-end

Come già detto in precedenza il controller sfrutta una South-Bound Interface (SBI) per interagire con i device, in questo caso, tramite l'API P4Runtime. Inizialmente, il codice P4 compilato, ossia la rappresentazione del programma P4 dopo il processo di compilazione (nella sperimentazione è rappresentato dai file p4info.txt e bmv2.json), viene copiato nel

pod SBI. Il file *p4info.txt* contiene le specifiche relative alle tabelle e alle azioni definite nel programma P4, come i campi di confronto e i parametri delle azioni, mentre il file *bmv2.json* descrive i tipi di header, il parser e altre componenti necessarie per la configurazione dei dispositivi. Questo primo passo permette di poter configurare dinamicamente i dispositivi di rete per instradare il traffico in base alle specifiche del programma P4.
Successivamente vengono registrati i dispositivi e i link nel Context database del controller SDN tramite lo script *run_test_01_bootstrap.sh*. Lo script richiama il file *test_functional_bootstrap.py* che ha più funzionalità: verifica che il contesto sia pronto; crea i dispositivi all'interno del controller aggiungendo a ciascuno le regole di connessione e configurazione; verifica che i dispositivi siano stati registrati nel database. A questo punto, siamo in grado di richiedere la prima parte dell'intento che sarà impelentato: una connessione tra due end points alla componente di Service tramite un servizio.
Per mantenere il processo agnostico rispetto ai dettagli della tecnologia, la componente sfrutta una definizione minima permettendo agli utenti di specificare solo quali sono i dispositivi finali. La richiesta del servizio è composta quindi solo dagli indirizzi IP e dalle porte che si vogliono connettere. Questi vengono tradotti in in regole P4 specifiche dal driver del dispositivo, come la creazione delle tabella di instradamento nei dispositivi lungo il percorso che determinano l'uscita dei pacchetti sulla base delle regole specificate.
La creazione del servizio è realizzata grazie alle funzioni messe a disposizione dalla componente di Service all'interno del controller descritte in seguito. L'implementazione di queste funzioni è presente nella repository pubblica [78].
Inizialmente, la funzione *CreateService* viene utilizzata per creare un servizio di connettività vuoto, specificandone solamente il tipo. Questa funzione salva l'identificativo del servizio creato nel database del Context e lo restituisce.
Successivamente, la funzione *UpdateService* aggiorna il servizio, popolando i campi richiesti come gli endpoint, i vincoli e le configurazioni di servizio, utilizzando i parametri passati [6]. La componente di Service si interfaccia con la componente Context per recuperare la versione più aggiornata del servizio e impostarne lo stato a "Planned" (pianificato). Infine, si rivolge alla PathComp per calcolare il percorso di rete.
Per eseguire questa operazione la PathComp utilizza informazioni di rete memorizzate nel

Context database come i nodi e i collegamenti presenti.
Una volta ottenuti i percorsi di rete con i relativi collegamenti calcolati dalla PathComp, la componente di Service crea un *Task Scheduler*, responsabile dell'esecuzione delle attività di installazione e smantellamento dei collegamenti, garantendo che queste vengano effettuate nell'ordine appropriato. Come ultima operazione, il *Task Scheduler* , esegue il metodo *Execute* per applicare le configurazioni necessarie ai dispositivi lungo il percorso stabilito tramite l'SBI. Dopo l'esecuzione, lo stato del servizio viene impostato su "Active" (attivo).
Parallelamente, il database della componente Context viene aggiornato con le nuove informazioni, e l'identificativo del servizio viene restituito all'entità chiamante.

4.2.2 Vincoli e configurazioni di servizio

Oltre alla specifica degli endpoints è possibile richiedere azioni supplementari come l'aggiunta di vincoli o configurazioni di servizio specifiche, come la posizione di un endpoints o il numero di giorni per cui un servizio deve rimanere attivo. Anche se nella documentazione ufficiale non sono menzionati, tutti i tipi di vincoli supportati si possono ritrovare nella repository pubblica [79].
Queste specifiche vengono inizializzate nella fase di creazione di un servizio e devono essere rispettate finchè quest'ultimo non verrà eliminato. Ciò semplifica la gestione dei servizi, in quanto la dichiarazione di informazioni aggiuntive può sostituire l'associazione di una politica.
Nella demo presa in considerazione i vincoli non erano specificati ma sono stati aggiunti per quanto riguarda la latenza e la capacità del percorso, come si può vedere nella Figura 4.17. Questa aggiunta non ha prodotto cambiamenti nella gestione del servizio, poiché le funzionalità che consentirebbero di monitorare e adeguare automaticamente il servizio in base alle condizioni della rete non sono ancora state implementate nel controller, ma sono previste per release future. Questo impedimento ha reso necessaria l'introduzione di una politica per far rispettare i vincoli richiesti dall'intento.

4.2.3 Politica

Per finire l'implementazione dell'intento viene aggiunta una politica.
Inizialmente alla creazione del servizio si associano le differenti Kpi tramite i KpiDescriptor. In ogni KpiDescriptor si devono specificare i valori numerici da rispettare insieme al tipo di Kpi (KpiSampleType). Quest'ultimo può essere predefinito, come i KpiSampleType per la latenza (KPISAMPLETYPE_SERVICE_LATENCY_MS) o la capacità (KPISAMPLETYPE_LINK_TOTAL_CAPACITY_GBPS, KPISAMPLETYPE_LINK_USED _CAPACITY_GBPS) (illustrati nel File pubblico [80]), oppure, utilizzando il tipo UNKNOWN, si può personalizzare tramite una descrizione.
Successivamente si richiama la funzione della componente di Monitoring *SetKpi* per ogni regola, passando il KpiDescriptor come parametro, così da associare il monitoraggio delle metriche richieste a una kpi nel Monitoring database e restituire il relativo identificatore. Se a uno stesso servizio si associano più KPI ognuna deve avere associato un KpiSampleType diverso, altrimenti, anche se con una descrizione differente, verrà associato lo stesso identificativo per monitorare metriche diverse creando conflitti. Infine al servizio è stata associata una politica specificando le diverse regole collegate tra loro da operatori booleani come AND/OR tramite file JSON. Per definire una politica va specificato l'id del contesto a cui si vuole associare e l'id del relativo servizio, successivamente si specificano le regole. Ogni regola è composta dall'identificatore della Kpi, l'operatore numerico (maggiore, minore o uguale [81]) insieme al valore limite per definire l'intervallo di valori non ammessi e infine l'azione da eseguire (le possibili azioni sono riportare nel file [82]). Appena creata la politica si trova nello stato di VALIDATED. Quando le metriche delle Kpi monitorate specificate nelle regole vengono sottoscritte dalla componente di Monitoring al Metrics database, la politica passa allo stato PROVISIONED. Se i requisiti richiesti non vengono più soddisfatti, la componente di Monitoring solleva un allarme che invia alla componente di Policy e la politica passa allo stato ACTIVE. A questo punto, la componente di Policy recupera il servizio interessato dalla componente Context e applica le azioni correttive specificate. Nel caso di ricalcolo del percorso viene richiamata la funzione *UpdateService* il cui funzionamento è stato descritto precedentemente. Durante questa fase lo stato della

politica passa a ENFORCED e la politica viene monitorata per un certo periodo per verificare l'efficacia effettiva delle azioni. Se le metriche continuano a non rispettare i requisiti, la politica passa allo stato INEFFECTIVE, segnalando la necessità di ulteriori interventi. In caso contrario, se l'azione risolve il problema, la politica viene contrassegnata come EFFECTIVE. Il flusso dello stato della politica è descritto anche nella Figura 3.2.

4.3 Sperimentazione

In questa sezione verranno descritti gli esperimenti svolti esponendo i codici, le topologie e i comandi usati in modo tale da permetterne la riproduzione. In alcuni casi è stata necessaria la modifica dei codici originali; i file sono riportati in Appendice.
Per verificare il comportamento del controller, non avendo a disposizione una rete reale, sono state utilizzate delle topologie di rete riprodotte tramite Mininet basate su BMv2[83]. BMv2 (Behavioral Model version 2) è un software open-source che implementa uno switch virtuale programmabile utilizzato principalmente per sviluppare e testare reti gestite dal paradigma SDN basate su protocolli programmabili, in particolare P4.
Per iniziare il lavoro si è partiti da una demo già preesistente, apportando in seguito le modifiche necessarie.

4.3.1 Esperimento 1

Inizialmente si è riprodotta la demo dell'Hackfest 3. Lo scopo di questa sperimentazione consiste nel creare un intento di rete, attraverso la definizione di un servizio, che connette due endpoint, client e server, tramite un collegamento che mantiene i valori di latenza sotto una determinata soglia grazie a una politica ad esso associata. I dettagli si possono ritrovare nella pagina del sito [68].
Dopo aver verificato che le componenti del controller siano in stato di running (*Kubectl get pods -n=tfs*), si istanzia la topologia [84] costituita da 4 nodi e 4 link, illustrata in Figura 4.4, sul container di Mininet collegato al controller tramite il comando *make start*.
Il file Objects.py mantiene le informazioni sulla topologia, sui dispositivi e sui servizi che saranno utilizzate dagli script (per il bootstrap, la creazione, l'eliminazione del servizio e

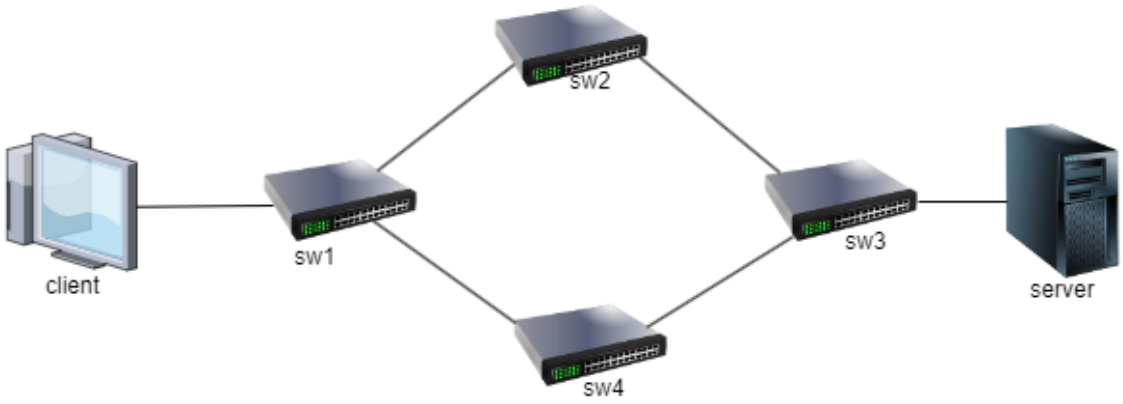


Figura 4.4: Topologia 4 switch e 2 percorsi

per la pulizia dell'ambiente alla fine dell'esperimento) per comunicare le configurazioni al controller.

Il primo comando da eseguire sul terminale del controller (*./setup.sh*) configura il pod Kubernetes dell'SBI inserendo le configurazioni di P4 necessarie al server per comunicare con gli switch. Questo comando crea l'ambiente di esecuzione e imposta i parametri di configurazione di P4, come le tabelle di instradamento e le regole di forwarding, e inizializza i driver degli switch per supportare la gestione dei dispositivi all'interno della rete.

Successivamente viene eseguito il comando *./run_test_01_bootstrap.sh* che richiama il file *test_functional_bootstrap.py*. Questo comando registra la topologia collegata a Mininet al controller, inserendo i dispositivi e i collegamenti nel Context database, così da poter proseguire con l'installazione del servizio.

Alla URL http://localhost:8080/webui si può accedere all'interfaccia grafica del controller e, selezionando il contesto *Context:(admin):Topology(admin)*, si può verificare la corretta registrazione nel controller della topologia di rete e delle configurazioni ad essa associate.

Prima di continuare con la creazione del servizio, in un altro terminale, dalla cartella *probe-tfs*, si copia ricorsivamente la cartella target nella directory corrente (*cp -r ∼/controller /src/tests/p4/probe/probe-tfs/target .*) necessaria per eseguire i comandi successivi. Dallo stesso terminale ci si collega al container di Mininet (*./connect-to-mininet.sh*) e si attiva lo script *tfsagent* (*./tfsagent.sh*) che permette di ascoltare gli eventi dalla componente di

Context e, quando un servizio viene registrato, crea la KpiDescriptor relativa alla latenza e la registra nel Monitoring database. In questo script il KpiSampleType specificato è UNKNOWN e il relativo commento è "*Latency value for service* {}". Successivamente si mette in attesa dei dati dal tfsping per ricevere le metriche relative alla latenza del collegamento e crea delle kpi da mandare alla componente di Monitoring.

A questo punto si può creare il servizio tra i due endpoints (*./run_test_02_create_service.sh*) e visualizzarlo sull'interfaccia web, come si può vedere nelle Figure 4.5 4.6. Gli end-

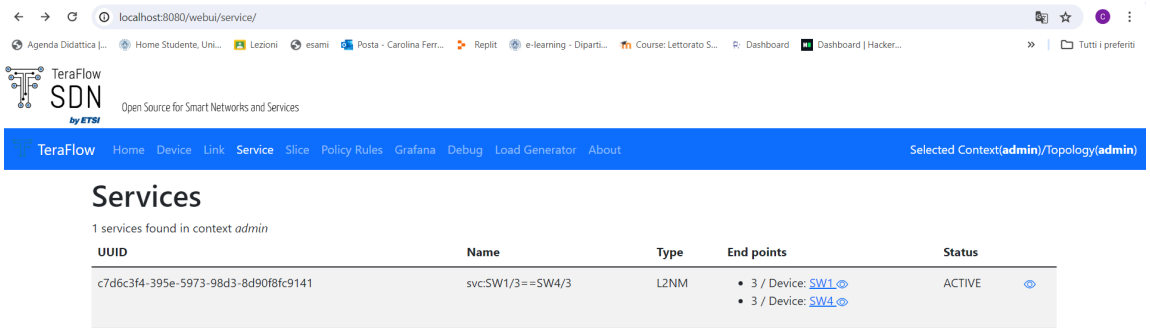


Figura 4.5: Servizio

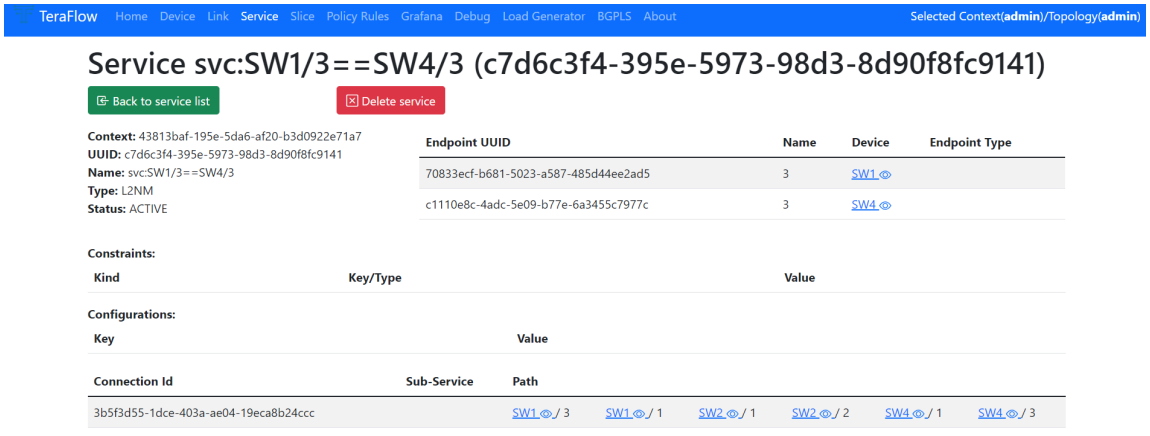


Figura 4.6: Servizio iniziale con topologia a 4 switch

points relativi a questa demo sono "SW1-port3" e "SW4-port3", gli switch della topologia collegati rispettivamente al client e al server instanziati su Mininet. A questo punto se

si esegue il comando *client ping server* sul terminale di Mininet il server risponde come atteso. Per implementare correttamente la politica è necessario modificare il file *manifests/policyservice.yaml*, come rappresentato in Figura 4.7. Per farlo, apriamo il file dal terminale con il comando *kubectl edit svc policyservice -n=tfs* per accedere alla configurazione del policyservice nel namespace tfs. In particolare si deve aggiungere *NodePort: 30060* così da consentire di esporre il servizio sulla porta 30060 che sarà utilizzata dallo script *addPolicy.sh* in seguito. Adesso è possibile installare la regola di politica. Questa

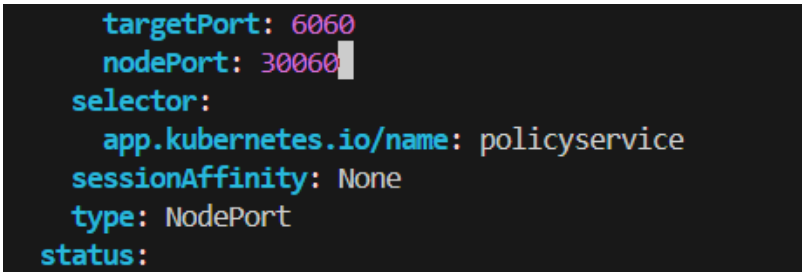
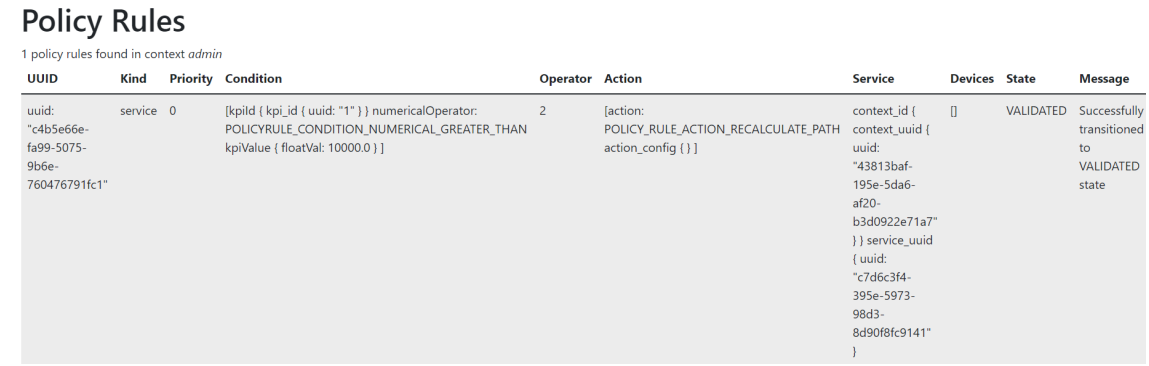


Figura 4.7: File policyservice modificato

viene definita utilizzando un file JSON (*policyAddService.json*) che specifica gli identificativi del contesto e del servizio a cui si vuole associare la politica, oltre alle regole che devono essere rispettate. In questo caso alla politica viene associata una sola regola rappresentata dall'identificativo KpiId, registrato in precedenza nel Monitoring database dall'agent, specificando la condizione che il valore non deve superare 10000. Se tale richiesta viene violata, la regola indicata è il ricalcolo del percorso. Lo script è richiamato eseguendo *./addPolicy.sh*. Dalla Web UI si può constatare che la politica è stata correttamente implementata nella Figura 4.8. Infine, per verificare l'efficacia della politica inserita, si aggiunge un ritardo su un'interfaccia di uno switch interno al percorso prestabilito. Ciò viene realizzato tramite il comando *switch2 tc qdisc add dev switch2-eth2 root netem delay 2000ms*. In questo caso, si applica un ritardo di 2000 ms sull'interfaccia eth2 dello switch switch2 con lo scopo di aumentare la latenza del percorso di rete e provocare una violazione dei vincoli imposti dalla regola della politica definita. Questo dovrebbe attivare l'azione corrispondente di ricalcolo del percorso. A questo punto, per verificare che ciò avvenga, viene eseguito *client ./tfsping*, uno script che utilizza il comando ping per calcolare la latenza del percorso e mandarla al *tfsagent*. Quest'ultimo trasmette la metrica



Policy Rules

1 policy rules found in context *admin*

UUID	Kind	Priority	Condition	Operator	Action	Service	Devices	State	Message
uuid: "c4b5e66e-fa99-5075-9b6e-760476791fc1"	service	0	[kpiId { kpi_id { uuid: "1" } } numericalOperator: POLICYRULE_CONDITION_NUMERICAL_GREATER_THAN kpiValue { floatVal: 10000.0 }]	2	[action: POLICY_RULE_ACTION_RECALCULATE_PATH action_config { }]	context_id { context_uuid { uuid: "43813baf-195e-5da6-af20-b3d0922e71a7" } } service_uuid { uuid: "c7d6c3f4-395e-5973-98d3-8d90f8fc9141" }	[]	VALIDATED	Successfully transitioned to VALIDATED state

Figura 4.8: Politica associata al servizio

alla componente di Monitoring, che ha il compito di verificare i valori e notificare la violazione della soglia imposta. In questo caso, il superamento del limite di latenza ha attivato correttamente l'azione prevista come si può vedere dal nuovo percorso in Figura 4.9. Si è

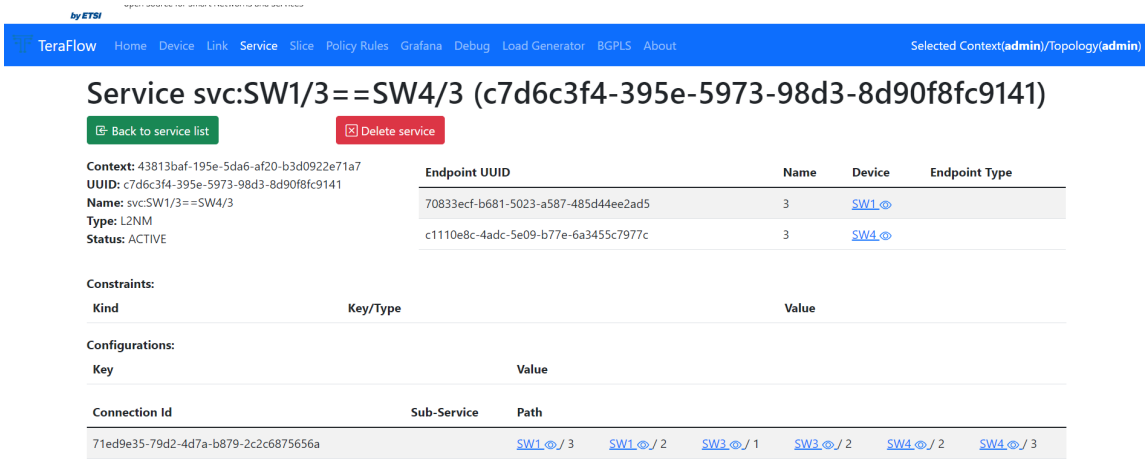


Figura 4.9: Servizio con cambiamento di percorso

fatta anche un'ulteriore prova, invece di aumentare la latenza si è direttamente disattivato un link tra due switch, in questo caso switch2 e switch4, tramite *link switch2 switch4 down* e si è potuto constatare che anche in questo caso è stata attivata l'azione della politica.

Prima di spengere la macchina virtuale del controller ci si deve assicurare di disattivare correttamente le risorse istanziate. Per prima cosa va rimossa la politica dalla quale

dipende il servizio *./removePolicy.sh*. Dopo aver eseguito il comando è opportuno verificare tramite l'interfaccia web che la politica non è più disponibile; vista l'instabilità del controller potrebbe essere necessario rieseguirlo una seconda volta prima di continuare. Successivamente si disattiva il servizio (*./run_test_03_ delete_service.sh*) e si rimuovono i dispositivi e l'ambiente allocati (*./run_test_04_ cleanup.sh*).

4.3.2 Esperimento 2

In questo esperimento la topologia di base da cui ha avuto inizio è stata quella descritta nel file del controller *8switch3path.py* [85]. Dopo lo script è stato modificato al fine di aggiungere dei link per creare 5 possibili percorsi al posto di 3; la topologia finale, con 8 nodi e 11 link, è illustrata nella Figura 4.10.

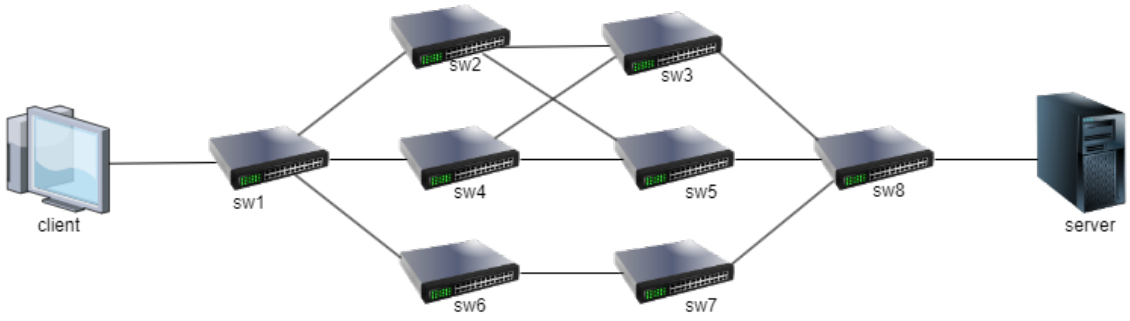


Figura 4.10: Topologia 8 switch e 5 percorsi

Prima di inizializzare la topologia è stato necessario modificare il file *docker-compose.yaml* per abilitare ulteriori porte di rete (50005,50006,50007,50008) al fine di stabilire le connessioni per la comunicazione tra Mininet containerizzato e il controller. Queste porte vengono esposte per consentire di comunicare con i dispositivi P4 emulati in Mininet, in modo che i comandi possano essere inviati e ricevuti correttamente.

Per questa sperimentazione sono stati utilizzati gli script presenti nella cartella P4 del controller [86]. E' stata necessaria l'aggiunta dei nuovi collegamenti al file *Objects.py* per una corretta sincronizzazione con il container Mininet. Queste modifiche consistono nell'inserimento di una interfaccia agli switch interessati (sw2, sw3, sw4, sw5) per poter

supportare un nuovo collegamento e la specifica di quest'ultimi in modo unidirezionale come riportato 6.2. Nello stesso file sono stati aggiunti alla definizione del servizio dei vincoli di configurazione relativi alla latenza e alla capacità, con la conseguente modifica anche del file *test_functional_create_service.py*. I cambiamenti sono evidenziati in Appendice 6.3.1.

Per quanto riguarda la parte iniziale i comandi per la configurazione dei dispositivi sono gli stessi della sezione precedente.

Successivamente viene definito un probe, necessario per il monitoraggio della rete e la raccolta di informazioni. Il probe in questione è costituito da due componenti principali: l'agent e il ping. L'agent è responsabile della creazione delle KpiId, della raccolta dei dati e dell'invio delle metriche al sistema di Monitoring del controller. Il ping, invece, viene utilizzato per testare la latenza del percorso di rete e la packet loss. L'implementazione utilizzata è disponibile al link [87] e rappresentano gli script utilizzati nella seconda parte della demo dell'Hackfest 3.

I file sono stati modificati per aggiungere la metrica della packet loss. Per quanto riguarda l'agent si è aggiunta la relativa KpiDescriptor da sottoscrivere alla componente di Monitoring ed è stata necessaria l'aggiunta del KpiSampleType per il KpiDescriptor della latenza. Pur essendo uno dei tipi predefiniti, non tutti sono implementati completamente, infatti i KpiSampleType in stato finale sono: UNKNOWN, PACKETS_TRANSMITTED, PACKETS_RECEIVED, BYTES_TRANSMITTED, BYTES_RECEIVED e LINK_TOTAL _CAPACITY_GB
Per questo motivo all'inizio la Kpi relativa alla latenza non aveva l'effetto desiderato; per sistemare questo problema è stata necessaria la modifica di alcuni file di configurazione del controller relativi alla gestione dei kpiSampleType. La stessa cosa non è stata possibile per la packet loss in quanto non era tra i tipi, anche solo parzialmente, implementati. La seconda parte dello script è stata adattata per ricevere la metrica aggiunta e inviare l'evento alla componente di Monitoring. Per quanto riguarda il file *ping2.py* si è calcolata la packet loss grazie ai risultati ricevuti dall'esecuzione del comando ping di Mininet. Affinchè ciò sia possibile vengono mantenute le informazioni del totale dei ping e quelli effettuati con successo e per calcolare la percentuale dei pacchetti persi, infine la metrica appena computata viene inviatra tramite la socket all'agent. Per un problema di comunicazione si è modifica-

to il percorso della socket, infatti inizialmente era */home/teraflow/ngsdn-tutorial/tmp/sock* ed è stata cambiata in *"/home/teraflow/ngsdn-tutorial/tmp/tfsping"*.
In un terminale relativo alla cartella si eseguono i seguenti comandi: *source ~/tfs-ctrl/tfs_runtime_env_vars.sh* per inserire le variabili di TeraFlow necessarie, se l'ambiente Python è disattivato *pyenv activate 3.9.18/envs/tfs*, e infine *python agent.py*. Quando si eseguirà il comando relativo alla creazione del servizio (*./run_test_02_create_service.sh*), sul terminale in cui è in esecuzione l'agent si vedrà l'output in Figura 4.12.
Il servizio richiesto ha come endpoints "SW1-port4" e "SW8-port4", come si può vedere in Figura 4.11.

Figura 4.11: Servizio iniziale con topologia a 8 switch

```
stream: New CREATE event:
context_id {
  context_uuid {
    uuid: "43813baf-195e-5da6-af20-b3d0922e71a7"
  }
}
service_uuid {
  uuid: "d5261206-1047-51c0-8ef2-89b4c601afe7"
}

loss: kpi_id {
  uuid: "1"
}

latency: kpi_id {
  uuid: "2"
}
```

Figura 4.12: Creazione kpi dallo script agent.py

Per creare la politica si riutilizzano gli script precedenti assicurandosi di cambiare nel file *policyAddService.json* gli id relativi al contesto e al servizio e di aggiungere alle regole quella relativa alla packet loss. Il relativo script è riposrtato 6.5 e la politica risultante è illustrata in Figura 4.13.

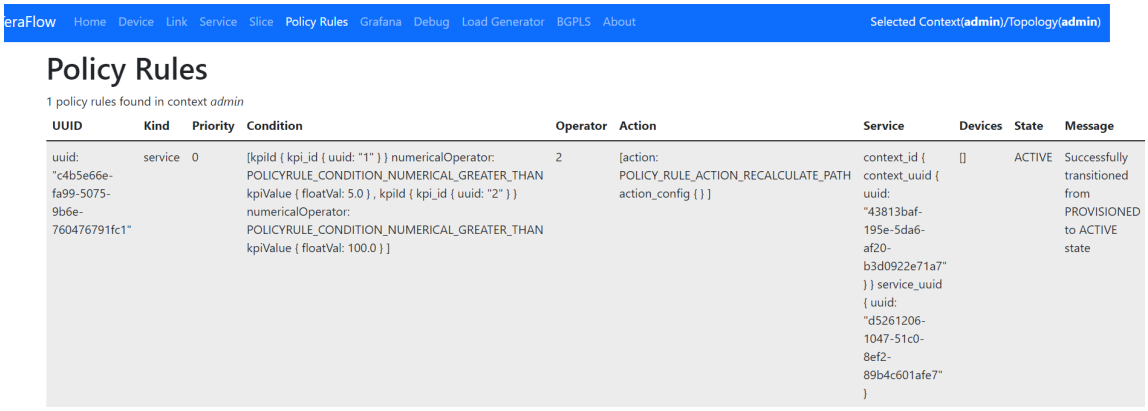


Figura 4.13: Politica associata al servizio

Sul terminale Mininet, oltre alle prove fatte aggiungendo dei ritardi e disattivando dei link, per verificare il funzionamento della seconda regola inserita ,si simula una packet loss. Questo avviene tramite il comando *switch4 tc qdisc add dev switch6-eth2 root netem loss 7%* che applica una perdita del 7% dei pacchetti sull'interfaccia eth2 dello switch 6. Al fine di verificare la politica viene attivato lo script *ping2.py* tramite Mininet (*client python ping2.py 10.0.0.2*). Anche in questo caso si è riscontato un cambiamento nel persorso come mostrato in Figura 4.14. Infine tramite i comandi esposti in precendenza, cambiando nello script *removePolicy.json* l'identificativo della politica, si sono disattivati in ordine la politica, il servizio e i dispositivi.

4.3.3 Esperimento 3

L'ultima topologia utilizzata è stata Abilene [88], raffigurata in Figura 4.15; una rete di trasporto creata da Internet2 con 11 nodi e 14 link. La configurazione della rete emulata in Mininet per questa topologia è disponibile su [89]. Per simulare l'ambiente, i valori di larghezza di banda vengono impostati a 30 Mbps e vengono effettuati dei cambiamenti per

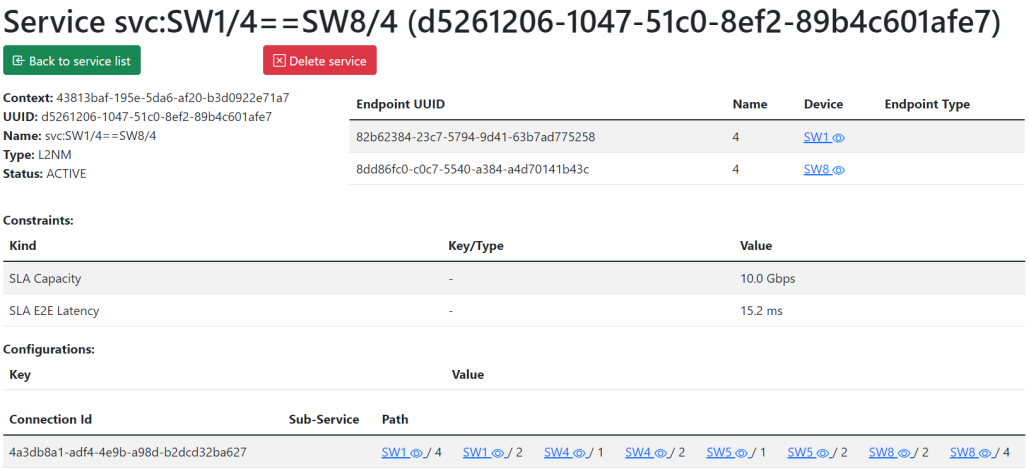


Figura 4.14: Servizio con cambiamento di percorso

garantire una corretta connessione con il container, nonché la configurazione del servizio da istanziare successivamente, come la configurazione ARP per i due endpoint. Lo script è riportato in Appendice 6.9 e La relativa topologia è rappresentata in Figura 4.16. Per poter utilizzare la topologia sono state aggiunte ulteriori porte a Mininet (5009, 5010, 5011).
Per quanto riguarda gli script sono stati riutilizzati quelli della sperimentazione precedente nella cartella P4 [86] con le modifiche necessarie al file *Objects.py*. I comandi per configurare i dispositivi sono i medesimi.
In questa sperimentazione è stata introdotta una nuova regola di politica relativa alla capacità. A tal fine è stato modificato nuovamente il file *agent.py* con l'aggiunta di una nuova KpiDescriptor, con il relativo KpiSampleType LINK_TOTAL_CAPACITY_GBPS, e si è avviato. Si è creato il servizio, illustrato in Figura 4.17, con endpoints "SW1-port3" e "SW8-port4" che nella topologia corrispondono alle città di New York e Kansas City. Successivamente si istanzia la politica, raffigurata in Figura 4.18, cambiando nel file JSON i campi relativi al contesto, al servizio e alle regole 6.7. Infine per verificarla , oltre ai test effettuati anche negli altri esperimenti, si è modificata la capacità tramite due differenti comandi: *switch4 tc qdisc add dev switch4-eth2 root tbf rate 1mbit burst 10kb latency 50ms* che limita la velocità di trasmissione a 1Mbps per secondo e imposta un ritardo massimo di 50ms per i pacchetti nella coda; *switch1 tc qdisc add dev switch1-eth1 root tbf rate*

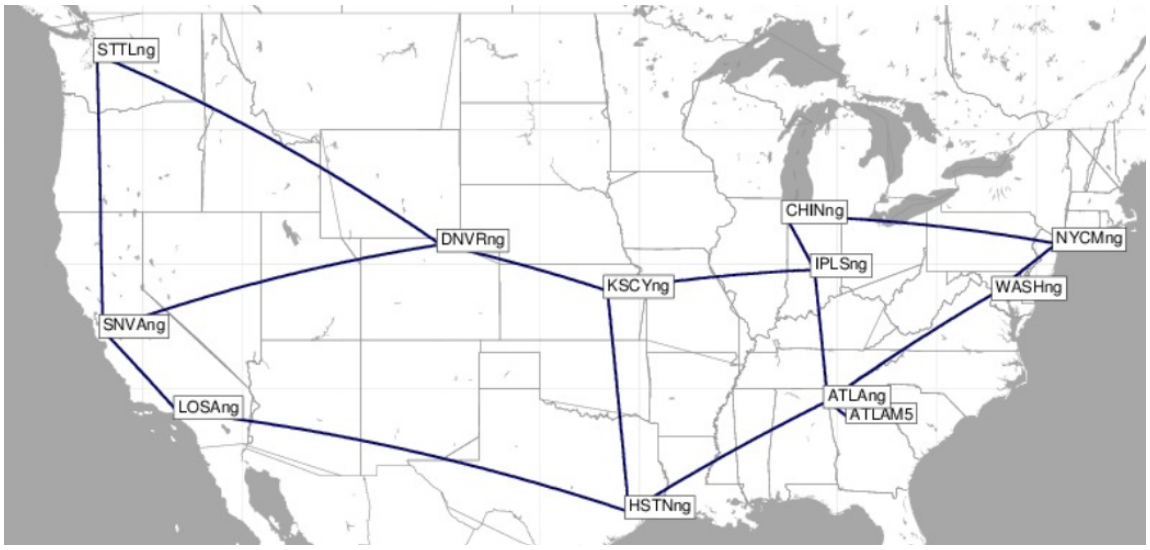


Figura 4.15: Topologia Abilene

1mbit burst 10kb limit 10000 che limita la velocità di trasmissione e specifica un limite sulla dimensione massima della coda a 10.000 byte.

Prima di attivare lo script del ping vanno effettuate delle modifiche. Mentre la latenza e la packet loss potevano essere calcolate tramite il comando ping, per la capacità questo non è possibile. Per calcolare la nuova metrica si è introdotto nel codice la chiamata anche al comando iperf, un comando per misurare attivamente l'ampiezza di banda disponibile nel percorso tra due endpoints. Prima di eseguire il codice su Mininet (*h0 python ping2.py 10.0.0.8*), è necessario attivare iperf sul dispositivo finale *h7* tramite il comando *iperf -s &*. L'opzione & permette di avviare iperf in background e riutilizzare lo stesso terminale per altre operazioni. Se gli script sono attivi dopo aver modificato il rate si verifica un cambiamento del percorso, come si può vedere nella Figura 4.19. Come negli altri esperimenti, con le solite modifiche ai file, si sono disattivati politica, servizio e dispositivi per ricreare l'ambiente iniziale.

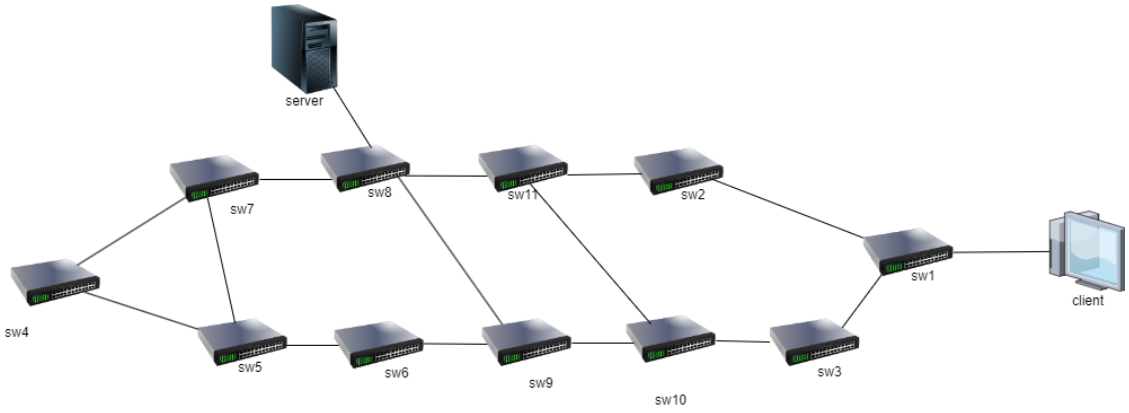


Figura 4.16: Topologia Abilene in Teraflow

Service svc:SW1/3==SW8/4 (be1fbc16-fad4-5832-9a32-754a7d134b81)

Back to service list | Delete service

Context: 43813baf-195e-5da6-af20-b3d0922e71a7
UUID: be1fbc16-fad4-5832-9a32-754a7d134b81
Name: svc:SW1/3==SW8/4
Type: L2NM
Status: ACTIVE

Endpoint UUID	Name	Device	Endpoint Type
70833ecf-b681-5023-a587-485d44ee2ad5	3	SW1	
8dd86fc0-c0c7-5540-a384-a4d70141b43c	4	SW8	

Constraints:

Kind	Key/Type	Value
SLA Capacity	-	10.0 Gbps
SLA E2E Latency	-	15.2 ms

Configurations:

Key	Value

Connection Id	Sub-Service	Path
e385f934-ae9b-445a-b7c4-531d61952188		SW1 / 3 SW1 / 1 SW2 / 1 SW2 / 2 SW11 / 1 SW11 / 2 SW8 / 3 SW8 / 4

Figura 4.17: Servizio iniziale con Abilene

Figura 4.18: Politica associata al servizio

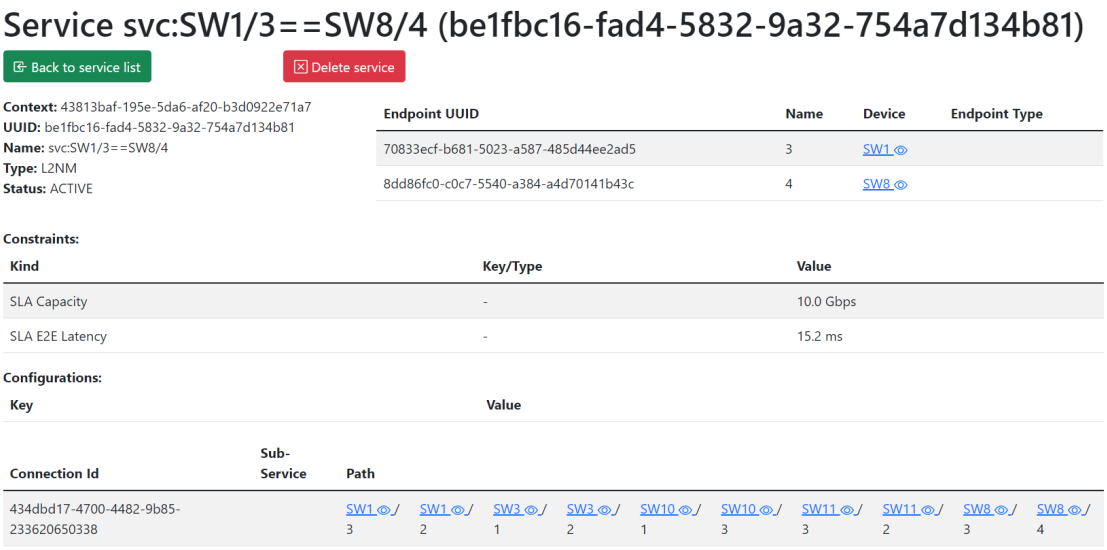


Figura 4.19: Servizio con un percorso modificato

5. Conclusioni

All'interno di questo elaborato sono state inizialmente descritte le motivazioni che stanno spingendo la ricerca di nuove tecnologie per superare le limitazioni dell'infrastruttura di rete tradizionale. Abbiamo presentato il paradigma SDN come uno dei più promettenti per conseguire tale obiettivo e sono state discusse le implementazioni più popolari per i controller SDN. Ci siamo concentrati sull'architettura e sulle funzionalità di TeraFlow, un controller SDN emergente progettato per le reti B5G, e abbiamo analizzato i suoi meccanismi di creazione e gestione dei servizi di rete. La sperimentazione si è focalizzata sulla creazione di servizi di connettività tra due end points, l'introduzione di politiche di rete specifiche, e la verifica della risposta del sistema a condizioni introdotte artificialmente. I test sono stati resi possibili dall'emulatore di rete Mininet. La valutazione è stata fatta su prove di raggiungibilità e di performance che, pur essendo abbastanza semplici, rappresentano un punto di partenza per analisi più complesse ed esegubili su reti più ampie.
In una rete reale, tuttavia, andrebbero svolti ulteriori accertamenti per garantire le stesse performance.
Durante la sperimentazione si sono riscontrate delle problematiche dovute alla mancanza di implementazione di alcune funzioni che hanno limitato lo sviluppo completo delle funzionalità desiderate.
Nonostante queste limitazioni, l'approccio sperimentale adottato ha permesso di ottenere risultati significativi, evidenziando la capacità del sistema di rispondere a condizioni di rete variabili e di applicare politiche di gestione del traffico in modo efficace.
Per lavori futuri si potrebbe pensare di implementare il servizio tramite interfaccia Web. Attualmente la creazione dei servizi tramite WebUI non supporta il tipo L2MV utilizzato in questa sperimetazione.
Si potrebbe inoltre sperimentare con topologie più grandi e politiche più complesse per approfondire ulteriormente le capacità e i limiti del sistema. Infine, si potrebbero verificare le stesse prove con altri protocolli e tipi di switch differenti da P4, ampliando così le possibilità di applicazione e la robustezza delle soluzioni proposte.

6. Appendice

6.1 Installazione ODL

Il controller viene eseguito all'interno di una Java Virtual Machine (JVM), quindi è necessario verificare quali versioni di Java supporta la distribuzione che si decide di installare. La versione stabile più recente di ODL è compatibile con le versioni di Java superiori alla 17, mentre per le versioni più datate quest'ultime non vanno bene. Per l'installazione si deve scaricare la distribuzione desiderata che si trova sul loro sito ufficiale nella pagina di download del software [90]. Successivamente è necessario fare l'unzip del file, navigare nella cartella e eseguire il seguente comando da terminale per avviare il controller.

```
./bin/karaf
```

La versione di ODL dell'immagine 6.1 è Potassium.

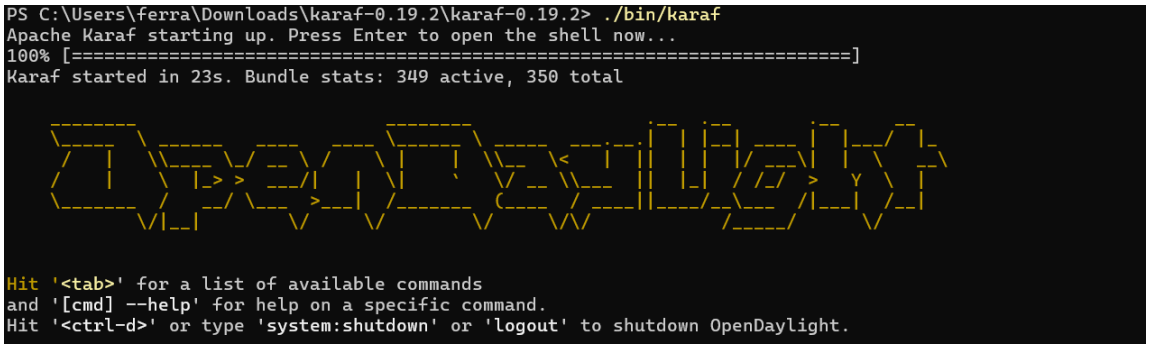


Figura 6.1: Installazione ODL

Successivamente, si può trovare una lista completa delle feature disponibili eseguendo il seguente comando.

```
feature:list
```

Per installarle invece

```
feature:install <feature1> <feature2>..
```

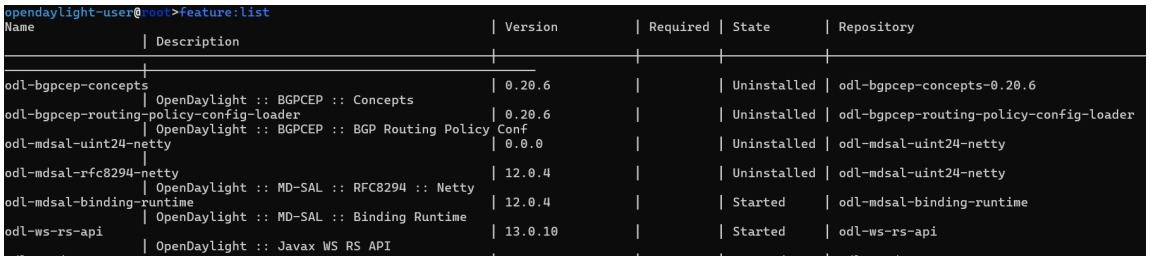


Figura 6.2: Alcune features disponibili

6.2 Topologia mininet 8sw aggiunta link

6.2.1 File Mininet

```
        self.addLink(switch2, switch5)
        self.addLink(switch4, switch3)

        # Host links
        self.addLink(client, switch1)   # Switch1: port 4
        self.addLink(server, switch8)   # Switch8: port 4
```

6.2.2 File Objects.py

```
# SW2_3 - SW5_3
LINK_SW2_SW5_UUID           = get_link_uuid(ENDPOINT_ID_SW2_3,
    ENDPOINT_ID_SW5_3)
LINK_SW2_SW5_ID             = json_link_id(LINK_SW2_SW5_UUID)
LINK_SW2_SW5                = json_link(LINK_SW2_SW5_UUID, [
   ENDPOINT_ID_SW2_3, ENDPOINT_ID_SW5_3])

# SW5_3 - SW2_3
LINK_SW5_SW2_UUID           = get_link_uuid(ENDPOINT_ID_SW5_3,
    ENDPOINT_ID_SW2_3)
```

```
LINK_SW5_SW2_ID             = json_link_id(LINK_SW5_SW2_UUID)
LINK_SW5_SW2                = json_link(LINK_SW5_SW2_UUID, [
   ENDPOINT_ID_SW5_3, ENDPOINT_ID_SW2_3])

# SW4_3 - SW3_3
LINK_SW4_SW3_UUID           = get_link_uuid(ENDPOINT_ID_SW4_3,
    ENDPOINT_ID_SW3_3)
LINK_SW4_SW3_ID             = json_link_id(LINK_SW4_SW3_UUID)
LINK_SW4_SW3                = json_link(LINK_SW4_SW3_UUID, [
   ENDPOINT_ID_SW4_3, ENDPOINT_ID_SW3_3])

# SW3_3 - SW4_3
LINK_SW3_SW4_UUID           = get_link_uuid(ENDPOINT_ID_SW3_3,
    ENDPOINT_ID_SW4_3)
LINK_SW3_SW4_ID             = json_link_id(LINK_SW3_SW4_UUID)
LINK_SW3_SW4
```

6.3 Aggiunta costraints

6.3.1 File Objects

```
SERVICE_SW1_SW8_UUID          = get_service_uuid(
   ENDPOINT_ID_SW1_4, ENDPOINT_ID_SW8_4)
SERVICE_SW1_SW8               = json_service_p4_planned(
   SERVICE_SW1_SW8_UUID)
SERVICE_SW1_SW8_ENDPOINT_IDS  = [DEVICE_SW1_ENDPOINT_IDS[3],
   DEVICE_SW8_ENDPOINT_IDS[3]]
constraints     = [
                {"sla_capacity": {"capacity_gbps": 10.0}},
                {"sla_latency": {"e2e_latency_ms": 15.2}}
            ]
```

```
SERVICES = [
    (SERVICE_SW1_SW8, SERVICE_SW1_SW8_ENDPOINT_IDS,
       constraints),
]
```

6.3.2 File test_functional_create_service.py

```
    for service, endpoints, constraints in SERVICES:
        # Insert Service (table entries)
        service_uuid = service['service_id']['service_uuid']['
           uuid']
        print('Creating Service {:s}'.format(service_uuid))
        service_p4 = copy.deepcopy(service)
        service_client.CreateService(Service(**service_p4))
        service_p4['service_endpoint_ids'].extend(endpoints)
        service_p4['service_constraints'].extend(constraints)
        service_client.UpdateService(Service(**service_p4))
```

6.4 Modifica setup in p4

```
export POD_NAME=$(kubectl get pods -n=tfs | grep device | awk
   '{print $1}')

kubectl exec ${POD_NAME} -n=tfs -c=server -- rm -rf /root/p4
kubectl exec ${POD_NAME} -n=tfs -c=server -- mkdir /root/p4

kubectl cp src/tests/p4/p4/p4info.txt tfs/${POD_NAME}:/root/p4
    -c=server
kubectl cp src/tests/p4/p4/bmv2.json tfs/${POD_NAME}:/root/p4
   -c=server
```

6.5 Script policyAddService packet loss


```
{
    "serviceId": {
        "context_id": {
            "context_uuid": {
                "uuid": "43813baf-195e-5da6-af20-b3d0922e71a7"
            }
        },
        "service_uuid": {
            "uuid": "be1fbc16-fad4-5832-9a32-754a7d134b81"
        }
    },
    "policyRuleBasic": {
        "priority": 0,
        "policyRuleId": {
            "uuid": {
                "uuid": "1"
            }
        },
        "booleanOperator": "POLICYRULE_CONDITION_BOOLEAN_OR",
        "policyRuleState": {
            "policyRuleStateMessage": ""
        },
        "actionList": [
            {
                "action": "POLICY_RULE_ACTION_RECALCULATE_PATH
                   ",
                "action_config": [
                    {
                        "action_key": "",
                        "action_value": ""
```

```
                }
            ]
        }
    ],
    "conditionList": [
        {
            "numericalOperator": "
               POLICYRULE_CONDITION_NUMERICAL_GREATER_THAN
               ",
            "kpiValue": {
                "floatVal": 5
            },
            "kpiId": {
                "kpi_id": {
                    "uuid": "1"
                }
            }
        },
        {
            "numericalOperator": "
               POLICYRULE_CONDITION_NUMERICAL_GREATER_THAN
               ",
            "kpiValue": {
                "floatVal": 100
            },
            "kpiId": {
                "kpi_id": {
                    "uuid": "2"
                }
            }
        }
    ]
```

```
    }
}
```

6.6 make file abilene

```
start: NGSDN_TOPO_PY := 4switch2path.py
start: _start

start-8: NGSDN_TOPO_PY := 8switch3path.py
start-8: _start

start-ab: NGSDN_TOPO_PY := abilene.py
start-ab: _start

start-v4: NGSDN_TOPO_PY := topo-v4.py
start-v4: _start
```

6.7 File addService con aggiunta capacità

```
{
    "serviceId": {
        "context_id": {
            "context_uuid": {
                "uuid": "43813baf-195e-5da6-af20-b3d0922e71a7"
            }
        },
        "service_uuid": {
            "uuid": "be1fbc16-fad4-5832-9a32-754a7d134b81"
        }
    },
```

```
"policyRuleBasic": {
    "priority": 0,
    "policyRuleId": {
        "uuid": {
            "uuid": "1"
        }
    },
    "booleanOperator": "POLICYRULE_CONDITION_BOOLEAN_OR",
    "policyRuleState": {
        "policyRuleStateMessage": ""
    },
    "actionList": [
        {
            "action": "POLICY_RULE_ACTION_RECALCULATE_PATH
               ",
            "action_config": [
                {
                    "action_key": "",
                    "action_value": ""
                }
            ]
        }
    ],
    "conditionList": [
        {
            "numericalOperator": "
               POLICYRULE_CONDITION_NUMERICAL_GREATER_THAN
               ",
            "kpiValue": {
                "floatVal": 5
            },
            "kpiId": {
```

```
                    "kpi_id": {
                        "uuid": "1"
                    }
                }
            },
            {
                "numericalOperator": "
                   POLICYRULE_CONDITION_NUMERICAL_GREATER_THAN
                   ",
                "kpiValue": {
                    "floatVal": 100
                },
                "kpiId": {
                    "kpi_id": {
                        "uuid": "2"
                    }
                }
            },
            {
                "numericalOperator": "
                   POLICYRULE_CONDITION_NUMERICAL_LESS_THAN",
                "kpiValue": {
                    "floatVal": 0.002
                },
                "kpiId": {
                    "kpi_id": {
                        "uuid": "3"
                    }
                }
            }
        ]
    }
```

```
}
```

6.8 implementazione probe con aggiunta capacità

6.8.1 agent.py

```
import os, threading, time, socket
from common.Settings import get_setting
from common.proto.context_pb2 import Empty, Timestamp
from common.proto.monitoring_pb2 import KpiDescriptor, Kpi,
   KpiId, KpiValue
from common.proto.kpi_sample_types_pb2 import KpiSampleType
from monitoring.client.MonitoringClient import
   MonitoringClient
from context.client.ContextClient import ContextClient

# ----- If you want to use .env file
#from dotenv import load_dotenv
#load_dotenv()
#def get_setting(key):
#    return os.getenv(key)


#### gRPC Clients
monitoring_client = MonitoringClient(get_setting('
   MONITORINGSERVICE_SERVICE_HOST'), get_setting('
   MONITORINGSERVICE_SERVICE_PORT_GRPC'))
context_client = ContextClient(get_setting('
   CONTEXTSERVICE_SERVICE_HOST'), get_setting('
   CONTEXTSERVICE_SERVICE_PORT_GRPC'))
```

```
### Locks and common variables
enabled_lock = threading.Lock()
kpi_id_lock = threading.Lock()
kpi_id_loss = KpiId()
kpi_id_latency = KpiId()
kpi_id_capacity = KpiId()
enabled = False

### Define the path to the Unix socket
socket_path = "/home/teraflow/ngsdn-tutorial/tmp/tfsping"
if os.path.exists(socket_path):
    os.remove(socket_path)

def thread_context_func():
    global kpi_id_loss
    global kpi_id_latency
    global kpi_id_capacity
    global enabled
    while True:
        # Listen to ContextService/GetServiceEvents stream
        events = context_client.GetServiceEvents(Empty())
        for event in events:
            event_service = event.service_id
            event_service_uuid = event_service.service_uuid.
               uuid
            event_type = event.event.event_type
            if event_type == 1:
                print(f"stream: New CREATE event:\n{
                   event_service}")
                #loss
                kpi_descriptor_loss = KpiDescriptor(
```

```
                        kpi_description = f"Loss Ratio for
                            service {event_service_uuid}",
                        service_id = event_service,
                        kpi_sample_type = KpiSampleType.
                            KPISAMPLETYPE_UNKNOWN
                        )
                with kpi_id_lock:
                    response_loss = monitoring_client.SetKpi(
                        kpi_descriptor_loss)
                    kpi_id_loss = response_loss
                    print("loss: "+str(kpi_id_loss))

                #latency
                kpi_descriptor_latency = KpiDescriptor(
                        kpi_description = f"Latency for
                            service {event_service_uuid}",
                        service_id = event_service,
                        kpi_sample_type = KpiSampleType.
                            KPISAMPLETYPE_SERVICE_LATENCY_MS
                        )
                with kpi_id_lock:
                    response_latency = monitoring_client.
                        SetKpi(kpi_descriptor_latency)
                    kpi_id_latency = response_latency
                    print("latency: "+str(kpi_id_latency))

                #capacity
                kpi_descriptor_capacity = KpiDescriptor(
                        kpi_description = f"Capacity for
                            service {event_service_uuid}",
                        service_id = event_service,
```

```
                        kpi_sample_type = KpiSampleType.
                            KPISAMPLETYPE_LINK_TOTAL_CAPACITY_GBPS

                        )
                with kpi_id_lock:
                    response_capacity = monitoring_client.
                        SetKpi(kpi_descriptor_capacity)
                    kpi_id_capacity = response_capacity
                    print("capacity: "+str(kpi_id_capacity))

                with enabled_lock:
                    enabled = True
            elif event_type == 3:
                print(f"stream: New REMOVE event:\n{
                    event_service}")
                with enabled_lock:
                    enabled = False

def thread_kpi_func():
    global kpi_id_loss
    global kpi_id_latency
    global kpi_id_capacity
    global enabled
    try:
        # Create socket object
        server_socket = socket.socket(socket.AF_UNIX, socket.
            SOCK_STREAM)

        # Bind the socket to the socket path
        server_socket.bind(socket_path)

        # Listen for incoming connections
```

```
        server_socket.listen(1)

        while True:
            print("Awaiting for new connection!")

            # Accept incoming connection
            connection, client_address = server_socket.accept
                ()

            # Read data from the connection
            data = connection.recv(1024)

            if data:
                with enabled_lock:
                    if enabled:
                        data = data.decode()
                        print(f"Received: {data}")
                        latency, loss, capacity = data.split('
                            ,')
                        with kpi_id_lock:

                            now = time.time()

                            new_timestamp = Timestamp()
                            new_timestamp.timestamp = now

                            new_value = KpiValue()
                            new_value.floatVal = float(latency
                                )

                            kpi_latency = Kpi (
                                    kpi_id = kpi_id_latency,
```

```
                                timestamp = new_timestamp,
                                kpi_value = new_value
                                )
                        print(kpi_latency)
                        response_latency =
                           monitoring_client.IncludeKpi(
                           kpi_latency)
                    with kpi_id_lock:
                        new_value.floatVal = float(loss)

                        kpi_loss = Kpi(
                            kpi_id=kpi_id_loss,
                            timestamp=new_timestamp,
                            kpi_value=new_value
                        )
                        print(kpi_loss)
                        response_loss = monitoring_client.
                           IncludeKpi(kpi_loss)

                    with kpi_id_lock:
                        new_value.floatVal = float(
                           capacity)

                        kpi_capacity = Kpi(
                            kpi_id=kpi_id_capacity,
                            timestamp=new_timestamp,
                            kpi_value=new_value
                        )
                        print(kpi_capacity)
                        response_capacity =
                           monitoring_client.IncludeKpi(
                           kpi_capacity)
```

```
            # Close the connection
            connection.close()


    except Exception as e:
        print(f"Error: {str(e)}")


def main():

    # Start Thread that listens to context events
    thread_context = threading.Thread(target=
       thread_context_func)
    thread_context.daemon = True
    thread_context.start()

    # Start Thread that listens to socket
    thread_kpi = threading.Thread(target=thread_kpi_func)
    thread_kpi.daemon = True
    thread_kpi.start()

    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        os.remove(socket_path)
        print("Script terminated.")

if __name__ == "__main__":
    main()
```

6.8.2 ping2.py

```
import socket, re, time, subprocess, sys, os

socket_path = "/tmp/tfsping"

#socket_path = "./tmp/sock"
def main():
    hostname = sys.argv[1]
   # server_host = sys.argv[2]
    count = 1
    wait = 5
    packet_sizes = [64, 512, 1024, 2048, 4096]

    total_pings = 0
    successful_pings = 0
    try:
        while True:
            start_time = time.time()

            try:
                # Run the ping command and capture the output
                result = subprocess.check_output(["ping", "-W"
                    , str(wait), "-c", str(count), hostname],
                    universal_newlines=True)
                response_time = float(re.findall(r"time
                    =([0-9.]+) ms", result)[0])

#cambia client_host e server_host
                iperf_result = subprocess.check_output(["iperf
                    ", "-c", hostname], universal_newlines=True
                    )
```

```
            bandwidth_mb = re.findall(r"(\d+\.?\d*)\s+
                Mbits/sec", iperf_result)[-1]
            #banda in gbps
            bandwidth = float(bandwidth_mb)/1000

        except subprocess.CalledProcessError as e:
            # If ping fails return negative response_time
            response_time = -1
            bandwidth = -1

        # Calculate new loss_ratio
        if response_time != -1:
            successful_pings += 1
        total_pings += 1
        moving_loss_ratio = round(((total_pings -
            successful_pings) / float(total_pings) * 100),
            2)

        print("Total pings: {}".format(total_pings))
        print("Successful pings: {}".format(
            successful_pings))

        print("Packet loss: {}\%".format(moving_loss_ratio
            ))
        print("Latency: {} ms".format(response_time))
        print("Bandwidth: {} Gbps/sec".format(bandwidth))

        data_latency = str(response_time)
        data_loss = str(moving_loss_ratio)
        data_bandwidth = str(bandwidth)
        data = data_latency+","+data_loss+","+
            data_bandwidth
```

```
            # Write results in socket
            try:
                client_socket = socket.socket(socket.AF_UNIX,
                    socket.SOCK_STREAM)
                client_socket.connect(socket_path)
                client_socket.send(data.encode())
                client_socket.close()
            except Exception as e:
                print(e)

            # Calculate the time taken by ping
            execution_time = time.time() - start_time
            # Wait the rest of the time
            wait_time = max(0, 6 - execution_time)
            time.sleep(wait_time)

    except KeyboardInterrupt:
        print("Script terminated.")

if __name__ == "__main__":
    main()
```

6.9 Rete emulata con la topologia Abilene

```
    def config(self, mac=None, ip=None, defaultRoute=None, lo=
        'up', gw=None,
               **_params):
        super(IPv4Host, self).config(mac, ip, defaultRoute, lo
            , **_params)
```

```
        self.cmd('ip -4 addr flush dev %s' % self.defaultIntf
            ())
        self.cmd('ip -6 addr flush dev %s' % self.defaultIntf
            ())
        self.cmd('ip -4 link set up %s' % self.defaultIntf())
        self.cmd('ip -4 addr add %s dev %s' % (ip, self.
            defaultIntf()))
        if gw:
            self.cmd('ip -4 route add default via %s' % gw)
        # Disable offload
        for attr in ["rx", "tx", "sg"]:
            cmd = "/sbin/ethtool --offload %s %s off" % (
                self.defaultIntf(), attr)
            self.cmd(cmd)

        def updateIP():
            return ip.split('/')[0]

        self.defaultIntf().updateIP = updateIP

class TutorialTopo(Topo):
    """Abilene Topology"""

    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        # Switches
        NewYork = self.addSwitch('s1', cls=StratumBmv2Switch,
            cpuport=CPU_PORT)
        Chicago = self.addSwitch('s2', cls=StratumBmv2Switch,
            cpuport=CPU_PORT)
```

```
        WashingtonDC = self.addSwitch('s3', cls=
            StratumBmv2Switch, cpuport=CPU_PORT)
        Seattle = self.addSwitch('s4', cls=StratumBmv2Switch,
            cpuport=CPU_PORT)
        Sunnyvale = self.addSwitch('s5', cls=StratumBmv2Switch
            , cpuport=CPU_PORT)
        LosAngeles = self.addSwitch('s6', cls=
            StratumBmv2Switch, cpuport=CPU_PORT)
        Denver = self.addSwitch('s7', cls=StratumBmv2Switch,
            cpuport=CPU_PORT)
        KansasCity = self.addSwitch('s8', cls=
            StratumBmv2Switch, cpuport=CPU_PORT)
        Houston = self.addSwitch('s9', cls=StratumBmv2Switch,
            cpuport=CPU_PORT)
        Atlanta = self.addSwitch('s10', cls=StratumBmv2Switch,
             cpuport=CPU_PORT)
        Indianapolis = self.addSwitch('s11', cls=
            StratumBmv2Switch, cpuport=CPU_PORT)


        # ... and now hosts
        NewYork_host = self.addHost( 'h0',cls=IPv4Host, mac="
            aa:bb:cc:dd:ee:11",
                                ip='10.0.0.1/24', gw='10.0.0.100'
                                    )
        Chicago_host = self.addHost( 'h1', cls=IPv4Host, mac="
            aa:bb:cc:dd:ee:22",
                                ip='10.0.0.2/24', gw='10.0.0.100'
                                    )
        WashingtonDC_host = self.addHost( 'h2',cls=IPv4Host,
            mac="aa:bb:cc:dd:ee:33",
```

```
                        ip='10.0.0.3/24', gw='10.0.0.100'
                           )
        Seattle_host = self.addHost( 'h3',cls=IPv4Host, mac="
           aa:bb:cc:dd:ee:44",
                        ip='10.0.0.4/24', gw='10.0.0.100'
                           )
        Sunnyvale_host = self.addHost( 'h4',cls=IPv4Host, mac=
           "aa:bb:cc:dd:ee:55",
                        ip='10.0.0.5/24', gw='10.0.0.100'
                           )
        LosAngeles_host = self.addHost( 'h5',cls=IPv4Host, mac
           ="aa:bb:cc:dd:ee:66",
                        ip='10.0.0.6/24', gw='10.0.0.100'
                           )
        Denver_host = self.addHost( 'h6', cls=IPv4Host, mac="
           aa:bb:cc:dd:ee:77",
                        ip='10.0.0.7/24', gw='10.0.0.100'
                           )
        KansasCity_host = self.addHost( 'h7',cls=IPv4Host, mac
           ="aa:bb:cc:dd:ee:88",
                        ip='10.0.0.8/24', gw='10.0.0.100'
                           )
        Houston_host = self.addHost( 'h8',cls=IPv4Host, mac="
           aa:bb:cc:dd:ee:99",
                        ip='10.0.0.9/24', gw='10.0.0.100'
                           )
        Atlanta_host = self.addHost( 'h9',cls=IPv4Host, mac="
           aa:bb:cc:dd:ee:10",
                        ip='10.0.0.10/24', gw='10.0.0.100'
                            )
        Indianapolis_host = self.addHost( 'h10',cls=IPv4Host,
           mac="aa:bb:cc:dd:ee:11",
```

```
                        ip='10.0.0.11/24', gw='10.0.0.100'
                            )

        # add edges between switches
        self.addLink( NewYork , Chicago, bw=30, delay='
            0.806374975652ms')
        self.addLink( NewYork , WashingtonDC, bw=30, delay='
            0.605826192092ms')
        self.addLink( Chicago , Indianapolis, bw=30, delay='
            1.34462717203ms')
        self.addLink( WashingtonDC , Atlanta, bw=30, delay='
            0.557636936322ms')
        self.addLink( Seattle , Sunnyvale, bw=30, delay='
            1.28837123738ms')
        self.addLink( Seattle , Denver, bw=30, delay='
            1.11169346865ms')
        self.addLink( Sunnyvale , LosAngeles, bw=30, delay='
            0.590813628707ms')
        self.addLink( Sunnyvale , Denver, bw=30, delay='
            0.997327682281ms')
        self.addLink( LosAngeles , Houston, bw=30, delay='
            1.20160833263ms')
        self.addLink( Denver , KansasCity, bw=30, delay='
            0.223328790403ms')
        self.addLink( KansasCity , Houston, bw=30, delay='
            1.71325092726ms')
        self.addLink( KansasCity , Indianapolis, bw=30, delay=
            '0.240899959477ms')
        self.addLink( Houston , Atlanta, bw=30, delay='
            1.34344500256ms')
        self.addLink( Atlanta , Indianapolis, bw=30, delay='
            0.544962634977ms')
```

```
        # add edges between switch and corresponding host
        self.addLink( NewYork , NewYork_host )
        self.addLink( Chicago , Chicago_host )
        self.addLink( WashingtonDC , WashingtonDC_host )
        self.addLink( Seattle , Seattle_host )
        self.addLink( Sunnyvale , Sunnyvale_host )
        self.addLink( LosAngeles , LosAngeles_host )
        self.addLink( Denver , Denver_host )
        self.addLink( KansasCity , KansasCity_host )
        self.addLink( Houston , Houston_host )
        self.addLink( Atlanta , Atlanta_host )
        self.addLink( Indianapolis , Indianapolis_host )

def main():
    net = Mininet(topo=TutorialTopo(), controller=None)
    net.start()

    #get hosts new york kansascity
    NewYork_host = net.hosts[0]
    NewYork_host.setARP('10.0.0.8', 'aa:bb:cc:dd:ee:88')
    KansasCity_host = net.hosts[1]
    KansasCity_host.setARP('10.0.0.1', 'aa:bb:cc:dd:ee:11')


    CLI(net)
    net.stop()
    print '#' * 80
    print 'ATTENTION: Mininet was stopped! Perhaps
       accidentally?'
```

```
    print 'No worries, it will restart automatically in a few
       seconds...'
    print 'To access again the Mininet CLI, use `make mn-cli`'
    print 'To detach from the CLI (without stopping), press
       Ctrl-D'
    print 'To permanently quit Mininet, use `make stop`'
    print '#' * 80


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description='Mininet topology script for 2x2 fabric
           with stratum_bmv2 and IPv4 hosts')
    args = parser.parse_args()
    setLogLevel('info')

    main()
```

Bibliografia

[1] Babak Darabinejad. An introduction to software-defined networking. *International Journal of Intelligent Information Systems*, 3:71, 11 2014.

[2] Bhargavi Goswami. Experimenting with onos scalability on software defined network. *Journal of Advanced Research in Dynamical and Control Systems*, 10:1820–1830, 01 2019.

[3] Gonzalez Carlos Javier. *Management of a heterogeneous distributed architecture with the SDN*. PhD thesis, 12 2017.

[4] T Kubernetes. Kubernetes. *Kubernetes. Retrieved May*, 24:2019, 2019.

[5] Alberto Mozo, Amit Karamchandani, Luis Cal, Sandra Gómez-Canaval, Antonio Pastor, and Lluis Gifre. A machine-learning-based cyberattack detector for a cloud-based sdn controller. *Applied Sciences*, 13:4914, 04 2023.

[6] D3.2: Final evaluation of life-cycle automation and high performance sdn components. *teraflow-h2020.eu*, 2023.

[7] K. Indrasiri and D. Kuruppu. *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[8] Protocol buffer delle componenti in teraflow. `https://gitlab.com/teraflow-h2020/controller/-/tree/develop/proto`.

[9] Ya Gao and Zhenling Wang. A review of p4 programmable data planes for network security. *Mobile Information Systems*, 2021(1):1257046, 2021.

[10] Panagiotis Famelis, Georgios P. Katsikas, Vasilios Katopodis, Carlos Natalino, Lluis Gifre Renom, Ricardo Martinez, Ricard Vilalta, Dimitrios Klonidis, Paolo Monti, Daniel King, and Adrian Farrel. P5: Event-driven policy framework for p4-based traffic engineering. In *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–3, 2023.

[11] B. Martini, M. Gharbaoui, and P. Castoldi. Intent-based zero-touch service chaining layer for software-defined edge cloud networks. *Computer Networks*, 212:109034, 2022.

[12] Andrea Biancini, Mauro Campanella, Luca Prete, and Fabio Farina. Software defined networking esperienze openflow e l'interesse per cloud. 04 2013.

[13] Maede Zolanvari. Sdn for 5g. 2015.

[14] Le prospettive del 5g. *aeit*, 104(11/12):39–41, 2018. `https://www.aeit.it/aeit/edicola/aeit/aeit2018/aeit2018_06_cisa/aeit2018_06_riv.pdf`.

[15] Laura Zanotti. Intent-based networking (ibn): significato e vantaggi del networking basato sugli intenti. *ZeroUno*, 2019.

[16] Yiming Wei, Mugen Peng, and Yaqiong Liu. Intent-based networks for 6g: Insights and challenges. *Digital Communications and Networks*, 6(3):270–280, 2020.

[17] Etsi. `https://www.etsi.org/`.

[18] Onos. `https://opennetworking.org/onos/`.

[19] Opendaylight. `https://www.opendaylight.org/`.

[20] Daniel Barattini. Supporto a micro-servizi per controller ad alta scalabilità e affidabilità. Master's thesis, Alma mater studiorum - universita' di Bologna, 2020.

[21] Routing information protocol. *IBM*, 2023. `https://www.ibm.com/docs/en/i/7.3?topic=routing-information-protocol`.

[22] Muhammad Fauzan Rafi Sidiq Widjonarto. Application of dijkstra algorithm on ospf routing protocol and its effect in modern networks, 2018. 13518147.

[23] Nicholas Brasini. *Analisi e sviluppo di un'interfaccia web per gestire controller SDN*. PhD thesis, Alma mater studiorum - universita' di Bologna, 2017.

[24] William Stallings. *Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud*. Pearson Education, 2015. Accessed: 6 September 2024.

[25] Diego Kreutz, Fernando Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *ArXiv e-prints*, 103, 06 2014.

[26] Open networking foundation. `https://opennetworking.org/`.

[27] Yustus Eko Oktian, SangGon Lee, HoonJae Lee, and JunHuy Lam. Distributed sdn controller system: A survey on design choice. *Computer Networks*, 121:100–111, 2017.

[28] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[29] B. Martini, M. Gharbaoui, and P. Castoldi. Intent-based zero-touch service chaining layer for software-defined edge cloud networks. *Computer Networks*, 212:109034, 2022.

[30] *MEC*. url: `https://www.etsi.org/technologies/multi-access-edge-computing`.

[31] Nfv. `https://www.etsi.org/technologies/nfv`.

[32] Ietf. `https://www.ietf.org/`.

[33] Hao Yu, Hamid Rahimi, Carl Janz, et al. Building a comprehensive intent-based networking framework: A practical approach from design concepts to implementation. *Journal of Network and Systems Management*, 32:47, 2024.

[34] Nathan Sousa, Nazrul Islam, Danny Perez, and Christian Esteve Rothenberg. Policy-driven network traffic rerouting through intent-based control loops. 07 2019.

[35] B. Martini, M. Gharbaoui, and P. Castoldi. Intent-based network slicing for sdn vertical services with assurance: Context, design and preliminary experiments. *Future Generation Computer Systems*, 142:101–116, 2023.

[36] Osgi. https://www.osgi.org/.

[37] Apache karaf. https://kafka.apache.org/.

[38] Ashwin Rajaratnam, Ruturaj Kadikar, Shanthi Prince, and M. Valarmathi. Software defined networks: Comparative analysis of topologies with onos. In *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pages 1377–1381, 2017.

[39] Davide Sanvito, Daniele Moro, Mattia Gullì, Ilario Filippini, Antonio Capone, and Andrea Campanella. Onos intent monitor and reroute service: enabling plug&play routing logic. *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 272–276, 2018.

[40] Putri Monika, Ridha Negara, and Danu Sanjoyo. Performance analysis of software defined network using intent monitor and reroute method on onos controller. *Bulletin of Electrical Engineering and Informatics*, 9(5):2065–2073, 2020.

[41] Lfnetworking. https://lfnetworking.org/.

[42] Giacomo Ondesca. *Monitoraggio di rete con tecnologie SDN (Software-Defined Networking)*. PhD thesis, Politecnico di Torino, 2021.

[43] Benedetta Contigiani. *Improving network management with Software-Defined Networking*. PhD thesis, Università degli studi di Camerino, 2016.

[44] Modulo nic di odl. https://test-odl-docs.readthedocs.io/en/stable-boron/user-guide/network-intent-composition-(nic)-user-guide.html#.

[45] Confluence. Network intent composition proposal, 2021.

[46] Kubernetes. hhttps://kubernetes.io/.

[47] Madiha H Syed, Eduardo B Fernandez, et al. The software container pattern. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, pages 24–26, 2015.

[48] *CNCF*. url: https://www.cncf.io/.

[49] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. *Kubernetes: up and running*. " O'Reilly Media, Inc.", 2022.

[50] D1.4: Final project periodic report. *teraflow-h2020.eu*, 2023.

[51] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[52] *Indagine statistica sull'uso dei microservizi*. url: https://www.statista.com/statistics/1233937/microservices-adoption-level-organization/.

[53] Zhijun Ding, Song Wang, and Changjun Jiang. Kubernetes-oriented microservice placement with dynamic resource allocation. *IEEE Transactions on Cloud Computing*, 11(2):1777–1793, 2023.

[54] Teraflow. https://www.teraflow-h2020.eu/%.

[55] Progetto horizon 2020. https://research-and-innovation.ec.europa.eu/funding/funding-opportunities/funding-programmes-and-open-calls/horizon-2020_en.

[56] TeraFlow. *5GPPP*, 2021. url: https://5g-ppp.eu/teraflow/.

[57] D2.2 - preliminary requirements, architecture design, techno-economic studies and data models. *teraflow-h2020.eu*, 2023.

[58] D4.1: Preliminary evaluation of teraflow security and b5g network integration. *teraflow-h2020.eu*, 2020.

[59] Teraflowsdn core components. https://www.teraflow-h2020.eu/blog/teraflowsdn-core-components, 2021.

[60] D3.1: Preliminary evaluation of life-cycle automation and high performance sdn components. *teraflow-h2020.eu*, 2022.

[61] Daniel Adanza, Lluis Gifre, Pol Alemany, Juan-Pedro Fernández-Palacios, Oscar González de Dios, Raul Muñoz, and Ricard Vilalta. Enabling traffic forecasting with cloud-native sdn controller in transport networks. *Computer Networks*, 250:110565, 2024.

[62] Grafana. `https://grafana.com/`.

[63] D5.3 final demonstrators and evaluation report. *teraflow-h2020.eu*, 2023.

[64] grpc. `https://grpc.io/`.

[65] F. Paolucci, A. Sgambelluri, M. Dallaglio, F. Cugini, and P. Castoldi. Demonstration of grpc telemetry for soft failure detection in elastic optical networks. In *2017 European Conference on Optical Communication (ECOC)*, pages 1–3, 2017.

[66] Ricard Vilalta, Noboru Yoshikane, Ramon Casellas, Ricardo Martínez, Shohei Beppu, Daiki Soma, Seiya Sumita, Takehiro Tsuritani, Itsuro Morita, and Raul Muñoz. Grpc-based sdn control and telemetry for soft-failure detection of spectral/spacial superchannels. In *45th European Conference on Optical Communication (ECOC 2019)*, pages 1–4, 2019.

[67] Hari Krishna and Rinki Sharma. Comparative study of orchestration using grpc api and rest api in server creation time: An openstack case. *International Journal of Computer Networks & Communications (IJCNC)*, 16(1), jan 2024. Available at SSRN: `https://ssrn.com/abstract=4710302`.

[68] Hackfest 3. `https://labs.etsi.org/rep/groups/tfs/-/wikis/TFS-HACKFEST-3`.

[69] Link alla macchina virtuale per l'hackfest 3. `https://drive.google.com/file/d/1OaukXmAC1uaeIAChkEpvBB9mSkUHimsR/view`.

[70] *Cartella ngsdn-tutorial*. url: `https://github.com/opennetworkinglab/ngsdn-tutorial`.

[71] Mininet. `https://mininet.org/`.

[72] Mininet overview. https://mininet.org/overview/.

[73] Francesco Cristiano. *Emulazione distribuita di reti di telecomunicazioni su piattaforma Mininte*. PhD thesis, Alma Mater Studiorum- Università di Bologna, 2013.

[74] Nikhil Handigol Bob Lantz and Vimal Jeyakumar Brandon Heller. *Introduction to Mininet*. url: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet.

[75] Mininet.org. *Mininet Walkthrough*. url: https://mininet.org/walkthrough/.

[76] P4. https://p4.org/.

[77] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[78] *File ServiceServiceServicerImpl*. url: https://labs.etsi.org/rep/tfs/controller/-/blob/master/src/service/service/ServiceServiceServicerImpl.py.

[79] *File Constraint*. url: https://labs.etsi.org/rep/tfs/controller/-/blob/master/src/common/tools/object_factory/Constraint.py.

[80] *File KpiSampleType*. url: https://labs.etsi.org/rep/tfs/controller/-/blob/master/src/ztp/target/generated-sources/grpc/kpi_sample_types/KpiSampleTypes.java.

[81] *File Operatori Numerici*. url: https://labs.etsi.org/rep/tfs/controller/-/blob/master/src/policy/src/main/java/org/etsi/tfs/policy/policy/model/NumericalOperator.java.

[82] *File Azioni Policy*. url: https://labs.etsi.org/rep/tfs/controller/-/blob/master/proto/policy_action.proto.

[83] P4 Language. *Behavioural model (bmv2) reference P4 software switch*, mar 2023. Available: https://github.com/p4lang/behavioral-model.

[84] Topologia mininet 4 switch. https://labs.etsi.org/rep/tfs/controller/-/blob/master/src/tests/hackfest3/mininet/4switch2path.py.

[85] *File con la topologia Mininet 8 switch e 3 path*. url: https://labs.etsi.org/rep/tfs/controller/-/blob/master/src/tests/p4/mininet/8switch3path.py.

[86] Cartella p4. https://labs.etsi.org/rep/tfs/controller/-/tree/master/src/tests/p4.

[87] *Cartella con l'implementazione del probe il Python*. url: https://labs.etsi.org/rep/tfs/controller/-/blob/master/src/tests/hackfest3/new-probe/solution.

[88] Wikipedia contributors. Abilene network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Abilene_Network&oldid=1220796057, 2024. [Online; accessed 17-August-2024].

[89] *File Mininet per la topologia Abilene*. url: https://github.com/sjas/assessing-mininet/blob/master/parser/topologies/Abilene.graphml-generated-Mininet-Topo.py.

[90] Installazione opendaylight. https://docs.opendaylight.org/en/latest/downloads.html.