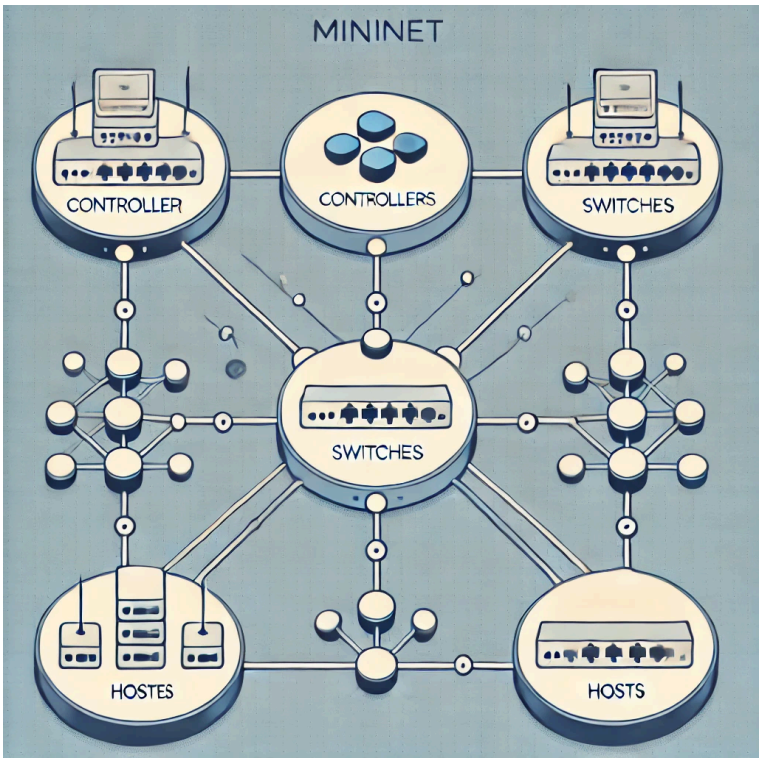
Relazione tecnica di progetto

Sperimentazioni di rete con Mininet



Alessandro Botta [0752081]
Domenico Puglisi [0750729]

Commissionato dal Professore: Fabrizio Giuliano

UNIVERSITÀ
DEGLI STUDI
DI PALERMO

a.a. 2024/25 CdL Informatica Triennale
16/01/2025

Abstract

Il presente documento descrive lo sviluppo e l'implementazione di un sistema di comunicazione di rete basato su tecnologia **SDN** (Software Defined Networking), realizzato da Alessandro Botta e Domenico Puglisi per l'anno accademico 2024/25 del CdL in Informatica. Il progetto si avvale di **Mininet** per emulare una rete composta da switch, router e host e utilizza il controller **Ryu** per configurare dinamicamente gli indirizzi IP e le tabelle di flusso attraverso un algoritmo di routing dinamico basato su **Dijkstra**. La specificità del progetto risiede nell'implementazione di un server HTTP con **Flask** per la misurazione del Round Trip Time (**RTT**) tra i nodi della rete, con i risultati memorizzati su un database **SQLite**. Tutti i dettagli implementativi, inclusi i codici sorgente, sono disponibili nel repository GitHub del progetto. Questo lavoro fornisce un'analisi dettagliata dell'architettura di rete e analizza la coerenza degli RTT misurati con le prestazioni teoriche attese.

Realizzazione Rete

Lo sviluppo della topologia di rete richiesta è stato strutturato nelle seguenti fasi:

1. **Configurazione dell'ambiente di sviluppo**, incluse le dipendenze utilizzate; ulteriori strumenti di sviluppo; ambiente di esecuzione per il codice da realizzare.
2. **Realizzazione della topologia** della rete in codice Python con le API di Mininet.
3. **Realizzazione del controller** per implementare all'interno della rete l'uso dell'algoritmo di Dijkstra come algoritmo di routing.
4. **Test della rete** ottenuta, inclusa l'integrazione con il Server HTTP in Flask sviluppato separatamente.

Configurazione Ambiente di Sviluppo

Nella fase iniziale, è stato essenziale identificare una versione di **Python** compatibile sia con la libreria **mininet** che con **ryu**, affinché tutti i moduli di codice potessero eseguiti su un unico sistema operativo comune. Tali moduli sono stati sviluppati all'interno della sottocartella `mininet_config` del progetto.

Per la gestione delle dipendenze e l'installazione dell'interprete Python adeguato, nonché la creazione del virtual environment del progetto, è stato scelto lo strumento **uv**. All'interno del file `pyproject.toml` sono dunque elencate le **dipendenze** richieste con l'esatta versione che è stata utilizzata per ciascuna di esse; nel file `uv.lock` sono specificati gli esatti **URL** usati per scaricare tali dipendenze dal Python Package Index, in maniera tale da garantire la riproducibilità dell'ambiente di sviluppo.
La versione dell'interprete scelta per lo sviluppo è di **Python 3.9**, l'ultima versione tuttora compatibile con le funzionalità di ryu utilizzate dal nostro controller. Difatti, la versione 3.10 di Python ha deprecato alcuni moduli preinstallati con l'interprete come **distutils**; ryu non è stato aggiornato di conseguenza e, sulle versioni moderne dell'interprete, la sua installazione fallisce. Per eseguire correttamente la sua installazione, è necessario il comando `pip install ryu` eseguito manualmente all'interno del virtual environment creato con **uv**.

Come ambiente di esecuzione, abbiamo fatto uso dell'immagine **ubuntu-server-20.04** fornita dagli sviluppatori di mininet. Per semplificare lo sviluppo ed il testing del codice, è stata impostata una regola di **port forwarding** per esporre la porta 22 del protocollo **SSH** sull'interfaccia di rete della macchina host, in maniera tale da poter trasferire i file di codice alla VM in SFTP ed eseguirli mediante il terminale locale, piuttosto che il TTY della VM.

Infine, con lo scopo di agevolare lo sviluppo in parallelo della soluzione, il codice realizzato è stato salvato in branch dedicati per le due fasi di sviluppo successive e soltanto dopo unito al resto del codice del progetto per eseguire il server HTTP in Flask sugli host mininet.

Topologia di Rete

L'intera topologia della rete è memorizzata all'interno del file `topology.py`.
Per ciascuna delle quattro subnet in cui saranno presenti dei dispositivi host, è stata adottata la seguente **convenzione** per l'assegnazione **statica** degli indirizzi IP: il gateway di ciascuna subnet avrà, come proprio indirizzo, **l'ultimo** IP assegnabile del proprio spazio di indirizzamento, il quale precede l'indirizzo di broadcast locale; i dispositivi host, invece, hanno ricevuto indirizzi progressivi a partire dal **primo** assegnabile all'interno della subnet in questione. Ciascuna di queste quattro subnet presenta una subnet mask /24; pertanto, per ognuna di esse, l'indirizzo del gateway termina con il valore 254, mentre l'indirizzo della subnet con il valore 0.

Per le subnet aventi subnet mask /30, dedicate alle connessioni dirette fra coppie dei cinque switch, è stata adottata una **convenzione differente**: gli unici due indirizzi IP assegnabili al loro interno sono stati allocati ai due switch rispettando l'ordine delle loro etichette (e.g. nella subnet 200.0.0.0/30, dedicata al collegamento fra switch 1 e switch 3, il primo indirizzo è stato assegnato allo switch 1, mentre il secondo allo switch 3).

Dopodiché, sono stati introdotti tutti i **link *interni*** a ogni subnet, collegando ciascun host al loro switch corrispondente e specificando i parametri di bandwidth (`bw`) e ritardo (`delay`). Inoltre, in preparazione alla fase successiva, è stato inserito nella tabella di routing di ciascun host l'indirizzo IP dello switch che ne gestisce la subnet come default gateway per l'indirect forwarding.

Per completare la topologia di rete, sono stati introdotti i **link che collegano** tra loro **i cinque switch**, privi di alcuna informazione nelle loro tabelle di routing; con questa configurazione, gli host appartenenti alla stessa subnet sono già in grado di comunicare, **ma non vi è reperibilità fra subnet diverse**.
Per ovviare a una limitazione dei controller OpenFlow in ryu, la quale verrà discussa in maggiore dettaglio nel paragrafo successivo, la classe `ProjectTopology` memorizza in un proprio attributo `link_list` una lista di oggetti che descrivono la configurazione di ciascun link della rete, incluse informazioni sulle loro capacità di trasmissione e dei due dispositivi da esso collegati.

L'assegnazione di indirizzi IP ai cinque switch della rete è stata realizzata con **chiamate POST** in protocollo HTTP ad un server web distribuito insieme al codice sorgente di ryu che implementa un'API RESTful per la configurazione dei dispositivi di rete mediante OpenFlow.

Il codice che descrive la topologia della rete, pertanto, memorizza con delle proprie strutture dati (una lista di oggetti `SwitchConfig`) le configurazioni da trasmettere al controller remoto affinché sia quest'ultimo le propaghi automaticamente ai dispositivi in questione; per fare ciò, è stato necessario impostare ciascuno switch in **modalità bridge OpenFlow** mediante l'apposito comando (*`ovs-vsctl set Bridge {switch_name} protocols=OpenFlow13`*). Nel costruttore dell'istanza della rete mininet, è stato specificato l'uso del controller esterno; all'interno della VM mininet, è stato eseguito tale controller con la componente di server web in ascolto sulla porta 8080, alla quale il codice di topologia si collega per trasmettere le configurazioni desiderate.

Tale struttura **non è**, però, **sufficiente** a consentire la comunicazione fra diverse subnet, in quanto non sono state fornite configurazioni per abilitare il routing dei pacchetti tra di loro; al fine di verificare la corretta configurazione dell'attuale topologia, nonché l'assegnazione degli indirizzi IP a ciascuno switch, sono state introdotte, sempre mediante chiamate all'API RESTful del controller ryu, delle **rotte statiche** che collegassero lo Switch 1 allo Switch 3 e, pertanto, la Subnet 1 alla Subnet 3.

Controller OpenFlow

È necessario giustificare le scelte effettuate nello sviluppo e configurazione del controller OpenFlow in ryu utilizzato per configurare gli switch ed implementare l'algoritmo di Dijkstra per il routing dei pacchetti.

Mininet consente l'instaurazione di **collegamenti a più controller** OpenFlow remoti contemporaneamente, ma tale funzionalità dev'essere supportata correttamente dai controller in questione, i quali potrebbero entrare in conflitto, trasmettendo configurazioni contrastanti ai dispositivi ad essi collegati.

Con la topologia realizzata al paragrafo precedente, non è stato possibile configurare adeguatamente il controller suggerito per il routing con algoritmo di Dijkstra affinché funzionasse al di fianco dell'API RESTful: il controller Dijkstra era in grado di connettersi agli switch e ricevere le loro informazioni, ma **non poteva trasmettere correttamente le rotte** calcolate, **inducendo** invece **un traffico infinito di pacchetti ARP** fra gli switch, il quale generava una quantità estremamente elevata di log nella console dell'API RESTful. Siccome il codice del controller in questione era privo di documentazione alcuna, dopo aver valutato le possibili alternative, si è deciso di **riscrivere il controller** con tutte e sole le funzionalità richieste, con il completo controllo su ogni suo aspetto.

Tale controller è implementato all'interno del file `our_dijkstra.py`, contenente una classe che **estende** `RestRouterAPI` i.e. il controller messo a disposizione dal codice sorgente di ryu; pertanto, **implementa tutte le funzionalità già discusse** nel paragrafo precedente e può sostituire il server usato finora, eliminando la problematica dell'uso di più controller simultaneamente e le interazioni fra di essi.

Per determinare le rotte ottimali di ciascuno switch, sono state introdotte due nuove rotte all'API RESTful:

- **`/dijkstra`**, la quale riceve in POST un oggetto JSON contenente informazioni sulle reti verso le quali generare rotte, gli switch che le gestiscono ed i parametri di trasmissione (banda e ritardo) per ogni link della rete.
- **`/dijkstra_unit`**, la quale esegue lo stesso calcolo di `/dijkstra` assegnando costo **unitario** a ciascun link, senza tenere in considerazione alcun parametro di trasmissione.

Entrambe le rotte restituiscono un **array JSON** contenente le configurazioni da fornire in una successiva chiamata alle rotte `/router` dell'API per popolare le tabelle di routing di ogni switch con delle rotte statiche. Il formato esatto che deve avere la richiesta POST alle rotte Dijkstra è specificato all'interno del codice sorgente del controller.

È importante notare che, sebbene l'API `ryu.topology` consenta di ottenere diverse informazioni riguardo i dispositivi OpenFlow connessi al controller, **non vi è come risalire all'indirizzo IP** di un dispositivo datone l'indirizzo MAC; l'indirizzo IP, tuttavia, è necessario per la creazione di una rotta statica mediante API RESTful ed è per questo motivo che, nel codice che crea la topologia di rete, è necessario tenere traccia di queste informazioni, in quanto devono essere fornite al controller nella richiesta POST.

È possibile verificare le configurazioni fornite dal controller chiamando la rotta `/router/{switch_id}` per ogni switch gestito tramite OpenFlow ed ispezionare il JSON risultante.

Server HTTP per la misurazione del RTT

Il seguente servizio, sfruttando Flask come framework HTTP e SQLite come database, permette di misurare il Round Trip Time (**RTT**) verso un determinato host (IP o hostname) e di consultare i risultati sia in tempo reale sia in forma storica.

Motivazione dello stack utilizzato

Dal punto di vista dello stack di tecnologie utilizzato, la scelta di **Flask** è motivata dalla necessità di disporre rapidamente di un server HTTP leggero e di facile integrazione con Python. **SQLite** è stato preferito a un database più complesso in quanto è sufficiente a gestire in modo efficiente e semplice i dati delle misure di RTT, garantendo immediatezza nell'accesso ai risultati. Il servizio fa inoltre uso del comando ping tramite chiamate con **subprocess.run**, evitando la necessità di eseguire codice con privilegi elevati per accedere a socket raw (ovvero interfacciarsi direttamente con il livello 2 o 4 della pila ISO OSI). Grazie a questa soluzione, ogni volta che l'utente avvia una nuova misurazione, un **thread dedicato** si occupa di eseguire ripetuti ping verso l'host scelto, con una frequenza configurabile. I risultati vengono salvati in tempo reale nel database, insieme ai relativi metadati (timestamp, indirizzo di destinazione, indirizzo sorgente, valore di RTT, durata totale del test).

Architettura

Per quanto riguarda l'architettura della soluzione, il server HTTP espone diverse route.

- La **rotta principale** (/) presenta una semplice pagina con un form dove l'utente può specificare l'host di destinazione e la durata della misura.
- L'avvio di una nuova misurazione (**/start_measurement**) crea e lancia un thread dedicato che gestisce la ripetizione dei ping
- La rotta **/get_current_data** restituisce, in formato JSON, i dati dell'attuale sessione di misura.
- La rotta **/get_history_data** è atta a consultare i risultati storici , essa filtra le misure registrate nel database in base all'host di destinazione, distinguendo le misure precedenti da quelle ancora in corso.
- Per consultare l'intero storico di un host è inoltre disponibile una pagina (**/show_history**) che, tramite la rotta /get_host_history_data, ritorna tutte le misure effettuate verso quello specifico indirizzo, e consente di visualizzarle in modo più dettagliato.

Calcolo del RTT

L'applicazione, per ogni misura avviata, calcola ciclicamente il **tempo di risposta** di un pacchetto **ICMP Echo Request/Reply**, raccogliendone i valori in un thread separato: per ogni risposta ricevuta, l'applicazione calcola quanto tempo è passato dall'invio del pacchetto Echo Request fino al momento in cui viene ricevuta la risposta Echo Reply. Questo intervallo di tempo è il RTT del pacchetto e fornisce una misura della latenza tra il dispositivo che invia il ping e il dispositivo di destinazione.

Questi dati sono **memorizzati su SQLite e, in parallelo**, forniti in tempo reale al browser per la tracciatura dei grafici. È sufficiente avviare il server in uno dei nodi della topologia per consentire la misurazione del RTT verso gli altri nodi, sia fornendo un IP, sia indicando un hostname riconosciuto in ambiente Mininet. Questo approccio rende immediata la verifica delle latenze nelle reti simulate e offre un supporto essenziale nello studio del comportamento e delle prestazioni di protocolli di routing o di configurazioni di rete sperimentali.

Focus sugli endpoint REST

Nonostante l'interfaccia web consenta una rappresentazione grafica di tali dati, l'attenzione principale è posta sulla gestione delle richieste HTTP e sulla logica di misura. Gli **endpoint REST** (Representational State Transfer), come *get_current_data* e *get_history_data*, rappresentano il **punto di incontro** tra l'utente e il motore di calcolo del RTT. Ciò facilita la raccolta ed esposizione di dati fondamentali per le analisi di rete, senza introdurre complessità eccessive per quanto riguarda il front end.

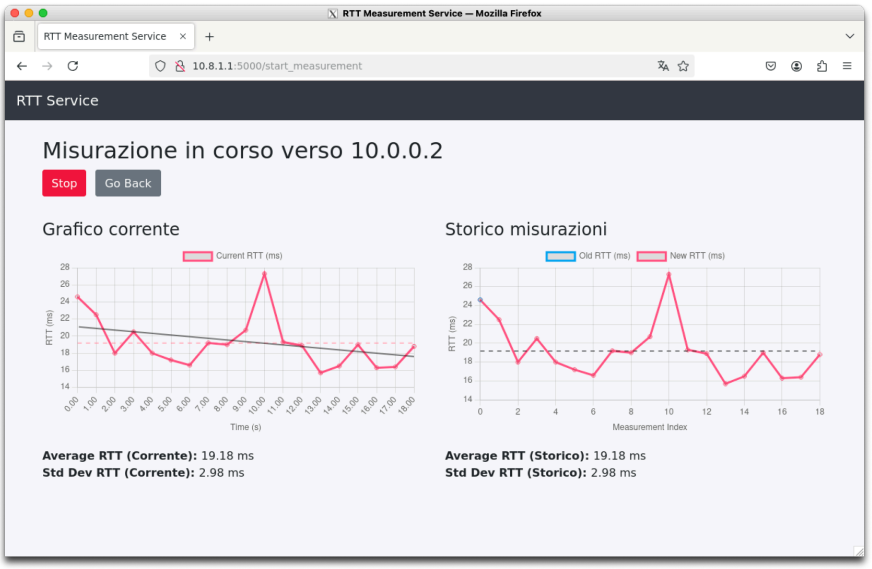
La scelta di adottare un thread separato per le misurazioni continua a mantenere reattivo il server Flask, che può così rispondere alle richieste di consultazione dei dati o di avvio/stop delle misurazioni senza blocchi.

Analisi della Soluzione Prodotta

Per verificare che ciascun host della rete riesca a comunicare con ogni altro, anche appartenendo a subnet distinte, è possibile eseguire il comando `pingall` della CLI di mininet:

```
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5
h2 -> h1 h3 h4 h5
h3 -> h1 h2 h4 h5
h4 -> h1 h2 h3 h5
h5 -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
mininet>
```

Per analizzare le caratteristiche della rete prodotta, è stato usato il Server HTTP realizzato con Flask descritto nella sezione precedente.



Il server è in esecuzione sull'host H5, accessibile al proprio indirizzo IP. L'host H4 è stato usato come client web per testare le funzionalità del sito.

In linea generale, i RTT misurati sperimentalmente risultano **inferiori** ai valori teorici dati dalla topologia della rete. Per esempio, l'RTT teorico calcolato seguendo la rotta ottimale fra H5 ed H4, assumendo una trasmissione di 1000B, è di circa 27ms, mentre il valore misurato sperimentalmente è poco più della metà: 15ms.
Questa osservazione è **vera per quasi ogni RTT osservato nell'ambiente mininet**; ipotizziamo sia dovuto a delle **inaccuratezze di emulazione** dell'architettura x86 su una macchina ARM, usata per eseguire la macchina virtuale mininet.

Per quanto riguarda le rotte assegnate dalla nostra implementazione dell'algoritmo di Dijkstra, abbiamo messo a confronto le rotte ottenute **assumendo costo unitario** per ogni hop con quelle ottenute calcolando il peso di ciascun link con la seguente funzione:

$$\frac{\alpha \cdot r}{\beta \cdot C}$$

Come punto di partenza, abbiamo assegnato alle costanti arbitrarie α e β il valore 1, valutando i risultati iniziali.

Effettuando una richiesta HTTP-GET alla rotta `/router/{switch_id}` del controller, è possibile **recuperare la configurazione** assegnata agli switch OpenFlow e mettere a confronto le informazioni di routing; così facendo, è possibile notare che le tabelle di routing ottenute prediligono già tutti i link con prestazioni di trasmissioni teoriche migliori, piuttosto che il cammino costituito dal minore numero di hop. Pertanto, abbiamo deciso di non manipolare ulteriormente i coefficienti.

La seguente tabella riporta i valori misurati effettuando un **ping tra** gli host **H4 ed H3** con le due possibili configurazioni. La dimensione del ping è stata della **massima quantità** consentita in byte, assegnata mediante parametro `-s` (65507B), con **trenta campionamenti** in entrambi i casi.

H4 Ping H3	**Valore Minimo**	**Valore Medio**	**Valore Massimo**	**Deviazione Std.**
Costo Unitario	1058ms	1061ms	1066ms	±1.8ms
Costo con Formula	1061ms	1068ms	1157ms	±16ms

È possibile notare che, sebbene il valore medio rimanga simile in entrambe le configurazioni, la **deviazione standard aumenta notevolmente** con la funzione di peso adottata; la rotta con un minore numero di hop risulta dunque **più stabile**.
La rotta ottenuta con questa funzione di peso richiede *tre* hop fra gli switch intermedi e non più soltanto due, sebbene vi sia un bottleneck meno significativo lungo tale rotta, dato dal link fra lo switch 3 e lo switch 4 con una capacità di trasmissione di 5Mbps, rispetto al link fra gli switch 1 e 3 con 1Mbps massimo.

Inoltre, per i test di banda effettuati con `iperf`, il vero bottleneck è stato il **link interno** che collega l'host H3 al proprio switch, con solo 1Mbps di banda; difatti, la capacità di banda calcolata in entrambe le configurazioni è di 1.5Mbps.
Ciò non è il caso, invece, per coppie di host come H1 e H4 che, in configurazione con rotte a costo unitario, degradano notevolmente in prestazioni, scendendo dai 6Mbps possibili a soli 2Mbps. Il miglioramento ottenuto applicando dunque lo stesso algoritmo con dei pesi più sofisticati spesso varia da host in host; **non vi è alcun fattore di miglioramento costante omogeneo** su tutta la rete.

Considerazioni finali

Il progetto realizzato ha offerto un'ottima occasione per esplorare sia gli aspetti teorici che pratici delle reti **SDN**. L'implementazione di una rete complessa mediante l'uso di Mininet e il controller Ryu ha permesso di osservare direttamente il comportamento delle reti in condizioni simulate, **validando e talvolta mettendo in discussione** le aspettative teoriche sui tempi di Round Trip Time e sulla gestione dei flussi di dati.

Il progetto ha affrontato alcune sfide significative relative alla **compatibilità** e alla **manutenzione** del software impiegato. Le limitazioni incontrate con Mininet e alcune componenti di Ryu hanno richiesto adattamenti significativi del codice e soluzioni creative per superare le barriere tecniche.

In conclusione, il progetto ha aperto la strada a numerose possibilità future di ricerca, specialmente per quanto riguarda l'**ottimizzazione** di algoritmi di **shortest path finding** come Dijkstra. Estendere questo lavoro per esplorare nuove configurazioni dell'algoritmo e/o altre configurazioni di rete rappresenterebbe un progetto stimolante nel dominio delle reti SDN.