UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Network and Cloud Infrastructures

SDN Project Work: Network Slicing

Candidato:

Vincenzo D'Angelo M63001595

Professori:

Giorgio Ventre

Roberto Canonico

Alessio Botta

Anno Accademico 2024/2025

Indice

1 Introduzione **1**

2 Definizione della topologia di rete **2**

3 Topology Slicing **4**
3.1 Implementazione 4
3.2 Verifica della Topology Slicing 7
3.3 Dashboard 10

4 Service Slicing **15**
4.1 Implementazione 15
4.2 Verifica del Service Slicing 18

5 Dynamic Slicing **20**
5.1 Implementazione 20
5.2 Verifica del Dynamic Slicing 21

1 Introduzione

Lo scopo di questo progetto è implementare il *network slicing* in un ambiente SDN (*Software Defined Networking*) [1] utilizzando Mininet [2] come emulatore di rete e Ryu [3] come controller.

SDN è un paradigma di rete che separa il piano di controllo dal piano dati, consentendo una gestione centralizzata, programmabile e dinamica delle risorse di rete. Questo approccio permette di configurare i flussi di traffico in modo flessibile, migliorando l'efficienza della rete e facilitando l'implementazione di politiche avanzate come il network slicing.

L'implementazione si concentrerà su due aspetti principali:

1. **Topology Slicing** – Limitare i percorsi di comunicazione tra specifici host per garantire l'isolamento dei flussi e ridurre interferenze tra slice.
2. **Service Slicing** – Dare priorità a determinati tipi di traffico, ad esempio video o dati critici, rispetto ad altri, garantendo la qualità del servizio (QoS).

Questi concetti sono comunemente adottati per ottenere isolamento del traffico, ottimizzare l'utilizzo delle risorse di rete e supportare scenari complessi come reti multi-tenant o servizi con requisiti diversi di latenza e larghezza di banda.

2 Definizione della topologia di rete

La topologia è stata realizzata in Mininet ed è definita nel file `topology.py`. Essa include quattro host (`H1`–`H4`) e quattro switch (`S1`–`S4`).

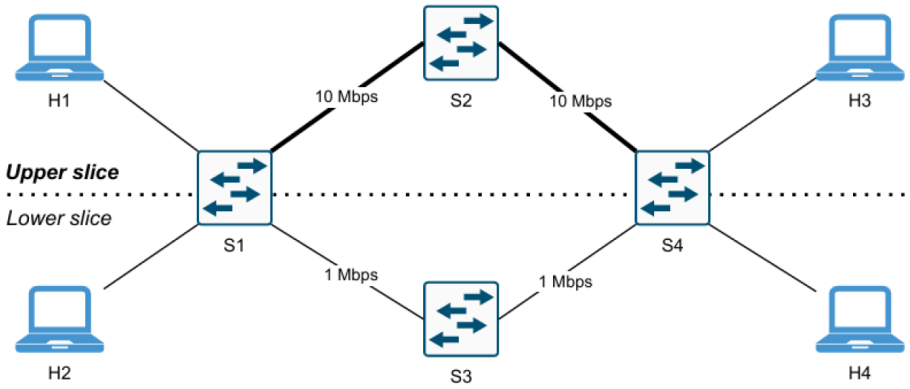


Figura 1: Topologia di rete

Ogni host è identificato da un MAC e un IP univoci:

```
self.h1 = self.net.addHost('h1', mac='00:00:00:00:00:01',
↪  ip='10.0.0.1')
self.h2 = self.net.addHost('h2', mac='00:00:00:00:00:02',
↪  ip='10.0.0.2')
self.h3 = self.net.addHost('h3', mac='00:00:00:00:00:03',
↪  ip='10.0.0.3')
self.h4 = self.net.addHost('h4', mac='00:00:00:00:00:04',
↪  ip='10.0.0.4')
```

Codice 1: Definizione degli host

```
self.s1 = self.net.addSwitch('s1')
self.s2 = self.net.addSwitch('s2')
self.s3 = self.net.addSwitch('s3')
self.s4 = self.net.addSwitch('s4')
```

Codice 2: Definizione degli switch

Gli switch sono collegati tramite link con parametri di larghezza di banda e latenza specifici per simulare differenti caratteristiche di rete:

- Slice "Up": H1 → s1 → s2 → s4 → H3, con link a 10 Mbps di banda
- Slice "Down": H2 → s1 → s3 → s4 → H4, con link a 1 Mbps di banda

```
self.net.addLink(self.h1, self.s1, delay='0.01ms', port1=1, port2=1)
self.net.addLink(self.h2, self.s1, delay='0.01ms', port1=1, port2=2)
self.net.addLink(self.h3, self.s4, delay='0.01ms', port1=1, port2=3)
self.net.addLink(self.h4, self.s4, delay='0.01ms', port1=1, port2=4)
```

Codice 3: Collegamenti host-switch

```
self.net.addLink(self.s1, self.s2, bw=10, delay='0.025ms', port1=3,
↪  port2=1)
self.net.addLink(self.s2, self.s4, bw=10, delay='0.025ms', port1=2,
↪  port2=1)

self.net.addLink(self.s1, self.s3, bw=1, delay='0.025ms', port1=4,
↪  port2=1)
self.net.addLink(self.s3, self.s4, bw=1, delay='0.025ms', port1=2,
↪  port2=2)
```

Codice 4: Collegamenti switch-switch

Tutti gli switch sono gestiti da un controller remoto, eseguito esternamente alla rete Mininet, che permette di programmare dinamicamente i flussi e separare i due slice.

```
self.net = Mininet(controller=RemoteController, link=TCLink)
c1 = self.net.addController('c1', controller=RemoteController,
↪  port=6653, ip='127.0.0.1')
c1.start()
```

Codice 5: Definizione del remote controller

L'uso di un Remote Controller permette di gestire le politiche di instradamento in modo centralizzato tramite Ryu.

La seguente tabella riassume la mappatura dei porti di ciascuno switch nella topologia:

Switch	Porta	Collegamento	Slice
S1	1	H1	Up
S1	2	H2	Down
S1	3	S2	Up
S1	4	S3	Down
S2	1	S1	Up
S2	2	S4	Up
S3	1	S1	Down
S3	2	S4	Down
S4	1	S2	Up
S4	2	S3	Down
S4	3	H3	Up
S4	4	H4	Down

Tabella 1: Mappatura dei porti degli switch

3 Topology Slicing

L'obiettivo di questa sezione è implementare una rete suddivisa in *slice* distinti, in modo che la comunicazione tra gli host avvenga esclusivamente attraverso percorsi predefiniti. In particolare:

- l'host `H1` deve comunicare unicamente con l'host `H3` attraverso il percorso superiore (upper slice);
- l'host `H2` deve comunicare unicamente con l'host `H4` attraverso il percorso inferiore (lower slice).

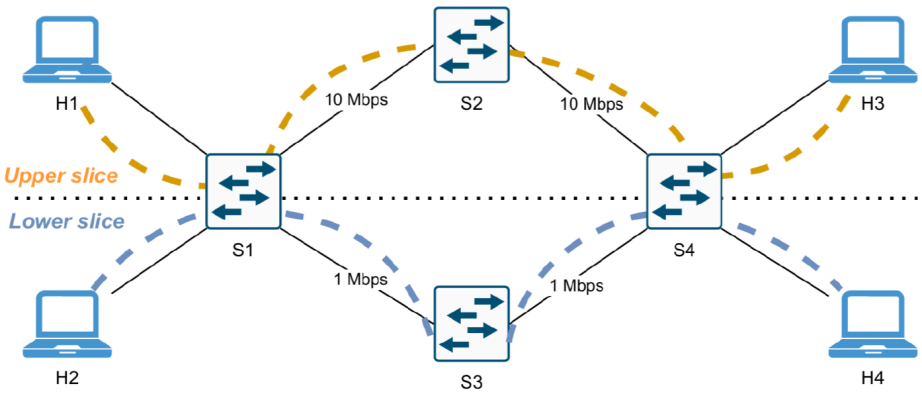


Figura 2: Topology Slicing

Questa segmentazione consente di ottenere un isolamento logico tra flussi diversi pur condividendo la stessa infrastruttura fisica.

3.1 Implementazione

Per imporre la separazione dei percorsi è stato sviluppato il modulo *Ryu* `controller_topo.py`.
In particolare, sono state installate regole di forwarding basate sugli indirizzi MAC degli host. Tali regole consentono:

- il traffico tra `H1` e `H3` solo tramite l'upper slice;
- il traffico tra `H2` e `H4` solo tramite il lower slice.

Le regole vengono inserite in ciascun datapath (switch) all'avvio, sfruttando il meccanismo di gestione degli eventi *EventOFPSwitchFeatures*. Per la gestione degli ARP, sono state aggiunte specifiche entry di flusso, così da garantire la risoluzione degli indirizzi solo all'interno delle slice consentite.

Switch S1

Lo switch `S1` gestisce entrambi gli slice in entrambe le direzioni e garantisce che i flussi siano correttamente separati lungo i rispettivi percorsi senza interferenze.

```
if dpid == 1:
            # h1->h3 slice
            self.add_mac_flow(datapath, self.h1, self.h3, out_port=3)
            self.add_mac_flow(datapath, self.h3, self.h1, out_port=1)
            self.add_arp_flow(datapath, self.h1, 3)
            self.add_arp_flow(datapath, self.h3, 1)

            # h2->h4 slice
            self.add_mac_flow(datapath, self.h2, self.h4, out_port=4)
            self.add_mac_flow(datapath, self.h4, self.h2, out_port=2)
            self.add_arp_flow(datapath, self.h2, 4)
            self.add_arp_flow(datapath, self.h4, 2)
```

Codice 6: Gestione delle regole di forwarding per lo switch `S1`

Funge da punto di ingresso per i pacchetti provenienti da `H1` e `H2` e da punto di uscita per i pacchetti provenienti da `H3` e `H4`.

- Slice `H1`↔`H3`
  - i pacchetti da `H1` a `H3` vengono inviati sulla porta 3 (verso `S2`);
  - i pacchetti da `H3` a `H1` vengono inoltrati sulla porta 1 (verso `H1`);
- Slice `H2`↔`H4`
  - i pacchetti da `H2` a `H4` vengono inviati sulla porta 4 (verso `S3`);
  - i pacchetti da `H4` a `H2` vengono inoltrati sulla porta 2 (verso `H2`);

Switch S2

Lo switch `S2` è il nodo intermedio dello slice superiore:

```
elif dpid == 2:
            self.add_mac_flow(datapath, self.h1, self.h3, out_port=2)
            self.add_mac_flow(datapath, self.h3, self.h1, out_port=1)
            self.add_arp_flow(datapath, self.h1, 2)
            self.add_arp_flow(datapath, self.h3, 1)
```

Codice 7: Gestione delle regole di forwarding per lo switch `S2`

Gestisce esclusivamente il traffico `H1` ↔ `H3`:

- i pacchetti da `H1` a `H3` vengono inviati sulla porta 2 (verso `S4`);
- i pacchetti da `H3` a `H1` vengono inoltrati sulla porta 1 (verso `S1`);

Switch S3

Lo switch S3 è il nodo intermedio dello slice inferiore:

```
elif dpid == 3:
            self.add_mac_flow(datapath, self.h2, self.h4, out_port=2)
            self.add_mac_flow(datapath, self.h4, self.h2, out_port=1)
            self.add_arp_flow(datapath, self.h2, 2)
            self.add_arp_flow(datapath, self.h4, 1)
```

Codice 8: Gestione delle regole di forwarding per lo switch S3

Gestisce esclusivamente il traffico H2 ↔ H4:

- i pacchetti da H2 a H4 vengono inviati sulla porta 2 (verso S4);
- i pacchetti da H4 a H2 vengono inoltrati sulla porta 1 (verso S1);

Switch S4

Lo switch S4 gestisce entrambi gli slice in entrambe le direzioni e garantisce che i flussi siano correttamente separati lungo i rispettivi percorsi senza interferenze.

```
elif dpid == 4:
            # h1->h3 slice
            self.add_mac_flow(datapath, self.h1, self.h3, out_port=3)
            self.add_mac_flow(datapath, self.h3, self.h1, out_port=1)
            self.add_arp_flow(datapath, self.h1, 3)
            self.add_arp_flow(datapath, self.h3, 1)

            # h2->h4 slice
            self.add_mac_flow(datapath, self.h2, self.h4, out_port=4)
            self.add_mac_flow(datapath, self.h4, self.h2, out_port=2)
            self.add_arp_flow(datapath, self.h2, 4)
            self.add_arp_flow(datapath, self.h4, 2)
```

Codice 9: Gestione delle regole di forwarding per lo switch S4

Funge da punto di ingresso per i pacchetti provenienti da H3 e H4 e da punto di uscita per i pacchetti provenienti da H1 e H2.

- Slice H1↔H3
  - i pacchetti da H1 a H3 vengono inviati sulla porta 3 (verso H3);
  - i pacchetti da H3 a H1 vengono inoltrati sulla porta 1 (verso S2);
- Slice H2↔H4
  - i pacchetti da H2 a H4 vengono inviati sulla porta 4 (verso H4);
  - i pacchetti da H4 a H2 vengono inoltrati sulla porta 2 (verso S3);

3.2 Verifica della Topology Slicing

Per controllare il corretto funzionamento della rete, è stato implementato un modulo di verifica dedicato all'interno del file `test_topology.py`. Questo script utilizza la topologia definita in topology.py tramite la classe Environment e svolge i seguenti compiti:

- **Avvio della rete**: Lo script inizializza l'ambiente Mininet e avvia la rete con il controller remoto e gli switch configurati per il topology slicing.
- **Identificazione degli host**: Vengono recuperati gli oggetti host H1, H2, H3 e H4 per eseguire i test di connettività.
- **Definizione dei test**: La verifica è strutturata tramite una lista di test in cui si specificano la sorgente, la destinazione e se la comunicazione dovrebbe riuscire. In particolare:
  - Comunicazioni consentite:
    - H1 $\leftrightarrow$ H3 (upper slice)
    - H2 $\leftrightarrow$ H4 (lower slice)
  - Comunicazioni non consentite:
    - H1 $\leftrightarrow$ H2
    - H1 $\leftrightarrow$ H4
    - H2 $\leftrightarrow$ H3
    - H3 $\leftrightarrow$ H4
- **Esecuzione dei test**: Lo script esegue innanzitutto un comando `pingAll` per verificare rapidamente la connettività generale. Successivamente, per ciascuna coppia di host, viene eseguito un ping di 2 pacchetti e i risultati vengono stampati in console indicando se la comunicazione è avvenuta correttamente o se è stata bloccata.

```
info('Pingall della rete:\n')
    net.pingAll()

for src, dst, should_work in tests:
    info(f"\n[TEST] Ping da {src.name} a {dst.name}(deve
    ↪  {'funzionare' if should_work else 'FALLIRE'}):\n")
    result = src.cmd(f'ping -c 2 {dst.IP()}')
    info(result)
```

Codice 10: Esecuzione dei test

- **Pulizia della rete**: Al termine dei test, lo script ferma la rete e pulisce eventuali residui Mininet, garantendo un ambiente pronto per successive esecuzioni.

Questo approccio consente di verificare in modo sistematico che ogni flusso dati sia instradato lungo lo slice corretto, confermando la corretta separazione dei percorsi tra gli host H1-H3 e H2-H4.

Di seguito sono mostrati alcuni esempi rappresentativi dei risultati ottenuti.

```
Pingall della rete:
*** Ping: testing ping reachability
h1 -> X h3 X
h2 -> X X h4
h3 -> h1 X X
h4 -> X h2 X
```

Figura 3: Verifica connettività generale con `pingAll()`

```
[TEST] Ping da h1 a h3 (deve funzionare):
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=1.54 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=1.08 ms

--- 10.0.0.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 1.082/1.309/1.537/0.227 ms
```

Figura 4: Verifica connettività H1 → H3

```
[TEST] Ping da h2 a h3 (deve FALLIRE):
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
From 10.0.0.2 icmp_seq=1 Destination Host Unreachable
From 10.0.0.2 icmp_seq=2 Destination Host Unreachable

--- 10.0.0.3 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1024ms
pipe 2
```

Figura 5: Verifica connettività H2 → H3

Successivamente, è stato eseguito un ping di 5 pacchetti da `H1` verso `H3` utilizzando il seguente comando da terminale:

```
mininet> h1 ping -c 5 10.0.0.3
```

In parallelo, è stato avviato `Wireshark` direttamente sull'host `H1`, al fine di catturare e analizzare il traffico generato e ricevuto da tale nodo.

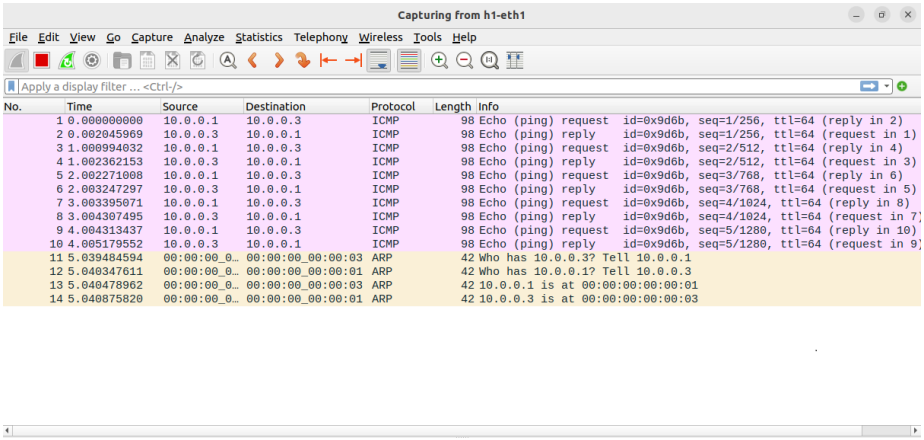


Figura 6: Verifica connettività H1 → H3

Dai risultati in Figura 6, si può osservare la presenza di 4 pacchetti ARP. Questo avviene perché, prima che `H1` possa inviare pacchetti ICMP a `H3`, deve risolvere l'indirizzo MAC della destinazione. In particolare, vengono generati due pacchetti ARP iniziali per la richiesta e altri due come risposta.
Il comando di ping produce 5 richieste ICMP ("*Echo Request*") e 5 risposte ICMP ("*Echo Reply*"), per un totale di 10 pacchetti ICMP catturati.

Lo stesso test è stato eseguito per verificare la connettività tra `H1` e `H2`, lanciando il seguente comando da terminale:

```
mininet> h1 ping -c 5 10.0.0.2
```

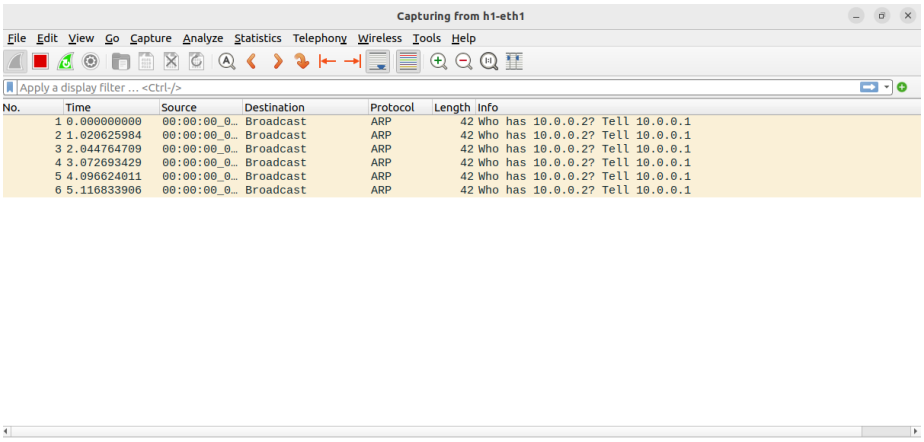


Figura 7: Verifica connettività H1 → H2

Prima di inviare i pacchetti ICMP, `H1` tenta di risolvere l'indirizzo MAC di `H2` tramite ARP. Come mostrato in Figura 7, i pacchetti ARP non ricevono

risposta, poiché il controller non ha installato regole per il flusso `H1→H2`.
Di conseguenza, `H1` non conosce l'indirizzo MAC di destinazione e non invia i pacchetti ICMP.

3.3 Dashboard

La dashboard è stata progettata per il monitoraggio in tempo reale della rete implementata. Consente di avere una panoramica immediata sull'utilizzo della banda di ciascuno switch e delle relative porte, fornendo strumenti pratici di analisi e debugging durante esperimenti di network slicing e traffic engineering.

La dashboard, sviluppata in HTML, CSS e JavaScript, si collega al controller Ryu tramite la porta 8765 e mantiene una sincronizzazione continua.

Per ogni switch e porta vengono mostrati:

- traffico ricevuto (`RX Bytes`),
- traffico trasmesso (`TX Bytes`),
- banda istantanea calcolata in Mbps,
- latenza stimata (in millisecondi).

Il controller Ryu raccoglie periodicamente le statistiche dagli switch OpenFlow attraverso i messaggi Port Stats, elabora i dati, calcola la banda istantanea e la latenza, e li invia tramite WebSocket ai client connessi.

```
def _monitor(self):
        while True:
            for dp in self.datapaths.values():
                self._request_port_stats(dp)
                self._send_echo(dp)
            hub.sleep(1)
            self._send_stats_to_ws()
```

Codice 11: Monitoraggio statistiche

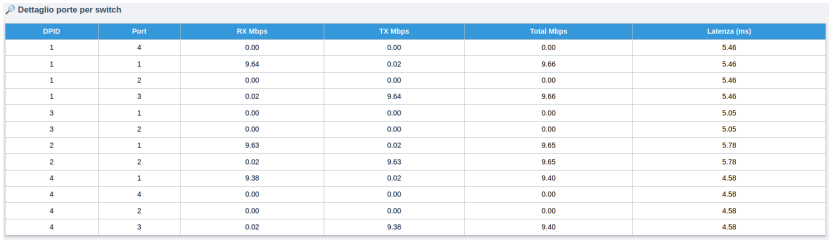
Il WebSocket Server garantisce la comunicazione real-time tra controller e frontend: ogni client riceve aggiornamenti automatici senza necessità di refresh.

```
def _start_ws_server(self):
        self.logger.info(f"Avvio WebSocket server
        ↪  ws://0.0.0.0:{WS_PORT}")
        self.server = WebsocketServer(port=WS_PORT, host='0.0.0.0')
        self.server.set_fn_new_client(self.new_client)
        self.server.set_fn_client_left(self.client_left)
        self.server.run_forever()
```

Codice 12: WebSocket Server

La dashboard presenta i dati in tre modalità complementari:

- **Tabella:** dettaglio completo per DPID, porta, RX/TX bytes, banda e latenza.



Dettaglio porte per switch

DPID	Port	RX Mbps	TX Mbps	Total Mbps	Latenza (ms)
1	4	0.00	0.00	0.00	5.46
1	1	9.64	0.02	9.66	5.46
1	2	0.00	0.00	0.00	5.46
1	3	0.02	9.64	9.66	5.46
3	1	0.00	0.00	0.00	5.05
3	2	0.00	0.00	0.00	5.05
2	1	9.63	0.02	9.65	5.78
2	2	0.02	9.63	9.65	5.78
4	1	9.38	0.02	9.40	4.58
4	4	0.00	0.00	0.00	4.58
4	2	0.00	0.00	0.00	4.58
4	3	0.02	9.38	9.40	4.58

Figura 8: Dettaglio completo delle statistiche di traffico per ciascuna porta degli switch OpenFlow misurate durante l'esperimento iperf.

- **Grafico a istogramma:** rappresentazione visiva dell'uso di banda per ogni porta, aggiornato in tempo reale.

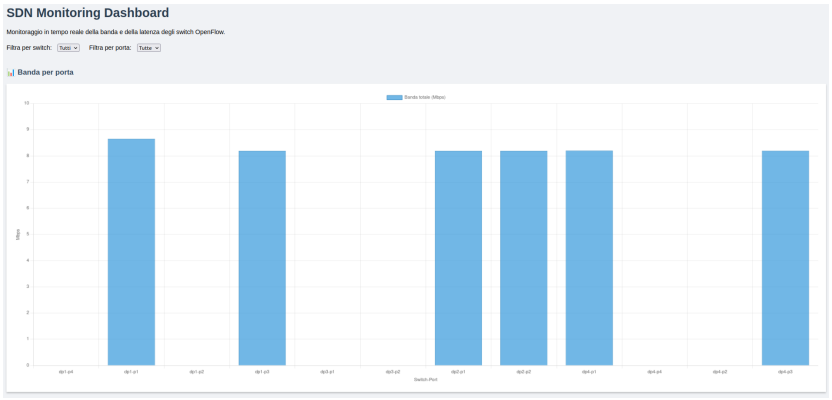


Figura 9: Istogramma della banda totale (Mbps) per porta degli switch OpenFlow, aggiornato in tempo reale tramite la dashboard.

- **Grafico a linee:** visualizzazione della latenza per ciascun switch nel tempo.

Figura 10: Grafico a linee della latenza (ms) per ciascun switch OpenFlow, aggiornato in tempo reale tramite la dashboard.

Inoltre, sono presenti filtri interattivi che consentono di selezionare switch e porte specifiche, permettendo un'analisi mirata e facilitando l'individuazione di eventuali colli di bottiglia o anomalie.

Banda La banda istantanea per ciascuna porta viene calcolata a partire dai byte trasmessi (`TX Bytes`) e ricevuti (`RX Bytes`) misurati tramite i messaggi `OFPPortStatsReply` di OpenFlow. Per ogni porta si memorizzano le statistiche correnti e quelle del ciclo precedente. La banda in Mbps viene calcolata come:

$$\text{RX Mbps} = \frac{(RX_{\text{attuale}} - RX_{\text{precedente}}) \times 8}{\Delta t \times 10^6}$$

$$\text{TX Mbps} = \frac{(TX_{\text{attuale}} - TX_{\text{precedente}}) \times 8}{\Delta t \times 10^6}$$

dove Δt è l'intervallo di tempo (in secondi) tra due misurazioni consecutive. La banda totale per la porta è quindi la somma di `RX Mbps` e `TX Mbps`.

```
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
    def port_stats_reply_handler(self, ev):
        dp = ev.msg.datapath
        now = time.time()
        for stat in ev.msg.body:
            port_no = stat.port_no
            if port_no < 1 or port_no > 65534:
                continue
            key = (dp.id, port_no)
            rx = stat.rx_bytes
            tx = stat.tx_bytes
            prev = self.prev_port_stats.get(key)
            if prev:
                dt = now - prev["time"]
                rx_mbps = (rx - prev["rx_bytes"]) * 8 / dt /
                ↪  1_000_000
                tx_mbps = (tx - prev["tx_bytes"]) * 8 / dt /
                ↪  1_000_000
            else:
                rx_mbps = tx_mbps = 0
            self.port_stats[key] = {"rx_mbps": rx_mbps, "tx_mbps":
            ↪  tx_mbps}
            self.prev_port_stats[key] = {"rx_bytes": rx, "tx_bytes":
            ↪  tx, "time": now}
```

Codice 13: Calcolo byte ricevuti e trasmessi

Latenza La latenza tra switch e host viene stimata usando pacchetti ARP o ICMP: il controller misura il tempo trascorso tra l'invio di una richiesta e la ricezione della risposta, aggiornando un valore medio nel tempo. Il valore di latenza per ciascun switch viene inviato periodicamente alla dashboard e visualizzato come grafico a linee, mostrando l'andamento temporale della latenza dei diversi switch.

```
@set_ev_cls(ofp_event.EventOFPEchoReply, MAIN_DISPATCHER)
    def echo_reply_handler(self, ev):
        now = time.time()
        dpid = ev.msg.datapath.id
        sent_time = self.echo_sent_time.get(dpid, None)
        if sent_time:
            latency = (now - sent_time) * 1000  # ms
            self.switch_latency[dpid] = latency
            self.logger.info(f"Latenza Switch {dpid}: {latency:.2f}
            ↪  ms")
```

Codice 14: Calcolo Latenza

Aggiornamento e invio dati Il controller Ryu raccoglie periodicamente le statistiche di banda e latenza, le elabora e costruisce un messaggio JSON contenente le informazioni principali per ciascuna porta, come identificativo dello switch, numero della porta, banda RX/TX e latenza.
Il messaggio viene inviato ai client WebSocket connessi alla dashboard, che aggiornano automaticamente grafici e tabella in tempo reale.

```
def _send_stats_to_ws(self):
        if not self.clients:
            return
        msg_data = [{"dpid": dpid, "port_no": p,
             "rx_mbps": stats["rx_mbps"],
             "tx_mbps": stats["tx_mbps"],
             "bandwidth_mbps": stats["rx_mbps"] + stats["tx_mbps"],
             "latency_ms": self.switch_latency.get(dpid, None)}   #
             ↪  aggiunto
            for (dpid, p), stats in self.port_stats.items()]
        msg = json.dumps({"type": "bandwidth_stats", "stats":
        ↪  msg_data})
        for client_id in list(self.clients):
            try:
                client_obj = next((c for c in self.server.clients if
                ↪  c['id'] == client_id), None)
                if client_obj:
                    self.server.send_message(client_obj, msg)
            except:
                self.clients.discard(client_id)
```

Codice 15: Invio dei dati ai client connessi alla dashboard

4 Service Slicing

Il *Service Slicing* implementato consente di separare e prioritizzare il traffico in base al tipo di servizio. In particolare, il traffico video (identificato come traffico UDP sulla porta 9999) viene instradato attraverso uno slice dedicato con priorità maggiore rispetto al traffico normale.

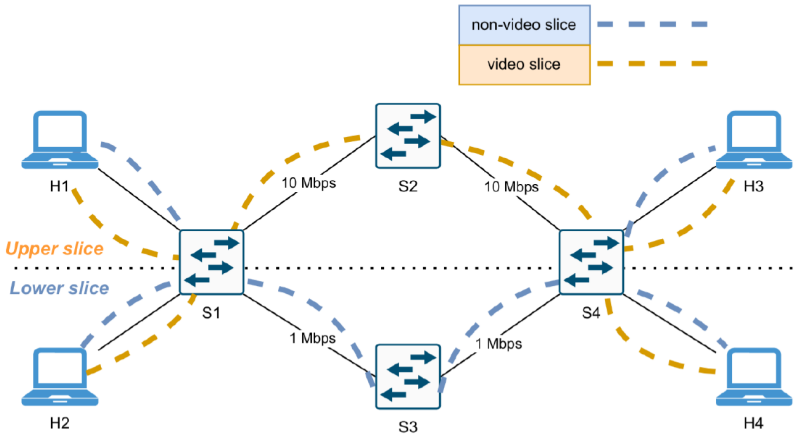


Figura 11: Service Slicing

Questo approccio permette di:

- Garantire **qualità del servizio (QoS)** per il traffico video.
- Consentire la **coesistenza di traffico differenti** sulla stessa infrastruttura di rete.
- Ottimizzare l'utilizzo dei collegamenti con larghezza di banda limitata.

4.1 Implementazione

Il controller che gestisce il *Service Slicing* è implementato nel file `controller_serv.py`. Analogamente al caso precedente, sono state installate regole statiche negli switch; questa volta, tuttavia, le regole riguardano esclusivamente il traffico UDP sulla porta 9999 (traffico video) e sono impostate con priorità elevata. In questo modo, i pacchetti UDP destinati alla porta 9999 vengono sempre processati prima del traffico normale, garantendo la separazione e la priorità del traffico video.

Switch S1

Lo switch `S1` inoltra i pacchetti UDP destinati a `H3` o `H4` verso lo slice superiore (tramite `S2`).

```
if dpid == 1:
            for host_port in [1, 2]:  # h1,h2
                match = parser.OFPMatch(
                    in_port=host_port,
                    eth_type=0x0800,
                    ip_proto=17,
                    udp_dst=UDP_PORT_STREAMING
                )
                actions = [parser.OFPActionOutput(3)]  # verso s2
                self.add_flow(datapath, priority=100, match=match,
                ↪  actions=actions)
```

Codice 16: Gestione delle regole di forwarding per lo switch `S1`

Switch S2

Lo switch `S2` instrada i pacchetti UDP ricevuti da `S1` verso `S4` e viceversa, garantendo che il traffico video percorra lo slice superiore

```
elif dpid == 2:
            match = parser.OFPMatch(
                in_port=1, # da s1
                eth_type=0x0800,
                ip_proto=17,
                udp_dst=UDP_PORT_STREAMING
            )
            actions = [parser.OFPActionOutput(2)]  # verso s4
            self.add_flow(datapath, priority=100, match=match,
            ↪  actions=actions)

            match = parser.OFPMatch(
                in_port=2, #da s4
                eth_type=0x0800,
                ip_proto=17,
                udp_dst=UDP_PORT_STREAMING
            )
            actions = [parser.OFPActionOutput(1)]  # verso s1
            self.add_flow(datapath, priority=100, match=match,
            ↪  actions=actions)
```

Codice 17: Gestione delle regole di forwarding per lo switch `S2`

Switch S3

Lo switch `S3` non inoltra pacchetti UDP, pertanto non viene installata alcuna regola.

Switch S4

Lo switch `S4` riceve i pacchetti UDP dagli host `H3` e `H4` e li inoltra verso `S2` quando destinati al traffico video.

```
elif dpid == 4:
            for host_port in [3, 4]:  # h3,h4
                match = parser.OFPMatch(
                    in_port=host_port,
                    eth_type=0x0800,
                    ip_proto=17,
                    udp_dst=UDP_PORT_STREAMING
                )
                actions = [parser.OFPActionOutput(1)]  # verso s2
                self.add_flow(datapath, priority=100, match=match,
                ↪  actions=actions)
```

Codice 18: Gestione delle regole di forwarding per lo switch `S4`

Tutti gli altri pacchetti vengono inviati al controller, che decide il percorso da utilizzare. Il controller applica un *flood controllato* sugli host e sui link definiti in base allo slice (superiore o inferiore), garantendo la comunicazione tra tutti gli host indipendentemente dal tipo di traffico.

Inizialmente, l'insieme dei link disponibili (`link_set`) su cui può essere inoltrato il traffico viene impostato sui link inferiori. Viene quindi effettuato un controllo sul tipo di traffico: se il pacchetto è UDP video, l'insieme dei link utilizzabili viene sostituito con quelli dello slice "up".

```
link_set = dw_links.get(dpid, set())
ip4 = pkt.get_protocol(ipv4.ipv4)

udp_video = ip4 and ip4.proto == 17 and udp_pkt and udp_pkt.dst_port
↪  == UDP_PORT_STREAMING

if udp_video:
            link_set = up_links.get(dpid, set())
```

Codice 19: scelta dei link da usare per il flood controllato

Se l'indirizzo MAC di destinazione è già noto, il pacchetto viene inoltrato direttamente sulla porta corrispondente. Se, invece, la destinazione non è ancora nota, il controller effettua un *flood controllato*:

- vengono considerate tutte le porte degli host collegate allo switch, ad eccezione della porta di ingresso;

- vengono aggiunti i link dello slice appropriato (normale o prioritario per UDP), sempre escludendo la porta di ingresso.

```
if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
            actions = [parser.OFPActionOutput(out_port)]
else:
            # flood controllato su host + link_set
            for p in host_ports.get(dpid, set()):
                if p != in_port:
                    actions.append(parser.OFPActionOutput(p))
            for p in link_set:
                if p != in_port:
                    actions.append(parser.OFPActionOutput(p))
```

Codice 20: Gestione dell'inoltro dei pacchetti

Questo meccanismo consente di instradare correttamente i pacchetti verso la destinazione, mantenendo l'isolamento dei flussi e la separazione tra slice.

Se il pacchetto ha una destinazione univoca (cioè viene inoltrato su una sola porta), viene creata una *flow entry dinamica* nello switch con priorità bassa. Questo permette ai pacchetti successivi con la stessa coppia MAC sorgente-destinazione di essere inoltrati direttamente dallo switch, senza coinvolgere nuovamente il controller, migliorando le prestazioni complessive della rete.

4.2 Verifica del Service Slicing

Per verificare il corretto funzionamento del *Service Slicing*, è necessario dimostrare che:

- Tutti gli host mantengono la capacità di comunicare tra loro, indipendentemente dallo slice utilizzato.
- Il traffico normale percorre lo slice inferiore senza interferire con il traffico video.

Come prima verifica, è stato eseguito un `pingall` per accertare che ogni host potesse raggiungere tutti gli altri, confermando la connettività di base della rete.

```
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

Figura 12: Verifica connettività generale con `pingAll()`

Successivamente, sono stati generati con `iperf` diversi flussi di traffico (UDP, TCP e ICMP) da `H1` verso `H3`. La cattura dei pacchetti effettuata con `Wireshark`, riportata in Figura 13, ha evidenziato la corretta separazione del

traffico: i pacchetti UDP sulla porta 9999 sono stati instradati sullo slice superiore (verso `s2`), mentre gli altri flussi hanno utilizzato lo slice inferiore (verso `s3`).

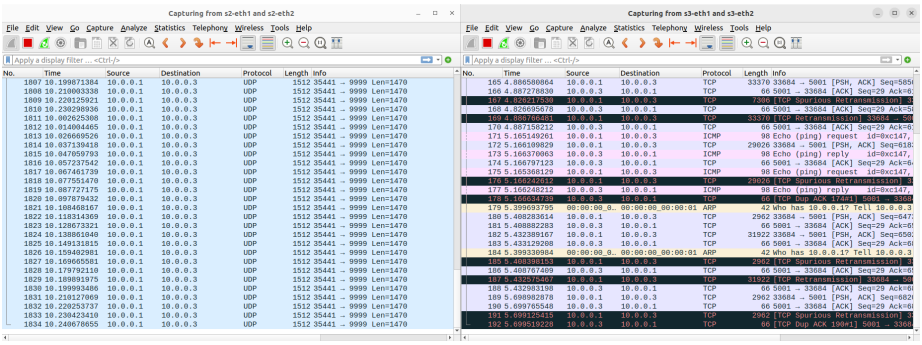


Figura 13: Verifica separazione del traffico

I test comparativi hanno confermato che i pacchetti video vengono instradati preferenzialmente attraverso lo slice superiore, sfruttando la maggiore larghezza di banda disponibile.

5 Dynamic Slicing

Il *Dynamic Service Slicing* estende il concetto di service slicing introdotto precedentemente. In questo caso, tutto il traffico normale può temporaneamente utilizzare lo slice video quando la banda è disponibile, mentre il traffico video mantiene sempre la priorità sullo slice superiore. Questo consente un uso più efficiente delle risorse di rete senza compromettere la qualità del traffico video.

5.1 Implementazione

Nel Dynamic Service Slicing, la logica del controller Ryu è stata estesa per introdurre un comportamento adattivo basato sulla disponibilità di banda. I pacchetti video continuano a utilizzare sempre lo *slice superiore*, mentre il traffico normale può temporaneamente instradarsi sullo *slice superiore* solo se la banda disponibile è inferiore a una soglia predefinita (`BANDWIDTH_THRESHOLD`). In caso contrario, il traffico normale viene instradato sullo *slice inferiore*.

```
if udp_video:
            link_set = up_links.get(dpid, set())
elif bw_bps < BANDWIDTH_THRESHOLD:
                link_set = up_links.get(dpid, set())
else:
            link_set = dw_links.get(dpid, set())
```

Codice 21: scelta dei link da usare per il flood controllato

Un aspetto cruciale del Dynamic Service Slicing riguarda il comportamento del *MAC learning* e del flooding controllato. Quando un pacchetto deve essere instradato sullo *slice inferiore*, il controller esegue un *flood controllato* sugli host e sui link dello slice inferiore. In questo caso non viene applicato il MAC learning, perché instradare il pacchetto direttamente su una porta fissa comprometterebbe la dinamicità dello slicing: un pacchetto passato una volta sullo slice inferiore verrebbe sempre indirizzato lì, anche quando la banda disponibile lo permetterebbe sullo slice superiore.
Al contrario, quando il pacchetto deve percorrere lo *slice superiore*, il flooding controllato viene applicato solo se il MAC di destinazione non è ancora noto, permettendo così l'installazione di regole dinamiche e la corretta separazione dei flussi.

```
if link_set == dw_links.get(dpid, set()):
            # flood controllato su host + link_set
            for p in host_ports.get(dpid, set()):
                if p != in_port:
                    actions.append(parser.OFPActionOutput(p))
            for p in link_set:
                if p != in_port:
                    actions.append(parser.OFPActionOutput(p))
else:
            # MAC learning normale
            if dst in self.mac_to_port[dpid]:
                out_port = self.mac_to_port[dpid][dst]
                actions = [parser.OFPActionOutput(out_port)]
            else:
                # se non conosciamo ancora il MAC, possiamo fare
                ↪  flood controllato anche sul link superiore
                for p in host_ports.get(dpid, set()):
                    if p != in_port:
                        actions.append(parser.OFPActionOutput(p))
                for p in link_set:
                    if p != in_port:
                       actions.append(parser.OFPActionOutput(p))
```

Codice 22: Gestione dell'inoltro dei pacchetti

Questa logica permette di utilizzare in modo flessibile la rete, dando priorità al traffico video ma sfruttando le risorse disponibili anche per il traffico normale. L'adattamento alla banda disponibile rende possibile instradare pacchetti normali sullo slice video senza interferire con i flussi prioritari.

5.2 Verifica del Dynamic Slicing

Per verificare il corretto funzionamento del *Dynamic Slicing*, è necessario dimostrare che:

- Tutti gli host mantengono la capacità di comunicare tra loro, indipendentemente dallo slice utilizzato.
- Quando è disponibile sufficiente banda, il traffico normale può percorrere lo slice superiore.
- Quando la banda disponibile è limitata, il traffico normale viene instradato sullo slice inferiore, garantendo priorità al traffico video.

Un primo test con `pingall` conferma la connettività di base della rete.

```
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

Figura 14: Verifica della connettività generale con `pingAll()`

Per verificare la logica dinamica, è stato generato un flusso TCP da `H1` verso `H3` con una banda impostata inferiore alla soglia definita per lo slicing dinamico. Come mostrato nella Figura 15, il traffico percorre lo slice superiore passando per lo switch `s2`, sfruttando la banda disponibile.

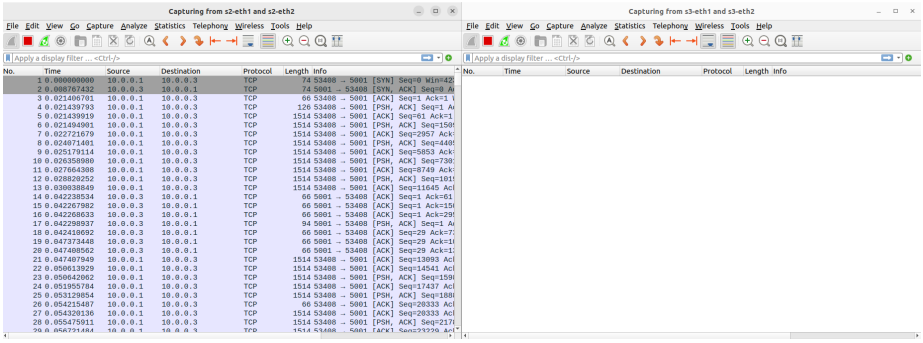


Figura 15: Cattura dei pacchetti su `s2` a seguito di un flusso TCP singolo

Infine, lo stesso traffico TCP è stato generato mentre un flusso UDP video sullo slice superiore occupava una banda superiore alla soglia stabilita. Come evidenziato dalle catture su `s2` e `s3` (Figura 16), il traffico TCP viene temporaneamente instradato sullo slice inferiore, confermando che il controller garantisce la priorità al traffico video senza interrompere la comunicazione degli altri flussi.

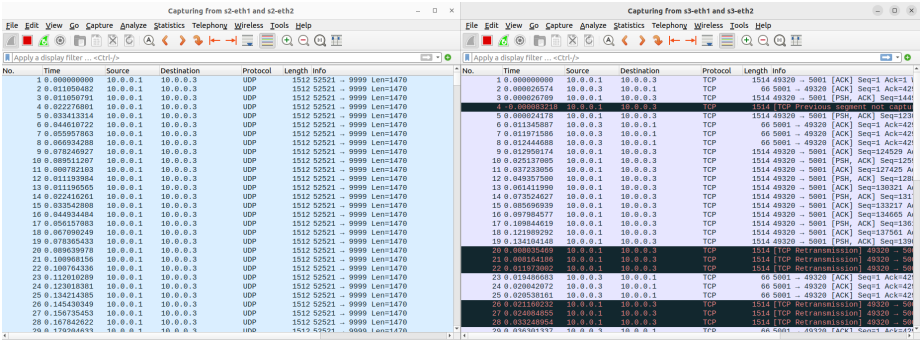


Figura 16: Instradamento dinamico del traffico TCP in presenza di traffico video sullo slice superiore

I test confermano la corretta separazione dei flussi e l'adattamento dinamico della rete. Il traffico video mantiene la priorità e la qualità anche in presenza di elevato traffico normale. L'utilizzo dinamico dello slice video consente di aumentare l'efficienza complessiva della rete, senza introdurre interferenze tra i diversi flussi.

Riferimenti

[1] Nick McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[2] Bob Lantz, Brandon Heller e Nick McKeown. “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks”. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IX)*. 2010.

[3] Ryu SDN Framework. *Ryu SDN Framework Official Documentation*. `https://ryu.readthedocs.io/en/latest/`. Accessed: 2025-09-18.