

# UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

---

*Corso di Laurea Magistrale in Ingegneria Informatica*

---



## ELABORATO DI NETWORKS AND CLOUD INFRASTRUCTURES

*PROJECT WORK SLICING*

a.a. 2024-25

Studente:

Fienga Luigi M63001733

<b>1. Topologia della Rete</b>	<b>3</b>
<b>2. Topology Slicing</b>	<b>5</b>
2.1 Controller Topology Slicing	5
2.2 Testing	10
<b>3. Service Slicing</b>	<b>14</b>
3.1 Controller Service Slicing	14
3.2 Testing	21
<b>4. Dynamic Slicing</b>	<b>23</b>
4.1 Controller Dynamic Slicing	23
4.2 Testing	29
<b>5. Codice</b>	<b>31</b>

# 1. Topologia della Rete

La rete implementata rispecchia quella che è stata presentata nella traccia del problema, ovvero da come si vede in *figura 1.1*, la parte superiore della rete è caratterizzata da una larghezza di banda superiore rispetto a quella inferiore, inoltre si evince una struttura topologicamente circolare.

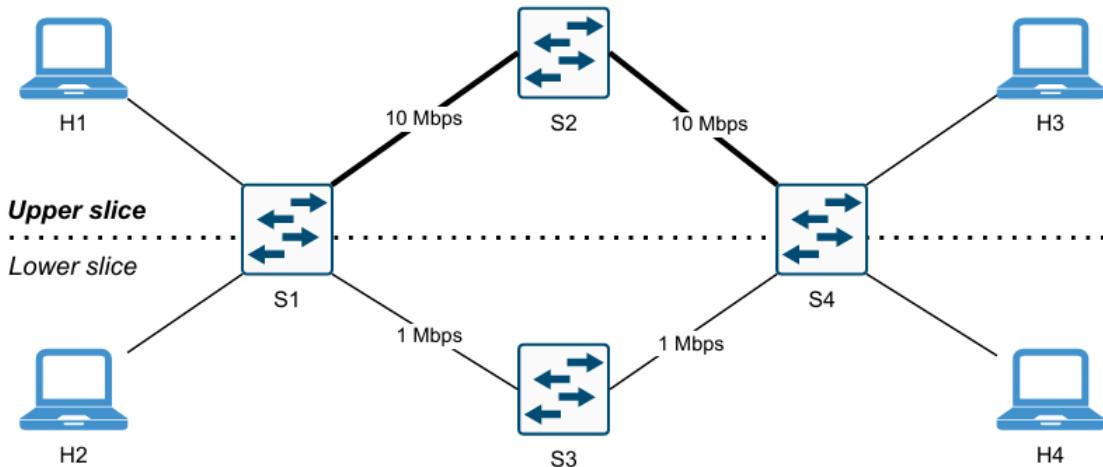


Figura 1.1 Rete Implementata

Gli switch sono controllati da uno **controller remoto**, il quale installerà le regole di instradamento opportune all'interno di ogni dispositivo, ovviamente sfruttando il protocollo **Openflow**.

```
class SliceTopo(Topo):
    def build(self):
        info('*** Creazione switch\n')
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')

        info('*** Creazione host\n')
        h1 = self.addHost('h1', ip='10.0.0.1/24', mac='00:00:00:00:01')
        h2 = self.addHost('h2', ip='10.0.0.2/24', mac='00:00:00:00:02')
        h3 = self.addHost('h3', ip='10.0.0.3/24', mac='00:00:00:00:03')
        h4 = self.addHost('h4', ip='10.0.0.4/24', mac='00:00:00:00:04')

        info('*** Creazione link\n')
        self.addLink(h1, s1)
        self.addLink(h2, s1)
        self.addLink(s1, s2, bw=10)
        self.addLink(s1, s3, bw=1)
        self.addLink(s2, s4, bw=10)
        self.addLink(s3, s4, bw=1)
        self.addLink(h3, s4)
        self.addLink(h4, s4)
```

Figura 1.2 Implementazione della topologia della rete

In *figura 1.2* è riportata la classe **SliceTopo** che implementa la topologia di rete vista prima ad alto livello.

Si sottolineano le seguenti caratteristiche:

- l'assegnazione degli indirizzi IP e MAC statici per ogni host
- la creazione degli switch collegati opportunamente rispetto alle bande viste precedentemente.

In particolare si ricapitolano di seguito le porte di ogni switch e i corrispondenti dispositivi collegati.

- Lo switch s1 è collegato con la seguente configurazione:
  - porta 1 con H1
  - porta 2 con H2
  - porta 3 con S2
  - porta 4 con S3
- Gli switch s2 e s3 sono collegati con la seguente configurazione:
  - porta 1 con S1
  - porta 2 con S4
- Lo switch s4 è collegato con la seguente configurazione:
  - porta 1 con S2
  - porta 2 con S3
  - porta 3 con H3
  - porta 4 con H4

```
def run():
    topo = SliceTopo()
    net = Mininet(
        topo=topo,
        controller=lambda name: RemoteController(name, ip='127.0.0.1', port=6653),
        switch=OVSSwitch,
        link=TCLink,
        autoSetMacs=False
    )

    info('*** Avvio rete\n')
    net.start()

    info('*** Test con ping all\n')
    net.pingAll()

    info('*** Avvio CLI\n')
    CLI(net)

    info('*** Arresto rete\n')
    net.stop()

if __name__ == '__main__':
    setLogLevel('info')
    run()
```

Figura 1.3 Avvio della topologia della rete

All'esecuzione del codice, la prima cosa è istanziare la topologia della rete definita con **Mininet**, poi viene effettuata la connessione al controller remoto all'indirizzo **127.0.0.1** sulla porta **6653**. Si specifica inoltre che lo switch sarà di tipo **OVSSwitch** (uno switch software

compatibile con OpenFlow) ed il link è di tipo **TCLink** che permette di simulare in maniera realistica le caratteristiche dei medesimi come la loro capacità.

La prima cosa che viene fatta dopo aver startato la rete è quella di verificare la raggiungibilità dei dispositivi mediante pingAll(), per poi avviare una CLI con mininet per interagire con la rete.

Si specifica infine che la seguente topologia è stata utilizzata per tutti i task del problema (Topology Slicing, Service Slicing e Dynamic Slicing).

## 2. Topology Slicing

Si ricorda che l'obiettivo di questo task è quello di garantire la comunicazione solo tra h1 <-> h3 e h2 <-> h4, dividendo la rete tra uno slice superiore che collega i dispositivi h1 e h3 tramite gli switch s1 <-> s2 <-> s4 e uno slice inferiore tramite gli switch s1 <-> s3 <-> s4 per garantire la connessione tra h2 e h4.

### 2.1 Controller Topology Slicing

Il seguente controller è stato implementato tramite **Ryu**, che utilizza il protocollo OpenFlow 1.3 per gestire la rete virtuale Mininet.

Riportiamo di seguito le **regole implementate** per ogni switch.

Su ogni switch è caricata la **regola di Default**:

Priorità	Match (Ingresso)	Match (Tipo)	Azione (Uscita)	Descrizione Flusso
0	<i>Qualsiasi</i>	<i>Qualsiasi</i>	DROP	Serve a garantire l'isolamento dello slice se quel pacchetto non rispecchia nessun'altra regola

La gestione del protocollo **ARP** è stata fatta in modo tale che il flooding del pacchetto sia fatto solo nello slice di appartenenza del dispositivo, ad esempio se h1 fa una richiesta ARP è perché presumibilmente deve conoscere l'indirizzo MAC di h3, quindi la sua richiesta viene inoltrata solo nello slice superiore.

### s1

Priorità	Match (Ingresso)	Match (Tipo)	Azione (Uscita)	Descrizione Flusso
100	Port 1 (da h1)	ARP (0x0806)	Output Port 3 (verso s2)	Upper Slice: h1 cerca h3
100	Port 2 (da h2)	ARP (0x0806)	Output Port 4 (verso s3)	Lower Slice: h2 cerca h4
100	Port 3 (da s2)	ARP (0x0806)	Output Port 1 (verso h1)	Upper Slice: Reply verso h1
100	Port 4 (da s3)	ARP (0x0806)	Output Port 2 (verso h2)	Lower Slice: Reply verso h2

### s2 e s3

Priorità	Match (Ingresso)	Match (Tipo)	Azione (Uscita)	Descrizione Flusso
100	<i>Qualsiasi</i>	ARP (0x0806)	FLOOD	Tunnel Upper Slice (Passa tutto l'ARP (0x0806))

#### s4

Priorità	Match (Ingresso)	Match (Tipo)	Azione (Uscita)	Descrizione Flusso
100	Port 1 (da s2)	ARP (0x0806)	Output Port 3 (verso h3)	Upper Slice: Request verso h3
100	Port 2 (da s3)	ARP (0x0806)	Output Port 4 (verso h4)	Lower Slice: Request verso h4
100	Port 3 (da h3)	ARP (0x0806)	Output Port 1 (verso s2)	Upper Slice: Reply da h3
100	Port 4 (da h4)	ARP (0x0806)	Output Port 2 (verso s3)	Lower Slice: Reply da h4

Le seguenti regole invece servono per il corretto **intradamento del traffico** rete in base alle esigenze descritte.

#### s1

Priorità	Match (Eth Src)	Match (Eth Dst)	Azione (Uscita)	Descrizione Flusso
200	h1	h3	Output Port 3 (verso s2)	Upper Slice: Andata
200	h2	h4	Output Port 4 (verso s3)	Lower Slice: Andata

200	h3	h1	Output Port 1 (verso h1)	Upper Slice: Ritorno
200	h4	h2	Output Port 2 (verso h2)	Lower Slice: Ritorno

s2

Priorità	Match (Eth Src)	Match (Eth Dst)	Azione (Uscita)	Descrizione Flusso
200	Qualsiasi	h3	Output Port 2 (verso s4)	Verso destra (Upper)
200	Qualsiasi	h1	Output Port 1 (verso s1)	Verso sinistra (Upper)

s3

Priorità	Match (Eth Src)	Match (Eth Dst)	Azione (Uscita)	Descrizione Flusso
200	Qualsiasi	h4	Output Port 2 (verso s4)	Verso destra (Lower)
200	Qualsiasi	h2	Output Port 1 (verso s1)	Verso sinistra (Lower)

#### s4

Priorità	Match (Eth Src)	Match (Eth Dst)	Azione (Uscita)	Descrizione Flusso
200	h1	h3	Output Port 3 (verso h3)	Upper Slice: Consegna a h3
200	h2	h4	Output Port 4 (verso h4)	Lower Slice: Consegna a h4
200	h3	h1	Output Port 1 (verso s2)	Upper Slice: Ritorno via s2
200	h4	h2	Output Port 2 (verso s3)	Lower Slice: Ritorno via s3

In merito alle priorità delle varie regole si specifica che, alla regola di default è stato assegnato il valore 0 in quanto qualunque pacchetto soddisfarebbe quella regola. Le regole ARP hanno avuto assegnazione 100 in quanto è considerato traffico di servizio e di ordine comune, a differenza delle regole specifiche di instradamento che hanno avuto priorità 200. Non verranno allegati tutti i codici di ogni singola regola implementata, però per fini dimostrativi, se ne riporta solo una, essendo l'implementazione delle altre pressoché simile a patto della variazione dei parametri.

```
# Upper Slice (ARP): h1 (port 1) -> s2 (port 3)
match = parser.OFPPMatch(in_port=1, eth_type=0x0806)
actions = [parser.OFPActionOutput(self.PORT_MAP[1]['s2'])]
self.add_flow(dp, 100, match, actions)
```

Figura 1.4 Regola per l'instradamento ARP da h1 verso s2

Per semplificare il lavoro di ricerca degli indirizzi MAC di ogni host e la conoscenza topologica della rete, sono stati realizzati due dizionari per tenere traccia di queste informazioni, presenti in *figura 1.5* dichiarati nella funzione di *\_\_init\_\_*.

```

# MAC Address degli Host come definito nel progetto
self.H = {
    'h1': '00:00:00:00:00:01',
    'h2': '00:00:00:00:00:02',
    'h3': '00:00:00:00:00:03',
    'h4': '00:00:00:00:00:04'
}

# MAPPATURA PORTE (Topology Map)
# Questa mappa riflette la topologia della rete
self.PORT_MAP = {
    # s1: 1->h1, 2->h2, 3->s2 (Upper), 4->s3 (Lower)
    1: {'h1': 1, 'h2': 2, 's2': 3, 's3': 4},
    # s2: 1->s1, 2->s4
    2: {'s1': 1, 's4': 2},
    # s3: 1->s1, 2->s4
    3: {'s1': 1, 's4': 2},
    # s4: 1->s2 (Upper), 2->s3 (Lower), 3->h3, 4->h4
    4: {'s2': 1, 's3': 2, 'h3': 3, 'h4': 4}
}

```

Figura 1.5 Dizionari

Per semplificare il lavoro di inserimento delle regole negli switch è stata implementata una funzione che ne semplifica il meccanismo, presente in *figura 1.6*.

```

def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPI_APPLY_ACTIONS, actions)]
    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                           match=match, instructions=inst)
    datapath.send_msg(mod)

```

Figura 1.6 add\_flow

Per quanto riguarda l'inserimento delle regole, questo viene fatto automaticamente dal controller non appena uno switch si connette al medesimo, ciò è possibile grazie alla funzione:

```

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev)

```

Come si evince dalla funzione il meccanismo di **automatismo** è possibile grazie all'attributo `@set_ev_cls`, il quale triggerà la funzione all'evento che uno switch ha risposto con le proprie feature memorizzate internamente.

## 2.2 Testing

Per prima cosa verifichiamo la raggiungibilità dei dispositivi, a tal fine si usa il comando `pingAll` che parte automaticamente all'avvio della rete.

```

test con ping all
*** Ping: testing ping reachability
h1 -> X h3 X
h2 -> X X h4
h3 -> h1 X X
h4 -> X h2 X
*** Results: 66% dropped (4/12 received)
*** Avvio CLI
*** Starting CLI:
mininet> █

```

Figura 1.7 pingAll

Vedendo la *figura 1.7* si può evincere che le proprietà di raggiungibilità per il seguente problema sono state rispettate correttamente.

Ciò che dobbiamo fare è controllare che effettivamente la comunicazione dei dispositivi avviene attraverso lo slicing corretto. Per fare ciò si è utilizzati *xterm* sulla CLI di mininet per aprire i terminali degli switch: s1, s2, s3 e s4. Sui quali si è utilizzati *tcpdump* per controllare il traffico in uscita su una data porta ethernet, in particolare:

- Per lo slice superiore si sono osservate le porte s1-eth3, s2-eth2, s3-eth2 e s4-eth3. Eseguendo *h1 ping -c4 h3* è stato prodotto il seguente risultato

```

"Node: s1" (root) - - x
listening on s1-eth3, link-type EN10MB (Ethernet), snapshot length 262144 bytes root@luigi-virtual-machine:/home/luigi/ncis_project/2# tcpdump -i s2-eth2 -nn
14:05:23.526038 IP fe80::e0d8:ceeff:feed:fb89 > ff02::2: ICMP6, router solicitation, use -v[v]... for full protocol decode
14:05:42.186735 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 1, length 64
14:05:42.187088 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 1, length 64
14:05:43.187673 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 2, length 64
14:05:43.187790 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 2, length 64
14:05:44.198815 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 3, length 64
14:05:44.200778 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 3, length 64
14:05:45.200416 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 4, length 64
14:05:45.200498 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 4, length 64
14:05:47.591906 ARP, Request who-has 10.0.0.3 tell 10.0.0.1, length 28
14:05:47.592036 ARP, Request who-has 10.0.0.1 tell 10.0.0.3, length 28
14:05:47.592105 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
14:05:47.592125 ARP, Reply 10.0.0.3 is-at 00:00:00:00:00:03, length 28
[...]
59
"Node: s2" (root) - - x
listening on s2-eth2, link-type EN10MB (Ethernet), snapshot length 262144 bytes root@luigi-virtual-machine:/home/luigi/ncis_project/2# tcpdump -i s2-eth2 -nn
14:05:42.186911 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 1, length 64
14:05:42.187049 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 1, length 64
14:05:43.187736 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 2, length 64
14:05:43.187785 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 2, length 64
14:05:44.198843 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 3, length 64
14:05:44.200767 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 3, length 64
14:05:45.200437 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 4, length 64
14:05:45.200493 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 4, length 64
14:05:47.591883 ARP, Request who-has 10.0.0.1 tell 10.0.0.3, length 28
14:05:47.592045 ARP, Request who-has 10.0.0.3 tell 10.0.0.1, length 28
14:05:47.592108 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
14:05:47.592124 ARP, Reply 10.0.0.3 is-at 00:00:00:00:00:03, length 28
[...]
"Node: s3" (root) - - x
listening on s3-eth2, link-type EN10MB (Ethernet), snapshot length 262144 bytes root@luigi-virtual-machine:/home/luigi/ncis_project/2# tcpdump -i s3-eth2 -nn
14:05:42.186958 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 1, length 64
14:05:42.187004 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 1, length 64
14:05:43.187745 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 2, length 64
14:05:43.187778 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 2, length 64
14:05:44.198853 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 3, length 64
14:05:44.198901 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 3, length 64
14:05:45.200446 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 4, length 64
14:05:45.200484 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 4, length 64
14:05:47.591042 ARP, Request who-has 10.0.0.1 tell 10.0.0.3, length 28
14:05:47.592116 ARP, Request who-has 10.0.0.3 tell 10.0.0.1, length 28
14:05:47.592122 ARP, Reply 10.0.0.3 is-at 00:00:00:00:00:03, length 28
14:05:47.592183 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
[...]
"Node: s4" (root) - - x
listening on s4-eth3, link-type EN10MB (Ethernet), snapshot length 262144 bytes root@luigi-virtual-machine:/home/luigi/ncis_project/2# tcpdump -i s4-eth3 -nn
14:05:42.186958 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 1, length 64
14:05:42.187004 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 1, length 64
14:05:43.187745 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 2, length 64
14:05:43.187778 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 2, length 64
14:05:44.198853 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 3, length 64
14:05:44.198901 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 3, length 64
14:05:45.200446 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 56324, seq 4, length 64
14:05:45.200484 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 56324, seq 4, length 64
14:05:47.591042 ARP, Request who-has 10.0.0.1 tell 10.0.0.3, length 28
14:05:47.592116 ARP, Request who-has 10.0.0.3 tell 10.0.0.1, length 28
14:05:47.592122 ARP, Reply 10.0.0.3 is-at 00:00:00:00:00:03, length 28
14:05:47.592183 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
[...]

```

Figura 1.8 switch h1 -> h3

Da come si evince in *figura 1.8* effettivamente il traffico di rete esegue lo slice superiore fino ad arrivare a destinazione h3.

```
mininet> h1 ping -c4 h3
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=3.60 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.185 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.224 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.229 ms

--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3055ms
rtt min/avg/max/mdev = 0.185/1.059/3.601/1.467 ms
mininet> h1 ping -c4 h3
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=1.34 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.237 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=2.15 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.206 ms

--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3014ms
rtt min/avg/max/mdev = 0.206/0.983/2.152/0.814 ms
mininet>
```

Figura 1.9 ping h1 verso h3

- Per lo slice inferiore si sono osservate le porte s1-eth4, s2-eth2, s3-eth2 e s4-eth4. Eseguendo *h2 ping -c4 h4* è stato prodotto il seguente risultato:

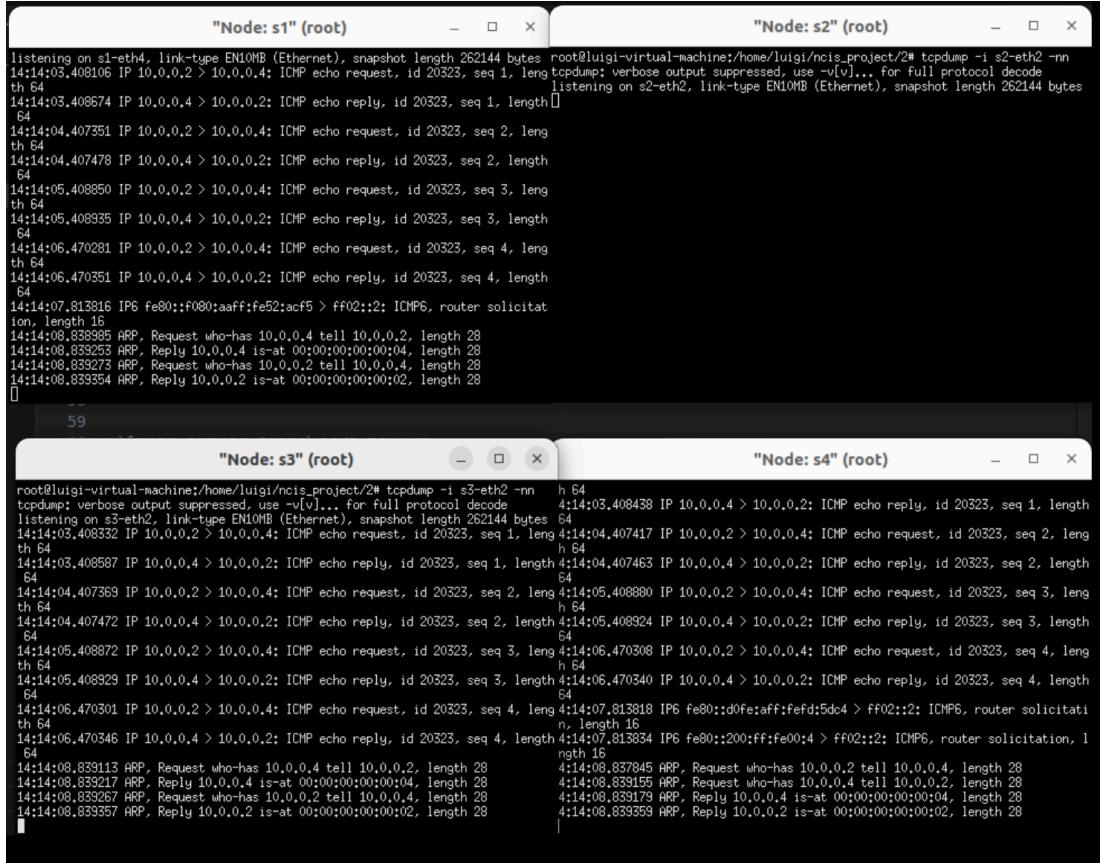


Figura 1.10 switch h2 -> h4

Emerge quindi in *figura 1.10* che effettivamente il traffico di rete passa per lo slice inferiore fino ad arrivare a destinazione h3.

```

mininet> h1 ping -c4 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time
pipe 4
mininet> h2 ping -c4 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=3.02 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.239 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.342 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.169 ms

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3064ms
rtt min/avg/max/mdev = 0.169/0.943/3.022/1.201 ms
mininet> 
```

Figura 1.11 ping h2 verso h4

In conclusione i requisiti di slicing sono stati soddisfatti, h1 può comunicare solo con h3 tramite lo slice superiore (e viceversa), lo stesso vale anche per h2 con h4. Si allegano infine solo per fine dimostrativo le immagini che non è possibile instaurare una connessione tra h1 e h4.

```

mininet> h1 ping -c4 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3077ms
pipe 4
mininet> 
```

Figura 1.12 ping h1 verso h4

```

"Node: s1" (root)           "Node: s2" (root)
root@luigi:~/virtual-machine:/home/luigi/ncis_project/2# tcpdump -i s1-eth3 -nn
tcpdump: verbose output suppressed, use -v[vv..] for full protocol decode
listening on s1-eth3, link-type EN10MB (Ethernet), snapshot length 262144 bytes
14:09:58.122407 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:09:58.150408 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:09:58.173857 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:09:58.173857 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:09:58.173857 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:10:01.221854 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:10:01.221854 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:10:02.248099 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:10:02.248099 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28

```

```

"Node: s3" (root)           "Node: s4" (root)
root@luigi:~/virtual-machine:/home/luigi/ncis_project/2# tcpdump -i s3-eth2 -nn
root@luigi:~/virtual-machine:/home/luigi/ncis_project/2# tcpdump -i s4-eth3 -nn
tcpdump: verbose output suppressed, use -v[vv..] for full protocol decode
listening on s3-eth2, link-type EN10MB (Ethernet), snapshot length 262144 bytes
14:09:57.122504 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:09:58.150505 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:09:59.173953 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:10:00.198356 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:10:01.221860 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:11:02.248152 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28
14:11:02.248152 ARP Request who-has 10.0.0.4 tell 10.0.0.1, length 28

```

Figura 1.13 switch h1 verso h4

### 3. Service Slicing

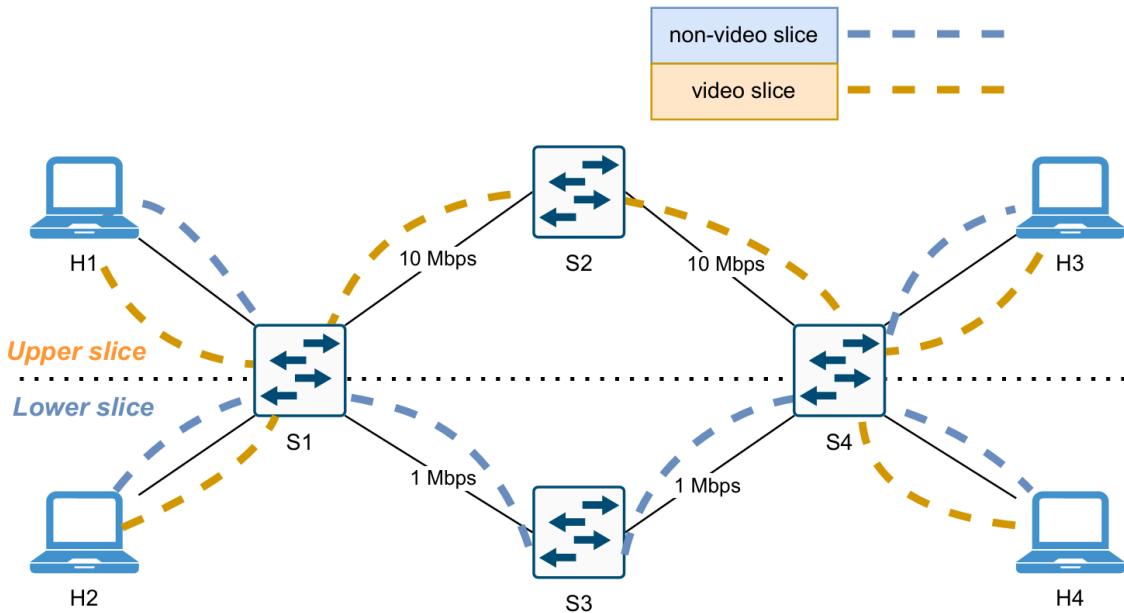


Figura 1.14 Topologia del Service Slicing

Per il secondo task la topologia della rete è rimasta la stessa, la differenza fondamentale sta nel fatto che:

- tutti i dispositivi possono comunicare tra di loro
- lo slicing avviene per la tipologia del traffico, il traffico video passa per l'upper slice (banda veloce) e il resto per il lower slice (banda lenta)
- il traffico video è individuato mediante UDP sul porto 9999
- essendo tutti i dispositivi raggiungibili tra di loro, gli switch s1 e s4 devono non solo instradare bene il traffico secondo lo slicing, se il pacchetto deve raggiungere un dispositivo dell'altro side, ma anche instradare correttamente i pacchetti tra h1 e h2 o h3 e h4 (per il traffico video o non).

#### 3.1 Controller Service Slicing

Di seguito riportiamo le **regole** installate all'interno degli switch da parte del controller.

Su ogni switch è caricata la **regola di Default**:

Priorità	Match (Ingresso)	Match (Tipo)	Azione (Uscita)	Descrizione Flusso
0	Qualsiasi	Qualsiasi	DROP	Serve a garantire l'isolamento dello slice se quel pacchetto non rispecchia nessun'altra regola

Per quanto riguarda **ARP**, visto che tutti i dispositivi devono essere raggiungibili quando uno switch come s1 o s4 riceve un pacchetto ARP, questo viene floodato su tutta la rete attraverso il lower slice.

### s1

Priorità	Match	Action	Descrizione
100	eth_type=0x0806 (ARP)	Output: 1 (h1), 2 (h2), 4 (s3)	Invia ARP agli host locali e alla slice "bassa". <b>Esclude S2 (porta 3).</b>

### s2

Al seguente switch non arriveranno mai pacchetti di richieste ARP, quindi nel caso "strano" in cui ci dovessero essere, gli stessi verranno droppati dalla regola di default.

### s3

Priorità	Match	Action	Descrizione
100	eth_type=0x0806 (ARP)	FLOOD	Inoltra l'ARP su tutte le porte attive (tranne quella di ingresso). S3 agisce come transito per il traffico di controllo.

s4

Priorità	Match	Action	Descrizione
100	eth_type=0x0806 (ARP)	Output: 3 (h3), 4 (h4), 2 (s3)	Invia ARP agli host locali e alla slice "bassa". <b>Esclude S2 (porta 1).</b>

**Regole di instradamento** per garantire il Service Slicing descritto

s1

Priorità	Match	Action	Descrizione
310	IP, UDP, dst_port=9999, src=h1, dst=h2	Output: h2	<b>Video Locale:</b> Traffico video diretto da h1 a h2 (non esce dallo switch).
310	IP, UDP, dst_port=9999, src=h2, dst=h1	Output: h1	<b>Video Locale:</b> Traffico video diretto da h2 a h1 (non esce dallo switch).

<b>300</b>	IP, UDP, dst_port=9999, in_port=1 (h1)	Output: 3 (s2)	<b>Video Uplink:</b> Traffico video da h1 verso h3/h4 instradato sulla Upper Slice.
<b>300</b>	IP, UDP, dst_port=9999, in_port=2 (h2)	Output: 3 (s2)	<b>Video Uplink:</b> Traffico video da h2 verso h3/h4 instradato sulla Upper Slice.
<b>300</b>	IP, in_port=3 (da s2), dst=h1	Output: 1 (h1)	<b>Video Return:</b> Traffico di ritorno dalla Upper Slice verso h1.
<b>300</b>	IP, in_port=3 (da s2), dst=h2	Output: 2 (h2)	<b>Video Return:</b> Traffico di ritorno dalla Upper Slice verso h2.
<b>210</b>	IP, dst=h1	Output: h1	<b>Standard Locale:</b> Traffico IP normale tra host locali (h2->h1).
<b>210</b>	IP, dst=h2	Output: h2	<b>Standard Locale:</b> Traffico IP normale tra host locali (h1->h2).
<b>200</b>	IP, dst=h3	Output: 4 (s3)	<b>Standard Uplink:</b> Tutto il resto del traffico verso h3 va sulla Lower Slice.
<b>200</b>	IP, dst=h4	Output: 4 (s3)	<b>Standard Uplink:</b> Tutto il resto del traffico verso h4 va sulla Lower Slice.

<b>200</b>	IP, in_port=4 (da s3), dst=h1	Output: h1	<b>Standard Return:</b> Traffico di ritorno dalla Lower Slice verso h1.
<b>200</b>	IP, in_port=4 (da s3), dst=h2	Output: h2	<b>Standard Return:</b> Traffico di ritorno dalla Lower Slice verso h2.

**s2**

Priorità	Match	Action	Descrizione
<b>300</b>	IP, in_port=1 (da s1)	Output: 2 (verso s4)	Instrada traffico video verso destinazione (Forward).
<b>300</b>	IP, in_port=2 (da s4)	Output: 1 (verso s1)	Instrada traffico video di ritorno (Backward).

**s3**

Priorità	Match	Action	Descrizione
<b>200</b>	IP, in_port=1 (da s1)	Output: 2 (verso s4)	Instrada traffico standard verso destinazione (Forward).
<b>200</b>	IP, in_port=2 (da s4)	Output: 1 (verso s1)	Instrada traffico standard di ritorno (Backward).

**s4**

Priorità	Match (Condizioni)	Action (Istruzioni)	Descrizione
<b>310</b>	IP, UDP, Port 9999, Src=H3, Dst=H4	Output: <b>Porta 4 (H4)</b>	<b>Video Locale:</b> Traffico video diretto da H3 a H4.
<b>310</b>	IP, UDP, Port 9999, Src=H4, Dst=H3	Output: <b>Porta 3 (H3)</b>	<b>Video Locale:</b> Traffico video diretto da H4 a H3.
<b>300</b>	IP, UDP, Port 9999, In_Port=3 (H3)	Output: <b>Porta 1 (S2)</b>	<b>Video Uplink:</b> Traffico video da H3 verso l'esterno (via Upper Slice).
<b>300</b>	IP, UDP, Port 9999, In_Port=4 (H4)	Output: <b>Porta 1 (S2)</b>	<b>Video Uplink:</b> Traffico video da H4 verso l'esterno (via Upper Slice).
<b>300</b>	IP, In_Port=1 (S2), Dst=H3	Output: <b>Porta 3 (H3)</b>	<b>Video Return:</b> Traffico di ritorno da S2 destinato a H3.
<b>300</b>	IP, In_Port=1 (S2), Dst=H4	Output: <b>Porta 4 (H4)</b>	<b>Video Return:</b> Traffico di ritorno da S2 destinato a H4.
<b>210</b>	IP, Dst=H4 (traffico locale)	Output: <b>Porta 4 (H4)</b>	<b>Standard Locale:</b> Traffico generico verso H4 (es. da H3).
<b>210</b>	IP, Dst=H3 (traffico locale)	Output: <b>Porta 3 (H3)</b>	<b>Standard Locale:</b> Traffico generico verso H3 (es. da H4).

<b>200</b>	IP, Dst=H1	Output: <b>Porta 2 (S3)</b>	<b>Standard Uplink:</b> Traffico generico verso H1 (via Lower Slice).
<b>200</b>	IP, Dst=H2	Output: <b>Porta 2 (S3)</b>	<b>Standard Uplink:</b> Traffico generico verso H2 (via Lower Slice).
<b>200</b>	IP, In_Port=2 (S3), Dst=H3	Output: <b>Porta 3 (H3)</b>	<b>Standard Return:</b> Traffico di ritorno da S3 destinato a H3.
<b>200</b>	IP, In_Port=2 (S3), Dst=H4	Output: <b>Porta 4 (H4)</b>	<b>Standard Return:</b> Traffico di ritorno da S3 destinato a H4.

In merito al codice implementato non ci sono metodi o dizionari diversi da quelli citati per il task precedente, sono state solo implementate le nuove regole precedentemente enunciate nella funzione *switch\_features\_handler* (sostituendo ovviamente le vecchie).

Una piccola specifica è per la priorità delle regole, il traffico di rete normale ha priorità minore di quello delle regole del traffico video perché qualunque pacchetto ricadrebbe in quella regola essendo molto generica, di conseguenza se si fosse messo un valore più alto rispetto a quella della regola video, il traffico video sarebbe trasmesso sul lower slice. Anche per il traffico locale si è scelto un valore di priorità maggiore rispetto a quello indirizzato sugli switch s2/s3 perchè è qualcosa di più specifico rispetto al restante traffico.

## 3.2 Testing

Di seguito sono riportate una serie di test effettuati per dimostrare il corretto slicing della rete.

Innanzitutto il pingAll ci permette di sapere che tutti i dispositivi sono raggiungibili tra di loro.

```
*** Test con ping all
*** 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)
*** Avvio CLI
*** Starting CLI:
mininet> █
```

Figura 1.15 pingAll Service Slicing

Per quanto riguarda la generazione del traffico video o normale è stato utilizzato *iperf* che permette di conoscere anche metriche come la banda disponibile, latenza e perdita di pacchetti; supportando sia traffico TCP e UDP.

### Traffico Video

Nella figura 1.16 è presente un esempio di come h1 comunichi con un server in h3 per la trasmissione video, la banda allocata per entrambe le trasmissioni è stata di circa 10 Mbits/sec quindi ci fa capire che la trasmissione è avvenuta nel upper slice.

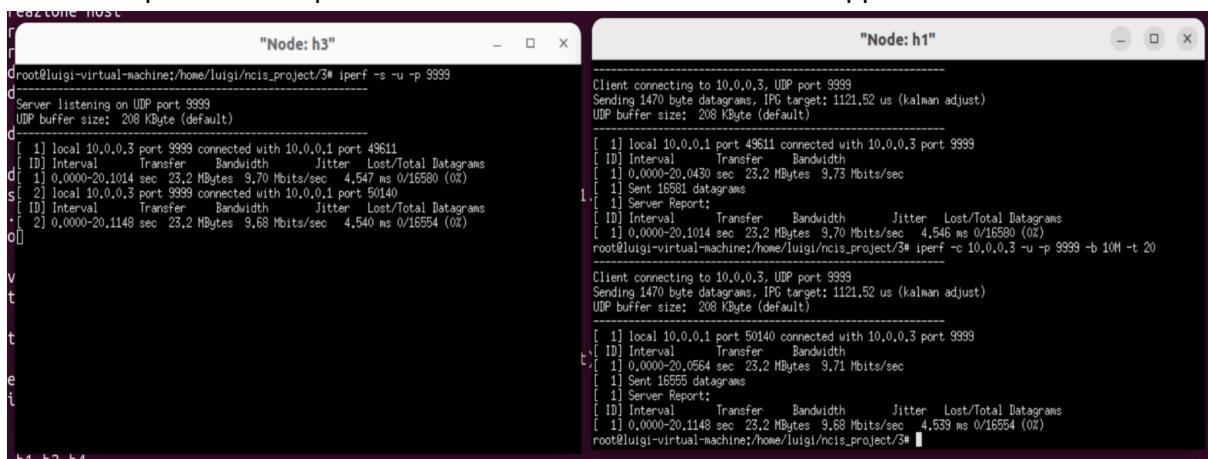


Figura 1.16 video h1 -> h3

Nella figura 1.17 è presente un esempio di comunicazione video tra h3 e h4, in particolare h4 trasmette video ad h3 per mezzo appunto solo di s4, avendo una banda allocata poco maggiore di 10 Mbits/sec.

```
"Node: h4"
root@luigi-virtual-machine:/home/luigi/ncis_project/3# iperf -c 10.0.0.3 -u -p 9999 -b 10M -t 10
Client connecting to 10.0.0.3, UDP port 9999
Sending 1470 byte datagrams, IPG target: 1121.52 us (kalman adjust)
UDP buffer size: 208 KByte (default)
[ 1] local 10.0.0.4 port 42572 connected with 10.0.0.3 port 9999
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 1] 0.0000-10.0018 sec 12.5 MBytes 10.5 Mbits/sec
[ 1] Sent 8921 datagrams
[ 1] Server Report:
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 1] 0.0000-10.0012 sec 12.5 MBytes 10.5 Mbits/sec 0.004 ms 0/8920 (0%)
root@luigi-virtual-machine:/home/luigi/ncis_project/3# 

"Node: h3"
root@luigi-virtual-machine:/home/luigi/ncis_project/3# iperf -s -u -p 9999
Server listening on UDP port 9999
UDP buffer size: 208 KByte (default)
[ 1] local 10.0.0.3 port 9999 connected with 10.0.0.4 port 42572
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 1] 0.0000-10.0012 sec 12.5 MBytes 10.5 Mbits/sec 0.005 ms 0/8920 (0%)
[ 1] Sent 8921 datagrams
[ 1] Server Report:
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 1] 0.0000-10.0012 sec 12.5 MBytes 10.5 Mbits/sec 0.004 ms 0/8920 (0%)
root@luigi-virtual-machine:/home/luigi/ncis_project/3# 
```

Figura 1.17 video h4 -> h3

### Traffico normale

Nella figura 1.18 è stata effettuata una trasmissione su h4, server UDP, sul porto 5000, dal dispositivo h1 che ha avviato una trasmissione con un banda di traffico pari a 10 Mbits/sec. Da come si evince dai risultati di iperf la bandwidth è stata effettivamente di 1.24 Mbits/sec quindi ci fa capire che il canale scelto è stato quello del lower slice, inoltre è stato anche lievemente congestionato il canale in quanto h4 riporta dei warning su degli ack non ricevuti e h1 presenta un jitter molto alto.

```
"Node: h4"
root@luigi-virtual-machine:/home/luigi/ncis_project/3# iperf -s -u -p 5000
Server listening on UDP port 5000
UDP buffer size: 208 KByte (default)
[ 1] local 10.0.0.4 port 5000 connected with 10.0.0.1 port 56627
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 1] 0.0000-10.9928 sec 1.28 MBytes 974 Kbytes/sec 45.587 ms 0/910 (0%)
[ 5] WARNING: ack of last datagram failed,
[ 2] local 10.0.0.4 port 5000 connected with 10.0.0.1 port 56627
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 2] 0.0000-0.0121 sec 2.87 KBytes 1.95 Mbits/sec 0.754 ms 919/921 (1e+02%)
[ 6] WARNING: ack of last datagram failed,
[ 3] local 10.0.0.4 port 5000 connected with 10.0.0.1 port 56627
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 3] 0.0000-0.0121 sec 2.87 KBytes 1.94 Mbits/sec 0.759 ms 930/932 (1e+02%)
[ 5] WARNING: ack of last datagram failed,
[ 4] local 10.0.0.4 port 5000 connected with 10.0.0.1 port 56627
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 4] 0.0000-0.0121 sec 2.87 KBytes 1.94 Mbits/sec 0.757 ms 941/943 (1e+02%)
[ 6] WARNING: ack of last datagram failed,
[ 5] local 10.0.0.4 port 5000 connected with 10.0.0.1 port 56627
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 5] 0.0000-0.0122 sec 2.87 KBytes 1.93 Mbits/sec 0.760 ms 952/954 (1e+02%)
[ 5] WARNING: ack of last datagram failed,
tes FROM 10.0.0.3: icmp_seq=2 ttl=64 time=0.111 ms
tes from 10.0.0.3: icmp seq=3 ttl=64 time=0.087 ms

"Node: h1"
root@luigi-virtual-machine:/home/luigi/ncis_project/3# iperf -c 10.0.0.4 -u -p 5000 -b 10M -t 10
Client connecting to 10.0.0.4, UDP port 5000
Sending 1470 byte datagrams, IPG target: 1121.52 us (kalman adjust)
UDP buffer size: 208 KByte (default)
[ 1] local 10.0.0.1 port 56627 connected with 10.0.0.4 port 5000
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 1] 0.0000-10.3908 sec 1.28 MBytes 1.03 Mbits/sec
[ 1] Sent 911 datagrams
[ 1] Server Report:
[ ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[ 1] 0.0000-10.9928 sec 1.28 MBytes 974 Kbytes/sec 45.586 ms 0/910 (0%)
root@luigi-virtual-machine:/home/luigi/ncis_project/3# 
```

Figura 1.18 traffico normale h1 -> h4

In quest'ultimo caso (*figura 1.19*) è stato fatto un esempio di trasmissione di traffico tramite TCP (h4->h2), vedendo la banda allocata pari a poco più di 1 Mbits/sec ci fa capire che è stato scelto il lower slice.

```

"Node: h2"
root@luigi-virtual-machine:/home/luigi/ncis_project/3# iperf -s -p 80
R Server listening on TCP port 80
TCP window size: 85,3 KByte (default)
S
[ 1] local 10.0.0.2 port 80 connected with 10.0.0.4 port 55266
n [ ID] Interval Transfer Bandwidth
n[ 1] 0,0000-30,8281 sec 3,58 MB/s 974 Kbytes/sec
n
y
y
y
y
1
c
m
n
10.0.0.3 (10.0.0.3) 30,8281 bytes/s data:
bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=0.324 ms
bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.111 ms
bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.087 ms

"Node: h4"
root@luigi-virtual-machine:/home/luigi/ncis_project/3# iperf -c 10.0.0.2 -p 80 -t 10
Client connecting to 10.0.0.2, TCP port 80
TCP window size: 128 KByte (default)
[ 1] local 10.0.0.4 port 55266 connected with 10.0.0.2 port 80
[ ID] Interval Transfer Bandwidth
[ 1] 0,0000-24,5796 sec 3,58 MB/s 1,22 Mbit/s
root@luigi-virtual-machine:/home/luigi/ncis_project/3#

```

*Figura 1.19 traffico tcp h4 -> h2*

## 4. Dynamic Slicing

Come possibile estensione è stato scelto di fare il punto del Dynamic Slicing, che consiste nell'utilizzare l'upper slice non solo esclusivamente per la trasmissione video ma anche per il traffico normale se ci sta abbastanza banda disponibile, assicurando ovviamente che il traffico video abbia priorità assoluta.

### 4.1 Controller Dynamic Slicing

Per implementare la seguente richiesta, è stato introdotto un **thread** nel controller che chiede le statiche dei flussi che passano per gli switch s1 e s4 ogni 2 secondi, se si evince che tra questi il traffico video è **minore di 1 Mbits/sec** allora può transitare il traffico normale nell'upper slice, se è **superiore** di tale soglia allora il traffico normale va nel lower slice per dare priorità a quello video, per far variare il transito del traffico normale bisogna cambiare dinamicamente le regole di instradamento.

Il valore di soglia pari a 1 Mbits/sec è stato scelto come valore empirico, presumibilmente il traffico video medio avrà bisogno di almeno quel valore di banda per una sua corretta trasmissione.

La seguente soluzione è stata prodotta integrando e migliorando la soluzione di Service Slicing.

Dove c'erano prima nella funzione di `__init__` solo i dizionari degli host e della topologia della rete, ora come si vede anche in *Figura 1.20* sono state aggiunte:

- dizionario **datapath** per tenere traccia degli switch attivi nella rete, serve al thread per un corretto monitoraggio
- **monitor\_interval** che definisce la frequenza con cui andare a controllare i flussi
- **bandwidth\_threshold** è la soglia definita di 1 Mbits/sec

- **`video_stats`** è un dizionario che tiene memoria per s1 e s4 il numero di byte del flusso video precedente.
- **`global_slice_state`** è una variabile globale che permette di sapere se il traffico normale deve andare nel Lower Slice o può passare nel Upper Slice, è stata inizializzata come Under Slice.
- **`monitor_thread`** è il thread allocato dal controller, il suo comportamento è definito nella funzione interna **`_monitor`**

```

def __init__(self, *args, **kwargs):
    super(DynamicSliceController, self).__init__(*args, **kwargs)

    self.datapaths = {}

    self.monitor_interval = 2
    self.bandwidth_threshold = 1000000 / 8 # 1 Mbps

    # Dizionario per i byte dei flussi VIDEO precedenti
    # Chiave: dpid -> Valore: byte totali video letti prima
    self.video_stats = {
        1: 0,
        4: 0
    }

    self.global_slice_state = 'LOWER'
    self.monitor_thread = hub.spawn(self._monitor)

    self.H = {'h1':'00:00:00:00:00:01', 'h2':'00:00:00:00:00:02', 'h3':'00:00:00:00:00:03', 'h4':'00:00:00:00:00:04'}
    self.PORT_MAP = {
        1: {'h1': 1, 'h2': 2, 's2': 3, 's3': 4},
        2: {'s1': 1, 's4': 2},
        3: {'s1': 1, 's4': 2},
        4: {'s2': 1, 's3': 2, 'h3': 3, 'h4': 4}
    }
}

```

Figura 1.20 `__init__` Dynamic Slicing

In Figura 1.22 è presente il comportamento del thread, il quale ogni 2 secondi chiede le statistiche dei flussi agli switch 1 e 4.

```

# --- MONITORING LOOP (Flow Stats) ---
def _monitor(self):
    while True:
        for dp in self.datapaths.values():
            if dp.id in [1, 4]:
                parser = dp.ofproto_parser
                # Statistiche sui FLUSSI
                req = parser.OFPFlowStatsRequest(dp)
                dp.send_msg(req)
        hub.sleep(self.monitor_interval)

```

Figura 1.22 `_monitor`

La gestione della risposta alla richiesta delle statistiche dei flussi è stata fatto mediante l'implementazione della funzione:

```

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev)

```

```

body = ev.msg.body
dpid = ev.msg.datapath.id

if dpid not in [1, 4]:
    return

current_video_bytes = 0

for flow in body:
    # 1. Filtriamo per Priorità 300 (la slice Video)
    if flow.priority == 300:

        if 'udp_dst' in flow.match and flow.match['udp_dst'] == 9999:
            current_video_bytes += flow.byte_count

# Recuperiamo il valore precedente
prev_bytes = self.video_stats[dpid]

if prev_bytes == 0:
    delta_bytes = 0
else:
    delta_bytes = current_video_bytes - prev_bytes
    if delta_bytes < 0: delta_bytes = 0 # Gestione reset contatori

self.video_stats[dpid] = current_video_bytes

# Calcolo velocità
video_speed = delta_bytes / self.monitor_interval

```

*Figura 1.23 \_flow\_stats\_reply\_handler - 1*

La funzione, come si vede in *Figura 1.23*, in primis controlla se la risposta proviene da s1 o s4 altrimenti si conclude, dopodichè filtra tra tutti i flussi prendendo solo quelli che soddisfano la priorità 300 e la porta di destinazione UDP 9999 (flusso video) salvando il numero di byte, noto il numero di Byte del flusso video precedente si calcola la velocità attuale.

```

# Aggiorniamo le velocità correnti per il confronto globale
if not hasattr(self, 'current_speeds'):
    self.current_speeds = {1: 0.0, 4: 0.0}
self.current_speeds[dpid] = video_speed

# Prendiamo il massimo tra i due switch
max_video_speed = max(self.current_speeds[1], self.current_speeds[4])

# Soglia (1 Mbps)
is_congested = max_video_speed > self.bandwidth_threshold

if is_congested and self.global_slice_state == 'UPPER':
    self.logger.info(f"*** VIDEO RILEVATO ({max_video_speed*8/1e6:.2f} Mbps). Traffico Standard -> LOWER.")
    self.apply_slice_policy('LOWER')

elif not is_congested and self.global_slice_state == 'LOWER':
    if max_video_speed < (self.bandwidth_threshold / 2):
        self.logger.info(f"*** VIDEO TERMINATO ({max_video_speed*8/1e6:.2f} Mbps). Traffico Standard -> UPPER.")
        self.apply_slice_policy('UPPER')

```

Figura 1.24 \_flow\_stats\_reply\_handler - 2

Il continuo del codice è presente in *figura 1.24*, dove verrà memorizzata la velocità attuale in un dizionario denominato ***current\_speeds***, preso il valore massimo tra le due velocità correnti (tra s1 e s4) se questo supera la soglia massimo ed il ***global\_slice\_state*** è attualmente ‘UPPER’ allora si cambieranno le regole di instradamento in modo tale da poter passare alla configurazione ‘LOWER’ per il traffico normale. Per ripassare alla configurazione precedente non basta solo che il traffico video non sia congestionato e la configurazione sia ‘LOWER’, ma è necessario che la ***max\_video\_speed*** sia minore alla metà della soglia (0.5MB). Questo serve a prevenire eventuali sbalzi del traffico video, in cui la banda oscillerà poco più o poco meno ad 1 MB di banda (in fase conclusiva di trasmissione), quindi senza questo controllo il traffico normale si sarebbe spostato tra ‘UPPER’ e ‘SLICE’ inutilmente.

```

def apply_slice_policy(self, target_slice):
    self.global_slice_state = target_slice
    for dpid, dp in self.datapaths.items():
        parser = dp.ofproto_parser
        if dpid == 1:
            out_port = self.PORT_MAP[1]['s3'] if target_slice == 'LOWER' else self.PORT_MAP[1]['s2']
            for dst in ['h3', 'h4']:
                match = parser.OFPMatch(eth_type=0x0800, eth_dst=self.H[dst])
                self.add_flow(dp, 250, match, [parser.OFPActionOutput(out_port)])
        elif dpid == 4:
            out_port = self.PORT_MAP[4]['s3'] if target_slice == 'LOWER' else self.PORT_MAP[4]['s2']
            for dst in ['h1', 'h2']:
                match = parser.OFPMatch(eth_type=0x0800, eth_dst=self.H[dst])
                self.add_flow(dp, 250, match, [parser.OFPActionOutput(out_port)])

```

Figura 1.25 apply\_slice\_policy

Per cambiare dinamicamente le regole si è implementata la funzione ***apply\_slice\_policy*** che in base al target fornito cambia la regola con priorità 250 all’interno degli switch s1 e s4. Di seguito è riportata una tabella sintetica delle possibili regole che vengono aggiunte.

<b>Switch (DPID)</b>	<b>Host Destinazione</b>	<b>target_slice</b>	<b>Slice Scelta</b>	<b>Porta di Uscita</b>	<b>Descrizione Azione</b>
<b>S1</b> (dpid=1)	<b>h3</b>	'LOWER'	Lower Slice (S3)	4 (s3)	Traffico spostato sul percorso lento (1 Mbps)
<b>S1</b> (dpid=1)	<b>h3</b>	'UPPER'	Upper Slice (S2)	3 (s2)	Traffico spostato sul percorso veloce (10 Mbps)
<b>S1</b> (dpid=1)	<b>h4</b>	'LOWER'	Lower Slice (S3)	4 (s3)	Traffico spostato sul percorso lento (1 Mbps)
<b>S1</b> (dpid=1)	<b>h4</b>	'UPPER'	Upper Slice (S2)	3 (s2)	Traffico spostato sul percorso veloce (10 Mbps)
<b>S4</b> (dpid=4)	<b>h1</b>	'LOWER'	Lower Slice (S3)	2 (s3)	Traffico spostato sul percorso lento (1 Mbps)

<b>S4</b> (dpid=4)	<b>h1</b>	'UPPER'	Upper Slice (S2)	1 (s2)	Traffico spostato sul percorso veloce (10 Mbps)
<b>S4</b> (dpid=4)	<b>h2</b>	'LOWER'	Lower Slice (S3)	2 (s3)	Traffico spostato sul percorso lento (1 Mbps)
<b>S4</b> (dpid=4)	<b>h2</b>	'UPPER'	Upper Slice (S2)	1 (s2)	Traffico spostato sul percorso veloce (10 Mbps)

```
@set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        if datapath.id not in self.datapaths:
            self.datapaths[datapath.id] = datapath
    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.datapaths:
            del self.datapaths[datapath.id]
            if datapath.id in self.video_stats:
                self.video_stats[datapath.id] = 0
```

Figura 1.26 \_state\_change\_heandler

La funzione in *figura 1.26* serve a tenere traccia se uno switch si connette o disconnette dal controller.

## 4.2 Testing

La prima cosa da verificare è la completa raggiungibilità dei dispositivi, che è stato dimostrato tramite *pingAll*.

```
*** 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 1.27 *pingAll*

Sono stati effettuati due test:

Nel **primo esempio** su h1 e h3 giravano rispettivamente dei server UDP sulla porta 9999 e TCP sulla porta 5000, h2 parlerà con h3 per il traffico normale per 20 sec e h4 parlerà con h1 per 10 secondi col traffico video per una banda richiesta di 8MB. Inizia prima il traffico normale e dopo circa 5 secondi inizia il traffico video. Come si vede in *Figura 1.28*, la banda di h2 complessiva è stata di 5.33 Mbits/sec, indice che ci fa capire che c'è stato il dynamic slicing e quindi il traffico normale è stato limitato per la banda inferiore, da h4 si può notare come la banda richiesta di 8 Mbits/sec è stata sempre soddisfatta. Inoltre sul terminale nello sfondo si può vedere come il controller detecta il traffico video per spostare il traffico normale nel lower slice. Si può visualizzare il seguente [video](#) per una maggiore chiarezza.

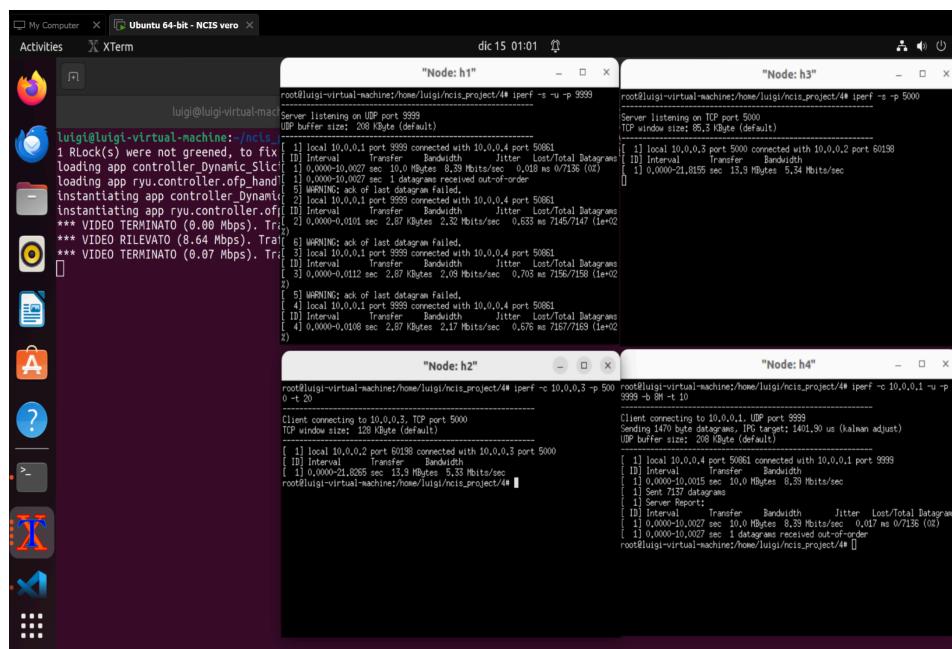


Figura 1.28 Esempio

Nel **secondo esempio** su h3 e h4 giravano rispettivamente dei server UDP sulla porta 5000 e UDP sulla porta 9999, h2 parlerà con h3 per il traffico normale per 20 sec con un banda richiesta di 10MB e h1 parlerà con h4 per 10 secondi col traffico video con una banda richiesta di 8MB. Inizia prima il traffico normale e dopo circa 5 secondi inizia il traffico video. Come si vede in *Figura 1.29*, la banda di h2 complessiva è stata di 5.92 Mbits/sec, indice che ci fa capire che c'è stato il dynamic slicing e quindi il traffico normale è stato limitato per la banda inferiore ad una certa per poi ritornare allo slice superiore, da h4 si può notare come la banda richiesta di 8 Mbits/sec è stata sempre soddisfatta. Inoltre sul terminale nello sfondo si può vedere come il controller detecta il traffico video per spostare il traffico normale nel lower slice. Si può visualizzare il seguente [video](#) per una maggiore chiarezza.

```

My Computer X Ubuntu 64-bit - NCIS vero
Activities X XTerm dic 15 01:19

"Node: h1" "Node: h3"
root@luigi-virtual-machine:/home/luigi/ncis_project/4# iperf -s -u -p 5000
[1] local 10.0.0.3 port 5000 connected with 10.0.0.2 port 46297
[1] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[1] 0.0000-20.062 sec 14.2 MBytes 5.92 Mbits/sec 4.524 ms 0/10103 (0%)
[1] 0.0000-20.062 sec 46 datagrams received out-of-order

"Node: h2" "Node: h4"
root@luigi-virtual-machine:/home/luigi/ncis_project/4# iperf -c 10.0.0.4 -u -p 9999 -b 10M -t 20
[1] local 10.0.0.2 port 46297 connected with 10.0.0.3 port 5000
[1] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[1] 0.0000-20.069 sec 14.2 MBytes 5.94 Mbits/sec
[1] Sent 10104 datagrams
[1] Server Report:
[1] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[1] 0.0000-20.069 sec 14.2 MBytes 5.92 Mbits/sec 4.523 ms 0/10103 (0%)
[1] 0.0000-20.062 sec 46 datagrams received out-of-order
root@luigi-virtual-machine:/home/luigi/ncis_project/4# 

```

*Figura 1.29 Esempio 2*

## 5. Codice

Per la visualizzazione dei codici completi è possibile consultare il seguente [github](#).