UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Corso di Networks and Cloud Infrastructures

Network Slicing with Ryu and Mininet

Anno Accademico 2025/2026

Docente
Giorgio Ventre

Studenti
Gennaro Iannicelli M63001668
Giuseppe Gatta M63001669

Contents

Chapter 1

Introduzione

Negli ultimi anni le reti di comunicazione hanno subito una profonda trasformazione, determinata dalla crescente diffusione di applicazioni digitali eterogenee e ad alta intensità di traffico. Servizi quali lo streaming video ad alta definizione, le piattaforme di videoconferenza, il cloud computing, l'edge computing e l'Internet of Things (IoT) impongono requisiti sempre più stringenti in termini di banda, latenza, affidabilità e continuità del servizio.

Le architetture di rete tradizionali, progettate per scenari relativamente statici, risultano poco flessibili e difficili da riconfigurare dinamicamente. La stretta integrazione tra piano di controllo e piano di inoltro rende complessa l'implementazione di politiche avanzate di gestione del traffico e qualità del servizio (Quality of Service, QoS).

In questo contesto emergono nuovi paradigmi, tra cui il Software-Defined Networking (SDN) e il Network Slicing, che consentono una

gestione centralizzata, programmabile e dinamica delle risorse di rete. Il presente progetto si colloca in tale contesto e mira a dimostrare sperimentalmente l'efficacia di questi concetti.

1.1 Traccia progetto

La traccia del progetto prevede l'implementazione del Network Slicing in un ambiente SDN utilizzando Mininet come emulatore di rete e Ryu come controller SDN basato su OpenFlow.

Il progetto è suddiviso in due parti principali:

- **Topology Slicing**: limitazione dei percorsi di comunicazione tra specifici host tramite regole di forwarding;
- **Service Slicing**: classificazione del traffico in base al servizio e assegnazione di priorità differenti.

Per il Topology Slicing è richiesto che:

- l'host H1 possa comunicare esclusivamente con H3 tramite un percorso superiore della rete;
- l'host H2 possa comunicare esclusivamente con H4 tramite un percorso inferiore;
- tutte le altre comunicazioni siano bloccate dal controller SDN.

Il Service Slicing prevede l'identificazione del traffico video come traffico UDP diretto alla porta 9999, che deve essere trattato come prioritario e instradato su una slice dedicata ad alta capacità. Il corretto funzionamento delle politiche deve essere verificato tramite strumenti di test e monitoraggio.

1.2 Introduzione teorica

Il Software-Defined Networking introduce una separazione netta tra il piano di controllo e il piano di inoltro. Il controllo della rete viene centralizzato in un'entità software, il controller SDN, che comunica con i dispositivi di rete tramite protocolli standardizzati come OpenFlow.

Il Network Slicing rappresenta un'evoluzione del paradigma SDN e consente la creazione di più reti logiche indipendenti, denominate slice, sulla stessa infrastruttura fisica. Ogni slice può essere configurata con requisiti specifici di banda, latenza, affidabilità e sicurezza, risultando particolarmente adatta a scenari come reti 5G e ambienti multi-servizio.

1.3 Obiettivi

Gli obiettivi principali del progetto sono:

- progettare una topologia SDN con almeno due slice distinte;

- implementare il Topology Slicing basato su indirizzi MAC;
- implementare il Service Slicing basato sulla classificazione del traffico;
- garantire una priorità maggiore al traffico video (UDP/9999);
- verificare sperimentalmente il comportamento della rete;
- analizzare le regole di flusso installate sugli switch OpenFlow.

1.4 Ambiente di sviluppo e tecnologie

Il progetto è stato sviluppato utilizzando strumenti open-source ampiamente adottati in ambito accademico. Mininet è stato utilizzato per l'emulazione della rete SDN, Ryu come controller SDN, Open vSwitch come switch OpenFlow, mentre i test e le analisi sono stati condotti mediante iPerf, tcpdump e ovs-ofctl.

Chapter 2

Implementazione

2.1 Descrizione della topologia

La topologia di rete realizzata è costituita da quattro host (H1, H2, H3, H4) e quattro switch OpenFlow (S1, S2, S3, S4), tutti controllati da un unico controller SDN basato su Ryu. Tale architettura è stata progettata con l'obiettivo di supportare esplicitamente il concetto di Network Slicing, consentendo la definizione di più slice logiche indipendenti sulla stessa infrastruttura fisica.

La rete è organizzata in modo da prevedere due percorsi logicamente distinti tra gli host, ciascuno dei quali rappresenta una slice separata:

- **Slice superiore**: consente la comunicazione esclusiva tra l'host H1 e l'host H3 attraverso il percorso S1 $\rightarrow$ S2 $\rightarrow$ S4. Questo cammino è associato alla slice ad alta capacità ed è destinato al

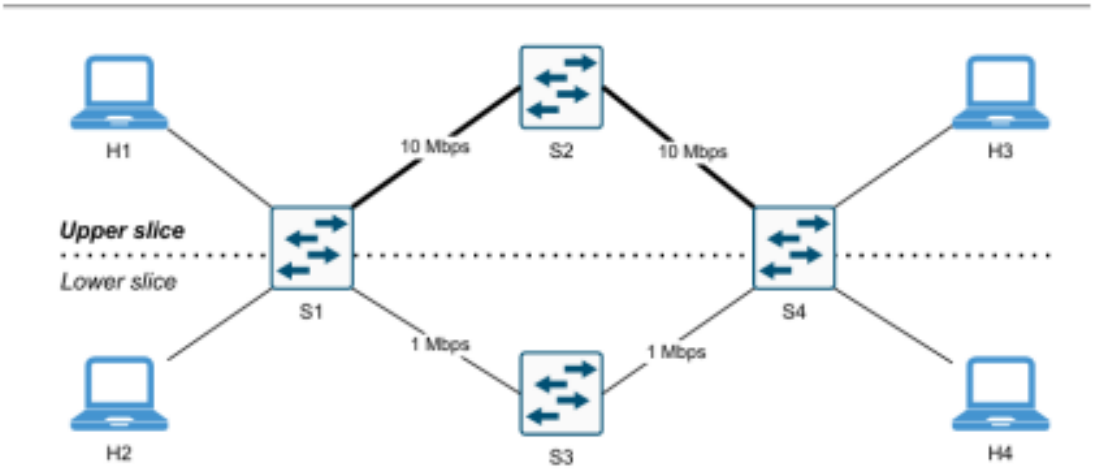
traffico prioritario.

- **Slice inferiore**: permette la comunicazione esclusiva tra l'host H2 e l'host H4 attraverso un percorso alternativo rispetto a quello della slice superiore, caratterizzato da capacità inferiore e destinato al traffico best-effort.

I collegamenti tra gli switch sono stati configurati in modo tale da definire chiaramente due cammini distinti e non sovrapposti per le comunicazioni tra le rispettive coppie di host. Questa scelta progettuale consente di ottenere un isolamento logico completo tra le slice, evitando interferenze tra i flussi di traffico appartenenti a slice diverse.

In particolare, l'host H1 è in grado di comunicare esclusivamente con H3, mentre l'host H2 può comunicare solo con H4. Tutti gli altri tentativi di comunicazione tra host vengono esplicitamente bloccati mediante regole di inoltro installate dinamicamente dal controller Ryu. Tali regole sono basate sul matching degli indirizzi MAC sorgente e destinazione e consentono di implementare efficacemente il Topology Slicing.

La topologia di rete implementata è illustrata nella Figura 2.1. Questa struttura consente di testare in modo controllato la gestione del traffico e il comportamento della rete in presenza di segmentazione logica, risultando rappresentativa di scenari reali quali reti 5G, ambienti edge computing e infrastrutture multi-tenant, dove l'isolamento tra servizi e utenti rappresenta un requisito fondamentale.



Codice

```
    from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import OVSKernelSwitch,
   RemoteController
from mininet.cli import CLI
from mininet.link import TCLink

class NetworkSlicingTopo (Topo):
    def __init__(self):
        # Initialize topology
        Topo.__init__(self)

        # Create template host, switch, and link
        host_config = dict(inNamespace=True)
        http_link_config = dict(bw=1)
        video_link_config = dict(bw=10)
        host_link_config = dict()
```

```
17
18          # Create switch nodes
19          for i in range (4):
20              sconfig = {"dpid": "%016x" % (i + 1)}
21              self.addSwitch("s%d" % (i + 1), **sconfig)
22
23          # Create host nodes
24          for i in range (4):
25              mac_addr = "00:00:00:00:00:0%d" % (i + 1)
26              self.addHost("h%d" % (i + 1), mac=mac_addr,
                    **host_config)
27
28          # Add switch links
29          self.addLink("s1", "s2", **video_link_config)
30          self.addLink("s2", "s4", **video_link_config)
31          self.addLink("s1", "s3", **http_link_config)
32          self.addLink("s3", "s4", **http_link_config)
33
34          # Add host links
35          self.addLink("h1", "s1", **host_link_config)
36          self.addLink("h2", "s1", **host_link_config)
37          self.addLink("h3", "s4", **host_link_config)
38          self.addLink("h4", "s4", **host_link_config)
39
40  topos = {"networkslicingtopo": (lambda:
        NetworkSlicingTopo())}
41
42  if __name__ == "__main__":
```

```
43      topo = NetworkSlicingTopo()
44      net = Mininet(
45          topo=topo,
46          switch=OVSKernelSwitch,
47          build=False,
48          autoSetMacs=True,
49          autoStaticArp=True,
50          link=TCLink,
51      )
52      controller = RemoteController("c1", ip="127.0.0.1",
           port=6633)
53      net.addController(controller)
54      net.build()
55      net.start()
56      CLI(net)
57      net.stop()
```

Spiegazione codice

Abbiamo definito una topologia di rete personalizzata in Mininet per simulare il concetto di *network slicing*. In particolare, abbiamo creato:

- quattro switch Open vSwitch, ciascuno identificato da un DPID univoco;
- quattro host, assegnando a ciascuno un indirizzo MAC statico e inserendoli in namespace separati.

Successivamente, abbiamo configurato i collegamenti tra gli switch in

modo da realizzare due percorsi distinti con caratteristiche di capacità differenti:

- un percorso ad alta capacità, con banda pari a 10 Mbps, dedicato al traffico video;
- un percorso a bassa capacità, limitato a 1 Mbps, destinato a traffico meno prioritario, come quello HTTP.

I collegamenti tra host e switch sono stati invece creati utilizzando i parametri di default. La rete è stata istanziata utilizzando `OVSKernelSwitch` ed è stata collegata a un controller remoto tramite `RemoteController`. Una volta avviata la topologia, abbiamo potuto interagire con essa tramite la CLI di Mininet e verificare da terminale che fosse stata istanziata correttamente.

```
Unable to contact the remote controller at 127.0.0.1:6633
mininet> nodes
available nodes are:
c1 h1 h2 h3 h4 s1 s2 s3 s4
mininet> links
h1-eth0<->s1-eth3 (OK OK)
h2-eth0<->s1-eth4 (OK OK)
h3-eth0<->s4-eth3 (OK OK)
h4-eth0<->s4-eth4 (OK OK)
s1-eth1<->s2-eth1 (OK OK)
s1-eth2<->s3-eth1 (OK OK)
s2-eth2<->s4-eth1 (OK OK)
s3-eth2<->s4-eth2 (OK OK)
mininet> net
h1 h1-eth0:s1-eth3
h2 h2-eth0:s1-eth4
h3 h3-eth0:s4-eth3
h4 h4-eth0:s4-eth4
s1 lo:  s1-eth1:s2-eth1 s1-eth2:s3-eth1 s1-eth3:h1-eth0 s1-eth4:h2-eth0
s2 lo:  s2-eth1:s1-eth1 s2-eth2:s4-eth1
s3 lo:  s3-eth1:s1-eth2 s3-eth2:s4-eth2
s4 lo:  s4-eth1:s2-eth2 s4-eth2:s3-eth2 s4-eth3:h3-eth0 s4-eth4:h4-eth0
c1
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=23793>
<Host h2: h2-eth0:10.0.0.2 pid=23795>
<Host h3: h3-eth0:10.0.0.3 pid=23797>
<Host h4: h4-eth0:10.0.0.4 pid=23799>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None,s1-eth4:None pid=23804>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None pid=23807>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None pid=23810>
<OVSSwitch s4: lo:127.0.0.1,s4-eth1:None,s4-eth2:None,s4-eth3:None,s4-eth4:None pid=23813>
<RemoteController c1: 127.0.0.1:6633 pid=23787>
mininet>
```

2.2 Topology Slicing

Il Topology Slicing è stato implementato imponendo vincoli rigorosi sulle comunicazioni consentite tra gli host, garantendo isolamento logico e controllo dei percorsi di rete.

2.2.1 Controller Ryu

Il controller utilizza una struttura `mac_to_port` per apprendere dinamicamente la posizione degli host. Solo le coppie di host autorizzate possono comunicare e vengono installate regole di forwarding sugli switch appartenenti alla slice corretta.

```
    from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER,
   MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types

class TopologySlicingMacToPort(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

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

```
        args, **kwargs)
15      self.mac_to_port = {}
16      # MAC address degli host
17      self.mac_h1 = "00:00:00:00:00:01"
18      self.mac_h2 = "00:00:00:00:00:02"
19      self.mac_h3 = "00:00:00:00:00:03"
20      self.mac_h4 = "00:00:00:00:00:04"
21
22      # Coppie autorizzate (slice)
23      self.allowed_pairs = {
24          (self.mac_h1, self.mac_h3),
25          (self.mac_h3, self.mac_h1),
26          (self.mac_h2, self.mac_h4),
27          (self.mac_h4, self.mac_h2),
28      }
29      # Mappatura switch ammessi
30      self.upper_path_switches = {1, 2, 4}
31      self.lower_path_switches = {1, 3, 4}
32
33  @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
        CONFIG_DISPATCHER)
34  def switch_features_handler(self, ev):
35      datapath = ev.msg.datapath
36      ofproto = datapath.ofproto
37      parser = datapath.ofproto_parser
38      match = parser.OFPMatch()
39      actions = [parser.OFPActionOutput(ofproto.
            OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
```

```
40          inst = [parser.OFPInstructionActions(ofproto.
               OFPIT_APPLY_ACTIONS, actions)]
41          mod = parser.OFPFlowMod(datapath=datapath,
               priority=0, match=match, instructions=inst)
42          datapath.send_msg(mod)
43
44      def add_flow(self, datapath, priority, match, actions
           , buffer_id=None):
45          ofproto = datapath.ofproto
46          parser = datapath.ofproto_parser
47          inst = [parser.OFPInstructionActions(ofproto.
               OFPIT_APPLY_ACTIONS, actions)]
48          if buffer_id:
49              mod = parser.OFPFlowMod(datapath=datapath,
                   buffer_id=buffer_id, priority=priority,
                   match=match, instructions=inst)
50          else:
51              mod = parser.OFPFlowMod(datapath=datapath,
                   priority=priority, match=match,
                   instructions=inst)
52          datapath.send_msg(mod)
53
54      @set_ev_cls(ofp_event.EventOFPPacketIn,
           MAIN_DISPATCHER)
55      def _packet_in_handler(self, ev):
56          msg = ev.msg
57          datapath = msg.datapath
58          dpid = datapath.id
```

```
59          parser = datapath.ofproto_parser
60          in_port = msg.match["in_port"]
61          pkt = packet.Packet(msg.data)
62          eth = pkt.get_protocols(ethernet.ethernet)[0]
63          dst = eth.dst
64          src = eth.src
65
66          if eth.ethertype == ether_types.ETH_TYPE_LLDP:
67              return
68
69          self.mac_to_port.setdefault(dpid, {})
70          self.mac_to_port[dpid][src] = in_port
71
72          if (src, dst) not in self.allowed_pairs:
73              return
74
75          # Verifica compatibilita dpid con la slice
76          if (src == self.mac_h1 and dst == self.mac_h3) or
                (src == self.mac_h3 and dst == self.mac_h1):
77              if dpid not in self.upper_path_switches:
78                  return
79          elif (src == self.mac_h2 and dst == self.mac_h4)
               or (src == self.mac_h4 and dst == self.mac_h2):
80              if dpid not in self.lower_path_switches:
81                  return
82
83          if dst in self.mac_to_port[dpid]:
84              out_port = self.mac_to_port[dpid][dst]
```

```
85          else:
86              out_port = datapath.ofproto.OFPP_FLOOD
87
88          actions = [parser.OFPActionOutput(out_port)]
89          match = parser.OFPMatch(in_port=in_port, eth_src=
               src, eth_dst=dst)
90          self.add_flow(datapath, 1, match, actions)
91
92          out = parser.OFPPacketOut(datapath=datapath,
               buffer_id=msg.buffer_id, in_port=in_port,
               actions=actions, data=msg.data if msg.buffer_id
                == datapath.ofproto.OFP_NO_BUFFER else None)
93          datapath.send_msg(out)
```

Il controller è stato sviluppato utilizzando il framework Ryu e il protocollo OpenFlow versione 1.3. Nel costruttore `__init__` abbiamo definito le principali strutture dati necessarie al funzionamento del controller, in particolare:

- `mac_to_port`, un dizionario utilizzato per apprendere dinamicamente l'associazione tra indirizzi MAC e porte degli switch;
- `allowed_pairs`, un insieme di coppie di indirizzi MAC autorizzate alla comunicazione, che rappresentano le slice di rete abilitate (H1–H3 e H2–H4).

Nel metodo `switch_features_handler` abbiamo installato una regola di *table-miss* con priorità minima, che inoltra al controller tutti i

pacchetti che non corrispondono ad alcuna regola presente nello switch. Questa regola è fondamentale, poiché consente al controller di ricevere i pacchetti sconosciuti e di decidere dinamicamente come gestirli. Il metodo `add_flow` è invece responsabile della creazione e dell'invio di messaggi `OFPFlowMod` agli switch, permettendo di installare regole di forwarding specificando:

- il *match*, ovvero le condizioni che un pacchetto deve soddisfare (ad esempio indirizzi MAC sorgente e destinazione e porta di ingresso);
- le *actions*, che definiscono l'operazione da eseguire sui pacchetti corrispondenti, come l'inoltro verso una determinata porta;
- la *priority* della regola, che ne stabilisce l'importanza rispetto alle altre.

In questo modo, una volta installata una regola di flusso, i pacchetti successivi con le stesse caratteristiche vengono gestiti direttamente dallo switch, senza coinvolgere il controller. Il metodo `packet_in_handler` gestisce invece i pacchetti ricevuti dal controller tramite eventi `PacketIn`. Il comportamento del controller segue i seguenti passaggi:

- **Apprendimento dei MAC**: il controller registra la porta di ingresso associata all'indirizzo MAC sorgente, memorizzandola in `mac_to_port[dpid][src]`;

- **Controllo delle coppie autorizzate**: viene verificato che la coppia (`src`, `dst`) appartenga all'insieme `allowed_pairs`; in caso contrario, il pacchetto viene bloccato;
- **Determinazione della porta di uscita**: se il MAC di destinazione è già noto, viene individuata la porta corretta per l'inoltro;
- **Installazione del flusso**: viene installata una regola nello switch tramite `add_flow`, evitando l'invio di pacchetti successivi al controller;
- **Inoltro del pacchetto**: il pacchetto viene infine inoltrato mediante un messaggio `PacketOut`.

Questo meccanismo consente di implementare un controllo centralizzato del traffico, garantendo l'isolamento tra le slice di rete e limitando le comunicazioni alle sole coppie di host autorizzate.

2.2.2 Verifica e Test

Per valutare il corretto funzionamento del meccanismo di slicing implementato, è stato innanzitutto eseguito un test di connettività globale mediante il comando `pingall`, il cui esito è riportato di seguito.

```
mininet> pingall
*** 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)
mininet>
```

I risultati ottenuti evidenziano che l'host H1 riesce a comunicare esclusivamente con H3, mentre H2 comunica unicamente con H4, in accordo con le politiche di slicing definite. Successivamente, sono stati effettuati test di connettività mirati, al fine di analizzare nel dettaglio lo scambio di pacchetti tra specifiche coppie di host.

```
mininet> h1 ping -c 4 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=2.19 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.063 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.155 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.120 ms

--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3031ms
rtt min/avg/max/mdev = 0.063/0.632/2.191/0.900 ms
```

Figure 2.1: Ping tra h1 e h3

```
mininet> h2 ping -c 4 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=1.08 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.061 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.119 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.113 ms

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3056ms
rtt min/avg/max/mdev = 0.061/0.343/1.081/0.426 ms
```

Figure 2.2: Ping tra h2 e h4

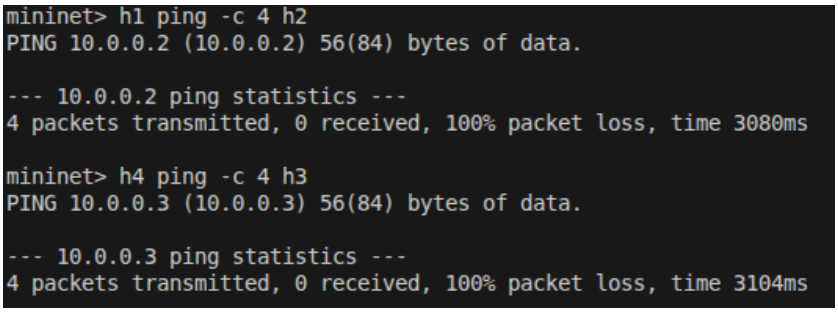
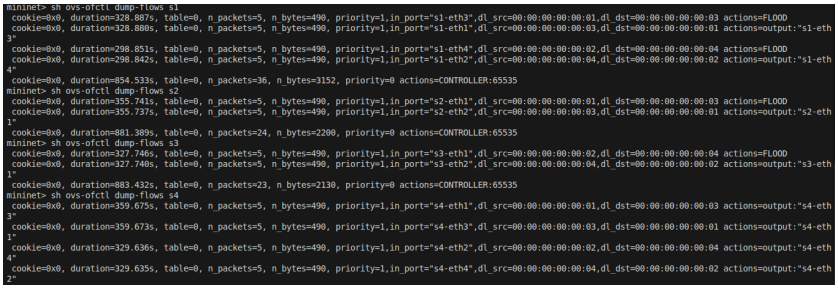


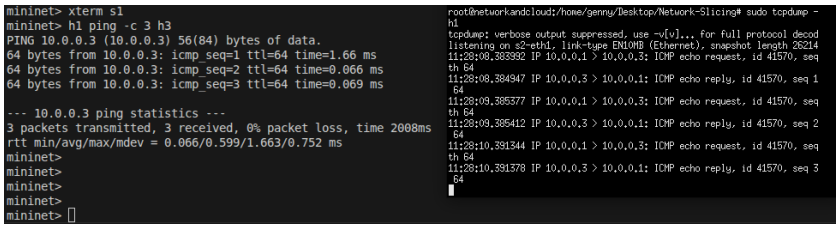
Figure 2.3: Ping tra h1 e h2 e tra h3 e h4

In un secondo momento, sono state esaminate le regole di forwarding installate sugli switch utilizzando il comando `sh ovs-ofctl dump-flows s`, dove `s` identifica lo switch di interesse, così da verificare la corretta installazione dei flussi da parte del controller.

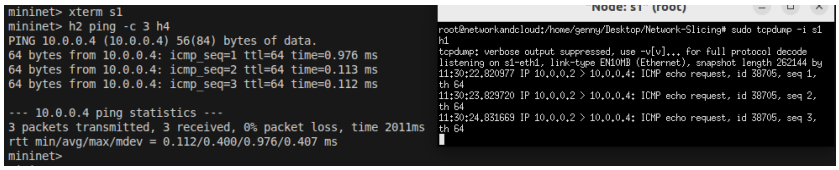


Per ottenere una verifica più accurata del comportamento della rete, è stato analizzato il traffico in transito sugli switch, con l'obiettivo di confermare che i pacchetti attraversassero gli slice dedicati. In particolare, per verificare che il traffico tra H1 e H3 utilizzasse il percorso superiore, è stato eseguito il comando `mininet> h1 ping -c 3 h3`; in contemporanea, il traffico sull'interfaccia `s1-eth2`, che collega lo switch s1 allo switch s2, è stato monitorato tramite `tcpdump`. L'output ha mostrato la presenza di pacchetti ICMP in transito, confermando che il traffico tra H1 e H3 percorre correttamente il ramo

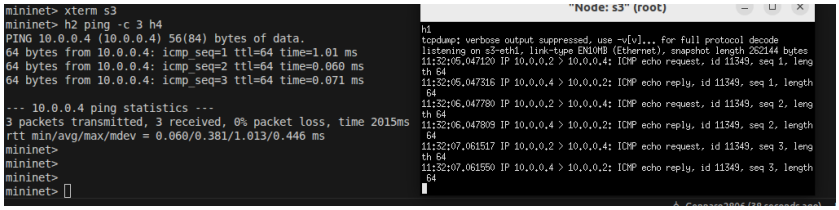
superiore della rete, come previsto dalla logica del controller.



In modo analogo, eseguendo il comando `mininet> h2 ping -c 3 h4` e analizzando il traffico sullo stesso percorso superiore, non sono stati rilevati pacchetti ICMP in transito, a conferma del fatto che il traffico tra H2 e H4 non utilizza tale percorso.



Infine, ripetendo il test di connettività tra H2 e H4 e monitorando il traffico sull'interfaccia `s3-eth1`, che collega lo switch s3 allo switch s1, tramite `tcpdump`, sono stati osservati pacchetti ICMP in transito. Questo risultato conferma che, in questo caso, il traffico tra H2 e H4 viene instradato correttamente lungo il percorso inferiore della rete.



2.3 Service Slicing

Il Service Slicing introduce una differenziazione dei flussi in base al servizio, assegnando priorità al traffico video identificato come UDP sulla porta 9999.

2.3.1 Controller Ryu

Il controller classifica i pacchetti in base al protocollo di trasporto e alla porta di destinazione. Il traffico video viene instradato su una slice dedicata con priorità elevata, mentre il traffico best-effort utilizza la slice a capacità ridotta.

```
1     from ryu.base import app_manager
2 from ryu.controller import ofp_event
3 from ryu.controller.handler import CONFIG_DISPATCHER,
    MAIN_DISPATCHER
4 from ryu.controller.handler import set_ev_cls
5 from ryu.ofproto import ofproto_v1_3
6 from ryu.lib.packet import packet
7 from ryu.lib.packet import ethernet
8 from ryu.lib.packet import ether_types
9 from ryu.lib.packet import udp
10 from ryu.lib.packet import tcp
11 from ryu.lib.packet import icmp
12
13
14 class ServiceSlicing(app_manager.RyuApp):
```

```
15      # Versione di OpenFlow utilizzata
16      OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
17
18      def __init__(self, *args, **kwargs):
19          super(ServiceSlicing, self).__init__(*args, **
                kwargs)
20
21          # Tabella MAC -> porta per ciascun datapath (
                switch)
22          # outport = self.mac_to_port[dpid][mac_address]
23          self.mac_to_port = {
24              1: {
25                  "00:00:00:00:00:01": 3,
26                  "00:00:00:00:00:02": 4
27              },
28              4: {
29                  "00:00:00:00:00:03": 3,
30                  "00:00:00:00:00:04": 4
31              },
32          }
33
34          # Porta UDP utilizzata per identificare una
                specifica slice
35          self.slice_TCport = 9999
36
37          # Mappatura slice -> porta di uscita per ciascun
                switch
38          # outport = self.slice_ports[dpid][slicenumber]
```

```
39          self.slice_ports = {
40              1: {1: 1, 2: 2},
41              4: {1: 1, 2: 2}
42          }
43
44          # Switch di bordo (edge / end switches)
45          self.end_swtiches = [1, 4]
46
47      @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
            CONFIG_DISPATCHER)
48      def switch_features_handler(self, ev):
49          """Gestione evento di connessione dello switch"""
50          datapath = ev.msg.datapath
51          ofproto = datapath.ofproto
52          parser = datapath.ofproto_parser
53
54          # Flow di default (table-miss): inoltra i
               pacchetti al controller
55          match = parser.OFPMatch()
56          actions = [
57              parser.OFPActionOutput(
58                  ofproto.OFPP_CONTROLLER,
59                  ofproto.OFPCML_NO_BUFFER
60              )
61          ]
62          self.add_flow(datapath, 0, match, actions)
63
64      def add_flow(self, datapath, priority, match, actions
```

```
    ):
65      """Installa una flow entry nello switch"""
66      ofproto = datapath.ofproto
67      parser = datapath.ofproto_parser
68
69      inst = [
70          parser.OFPInstructionActions(
71              ofproto.OFPIT_APPLY_ACTIONS,
72              actions
73          )
74      ]
75
76      mod = parser.OFPFlowMod(
77          datapath=datapath,
78          priority=priority,
79          match=match,
80          instructions=inst
81      )
82
83      datapath.send_msg(mod)
84
85  def _send_package(self, msg, datapath, in_port,
      actions):
86      """Invia il pacchetto immediatamente tramite
          PacketOut"""
87      data = None
88      ofproto = datapath.ofproto
89
```

```
90          if msg.buffer_id == ofproto.OFP_NO_BUFFER:
91              data = msg.data
92
93          out = datapath.ofproto_parser.OFPPacketOut(
94              datapath=datapath,
95              buffer_id=msg.buffer_id,
96              in_port=in_port,
97              actions=actions,
98              data=data,
99          )
100
101         datapath.send_msg(out)
102
103     @set_ev_cls(ofp_event.EventOFPPacketIn,
            MAIN_DISPATCHER)
104     def _packet_in_handler(self, ev):
105         """Gestione dei pacchetti ricevuti dal controller
                """
106         msg = ev.msg
107         datapath = msg.datapath
108         ofproto = datapath.ofproto
109         in_port = msg.match["in_port"]
110
111         pkt = packet.Packet(msg.data)
112         eth = pkt.get_protocol(ethernet.ethernet)
113
114         # Ignora pacchetti LLDP
115         if eth.ethertype == ether_types.ETH_TYPE_LLDP:
```

```
116             return
117
118         dst = eth.dst
119         src = eth.src
120         dpid = datapath.id
121
122         # Forwarding basato su MAC conosciuti
123         if dpid in self.mac_to_port:
124             if dst in self.mac_to_port[dpid]:
125                 out_port = self.mac_to_port[dpid][dst]
126                 actions = [
127                     datapath.ofproto_parser.
                            OFPActionOutput(out_port)
128                 ]
129
130                 match = datapath.ofproto_parser.OFPMatch(
131                     eth_dst=dst
132                 )
133
134                 self.add_flow(datapath, 1, match, actions
                        )
135                 self._send_package(msg, datapath, in_port
                        , actions)
136
137             # UDP verso porta slice dedicata
138             elif (
139                 pkt.get_protocol(udp.udp) and
140                 pkt.get_protocol(udp.udp).dst_port ==
```

```
                self.slice_TCport
141         ):
142             slice_number = 1
143             out_port = self.slice_ports[dpid][
                    slice_number]
144
145             match = datapath.ofproto_parser.OFPMatch(
146                 in_port=in_port,
147                 eth_dst=dst,
148                 eth_type=ether_types.ETH_TYPE_IP,
149                 ip_proto=0x11,  # UDP
150                 udp_dst=self.slice_TCport,
151             )
152
153             actions = [
154                 datapath.ofproto_parser.
                        OFPActionOutput(out_port)
155             ]
156
157             self.add_flow(datapath, 2, match, actions
                    )
158             self._send_package(msg, datapath, in_port
                    , actions)
159
160         # UDP verso altre porte
161         elif pkt.get_protocol(udp.udp):
162             slice_number = 2
163             out_port = self.slice_ports[dpid][
```

```
                    slice_number]
164
165                 match = datapath.ofproto_parser.OFPMatch(
166                     in_port=in_port,
167                     eth_dst=dst,
168                     eth_src=src,
169                     eth_type=ether_types.ETH_TYPE_IP,
170                     ip_proto=0x11,  # UDP
171                     udp_dst=pkt.get_protocol(udp.udp).
                            dst_port,
172                 )
173
174                 actions = [
175                     datapath.ofproto_parser.
                            OFPActionOutput(out_port)
176                 ]
177
178                 self.add_flow(datapath, 1, match, actions
                        )
179                 self._send_package(msg, datapath, in_port
                        , actions)
180
181             # Traffico TCP
182             elif pkt.get_protocol(tcp.tcp):
183                 slice_number = 2
184                 out_port = self.slice_ports[dpid][
                        slice_number]
185
```

```
186                 match = datapath.ofproto_parser.OFPMatch(
187                     in_port=in_port,
188                     eth_dst=dst,
189                     eth_src=src,
190                     eth_type=ether_types.ETH_TYPE_IP,
191                     ip_proto=0x06,  # TCP
192                 )
193
194                 actions = [
195                     datapath.ofproto_parser.
                        OFPActionOutput(out_port)
196                 ]
197
198                 self.add_flow(datapath, 1, match, actions
                    )
199                 self._send_package(msg, datapath, in_port
                    , actions)
200
201             # Traffico ICMP
202             elif pkt.get_protocol(icmp.icmp):
203                 slice_number = 2
204                 out_port = self.slice_ports[dpid][
                    slice_number]
205
206                 match = datapath.ofproto_parser.OFPMatch(
207                     in_port=in_port,
208                     eth_dst=dst,
209                     eth_src=src,
```

```
210                     eth_type=ether_types.ETH_TYPE_IP,
211                     ip_proto=0x01,  # ICMP
212                 )
213
214                 actions = [
215                     datapath.ofproto_parser.
                        OFPActionOutput(out_port)
216                 ]
217
218                 self.add_flow(datapath, 1, match, actions
                    )
219                 self._send_package(msg, datapath, in_port
                    , actions)
220
221         # Flood sui nodi non terminali
222         elif dpid not in self.end_swtiches:
223             out_port = ofproto.OFPP_FLOOD
224             actions = [
225                 datapath.ofproto_parser.OFPActionOutput(
                        out_port)
226             ]
227
228             match = datapath.ofproto_parser.OFPMatch(
229                 in_port=in_port
230             )
231
232             self.add_flow(datapath, 1, match, actions)
233             self._send_package(msg, datapath, in_port,
```

```
            actions)
```

Nel costruttore `__init__` vengono inizializzate le principali strutture dati necessarie alla gestione dello slicing del traffico. In particolare, abbiamo definito:

- `mac_to_port`, un dizionario statico che associa indirizzi MAC alle porte di uscita sugli switch terminali (ID 1 e 4), utilizzato per determinare rapidamente la porta corretta verso un host noto;
- `slice_video`, una variabile che identifica la slice dedicata al traffico video;
- `slice_TCport`, che specifica la porta UDP 9999, utilizzata come criterio per riconoscere il traffico video;
- `ports`, una mappa che associa a ciascuna slice (1 o 2) la porta di uscita corretta per ogni switch;
- `end_switches`, una lista contenente gli switch terminali (1 e 4), ovvero quelli direttamente connessi agli host, sui quali viene applicata la logica di slicing.

Per quanto riguarda la gestione degli switch, al momento della connessione di ciascun datapath al controller viene installata una regola di *table-miss*, che inoltra al controller tutti i pacchetti che non corrispondono ad alcuna regola presente nella tabella di flusso. Questo mec-

canismo consente al controller di analizzare e gestire dinamicamente i pacchetti sconosciuti.

Il metodo `add_flow` è responsabile della creazione e dell'invio di messaggi `OFPFlowMod` agli switch, permettendo l'installazione di nuove regole di forwarding. Ogni regola specifica:

- le condizioni di *match*, come il tipo di protocollo, la porta UDP o gli indirizzi MAC;
- le *actions*, che definiscono l'operazione da eseguire sui pacchetti corrispondenti, ad esempio l'inoltro su una determinata porta;
- la *priority*, che determina l'ordine di valutazione delle regole.

Il metodo `packet_in_handler` gestisce i pacchetti ricevuti tramite eventi `PacketIn` e applica la logica di slicing in base alle caratteristiche del traffico:

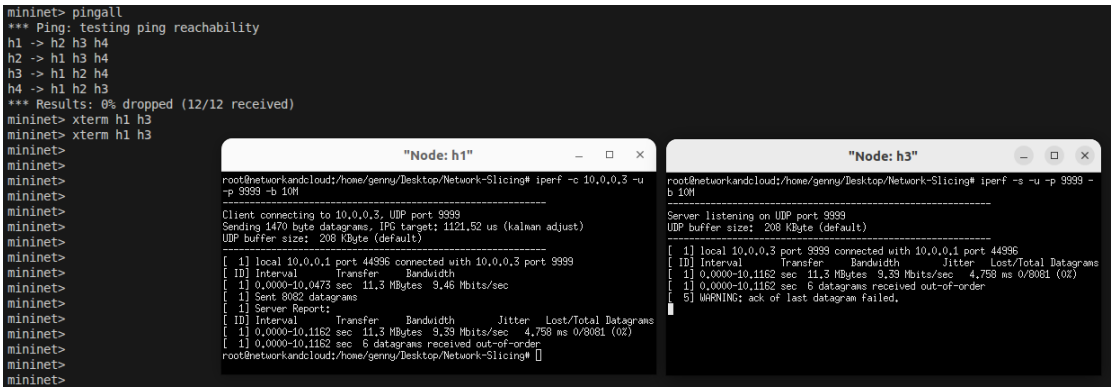
- **MAC noto**: se il pacchetto è destinato a un indirizzo MAC presente in `mac_to_port`, viene inoltrato direttamente sulla porta associata;
- **Traffico video**: i pacchetti UDP diretti alla porta 9999 vengono classificati come traffico prioritario e instradati sulla slice 1;
- **Traffico UDP generico**: i pacchetti UDP destinati a porte diverse dalla 9999 vengono instradati sulla slice 2, considerata come percorso standard;

- **Traffico TCP o ICMP**: anch'esso viene instradato sulla slice 2, seguendo la logica di separazione dei servizi;

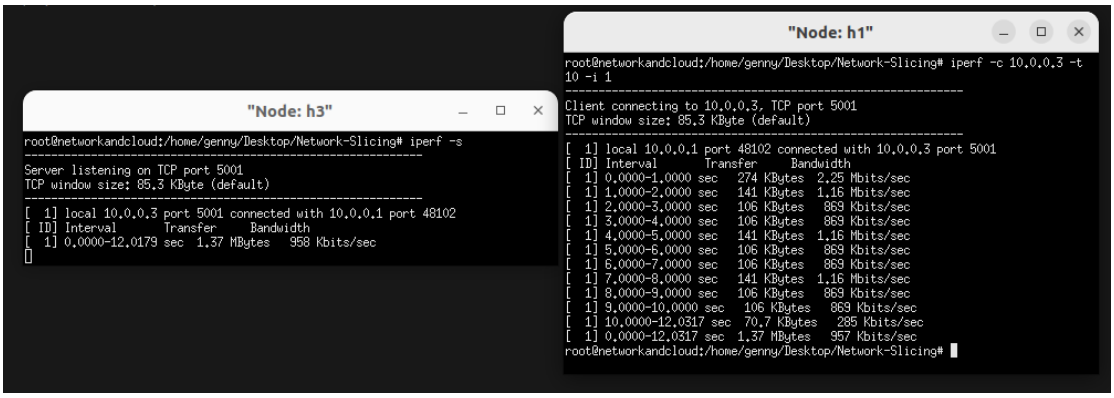
- **Switch non terminali**: se il pacchetto proviene da uno switch non incluso in `end_switches`, viene inoltrato in modalità flood, al fine di garantire la corretta propagazione iniziale del traffico.

2.3.2 Verifica e Test

Per verificare il corretto funzionamento del controller sono stati eseguiti diversi test sperimentali. Oltre ai tradizionali test di connettività basati sul comando `ping`, sono state effettuate verifiche specifiche per accertare che il controller fosse in grado di riconoscere correttamente il tipo di traffico e di instradarlo secondo le politiche di slicing definite. In particolare, è stato generato traffico UDP tra gli host H1 e H3, simulando uno scenario client-server, mediante l'utilizzo dello strumento `iperf`. A tal fine, sul nodo H3 è stato avviato un server UDP in ascolto sulla porta 9999 tramite il comando `iperf -s -u -p 9999`, mentre su H1 è stato eseguito il comando `iperf -c 10.0.0.3 -u -p 9999 -b 10M`, generando un flusso di traffico video con banda elevata.

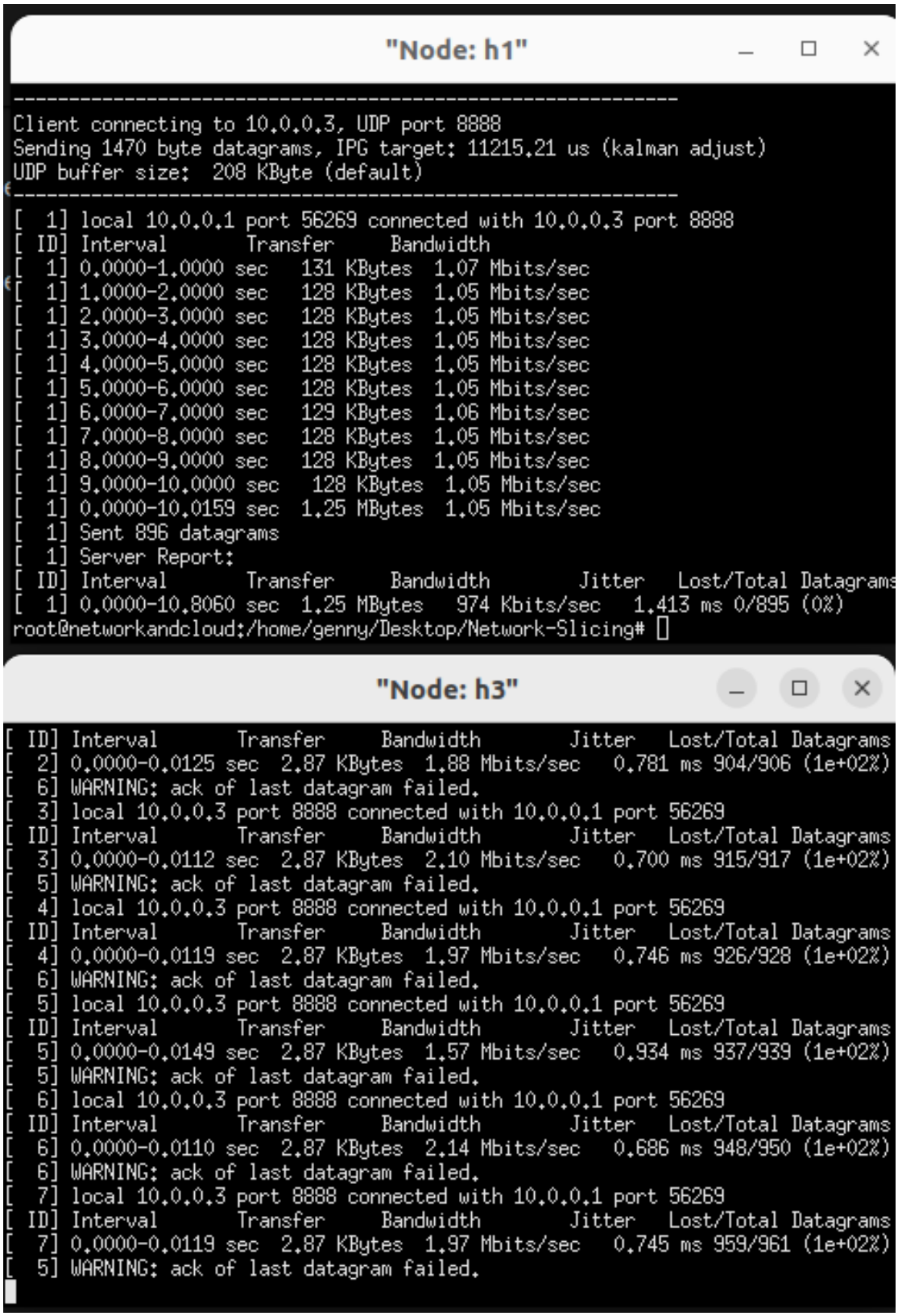


I risultati ottenuti mostrano che il controller è in grado di riconoscere correttamente il traffico video e di instradarlo lungo il percorso prioritario ad alta capacità. In particolare, si osserva che la banda di trasmissione misurata risulta prossima ai 10 Mbps, valore coerente con la configurazione dei link dedicati a tale slice. In modo analogo, è possibile simulare traffico TCP, che non viene classificato come prioritario; di conseguenza, questo tipo di traffico viene instradato attraverso il percorso standard a capacità ridotta. Quindi:



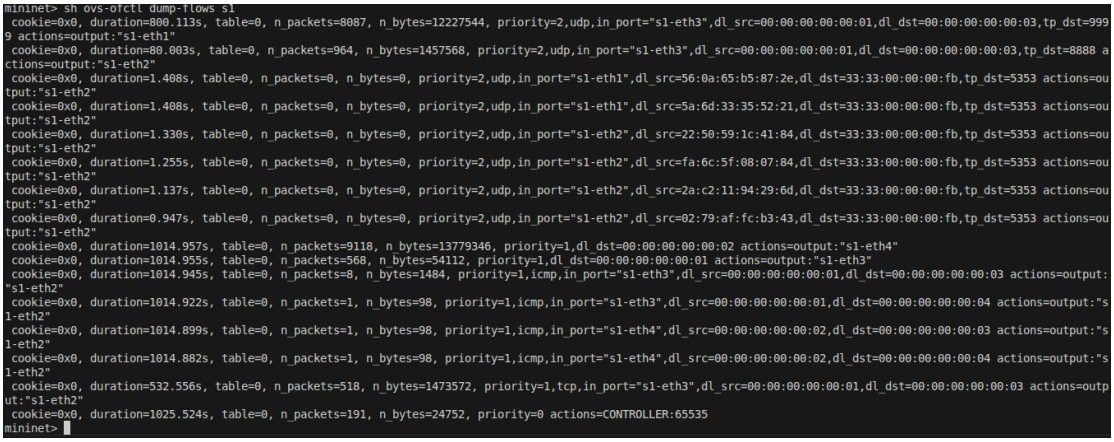
Dai risultati ottenuti è possibile dedurre che il traffico TCP, ad esempio sulla porta 8888, non viene classificato come prioritario dal controller. Di conseguenza, tale traffico non viene instradato attraverso

lo slice ad alta capacità, ma viene deviato sullo slice a bassa banda, come confermato dalla velocità di trasmissione osservata, pari a circa 1 Mbps. Un'analoga verifica può essere effettuata anche per il traffico UDP destinato a porte diverse dalla 9999, che non viene riconosciuto come traffico prioritario:



Un ulteriore tipo di test è stato condotto utilizzando il comando `sh`

`ovs-ofctl dump-flows s*`, che consente di visualizzare tutti i flussi installati sui vari switch. Questo permette di verificare in dettaglio le regole di forwarding configurate dal controller e di controllare che il traffico venga instradato secondo le slice definite.



2.3.3 Interpretazione regole di flusso su S1

Regola 1

```
1    REGOLA 1
2 cookie=0x0, duration=20.386s, table=0, n_packets=8204,
   n_bytes=12404448, priority=2,udp,in_port="s1-eth3",
   dl_dst=00:00:00:00:00:03,tp_dst=9999 actions=output:"s1
   -eth1"
```

La regola in esame ha priorità 2, la più alta nello switch, e corrisponde ai pacchetti UDP in ingresso sulla porta `s1-eth3`, con destinazione MAC `00:00:00:00:00:03` e porta UDP di destinazione 9999. L'azione associata consiste nell'inoltrare i pacchetti sull'interfaccia `s1-eth1`. Le statistiche mostrano che sono stati trasmessi

8105 pacchetti, per un totale di circa 12 MB di dati. Con tutta probabilità, questa regola corrisponde allo slice dedicato al traffico video ad alta priorità.

Regola 2

```
1    REGOLA 2
2 cookie=0x0, duration=20.386s, table=0, n_packets=8204,
   n_bytes=12404448, priority=2,udp,in_port="s1-eth3",
   dl_dst=00:00:00:00:00:03,tp_dst=9999 actions=output:"s1
   -eth1"
```

La regola considerata ha priorità 1, inferiore rispetto alla precedente, e corrisponde ai pacchetti UDP diretti alla porta 8888, ad esempio traffico di test o appartenente allo slice "standard". I pacchetti vengono inoltrati dall'interfaccia `s1-eth3` verso `s1-eth2`. Il numero di pacchetti e di byte trasmessi risulta inferiore rispetto a quanto osservato per la regola del traffico video, indicando che si tratta molto probabilmente di traffico non prioritario o appartenente a uno slice a banda ridotta.

Regola 3

```
1    REGOLA 3
2 cookie=0x0, duration=568.662s, table=0, n_packets=3,
   n_bytes=294, priority=1,icmp,in_port="s1-eth4",dl_src
   =00:00:00:00:00:02,dl_dst=00:00:00:00:00:04 actions=
   output:"s1-eth2"
```

Una delle regole analizzate riguarda il traffico ICMP (simile al traffico TCP in termini di priorità), con priorità 1. In questo caso, i pacchetti provenienti dall'interfaccia `s1-eth4`, con indirizzo sorgente `00:00:00:00:00:02` e destinazione `00:00:00:00:00:04`, vengono inoltrati verso l'interfaccia `s1-eth2`. La regola registra solo pochi pacchetti (3) per un totale di 294 byte, suggerendo che si tratta di traffico non prioritario, probabilmente associato a uno slice a banda inferiore dedicato al traffico "standard".

Conclusione

Il controller installa flussi con priorità differenziate a seconda del tipo di traffico. Il traffico UDP video, destinato alla porta 9999, viene instradato su un'interfaccia dedicata (`s1-eth1`), corrispondente a una slice con priorità elevata. Al contrario, il traffico diretto ad altre porte, come UDP 8888 o TCP generico, viene instradato su `s1-eth2`, probabilmente associata a uno slice best-effort. Le statistiche relative al numero di pacchetti e ai byte trasmessi confermano che le regole sono attive e che il traffico ha effettivamente attraversato i percorsi previsti.

2.3.4 Interpretazione regole s2 e s3

```
mininet> sh ovs-ofctl dump-flows s2
 cookie=0x0, duration=510.634s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s2-eth1",dl_src=12:df:26:63:37:b2,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=
output:"s2-eth2"
 cookie=0x0, duration=510.255s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s2-eth2",dl_src=fa:c0:1c:3f:17:2c,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=
output:"s2-eth2"
 cookie=0x0, duration=506.360s, table=0, n_packets=658, n_bytes=980698, priority=1,in_port="s2-eth1" actions=FLOOD
 cookie=0x0, duration=504.311s, table=0, n_packets=451412, n_bytes=38153286, priority=1,in_port="s2-eth2" actions=FLOOD
 cookie=0x0, duration=512.410s, table=0, n_packets=4, n_bytes=354, priority=0 actions=CONTROLLER:65535
mininet> sh ovs-ofctl dump-flows s3
 cookie=0x0, duration=561.222s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s3-eth2",dl_src=d2:a7:d4:11:d2:ab,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=
FLOOD
 cookie=0x0, duration=560.839s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s3-eth2",dl_src=da:46:73:7c:70:2a,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=
FLOOD
 cookie=0x0, duration=560.767s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s3-eth2",dl_src=12:df:26:63:37:b2,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=
FLOOD
 cookie=0x0, duration=560.718s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s3-eth1",dl_src=9a:2b:fe:52:a5:4d,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=
FLOOD
 cookie=0x0, duration=560.396s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s3-eth1",dl_src=3e:43:a3:b9:99:f5,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=
FLOOD
 cookie=0x0, duration=555.987s, table=0, n_packets=23, n_bytes=2276, priority=1,in_port="s3-eth2" actions=FLOOD
 cookie=0x0, duration=555.475s, table=0, n_packets=528180, n_bytes=45376628, priority=1,in_port="s3-eth1" actions=FLOOD
 cookie=0x0, duration=562.549s, table=0, n_packets=7, n_bytes=675, priority=0 actions=CONTROLLER:65535
```

L'analisi degli switch s2 e s3 evidenzia la presenza di flussi con `priority=1` e azione `FLOOD`, ossia i pacchetti vengono inoltrati su tutte le porte tranne quella di ingresso. Ciò indica che il traffico che arriva su queste porte non corrisponde ad alcun flusso specifico di forwarding e viene quindi propagato come broadcast. Inoltre, è sempre presente un flusso con `priority=0` e azione `CONTROLLER:6553`, mediante il quale i pacchetti vengono inviati al controller. Questa regola è necessaria affinché il controller possa decidere dinamicamente come gestire pacchetti nuovi o non riconosciuti.

2.3.5 Interpretazione regole s4

```
mininet> sh ovs-ofctl dump-flows s4
 cookie=0x0, duration=402.715s, table=0, n_packets=585313, n_bytes=50059682, priority=2,dl_dst=00:00:00:00:00:03 actions=output:"s4-eth1"
 cookie=0x0, duration=350.554s, table=0, n_packets=648, n_bytes=979776, priority=2,dl_dst=00:00:00:00:00:04 actions=output:"s4-eth2"
 cookie=0x0, duration=94.871s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s4-eth1",dl_src=d2:a7:d4:11:d2:ab,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=o
utput:"s4-eth2"
 cookie=0x0, duration=94.551s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s4-eth1",dl_src=12:df:26:63:37:b2,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=o
utput:"s4-eth2"
 cookie=0x0, duration=94.482s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s4-eth2",dl_src=9a:2b:fe:52:a5:4d,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=o
utput:"s4-eth2"
 cookie=0x0, duration=94.414s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s4-eth2",dl_src=6a:fc:50:6f:cf:4e,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=o
utput:"s4-eth2"
 cookie=0x0, duration=94.143s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s4-eth2",dl_src=3e:43:a3:b9:99:f5,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=o
utput:"s4-eth2"
 cookie=0x0, duration=94.141s, table=0, n_packets=0, n_bytes=0, priority=1,udp,in_port="s4-eth2",dl_src=fa:c0:1c:3f:17:2c,dl_dst=33:33:00:00:00:fb,tp_dst=5353 actions=o
utput:"s4-eth2"
 cookie=0x0, duration=600.321s, table=0, n_packets=79, n_bytes=8517, priority=0 actions=CONTROLLER:65535
```

Il comportamento osservato risulta coerente con la logica implementata dal controller: sugli switch terminali (s1 e s4) sono presenti flussi

specifici con priorità 1 o 2, corrispondenti alle regole di slicing definite, mentre sugli switch intermedi (s2 e s3) si osserva solo flooding, poiché il controller li gestisce come switch di transito privi di regole specifiche.

Chapter 3

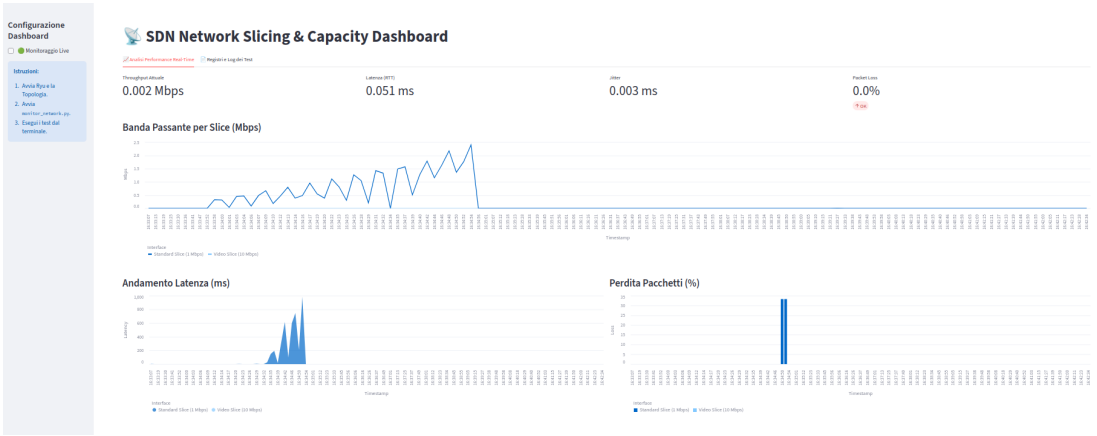
Performance analysis

3.1 Dashboard streamlit

Al fine di analizzare e validare il comportamento della rete SDN implementata tramite Mininet e controller Ryu, è stata sviluppata una dashboard interattiva basata su Streamlit. Tale dashboard ha lo scopo di fornire una visione centralizzata e in tempo reale delle principali metriche di performance della rete, tra cui throughput, latenza, jitter e packet loss, consentendo di valutare l'efficacia delle politiche di controllo e slicing adottate. La dashboard permette inoltre di visualizzare i risultati dei test di rete eseguiti da terminale, facilitando l'analisi sperimentale e il confronto tra diversi scenari di configurazione.

L'architettura della dashboard si basa su diversi moduli cooperanti. Il file `dashboard.py` rappresenta l'interfaccia grafica principale e si occupa della visualizzazione delle metriche e dei risultati

dei test. Il modulo `monitor_network.py` opera in background ed è responsabile del monitoraggio continuo della comunicazione tra la topologia Mininet e i controller Ryu, raccogliendo dati utili alla valutazione delle prestazioni della rete. Il file `run_tests.py` automatizza l'esecuzione dei test di rete, includendo misure di connettività tramite `ping` tra gli host e test di banda mediante `iperf`. Infine, il modulo `capacity_test.py` esegue un test specifico sullo slice inferiore, configurato con una capacità massima di 1 Mbps, con l'obiettivo di verificare il corretto comportamento del sistema in condizioni di congestione, evidenziando la comparsa di packet loss al superamento della soglia di banda assegnata.



3.1.1 Implementazione

Dashboard

```
    import streamlit as st
import pandas as pd
import time
```

```
import os

# Configurazione della pagina per una visualizzazione ampia
st.set_page_config(
    page_title="SDN Slicing Dashboard",
    layout="wide",
    page_icon=""
)

# Percorsi dei file
DATA_FILE = "traffic_data.csv"
STANDARD_LOG = "test_results.log"
CAPACITY_LOG = "capacity_test_results.log"

# --- SIDEBAR ---
st.sidebar.title("Configurazione Dashboard")
auto_refresh = st.sidebar.checkbox(" Monitoraggio Live", value=True)
st.sidebar.info("""
**Istruzioni:**
1. Avvia Ryu e la Topologia.
2. Avvia `monitor_network.py`.
3. Esegui i test dal terminale.
""")

st.title(" SDN Network Slicing & Capacity Dashboard")

```

```
30 # Funzione per caricare i dati CSV
31 def load_data():
32     if not os.path.exists(DATA_FILE):
33         return pd.DataFrame()
34     return pd.read_csv(DATA_FILE)
35
36 df = load_data()
37
38 # --- DEFINIZIONE TAB ---
39 tab1, tab2 = st.tabs([" Analisi Performance Real-Time", "
       Registri e Log dei Test"])
40
41 # --- TAB 1: GRAFICI E METRICHE ---
42 with tab1:
43     if not df.empty:
44         # Metriche in primo piano (ultimi valori rilevati
              )
45         last_row = df.iloc[-1]
46         m1, m2, m3, m4 = st.columns(4)
47
48         m1.metric("Throughput Attuale", f"{last_row['Mbps
              ']} Mbps")
49
50         # Mostra latenza e jitter solo se il link e'
              attivo (loss < 100)
51         if last_row['Loss'] < 100:
52             m2.metric("Latenza (RTT)", f"{last_row['
                  Latency']} ms")
```

```
53              m3.metric("Jitter", f"{last_row['Jitter']} ms
                    ")
54          else:
55              m2.metric("Latenza (RTT)", "N/A")
56              m3.metric("Jitter", "N/A")
57
58          m4.metric("Packet Loss", f"{last_row['Loss']}%",
59                    delta=" CRITICO" if last_row['Loss'] >
                        10 else "OK",
60                    delta_color="inverse")
61
62          # --- Grafico Banda ---
63          st.subheader("Banda Passante per Slice (Mbps)")
64          st.line_chart(df, x="Timestamp", y="Mbps", color
                ="Interface")
65
66          # --- Grafici Qualita' (Latenza e Loss) ---
67          col_left, col_right = st.columns(2)
68
69          with col_left:
70              st.subheader("Andamento Latenza (ms)")
71              # Puliamo i dati per non mostrare picchi d'
                    errore quando il link cade
72              df_clean = df[df['Loss'] < 100]
73              st.area_chart(df_clean, x="Timestamp", y="
                    Latency", color="Interface")
74
75          with col_right:
```

```
76              st.subheader("Perdita Pacchetti (%)")
77              st.bar_chart(df, x="Timestamp", y="Loss",
                    color="Interface")
78      else:
79          st.warning(" Nessun dato rilevato nel file CSV.
                Assicurati che 'monitor_network.py' sia in
                esecuzione con sudo.")
80
81  # --- TAB 2: VISUALIZZAZIONE LOG ---
82  with tab2:
83      st.header("Visualizzazione Risultati Test")
84
85      # Selezione del tipo di test da visualizzare
86      test_choice = st.radio(
87          "Seleziona il report da visualizzare:",
88          ["Verifica Standard (iperf/ping)", "Capacity
                Stress Test (Analisi Saturazione)"],
89          horizontal=True
90      )
91
92      # Determina quale file aprire
93      selected_file = STANDARD_LOG if "Verifica" in
            test_choice else CAPACITY_LOG
94
95      col_btn1, col_btn2 = st.columns([1, 5])
96      with col_btn1:
97          if st.button(" Aggiorna Log"):
98              st.rerun()
```

```
99
100     if os.path.exists(selected_file):
101         with open(selected_file, "r") as f:
102             log_content = f.read()
103             # Visualizzazione del log in un'area di testo
                    spaziosa
104             st.text_area(f"Contenuto di: {selected_file
                   }", log_content, height=600)
105
106             # Funzione di download per la relazione
107             st.download_button(
108                 label=" Scarica Log per la Relazione",
109                 data=log_content,
110                 file_name=selected_file,
111                 mime="text/plain"
112             )
113     else:
114         st.info(f"Il file `{selected_file}` non esiste
                ancora. Esegui lo script corrispondente nel
                terminale.")
115
116 # --- LOGICA DI REFRESH ---
117 if auto_refresh:
118     time.sleep(1)
119     st.rerun()
```

Il file della dashboard Streamlit costituisce il punto di accesso principale per l'analisi visuale delle performance della rete SDN. L'applicazione

fornisce un'interfaccia web interattiva che consente di monitorare in tempo quasi reale le metriche di throughput, latenza, jitter e packet loss, a partire dai dati raccolti e salvati in formato CSV dal modulo di monitoraggio.

La dashboard è strutturata in due sezioni principali. La prima è dedicata all'analisi delle performance in tempo reale e presenta indicatori sintetici delle ultime misurazioni disponibili, affiancati da grafici temporali che mostrano l'andamento della banda passante per slice e la qualità del collegamento. Un aspetto rilevante è la gestione dei casi di link non attivo, in cui valori di latenza e jitter vengono esclusi dalla visualizzazione per evitare interpretazioni fuorvianti dei dati.

La seconda sezione consente la consultazione dei log generati dai test eseguiti da terminale, distinguendo tra test standard (ping e iperf) e test di capacità. I risultati vengono caricati dinamicamente dai file di log e possono essere visualizzati direttamente nella dashboard o scaricati per l'inclusione nella documentazione sperimentale.

Infine, la dashboard implementa un meccanismo di aggiornamento automatico opzionale che permette di mantenere la visualizzazione sincronizzata con l'evoluzione dello stato della rete, rendendo lo strumento particolarmente adatto al monitoraggio continuo durante le fasi di test e validazione.

Monitor network

```
1    import time
```

```
import csv
import os
import subprocess

DATA_FILE = "traffic_data.csv"

# Associazioni: Interfaccia -> [Etichetta, Host Sorgente, IP Destinazione]
INTERFACES = {
    "s1-eth1": {"label": "Video Slice (10 Mbps)", "src": "h1", "target": "10.0.0.3"},
    "s1-eth2": {"label": "Standard Slice (1 Mbps)", "src": "h2", "target": "10.0.0.4"}
}

THRESHOLD_MBPS = 0.05
HEARTBEAT_INTERVAL = 5

def get_host_pid(host):
    pid = subprocess.getoutput(f"ps -eo pid,cmd | grep 'mininet:{host}' | grep -v grep | awk '{{print $1}}'").strip()
    return pid

def get_tx_bytes(interface):
    path = f"/sys/class/net/{interface}/statistics/tx_bytes"
    try:
```

```
24          with open(path, "r") as f: return int(f.read())
25      except: return None
26
27  def get_performance_stats(src_host, target_ip):
28      """Esegue il ping dall'interno del namespace dell'
            host sorgente."""
29      pid = get_host_pid(src_host)
30      if not pid: return 0, 0, 100
31
32      # Usiamo ping standard con timeout brevi per non
            bloccare il monitor
33      cmd = f"sudo nsenter -t {pid} -n ping -c 3 -i 0.2 -W
            1 {target_ip}"
34      res = subprocess.getoutput(cmd)
35
36      try:
37          # Parsing rtt min/avg/max/mdev =
                0.052/0.062/0.072/0.010 ms
38          if "avg" in res:
39              stats_line = [line for line in res.split('\n
                    ') if 'rtt' in line][0]
40              values = stats_line.split('=')[1].strip().
                    split('/')
41              latency = float(values[1])
42              jitter = float(values[3].split()[0]) # mdev e
                    ' una buona approssimazione del jitter
43
```

```
44              loss_line = [line for line in res.split('\n')
                    if 'packet loss' in line][0]
45              loss = float(loss_line.split('%')[0].split()
                   [-1])
46              return latency, jitter, loss
47          else:
48              return 0, 0, 100
49      except:
50          return 0, 0, 100
51
52  def monitor():
53      with open(DATA_FILE, "w", newline='') as f:
54          writer = csv.writer(f)
55          writer.writerow(["Timestamp", "Interface", "Mbps
               ", "Latency", "Jitter", "Loss"])
56
57      print("Monitoraggio Intelligente (Ping dai Namespace)
            avviato...")
58      prev_bytes = {iface: get_tx_bytes(iface) for iface in
            INTERFACES}
59      last_write_time = 0
60      was_active = {iface: False for iface in INTERFACES}
61
62      while True:
63          time.sleep(1)
64          current_time_str = time.strftime("%H:%M:%S")
65          current_time_unix = time.time()
66          data_to_write = []
```

```
67          should_heartbeat = (current_time_unix -
               last_write_time) >= HEARTBEAT_INTERVAL
68
69          for iface, cfg in INTERFACES.items():
70              curr = get_tx_bytes(iface)
71              if curr is None: continue
72
73              mbps = round(((curr - prev_bytes[iface]) * 8)
                    / 1000000.0, 3)
74              prev_bytes[iface] = curr
75
76              if mbps > THRESHOLD_MBPS or was_active[iface]
                    or should_heartbeat:
77                  lat, jit, loss = get_performance_stats(
                       cfg['src'], cfg['target'])
78                  data_to_write.append([current_time_str,
                       cfg['label'], mbps, lat, jit, loss])
79                  was_active[iface] = (mbps >
                       THRESHOLD_MBPS)
80
81          if data_to_write:
82              with open(DATA_FILE, "a", newline='') as f:
83                  writer = csv.writer(f)
84                  writer.writerows(data_to_write)
85              last_write_time = current_time_unix
86
87  if __name__ == "__main__":
88      if os.geteuid() != 0: print("Usa sudo!"); exit()
```

```
89    monitor()
```

Il modulo `monitor_network.py` è responsabile della raccolta continua delle metriche di performance della rete e rappresenta il collegamento tra l'infrastruttura Mininet, i controller Ryu e la dashboard Streamlit. Il suo compito principale è quello di misurare in modo non invasivo il traffico e la qualità del collegamento per ciascuna slice di rete, salvando i risultati in un file CSV successivamente utilizzato per la visualizzazione.

Il monitoraggio avviene a livello di interfaccia di switch, associando a ciascuna interfaccia una specifica slice e una coppia di host sorgente e destinazione. In particolare, per ogni slice vengono raccolte le seguenti informazioni:

- throughput, calcolato a partire dai byte trasmessi sull'interfaccia;
- latenza media (RTT), jitter e packet loss, misurati tramite ping;
- timestamp della rilevazione, utile per l'analisi temporale delle prestazioni.

Un aspetto rilevante del modulo è l'esecuzione dei comandi di misura direttamente all'interno dei namespace di rete degli host Mininet. Questo approccio, ottenuto tramite l'uso di `nsenter`, consente di effettuare misurazioni realistiche, evitando interferenze con lo stack di rete dell'host fisico e garantendo coerenza con la topologia emulata.

Per ridurre il carico computazionale e la quantità di dati prodotti, il monitor implementa una logica di campionamento intelligente:

- le metriche di qualità vengono calcolate solo quando il traffico supera una soglia minima di throughput;
- viene comunque forzata una scrittura periodica (heartbeat) per mantenere aggiornata la visualizzazione;
- vengono gestiti esplicitamente i casi di link inattivo, assegnando valori di packet loss pari al 100%.

Infine, il modulo richiede l'esecuzione con privilegi di amministratore, necessari per l'accesso alle statistiche di rete del sistema e ai namespace Mininet. Il file CSV generato costituisce la base dati condivisa con la dashboard e consente un monitoraggio continuo e sincronizzato dello stato della rete durante l'esecuzione dei test.

Run_tests.py

```
1     import subprocess
2 import time
3 import os
4 import sys
5
6 LOG_FILE = "test_results.log"
7
8 def fix_perms(path):
9     if os.path.exists(path) and "SUDO_UID" in os.environ:
```

```
10          os.chown(path, int(os.environ["SUDO_UID"]), int(
                os.environ["SUDO_GID"]))
11
12  def log(msg):
13      print(msg)
14      with open(LOG_FILE, "a") as f: f.write(msg + "\n")
15      fix_perms(LOG_FILE)
16
17  def run_host_cmd(host, command, wait=False):
18      """
19      Esegue un comando nell'host.
20      Se wait=False, prova a lanciarlo in background reale
            usando nohup.
21      """
22      pid = subprocess.getoutput(f"ps -eo pid,cmd | grep '
            mininet:{host}' | grep -v grep | awk '{{print $1
            }}'").strip()
23      if not pid: return "Host non trovato"
24
25      if "iperf -s" in command:
26          # Forza l'esecuzione in background totale per
                evitare blocchi
27          full_cmd = f"sudo nsenter -t {pid} -n nohup {
                command} > /dev/null 2>&1 &"
28          subprocess.Popen(full_cmd, shell=True)
29          return f"Server iPerf avviato su {host}"
30      else:
31          full_cmd = f"sudo nsenter -t {pid} -n {command}"
```

```
32          return subprocess.getoutput(full_cmd)
33
34  def main():
35      # Pulizia iniziale
36      subprocess.run("sudo pkill iperf", shell=True, stderr
            =subprocess.DEVNULL)
37      with open(LOG_FILE, "w") as f: f.write("--- REPORT
            TEST SDN AVANZATO ---\n")
38      fix_perms(LOG_FILE)
39
40      log(f"Inizio sessione: {time.strftime('%H:%M:%S')}")
41
42      # --- TEST 1: PING ---
43      log("\n[1] Verifica Connettivita' (H1 -> H3):")
44      res_ping = run_host_cmd("h1", "ping -c 4 10.0.0.3")
45      log(res_ping)
46
47      # --- TEST 2: VIDEO SLICE (UDP 9999) ---
48      log("\n[2] Test Video Slice (UDP 9999) - Target 10
            Mbps:")
49      run_host_cmd("h3", "iperf -s -u -p 9999") # Avvia
            server
50      time.sleep(2) # Attesa cruciale per attivazione
            server
51
52      log("Generazione traffico in corso...")
53      # Eseguiamo il client (questo deve essere bloccante
            per durare 10s)
```

```
54      res_video = run_host_cmd("h1", "iperf -c 10.0.0.3 -u
            -p 9999 -b 10M -t 10")
55      log(res_video)
56
57      # --- TEST 3: STANDARD SLICE (TCP) ---
58      log("\n[3] Test Standard Slice (TCP) - Target 1Mbps
            :")
59      run_host_cmd("h4", "iperf -s") # Avvia server su H4
60      time.sleep(2)
61
62      log("Generazione traffico in corso...")
63      res_tcp = run_host_cmd("h2", "iperf -c 10.0.0.4 -t
            10")
64      log(res_tcp)
65
66      # Pulizia finale
67      subprocess.run("sudo pkill iperf", shell=True)
68      log(f"\nFine test: {time.strftime('%H:%M:%S')}")
69
70  if __name__ == "__main__":
71      if os.geteuid() != 0:
72          print("Esegui con sudo!"); sys.exit()
73      main()
```

Il modulo `run_tests.py` automatizza l'esecuzione dei principali test di rete utilizzati per la valutazione delle performance delle slice SDN e si occupa della raccolta strutturata dei risultati in un file di log. Il suo obiettivo è garantire test ripetibili e coerenti, riducendo l'intervento

manuale durante le fasi sperimentali.

Lo script esegue i comandi direttamente all'interno dei namespace di rete degli host Mininet, sfruttando `nsenter` per assicurare che le misurazioni riflettano fedelmente il comportamento della topologia emulata. Un aspetto rilevante è la gestione differenziata dei processi `iperf`, che vengono avviati in modalità server o client a seconda del ruolo dell'host e del tipo di test.

I test eseguiti includono:

- una verifica preliminare di connettività tramite `ping`, utile a validare la corretta configurazione della rete;
- un test sullo slice ad alta priorità (video slice), basato su traffico UDP con banda target di 10 Mbps;
- un test sullo slice standard, basato su traffico TCP con capacità nominale di 1 Mbps.

Per evitare blocchi nell'esecuzione dello script, i server `iperf` vengono avviati in background reale tramite `nohup`, mentre i client vengono eseguiti in modalità bloccante per garantire una durata controllata dei test. I risultati di ciascun test vengono registrati in modo sequenziale all'interno di un file di log, che può essere successivamente visualizzato e scaricato dalla dashboard Streamlit.

Infine, il modulo implementa una gestione esplicita dei privilegi e dei permessi sui file di output, assicurando che i log generati durante

l'esecuzione con `sudo` rimangano accessibili anche all'utente non privilegiato. Questo rende il flusso di test facilmente integrabile con gli altri componenti del sistema di monitoraggio.

3.1.2 Capacity_test.py

```
    import subprocess
import time
import os
import sys
import pandas as pd

# File di riferimento
LOG_FILE = "capacity_test_results.log"
DATA_FILE = "traffic_data.csv"

def fix_perms(path):
    """Sblocca i permessi del log per l'utente non-root.
       """
    if os.path.exists(path) and "SUDO_UID" in os.environ:
        os.chown(path, int(os.environ["SUDO_UID"]), int(
           os.environ["SUDO_GID"]))

def log(msg):
    """Stampa a video e scrive nel file di log."""
    print(msg)
    with open(LOG_FILE, "a") as f: f.write(msg + "\n")
    fix_perms(LOG_FILE)
```

```
21
22 def get_loss_from_csv():
23     """Legge il packet loss in tempo reale dal CSV senza
          bloccare il file."""
24     try:
25         if not os.path.exists(DATA_FILE):
26             return 0.0
27
28         # Apriamo in modalita' lettura con 'newline' per
              evitare conflitti
29         with open(DATA_FILE, "r", encoding="utf-8",
              errors="ignore") as f:
30             lines = f.readlines()
31             if len(lines) < 2: # Solo header o vuoto
32                 return 0.0
33
34             # Scorriamo il file al contrario per trovare
                  l'ultima riga della Standard Slice
35             for line in reversed(lines):
36                 parts = line.strip().split(',')
37                 # Assicurati che l'indice coincida con la
                      colonna 'Loss'
38                 # Se il CSV e': Timestamp, Interface,
                      Mbps, Latency, Jitter, Loss -> Loss e'
                      l'indice 5
39                 if len(parts) >= 6 and "Standard" in
                      parts[1]:
40                     try:
```

```
41                          loss_value = float(parts[5])
42                          return loss_value
43                      except ValueError:
44                          continue
45          return 0.0
46      except Exception as e:
47          print(f"Errore accesso CSV: {e}")
48          return 0.0
49
50  def run_host_cmd(host, command):
51      """Esegue un comando nel namespace dell'host Mininet.
            """
52      pid = subprocess.getoutput(f"ps -eo pid,cmd | grep '
            mininet:{host}' | grep -v grep | awk '{{print $1
            }}'").strip()
53      if not pid: return "Host non trovato"
54      return subprocess.getoutput(f"sudo nsenter -t {pid} -
            n {command}")
55
56  def main():
57      # Pulizia iniziale di eventuali processi iperf
            rimasti appesi
58      subprocess.run("sudo pkill iperf", shell=True, stderr
            =subprocess.DEVNULL)
59
60      with open(LOG_FILE, "w") as f:
61          f.write(f"--- ANALISI DI SATURAZIONE: {time.
                strftime('%H:%M:%S')} ---\n")
```

```
62      fix_perms(LOG_FILE)
63
64      log("Avvio Server iPerf su H4 (Destinazione)...")
65      pid_h4 = subprocess.getoutput(f"ps -eo pid,cmd | grep
            'mininet:h4' | grep -v grep | awk '{{print $1}}'")
           .strip()
66      subprocess.Popen(f"sudo nsenter -t {pid_h4} -n iperf
           -s -u > /dev/null 2>&1", shell=True)
67      time.sleep(2)
68
69      # Parametri del test
70      current_bw = 0.2
71      step = 0.1  # Step piu' piccolo (100Kbps) per una
           precisione maggiore
72      threshold = 10.0
73
74      log(f"Target: Rilevare il limite della Standard Slice
            (Soglia Loss > {threshold}%)")
75      log
           ("-------------------------------------------------------------
76
77      while True:
78          log(f">>> Incremento carico: {current_bw:.2f}
               Mbps")
79
80          # Genera traffico per 4 secondi
```

```
81          run_host_cmd("h2", f"iperf -c 10.0.0.4 -u -b {
               current_bw}M -t 4")
82
83          # Breve attesa per permettere al monitor di
               scrivere nel CSV
84          time.sleep(1.2)
85
86          # Controllo della perdita pacchetti
87          current_loss = get_loss_from_csv()
88          log(f"   [Monitor] Perdita rilevata: {
               current_loss}%")
89
90          if current_loss > threshold:
91              log(f"\n[!!!] SOGLIA SUPERATA: {current_loss
                   }%")
92              log(f"Il limite della slice e' stato
                   raggiunto a {current_bw:.2f} Mbps.")
93              break
94
95          # Sicurezza per evitare loop infiniti se qualcosa
               non va nel monitor
96          if current_bw >= 5.0:
97              log("\n[?] Test interrotto: raggiunto limite
                   di sicurezza di 5 Mbps senza rilevare loss
                   .")
98              break
99
100         current_bw += step
```

```
101
102     # Pulizia e chiusura
103     subprocess.run("sudo pkill iperf", shell=True)
104     log("\n--- TEST TERMINATO CON SUCCESSO ---")
105
106 if __name__ == "__main__":
107     if os.geteuid() != 0:
108         print("Errore: lo script deve essere eseguito con
                sudo!")
109     else:
110         main()
```

Il modulo `capacity_test.py` implementa un test di saturazione controllata finalizzato a verificare il corretto funzionamento del meccanismo di network slicing sullo slice a bassa capacità. In particolare, il test è progettato per dimostrare che, superata la banda assegnata (1 Mbps), il sistema manifesti un degrado delle prestazioni misurabile in termini di packet loss.

Il test genera traffico UDP crescente verso lo slice standard, aumentando progressivamente la banda richiesta a piccoli passi. Dopo ogni fase di generazione del traffico, il modulo non misura direttamente la qualità del collegamento, ma interroga i dati prodotti dal sistema di monitoraggio, leggendo le informazioni più recenti dal file CSV condiviso.

Gli aspetti chiave del test includono:

- incremento graduale del carico di traffico per individuare il punto

di collasso dello slice;

- analisi robusta del packet loss basata su più campioni recenti, al fine di evitare falsi positivi dovuti a fluttuazioni temporanee;
- condizione di arresto basata sul superamento di una soglia di perdita pacchetti prefissata (10%);
- meccanismo di timeout di sicurezza per prevenire esecuzioni prolungate o bloccanti.

Il test sfrutta l'esecuzione dei comandi `iperf` all'interno dei namespace degli host Mininet, garantendo coerenza con la topologia emulata e con le politiche di slicing applicate dal controller SDN. I risultati vengono registrati in un file di log dedicato, che può essere successivamente analizzato o visualizzato tramite la dashboard Streamlit.

Nel complesso, questo modulo consente di validare sperimentalmente il rispetto dei vincoli di banda imposti allo slice inferiore e fornisce una dimostrazione concreta dell'efficacia delle politiche di controllo del traffico adottate.

Chapter 4

Conclusioni

Il progetto ha dimostrato come le funzionalità di *Topology Slicing* e *Service Slicing* possano essere implementate in modo efficace all'interno di un'architettura SDN utilizzando Mininet e controller Ryu. Per garantire modularità, chiarezza progettuale e facilità di validazione sperimentale, si è scelto di adottare due controller distinti, ciascuno con una responsabilità ben definita. Il controller dedicato al Topology Slicing realizza l'isolamento dei percorsi di rete attraverso regole statiche basate sugli indirizzi MAC, instradando il traffico tra coppie di host predefinite e consentendo la creazione di slice topologici indipendenti. Il controller per il Service Slicing, invece, opera a un livello più dinamico, classificando il traffico in base al protocollo di trasporto e alla porta di destinazione, e applicando politiche di priorità per garantire un trattamento preferenziale al traffico video.

L'adozione di controller separati ha permesso di semplificare il de-

bug e il testing delle singole funzionalità, oltre a rendere ciascun componente facilmente manutenibile e riutilizzabile in contesti differenti. Questo approccio riflette le architetture SDN reali, nelle quali il controllo della rete è spesso suddiviso in più moduli o controller specializzati, ciascuno responsabile di aspetti specifici come instradamento, qualità del servizio e gestione delle risorse.

Un contributo fondamentale del progetto è rappresentato dal sistema di monitoraggio delle prestazioni, che consente di validare sperimentalmente le politiche di slicing implementate. Attraverso un modulo di monitoraggio in background e una dashboard interattiva basata su Streamlit, è stato possibile osservare in tempo quasi reale metriche quali throughput, latenza, jitter e packet loss, nonché analizzare i risultati dei test eseguiti da terminale. In particolare, il test di capacità ha permesso di evidenziare il comportamento dello slice a bassa priorità in condizioni di congestione, mostrando la comparsa di perdita di pacchetti al superamento della banda assegnata e confermando l'efficacia delle politiche di controllo applicate.

Nel complesso, il progetto fornisce una dimostrazione concreta di come i principi dell'SDN possano essere utilizzati per realizzare slicing di rete flessibile e misurabile, integrando il piano di controllo con strumenti di monitoraggio avanzati che risultano essenziali per l'analisi, la validazione e l'ottimizzazione delle prestazioni di rete.