



**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

Network and Cloud Infrastructures

Project Work Slicing

Esposito Fabio M63001806

Fontanella Gianluca M63001818

Grandioso Nicola M63001814

Iovine Giovanna M63001821

Prof. Giorgio Ventre

Academic Year 2024/25

SDN Project Work: Network Slicing with Ryu and Mininet	3
Chapter 1	3
Topology	3
Chapter 2	6
Topology slicing	6
Objective	6
Solution	6
Testing.....	9
Chapter 3	12
Service slicing	12
Objective	12
Solution	12
Testing.....	15
Chapter 4	19
Dashboard for visualization.....	19
Objective	19
Solution 1 (topology slicing)	19
1. Controller modifies (controller_fin.py)	19
2. Flask Server (server2.py)	22
3. HTML Page (table.html)	23
Testing.....	24
Solution 2 (video slicing)	27
Testing 2	29
Chapter 5	32
Advanced Traffic Generation and Classification.....	32
Objective	32
Solution	32
Testing.....	35

SDN Project Work: Network Slicing with Ryu and Mininet

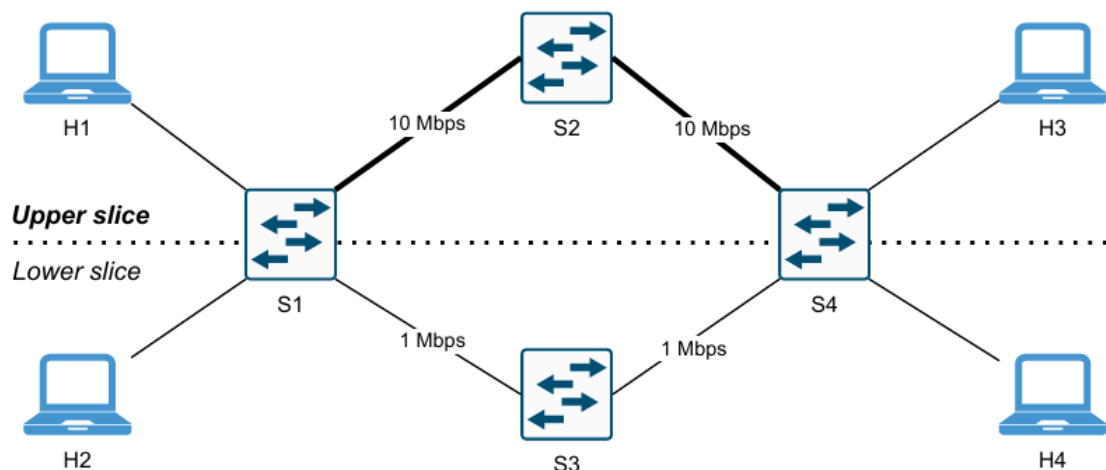
The purpose of this project is to implement network slicing in an SDN environment using Mininet as the network emulator and Ryu as the SDN controller. The implementation will focus on:

1. Topology Slicing – Restricting communication paths between specific hosts.
2. Service Slicing – Prioritizing specific types of traffic (e.g., video) over others.

Chapter 1

Topology

We decided to implement the following topology. It is a very simple design that allow us have a great example of slicing we can often test in different scenarios.



We report the code below.

```
import threading
import random
import time
from mininet.log import setLogLevel, info
from mininet.topo import Topo
from mininet.net import Mininet, CLI
from mininet.node import OVSKernelSwitch, Host
from mininet.link import TCLink, Link
from mininet.node import RemoteController #Controller

class Environment(Topo):
```

```

def __init__(self):
    "Create a network."
    self.net = Mininet(controller=RemoteController, link=TCLink)

    info("*** Starting controller\n")
    c1 = self.net.addController( 'c1', controller=RemoteController)
#Controller

```

We start the controller to manage the traffic of the net.

```

    c1.start()

    info("*** Adding hosts and switches\n")
    self.h1 = self.net.addHost('h1', mac='00:00:00:00:00:01', ip=
'10.0.0.1')
    self.h2 = self.net.addHost('h2', mac='00:00:00:00:00:02', ip=
'10.0.0.2')
    self.h3 = self.net.addHost('h3', mac='00:00:00:00:00:03', ip=
'10.0.0.3')
    self.h4 = self.net.addHost('h4', mac='00:00:00:00:00:04', ip=
'10.0.0.4')
    self.s1 = self.net.addSwitch('s1', cls=OVSKernelSwitch)
    self.s2 = self.net.addSwitch('s2', cls=OVSKernelSwitch)
    self.s3 = self.net.addSwitch('s3', cls=OVSKernelSwitch)
    self.s4 = self.net.addSwitch('s4', cls=OVSKernelSwitch)

    info("*** Adding links\n")
    self.net.addLink(self.h1, self.s1)
    self.net.addLink(self.h2, self.s1)
    self.net.addLink(self.s1, self.s2, bw=10)
    self.net.addLink(self.s1, self.s3, bw=1)
    self.net.addLink(self.s2, self.s4, bw=10)
    self.net.addLink(self.s3, self.s4, bw=1)
    self.net.addLink(self.s4, self.h3)
    self.net.addLink(self.s4, self.h4)

    info("*** Starting network\n")
    self.net.build()
    self.net.start()

...
if __name__ == '__main__':

    setLogLevel('info')
    info('starting the environment\n')
    env = Environment()

```

```
info("*** Running CLI\n")  
CLI(env.net)
```

In this case, we first created the four hosts and the four switches. Then, based on the given topology, we established the links between them.

Chapter 2

Topology slicing

Objective

Implement a sliced network where specific hosts can only communicate through predefined paths. Specifically:

- Host H1 should communicate only with H3 via an upper path (slice).
- Host H2 should communicate only with H4 via a lower path (slice).

Solution

We want to implement a specific logic for routing traffic through the network. This responsibility lies with the controller, which we intend to modify to achieve our goal. In the base solution, packet forwarding is based on a learning mechanism. However, we aim to enforce a predefined route instead. There will no longer be flooding to learn the next hop; instead, the route will be determined in advance based on a **source-destination mapping**.

```
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, ethernet, arp
from ryu.ofproto import ether as ether_types

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # Table-miss flow entry
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)
```

```

def add_flow(self, datapath, priority, match, actions, buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]

    if buffer_id is not None:
        mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                priority=priority, match=match,
                                instructions=inst)
    else:
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)

    datapath.send_msg(mod)

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofp = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']
    dpid = datapath.id

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    if eth.ethertype == ether_types.ETH_TYPE_LLDP:
        return

    src = eth.src
    dst = eth.dst

    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"

    not_allowed = {
        (H1, H2), (H1, H4), (H2, H1), (H2, H3),
        (H3, H2), (H3, H4), (H4, H1), (H4, H3)
    }

    self.mac_to_port.setdefault(dpid, {})
    self.mac_to_port[dpid][src] = in_port

```

```

if (src, dst) in not_allowed:
    match = parser.OFPMatch(eth_src=src, eth_dst=dst)
    self.add_flow(datapath, 1, match, []) # Drop
    return
out_port=0

if dpid == 1:
    if src == H1:

        out_port = 3
    elif src == H2:
        out_port = 4
    elif src == H3:
        out_port = 1
    elif src == H4:
        out_port = 2
elif dpid == 2:
    if src == H1:
        out_port = 2
    else:
        out_port = 1

elif dpid == 3:
    if src == H2:
        out_port = 2
    else:
        out_port = 1

elif dpid == 4:
    if src == H1:
        out_port = 3
    elif src == H2:
        out_port = 4
    elif src == H3:
        out_port = 1
    elif src == H4:
        out_port = 2

# Aggiorna mac_to_port per la destinazione solo se non è flood

self.mac_to_port[dpid][dst] = out_port

actions = [parser.OFPActionOutput(out_port)]

# Crea un match più specifico per evitare conflitti
match = parser.OFPMatch(in_port=in_port, eth_src=src, eth_dst=dst)

```



```

# Installa la regola nel flow table
if msg.buffer_id != ofp.OFP_NO_BUFFER:
    self.add_flow(datapath, 10, match, actions, msg.buffer_id)
    return
else:
    self.add_flow(datapath, 10, match, actions)

# Invia il pacchetto
data = None
if msg.buffer_id == ofp.OFP_NO_BUFFER:
    data = msg.data

out = parser.OFPPacketOut(datapath=datapath,
                           buffer_id=msg.buffer_id,
                           in_port=in_port,
                           actions=actions,
                           data=data)
datapath.send_msg(out)

```

This code implements a simple Ryu OpenFlow 1.3 controller application that acts as a learning switch with added **custom traffic filtering and static routing logic**.

At startup, the controller installs a **table-miss flow entry** to direct unmatched packets to the controller.

The application maintains a **set of disallowed source-destination MAC address pairs**, representing blocked flows. When such a pair is detected, the controller installs a **drop rule in the switch to block future traffic** between those addresses.

For **allowed traffic**, rather than relying on dynamic learning or flooding, the controller uses **predefined static rules based on the switch ID (dpid) and the source MAC address** to determine the **correct output port**. It then **installs a flow entry to handle subsequent packets directly at the switch level**, reducing controller load and improving performance.

Finally, the **current packet is sent out through the chosen port**.

Overall, this design enforces static path routing and selective traffic blocking, giving the controller **fine-grained control over how packets move through the network**.

Testing

How to start the network?

1. On terminal 1, execute the command `“ryu-manager controller7.py”`

2. On terminal 2, execute the command “`sudo python3 topology.py`”

We will execute our test commands in the section where the network topology is created.

To verify the correct behavior of our network, we use the **pingall** command. This command tests the reachability between all hosts by sending packets from each host to every other host. It helps confirm that the static routing and traffic control mechanisms are functioning as expected.

```
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>
```

This test immediately shows that host h1 can communicate only with host h3, and host h2 only with host h4, as required by our specifications.

To gain a more detailed understanding of the network behavior, we also used the **ping** command between each pair of hosts individually. The output confirmed the unreachability of the denied routes, verifying that our filtering and static routing logic is working correctly.

```
mininet> h1 ping 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=0.245 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.082 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.107 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.065 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=0.057 ms
64 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=0.067 ms
64 bytes from 10.0.0.3: icmp_seq=7 ttl=64 time=0.070 ms
64 bytes from 10.0.0.3: icmp_seq=8 ttl=64 time=0.064 ms
64 bytes from 10.0.0.3: icmp_seq=9 ttl=64 time=0.066 ms
64 bytes from 10.0.0.3: icmp_seq=10 ttl=64 time=0.051 ms
64 bytes from 10.0.0.3: icmp_seq=11 ttl=64 time=0.162 ms
64 bytes from 10.0.0.3: icmp_seq=12 ttl=64 time=0.072 ms
64 bytes from 10.0.0.3: icmp_seq=13 ttl=64 time=0.062 ms
64 bytes from 10.0.0.3: icmp_seq=14 ttl=64 time=0.053 ms
^C
```

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 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
From 10.0.0.1 icmp_seq=5 Destination Host Unreachable
From 10.0.0.1 icmp_seq=6 Destination Host Unreachable
From 10.0.0.1 icmp_seq=7 Destination Host Unreachable
From 10.0.0.1 icmp_seq=8 Destination Host Unreachable
From 10.0.0.1 icmp_seq=9 Destination Host Unreachable
```

By running a simple ping from h1 to both a reachable and an unreachable host, we observe two distinct outputs: one indicating **successful** communication and the other indicating **failure**. This second test further confirms that the network is functioning correctly according to the traffic slicing rules we implemented.

To verify that packets follow the correct path, we performed a ping from h1 to h3 and monitored the traffic on the relevant links. Although we observed activity on both the link entering switch 3 (lower path) and switch 2 (upper path), only the ICMP packets traversed the upper path through switch 2. This confirms that the forwarding logic enforces the intended routing behavior.

Left Screenshot (s2-eth1): Captured 40 packets, all ICMP Echo (ping) from 10.0.0.3 to 10.0.0.1.

No.	Time	Source	Destination	Protocol	Length	Info
28	9.215473612	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping)
29	10.245421107	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping)
30	10.245562212	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping)
31	11.263306599	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping)
32	11.263357508	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping)
33	12.287250650	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping)
34	12.287296195	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping)
35	13.311393070	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping)
36	13.311433165	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping)
37	14.336341367	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping)
38	14.336386762	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping)
39	15.359390383	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping)
40	15.359436299	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping)

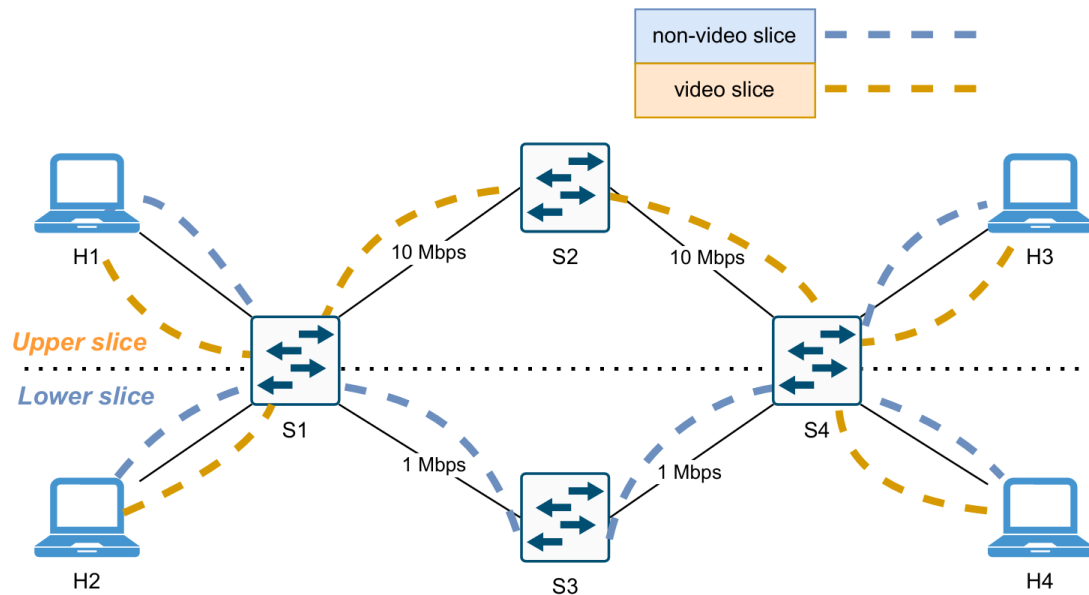
Right Screenshot (s3-eth1): No packets captured.

Chapter 3

Service slicing

Objective

Implement traffic-based slicing where video traffic (identified as UDP traffic on port 9999) is prioritized over other types of traffic.



Solution

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofp = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']
    dpid = datapath.id

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
    eth_type = eth.ethertype
    if eth.ethertype == ether_types.ETH_TYPE_LLDP:
        return

    src = eth.src
    dst = eth.dst

    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.mac_to_port.setdefault(dpid, {})
    self.mac_to_port[dpid][src] = in_port
    out_port = []
    is_video_traffic = False

    ip_pkt = pkt.get_protocol(ipv4.ipv4)
    udp_pkt = pkt.get_protocol(udp.udp)

    # verifico che siano effettivamente pacchetti ip-udp e controllo porta per discriminare traffico video
    # in base a se traffico e video o broadcast o altro andiamo ad assegnare porte di uscita.
    if ip_pkt and udp_pkt:
        if udp_pkt.dst_port == 9999:
            is_video_traffic = True
```

```

if dpid == 1:
    if src == H1:
        if is_video_traffic:
            if dst == H3 or dst == H4:
                out_port.append(3)
            elif dst == H2:
                out_port.append(2)
            else:
                #traffico broadcast
                out_port.append(2)
                out_port.append(3)
        else:
            if dst == H3 or dst == H4:
                out_port.append(4)
            elif dst == H2:
                out_port.append(2)
            else:
                #traffico broadcast
                out_port.append(2)
                out_port.append(4)
    elif src == H2:
        if is_video_traffic:
            if dst == H3 or dst == H4:
                out_port.append(3)
            elif dst == H1:
                out_port.append(1)
            else:
                #traffico broadcast
                out_port.append(1)
                out_port.append(3)
        else:
            if dst == H3 or dst == H4:
                out_port.append(4)
            elif dst == H1:
                out_port.append(1)
            else:
                #traffico broadcast
                out_port.append(1)
                out_port.append(4)

```

```

elif src == H3 or src == H4:
    if dst == H1:
        out_port.append(1)
    elif dst == H2:
        out_port.append(2)
    else:
        #traffico broadcast
        out_port.append(1)
        out_port.append(2)

```

```

elif dpid == 2:
    if in_port == 1:
        out_port.append(2)
    else:
        out_port.append(1)

```

```

elif dpid == 3:
    if in_port == 1:
        out_port.append(2)
    else:
        out_port.append(1)

```

```

elif dpid == 4:
    if src == H3:
        if is_video_traffic:
            if dst == H1 or dst == H2:
                out_port.append(1)
            elif dst == H4:
                out_port.append(4)
            else:
                #traffico broadcast
                out_port.append(1)
                out_port.append(4)
        else:
            if dst == H1 or dst == H2:
                out_port.append(2)
            elif dst == H4:
                out_port.append(4)

```

```

        else:
            #traffico broadcast
            out_port.append(2)
            out_port.append(4)
    elif src == H4:
        if is_video_traffic:
            if dst == H1 or dst == H2:
                out_port.append(1)
            elif dst == H3:
                out_port.append(3)
            else:
                #traffico broadcast
                out_port.append(1)
                out_port.append(3)
        else:
            if dst == H1 or dst == H2:
                out_port.append(2)
            elif dst == H3:
                out_port.append(3)
            else:
                #traffico broadcast
                out_port.append(2)
                out_port.append(3)
    elif src == H1 or src == H2:
        if dst == H3:
            out_port.append(3)
        elif dst == H4:
            out_port.append(4)
        else:
            #traffico broadcast
            out_port.append(3)
            out_port.append(4)

```

```

# Aggiorna mac_to_port per la destinazione solo se non è flood
actions = []

```

Since the goal is to **prioritize video traffic**, explicitly identified as traffic with destination **port 9999** and using the **UDP transport protocol**, the objective is to ensure that this traffic utilizes the path with the highest bandwidth. This implies that within the routers, there is a need to differentiate incoming packets by leveraging the information found in the packet headers.

The proposed solution involves modifying the `packet_in_handler` function, which manages the arrival of packets at the switch, to distinguish video traffic from non-video traffic. In the case of video traffic, **only the output ports that allow routing the traffic along the priority path (and, of course, those leading to hosts that can all receive video traffic) are considered as possible exits**. To develop this first point, we have created a list called `out_port` to store the switch's output ports where the packet should be directed based on the type of traffic associated with that packet. Within nested `else-if` statements, for each switch, we populate this list based on the destination host, the source host, and the type of packet. Additionally, we have handled the case where the destination host is `FF:FF:FF:FF`, i.e., broadcast traffic. In this scenario, we do not discriminate based on the destination host but still adhere to the priority constraints, having first checked whether the packet is video traffic or not.

```
## ora devo creare i match, ossia le corrispondenze da dare allo switch per fargli capire quando arriva il pacchetto di quel tipo
## ossia che fa "match" cosa deve fare. Il cosa deve fare sono le actions, ossia semplicemente inviare pkt sulle porte nell'array out_port

if is_video_traffic:
    # se il traffico è classificato come video, vuol dire che il pacchetto arrivato è ip - udp e ha porto dst 9999 e quindi creo match
    match= parser.OFPMatch(in_port=in_port, eth_src=src, eth_dst=dst, eth_type=ether_types.ETH_TYPE_IP, ip_proto=17, udp_dst=9999)
else:
    # può capitare che il pacchetto arrivato sia ip (esempio icmp)...
    if eth_type==ether_types.ETH_TYPE_IP:
        ip_pkt = pkt.get_protocol(ipv4.ipv4)
        # ...ma può non essere udp, devo verificarlo
        if ip_pkt.proto == 17:
            udp_pkt = pkt.get_protocol(udp.udp)
            # se ip - udp non traffico video devo dare alla dest port semplicemente quella che arriva
            match= parser.OFPMatch(in_port=in_port, eth_src=src, eth_dst=dst, eth_type=ether_types.ETH_TYPE_IP, ip_proto=17, udp_dst=udp_pkt.dst_port)
        else:
            # se ip ma non udp lascio il particolare protocollo senza assegnare udp alla regola
            match=parser.OFPMatch(in_port=in_port, eth_src=src, eth_dst=dst, eth_type=ether_types.ETH_TYPE_IP, ip_proto=ip_pkt.proto)
    else:
        # pacchetti di altro tipo (es. ARP)
        match = parser.OFPMatch(in_port=in_port, eth_src=src, eth_dst=dst, eth_type= ether_type)

# traffico unicast
if len(out_port) == 1:
    self.mac_to_port[dpid][dst] = out_port[0]

for porta in out_port:
    actions.append(parser.OFPACTIONOutput(porta))
```

```
# Installa la regola nel flow table
if msg.buffer_id != ofp.OFP_NO_BUFFER:
    self.add_flow(datapath, 10, match, actions, msg.buffer_id)
    return
else:
    self.add_flow(datapath, 10, match, actions)

# Invia il pacchetto
data = None
if msg.buffer_id == ofp.OFP_NO_BUFFER:
    data = msg.data

out = parser.OFPPacketOut(datapath=datapath,
                           buffer_id=msg.buffer_id,
                           in_port=in_port,
                           actions=actions,
                           data=data)
datapath.send_msg(out)
```

In the second step of the solution, with the output ports available, we proceed to insert the rules into the router (we create the matches). We note that for video traffic, we insert a rule that specifies how to **route packets that use the UDP protocol (17), port 9999, and are IP packets**. In the case of non-video traffic, further distinctions are necessary regarding which rules to insert into the switch. If the incoming packet is IP, uses the UDP transport protocol, but the destination port is different from 9999, it means it is not video traffic, and I must route it correctly using the information previously retrieved from the switch. If it is only IP, it is possible that the transport protocol is different from UDP, so I need a rule to differentiate this type of traffic. Finally, we insert a separate rule for traffic that is not IP.

CAUTION: Why is such a detailed distinction of traffic necessary, and why is it not enough to simply separate video traffic from non-video traffic when creating the rules? **In writing the following rules, a granular approach was adopted. Packet matching is based on identifying video traffic as UDP on port 9999.**

If an incoming packet does not belong to video traffic, creating overly general rules would lead to problems. For example, defining a generic rule for all IP packets would also include UDP video traffic, resulting in incorrect matching.

We followed this approach:

- If the packet is IP, we check whether it uses the UDP protocol.
 - If it is **not UDP**, we match it based solely on the transport protocol.
 - If it **is UDP**, we need to be more specific and inspect the port number. Since this is **non-video traffic**, the port will necessarily be different from 9999. Therefore, in this case, we perform matching based on both the transport protocol and the port.

Thanks to this selective and hierarchical approach, we are able to correctly route the traffic.

Lastly, the precaution of using a `for` loop over `out_port` is necessary in the case of broadcast traffic.

Note that the responses to video packets are only reply packets and not actual video traffic, and therefore they are routed through the lower-priority link.

Testing

- Ensure that all hosts can ping each other (basic connectivity).


```
mininet> 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)
```

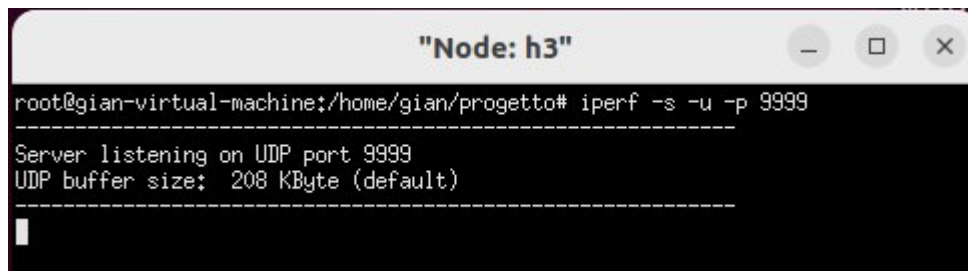
- Use iPerf to generate UDP traffic on port 9999 and verify that it is routed through the dedicated slice.

To simulate video traffic, we use the `iperf` command.

Assume that **h1** acts as the client and **h3** as the server.

For convenience, we open a terminal on each host using: `xterm h1 h3`

To start the server on **h3**, we run:



```
root@gian-virtual-machine:/home/gian/progetto# iperf -s -u -p 9999
-----
Server listening on UDP port 9999
UDP buffer size: 208 KByte (default)
-----
█
```

This sets up a UDP server listening on port 9999.

On the client side (**h1**), we use:

```
iperf -c 10.0.0.3 -u -p 9999 -b 15M
```

This sends UDP packets to **h3** (with IP address `10.0.0.3` as defined in the topology) on port 9999.

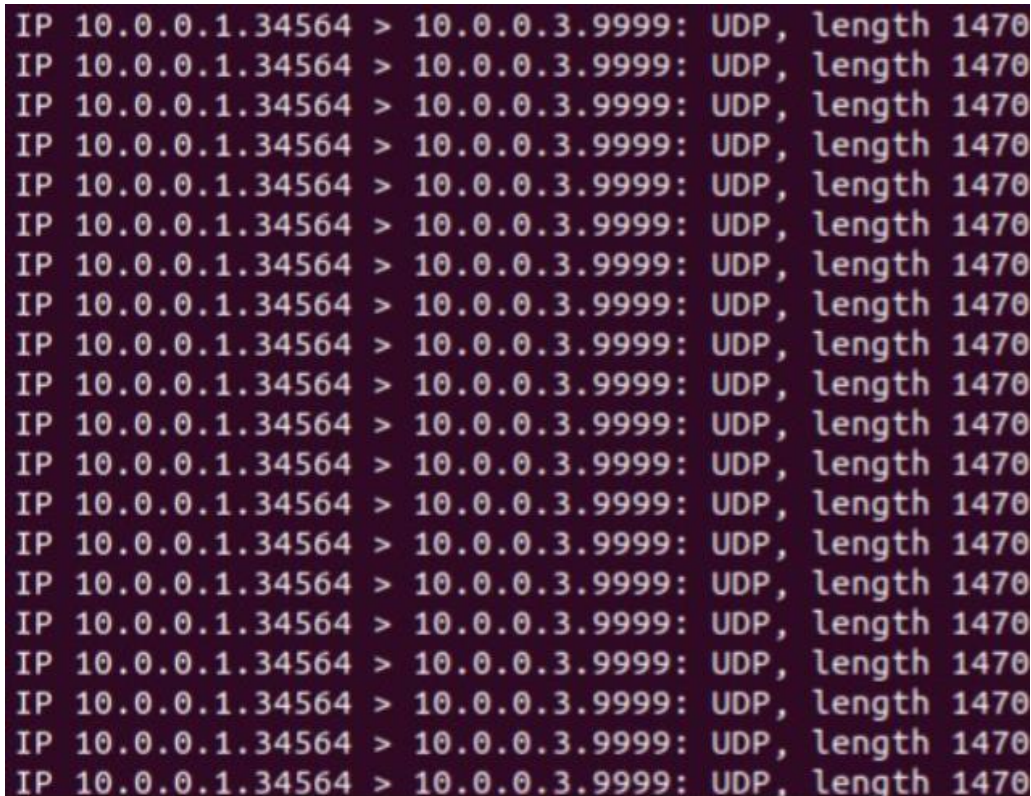
The `-b` option specifies the bandwidth. Note that we intentionally set a value **higher than 10 Mbit/s** to demonstrate that the link is **limited to 10 Mbit/s**.

To verify that the traffic actually follows the intended path—passing through switch **S1**, then **S2**, and finally **S4**—we use the `tcpdump` command to observe traffic on specific switch interfaces.

For example, on **S1**:


```
s1 tcpdump -i s1-eth3
```

This command monitors interface eth3 on switch S1.



```
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
IP 10.0.0.1.34564 > 10.0.0.3.9999: UDP, length 1470
```

We must verify that traffic is being forwarded through the correct switch port (and not through the others).

The same procedure is applied to the other switches (**S2**, **S4**) to confirm that the traffic is routed as expected.

- Compare throughput for video traffic vs. other traffic, confirming that video traffic has priority.

To compare video traffic with other types of traffic—and to demonstrate that video traffic is indeed prioritized—we measure and display the bandwidth usage for both video and non-video traffic:

- For **video traffic**, we repeat the previously described procedure by sending UDP packets to a server listening on port 9999.

```

root@gian-virtual-machine:/home/gian/progetto# iperf -c 10.0.0.3 -u -p 9999 -b 15M
-----
Client connecting to 10.0.0.3, UDP port 9999
Sending 1470 byte datagrams, IPG target: 747.68 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.1 port 45418 connected with 10.0.0.3 port 9999
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-10.0040 sec 11.4 MBytes 9.55 Mbits/sec
[ 1] Sent 8128 datagrams
[ 1] Server Report:
[ ID] Interval      Transfer    Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-10.0604 sec 11.4 MBytes 9.50 Mbits/sec  4.477 ms 0/8127 (0%)
[ 1] 0.0000-10.0604 sec 6 datagrams received out-of-order
root@gian-virtual-machine:/home/gian/progetto# █

```

- For **non-video traffic**, we send packets to a different server (not UDP 9999) on **h3**, and the iPerf report will show the resulting bandwidth.

```

root@gian-virtual-machine:/home/gian/progetto# iperf -c 10.0.0.3 -u -p 9998 -b 15M
-----
Client connecting to 10.0.0.3, UDP port 9998
Sending 1470 byte datagrams, IPG target: 747.68 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.1 port 45836 connected with 10.0.0.3 port 9998
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-10.0954 sec 1.16 MBytes 966 Kbits/sec
[ 1] Sent 830 datagrams
[ 1] Server Report:
[ ID] Interval      Transfer    Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-9.7106 sec 1.16 MBytes 1.00 Mbits/sec 45.447 ms 0/829 (0%)
[ 1] 0.0000-9.7106 sec 333 datagrams received out-of-order
root@gian-virtual-machine:/home/gian/progetto# █

```

Chapter 4

Dashboard for visualization

Objective

Develop a real-time monitoring dashboard to visualize network statistics (e.g., traffic flow bandwidth, delay.. for the different flows).

Solution 1 (topology slicing)

Our idea is to create two separate dashboards: the first for topology slicing and the second for displaying service slicing statistics. However, the implementation is largely the same, so we will focus on discussing the first one and simply report the second for completeness.

We can divide the solution into three parts: modifications to the controller, the Flask server, and the display on the HTML page.

1. **Controller modifies** (controller_fin.py)

The statistics are obtained from the flow entries in the switches' flow tables. This can only be done by the controller, so the first step was to modify it.

To do this, we added three new attributes to the `__init__(self, *args, **kwargs)` method.

```
self.datapaths = {}
self.flow_stats = {} # Per memorizzare flussi e byte precedenti
self.FLASK_SERVER_URL = 'http://127.0.0.1:5000/update_stats'
self.monitor_thread = hub.spawn(self._monitor)
```

The first one is used to store the IDs of the network switches. The second one stores the byte and flow counts, which are needed to calculate *bandwidth*. Finally, we added the URL for connecting to the Flask server and the route we defined to send the statistics (which we'll discuss later).

We also started a **greenthread** using the routine provided by Ryu. This allows us to run a separate thread alongside the main process, so we can collect statistics without interrupting the controller's execution.

Then, we modified the **switch_features_handler(self, ev)** method so that each time a new switch connects to the controller, its ID is stored in the datapath list.

```

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # Memorizza datapath per monitoraggio stats
    self.datapaths[datapath.id] = datapath

    # Table-miss flow entry
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                      ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)

```

Finally, we added some methods that allow us to start the thread, periodically request statistics (once every second—making it essentially “real-time”), and wait for the responses.

```

def _monitor(self):
    """Richiede periodicamente le statistiche di flusso dagli switch."""
    while True:
        for dp in self.datapaths.values():
            self._request_flow_stats(dp)
        hub.sleep(1)

def _request_flow_stats(self, datapath):
    parser = datapath.ofproto_parser
    req = parser.OFPFlowStatsRequest(datapath)
    datapath.send_msg(req)

```

As we can see, every second we send a flow statistics request to each switch. To do this, we used the **OFPSFlowStatsRequest** message, then we send it to the switch to obtain the statistics for each flow.

Then we use the decorator **@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)** on the method **_flow_stats_reply_handler(self, ev)** so that each time we have the response from a switch we can calculate bandwidth and send data to the Flask Server.

```

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev):
    dpid = ev.msg.datapath.id
    current_time = time.time()
    body = ev.msg.body

    stats_to_send = []

```

First, the switch is identified by retrieving its ID. Then, we take a **timestamp** to know exactly when the statistics were collected. After that, we extract the body of the

message, which contains the flow information. At this point, the controller prepares a list of statistics to send.

```
for stat in body:
    if stat.priority != 10:
        continue # Considera solo i flussi con priorità 10

    match = stat.match
    src = match.get('eth_src')
    dst = match.get('eth_dst')
    byte_count = stat.byte_count
```

For each flow received, we focus only on the ones that were installed by us. From each entry, we extract the **source** and **destination** from the match field. It's important to note that we want to display, for each switch, the bandwidth used by flows going from a specific source to a specific destination.

Finally, we retrieve the **number of bytes** transmitted by each flow from the entry's statistics.

We ignore flows which haven't got a source or a destination.

```
if not src or not dst:
    continue # Ignora flussi senza src/dst

key = (dpid, src, dst)
last = self.flow_stats.get(key, None)

bandwidth_kbps = 0
if last:
    time_diff = current_time - last['time']
    if time_diff > 0:
        delta_bytes = byte_count - last['bytes']
        bandwidth_kbps = (delta_bytes * 8) / (time_diff * 1000)
```

Then, we create an entry in `flow_stats` corresponding to a key defined by the datapath ID, the source, and the destination.

If we have already seen that flow before, we calculate how much time has passed since the last measurement. If a non-zero amount of time has passed, we compute the number of bytes transmitted during that interval. Then, to calculate the bandwidth, we convert the bytes to bits (by multiplying by 8) and divide by the elapsed time in seconds. This gives us the bandwidth in bits per second (bps).

Notice that we divide by 1000 so that we can express the bandwidth in kilo bits per second (**kbps**).

```

# Salva l'ultima statistica
self.flow_stats[key] = {'bytes': byte_count, 'time': current_time}

stats_to_send.append({
    'dpid': dpid,
    'src': src,
    'dst': dst,
    'bandwidth_kbps': round(max(0, bandwidth_kbps), 2)
})

if stats_to_send:
    self.logger.info("Invio statistiche dei flussi al server Flask")
    hub.spawn(self._send_stats_to_flask, stats_to_send)

```

Then, we store the statistics for the next computation, prepare the dictionary to send to the server, and—if there is data to send, after that we parsed every flow entry that the switch sent us—we start a green thread to send it to the server using the `_send_stats_to_flask` method.

```

def _send_stats_to_flask(self, stats):
    """Invia le statistiche a Flask in un greenthread separato."""
    try:
        requests.post(self.FLASK_SERVER_URL, data=json.dumps(stats),
                      headers={'Content-Type': 'application/json'})
    except requests.exceptions.RequestException as e:
        self.logger.error("Errore nell'invio delle statistiche al server: %s", e)

```

This method performs an **HTTP POST request** to send data to a Flask server. Before sending, it converts the dictionary of statistics into a **JSON-formatted string**. It also includes exception handling to catch and log any errors that occur during the data transmission.

2. Flask Server (server2.py)

The server is very **lightweight**; in fact, it has only three routes.

```

1 from flask import Flask, request, jsonify, render_template
2 import time
3
4 app = Flask(__name__)
5
6 # Struttura: {(dpid, src, dst): {'bandwidth_kbps': ..., 'last_updated': ...}}
7 flow_stats = {}
8
9 @app.route('/update_stats', methods=['POST'])
10 def update_stats():
11     stats_list = request.get_json()
12
13     for stat in stats_list:
14         key = (stat['dpid'], stat['src'], stat['dst'])
15
16         flow_stats[key] = {
17             'bandwidth_kbps': stat['bandwidth_kbps'],
18             'last_updated': time.strftime('%H:%M:%S')
19         }
20
21     return 'Dati aggiornati', 204

```

The first route handles **POST** requests from the controller, allowing us to start saving the flow statistics in the `flow_stats` dictionary.

The second route, which is called by JavaScript on the HTML page via a **GET** request, provides the data received from the controller so it can be displayed on the page.

```
@app.route('/flow_data')
def flow_data():
    # Converte i dati in formato JSON leggibile per JS
    response = []
    for (dpid, src, dst), info in flow_stats.items():
        response.append({
            'switch': f'Switch {dpid}',
            'src': src,
            'dst': dst,
            'bandwidth_kbps': info['bandwidth_kbps'],
            'last_updated': info['last_updated']
        })
    return jsonify(response)
```

By default, the server runs on the loopback interface at IP address `127.0.0.1` on port `5000`. Through the virtual machine, we can access this interface and call the third route, which we added to render the HTML page.

```
@app.route('/')
def index():
    return render_template('table.html')
```

3. HTML Page (table.html)

In this case, we created a user-friendly interface where the user can view the flows—those that are allowed—from each source to each destination as they pass through the switches. In particular, the user can also filter the entries to display only those related to a specific switch.

Additionally, there is a chart that shows the **trend of the bandwidth** of the flows over time.

The HTML file actually contains the CSS, HTML, and JavaScript code all in one. The most important part is the section that makes the request to the Flask server using the route we mentioned earlier.

```
function updateTable() {
    fetch('/flow_data')
        .then(response => response.json())
```

Thanks to this we can update in real time the content of tables and charts.

Testing

To test our solution, we start the Ubuntu 22.04 virtual machine. Then, we launch the Ryu controller using the following command:

```
ryu-manager controller_fin.py
```

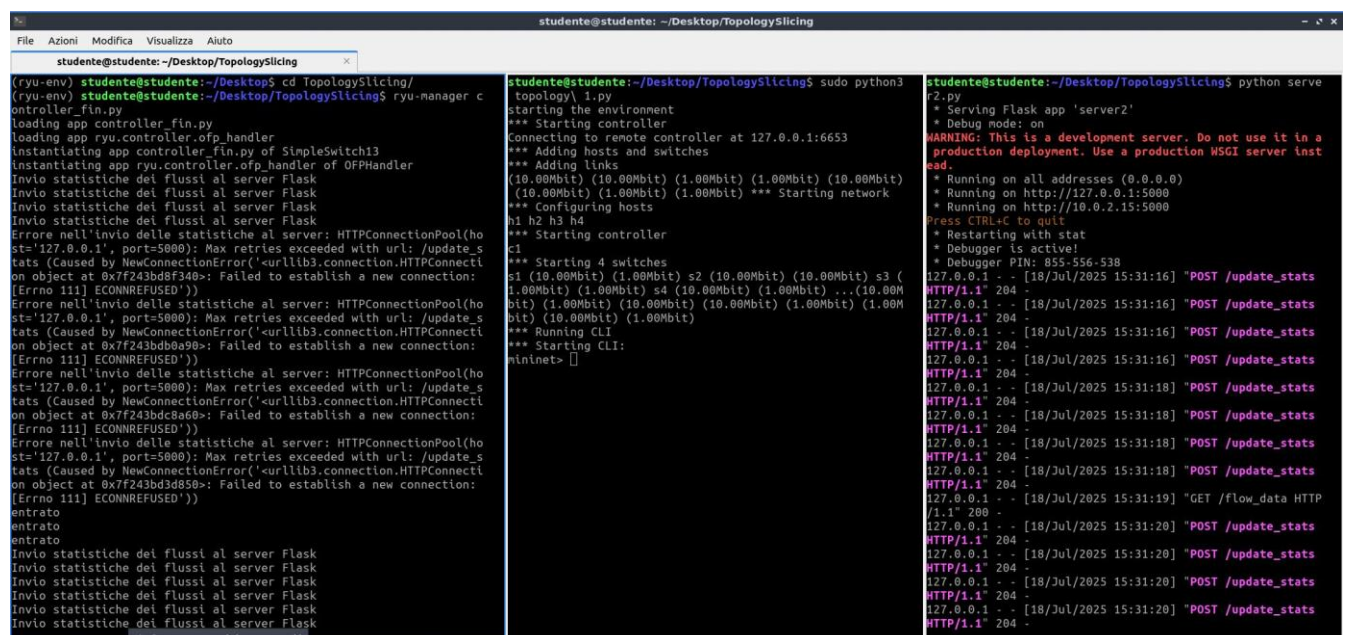
Then, we create the network topology using Mininet with the following command:

```
sudo python3 topology\ 1.py
```

And finally, we start the Flask server simply by running the following command:

```
python server2.py
```

These are the preliminary steps:

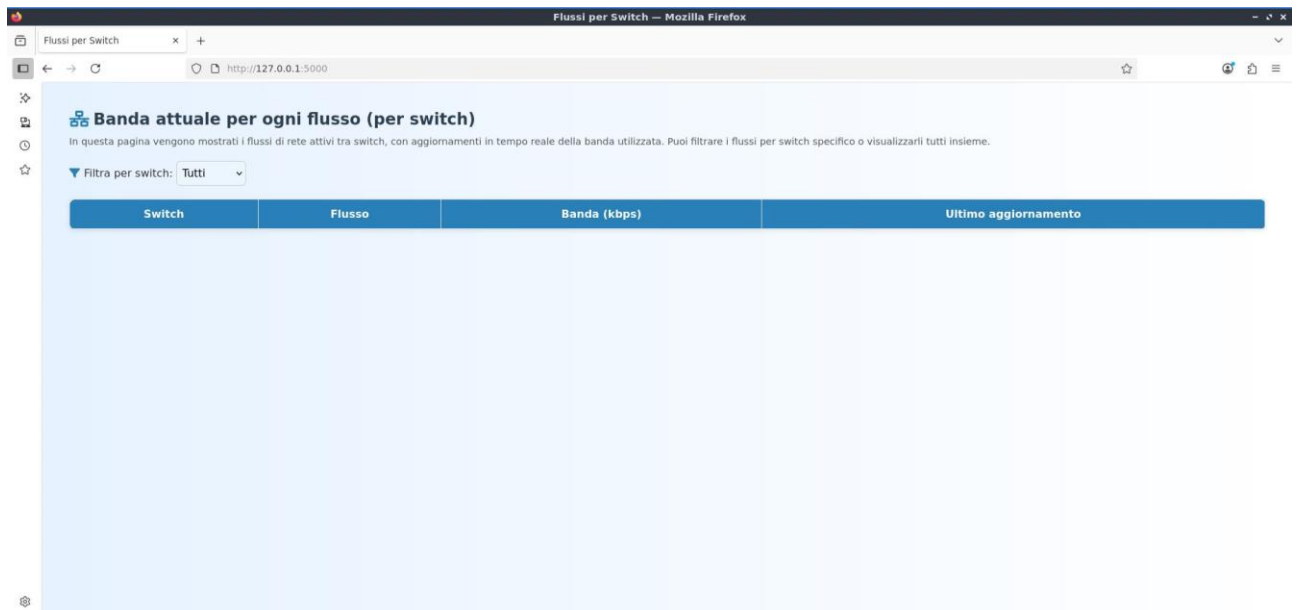


```
studente@studente: ~/Desktop/TopologySlicing
(ryu-env) studente@studente:~/Desktop$ cd TopologySlicing/
(ryu-env) studente@studente:~/Desktop/TopologySlicing$ ryu-manager c
ontroller_fin.py
loading app controller_fin.py
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
Invio statistiche dei flussi al server Flask
Invio statistiche dei flussi al server Flask
Invio statistiche dei flussi al server Flask
Errore nell'invio delle statistiche al server: HTTPConnectionPool(ho
sts='127.0.0.1', port=5000): Max retries exceeded with url: /update_s
tats (Caused by NewConnectionError('<urllib3.connection.HTTPConnecti
on object at 0x7f243bd8f340>: Failed to establish a new connection:
[Errno 111] ECONNREFUSED'))
Errore nell'invio delle statistiche al server: HTTPConnectionPool(ho
sts='127.0.0.1', port=5000): Max retries exceeded with url: /update_s
tats (Caused by NewConnectionError('<urllib3.connection.HTTPConnecti
on object at 0x7f243bd8a90>: Failed to establish a new connection:
[Errno 111] ECONNREFUSED'))
Errore nell'invio delle statistiche al server: HTTPConnectionPool(ho
sts='127.0.0.1', port=5000): Max retries exceeded with url: /update_s
tats (Caused by NewConnectionError('<urllib3.connection.HTTPConnecti
on object at 0x7f243bd8c8a0>: Failed to establish a new connection:
[Errno 111] ECONNREFUSED'))
Errore nell'invio delle statistiche al server: HTTPConnectionPool(ho
sts='127.0.0.1', port=5000): Max retries exceeded with url: /update_s
tats (Caused by NewConnectionError('<urllib3.connection.HTTPConnecti
on object at 0x7f243bd3d850>: Failed to establish a new connection:
[Errno 111] ECONNREFUSED'))
entrato
entrato
Invio statistiche dei flussi al server Flask
Invio statistiche dei flussi al server Flask
Invio statistiche dei flussi al server Flask
Invio statistiche dei flussi al server Flask
Invio statistiche dei flussi al server Flask
Invio statistiche dei flussi al server Flask

studente@studente:~/Desktop/TopologySlicing$ sudo python3
topology\ 1.py
starting the environment
*** Starting controller
connecting to remote controller at 127.0.0.1:6653
*** Adding hosts and switches
*** Adding links
(10.00Mbit) (10.00Mbit) (1.00Mbit) (1.00Mbit) (10.00Mbit)
(10.00Mbit) (1.00Mbit) (1.00Mbit) (1.00Mbit) *** Starting network
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c1
*** Starting 4 switches
s1 (10.00Mbit) (1.00Mbit) s2 (10.00Mbit) (10.00Mbit) s3 (
1.00Mbit) (1.00Mbit) s4 (10.00Mbit) (1.00Mbit) ... (10.00M
bit) (1.00Mbit) (10.00Mbit) (10.00Mbit) (1.00Mbit) (1.00M
bit) (1.00Mbit) (1.00Mbit)
*** Running CLI
*** Starting CLI:
mininet> []

studente@studente:~/Desktop/TopologySlicing$ python serve
r2.py
* Serving Flask app 'server2'
* Debug mode: on
WARNING: This is a development server. Do not use it in a
production deployment. Use a production WSGI server inst
ead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://10.0.2.15:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 855-556-538
127.0.0.1 - - [18/Jul/2025 15:31:16] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:16] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:16] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:16] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:18] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:18] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:18] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:18] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:19] "GET /flow_data HTTP
/1.1" 200 -
127.0.0.1 - - [18/Jul/2025 15:31:20] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:20] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:20] "POST /update_stats
HTTP/1.1" 204 -
127.0.0.1 - - [18/Jul/2025 15:31:20] "POST /update_stats
HTTP/1.1" 204 -
```

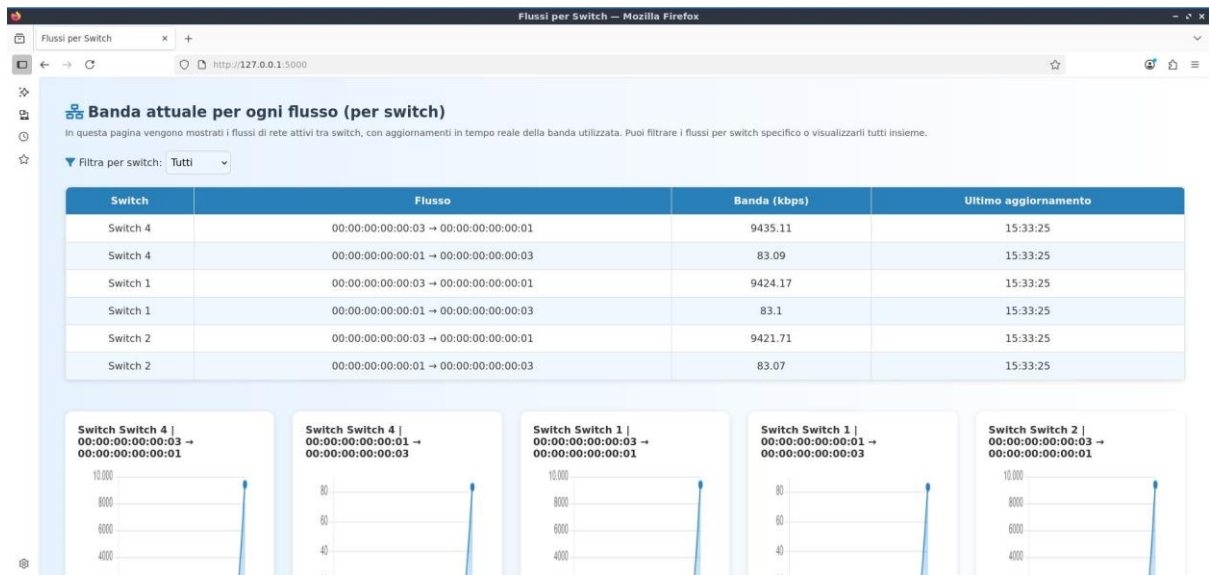
The main interface of our dashboard displays a table with the four fields shown in the image, along with a dropdown filter to choose the switch you want to analyze.



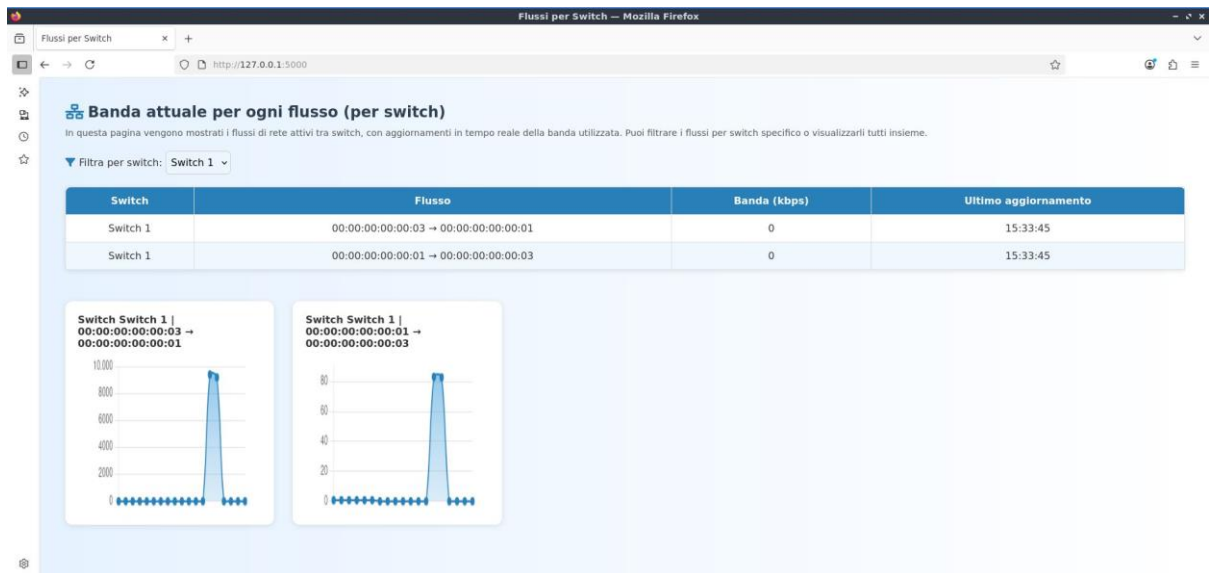
We can see some examples of use, for example we can start just doing a ping from h1 to h3:



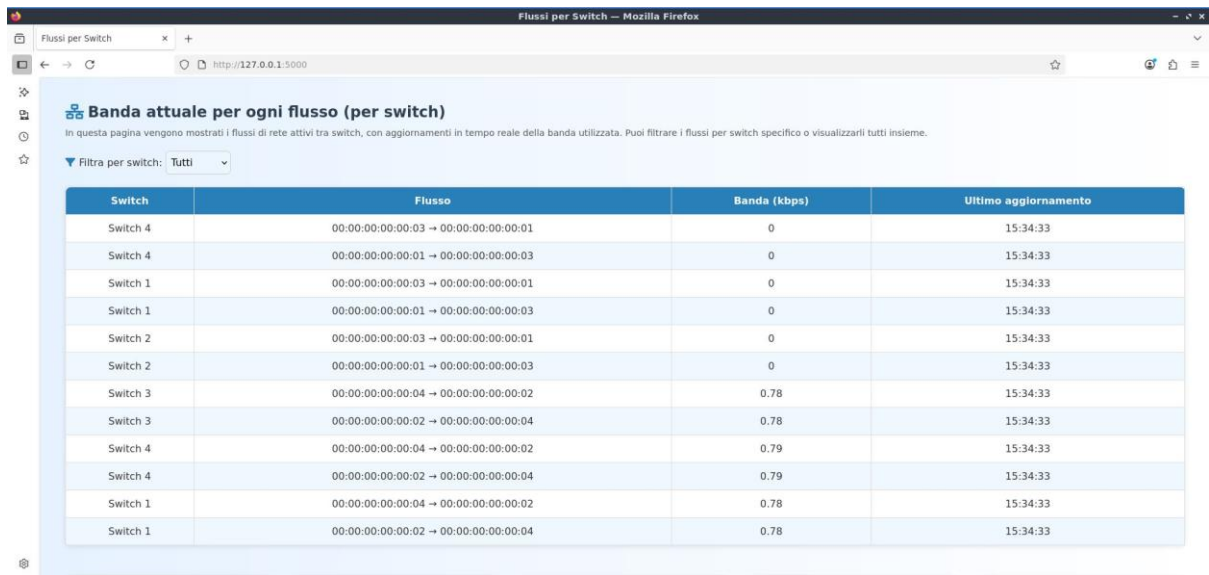
Thanks to our dashboard, you can also verify that only the correct switches are traversed by the different flows.



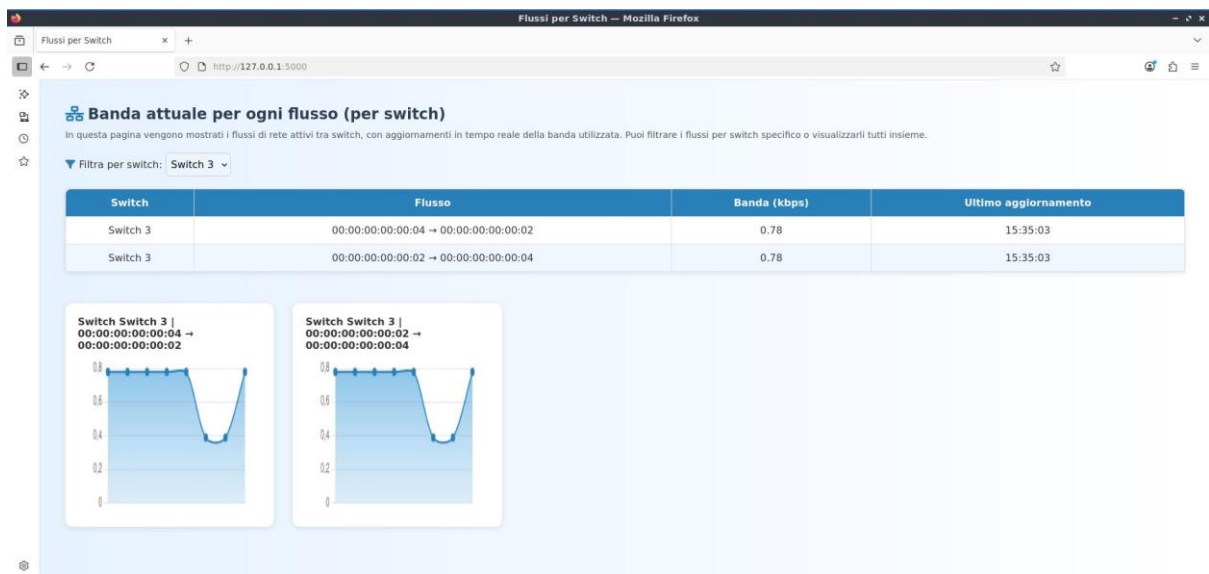
We can also see an example where we generate traffic using D-ITG.



Then this is an example of the use of switch filter.



Then, we can observe that when we ping from h2 to h4, the switches traversed by the flows change. In particular, the traffic now follows the path that goes through the third switch.



Solution 2 (video slicing)

As for the dashboard in this case, we actually did the same things—the only difference is the type of JSON object we send to the server.

```

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev):
    dpid = ev.msg.datapath.id
    current_time = time.time()
    body = ev.msg.body

    stats_to_send = []

    for stat in body:
        if stat.priority != 10:
            continue

        match = stat.match
        src = match.get('eth_src')
        dst = match.get('eth_dst')
        eth_type = match.get('eth_type')
        ip_proto = match.get('ip_proto')
        udp_dst = match.get('udp_dst')
        byte_count = stat.byte_count

        if not src or not dst:
            continue

```

We need more detailed information about each flow because we distinguish between different flows based on the transport protocol and port numbers.

So, the rest of the code is identical to the case we saw earlier. As for the Flask server, it remains the same, but in this case, we need to parse a different JSON object.

```

@app.route('/update_stats', methods=['POST'])
def update_stats():
    stats_list = request.get_json()

    for stat in stats_list:
        key = (stat['dpid'], stat['src'], stat['dst'])

        flow_stats[key] = {
            'bandwidth_kbps': stat['bandwidth_kbps'],
            'eth_type': stat.get('eth_type'),
            'ip_proto': stat.get('ip_proto'),
            'udp_dst': stat.get('udp_dst'),
            'last_updated': time.strftime('%H:%M:%S')
        }

    return 'Dati aggiornati', 204

```

```
@app.route('/flow_data')
def flow_data():
    # Converte i dati in formato JSON leggibile per JS
    response = []
    for (dpid, src, dst), info in flow_stats.items():
        response.append({
            'switch': f'Switch {dpid}',
            'src': src,
            'dst': dst,
            'bandwidth_kbps': info['bandwidth_kbps'],
            'eth_type': info.get('eth_type'),
            'ip_proto': info.get('ip_proto'),
            'udp_dst': info.get('udp_dst'),
            'last_updated': info['last_updated']
        })
    return jsonify(response)
```

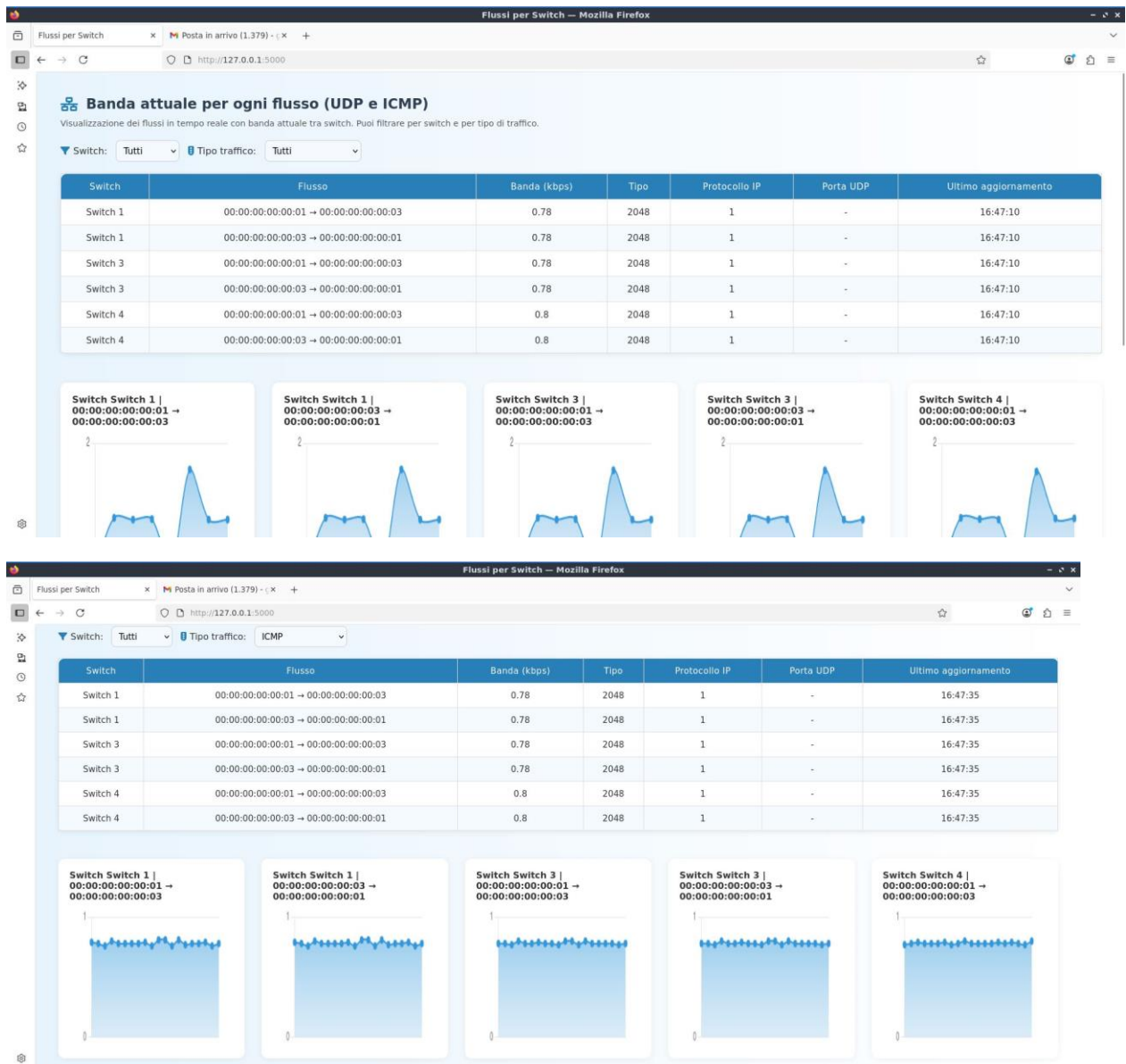
As for the HTML page, the only difference is that in this case we distinguish flows based on the protocol (UDP or ICMP) and the port (9999 or not). We also added a dropdown filter to choose which flows to view.

Testing 2

So we start the controller, the server and the topology.

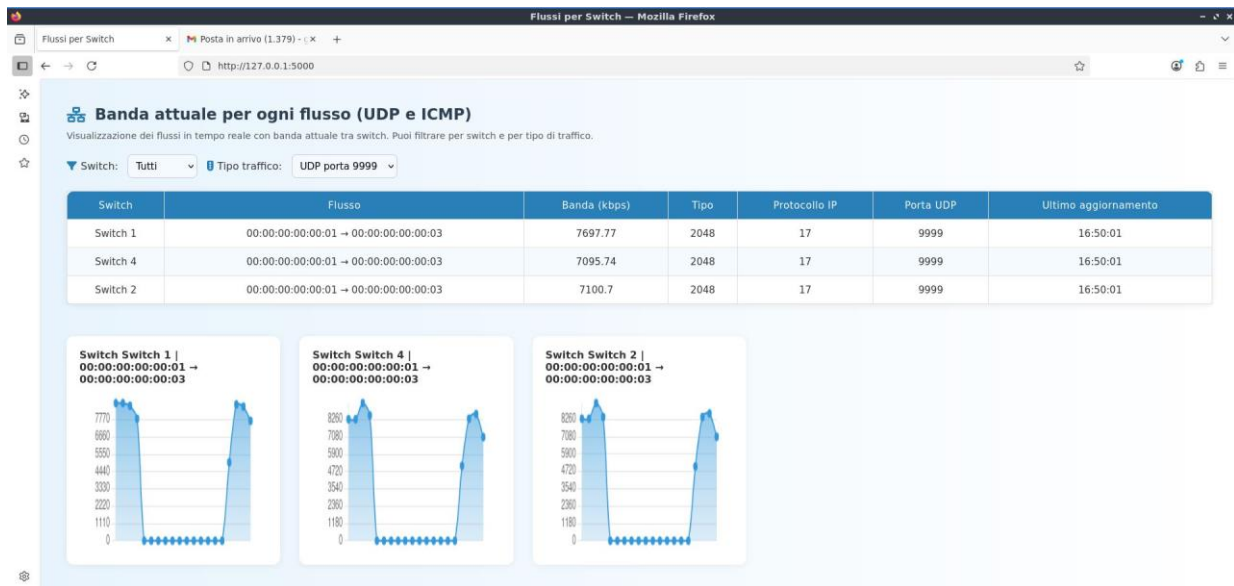
[illegible]

We began testing by viewing all switches while filtering only ICMP flows. To test this, we performed a ping from h1 to h3.



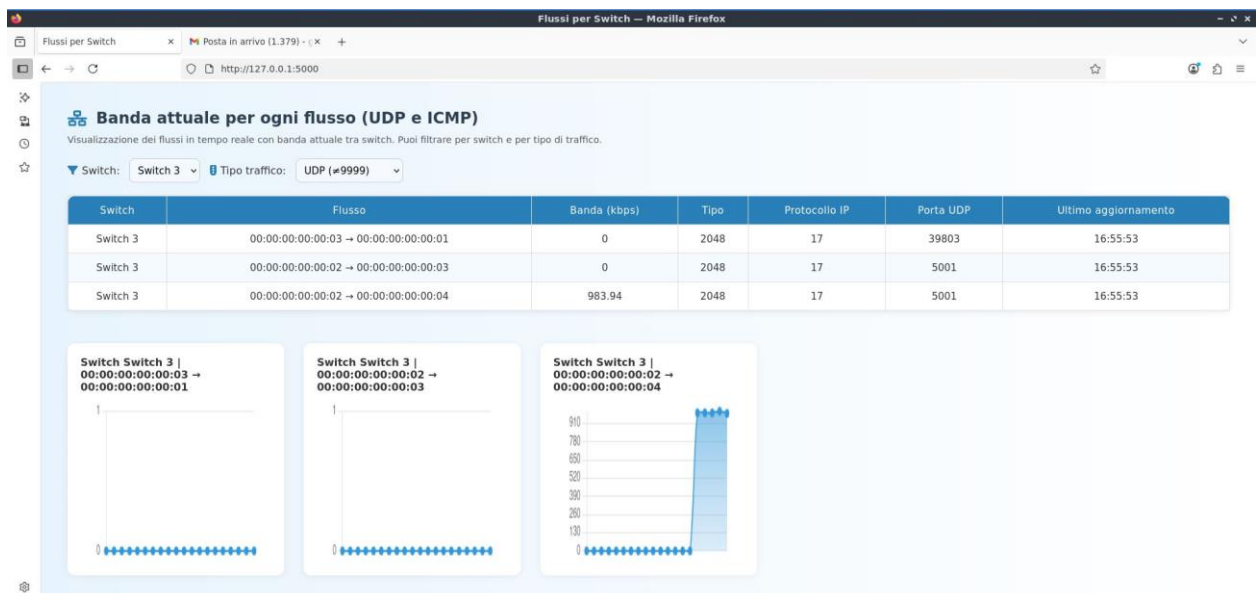
So, we can see that since the traffic uses the ICMP protocol, it is routed through switch 3 (the lower path).

Then, we generated video traffic from H3 to H1 using iperf. We ran `iperf -s -u -p 9999` in the H3 shell and `iperf -c 10.0.0.3 -u -p 9999 -b 8M` in the H1 shell. On the dashboard, we filtered the traffic to show only flows using the UDP protocol and port 9999.



Notice that traffic is routed through switch 3 (the upper path).

While if we try to use a different port (for example 5001), we can see:



Chapter 5

Advanced Traffic Generation and Classification

Objective

- Use D-ITG (Distributed Internet Traffic Generator) to generate diverse traffic profiles (e.g., video and normal traffic).
- Classify traffic based on packet statistics rather than just port numbers.

Solution

Instead of using UDP/TCP port numbers (e.g., port 80 for HTTP, 9999 for video), the controller uses measured behavior to infer the class of traffic dynamically. This is more flexible and realistic, especially in environments where ports might be shared or obfuscated.

Looking at our topology, what we want to modify in order to reach this goal, is the behavior of the controller.

In the first solution presented, the routing rules were decided looking at the couple source address-destination address. Where we implemented a service slicing instead, it was possible to identify the video traffic looking at the destination port number. This time, we want to identify traffic types looking at another information: statistics.

We are using Ryu, which is an SDN controller that uses the OpenFlow protocol to manage and monitor network traffic. To get traffic statistics, Ryu sends

OFPFlowStatsRequest messages to switches, which reply with **OFPFlowStatsReply** containing metrics like byte count, packet count, and flow duration. These statistics allow the controller to compute throughput and monitor traffic load. This enables traffic classification and routing decisions based on actual flow behavior, rather than fixed parameters like port numbers.

In the previous chapter, we collected these information, to send it to the remote server flask.

We want to start from it, to **modify the routing rules**. As we did for the dashboard, the piece of information we focus on about statistics is the bandwidth.

We set a **threshold**, which allow us divide the traffic flow in two categories, we route on the two slices.

```
### MODIFICA: Aggiunti per la classificazione dinamica
# 1. Traccia lo stato attuale di ogni flusso ('normal' o 'video')
self.flow_path_state = {}
# 2. Soglia in Kbps per considerare un flusso come "video" (es. 5 Mbps)
self.VIDEO_BANDWIDTH_THRESHOLD_KBPS = 500
```


We want our controller to understand what's happening in the network, and dynamically redirect the flow. Statistics, is indeed, something we now a posteriori, not something we can predict and now statically since the beginning. It is the reason why, the following modification to the previous code was necessary.

```
### MODIFICA: Aggiunto il parametro 'command' per poter MODIFICARE i flussi
def add_flow(self, datapath, priority, match, actions, buffer_id=None, command=ofproto_v1_3.OFPFC_ADD):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                        actions)]
    if buffer_id is not None:
        mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                priority=priority, match=match,
                                instructions=inst, command=command) # command aggiunto
    else:
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst, command=command) # command aggiunto
    datapath.send_msg(mod)
```

In `_flow_stats_reply_handler`, we have the new business logic to **classify different types of traffic** looking at the bandwidth dynamically.

```
@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev):
    dpid = ev.msg.datapath.id
    datapath = ev.msg.datapath
    ### MODIFICA: Aggiunti parser e ofproto per poter modificare i flussi da qui
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    current_time = time.time()
    body = ev.msg.body
```

If the current bandwidth exceeds a defined threshold and the flow is currently marked as normal, it is **promoted to video**, triggering a flow modification (`OFPFC_MODIFY_STRICT`) that reroutes it through the **optimized path for video traffic**. Conversely, if the bandwidth drops below the threshold and the flow was previously marked as video, it is downgraded to normal and rerouted accordingly. This adaptive logic enables **real-time traffic management** based on actual usage rather than static rules.

```

class SimpleSwitch13(app_manager.RyuApp):
    def _flow_stats_reply_handler(self, ev):
        for stat in body:

            ### MODIFICA: Aggiunta la logica di classificazione e modifica dei flussi
            current_state = self.flow_path_state.get(key, 'normal')

            # Se la banda supera la soglia e il flusso è 'normal', lo promuoviamo a 'video'
            if bandwidth_kbps > self.VIDEO_BANDWIDTH_THRESHOLD_KBPS and current_state == 'normal':
                self.logger.info(f"[DPID {dpid}] PROMUOVO flusso {src}->{dst} a 'video' (Banda: {bandwidth_kbps:.2f} Kbps)")
                # La logica di routing è richiamata qui, specificando che è traffico video
                out_port = self._get_forwarding_path(dpид, src, dst, is_video_traffic=True)
                if out_port:
                    actions = [parser.OFPACTIONOutput(p) for p in out_port]
                    self.add_flow(datapath, 10, match, actions, command=ofproto.OFPFC_MODIFY_STRICT)
                    self.flow_path_state[key] = 'video'

            # Se la banda scende sotto la soglia e il flusso era 'video', lo retrocediamo a 'normal'
            elif bandwidth_kbps < self.VIDEO_BANDWIDTH_THRESHOLD_KBPS and current_state == 'video':
                self.logger.info(f"[DPID {dpid}] RETROCEDO flusso {src}->{dst} a 'normal' (Banda: {bandwidth_kbps:.2f} Kbps)")
                out_port = self._get_forwarding_path(dpид, src, dst, is_video_traffic=False)
                if out_port:
                    actions = [parser.OFPACTIONOutput(p) for p in out_port]
                    self.add_flow(datapath, 10, match, actions, command=ofproto.OFPFC_MODIFY_STRICT)
                    self.flow_path_state[key] = 'normal'

```

The forwarding rules are not in the packet_in_handler function anymore, but in a separated method, where we do not change anything from the previous solutions. Note that all the forwarding table is reported in following snippet of code, to avoid redundancy.

```

### MODIFICA: La logica di routing è stata estratta in una funzione separata
# per evitare duplicazione di codice e renderla riutilizzabile.
def _get_forwarding_path(self, dpид, src, dst, in_port=None, is_video_traffic=False):
    """Questa funzione contiene la TUA logica di routing originale."""
    out_port = []
    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"

    # Tutta la tua logica if/elif/else qui, intatta.
    if dpид == 1:
        if src == H1:
            if is_video_traffic:
                if dst == H3 or dst == H4: out_port.append(3)
                elif dst == H2: out_port.append(2)
                else: out_port.extend([2, 3])
            else:
                if dst == H3 or dst == H4: out_port.append(4)
                elif dst == H2: out_port.append(2)
                else: out_port.extend([2, 4])

```

We handle incoming packets (PacketIn events) and set up initial forwarding rules for new flows. When a packet arrives, it extracts source and destination MAC addresses and determines the output port using a default assumption that the flow is normal traffic (no video classification is done here). A match is built based on the packet type (e.g., IP/UDP), and if the flow is new, it is registered in the controller's flow_path_state as normal. Finally, the controller installs a **flow rule with the appropriate actions so that similar future packets are handled directly by the switch**, improving performance.

Last version instead hardcodes routing logic per switch using MAC addresses and assigns output ports manually. It classifies traffic based on switch-level throughput rather than individual flow stats.

```
class SimpleSwitch13(app_manager.RyuApp):
    def _packet_in_handler(self, ev):

        ### MODIFICA: Semplificazione radicale di questa sezione.
        # 1. Non proviamo più a indovinare se il traffico è video qui.
        # 2. Tutti i nuovi flussi sono considerati 'normal' di default.
        out_port = self._get_forwarding_path(dpid, src, dst, in_port, is_video_traffic=False)
        if not out_port:
            self.logger.info(f"Nessun percorso trovato per {src}->{dst} in DPID {dpid}")
            return

        actions = [parser.OFPACTIONOutput(p) for p in out_port]

        # La tua logica di creazione del match è corretta e la manteniamo
        match = None
        if eth_type == ether_types.ETH_TYPE_IP:
            ip_pkt = pkt.get_protocol(ipv4.ipv4)
            if ip_pkt.proto == 17:
                udp_pkt = pkt.get_protocol(udp.udp)
                match = parser.OFPMATCH(in_port=in_port, eth_src=src, eth_dst=dst,
                                       eth_type=ether_types.ETH_TYPE_IP, ip_proto=17,
                                       udp_dst=udp_pkt.dst_port)
            else:
                match = parser.OFPMATCH(in_port=in_port, eth_src=src, eth_dst=dst,
                                       eth_type=ether_types.ETH_TYPE_IP, ip_proto=ip_pkt.proto)
        else:
            match = parser.OFPMATCH(in_port=in_port, eth_src=src, eth_dst=dst, eth_type=eth_type)

        ### MODIFICA: Registriamo il nuovo flusso come 'normal'.
        if match:
            key = (dpid, src, dst, match.get('eth_type'), match.get('ip_proto'), match.get('udp_dst'))
            if key not in self.flow_path_state:
                self.logger.info(f"[DPID {dpid}] Nuovo flusso {src}->{dst}. Intradato su percorso 'normal'.")
                self.flow_path_state[key] = 'normal'
```

Testing

To test the controller's traffic classification and routing logic, **D-ITG** (Distributed Internet Traffic Generator) is used to simulate realistic network traffic. It allows the generation of multiple traffic profiles with customizable parameters such as bandwidth, packet size, protocol type, and inter-arrival time. By creating distinct flows—e.g., a high-bandwidth video stream and a low-bandwidth normal flow—D-ITG helps validate whether the controller correctly identifies and routes traffic based on runtime statistics (e.g., bandwidth usage) rather than static attributes like port numbers. This provides a controlled environment to evaluate dynamic flow management.

After having created the topology and started the controller, as for the previous tests, we followed the steps below

1. Open a terminal for h1: **xterm h1**
2. Open a terminal for h3: **xterm h3**
3. Open a terminal for s3: **xterm s3**

We want h1 to be a receiver, so we execute from its terminal the command **ITGRecv**.

H3 instead is a sender, so from its terminal we execute the command

ITGSend -a 10.0.0.1 -T UDP -C 63 -c 500 -t 10000

Which instructs D-ITG to generate a UDP traffic flow with the following parameters:

- -a <IP>: Destination IP address — the traffic will be sent to the host with this IP
- -T UDP: Transport protocol — sets the transport layer to UDP.
- -C 63: Packet rate — sends 1000 packets per second.
- -c 500: Packet size — each packet will be 1000 bytes.
- -t 10000: Duration — the traffic will be generated for 10,000 milliseconds (i.e., 10 seconds).

Bandwidth (bps)=Packets per second*Packet size (bytes)*8

We used the terminal where the topology was created to observe traffic on s2, and the terminal of s3, to observe traffic on it. We monitored the network interfaces eth1 on switches S2 and S3 using **tcpdump**, a tool that captures and analyzes network packets. By watching the traffic flow, we noticed that when the traffic coming from S3 exceeds 500 kbps, it gets redirected to switch S2.

```
giovanna@giovanna-VirtualBox: ~/Scari...
12:11:38.658123 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.664882 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.671640 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.678324 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.685735 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.692486 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.699246 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.706772 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.713501 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.720284 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.726993 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.733730 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.740492 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.747252 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.754000 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.761218 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:38.767962 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500

giovanna@giovanna-VirtualBox: ~/Scari...
"Node: h1"
ot@giovanna-VirtualBox:/home/giovanna/Scaricati# ITGRecv
ITGRecv version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
Press Ctrl-C to terminate
Listening on UDP port : 8999
Listening on UDP port : 8999
Listening on UDP port : 8999

"Node: s3" (root)
12:11:33.709791 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.720541 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.729611 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.738328 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.745091 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.750032 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.762238 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.768948 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.775717 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.782442 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.789150 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.795866 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.802615 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.810398 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.822296 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.829143 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.836639 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.845426 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.852174 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.858890 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.865687 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500
12:11:33.872432 IP 10.0.0.3.33866 > 10.0.0.1.8999: UDP, length 500

root@giovanna-VirtualBox:/home/giovanna/Scaricati# ITGSend -a 10.0.0.1 -T UDP -C 150 -c 500 -t 10000
ITGSend version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
Started sending packets of flow ID: 1
Finished sending packets of flow ID: 1

root@giovanna-VirtualBox:/home/giovanna/Scaricati# ITGSend -a 10.0.0.1 -T UDP -C 150 -c 500 -t 10000
ITGSend version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
Started sending packets of flow ID: 1
```

To observe how the network behaviour changes, when the traffic does, we also tested it changing the parameters in the sending command, as below:

ITGSSend -a 10.0.0.1 -T UDP -C 1000 -c 1000 -t 20000

The screenshot shows a terminal window with three panes. The top-left pane displays a list of network traffic entries, each starting with a timestamp, IP addresses, and protocol details. The top-right pane shows the output of the command `ITGRecv` on `Node: h1`, indicating it is listening on UDP port 8999. The bottom pane shows the output of the command `ITGSSend -a 10.0.0.1 -T UDP -C 1000 -c 1000 -t 20000` on `Node: h3`, showing the command's execution and the version of the tool (2.8.1).

```
12:14:52.803464 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.804576 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.805686 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.806856 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.807903 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.809011 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.810119 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.811308 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.812410 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.813520 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.814965 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.816116 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.817187 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.818259 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.819372 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.820478 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000
12:14:52.821598 IP 10.0.0.3.58665 > 10.0.0.1.8999: UDP, length 1000

root@giovanna-VirtualBox:~/home/giovanna/Scaricati# ITGRecv
ITGRecv version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
Press Ctrl-C to terminate
Listening on UDP port : 8999
nish on UDP port : 8999
stening on UDP port : 8999
nish on UDP port : 8999

root@giovanna-VirtualBox:~/home/giovanna/Scaricati# ITGSSend -a 10.0.0.1 -T UDP -C 1000 -c 1000 -t 20000
ITGSSend version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
Started sending packets of flow ID: 1
Finished sending packets of flow ID: 1
root@giovanna-VirtualBox:~/home/giovanna/Scaricati#
```

The first test produces a low-bandwidth, short-duration load on the network. The second command, by contrast, sends a much heavier flow: 1,000 packets per second of 1,000 bytes each for 20 seconds, creating a higher bandwidth and longer-lasting traffic burst. Essentially, the first test simulates normal or low traffic conditions, while the second represents a more intense load, useful for testing how the controller reacts to traffic that might exceed certain thresholds.