



UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Computer Engineering

SDN Network for replicated service

**Daniel Deiana
Giovanni Barbieri
Alessio di Ricco**

ANNO ACCADEMICO 2023/2024

Indice

1	Report	5
1.1	Project requirements	5
1.2	Implementation	5
1.3	Example Network	6
1.4	Main design choices	6
1.4.1	Out of Band Controller	6
1.4.2	Proactive vs Reactive approach	6
1.4.3	Dynamic establishing of paths	6
1.4.4	Load Balancing Policy	7
1.5	Testing	8
1.5.1	Sample Application used for testing	8
1.5.2	Transport protocol used in the sample application	9
1.5.3	Correct functioning test	9
1.5.4	Wrong functioning test	9

Capitolo 1

Report

1.1 Project requirements

It was requested to implement an SDN network that permits to a client to subscribe to a certain service and to implement those requirements:

- exposes a RESTful interface that allows a user to subscribe. Within the subscription, the user must specify its IP address and the number K of replica servers that must process its requests
- Upon a subscription, assigns K servers to the client. When selecting the K server, the load - i.e., the number of clients to which that server is assigned - has to be balanced among servers.
- Implements the mapping between `VIRTUAL_IP` and the K servers, and merges the responses of the servers into a single packet.

1.2 Implementation

To achieve the functional requirements we implemented a floodlight module called `ReverseProxy`. We implemented the REST API for the subscription of clients (unsubscription was not implemented although it is required in a real implementation). The logic implemented in this module is used to create the `DIRECT PATH` to route the client messages to the assigned physical servers (we assign K physical servers as said in specifications). The logic also implements the methods that instantiate `REVERSE` paths from the servers to the controller. As said in specs, we need to replicate the request of the client and send it to the servers, in this phase a packet in is generated because we do not install flow rules that match the `VIRTUAL_IP` as destination, the controller after receiving this packet in creates K replicas of the request and in each replica we set the destination address of every assigned physical server. After this the replicas follow the routes instantiated before and reach the servers. The second phase is the one where we need to

merge the response packets, the merging needs to be done by the controller, the PACKET_IN are generated by the responses because in the REVERSE PATH the source address that matched is the one of VIRTUAL IP, this was done because we will use those reverse path to send the merged response that has the VIRTUAL IP as src address, So when the controller (that maintains a state for pending request of a client) sees that has received all responses, generates and sends only 1 packet specifying as MAC and IP source addresses the one of the VIRTUAL server, the packet follows the reverse route and finally reaches the client.

1.3 Example Network

The network that we used was composed by 2 Hosts and 3 Servers, we have 4 switches that are instantiating a remote TCP connection with the SDN Controller, the following topology is implemented in mininet and it is generated using the script `topology.py`

1.4 Main design choices

1.4.1 Out of Band Controller

The controller is not in the same network as the switches, servers and hosts. This was chosen because there are no particular reasons to choose for the in-band option and with the out of band approach we can have better management and achieve separation of control flow from the data flow.

1.4.2 Proactive vs Reactive approach

We have chosen a proactive approach for the installing of rules, 'so the rules for both REVERSE and DIRECT path so the rules are installed as soon as the client makes the subscription, the rules have infinite timeout so the client that makes a request will not have to wait for the creation of both the K's direct and reverse path (creating those rules periodically has a considerable overhead if we consider a complicated network and if the requested servers are many), the rules can be removed in an implementation of the unsubscribe operation.

1.4.3 Dynamic establishing of paths

The paths that we install are dinamically set based on the current topology that is used, that was considered because it is common that in a network that implements those kind of service changes are made frequently. This was achieved using the `IRoutingService` provided by floodlight.

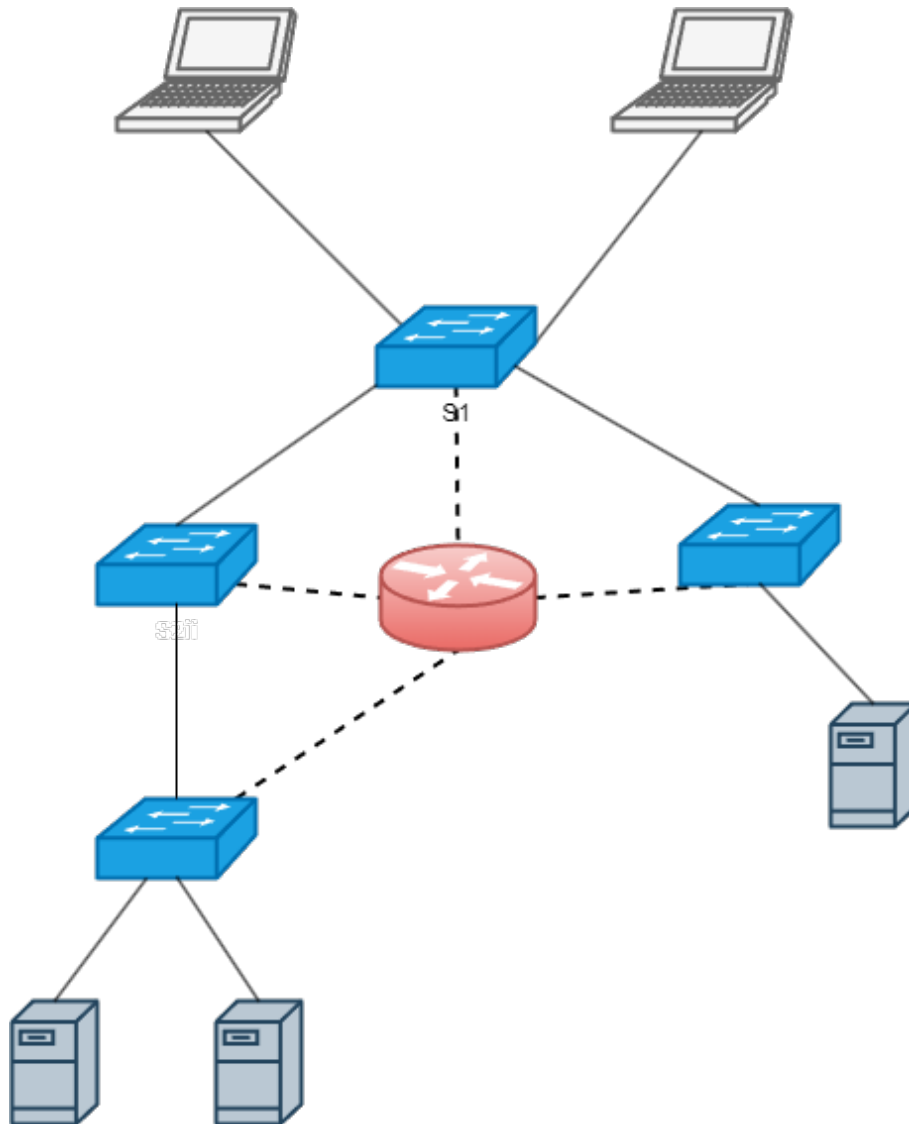


Figura 1.1: Sample network used for testing the module.

1.4.4 Load Balancing Policy

The load balancing requested for the servers was implemented by using a data structure that for each server saves the List of clients that are served. This was done to implement a queue size policy, so when a subscribe is issued the K servers are chosen among the first K servers ordered incrementally by queue size. That is preferable wrt. to a round robin approach because we account for unsubscriptions (a round robin approach has no idea of the queue size).

1.5 Testing

First off, we tested the subscription. Inside the client h1 we can simply use the following command `curl -X POST -H "Content-Type: application/json" -d '{"ClientIP":"10.0.0.2", "K":"2"}' http://127.0.0.1:8080/api/subscribe/json`

```
ip subscribe:10.0.0.2
Inserted new clients [10.0.0.71=[10.0.0.2], 10.0.0.70=[10.0.0.2], 10.0.0.69=[]]
Assigned servers: [10.0.0.71, 10.0.0.70]
```

Figure 1.2: Subscribe output showed in the controller stdout.

After the subscribe we can check if the rules are correctly instantiated in the OpenFlow switches by using the `dpctl dump-flows` command in Mininet console.

```
mininet> dpctl dump-flows
*** s1 ***
cookie=0x0, duration=328.668s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=ee:4c:d7:39:16:f0,d_l_dst=92:95:5f:a3:a0:4a,nw_src=10.0.0.2,nw_dst=10.0.0.71 actions=output:"s1-eth4"
cookie=0x0, duration=328.659s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=ee:4c:d7:39:16:f0,d_l_dst=e6:12:0a:44:65:dc,nw_src=10.0.0.2,nw_dst=10.0.0.70 actions=output:"s1-eth3"
cookie=0x0, duration=328.658s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=00:00:00:00:00:fe,d_l_dst=ee:4c:d7:39:16:f0,nw_src=10.0.0.254,nw_dst=10.0.0.2 actions=output:"s1-eth1"
cookie=0x0, duration=521.259s, table=0, n_packets=431, n_bytes=28582, priority=0 actions=CONTROLLER:65535
*** s2 ***
cookie=0x0, duration=328.670s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=ee:4c:d7:39:16:f0,d_l_dst=e6:12:0a:44:65:dc,nw_src=10.0.0.2,nw_dst=10.0.0.70 actions=output:"s2-eth1"
cookie=0x0, duration=328.669s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=00:00:00:00:00:fe,d_l_dst=ee:4c:d7:39:16:f0,nw_src=10.0.0.254,nw_dst=10.0.0.2 actions=output:"s2-eth2"
cookie=0x0, duration=521.281s, table=0, n_packets=115, n_bytes=9929, priority=0 actions=CONTROLLER:65535
*** s4 ***
cookie=0x0, duration=328.675s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=ee:4c:d7:39:16:f0,d_l_dst=e6:12:0a:44:65:dc,nw_src=10.0.0.2,nw_dst=10.0.0.70 actions=output:"s4-eth2"
cookie=0x0, duration=328.674s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=00:00:00:00:00:fe,d_l_dst=ee:4c:d7:39:16:f0,nw_src=10.0.0.254,nw_dst=10.0.0.2 actions=output:"s4-eth1"
cookie=0x0, duration=521.275s, table=0, n_packets=647, n_bytes=37954, priority=0 actions=CONTROLLER:65535
*** s3 ***
cookie=0x0, duration=328.691s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=ee:4c:d7:39:16:f0,d_l_dst=92:95:5f:a3:a0:4a,nw_src=10.0.0.2,nw_dst=10.0.0.71 actions=output:"s3-eth1"
cookie=0x0, duration=328.687s, table=0, n_packets=0, n_bytes=0, ip,d_l_src=00:00:00:00:00:fe,d_l_dst=ee:4c:d7:39:16:f0,nw_src=10.0.0.254,nw_dst=10.0.0.2 actions=output:"s3-eth2"
cookie=0x0, duration=521.280s, table=0, n_packets=93, n_bytes=7459, priority=0 actions=CONTROLLER:65535
mininet>
```

Figure 1.3: flow tables after a client subscription specifying **k=2 DIRECT PATH** flows are marked in Yellow to the left.

1.5.1 Sample Application used for testing

The application we used for testing was written in C. The client executes `client.c` code that simply does a request to the specified `VIRTUAL_IP` using standard Unix sockets, it simply sends and hello message to the service and expects to receive one response containing a server hello. The server is also implemented in C and responds to the requests sending the server Hello message.

1.5.2 Transport protocol used in the sample application

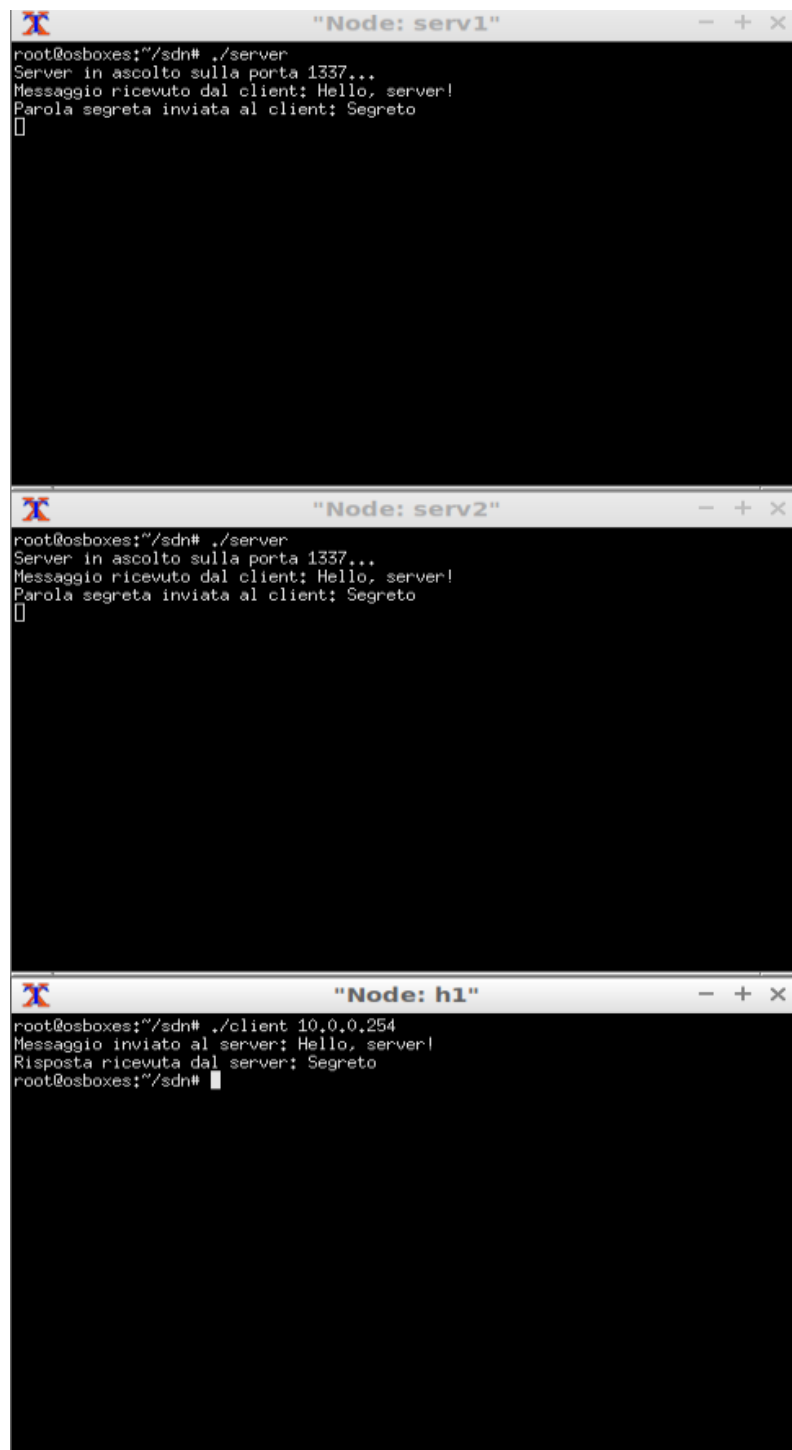
We have chosen to use UDP protocol in the client server application described above. The reason for that was that the logic implemented by the controller only operates on the IP packets, so if we used TCP we wouldn't have idea if the packet flowing back are containing the actual response or are just part of the handshakes that are part of the protocol, so for this testing UDP was the correct choice because we do not have handshakes. Another assumption is that the responses are carried only using one single UDP Datagram for simplicity. If we want to implement this merging logic for a service that uses TCP we can simply extend the controller module.

1.5.3 Correct functioning test

The test for the correct functioning was simply to make a request to virtual ip haven choosen to register using 2 replicated servers. The output of the client and the servers is shown below

1.5.4 Wrong functioning test

We tested the situation were we requested at subscription time for 2 replica server but, at the time of the request one of them is down, so we do not expect a response from the perspective of the client (if this happens in a real situation we need to implement some timeout mechanism to check if the server is still alive, otherwise the controller waits forever for the response).



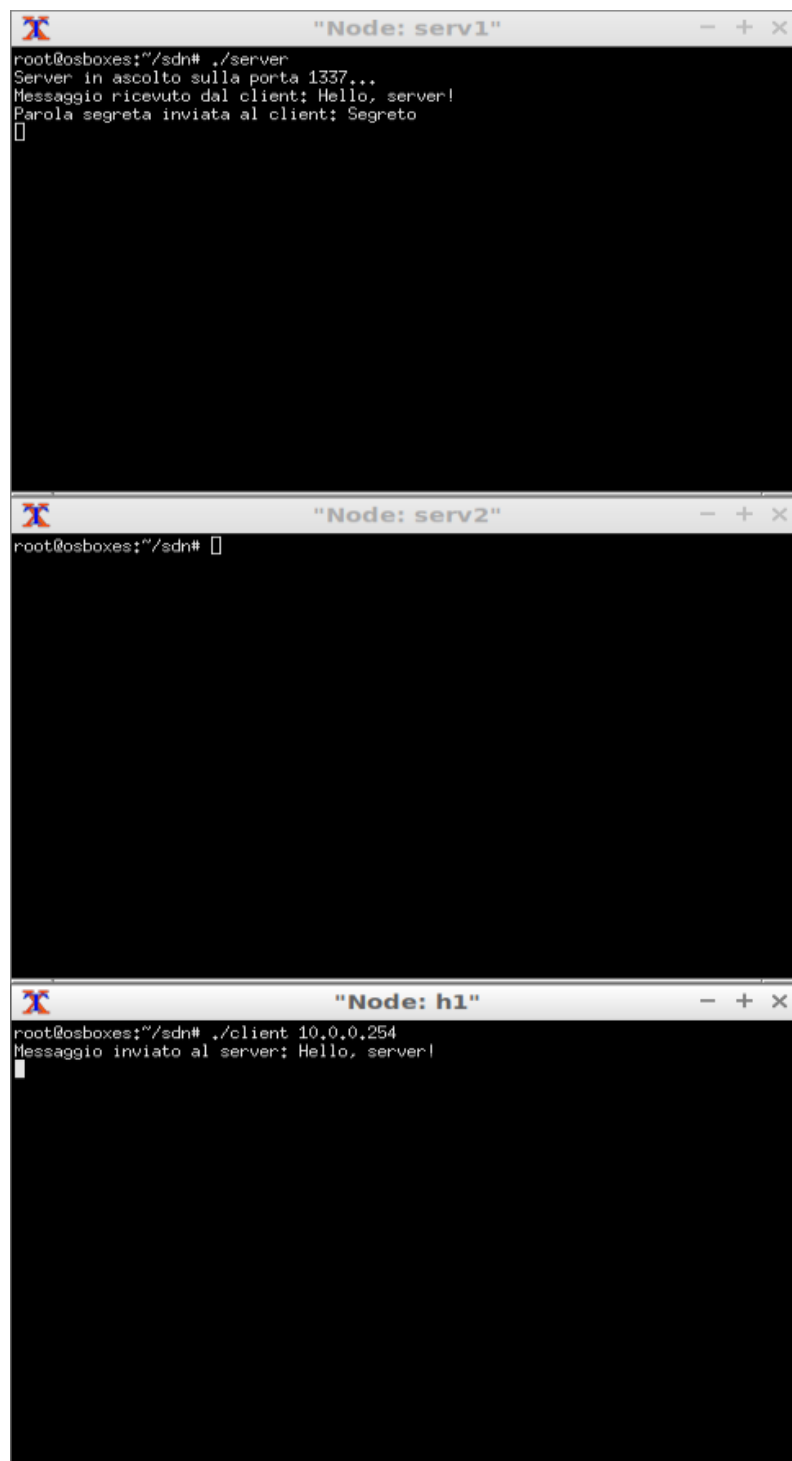
The image displays three terminal windows stacked vertically, each showing the execution of a network application. The top window, titled "Node: serv1", shows a server listening on port 1337, receiving a "Hello, server!" message, and sending back "Segreto". The middle window, titled "Node: serv2", shows the same server behavior. The bottom window, titled "Node: h1", shows a client connecting to the server, sending "Hello, server!", and receiving "Segreto" in response.

```
root@osboxes:~/sdn# ./server
Server in ascolto sulla porta 1337...
Messaggio ricevuto dal client: Hello, server!
Parola segreta inviata al client: Segreto
[]
```

```
root@osboxes:~/sdn# ./server
Server in ascolto sulla porta 1337...
Messaggio ricevuto dal client: Hello, server!
Parola segreta inviata al client: Segreto
[]
```

```
root@osboxes:~/sdn# ./client 10.0.0.254
Messaggio inviato al server: Hello, server!
Risposta ricevuta dal server: Segreto
root@osboxes:~/sdn#
```

Figura 1.4: Correct functioning



The image displays three terminal windows stacked vertically, each with a title bar and a standard Linux-style icon. The top window, titled "Node: serv1", shows the execution of a server program. It reports that the server is listening on port 1337, receives a "Hello, server!" message from a client, and attempts to send back a secret word "Segreto". The middle window, titled "Node: serv2", shows a root prompt with no visible output. The bottom window, titled "Node: h1", shows the execution of a client program that sends a "Hello, server!" message to the server at IP 10.0.0.254. The lack of a response in the middle window indicates a communication failure.

```
"Node: serv1"
root@osboxes:~/sdn# ./server
Server in ascolto sulla porta 1337...
Messaggio ricevuto dal client: Hello, server!
Parola segreta inviata al client: Segreto
[]

"Node: serv2"
root@osboxes:~/sdn# []

"Node: h1"
root@osboxes:~/sdn# ./client 10.0.0.254
Messaggio inviato al server: Hello, server!
[]
```

Figura 1.5: Wrong functioning