Department of Information Engineering and
Computer Science
University of Trento, Italy

Softwarized & Virtualized Mobile Networks

On-demand SDN slicing

Authors:
De Menego Luca
Greco Matteo
Marchioro Nicola

Jan 25, 2023

Contents

1 Introduction

The project involves the development of a solution based on `ComNetsEmu` allowing to activate and de-activate SDN slices on-demand. The slicing approach is topology-based, hence a process needs to be defined that builds multiple virtual networks on top of a physical network infrastructure. This kind of separation is quite useful not only because it allows to spread resources to different groups in a fine-grained manner, but also because it improves security, reduces costs and allows for a more efficient use of resources. Obviously, there is a downside: it also increases the complexity of the network, so it must be properly managed.

There is another vital requirement that must be met: the slicing approach should allow to configure the QoS (Quality of Service) settings for the created virtual networks. In detail, a client should be able to directly control the links bandwidth by limiting it if needed.

The functions exposed by the project can be provided via Command Line Interface or through a web application.

2 Backend

The developed solution is divided in two parts, namely backend and frontend. The backend contains the business logic, and it exposes a RESTful API providing all functionalities a client may need to interact with the infrastructure.

2.1 MiniNET Topology

In this project, we have used MiniNet, an open-source network emulator that allows users to create and test virtual networks on their own computer. It is designed to mimic the behavior of a real network, but it runs entirely in software. It allowed us to create a virtual network topology for experimentation and testing. In detail, the topology consists of five switches connected in a mesh configuration. In a mesh topology, all devices are connected to every other device, providing multiple paths for data transmission. Each switch is then connected to a different host. A visual representation can be seen in Figure 1.

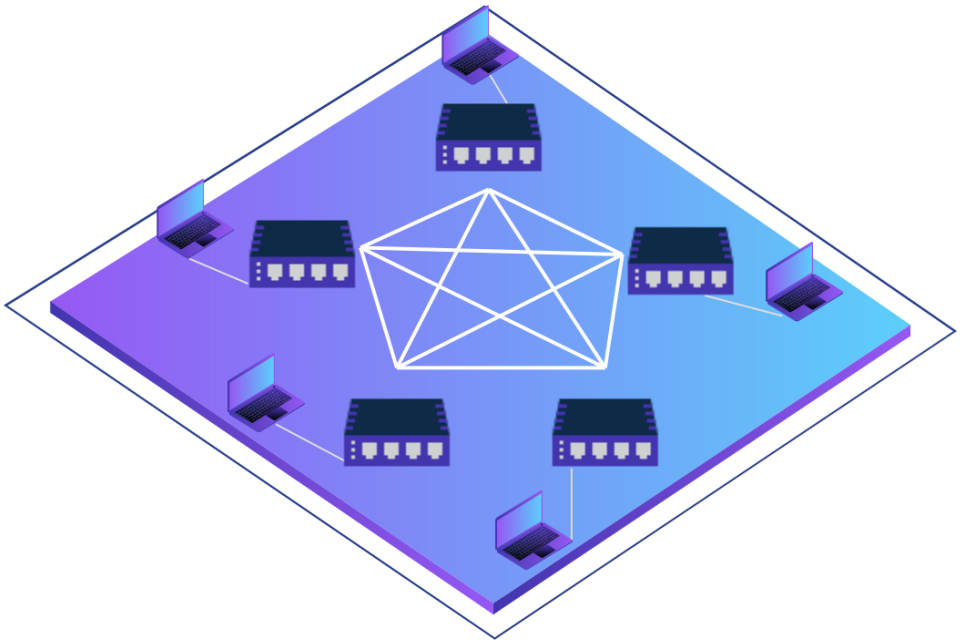

Figure 1: Visual representation of the starting topology

2.2 Architecture

The Ryu architecture used in this project consists of several components that work together to manage the network. In detail, we have:

- `ryu_application`: the entry point for the Ryu application, responsible for starting the main controller and all the other components;
- `switch_stp_rest`: a custom controller module developed by us, that mainly applies the slicing to the network and exposes the RESTful API;
- `stplib`: a module developed by Ryu that implements the Spanning Tree Protocol (STP), which allows for loop prevention;
- `rest_topology`, `rest_qos`, `ofctl_rest`, `rest_conf_switch`: RESTful API modules developed by Ryu providing interfaces for interacting with the network and configuring its elements;
- `ws_topology`: a module exposing a WebSocket interface, that allows real-time network monitoring.

The use of these different modules allows for a high degree of flexibility and customization in managing the network.

2.3 Spanning Tree Protocol

STP is a networking protocol used to ptrevent loops in a network, that can occur when there are multiple paths between two devices, as in our case. This can cause issues such as broadcast storms, making the network become unavailable. The Spanning Tree Protocol creates a logical topology of the network, where only one path between all devices is actually active. The result will be a tree structure, and the device at the root of this tree is called "root bridge".

Each device in the network has a Bridge ID composed of its MAC and a priority value. The tree is constructed based on these values: in general we want the device with lowest BID to be elected as root bridge. This information is collected by all devices through the exchange of Bridge Protocol Data Units (BPDUs). When all these packets have been exchanged, the root bridge can be univocally elected and all other devices can calculate the best path to it. After the calculation, each device will only have one designated active port allowing to forward traffic to the root: all other ports become blocked, i.e. they can only receive BPDU and not forward traffic.

This protocol has been integrated within our architecture by using a module pre-built by Ryu called `stplib`. There are two main things that must be configured when using this module:

- the switches' priority - in our case we filled the priority with random values, to simulate the fact that in a real network topology some links may be better than others, hence some switches may have higher priority;
- the time needed for the Spanning Tree calculation - it is usually set between 30 and 50 seconds, but since our topology is quite small we set the `fwd_delay` to 8 seconds.

Figure 2 shows how the result of the Spanning Tree Protocol application may look like.

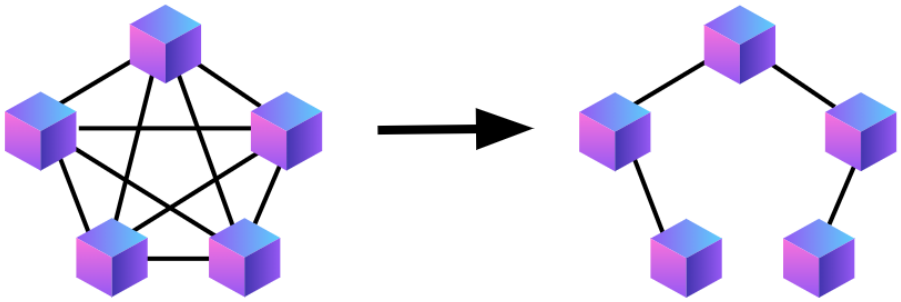

Figure 2: Visual representation of a possible result of a STP application

2.4 Slicing

Slicing refers to the process of creating multiple virtual networks on top of a physical network infrastructure. Each slice typically has its own set of rules and policies for managing traffic, without interfering with other slices. In this project slicing has been implemented by using a SDN controller that logically divides the network. In detail, we managed to do it by exploiting the STP protocol and logically enabling/disabling links. More information about this is provided in Subsection 2.4.1.

2.4.1 Topology Slicing

While for instance service slicing is focused on providing different sets of network services to different slices, topology slicing mainly focuses on providing different logical topologies to different slices. As previously described, this has been implemented by us using the `stplib` Ryu's module. The stplib API provides functions such as `link_up` and `link_down`, which allow for the creation and removal of virtual links between switches. By using these functions, we were able to create virtual networks with their own logical topologies, such as a tree or bus.

For example, the `link_up` function can be used to add a new link between two switches, creating a new path for data to travel through the network. Similarly, the `link_down` function can be used to remove a link between two virtual switches, breaking an existing path and changing the logical topology of the network. Following is a template representing a slicing, while Figure 3 shows the snippet of code responsible of handling it:

```
{
    "1": {"5": [1,3], "1": [5], "3": [5]},
    "2": {"1": [5], "5": [1]},
    "3": {},
    "4": {"1": [5], "5": [1]},
    "5": {}
}
```

In this template:

- the outer key represents the Switch ID;
- the inner key represents the Input port;
- the values within the arrays are the possible output ports.

Apart from the use of these functions provided by `stplib`, we also enforce the slice application whenever a packet is received, by sending it only to the ports the slice allows us to. Figure 4 shows how this is achieved.

It's worth mentioning that this implementation is a simple one, it may not cover all the features that a production network may need, but it provides a solid foundation for further development and experimentation.

```
for switch in self.get_switches():
    switch_id = dpid_lib.str_to_dpid(switch["dpid"])
    bridge = self.stp.bridge_list[switch_id]
    for port in switch["ports"]:
        port_id = self.str_to_port_no(port["port_no"])
        port = self.dpset.get_port(switch_id, port_id)
        if (port_id not in flatten(list(self.slice_to_port[str(switch_id)].values()))
                and str(port_id) not in list(self.slice_to_port[str(switch_id)].keys())):
            # disable the port
            bridge.link_down(port)
        else:
            # enable the port
            bridge.link_up(port)
```

Figure 3: Code applying a given topology slicing

```
# Check if the slice allows this communication
if str(in_port) in self.slice_to_port[str(dpid)]:
    # Learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    # If the destination is known, send the packet to the destination
    if (dst in self.mac_to_port[dpid]
            and self.mac_to_port[dpid][dst] in self.slice_to_port[str(dpid)][str(in_port)]):
        out_port = [self.mac_to_port[dpid][dst]]
    else:
        # Flood the packet to all possible ports (based on the slice restrictions)
        out_port = self.slice_to_port[str(dpid)][str(in_port)]

    # Create the actions list: send the packet to each port in out_port
    actions = [parser.OFPActionOutput(int(out)) for out in out_port]
else:
    # The slice doesn't allow this communication, no action is taken
    self.logger.info("Can't communicate due to slice restrictions, switch %s, in_port: %s, sl
    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=[], data=None)
    datapath.send_msg(out)
    return
```

Figure 4: Code enforcing a slice in the packet in handler

2.4.2 QoS configuration

In addition to topology slicing, we also implemented Quality of Service (QoS) configuration by using virtual queues in Ryu. QoS is a set of techniques used to manage and prioritize network traffic, ensuring that critical applications and services receive the necessary bandwidth and resources. In SDN, this can be achieved by creating virtual queues on the switches and managing them through the SDN controller.

Ryu provides a RESTful API for creating and managing virtual queues on virtual switches. By using this API, we were able to create and configure them with different parameters such as the maximum and minimum rate. Once the virtual queues are configured, we can use the Ryu controller to apply different QoS policies to different slices of the network. For example, we can configure a high priority virtual queue for a critical application and a low priority virtual queue

for non-critical traffic. This allows us to ensure that critical applications and services receive the necessary bandwidth and resources, while also managing the overall network traffic. Following is a template representing the QoS configuration for a certain slice, while Figure 5 shows the snippet of code responsible of handling it:

```
[{
  ‘‘switch_id’’: 1,
  ‘‘port_name’’: ‘‘s1-eth5’’
  ‘‘match’’: [{
    ‘‘nw_dst’’: 10.0.0.2,
    ‘‘nw_src’’: 10.0.0.1
  }, ...],
  ‘‘queues’’: [{
    ‘‘queue’’: ‘‘0’’,
    ‘‘max_rate’’: ‘‘500000’’
  }, ...],
}, ...]
```

In this case the template is quite self-explanatory:

- the Switch ID and Port Name identify specifically the switch we want to address;
- source and destination tell us when the queue should be applied, based on the packet's information;
- the max rate is the maximum bandwidth allowed when the queue is in use.

2.5 Ryu Events

In order to monitor the network infrastructure, we have used Ryu events to gather information about the network's state.

Ryu provides a built-in event system that allows different components of the controller to communicate with each other. These events can be used to inform the controller of changes in the network's state, such as link failures, and to trigger actions, such as reconfiguring the network. Some of the events are already available and exposed by different components, while some of them have been created by us specifically for this project.

In our case, we have mostly used Ryu events to gather information about the network's state, such as the status of the links, and we handle and publish this information in a websocket. This allows us to monitor the network in real-time and to react quickly to any changes in the network's state.

2.6 Rest API

In addition to monitoring the network using Ryu events, we also developed a RESTful API that exposes important information about the network state and provides various functions to manage slices and Quality of Service (QoS) configurations, such as routes to create, delete or update slices and virtual queues.

The RESTful API provides an interface for interacting with the network, allowing users to retrieve information about the network's state and to configure the network. The API is designed to be easy to use and understand, using standard HTTP methods, such as GET, POST, and DELETE, to retrieve and modify data.

```
requests.put(
    'http://localhost:8080/v1.0/conf/switches/{switch_id}/ovsdb_addr',
    data='"tcp:127.0.0.1:6632"'
)

requests.post('http://localhost:8080/qos/queue/{switch_id}', json.dumps({
    "port_name": qos_configuration["port_name"],
    "type": "linux-htb",
    "max_rate": "10000000000",
    "queues": [{"max_rate": qos_configuration['max_rate']}]
}))

requests.post('http://localhost:8080/qos/rules/{switch_id}', json.dumps({
    "match": {
        "nw_dst": qos_configuration["nw_dst"],
        "nw_src": qos_configuration["nw_src"],
    },
    "actions": {
        "queue": qos_configuration["queue"]
    }
}))
```

Figure 5: Code applying a given QoS configuration on a slice

The API we developed has been documented using OpenAPI 3.0, which is a specification for describing RESTful APIs. OpenAPI provides a standardized format for documenting the API's endpoints, parameters, and responses, making it easy for developers to understand and use the API. The documentation is available under http://localhost:8080/docs/index.html when starting the application, and it is also available online in SwaggerHub at https://app.swaggerhub.com/apis/lucademenego99/on-demand-sdn-slices/1.0.0. Figure 6 shows the most important endpoints exposed by it.

SwaggerHub is a platform for designing, building, and documenting RESTful APIs. It allows developers to create, test, and document APIs using the OpenAPI specification. The platform also provides a variety of tools for collaborating on API development, such as versioning, commenting, and reviewing. By using SwaggerHub, we were able to create and publish the documentation for the API, making it accessible to other developers.

2.7 WebSocket

The ws_topology component is a part of the Ryu application that is responsible for listening to Ryu events and publishing the information these events provide to a WebSocket. A WebSocket is a protocol for real-time, bidirectional communication between a web browser and a server. It allows for low-latency, high-frequency communication, which is ideal for applications that require real-time updates, such as network monitoring. In this particular case, this allows us to monitor the network in real-time and to react quickly to any changes in the network's state.

We use the WebSocket by opening a connection with it in a specially-crafted web application exposing the main functionalities of the project. This web application allows users to monitor the network's state in real-time, and to interact with the network through the RESTful API. The web application can be accessed by opening a browser and connecting to http://localhost:8080 when the application is running.

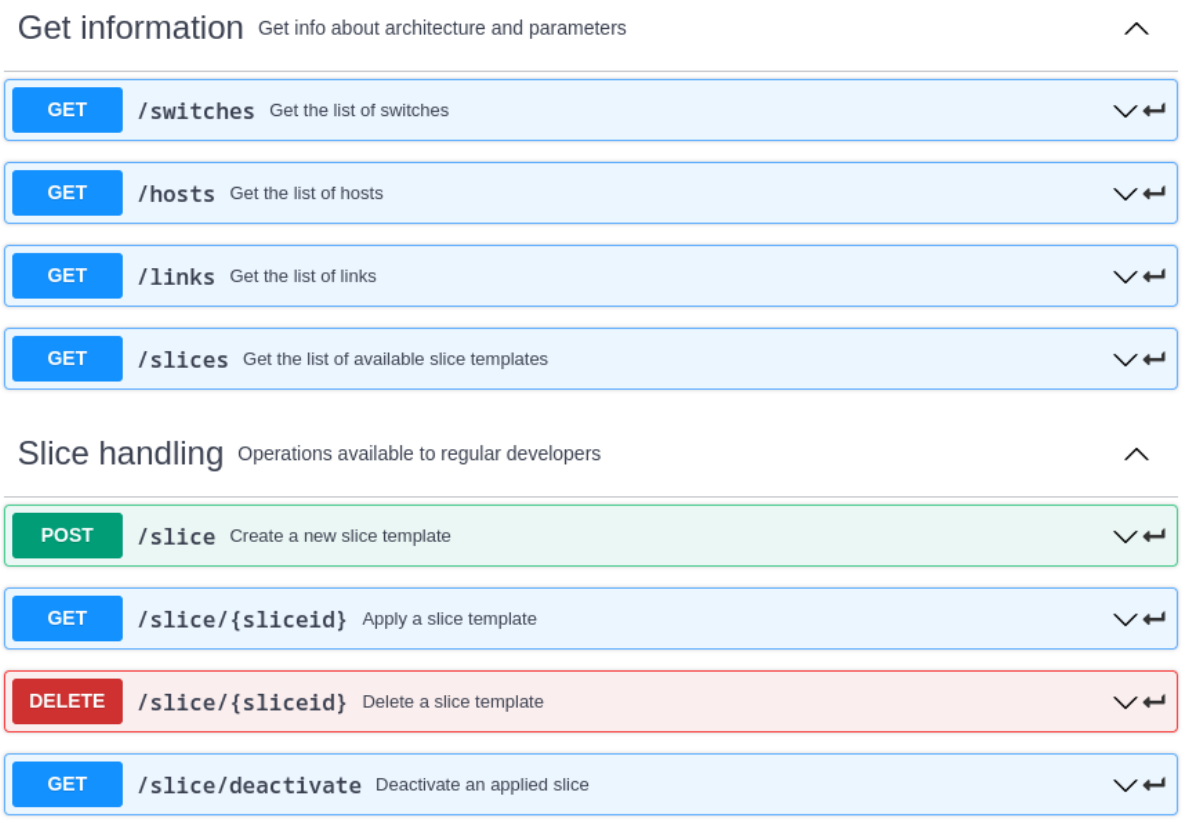

Figure 6: List of the main endpoints exposed by the developed RESTful API

3 Frontend

The second part of the project involves the frontend, i.e. a simple web application that by opening a connection with the exposed WebSocket explained in Subsection 2.7 and by communicating with the developed RESTful API provides all main backend functionalities.

3.1 Events Handling

In the front-end, we open the websocket by using JavaScript and the WebSocket API. This is done by creating a new WebSocket object, and specifying the URL of the server to connect to. Once the connection is established, we can listen to events emitted by the server. The main events that we use to update the Web application are:

- SliceUpdate, which is generated when a slice is activated or deactivated by any client. When the event is captured it triggers an update of the topology preview presented on the homepage.
- SliceListUpdate, which is triggered when a slice is added or removed to the template list (slice_templates.json). When it occurs, it reloads the template list in the side menu of the homepage.

3.2 D3.js

When receiving new information about the topology through the WebSocket, we update the user interface by using the D3.js library. D3.js is a JavaScript library that allows for the manipulation of documents based on data, and it is often used for creating interactive data visualizations.

We use D3.js to bind the new data received from the WebSocket to the elements in the user interface, and to update the visual representation of the network's topology (e.g. switches and links). This allows us to display the current state of the network, and to update it in real-time as new information is received.

3.3 Functionalities

The web application consists of two pages, namely the *Homepage* and the *New slice* page. On the *Homepage*, the user is able to perform the following tasks:

- View the network topology and the configurations of the various switches in case slices have been applied.
- Apply predefined slices, which simulate common network topologies such as Bus, Tree and Star.
- Apply custom slices, which can be created in the *New slice* page.
- Deactivate applied slices, allowing the network to revert back to a mesh network configuration.

When the user navigates to the *New slice* he has the following capabilities:

- Create a new slice: the user can create a custom slice by that meets their specific requirements and needs. Specifically, he can select the source and destination hosts, choose the route to connect those hosts by adding switches to the slice and can also add a QoS rule specifying the rate.
- Preview new slices before saving: the user can preview the custom slice to verify that the configuration is as desired before saving it. This allows the user to make any necessary changes before the custom slice is saved.
- Name custom slices for identification: the user can name the custom slice to make it easier to identify and locate later. This helps with organization and reduces the time required to locate a specific custom slice.
- Save custom slices for future use: the user can save the custom slice for future use. This allows the user to easily reapply the custom slice to the network topology without having to reconfigure it each time it is needed.

4 Solution in action

As an example of how the developed project can be used, we provide here a walk-through of what must be done to set it up and use its main features. First of all, the project relies on ComNetsEmu, a testbed and network emulator that already provides all the dependencies we need to start the Ryu application and the MiniNET topology. More information about it can be found here: https://git.comnets.net/public-repo/comnetsemu. We personally installed it by cloning the repository and using Vagrant.

If installed in this way, the project can be easily started by following these steps:

1. start the Virtual Machine and get access to a shell
2. `cd comnetsemu/app/realizing_network_slicing/`
3. `git clone https://github.com/lucademenego99/on-demand-sdn-slices.git`

4. `cd on-demand-sdn-slices`
5. `sudo ovs-vsctl set-manager ptcp:6632`
6. `ryu run -observe-links ryu_application.py`
7. open another terminal
8. `sudo ./mesh-network.py`

If you clone the project to a different directory, you may get some errors when trying to run the Ryu application. In fact, the file `ryu_application.py` imports the various modules through an absolute path. In that case, please manually update the paths to the various components.

4.1 Homepage

If everything works as expected, the Ryu Controller should start running the STP calculation, and the web application should be available at `http://localhost:8080`. When the STP calculation is finished (i.e. when the ports are either in FORWARD or BLOCK state), the web application can be opened and used to change the infrastructure. For instance, let's apply the *Islands* slice:

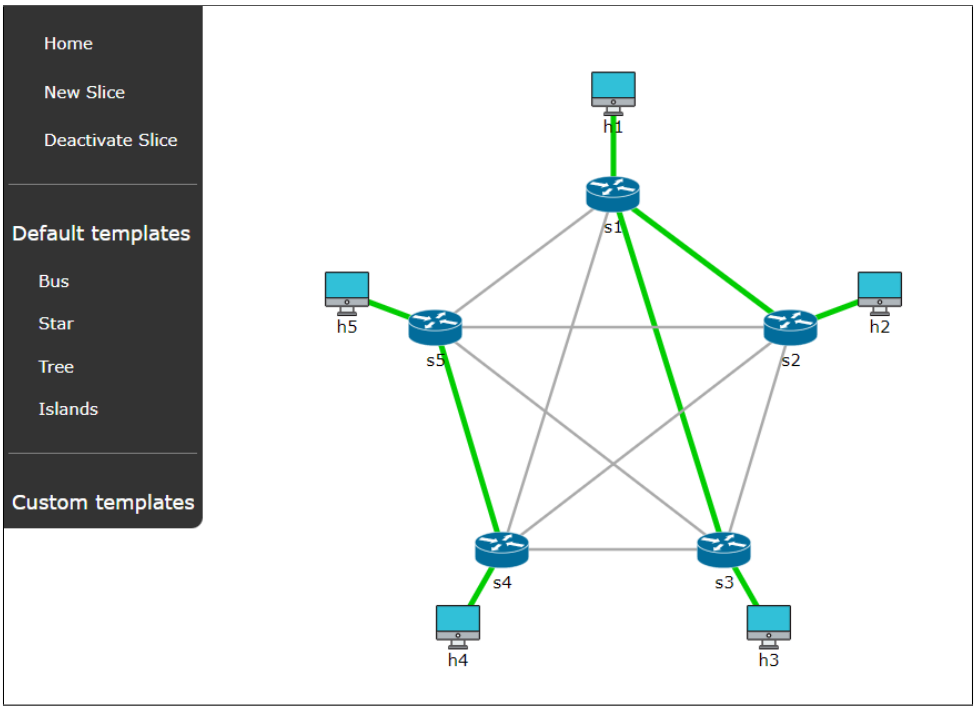

Figure 7: Homepage of the Web application

When clicking the button the web application will call the RESTful API exposed by the backend requesting for the slice employment, and after a moment the WebSocket will receive a message with information about the applied slice. Now the active links you see in the page will accurately represent the slice. You can also check the slice has really been applied by going into the MiniNET Command Line Interface and perform a pingall:

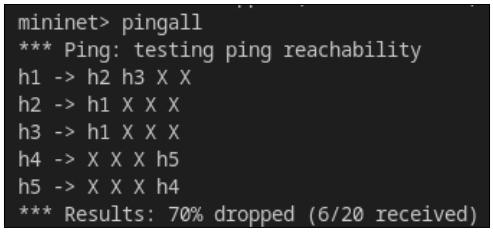

Figure 8: Result of a pingall command run with "Islands" slice applied

To deactivate the slice, we can simply click the button called *Deactivate slice*. After removing the slice we can test it by executing a pingall.

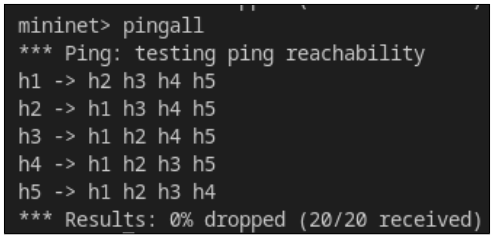

Figure 9: Result of a pingall on the mininet terminal with no slice applied

Among all the templates, Star is the one that provides a complete QoS configuration. After having enabled it, the user can view a list of all installed virtual queues by clicking on a switch. For example, switch 2 has a QoS configuration that limits bandwidth to `1000 KBit/sec`, and this can be confirmed by using `iperf`, a network testing tool mainly used to measure the performance and quality of network connections. Running an `iperf h1 h2` from the Mininet console will confirm the set limited bandwidth, as illustrated in Figure 10.

Figure 10: Result of an iperf between h1 and h2, when Star is the applied slice1

4.2 New Slice

Users can also create new custom slices by navigating to the *New slice* page. To create a custom slice you can select the source and destination hosts and decide if you want to add a QoS rule by specifying the max-rate in bps.

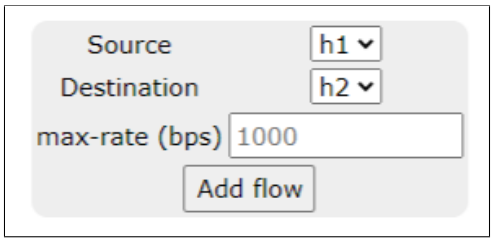

Figure 11: Source and destination hosts selection, max rate configuration in bps

After clicking the button on the table called *Add flow*, an initial representation of the route between the selected hosts is shown on the page. By clicking the "+" button the user can add a switch to the flow, effectively selecting how the source and destination should be connected. To add multiple flows the user can select others source and destination hosts and configure them in

the same ways as before. Furthermore, the user can have a preview of the flow that has been configured by clicking on the *Details icon*.

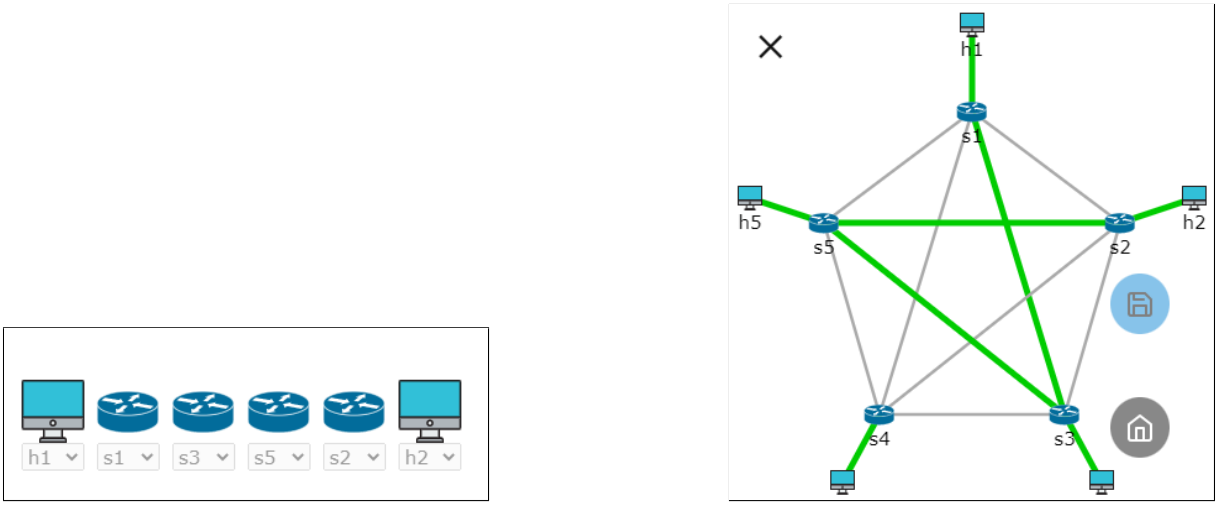

Figure 12: Left: depiction of the added flow. Right: slice preview when Details icon is selected

When the user is satisfied with the changes, he can name the slice and save it for future use. This allows the user to easily apply the custom slice to the network topology without having to reconfigure it each time it is needed.

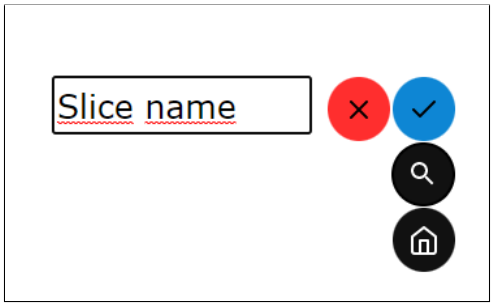

Figure 13: Naming and confirmation options when saving slice

5 Conclusion

In conclusion, we have developed a project that demonstrates the use of Software-Defined Networking (SDN) for topology slicing and Quality of Service (QoS) configuration. The project uses Ryu, a Python-based SDN controller, and MiniNet, an emulator for creating a virtual network.

We have implemented topology slicing and QoS configuration by using various Ryu functionalities, allowing for the management and prioritization of network traffic. In addition to the controller's functionality, we have also developed a RESTful API that exposes important information about the network state and provides various functions to manage slices and QoS configurations. We have also exposed this functionality through a WebSocket, which allows us to monitor the network in real-time and to react quickly to any changes in the network's state.

Finally, we have also developed a web application that allows users to monitor the network's state in real-time, and to interact with the network through the RESTful API. This web application is developed with D3.js library, which allows us to display the current state of the network, and to update it in real-time as new information is received.