UNIVERSITÀ DI TRENTO

Department of Information Engineering and Computer Science

ON-DEMAND SDN SLICING

Networking - Module II

Alessio Raffini - Francesco Vinciati - Marco Zani

Anno accademico 2023/2024

Abstract

This report presents the design and implementation of an on-demand Software-Defined Networking (SDN) slicing solution utilizing **Comnetsemu**, a virtual environment for network simulation, employing *Mininet* and *Ryu*. Developed as part of the Module 2 of the Networking course at the University of Trento, this project focuses on addressing the dynamic resource allocation challenges in modern network environments. Leveraging Python scripts and a graphical user interface (GUI), the system allows for the creation and management of network slices tailored to specific requirements. The report outlines the architecture of the solution, detailing the integration of Mininet and Ryu, along with the functionalities of the GUI. Furthermore, it discusses the implementation challenges encountered and evaluates the performance of the proposed solution through simulation experiments. The results demonstrate the effectiveness and scalability of the developed on-demand SDN slicing approach, showcasing its potential for optimizing resource utilization and enhancing network management in diverse environments.

Contents

1 Project requirements

The goal of this project is to implement a network slicing approach utilizing ComNetsEmu. The primary objective is to enable dynamic activation and deactivation of network slices through Command Line Interface (CLI) or Graphical User Interface (GUI) commands. Key requirements for the project include:

- **Single SDN Controller:** The solution must utilize a single SDN controller, such as RYU, to manage the network slicing operations efficiently.
- **Dynamic Activation and Deactivation:** The system should support on-demand activation and deactivation of different network slices. This means that users can dynamically create, modify, activate, and deactivate slices based on changing network requirements.
- **Slice Description:** The project allows flexibility in describing network slices. While specific templates can be employed, the system must enable the identification of essential parameters including:
  - **Flows:** The ability to specify and differentiate traffic flows within each network slice, defining their characteristics and requirements.
  - **Topology:** Defining the network topology associated with each slice, including the arrangement of switches, routers, and hosts, ensuring isolation and customization.
  - **Link Capacity Allocation:** Each slice should be allocated a percentage of link capacity, allowing for efficient resource utilization and QoS management.

The project team has the freedom to determine the specific methodologies and algorithms for implementing the aforementioned requirements. Additionally, the project should aim for a user-friendly interface, facilitating seamless interaction with the slicing functionalities through either CLI or GUI commands. This section outlines the fundamental objectives and guidelines that will govern the development and implementation of the On Demand SDN Slicing solution within the ComNetsEmu environment.

2 Implementation

In this project, an intuitive graphical user interface (GUI) has been developed using the GTK4 module to ensure a pleasant user experience, particularly for users less familiar with command line interfaces (CLI). The GUI provides a user-friendly environment for interacting with the network slicing functionalities, offering ease of navigation and control.
For network topology, a template based on 5 interconnected switches and a total of 10 hosts (2 per switch) has been employed. This standard template serves as a basis for network slicing operations. However, users have the flexibility to create customized slices

according to their specific requirements. Through the dedicated interface, users can select which hosts and switches to include in their slice, tailoring the network configuration to suit their needs.
The Ryu module has been utilized to implement the SDN controller, providing the necessary intelligence and management capabilities for the network slicing operations. Leveraging Ryu's features, the system effectively orchestrates the network resources and maintains the desired slice configurations.
The GUI facilitates dynamic activation and deactivation of network slices. Users can seamlessly turn on or off one or more slices, with real-time feedback on the percentage of link utilization for each slice. Additionally, the interface displays detailed information regarding the devices connected within each slice, thereby visualizing the topology created within that slice.
Overall, the combination of a user-friendly GUI, customizable network slicing options, and efficient control through the Ryu controller enhances the usability and effectiveness of the On Demand SDN Slicing solution developed for ComNetsEmu.

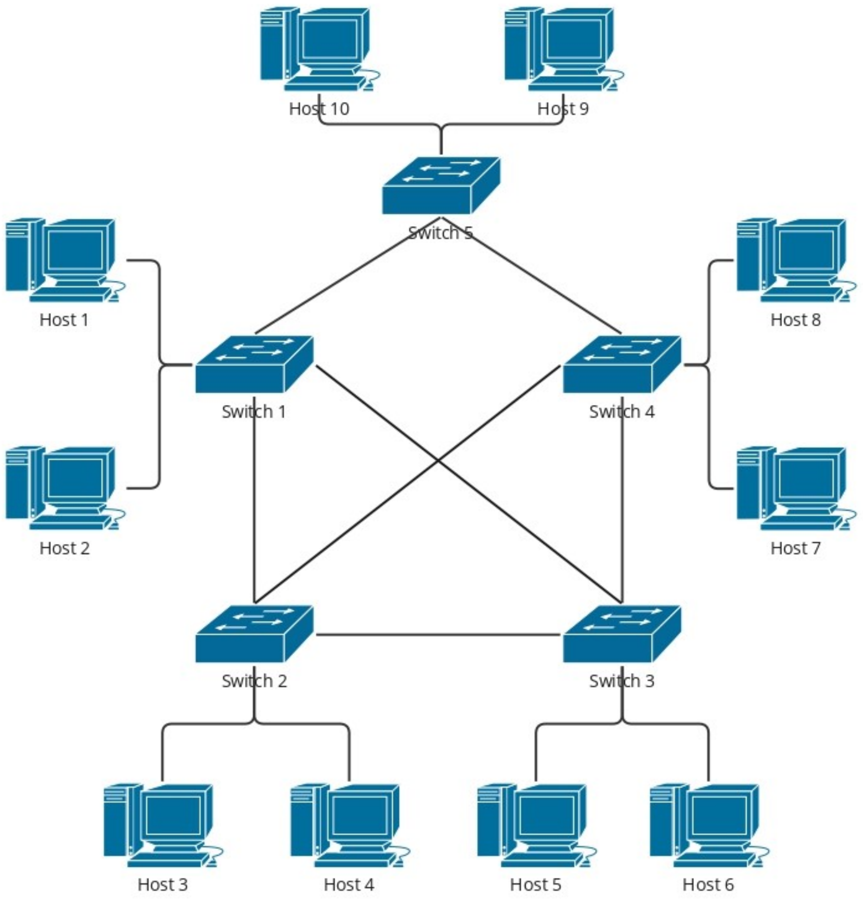


Figure 1 - Network topology

2.1 QoS

The implementation of Quality of Service (QoS) has been applied to all links through a user-defined maximum bandwidth setting for each individual slice. Specifically, users can configure the percentage of maximum bandwidth utilization based on 10 predefined slots (10%/20%/30%/.../100%).
This setting is then received by the controller, which applies it to the respective links upon packet reception using OpenFlow protocol to send the configuration to the network. Implementing QoS in this manner ensures that network resources are allocated efficiently and according to user-defined priorities, enhancing the overall performance and reliability of the network.

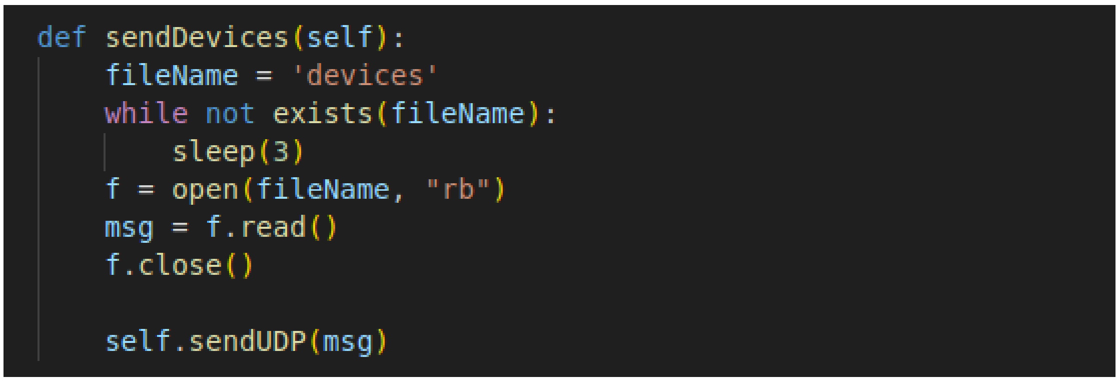
3 Backend

The back-end is composed by the class **Slicer** who manages, creates and shares the slices configurations. In detail, the slicer object contains the Udp socket used for exchanging data with the controller, the list of profiles, and index to keep track of the active profile and a datastructure representing the network topology.

3.1 Slicer

The slicer offers 2 main functions: The initialization and the ability to toggle a specific profile.

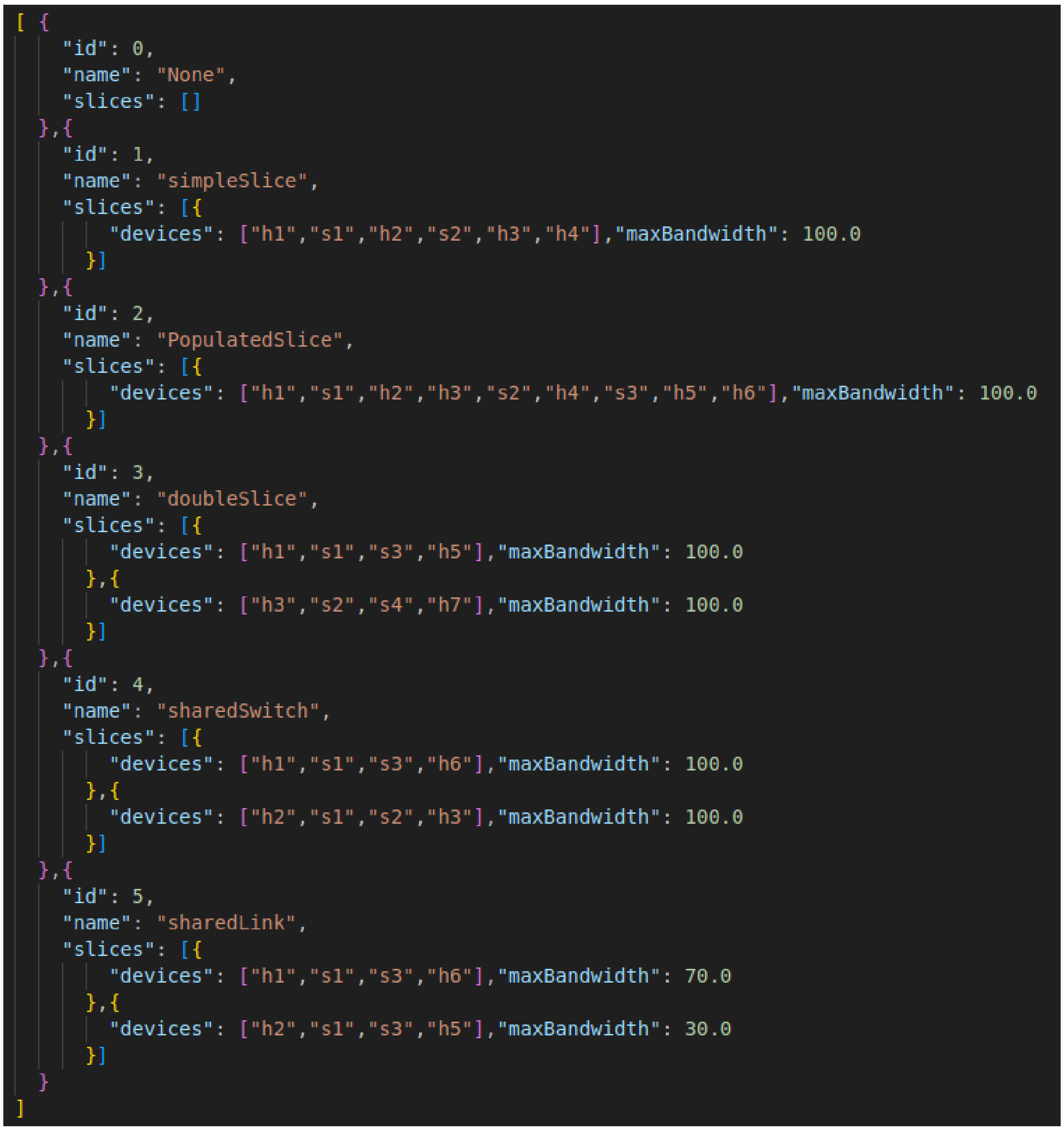
Initialization The inizialization requires the slicer to load all the data from the *profiles.json* file and convert it to usable Profile objects. The JSON file is organized to store id and name of a profile, plus an array of slices, each element containing the list of devices and a minBandwidth value Afterwards the slicer creates an empty Topology object and loads the different links in the network topology from mininet. Lastly it takes the association between hostname, and its IP and MAC addresses and sends it to the controller using the *sendDevices()* function.



```
def sendDevices(self):
    fileName = 'devices'
    while not exists(fileName):
        sleep(3)
    f = open(fileName, "rb")
    msg = f.read()
    f.close()

    self.sendUDP(msg)
```

Figure 2 - sendDevices() function

In the next image, it's possible to see how profiles are configured from a software point of view, using a JSON structure, within the profiles.json file.



```
[ {
    "id": 0,
    "name": "None",
    "slices": []
  },{
    "id": 1,
    "name": "simpleSlice",
    "slices": [{
        "devices": ["h1","s1","h2","s2","h3","h4"],"maxBandwidth": 100.0
      }]
  },{
    "id": 2,
    "name": "PopulatedSlice",
    "slices": [{
        "devices": ["h1","s1","h2","h3","s2","h4","s3","h5","h6"],"maxBandwidth": 100.0
      }]
  },{
    "id": 3,
    "name": "doubleSlice",
    "slices": [{
        "devices": ["h1","s1","s3","h5"],"maxBandwidth": 100.0
      },{
        "devices": ["h3","s2","s4","h7"],"maxBandwidth": 100.0
      }]
  },{
    "id": 4,
    "name": "sharedSwitch",
    "slices": [{
        "devices": ["h1","s1","s3","h6"],"maxBandwidth": 100.0
      },{
        "devices": ["h2","s1","s2","h3"],"maxBandwidth": 100.0
      }]
  },{
    "id": 5,
    "name": "sharedLink",
    "slices": [{
        "devices": ["h1","s1","s3","h6"],"maxBandwidth": 70.0
      },{
        "devices": ["h2","s1","s3","h5"],"maxBandwidth": 30.0
      }]
  }
]
```

Figure 3 - Profiles.json file

At the topology level, we can represent the described profiles as follows.

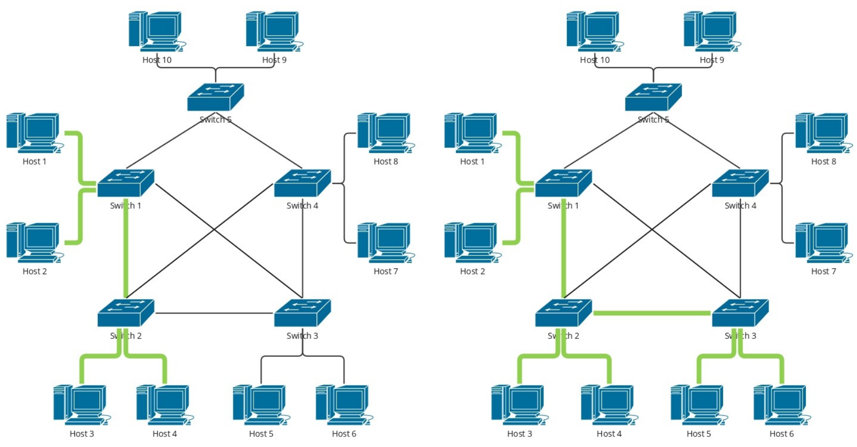


Figure 4 - Slice 1 (left) & Slice 2 (right)

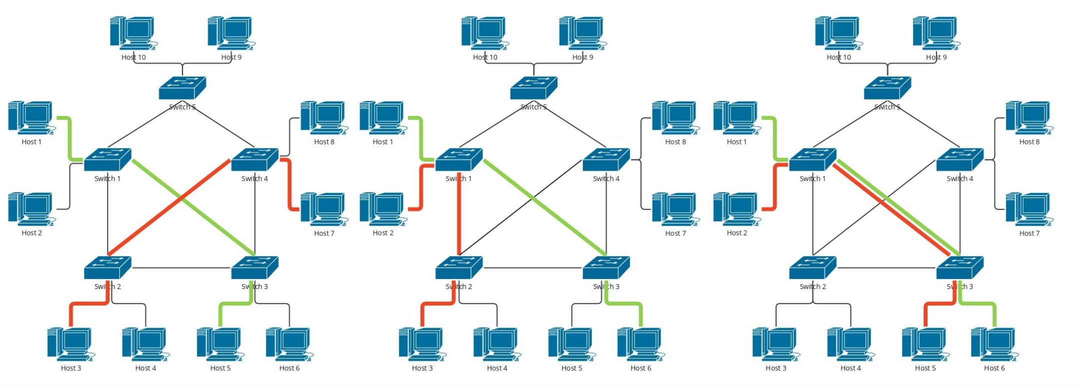
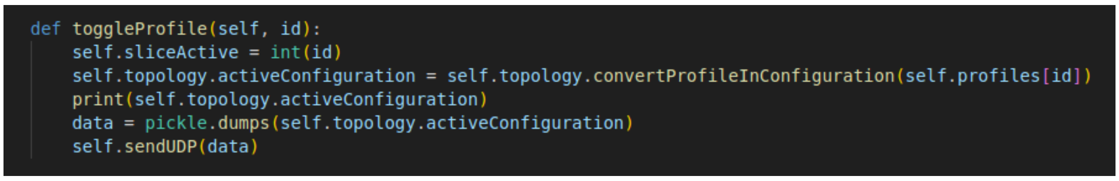


Figure 5 - Respectively Slice 3, 4 and 5

Enabling a profile The other function, *toggleProfile()*, uses the index passed as argument, and requests to the Topology to convert the profile using that index into an usable controller configuration. It then converts the configuration into a stream of bytes and sends it to the controller using the *sendUDP()* function. The convertion into a bytestream using the pickle library is important because it automatically manages the convertion from bytestream into a dictionary instead of needing to manually parse a str into the dictionary.



```
def toggleProfile(self, id):
    self.sliceActive = int(id)
    self.topology.activeConfiguration = self.topology.convertProfileInConfiguration(self.profiles[id])
    print(self.topology.activeConfiguration)
    data = pickle.dumps(self.topology.activeConfiguration)
    self.sendUDP(data)
```

Figure 6 - toggleProfile() function

3.2 Topology

The Topology class is responable with managing the information regarding the network topology, keeping a copy of the current active configuration and converting a given profile into a controller configuration.

Storing devices data Each end point is saved as a dictionary using the name as key, and using as value a list of tuples containing the other connection device and the port used. Because the data passed from mininet lists only links, the topology is populated using the *addLink* function, who tries to source and destination devices, if not already present, to the topology, and then appends the link data to each device connections list, always if not already present.

Converting profiles into a configuration The *convertProfileInConfiguration()* function accepts a profile as argument, then for each slice extracts all the contained devices list,their amount and how much bandwidth must be reserved. Then for each device extract all the data from the topology regarding that element. With all this informations, uses the functions provided by the *floydWarshall.py* file to build up the configuration. In order:

1. Initializes an empty table the size of all devices in the slice
2. Uses the Floyd-Warshall algorithm to create the next hop matrix
3. Removes all cells where there is no connection
4. Converts the matrix in a dictionary
5. Inserts the port necessary for the next hop
6. Removes all the records that do not regard switches After performing this actions on all slices, it lastly compares all the reserved bandwidth requiremes, so that on every link can be detected the most efficient bandwidth separation.

3.3 Floyd-Warshall

The *floydWarshall.py* file is an utility file providing a list of function used to calculate the shortest path between all couples of nodes and converting this table in a configuration usable by the controller.

- **initMatrix()** generates an adjacency matrix n x n where n is the number of devices in the slice.
- **compute_next_hop()** applies the Floyd-Warshall algorithm, which uses a Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

- **shrinkTable()** converts the matrix into a list of lists, where at each index corresponds a device, and inside the list are contained tuples indicating which devices are reachable forwarding which device.

  This convertion removes all the empty cells of the matrix, compacting the data.

- **convertToDict()** converts the list of lists into a dictionary, using the device associated with the index as key, and storing the sublist as value.

- **convertPorts()** exchanges the device name used inside the tuples in the dictionary values for the port to use to reach them.

- **extractSwitches()** removes all keys regarding devices that are not switches, since the controller has no use for them.

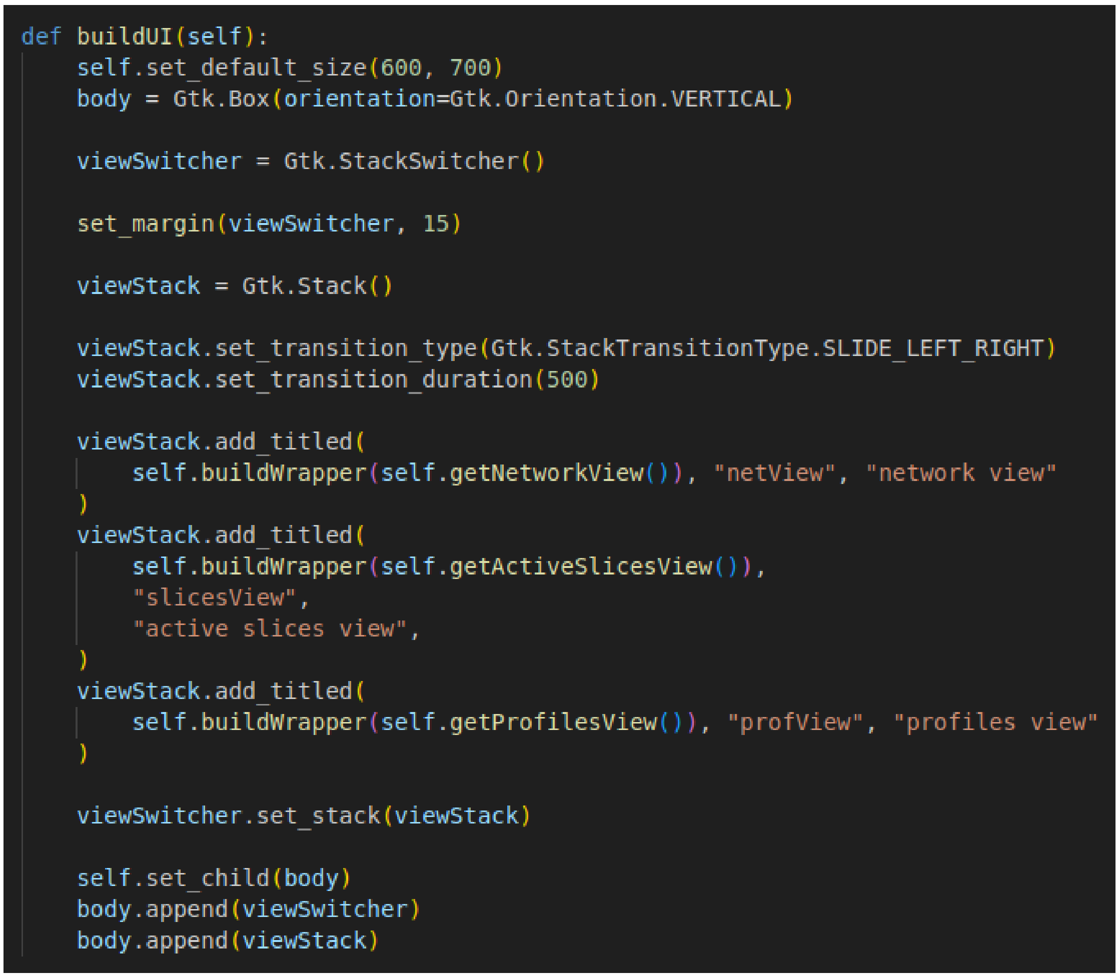
- **add_min_bandwidth()** compares all links which are common between slices, if any are found, it enforces the each link to split bandwidth according to the required value.

4 UI components

Found inside the *slicer_ UI.py* file, the class **Visualizer** is responsable for presenting the application window after calling the class **SlicerWindow** which is responsable to build the UI, while also initializing the Slicer, importing the topology from mininet and sending the list of devices in the net to the controller.It also manages the event *updateProfiles*.

4.1 Applications views

The user interface offers three views: network, profiles and active slices, interchangeable like a carousel using the related buttons. This allows the application to maintain a coherent look and behavior. Each view is loaded individually and incapsulated inside a wrapper. *BuildUi()* is in charge of building the interface by integrating the wrappers of the different sections of the UI.

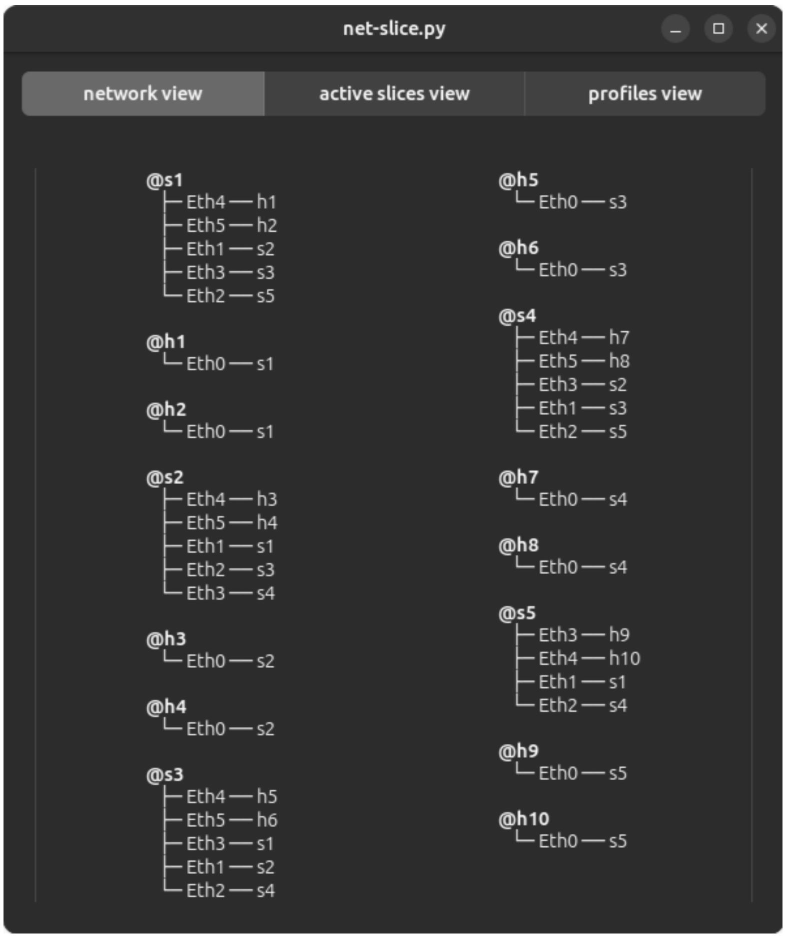


```python
def buildUI(self):
    self.set_default_size(600, 700)
    body = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)

    viewSwitcher = Gtk.StackSwitcher()

    set_margin(viewSwitcher, 15)

    viewStack = Gtk.Stack()

    viewStack.set_transition_type(Gtk.StackTransitionType.SLIDE_LEFT_RIGHT)
    viewStack.set_transition_duration(500)

    viewStack.add_titled(
        self.buildWrapper(self.getNetworkView()), "netView", "network view"
    )
    viewStack.add_titled(
        self.buildWrapper(self.getActiveSlicesView()),
        "slicesView",
        "active slices view",
    )
    viewStack.add_titled(
        self.buildWrapper(self.getProfilesView()), "profView", "profiles view"
    )

    viewSwitcher.set_stack(viewStack)

    self.set_child(body)
    body.append(viewSwitcher)
    body.append(viewStack)
```

Figure 7 - buildUI() function

4.1.1 Network view

This view has been created with the purpouse of presenting in a minimal and clear way, which devices are contained in the network and which port do they use to connect with each other. This is done by formatting the data contained in the slicer topology.
The data is separeted in two columns balanced to occupy less space possible using the *splitArray()* function.

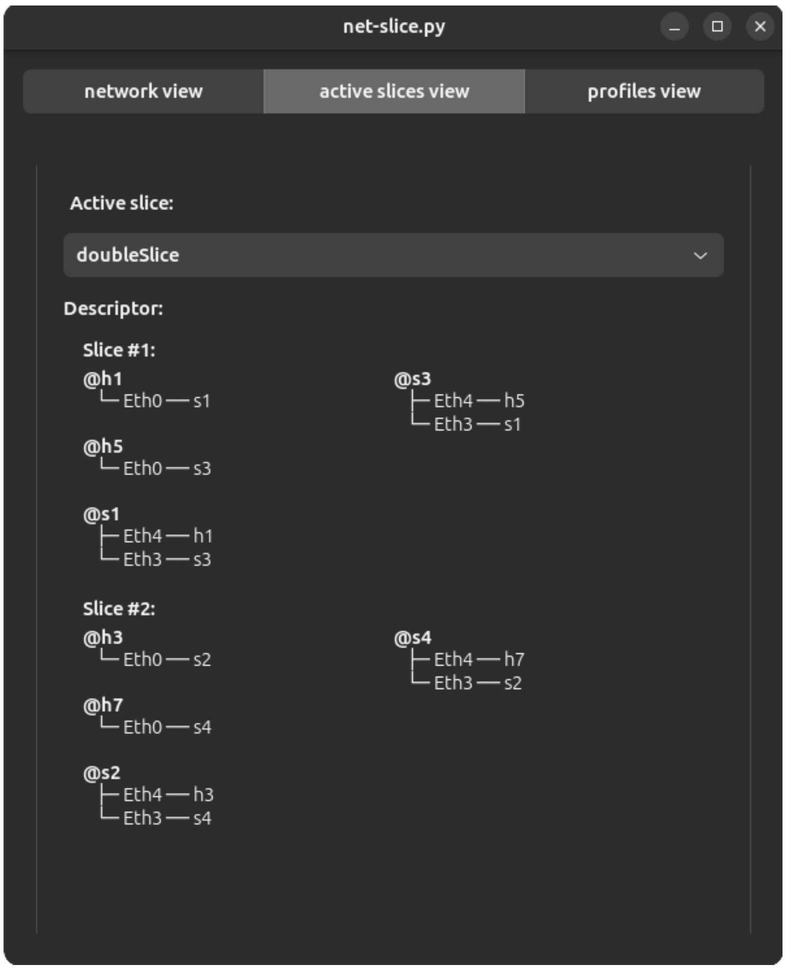


```
@s1                      @h5
 ├─Eth4 ── h1             └─Eth0 ── s3
 ├─Eth5 ── h2
 ├─Eth1 ── s2            @h6
 ├─Eth3 ── s3             └─Eth0 ── s3
 └─Eth2 ── s5
                         @s4
@h1                       ├─Eth4 ── h7
 └─Eth0 ── s1             ├─Eth5 ── h8
                          ├─Eth3 ── s2
@h2                       ├─Eth1 ── s3
 └─Eth0 ── s1             └─Eth2 ── s5

@s2                      @h7
 ├─Eth4 ── h3             └─Eth0 ── s4
 ├─Eth5 ── h4
 ├─Eth1 ── s1            @h8
 ├─Eth2 ── s3             └─Eth0 ── s4
 └─Eth3 ── s4
                         @s5
@h3                       ├─Eth3 ── h9
 └─Eth0 ── s2             ├─Eth4 ── h10
                          ├─Eth1 ── s1
@h4                       └─Eth2 ── s4
 └─Eth0 ── s2
                         @h9
@s3                       └─Eth0 ── s5
 ├─Eth4 ── h5
 ├─Eth5 ── h6            @h10
 ├─Eth3 ── s1             └─Eth0 ── s5
 ├─Eth1 ── s2
 └─Eth2 ── s4
```

Figure 8 - Network View

4.1.2 Active slice view

Used for selecting the slice to apply on the network, the active slice view presents a dropdown menu listing all the profiles that have been loaded from the Slicer. This component is custom made using object inheritance to simplify the component usage. Indeed the normal usage of the GTK dropdown requires the creation of a data model and factory, plus the specification of setups and binding of each dropdown element.



Active slice:

doubleSlice

Descriptor:

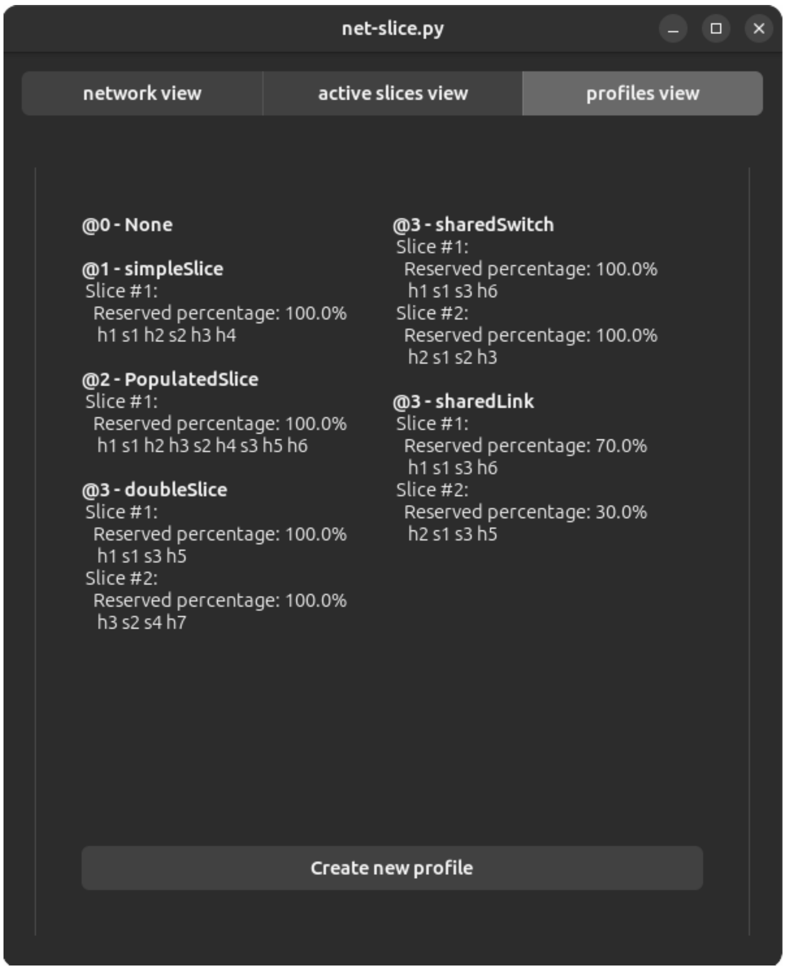
```
Slice #1:
@h1                          @s3
 └─Eth0──s1                   ├─Eth4──h5
                              └─Eth3──s1
@h5
 └─Eth0──s3

@s1
 ├─Eth4──h1
 └─Eth3──s3

Slice #2:
@h3                          @s4
 └─Eth0──s2                   ├─Eth4──h7
                              └─Eth3──s2
@h7
 └─Eth0──s4

@s2
 ├─Eth4──h3
 └─Eth3──s4
```

Figure 9 - Active Slice view

The **SimpleDropDown** widget instead only takes the list of elements and the index of the selected element as constructor's arguments, while automatically managing all the previous requirements. This widget also manages the signal dropDownElementSelected used to update the active configuration through the *updateActiveSlice()* function.
Upon selecting a slice, two functions are called, one to update the dropdown widget with the selected item (*updateDropDown()*) and another to display the network configuration (*updateActiveSlice()*) using a similar formatting as *formatDevices()*.

4.1.3 Profiles view

This last view provides the list of available profiles with the different slices, capacity limits and all devices contained inside each slice, all formatted using the *formatProfiles()* fucntion.
At the bottom of the panel is available a button used to launch a new window used for creating a new profile using the **NewProfileWindow** class. This is more explored in the next chapter. With the creation of a new window, the primary application transfers full focus to it, thereby directing user attention to the new widget while preventing interference with the main program. This is all done in the spawnNewProfileWidget() function, who also connects the newProfileWindowClosed event to the newProfileWindowsClosed() function, responable of inserting the newly created profile inside the slicer list and emitting the updateProfiles signal.

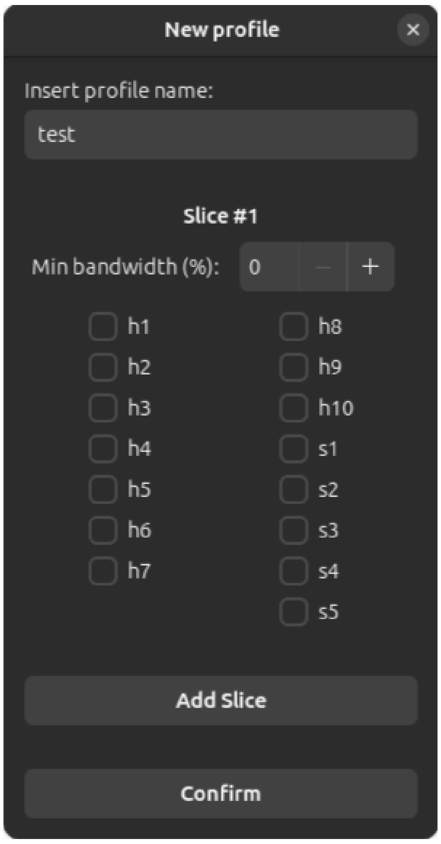


```
@0 - None

@1 - simpleSlice
 Slice #1:
   Reserved percentage: 100.0%
    h1 s1 h2 s2 h3 h4

@2 - PopulatedSlice
 Slice #1:
   Reserved percentage: 100.0%
    h1 s1 h2 h3 s2 h4 s3 h5 h6

@3 - doubleSlice
 Slice #1:
   Reserved percentage: 100.0%
    h1 s1 s3 h5
 Slice #2:
   Reserved percentage: 100.0%
    h3 s2 s4 h7

@3 - sharedSwitch
 Slice #1:
   Reserved percentage: 100.0%
    h1 s1 s3 h6
 Slice #2:
   Reserved percentage: 100.0%
    h2 s1 s2 h3

@3 - sharedLink
 Slice #1:
   Reserved percentage: 70.0%
    h1 s1 s3 h6
 Slice #2:
   Reserved percentage: 30.0%
    h2 s1 s3 h5
```

Figure 10 - Profiles view

4.2 New Profile window

The window presents itself only with the textfield used to capture the profile name, plus two buttons for adding slices and saving the profile. In case the profile is saved without any slices inside, all the changes are ignored and no profile is created in the main application.
When clicking the "*add slice*" button, a small widget is added to the window using the **NewSliceBox** class. This class builds a SpinButton used for keeping track of the minimum bandwidth reserved for the slice, plus a selection of check buttons listing the different devices who can be contained inside the slice Each time a new slice is created, the program removes previously selected hosts and reduces the maximum value for reserved bandwidth. This behavior blocks the possibility of having the same host on multiple slices, or slices who compete for the same bandwidth. If the maximum amount of reserved bandwidth is reached, or all hosts have been selected, the "add slice" button doesn't perform any action.
Upon confirming the customizations using the apropriate button, the window extracts the data from the form, creates a new profile storing it internally, and emits the *newProfileWindowClosed* signal, warning the main window that the new profile is ready to be extracted and the new profile window to be subsequently destroyed.


New profile

Insert profile name:

test

Slice #1

Min bandwidth (%): 0

Figure 11 - Profiles view

5 User manual

To launch the code, it's necessary to open three terminals: the first two are used to connect to Comnetsemu via Vagrant, while the third is opened directly from the host machine and navigates to the project directory. Subsequently, in the first terminal (Vagrant), execute the command ***sudo python3 network.py*** inside the project directory to initiate the network using Mininet. In the second terminal (also Vagrant), run the command ***ryu-manager controller.py*** to start the controller developed using Ryu. Finally, in the third terminal, execute the command ***python3 slicer_UI.py*** to launch the graphical user interface for managing slice profiles and QoS. For further details on the interface functionalities, refer to Chapter 4.