Department of Information Engineering and Computer Science

Softwarized and Virtualized Mobile Networks

$\sim \cdot \sim$

Academic Year 2021–2022

Morphing Slices Documentation

Professor
Granelli Fabrizio

Student
Vinci Nicolò
220229

July 16, 2022

Contents

Introduction

The document provides a detailed description of the Softwarized and Virtualized Mobile Networks project called Morphing Slices. The document is organized as followed:

1. Introduction: it defines a general overview of the theory, the project purpose, the document organization and why the document has been written.
2. Getting Started: it describes how to start the project and which operative system are supported by the project and provides the necessary steps for running the project in Windows.
3. Development Workflow: it reports the workflow followed during the development phase of the project.
4. New Topology Building System: it outlines in details the new innovative way to deploy networks in Comnetsemu.
5. Tested Scenarios: it describes the three tested scenarios and their corresponding results.
6. Known Issues: it reports the issues encountered during the developing.
7. Conclusions: it sums up the entire document focusing on the obtained results and gives a consideration on the new way to deploy networks.

The project is developed on top of Comnetsemu [3] which is a virtual testbed and network emulator for Network Function Virtualization and Software Defined Network. In turn, Comnetsemu extends the network emulator Mininet [12] and allows to deploy Docker container [6] inside its hosts. Docker allows a lightweight virtualization starting container in an isolated environment. A similar approach is taken by Containernet [4] which is a Mininet fork. However, Comnetsemu is focused more on adding essential features to Mininet for a better emulation. To read more about the comparison between Comnetsemu and Mininet: `https://git.comnets.net/public-repo/comnetsemu/-/blob/master/doc/comparison.md`.
`https://git.comnets.net/public-repo/comnetsemu/-/blob/master/doc/faq.md`.
So, the project is strictly related to two paradigms:

- Software Defined Network (SDN): it defines the separation between the Data plane and Control plane. In the networking world, the Data plane involves any activity that results in sending a packet from the end user. Instead, the Control plane handles the Data plane activities without involving end user packets. For example, the Control plane is in charge to make the routing table and the Data plane has to forward the packets following it.
- Network Function Virtualization (NFV): it defines how to virtualize node network functions into building blocks that may be connected, or chained, together to create communication services.

The Network Slicing concept is also implemented in the project and it uses the SDN and NFV paradigms. Network slicing means splitting the network in different slices providing logical end-to-end connection. It allows to deploy multiple logical, self-contained networks on top a common hardware. It supports flexible on-demand networks and isolation among them. Within a slice, the forwarding rules can be defined with flows. Then, they may be manipulated dynamically to redirect the traffic. The OpenFlow protocol [13] was created to deploy the Network Slicing concept. So, the figure 1 summarizes the theoretical idea behind the project.

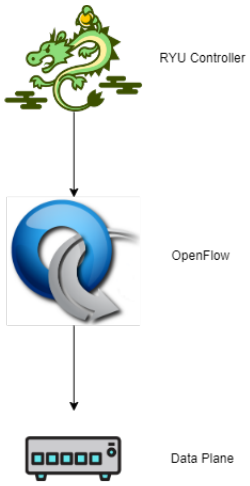


Figure 1: Introduction summary

The goal of the project is to enable the RYU SDN controller[18] to build network slices and dynamically modify the netowkr traffic. The SDN controller will not only slice, but reprogram connectivity within the slice. Considering to have the same service deployed in two different servers, a service migration can be performed manipulating and reprogramming the flows within the slice. So, the project aims at demonstrating a dynamic, stateful and transparent service migration:

- Dynamic: migration is performed after a user input.
- Stateful: service state is kept after the migration.
- Transparent: client is not aware of the migration.

It has been decided to write the document for giving a detailed documentation about the project. By the way, reading the document it can be useful for the future Comnetsemu project developers. A very smooth development workflow and new convenient method of developing networks have been described. So, future developers may be inspired by the document during their projects.

Chapter 1

Getting Started

The project can be run on Linux-based and Windows operative system. However, some extra steps need to be taken on Windows. They are described in the section 1.1. On Windows, it is recommended to first reading the section 1.1 and then follow the steps below. On a Linux-based, the below steps can be performed immediately. Firstly, it is necessary to clone Github repository:

```
$ git clone --recursive https://github.com/nico989/SVMN.git
$ cd SVMN
```

The option *recursive* is to clone recursively any git submodule in a repository. The project imports as a git submodule the original Comnetsemu Github repository from:
`https://github.com/stevelorenz/comnetsemu`.
Then, the folder *scripts* needs the execution permissions:

```
$ chmod -R +x scripts
```

At the end, the initialization script must be run:

```
$ scripts/init.sh
```

At the first time the *init.sh* is executed, the local project folder might not be copied in the Comnetsemu virtual machine. So, the error showed at figure may be prompted.

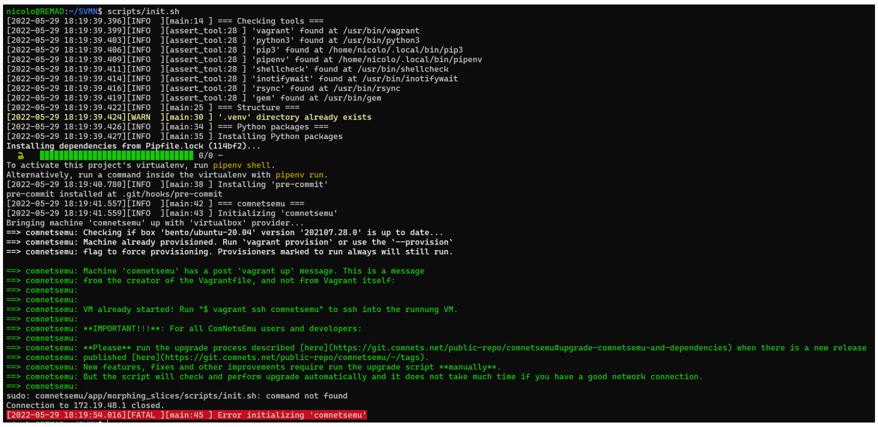


Figure 1.1: First *init.sh* run

To copy the local project folder in Comnetsemu the first time, the *dev.sh* script must be run before the *init.sh* script:

```
$ scripts/dev.sh
```

Now, the *init.sh* script can finish correctly. Read the chapter 2 for more details about the *dev.sh* script. The *init.sh* script performs different preliminary operations:

1. It checks if the required packages are installed.

```
# Assert tool(s)
INFO "=== Checking tools ==="
assert_tool vagrant
assert_tool python3
assert_tool pip3
assert_tool pipenv
assert_tool shellcheck
assert_tool inotifywait
assert_tool rsync
assert_tool gem
```

Figure 1.2: Packages required

2. Then, it creates the Python virtual environment folder and install the Python packages inside it. It also installs the pre-commit package, check the chapter 2 for more information.

```
# Create .venv directory
INFO "=== Structure ==="
if [ ! -d "$VENV_DIR" ]; then
    INFO "Creating '.venv' directory"
    mkdir "${VENV_DIR}"
else
    WARN "'.venv' directory already exists"
fi

# Install Python packages
INFO "=== Python packages ==="
INFO "Installing Python packages"
(pipenv install --dev) || { FATAL "Error installing Python packages"; exit 1; }
# Install pre-commit
INFO "Installing 'pre-commit'"
(pipenv run pre-commit install) || { FATAL "Error installing 'pre-commit'"; exit 1; }
```

Figure 1.3: Set Python virtual environment

3. At the end, it starts the Comnetsemu virtual machine and run the initialization script for it.

```
# Create and configure comnetsemu
INFO "=== comnetsemu ==="
INFO "Initializing 'comnetsemu'"
(cd "${__DIRNAME}/../comnetsemu" && vagrant up) || { FATAL "Error initializing 'comnetsemu'"; exit 1; }
(cd "${__DIRNAME}/../comnetsemu" && vagrant ssh -- -t 'sudo comnetsemu/app/morphing_slices/scripts/init.sh') || { FATAL "Error initializing 'comnetsemu'"; exit 1; }
```

Figure 1.4: Run and configure Comnetsemu

So, there is another installation script located at *src/scripts/init.sh*. The script performs four crucial steps for the Comnetsemu virtual machine:

1. It updates the Comnetsemu machine and installs the *parallel* package.

```
# Packages
INFO "Installing packages"
sudo apt-get update \
&& sudo apt-get install -y \
parallel
```

Figure 1.5: Update and install Comnetsemu packages

2. If they are not present yet, it builds up the Docker Images for the Mininet hosts.

```
# Check Docker image(s)
if [[ "$(docker images -q ${DEV_HOST_IMAGE} 2> /dev/null)" == "" ]]; then
    WARN "Docker image '${DEV_HOST_IMAGE}' does not exists"
    INFO "Building Docker image '${DEV_HOST_IMAGE}'"
    docker build --rm -t ${DEV_HOST_IMAGE} -f "$( readlink -m "${__DIRNAME}/../docker/Dockerfile.dev_host" )" .
fi
if [[ "$(docker images -q ${DEV_SERVER_IMAGE} 2> /dev/null)" == "" ]]; then
    WARN "Docker image '${DEV_SERVER_IMAGE}' does not exists"
    INFO "Building Docker image '${DEV_SERVER_IMAGE}'"
    docker build --rm -t ${DEV_SERVER_IMAGE} -f "$( readlink -m "${__DIRNAME}/../docker/Dockerfile.dev_server" )" .
fi
```

Figure 1.6: Build Docker images

3. It installs the Python packages from the requirements file generated by *scripts/dev.sh* and two more packages.

```
INFO "Installing Python dependencies from 'requirements.txt'"
sudo pip install -r "${__DIRNAME}/../requirements.txt"
sudo pip install eventlet==0.30.2 Flask==2.0.3
```

Figure 1.7: Install Python packages in Comnetsemu

4. It patches the RYU Python package. Check the chapter 5 for more details.

```
INFO "Checking Ryu"
# Ryu gui topology html directory
RYU_GUI_TOPOLOGY_HTML_DIR="$(dirname "$(python3 -c "import ryu; print(ryu.__file__)")")/app/gui_topology/html"
readonly RYU_GUI_TOPOLOGY_HTML_DIR
if [ ! -d "${RYU_GUI_TOPOLOGY_HTML_DIR}" ]; then
    WARN "Ryu not fixed, fixing..."
    # Html folder
    sudo mkdir "${RYU_GUI_TOPOLOGY_HTML_DIR}"
    sudo wget --quiet --show-progress -P "${RYU_GUI_TOPOLOGY_HTML_DIR}" https://raw.githubusercontent.com/faucetsdn/ryu/master/ryu/app/gui_topology/html/index.html
    sudo wget --quiet --show-progress -P "${RYU_GUI_TOPOLOGY_HTML_DIR}" https://raw.githubusercontent.com/faucetsdn/ryu/master/ryu/app/gui_topology/html/router.svg
    sudo wget --quiet --show-progress -P "${RYU_GUI_TOPOLOGY_HTML_DIR}" https://raw.githubusercontent.com/faucetsdn/ryu/master/ryu/app/gui_topology/html/ryu.topology.css
    sudo wget --quiet --show-progress -P "${RYU_GUI_TOPOLOGY_HTML_DIR}" https://raw.githubusercontent.com/faucetsdn/ryu/master/ryu/app/gui_topology/html/ryu.topology.js
    INFO "Ryu fixed"
else
    INFO "Ryu already fixed"
fi
```

Figure 1.8: Fix RYU Python package

1.1 Steps for Windows

First of all, the user must install Windows Subsystem for Linux (WSL) version 2 following the Microsoft official guide:
`https://docs.microsoft.com/it-it/windows/wsl/install`
Then, the next configurations must be done in the WSL and not in Windows directly:

1. Install Vagrant in WSL:
   `https://www.vagrantup.com/downloads`.
   It can be installed from the Ubuntu repositories as well, but the version may be older than the latest. So, it is recommended to download the zip file and install the deb package manually. Vagrant must be installed in Windows as well.

2. Configure Vagrant to run in WSL following the official guide:
   `https://www.vagrantup.com/docs/other/wsl`.
   Two environment variable must be set for the basics configuration:

   ```
   $ export VAGRANT_WSL_ENABLE_WINDOWS_ACCESS="1"
   $ export PATH="$PATH:/mnt/c/Program Files/Oracle/VirtualBox"
   ```

3. Install the Vagrant plugin for VirtualBox in the WSL:
   `https://github.com/Karandash8/virtualbox_WSL2`.
   The following command can be run to install the plugin:

```
$ vagrant plugin install virtualbox_WSL2
```

4. The line 565 of *platform.rb* file located at
   /opt/vagrant/embedded/gems/[VAGRANT_VERSION]/gems/vagrant-[VAGRANT_VERSION]/lib/vagrant/util/platform.rb
   must be replaced from:

   ```
   if info && (info[:type] == "drvfs" || info[:type] == "9p")
   ```

   To:

   ```
   if info && (info[:type] == "drvfs" || info[:type] == "9p"
   || info[:type] == "ext4")
   ```

   More information about the issue:
   `https://github.com/hashicorp/vagrant/issues/11623`

After completing all the steps, the repository can be cloned and scripts can be run as described above.

Chapter 2

Development Workflow

The chapter describes the workflow followed during the project development. The purpose is to suggest a smooth and comfortable way of working in the Comnetsemu environment. Then, it is up to the developers to follow the proposed approach or create a new one.

2.1 Git and GitHub

Git has been used for versioning the code and the repository is hosted on GitHub. Git allows to keep tracking and managing the code. GitHub is useful to host on the web the repository and integrate different workflows [9]. A GitHub workflow is a configurable automated process that run one or more jobs and it is defined by a YAML file. The project integrates two workflows useful for testing and deployment purpose:

- Continuous Integration (CI) [2]: it is defined by the *ci.yml* file located in the *.github/-workflows* folder. CI workflow automatically builds and tests the software every time a developer pushes changes to the main branch. So, every time the workflow performs the steps defined in the YAML file after a push action or pull request. The CI mainly tests the installation of packages for this project.

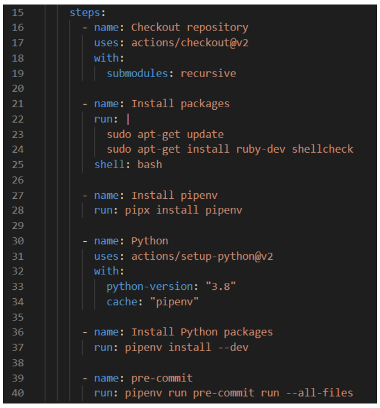


Figure 2.1: CI steps

The result can be checked on the GitHub user interface under the tab *Action* and the item *ci*.

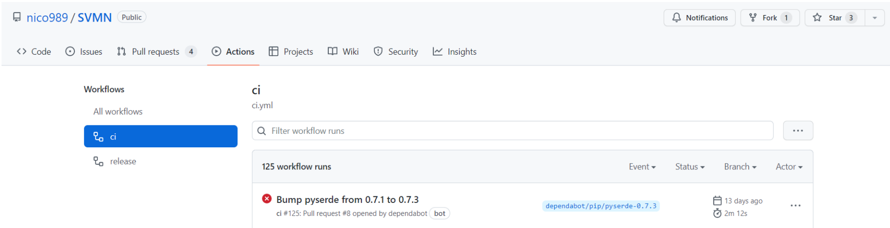


Figure 2.2: CI result

If there is a red cross like in the figure 2.2, it means that something went wrong. Otherwise, if all went good, there will be a green tick.

- Continuous Deployment (CD) [1]: it is defined by the *release.yml* file located in the *.github/-workflows* folder. It generates *morphing_slices.tar.gz* file for any new tagged version. So, the workflow is triggered every time someone performs a push action with the option *-v* for defining the version.

```
$ git tag v0.0.30
$ git push ——origin tags
```

After the pushing request, the steps reported in the figure 2.3 are taken.

```yaml
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
        with:
          submodules: recursive

      - name: Environment
        run: echo "RELEASE_VERSION=${GITHUB_REF#refs/*/}" >> $GITHUB_ENV

      - name: Install pipenv
        run: pipx install pipenv

      - name: Python
        uses: actions/setup-python@v2
        with:
          python-version: "3.8"
          cache: "pipenv"

      - name: Scripts permissions
        run: chmod -R +x scripts

      - name: Generate morphing_slices.tar.gz
        run: scripts/prod.sh

      - name: Release
        uses: softprops/action-gh-release@v1
        with:
          body: Morphing Slices ${{ env.RELEASE_VERSION }}
          files: |
            morphing_slices.tar.gz
```

Figure 2.3: CD steps

The most important is the one highlighted in red. It executes the script *prod.sh* located in the folder *scripts*. The script core function is to generate the *.tar.gz* file from the *src* folder.

```bash
# Generate final tar|gz file
INFO "Generating '${TAR_GZ_NAME}'"
find "${TMP_DIR}" -printf "%P\n" | tar -czf "${TAR_GZ_NAME}" --no-recursion -C "${TMP_DIR}" -T -
```

Figure 2.4: Folder generation

The result can be checked on the GitHub user interface under the tab *Action* and the item *release*.

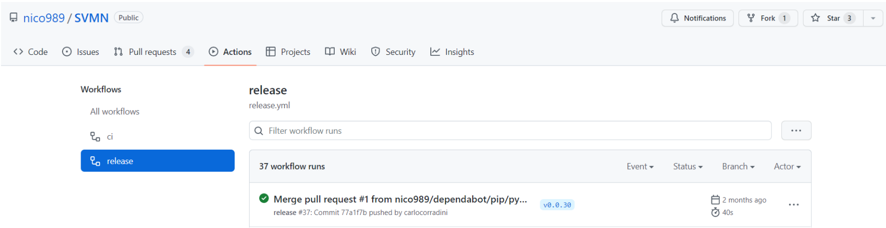


Figure 2.5: CD result

If there is a red cross like in the figure 2.2, it means that something went wrong. Otherwise, if the *.tar.gz* file has been created correctly, there will be a green tick as reported in figure 2.5. So, any project version can be easily downloaded from the GitHub repository.

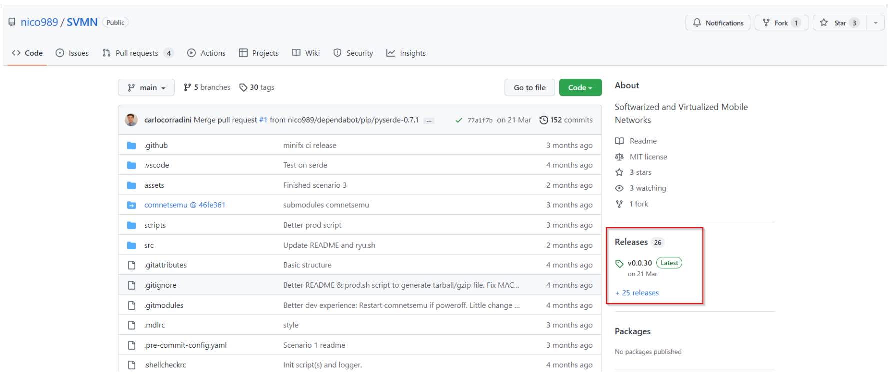


Figure 2.6: CD release

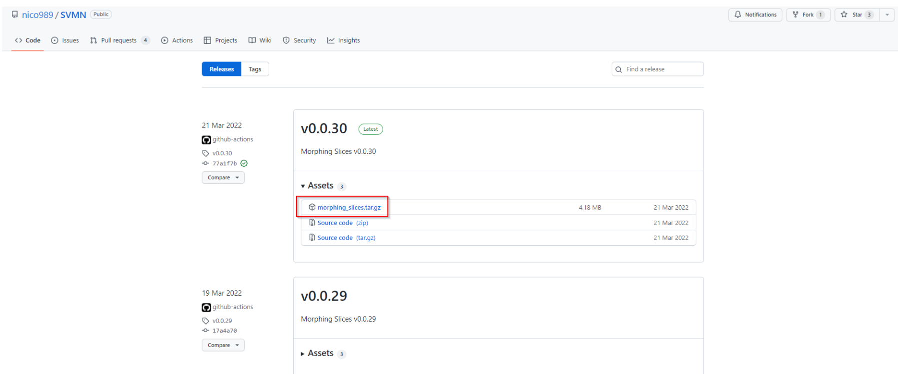


Figure 2.7: CD release download

Then, other two tools are used to maintain the code and the project itself:

- Pre-commit [15]: it is a framework to manage and maintain multi-language Pre-commit hooks. A Git hook script is useful to identify simple issues before submitting the code. Hooks are defined in the *pre-commit-config.yaml* file. They run automatically on every commits pointing out issues such as missing semicolons, trailing whitespace, and debug statements.

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.1.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-docstring-first
      - id: check-yaml
      - id: debug-statements
      - id: name-tests-test
      - id: requirements-txt-fixer
  - repo: https://github.com/jumanjihouse/pre-commit-hooks
    rev: 2.1.5
    hooks:
      - id: script-must-have-extension
      - id: markdownlint
      - id: shellcheck
  - repo: https://github.com/psf/black
    rev: 22.1.0
    hooks:
      - id: black
  - repo: https://github.com/Lucas-C/pre-commit-hooks-nodejs
    rev: v1.1.1
    hooks:
      - id: dockerfile_lint
```

Figure 2.8: Pre-commit YAML file

- Dependabot [5]: it is defined by the *dependabot.yml* file located in the *.github* folder. It is used to check the third-party libraries updates. It checks the *pip* and *docker* dependencies weekly. A pull request will be created, if an update is found.

```yaml
updates:
  # Enable version updates for pipenv
  - package-ecosystem: "pip"
    # Look for a `Pipfile` in the `root` directory
    directory: "/"
    # Check for updates once a week
    schedule:
      interval: "weekly"

  # Enable version updates for Docker
  - package-ecosystem: "docker"
    # Look for a `Dockerfile` in the `root` directory
    directory: "/"
    # Check for updates once a week
    schedule:
      interval: "weekly"
```

Figure 2.9: Dependabot YAML file

2.2 Comentsemu Development

In this section the development process in the Comnetsemu virtual machine is described. It might be annoying to copy and paste every time the local project in the Comnetsemu virtual machine. However, Comnetsemu has a shared folder called *comnetsemu* already set up. It means that any file in that folder can be used in the virtual machine as well. So, the simple idea is to set up a watcher which automatically copies and pastes any update from the *src* folder in the *comnetsemu* shared folder.

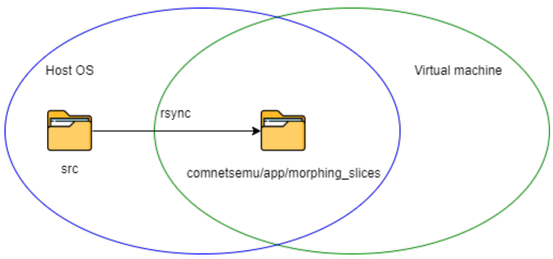


Figure 2.10: Development schema

The script *dev.sh* located in the folder *scripts* must be run to start the development process:

1. First, it checks from the Vagrant status if the Comnetsemu virtual machine has been started. If it is powered off, the script will restart Comnetsemu.

```
# Check Vagrant status
INFO "=== Checking Vagrant ==="
case "${VAGRANT_STATUS}" in
    running)
        INFO "'comnetsemu' is running"
    ;;
    poweroff)
        # Restarting comnetsemu
        INFO "'comnetsemu' is poweroff, restart..."
        INFO "Restarting 'comnetsemu'"
        (cd "${__DIRNAME}/../comnetsemu" && vagrant up) || { FATAL "Error restarting 'comnetsemu'"; exit 1; }
    ;;
    *)
        ERROR "Vagrant status '${VAGRANT_STATUS}' not checked"
        exit 1
    ;;
esac
```

Figure 2.11: Check Comnetsemu staus

2. It generates the *requirements.txt* file which contains the required packages to be installed.

```
# Generate requirements.txt
INFO "Generating 'requirements.txt' from 'pipenv'"
pipenv lock -r > "${__DIRNAME}/../src/requirements.txt"
```

Figure 2.12: Generate requirements

3. Then, it synchronizes the destination directory *morphing_slices* in the shared folder *comnetsemu/app* with the local directory *src*. The first time, the local directory is copied in the shared folder. The utility *rsync* [17] is used for synchronization purpose.

```
# Sync watch and destination directories
INFO "=== Sync ==="
function sync() {
    DEBUG "Syncing '$WATCH_DIR' with destination '$DEST_DIR'"
    rsync --archive --verbose --compress --delete --human-readable --quiet "$WATCH_DIR" "$DEST_DIR"
}

# Sync directory
INFO "Syncing directory '$WATCH_DIR'"
sync
```

Figure 2.13: Rsync synchronization

4. It initializes Comnetsemu running the *init.sh* script located in the *scripts* folder.

```
# Initialize comnetsemu
INFO "Initializing 'comnetsemu'"
(cd "${__DIRNAME}/../comnetsemu" && vagrant ssh -- -t 'cd comnetsemu/app/morphing_slices && sudo scripts/init.sh') || { FATAL "Error initializing 'comnetsemu'"; exit 1; }
```

Figure 2.14: Initialize Comnetsemu

5. At the end, the watcher starts to monitor the *src* local folder. The command *inotifywait* [10] is used to monitor any change in *src*. Every time a change occurs, only the change will be reflected in the destination folder by *rsync*.

```
# Watcher
INFO "Starting watcher on '$WATCH_DIR' with destination '$DEST_DIR'";
while inotifywait --recursive --quiet --event modify,create,delete,move "$WATCH_DIR"; do
    sync
done
INFO "Watcher stopped";
```

Figure 2.15: Start watcher

When the *dev.sh* script runs, the output looks like the one showed in the figure 2.16 at the end.

Figure 2.16: Run *dev.sh*

If a change occurs, it will be prompted in the output as shown in the figure 2.17.

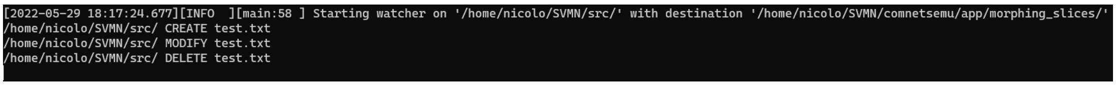


Figure 2.17: Change occurrence

Chapter 3

New Topology Building System

The project brings a new innovative way to deploy virtual networks in the Comnetsemu environment. Comnetsemu basically uses the Mininet python API to create the network topology and emulate the nodes. It is needed to write a long python file to emulate a big network topology and it is not easily readable for future developers or debugging purpose. So, the idea is to take an approach similar to Kubernetes [11] having a YAML or JSON file to describe the network topology to achieve a better readability and avoid python boiler plate code. Therefore, it is more easy to update the topology modifying the YAML/JSON file than a long python code. The idea of the new building system is to parse the YAML/JSON file and create the network using the Comnetsemu python API. So, there are three main python scripts:

- *src/topology.py.*
- *src/topology/topologyParser.py.*
- *src/topology/topologyBuilder.py.*

Hence, a new layer has been added to improve the usability in deploying network topology in the Comnetsemu environment. Future developers will need to write a single YAML/JSON file for the network topology. A practical application of the new topology building system can be seen in the chapter 4,

3.1 src/topology.py

This is entry point for building the topology. Firstly, a parser has been defined in order to get the YAML/JSON file as input.

```python
if __name__ == "__main__":
    # Arguments
    parser = argparse.ArgumentParser(
        description="Topology builder",
        formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=40),
    )
    parser.add_argument(
        "--file", help="Topology file", required=True, action="store", type=str
    )
    args = parser.parse_args()
```

Figure 3.1: Main *topology.py*

The input file is parsed in the *getTopology* function to get a topology object in order to feed the *buildTopology* function to emulate the network. The latter is described in the section 3.3.

```python
    # Topology
    logger.info(f"Analyzing topology file '{args.file}'")
    topology = getTopology(args.file)
    (network, manager) = topologyBuilder.buildTopology(topology)
```

Figure 3.2: Get and build topology

At the end, the Containernet object returned by *buildTopology* and Mininet command line interface are started. The *cleanTopology* function is called when the user closes the CLI.

```python
    # Network
    network.start()
    cli.CLI(network)
    topologyBuilder.cleanTopology(topology, network, manager)

```

Figure 3.3: Start network and Mininet CLI

Regarding the *getTopology* function, it takes in input the YAML/JSON file, it analyzes it with the *fileFormat* function and calls the *parseTopology* to deserialize the file in the topology object. More details of the *parseTopology* are reported in the section 3.2.

```python
def getTopology(file: str) -> topologyBuilder.Topology:
    topology = None
    format = fileFormat(file)

    with open(file) as f:
        topology = f.read()

    return topologyParser.parseTopology(topology, format)
```

Figure 3.4: *getTopology* function

To conclude, the *fileFormat* function analyzes the file type provided as input by the user. The accepted extensions are *.yaml*, *.yml* and *.json*, because the *parseTopology* is able to deserialize only them. If a user inputs another file type, an exception is returned.

```python
def fileFormat(file: str) -> topologyParser.Format:
    _, file_extension = os.path.splitext(file)
    if file_extension == ".yaml" or file_extension == ".yml":
        return topologyParser.Format.YAML
    elif file_extension == "json":
        return topologyParser.Format.JSON
    else:
        raise ValueError(f"Unknown file extension {file_extension}")
```

Figure 3.5: *fileFormat* function

3.2 src/topology/topologyParser.py

The script aims to deserialize the input file into the custom topology object. It relies on the *pyserde* [16] python library which allows to deserialize YAML/JSON file into python object and viceversa. The library provides two functions called *from_yaml* and *from_json* to do the deserialization and get the topology object.

```python
class Format(enum.Enum):
    YAML = enum.auto()
    JSON = enum.auto()


def parseTopology(topology: str, format: Format) -> Topology:
    if format is Format.YAML:
        return from_yaml(Topology, topology)
    elif format is Format.JSON:
        return from_json(Topology, topology)
    else:
        raise ValueError(f"Unknown format {format}")
```

Figure 3.6: *parseTopology* function

Above the function, different classes have been defined and tagged with the *@serde* decorator. They represent the attributes which can be defined in the network topology file. Then, each class has its own field attributes which can be optional or not. The topology object has been declared and it uses all classes defined in the script. It represents the entire network topology input file.

```python
@serde(tagging=InternalTagging("type"))
class Topology:
    network: Network
    controllers: List[Union[ControllerLocal, ControllerRemote]]
    switches: List[Switch]
    hosts: List[Host]
```

Figure 3.7: Topology object

So, it means that in the input file there are:

- A Network attribute: the *autoArp* and *autoMac* flag for the network can be set.

```python
@serde
class Network:
    autoMac: bool = field(default=False)
    autoArp: bool = field(default=False)
```

Figure 3.8: Network object

- A list of Controllers attribute: more than one controller can be defined for the same Mininet network. A controller can be local or remote. The local one has only the name attribute. Instead, the remote has name, IP address and port attributes.

```python
@serde
class ControllerLocal:
    name: str


@serde
class ControllerRemote:
    name: str
    ip: ipaddress.IPv4Address
    port: int
```

Figure 3.9: Controller objects

- A list of Switches attribute: more than one switch can be emulated. A name and list of network links can be defined for each switch.

```python
@serde
class Switch:
    name: str
    links: List[NetworkLink] = field(default_factory=list)
```

Figure 3.10: Switch object

A network link is defined by some optional parameters like bandwidth, delay, fromInterface, toInterface. The only mandatory field is the node to which the link is attached.

```python
@serde
class NetworkLink:
    node: str
    bandwidth: Optional[int]
    delay: Optional[str]
    fromInterface: Optional[str]
    toInterface: Optional[str]
```

Figure 3.11: Network link object

- A list of Hosts: an host represents an emulated node in the network. It must have a name and IP address. Optionally, a MAC address and Docker image might be defined for it. In turn, the host can deploy a list of Docker containers inside itself. A host might also have a list of network interfaces. Comnetsemu assigns one network interface by default, but the user may want to add more.

```python
@serde
class Host:
    name: str
    ip: ipaddress.IPv4Interface
    mac: Optional[str]
    image: Optional[str]
    containers: List[Container] = field(default_factory=list)
    interfaces: List[NetworkInterface] = field(default_factory=list)
```

Figure 3.12: Host object

A container object must be defined by a name and Docker image. It may have a command to be executed when the container starts.

```python
@serde
class Container:
    name: str
    image: str
    cmd: Optional[str]
    wait: Optional[bool]
```

Figure 3.13: Container object

A network interface object must have a name and IP address. Furthermore, a MAC address might be defined.

```python
@serde
class NetworkInterface:
    name: str
    ip: ipaddress.IPv4Interface
    mac: Optional[str]
```

Figure 3.14: Network interface object

3.3 src/topology/topologyBuilder.py

The *buildTopology* function is defined in the script. It takes the topology object and builds every component of it. Firstly, it initializes the containernet object. Then, it builds controllers, switches, hosts, network links and network interfaces. The virtual network function manager object is initialized. In the Comnetsemu environment, the VNFManager has the capability to start or stop any host. At the end, Docker containers inside hosts are built up.

```python
def buildTopology(topology: Topology) -> Tuple[Containernet, VNFManager]:
    logger.info("=== NETWORK ===")
    logger.info(f"Network: {to_json(topology.network)}")
    network = Containernet(
        switch=node.OVSKernelSwitch,
        autoSetMacs=topology.network.autoMac,
        autoStaticArp=topology.network.autoArp,
        link=mnlink.TCLink,
    )

    logger.info("=== CONTROLLERS ===")
    buildControllers(network, topology)
    logger.info("=== SWITCHES ===")
    buildSwitches(network, topology)
    logger.info("=== HOSTS ===")
    hostInstances = buildHosts(network, topology)

    logger.info("=== NETWORK LINKS ===")
    buildNetworkLinks(network, topology)
    logger.info("=== NETWORK INTERFACES ===")
    buildNetworkInterfaces(hostInstances)

    # Manager
    manager = VNFManager(network)

    logger.info("=== CONTAINERS ===")
    buildContainers(manager, topology)

    return (network, manager)
```

Figure 3.15: *buildTopology* function

So, the functions to build the entire network are:

- *buildControllers*: it builds all the controllers. For each controller, it checks if it is remote or local. Then, it uses the Mininet function *addController* to add it to the network.

```python
def controllerType(controller: Union[ControllerLocal, ControllerRemote]):
    if isinstance(controller, ControllerLocal):
        return node.Controller
    elif isinstance(controller, ControllerRemote):
        return node.RemoteController
    else:
        raise Exception(f"Unknown class instance {type(controller)}")


def buildControllers(network: Containernet, topology: Topology) -> None:
    for controller in topology.controllers:
        logger.info(f"Controller {controller.name}: {to_json(controller)}")
        params = {}

        if isinstance(controller, ControllerRemote):
            params["ip"] = str(controller.ip)
            params["port"] = controller.port

        network.addController(
            controller.name, controller=controllerType(controller), **params
        )
```

Figure 3.16: *buildControllers* function

- *buildSwitches*: it builds all the switches. For each switch, it is assigned the datapath id and OpenFlow protocol version 1.0. Then, it uses the Mininet function *addSwitch* to add it to the network.

```python
def buildSwitches(network: Containernet, topology: Topology) -> None:
    for index, switch in enumerate(topology.switches):
        logger.info(f"Switch {switch.name}: {to_json(switch)}")
        params = {"protocols": "OpenFlow10", "dpid": "%016x" % (index + 1)}

        network.addSwitch(switch.name, **params)
```

Figure 3.17: *buildSwitches* function

- *buildHosts*: it builds all the hosts defining IP address, Docker image and Docker arguments. Then, it uses the Mininet function *addDockerHost* to add it to the network.

```python
def buildHosts(network: Containernet, topology: Topology) -> Dict[DockerHost, Host]:
    hostInstances: Dict[DockerHost, Host] = {}

    for host in topology.hosts:
        logger.info(f"Host {host.name}: {to_json(host)}")
        docker_args = {"hostname": host.name, "pid_mode": "host"}
        params = {
            "ip": host.ip.with_prefixlen,
            "dimage": host.image if host.image else "dev_host:latest",
            "inNamespace": True,
            "docker_args": docker_args,
        }

        if host.mac:
            params["mac"] = host.mac

        instance = network.addDockerHost(host.name, **params)

        # Add instance
        hostInstances[instance] = host

    return hostInstances
```

Figure 3.18: *buildHosts* function

- *buildNetworkLinks*: it builds all the network links. For each switch, the links between it and the nodes are built up. Then, it uses the Mininet function *addLink* to add it to the network.

```python
def buildNetworkLinks(network: Containernet, topology: Topology) -> None:
    for switch in topology.switches:
        logger.info(f"Switch {switch.name}:")

        for link in switch.links:
            logger.info(f" {to_json(link)}")
            params = {}

            if link.bandwidth:
                params["bw"] = link.bandwidth
            if link.delay:
                params["delay"] = link.delay
            if link.fromInterface and link.toInterface:
                params["intfName1"] = link.fromInterface
                params["intfName2"] = link.toInterface

            network.addLink(switch.name, link.node, **params)
```

Figure 3.19: *buildNetworkLinks* function

- *buildNetworkInterfaces*: it builds any extra network interface for any host. It uses the *ip addr* Linux command to assign the IP address to the interface. Moreover, it uses the *macchanger* commmand to assign the MAC address to the interface.

```python
def buildNetworkInterfaces(hostInstances: Dict[DockerHost, Host]) -> None:
    for instance, host in hostInstances.items():
        logger.info(f"Host {host.name}:")

        for interface in host.interfaces:
            logger.info(f" {to_json(interface)}")

            instance.cmd(f"ip addr add {interface.ip} dev {interface.name}")
            if interface.mac:
                instance.cmd(f"macchanger -m {interface.mac} {interface.name}")
```

Figure 3.20: *buildNetworkInterfaces* function

- *buildContainers*: it builds Docker containers inside any host. It allows to do the Docker in Docker virtualization. For each host, all the declared containers are created using the VFNManager.

```python
def buildContainers(manager: VNFManager, topology: Topology) -> None:
    for host in topology.hosts:
        logger.info(f"Host {host.name}:")

        for container in host.containers:
            logger.info(f" {to_json(container)}")

            manager.addContainer(
                name=container.name,
                dhost=host.name,
                dimage=container.image,
                dcmd=container.cmd if container.cmd else "",
                wait=container.wait if container.wait else False,
            )
```

Figure 3.21: *buildContainers* function

At the end of the script, the *cleanTopology* function has been defined. For each host, it removes the inside Docker containers. Then, it stops the network and the VNFManager.

```python
def cleanTopology(
    topology: Topology, network: Containernet, manager: VNFManager
) -> None:
    logger.info("=== CLEANING ===")
    # Container
    for host in topology.hosts:
        for container in host.containers:
            logger.info(f"Container: {container.name}")
            manager.removeContainer(container.name, True)
    # Stop network
    network.stop()
    manager.stop()
```

Figure 3.22: *cleanTopology* function

3.4 Build a network topology

The command to build the network topology defined in a YAML/JSON files is:

```
$ sudo python3 topology.py --file scenarios/1/topology.yaml
```

The output should be similar to the figure 3.23.

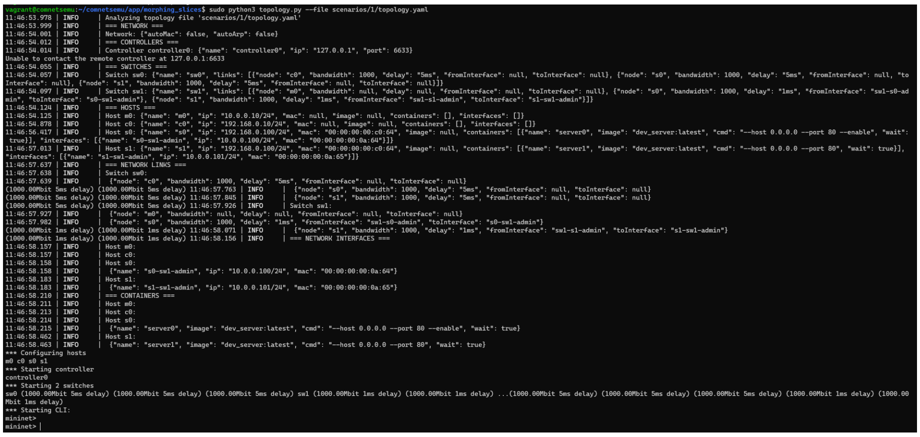


Figure 3.23: Building topology output

Chapter 4

Tested Scenarios

Some possible tested scenarios and their results are reported in this chapter. First of all, the deployed service and how the migration works are described to understand better why the migration is dynamic, stateful and transparent. After that, each scenario has been defined by:

- topology.yaml: YAML file which implements the network topology with the new topology building system demonstrated in the chapter 3.
- OpenFlow: a software component which implements the OpenFlow protocol. It is needed to define slices and flows. FlowVisor [8] and ovs-ofctl [14] have been used during testing.
- RYU Controller: a python script which implements a RYU controller. It is used for handling the traffic within slices and flows. The controller might also add, update, modify or delete flows redirecting dynamically the network traffic.

Any host in Mininet network is deployed as a Docker container. So, three custom Docker images located at *src/docker* were developed:

- Dockerfile.dev_host: it is the default Mininet Docker image with network tools.
- Dockerfile.dev_server: it deploys the developed service to be migrated.
- Dockerfile.flowvisor: it installs and deploys FlowVisor for slicing the network.

4.1 Tested Service

The service is a python Flask [7] web application located at *src/server.py* and deployed in the network through the Dockerfile.dev_server. The application increments a counter every time a user performs an HTTP post request to the API */api/counter*.

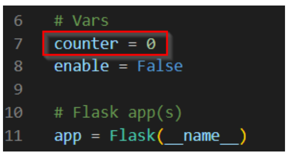


Figure 4.1: Flask counter

The web application exposes four APIs:

- */api/counter* [GET]: it returns the current counter value.

```python
# APP
@app.route("/api/counter", methods=["GET"])
def get_counter():
    """
    Return counter.
    """

    global counter
    return jsonify(counter=counter)
```

Figure 4.2: Flask */api/counter* [GET]

- */api/counter* [POST]: it increments and returns the counter value.

```python
@app.route("/api/counter", methods=["POST"])
def post_counter():
    """
    Increment and return counter.
    """

    global counter
    counter += 1
    return jsonify(counter=counter)
```

Figure 4.3: Flask */api/counter* [POST]

- */api/admin/disable* [POST]: it logically disables the service setting to *False* the global variable *enable* and returns the current counter value.

```python
@app.route("/api/admin/disable", methods=["POST"])
def disable():
    """
    Disable service.
    """

    global enable
    enable = False
    return jsonify(counter=counter)
```

Figure 4.4: Flask */api/admin/disable* [POST]

- */api/admin/migrate* [POST]: it performs the migration detailed in the section 4.2. It retrieves the IP address of the current enabled server from the body parameter. Then, it disables it and gets the current counter value calling the API */api/admin/disable*. At the end, it stores the current counter value and set the global variable *enable* to *True* to become the new enabled server.

```python
# APPMGR
@app.route("/api/admin/migrate", methods=["POST"])
def migrate():
    """
    Migrate service and return counter.
    """

    global counter, enable

    # Request body
    body = request.get_json(force=True)
    # Server
    server = body["server"]
    # Disable other server & obtain counter
    response = requests.post(f"{server}/api/admin/disable").json()
    # Set counter value
    counter = response["counter"]
    # Enable this server
    enable = True

    return jsonify(counter=counter)
```

Figure 4.5: Flask */api/admin/migrate* [POST]

4.2 How Migration Works

Firstly, there are at least two running Flask web services in every scenario and only one is enabled to receive HTTP requests. Morevoer, if a user performs a GET or POST request on the endpoint */api/counter*, it gets a 503 HTTP status code, because it is intercepted and blocked by the middleware reported in figure 4.6.

```python
@app.before_request
def before_request_func():
    global enable

    # Check URL admin request
    admin = re.search(r"\badmin\b", request.url)

    # Return 503 if disabled
    if not enable and not admin:
        return "Service Unavailable", 503
```

Figure 4.6: Flask middleware

Ideally, the migration should be performed by a network administrator and it works as following:

1. The administrator performs an HTTP post request at */api/admin/migrate* to the disabled server. He has to provide the IP address of the enabled server in the body following the JSON format.

2. The disabled server retrieves the IP address and performs an HTTP post request at *[Enabled Server IP Address]/api/admin/disable*.

3. The enabled server sets its own global variable *enable* to false and returns the counter current value in the response.

4. The disabled server stores the counter value and sets to true its own global variable *enable*.

4.3 Scenario 1

The Scenario 1 is located in the folder *src/scenarios/1* and the network topology is reported in figure 4.7.

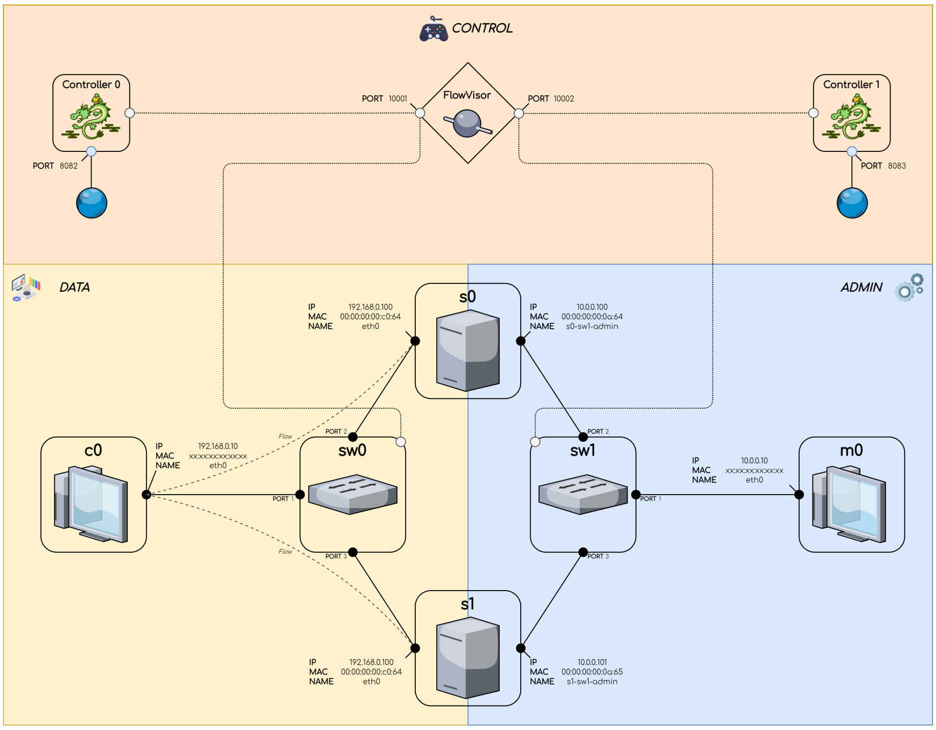


Figure 4.7: Network topology

There are several devices in the network:

- Client *c0*: it performs the HTTP requests to the enabled server.
- Servers *s0* and *s1*: they implement the Flask web application. By default, *s0* is the enabled server and *s1* is disabled. After the migration, *s1* will become enabled getting the current counter value from *s0*.
- Switch *sw0*: it connects *c0* with the two servers *s0* and *s1*.
- Manager *m0*: it emulates the network administrator and executes the command for the migration.
- Switch *sw1*: it connects *m0* with the two servers *s0* and *s1*.
- Controller 0: it handles the network traffic of the *data* slice telling to the switch *sw0* how to forward the traffic.
- Controller 1: it handles the network traffic of the *admin* slice telling to the switch *sw1* how to forward the traffic.

So, the OpenFlow protocol defines slices and flows and it is implemented by FlowVisor in the scenario. The *data* slice were created to emulate the traffic generated by a standard user. Instead, the *admin* slice emulates the administrator traffic to perform the service migration. Slices and flows are defined by the script *src/scenarios/1/flowvisor.sh*.

```
# FlowVisor admin slice & flow
INFO "Creating FlowVisor admin slice"
fvctl_exec add-slice --password=password slice_service_migration_admin tcp:localhost:10002 admin@slice_service_migration_admin
INFO "Creating FlowVisor admin flow"
fvctl_exec add-flowspace admin 2 1 any slice_service_migration_admin=7

# FlowVisor data slice
INFO "Creating FlowVisor data slice"
fvctl_exec add-slice --password=password slice_service_migration tcp:localhost:10001 admin@slice_service_migration

# Flowvisor data flow
INFO "Creating FlowVisor data flow"
fvctl_exec add-flowspace dpid1-c0 1 1 in_port=1 slice_service_migration=7
fvctl_exec add-flowspace dpid1-s 1 1 in_port="${SERVERS_PORT[IDX_SERVER]}" slice_service_migration=7
```

Figure 4.8: Slices and flows

The approach for the migration is to adjust dynamically the flow between the switch *sw0* and servers in order to always connect the client to the enabled server. As reported in figure 4.8, one flow was created for the *admin* slice and two for the *data* slice. So, in the *data* slice there is one flow to connect the client *c0* to the switch *sw0* and the other one is to connect the switch *sw0* to the enabled server. The latter is a dynamic flow, because the OpenFlow attribute *in_port* is automatically changed towards the enabled server for performing the service migration. Hence, the migration is straightforward in theory, but the implementation is a bit tricky. Firstly, FlowVisor needs a container itself for running and the script *src/scripts/flowvisor.sh* deploys a Docker container for FlowVisor. Secondly, the manager *m0* needs to send the command to the disabled server for starting the migration. So, the script *src/scripts/flowvisor.py* is a Flask web server which accepts HTTP post request at */api/migrate* and sends the command to *m0* using the Docker command line.

```python
@app.route("/api/migrate", methods=["POST"])
def migrate():
    """
    Migrate service via Docker.
    """

    # Request body
    body = request.get_json(force=True)
    # From server
    from_server = body["from"]
    # To server
    to_server = body["to"]

    # Call Docker
    subprocess.call(
        f'docker exec m0 curl -X POST -H "Content-Type:application/json" -d \'{{ "server": "http://{from_server}" }}\' {to_server}/api/admin/migrate',
        shell=True,
    )

    return ("", 200)
```

Figure 4.9: Flask *flowvisor.py*

The web server is run by the *src/scripts/flowvisor.sh* before the FlowVisor container deployment.

```
# Start server
INFO "Starting Docker server..."
python3 "${__DIRNAME}/flowvisor.py" --port $PORT > /dev/null 2>&1 &
SERVER_PID="$(jobs -p)"
readonly SERVER_PID
INFO "Docker server started with PID $SERVER_PID"
```

Figure 4.10: Run Flask *flowvisor.py*

Thirdly, the Controller 0 has to keep tracking of the new flow. The Controller 0 is a simple self-learning controller and for each datapath id, the controller maps the MAC address to the attribute *in_port*. It is able to automatically learn the mapping between MAC address and the OpenFlow attribute *in_port* using the OpenFlow Flood.

```python
# Learn mac address
self.mac_to_port.setdefault(dpid, {})
self.mac_to_port[dpid][src] = in_port

# Find out_port
if dst in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][dst]
else:
    out_port = ofproto.OFPP_FLOOD

actions = [parser.OFPActionOutput(out_port)]
```

Figure 4.11: Self-learning controller

As soon as the controller receives packet, it tries to lookup the *in_port* of the destination MAC address specified in the packet. If it is able to find the *in_port*, it sets the OpenFlow actions attribute *out_port* to it. Otherwise, it starts an OpenFlow Flood to discover the unknown *in_port* flooding the network. The controller self-learning code is reported in figure 4.11. When the flow changes, the mapping stored by the controller has to be updated as well. The script *src/scenarios/1/migrate.py* is a Flask web server run by the controller in a new python thread to handle incoming HTTP request for updating the mapping. As soon as the controller is launched, it spawns a new thread for the web server as reported in figure 4.12.

```python
class Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Controller, self).__init__(*args, **kwargs)
        CONF = cfg.CONF
        CONF.register_opts(
            [
                cfg.IntOpt("port", default=None),
            ]
        )

        self.mac_to_port = {}

        if CONF.port:
            migrator_port = int(CONF.port)
            self.thread_migration = threading.Thread(
                target=self.thread_migration_cb,
                daemon=True,
                kwargs={"port": migrator_port},
            )
            self.thread_migration.start()

    def thread_migration_cb(self, port: int):
        migrator.start(port, self.migration_cb)

    def migration_cb(self, dpid: int, mac: str, port: int):
        self.mac_to_port[dpid][mac] = port
```

Figure 4.12: Spawn *migrator.py*

Hence, the *migrator.py* accepts HTTP post request at */api/migrate* and call the method *migration_cb* to update the mapping. The datapath id, MAC address and the new *in_port* has to be passed in the body of the request.

```python
@app.route("/api/migrate", methods=["POST"])
def migrate():
    """
    Migrate service.
    """

    # Request body
    body = request.get_json(force=True)
    # Switch dpid
    dpid = int(body["dpid"])
    # Server mac
    mac = body["mac"]
    # Server port
    port = int(body["port"])

    # Notify callback
    if cb:
        cb(dpid, mac, port)

    return ("", 200)


def start(port: int, callback):
    global cb
    cb = callback
    app.run(port=port)
```

Figure 4.13: *migrator.py* API

To sum up, there is a while loop for migration in the script *src/scenarios/1/flowvisor.sh*. If the user press Enter:

- An HTTP post request has been performed to the *src/script/flowvisor.py* Flask web server to launch the migration command from the manager *m0*.
- An HTTP post request has been performed to the *src/scenarios/1/migrator.py* Flask web server to update the controller mapping.

```bash
# Migration loop
while read -n1 -r -p "Press 'Enter' to migrate or 'q' to exit" && [[ $REPLY != q ]]; do
    # Old ip
    OLD_IP=${SERVERS_IP[IDX_SERVER]}
    # Old port
    OLD_PORT=${SERVERS_PORT[IDX_SERVER]}
    # Update idx server
    IDX_SERVER="$(next_idx_server "$IDX_SERVER")"
    # New ip
    NEW_IP=${SERVERS_IP[IDX_SERVER]}
    # New port
    NEW_PORT=${SERVERS_PORT[IDX_SERVER]}

    INFO "Migrating from { ip: $OLD_IP, port: $OLD_PORT } to { ip: $NEW_IP, port: $NEW_PORT }"

    # Docker manager
    curl -X POST -H \"Content-Type:application/json\" -d "{ \"from\": \"$OLD_IP\", \"to\": \"$NEW_IP\" }" localhost:12345/api/migrate

    # Controller flow
    curl -X POST -H \"Content-Type:application/json\" -d "{ \"dpid\": \"1\", \"mac\": \"$SERVER_MAC\", \"port\": \"$NEW_PORT\" }" localhost:9876/api/migrate

    # FlowVisor
    fvctl_exec update-flowspace --match=in_port="$NEW_PORT" dpid1-s
    fvctl_stop
    fvctl_start

    INFO "Successfully migrated from { ip: $OLD_IP, port: $OLD_PORT } to { ip: $NEW_IP, port: $NEW_PORT }"
done
```

Figure 4.14: While loop migration

4.3.1 Running the Scenario

Repeat the following commands three times, because three terminals in the Comnetsemu machine are needed.

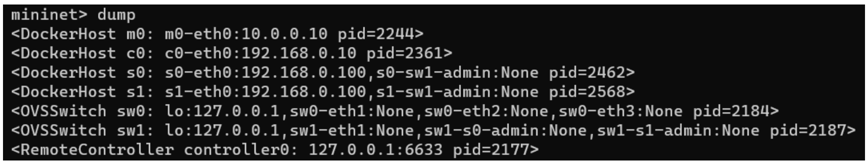
```
$ cd SVMN/comnetsemu
$ vagrant ssh
$ cd comnetsmu/app/morphing_slices
```

Then, follow the steps:

1. Generate the network from the topology.yaml file in the first terminal:

```
$ sudo python3 topology.py −−file scenarios/1/topology.yaml
```

   The output should be similar to figure 4.15 executing the Mininet command *dump*.



```
mininet> dump
<DockerHost m0: m0-eth0:10.0.0.10 pid=2244>
<DockerHost c0: c0-eth0:192.168.0.10 pid=2361>
<DockerHost s0: s0-eth0:192.168.0.100,s0-sw1-admin:None pid=2462>
<DockerHost s1: s1-eth0:192.168.0.100,s1-sw1-admin:None pid=2568>
<OVSSwitch sw0: lo:127.0.0.1,sw0-eth1:None,sw0-eth2:None,sw0-eth3:None pid=2184>
<OVSSwitch sw1: lo:127.0.0.1,sw1-eth1:None,sw1-s0-admin:None,sw1-s1-admin:None pid=2187>
<RemoteController controller0: 127.0.0.1:6633 pid=2177>
```

Figure 4.15: Mininet dump

2. In the second terminal, start the FlowVisor container attaching the folder *scenarios/1* as volume:

```
$ scripts/flowvisor.sh −−volume scenarios/1
```

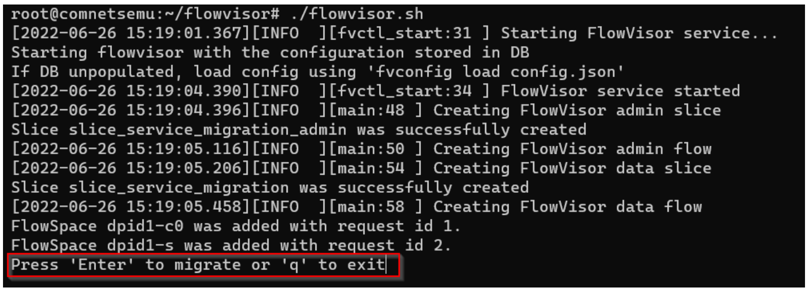
```
vagrant@comnetsemu:~/comnetsemu/app/morphing_slices$ scripts/flowvisor.sh --volume scenarios/1
[2022-06-26 15:18:26.735][INFO  ][main:62 ] Starting Docker server...
[2022-06-26 15:18:26.742][INFO  ][main:66 ] Docker server started with PID 10045
[2022-06-26 15:18:26.747][INFO  ][main:69 ] Run Flowvisor image 'flowvisor:latest'
[2022-06-26 15:18:26.754][INFO  ][main:71 ] Mounted shared volume /home/vagrant/comnetsemu/app/morphing_slices/scenarios/1
root@comnetsemu:~/flowvisor#
```

Figure 4.16: Run FlowVisor container

   The script for creating the Docker container is located in *src/scripts/flowvisor.sh*.

   Then, run the script *src/scenarios/1/flowvisor.sh* in the FlowVisor container:

```
$ ./flowvisor.sh
```



```
root@comnetsemu:~/flowvisor# ./flowvisor.sh
[2022-06-26 15:19:01.367][INFO  ][fvctl_start:31 ] Starting FlowVisor service...
Starting flowvisor with the configuration stored in DB
If DB unpopulated, load config using 'fvconfig load config.json'
[2022-06-26 15:19:04.390][INFO  ][fvctl_start:34 ] FlowVisor service started
[2022-06-26 15:19:04.396][INFO  ][main:48 ] Creating FlowVisor admin slice
Slice slice_service_migration_admin was successfully created
[2022-06-26 15:19:05.116][INFO  ][main:50 ] Creating FlowVisor admin flow
[2022-06-26 15:19:05.206][INFO  ][main:54 ] Creating FlowVisor data slice
Slice slice_service_migration was successfully created
[2022-06-26 15:19:05.458][INFO  ][main:58 ] Creating FlowVisor data flow
FlowSpace dpid1-c0 was added with request id 1.
FlowSpace dpid1-s was added with request id 2.
Press 'Enter' to migrate or 'q' to exit
```

Figure 4.17: Run *flowvisor.sh* migration script

   The script is already in the container, because the folder has been mounted in the previous step.

3. In the third terminal, run the two RYU controllers:

```
$ parallel —j 2 ——ungroup ::: \
'scripts/ryu.sh ——controller scenarios/1/controller.py ——ofport 10001 \
——port 8082 ——config scenarios/1/controller.cfg' \
'scripts/ryu.sh ——controller scenarios/1/controller.py ——ofport 10002 \
——port 8083'
```

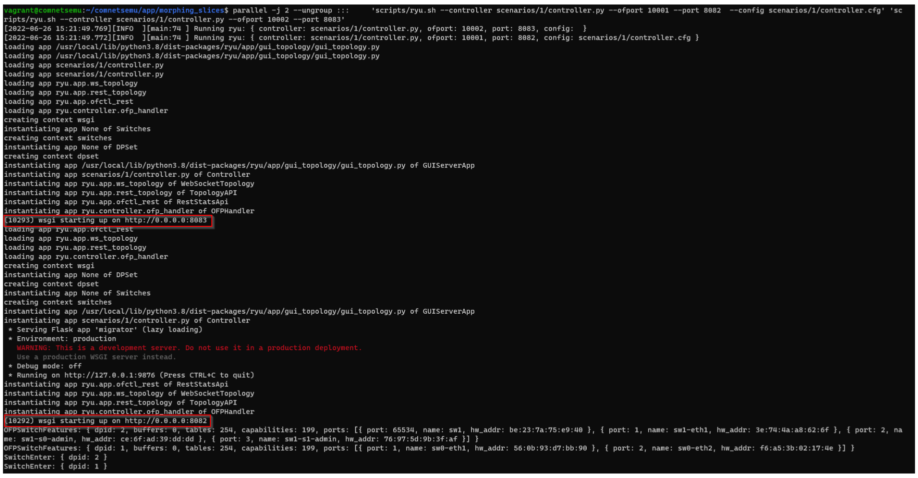


Figure 4.18: Run controllers

The RYU Graphical User Interfaces can be accessed from the browser at the link *http://localhost:8082* and *http://localhost:8083*.
The RYU GUI shows the datapath handled by the controller. So, the Controller 0 handles the datapath 1 related to the switch *sw0* as reported in the figure 4.19.

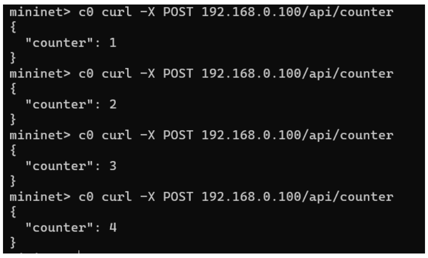
Figure 4.19: RYU GUI at *http://localhost:8082*

The Controller 1 handles the datapath 2 related to the switch *sw1* as reported in the figure 4.20.

Figure 4.20: RYU GUI at *http://localhost:8083*

Any request can be performed to the enabled server in the terminal where the network topology has been deployed. So, the client can increment the counter.

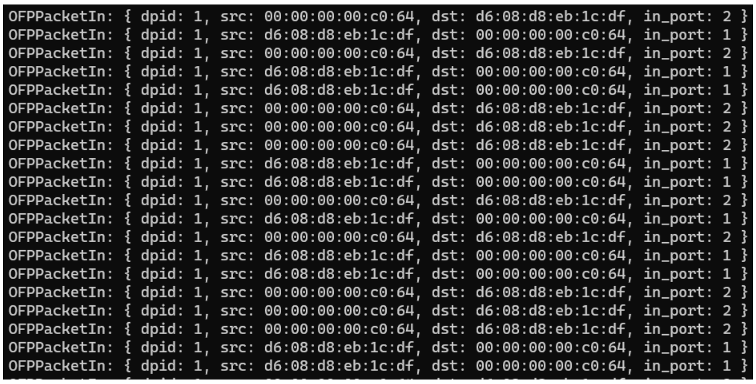
```
$ c0 curl −X POST 192.168.0.100/api/counter
```



```
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 1
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 2
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 3
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 4
}
```

Figure 4.21: Client increments counter

At the beginning, the server *s0* is enabled and its *in_port* is the 2. Indeed, *s0* is replying to the client checking from the controller logs reported in the figure 4.22. Instead, the server *s1* is disabled and its *in_port* is the 3.



```
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 2 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
```

Figure 4.22: *s0* replies to client

The migration can be done just pressing *Enter* in the FlowVisor terminal.

```
root@comnetsemu:~/flowvisor# ./flowvisor.sh
[2022-06-26 15:19:01.367][INFO  ][fvctl_start:31 ] Starting FlowVisor service...
Starting flowvisor with the configuration stored in DB
If DB unpopulated, load config using 'fvconfig load config.json'
[2022-06-26 15:19:04.390][INFO  ][fvctl_start:34 ] FlowVisor service started
[2022-06-26 15:19:04.396][INFO  ][main:48 ] Creating FlowVisor admin slice
Slice slice_service_migration_admin was successfully created
[2022-06-26 15:19:05.116][INFO  ][main:50 ] Creating FlowVisor admin flow
[2022-06-26 15:19:05.206][INFO  ][main:54 ] Creating FlowVisor data slice
Slice slice_service_migration was successfully created
[2022-06-26 15:19:05.458][INFO  ][main:58 ] Creating FlowVisor data flow
FlowSpace dpid1-c0 was added with request id 1.
FlowSpace dpid1-s was added with request id 2.
Press 'Enter' to migrate or 'q' to exit
```

Figure 4.23: FlowVisor migration

The client can continue to increment the counter where it has left without noticing the migration.

```
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 1
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 2
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 3
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 4
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 5
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 6
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 7
}
```

Figure 4.24: *c0* increments the counter after migration

However, the server *s1* is the enabled and the *s0* is disabled. It can be checked from the controller logs as reported in figure 4.25.

```
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: ff:ff:ff:ff:ff:ff, in_port: 3 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: d6:08:d8:eb:1c:df, dst: 00:00:00:00:c0:64, in_port: 1 }
OFPPacketIn: { dpid: 1, src: 00:00:00:00:c0:64, dst: d6:08:d8:eb:1c:df, in_port: 3 }
```

Figure 4.25: *s1* replies to client

4.4 Scenario 2

The Scenario 2 is located in the folder *src/scenarios/2* and the network topology is reported in figure 4.26.

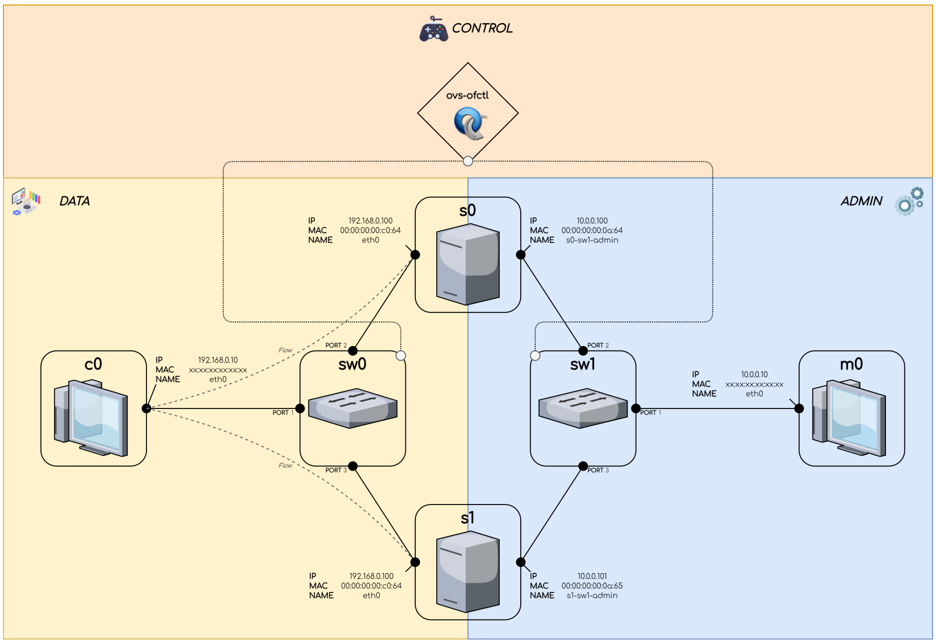


Figure 4.26: Network topology

There are several devices in the network:

- Client *c0*: it performs the HTTP requests to the enabled server.
- Servers *s0* and *s1*: they implement the Flask web application. By default, *s0* is the enabled server and *s1* is disabled. After the migration, *s1* will become enabled getting the current counter value from *s0*.
- Switch *sw0*: it connects *c0* with the two servers *s0* and *s1*.
- Manager *m0*: it emulates the network administrator and executes the command for the migration.
- Switch *sw1*: it connects *m0* with the two servers *s0* and *s1*.

The OpenFlow protocol is implemented by ovs-ofctl in the scenario which defines flows for both *data* and *admin* subnets. Flows are defined by the script *src/scenarios/2/openflow.sh*.

```
# OpenFlow admin flow
INFO "Creating OpenFlow admin flow"
ofctl_exec add-flow sw1 actions=normal

# OpenFlow data flow
INFO "Creating OpenFlow data flow"
ofctl_exec add-flow sw0 in_port=1,actions=output:"${SERVERS_PORT[IDX_SERVER]}"
ofctl_exec add-flow sw0 in_port=2,actions=output:1
ofctl_exec add-flow sw0 in_port=3,actions=output:1
```

Figure 4.27: ovs-ofctl flows

As shown in figure 4.27, one flow is created for the *admin* and three flows for the *data* defined by the OpenFlow attribute *in_port* and the action *out_port*. The migration works adjusting dynamically the flow between the switch *sw0* and servers. So, there is a while loop to wait the user input to perform the migration. After pressing *Enter*, the *out_port* of the flow will be changed and the manager *m0* will perform the migration between servers *s0* and *s1* using Docker command line.

```
# Migration loop
while read -n1 -r -p "Press 'Enter' to migrate or 'q' to exit" && [[ $REPLY != q ]]; do
    # Old ip
    OLD_IP=${SERVERS_IP[IDX_SERVER]}
    # Old port
    OLD_PORT=${SERVERS_PORT[IDX_SERVER]}
    # Update idx server
    IDX_SERVER="$(next_idx_server "$IDX_SERVER")"
    # New ip
    NEW_IP=${SERVERS_IP[IDX_SERVER]}
    # New port
    NEW_PORT=${SERVERS_PORT[IDX_SERVER]}

    INFO "Migrating from { ip: $OLD_IP, port: $OLD_PORT } to { ip: $NEW_IP, port: $NEW_PORT }"

    # Docker manager
    docker exec --detach m0 curl -X POST -H \"Content-Type:application/json\" -d "{ \"server\": \"http://$OLD_IP\" }" "$NEW_IP/api/admin/migrate"

    # OpenFlow data flow
    ofctl_exec mod-flows sw0 in_port=1,actions=output:"$NEW_PORT"

    INFO "Successfully migrated from { ip: $OLD_IP, port: $OLD_PORT } to { ip: $NEW_IP, port: $NEW_PORT }"
done
```

Figure 4.28: ovs-ofctl migration

To sum up, there are no remote RYU controllers to manage the traffic, but the migration can be performed just adjusting flows.

4.4.1 Running the Scenario

Repeat the following commands two times, because two terminals are needed for the scenario.

```
$ cd SVMN/comnetsemu
$ vagrant ssh
$ cd comnetsmu/app/morphing_slices
```

Then, follow the steps:

- Generate the network from the topology.yaml file in the first terminal:

  ```
  $ sudo python3 topology.py --file scenarios/2/topology.yaml
  ```

  The output should be similar to figure 4.29 executing the Mininet command *dump*.

```
mininet> dump
<DockerHost m0: m0-eth0:10.0.0.10 pid=4066>
<DockerHost c0: c0-eth0:192.168.0.10 pid=4178>
<DockerHost s0: s0-eth0:192.168.0.100,s0-sw1-admin:None pid=4288>
<DockerHost s1: s1-eth0:192.168.0.100,s1-sw1-admin:None pid=4399>
<OVSSwitch sw0: lo:127.0.0.1,sw0-eth1:None,sw0-eth2:None,sw0-eth3:None pid=4011>
<OVSSwitch sw1: lo:127.0.0.1,sw1-eth1:None,sw1-s0-admin:None,sw1-s1-admin:None pid=4014>
<Controller controller0: 127.0.0.1:6653 pid=4003>
```

Figure 4.29: Mininet dump

- In the second terminal, run the ovs-ofctl script called *openflow.sh*.

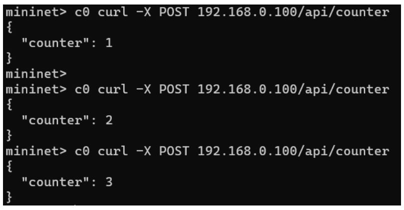
  ```
  $ scenarios/2/openflow.sh
  ```

```
vagrant@comnetsemu:~/comnetsemu/app/morphing_slices$ scenarios/2/openflow.sh
[2022-07-02 14:47:08.458][INFO  ][main:27 ] Creating OpenFlow admin flow
[2022-07-02 14:47:08.498][INFO  ][main:31 ] Creating OpenFlow data flow
Press 'Enter' to migrate or 'q' to exit
```

Figure 4.30: *openflow.sh* output

Any request can be performed to the enabled server in the terminal where the network topology has been deployed. So, the client can increment the counter.

```
$ c0 curl −X POST 192.168.0.100/api/counter
```

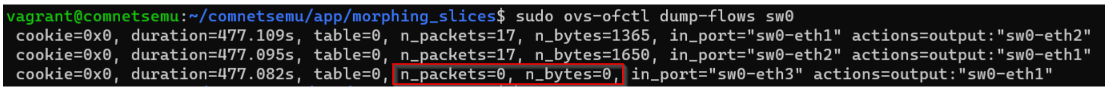


```
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 1
}
mininet>
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 2
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 3
}
```

Figure 4.31: Client increments counter

Flows can be checked in another terminal typing:

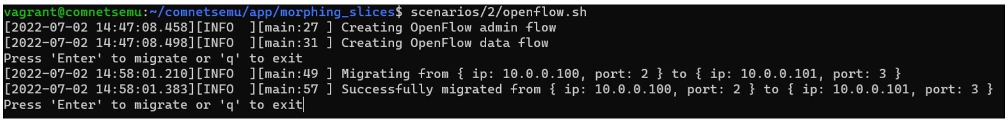
```
$ sudo ovs−ofctl dump−flows sw0
```



```
vagrant@comnetsemu:~/comnetsemu/app/morphing_slices$ sudo ovs-ofctl dump-flows sw0
 cookie=0x0, duration=477.109s, table=0, n_packets=17, n_bytes=1365, in_port="sw0-eth1" actions=output:"sw0-eth2"
 cookie=0x0, duration=477.095s, table=0, n_packets=17, n_bytes=1650, in_port="sw0-eth2" actions=output:"sw0-eth1"
 cookie=0x0, duration=477.082s, table=0, n_packets=0, n_bytes=0, in_port="sw0-eth3" actions=output:"sw0-eth1"
```

Figure 4.32: Check flows

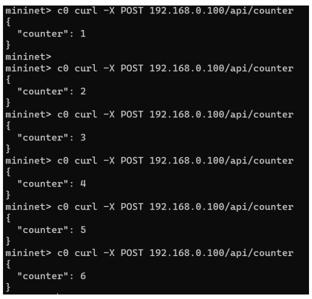
As shown in figure 4.32, there is no traffic in the flows with *in_port sw0-eth3*, because it is related to the server *s1* which is disabled at the beginning. The migration can be done just pressing *Enter* in the ovs-ofctl terminal.



```
vagrant@comnetsemu:~/comnetsemu/app/morphing_slices$ scenarios/2/openflow.sh
[2022-07-02 14:47:08.458][INFO  ][main:27 ] Creating OpenFlow admin flow
[2022-07-02 14:47:08.498][INFO  ][main:31 ] Creating OpenFlow data flow
Press 'Enter' to migrate or 'q' to exit
[2022-07-02 14:58:01.210][INFO  ][main:49 ] Migrating from { ip: 10.0.0.100, port: 2 } to { ip: 10.0.0.101, port: 3 }
[2022-07-02 14:58:01.383][INFO  ][main:57 ] Successfully migrated from { ip: 10.0.0.100, port: 2 } to { ip: 10.0.0.101, port: 3 }
Press 'Enter' to migrate or 'q' to exit
```

Figure 4.33: Perform migration

The client can continue to increment the counter where it has left without noticing the migration.



```
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 1
}
mininet>
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 2
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 3
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 4
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 5
}
mininet> c0 curl -X POST 192.168.0.100/api/counter
{
  "counter": 6
}
```

Figure 4.34: *c0* increments the counter after migration

Now, there is traffic in the flows with *in_port sw0-eth3*, because the server *s1* has become enabled.

```
vagrant@comnetsemu:~/comnetsemu/app/morphing_slices$ sudo ovs-ofctl dump-flows sw0
 cookie=0x0, duration=893.809s, table=0, n_packets=17, n_bytes=1650, in_port="sw0-eth2" actions=output:"sw0-eth1"
 cookie=0x0, duration=893.796s, table=0, n_packets=16, n_bytes=1608, in_port="sw0-eth3" actions=output:"sw0-eth1"
 cookie=0x0, duration=893.823s, table=0, n_packets=33, n_bytes=2688, in_port="sw0-eth1" actions=output:"sw0-eth3"
```

Figure 4.35: Recheck flows

4.5 Scenario 3

The Scenario 3 is located in the folder *src/scenarios/3* and the network topology is reported in figure 4.36.

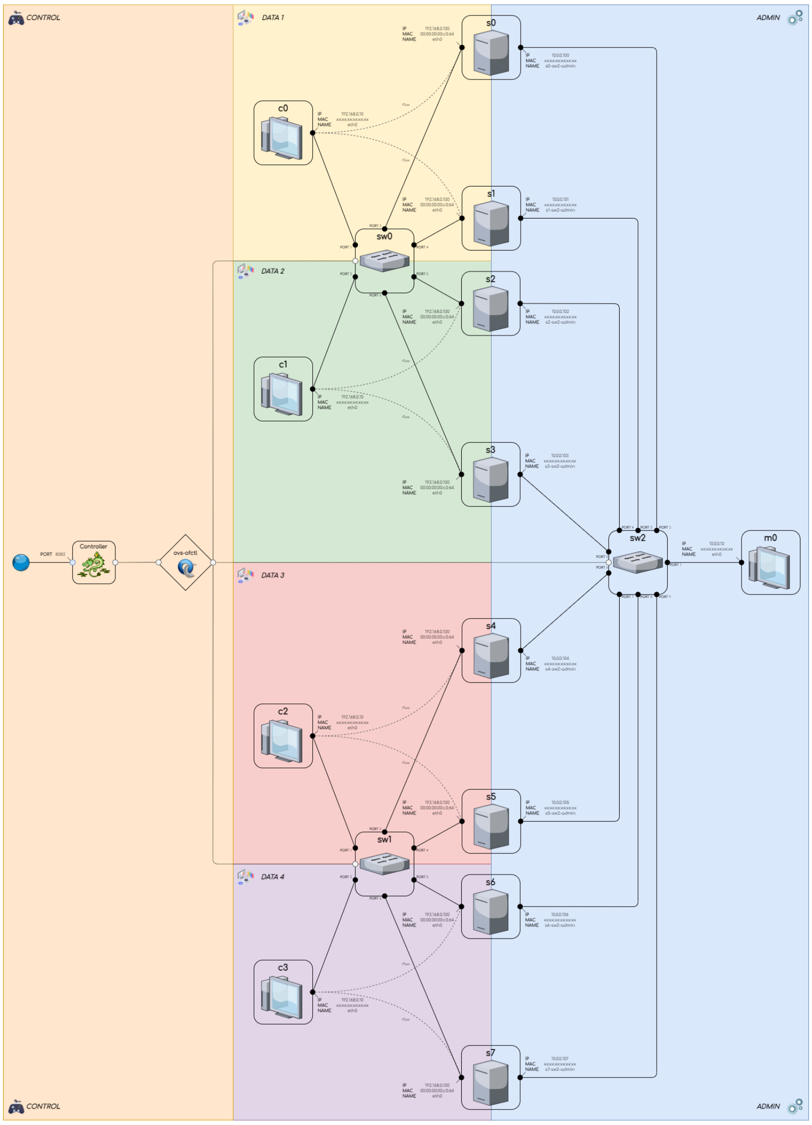


Figure 4.36: Network topology

There are a lot of devices in the network to prove the project scalability:

- Client *c0, c1, c2, c3*: they perform the HTTP requests to the enabled server in the correspondent slice.
- Servers *s0, s1, s2, s3, s4, s5, s6, s7*: they implement the Flask web application. By default, *s0, s2, s4, s6* are the enabled servers and *s1, s3, s5, s7* are disabled.
- Switch *sw0*: it connects *c0* with the two servers *s0* and *s1*. Moreover, it connects *c1* with the two servers *s2* and *s3*.
- Switch *sw1*: it connects *c2* with the two servers *s4* and *s5*. Moreover, it connects *c3* with the two servers *s6* and *s7*.
- Manager *m0*: it emulates the network administrator and executes the command for the migration.
- Switch *sw2*: it connects *m0* with the all the servers.
- Controller 0: it handles the network traffic.

The OpenFlow protocol is implemented by ovs-ofctl which defines slices and flows for the entire network. Slices and flows are defined by the script *src/scenarios/3/openflow.sh*.

```
# OpenFlow admin slice
INFO "Create admin slice on switch sw2"
sudo ovs-vsctl set port sw2-eth1 qos=@admin -- \
--id=@admin create qos type=linux-htb \
queues:777=@adm -- \
--id=@adm create queue other-config:max-rate=1000000
# OpenFlow admin flow
INFO "Creating OpenFlow admin flow"
ofctl_exec add-flow sw2 actions=set_queue:777,normal

# OpenFlow data slices sw0
INFO "Create slices data_1 and data_2 on switch sw0"
sudo ovs-vsctl set port sw0-eth1 qos=@data -- \
--id=@data create qos type=linux-htb \
queues:123=@data1 \
queues:234=@data2 -- \
--id=@data1 create queue other-config:max-rate=1000000 -- \
--id=@data2 create queue other-config:max-rate=1000000
# OpenFlow data_1 flows
INFO "Creating OpenFlow data_1 flows"
ofctl_exec add-flow sw0 in_port=1,actions=set_queue:123,output:"${SLICE_1_SERVERS_PORT[SLICE_1_IDX_SERVER]}"
ofctl_exec add-flow sw0 in_port=3,actions=set_queue:123,output:1
ofctl_exec add-flow sw0 in_port=4,actions=set_queue:123,output:1
# OpenFlow data_2 flows
INFO "Creating OpenFlow data_2 flows"
ofctl_exec add-flow sw0 in_port=2,actions=set_queue:234,output:"${SLICE_2_SERVERS_PORT[SLICE_2_IDX_SERVER]}"
ofctl_exec add-flow sw0 in_port=5,actions=set_queue:234,output:2
ofctl_exec add-flow sw0 in_port=6,actions=set_queue:234,output:2

# OpenFlow data slices sw1
INFO "Create slices data_3 and data_4 on switch sw1"
sudo ovs-vsctl set port sw1-eth1 qos=@data -- \
--id=@data create qos type=linux-htb \
queues:123=@data3 \
queues:234=@data4 -- \
--id=@data3 create queue other-config:max-rate=1000000 -- \
--id=@data4 create queue other-config:max-rate=1000000
# OpenFlow data_3 flows
INFO "Creating OpenFlow data_1 flows"
ofctl_exec add-flow sw1 in_port=1,actions=set_queue:123,output:"${SLICE_3_SERVERS_PORT[SLICE_3_IDX_SERVER]}"
ofctl_exec add-flow sw1 in_port=3,actions=set_queue:123,output:1
ofctl_exec add-flow sw1 in_port=4,actions=set_queue:123,output:1
# OpenFlow data_4 flows
INFO "Creating OpenFlow data_2 flows"
ofctl_exec add-flow sw1 in_port=2,actions=set_queue:234,output:"${SLICE_4_SERVERS_PORT[SLICE_4_IDX_SERVER]}"
ofctl_exec add-flow sw1 in_port=5,actions=set_queue:234,output:2
ofctl_exec add-flow sw1 in_port=6,actions=set_queue:234,output:2
```

Figure 4.37: ovs-ofctl slices and flows

There are the following slices:

- *data 1, data 2, data 3, data 4*: in each *data* slice, there are one client and two servers.
- *admin*: where the manager *m0* handles the server migration for every *data* slice.

When the script *src/scenarios/3/openflow.sh* runs, a while loop waits for the user input to perform the migration. The user can migrate in two modes:

- Update: the flow is updated with the new *in_port* attribute.
- Delete: the flow is removed and created with the new *in_port* attribute.

After the user picks the mode and in which slice to migrate:

- the manager *m0* performs the server migration via the Docker command line.
- the controller modifies the network traffic updating the flow, performing an HTTP post request to the *migrator.py* Flask web server described below.

```
# Docker manager
docker exec -d m0 curl -X POST -H \"Content-Type:application/json\" -d "{ \"server\": \"http://$OLD_IP\" }" "$NEW_IP"/api/admin/migrate
# Controller flow
curl -X POST -H \"Content-Type:application/json\" -d "{ \"mode\": \"$MIGRATION_MODE\", \"dpid\": \"$DATAPATH_ID\", \"in_port\": \"$CLIENT_PORT\", \"out_port\": \"$NEW_PORT\" }" "localhost:$CONTROLLER_PORT/api/migrate"
```

Figure 4.38: Operations for migration

So, ovs-ofctl only creates slices and flows without updating them. The migration is entirely handled by the Controller 0 for any *data* slice dynamically adjusting the flow to always connect the client to the enabled server. The Controller 0 stores how to forward the traffic for switch *sw0* and *sw1* connecting the different *in_ports* in the dictionary *self.in_to_out*.

```
        self.in_to_out = {
            1: {1: 3, 2: 5, 3: 1, 4: 1, 5: 2, 6: 2},
            2: {1: 3, 2: 5, 3: 1, 4: 1, 5: 2, 6: 2},
        }
```

Figure 4.39: Port mapping for switch *sw0* and *sw1*

Furthermore, the Controller 0 runs a Flask web server *src/secenarios/3/migrate.py* in a new python thread to handle incoming HTTP request to update the flows and mapping. As soon as the controller is launched, the Flask web server shown in the figure 4.40 starts.

```python
@app.route("/api/migrate", methods=["POST"])
def migrate():
    """
    Migrate service.
    """

    # Request body
    body = request.get_json(force=True)
    # Mode
    mode = int(body["mode"])
    # Switch dpid
    dpid = int(body["dpid"])
    # In port
    in_port = int(body["in_port"])
    # Out port
    out_port = int(body["out_port"])

    # Notify callback
    if cb:
        cb(mode, dpid, in_port, out_port)

    return ("", 200)


def start(port: int, callback):
    global cb
    cb = callback
    app.run(port=port)
```

Figure 4.40: *migrator.py*

The *migrator.py* accepts HTTP post request at */api/migrate* and call the method *migration_cb* to update or modify the flows and mapping. The mode, datapath id, MAC address and the new in port has to be passed in the body of the request. The mode parameter can be update or delete. Then, the method *migration_cb* shown in the figure 4.41 executes the user request.

```python
    def migration_cb(self, mode: int, dpid: int, in_port: int, out_port: int):
        datapath: Datapath = self.get_datapath(dpid)

        if mode == MIGRATION_MODE_UPDATE:
            match = datapath.ofproto_parser.OFPMatch(in_port=in_port)
            actions = [datapath.ofproto_parser.OFPActionOutput(out_port)]
            self.update_flow(datapath, match, actions)
        elif mode == MIGRATION_MODE_DELETE:
            self.delete_flow(datapath, in_port)
        else:
            self.logger.warn(f"Unknown migration mode {mode}")

        self.in_to_out[dpid][in_port] = out_port
```

Figure 4.41: *migrator_cb*

Hence, the Controller 0 is able to update the network traffic and flows in any *data* slice on its own. At the beginning, the Controller 0 can create flows considering the incoming OpenFlow traffic with the method *add_flow* shown in the figure 4.42.

```python
    def add_flow(
        self, datapath: Datapath, priority: int, match, actions, buffer_id=None
    ):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        self.logger.info(
            f"Add flow: {{ dpid: {datapath.id}, priority: {priority}, match: {match}, actions: {actions} }}"
        )

        if buffer_id:
            mod = parser.OFPFlowMod(
                datapath=datapath,
                match=match,
                command=ofproto.OFPFC_ADD,
                priority=priority,
                flags=ofproto.OFPFF_SEND_FLOW_REM,
                actions=actions,
                buffer_id=buffer_id,
            )
        else:
            mod = parser.OFPFlowMod(
                datapath=datapath,
                match=match,
                command=ofproto.OFPFC_ADD,
                priority=priority,
                flags=ofproto.OFPFF_SEND_FLOW_REM,
                actions=actions,
            )

        datapath.send_msg(mod)
```

Figure 4.42: Create flow

The Controller 0 can update a flow with the method *update_flow* reported in figure the 4.43.

```python
    def update_flow(self, datapath: Datapath, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        self.logger.info(
            f"Update flow: {{ dpid: {datapath.id}, match: {match}, actions: {actions} }}"
        )

        mod = parser.OFPFlowMod(
            datapath=datapath,
            match=match,
            command=ofproto.OFPFC_MODIFY,
            actions=actions,
        )

        datapath.send_msg(mod)
```

Figure 4.43: Update flow

The Controller 0 can delete a flow with the method *delete_flow* reported in figure the 4.44.

```python
    def delete_flow(self, datapath: Datapath, in_port: int):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        self.logger.info(
            f"Delete flow: {{ dpid: {datapath.id}, in_port={in_port}, out_port: {self.in_to_out[datapath.id][in_port]} }}"
        )

        match = parser.OFPMatch(in_port=in_port)
        mod = parser.OFPFlowMod(
            datapath=datapath,
            match=match,
            command=ofproto.OFPFC_DELETE,
        )

        datapath.send_msg(mod)
```

Figure 4.44: Delete flow

After deleting a flow, it will be recreated again with new *in_port* attribute considering the incoming OpenFlow traffic.

4.5.1 Running the Scenario

Repeat the following commands three times, because three terminals in the Comnetsemu machine are needed.

```
$ cd SVMN/comnetsemu
$ vagrant ssh
$ cd comnetsmu/app/morphing_slices
```

Then, follow the steps:

1. Generate the network from the topology.yaml file in the first terminal:

   ```
   $ sudo python3 topology.py --file scenarios/3/topology.yaml
   ```

   The output should be similar to figure 4.45 executing the Mininet command *dump*.

```
mininet> dump
<DockerHost m0: m0-eth0:10.0.0.10 pid=5652>
<DockerHost c0: c0-eth0:192.168.0.10 pid=5767>
<DockerHost c1: c1-eth0:192.168.0.10 pid=5881>
<DockerHost c2: c2-eth0:192.168.0.10 pid=5988>
<DockerHost c3: c3-eth0:192.168.0.10 pid=6095>
<DockerHost s0: s0-eth0:192.168.0.100,s0-sw2-admin:None pid=6195>
<DockerHost s1: s1-eth0:192.168.0.100,s1-sw2-admin:None pid=6299>
<DockerHost s2: s2-eth0:192.168.0.100,s2-sw2-admin:None pid=6400>
<DockerHost s3: s3-eth0:192.168.0.100,s3-sw2-admin:None pid=6506>
<DockerHost s4: s4-eth0:192.168.0.100,s4-sw2-admin:None pid=6610>
<DockerHost s5: s5-eth0:192.168.0.100,s5-sw2-admin:None pid=6707>
<DockerHost s6: s6-eth0:192.168.0.100,s6-sw2-admin:None pid=6809>
<DockerHost s7: s7-eth0:192.168.0.100,s7-sw2-admin:None pid=6910>
<OVSSwitch sw0: lo:127.0.0.1,sw0-eth1:None,sw0-eth2:None,sw0-eth3:None,sw0-eth4:None,sw0-eth5:None,sw0-eth6:None pid=5592>
<OVSSwitch sw1: lo:127.0.0.1,sw1-eth1:None,sw1-eth2:None,sw1-eth3:None,sw1-eth4:None,sw1-eth5:None,sw1-eth6:None pid=5595>
<OVSSwitch sw2: lo:127.0.0.1,sw2-eth1:None,sw2-s0-admin:None,sw2-s1-admin:None,sw2-s2-admin:None,sw2-s3-admin:None,sw2-s4-admin:None,sw2-s5-admin:None,sw2-s
6-admin:None,sw2-s7-admin:None pid=5598>
<RemoteController controller0: 127.0.0.1:6653 pid=5585>
```

Figure 4.45: Mininet dump

2. In the second terminal, run the ovs-ofctl script called *openflow.sh*.

```
$ scenarios/3/openflow.sh
```

Pick between *UPDATE* or *DELETE* mode for the migration and the output should be similar to figure 4.46.

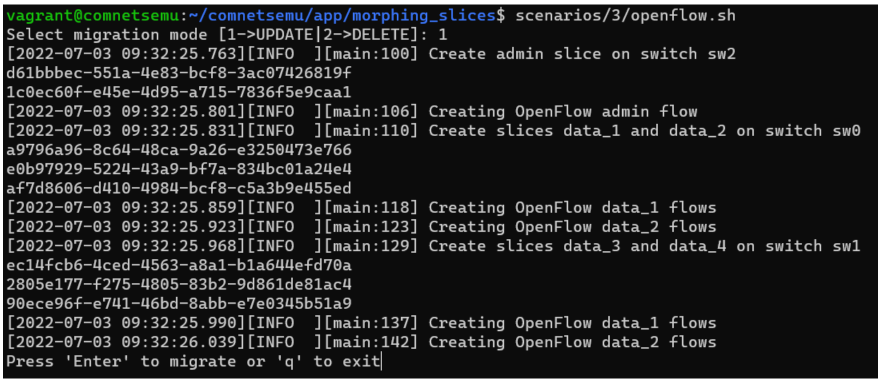


Figure 4.46: Run *openflow.sh*

3. In the third terminal, run the RYU controller:

```
$ scripts/ryu.sh --controller scenarios/3/controller.py --port 8082 \
--config scenarios/3/controller.cfg
```

Figure 4.47: Run RYU Controller 0

The RYU Graphical User Interface can be accessed from the browser at the link *http://localhost:8082*.

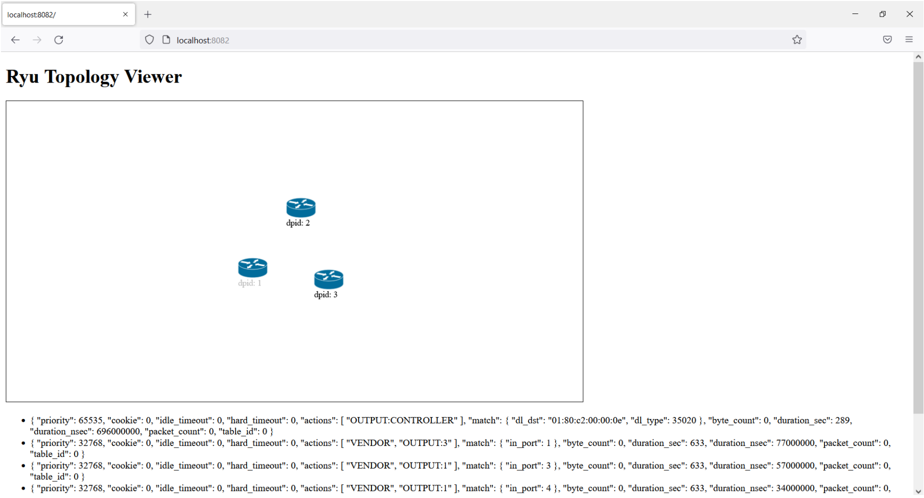


Figure 4.48: Run RYU at textithttp://localhost:8082

Requests can be performed by client *c0*, *c1*, *c2*, *c3*.

```
$ c1 curl —X POST 192.168.0.100/api/counter
```

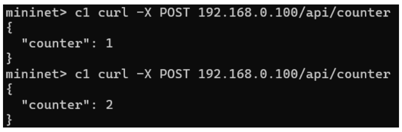


Figure 4.49: Client *c1* increments counter

Slices and flows can be checked in another terminal typing:

```
$ sudo ovs—ofctl dump—flows sw0
$ sudo ovs—ofctl dump—flows sw1
```

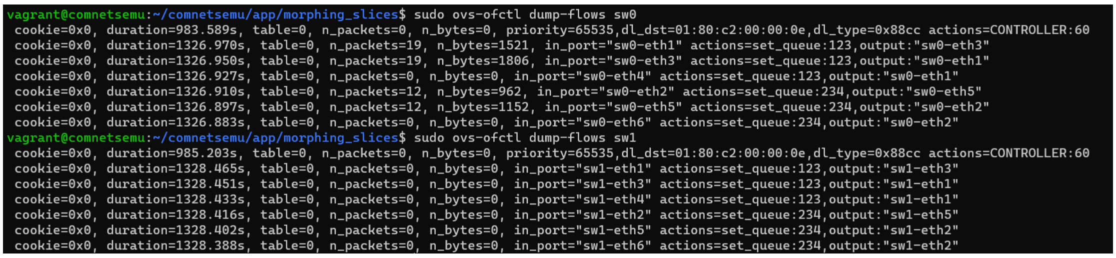


Figure 4.50: Check slices and flows

In the second terminal, the migration can be performed picking the *data* slice corresponding to the client. In the example, the *data* slice of the client *c1* is the number 2 according to the network topology.

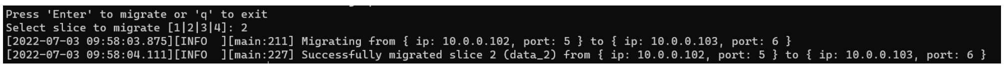


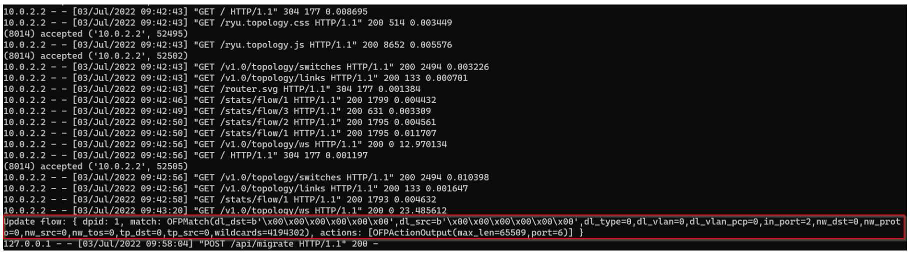
Figure 4.51: Perform migration

So, the flow is updated by the Controller 0 with the method *update_flow* shown in the figure 4.43. The client *c1* can continue to increment the counter where it has left without noticing the migration.

```
mininet> c1 curl -X POST 192.168.0.100/api/counter
{
  "counter": 1
}
mininet> c1 curl -X POST 192.168.0.100/api/counter
{
  "counter": 2
}
mininet> c1 curl -X POST 192.168.0.100/api/counter
{
  "counter": 3
}
mininet> c1 curl -X POST 192.168.0.100/api/counter
{
  "counter": 4
}
mininet> c1 curl -X POST 192.168.0.100/api/counter
{
  "counter": 5
}
```

Figure 4.52: *c1* increments counter after migration

The update operation can be checked in the Controller 0 logs as shown in the figure 4.53.

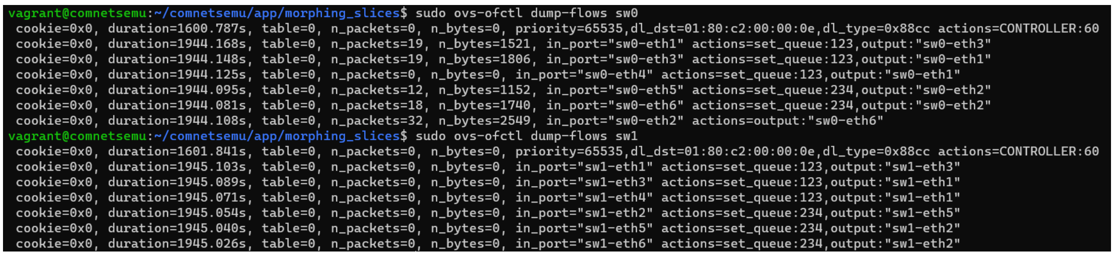


```
10.0.2.2 - - [03/Jul/2022 09:42:43] "GET / HTTP/1.1" 304 177 0.008695
10.0.2.2 - - [03/Jul/2022 09:42:43] "GET /ryu.topology.css HTTP/1.1" 200 514 0.003449
(8014) accepted ('10.0.2.2', 52495)
10.0.2.2 - - [03/Jul/2022 09:42:43] "GET /ryu.topology.js HTTP/1.1" 200 8652 0.005576
(8014) accepted ('10.0.2.2', 52502)
10.0.2.2 - - [03/Jul/2022 09:42:43] "GET /v1.0/topology/switches HTTP/1.1" 200 2494 0.003226
10.0.2.2 - - [03/Jul/2022 09:42:43] "GET /v1.0/topology/links HTTP/1.1" 200 133 0.000701
10.0.2.2 - - [03/Jul/2022 09:42:43] "GET /router.svg HTTP/1.1" 304 177 0.001384
10.0.2.2 - - [03/Jul/2022 09:42:46] "GET /stats/flow/1 HTTP/1.1" 200 1799 0.004432
10.0.2.2 - - [03/Jul/2022 09:42:49] "GET /stats/flow/3 HTTP/1.1" 200 631 0.003309
10.0.2.2 - - [03/Jul/2022 09:42:50] "GET /stats/flow/2 HTTP/1.1" 200 1795 0.004561
10.0.2.2 - - [03/Jul/2022 09:42:50] "GET /stats/flow/1 HTTP/1.1" 200 1795 0.011707
10.0.2.2 - - [03/Jul/2022 09:42:56] "GET /v1.0/topology/ws HTTP/1.1" 200 0 12.970134
10.0.2.2 - - [03/Jul/2022 09:42:56] "GET / HTTP/1.1" 304 177 0.001197
(8014) accepted ('10.0.2.2', 52505)
10.0.2.2 - - [03/Jul/2022 09:42:56] "GET /v1.0/topology/switches HTTP/1.1" 200 2494 0.010398
10.0.2.2 - - [03/Jul/2022 09:42:56] "GET /v1.0/topology/links HTTP/1.1" 200 133 0.001647
10.0.2.2 - - [03/Jul/2022 09:42:58] "GET /stats/flow/1 HTTP/1.1" 200 1793 0.004632
10.0.2.2 - - [03/Jul/2022 09:43:20] "GET /v1.0/topology/ws HTTP/1.1" 200 0 23.485612
Update flow: { dpid: 1, match: OFPMatch(dl_dst=b'\x00\x00\x00\x00\x00\x00',dl_src=b'\x00\x00\x00\x00\x00\x00',dl_type=0,dl_vlan=0,dl_vlan_pcp=0,in_port=2,nw_dst=0,nw_prot
o=0,nw_src=0,nw_tos=0,tp_dst=0,tp_src=0,wildcards=4194302), actions: [OFPActionOutput(max_len=65509,port=6)] }
127.0.0.1 - - [03/Jul/2022 09:58:04] "POST /api/migrate HTTP/1.1" 200 -
```

Figure 4.53: Controller 0 logs

Then, slices and flows can be rechecked again.

```
$ sudo ovs—ofctl dump—flows sw0
$ sudo ovs—ofctl dump—flows sw1
```



```
vagrant@comnetsemu:~/comnetsemu/app/morphing_slices$ sudo ovs-ofctl dump-flows sw0
 cookie=0x0, duration=1600.787s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
 cookie=0x0, duration=1944.168s, table=0, n_packets=19, n_bytes=1521, in_port="sw0-eth1" actions=set_queue:123,output:"sw0-eth3"
 cookie=0x0, duration=1944.148s, table=0, n_packets=19, n_bytes=1806, in_port="sw0-eth3" actions=set_queue:123,output:"sw0-eth1"
 cookie=0x0, duration=1944.125s, table=0, n_packets=0, n_bytes=0, in_port="sw0-eth4" actions=set_queue:123,output:"sw0-eth1"
 cookie=0x0, duration=1944.095s, table=0, n_packets=12, n_bytes=1152, in_port="sw0-eth5" actions=set_queue:234,output:"sw0-eth2"
 cookie=0x0, duration=1944.081s, table=0, n_packets=18, n_bytes=1740, in_port="sw0-eth6" actions=set_queue:234,output:"sw0-eth2"
 cookie=0x0, duration=1944.108s, table=0, n_packets=32, n_bytes=2549, in_port="sw0-eth2" actions=output:"sw0-eth6"
vagrant@comnetsemu:~/comnetsemu/app/morphing_slices$ sudo ovs-ofctl dump-flows sw1
 cookie=0x0, duration=1601.841s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
 cookie=0x0, duration=1945.103s, table=0, n_packets=0, n_bytes=0, in_port="sw1-eth1" actions=set_queue:123,output:"sw1-eth3"
 cookie=0x0, duration=1945.089s, table=0, n_packets=0, n_bytes=0, in_port="sw1-eth3" actions=set_queue:123,output:"sw1-eth1"
 cookie=0x0, duration=1945.071s, table=0, n_packets=0, n_bytes=0, in_port="sw1-eth4" actions=set_queue:123,output:"sw1-eth1"
 cookie=0x0, duration=1945.054s, table=0, n_packets=0, n_bytes=0, in_port="sw1-eth2" actions=set_queue:234,output:"sw1-eth5"
 cookie=0x0, duration=1945.040s, table=0, n_packets=0, n_bytes=0, in_port="sw1-eth5" actions=set_queue:234,output:"sw1-eth2"
 cookie=0x0, duration=1945.026s, table=0, n_packets=0, n_bytes=0, in_port="sw1-eth6" actions=set_queue:234,output:"sw1-eth2"
```

Figure 4.54: Recheck slices and flows

Chapter 5

Known Issues

The known issues faced during the project development are listed below:

- If something went wrong running the installation script *scripts/init.sh* on Windows OS as described in the chapter 1, the Virtual Box window should be kept in evidence by the mouse.
- The files for the RYU Graphical User Interface miss, installing RYU from pip. So, they are downloaded manually as shown in the figure 1.8 in the chapter 1.
- FlowVisor only implements the OpenFlow protocol version 1.0, which is the oldest one. Moreover, it does not allow to define directly the OpenFlow resulting action. It needs to be restarted to update any slice or flow and the built-in update command does not work as expected.
- RYU and Mininet documentation are not always clear. Indeed, it has been put a lot of effort to deploy the last RYU controller described in the scenario 3 at 4.5 in the chapter 4.

Conclusions

The project has been developed to be cross platform and run on any operating system. On Windows might be longer the process, but it is just a matter of downloading a plugin and configuring correctly the environment.
Then, a new development workflow has been proposed in the Comnetsemu environment. It is very interesting, because it allows you to develop the code and immediately run it in the Comnetsemu virtual machine. So, the developer does not need to take care of copying and pasting every time the code inside the Comnetsemu virtual machine.
The core of the project is the new topology building system described in the chapter 3. It is the innovation introduced by the project that can help to create Mininet networks in Comnetsemu easier. Furthermore, the network topology files are more readable and it is simple to update and debug them.
At the end, different scenarios have been tested and results have been reported. The first scenario is the most complicated in terms of service migration and OpenFlow protocol, because FlowVisor is deployed by a new Docker container. It might be difficult to develop it, because it needs to create the communication between the FlowVisor Docker container and the RYU controller. Instead, the third scenario is the best option in terms of service migration, because it only relies on the RYU controller to perform the service migration. Moreover, slices and flows are handled by ovs-ofctl and there is no need to deploy a new Docker container outside the Mininet network as for FlowVisor. So, the RYU controller can modify directly slices and flows without communicating with a Docker container in a new virtualization level. The second scenario is the most simple one. It is shown to introduce ovs-ofctl to lead at the final best scenario 3.

Bibliography

[1] *CD Workflow*. URL: https://docs.github.com/en/actions/deployment/about-deployments/about-continuous-deployment.

[2] *CI Workflow*. URL: https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration.

[3] *Comnetsemu Repository*. URL: https://git.comnets.net/public-repo/comnetsemu.

[4] *Containernet*. URL: https://containernet.github.io/.

[5] *Dependabot*. URL: https://docs.github.com/en/code-security/dependabot.

[6] *Docker*. URL: https://www.docker.com/.

[7] *Flask*. URL: https://flask.palletsprojects.com/en/2.1.x.

[8] *FlowVisor*. URL: https://github.com/OPENNETWORKINGLAB/flowvisor.

[9] *GitHub Workflows*. URL: https://docs.github.com/en/github-ae@latest/actions/using-workflows/about-workflows.

[10] *inotifywait*. URL: https://linux.die.net/man/1/inotifywait.

[11] *Kubernetes*. URL: https://kubernetes.io/.

[12] *Mininet*. URL: http://mininet.org/.

[13] *OpenFlow Protocol*. URL: https://en.wikipedia.org/wiki/OpenFlow.

[14] *ovs-ofctl*. URL: https://www.man7.org/linux/man-pages/man8/ovs-ofctl.8.html.

[15] *Pre-commit*. URL: https://pre-commit.com/.

[16] *PySerde*. URL: https://github.com/yukinarit/pyserde.

[17] *rsync*. URL: https://linux.die.net/man/1/rsync.

[18] *RYU Controller Documentation*. URL: https://ryu.readthedocs.io/en/latest/.

List of Figures