UNIVERSITÀ DI PISA

DEPARTMENT OF INFORMATION ENGINEERING

MSc in Computer Engineering

Advanced Computer Networking Project

Alessio Simoncini, Lorenzo Vezzani

ACADEMIC YEAR 2025/2026

Contents

Chapter 1

Introduction

1.1 Aims and scope of the documentation

This work has been developed as part of the *Advanced Computer Networking* course project, academic year 2025/2026.

1.2 Project description

The project consists in the design and implementation of a network composed by a hierarchy of Autonomous Systems (AS). The purposes of the project are using templates for network provisioning, and the study of the BGP protocol, in particular eBGP, iBGP and the dynamic configuration of BGP attributes.

1.3 Objectives

Three main objectives are given in the project specifications:

1. **Configuration templates** — Develop reusable configuration templates for all routers in the network using Jinja2 scripting. These templates should support parameterization for device-specific details such as AS numbers, router IDs and interface addresses and BGP neighbor relationships.

2. **Network emulation with Containerlab** — Use Containerlab to instantiate the described network topology. Each router should be based on FRR, end nodes will be based on Alpine.

3. **Network Automation System for AS 65020** — Design and implement a network automation system that operates within a Manager node in AS65020, which should receive traffic prediction, determine optimal traffic distribution across upstream links and adjust BGP attributes influence inbound and outbound traffic paths.

The basic structure of the network to emulate is shown and discussed in the following chapter.

1.4 Environment

As requested by project specification, the environment used for carrying out the network emulation is ContainerLab. Containerlab is an open-source tool used to create virtual networking labs based on container technology. It allows users to define network topologies through YAML files; network devices, including routers, switches, and hosts, are then emulated as containers.

1.5 Code repository

The code developed for the project can be found in the repository at:
`https://github.com/AlessioSmn/Network-Provisioning-Automation` .

Chapter 2

Network structure

2.1 Network overview

2.1.1 Base network

The network is composed by a central AS (AS65020), which will be the focus of later tasks, two customer ASes, two upstream ASes and the Internet. The basic structure of the network to implement is represented in Figure 2.1:

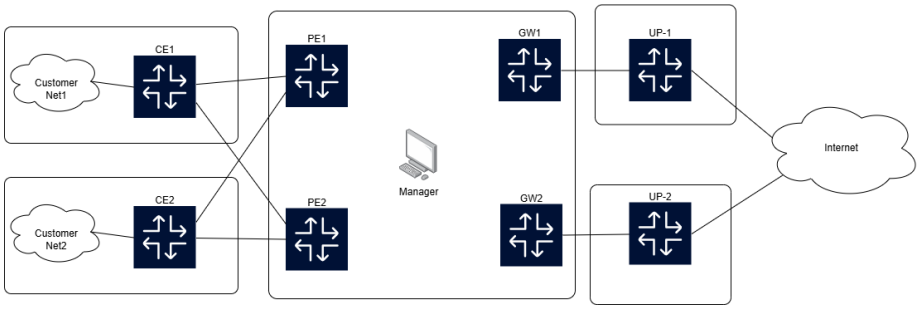

Figure 2.1: Starting network

2.1.2 Developed network

The network developed for the project expands the base network by specifying the structure of the various networks: customer networks, the core network and the Internet. All aspects are described in detail, along with more precise images, in the following section. The structure of the implemented network is represent in Figure 2.2, along with interface names and addresses:

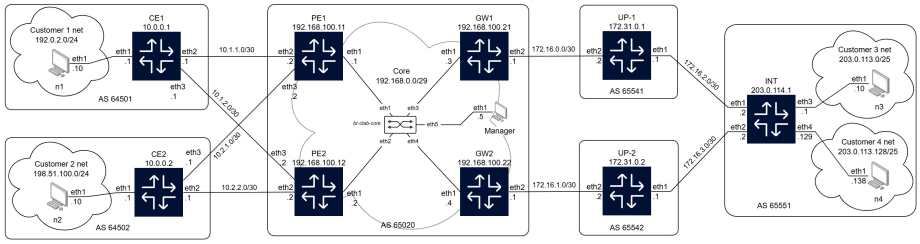

Figure 2.2: Starting network

Management network

When deploying the network, Containerlab, based on Docker, creates all nodes as containers and automatically configures a management interface (`eth0`), assigns it an IP address, and installs a default route through that interface. To the best of our knowledge, there is no supported way to prevent the creation of this interface and its associated default route. For this reason, we decided to explicitly use it as the management interface.

Management addresses The management network prefix can be configured in the topology file, as shown in Listing 2.1:

```
1 mgmt:
2   network: acn_prj_mgmt              # name
3   ipv4-subnet: 10.255.255.0/24       # ipv4 range
```

Listing 2.1: Management network

By default, Docker assigns management IP addresses dynamically. However, fixed IP addresses can be explicitly specified for each node, as shown in Listing 2.2. This configuration is particularly useful for debugging purposes, as it allows consistent addressing across multiple deployments.

```
1     PE1:
2       group: frr_nodes
3       mgmt-ipv4: 10.255.255.111
```

Listing 2.2: Management IP address

Management VRF Another possible source of problem is the default route towards the management interface, which is installed as a route of type `kernel`, and thus cannot be removed.
To avoid unintended traffic being forwarded toward the management network, we configured the startup scripts of all nodes to optionally move the management interface into a dedicated and isolated Virtual Routing and Forwarding instance (VRF). To allow us to avoid that default route and consequent traffic towards the

management network, we configured the startup files of all nodes to be able to move the management interface into a dedicated and separated Virtual Routing Function (VRF). If the environment variable `CLAB_MGMT_VRF` is specified in the topology file, as represented in Listing **??**,

```
topology:
  groups:
    frr_nodes: # applied to all nodes of the group
      env:
        CLAB_MGMT_VRF: MGMT
        CLAB_MGMT_GW: 10.255.255.1
```

Listing 2.3: Management VRF in .clab.yml

the startup script receives this environment variable and, when defined, creates the management VRF, moves the `eth0` interface into it and moves the default route, as represented in Listing 2.4:

```
# VRF Management (if specified)
if [ -n "${CLAB_MGMT_VRF}" ]; then

    # VRF creation
    ip link add "${CLAB_MGMT_VRF}" type vrf table 100
    ip link set "${CLAB_MGMT_VRF}" up

    # Interface assigned to VRF
    ip link set eth0 master "${CLAB_MGMT_VRF}"

    # Default route moved
    ip route add default via ${CLAB_MGMT_GW} vrf "${CLAB_MGMT_VRF}"
fi
```

Listing 2.4: Management VRF in .clab.yml

In the following table the management network IP addresses of each node are written down:

Node	Address
CE1	10.255.255.101
CE2	10.255.255.102
PE1	10.255.255.111
PE2	10.255.255.112
GW1	10.255.255.121
GW2	10.255.255.122
UP1	10.255.255.131
UP2	10.255.255.132
INT	10.255.255.141
n1	10.255.255.201
n2	10.255.255.202
n3	10.255.255.203
n4	10.255.255.204
MNGR	10.255.255.205

Table 2.1: Management IP addresses

2.2 Node images

Three types of nodes are present in the network: `FRR` routers, `Alpine` hosts and a single `Manager` node. All nodes are based on a custom-built images in order to at least have the OpenSSH package: all nodes are configured to be accessible via SSH via their management IP address, by using:

- username: `root`
- password: `admin`

2.2.1 FRR nodes

Based on FRRouting image, or FRR, version `10.4.1`. OpenSSH is added, username and password are set, and keys are generated. The final image is build by the `/shell/images.sh` script and named `frr-ssh:10.4.1`.
The FRR dockerfile is found at `./config/frr/dockerfile`.

2.2.2 Alpine nodes

Based on an Alpine image, version `3.19.1`. OpenSSH is added, username and password are set, and keys are generated. The final image is build by the `/shell/images.sh` script and named `alpine-ssh:3.19.1`.
The FRR dockerfile is found at `./config/alpine/dockerfile`.

2.2.3 Manager node

Based on an Alpine image, version `3.19.1`. Most importantly OpenSSH and Paramiko are added: username and password are set, and keys are generated to enable SSH access, while Paramiko is needed to be able to execute Python scripts on the node. The final image is build by the `/shell/images.sh` script and named `python-ssh:3.12-alpine`.
The FRR dockerfile is found at `./config/mngr/dockerfile`.

2.3 Network details

2.3.1 Customer side

Each customer network in the base structure has been implemented with a simple point to point link between the Customer Edge router (CE1 / CE2) and the relative end host (n1 / n2).

Customer 1 addresses

The customer 1 network has been given the **ASN 64501**.
The customer 1 part of the network is shown in detail in Figure 2.3:

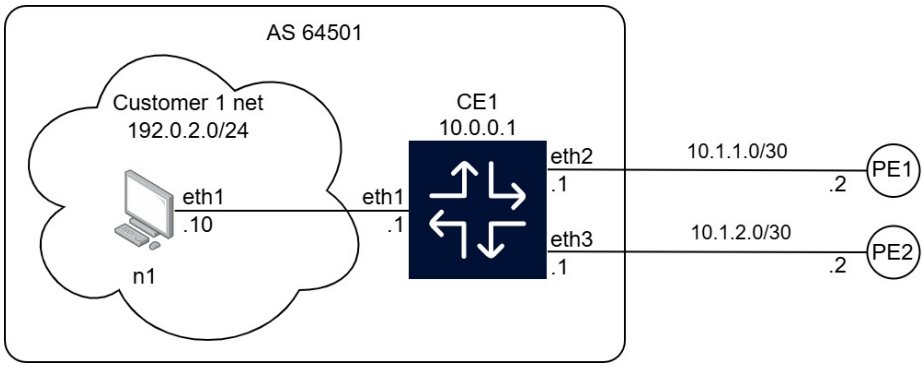

Figure 2.3: Network - customer 1

- `CE1 loopback address`: 10.0.0.1/32 (lo)
- `n1 - CE1 P2P link`: 192.0.2.0/24
  - `n1`: 192.0.2.10 (eth1)
  - `CE1`: 192.0.2.1 (eth1)
- `CE1 - PE1 P2P link`: 10.1.1.0/30
  - `CE1`: 10.1.1.1 (eth2)
  - `PE1`: 10.1.1.2 (eth2)

- `CE1 - PE2 P2P link`: 10.1.2.0/30
  - `CE1`: 10.1.2.1 (eth3)
  - `PE2`: 10.1.2.2 (eth3)

Customer 2 addresses

The customer 2 network has been given the **ASN 64502**.
The customer 2 part of the network is shown in detail in Figure 2.4:

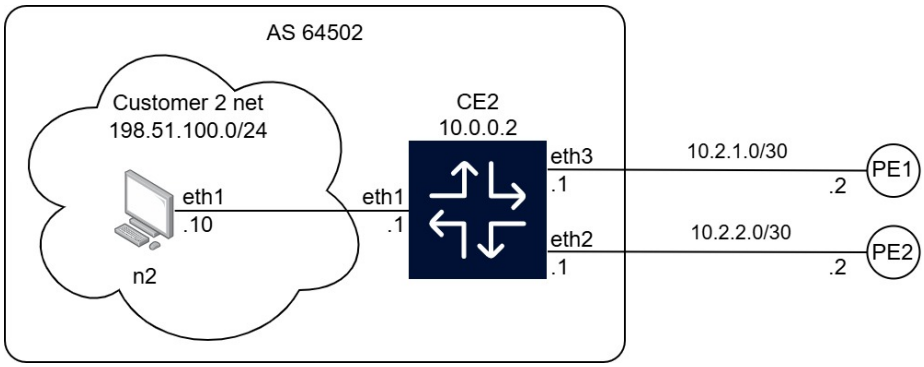

Figure 2.4: Network - customer 2

- `CE2 loopback address`: 10.0.0.2/32 (lo)
- `n2 - CE2 P2P link`: 198.51.100.0/24
  - `n2`: 198.51.100.10 (eth1)
  - `CE2`: 198.51.100.1 (eth1)
- `CE2 - PE1 P2P link`: 10.2.1.0/30
  - `CE2`: 10.2.1.1 (eth3)
  - `PE1`: 10.2.1.2 (eth3)
- `CE2 - PE2 P2P link`: 10.2.2.0/30
  - `CE2`: 10.2.2.1 (eth2)
  - `PE2`: 10.2.2.2 (eth2)

2.3.2 Core part

The core of the AS 65020 is implemented as a single LAN, using a switch called `br-clab-core` in the topology file; the LAN IP prefix is `192.168.0.0/29`. The central part of the network is shown in detail in Figure 2.5:

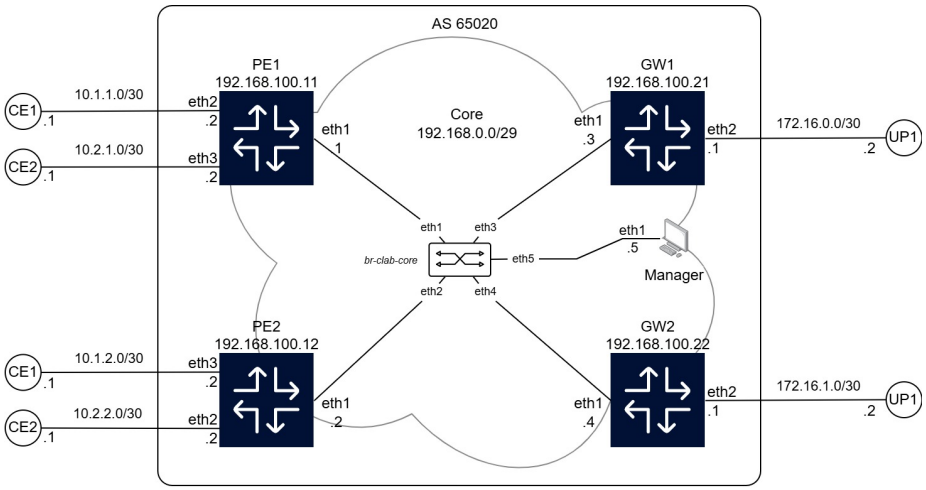

Figure 2.5: Network - central part

Core addresses

- `Core LAN`: 192.168.0.0/29
  - `PE1`: 192.168.0.1 (eth1)
  - `PE2`: 192.168.0.2 (eth1)
  - `GW1`: 192.168.0.3 (eth1)
  - `GW2`: 192.168.0.4 (eth1)
  - `MNGR`: 192.168.0.5 (eth1)

PE1

- `PE1 loopback address`: 192.168.100.11/32 (lo)
- `CE1 - PE1 P2P link`: 10.1.1.0/30
  - `CE1`: 10.1.1.1 (eth2)
  - `PE1`: 10.1.1.2 (eth2)
- `CE2 - PE1 P2P link`: 10.2.1.0/30
  - `CE2`: 10.2.1.1 (eth3)
  - `PE1`: 10.2.1.2 (eth3)

PE2

- `PE2 loopback address`: 192.168.100.12/32 (lo)
- `CE1 - PE2 P2P link`: 10.1.1.0/30
  - `CE1`: 10.1.2.1 (eth3)
  - `PE2`: 10.1.2.2 (eth3)
- `CE2 - PE2 P2P link`: 10.2.1.0/30
  - `CE2`: 10.2.2.1 (eth2)
  - `PE2`: 10.2.2.2 (eth2)

GW1

- `GW1 loopback address`: 192.168.100.21/32 (lo)
- `GW1 - UP1 P2P link`: 172.16.0.0/30
  - `GW1`: 172.16.0.1 (eth2)
  - `UP1`: 172.16.0.2 (eth2)

GW2

- `GW2 loopback address`: 192.168.100.21/32 (lo)
- `GW2 - UP2 P2P link`: 172.16.1.0/30
  - `GW2`: 172.16.1.1 (eth2)
  - `UP2`: 172.16.1.2 (eth2)

2.3.3 Upstream side

The internet network represented in the base network is implemented via a router, called `INT`, connected to both upstream routers and then connected to two hosts (n3 and n4): these addiotional networks and hosts have been created to be able to properly test load balancing on upstream links, by having many possible destination networks.
The upstream 1 network (router) has been given the **ASN 65541**; The upstream 2 network (router) has been given the **ASN 65542**; The Internet networks have been given the **ASN 65551**.
The upstream side of the network is shown in detail in Figure 2.6:

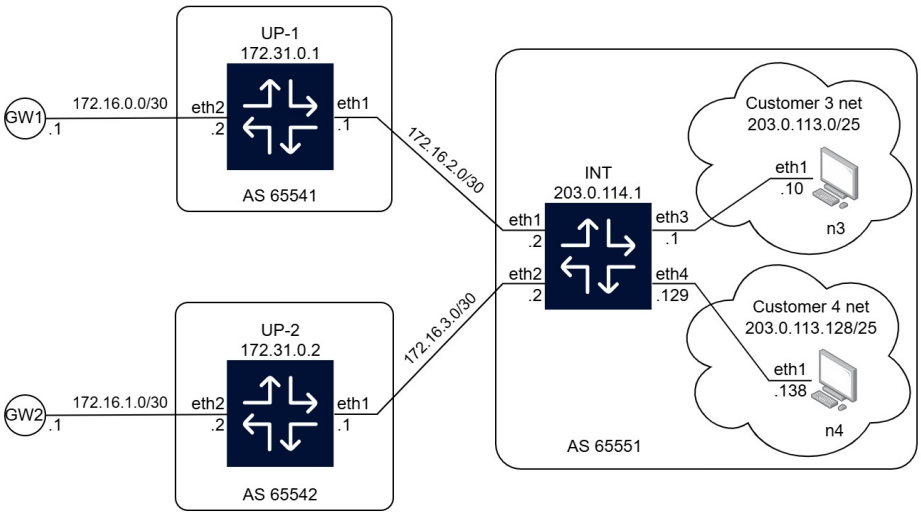

Figure 2.6: Network - upstream side

Upstreams

- `UP1 loopback address`: 172.31.0.1/32 (lo)
- `UP2 loopback address`: 172.31.0.2/32 (lo)
- `GW1 - UP1 P2P link`: 172.16.0.0/30
  - `GW1`: 172.16.0.1 (eth2)
  - `UP1`: 172.16.0.2 (eth2)
- `GW2 - UP2 P2P link`: 172.16.1.0/30
  - `GW2`: 172.16.1.1 (eth2)
  - `UP2`: 172.16.1.2 (eth2)
- `UP1 - INT P2P link`: 172.16.2.0/30
  - `UP1`: 172.16.2.1 (eth1)
  - `INT`: 172.16.2.2 (eth1)
- `UP2 - INT P2P link`: 172.16.3.0/30
  - `UP2`: 172.16.3.1 (eth1)
  - `INT`: 172.16.3.2 (eth2)

Internet

- `INT loopback address`: 203.0.114.1/32 (lo)
- `INT - n3 P2P link`: 203.0.113.0/25
  - `INT`: 203.0.113.1 (eth3)
  - `n3`: 203.0.113.10 (eth1)
- `INT - n4 P2P link`: 203.0.113.128/25
  - `INT`: 203.0.113.129 (eth4)
  - `n4`: 203.0.113.138 (eth1)

Chapter 3

Implementation

3.1 Containerlab topology file

The topology file is named `acn.clab.yml`, and it's located in the root of the project directory (`./acn.clab.yml`).

```
topology:

  groups:
    frr_nodes:
      # ...
    alp_nodes:
      # ...
    pyt_nodes:
      # ...
    bridges:
      kind: bridge

  nodes:
    # All previously described nodes
    # ...
    PE1:
      group: frr_nodes
      mgmt-ipv4: 10.255.255.111
    # ...

  links:
    # List of all links
```

Listing 3.1: acn.clab.yml structure

The list of links present in the topology file implements the connections described in section 2.3.

3.1.1 Node types

To implement the three types of nodes discussed in previous sections (FRR, Alpine and Alpine-Python) three node types have been defined in the network

topology: the keyword `groups` allows us to define a set of attributes to then be applied to any number of nodes, thus avoiding having to repeat the same exact configuration for each node. This is used in conjunction with the keyword `__clabNodeName__`, that is substituted at compile-time with the actual node name (CE1, PE1, etc.).

FRR node

The node is based on the image `frr-ssh:10.4.1`, discussed in section 2.2.1. Some file are needed for an FRR node:

- `frr.conf` — Startup configuration file
- `daemons` — Daemon file, used at startup to setup particular daemons (bgpd, ospfd among others)
- `vtysh.conf` — To enable vtysh
- `frr-cfg.sh` — To start the main FRR process, start SSH and possibly setup the management VRF

The node configuration within the topology file is represented in Listing 3.2:

```
1   frr_nodes:
2     env:
3       CLAB_MGMT_VRF: MGMT
4       CLAB_MGMT_GW: 10.255.255.1
5     kind: linux
6     image: frr-ssh:10.4.1
7     binds:
8       - ./config/startup/__clabNodeName__.conf:/etc/frr/frr.conf
9       - ./config/frr/daemons:/etc/frr/daemons
10      - ./config/frr/vtysh.conf:/etc/frr/vtysh.conf
11      - ./config/frr/frr-cfg.sh:/frr-cfg.sh
12    exec:
13      - bash /frr-cfg.sh
```

Listing 3.2: FRR node definition

Alpine node

The node is based on the image `alpine-ssh:3.19.1`, discussed in section 2.2.2. Some file are needed for an Alpine node:

- `conf.sh` — Startup configuration file
- `alp-cfg.sh` — Start SSH and possibly setup the management VRF

The node configuration within the topology file is represented in Listing 3.3:

```
1     alp_nodes:
2       kind: linux
3       image: alpine-ssh:3.19.1
4       binds:
5         - ./config/alpine/alp-cfg.sh:/alp-cfg.sh
6         - ./config/startup/__clabNodeName__.sh:/conf.sh
7       exec:
8         - sh /alp-cfg.sh
```

Listing 3.3: Alpine node definition

Alpine-Python node

The node is based on the image `python-ssh:3.12-alpine`, discussed in section 2.2.3. Some file are needed for the manager node:

- `conf.sh` — Startup configuration file
- `alp-cfg.sh` — Start SSH and possibly setup the management VRF
- `main.py` — Python file which implements the functionalities required in Task 3.
- `traffic-matrices.json` — Traffic matrix

The node configuration within the topology file is represented in Listing 3.4:

```
1     pyt_nodes:
2       kind: linux
3       image: python-ssh:3.12-alpine
4       binds:
5         - ./config/alpine/alp-cfg.sh:/alp-cfg.sh
6         - ./config/startup/__clabNodeName__.sh:/conf.sh
7         - ./config/mngr/main.py:/main.py
8         - ./config/mngr/traffic-matrices.json:/traffic-matrices.
              json
9       exec:
10        - sh /alp-cfg.sh
```

Listing 3.4: Manager node definition

3.2 Node configurations

3.2.1 FRR nodes

All FRR nodes interfaces have been configured as previously described in sections under 2.3.

OSPF The Open Shortest Path First (OSPF) routing protocol has been configured in the central AS (AS 65020): this has been done in order to have all nodes share their loopback addresses with each other, which in turn is needed for establishing **iBGP** sessions using loopback interfaces.
OSPF is configured in single-area mode. Each router (PE1, PE2, GW1, GW2) participates in the core network and advertises its loopback interface, which is configured as a passive interface.

BGP All routers have been configured with BGP; For `router-id` the loopback addresses have been used. For the BGP speakers of the AS 65020 a full-mesh topology has been setup. All the BGP relations are represented in the following table:

Type	Node 1	Node 1 int.	Node 2	Node 2 int.
eBGP	CE1	10.1.1.1	PE1	10.1.1.2
eBGP	CE1	10.1.2.1	PE2	10.1.2.2
eBGP	CE2	10.2.1.1	PE1	10.2.1.2
eBGP	CE2	10.2.2.1	PE2	10.2.2.2
iBGP	PE1	192.168.100.11	PE2	192.168.100.12
iBGP	PE1	192.168.100.11	GW1	192.168.100.21
iBGP	PE1	192.168.100.11	GW2	192.168.100.22
iBGP	PE2	192.168.100.12	GW1	192.168.100.21
iBGP	PE2	192.168.100.12	GW2	192.168.100.22
iBGP	GW1	192.168.100.21	GW2	192.168.100.22
eBGP	GW1	172.16.0.1	UP1	172.16.0.2
eBGP	GW2	172.16.1.1	UP2	172.16.1.2
eBGP	UP1	172.16.2.1	INT	172.16.2.2
eBGP	UP2	172.16.3.1	INT	172.16.3.2

Table 3.1: BGP relations

3.2.2 Alpine nodes

All Alpine nodes interfaces have been configured as previously described in sections under 2.3. The only additional configuration consists of a default route on the customer hosts (n1, n2, n3 and n4), directed toward their respective routers (CE1, CE2 and INT), which consequently operate as default gateways.

3.3 Templates

As required by Task 1 of the project specification, all configuration files are generated with templates. Due to the presence of different node types, each characterized by a different configuration syntax, two distinct templates have been defined: one for FRR-based routers and one for Alpine nodes.

The configuration generation process is based on a single Python script responsible for rendering the templates using the provided input data. The configuration data are stored in the directory `./template/data/*.yaml`, where each file is named according to the corresponding node (i.e., `<node-name>.yaml`). Each data file explicitly specifies the template to be used through the field:

```
type: frr | alpine
```

During execution, the Python script reads this field and automatically selects the appropriate template file, rendering it with the associated data to generate the final device configuration. The content of the Python script (`./template/generator.py`) is shown in Listing 3.5

```
TEMPLATE_FRR = 'template_frr.j2'
TEMPLATE_ALP = 'template_alp.j2'
BASE_DIR = Path(__file__).resolve().parent

# === Load data
data_filename = sys.argv[1]
with open(data_filename) as file:
    data = yaml.safe_load(file)

# Default type: FRR
cont_type = 'FRR'
if 'type' in data:
    if data['type'].lower() == 'alpine':
        cont_type = 'alpine'

# === Load template
environment = Environment(
    loader=FileSystemLoader(str(BASE_DIR)),
    trim_blocks=True,
    lstrip_blocks=True
)
template = None
if cont_type == 'FRR':
    template = environment.get_template(TEMPLATE_FRR)
elif cont_type == 'alpine':
    template = environment.get_template(TEMPLATE_ALP)

# === Generate content
content = template.render(data)

# === Print and save to file

output_dir = BASE_DIR / ".." / "config" / "startup"
output_dir = output_dir.resolve()

config_filename = data['config_filename']
output_path = output_dir / config_filename
output_dir.mkdir(parents=True, exist_ok=True)

with open(output_path, "w") as f:
    f.write(content)
```

Listing 3.5: Template processing

Note that some checks and debug lines have been omitted from this Listing.

Complete automation the previously described process creates the configuration file for a given node: this process is easily repeated on all nodes via a simple shell script (`./shell/template.sh`), as represented in Listing 3.6:

```
# Generate startup-config files
for file in template/data/*.yaml; do
    python3 template/generator.py "$file"
    echo "$file" processed
done
```

Listing 3.6: template.sh file

The FRR template is discussed in the following three subsections (*Interfaces, OSPF, BGP*), while the Alpine template is discussed in the last subsection (*Alpine template*).

3.3.1 FRR template

Interfaces

The template section dealing with interfaces loops over a list of interfaces, which must be called `interfaces`, and for each interface sets:

- IP address — field `ipv4`
- description — field `description`
- working layer — field `switchport` set to `False` for L3, set to `True` for L2

and finally activates it. The part of the template dealing with interfaces is represent in Listing 3.7:

```
{# ===Interfaces === -#}
{% if interfaces -%}
{% for int in interfaces -%}
interface {{int.name}}
    {% if int.ipv4 %}
    ip address {{int.ipv4}}
    {% endif -%}
    {% if int.description %}
    description {{int.description}}
    {% endif %}
    {% if int.switchport == False %}
    no switchport
    {% endif -%}
    no shutdown
!
{% endfor -%}
{% endif -%}
```

Listing 3.7: FRR template - interfaces

An example of a parameters file to be passed, compliant with the previously described part of template is represented in the following Listing 3.8:

```
interfaces:
- name: lo
  ipv4: 192.168.100.11/32
  switchport: False
  description: LOOPBACK
- name: eth1
  ipv4: 192.168.0.1/29
  switchport: False
  description: to_CORE
```

Listing 3.8: Interfaces parameters

The resulting part of the .conf file is represented in the following Listing 3.9:

```
interface lo
    ip address 192.168.100.11/32
    description LOOPBACK
    no switchport
    no shutdown
!
interface eth1
    ip address 192.168.0.1/29
    description to_CORE
    no switchport
    no shutdown
!
```

Listing 3.9: Configuration example

OSPF

The template section dealing with OSPF firstly checks if there's a section dedicated to OSPF (field `ospf`): if that field is not present OSPF is not configured. If instead the `ospf` field is present, the template configures the router-id (field `router_id`), then loops over a list of networks (field `networks`) and configures the advertising of those network prefixes, along with their relative area.
Then the template loops over a list of passive interfaces (field `passive_interfaces`) and marks them as passive, with regard to OSPF. The part of the template dealing with OSPF is represented in Listing 3.10:

```
{# === OSPF === -#}
{% if ospf -%}
router ospf
    router-id {{ospf.router_id}}
    {% for net in ospf.networks -%}
    network {{net.network}} area {{net.area}}
    {% endfor %}
!
{% for pi in ospf.passive_interfaces -%}
interface {{pi}}
    ip ospf passive
```

```
{% endfor -%}
{% endif -%}
```

Listing 3.10: FRR template - OSPF

An example of a parameters file to be passed, compliant with the previously described part of template is represented in the following Listing 3.11:

```
ospf:
  router_id: 192.168.100.11
  passive_interfaces: [lo, eth0, eth2, eth3]
  networks:
  - network: 192.168.0.0/29
    area: 0
  - network: 192.168.100.11/32
    area: 0
```

Listing 3.11: OSPF parameters

The resulting part of the .conf file is represented in the following Listing 3.12:

```
router ospf
    router-id 192.168.100.11
network 192.168.0.0/29 area 0
network 192.168.100.11/32 area 0
!
interface lo
    ip ospf passive
interface eth0
    ip ospf passive
interface eth2
    ip ospf passive
interface eth3
    ip ospf passive
!
```

Listing 3.12: OSPF Configuration example

BGP

The template section dealing with BGP firstly checks if there's a section dedicated to BGP (field `bgp`): if that field is not present BGP is not configured.
If instead the `bgp` field is present, the template configures the router-id (field `ID`), the current router's ASN (field `AS`) and configures BGP to install ECMP routes among different AS paths of the same length, via the command `bestpath as-path multipath-relax`. Also two other fields are needed for the neighbors configuration:

- `lo_iBGP` — indicates whether to use the loopback interface for iBGP sessions
- `lo_eBGP` — indicates whether to use the loopback interface for eBGP sessions

- `lo` — the loopback interface name

The template loops over a list of BGP neighbors (field `neighbors`) and for each one configures:

- Neighbor IP address — field neighbor.`ipv4`
- Neighbor ASN — if the field neighbor.`AS` is present, that value is used; otherwise it is intended as a **i**BGP session, thus setting the remote-as field to the same ASN as the current router.
- Interface used — if the loopback interface must be used for the current neighbor it is set with the `update-source` command.
- Next-hop modification — for all iBGP peerings the router is configured to overwrite the Next-Hop field by putting its address as Next-Hop, before sending the route advertisement to its iBGP peer; this is done with the `next-hop-self` command

The template then loops over a list of networks (field `networks`) and configure each one of them to be advertised via BGP updates. Finally, the `no bgp ebgp-requires-policy` rule is added: this allows us to let BGP run without needing all BGP updates to have a policy.
The part of the template dealing with BGP is represented in Listing 3.13:

```
{# === BGP === -#}
{% if bgp -%}
router bgp {{bgp.AS}}
    bgp router-id {{bgp.ID}}
    no bgp ebgp-requires-policy
    bgp bestpath as-path multipath-relax
{% for nb in bgp.neighbors -%}
    {% if nb.AS %}
    neighbor {{nb.ipv4}} remote-as {{nb.AS}}
    {% if bgp.lo_eBGP %}
    neighbor {{nb.ipv4}} update-source {{bgp.lo}}
    {% endif -%}
    {% else %}
    neighbor {{nb.ipv4}} remote-as {{bgp.AS}}
    neighbor {{nb.ipv4}} next-hop-self
    {% if bgp.lo_iBGP %}
    neighbor {{nb.ipv4}} update-source {{bgp.lo}}
    {% endif -%}
    {% endif -%}
{% endfor -%}
{% for net in bgp.networks %}
    network {{net.prefix}} mask {{net.mask}}
{% endfor %}
!
{% for nb in bgp.neighbors -%}
clear ip bgp {{nb.ipv4}} soft in
clear ip bgp {{nb.ipv4}} soft out
{% endfor -%}
{% endif -%}
```

```
!
```

Listing 3.13: FRR template - BGP

The template also loops again over the list of neighbors (field `neighbors`) and clears all routes received and sent to each neighbor: this doesn't regard the actual BGP configuration, it is done to enforce a refreshing of all BGP routes in the system, in order to avoiding possible errors in the configuration phase.
An example of a parameters file to be passed, compliant with the previously described part of template is represented in the following Listing 3.14:

```
bgp:
  AS: 65020
  ID: 192.168.100.11
  lo_iBGP: True
  lo_eBGP: False
  lo: lo

  neighbors:
  - ipv4: 10.1.1.1
    AS: 64501
  - ipv4: 10.2.1.1
    AS: 64502
  - ipv4: 192.168.100.12 # No AS specified -> same AS
  - ipv4: 192.168.100.21
  - ipv4: 192.168.100.22
```

Listing 3.14: BGP parameters

The resulting part of the .conf file is represented in the following Listing 3.15:

```
router bgp 65020
    bgp router-id 192.168.100.11
    neighbor 10.1.1.1 remote-as 64501
    neighbor 10.2.1.1 remote-as 64502
    neighbor 192.168.100.12 remote-as 65020
    neighbor 192.168.100.12 next-hop-self
    neighbor 192.168.100.12 update-source lo
    neighbor 192.168.100.21 remote-as 65020
    neighbor 192.168.100.21 next-hop-self
    neighbor 192.168.100.21 update-source lo
    neighbor 192.168.100.22 remote-as 65020
    neighbor 192.168.100.22 next-hop-self
    neighbor 192.168.100.22 update-source lo
    no bgp ebgp-requires-policy
!
clear ip bgp 10.1.1.1 soft in
clear ip bgp 10.1.1.1 soft out
clear ip bgp 10.2.1.1 soft in
clear ip bgp 10.2.1.1 soft out
clear ip bgp 192.168.100.12 soft in
clear ip bgp 192.168.100.12 soft out
clear ip bgp 192.168.100.21 soft in
clear ip bgp 192.168.100.21 soft out
clear ip bgp 192.168.100.22 soft in
clear ip bgp 192.168.100.22 soft out
```

```
!
```

Listing 3.15: BGP configuration example

3.3.2 Alpine template

The template designed for Alpine nodes configures three aspects:

- Interfaces — field `interfaces` — Given a list of interfaces, the IP address (`ipv4`) is configured for each one (`name`)
- Static route — field `ipv4_routes` — All specified routes are added; both the destination (`destination`) and next-hop field are requested (`next_hop`)
- Python — field `python` — Designed for the manager node. It requires a python filename (`file`), which is run in the background and its output is redirected to a log file (`log`); if specified (`messages`), some messages can be displayed via echo message

The template, once compiled, generates a shell file. The template used for Alpine nodes, both standard and Python-Alpine, is shown in Listing 3.16:

```
# === Interfaces
{% if interfaces -%}
{% for int in interfaces -%}
ip link set {{ int.name }} up
{% if int.ipv4 -%}
ip addr add {{ int.ipv4 }} dev {{ int.name }}
{% endif -%}
{% endfor -%}
{% endif %}

{% if ipv4_routes -%}
# === Static routes
{% for route in ipv4_routes -%}
ip route replace {{ route.destination }} via {{ route.next_hop }}
{% endfor -%}
{% endif %}

{% if python -%}
# === Python script
# Run it in background, redirect output to log file
nohup python3 -u {{python.file}} > {{python.log}} 2>&1 &
# Messages
{% for msg in python.msgs -%}
echo {{msg}}
{% endfor -%}
{% endif %}
```

Listing 3.16: Alpine template

An example of a parameters file to be passed, compliant with the previously described template is represented in the following Listing 3.17:

```yaml
type: alpine

interfaces:
- name: eth1
  switchport: False
  ipv4: 192.168.0.5/29

ipv4_routes:
- destination: 0.0.0.0/0
  next_hop: 192.0.2.1

python:
  file: /main.py
  log: /log_file.log
  msgs:
   - "Python code is running in the background"
   - "Output is being redirected to ./log_file.log"
   - "Type tail -f log_file.log to see the live output"
```

Listing 3.17: Alpine parameters

The resulting part of the .conf file is represented in the following Listing 3.18:

```
# === Interfaces
ip link set eth1 up
ip addr add 192.168.0.5/29 dev eth1

# === Static routing
ip route replace 0.0.0.0/0 via 192.0.2.1

# === Python file
# Run it in background, redirect output to log file
nohup python3 -u /main.py > /log_file.log 2>&1 &
# Messages
echo Python code is running in the background
echo Output is being redirected to ./log_file.log
echo Type tail -f log_file.log to see the live output
```

Listing 3.18: Alpine configuration example

3.4 Network automation

The entire network automation procedure is contained inside a python script called `main.py`. The Manager node executes this Python script passed to it with `bind` CONTAINERlab command. The script is executed in launched in background at the network deploy using bash script called `mngr.sh`.
Moreover, a file called `traffic-matrices.json` is passed to the manger as well. In this json file, the traffic estimation (Gb/s) for each route is included.

```
{ "from": "192.0.2.0/24", "to": "203.0.113.0/25", "traf": 12 },
{ "from": "198.51.100.0/24", "to": "203.0.113.0/25", "traf": 7 },
{ "from": "192.0.2.0/24", "to": "203.0.113.128/25", "traf": 5 },
{ "from": "198.51.100.0/24", "to": "203.0.113.128/25", "traf": 6 },
```

```
5 { "from": "203.0.113.0/25", "to": "192.0.2.0/24", "traf": 9 },
6 { "from": "203.0.113.128/25","to": "192.0.2.0/24", "traf": 3 },
7 { "from": "203.0.113.0/25", "to": "198.51.100.0/24", "traf": 6 },
8 { "from": "203.0.113.128/25", "to": "198.51.100.0/24", "traf": 10 }
```

Listing 3.19: A Traffic Matrix

In the Python script, there is an infinite loop. In every iteration, a new traffic matrix is obtained, and a simple greedy algorithm splits the traffic. The greedy algorithm exploits two buckets in order to determine an efficient traffic distribution across upstream links. The algorithm operates iteratively[1] according to the following steps:

1. Retrieve a route from the set of available routes.
2. Among the two buckets, select the one with the minimum amount of expected traffic.
3. Insert the selected route into the chosen bucket and update the bucket's total expected traffic by adding the traffic associated with the route.
4. If there are still unprocessed routes, return to step 1; otherwise, terminate the algorithm.

An additional constraint is imposed on the route allocation process. Consider the case in which a route from CU1 to the Internet is assigned to a given gateway (GW). If a subsequent route from CU2 to the Internet is processed, it must be served by the same gateway previously selected. In other words, it is not allowed to assign the route (CU1 → Internet) to GW1 and the route (CU2 → Internet) to GW2, unless additional mechanisms such as traffic engineering or segment routing are employed.
Once the algorithm has been executed, *local_pref* is exploited to balance the outbound traffic and a filtering method for the inbound traffic. In particular, bucket1 is used to determine which routes must have a 200 local-pref to GW1 (to-internet routes) and which routes must be filtered in GW2 (from-internet routes). The opposite goes for bucket2.
Note that we use route filtering instead of **MED** to balance inbound traffic, since the network topology does not allow MED to be applied effectively. In particular, MED is unsuitable for our scenario because it cannot discriminate among multiple entry points within the core AS. UP1 and UP2 each have only a single link toward the core (to GW1 and GW2, respectively); therefore, when traffic destined to a customer network reaches a given UP, there is only one possible path into the core. As a consequence, the only way to control which UP handles traffic for a given customer network is to select a single GW to announce that network. This ensures that only the corresponding UP advertises the route to the Internet.

[1] Between every iteration `time.sleep(180)` is called.

3.4.1 Local Preference Setting

The *Local Preference* is set using two route-maps, **SET-LP-GW1** and **SET-LP-GW2**, respectively applied on `GW1` and `GW2`. For each entry inside a bucket, an *IP prefix-list* is created, together with a new rule in the corresponding route-map. This rule is applied when a match with the previously defined prefix-list occurs and consists solely in setting the local preference value to 200. If the route map is created in GW1, this has to be applied to the routes received from UP1. The opposite goes for GW2 and UP2.

```
1  def configure_bgp_local_pref(gw_ip, route_out_map_name, bucket):
2
3      my_up = UP1_IP if gw_ip == GW1_IP else UP2_IP
4      my_up_as = UP1_AS if gw_ip == GW1_IP else UP2_AS
5
6      cmds = ['configure terminal']
7
8      seq = 10
9
10     for entry in bucket:
11         # from internet route, not interesting here
12         if(entry == CU1_NET or entry == CU2_NET):
13             continue
14
15         prefix = entry
16         pl_name = f'PL-{prefix.replace("/", "_")}'
17
18         cmds.append(f'ip prefix-list {pl_name} permit {prefix}')
19         cmds.append(f'route-map {route_out_map_name} permit {seq}')
20         cmds.append(f' match ip address prefix-list {pl_name}')
21         cmds.append(f' set local-preference 200')
22
23         seq += 10
24
25     # catch all preference
26     cmds.append(f'route-map {route_out_map_name} permit 999')
27
28     cmds.append(f'router bgp {MY_AS}')
29     cmds.append(f' neighbor {my_up} remote-as {my_up_as}')
30     cmds.append(f' neighbor {my_up} route-map {route_out_map_name}
           in')
31
32     cmds.append('end')
33
34     vtysh_cmd = "vtysh \\\n" + " \\\n".join([f'    -c "{c}"' for c
           in cmds])
```

Listing 3.20: Local Preference Route Map Creation

After the command building, this will be passed to the target gateway using `paramiko` library.

3.4.2 Filtering method setting

Similarly to local preference, filtering is set using 2 route-maps as well: **SET-FILTERING-GW1** and **SET-FILTERING-GW2**. After ip-prefix list is created, a new **deny** rule, matching with the ip-prefix, is defined. This route-map is applied to all iBGP neighbors of the interested gateway[2].

```
def configure_filtering(gw_ip, route_in_map_name, bucket):

    cmds = ['configure terminal']

    seq = 10

    for entry in bucket:
        # to internet route, not interesting here
        if not (entry == CU1_NET or entry == CU2_NET):
            continue

        # buiding filter using route map

        prefix = entry
        pl_name = f'PL-{prefix.replace("/", "_")}'

        cmds.append(f'ip prefix-list {pl_name} permit {prefix}')
        cmds.append(f'route-map {route_in_map_name} deny {seq}')
        cmds.append(f' match ip address prefix-list {pl_name}')

        seq += 10

    # catch-all route map
    cmds.append(f'route-map {route_in_map_name} permit 999')

    # appling the route map
    cmds.append(f' neighbor {my_up} remote-as {my_up_as}')
    cmds.append(f' neighbor {my_up} route-map {
        route_inbound_map_name} out')

    cmds.append('end')

    vtysh_cmd = "vtysh \\\n" + " \\\n".join([f' -c "{c}"' for c in
        cmds])
```

Listing 3.21: Filtering Route Map Creation

3.4.3 BGP Clearing

After route-maps have been created and therefore new rules have to be applied, it is important to clear BGP session with a `soft BGP clear`.

```
def clear_bgp(peer_ip, ssh):

    cmd = f
        "vtysh
```

[2]Note that in this case bucket1 is used to create deny rules on GW2, and bucket2 for GW1.

```
        -c 'clear ip bgp {peer_ip} soft in'
        -c 'clear ip bgp {peer_ip} soft out'"

    stdin, stdout, stderr = ssh.exec_command(cmd)
```

Listing 3.22: BGP clearing

3.4.4 Route Maps Clearing

In loop iteration *i*, iteration *i-1* route-map have to be removed to avoid conflicts between rules.

```
def remove_all_route_maps(ip):

    # ... ssh connection via paramiko ...

    cmd = "vtysh -c 'show route-map'"
    stdin, stdout, stderr = ssh.exec_command(cmd)
    output = stdout.read().decode()

    route_maps = set(
        re.findall(r"^route-map:\s+(\S+)", output, re.MULTILINE)
    )

    cmds = ['configure terminal']

    for rm in route_maps:
        cmds.append(f"no route-map {rm}")

    cmds.append("end")

    vtysh_cmd = "vtysh \\\n" + " \\\n".join([f' -c "{c}"' for c in
        cmds])
```

Listing 3.23: Rout-Map removing

Chapter 4

Testing

4.1 Network deploy

Here is the network deploy, showed via `containerlab graph` command.

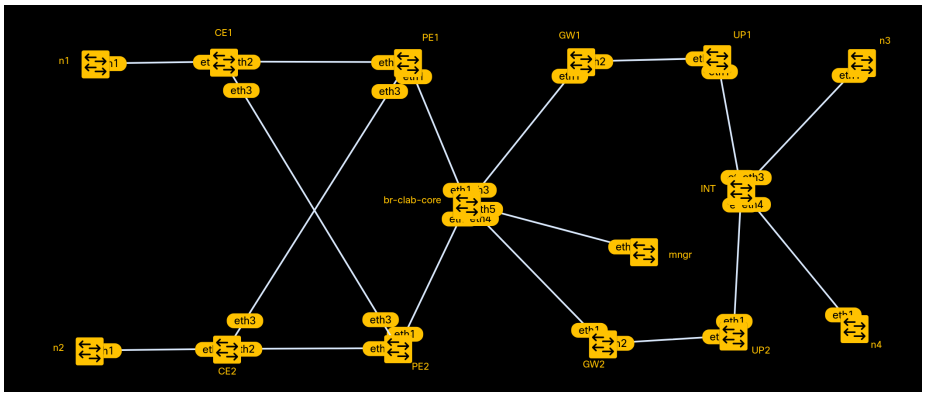

Figure 4.1: containerlab graph

4.2 Nodes configuration

In order to show the correct nodes' configuration, here is `show bgp summary` and `show ip route`, executed on various nodes of the network. A `ping` command is called as well, both from n1 and n2, respectively to n3 and n4, as a further confirmation of the correct configuration.

```
CE1# show ip bgp summary

IPv4 Unicast Summary:
BGP router identifier 10.0.0.1, local AS number 64501 VRF default vrf-id 0
BGP table version 15
RIB entries 7, using 896 bytes of memory
Peers 2, using 33 KiB of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
10.1.1.2        4      65020        44        40       15    0    0 00:26:09            3        4 N/A
10.1.2.2        4      65020        43        40       15    0    0 00:26:09            3        4 N/A

Total number of neighbors 2
```

Figure 4.2: CE1 - show ip bgp summary

```
PE1# show ip bgp summary

IPv4 Unicast Summary:
BGP router identifier 192.168.100.11, local AS number 65020 VRF default vrf-id 0
BGP table version 5
RIB entries 7, using 896 bytes of memory
Peers 5, using 83 KiB of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
10.1.1.1        4      64501         8         9        5    0    0 00:01:12            1        4 N/A
10.2.1.1        4      64502         8        10        5    0    0 00:01:13            1        4 N/A
192.168.100.12  4      65020         5         5        5    0    0 00:00:19            2        2 FRRouting/10.4.1_git
192.168.100.21  4      65020         4         5        5    0    0 00:00:29            1        2 FRRouting/10.4.1_git
192.168.100.22  4      65020         7         7        5    0    0 00:00:29            1        2 FRRouting/10.4.1_git

Total number of neighbors 5
```

Figure 4.3: PE1 - show ip bgp summary

```
GW1# show ip bgp summary

IPv4 Unicast Summary:
BGP router identifier 192.168.100.21, local AS number 65020 VRF default vrf-id 0
BGP table version 6
RIB entries 3, using 384 bytes of memory
Peers 4, using 66 KiB of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
172.16.0.2      4      65541        36        39        6    0    0 00:04:13            2        2 N/A
192.168.100.11  4      65020        20        17        6    0    0 00:03:30            0        2 FRRouting/10.4.1_git
192.168.100.12  4      65020        20        17        6    0    0 00:03:25            0        2 FRRouting/10.4.1_git
192.168.100.22  4      65020        24        23        6    0    0 00:03:30            0        2 FRRouting/10.4.1_git

Total number of neighbors 4
```

Figure 4.4: GW1 - show ip bgp summary

```
UP1# show ip bgp summary

IPv4 Unicast Summary:
BGP router identifier 172.31.0.1, local AS number 65541 VRF default vrf-id 0
BGP table version 36
RIB entries 7, using 896 bytes of memory
Peers 2, using 33 KiB of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
172.16.0.1      4      65020       179       167       36    0    0 00:24:46            1        4 N/A
172.16.2.2      4      65551        57        57       36    0    0 00:24:46            3        4 N/A

Total number of neighbors 2
```

Figure 4.5: UP1 - show ip bgp summary

```
lorenzo@Lorenzo-Virtual-Machine:~/Network-Provisioning-Automation$ docker exec -it acn-prj-n1 sh
/ # ping 203.0.113.10
PING 203.0.113.10 (203.0.113.10): 56 data bytes
64 bytes from 203.0.113.10: seq=0 ttl=59 time=0.186 ms
64 bytes from 203.0.113.10: seq=1 ttl=59 time=0.454 ms
64 bytes from 203.0.113.10: seq=2 ttl=59 time=0.281 ms
64 bytes from 203.0.113.10: seq=3 ttl=59 time=0.331 ms
```

Figure 4.6: ping - from n1, to n3

```
lorenzo@Lorenzo-Virtual-Machine:~/Network-Provisioning-Automation$ docker exec -it acn-prj-n2 sh
/ # ping 203.0.113.138
PING 203.0.113.138 (203.0.113.138): 56 data bytes
64 bytes from 203.0.113.138: seq=0 ttl=59 time=0.243 ms
64 bytes from 203.0.113.138: seq=1 ttl=59 time=0.308 ms
64 bytes from 203.0.113.138: seq=2 ttl=59 time=0.311 ms
64 bytes from 203.0.113.138: seq=3 ttl=59 time=0.307 ms
```

Figure 4.7: ping - from n2, to n4

```
PE1# show ip route
Codes: K - kernel route, C - connected, L - local, S - static,
       R - RIP, O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric, t - Table-Direct,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

IPv4 unicast VRF default:
C>* 10.1.1.0/30 is directly connected, eth2, weight 1, 00:01:36
L>* 10.1.1.2/32 is directly connected, eth2, weight 1, 00:01:36
C>* 10.2.1.0/30 is directly connected, eth3, weight 1, 00:01:36
L>* 10.2.1.2/32 is directly connected, eth3, weight 1, 00:01:36
B>* 192.0.2.0/24 [20/0] via 10.1.1.1, eth2, weight 1, 00:01:34
O   192.168.0.0/29 [110/10] is directly connected, eth1, weight 1, 00:00:51
C>* 192.168.0.0/29 is directly connected, eth1, weight 1, 00:01:36
L>* 192.168.0.1/32 is directly connected, eth1, weight 1, 00:01:36
O   192.168.100.11/32 [110/0] is directly connected, lo, weight 1, 00:01:36
L * 192.168.100.11/32 is directly connected, lo, weight 1, 00:01:36
C>* 192.168.100.11/32 is directly connected, lo, weight 1, 00:01:36
O>* 192.168.100.12/32 [110/10] via 192.168.0.2, eth1, weight 1, 00:00:51
O>* 192.168.100.21/32 [110/10] via 192.168.0.3, eth1, weight 1, 00:00:46
O>* 192.168.100.22/32 [110/10] via 192.168.0.4, eth1, weight 1, 00:00:51
B>* 198.51.100.0/24 [20/0] via 10.2.1.1, eth3, weight 1, 00:01:34
B>  203.0.113.0/25 [200/0] via 192.168.100.21 (recursive), weight 1, 00:00:44
  *                          via 192.168.0.3, eth1, weight 1, 00:00:44
B>  203.0.113.128/25 [200/0] via 192.168.100.21 (recursive), weight 1, 00:00:44
  *                            via 192.168.0.3, eth1, weight 1, 00:00:44
```

Figure 4.8: PE1 - show ip route

```
GW1# show ip route
Codes: K - kernel route, C - connected, L - local, S - static,
       R - RIP, O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric, t - Table-Direct,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

IPv4 unicast VRF default:
C>* 172.16.0.0/30 is directly connected, eth2, weight 1, 00:02:09
L>* 172.16.0.1/32 is directly connected, eth2, weight 1, 00:02:09
B>  192.0.2.0/24 [200/0] via 192.168.100.11 (recursive), weight 1, 00:01:17
  *                        via 192.168.0.1, eth1, weight 1, 00:01:17
                         via 192.168.100.12 (recursive), weight 1, 00:01:17
  *                        via 192.168.0.2, eth1, weight 1, 00:01:17
O   192.168.0.0/29 [110/10] is directly connected, eth1, weight 1, 00:01:19
C>* 192.168.0.0/29 is directly connected, eth1, weight 1, 00:02:09
L>* 192.168.0.3/32 is directly connected, eth1, weight 1, 00:02:09
O>* 192.168.100.11/32 [110/10] via 192.168.0.1, eth1, weight 1, 00:01:19
O>* 192.168.100.12/32 [110/10] via 192.168.0.2, eth1, weight 1, 00:01:19
O   192.168.100.21/32 [110/0] is directly connected, lo, weight 1, 00:02:09
L * 192.168.100.21/32 is directly connected, lo, weight 1, 00:02:09
C>* 192.168.100.21/32 is directly connected, lo, weight 1, 00:02:09
O>* 192.168.100.22/32 [110/10] via 192.168.0.4, eth1, weight 1, 00:01:19
B>  198.51.100.0/24 [200/0] via 192.168.100.11 (recursive), weight 1, 00:01:17
  *                           via 192.168.0.1, eth1, weight 1, 00:01:17
                            via 192.168.100.12 (recursive), weight 1, 00:01:17
  *                           via 192.168.0.2, eth1, weight 1, 00:01:17
B>* 203.0.113.0/25 [20/0] via 172.16.0.2, eth2, weight 1, 00:02:07
B>* 203.0.113.128/25 [20/0] via 172.16.0.2, eth2, weight 1, 00:02:07
```

Figure 4.9: GW1 - show ip route

```
CE1# show ip route
Codes: K - kernel route, C - connected, L - local, S - static,
       R - RIP, O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric, t - Table-Direct,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

IPv4 unicast VRF default:
L * 10.0.0.1/32 is directly connected, lo, weight 1, 00:01:30
C>* 10.0.0.1/32 is directly connected, lo, weight 1, 00:01:30
C>* 10.1.1.0/30 is directly connected, eth2, weight 1, 00:01:30
L>* 10.1.1.1/32 is directly connected, eth2, weight 1, 00:01:30
C>* 10.1.2.0/30 is directly connected, eth3, weight 1, 00:01:30
L>* 10.1.2.1/32 is directly connected, eth3, weight 1, 00:01:30
C>* 192.0.2.0/24 is directly connected, eth1, weight 1, 00:01:30
L>* 192.0.2.1/32 is directly connected, eth1, weight 1, 00:01:30
B>* 198.51.100.0/24 [20/0] via 10.1.1.2, eth2, weight 1, 00:01:28
  *                        via 10.1.2.2, eth3, weight 1, 00:01:28
B>* 203.0.113.0/25 [20/0] via 10.1.1.2, eth2, weight 1, 00:00:38
  *                       via 10.1.2.2, eth3, weight 1, 00:00:38
B>* 203.0.113.128/25 [20/0] via 10.1.1.2, eth2, weight 1, 00:00:38
  *                         via 10.1.2.2, eth3, weight 1, 00:00:38
```

Figure 4.10: CE1 - show ip route

```
UP1# show ip route
Codes: K - kernel route, C - connected, L - local, S - static,
       R - RIP, O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric, t - Table-Direct,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

IPv4 unicast VRF default:
C>* 172.16.0.0/30 is directly connected, eth2, weight 1, 00:01:47
L>* 172.16.0.2/32 is directly connected, eth2, weight 1, 00:01:47
C>* 172.16.2.0/24 is directly connected, eth1, weight 1, 00:01:47
L>* 172.16.2.1/32 is directly connected, eth1, weight 1, 00:01:47
L * 172.31.0.1/32 is directly connected, lo, weight 1, 00:01:47
C>* 172.31.0.1/32 is directly connected, lo, weight 1, 00:01:47
B>* 192.0.2.0/24 [20/0] via 172.16.0.1, eth2, weight 1, 00:00:55
B>* 198.51.100.0/24 [20/0] via 172.16.0.1, eth2, weight 1, 00:00:55
B>* 203.0.113.0/25 [20/0] via 172.16.2.2, eth1, weight 1, 00:01:45
B>* 203.0.113.128/25 [20/0] via 172.16.2.2, eth1, weight 1, 00:01:45
```

Figure 4.11: UP1 - show ip route

4.3 Automation testing

Network Legend

- **Customer Network 1 (CuNet1)**: 192.0.2.0/24
- **Customer Network 2 (CuNet2)**: 198.51.100.0/24
- **External Network 1 (ExtNet1)**: 203.0.113.0/25
- **External Network 2 (ExtNet2)**: 203.0.113.128/25

Traffic Matrix 1

From / To	CuNet1	CuNet2	ExtNet1	ExtNet2
CuNet1	–	–	12	5
CuNet2	–	–	7	6
ExtNet1	9	6	–	–
ExtNet2	3	10	–	–

Traffic Matrix 2

From / To	CuNet1	CuNet2	ExtNet1	ExtNet2
CuNet1	–	–	8	3
CuNet2	–	–	25	6
ExtNet1	5	4	–	–
ExtNet2	2	7	–	–

The greedy algorithm will provide the following result, w.r.t. the previous matrices:

Matrix 1 Buckets

- **Bucket 1**: ExtNet1, CuNet2
- **Bucket 2**: ExtNet2, CuNet1

Matrix 2 Buckets

- **Bucket 1**: ExtNet1
- **Bucket 2**: ExtNet2, CuNet1, CuNet2

Chapter 5

Conclusion

[TODO] Parte potenzialmente molto inutile
[TODO] Dire che il ciclo di prevelare matrice di traffico e applicare l'algoritmo è fatto ogni X minuti, ma è chiaramente a scopi dimostrativi di testing. Una implementazione più seria prendere i dati da stime sui dati di traffico + possibli previsioni su altre dati, e probabilmente sarebbe aggiornato solo se necessario e meno frequentemente.

Chapter 6

References / bibliography

[TODO] refrence to: general documents, papers, projects, short paper, websites, libraries used
[TODO] Se non ci sono, togliere

Appendix A

Templates

[TODO] Se vogliamo

Appendix B

Routing tables

[TODO] Se vogliamo

Appendix C

BGP configurations

[TODO] Se vogliamo