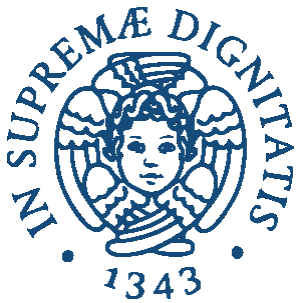

Università di Pisa

Master Of Science in Computer Engineering

Advanced Computer Networking

Automating traffic distribution via BGP

Project members:
Francesco De Lucchini
Antonio Andrea Salvalaggio

Academic Year 2025/2026

Contents

1 Introduction

1.1 Problem description

Consider the network in Figure 1, where:

- Rectangles represent autonomous systems (AS)
- All routers in AS 65020 belong to the same LAN, and so does the *Manager* node
- `CE`: Customer Edge, `PE`: Provider Edge, `GW`: GateWay, `UP`: UPstream

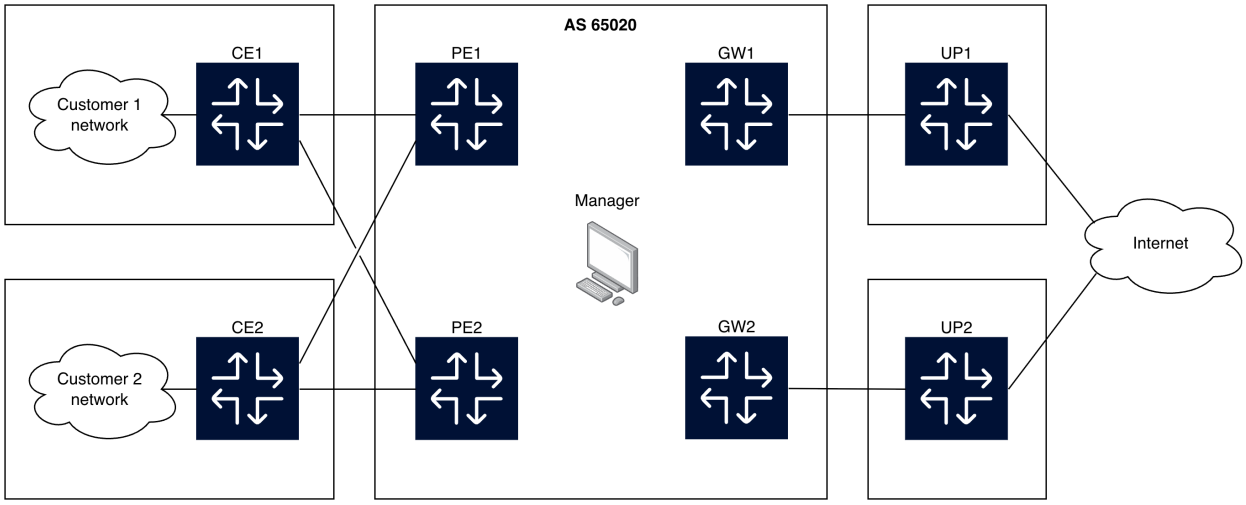

Figure 1: Schema of the network to implement

The project consists in:

1. Developing a Jinja2 reusable configuration template for the routers used in the network, parametrizing device-specific details such as AS numbers, router identifiers, interfaces, BGP neighbors and advertised networks. The routers should be based on FRR;
2. Implementing and testing the network described in Figure 1 in ContainerLab;
3. Designing a software, operating within the *Manager* node, that receives traffic predictions and utilizes BGP attributes (e.g., `MED`, `LOCAL_PREF`) to optimally distribute the traffic across upstream links.

2 Configuration template

To facilitate the creation of the configuration files for all the routers in the system, a python script has been developed. The script uses the Jinja2 package to populate a reusable FRR configuration template with the parameters provided through a separate YAML file.

2.1 The configuration generator script

The python script (`frr-config-generator/generator.py`) takes up to 3 arguments:

1. The location of the configuration data. It can be a single YAML file or a directory of YAML files, in which case all files will be processed one after the other. It defaults to `"./input/"`
2. The output location. It must be a directory and defaults to `"./output/"`
3. The Jinja2 template to be used. It must be a valid Jinja2 template file (`.j2`). It defaults to `"./template.j2"`

2.2 The Jinja2 template

The Jinja2 template (`frr-config-generator/template.j2`) parametrizes the configuration for a standard FRR router. Mainly, it allows to configure both the interfaces and BGP routing information.

The associated configuration data is provided through one or more YAML files with the following structure:

```
devices:
 - hostname: device1
   interfaces: ...
   bgp: ...
 - hostname: device2
   ...
```

There is a top-level `devices` key containing one entry for every router to configure.

The `hostname` key defines the device's hostname, and is used by the generation script to determine the filename for the output configuration (`<hostname>.conf`).

The `interfaces` key contains a list of interfaces each with the following structure:

- `name`: the interface name (e.g.: `eth0`);
- `ipv4_address`: the IPv4 address and mask associated with the interface (e.g.: `10.0.0.1/24`);
- `ipv6_address`: the IPv6 address and prefix associated with the interface.

The `setup_bgp_localpref_map` key is used to add the route-map needed by the load balancing system discussed in section 4.3.

The `bgp` key contains the following data:

- `as_number`: the device's own AS number (e.g.: `65000`);
- `disable_ebgp_requires_policy` and `disable_import_check`: `True` by default, they respectively add `no bgp ebgp-requires-policy` and `no bgp network import-check` to the configuration. The use of these parameters is discussed in section 3.1;
- `router_id`: the device's own router-id (e.g.: `192.168.1.1`);
- `neighbors`: a list of neighbors, each containing the `ip` and `remote_as` number of the neighbors. `next_hop_self` can be used to set the `next-hop-self` flag. `use_localpref_map` is used for the changes needed by the load balancing system discussed in section 4.3. The list can also contain peer groups by setting the following: `is_group` to `True`, `group_name`, `remote_as`, `next_hop_self` and `group_members` (a list of ips);
- `networks`: the list of network prefixes to advertise in the form ip/mask.

The file `frr-config-generator/input/settings.yaml` contains the configuration data used for all the routers of the discussed network. The configuration data for the `PE1` router is provided as an example:

```yaml
  - hostname: PE1
    interfaces:
      - name: eth1
        ipv4_address: 10.0.1.0/31
      - name: eth2
        ipv4_address: 10.0.3.0/31
      - name: eth3
        ipv4_address: 172.16.0.1/24
      - name: lo0
        ipv4_address: 192.168.3.1/32
    bgp:
      as_number: 65020
      router_id: 192.168.3.1
      neighbors:
        - ip: 10.0.1.1
          remote_as: 65001
        - ip: 10.0.3.1
          remote_as: 65002
        - is_group: True
          group_name: IBGP-PEERS
          remote_as: 65020
          next_hop_self: True
          group_members:
            - 172.16.0.2
            - 172.16.0.3
            - 172.16.0.4
```

3 Network implementation

3.1 Containerlab

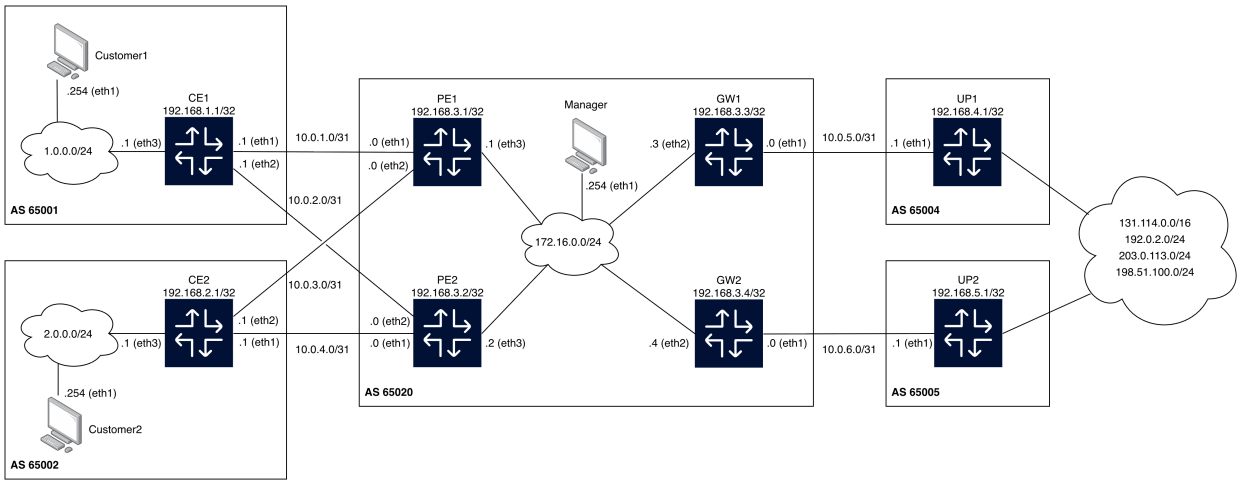

Figure 2: Schema of the implemented network

The implemented network is shown in Figure 2. The main implementation choices are explained below:

- We chose `1.0.0.0/24` as *Customer 1*'s network, `2.0.0.0/24` as *Customer 2*'s network and a few networks (`131.114.0.0/16`, `192.0.2.0/24`, `198.51.100.0/24`, `203.0.113.0/24`) as "*the internet*". We also added a linux node in each customer network for testing purposes
- The core AS number is `65020` because it was defined like that in the project specifications. Other AS numbers are taken from the private/free-use range `64512-65534`, but that is not mandatory, any AS number should do
- To implement the unique router identifiers used in BGP, we assigned `192.168.<AS index>.<router index>/32` addresses to loopback interfaces
- Point-to-point links follow the `10.0.<link index>.<interface>/31` addressing
- `eth0` interfaces are avoided since they are automatically connected to the management network created by ContainerLab
- The LAN of `AS 65020` is implemented using a linux bridge (`br-core`). Note that ContainerLab does not create bridges automatically, they must be created manually before running `containerlab deploy`

To configure the routers, we bind the required configuration files to the router's filesystem and then execute a bash script. The instantiation of `CE1` (`network/network.clab.yaml`) is shown below as an example:

```
62      CE1:
63        kind: linux
64        image: local/frr-ssh:latest
65        binds:
66          - ./config/frr/nodes/__clabNodeName__.conf:/etc/frr/frr.conf
67          - ./config/frr/daemons:/etc/frr/daemons
68          - ./config/frr/vtysh.conf:/etc/frr/vtysh.conf
69          - ./config/frr/configure.sh:/tmp/configure.sh
70        exec:
71          - bash /tmp/configure.sh
```

In particular:

- The `daemons` file enables the BGP daemon (all FRR daemons are disabled by default)
- The `vtysh.conf` file instructs the FRR shell (`vtysh`) to use a single configuration file (`/etc/frr/frr.conf`) instead of splitting the configuration into multiple files, one for each daemon
- The `configure.sh` script starts the SSH daemon and separates the management traffic from the data plane traffic by moving the management interface (`eth0`) into its own routing table
- The `__clabNodeName__.conf` file (in this case, `CE1.conf`) contains all the configuration settings, as explained in Section 2.2. All the configurations contain the lines:

  ```
  no bgp ebgp-requires-policy
  no bgp network import-check
  ```

  This is because FRR by default filters out every BGP update. Since we control the whole internet, for testing purposes, we instead allow everything. Specifically, we also assumed that the two customers are willing to advertise their networks one with each other.

  Moreover, when iBGP is used (i.e., between the routers of `AS 65020`), one can find:

  ```
  neighbor IBGP-PEERS next-hop-self
  ```

  This is because, by default, when an eBGP router advertises a route into an AS, it sets the `NEXT_HOP` attribute to its own interface IP address. When the route is passed via iBGP to other routers, the next hop, that usually is only directly reachable from the edge router that received the update, does not change. This command overwrites the `NEXT_HOP` field with the IP of the iBGP router propagating the update, fixing the reachability issues

- Finally, the docker image `local/frr-ssh:latest` is a custom FRR image with SSH installed and a `netadmin` user that always logs in to vtysh. This will be useful in Section 4.3 to programmatically adjust a router's configuration.

```
FROM quay.io/frrouting/frr:10.4.1

# shadow is required to run useradd
RUN apk add --no-cache openssh shadow

# Generate host keys (required for sshd to run)
RUN ssh-keygen -A

# Add an user that always logs int to vtysh (frrvty group is required for
↪ permissions)
RUN useradd -m -s /usr/bin/vtysh -G frrvty netadmin
RUN echo "netadmin:super-strong-password" | chpasswd
```

The nodes added for testing purposes (i.e., *Customer1*, *Customer2*), and the *Manager* node, are implemented as Alpine linux containers. Specifically, the *Manager* node runs a patched Alpine linux image with the python library `netmiko` pre-installed (again, this will be useful in Section 4.3).

The instantiation of the *Manager* node (`network/network.clab.yaml`) is shown below as an example:

```
    manager:
      kind: linux
      image: local/manager:latest
      env:
        MY_IP: 172.16.0.254/24
        MY_GW: 172.16.0.3
      binds:
        - ../automation-system:/automation-system
        - ./config/alpine/configure.sh:/tmp/configure.sh
      exec:
        - sh /tmp/configure.sh
```

In particular, the Alpine nodes receive their IP address and gateway as environment variables, and, at startup, they run a *sh* script similar to the one ran by FRR routers (separates the management traffic from the data plane traffic).

The gateway of *Customer1* and *Customer2* is the only router they are attached to, i.e., `CE1` and `CE2`, respectively. For the *Manager* node, there are multiple candidate gateways (all the ones in the same LAN); we arbitrarily chose `GW1`.

3.2 Testing

3.2.1 Customer edge routers

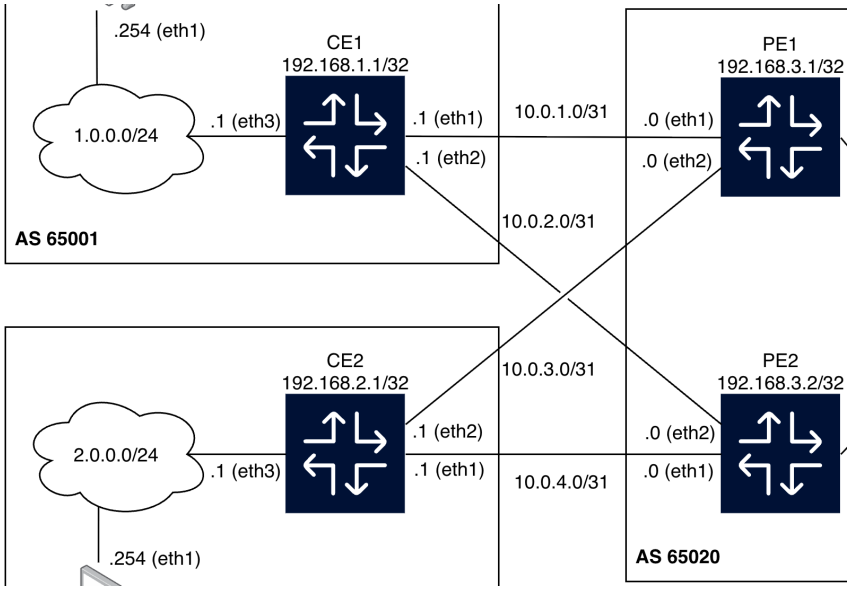

Figure 3: Portion of the network relevant for the Customer Edge routers

Let's inspect `CE1` as an example:

```
# show ip bgp summary (edited)
BGP router identifier 192.168.1.1, local AS number 65001
Neighbor        V         AS   State/PfxRcd   PfxSnt
10.0.1.0        4      65020              2        3
10.0.2.0        4      65020              2        3
```

```
# show ip bgp (edited)
     Network          Next Hop            Metric LocPrf Weight Path
 *>  1.0.0.0/24       0.0.0.0                  0         32768 i
 *>  2.0.0.0/24       10.0.1.0                                 0 65020 65002 i
 *=                   10.0.2.0                                 0 65020 65002 i
 *>  131.114.0.0/16   10.0.1.0                                 0 65020 65004 i
 *=                   10.0.2.0                                 0 65020 65004 i
 *>  192.0.2.0/24     10.0.1.0                                 0 65020 65004 i
 *=                   10.0.2.0                                 0 65020 65004 i
 *>  198.51.100.0/24  10.0.1.0                                 0 65020 65004 i
 *=                   10.0.2.0                                 0 65020 65004 i
 *>  203.0.113.0/24   10.0.1.0                                 0 65020 65004 i
 *=                   10.0.2.0                                 0 65020 65004 i
```

We can see that:

- The router identifier and AS number correspond to the expected values
- The two neighbors, `PE1` and `PE2`, are up and running (there are received prefixes)
- The router correctly received all the networks shared by the upstream routers and also the network of the second customer. All received prefixes have two equal-cost paths, one via `PE1` and one via `PE2`. Note also that multipath is enabled (`=` symbol), therefore, outbound traffic will be automatically balanced between the two paths

3.2.2 Provider edge routers

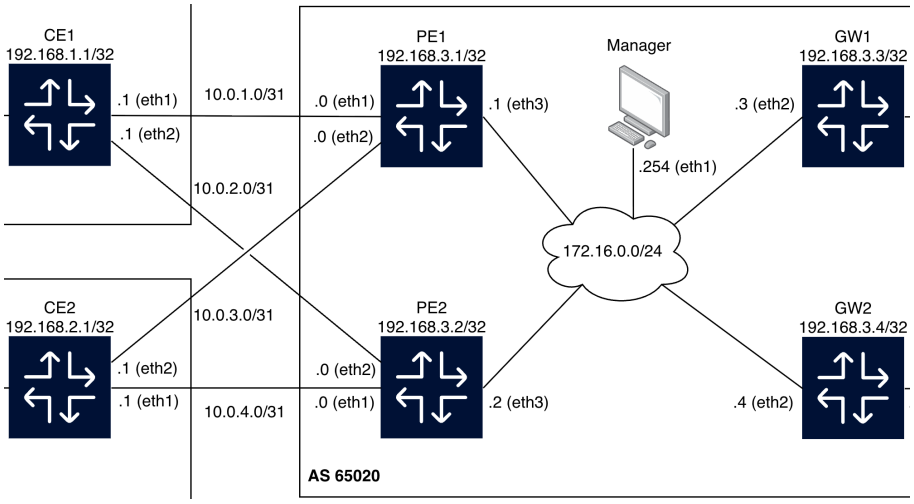

Figure 4: Portion of the network relevant for the Provider Edge routers

Let's inspect `PE1` as an example:

```
# show ip bgp summary (edited)
BGP router identifier 192.168.3.1, local AS number 65020
Neighbor        V         AS   State/PfxRcd   PfxSnt
172.16.0.2      4      65020              2        2
172.16.0.3      4      65020              1        2
172.16.0.4      4      65020              1        2
10.0.1.1        4      65001              1        3
10.0.3.1        4      65002              1        3
```

```
# show ip bgp (edited)
     Network          Next Hop            Metric LocPrf Weight Path
 *>  1.0.0.0/24       10.0.1.1                 0             0 65001 i
 * i                  172.16.0.2               0    100      0 65001 i
 *>  2.0.0.0/24       10.0.3.1                 0             0 65002 i
 * i                  172.16.0.2               0    100      0 65002 i
 *>i 131.114.0.0/16   172.16.0.3               0    100      0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>i 192.0.2.0/24     172.16.0.3               0    100      0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>i 198.51.100.0/24  172.16.0.3               0    100      0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>i 203.0.113.0/24   172.16.0.3               0    100      0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
```

The router correctly received all the advertised prefixes. For each prefix, there are two possible paths: when possible, the eBGP path is preferred. For upstream prefixes, instead, `GW1` (`172.16.0.3`) is chosen over `GW2` (`172.16.0.4`) only because its router id is lower (last tie-break rule of BGP).

Note that, in this case, to reach upstream networks, multipath is not enabled: even though there always are two possible paths, they go through different ASs, and, therefore, they are not considered of equal cost, even if all the other cost metrics match. This constraint can be relaxed with `bgp bestpath as-path multipath-relax`, but we decided against it as discussed in section 4.1.

3.2.3 Gateway routers

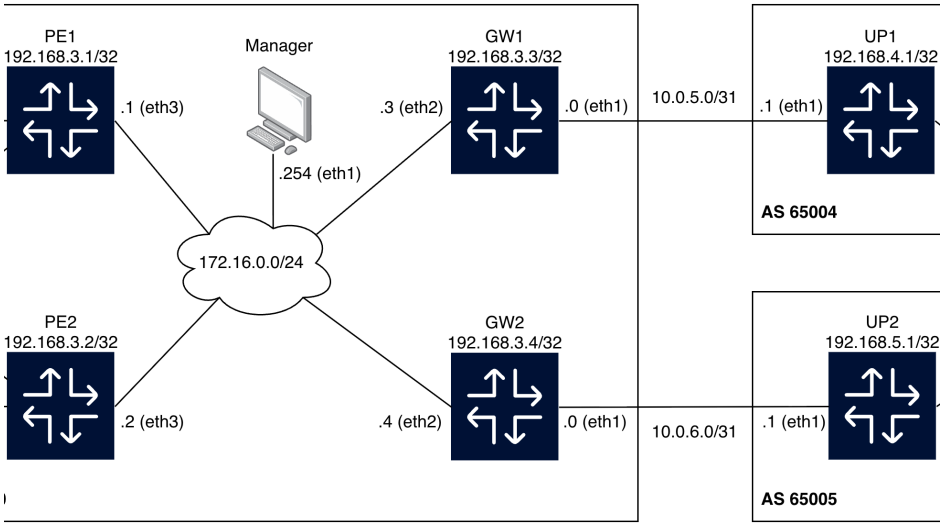

Figure 5: Portion of the network relevant for the GateWay routers

Let's inspect `GW1` as an example:

```
# show ip bgp summary (edited)
BGP router identifier 192.168.3.3, local AS number 65020
Neighbor        V         AS   State/PfxRcd   PfxSnt
172.16.0.1      4      65020              2        1
172.16.0.2      4      65020              2        1
172.16.0.4      4      65020              1        1
10.0.5.1        4      65004              1        3
```

```
# show ip bgp (edited)
     Network          Next Hop            Metric LocPrf Weight Path
 *>i 1.0.0.0/24       172.16.0.1               0    100      0 65001 i
 *=i                  172.16.0.2               0    100      0 65001 i
 *>i 2.0.0.0/24       172.16.0.1               0    100      0 65002 i
 *=i                  172.16.0.2               0    100      0 65002 i
 *>  131.114.0.0/16   10.0.5.1                 0             0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>  192.0.2.0/24     10.0.5.1                 0             0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>  198.51.100.0/24  10.0.5.1                 0             0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>  203.0.113.0/24   10.0.5.1                 0             0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
```

3.2.4 Upstream routers

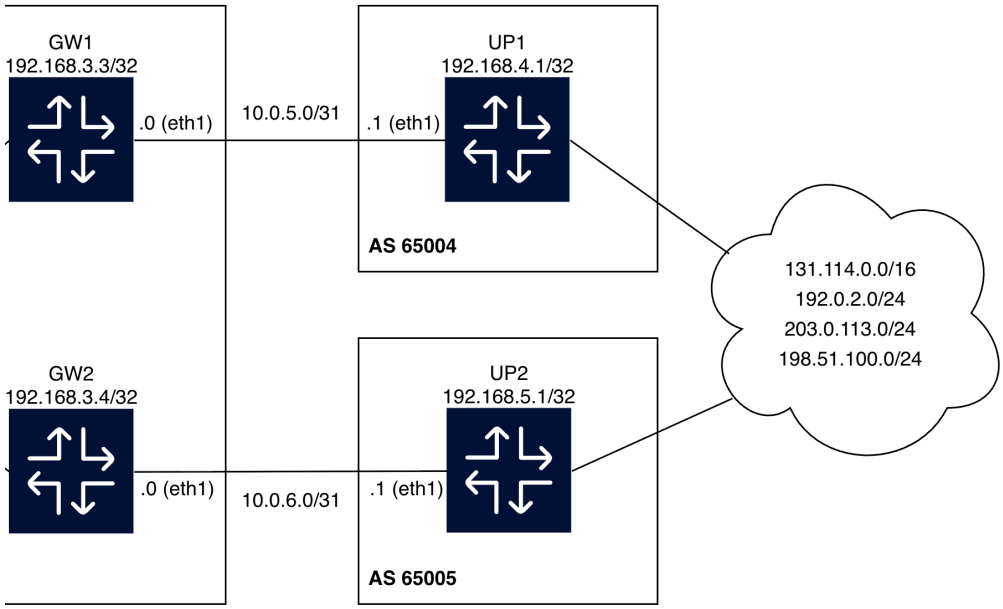

Figure 6: Portion of the network relevant for the UPstream routers

Let's inspect `UP1` as an example:

```
# show ip bgp summary (edited)
BGP router identifier 192.168.4.1, local AS number 65004
Neighbor        V         AS   State/PfxRcd   PfxSnt
10.0.5.0        4      65020              2        3
```

```
# show ip bgp (edited)
     Network          Next Hop            Metric LocPrf Weight Path
 *>  1.0.0.0/24       10.0.5.0                               0 65020 65001 i
 *>  2.0.0.0/24       10.0.5.0                               0 65020 65002 i
 *>  131.114.0.0/16   0.0.0.0                  0         32768 i
 *>  192.0.2.0/24     0.0.0.0                  0         32768 i
 *>  198.51.100.0/24  0.0.0.0                  0         32768 i
 *>  203.0.113.0/24   0.0.0.0                  0         32768 i
```

3.2.5 Test nodes

Let's now inspect the routing table of the test nodes. As expected, each node can reach the network they are directly connected to, while every other request goes to the default gateway:

```
# Customer1
default via 1.0.0.1 dev eth1 # CE1
1.0.0.0/24 dev eth1 scope link  src 1.0.0.254
```

```
# Customer2
default via 2.0.0.1 dev eth1 # CE2
2.0.0.0/16 dev eth1 scope link  src 2.0.0.254
```

```
# Manager
default via 172.16.0.3 dev eth1 # GW1
172.16.0.0/24 dev eth1 scope link  src 172.16.0.254
```

As expected, from *Customer1* (and also from *Customer2*), we are able to ping every shared network:

```
64 bytes from 2.0.0.1: seq=0 ttl=62 time=0.204 ms
64 bytes from 131.114.0.1: seq=0 ttl=61 time=0.222 ms
64 bytes from 192.0.2.1: seq=0 ttl=61 time=0.239 ms
64 bytes from 198.51.100.1: seq=0 ttl=61 time=0.266 ms
64 bytes from 203.0.113.1: seq=0 ttl=61 time=0.298 ms
```

The *Manager* node, in contrast, is only able to ping internally to its own LAN. This is expected: while ICMP requests reach external destinations, the corresponding reply packets are discarded on the return path because no route exist (the `172.16.0.0/24` prefix is unknown to the outside network).

```
64 bytes from 172.16.0.1: seq=0 ttl=64 time=0.140 ms
64 bytes from 172.16.0.2: seq=0 ttl=64 time=0.234 ms
64 bytes from 172.16.0.3: seq=0 ttl=64 time=0.233 ms
64 bytes from 172.16.0.4: seq=0 ttl=64 time=0.280 ms
```

4 Traffic distribution

To provide dynamic load balancing between the two upstream links a Network Automation System has been implemented on the manager node inside `AS 65020`. This systems takes traffic prediction matrices as inputs and uses them to compute the optimal traffic distribution across the two upstream links. It then automatically adjusts the BGP configuration of routers the inside `AS 65020` to enforce such distribution.

The system only changes BGP attributes (i.e., `LOCAL_PREF`) to influence path priority. Thus, if an upstream link becomes unavailable (e.g., `GW1-UP1`), the other one (e.g., `GW2-UP2`) will serve as backup and automatically take over all the traffic.

The system's load balancing is designed to only affect outbound traffic as only one network per customer is present (two total). If needed the system can be easily modified to also try load balancing incoming traffic (from upstream routers) by altering its advertisements of the client networks (or some of their subnets) to make one of the links more desirable (e.g.: by lengthening its `AS_PATH`).

The automation system consists of a python script (`distribute_traffic.py`) that takes a prediction matrix in a `.json` file as an input. The prediction matrix must be in the following format:

```
{
    "Customer": {
        "Destination": Load, // Load values must be integers
        ...
    },
    ...
}
```

This is an example of a prediction matrix:

```
{
    "C1": {
        "131.114.0.0/16": 100,
        "198.51.100.0/24": 25,
    },
    "C2": {
        "131.114.0.0/16": 20,
        "192.0.2.0/24": 50,
        "203.0.113.0/24": 40
    }
}
```

4.1 Balancing strategies

In the analyzed network, there are two areas that may require balancing: the `CE-PE` links and the `GW-UP` links.

In the first case, as there is a single `CE` router with multiple outbound links to different `PE` routers, it is possible to use BGP multipath to allow for automatic load balancing as shown in section 3.2.1. This makes it unnecessary for the Network Automation System to compute the optimal distribution on customer links and update `PE` routers accordingly (i.e.: influencing `MED` BGP attributes).

The same approach could be used on `PE1` and `PE2` by allowing multipathing on unequal cost paths (i.e.: `bgp bestpath as-path multipath-relax` to allow paths through different ASs).

We decided against this approach for two main reasons:

- The first is that doing so would allow for very little control over the actual balancing, leaving everything in the hands of the single `PE` nodes, which do not coordinate with each other. This may or may not produce a good balancing depending on the specific traffic distribution.
- The second is that this does not take into account traffic predictions, thus explicitly violating the received requirements and possibly making poor choices due to missing information.

Our solution is instead to manually define which gateway should be used to reach each destination by increasing the advertised `LOCAL_PREF` of the preferred routes. This way, the chosen routes have higher priority, and, therefore, they are used (and advertised by) `PE` routers.

4.2 Determining the optimal distribution

To determine the optimal distribution across upstream links, the system first merges all the customers' predictions together by computing for each destination the total prediction across all customers. This is needed as the systems does not differentiate paths based on the customer, so the balancing must be done based only on destination.

The load balancing problem (distributing N paths of different load across two links minimizing the difference in load between the two) can be solved by reducing it to a 0-1 Knapsack problem with the following parameters:

- $Capacity = \sum_{i=1}^{N} Load_i/2$
- $Weight_i = Load_i$
- $Value_i = Load_i$

This means choosing the paths that maximize the load on a link, while keeping it at or below half of the total. All the remaining paths will then belong to the other link. This way the algorithms maximizes how close both links are to half of the total load.

The systems solves the Knapsack program using a well known dynamic programming algorithm with $\mathcal{O}(N \cdot Capacity)$ time and space complexity.

4.3 Automatic router reconfiguration

The computed distribution is then enforced by increasing the `LOCAL_PREF` advertised by `GW1` and `GW2` for the destinations assigned to their respective links.

This is done running the following configuration commands on the routers' vtysh console to edit the prefix list containing the destinations that should have higher `LOCAL_PREF`:

```
# Enter configuration mode
configure terminal

# Clear LOCALPREF_PREFIX_LIST of all entries
no ip prefix-list LOCALPREF_PREFIX_LIST

# Add all destinations assigned to this router to the prefix list
ip prefix-list LOCALPREF_PREFIX_LIST permit <destination 1>
ip prefix-list LOCALPREF_PREFIX_LIST permit <destination 2>
# ...
# If no destinations are present the prefix list reamins undefined, defaulting
↪  to deny everything

# Exit configuration mode
end
```

It is important to note how this script always clears the previous configuration. If it were not the case, running the script multiple times with different predictions would not balance the traffic correctly, as the included networks could be different, potentially leaving lingering effects from previous balancings. This also means that prediction matrices must contain all the predicted traffic, not just new predictions.

As the python script that computes and enforces the distribution runs on a single centralized manager node, it cannot directly run commands on the gateway routers. The script therefore uses the *Netmiko* library to establish a ssh connection with the `netadmin` user on the routers (that for convenience has been set to use vtysh as default shell) and execute the commands remotely.

The prefix list defined this way is used by a route map to match the destinations in incoming advertisements from the upstream router and decide whether to increase `LOCAL_PREF`.

In order to do this, the following statements are included in the configuration for both `GW1` and `GW2`:

```
route-map LOCALPREF_MAP permit 10
    match ip address prefix-list LOCALPREF_PREFIX_LIST
    set local-preference 200
route-map LOCALPREF_MAP permit 999

router bgp 65020
    ...
    neighbor <neighbor> route-map LOCALPREF_MAP in
    ...
```

4.4 Testing results

We fist analyze the default behavior of the network by running a `traceroute` from *Customer1* to every upstream network:

```
traceroute to 131.114.0.1 (131.114.0.1), 30 hops max, 46 byte packets
 1  1.0.0.1 (1.0.0.1)  0.015 ms  0.035 ms  *                    // CE1
 2  10.0.2.0 (10.0.2.0)  0.015 ms  0.146 ms  0.092 ms           // PE2
 3  172.16.0.3 (172.16.0.3)  0.006 ms  0.025 ms  0.005 ms       // GW1
 4  131.114.0.1 (131.114.0.1)  0.008 ms  0.024 ms  0.007 ms     // UP1
```

```
traceroute to 192.0.2.1 (192.0.2.1), 30 hops max, 46 byte packets
 1  1.0.0.1 (1.0.0.1)  0.012 ms  0.037 ms  *                    // CE1
 2  10.0.2.0 (10.0.2.0)  0.014 ms  0.039 ms  0.012 ms           // PE2
 3  172.16.0.3 (172.16.0.3)  0.006 ms  0.027 ms  0.043 ms       // GW1
 4  192.0.2.1 (192.0.2.1)  0.005 ms  0.029 ms  0.007 ms        // UP1
```

```
traceroute to 198.51.100.1 (198.51.100.1), 30 hops max, 46 byte packets
 1  1.0.0.1 (1.0.0.1)  0.014 ms  0.033 ms  *                    // CE1
 2  10.0.1.0 (10.0.1.0)  0.010 ms  0.041 ms  0.014 ms           // PE1
 3  172.16.0.3 (172.16.0.3)  0.006 ms  0.025 ms  0.006 ms       // GW1
 4  198.51.100.1 (198.51.100.1)  0.005 ms  0.066 ms  0.007 ms   // UP1
```

```
traceroute to 203.0.113.1 (203.0.113.1), 30 hops max, 46 byte packets
 1  1.0.0.1 (1.0.0.1)  0.015 ms  0.036 ms  *                    // CE1
 2  10.0.2.0 (10.0.2.0)  0.013 ms  0.043 ms  0.006 ms           // PE2
 3  172.16.0.3 (172.16.0.3)  0.006 ms  0.027 ms  0.007 ms       // GW1
 4  203.0.113.1 (203.0.113.1)  0.006 ms  0.023 ms  0.005 ms     // UP1
```

As explained in Section 3.2.1, due to BGP Equal Cost MultiPath, the traffic to the second hop is automatically balanced between the two possible `CE1 - PEx` links.

Moreover, as explained in Section 3.2.2, the traffic always goes through the `GW1 - UP1` link.

All of the above is also true for *Customer2*.

Consider now the traffic prediction below (`automation-system/predictions/test.json`):

```
{
    "C1": {
        "198.51.100.0/24": 150,
        "203.0.113.0/24": 35
    },
    "C2": {
        "131.114.0.0/16": 20,
        "192.0.2.0/24": 50,
        "203.0.113.0/24": 40
    }
}
```

By running `python3 distribute_traffic.py predictions/test.json` on the *Manager* node, one obtains the following output:

```
Total load: 295
UP1: 145
['203.0.113.0/24', '131.114.0.0/16', '192.0.2.0/24']
UP2: 150
['198.51.100.0/24']

Sending commands to 172.16.0.3: ...

Output: configure terminal
GW1(config)# no ip prefix-list LOCALPREF_PREFIX_LIST
GW1(config)# ip prefix-list LOCALPREF_PREFIX_LIST permit 203.0.113.0/24
GW1(config)# ip prefix-list LOCALPREF_PREFIX_LIST permit 131.114.0.0/16
GW1(config)# ip prefix-list LOCALPREF_PREFIX_LIST permit 192.0.2.0/24
GW1(config)# end
GW1#

Sending commands to 172.16.0.4: ...

Output: configure terminal
GW2(config)# no ip prefix-list LOCALPREF_PREFIX_LIST
GW2(config)# ip prefix-list LOCALPREF_PREFIX_LIST permit 198.51.100.0/24
GW2(config)# end
GW2#
```

Let's now check the BGP routes on PE1 with the command `show ip bgp`.

Old routes (before balancing the traffic):

```
     Network          Next Hop            Metric LocPrf Weight Path
 *>  1.0.0.0/24       10.0.1.1                 0             0 65001 i
 * i                  172.16.0.2               0    100      0 65001 i
 *>  2.0.0.0/24       10.0.3.1                 0             0 65002 i
 * i                  172.16.0.2               0    100      0 65002 i
 *>i 131.114.0.0/16   172.16.0.3               0    100      0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>i 192.0.2.0/24     172.16.0.3               0    100      0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>i 198.51.100.0/24  172.16.0.3               0    100      0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
 *>i 203.0.113.0/24   172.16.0.3               0    100      0 65004 i
 * i                  172.16.0.4               0    100      0 65005 i
```

New routes (after balancing the traffic):

```
     Network          Next Hop            Metric LocPrf Weight Path
 *>  1.0.0.0/24       10.0.1.1                 0             0 65001 i
 * i                  172.16.0.2               0    100      0 65001 i
 *>  2.0.0.0/24       10.0.3.1                 0             0 65002 i
 * i                  172.16.0.2               0    100      0 65002 i
 *>i 131.114.0.0/16   172.16.0.3               0    200      0 65004 i // GW1
 *>i 192.0.2.0/24     172.16.0.3               0    200      0 65004 i // GW1
 *>i 198.51.100.0/24  172.16.0.4               0    200      0 65005 i // GW2!
 *>i 203.0.113.0/24   172.16.0.3               0    200      0 65004 i // GW1
```

This is perfectly coherent with how how the traffic was distributed:

```
Total load: 295
UP1: 145
['203.0.113.0/24', '131.114.0.0/16', '192.0.2.0/24']
UP2: 150
['198.51.100.0/24']
```

The highlighted lines are exactly the same on PE2.

We can now check that all the changes were successfully applied by repeating the same `traceroute` experiment performed at the beginning of this chapter. Apart from the second hop (which is multipath, and, therefore, variable in its nature), the highlighted lines are the ones that changed w.r.t the first experiment.

```
traceroute to 131.114.0.1 (131.114.0.1), 30 hops max, 46 byte packets
 1  1.0.0.1 (1.0.0.1)  0.016 ms  0.093 ms  *                    // CE1
 2  10.0.1.0 (10.0.1.0)  0.014 ms  0.038 ms  0.006 ms          // PE1
 3  172.16.0.3 (172.16.0.3)  0.005 ms  0.031 ms  0.006 ms      // GW1
 4  131.114.0.1 (131.114.0.1)  0.007 ms  0.074 ms  0.007 ms    // UP1
```

```
traceroute to 192.0.2.1 (192.0.2.1), 30 hops max, 46 byte packets
 1  1.0.0.1 (1.0.0.1)  0.014 ms  0.038 ms  *                    // CE1
 2  10.0.1.0 (10.0.1.0)  0.014 ms  0.042 ms  0.006 ms          // PE1
 3  172.16.0.3 (172.16.0.3)  0.006 ms  0.045 ms  0.005 ms      // GW1
 4  192.0.2.1 (192.0.2.1)  0.006 ms  0.026 ms  0.005 ms        // UP1
```

```
traceroute to 198.51.100.1 (198.51.100.1), 30 hops max, 46 byte packets
 1  1.0.0.1 (1.0.0.1)  0.014 ms  0.076 ms  *                    // CE1
 2  10.0.2.0 (10.0.2.0)  0.013 ms  0.041 ms  0.005 ms          // PE2
 3  172.16.0.4 (172.16.0.4)  0.005 ms  0.029 ms  0.006 ms      // GW2
 4  198.51.100.1 (198.51.100.1)  0.008 ms  0.028 ms  0.005 ms  // UP2
```

```
traceroute to 203.0.113.1 (203.0.113.1), 30 hops max, 46 byte packets
 1  1.0.0.1 (1.0.0.1)  0.015 ms  0.038 ms  *                    // CE1
 2  10.0.1.0 (10.0.1.0)  0.059 ms  0.042 ms  0.007 ms          // PE1
 3  172.16.0.3 (172.16.0.3)  0.007 ms  0.071 ms  0.006 ms      // GW1
 4  203.0.113.1 (203.0.113.1)  0.005 ms  0.078 ms  0.007 ms    // UP1
```

Before applying traffic predictions, by default, all upstream traffic was routed to `GW1`. We can see now that this is no longer true: as expected, all traffic to `198.51.100.0/24` is instead routed to `GW2`.