ASSIGNMENT 7 – LAB 9: NETWORK AUTOMATION

MAIN TOPICS: why is automation important in DS, SDN and O-Ran concepts with NETCONF/YANG and Ansible, ncclient

INITIAL PART OF SET UP OF THE ENVIRONMENT (the git has been already cloned) (Non assignment)
(Terminal that I used on VSC)

```
pip3 install --user ncclient netconf-console2 paramiko ansible
```

```
sudo usermod -aG docker $USER
```

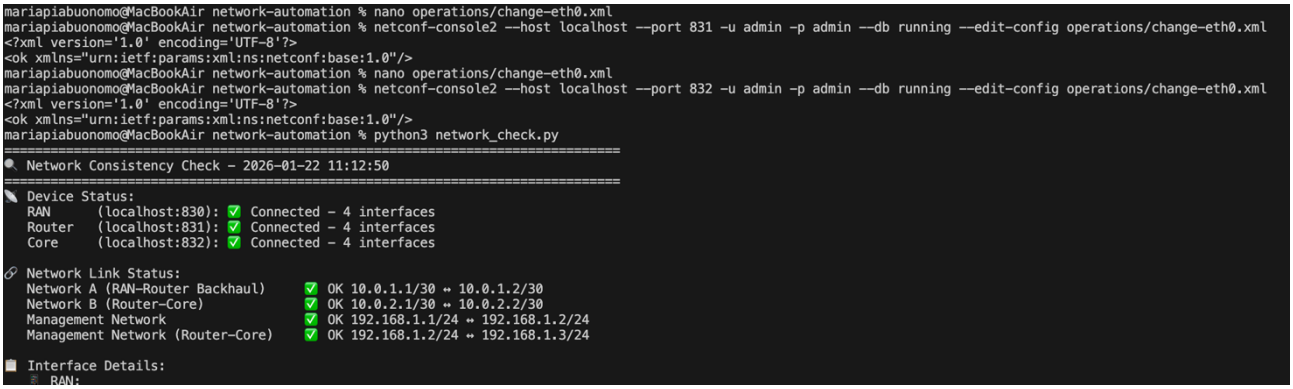
```
docker --version
docker compose version   # or: docker-compose --version
python3 --version
pip3 --version
ansible --version
netconf-console2 --help  # should print help if installed
```

```
cd network-automation/

docker-compose up -d (to leave it in the same terminal)
```

```
#docker-compose up
```

EXERCISE 1 (Not assignment)
docker ps (I check that all the ports are up. To remember: RAN = 830, ROUTER = 831, CORE = 832)
For each device I have modified the file change-eth0.xml by changing the interfaces and the IPs depending on if I wanted to work on RAN, ROUTER or CORE (always with the running mode).



The scheme I followed is the following:

- RAN:

eth0 – 192.168.1.1/24
backhaul0 – 10.0.1.1/30

- ROUTER:

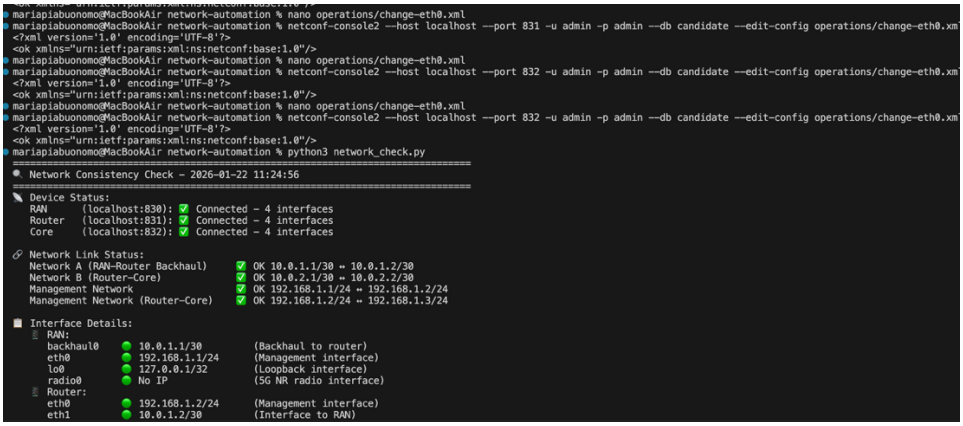
eth0 – 192.168.1.2/24
eth1 – 10.0.1.2/30

eth2 – 10.0.2.1/30

- CORE:

eth0 – 192.168.1.3/24
eth1 – 10.0.2.2/30
And then I always updated them with:
netconf-console2 --host localhost --port 831 -u admin -p admin --db running --edit-config operations/change-eth0.xml

The system works immediately because running controls the active data plane. Upon restart, the device loads the startup datastore. All your changes in running will be lost. If BBU backhaul0 doesn't match Router eth1, the link Layer 3 connectivity fails. NETCONF (without a validation script) does not inherently "know" if two different devices have matching subnets; it only checks if the XML is syntactically valid.
If we work with candidate...



Would the current configuration allow the system to work properly?

No. If you have only updated the candidate datastore, the system is still physically operating on the running configuration. Since the running datastore still contains the previous exercise's IPs (or the initial ones), the new network plan (the 100.0 and 200.0 subnets) is not yet active. The devices cannot communicate using the new IP scheme until a <commit/> is issued.

Are there differences between the datastores? Why?

Yes. The Difference: The candidate datastore contains my new XML edits, while running remains unchanged.

This is based on the "Shared Data Model" design. It allows a network engineer to upload multiple complex changes, verify them, and check for syntax errors without crashing the live network. It acts as a transactional buffer.

What would happen if a device gets restarted in such a case?

The device would boot up using the startup datastore. All changes sitting in the candidate datastore would be lost (wiped). The device would revert to whatever configuration was last saved to startup.

What would happen if BBU backhaul0 and Router eth1 didn't match?

Layer 3 connectivity would fail. The BBU would be unable to send traffic to the Router because they would be on different logical subnets.

How does this approach scale?

Managing 100 RAN devices, 50 routers, and 1 core device using individual netconf-console2 commands is extremely inefficient.

From the point of view of time and effort in fact, manually editing 151 XML files is a recipe for typos. One wrong bit in a mask could take down a whole cell site.

From the Configuration Drift POV, if we update devices one by one in the running datastore, the network exists in a "broken" partial state for the duration of the update (which could take hours).

And with the commit for each interface:

Activity Analysis (Post-Commit)

Would the system work properly?

Yes. Now that the configuration is in running, the device's interfaces are actually using the new IP addresses (10.0.100.x and 10.0.200.x). If your configuration was logically consistent, the data plane is now operational.

Are there differences between datastores?

Yes. candidate and running are now identical. However, startup still contains the old configuration.

Why?

The <commit> operation is designed to affect the active state immediately without making it permanent. This allows you to test the changes in a live environment before deciding to "save" them.

What happens if the device restarts?

The changes are lost. Upon reboot, the device ignores the running datastore and loads the startup datastore. You would be back at the old IP addresses.

EXERCISE 2 (not assignment) (su VSC si chiama LAB9_ES2.py)

Automation with python

```
(.venv) mariapiabuonomo@MacBookAir esercizidistr % /Users/mariapiabuonomo/Desktop/esercizidistr/.venv/bin/python /Users/mariapiabuonomo/Desktop/esercizidistr/network-automation/LAB9_ES2.py
Connected to 127.0.0.1:830
--- Updating device to 192.168.1.1 ---
Success: eth0 set to 192.168.1.1
get-config response:
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>eth0</name>
      <description>Management interface</description>
      <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:ethernetCsmacd</type>
      <enabled>true</enabled>
      <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <enabled>true</enabled>
        <address>
          <ip>192.168.1.1</ip>
          <prefix-length>24</prefix-length>
        </address>
      </ipv4>
    </interface>
  </interfaces>
</data>

Connected to 127.0.0.1:831
--- Updating device to 192.168.1.20 ---
Success: eth0 set to 192.168.1.20
get-config response:
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>eth0</name>
      <description>Management interface</description>
      <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:ethernetCsmacd</type>
      <enabled>true</enabled>
      <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <enabled>true</enabled>
        <address>
          <ip>192.168.1.20</ip>
          <prefix-length>24</prefix-length>
        </address>
        <address>
          <ip>192.168.100.20</ip>
          <prefix-length>24</prefix-length>
        </address>
      </ipv4>
    </interface>
  </interfaces>
```

```
        </address>
      </ipv4>
    </interface>
  </interfaces>
</data>

Connected to 127.0.0.1:832
--- Updating device to 192.168.1.30 ---
Success: eth0 set to 192.168.1.30
get-config response:
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>eth0</name>
      <description>Management interface</description>
      <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:ethernetCsmacd</type>
      <enabled>true</enabled>
      <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <enabled>true</enabled>
        <address>
          <ip>192.168.1.30</ip>
          <prefix-length>24</prefix-length>
        </address>
        <address>
          <ip>192.168.100.30</ip>
          <prefix-length>24</prefix-length>
        </address>
      </ipv4>
    </interface>
  </interfaces>
</data>
</data>

(.venv) mariapiabuonomo@MacBookAir esercizidistr % cd network-automation
(.venv) mariapiabuonomo@MacBookAir network-automation % python3 network_check.py
================================================================================
🔍 Network Consistency Check - 2026-01-22 12:43:22
================================================================================
📡 Device Status:
   RAN      (localhost:830): ✅ Connected - 4 interfaces
   Router   (localhost:831): ✅ Connected - 4 interfaces
   Core     (localhost:832): ✅ Connected - 4 interfaces

🔗 Network Link Status:
   Network A (RAN-Router Backhaul)     ✅ OK 10.0.1.1/30 ↔ 10.0.1.2/30
   Network B (Router-Core)             ✅ OK 10.0.2.1/30 ↔ 10.0.2.2/30
   Management Network                  ✅ OK 192.168.1.1/24 ↔ 192.168.1.20/24
   Management Network (Router-Core)    ✅ OK 192.168.1.20/24 ↔ 192.168.1.30/24

📋 Interface Details:
   📱 RAN:
      backhaul0     🟢 10.0.1.1/30         (Backhaul to router)
      eth0          🟢 192.168.1.1/24      (Management interface)
      lo0           🟢 127.0.0.1/32        (Loopback interface)
      radio0        🟢 No IP               (5G NR radio interface)
   📱 Router:
      eth0          🟢 192.168.1.20/24     (Management interface)
      eth1          🟢 10.0.1.2/30         (Interface to RAN)
      eth2          🟢 10.0.2.1/30         (Interface to Core)
      lo0           🟢 127.0.0.1/32        (Loopback interface)
   📱 Core:
      eth0          🟢 192.168.1.30/24     (Management interface)
      eth1          🟢 10.0.2.2/30         (Interface to Router)
      eth2          🟢 203.0.113.1/24      (Interface to Internet/External)
      lo0           🟢 127.0.0.1/32        (Loopback interface)
```

OBSERVATIONS:

How does this approach scale?

This approach is infinitely more scalable than manual entry, but it can be further optimized.

Time Spent:

- Manual: Managing 100 RAN, 50 Routers, and 1 Core (151 total) manually would take hours and likely result in several configuration errors.
- Sequential Python: Taking ~2 seconds per device, the script would finish in about 5 minutes.
- Parallel Python: By using threading, we could connect to all 151 devices simultaneously. The entire network could be updated in under 10 seconds.

Consistency: The primary benefit isn't just speed, it's reliability. The script ensures that every single device receives the exact same logic. If the script works for 3 devices, it will work for 3,000.

Error Handling: In a large-scale environment, we would enhance the script to log "failed" connections to a file, allowing to quickly identify and troubleshoot only the 2 or 3 devices that had issues, rather than checking all 151.

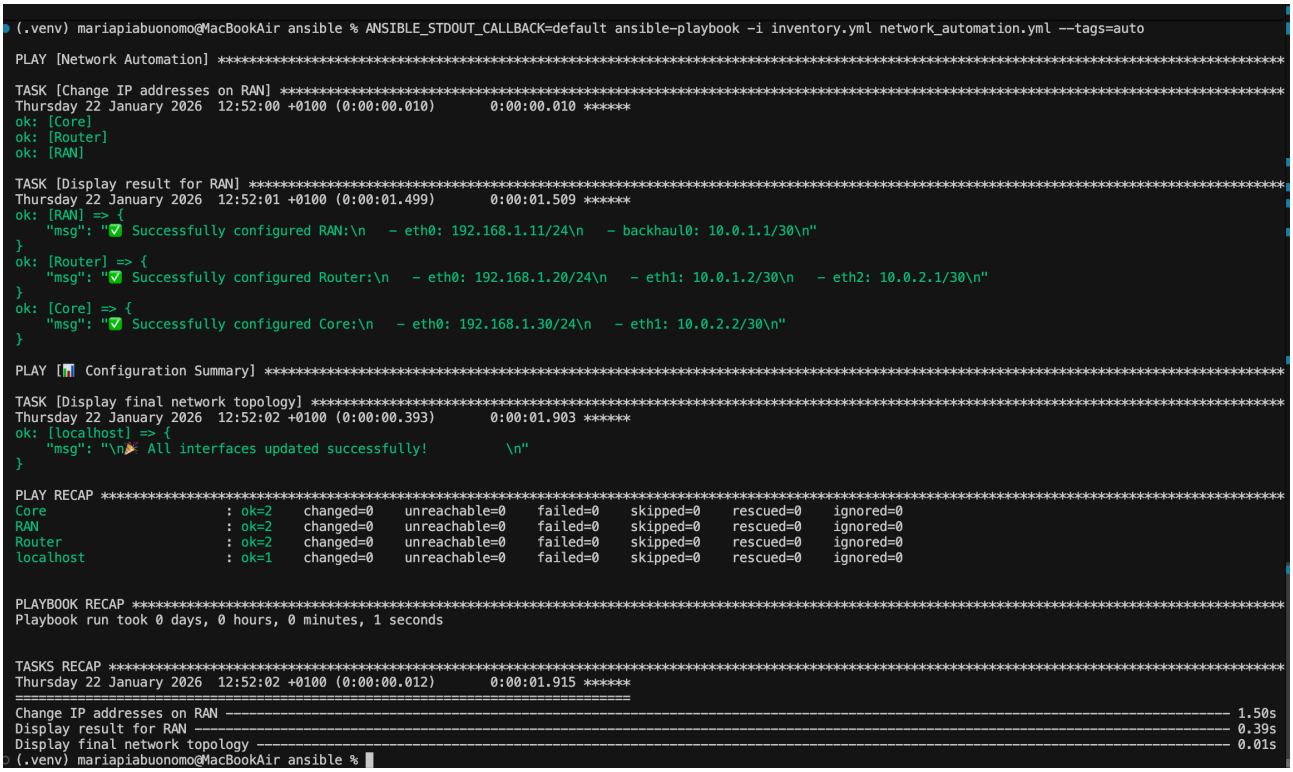
EXERCISE 3 (not assignment)
Automate with ansible:
At first I had this error while running:

ansible-playbook -i inventory.yml network_automation.yml --tags=auto

[ERROR]: The 'community.general.yaml' callback plugin has been removed. The plugin has been superseded by the option `result_format=yaml` in callback plugin ansible.builtin.default from ansible-core 2.13 onwards. This feature was removed from collection 'community.general' version 12.0.0.

So I had to use: ANSIBLE_STDOUT_CALLBACK=default ansible-playbook -i inventory.yml network_automation.yml --tags=auto

This was because of a version mismatch between my Ansible core installation and the community collections installed on my system.



OBSERVATIONS:

What is the playbook doing?

The playbook network_automation.yml is acting as a Declarative Orchestrator.

- Inventory Mapping: It reads inventory.yml to find the connection details (IPs/Ports) for the RAN, Router, and Core.
- It pushes the XML configuration defined in the tasks to ensure the eth0 management interfaces match the IPs specified in the inventory.
- Abstraction: It uses the ansible.netcommon.netconf_config module, which handles the "heavy lifting" of the NETCONF session (opening the SSH channel, sending the RPC, and parsing the <ok/> response) so you don't have to write that logic manually in Python.

Is the playbook working with candidate, running, startup, or all?

By default, this playbook works with Candidate and Running:

- Candidate: Ansible pushes the configuration here first to "test" if the XML is valid.
- Running: Immediately after a successful push to the candidate, Ansible issues a <commit/> to make the changes active in the running datastore.
- Startup: No. Unless the task specifically includes save: true or copy_config: true, **Ansible does not update the startup datastore.**

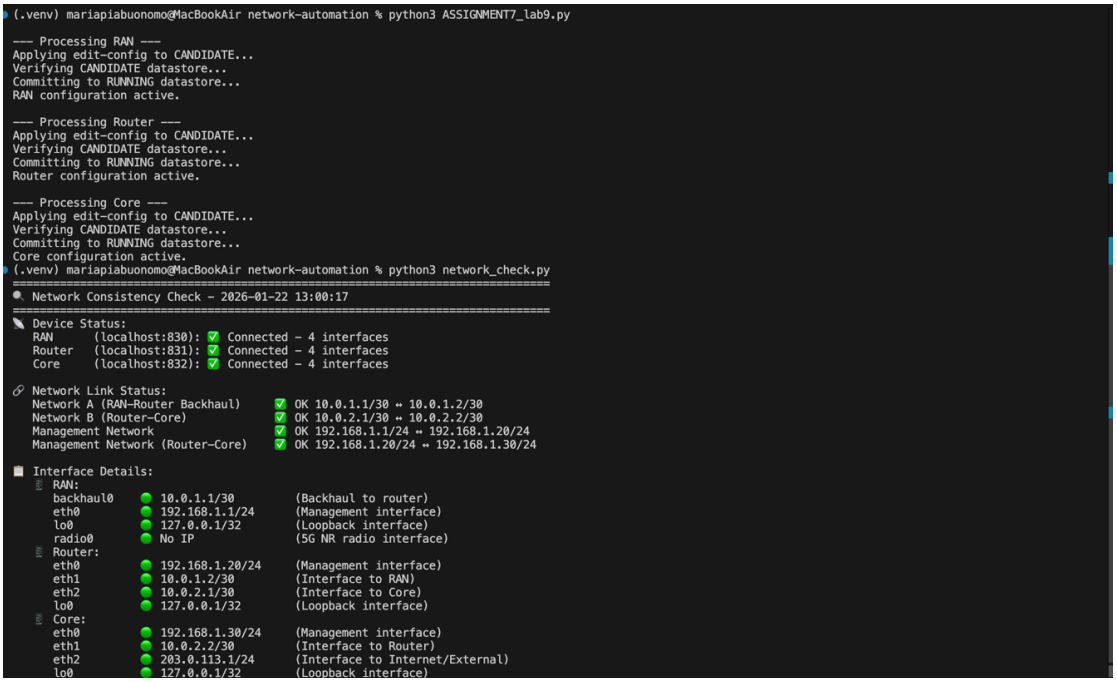
How does this approach scale?

Ansible is designed for "Massive Parallelism."

- If you need to manage 100 RAN devices and 50 routers, you don't touch the playbook (.yml). You only add those 150 devices to the inventory.yml.
- Forks: Ansible can connect to many devices at once using "forks." Instead of waiting for Device 1 to finish before starting Device 2 (Sequential), it can update 5, 50, or even 100 devices simultaneously.
- Group Logic: You can group devices in the inventory (e.g., [all_ran_nodes]). This allows you to apply a specific change to all 100 RAN devices with a single task, rather than looping through a list in Python.

ASSIGNMENT

Output:



COMMENTS:

Would the current configuration allow the system to work properly?

Yes. Because we performed a commit operation, the configuration has moved from the candidate (draft) to the running (active) datastore. The device's network stack is now using the new IPs (10.0.1.x and 10.0.2.x), allowing Layer 3 connectivity between the BBU, Router, and Core.

Are there differences between the datastores? Why?

Yes. * Running vs. Candidate: These are now identical after the commit.
Startup: This datastore is different (it contains the old configuration).
Why: In NETCONF, the commit operation only targets the running datastore. It does not automatically persist changes to the startup datastore. You would need a separate copy-config operation to synchronize them.

What would happen if a device gets restarted in such a case?
The configuration would be lost. On boot, a device ignores its running state and loads the configuration from the startup datastore. Since we haven't saved our changes to startup, the device would revert to the old IP addresses, and the network would break.

How does this approach scale? (151 Devices)
Manual vs. Automation: Manually configuring 151 devices via CLI would take hours and is prone to typos. The Python script handles all devices in under a minute.
Sequential vs. Parallel: This script is *sequential* (one device at a time). For a massive RAN deployment (100+ devices), we would use Ansible or Python Multi-threading to connect to all devices at once.
Impact: The time spent shifts from "repetitive typing" to "validating logic." Automation ensures that if the logic is correct for 1 device, it is correct for 1,000.