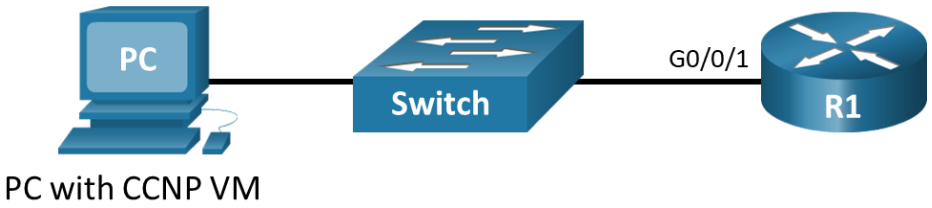
Lab - Use NETCONF to Access an IOS XE Router

Topology



Addressing Table

Device	Interface	IP Address	Subnet Mask
R1	G0/0/1	192.168.1.1	255.255.255.0
PC	NIC	DHCP	DHCP

Objectives

Part 1: Build the Network and Verify Connectivity

Part 2: Use a NETCONF Session to Gather Information

Part 3: Use ncclient to Connect to NETCONF

Part 4: Use ncclient to Retrieve the Configuration

Part 5: Use ncclient to Configure a Device

Background / Scenario

The Network Configuration Protocol (NETCONF), defined in RFCs 4741 and 6241, uses YANG data models to communicate with various devices on the network. YANG (yet another next generation) is a data modeling language. This language defines the data that is sent over network management protocols, like NETCONF. When using NETCONF to access an IOS XE device, the data is returned in XML format.

In this lab, you will use a NETCONF client, ncclient, which is a Python module for client-side scripting. You will use ncclient to verify NETCONF is configured, retrieve a device configuration, and modify a device configuration.

Required Resources

- 1 Router (Cisco 4221 with Cisco IOS XE Release 16.9.4 universal image or comparable)
- 1 Switch (Optional: any switch available for connecting R1 and the PC)
- 1 PC (Choice of operating system with Cisco Networking Academy CCNP VM running in a virtual machine client and terminal emulation program)
- Ethernet cables as shown in the topology

Instructions

Part 1: Build the Network and Verify Connectivity

Note: Skip Part 1 if you already completed it in a previous lab. However, be sure your CCNP VM can ping R1.

In Part 1, you will cable the devices, start the CCNP VM, configure R1 for NETCONF and RESTCONF access over an SSH connection. You will then verify connectivity between the CCNP VM and R1 as well as test an SSH connection to R1.

Step 1: Cable the network as shown in the topology.

Connect the devices as shown in the topology diagram, and cable as necessary.

Step 2: Start the CCNP VM.

Note: If you have not completed **Lab - Install the CCNP Virtual Machine**, do so now before continuing with this lab.

a. Open VirtualBox. Start the **CCNP VM** virtual machine.

b. Enter the password **StudentPass** to log into the VM as necessary.

Step 3: Configure R1.

Console into R1 and paste the following configuration into the CLI to configure basic settings and enable NETCONF, RESTCONF, and SSH.

```
enable
configure terminal
hostname R1
no ip domain lookup
line con 0
logging synchronous
 exec-timeout 0 0
 logging synchronous
line vty 0 15
 exec-t 0 0
 logg sync
 login local
 transport input ssh
ip domain name example.netacad.com
crypto key generate rsa modulus 2048
username cisco priv 15 password cisco123!
interface GigabitEthernet0/0/1
 description Link to PC
 ip address 192.168.1.1 255.255.255.0
 no shutdown
ip dhcp excluded-address 192.168.1.1 192.168.1.10
!Configure a DHCP server to assign IPv4 addressing to the CCNP VM
ip dhcp pool LAN
 network 192.168.1.0 /24
 default-router 192.168.1.1
```

```
 domain-name example.netacad.com
 end
copy run start
```

Step 4: Verify that the CCNP VM can ping the default gateway.

a. In the **CCNP VM**, open a terminal window.

b. Verify the **CCNP VM** is connected to R1 by either entering **ip address** to verify that the CCNP VM received IP addressing from the DHCP server, or simply by pinging R1 at 192.168.1.1. Enter **Ctrl+C** to break out of the ping, as shown in the example output below.

```
student@CCNP:~$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
default qlen 1000
    link/ether 00:50:56:b3:72:3b brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.15/24 brd 192.168.1.255 scope global dynamic noprefixroute ens160
       valid_lft 79564sec preferred_lft 79564sec
    inet6 fe80::1ae4:952f:402d:6b1/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
default qlen 1000
    link/ether 00:50:56:b3:26:b6 brd ff:ff:ff:ff:ff:ff
    inet 192.168.50.183/24 brd 192.168.50.255 scope global dynamic noprefixroute
ens192
       valid_lft 70687sec preferred_lft 70687sec
    inet6 fe80::4c87:a2b3:aa9:5470/64 scope link noprefixroute
       valid_lft forever preferred_lft forever


student@CCNP:~$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=255 time=0.703 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=255 time=0.748 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=255 time=0.757 ms
^C
--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2033ms
rtt min/avg/max/mdev = 0.703/0.736/0.757/0.023 ms
```

c. If the **CCNP VM** has not received IPv4 addressing, check your physical connections between the host PC and R1. Also, verify that R1 is configured correctly according to the previous step.

Step 5: Establish an SSH Connection to R1.

a. Open the PuTTY SSH Client.

b. Enter the IPv4 address for the default gateway, 192.168.1.1, and click **Open**.

c. You should be able to login to R1 with the username **cisco** and password **cisco123!**. If not, verify that your SSH configuration is correct on R1.

d. Keep your PuTTY session open.

Part 2: Use a NETCONF Session to Gather Information

In Part 2, you will check to see if NETCONF is already running, enable NETCONF if it is not, and verify that NETCONF is ready for an SSH connection. Then you will connect the NETCONF process, start a NETCONF session, gather interface information, and close the session.

Step 1: Check if NETCONF is running on R1.

a. NETCONF may already be running if another student enabled it, or if a later IOS version enables it by default. From the PuTTY terminal, use the **show platform software yang-management process** command to see if the NETCONF SSH daemon (**ncsshd**) is running.

```
R1# show platform software yang-management process
confd            : Not Running
nesd             : Not Running
syncfd           : Not Running
ncsshd           : Not Running
dmiauthd         : Not Running
nginx            : Running
ndbmand          : Not Running
pubd             : Not Running
```

b. If NETCONF is not running, as shown in the output above, enter the global configuration command **netconf-yang**.

```
R1# config t
R1(config)# netconf-yang
```

c. Now verify that NETCONF is running again.

```
R1# show platform software yang-management process
confd            : Running
nesd             : Running
syncfd           : Running
ncsshd           : Running
dmiauthd         : Running
nginx            : Running
ndbmand          : Running
pubd             : Running
```

Step 2: Access the NETCONF process through an SSH terminal.

You can establish an SSH session with R1 using PuTTY. However, the copying and pasting functionality in the PuTTY terminal window inside the **CCNP VM** can be problematic. Therefore, use a command line window to start the SSH session with R1.

a. Enter the following command in a terminal window. Then enter **cisco123!** as the password. If you type the command incorrectly, use the up-arrow key to recall the command that you enter, then find and edit your mistake.

```
student@CCNP:~$ ssh -oHostKeyAlgorithms=+ssh-dss cisco@192.168.1.1 -p 830 -s
netconf
cisco@192.168.1.1's password:
```

b. R1 will respond with a hello message that includes all of its capabilities. The end of the message is identified with **]]>]]>**.

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.0</capability>
<capability>urn:ietf:params:netconf:base:1.1</capability>
<capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
<capability>urn:ietf:params:netconf:capability:xpath:1.0</capability>
<capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
<capability>urn:ietf:params:netconf:capability:validate:1.1</capability>
<capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
<capability>urn:ietf:params:netconf:capability:notification:1.0</capability>
<capability>urn:ietf:params:netconf:capability:interleave:1.0</capability>
<capability>urn:ietf:params:netconf:capability:with-defaults:1.0?basic-
mode=explicit&amp;also-supported=report-all-tagged</capability>
<capability>urn:ietf:params:netconf:capability:yang-library:1.0?revision=2016-06-
21&amp;module-set-id=d26d4d9ff23c0afcad7994ca2b832a9b</capability>
<capability>http://tail-f.com/ns/netconf/actions/1.0</capability>
<capability>http://tail-f.com/ns/netconf/extensions</capability>
(output omitted)
<capability>urn:ietf:params:xml:ns:yang:ietf-netconf-with-defaults?module=ietf-
netconf-with-defaults&amp;revision=2011-06-01</capability>
<capability>
        urn:ietf:params:netconf:capability:notification:1.1
<output omitted>
      </capability>
</capabilities>
<session-id>20</session-id></hello>]]>]]>
```

Step 3: Start a NETCONF session by sending a hello message from the client.

To start a NETCONF session, the client needs to send its own hello message. The hello message should include the NETCONF base capabilities version the client wants to use.

a. Copy and paste the following XML code into the SSH session. Notice that the end of the client hello message is identified with a **]]>]]>**.

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <capabilities>
   <capability>urn:ietf:params:netconf:base:1.0</capability>
 </capabilities>
</hello>
]]>]]>
```

b. On R1, use the **show netconf-yang sessions** command to verify that a NETCONF session has been started.

```
R1# show netconf-yang sessions
R: Global-lock on running datastore
```

```
C: Global-lock on candidate datastore
S: Global-lock on startup datastore

Number of sessions : 1

session-id  transport    username             source-host           global-lock
--------------------------------------------------------------------------------
20          netconf-ssh  cisco                192.168.1.12          None
```

Step 4: Send RPC messages to an IOS XE device.

During an SSH session, a NETCONF client can use Remote Procedure Call (RPC) messages to send NETCONF operations to the IOS XE device. The table lists some of the more common NETCONF operations.

Operation	Description
<get>	Retrieve running configuration and device state information
<get-config>	Retrieve all or part of specified configuration data store
<edit-config>	Loads all or part of a configuration to the specified configuration data store
<copy-config>	Replace an entire configuration data store with another
<delete-config>	Delete a configuration data store
<commit>	Copy candidate data store to running data store
<lock> / <unlock>	Lock or unlock the entire configuration data store system
<close-session>	Graceful termination of NETCONF session
<kill-session>	Forced termination of NETCONF session

a. Copy and paste the following RPC get message XML code into the terminal SSH session to retrieve information about the interfaces on R1.

```
<rpc message-id="103" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <get>
  <filter>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"/>
   </filter>
 </get>
</rpc>
]]>]]>
```

b. Recall that XML does not require indention or white space. Therefore, R1 will return a long string of XML data.

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-
id="103"><data><interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-
interfaces"><interface><name>GigabitEthernet0</name><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"></ipv4><ipv6
xmlns="urn:ietf:params:xml:ns:yang:ietf-
ip"></ipv6></interface><interface><name>GigabitEthernet0/0/0</name><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"></ipv4><ipv6
xmlns="urn:ietf:params:xml:ns:yang:ietf-
ip"></ipv6></interface><interface><name>GigabitEthernet0/0/1</name><description>Link
to PC</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"><address><ip>192.168.1.1</ip><netmask>255.255.255.0</netmask></address></ipv4><ipv
6 xmlns="urn:ietf:params:xml:ns:yang:ietf-
ip"></ipv6></interface></interfaces></data></rpc-reply>]]>]]>
```

c. Copy the XML that was returned, but do not include the final “]]>]]>” characters. These characters are not part of the XML that is returned by the router.

d. Search the internet for “prettify XML”. Find a suitable site and use its tool to transform your XML into a more readable format, such as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="103">
   <data>
      <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
         <interface>
            <name>GigabitEthernet0</name>
            <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" />
            <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip" />
         </interface>
         <interface>
            <name>GigabitEthernet0/0/0</name>
            <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" />
            <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip" />
         </interface>
         <interface>
            <name>GigabitEthernet0/0/1</name>
            <description>Link to PC</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">
               <address>
                  <ip>192.168.1.1</ip>
                  <netmask>255.255.255.0</netmask>
               </address>
            </ipv4>
            <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip" />
         </interface>
      </interfaces>
   </data>
</rpc-reply>
```

Step 5: Close the NETCONF session.

To close the NETCONF session, the client needs to send the following RPC message:

```
<rpc message-id="9999999" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <close-session />
</rpc>
```

You will be returned to the terminal prompt. Return to the router prompt and show the open netconf sessions. You will see that the session has been closed.

Part 3: Use ncclient to Connect to NETCONF

Working with NETCONF does not require working with raw NETCONF RPC messages and XML. In Part 3, you will learn how to use the **ncclient** Python module to easily interact with network devices using NETCONF. In addition, you will learn how to identify which YANG models are supported by the device. This information is helpful when building a production network automation system that requires specific YANG models to be supported by the given network device.

Step 1: Verify that ncclient is installed and ready for use.

Enter the command **pip3 list --format=columns** to see all Python modules currently installed in the CCNP VM. Your output may differ from the following. But you should see **ncclient** listed, as shown. If not, use the **sudo pip3 install ncclient** command to install it.

```
student@CCNP:~$ pip3 list --format=columns
Package               Version
--------------------- ------------------
apturl                0.5.2
asn1crypto            0.24.0
(output omitted)
ncclient              0.6.7
netifaces             0.10.4
netmiko               3.0.0
oauth                 1.0.1
olefile               0.45.1
paramiko              2.7.1
pexpect               4.2.1
Pillow                5.1.0
pip                   9.0.1
(output omitted)
requests              2.22.0
requests-unixsocket   0.1.5
scp                   0.13.2
(output omitted)
xmltodict             0.12.0
zope.interface        4.3.2
```

Step 2: Create a script to use ncclient to connect the NETCONF service.

The ncclient module provides a **manager** class with a **connect()** method to setup the remote NETCONF connections. After a successful connection, the returned object represents the NETCONF connection to the remote device.

a. In Python IDLE, create a new Python script file called **ncclient-netconf.py**.

b. In the new Python script file editor, import the manager class from the **ncclient** module. Then create a variable **m** to represent the **connection()** method. The **connection()** method includes all the information that is required to connect to the NETCONF service running on R1. Note that the port is 830 for NETCONF.

```
from ncclient import manager


m = manager.connect(
    host="192.168.1.1",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )
```

If the **hostkey_verify** is set to True, R1 will ask you to verify the SSH fingerprint. In a lab environment, it is safe to set this value to False, as we have done here.

c. Save and run the program to verify that there are no errors. You will not see any output yet. But you can verify that the NETCONF session is active on R1 by entering the **show netconf-yang sessions** command.

```
================= RESTART: /home/student/ncclient-netconf.py =================
>>>
```

Step 3: Add a print function to the script so that the NETCONF capabilities for R1 are listed.

The **m** object returned by the **manager.connect()** function represents the NETCONF remote session. As you saw in Part 2, in every NETCONF session, the server first sends its list of capabilities which is a list, in XML format, of supported YANG models. With the **ncclient** module, the received list of capabilities is stored in the **m.server_capabilities** list.

a. Use a **for** loop and a print function to display the device capabilities:

```
print("#Supported Capabilities (YANG models):")
for capability in m.server_capabilities:
    print(capability)
```

b. Save and run the program. The output is the same output you got from sending the complex hello message in Part 2, Step 3, but without the opening and closing <capability> XML tag on each line.

```
================= RESTART: /home/student/ncclient-netconf.py =================
#Supported Capabilities (YANG models):
urn:ietf:params:netconf:base:1.0
urn:ietf:params:netconf:base:1.1
urn:ietf:params:netconf:capability:writable-running:1.0
urn:ietf:params:netconf:capability:xpath:1.0
urn:ietf:params:netconf:capability:validate:1.0
urn:ietf:params:netconf:capability:validate:1.1
urn:ietf:params:netconf:capability:rollback-on-error:1.0
urn:ietf:params:netconf:capability:notification:1.0
Urn:ietf:params:netconf:capability:interleave:1.0
<output omitted>
```

Part 4: Use ncclient to Retrieve the Configuration

In Part 4, you will use the NETCONF ncclient to retrieve the configuration of R1, update the configuration, and create a new interface. You will also learn why the transactional support of NETCONF is important for getting consistent network changes.

Step 1: Use the get_config() function to retrieve the running configuration for R1.

a. You can use the **get_config()** method of the **m** NETCONF session object to retrieve the configuration for R1. The **get_config()** method expects a source string parameter that specifies the source NETCONF datastore. Use a print function to display the results. The only NETCONF datastore currently on R1 is the running datastore. You can verify this with the **show netconf-yang datastores** command. If you want to skip displaying the output from Part 3, comment out the block of statements that print the capabilities, as shown in the following:

```
'''
print("#Supported Capabilities (YANG models):")
for capability in m.server_capabilities:
    print(capability)
'''

netconf_reply = m.get_config(source="running")
print(netconf_reply)
```

b. Save and run your program. The output will be well over 100 lines, so IDLE may compress them. Double-click the **Squeezed text** message in the IDLE shell window to expand the output.

```
================= RESTART: /home/student/ncclient-netconf.py =================
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-
id="urn:uuid:b146b877-ba9a-45bc-8219-31732d1708dd"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"><data><native
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native"><version>16.9</version><boot-
start-marker/><boot-end-
marker/><service><timestamps><debug><datetime><msec></msec></datetime></debug><log><da
tetime><msec/></datetime></log></timestamps></service><hostname>R1</hostname>
<output omitted>
```

c. Notice that the returned XML is not formatted. You can copy it out to the same site you found in Part 2 to prettify the XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="urn:uuid:b146b877-ba9a-
45bc-8219-31732d1708dd">
   <data>
      <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
         <version>16.9</version>
         <boot-start-marker />
         <boot-end-marker />
         <service>
            <timestamps>
               <debug>
                  <datetime>
                     <msec />
                  </datetime>
```

```
                </debug>
                <log>
                   <datetime>
                      <msec />
                   </datetime>
                </log>
             </timestamps>
          </service>
          <hostname>R1</hostname>
<output omitted>
```

Step 2: Use Python to prettify the XML.

Python has built in support for working with XML files. The **xml.dom.minidom** module can be used to prettify the output with the **toprettyxml()** function.

a. At the beginning of your script, add a statement to import the **xml.dom.minidom** module. Then replace the simple print function **print(netconf_reply)** with a version that prints prettified XML output.

```
import xml.dom.minidom
print(xml.dom.minidom.parseString(netconf_reply.xml).toprettyxml())
```

b. Save and run your program. Expand the output to see the XML displayed in a more readable format.

```
================== RESTART: /home/student/ncclient-netconf.py ==================
<?xml version="1.0" ?>
<rpc-reply message-id="urn:uuid:0fc605b5-bf01-44c2-9830-1c092db3afa7"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <version>16.9</version>
            <boot-start-marker/>
            <boot-end-marker/>
            <service>
                <timestamps>
                    <debug>
                        <datetime>
                            <msec/>
                        </datetime>
                    </debug>
                    <log>
                        <datetime>
                            <msec/>
                        </datetime>
                    </log>
                </timestamps>
            </service>
            <hostname>R1</hostname>
```

Step 3: Use a filter with get_config() to only retrieve a specific YANG model.

A network administrator may only want to retrieve a portion of the running configuration on a device. NETCONF supports returning only data that is defined in a filter parameter of the **get_conf()** function.

a. Create a variable called **netconf_filter** that only retrieves data defined by the Cisco IOS XE Native YANG model.

```
netconf_filter = """
<filter>
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native" />
</filter>
"""
netconf_reply = m.get_config(source="running", filter=netconf_filter)
```

b. Save and run your program. Expand the output to see the XML displayed in a more readable format. The start of the output is the same, as shown below. However, the rest of the output only includes specified YANG models.

```
================= RESTART: /home/student/ncclient-netconf.py =================
<?xml version="1.0" ?>
<rpc-reply message-id="urn:uuid:a3e46a28-07b4-4e8d-964c-647cb565e7a1"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <version>16.9</version>
            <boot-start-marker/>
            <boot-end-marker/>
            <service>
                <timestamps>
                    <debug>
                        <datetime>
                            <msec/>
                        </datetime>
                    </debug>
                    <log>
                        <datetime>
                            <msec/>
                        </datetime>
                    </log>
                </timestamps>
            </service>
            <hostname>R1</hostname>
(output omitted)
```

c. Filtering the retrieved data to only display the native YANG module significantly reduces your output. This is because the native YANG module only includes a subset of all the Cisco IOX XE YANG models.

Part 5: Use ncclient to Configure a Device

In Part 5, you will use **ncclient** to configure R1 using the **edit_config()** method of the **manager** module.

Step 1: Use ncclient to edit the hostname for R1.

Note: It is fine if multiple students are accessing R1 at the same time and changing the hostname. You will be looking for the **<ok/>** response from NETCONF which confirms your configuration was sent and applied successfully. The current hostname is unimportant.

a. To update an existing setting in the configuration for R1, you can extract the setting location from the configuration retrieved in Part 4. For this step, you will set a variable to change the **<hostname>** value.

```
================= RESTART: /home/student/ncclient-netconf.py =================
<?xml version="1.0" ?>
<rpc-reply message-id="urn:uuid:a3e46a28-07b4-4e8d-964c-647cb565e7a1"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
(output omitted)
            <hostname>R1</hostname>
(output omitted)
```

b. Previously, you defined a <filter> variable. To modify a device configuration, you will define a <config> variable. Add the following variable to your **ncclient_netconf.py** script. You can use NEWHOSTNAME or whatever hostname you wish.

```
netconf_hostname = """
<config>
  <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
     <hostname>NEWHOSTNAME</hostname>
  </native>
</config>
"""
```

c. Use the **edit_config()** function of the **m** NETCONF session object to send the configuration and store the results in the **netconf_reply** variable so that they can be printed. The parameters for the **edit_config()** function are as follows:

- **target** - the targeted NETCONF datastore to be updated
- **config** - the configuration modification that is to be sent

```
netconf_reply = m.edit_config(target="running", config=netconf_hostname)
```

d. The **edit_config()** function returns an XML RPC reply message with **<ok/>** indicating that the change was successfully applied. Repeat the previous print statement to display the results.

```
print(xml.dom.minidom.parseString(netconf_reply.xml).toprettyxml())
```

e. Save and run your program. You should get output similar to the output displayed below. You can also verify that the hostname for R1 has changed by accessing the R1 CLI.

```
================= RESTART: /home/student/ncclient-netconf.py =================
<?xml version="1.0" ?>
<rpc-reply message-id="urn:uuid:8d8e1f04-1aa4-47c9-92d5-2a8981322f8f"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <ok/>
</rpc-reply>


>>>
```

f. Change the hostname of the router back to R1.

Step 2: Use ncclient to create a new loopback interface on R1.

a. Comment out the code from the previous step if you want to avoid changing the hostname again.

b. Create a new <config> variable to hold the configuration for a new loopback interface. Add the following to your **ncclient_netconf.py** script. If multiple students are accessing R1 at the same time, use the loopback assigned to you by your instructor. Otherwise, you can use loopback 1, as shown below.

Note: Replace [Student Name] with your name. Leave the backslash (\ )in the **description** command. The escape character, backslash (\ ), allows an alternative interpretation of the single quote (') as the apostrophe. The escape characters allow special characters to be used inside a string declaration. The single quote is normally used to note the beginning or the end of a string declaration.

```
netconf_loopback = """
<config>
 <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <interface>
   <Loopback>
    <name>1</name>
    <description>[Student Name]\'s loopback</description>
    <ip>
     <address>
      <primary>
       <address>10.1.1.1</address>
       <mask>255.255.255.0</mask>
      </primary>
     </address>
    </ip>
   </Loopback>
  </interface>
 </native>
</config>
"""
```

c. Comment out the previous netconf_reply variable if you want to avoid changing the hostname again. Add the following **edit_config()** function to your **ncclient_netconf.py** to send the new loopback configuration to R1 and then print out the results.

```
netconf_reply = m.edit_config(target="running", config=netconf_loopback)
print(xml.dom.minidom.parseString(netconf_reply.xml).toprettyxml())
```

d. Save and run your program. You should get output similar to the following:

```
================== RESTART: /home/student/ncclient-netconf.py ==================
<?xml version="1.0" ?>
<rpc-reply message-id="urn:uuid:6407b416-2851-4eca-bbfa-7afbc6e8fbcf"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <ok/>
</rpc-reply>


>>>
```

e. On R1, verify that the new loopback interface was created.

```
R1# show ip interface brief
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0/0   unassigned      YES unset  down                  down
```

```
GigabitEthernet0/0/1   192.168.1.1     YES manual up                    up
Serial0/1/0            unassigned      YES unset  up                    up
Serial0/1/1            unassigned      YES unset  up                    up
GigabitEthernet0       unassigned      YES unset  down                  down
Loopback1              10.1.1.1        YES other  up                    up
```

Step 3: Attempt to create a new loopback interface with the same IPv4 address.

a. Create a new variable called **netconf_newloop**. It will hold a configuration that creates a new loopback 2 interface but with the same IPv4 address as on loopback 1: 10.1.1.1./24. At the router CLI, this would create an error because of the attempt to assign a duplicate IP address to an interface.

Note: If multiple students are accessing R1 at the same time, increment by 1 the loopback assigned to you by your instructor. Otherwise, you can use loopback 2, as shown below.

```
netconf_newloop = """
<config>
 <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <interface>
   <Loopback>
    <name>2</name>
    <description>[Student Name]'s loopback</description>
    <ip>
     <address>
      <primary>
       <address>10.1.1.1</address>
       <mask>255.255.255.0</mask>
      </primary>
     </address>
    </ip>
   </Loopback>
  </interface>
 </native>
</config>
"""
```

b. Add the following edit_config() function to your ncclient_netconf.py to send the new loopback configuration to R1. You do not need a print statement for this step.

```
netconf_reply = m.edit_config(target="running", config=netconf_newloop)
```

c. Save and run the program. You should get error output similar to the following:

```
============ RESTART: /home/student/Documents/ncclient-netconf.py ============
Traceback (most recent call last):
  File "/home/student/Documents/ncclient-netconf.py", line 71, in <module>
    netconf_reply2=m.edit_config(target='running', config = netconf_newloop)
  File "/home/student/.local/lib/python3.6/site-packages/ncclient/manager.py", line
236, in execute
     huge_tree=self._huge_tree).request(*args, **kwds)
  File "/home/student/.local/lib/python3.6/site-packages/ncclient/operations/edit.py",
line 69, in request
     return self._request(node)
```

```
ncclient.operations.rpc.RPCError: inconsistent value: Device refused one or more
commands
>>>
```

d. Unlike using **netmiko** or copying and pasting a script, NETCONF will not apply any of the configuration that is sent if one or more commands are rejected. To verify this, enter the **show ip interface brief** command on R1. Notice that your new interface was not created.

```
R1# show ip interface brief
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0/0   unassigned      YES unset  down                  down
GigabitEthernet0/0/1   192.168.1.1     YES manual up                    up
Serial0/1/0            unassigned      YES unset  up                    up
Serial0/1/1            unassigned      YES unset  up                    up
GigabitEthernet0       unassigned      YES unset  down                  down
Loopback1              10.1.1.1        YES other  up                    up
```

Part 6: Modify the Program Used in this Lab

The following is the complete program that was created in this lab. The code from Part 5, Step 3 is commented so that you can run the script without error. Your script may look different. Practice your Python skills by modifying the program to send different verification and configuration commands.

```
from ncclient import manager
import xml.dom.minidom

m = manager.connect(
    host="192.168.1.1",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )

print("#Supported Capabilities (YANG models):")
for capability in m.server_capabilities:
    print(capability)


netconf_reply = m.get_config(source="running")
print(xml.dom.minidom.parseString(netconf_reply.xml).toprettyxml())

netconf_filter = """
<filter>
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native" />
</filter>
"""
netconf_reply = m.get_config(source="running", filter=netconf_filter)
print(xml.dom.minidom.parseString(netconf_reply.xml).toprettyxml())
```

```
netconf_hostname = """
<config>
  <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
     <hostname>NEWHOSTNAME</hostname>
  </native>
</config>
"""

netconf_reply = m.edit_config(target="running", config=netconf_hostname)
print(xml.dom.minidom.parseString(netconf_reply.xml).toprettyxml())

netconf_loopack = """
<config>
 <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <interface>
   <Loopback>
    <name>1</name>
    <description>[Student Name]'s loopback</description>
    <ip>
     <address>
      <primary>
       <address>10.1.1.1</address>
       <mask>255.255.255.0</mask>
      </primary>
     </address>
    </ip>
   </Loopback>
  </interface>
 </native>
</config>
"""

netconf_reply = m.edit_config(target="running", config=netconf_loopack)
print(xml.dom.minidom.parseString(netconf_reply.xml).toprettyxml())

'''
netconf_newloop = """
<config>
 <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <interface>
   <Loopback>
    <name>2</name>
    <description>[Student Name]’s loopback</description>
    <ip>
```

```
      <address>
       <primary>
        <address>10.1.1.1</address>
        <mask>255.255.255.0</mask>
       </primary>
      </address>
     </ip>
    </Loopback>
   </interface>
  </native>
</config>
"""

netconf_reply = m.edit_config(target="running", config=netconf_newloop)
'''
```

Router Interface Summary Table

Router Model	Ethernet Interface #1	Ethernet Interface #2	Serial Interface #1	Serial Interface #2
1800	Fast Ethernet 0/0 (F0/0)	Fast Ethernet 0/1 (F0/1)	Serial 0/0/0 (S0/0/0)	Serial 0/0/1 (S0/0/1)
1900	Gigabit Ethernet 0/0 (G0/0)	Gigabit Ethernet 0/1 (G0/1)	Serial 0/0/0 (S0/0/0)	Serial 0/0/1 (S0/0/1)
2801	Fast Ethernet 0/0 (F0/0)	Fast Ethernet 0/1 (F0/1)	Serial 0/1/0 (S0/1/0)	Serial 0/1/1 (S0/1/1)
2811	Fast Ethernet 0/0 (F0/0)	Fast Ethernet 0/1 (F0/1)	Serial 0/0/0 (S0/0/0)	Serial 0/0/1 (S0/0/1)
2900	Gigabit Ethernet 0/0 (G0/0)	Gigabit Ethernet 0/1 (G0/1)	Serial 0/0/0 (S0/0/0)	Serial 0/0/1 (S0/0/1)
4221	Gigabit Ethernet 0/0/0 (G0/0/0)	Gigabit Ethernet 0/0/1 (G0/0/1)	Serial 0/1/0 (S0/1/0)	Serial 0/1/1 (S0/1/1)
4300	Gigabit Ethernet 0/0/0 (G0/0/0)	Gigabit Ethernet 0/0/1 (G0/0/1)	Serial 0/1/0 (S0/1/0)	Serial 0/1/1 (S0/1/1)

Note: To find out how the router is configured, look at the interfaces to identify the type of router and how many interfaces the router has. There is no way to effectively list all the combinations of configurations for each router class. This table includes identifiers for the possible combinations of Ethernet and Serial interfaces in the device. The table does not include any other type of interface, even though a specific router may contain one. An example of this might be an ISDN BRI interface. The string in parenthesis is the legal abbreviation that can be used in Cisco IOS commands to represent the interface.