KEYSIGHT

NAF AUTOCON2: WS:D1 LAB INSTRUCTIONS DOCUMENT

INTRODUCTION

This document describes the Ixia-C Community Edition / Keysight Elastic Network Generator (in short KENG) lab exercises for the 2024 AutoCon 2 WS-D1 workshop. It covers the environment configuration required for the lab exercises, the infrastructure tools required to manage the test topology, and the test tools required to generate traffic and emulate protocols.

The document assumes the students have prior knowledge about Linux OS configuration, about container / virtual machine concepts, and about network test principles. The lab exercises start from a single all-in-one Ubuntu Server 22.04 instance with minimal installation (provisioned with 8 CPU / 16 GB RAM / 128 GB HDD) and instructs the students on how to configure all elements required for a successful test execution (including the scripting clients, the network topology, the test tools, and the devices under test).

Lab	OTG Test Tool	OTG Test Tool Components	OTG API Client	Infrastructure	DUT	Learning Objective	Duration
00	n / a	n / a	n / a	n / a	n / a	DOCKER PYTHON	~ 10 min
01	Ixia-c	KENG Controller Ixia-C Traffic Engine	OTGEN + SNAPPI	DOCKER CLIENT	B2B	SNAPPI IXIA-C OTGEN	~ 30 min
02	Ixia-c	KENG Controller Ixia-C Traffic Engine Ixia-C Protocol Engine	SNAPPI	DOCKER COMPOSE	B2B	DOCKER COMPOSE SNAPPI PROTOCOLS SNAPPI CAPTURES REST STATES REST STATS	~ 20 min
03	Ixia-c	KENG Controller Ixia-C Traffic Engine Ixia-C Protocol Engine	SNAPPI	CONTAINERLAB	Nokia SRL	CONTAINERLAB IXIA-C-ONE DEPLOYMENT EGRESS TRACKING	~ 20 min
04	Ixia-c	Ixia-C-One	SNAPPI	CONTAINERLAB	Nokia SRL	SNAPPI PROTOCOLS SNAPPI TRAFFIC CONTROL ACTIONS	~ 20 min
05	Ixia-c	KENG Operator KENG Controller Ixia-C GNMI Server Ixia-C Traffic Engine Ixia-C Protocol Engine	SNAPPI GOSNAPPI	KIND / KNE	B2B	IXIA-C IN K8S KENG-OPERATOR GOSNAPPI GRPC	~ 20 min
Demo	HW Ports	KENG Controller Ixia-C GNMI Server KENG-Layer23-HW Server	SNAPPI	DOCKER COMPOSE	HW DUT	KENG for HW	DEMO ~ 10 min

The previous table includes the lab exercises proposed for this training session. Please make sure to execute each command with its full syntax and understand why every step is necessary. In case the output of one command is different than expected please ask the instructor for clarifications.

|		
✅	This section summarizes the objectives of the lab exercise.	HINT: Carefully read this section to understand the goals and structure of the lab.
▶_	This section includes commands that must be executed exactly as-is.	
ℹ️	This section includes additional information for future study.	HINT: Bookmark the links for future reference.
⚠️	This section includes frequent errors or other caveats.	HINT: Pay attention to these quirks and avoid them whenever possible.
❓	This section includes questions about the lab exercise.	HINT: Answer the question for yourself before proceeding forward.

Lab #00 – ENVIRONMENT CONFIGURATION

> This lab starts from a minimal installation of Ubuntu 22.04 and guides the students through the configuration of most important prerequisites for the rest of the labs.
>
> ☑ Please note that a typical test environment usually consists of one separate host for script execution and other hosts for running the test topology. The following lab exercises will however utilize an all-in-one deployment which combines the script execution environment with the test topology environment. This is very useful for learning purposes, and it is also an environment used by some network developers.
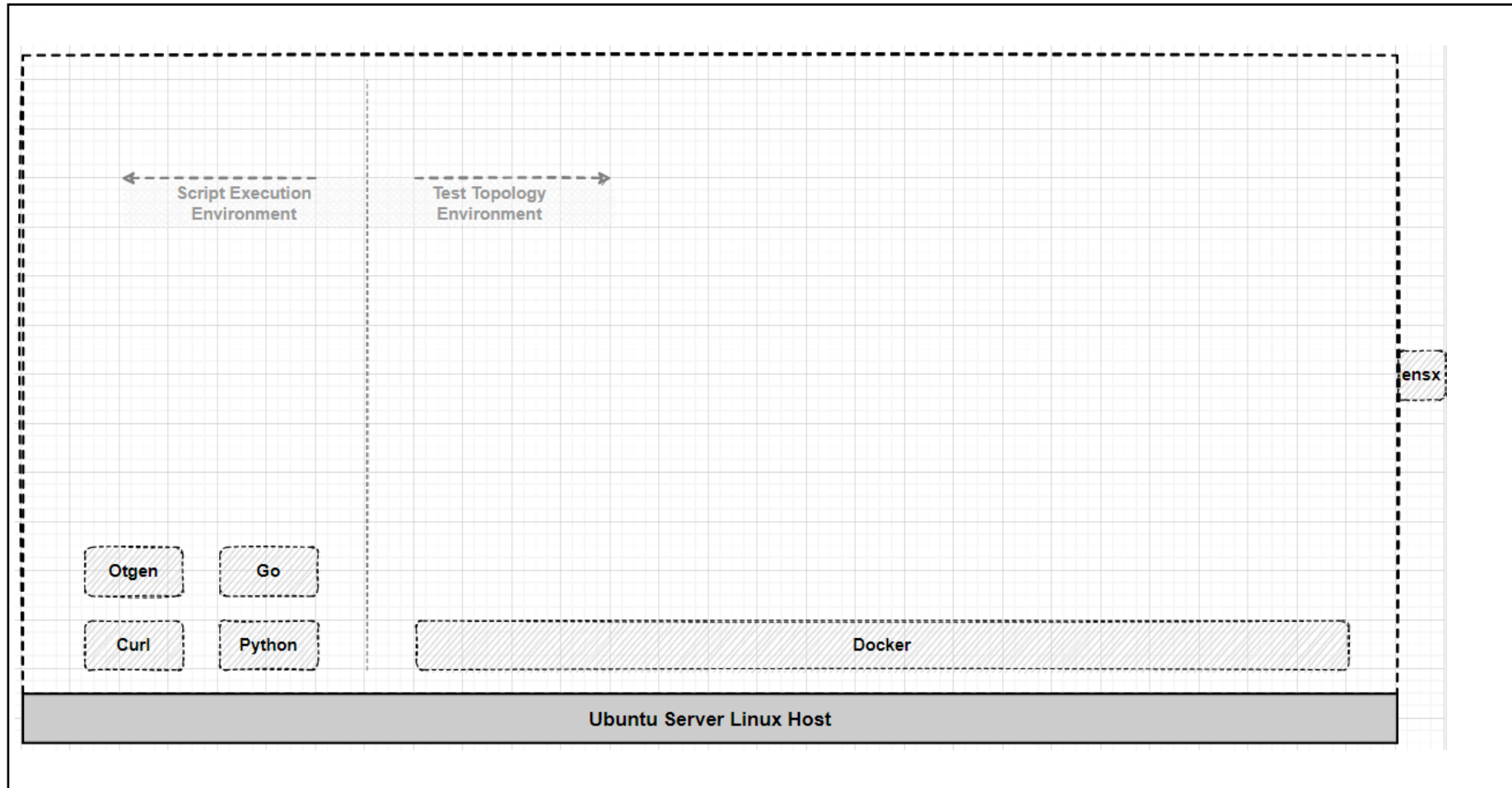
PREPARATION: Connect to your lab machine

Login to the console of your assigned server. Feel free to use any SSH client terminal with the provided key. You should find a CloudShare link in your email with instructions for connecting to your virtual environment. If it's not there, please reach out to your instructor.

```
history
ip address
```
#0-01

PREPARATION: Check DOCKER

Docker and docker-compose were installed as Ubuntu packages (e.g. "sudo apt install docker.io && sudo apt install docker-compose"). Check the version of the Docker Container Engine already installed on this machine.

```
docker version && docker-compose version
```
#0-02

Verify the list of container images which are already loaded in the local registry. You will notice no images exist in the registry.

```
docker images
```
#0-03

Verify the list of containers which are already running on this host. You will notice there are no running containers.

```
docker ps
```
#0-04

PREPARATION: Check PYTHON and clone repository

Check the version of Python already installed on this Ubuntu machine.

```
python3 --version
```
#0-05

Install the snappi library.

```
python3 -m pip install snappi==1.14.0
```
#0-06

ℹ️ SNAPPI is an auto-generated python SDK which can be executed against any traffic generator conforming to OTG API standard

---

Clone the git repository associated with this workshop

```
git clone https://github.com/open-traffic-generator/ac2-workshop.git
```
#0-07

Lab #01 – DOCKER RUN SIMPLE B2B TRAFFIC

This lab uses snappi to control the free Ixia-c Community Edition (OTG Test Tool) which is deployed via plain Docker Engine commands and utilized to send raw traffic in a back-to-back topology. This lab consists of 1x KENG Controller and 2x Ixia-c Traffic Engine containers.

☑ The test script has been already created before this lab. The test only includes raw traffic (no protocol emulation) and performs the following actions:
  ➢ Validates that total packet sent and received on both interfaces is as expected using the port metrics.
  ➢ Sends 2000 packets between the two ports at a rate of 100 packets per second for a total of 20 seconds.
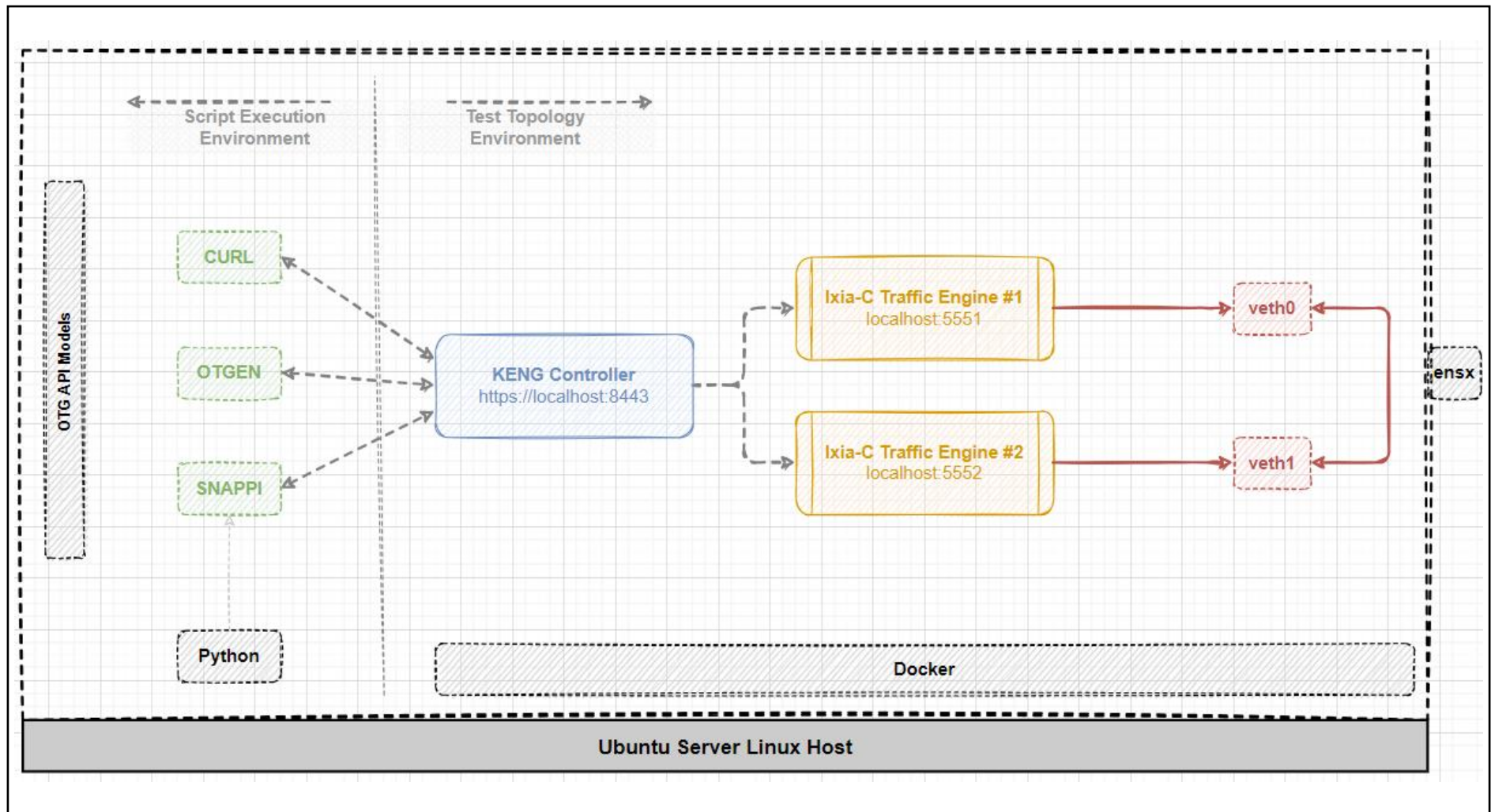
CONFIGURATION: Download docker images

> ℹ️ Keysight publishes a free version of its software on the Git Hub Container Registry. The free version supports control plane BGP emulation and full data plane traffic capabilities - https://ixia-c.dev/#community-edition. New versions are published every few weeks. Additional information can be found on the ixia-c releases page

Pull the desired version of the Ixia-c Controller and Ixia-c Traffic Engine from the GitHub Container Registry.

```
docker pull ghcr.io/open-traffic-generator/keng-controller:1.14.0-1

docker pull ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
```
#1-01

Check once again the list of images which are loaded in the local registry and the list of containers which are already running.

```
docker images

docker ps
```
#1-02

> ⚠️ Please pay close attention to the version tag associated with these images. You will notice the most recent versions have been downloaded by using a specific version tag. It is worth mentioning that the OTG API model is under active development, and it is often advisable to pull a specific software version by using the exact version tag instead of latest version

CONFIGURATION: Create the test interfaces

A virtual interface pair must be created on the server. These will be sending and receiving traffic. Use the "ip link" command to see the newly created virtual interfaces

```
sudo ip link add name veth0 type veth peer name veth1&& sudo ip link set dev veth0 up && sudo ip link set dev veth1 up

ip link
```
#1-03

DEPLOYMENT: Start the IXIA-C containers

One basic way to deploy the test tool is to start the containers one by one. During this process multiple parameters must be specified. Start the KENG Controller.

```
docker run -d --name controller --network=host ghcr.io/open-traffic-generator/keng-controller:1.14.0-1 --http-port 8443 --accept-eula
```
#1-04

Continue the manual deployment process by starting two Ixia-c Traffic Engine containers. Analyze the parameters that have been used during the deployment.

```
docker run -d --name traffic-engine-1 --network=host -e ARG_IFACE_LIST=virtual@af_packet,veth0 -e OPT_NO_HUGEPAGES=Yes --privileged -e \
OPT_LISTEN_PORT=5551 ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
```
#1-05

```
docker run -d --name traffic-engine-2 --network=host -e ARG_IFACE_LIST=virtual@af_packet,veth1 -e OPT_NO_HUGEPAGES=Yes --privileged -e \
OPT_LISTEN_PORT=5552 ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
```
#1-06

All the previous commands specify the names of the containers, and the name of the images used to boot the container. All the containers are using host networking which is a type of network attachment that ensures direct connection between the container and the Linux host.

Each container listens for connections on a default port (TCP 8443 for Ixia-c Controller and TCP 5555 for Ixia-c Traffic Engine). The default port can be overridden with a specific command parameter as seen for the two Ixia-c Traffic Engine containers. Please note that all containers sharing the same namespace must listen on different TCP ports.

Furthermore, each Ixia-c Traffic Engine container must have one (or more) test interfaces. In this lab we are using virtual test interfaces veth0 and veth1. This means the test traffic stays inside the Linux host but in most cases, this type of test will probably use an existing interface which in turn gets connected to other network devices.

⚠️ Frequent errors encountered when starting the containers include:
(1) Not specifying the correct management network or not having reachability to that network.
(2) Not specifying the correct interfaces (their name / their type / their order) for the test networks.
(3) Not specifying the different listening ports when multiple similar containers are sharing the same namespace.
(4) Not specifying the correct version of the container image from the registry (when multiple versions of the same image exist).
(5) Not having the test interfaces successfully created before executing the docker run commands

Check the list of containers which are actively running on the host.

```
#1-07    docker ps -a
```

Notice the IMAGE used by each container and their NAMES. Notice when the containers were CREATED and what their current STATUS is.

TEST EXECUTION: Analyze the previously created SNAPPI scripts

Change directory to lab-01

```
#1-08    cd ~/ac2-workshop/lab-01
```

Inspect the contents of the test script (you can advance the output of the MORE command by pressing SPACE and you can exit by pressing the Q key).

> ℹ A rendering of the OTG models can be found here

```
#1-09    vim lab-01_test.py
```

> ℹ If you use "vi" or "vim" editors, you can exit by hitting "Esc" -> ":" -> "q!" -> Enter

> ⚠ Please pay close attention to the snappi API location at line #19 which is using the address / port of where the KENG Controller container is running. Also, under lines #32 and #33 we have the location of the Ixia-c Traffic Engine test ports.

```
print("")
api = snappi.api(location="https://127.0.0.1:8443", verify=False)
print("%s Starting connection to controller                    ... " % datetime.now())
```

```
port1, port2 = (
    configuration.ports
    .port(name="Port-1", location="127.0.0.1:5551")
    .port(name="Port-2", location="127.0.0.1:5552")
)
```

TEST EXECUTION: Run the SNAPPI script

Run the script.

```
python3 lab-01_test.py
```
#1-10

Analyze the output of the previous commands and answer the following questions.

> **(Q#01)** How much time did it take to execute the entire test?
> **(Q#02)** How much time was the test supposed to send traffic? What is the overhead from a test duration perspective?
> **(Q#03)** How often does the script fetch the statistics?

Check the interface counters and save it to a file to verify the amount of traffic that has been sent and received

```
cat /proc/net/dev > counters1.log
```
#1-11

Analyze the output of the previous commands and answer the following questions.

> **(Q#04)** Which interfaces were used to send / received traffic? How were these interfaces selected in the previous configuration?
> **(Q#05)** How many packets were sent and received on these interfaces? Does it match the value configured in the test script?
> **(Q#06)** How many bytes were sent and received on these interfaces? Does it correspond to the configured frame size?

Get the statistics from the controller user REST API curl commands

```
curl -k -d '{"choice":"flow"}' -X POST https://127.0.0.1:8443/monitor/metrics
```
#1-12
```
curl -k -d '{"choice":"port"}' -X POST https://127.0.0.1:8443/monitor/metrics
```

> ℹ️ To build the body of the REST request, you can check the get_metrics format (get_metrics).
> So you can have something like this curl -k -d '{"choice":"port","port":{"port_names":["Port-2"]}}' -X POST https://127.0.0.1:8443/monitor/metrics

> ℹ️ More than 4000 packets may be seen on the port because of the IPv6 Neighbor Discovery messages

TEST EXECUTION: Change the SNAPPI script

Edit the file with a text editor such as VIM to perform the changes below. Please use the model rendering for your reference.

    (a) Send traffic with a rate of 200 fps instead of 100 fps.
    (b) Send traffic with a duration of 5 seconds instead of 20 seconds.
    (c) Send traffic with a frame size of 512 bytes instead of 128 bytes in each direction.
    (d) Send traffic with ETH + IP packet structure instead of ETH + IP + UDP packet structure for one of the flows
    (e) Send traffic with both flows in the same direction instead of sending bidirectional traffic.

```
vim lab-01_test.py
```
#1-13

> ℹ️ For the UDP header change, remove "udp" headers from flow 1 at flow packet configuration lines
> For duration use the formula "total packets / packet rate"
> For the flow direction look at lines 50 and 51 and change the Tx Rx endpoints

Verify your changes with git diff

```
... b/lab-01/lab-01_test.py
@@ -47,12 +47,12 @@ def Traffic_Test():
         # Configure source and destination ports for each traffic flow
         flow1.tx_rx.port.tx_name = port1.name
         flow1.tx_rx.port.rx_names = [port2.name]
-        flow2.tx_rx.port.tx_name = port2.name
-        flow2.tx_rx.port.rx_names = [port1.name]
+        flow2.tx_rx.port.tx_name = port1.name
+        flow2.tx_rx.port.rx_names = [port2.name]

         # Configure packet size, rate, and duration for both flows
-        flow1.size.fixed = 128
-        flow2.size.fixed = 128
+        flow1.size.fixed = 512
+        flow2.size.fixed = 512
         for f in configuration.flows:
             # Send 2000 packets per test and then stop
-            f.duration.fixed_packets.packets = 2000
+            f.duration.fixed_packets.packets = 1000
             # Send  100 packets per second
-            f.rate.pps = 100
+            f.rate.pps = 200

         # Configure packet with Ethernet, IPv4, and UDP headers for both flows
         eth1 = flow1.packet.add().ethernet
         ip1 = flow1.packet.add().ipv4
-        udp1 = flow1.packet.add().udp

         flow2.packet.ethernet().ipv4().udp()
         eth2, ip2, udp2 = flow2.packet[0], flow2.packet[1], flow2.packet[2]
@@ -76,15 +75,11 @@ def Traffic_Test():
         ip2.src.value, ip2.dst.value = "10.0.0.2", "10.0.0.1"

         # Configure UDP Ports Source as incrementing
-        udp1.src_port.increment.start = 5100
-        udp1.src_port.increment.step  = 2
-        udp1.src_port.increment.count = 10
         udp2.src_port.increment.start = 5200
         udp2.src_port.increment.step  = 4
         udp2.src_port.increment.count = 10

         # Configure UDP Ports Destination as value list
-        udp1.dst_port.values = [6100, 6125, 6150, 6170, 6190]
         udp2.dst_port.values = [6200, 6222, 6244, 6266, 6288]
```

Use python to execute the modified script version.

```
python3 lab-01_test.py
```
#1-15

Check once again the interface counters to verify the amount of traffic that has been sent and received and compare the 2 outputs

```
cat /proc/net/dev > counters2.log
diff counters1.log counters2.log
```
#1-16

Verify once again the list of containers which are actively running on the host.

```
docker ps -a
```
#1-17

TEST EXECUTION: Run using OTGEN TOOL

Install the otgen tool

```
bash -c "$(curl -sL https://get.otgcdn.net/otgen)" -- -v 0.6.2
```
#1-18

ℹ️ OTGEN is an easy-to-use CLI utility for controlling OTG-compliant test tools

We will save the current config into a JSON file which will then feed into otgen tool

```
curl -k https://127.0.0.1:8443/config > ./lab-01-config.json
```
#1-19

We can look at the configuration once again.

```
>_   more lab-01-config.json
```
#1-20

Run the exported configuration using otgen tool and display the flow output to a table

```
>_   otgen run -k -a https://127.0.0.1:8443 -f lab-01-config.json -m flow | otgen transform -m flow  | otgen display --mode table
```
#1-21

```
+---------------------------+-----------+-----------+
|            NAME           | FRAMES TX | FRAMES RX |
+---------------------------+-----------+-----------+
| Flow #1 - Port 1 > Port 2 |      1000 |      1000 |
| Flow #2 - Port 2 > Port 1 |      1000 |      1000 |
+---------------------------+-----------+-----------+
```

Run again, but now display the port statistics to table and notice how only port 1 is sending traffic

```
>_   otgen run -k -a https://127.0.0.1:8443 -f lab-01-config.json -m port | otgen transform -m port | otgen display --mode table
```
#1-22

```
+--------+-----------+-----------+
|  NAME  | FRAMES TX | FRAMES RX |
+--------+-----------+-----------+
| Port-1 |      2000 |         0 |
| Port-2 |         0 |      2000 |
+--------+-----------+-----------+
```

Modify the JSON file to send 10000 packets at 1000 packets per second for "Flow #1 - Port 1 > Port 2", then rename the second flow to "Flow #2 - Port 1 > Port 2". This should be under lines 118 and 123 then line 280

```
>_   vi lab-01-config.json
```
#1-23

```
    "choice": "fixed",
    "fixed": 512                                      "duration": {
  },                                                    "choice": "fixed_packets",
  "rate": {                                             "fixed_packets": {
    "choice": "pps",                                      "packets": 1000,
    "pps": "1000"                                         "gap": 12
  },                                                    }
  "duration": {                                       },
    "choice": "fixed_packets",                        "metrics": {
    "fixed_packets": {                                  "enable": true,
      "packets": 10000,                                 "loss": false,
      "gap": 12                                         "timestamps": false
    }                                                 },
  },                                                  "name": "Flow #2 - Port 1 > Port 2"
  "metrics": {                                      }
    "enable": true
```

```
otgen run -k -a https://127.0.0.1:8443 -f lab-01-config.json -m flow | otgen transform -m flow  | otgen display --mode table
```
#1-24

```
+---------------------------+-----------+-----------+
|            NAME           | FRAMES TX | FRAMES RX |
+---------------------------+-----------+-----------+
| Flow #1 - Port 1 > Port 2 |     10000 |     10000 |
| Flow #2 - Port 1 > Port 2 |      1000 |      1000 |
+---------------------------+-----------+-----------+
```

We can use otgen utility to create basic configurations too. First, we need to create 3 variables which are used by the otgen tool

```
export OTG_API="https://localhost:8443"

export OTG_LOCATION_P1="localhost:5551"

export OTG_LOCATION_P2="localhost:5552"
```
#1-25

Please note these following commands will remove any existing controller configuration. Here we'll have a simple flow of 2000 packets at a rate of 100 frames per second with 1.1.1.1 as source and 2.2.2.2 as destination. We're also displaying the "flow" metrics in a table

```
otgen create flow -s 1.1.1.1 -d 2.2.2.2 -p 80 --rate 100 --count 2000 | otgen run --insecure --metrics flow | otgen transform --metrics flow –counters \
frames | otgen display --mode table
```
#1-26

You can analyze the new controller configuration created by otgen tool

```
curl -k https://127.0.0.1:8443/config
```
#1-27

CLEANUP

Stop and remove the previously created containers.

```
docker stop traffic-engine-1 traffic-engine-2 controller && docker rm traffic-engine-1 traffic-engine-2 controller
```
#1-28

Validate that the containers were indeed fully removed from the host.

```
docker ps -a
docker images
```
#1-29

Please note these commands have only removed the previously created containers. They did not remove the images from the registry.

Remove the veth link. This command will also remove the veth interfaces.

```
sudo ip link delete veth0
```
#1-30

Check the existing interfaces and make sure that veth interfaces are removed.

```
ip link && ip addr
```
#1-31

Lab #02 – DOCKER COMPOSE B2B PROTOCOLS AND TRAFFIC

This lab uses snappi to control the free Ixia-c Community Edition (OTG Test Tool) which is deployed via Docker Compose orchestration and utilized to create BGP peers send traffic in a back-to-back topology. This lab consists of 1x KENG Controller, 2x Ixia-C Traffic-Engine and 2x Ixia-C Protocol-Engine containers.

☑ This test includes traffic and protocol emulation and performs the following actions:
➢ Creates one BGPv4 peer on each pair of PE+TE containers. They both advertise several IPv4 and IPv6 unicast prefixes.
➢ Then "device" bidirectional flows are created using these routes as sources and destinations.
➢ Validates that the BGP peering gets established, routes are being received and traffic has no packet loss.
➢ Packet captures will be taken.
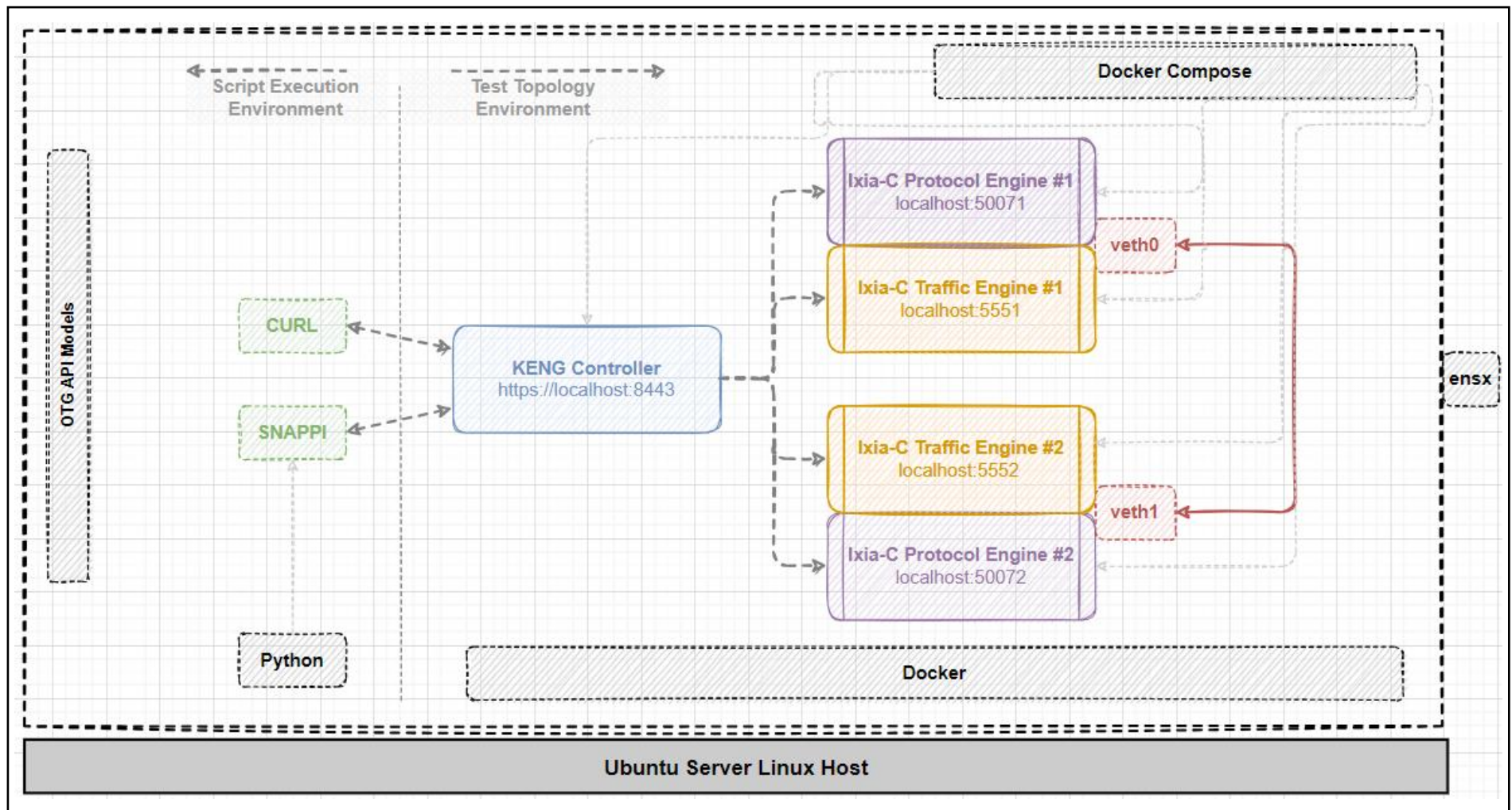
DEPLOYMENT:

`>_`	cd ~/ac2-workshop/lab-02
#2-01	cat compose.yml

Let's analyze the compose.yaml file.

> ℹ️ With docker-compose we can deploy all these containers with a single file (e.g. compose.yml). Notice how each traffic-engine and protocol-engine pair bind to the same host interface (e.g. veth0) as they share the same network.

```yaml
traffic_engine_1:
    image: ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
    restart: always
    privileged: true
    ports:
      - "5551:5551"
      - "50071:50071"          You, 7 days ago • labs added
    environment:
      - OPT_LISTEN_PORT=5551
      - ARG_IFACE_LIST=virtual@af_packet,veth0
      - OPT_NO_HUGEPAGES=Yes
      - OPT_NO_PINNING=Yes
      - WAIT_FOR_IFACE=Yes
traffic_engine_2:
    image: ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
    restart: always
    privileged: true
    ports:
      - "5552:5552"
      - "50072:50071"
    environment:
      - OPT_LISTEN_PORT=5552
      - ARG_IFACE_LIST=virtual@af_packet,veth1
      - OPT_NO_HUGEPAGES=Yes
      - OPT_NO_PINNING=Yes
      - WAIT_FOR_IFACE=Yes
protocol_engine_1:
    image: ghcr.io/open-traffic-generator/ixia-c-protocol-engine:1.00.0.405
    restart: always
    privileged: true
    network_mode: service:traffic_engine_1
    environment:
      - INTF_LIST=veth0
```

Deploy the containers using docker-compose, then run the script which creates veth0 and veth1 interfaces and inter-connects them

```
docker-compose -f compose.yml up -d

sudo bash connect_containers_veth.sh lab-02_traffic_engine_1_1 lab-02_traffic_engine_2_1 veth0 veth1
```
#2-02

ℹ️ Here "docker-compose" has been installed as a single package, however it can be installed as a docker plugin in which case will be used as "docker compose" (notice the missing dash between the two words). The docker-compose will also download any missing docker images

Check your deployment

```
docker ps
```
#2-03

```
CONTAINER ID   IMAGE                                                    COMMAND              CREATED          STATUS             PORTS
                                                                         NAMES
0adda559cb54   ghcr.io/open-traffic-generator/ixia-c-protocol-engine:1.00.0.405   "/docker_im/opt/Ixia…"   About a minute ago   Up About a minute
                                                                         lab-02_protocol_engine_1_1
99f2c492d9d0   ghcr.io/open-traffic-generator/ixia-c-protocol-engine:1.00.0.405   "/docker_im/opt/Ixia…"   About a minute ago   Up About a minute
                                                                         lab-02_protocol_engine_2_1
2dd9543da737   ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99     "./entrypoint.sh"    About a minute ago   Up About a minute   0.0.0.0:5552->5552/tcp,
 :::5552->5552/tcp, 0.0.0.0:50072->50071/tcp, [::]:50072->50071/tcp    lab-02_traffic_engine_2_1
1e8ca2af4fe5   ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99     "./entrypoint.sh"    About a minute ago   Up About a minute   0.0.0.0:5551->5551/tcp,
 :::5551->5551/tcp, 0.0.0.0:50071->50071/tcp, :::50071->50071/tcp    lab-02_traffic_engine_1_1
6b9f14c87c65   ghcr.io/open-traffic-generator/keng-controller:1.14.0-1           "./bin/controller --…"   About a minute ago   Up About a minute
                                                                         lab-02_controller_1
211d234df489   kindest/node:v1.26.0                                     "/usr/local/bin/entr…"   5 days ago       Up 5 days           127.0.0.1:34169->6443/t
cp                                                                       kne-control-plane
```

ℹ️ Unlike the previous lab where we used simple "docker run" commands with "host" docker networking, the docker-compose plugin will be using a custom bridge type for the 2 pairs of traffic-engine and protocol-engine.

ℹ️ Here the port forwarding is enabled for the PE (Protocol-Engine) and TE (Traffic-Engine) containers. The protocol-engine can only listen on 50071 so we couldn't have used default host docker network. The external (host) port will be different for each PE container.

The veth0 on TE1 and veth1 on TE2 are the test interfaces specified in the compose.yml. To inter-connect them we had to run a small utility script which uses IP namespaces (ip netns) for the TE containers.

Let's check the docker logs for traffic engine 1 to verify if the interface veth0 was found

```
>_    docker logs lab-02_traffic_engine_1_1
#2-04
```

```
2024-11-04 18:07:56.135051 [INFO] [Init] Enabling IPv6 support on the environment
2024-11-04 18:07:56.141934 [INFO] [Init] Skipped checking for DPDK drivers...
2024-11-04 18:07:56.145980 [INFO] [Init] Skipped checking for hugepages...
2024-11-04 18:07:56.172088 [INFO] [Init] Setting the listen address to *
2024-11-04 18:07:56.180219 [INFO] [Init] Setting the listen port to 5551
2024-11-04 18:07:56.190968 [INFO] [Init] Enumerating the interfaces:
2024-11-04 18:07:56.194807 [INFO] [Init] Waiting for interface veth0
2024-11-04 18:08:19.303155 [INFO] [Init] Interface veth0 found!
2024-11-04 18:08:19.326449 [INFO] [Init] Interface 1 virtual : af_packet
2024-11-04 18:08:19.330582 [INFO] [Init] No PCI interfaces present. PCI probing will be disabled
./entrypoint.sh: line 337: /sys/fs/cgroup/cpuset/cpuset.cpus: No such file or directory
```

Let's check the docker logs for traffic engine 2 to verify if the interface veth1 was found

```
>_    docker logs lab-02_traffic_engine_2_1
#2-05
```

```
2024-11-04 18:07:56.127903 [INFO] [Init] Enabling IPv6 support on the environment
2024-11-04 18:07:56.138733 [INFO] [Init] Skipped checking for DPDK drivers...
2024-11-04 18:07:56.142590 [INFO] [Init] Skipped checking for hugepages...
2024-11-04 18:07:56.168061 [INFO] [Init] Setting the listen address to *
2024-11-04 18:07:56.175443 [INFO] [Init] Setting the listen port to 5552
2024-11-04 18:07:56.185830 [INFO] [Init] Enumerating the interfaces:
2024-11-04 18:07:56.189940 [INFO] [Init] Waiting for interface veth1
2024-11-04 18:08:19.818616 [INFO] [Init] Interface veth1 found!
2024-11-04 18:08:19.844581 [INFO] [Init] Interface 1 virtual : af_packet
2024-11-04 18:08:19.848755 [INFO] [Init] No PCI interfaces present. PCI probing will be disabled
./entrypoint.sh: line 337: /sys/fs/cgroup/cpuset/cpuset.cpus: No such file or directory
2024-11-04 18:08:19.867189 [INFO] [Init] Core pinning disabled
```

⚠️ If the interfaces are not found in the logs you may have to redeploy or just restart using "docker-compose restart" command

Let's check the custom bridge network created by docker-compose

```
docker network ls
```

`docker inspect lab-02_default`

The traffic engine containers are getting assigned a management interface each.

```
"Containers": {
    "1e8ca2af4fe5258a36935be5a594c5e5ecd8427ac6de13de1718498d4c2f561e": {
        "Name": "lab-02_traffic_engine_1_1",
        "EndpointID": "c390bcd807708075c9b0626dc83e8b786aece9bfdb3a8196e0747a0df92097c2",
        "MacAddress": "02:42:ac:13:00:02",
        "IPv4Address": "172.19.0.2/16",
        "IPv6Address": ""
    },
    "2dd9543da737ed9ab81e92627ea66341f4b27cb56e2d1fa3c98a8c9bd1be7339": {
        "Name": "lab-02_traffic_engine_2_1",
        "EndpointID": "652eb7f69253825f908f96a284007233f1a615f29fecbef424c276461ca7c83d",
        "MacAddress": "02:42:ac:13:00:03",
        "IPv4Address": "172.19.0.3/16",
        "IPv6Address": ""
    }
},
"Options": {},
"Labels": {
    "com.docker.compose.network": "default",
    "com.docker.compose.project": "lab-02",
    "com.docker.compose.version": "1.29.2"
}
```

We can further inspect one of the traffic-engines and see the IP address assigned by the docker bridge
Run "ip addr" inside the container to get the IP address assigned to the management interface and we can also see the veth interface created by the utility script

```
docker exec -it lab-02_traffic_engine_1_1 ip addr
```

```
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
1309: eth0@if1310: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:13:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.19.0.2/16 brd 172.19.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe13:2/64 scope link
       valid_lft forever preferred_lft forever
1314: veth0@if1313: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:d0:87:b8:a2:ec brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::f8d0:87ff:feb8:a2ec/64 scope link
       valid_lft forever preferred_lft forever
```

EXECUTION:

Let's look at the script. Notice the KENG controller is set to "https://localhost:8443" but the ports location is different than before. This time we include the protocol engine port.

> vi lab-02_test.py

#2-08

```python
api = snappi.api(location="https://localhost:8443", verify=False)

c = ebgp_route_prefix_config(api, test_const)

api.set_config(c)

start_protocols(api)

wait_for(lambda: bgp_metrics_ok(api, test_const),"correct bgp peering")
```

The Ixia-c ports location attribute will include the port forwarding information for traffic-engine and protocol-engine containers in the format below. These are then used in the compose.yml file.

```
def ebgp_route_prefix_config(api, tc):
    c = api.config()
    ptx = c.ports.add(name="ptx", location="localhost:5551+localhost:50071")
    prx = c.ports.add(name="prx", location="localhost:5552+localhost:50072")

    # capture configuration

    rx_capture = c.captures.add(name="prx_capture")
```

```
traffic_engine_1:
    image: ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
    restart: always
    privileged: true
    ports:
        - "5551:5551"
        - "50071:50071"
    environment:
        - OPT_LISTEN_PORT=5551
        - ARG_IFACE_LIST=virtual@af_packet,veth0
        - OPT_NO_HUGEPAGES=Yes
        - OPT_NO_PINNING=Yes
        - WAIT_FOR_IFACE=Yes
traffic_engine_2:
    image: ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
    restart: always
    privileged: true
    ports:
        - "5552:5552"
        - "50072:50071"
    environment:
```

> `python3 lab-02_test.py`

#2-09

We can manually poll for various metrics and states and using REST commands.

> `curl -k -d '{"choice":"bgpv4"}' -X POST https://127.0.0.1:8443/monitor/metrics`

#2-10

For example, you can see the ARP table for the IPv4 interfaces or the learned BGP prefixes. The correct syntax for the HTTP "states" requests can be found in the OTG API model rendering page here

```
curl -k -d '{"choice":"ipv4_neighbors"}' -X POST https://127.0.0.1:8443/monitor/states
```
#2-11

```
curl -k -d '{"choice":"bgp_prefixes"}' -X POST https://127.0.0.1:8443/monitor/states
```
#2-12

Now let's capture the packets on the Rx port.
Open the file and uncomment the highlighted lines below

```
vi lab-02_test.py
```
#2-13

```python
start_protocols(api)

wait_for(lambda: bgp_metrics_ok(api, test_const),"correct bgp peering")

wait_for(lambda: bgp_prefixes_ok(api, test_const),"correct bgp prefixes")

start_capture(api)

start_transmit(api)

wait_for(lambda: flow_metrics_ok(api, test_const), "flow metrics",2,90)

stop_capture(api)

get_capture(api, "prx", "prx.pcap")
get_capture(api, "ptx", "ptx.pcap")
```

Notice the capture settings present in the snappi configuration

```
def ebgp_route_prefix_config(api, tc):
    c = api.config()
    ptx = c.ports.add(name="ptx", location="localhost:5551+localhost:50071")
    prx = c.ports.add(name="prx", location="localhost:5552+localhost:50072")

    # capture configuration

    rx_capture = c.captures.add(name="prx_capture")
    rx_capture.set(port_names=["prx"],format="pcap",overwrite=True)

    tx_capture = c.captures.add(name="ptx_capture")
    tx_capture.set(port_names=["ptx"],format="pcap",overwrite=True)

    dtx = c.devices.add(name="dtx")
```

Rerun the test

> python3 lab-02_test.py

#2-14

Let's install tshark so we can open the pcap. Please hit "Enter" on the next few windows to continue.

> sudo apt install tshark -y

#2-15 ll

> tshark -r prx.pcap

#2-16

The capture contains all the packets received by the otg port name "prx". You will most likely see just the udp packets (traffic flow).

```
1994    4.979969 ::10:10:10:1 → ::20:20:20:1 UDP 128 5000 → 6000 Len=62
1995    4.984960    10.10.10.1 → 20.20.20.1    UDP 128 5000 → 6000 Len=82
1996    4.984977 ::10:10:10:1 → ::20:20:20:1 UDP 128 5000 → 6000 Len=62
1997    4.989953    10.10.10.1 → 20.20.20.1    UDP 128 5000 → 6000 Len=82
1998    4.989967 ::10:10:10:1 → ::20:20:20:1 UDP 128 5000 → 6000 Len=62
1999    4.994959    10.10.10.1 → 20.20.20.1    UDP 128 5000 → 6000 Len=82
2000    4.994977 ::10:10:10:1 → ::20:20:20:1 UDP 128 5000 → 6000 Len=62
```

Let's make a small change to the script to start the capture before we start the protocols. By doing this, we can also capture the TCP packets exchanged during the BGP session.

We can use "nano" editor to **move** the "start_capture" function just above "start_protocols".

```
nano lab-02_test.py
```
#2-17

```python
api.set_config(c)

start_capture(api)

start_protocols(api)

wait_for(lambda: bgp_metrics_ok(api, test_const),"correct bgp peering")

wait_for(lambda: bgp_prefixes_ok(api, test_const),"correct bgp prefixes")

start_transmit(api)
```

ℹ️ Use "nano" editor as if you were using "Notepad" or a similar text editor. Once done with the changes hit "CTRL-X" -> "Y" -> "Enter" to save over the existing file

Rerun the test

```
python3 lab-02_test.py
```
#2-18

Open the capture by applying a "read" filter and notice the TCP packets.

```
tshark -r prx.pcap -Y tcp
```
#2-19

```
 2    1.012589        1.1.1.1 → 1.1.1.2        TCP 62 179 → 28647 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460
 3    1.018449        1.1.1.1 → 1.1.1.2        TCP 62 42446 → 179 [SYN] Seq=0 Win=0 Len=0 MSS=1460
 4    1.019036        1.1.1.1 → 1.1.1.2        TCP 58 42446 → 179 [ACK] Seq=1 Ack=1 Win=5840 Len=0
 5    1.512555        1.1.1.1 → 1.1.1.2        BGP 115 OPEN Message
 6    1.528707        1.1.1.1 → 1.1.1.2        BGP 115 OPEN Message
 7    2.028662        1.1.1.1 → 1.1.1.2        BGP 79 NOTIFICATION Message
 8    2.028671        1.1.1.1 → 1.1.1.2        TCP 58 42446 → 179 [FIN, ACK] Seq=79 Ack=80 Win=5762 Len=0
 9    2.032867        1.1.1.1 → 1.1.1.2        TCP 58 179 → 28647 [ACK] Seq=58 Ack=77 Win=5840 Len=0
10    2.400848        1.1.1.1 → 1.1.1.2        BGP 376 KEEPALIVE Message, KEEPALIVE Message, UPDATE Message, UPDATE Message
11    2.513452        1.1.1.1 → 1.1.1.2        TCP 62 179 → 20046 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460
12    2.806828        1.1.1.1 → 1.1.1.2        TCP 58 179 → 28647 [ACK] Seq=376 Ack=376 Win=5840 Len=0
13    3.023265        1.1.1.1 → 1.1.1.2        BGP 115 OPEN Message
14    3.023502        1.1.1.1 → 1.1.1.2        TCP 58 179 → 20046 [FIN, ACK] Seq=58 Ack=58 Win=5783 Len=0
15    3.023515        1.1.1.1 → 1.1.1.2        TCP 58 [TCP Out-Of-Order] 179 → 20046 [FIN, ACK] Seq=58 Ack=58 Win=5840 Len=0
16    3.023767        1.1.1.1 → 1.1.1.2        TCP 58 179 → 20046 [ACK] Seq=59 Ack=59 Win=5840 Len=0
17    3.023794        1.1.1.1 → 1.1.1.2        TCP 58 [TCP Dup ACK 16#1] 179 → 20046 [ACK] Seq=59 Ack=59 Win=5840 Len=0
```

CLEANUP:

Remove all the containers using docker compose

```
docker-compose down
```
#2-20

Remove the namespaces created by the utility script

```
sudo ip netns del lab-02_traffic_engine_1_1 && sudo ip netns del lab-02_traffic_engine_2_1
```
#2-21

Lab #03 – CONTAINERLAB DUT EGRESS TRACKING

This lab uses snappi to control the free Ixia-c Community Edition (OTG Test Tool) which is deployed via ContainerLab orchestration and utilized to create sub-interfaces and send traffic to a DUT (Device Under Test). This lab consists of 1x KENG Controller, 2x Ixia-C Traffic-Engine and 2x Ixia-C Protocol-Engine and 1x Nokia SRL containers.

This test includes traffic and protocol emulation and performs the following actions:
➢ Creates IPv4 interfaces and sub-interfaces on the OTG ports.
➢ Flows are also created with the goal of tracking any changes that might be introduced by the DUT to the received packets (egress tracking)

CONFIGURATION:

> ℹ️ With containerlab we will simplify the deployment process where the underlying networking and DUT configuration is handled automatically. We're using the NokiaSRL as DUT

Install containerlab

```
bash -c "$(curl -sL https://get.containerlab.dev)" -- -v 0.59.0
```
#3-00

DEPLOYMENT:

Let's analyze the lab-03.yml containerlab deployment file. Notice the similarities with the docker-compose deployment file.

```
cd ~/ac2-workshop/lab-03 && cat lab-03.yml
```
#3-01

> **ℹ** Unlike the previous lab, here we don't need to expose the external ports and we're using eth1 and eth2 as interface names. The management network will be handled by containerlab.

```
name: lab-03
topology:
  nodes:
    controller:
      kind: linux
      image: ghcr.io/open-traffic-generator/keng-controller:1.14.0-1
      cmd: --accept-eula --http-port 8443
      ports:
        - 8443:8443
    te1:
      kind: linux
      image: ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
      env:
        OPT_LISTEN_PORT: 5551
        ARG_IFACE_LIST: virtual@af_packet,eth1
        OPT_NO_HUGEPAGES: Yes
        OPT_NO_PINNING: Yes
        WAIT_FOR_IFACE: Yes
    te2:
      kind: linux
      image: ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
      env:
        OPT_LISTEN_PORT: 5552
        ARG_IFACE_LIST: virtual@af_packet,eth2
        OPT_NO_HUGEPAGES: Yes
        OPT_NO_PINNING: Yes
        WAIT_FOR_IFACE: Yes
    pe1:
      kind: linux
      image: ghcr.io/open-traffic-generator/ixia-c-protocol-engine:1.00.0.405
      network-mode: container:te1
      startup-delay: 5
      env:
        INTF_LIST: eth1
    pe2:
      kind: linux
      image: ghcr.io/open-traffic-generator/ixia-c-protocol-engine:1.00.0.405
      network-mode: container:te2
```

Let's deploy this lab

```
>_   sudo containerlab deploy -t lab-03.yml
```
#3-02

The Nokia SRL comes with a configuration file (lab-03-srl.cfg) which contains the VLAN subnetting and the DSCP policies required in this test.
In the end we should see all 6 containers running. We will use the highlighted names in the controller address and port location attributes.

> ℹ️ Ixia-c protocol engine containers have no management interface as they are using the same network as their corresponding traffic engines. This is found in the "network-mode" line from the deployment file.

```
+---+----------------------+--------------+----------------------------------------------------------+---------------+---------+----------------+---+----
----------------+
| # |         Name         | Container ID |                          Image                           |     Kind      |  State  |  IPv4 Address  | |
IPv6 Address    |
+---+----------------------+--------------+----------------------------------------------------------+---------------+---------+----------------+---+----
----------------+
| 1 | clab-lab-03-controller | 73e613c90f44 | ghcr.io/open-traffic-generator/keng-controller:1.14.0-1   | linux         | running | 172.20.20.2/24 | 3fff
:172:20:20::2/64 |
| 2 | clab-lab-03-pe1      | 447607ca677f | ghcr.io/open-traffic-generator/ixia-c-protocol-engine:1.00.0.405 | linux | running | N/A            | N/A
                 |
| 3 | clab-lab-03-pe2      | 5b9d47e1c9c5 | ghcr.io/open-traffic-generator/ixia-c-protocol-engine:1.00.0.405 | linux | running | N/A            | N/A
                 |
| 4 | clab-lab-03-srl      | ffe9a6d78d03 | ghcr.io/nokia/srlinux:latest                             | nokia_srlinux | running | 172.20.20.5/24 | 3fff
:172:20:20::5/64 |
| 5 | clab-lab-03-te1      | afdc3ed2a1ee | ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99 | linux    | running | 172.20.20.3/24 | 3fff
:172:20:20::3/64 |
| 6 | clab-lab-03-te2      | f7963c3c6e98 | ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99 | linux    | running | 172.20.20.4/24 | 3fff
:172:20:20::4/64 |
+---+----------------------+--------------+----------------------------------------------------------+---------------+---------+----------------+---+----
----------------+
```

EXECUTION:

Let's open the script file and set the KENG Controller address and the port location attributes. Use "clab-lab-03-te1:5551+clab-lab-03-te1:50071" and "clab-lab-03-te2:5552+clab-lab-03-te2:50071" for the ports and "https://clab-lab-03-controller:8443" for the KENG Controller.

```
>_   vi lab-03-1_test.py
```
#3-03

```
api = snappi.api(location="https://clab-lab-03-controller:8443", verify=False)

c = otg_config(api, test_const)

api.set_config(c)

start_protocols(api)

start_transmit(api)

wait_for(lambda: flow_metrics_ok(api, test_const), "flow metrics",2,90)
def otg_config(api, tc):
    c = api.config()

    p1 = c.ports.add(name="p1", location="clab-lab-03-te1:5551+clab-lab-03-te1:50071")
    p2 = c.ports.add(name="p2", location="clab-lab-03-te2:5552+clab-lab-03-te2:50071")
```

> ℹ️ Here we are using the names which were created by containerlab in the management network.

Open the test file and notice how we're now tracking on the Rx port at the VLAN ID.

> 🖳 cat lab-03-1_test.py
>
> #3-04

> ℹ️ On the Tx side the packets are not tagged, however we're expecting them received on the Rx with VLAN tagging. The "egress packet" is a way to tell the OTG receiving port on where it should look for values.

```
f.egress_packet.ethernet()
eg_vlan = f.egress_packet.add().vlan
eg_ip = f.egress_packet.add().ipv4

eg_vlan.id.metric_tags.add(name="vlanIdRx")
# eg_ip.priority.raw.metric_tags.add(name="dscpValuesRx", length=6)
```

Let's run the test and see if we see the 3 subnet VLAN IDs on the Rx side.

```
--------------------------------------------------------------------------------
Flow Metrics
--------------------------------------------------------------------------------
Name          State        Frames Tx    Frames Rx    FPS Tx    FPS Rx    Bytes Tx    Bytes Rx
f1            started      822          821          99        100       105216      108372
--------------------------------------------------------------------------------


----------------------------------------------------------------------
Tagged Metrics
----------------------------------------------------------------------
Tracked Value  Frames Rx    FPS Rx       Bytes Rx
101            274          33           36168
102            274          33           36168
103            273          33           36036
----------------------------------------------------------------------
```

The Nokia-SRL is configured to rewrite anything which is received as DSCP 10 (AF11) into DSCP 20 (AF22) everything else to DSCP 0. Let's make a copy of the script and change the script to track this.

We are sending 4 DSCP values

```
f_eth = f.packet.add().ethernet
f_ip = f.packet.add().ipv4
f_eth.src.value = d1_eth.mac
f_ip.src.value = tc["1Ip"]
f_ip.dst.increment.set(start = tc["2IpStart"], step = "0.0.1.0", count = tc["2SubnetCount"])
f_ip.priority.dscp.phb.values = [10, 14, 22, 24]
```

> **ℹ** There is a limited number of bits (12) we can use for egress tracking. Therefore, we cannot enable both VLAN id and IP DSCP for this. VLAN ID + DSCP would 18 in total

> **⚠** An error will result if we're exceeding the maximum number of bits reserved for "egress" tracking

Comment out the VLAN and enable the DSCP egress tracking

```
f.egress_packet.ethernet()
eg_vlan = f.egress_packet.add().vlan
eg_ip = f.egress_packet.add().ipv4

# eg_vlan.id.metric_tags.add(name="vlanIdRx")
eg_ip.priority.raw.metric_tags.add(name="dscpValuesRx", length=6)
```

Let's run the second test and see if we see the DSCP remarking. In case you see a "MAC address resolution error" please retry running the test.
As you can see 75% of the packets get rewritten.

```
>_  python3 lab-03-2_test.py
```
#3-07

```
---------------------------------------------------------------------------------------------------------
Flow Metrics
---------------------------------------------------------------------------------------------------------
Name           State        Frames Tx    Frames Rx    FPS Tx     FPS Rx      Bytes Tx      Bytes Rx
f1             stopped      1000         1000         100        100         128000        132000
---------------------------------------------------------------------------------------------------------



-------------------------------------------------------------------
Tagged Metrics
-------------------------------------------------------------------
Tracked Value  Frames Rx      FPS Rx         Bytes Rx
0x00           750            75             99000
0x14           250            24             33000
-------------------------------------------------------------------
```

Let's change the distribution to 20% by adding one more DSCP value to the list. Edit the file and add one more DSCP value on the Tx side.

```
>_   vim lab-03-2_test.py
#3-08
```

```
f_eth = f.packet.add().ethernet
f_ip = f.packet.add().ipv4
f_eth.src.value = d1_eth.mac
f_ip.src.value = tc["1Ip"]
f_ip.dst.increment.set(start = tc["2IpStart"], step = "0.0.1.0", count = tc["2SubnetCount"])
f_ip.priority.dscp.phb.values = [10, 14, 22, 24, 32]
```

Now 0x14 DSCP packets account to just to 20% of the traffic.

```
---------------------------------------------------------
Tagged Metrics
---------------------------------------------------------
Tracked Value  Frames Rx     FPS Rx        Bytes Rx
0x00           800           80            105600
0x14           200           20            26400
---------------------------------------------------------
```

Let's change the marking policy on the DUT to remark everything to DSCP 30 (AF33). Connect to the DUT and use password "**NokiaSrl1!**".

```
>_   ssh admin@clab-lab-03-srl
#3-09
```

Make sure that you are in the cli ("srl" prompt). If not, type "sr_cli" after ssh.

```
     enter candidate

>_   set qos rewrite-rules dscp-policy test-rewrite map FC1 dscp 30

#3-10  commit now

     quit
```

Rerun the test

```
>_   python3 lab-03-2_test.py
#3-11
```

DSCP 30 (AF33) is 0x1e at 20% received packets

```
-------------------------------------------------------
Tagged Metrics
-------------------------------------------------------
Tracked Value  Frames Rx      FPS Rx        Bytes Rx
0x00           800            80            105600
0x1e           200            20            26400
-------------------------------------------------------
```

CLEANUP:

Destroy the lab

```
>_   sudo containerlab destroy -t lab-03.yml --cleanup
#3-12
```

Lab #04 – CONTAINERLAB DUT CONVERGENCE

This lab uses snappi to control the free Ixia-c Community Edition (OTG Test Tool) which is deployed via Container Lab orchestration and utilized to create sub-interfaces and send traffic to a DUT. This lab consists of 1x Ixia-C-One and 1x Nokia SRL container.

These tests include traffic and protocol emulation and performs the following actions:

☑
➢ Create BGPv4 peers on the first two Ixia-c-One interfaces as we did in Lab #02 but now, we're sending them through a DUT.
➢ They both advertise several IPv4 unicast prefixes. Then bidirectional flows are created using these routes as sources and destinations
➢ Validate that the BGP peering gets established, routes are being received and traffic has no packet loss
➢ Verify convergence when different actions are taken (link down and preferred route withdraw)
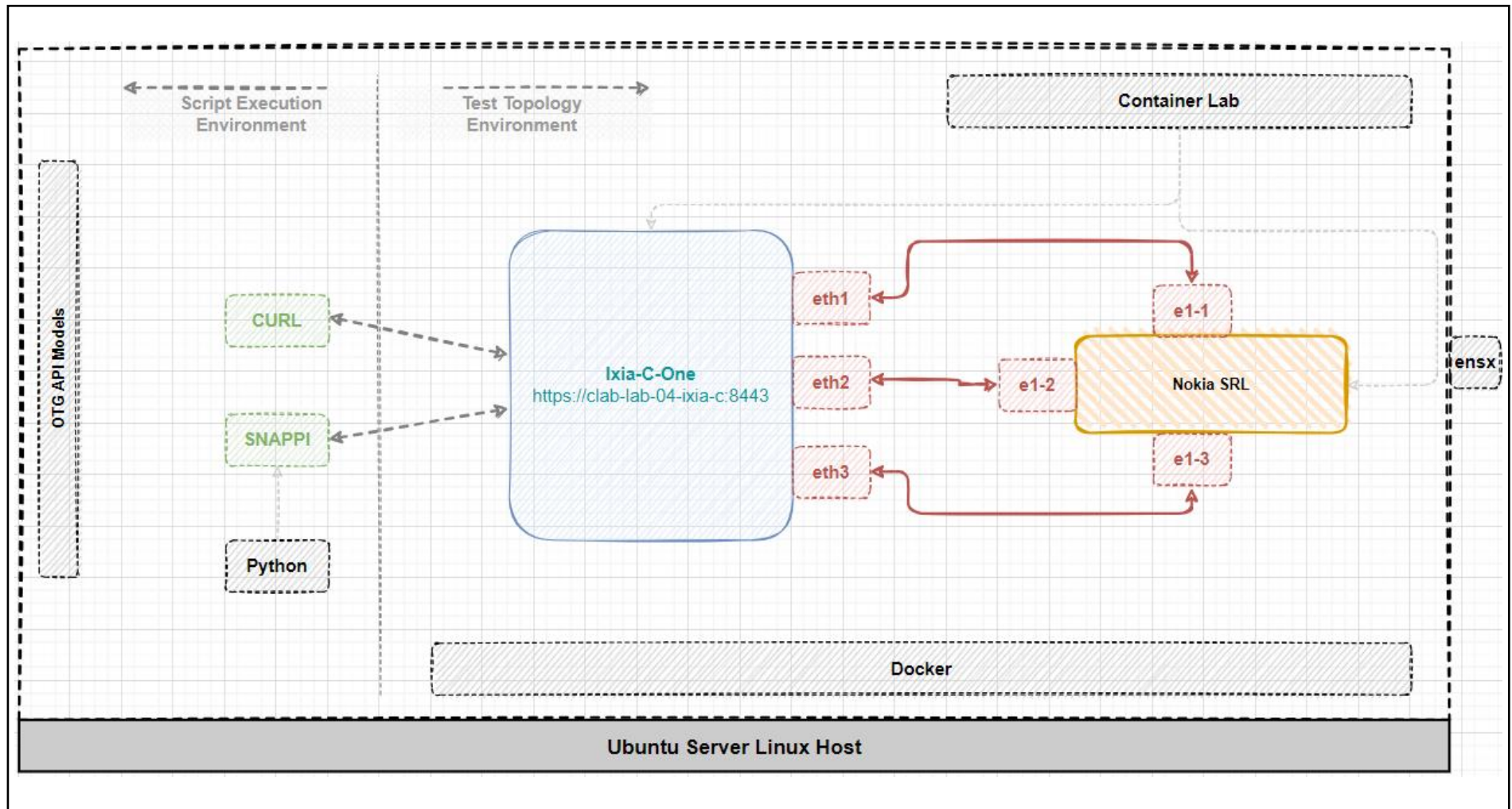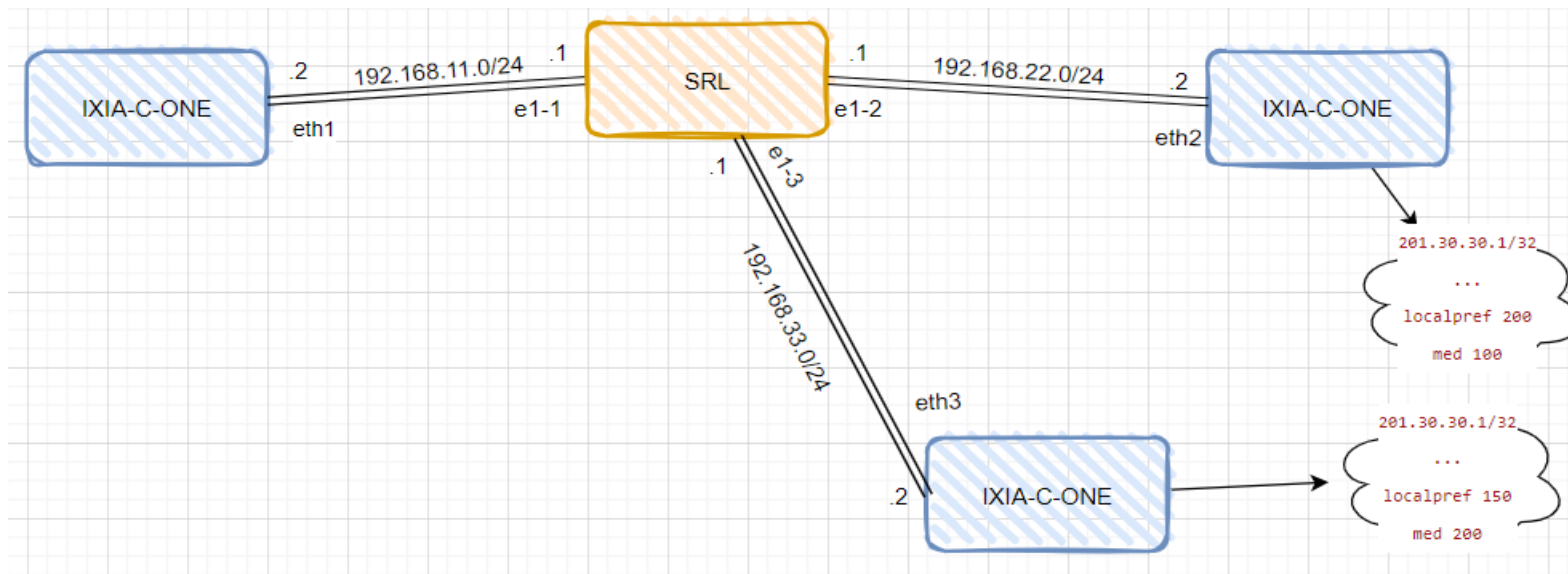
CONFIGURATION:

Let's analyze the lab-04.yml containerlab deployment file.

⌨ cd ~/ac2-workshop/lab-04 && cat lab-04.yml

`#4-01`

ℹ As traffic generator we are now using an all-in-one container named "ixia-c-one" which has all the components with individual interfaces connected to the DUT.

```yaml
name: lab-04
topology:
  nodes:
    ixia-c:
      kind: keysight_ixia-c-one
      image: ghcr.io/open-traffic-generator/ixia-c-one:1.14.0-1
    srl:
      kind: nokia_srlinux
      image: ghcr.io/nokia/srlinux:latest
      startup-config: lab-04-srl.cfg        You, last week • labs
  links:
    - endpoints: ["ixia-c:eth1","srl:e1-1"]
    - endpoints: ["ixia-c:eth2","srl:e1-2"]
    - endpoints: ["ixia-c:eth3","srl:e1-3"]
```

DEPLOYMENT:

Let's deploy this lab

```
sudo containerlab deploy -t lab-04.yml
```
#4-02

Any missing docker images will be automatically downloaded and we should see the two containers running.

```
+---+------------------+--------------+-------------------------------------------+-------------------+---------+----------------+----------------------+
| # |       Name       | Container ID |                   Image                   |       Kind        |  State  |  IPv4 Address  |     IPv6 Address     |
+---+------------------+--------------+-------------------------------------------+-------------------+---------+----------------+----------------------+
| 1 | clab-lab-04-ixia-c | 56a9bf6c546f | ghcr.io/open-traffic-generator/ixia-c-one:1.14.0-1 | keysight_ixia-c-one | running | 172.20.20.3/24 | 3fff:172:20:20::3/64 |
| 2 | clab-lab-04-srl    | dccadc0bec08 | ghcr.io/nokia/srlinux:latest              | nokia_srlinux     | running | 172.20.20.2/24 | 3fff:172:20:20::2/64 |
+---+------------------+--------------+-------------------------------------------+-------------------+---------+----------------+----------------------+
```

EXECUTION:

Let's open the script file and find the KENG controller address and the port location attributes.

```
vim lab-04-1_test.py
```
#4-03

```python
    "startDstRoute": "201.30.30.1",
}

api = snappi.api(location="https://clab-lab-04-ixia-c:8443", verify=False)
c = ibgp_route_prefix_config(api, test_const)


def ibgp_route_prefix_config(api, tc):
    c = api.config()
    p1 = c.ports.add(name="p1", location="eth1")
    p2 = c.ports.add(name="p2", location="eth2")
    p3 = c.ports.add(name="p3", location="eth3")
```

In this script we're configuring iBGP on eth2 and eth3 then we're advertising the same route but with different parameters. We're sending traffic towards this route then we introduce different events to trigger DUT convergence.

Let's make sure the "link-down" function is enabled and we set BGP "localpref" and BGP"med" according to the above diagram. Save the file.

```python
while True:
    get_flow_metrics(api)
    get_port_metrics(api)
    if time.time() - start > (test_const["pktCount"]/test_const["pktRate"])/2:
        break
    time.sleep(2)

# withdraw_routes(api)

link_operation(api, "down")

time.sleep(2)

get_bgp_prefixes(api)

wait_for(lambda: traffic_stopped(api), "traffic stopped",2,90)

get_convergence_time(api,test_const)

link_operation(api, "up")


    d2_bgpv4_peer_rrv4.addresses.add(
        address=tc["startDstRoute"], prefix=32, count=tc["routeCount"], step=1
    )
    d2_bgpv4_peer_rrv4.advanced.set(
        local_preference = 200,
        multi_exit_discriminator = 100
    )

    d3_eth = d3.ethernets.add(name="d3_eth")
```

```
d2_bgpv4_peer_rrv4.addresses.add(                      d3_bgpv4_peer_rrv4.addresses.add(
    address=tc["startDstRoute"], prefix=32, count=tc["routeCount"], step=1        address=tc["startDstRoute"], prefix=32,
)                                                      )
d2_bgpv4_peer_rrv4.advanced.set(                       d3_bgpv4_peer_rrv4.advanced.set(
    local_preference = 200,                                local_preference = 150,
    multi_exit_discriminator = 100                         multi_exit_discriminator = 200
)                                                      )
```

Check your script changes. Use "q" to escape.

```
>_  git diff lab-04-1_test.py
```

#4-04

ⓘ The link operation events are built according to the OTG control models found here

```
def link_operation(api, operation):
    print("%s Bringing %s port 2     ..." % (datetime.now(),operation))
    cs = api.control_state()
    cs.choice = cs.PORT
    cs.port.choice = cs.port.LINK
    cs.port.link.port_names = ["p2"]
    if operation == "down":
        cs.port.link.state = cs.port.link.DOWN
    else:
        cs.port.link.state = cs.port.link.UP
    api.set_control_state(cs)
```

Optionally before running your script, you can open another terminal and connect to the DUT to visualize the routes advertised by the OTG BGP peers. Nokia SRL password: "NokiaSrl1!"

```
>_  ssh admin@clab-lab-04-srl
```

#4-05

Run your script and see the convergence time.

In the DUT terminal, you will initially see the best route next hop is 192.168.22.2 peer (eth2) but after the link-down event, the eth3 peer 192.168.33.2 becomes the only peer advertising the routes.

Status	Network	Next Hop	MED	LocPref	
u*>	10.10.10.1/32	0.0.0.0	-		i
u*>	101.10.10.1/32	192.168.11.2	-	100	i
u*>	101.10.10.2/32	192.168.11.2	-	100	i
u*>	101.10.10.3/32	192.168.11.2	-	100	i
u*>	101.10.10.4/32	192.168.11.2	-	100	i
u*>	101.10.10.5/32	192.168.11.2	-	100	i
u*>	192.168.11.0/24	0.0.0.0	-		i
u*>	192.168.22.0/24	0.0.0.0	-		i
u*>	192.168.33.0/24	0.0.0.0	-		i
u*>	201.30.30.1/32	192.168.22.2	100	200	i
*	201.30.30.1/32	192.168.33.2	200	150	i
u*>	201.30.30.2/32	192.168.22.2	100	200	i
*	201.30.30.2/32	192.168.33.2	200	150	i
u*>	201.30.30.3/32	192.168.22.2	100	200	i
*	201.30.30.3/32	192.168.33.2	200	150	i
u*>	201.30.30.4/32	192.168.22.2	100	200	i
*	201.30.30.4/32	192.168.33.2	200	150	i
u*>	201.30.30.5/32	192.168.22.2	100	200	i
*	201.30.30.5/32	192.168.33.2	200	150	i

Status	Network	Next Hop	MED	LocPref	
u*>	10.10.10.1/32	0.0.0.0	-		i
u*>	101.10.10.1/32	192.168.11.2	-	100	i
u*>	101.10.10.2/32	192.168.11.2	-	100	i
u*>	101.10.10.3/32	192.168.11.2	-	100	i
u*>	101.10.10.4/32	192.168.11.2	-	100	i
u*>	101.10.10.5/32	192.168.11.2	-	100	i
u*>	192.168.11.0/24	0.0.0.0	-		i
u*>	192.168.33.0/24	0.0.0.0	-		i
	201.30.30.1/32	192.168.22.2	100	200	i
u*>	201.30.30.1/32	192.168.33.2	200	150	i
	201.30.30.2/32	192.168.22.2	100	200	i
u*>	201.30.30.2/32	192.168.33.2	200	150	i
	201.30.30.3/32	192.168.22.2	100	200	i
u*>	201.30.30.3/32	192.168.33.2	200	150	i
	201.30.30.4/32	192.168.22.2	100	200	i
u*>	201.30.30.4/32	192.168.33.2	200	150	i
	201.30.30.5/32	192.168.22.2	100	200	i
u*>	201.30.30.5/32	192.168.33.2	200	150	i

Before the event, the traffic is being received on p2 (eth2) then after the event is switched to p3 (eth3)

```
2024-11-04 23:15:15.859195 Getting flow metrics     ...


-----------------------------------------------------------------------------------------------
Flow Metrics
-----------------------------------------------------------------------------------------------
Name          State          Frames Tx      Frames Rx      FPS Tx        FPS Rx        Bytes Tx      Bytes Rx
bgpFlow       stopped        15000          14986          1000          999           1500000       1498600
-----------------------------------------------------------------------------------------------


2024-11-04 23:15:15.870188 Getting port metrics     ...


-----------------------------------------------------------------------------------------------------------------------------
Port Metrics
-----------------------------------------------------------------------------------------------------------------------------
Name     State        Frames Tx    Frames Rx    FPS Tx     FPS Rx    Bytes Tx    Bytes Rx    Bytes Tx Rate   Bytes Rx Rate
p1       stopped      15000        4            999        0         1500000     480         99999           0
p2       stopped      0            8136         0          0         0           813562      0               0
p3       stopped      0            6855         0          999       0           685557      0               99994
-----------------------------------------------------------------------------------------------------------------------------


Done waiting for traffic stopped
2024-11-04 23:15:15.888525 Convergence time was 0.014s
2024-11-04 23:15:15.888587 Bringing up port 2    ...
```

You can always use curl commands to check the otg BGP status, routes and flow metrics

> curl -k -d '{"choice":"bgpv4"}' -X POST https://clab-lab-04-ixia-c:8443/monitor/metrics

#4-08

> curl -k -d '{"choice":"bgp_prefixes"}' -X POST https://clab-lab-04-ixia-c:8443/monitor/states

#4-09

> curl -k -d '{"choice":"flow"}' -X POST https://clab-lab-04-ixia-c:8443/monitor/metrics

#4-10

The convergence time is calculated based on the number of lost packets divided by their FPS Tx rate.
Let's change the number of advertised routes to 50 to see if this number changes.

```
"2Prefix": 24,
"3Mac": "00:00:01:01:03:01",
"3Ip": "192.168.33.2",
"3Gateway": "192.168.33.1",
"3Prefix": 24,
"routeCount": 50,
"1AdvRoute": "101.10.10.1",
"startDstRoute": "201.30.30.1",
```

Rerun the test

>   python3 lab-04-1_test.py

#4-12

```
Done waiting for traffic stopped
2024-11-04 23:25:32.899369 Convergence time was 0.153s
2024-11-04 23:25:32.899414 Bringing up port 2    ...
```

The convergence time is longer is because there are more destinations to be processed. Also, the convergence time includes the time taken for the DUT to detect the "link down" event and recalculate the routing table.

Let's enable the "withdraw-routes" function. This is going to be a soft reset, as the eth2 BGP peer will send a BGP Update message indicating that its routes are no longer reachable. Make sure to comment out the "link_operation" lines.

>   vim lab-04-1_test.py

#4-13

```
0    def withdraw_routes(api):        You, 3 hours ago • changes to lab-04
1        print("%s Withdraw routes from port 2     ..." % datetime.now())
2        cs = api.control_state()
3        cs.choice = cs.PROTOCOL
4        cs.protocol.choice = cs.protocol.ROUTE
5        cs.protocol.route.names = ["d2_bgpv4_peer_rrv4"]
6        cs.protocol.route.state = cs.protocol.route.WITHDRAW
7        api.set_control_state(cs)
```

```
            break
        time.sleep(2)

    withdraw_routes(api)

    # link_operation(api, "down")

    time.sleep(2)

    get_bgp_prefixes(api)

    wait_for(lambda: traffic_stopped(api), "traffic stopped",2,90)

    get_convergence_time(api,test_const)

    # link_operation(api, "up")
```

Rerun the test

Before the routes withdraw event all the traffic flows towards the eth2 BGP peer as is considered the "best route"

```
--------------------------------------------------------------------------------------
Flow Metrics
--------------------------------------------------------------------------------------
Name          State        Frames Tx     Frames Rx     FPS Tx      FPS Rx      Bytes Tx     Bytes Rx
bgpFlow       started      6070          6060          1000        1000        607000       606000
--------------------------------------------------------------------------------------


2024-11-04 23:30:16.555149 Getting port metrics    ...


--------------------------------------------------------------------------------------------------------------
Port Metrics
--------------------------------------------------------------------------------------------------------------
Name       State       Frames Tx     Frames Rx     FPS Tx      FPS Rx      Bytes Tx     Bytes Rx     Bytes Tx Rate  Bytes Rx Rate
p1         started     6086          1             1000        0           608600       88           100000         0
p2         stopped     0             6074          0           999         0            607400       0              99999
p3         stopped     0             0             0           0           0            0            0              0
--------------------------------------------------------------------------------------------------------------
```

After the route withdraw the traffic is received on eth2 and at the end the convergence time is calculated

```
------------------------------------------------------------------------------------------------
Flow Metrics
------------------------------------------------------------------------------------------------
Name         State       Frames Tx    Frames Rx    FPS Tx      FPS Rx      Bytes Tx    Bytes Rx
bgpFlow      stopped     15000        14989        999         999         1500000     1498900
------------------------------------------------------------------------------------------------


2024-11-04 23:30:27.023758 Getting port metrics    ...


---------------------------------------------------------------------------------------------------------------------
Port Metrics
---------------------------------------------------------------------------------------------------------------------
Name    State       Frames Tx    Frames Rx    FPS Tx    FPS Rx    Bytes Tx    Bytes Rx    Bytes Tx Rate  Bytes Rx Rate
p1      stopped     15000        2            1000      0         1500000     461         100000         0
p2      stopped     0            8351         0         0         0           835337      0              0
p3      stopped     0            6641         0         999       0           664331      0              99998
---------------------------------------------------------------------------------------------------------------------


Done waiting for traffic stopped
2024-11-04 23:30:27.040595 Convergence time was 0.011s
```

> ℹ️ The reason we might see a quicker convergence when we just withdraw the routes is because now the packets are getting forwarded as the port is still up. We're losing a few packets because there is a small delay when the original routes are withdrawn from the Forwarding Information Base (data plane forwarding table) and the other calculated routes are installed.

CLEANUP:

Destroy the lab

```
sudo containerlab destroy -t lab-04.yml --cleanup
```

#4-15

Lab #05 – IXIA-C IN KUBERNETES

This lab uses snappi and gosnappi to control the free Ixia-c Community Edition which is deployed via KNE (Kubernetes Network Emulation) orchestration in a local Kubernetes cluster. This lab consists of 1x Kind Cluster, 1x KENG Operator pod, 1x KENG Controller pod and 2x pairs of Traffic Engine – Protocol Engine pods.

☑ These tests include traffic and protocol emulation and perform the following actions:
  ➢ Creates one BGPv4 peer on each pair of PE+TE containers. They both advertise several IPv4 and IPv6 unicast prefixes.
  ➢ Then bidirectional flows are created using these routes as sources and destinations.
  ➢ Validates that the BGP peering gets established, routes are being received and traffic has no packet loss.
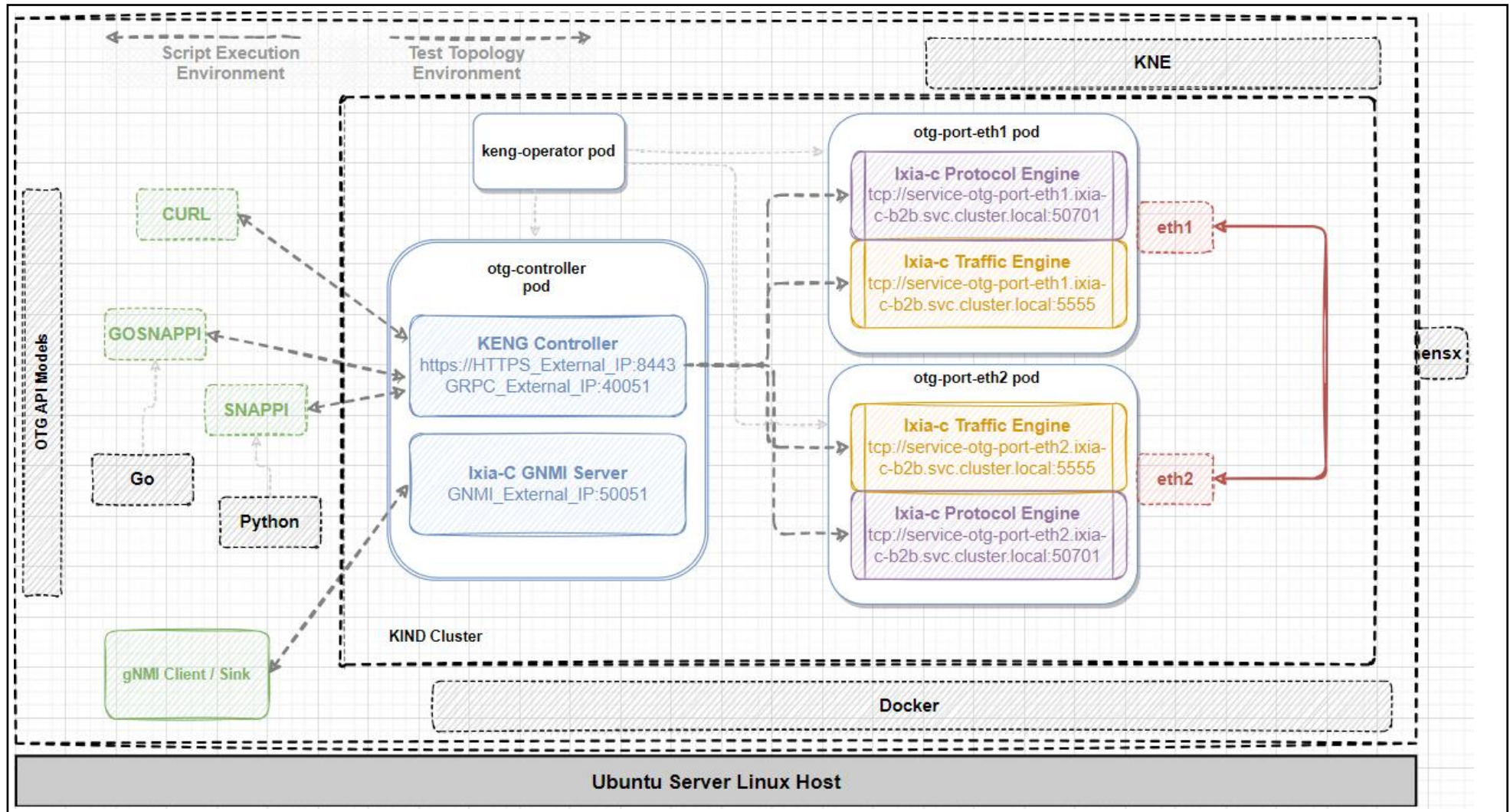
CONFIGURATION:

Install KIND (Kubernetes in Docker). This will allow us to create a Kubernetes (K8S) cluster with a docker container as a node.

```
[ $(uname -m) = x86_64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.24.0/kind-linux-amd64

chmod +x ./kind

sudo mv ./kind /usr/local/bin/kind
```
#5-01

> ℹ **KIND** is a tool for running local Kubernetes clusters using Docker container "nodes".
> **KIND** was primarily designed for testing Kubernetes itself, but may be used for local development or CI

Let's install the kubectl utility for controlling the K8S cluster

```
curl -LO https://dl.k8s.io/release/v1.27.3/bin/linux/amd64/kubectl

sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```
#5-02

Let's install GO language

```
cd ~ && wget https://go.dev/dl/go1.23.2.linux-amd64.tar.gz

sudo rm -rf /usr/local/go && sudo tar -C /usr/local -xzf go1.23.2.linux-amd64.tar.gz
```
#5-03

Continue with GO installation

```
export PATH=$PATH:/usr/local/go/bin  && echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.bashrc

go version
```
#5-04

Install KNE (Kubernetes Network Emulation). This might take 1-2 minutes.

```
wget https://github.com/openconfig/kne/archive/refs/tags/v0.2.1.tar.gz && tar -xvzf v0.2.1.tar.gz
cd kne-0.2.1 && make install
```
#5-05

We used containerlab to create network topologies in the docker environment. Similarly, **KNE** (Kubernetes Network Emulation) is a way to create topologies within a Kubernetes cluster (https://github.com/openconfig/kne)

Install gNIM client (gnmi-c)

```
bash -c "$(curl -sL https://get-gnmic.openconfig.net)"
```
#5-06

We will use gNIMc to interact with the Ixia-c gNMI server pod for statistics and states.

DEPLOYMENT:

Let's look at the deployment YAML file.

```
cd ~/ac2-workshop/lab-05 && cat kind-bridge.yaml
```
#5-07

We have the cluster config, the ingress load balancer the CNI and the KENG-operator. The ingress is needed for enabling external IP addresses for the services so that a script can interact with KENG controller pod. The CNI is needed to create the links between the test pods.

```
# kind-bridge.yaml cluster config file sets up a kind cluster where default PTP CNI plugin
# is swapped with the Bridge CNI plugin.
# Bridge CNI plugin is required by some Network OSes to operate.
cluster:
  kind: Kind
  spec:
    name: kne
    recycle: True
    version: v0.17.0
    image: kindest/node:v1.31.0
    config: /home/ubuntu/kne-0.2.1/manifests/kind/config.yaml
    additionalManifests:
      - /home/ubuntu/kne-0.2.1/manifests/kind/bridge.yaml
ingress:
  kind: MetalLB
  spec:
    manifest: /home/ubuntu/kne-0.2.1/manifests/metallb/manifest.yaml
    ip_count: 100
cni:
  kind: Meshnet
  spec:
    manifest: /home/ubuntu/kne-0.2.1/manifests/meshnet/grpc/manifest.yaml
controllers:
  - kind: IxiaTG
    spec:
      operator: ixiatg-operator.yaml
      configMap: ixiatg-configmap.yaml
```

> ℹ️ The **KENG-operator** (https://github.com/open-traffic-generator/keng-operator) is used by KNE to create the test pods. All the necessary version information is stored in the ixiatg-configmap file. The operator version must be compatible with the KENG-controller and this information can be found in the ixia-c release notes.

Let's look at the ixiatg-configmap.yaml. The release tag below will be used by the KENG-operator when deploying OTG pods requested in the KNE topology file (textproto file)

```
>_   cat ixiatg-configmap.yaml
```
#5-08

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: ixiatg-release-config
  namespace: ixiatg-op-system
data:
  versions: |
    {
      "release": "1.14.0-1",
      "images": [
        {
          "name": "controller",
          "path": "ghcr.io/open-traffic-generator/keng-controller",
          "tag": "1.14.0-1"
        },
        {
          "name": "gnmi-server",
          "path": "ghcr.io/open-traffic-generator/otg-gnmi-server",
          "tag": "1.14.15"
        },
        {
          "name": "traffic-engine",
          "path": "ghcr.io/open-traffic-generator/ixia-c-traffic-engine",
          "tag": "1.8.0.99"
        },
        {
          "name": "protocol-engine",
          "path": "ghcr.io/open-traffic-generator/ixia-c-protocol-engine",
          "tag": "1.00.0.405"
        }]
    }
```

Now we can create the K8S cluster using the provided YAML file.

```
>_  kne deploy kind-bridge.yaml
```
#5-09

Let's check to make sure everything is up and running.

```
kubectl get pods -A
```
#5-10

```
NAMESPACE           NAME                                              READY   STATUS    RESTARTS   AGE
ixiatg-op-system    ixiatg-op-controller-manager-5fdc78d8d7-zqgkf     2/2     Running   0          13m
kube-system         coredns-6f6b679f8f-8l5gn                          1/1     Running   0          14m
kube-system         coredns-6f6b679f8f-lbh4n                          1/1     Running   0          14m
kube-system         etcd-kne-control-plane                            1/1     Running   0          14m
kube-system         kindnet-zbrtj                                     1/1     Running   0          14m
kube-system         kube-apiserver-kne-control-plane                  1/1     Running   0          14m
kube-system         kube-controller-manager-kne-control-plane         1/1     Running   0          14m
kube-system         kube-proxy-hvqcv                                  1/1     Running   0          14m
kube-system         kube-scheduler-kne-control-plane                  1/1     Running   0          14m
local-path-storage  local-path-provisioner-57c5987fd4-rsvxn           1/1     Running   0          14m
meshnet             meshnet-nb688                                     1/1     Running   0          13m
metallb-system      controller-d54f47bcc-8wctj                        1/1     Running   0          14m
metallb-system      speaker-kw45s                                     1/1     Running   0          14m
```

With this shell script we are pulling the docker images then load them to the KIND node.

```
./load.sh
```
#5-11

Let's see the back-to-back topology we're going to create. The "otg" node version must match the one we have in the ixiatg-configmap.yaml

```
cat ixia-c-b2b.textproto
```
#5-12

```
# proto-file: github.com/openconfig/kne/proto/topo.proto
# proto-message: Topology
name: "ixia-c-b2b"
nodes: {
    name: "otg"
    vendor: KEYSIGHT
    version: "1.14.0-1" # Please update this with the local version from ixiatg-configmap.yaml
}
links: {
    a_node: "otg"
    a_int: "eth1"
    z_node: "otg"
    z_int: "eth2"
}
```

Let's create the test topology with the 2 back-to-back pods. The otg interfaces names "eth1" and "eth2" will be used by the KENG-operator to create the otg port pods. These will be prefaced by "**otg-port-**"

```
kne create ixia-c-b2b.textproto
```
#5-13

We can check the status of the pods and their Kubernetes (K8S) services

```
kubectl get pods -n ixia-c-b2b -o wide
kubectl get services -n ixia-c-b2b
```
#5-14

```
ubuntu@ip-10-0-10-138:~/ac2-workshop/lab-05$ kubectl get pods -n ixia-c-b2b -o wide
NAME              READY   STATUS    RESTARTS   AGE     IP            NODE                NOMINATED NODE   READINESS GATES
otg-controller    2/2     Running   0          4m54s   10.244.0.10   kne-control-plane   <none>           <none>
otg-port-eth1     2/2     Running   0          4m54s   10.244.0.12   kne-control-plane   <none>           <none>
otg-port-eth2     2/2     Running   0          4m54s   10.244.0.11   kne-control-plane   <none>           <none>
ubuntu@ip-10-0-10-138:~/ac2-workshop/lab-05$ kubectl get services -n ixia-c-b2b
NAME                          TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)                          AGE
service-gnmi-otg-controller   LoadBalancer   10.96.178.252   172.18.0.50    50051:31428/TCP                  4m58s
service-grpc-otg-controller   LoadBalancer   10.96.79.237    172.18.0.51    40051:31034/TCP                  4m58s
service-https-otg-controller  LoadBalancer   10.96.211.182   172.18.0.52    8443:30373/TCP                   4m58s
service-otg-port-eth1         LoadBalancer   10.96.230.78    172.18.0.54    5555:30470/TCP,50071:30719/TCP   4m58s
service-otg-port-eth2         LoadBalancer   10.96.240.32    172.18.0.53    5555:31339/TCP,50071:32697/TCP   4m58s
```

Describe one of the test pods. Notice we have the protocol-engine and the traffic-engine containers. You can see the images match those from the ixiatg-configmap.yaml

```
kubectl describe pods otg-port-eth1 -n ixia-c-b2b
```
#5-15

```
         /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-2vrtc (ro)
Containers:
  otg-port-eth1-traffic-engine:
    Container ID:   containerd://d1ed6004623c9b659aab211d5a2dcfae3430da073d5de42d4f6ad1264a98383a
    Image:          ghcr.io/open-traffic-generator/ixia-c-traffic-engine:1.8.0.99
    Image ID:       docker.io/library/import-2024-11-08@sha256:f4effb845334bbfc3f5b306b17ab8bf4f3c32fc88305a904561614f119acbb94
```

```
otg-port-eth1-protocol-engine:
  Container ID:   containerd://f1d74bbe43793ac25be5446f0cd74eaaa761ae0637b404d0fbe769fbaf31ff10
  Image:          ghcr.io/open-traffic-generator/ixia-c-protocol-engine:1.00.0.405
  Image ID:       docker.io/library/import-2024-11-08@sha256:d8c7ffd8f8c5307caf769092079cfd3f476e41962c1d1a80be4f3d6237274ba5
```

EXECUTION:

Let's see the External IP address used by the controller for the HTTPS service

> kubectl get services -n ixia-c-b2b

#5-16

```
NAME                         TYPE          CLUSTER-IP       EXTERNAL-IP    PORT(S)                         AGE
service-gnmi-otg-controller  LoadBalancer  10.96.23.16      172.18.0.85    50051:32430/TCP                 79s
service-grpc-otg-controller  LoadBalancer  10.96.225.154    172.18.0.86    40051:30970/TCP                 79s
service-https-otg-controller LoadBalancer  10.96.8.249      172.18.0.87    8443:30225/TCP                  79s
service-otg-port-eth1        LoadBalancer  10.96.129.89     172.18.0.88    5555:30703/TCP,50071:31422/TCP  79s
service-otg-port-eth2        LoadBalancer  10.96.236.9      172.18.0.89    5555:31340/TCP,50071:31332/TCP  79s
```

Now we need to use this IP address in our test files

> vim lab-05-1_test.py

#5-17

```python
print("")
api = snappi.api(location="https://172.18.0.87:8443", verify=False)
print("%s Starting connection to controller                    ... " % datetime.now())
```

The location attribute for the ports is set to "eth1" and "eth2". This information is the same as the one used in the topology file / links section.

```python
port1, port2 = (
    configuration.ports
    .port(name="Port-1", location="eth1")
    .port(name="Port-2", location="eth2")
)
```

Let's run the test

```
>_   python3 lab-05-1_test.py
```
#5-18

Make the change to the second file and run it

```
>_   vim lab-05-2_test.py
```
#5-19

```python
import logging as log
import snappi
from datetime import datetime
import time


def Test_ebgp_route_prefix():
    test_const = {
        "controller_location": "https://172.18.0.87:8443",
        "p1_location": "eth1",
        "p2_location": "eth2",
        "pktRate": 200,
        "pktCount": 1000,
        "pktSize": 128,
        "trafficDuration": 20,
        "txMac": "00:00:01:01:01:01",
```

```
>_   python3 lab-05-2_test.py
```
#5-20

Let's make the change to the "go" test file and run it

```
>_   vim lab-05-3_test.go
```
#5-21

```
func TestEbgpRoutePrefix(t *testing.T) {

        testConst := map[string]interface{}{
                "controller_location": "https://172.18.0.87:8443",
                // "controller_location": "172.18.0.62:40051",
                "p1_location":  "eth1",
                "p2_location":  "eth2",
                "pktRate":      uint64(200),
                "pktCount":     uint32(12000),
```

Run the gosnappi test. At the first "go" run, it might take a minute to compile the code

> go test -v lab-05-3_test.go

#5-22

ⓘ The **KENG-controller** can be configured using HTTP or gRPC protocols.

Now we want to change the API transport from HTTPS to gRPC. Let's see the External IP address used by the controller for the gRPC service

> kubectl get services -n ixia-c-b2b

#5-23

```
NAME                          TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)                          AGE
service-gnmi-otg-controller   LoadBalancer  10.96.23.16     172.18.0.85   50051:32430/TCP                  12m
service-grpc-otg-controller   LoadBalancer  10.96.225.154   172.18.0.86   40051:30970/TCP                  12m
service-https-otg-controller  LoadBalancer  10.96.8.249     172.18.0.87   8443:30225/TCP                   12m
service-otg-port-eth1         LoadBalancer  10.96.129.89    172.18.0.88   5555:30703/TCP,50071:31422/TCP   12m
service-otg-port-eth2         LoadBalancer  10.96.236.9     172.18.0.89   5555:31340/TCP,50071:31332/TCP   12m
```

Make the change to the go file and set the transport to gRPC.

> vim lab-05-3_test.go

#5-24

```go
func TestEbgpRoutePrefix(t *testing.T) {

        testConst := map[string]interface{}{
                // "controller_location": "https://172.18.0.87:8443",
                "controller_location": "172.18.0.86:40051",
                "p1_location":  "eth1",
                "p2_location":  "eth2",
                                              _

        "txAdvRouteV6": "::10:10:10:1",
        "rxAdvRouteV6": "::20:20:20:1",
}

api := gosnappi.NewApi()

// api.NewHttpTransport().SetLocation(testConst["controller_location"].(string))
api.NewGrpcTransport().SetLocation(testConst["controller_location"].(string))

c := ebgpRoutePrefixConfig(testConst)
```

Run the gosnappi test.

> `go test -v lab-05-3_test.go`

#5-25

ℹ️ The KENG-controller pod includes the "gnmi" container which can be used for metrics and states retrieval. See more details on gNMI

> `kubectl describe pods otg-controller -n ixia-c-b2b`

#5-26

Let's see the External IP address used by the controller for the gNMI service

> `kubectl get services -n ixia-c-b2b`

#5-27

```
NAME                          TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                        AGE
service-gnmi-otg-controller   LoadBalancer   10.96.23.16    172.18.0.85   50051:32430/TCP                36m
service-grpc-otg-controller   LoadBalancer   10.96.225.154  172.18.0.86   40051:30970/TCP                36m
service-https-otg-controller  LoadBalancer   10.96.8.249    172.18.0.87   8443:30225/TCP                 36m
service-otg-port-eth1         LoadBalancer   10.96.129.89   172.18.0.88   5555:30703/TCP,50071:31422/TCP 36m
service-otg-port-eth2         LoadBalancer   10.96.236.9    172.18.0.89   5555:31340/TCP,50071:31332/TCP 36m
```

Let's retrieve the flow statistics using the gnmi-c client.

gnmic -a 172.18.0.85:50051 --skip-verify -u admin -p admin get --path "flows"

#5-28

The flow statistics follow these Flow YANG models

Let's retrieve the BGP statistics using the gnmi-c client

gnmic -a 172.18.0.85:50051 --skip-verify -u admin -p admin get --path "bgp-peers"

#5-29

The BGP statistics follow these BGP YANG models

CLEANUP:

kind delete cluster --name kne

#5-30

Lab-Demo #06 – KENG DEMO WITH HARDWARE

☑ This lab uses SNAPPI to control the configuration of the Keysight AresONE 400G ports connected to a DUT. This is a repeat of the convergence part of Lab#04 and we're using docker-compose to deploy the KENG. The goal of this test is to demonstrate running at full 400 Gbps line rate and measure BGP data plane convergence.
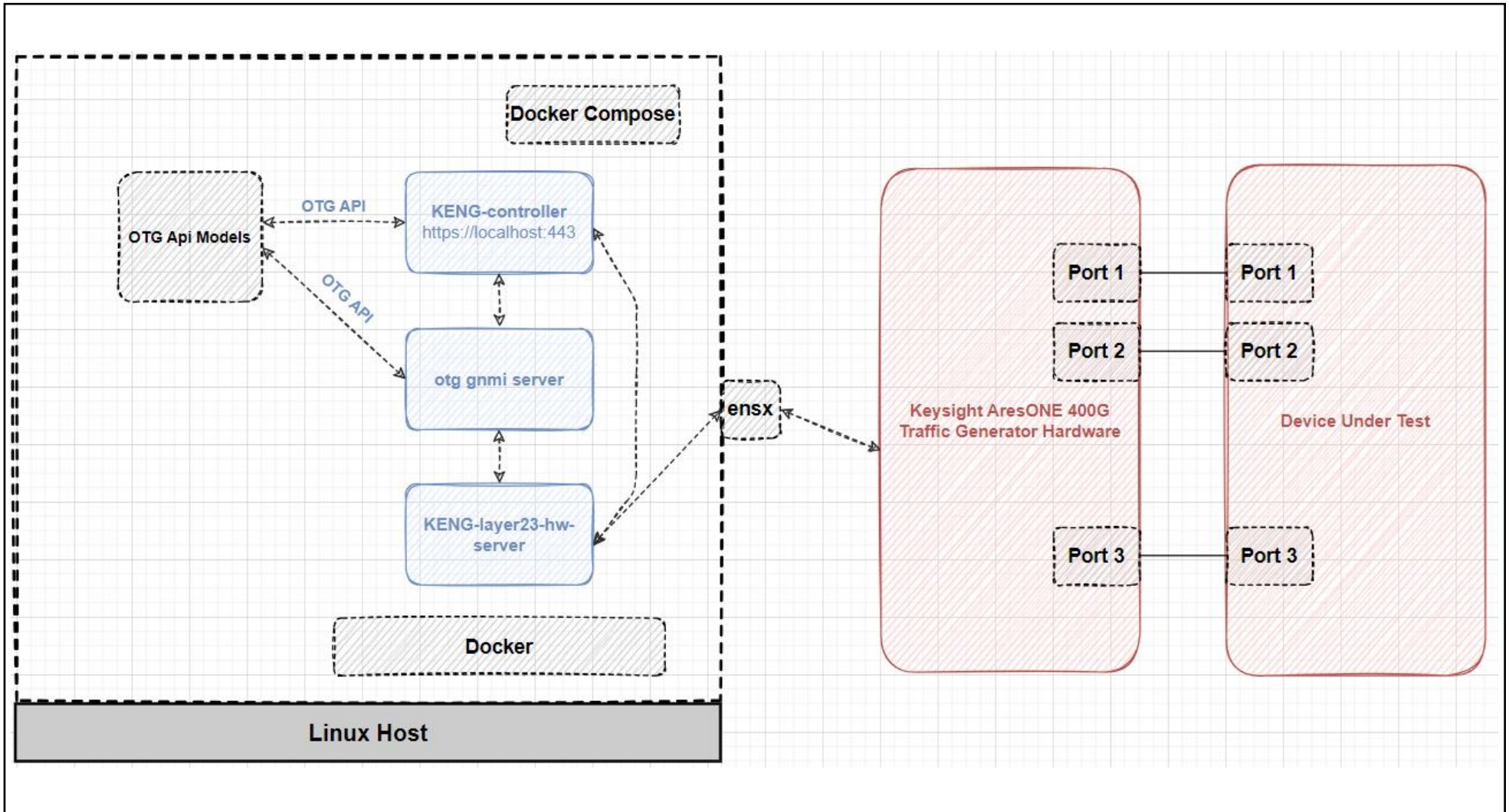
DEPLOYMENT:

```
cd ~/ac2-workshop/lab-06-demo

#6-01    docker-compose up -d
```

```
docker ps

#6-02
```

```
CONTAINER ID    IMAGE                                              COMMAND              CREATED        STATUS        PORTS
                                                     NAMES
c0dbff385550    ghcr.io/open-traffic-generator/otg-gnmi-server:1.14.15    "bin/gnmid -http-ser…"  7 seconds ago  Up 6 seconds  0.0.0.0:50051->50051/tcp, :::50051->5005
1/tcp                                                lab-06-demo_gnmi-server_1
c3f92ac74844    ghcr.io/open-traffic-generator/keng-controller:1.14.0-1  "./bin/controller --…"  7 seconds ago  Up 7 seconds  0.0.0.0:8443->8443/tcp, :::8443->8443/tc
p, 0.0.0.0:40051->40051/tcp, :::40051->40051/tcp    lab-06-demo_controller_1
5078bd1c4fb1    ghcr.io/open-traffic-generator/keng-layer23-hw-server:1.14.0-1  "dotnet otg-ixhw.dll…"  8 seconds ago  Up 7 seconds  0.0.0.0:5000-5001->5000-5001/tcp, :::500
0-5001->5000-5001/tcp                                lab-06-demo layer23-hw-server 1
```

We're setting the frame rate as line rate percentage of the hardware transmitting port. This will result in very high rate so it's better to use a fixed traffic duration instead of fixed number of packets. The actual packet per second rate is extracted from the flow statistics during the transmission and used in the convergence time calculation.

```
vim lab-06_test.py

#6-03
```

```python
f = c.flows.add()
f.duration.fixed_seconds.seconds = tc["trafficDuration"]
f.rate.percentage = tc["lineRatePercentage"]
f.size.fixed = tc["pktSize"]
f.metrics.enable = True
```

EXECUTION:

Run the test

```
python3 lab-06_test.py
```

#6-04

The total observed bitrate is "Bytes Tx Rate" * 8 ~ 394.7Gbps

```
-----------------------------------------------------------------------------------
Flow Metrics
-----------------------------------------------------------------------------------
Name         State      Frames Tx      Frames Rx      FPS Tx      FPS Rx      Bytes Tx          Bytes Rx
bgpFlow      started    1973684210     1973660635     2620063     2631253     2960490952500     2960490952500
-----------------------------------------------------------------------------------


2024-11-05 02:21:32.246974 Getting port metrics    ...


-----------------------------------------------------------------------------------
Port Metrics
-----------------------------------------------------------------------------------
Name     State     Frames Tx      Frames Rx      FPS Tx      FPS Rx     Bytes Tx        Bytes Rx        Bytes Tx Rate   Bytes Rx Rate
p1       None      1968970411     7              32894732    0          2953455608914   822             49342100000     0
p2       None      6              1057577568     0           0          506             1586366340892   0               0
p3       None      5              911367713      0           5452483    388             1367051560469   0               8178724000
-----------------------------------------------------------------------------------


Done waiting for traffic stopped
2024-11-05 02:21:39.248060 Convergence time was 0.0007166800183470085s
```

CLEANUP:

```
docker-compose down
```

#6-05

SUMMARY

This document has described the Ixia-c Community Edition / Keysight Elastic Network Generator (in short KENG) lab exercises for the 2024 Autocon2 WS-D1 workshop. It has covered the environment configuration required for the lab exercises, the infrastructure tools required to manage the test topology, and the test tools required to generate traffic and emulate protocols.

These lab exercises have included various OTG Test Tools (such as IXIA-C / KENG), various OTG API Clients (such as OTGEN / GOSNAPPI / SNAPPI / CURL JSON), various infrastructure options (such as DOCKER CLIENT / DOCKER COMPOSE / CONTAINER LAB / KIND-KNE), and various test topologies (such as B2B / DUT).