Python Click API documentation

Jump to bottom

Roberto Riggio edited this page on Aug 18, 2019 · 1 revision

Table of Contents

Overview

In this section we shall first introduce some fundamental concepts like *CPPs*, and *LVNFs*, then we shall describe all the Click primitives available to applications developers. Notice how all the methods described in the section are class methods of the `EmpowerApp` class and as such can be invoked only from applications extending this base class.

Prerequisites

Notice that in addition to the EmPOWER Runtime, in order to properly use the LVNF Functions supported by the EmPOWER framework you must also run an OpenFlow Controller implementing the Intent based networking interface necessary for Service Function Chaining. Instructions for setting up this controller can be found here.

Notice also that this tutorial assumes that the entire backhaul of your network is OpenFlow enabled. This includes the WTPs/CPPs (which already run by default an OpenVSwitch instance) as well as all the other switches interconnecting the WTPs/CPPs. All these switches must be configured to point to EmPOWER Ryu Controller.

Click Packet Processor

It is the a machine combining switching with packet processing capabilities. Switching is provided by OpenVSwitch, while the packet processing capabilities are provided by Click. CPPs are typically equipped with multiple ethernet ports. Instructions about how to build a CPP can be found here.

The list of CPPs available to the application can be accessed using:

```
self.cpps()
```

Light Virtual Network Function Abstraction

Light Virtual Network Functions (LVNFs) are essentially Click scripts running in a certain CPP. LVNFs are instantiated starting from VNF templates called Images.

Each LVNF Image consists of a Click script together with some additional information such as the number of input/output virtual ports used by the LVNF, and the list of Click handlers exposed by the Image.

Handlers are used in order to manipulate the internal state of the LVNF. For example, in the case of a Firewall LVNF, specific handlers shall be defined in order to allow the network operator to add/remove firewall rules.

For example consider the following code:

```
# Create Image
img = Image(vnf="in_0 -> Null() -> out_0")
```

The Image consists of a very simple Click script (the Null element) which uses one input and one output virtual port. Notice how *in_0* and *out_0* are the names of the two Click elements in charge of the network I/O. Upon deployment, the Image will be extended with the missing elements declaration. The actual Click script that will be deployed on the CPP will be the following:

```
in_0 :: FromHost(br0-0-0)
out_0 :: ToHost(br0-0-0)
in_0 -> Null() -> out_0
```

The virtual network interface *br0−0−0* is dynamically created when the Click instance is spawned on the target CPP. The interface name is dynamically generated in such a way to avoid collision with other LVNFs running on the same CPP. In particular br0 is the name of the bridge to which the interface will be connected, the first number is an incremental number, while the second number is the LVNF port number. Notice however that the users need not to know the actual interface(s) used by the LVNF.

The LVNF can be spawned using the following code:

```
# Spawn LVNF
self.spawn_lvnf(img, cpp)
```

This will result in a new Click instance to be started on the target CPP using the Click script above. After the Click instance is running the LVNF Agent will add the LVNF ports to the OpenVSwitch bridge on the CPP.

At this point the LVNF is ready to receive and process traffic.

The LVNF can be removed using the following code:

```
# Delete an LVNF
self.delete_lvnf(lvnf.lvnf_id)
```

A complete application deploying and removing a sample LVNF is reported in the code below:

```
from empower.core.image import Image
from empower.core.app import EmpowerApp
from empower.core.app import DEFAULT_PERIOD


class LvnfDeploy(EmpowerApp):

    def __init__(self, **kwargs):
        EmpowerApp.__init__(self, **kwargs)
        self.cppup(callback=self.cpp_up_callback)
        self.lvnfjoin(callback=self.lvnf_join_callback)
        self.lvnfleave(callback=self.lvnf_leave_callback)

    def cpp_up_callback(self, cpp):
        """Called when a new cpp connects to the controller."""

        self.log.info("CPP %s connected!" % cpp.addr)

        # Create Image
        img = Image(vnf="in_0 -> Null() -> out_0")

        # Spawn LVNF
        self.spawn_lvnf(img, cpp)

    def lvnf_join_callback(self, lvnf):
        """Called when an LVNF associates to a tennant."""

        self.log.info("LVNF %s joined %s" % (lvnf.lvnf_id, lvnf.tenant_id))

        # Stop LVNF
        lvnf.stop()
    def lvnf_leave_callback(self, lvnf):
        """Called when an LVNF leaves a tenant."""

        self.log.info("LVNF %s stopped" % lvnf.lvnf_id)


def launch(tenant_id, every=DEFAULT_PERIOD):
    """ Initialize the module. """

    return LvnfDeploy(tenant_id=tenant_id, every=every)
```

Virtual Network Port Abstraction

LVNFs, as well as Images, do not define which kind of traffic they should process.

Users can define the logical sequence of LVNFs a certain portion of the flow-space must traverse, using the LVNF's Virtual Network Ports.

LVNFs can have one or more Virtual Ports.

Each Virtual Port, is associated to one, and only one, virtual network interface in a CPP which in time is attached to one actual port of an OpenVSwitch bridge.

Users do not need to care about the virtual interface name on a CPP nor about the switch data-path identifiers (DPID) or physical port number. Instead, they can perform LVNF chaining by using the virtual port identifiers

For example in the case of the LVNF above, the following code access the first virtual port of the LVNF:

```
port = lvnf.ports[0]
print(port)
```

This will print something like

```
00:00:24:D1:61:ED ovs_port 43 virtual_port 0 hwaddr A6:0A:2C:D0:D3:74 iface vnf-br0-46-
0
```

This means that the LVNF is attached to DPID `00:00:24:D1:61:ED` on port *43*. The virtual port id is *0* and the virtual interface mac address and name are, respectively, `A6:0A:2C:D0:D3:74` and *vnf-br0-46-0*.

Now assume that you want all the HTTP requests made by a certain wireless client to be redirected to a certain LVNF. This can be achieved with the following code:

```
lvap.ports[0].next["tp_dst=80"] = lvnf.ports[0]
```

This will result in all the traffic generated by the specified LVAP and matching the specified rule to be forwarded to the first port of the LVNF. From a technical point of view this is done by tagging all the matching traffic at the source point with a unique VLAN id and then installing in the network a set of rules to forward the traffic between the two points.

Events

The following table lists the EmPOWER Click events.

Primitive	Parameters	Description
lvnf_join	lvnf	Callback when a LVNF joins a tenant.
lvnf_leave	lvnf	Callback when a LVNF leaves a tenant.
cpp_up	vbs	Callback when a CPP comes online.
cpp_down	vbs	Callback when a CPP goes offline.

Give a look at this Network App for some examples about how to use events.

LVNF Primitives

The following table lists the EmPOWER LVNF primitives. All primitives are non–blocking and, as such, they immediately return control to the calling Network App.

An optional callback method can be specified for Poller/Trigger primitives. The specified callback method will be called when the primitive returns.

All polling primitives are executed periodically with the period set by the *every* parameter (in ms). Specifying `every = -1` will result in a single query being issued.

Primitive	Parameters	Mode	Description
lvnf_stats	lvnf, every	Polling	Fetch LVNF statistics.

Clone this wiki locally

```
https://github.com/clicknf/clicknf.github.io.wiki.git
```