实验室 2 - 第 1 部分

OpenFlow python 控制器 (POX)

目标

将外部控制器连接到 mininet optical，并用 python 编写一个简单的控制器。

你将学到什么

- 使用基于 python 的 POX 控制器框架创建一个简单的 openflow 控制器。
- 使用 python 在 Mininet 中构建树状拓扑结构。
- 使用控制器在树状拓扑网络中转发流量。(第 2 层转发应用程序)
- 修改第 2 层转发应用程序，使其具有阻止特定流量的功能（防火墙应用程序）

先决条件

在 Mininet 中定义拓扑。Arp 知识

开流控制器

在本实验室中，我们将在 Mininet 中创建一个基本的网络树形拓扑。我们将使用 Mininet 外部的控制器，而不是在 Openflow 交换机中手动插入规则。我们将使用的 Openflow 控制器名为 POX，是用 Python (https://noxrepo.github.io/pox-doc/html/#) 编写的。因此，它使用 Python 面向对象方法来掩盖控制器的复杂性。这样，就可以用短短几行代码轻松编写复杂的控制器。
我们使用 POX *_handle_ConnectionUp()* 和 *_handle_PacketIn()函数*。这些函数分别侦听来自交换机的连接请求和从交换机传输到控制器的数据包。

在 ~/pox/pox/misc/ 目录下创建以下文件 *simple_controller.py*

```
from pox.core import core
import pox.lib.packet as pkt
```

```
import pox.openflow.libopenflow_01 as
of from pox.lib.util import dpidToStr
from pox.lib.addresses import

EthAddr log = core.getLogger()

table={}

def launch ():
    core.openflow.addListenerByName("ConnectionUp", _handle_ConnectionUp)
    core.openflow.addListenerByName("PacketIn", _handle_PacketIn)
    log.info("Switch running.")

def _handle_ConnectionUp ( 事件):
    log.info("Starting Switch %s", dpidToStr(event.dpid))
    msg = of.ofp_flow_mod(command = of.OFPFC_DELETE)
    event.connection.send(msg)


def _handle_PacketIn ( event):
    dpid = event.connection.dpid
    sw=dpidToStr(event.dpid)
    inport = event.port
    数据包 = 事件解析
    print("Event: switch %s port %s packet %s" % (sw, inport, packet))
```

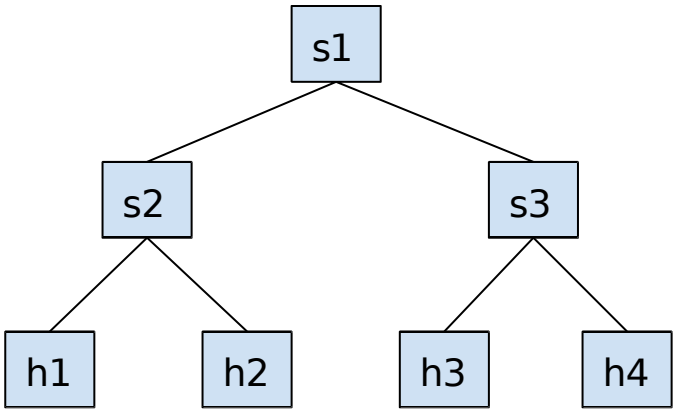
要运行该控制器，请输入

```
cd ~/pox
./pox.py log.level --DEBUG misc.simple_controller
```

该控制器所做的只是报告来自交换机的连接请求以及交换机转发给它的任何数据包。我们将在 Mininet 中创建的简单拓扑中使用它。

创建树形拓扑

Mininet 树状拓扑结构使用 2 层交换机，4 台主机 h1、h2、h3 和 h4 与之相连，如下图所示。



您将在下面的程序中看到，我们在实例化 Mininet 时定义了 controller=RemoteController 变量。Mininet 将尝试连接到位于同一台机器（localhost 或 ip 地址 127.0.0.1）上监听 TCP 端口 6633 的控制器。所有 openflow 交换机都将使用自己专用的 Openflow 通道连接到控制器，并接受来自控制器的规则。我们为四台主机分别定义了 IP 地址和简单的 MAC 地址。我们将四台主机连接到 2 台底层交换机 s2 和 s3，并创建从 s2 和 s3 到根交换机 s1 的链接。

这是准备 mininet 拓扑的脚本

```
从 mininet.net 导入 Mininet
from mininet.node import RemoteController
from mininet.cli import CLI
从 mininet.log 导入 setLogLevel, info

def treeTopo()：
    net = Mininet( controller=RemoteController )

    info( '*** Adding controller\n' )
    net.addController('c0')

    info( '*** Adding hosts\n' )
    h1 = net.addHost( 'h1', ip='10.0.0.1', mac='00:00:00:00:00:01' )
    h2 = net.addHost( 'h2', ip='10.0.0.2', mac='00:00:00:00:00:02' )
    h3 = net.addHost( 'h3', ip='10.0.0.3', mac='00:00:00:00:00:03' )
    h4 = net.addHost( 'h4', ip='10.0.0.4', mac='00:00:00:00:00:04' )

    info( '*** Adding switches\n' )
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )
    s3 = net.addSwitch( 's3' )
```

```
    info( '*** Creating links\n' )
    net.addLink( h1, s1 )   # 带宽（Mb/s）和延迟可按如下方式添加 # net.addLink(
                                h1, s1 ,cls=TCLink, bw=50,delay='10ms')

    net.addLink(  h2,  s1
    )   net.addLink(   h3,
    s2   )   net.addLink(
    h4, s2 )

    根 = s1 层 1 =
    [s2,s3]

    for idx,l1 in enumerate(layer1):
        net.addLink( root,l1 )

    info( '*** Starting network\n')

######################################## 这一部分到这里创建了 mininet
拓扑
    net.start()   # 此命令启动 mininet 中的拓扑结构

    info( '*** 运行 CLI\n' )
    CLI( net ) # 该命令创建 mininet 提示符，这样我们就可以通过同一终端（或外
部控制器链接）与 # mininet 交互。

    info( '*** 停止网络' )
    net.stop() # 在 mininet 提示符下输入退出命令后调用该命令

如果 name_== 'main'_:
    setLogLevel( 'info' )
    treeTopo()
```

标准的以太网交换机会向所有端口广播，以了解所有主机和设备的位置，而我们刚才描述的拓扑结构则不同，它是哑巴结构，没有转发规则。它必须依靠手动或控制器插入的规则来转发流量。

运行 simple_controller 和 tree0.py 拓扑

分别运行控制器和 tree0.py 拓扑生成器。

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.simple_controller
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
INFO:misc.simple_controller:Switch running.
DEBUG:core:POX 0.7.0 (gar) going up...
DEBUG:core:Running on CPython (3.8.5/Jul 28 2020 12:59:40)
DEBUG:core:Platform is Linux-5.4.0-42-generic-x86_64-with-glibc2.29
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:misc.simple_controller:Starting Switch 00-00-00-00-00-01
INFO:openflow.of_01:[00-00-00-00-00-02 3] connected
INFO:misc.simple_controller:Starting Switch 00-00-00-00-00-02
INFO:openflow.of_01:[00-00-00-00-00-03 4] connected
INFO:misc.simple_controller:Starting Switch 00-00-00-00-00-03
```

```
mininet@mininet-vm:~$ sudo python tree0.py
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts
*** Adding switches
*** Creating links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Running CLI
*** Starting CLI:
```

左为控制器。右为 Mininet 拓扑

您将看到交换机 s1、s1 和 s2 与控制器注册。网络上尚未产生任何流量。我们可以通过从 h1 ping 主机 h2 来产生简单的流量。

```
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
```

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
```

左为控制器。右为 Mininet 拓扑

从 h1 到 h2 的 ping 失败。为什么？因为交换机的流量表为空，不知道如何在 h2 和 h1 之间传输流量。请参考实验室 2 回答此问题。Ping 使用 IP 地址指向主机，而交换机使用以太网地址指向主机。Arp 用于关联主机上的主机 IP 地址和以太网地址。

考虑一下：在一个大型开放式网络中，用户试图发送机密流量，那么主机如何了解远程主机的以太网/MAC 地址，这种方法（Arp）的安全性如何?

转发应用程序

增强外壳控制器，以学习第 2 层转发规则。

我们可以使用 simple_controller 中的 *table{}* 数据结构来记录主机在网络中的位置。当我们生成 ping 时，源机器上的网络协议栈会发送一个 arp 请求，询问 "*哪个主机（即以太网地址）拥有此 IP 地址？*我们修改控制器，将该请求淹没到所有端口（除了它进来的端口）。拥有该 IP 地址的主机会回应 "*我就是拥有该 IP 地址的主机（即以太网地址）*"，控制器会将该回应泛洪到所有其他端口。这样，即使只有一台主机提出 Arp 请求问题，所有主机也都知道了答案。

然后，控制器可在每个交换机中生成与源地址和目标以太网地址相匹配的流量表规则，并将流量转发到正确的端口。

```
def _handle_PacketIn ( event):
    dpid = event.connection.dpid
    sw=dpidToStr(event.dpid)
    inport = event.port
    数据包 = 事件解析
    print("Event: switch %s port %s packet %s" % (sw, inport, packet))

    # 学习源文件表[（事件.连接，数据包.源文件）] = 事件
    .端口

    dst_port = table.get((event.connection,packet.dst))

    如果 dst_port 为空：
        # 交换机不知道目的地，因此将信息全部发送出去。
端口。
        # 我们可以使用特殊端口 OFPP_FLOOD 或 OFP_ALL。# 但并非所有
        交换机都支持 OFPP_FLOOD。
        msg = of.ofp_packet_out(data = event.ofp)
        msg.actions.append(of.ofp_action_output(port = of.OFPP_ALL))
        event.connection.send(msg)
    否则
        # 交换机知道目的地，因此可以路由数据包。我们还在交换机中安装了转发规
则
        msg = of.ofp_flow_mod()
        msg.priority=100
        msg.match.dl_dst = packet.src
        msg.match.dl_src = packet.dst
        msg.actions.append(of.ofp_action_output(port = event.port))
        event.connection.send(msg)

        # 我们必须转发接收到的数据包... msg =
        of.ofp_packet_out()
        msg.data = event.ofp
        msg.actions.append(of.ofp_action_output(port = dst_port))
        event.connection.send(msg)

        log.debug("Installing %s <-> %s" % (packet.src, packet.dst))
```

这次从 h1 到 h2 的 ping 成功了。控制器在两台主机之间正确传输 arp 请求和响应数据包，ping 流量也正确传输。

```
Event: switch 00-00-00-00-00-01 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-02 port 3 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-03 port 1 packet [00:00:00:00:00:01>ff:ff:ff:ff:ff:ff ARP]
Event: switch 00-00-00-00-00-01 port 2 packet [00:00:00:00:00:02>00:00:00:00:00:01 ARP]
DEBUG:misc.simple_controller:Installing 00:00:00:00:00:02 <-> 00:00:00:00:00:01
```

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=23.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.116 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.118 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.122 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.122 ms
```

防火墙应用程序

防火墙规则和 POX 实施

利用 simple_controller，我们现在可以构建稍微复杂一些的功能，例如防火墙。防火墙会在特定条件下阻止流量。在下面的练习中，条件是

1. 默认情况下，所有主机都可以与其他主机通信，但...
2. h1 和 h2 可能无法相互通信
3. h3 和 h4 可能无法相互通信

我们用 2 个图元来修改 simple_controller 的规则 2 和 3。

rules=[['00:00:00:00:00:01','00:00:00:00:00:02'],['00:00:00:00:00:03', '00:00:00:00:00:04']]

我们在 simple_controller 的初始化阶段将这些规则插入开关中，此时 _handle_ConnectionUp()被触发。

```
def _handle_ConnectionUp ( 事件)：
    log.info("Starting Switch %s", dpidToStr(event.dpid))
    msg = of.ofp_flow_mod(command = of.OFPFC_DELETE)
    event.connection.send(msg)

    为 rules 中的 rule：
        block = of.ofp_match()
        block.dl_src = EthAddr(rule[0])
        block.dl_dst = EthAddr(rule[1])
        flow_mod = of.ofp_flow_mod()
        flow_mod.match = block
        flow_mod.priority = 32000
        event.connection.send(flow_mod)
```

你会注意到，这些区块规则的优先级很高（32000），并且优先于随后学习的优先级较低的规则，我

们已经将优先级定义为 100。

根据拓扑结构测试防火墙应用程序

为了测试拓扑结构和防火墙应用程序，我们使用 Mininet pingall 应用程序从每台主机 ping 到其他每台主机。

mininet> pingall
**** Ping：测试 ping 的可达性 h1*
-> X h3 h4
h2 -> X h3
h4 h3 -> h1
h2 X h4 ->
h1 h2 X
**** 结果：33% 被放弃（8/12 收到）*

在这里，我们可以看到 h1 可以连接到 h3 和 h4，但被阻止连接到 h2，这是意料之中的。同样，防火墙也会阻止 h2 连接 h1，阻止 h3 连接 h4，阻止 h4 连接 h3。

您应该检查交换机中的流量表规则。您会注意到用于执行流量阻塞的优先级较高的规则，以及用于传递流量的优先级较低的规则。

mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
cookie=0x0, duration=171.743s, table=0, n_packets=27, n_bytes=2142, priority=32000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=drop
cookie=0x0, duration=171.742s, table=0, n_packets=0, n_bytes=0, priority=32000,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:04 actions=drop
cookie=0x0, duration=113.474s, table=0, n_packets=0, n_bytes=0, priority=100,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=输出: "s1-eth2"
cookie=0x0, duration=113.474s, table=0, n_packets=12, n_bytes=728, priority=100,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output: "s1-eth1"
cookie=0x0, duration=72.563s, table=0, n_packets=6, n_bytes=532, priority=100,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03 actions=output: "s1-eth3"
cookie=0x0, duration=72.563s, table=0, n_packets=7, n_bytes=574, priority=100,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01 actions=output: "s1-eth1"
cookie=0x0, duration=54.864s, table=0, n_packets=3, n_bytes=238,

priority=100,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:04
actions=output: "s1-eth3"

cookie=0x0, duration=54.864s, table=0, n_packets=4, n_bytes=280, priority=100,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:01 actions=output: "s1-eth1"
cookie=0x0, duration=44.847s, table=0, n_packets=3, n_bytes=238, priority=100,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:03 actions=output: "s1-eth3"
cookie=0x0, duration=44.847s, table=0, n_packets=4, n_bytes=280, priority=100,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:02 actions=output: "s1-eth2"
cookie=0x0, duration=44.832s, table=0, n_packets=3, n_bytes=238, priority=100,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:04 actions=output: "s1-eth3"
cookie=0x0, duration=44.832s, table=0, n_packets=4, n_bytes=280, priority=100,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:02 actions=output: "s1-eth2"