OpenDayLight Tutorial

From OpenFlow Wiki

Jump to: navigation, search

Following is an OpenDayLight Application Developers' tutorial for beginners.

Contents

[hide]

- 1 Setup

- 1 Setup
- 2 Basic theory
- 3 Development in OpenDayLight
- 4 Workflow, dependencies and information sharing
- 5 Creating a new module
- 6 Sample application: MAC Hub or Learning Switch
- 7 Credits

Setup

Following are basic environment setup to take care of before starting development.

- Download **this VM** (It already has a version of OpenDayLight, updated on 10 June 2013, compiled through Maven and built with Eclipse). The usage of the VM is similar to what is extensively explained in OpenFlow_Tutorial#Set_up_Virtual_Machine
  - The OVA file has an Ubuntu 12.10 VM with many other controllers installed.
  - **Username and passwd are both "ubuntu"**

- In case you want to do it from scratch, I recommend the following two excellent tutorials from Brent Salisbury:
  - OpenDaylight OpenFlow Controller setup tutorial
  - Import, Build, and Run OpenDaylight in Eclipse

- Run Mininet in your environment using the following command:

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

- Run OpenDayLight within Eclipse:
  - Double click on "distribution.opendaylight" in the Project Explorer of Eclipse
  - Right click on "opendaylight-application.launch" and click "Run As" and then click on "1. opendaylight-application.launch".

- With OpenDayLight running within Eclipse, you can now visit http://192.168.56.101:8080 (assuming that is the IP address of the VM) to get the web front of the controller platform, implemented through REST. Login as "admin" and use password "admin". Play around with the interface.

Basic theory

The high-level overview of the controller is available **here**. The Service Abstraction Layer is your friend for most development aspects.

OpenDayLight uses the following software tools/paradigms. It is important to become familiar with them:

- **Maven**: OpenDayLight uses Maven for easier build automation. Maven uses pom.xml (Project Object Model for this bundle) to script the dependencies between bundles and also to describe what bundles to load on start.
- **OSGi**: This framework in the backend of OpenDayLight allows dynamically loading bundles and packaged Jar files, and binding bundles together for information exchange.
- **Java interfaces**: Java Interfaces are used for event listening, specifications and forming patterns. This is the main way in which specific bundles implement call-back functions for events and also to indicate awareness of specific state.

The controller platform has certain core bundles, each of which export important services through Java interfaces. Here is a brief list of important ones that come in handy while developing network services. See **this link** for more information.

Bundle	Exported interface	Description
arphandler	IHostFinder	Component responsible for learning about host location by handling ARP.
hosttracker	IfIptoHost	Track the location of the host relatively to the SDN network.
switchmanager	ISwitchManager	Component holding the inventory information for all the known nodes (i.e., switches) in the controller.
topologymanager	ITopologyManager	Component holding the whole network graph.
usermanager	IUserManager	Component taking care of user management.
statisticsmanager	IStatisticsManager	Component in charge of using the SAL ReadService to collect several statistics from the SDN network.
sal	IReadService	Interface for retrieving the network node's flow/port/queue hardware view
sal	ITopologyService	Topology methods provided by SAL toward the applications
sal	IFlowProgrammerService	Interface for installing/modifying/removing flows on a network node
sal	IDataPacketService	Data Packet Services SAL provides to the applications
web	IDaylightWeb	Component tracking the several pieces of the UI depending on bundles installed on the system.

Development in OpenDayLight

Most controller platforms expose some native features to allow these key features:

- Ability to listen to asynchronous events (e.g., PACKET_IN, FLOW_REMOVED) and to register call back functions (e.g., receiveDataPacket)
- Ability to parse incoming packets (e.g., ARP, ICMP, TCP) and fabricate packets to send out into the network
- Ability to create and send an OpenFlow/SDN message (e.g., PACKET_OUT, FLOW_MOD, STATS_REQUEST) to the programmable dataplane.

With OpenDayLight you can achieve all of those using a combination of SAL implementation classes and listener interfaces. For instance, to allow a module to receive packets sent by the switch to the controller, the class needs to implement IListenDataPacket:

```
public class TutorialL2Forwarding implements IListenDataPacket
```

When the class implements that *IListenDataPacket* interface, any packet that arrives at a node without a pre-cached rule will result in a callback of *receiveDataPacket*. So overriding that method allows the application to get a copy of the packet.

```
public PacketResult receiveDataPacket(RawPacket inPkt) { ... }
```

Once the packet is received, you can decode the packet using the *dataPacketService*. Usage of command is shown below. Today, according to the *org.opendaylight.controller.sal.packet* package, we can inspect the packet headers the following packet types: ARP, Ethernet, ICMP, IPv4, LLDP, TCP, UDP.

```
Packet formattedPak = this.dataPacketService.decodeDataPacket(inPkt);
```

Along with the packet header fields, you can extract the incoming node (i.e., switch) and node-connector (i.e., switchport) using the following:

```
NodeConnector incoming_connector = inPkt.getIncomingNodeConnector();
```

I use the following two useful commands to extract header details to use in the MAC learning switch:

```
byte[] srcMAC = ((Ethernet)formattedPak).getSourceMACAddress();
long srcMAC_val = BitBufferHelper.toNumber(srcMAC);
```

I can use the *dataPacketService* to send a packet out of a node-connector (switchport). In the following example, I resend the incoming packet out of node-connector 'p':

```
RawPacket destPkt = new RawPacket(inPkt);
destPkt.setOutgoingNodeConnector(p);
this.dataPacketService.transmitDataPacket(destPkt);
```

Besides a PACKET_OUT, we can also perform a FLOW_MOD style rule insertion into a switch. For that, we use the Match, Action and Flow classes. Here is an example of how to create a match rule where the IN_PORT matches the value extracted from the PACKET_IN:

```
Match match = new Match();
match.setField(MatchType.IN_PORT, incoming_connector);
```

There are several other fields you can match one. Following are what I see in the *org.opendaylight.controller.sal.match*: "inPort", "dlSrc", "dlDst", "dlVlan", "dlVlanPriority", "dlOuterVlan", "dlOuterVlanPriority", "dlType", "nwTOS", "nwProto", "nwSrc", "nwDst", "tpSrc", "tpDst".

In the package *org.opendaylight.controller.sal.action*, I see the following supported actions: "drop", "loopback", "flood", "floodAll", "controller", "interface", "software path", "harware path", "output", "enqueue", "setDlSrc", "setDlDst", "setVlan", "setVlanPcp", "setVlanCif", "stripVlan", "pushVlan", "setDlType", "setNwSrc", "setNwDst", "setNwTos", "setTpSrc", "setTpDst", "setNextHop". Here is an example of creating an action list for the flow rule, for OUTPUT to an *outgoing_connector* that is pre-calculated.

```
List<Action> actions = new ArrayList<Action>();
actions.add(new Output(outgoing_connector));
```

Once the match rule is formed and action list is created, we can create a Flow and add it to a particular switch. In this case, the rule is inserted to the switch where the original packet arrived:

```
Flow f = new Flow(match, actions);
Status status = programmer.addFlow(incoming_node, f);
if (status.isSuccess())
    logger.trace("Installed flow {} in node {}", f, incoming_node);
else
    //What to do in case of flow insertion failure?
```

Once the packet is appropriately handled, the controller application has 3 choices on returning status back to the SAL. It can be

- return PacketResult.CONSUME: Packet has been processed and none in the chain after us should handle it
- return PacketResult.KEEP_PROCESSING: Packet has been processed, but still pass onto other bundles
- return PacketResult.IGNORED: Packet has been ignored and ready to pass to other bundles

One other feature that was useful was generating the list of ports in a node/switch. Assuming *incoming_node* is the switch whose ports I want to list, I use the SwitchManager class as follows:

```
Set<NodeConnector> nodeConnectors =
this.switchManager.getUpNodeConnectors(incoming_node);
```

Workflow, dependencies and information sharing

- **pom.xml**: Generally the pom.xml is used to indicate the inter-bundle dependencies. The syntax for that must be fairly apparent from the format of the pom.xml in the arphandler directory

- **Activator.java**: This handles the configuration of the dependencies during run-time. The configureInstance function in the bottom of Activator.java of each bundle has two main tasks:
    1. tells the list of Java interfaces implemented by a bundle 'X' using *setInterface* call
    2. registers other bundles that the bundle 'X' uses using *add* call

- **set/unset binding**: The bundle 'X' gets a reference to the instantiated object of other needed bundle 'Y' through the *setY* and *unsetY* calls in the main class. For example, TutorialL2Forwarding needs data from SwitchManager. This is accomplished by defining two methods *setSwitchManager* and *unsetSwitchManager*. These are called by OSGi and object references are appropriately passed.

Now that you're updated on the theory, you are now ready to create a new module.

Creating a new module

Assume that the main folder where opendaylight is installed is /home/ubuntu/opendaylight, follow these instructions to add a new module/bundle:

- Copying an existing module
    - Generally one can right click on an existing project folder in Eclipse (e.g., arphandler) and select "Copy" and then "Paste" it in a new name (say, "tutorial_L2_forwarding").
    - Replace all instances of the "ARPHandler" class with "TutorialL2Forwarding" and other instances of "arphandler" with "tutorial_L2_forwarding" in all the files in the directory ~/opendaylight/opendaylight/tutorial_L2_forwarding, including pom.xml
- Alternatively, you can copy the files in **this tar file** into the ~/opendaylight/opendaylight directory and "Import" that project into Eclipse.
- Changes to make to integrate this new project with OpenDayLight controller:
    - Include "tutorial_L2_forwarding" project in the distribution's POM file: ~/opendaylight/opendaylight/distribution/opendaylight/pom.xml. Right below the "arphandler", you can add the following line:

```
<module>../../tutorial_L2_forwarding</module>
```

- - Similarly add the following line into the logger's configuration file: ~/opendaylight/opendaylight/distribution/opendaylight/src/main/resources/configuration/logback.xml

```
<logger name="org.opendaylight.controller.tutorial_L2_forwarding" level="INFO"/>
```

- - In Eclipse, you need to add the new module to the opendaylight-application.launch file. You can do that by right clicking on the launch file and selecting "Run As" -> "Run Configurations". In the "Source" tab of the popup dialog, click "Add" -> "Java Project" -> Choose "org.opendaylight.controller.tutorial_L2_forwarding".

All the changes made to the main ~/opendaylight directory can also be seen in **this patch file**. Now you're all set to edit the code of the application.

Sample application: MAC Hub or Learning Switch

For the purposes of this tutorial, you should attempt to build a hub and/or a MAC learning switch using the above code snippets. Here is the main logic:

- A **hub** will flood any packet that arrives on a particular port of a switch out of all the other ports.
  - On packet_in to a switch,
    - Make list of all ports on the switch
    - Send packet out of all those ports, except the in_port
- A **learning switch** will keep track of where the host with each MAC address is located and accordingly send packets towards the destination and not FLOOD.
  - Create a HashMap called mac_to_port
  - On packet_in to a switch,
    - Parse packet to reveal src and dst MAC addr
    - Store in the hashmap the mapping between src_mac and the in_port
    - Lookup dst_mac in mac_to_port map to find next hop
    - If next hop is found, create flow_mod and send
    - Else, flood like hub.

The implementation of the sample application (i.e., the *tutorial_L2_forwarding* bundle) is available at **this link**. Once you integrate it with the controller and Run, you will see that mininet ping succeeds:

```
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
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable
64 bytes from 10.0.0.2: icmp_req=5 ttl=64 time=0.529 ms
64 bytes from 10.0.0.2: icmp_req=6 ttl=64 time=0.133 ms
64 bytes from 10.0.0.2: icmp_req=7 ttl=64 time=0.047 ms
```

The OSGi console shows the following sample log:

```
2013-06-13 21:00:31.844 PDT [SwitchEvent Thread] INFO o.o.c.t.i.TutorialL2Forwarding
- Installed flow Flow[match = Match[[DL_DST(00:00:00:00:00:01,null),
IN_PORT(OF|2@OF|00:00:00:00:00:00:00:01,null)]],
actions = [OUTPUT[OF|1@OF|00:00:00:00:00:00:00:01]], priority = 0, id = 0,
idleTimeout = 0, hardTimeout = 0] in node OF|00:00:00:00:00:00:00:01
```

Credits

The above tutorial was put together by Srini Seetharaman.

Quick Navigation

- OpenFlow Specs
- Bug Tracking
- Wiki
- Legal
- Log in

OpenFlow White Paper

Download the OpenFlow Whitepaper (PDF)

OpenFlow Specification

Download v1.1.0 Implemented (PDF)

Wiki Tools

Personal tools

- Log in

Navigation

- OpenFlow.org
- OpenFlow wiki
- Recent changes
- Random page
- Help

Projects

- SIGCOMM Demo
- GENI Demo

Toolbox

- What links here
- Related changes
- Special pages
- Printable version
- Permanent link