MAWLANA BHASHANI SCIENCE AND TECHNOLOY UNIVERSITY

Santosh,Tangail – 1902

Course Title : Introduction to Telecommunication System

Assignment No : 3

Submitted by,

Name :Tahmina Afroze

ID: IT-17014

Session: 2016-17

Dept. of ICT,MBSTU.

Submitted to,

NAZRUL ISLAM

Assistant Professor

Dept. of ICT,MBSTU.

Assignment Name : Controller Rest API

Theory:

1. Reactive Flow Instantiation: When a new flow comes into the switch, the OpenFlow agent software on the switch does a lookup in the flow tables. If no match for the flow is found, the switch creates an OFP packet-in packet and sends it off to the controller for instructions. Reactive mode reacts to traffic, consults the OpenFlow controller and creates a rule in the flow table based on the instruction. This behavior was tested on previous lab.

2. Proactive Flow Instantiation: Rather than reacting to a packet, an OpenFlow controller could populate the flow tables ahead of time for all traffic matches that could come into the switch. By pre-defining all of the flows and actions ahead of time in the switches flow tables, the packet-in event never occurs. The result is all packets are forwarded at line rate. Proactive OpenFlow flow tables eliminate any latency induced by consulting a controller on every flow.

3.Hybrid flow instantiation: A combination of both would allow for flexibility of reactive for particular sets a granular traffic control that while still preserving low-latency forwarding for the rest of the traffic.

Controller: REST API

1. Application program interface (API) is an interface presented by software (such as a network operating system) that provides the capability to collect information from or make a change to an underlying set of resources.

2. APIs in the context of SDN: In an open SDN model, a common interface discussed is the northbound interface (NBI). The NBI is the interface between software applications, such as operational support systems, and a centralized SDN controller. One of the common API technologies used at the northbound interface is the Representational State Transfer (REST) API. REST APIs use the HTTP/HTTPS protocol to execute common operations on resources represented by Uniform Resource Identifier (URI) strings. An application may use REST APIs to send an HTTP/HTTPS GET message via an SDN controller's IP address. That message would contain a URI string referencing the relevant network device and comprising an HTTP payload with a JSON header that has the proper parameters for a particular interface and statistic.

3. Datapath Identifier of Openflow Switch: Each OpenFlow instance on a switch is identified by a Datapath Identifier. This is a 64 bit number determined as follows according to the OpenFlow specification: “The datapath_id fleld uniquely identifies a datapath. The lower 48 bits are intended for the switch MAC address, while the top 16 bits are up to the implementer. An example use of the top 16 bits would be a VLAN ID to distinguish multiple virtual switch instances on a single physical switch.”

Using REST APIs:

REST API can be used in different ways:

1. A tool to generate REST API calls:

i) The Chrome browser, for example, has multiple plug-ins to generate REST API messages. These include Postman and the Advanced REST Client.

Ii) Firefox has the RESTClient add-on for the same functionality.

2. Command-line interface, the curl utility may also be used. Although the formatting of the REST API varies form one controller to another, the following items are common: URI string for the requested, HTTP method (e.g., GET, POST, PUT, and DELETE) and JSON/XML payload and/or parameters. The Ryu documentation provides examples illustrating how to send a valid REST API message.

RYU.APP.OFCTL_REST

(Information available at http://ryu.readthedocs.io/en/latest/app/ofctl_rest.html) ryu.app.ofctl_rest provides REST APIs for retrieving the switch stats and updating the switch stats. This application helps to debug application and get various statistics. Valid actions are:

1. Retrieve the switch stats

i) Get all switches

ii) Get the desc stats

iii) Get all flows stats

iv) Get flows stats filtered by fields

v) Get aggregate flow stats

vi) Get aggregate flow stats filtered by fields

vii) Get table stats

viii) Get table features

ix) Get ports stats

x) Get ports description

xi) Get queues stats

xii) Get queues config

xiii) Get queues description

xiv) Get groups stats

xv) Get group description stats

xvi) Get group features stats

xvii) Get meters stats

xviii) Get meter config stats

xix) Get meter description stats

xx) Get meter features stats

2. Update the switch stats

i) Add a flow entry

ii) Modify all matching flow entries

iii) Modify flow entry strictly

iv) Delete all matching flow entries

v) Delete flow entry strictly

vi) Delete all flow entries

vii) Add a group entry

viii) Modify a group entry

viii) Delete a group entry

ix) Modify the behavior of the port

x) Add a meter entry

xi) Modify a meter entry

xii) Delete a meter entry

xiii) Modify role

3. Support for experimenter multipart

i) Send a experimenter message

4. Reference: Description of Match and Actions

i) Description of Match on request messages

ii) Description of Actions on request messages

Installing curl:

1. Open the Synaptic Package Manager (Navigator ->System-> Synaptic Package Manager)

2. Setup the proxy:

   - **o** Click on settings-> Preference -> Network
   - **o** Click on manual proxy configuration
   - **o** HTT and FTP Proxy: proxy.rmit.edu.au Port: 8080

3. Search for Quick filter `curl`

4. Click on Mark for installation

5. Then click on Apply and wait until the package is installed

Question 5.1: Explain the advantages of REST API of the Controller

Ans:

The advantages of REST for development

- It is usually simple to build and adapt.

- Low use of resources.
- Process instances are created explicitly.
- With the initial URI, the client does not require routing information.
- Clients can have a generic ‘listener’ interface for notifications.

Question 5.3: Provide two real service examples for explaining the usage of reactive flow instantiation.

Ans: A few operator examples
We can create a `Flux` using the `just(T…)` and `fromIterable(Iterable<T>)` Reactor factory methods. Remember that given a `List`, `just` would *just* emit the list as one whole, single emission, while `fromIterable` will emit each element *from* the *iterable* list:

```
public class ReactorSnippets {
  private static List<String> words = Arrays.asList(
        "the",
        "quick",
        "brown",
        "fox",
        "jumped",
        "over",
        "the",
        "lazy",
        "dog"
        );

  @Test
  public void simpleCreation() {
     Flux<String> fewWords = Flux.just("Hello", "World");
     Flux<String> manyWords = Flux.fromIterable(words);

     fewWords.subscribe(System.out::println);
     System.out.println();
     manyWords.subscribe(System.out::println);
  }
}
```

Like in the corresponding RxJava examples, this prints

```
Hello
World
the
quick
brown
fox
jumped
over
the
lazy
dog
```

In order to output the individual letters in the fox sentence we'll also need `flatMap` (as we did in RxJava by Example), but in Reactor we use `fromArray` instead of `from`. We then want to filter out duplicate letters and sort them using `distinct` and `sort`. Finally, we want to output an index for each distinct letter, which can be done using `zipWith` and `range`:

```
@Test
public void findingMissingLetter() {
  Flux<String> manyLetters = Flux
        .fromIterable(words)
        .flatMap(word -> Flux.fromArray(word.split("")))
        .distinct()
        .sort()
        .zipWith(Flux.range(1, Integer.MAX_VALUE),
              (string, count) -> String.format("%2d. %s", count, string));

  manyLetters.subscribe(System.out::println);
}
```

This helps us notice the **s** is missing as expected:

```
1. a
2. b
...
18. r
19. t
20. u
...
25. z
```

One way of fixing that is to correct the original words array, but we could also manually add the "s" value to the `Flux` of letters using `concat/concatWith` and a `Mono`:

```
@Test
public void restoringMissingLetter() {
  Mono<String> missing = Mono.just("s");
  Flux<String> allLetters = Flux
        .fromIterable(words)
        .flatMap(word -> Flux.fromArray(word.split("")))
        .concatWith(missing)
        .distinct()
        .sort()
        .zipWith(Flux.range(1, Integer.MAX_VALUE),
              (string, count) -> String.format("%2d. %s", count, string));

  allLetters.subscribe(System.out::println);
}
```

This adds the missing **s** just before we filter out duplicates and sort/count the letters:

```
1. a
2. b
...
18. r
19. s
20. t
...
26. z
```

Question 5.4: Provide two real service examples for explaining the usage of proactive flow instantiation.

Ans:

Proactive flow instantiation – Rather than reacting to a packet, an OpenFlow controller could populate the flow tables ahead of time for all traffic matches that could come into the switch. Think of a typical routing table today. You have longest prefix matching that will match the most granular route to a destination prefix in a prefix tree lookup. By pre-defining all of your flows and actions ahead of time in the switches flow tables, the packet-in event never occurs. The result is all packets are forwarded at line rate, if the flow table is in TCAM, by merely doing a flow lookup in the switches flow tables. That is the same hardware that populates its forwarding tables today from "routing by rumor" in todays routing protocols and "flood and spray" layer2 learning standards. Proactive OpenFlow flow tables eliminates any latency induced by consulting a controller on every flow.

Question 5.5: what is the difference between sdn and openflow?

Ans:

Openflow is just the protocol used for communicatuin between the controller and a switch. This is considered the southbound communication. SDN is too vague and over used. There is no direct comparison. However openflow would fall under the SDN umbrella.

OpenFlow is just a protocol that manages the communication between the controlling plan and forwarding plan but I don't know where SDN takes place here. Is it the controller itself? How do I distinguish between them and when to say OpenFlow or SDN.

Software-defined networking (SDN) and OpenFlow aren't the same thing. We'll clarify the technical differences and discuss a more important distinction: SDN emphasizes applications that drive network usability and business requirements, while OpenFlow is a technology to link an SDN controller and network devices.

Many people don't understand the difference between OpenFlow and software-defined networking (SDN). This isn't surprising because the two technologies are closely related. However, they aren't interchangeable. OpenFlow is protocol that configures network switches using a process like an API. SDN is a term that describes providing programmable interfaces within a network infrastructure to enable a high degree of automation in provisioning network services. The SDN term is being abused by marketers who want to apply it to a wide range of technologies.

In fact, SDN can be explicitly defined. There are three architectural layers to an SDN network: the physical network, the SDN applications and the SDN controller. Let's look at each.