H2020 i**P** **O**ver I**c**N- the bet**T**er IP (POINT)

# Design-SDN

## POINT SDN Application Design


**List of Authors:**

George Petropoulos

# 1.Architecture

The POINT-enabled SDN application implements all the necessary functionalities to seamlessly bootstrap and operate an ICN topology over an existing SDN network, without disrupting any existing IP protocols and services. It is implemented as an Opendaylight Boron release[1] application, hence follows the controller's design and development principles. Opendaylight SDN controller utilizes OSGi-compliant Karaf container to manage the deployed artifacts, and all Opendaylight functions and protocols are implemented using Java and deployed as Karaf bundles into the core container. The POINT SDN application is a multi-module Maven[2] project, consisting of the following modules, which are deployed as Karaf bundles to the Opendaylight Karaf container (Figure 1):

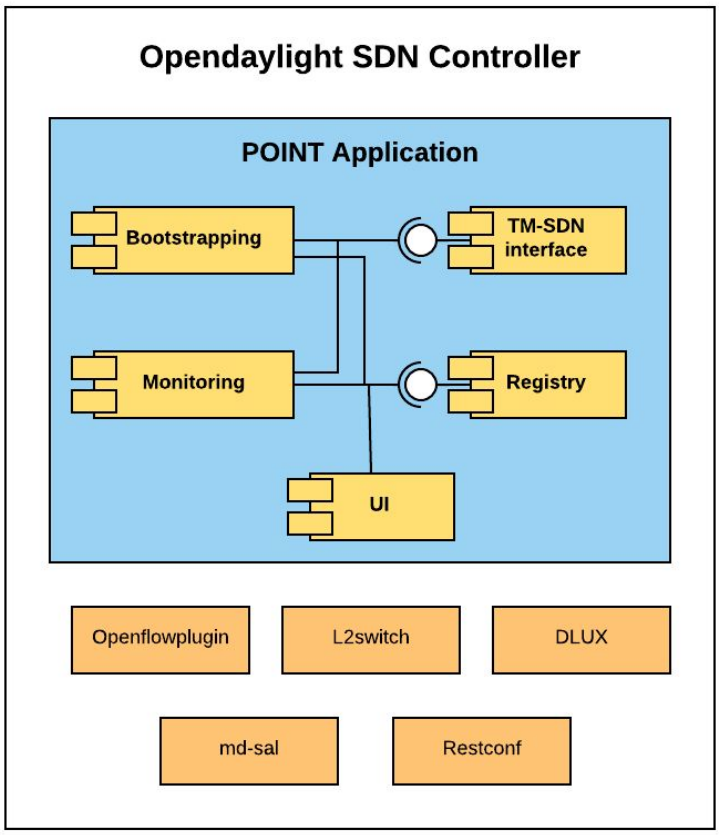


Figure 1: The POINT application component diagram

- **Registry**: It is the persistence module of the POINT application. It implements a db-like approach, in which stateful information are stored in the Opendaylight datastore in a key-value format. It also implements relevant methods to save, and retrieve stored instances.

[1] https://www.opendaylight.org/odlboron
[2] https://maven.apache.org/

- **Bootstrapping**: It is the module responsible to bootstrap the ICN topology on top of an existing SDN network. It implements appropriate methods to retrieve the topology information and topology updates, request unique resources by the ICN Resource Manager, and configure SDN switches with appropriate flow rules, following the approach of [1].
- **Monitoring**: The monitoring module of the ICN SDN application. It implements functions to monitor topology status, and specifically SDN switch port status, and inform the ICN Topology Manager for resilience purposes. In addition, the SDN switches' port statistics are also aggregated and sent to the ICN Topology Manager for traffic engineering purposes.
- **TM-SDN**: The module implementing the client side of the TM-SDN interface. This includes the protocol format and the methods to send and receive information from the ICN Topology Manager.
- **User Interface (UI)**: The user interface of the ICN application. It provides information about the SDN topology and ICN-enabled nodes, as well as means to initiate bootstrapping and monitoring functions.

The aforementioned components depend on existing Opendaylight modules and functionalities to perform their functions. Specifically, Openflowplugin is required for the Openflow rule configuration and topology management functions, L2switch is essential for tracking the ICN hosts, MD-SAL and Restconf are useful for core Opendaylight functionalities, such as datastore persistence and application's REST API. In addition the ICN UI extends Opendaylight's DLUX module, and creates a new menu and tab in its user interface.

The functionalities and specific methods per component are presented in details in later sections.

# 2.Registry

Registry is the persistence module of the ICN application, implementing the model, Java APIs and REST APIs to insert, update and query stateful information from the Opendaylight datastore.

## 2.1.Model

The model of the stateful information is defined using YANG file and structure. The registry YANG file is available at https://github.com/point-h2020/point-cycle1/blob/release-3.0.0/sdn/registry/model/src/main/yang/registry.yang (TODO update with correct release link) and implements 3 containers:

Node Registry:

- Matches the Openflow node name with the given ICN Node identifier

- Key: nodeName, Value: nodeIdentifier

Link Registry:

- Matches the Openflow link name with the given ICN Link Identifier
- Key: linkName, Value: linkIdentifier

Node Connector Registry:

- Stores information about the links and its 2 edge nodes
- Key: nodeConnectorName, Value: srcNode, dstNod
-

## 2.2.Java APIs

The complete documentation of the Java classes, and public and private methods is available at the JavaDoc documentation at (TODO add link). The main idea is that each registry requires a utility class which implements the following essential persistence functions:

- `public void initializeDataTree()`: The method which initializes the registry data tree.
- `public void writeToNodeRegistry(input)`: The method which writes to the registry a specific entry.
- `public void deleteFromNodeRegistry(key)`: The method which deletes from the registry a specific entry for a given key.
- `public NodeRegistryEntry readFromNodeRegistry(key)`: The method which reads from the registry an entry with a specific key.

## 2.3.REST APIs

The Registry module exposes the following REST APIs, based on the base URI (http://localhost:8181/restconf), which can be accessed with HTTP GET requests.

- `/operational/registry:node-registry`
  The method which fetches all configured node identifiers for all SDN switches and end-hosts.
- `/operational/registry:node-registry/node-registry-entry/{entry-id}`
  The method which fetches the configured node identifier for a specific SDN switch/end-host with the given `{entry-id}`.
- `/operational/registry:link-registry`
  The method which fetches all configured link identifiers for all SDN switches and end-hosts.
- `/operational/registry:link-registry/link-registry-entry/{entry-id}`
  The method which fetches the configured link identifier for a specific SDN switch/end-host link with the given `{entry-id}`.

- `/operational/registry:node-connector-registry`
  The method which fetches all configured node and link identifiers for all SDN switches and end-hosts.
- `/operational/registry:node-connector-registry/node-connector-registry-entry/{entry-id}`
  The method which fetches the configured node and link identifier for a specific SDN switch/end-host link with the given `{entry-id}`.

# 3.Bootstrapping

The Bootstrapping module is responsible to bootstrap the ICN topology on top of an existing SDN network. It implements appropriate methods to retrieve the topology information and topology updates, request unique resources by the ICN Resource Manager, and configure SDN switches with appropriate flow rules. In the following subsections, the sequence diagram of the ICN topology bootstrapping process is presented, as well as the key Java APIs of the implemented module and HTTP REST APIs which can be used by external entities or the SDN controller administrator.

## 3.1.Sequence Diagram

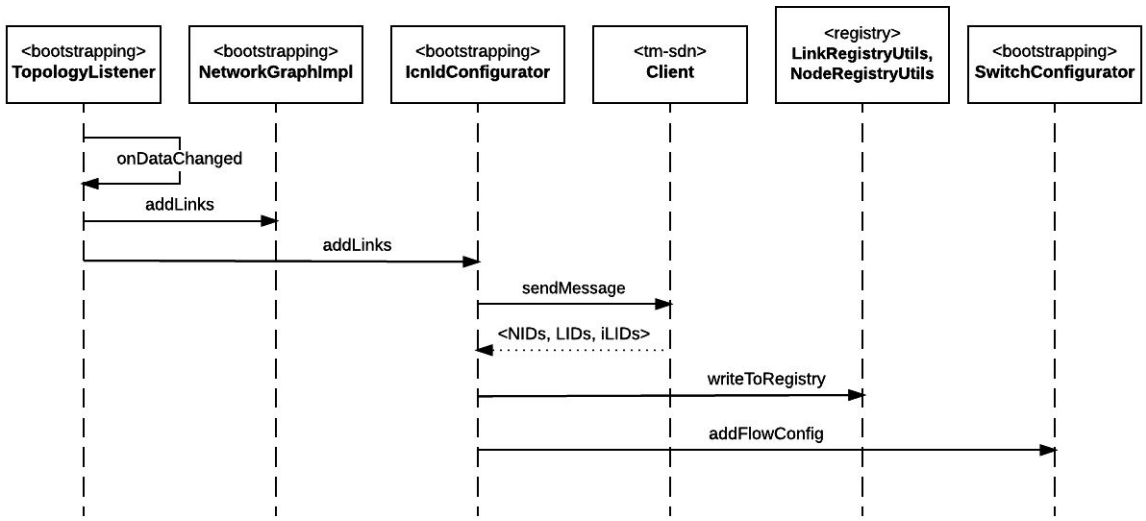


Figure 2: Bootstrapping sequence diagram for new links

The core ICN over SDN bootstrapping protocol has been specified in Deliverable 2.3 [TODO add D2.3 reference]. The mechanism to bootstrap SDN switches with ICN functionality requires the SDN controller to monitor the underlying topology. In case new nodes or links are added, modified or removed, it must have a topology monitoring function which compares the topology's previous state and identifies the network resources to be configured accordingly. This

is achieved by `TopologyListener` class (Figure 2), which maintains an internal listener for topology changes into the Opendaylight's datastore, as well as the topology state. When data in the datastore are changed, and in our example new links are added to the topology, then it (a) first triggers the `addLinks()` method of `NetworkGraphImpl` class, which maintains the topology graph and (b) the `addLinks()` method of the `IcnIdConfigurator` class, which handles the assignment of ICN resources. `IcnIdConfigurator` implements private methods which maintain the list of assigned resources, and in the case that the new link has not been assigned a unique ICN identifier in the past, it then calls the `sendMessage()` method of `Client` class of the tm-sdn module. This method sends a bootstrapping ResourceRequest message to the resource management function of the Topology Manager, and receives the ResourceResponse with the assigned unique Node IDs, Link IDs, and internal LIDs [D2.3 reference]. These message may contain more than one ResourceRequest messages for the respective number of network resources.
The assigned resources are returned to the `IcnIdConfigurator` , which then writes them to the Node and Link Registries using the `writeToRegistry()` functions, and prepares the arbitrary bitmask rules. This includes private methods to translate ICN-specific identifiers (Link IDs) to Openflow-specific matches and actions, based on [1]. Then it calls the `addFlowConfig()` function of `SwitchConfigurator` class, which inserts the created flow rules to the CONFIG datastore of the SDN controller, and eventually to the respective SDN switches.

## 3.2.Java APIs

The complete documentation of the Java classes, public and private methods, and their input parameters, is available at the JavaDoc documentation at (TODO add link). Below we present a list of representative classes and public methods which are essential to understand the module's functionality.

- `BootstrappingServiceImpl` : The bootstrapping service class. It implements all the REST APIs for the bootstrapping module.
    - `public Future<RpcResult<Void>> configureTm(ConfigureTmInput input)` : The method which configures the Topology Manager parameters.
    - `public Future<RpcResult<Void>> activateApplication(ActivateApplicationInput input)` : The method which activates the bootstrapping application.
    - `public Future<RpcResult<Void>> configureSwitch(ConfigureSwitchInput input)` : The method which configures a switch manually, without any resource management function.
    - `public Future<RpcResult<NodeLinkInformationOutput>> nodeLinkInformation (NodeLinkInformationInput input)` : The method which requests the node id and link id of a certain link. It either finds it in

the registry, or requests it from the resource manager directly and updates the registry itself. It is used by the User Interface module.

- `TopologyListener` : The class which listens for topology changes.
  - `public void onDataChanged(AsyncDataChangeEvent<InstanceIdentifier<?>, DataObject> dataChangeEvent)` : The method which monitors changes in data store for topology. In case of new nodes and links, then calls the respective functions to add them to the SDN and ICN topology, otherwise the ones removing them from those.
- `NetworkGraphImpl` : The class which maintains the SDN network graph.
  - `public synchronized void addLinks(List<Link> links)` : The method which adds links to the graph.
  - `public synchronized void removeLinks(List<Link> links)` : The method which removes links from the graph.
  - `public List<Link> getShortestPath(String srcNodeId, String dstNodeId)` : The method which returns the shortest path based on Dijkstra algorithm for the given source and destination nodes.
- `IcnIdConfigurator` : The ICN configurator class. It includes all the required methods to map Opendaylight/SDN information to ICN ones and maintains the ICN topology.
  - `public void addLinks(List<Link> links)` : The method which adds links to the ICN topology. If bootstrapping application is active, then for each link in the list, it calls the relevant method. Otherwise it adds them to the to-be-configured list.
  - `public void removeLinks(List<Link> links)` : The method which removes links from the ICN topology.
- `SwitchConfigurator` : The class which includes all the methods to add and delete a flow from an SDN switch.
  - `public void addFlowConfig(String edge_switch, String edgeNodeConnector, String srcAddress, String dstAddress)` : The method which installs a flow to the switch, through the configuration datastore of the controller.
  - `public void deleteFlowConfig(String edge_switch, String edgeNodeConnector)` : The method which deletes a flow from the switch, through the configuration datastore of the controller.

## 3.3.REST APIs

The Bootstrapping module exposes the following REST APIs, based on the base URI (http://localhost:8181/restconf), which can be accessed with HTTP POST requests.

- `/operations/bootstrapping:configureTm`
  Method to configure the TM parameters. Example JSON body as indicated below.

```
{ input: { tmIp: "192.168.1.2", tmPort: 12345, tmOpenflowId:
"host:00:00:00:00:00:01", tmAttachedSwitchId: "openflow:1",
tmNodeId: "00000001", tmLid: 1, tmInternalLid: 0 } }
```

- `/operations/bootstrapping:activateApplication`
  Method to activate/deactivate the automatic bootstrapping application. Example JSON body as indicated below.

  ```
  { input: { status: true } }
  ```

- `/operations/bootstrapping:configureSwitch`
  Method to configure an SDN switch with ABM rules manually. Example JSON body as indicated below.

  ```
  { input: { switchId: "openflow:1", portId: "openflow:1:2",
  linkId: 10 } }
  ```

- `/operations/bootstrapping:nodeLinkInformation`
  Method to request and retrieve the configured ICN parameters for a specific link between two nodes. Example JSON body as indicated below.

  ```
  { input: { srcNode: "openflow:1", dstNode: "openflow:2" } }
  ```

# 4.Monitoring

The Monitoring module implements functions to monitor topology status, and specifically SDN switch port status, and inform the ICN Topology Manager for resilience purposes. In addition, the SDN switches' port statistics are also aggregated, based on Openflow statistics, and sent to the ICN Topology Manager for traffic engineering purposes. These measurements include received and transmitted bytes and packets per interface.

## 4.1.Sequence Diagram

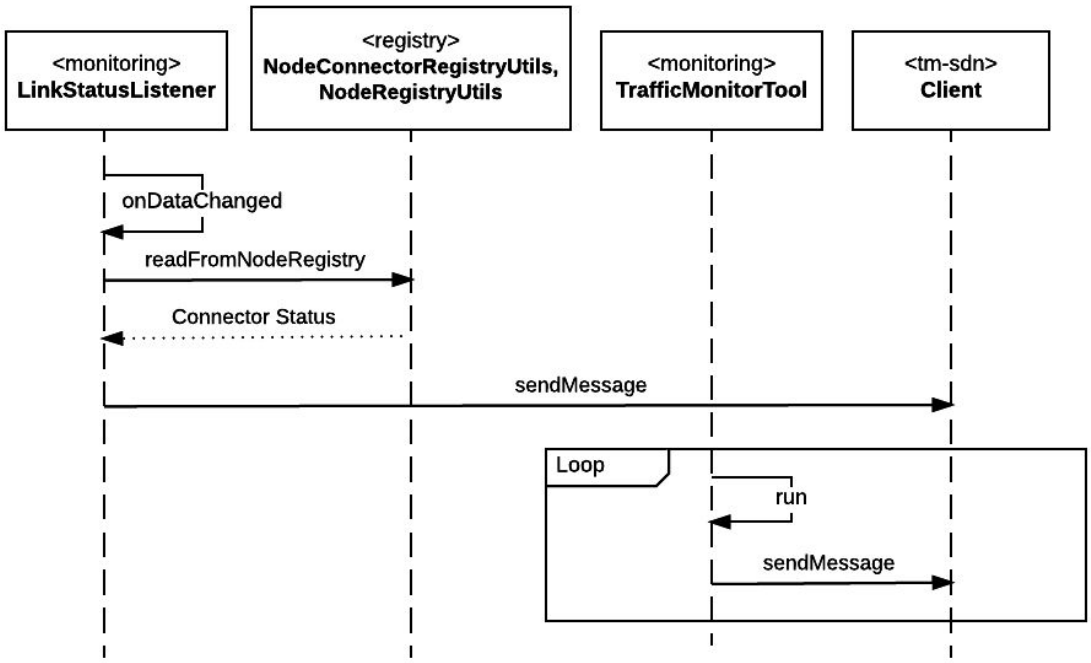


Figure 3: Monitoring sequence diagram

The TM-SDN interface has been specified in Deliverable 2.3 [TODO add D2.3 reference]. Specific messages of this interface were presented in previous section, but will be further specified in the following one. This section focuses on the monitoring messages, and specifically the ones needed for link state monitoring and traffic monitoring functionalities (Figure 3).

For the link state monitoring function, a relevant class, the `LinkStatusListener` has been implemented, which implements a listener to the Opendaylight data store, maintains the current topology state and in case a link is added or removed, it then takes appropriate measures to inform the ICN topology Manager. First it queries the NodeConnector and Node registries to identify whether the links or nodes are part of the ICN topology, and in case they are, it then also checks the connectors' status and whether it has been updated. In this case, it then calls the `sendMessage()` method of the tm-sdn module Client class to send a LinkStateMonitoring message to the ICN Topology Manager. It is assumed that ICN Topology Manager simply updates its topology and implements resilience functionality to resolve any potential issues.

For the traffic monitoring function, an implemented timer task, `TrafficMonitorTool`, periodically reads the Opendaylight datastore, aggregates the links' statistics, in terms of received and transmitted bytes and packets, and sends them to the Topology Manager, using the `sendMessage()` method of the tm-sdn module `Client` class.

## 4.2.Java APIs

The complete documentation of the Java classes, public and private methods, and their input parameters, is available at the JavaDoc documentation at (TODO add link). Below we present a list of representative classes and public methods which are essential to understand the module's functionality.

- `MonitoringServiceImpl` : The class which implements the REST API for monitoring module.
  - `public Future<RpcResult<Void>> init(InitInput input)` : The method which initializes the monitoring functions through the REST API.
- `LinkStatusListener` : The class which implements link status listener.
  - `public void onDataChanged(AsyncDataChangeEvent<InstanceIdentifier<?>, DataObject> change)` : The method which monitors changes in the datastore topology, and sends ADD or RMV link state notifications to the topology manager.
- `TrafficMonitorTool` : The class which implements the traffic monitoring timer task.
  - `public void run()` : The method which runs the monitoring task. It monitors the node connector statistics, specifically packets and bytes received and transmitted and sends them to the TM.

## 4.3.REST APIs

The Monitoring module exposes the following REST API, based on the base URI (http://localhost:8181/restconf), which can be accessed with HTTP POST request.

- `/operations/monitoring:init`

  Method which activates/deactivates the link state and traffic monitoring functionalities. Example JSON body as indicated below.

  ```
  { input: { trafficmonitor-enabled: true, trafficmonitor-period: 30, linkmonitor-enabled: true } }
  ```

# 5.TM-SDN

The TM-SDN interface has been specified in Deliverable 2.3 [TODO add D2.3 reference]. It includes messages for bootstrapping and monitoring purposes. In the POINT SDN application, the client side of the interface is implemented, using the Netty asynchronous NIO client server

framework[3]. The protocol format is specified using Google protocol buffers[4] and is specified in the messages.proto file, available at https://github.com/point-h2020/point-cycle1/blob/release-3.0.0/sdn/tm-sdn/messages.proto (TODO add correct link).
The key class of the tm-sdn module is the `Client` class, which implements the following method:

- public static TmSdnMessages.TmSdnMessage.ResourceOfferMessage sendMessage(TmSdnMessages.TmSdnMessage message): The method which creates the bootstrap to connect to the server. It initializes the pipeline with all the required parameters and functions for encoding, decoding, etc. It then sends the message to the server. If the message is a Resource Request, then it also returns the received Resource Offer.

# 6.User Interface

[TODO add figure]
The user interface module is implemented in Angular JS framework[5] and extends the Opendaylight's user interface module, DLUX. It consists of 2 sub-modules, (a) icnsdnui-bundle, which specifies relevant Angular-specific parameters for the module, and (b) icnsdnui-module, which includes all the javascript files which implement services and appropriate functions, as well as include the static resources, including html files, to be shown in the web application.
A brief description of the Javascript files is presented below:

- icnsdnui.module.js: Includes all the required Angular dependencies and settings to render the ICN-SDN user interface page.
- icnsdnui.controller.js: Includes all the implemented methods, which are called on button user interface input (button clicks in this case).
- icnsdnui.services.js: Implements all the relevant functions which interact with the exposed REST APIs, including the ones for bootstrapping initialization, monitoring initialization and ICN topology update with node and link identifiers.
- icnsdnui.directives.js: Implements the ICN topology graph rendering using the vis library[6]

# 7.ABM Flow Rule Configuration

Besides the aforementioned functionalities which implement an automated way to bootstrap and manage an ICN topology over an SDN network, the SDN controller administrator can still configure the SDN switches using the default Opendaylight API of the following URI: http://localhost:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:X/table/Y/flow/

[3] http://netty.io/
[4] https://developers.google.com/protocol-buffers/
[5] https://angularjs.org/
[6] http://visjs.org/

Z. In this URI, X indicates the identifier of the Openflow switch, Y the table identifier and Z the flow identifier to be configured.
Besides IPv4/6-specific flow rules, since Opendaylight Boron SR1 version, the administrator may also set IPv6 arbitrary bitmask masks to the configured Openflow rules, which enables the configuration of ICN-aware flow rules, based on the mechanism of [1]. An example of such rule is presented below:

```
{
  "flow": {
    "barrier": "false",
    "cookie": "10",
    "cookie_mask": "10",
    "flow-name": "example",
    "id": "2",
    "priority": "1030",
    "table_id": "0",
    "instructions": {
      "instruction": {
        "apply-actions": {
          "action": {
            "order": "0",
            "output-action": { "output-node-connector": "3" }
          }
        },
        "order": "0"
      }
    },
    "match": {
      "ethernet-match": {
        "ethernet-type": { "type": "34525" },
        "ethernet-destination": { "address": "00:00:00:00:00:00" }
      },
      "ipv6-source-address-no-mask":
"0000:4000:0000:0000:0000:0000:0000:0000",
      "ipv6-source-arbitrary-bitmask":
"0000:4000:0000:0000:0000:0000:0000:0000",
      "ipv6-destination-address-no-mask":
"0000:0000:0000:0000:0000:0000:0000:0000",
      "ipv6-destination-arbitrary-bitmask":
"0000:0000:0000:0000:0000:0000:0000:0000"
    }
  }
}
```

The SDN controller administrator would have to send an HTTP PUT request to http://localhost:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:1/table/0/flow/2, using an appropriate application, e.g. POSTMAN, and including headers for Basic authorization (admin/admin), Content-type and Accept as application/json, and sample body as indicated above.
In the body, you could replace the `id`, `output-node-connector` , and `ipv6-*-address-*` fields with the generated parameters.

An example JSON body for ABM flow rules with more output ports, then use the following format. Notice that action is now an array of actions and relevant output-node-connectors.

```
{
  "flow": {
    "barrier": "false",
    "cookie": "10",
    "cookie_mask": "10",
    "flow-name": "example",
    "id": "2",
    "priority": "1030",
    "table_id": "0",
    "instructions": {
      "instruction": {
        "apply-actions": {
          "action": [
            {
              "order": "0",
              "output-action": { "output-node-connector": "1" }
            },
            {
              "order": "1",
              "output-action": { "output-node-connector": "2" }
            }
          ]
        },
        "order": "0"
      }
    },
    "match": {
      "ethernet-match": {
        "ethernet-type": { "type": "34525" },
        "ethernet-destination": { "address": "00:00:00:00:00:00" }
      },
      "ipv6-source-address-no-mask":
"0000:4000:0000:0000:0000:0000:0000:8000",
      "ipv6-source-arbitrary-bitmask":
```

```
"0000:4000:0000:0000:0000:0000:0000:8000",
      "ipv6-destination-address-no-mask":
"0000:0000:0000:0000:0000:0000:0000:0000",
      "ipv6-destination-arbitrary-bitmask":
"0000:0000:0000:0000:0000:0000:0000:0000"
    }
  }
}
```

Finally, an example flow rule with an output port, as well as redirecting to the next table, is also presented below:

```
{
  "flow": {
    "barrier": "false",
    "cookie": "10",
    "cookie_mask": "10",
    "flow-name": "example",
    "id": "2",
    "priority": "1030",
    "table_id": "0",
  "instructions": {
    "instruction": [
      {
        "order": "0",
        "apply-actions": {
          "action": {
            "order": "0",
            "output-action": { "output-node-connector": "1" }
          }
        }
      },
      {
        "order": "1",
        "go-to-table": { "table_id": "1" }
      }
    ]
  },
    "match": {
      "ethernet-match": {
        "ethernet-type": { "type": "34525" },
        "ethernet-destination": { "address": "00:00:00:00:00:00" }
      },
      "ipv6-source-address-no-mask":
```

```
"0000:4000:0000:0000:0000:0000:0000:0000",
      "ipv6-source-arbitrary-bitmask":
"0000:4000:0000:0000:0000:0000:0000:0000",
      "ipv6-destination-address-no-mask":
"0000:0000:0000:0000:0000:0000:0000:0000",
      "ipv6-destination-arbitrary-bitmask":
"0000:0000:0000:0000:0000:0000:0000:0000"
    }
  }
}
```

[1] Reed, Martin J., et al. "Stateless multicast switching in software defined networks." *arXiv preprint arXiv:1511.06069* (2015).