IEEE NFV-SDN

IEEE Conference on Network Function Virtualization and Software Defined Networks
12-14 November 2019 // Dallas, Texas, USA

IEEE NFV-SDN 2019 Tutorial: Hands-on Tutorial: Controlling and Monitoring Network Equipment

Ricard Vilalta (CTTC/CERCA)

Short course Materials

- Please go to github repository to get latest version:
  - https://github.com/rvilalta/OFC_SC472
  - For a perfect hands-on experience, a VirtualBox VM image is needed. Please download the course VM from the link below and make sure the VM is installed and loads/starts up on your PC before travelling to OFC:
  - http://tiny.cc/NetControl2020
  - Login: osboxes
  - Password: osboxes.org
  - Login: root
  - Password: osboxes.org

Agenda

- 9h – 9h05 Motivation
- 9h05-9h20 YANG Data Modelling Language
  - Modelling a network
  - Using pyang and its plugins
  - Pyangbind to write code in python
- 9h20-10h Netconf
  - Understanding Netconf protocol
  - Use Confd as a Netconf Server
  - Create a Netconf Client
  - Create a Netconf Server with basic commands
- 10h-10h30 RESTconf
  - Understanding RESTconf protocol
  - Generate topology/connection OpenAPI
  - Generate connection Server Stub
- 10h30-11h Coffee break

We are ready for Control and monitoring of Networks II

- 11h-11h15 Using ONOS with RESTconf
  - Introduction to ONOS northbound REST API, Mininet config
  - ONOS client (topology & flows)
- 11h15-11h30 ONF Transport API
  - Understanding TAPI model
  - TAPI Topology client
- 11h30 – 12h gRPC
  - Understanding gRPC and Protocol Buffers
  - Usage of protobufs
  - Create a gRPC client/server
  - gRPC streams
- 12h-12h30 OpenConfig
  - Data Model Principles
  - RPCs and gNMI
- Conclusion

TRANSPORT SDN - MOTIVATION

What we see in the market ?

A New Kind of Business Customer

- **Private and public cloud**
- **Elastic compute and storage**

Hypergrowth - Mobile, Video, Cloud…

A New Kind of Consumer

- **Living in the cloud**
- **Streaming Downloads**
- **Driving new network loads**

A New Kind of Servi
Provider

SaaS, IaaS, PaaS
Elastic compute & sto e
Multi-tenant

Elastic Compute & Storage

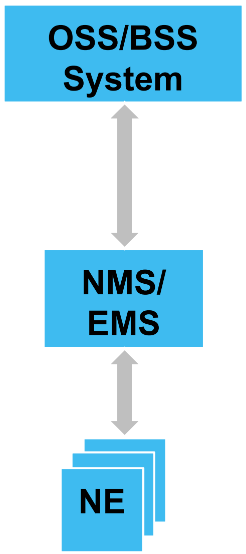
Requires an Elastic Network

A New Network is Required

- **Bandwidth-on-demand to match the compute/storage on-demand technology**
- **Multi-tenant**
- **Higher utilization, greater efficiency**
- **Scalable and resilient**
- **Faster service deployment**

Why is SDN different from traditional Architectures?

Traditional Architecture



The difference is NOT:

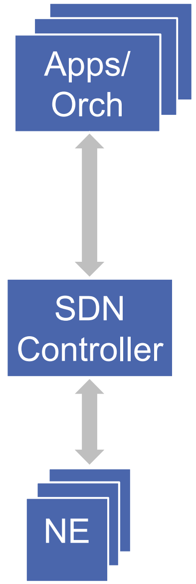
- Standardized Management Interfaces
- Standardized Architecture
- Partly not the Open Interfaces
- Partly not even the use cases

The difference is:

A new way of thinking

- Application focused
- Application takes control over the service
- Open Source based development
- Simplification through Abstraction & Virtualization

SDN Architecture



Why do we need SDN in Transport?

Principles of SDN

Programmability:
- Programmable interfaces
- Applications focused architecture
- Abstraction & Virtualization
- Multi-Tenant capabilities

Openness:
- Open Stanadsrds & Interfaces
- Open Source SW

Integration focused:
- Multi-layer
- Multi-vendor

What it Enables in Transport Network

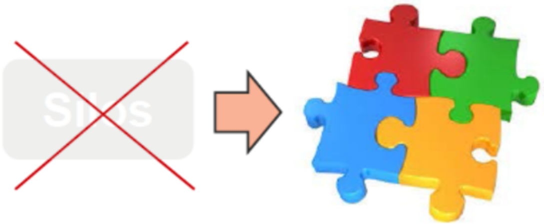
Innovation:
- Opens doors for new service models
- Service differentiation through new application

Simplified Architectures:
- Integrated E2E / Multi-layer service creation
- Automatic reaction on errors or any changes

Financial Benefits:
- Opex: efficient service setup
- Capex: fast ROI / hardware utilization
- New revenue opportunities

Keys to success

Break Silos

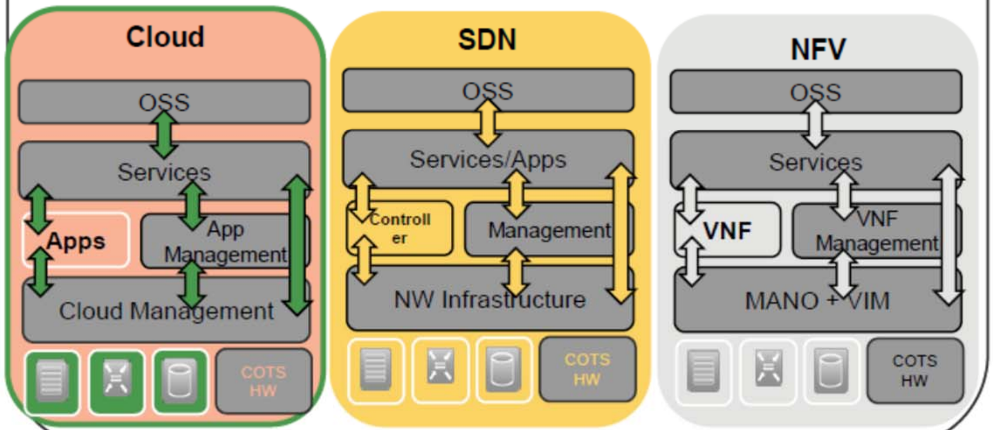


Metrics of Success

1) Multi-vendor Interoperability
2) Industry Adoption

Avoid moving from silos of HW to silos of SW

Avoid new silos of Management



YANG

Unified Information and Data Modeling (1)

- **Some deployments of optical transport networks are purely managed**, without a dedicated control plane.
  - The need of better management frameworks and protocols has long been established.
- From the perspective of an operator, the configuration of a control plane (e.g., definition of routing policies, configuration of routing peers) remains a management task.
- There is a need to have better configuration management, a clear separation of configuration and operational data, while enabling high level constructs more adapted to operators' workflows supporting network-wide transactions.
- While such frameworks are initially focused on management tasks, it is reasonable to *adopt them holistically, covering most aspects related to device and network control*
  - Increase of information and data modelling bound to the rise of network programmability.

- In general, a device (or system)
  - Information Model macroscopically describes the device capabilities, in terms of operations and configurable parameters, using high level abstractions without specific details on aspects such as a particular syntax or encoding.
  - Data Model determines the structure, syntax and semantics of the data that is externally visible.

Unified Information and Data Modeling (2) : Goals

- **Unified information and data modeling language** to describe a device capabilities, attributes, operations to be performed on a device or system and notifications
  - A common language with associated tools
  - Enabling complex models with complex semantics, flexible, supporting extensions and augmentations
  - A “best-practice” and guidelines for model authors
- **An architecture for remote configuration and control**
  - Client / Server, supporting multiple clients, access lists, transactional semantics, roll-back
- An **associated transport protocol** provides primitives to view and manipulate the data, providing a suitable encoding as defined by the data-model.
  - Flexible, efficient, allowin
  - *Ideally, data models should be protocol independent*
- **Standard, agreed upon models for devices**
  - Huge activity area
  - Hard to reach consensus (controversial aspects)
  - Some models do exist. Most stable ones cover mature aspects (interface configuration, RIB, BGP routing)

The YANG Language I

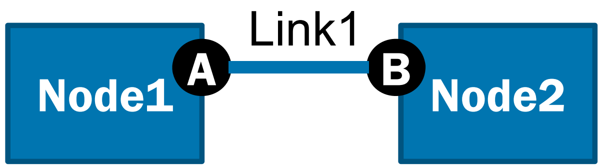
- YANG is a data modeling language, initially conceived to model configuration and state data for network devices
  - Models define the device configurations & notifications, capture semantic details and are easy to understand.
  - Significant adoption as data modelling language, across frameworks and Open Source projects
  - Ongoing notable effort across the SDOs to model constructs (e.g. topologies, protocols), including optical devices, such as transceivers, ROADMs,… Literally hundreds of emerging standards across SDOs.
- A Yang model includes a header, imports and include statements, type definitions, configurations and operational data declarations as well as actions (RPC) and notifications.
  - The language is expressive enough to:
    - Structure data into data trees within the so called datastores, by means of encapsulation of containers and lists, and to define constrained data types (e.g. following a given textual pattern).
    - Condition the presence of specific data to the support of optional features.
    - Allow the refinement of models by extending and constraining existing models (by inheritance/augmentation), resulting in a hierarchy of models.
    - Define configuration and/or state data.

The YANG Language II

- YANG has become the data modeling language of choice for multiple network control and management aspects
  - Covering devices, networks, and services, even pre-existing protocols.
  - Due in part, for its features and flexibility and the availability of tools.
  - Examples:
    - An SDN controller may export the underlying optical topology in a format that is unambiguously determined by its associated YANG schema,
    - A high-level service may be described so that an SDN controller is responsible for mediating and associating high-level service operations to per-device configuration operations.

A YANG model for network topology

- A network consists of:
  - Nodes and Links
- A node consists of:
  - node-id and ports
- A port consists of:
  - port-id and type of port
- A link consists of:
  - link-id, reference to source node, reference to target node, reference to source port and reference to target port.



topology.yang

```
module topology {

  namespace "urn:topology";
  prefix "topology";
  organization
    "CTTC";
  contact
    "ricard.vilalta@cttc.es";
  description
    "Basic example of network
topology";

  revision "2018-08-24" {
    description "Basic example
of network topology";
    reference "";
  }

typedef layer-protocol-name {
    type enumeration {
        enum "ETH";
        enum "OPTICAL";
    }
  }

...
```

```
...

grouping port {
    leaf port-id {
      type string;
    }
    leaf layer-protocol-name {
      type layer-protocol-name;
    }

  }

  grouping node {
    leaf node-id {
      type string;
    }
    list port {
      key "port-id";
      uses port;
    }
  }
```

```
...

grouping link {
    leaf link-id {
      type string;
    }
    leaf source-node {
      type leafref {
        path "/topology/node/node-id";
      }
    }
    leaf target-node {
      type leafref {
        path "/topology/node/node-id";
      }
    }
    leaf source-port {
      type leafref {
        path "/topology/node/port/port-id";
      }
    }
    leaf target-port {
      type leafref {
        path "/topology/node/port/port-id";
      }
    }
  }
```

```
...

grouping topology {
    list node {
      key "node-id";
      uses node;
    }
    list link {
      key "link-id";
      uses link;
    }
  }

  /**
   * Container/lists
   */
  container topology {
    uses topology;
  }
```

[Tool] pyang

- An extensible YANG validator and converter in python https://github.com/mbj4668/pyang
  - Check correctness, to transform YANG modules into other formats, and to generate code from the modules

```
# pyang -f tree topology.yang

module: topology
  +--rw topology
    +--rw node* [node-id]
    | +--rw node-id    string
    | +--rw port* [port-id]
    |    +--rw port-id              string
    |    +--rw layer-protocol-name?   layer-protocol-name
    +--rw link* [link-id]
      +--rw link-id       string
      +--rw source-node?   -> /topology/node/node-id
      +--rw target-node?   -> /topology/node/node-id
      +--rw source-port?   -> /topology/node/port/port-id
      +--rw target-port?   -> /topology/node/port/port-id
```

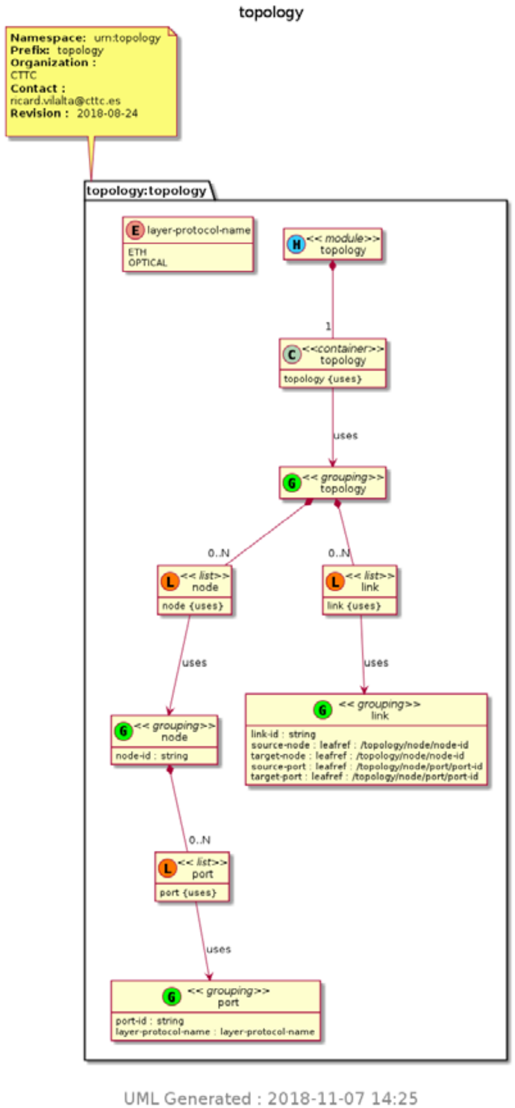
```
# pyang -f sample-xml-skeleton –sample-xml-skeleton-annotations
topology.yang

<?xml version='1.0' encoding='UTF-8'?>
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<topology xmlns="urn:topology">
 <node>
  <!-- # entries: 0.. -->
  <node-id><!-- type: string --></node-id>
  <port>
   <!-- # entries: 0.. -->
   <port-id><!-- type: string --></port-id>
   <layer-protocol-name><!-- type: layer-protocol-name --></layer-protocol-name>
  </port>
 </node>
 <link>
  <!-- # entries: 0.. -->
  <link-id><!-- type: string --></link-id>
  <source-node><!-- type: leafref --></source-node>
  <target-node><!-- type: leafref --></target-node>
  <source-port><!-- type: leafref --></source-port>
  <target-port><!-- type: leafref --></target-port>
 </link>
</topology>
</data>
```

UML diagram

- PlantUML is an opensource tool to create UML diagrams
- Pyang is able to create an UML diagram of the desired yang module
- Only a certain version of PlantUML is compatible with provided output:

http://sourceforge.net/projects/plantuml/files/plantuml.7997.jar/download

```
# pyang -f uml topology.yang -o topology.uml
# java -jar plantuml.jar topology.uml
```



From YANG to code: pyangbind

- PyangBind is a plugin for Pyang that generates a Python class hierarchy from a YANG data model. The resulting classes can be directly interacted with in Python. Particularly, PyangBind will allow you to:
  - Create new data instances - through setting values in the Python class hierarchy.
  - Load data instances from external sources - taking input data from an external source and allowing it to be addressed through the Python classes.
  - Serialise populated objects into formats that can be stored, or sent to another system (e.g., a network element).
- Please install from sources. It includes new serialization to XML.

```
$ export PYBINDPLUGIN=`/usr/bin/env python -c \
'import pyangbind; import os; print ("{}/plugin".format(os.path.dirname(pyangbind.__file__)))'`
$ echo $PYBINDPLUGIN
$ pyang -f pybind topology.yang –plugindir $PYBINDPLUGIN -o binding_topology.py
```

Source: https://github.com/robshakir/pyangbind

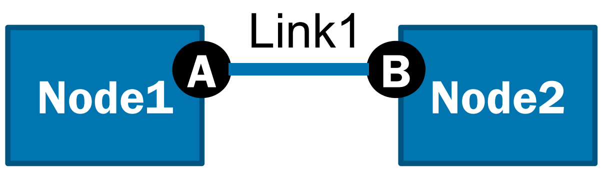
How to Create a topology

- Create an XML and a JSON that is compliant with topology.yang
- Use the proposed simple network topology
- Import the generated pyangbind bindings
- Use pyangbind serializers

Basic pyangbind tutorial:
https://github.com/robshakir/pyangbind#getting-started

```
$ python3 topology.py
```

```
from binding_topology import topology
from pyangbind.lib.serialise import pybindIETFXMLEncoder
import pyangbind.lib.pybindJSON as pybindJSON

topo = topology()
node1=topo.topology.node.add("node1")
node1.port.add("node1portA")
node2=topo.topology.node.add("node2")
node2.port.add("node2portA")
link=topo.topology.link.add("link1")
link.source_node = "node1"
link.target_node = "node2"
link.source_port = "node1portA"
link.target_port = "node2portA"

print(pybindIETFXMLEncoder.serialise(topo))
print(pybindJSON.dumps(topo))
```



Topology XML

```xml
<topology xmlns="urn:topology">
 <topology>
  <node>
   <node-id>node1</node-id>
   <port>
    <port-id>node1portA</port-id>
   </port>
  </node>
  <node>
   <node-id>node2</node-id>
   <port>
    <port-id>node2portA</port-id>
   </port>
  </node>
  <link>
   <target-node>node2</target-node>
   <source-port>node1portA</source-port>
   <link-id>link1</link-id>
   <source-node>node1</source-node>
   <target-port>node2portA</target-port>
  </link>
 </topology>
</topology>
```

Topology JSON


```
{
  "topology": {
    "node": {
      "node1": {
        "node-id": "node1",
        "port": {
          "node1portA": {
            "port-id": "node1portA"
          }
        }
      },
      "node2": {
        "node-id": "node2",
        "port": {
          "node2portA": {
            "port-id": "node2portA"
          }
        }
      }
    },
    "link": {
      "link1": {
        "link-id": "link1",
        "source-port": "node1portA",
        "target-node": "node2",
        "target-port": "node2portA",
        "source-node": "node1"
      }
    }
  }
}
```

Exercise: Create a connection data model

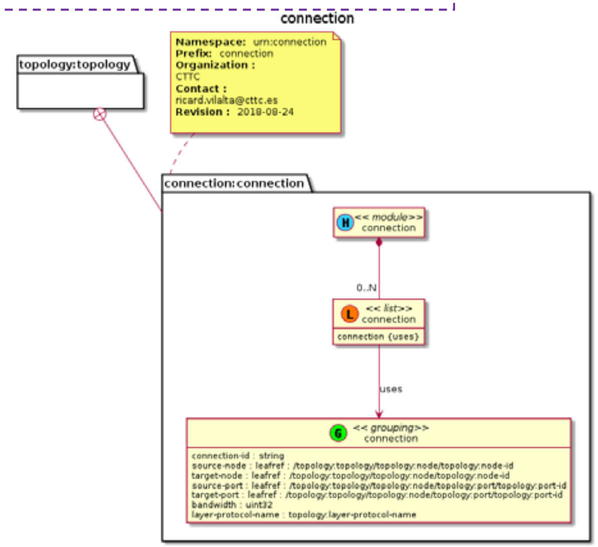
- Create a YANG data model for connection.
  - Connection consists of:
    - connection-id (string)
    - source-node, source-port, destination-node, destination-port (leaf-ref)
    - Bandwith (uint32)
    - layer-protocol-name (from topology.yang)

- Validate model with pyang
- Create pyangbind bindings
- Create xml using bindings

- Time: 10 min

Solution: connection.yang

```
module connection {
  namespace "urn:connection";
  prefix "connection";
  import topology {
    prefix "topology";
  }
  organization
    "CTTC";
  contact
    "ricard.vilalta@cttc.es";
  description
    "Basic example of network topology";
  revision "2018-08-24" {
    description "Basic example of network
topology";
    reference "";
  }
...
```



UML Generated : 2018-11-08 09:13

```
...
grouping connection {
    leaf connection-id {
      type string;
    }
    leaf source-node {
      type leafref {
        path "/topology:topology/topology:node/topology:node-id";
      }
    }
    leaf target-node {
      type leafref {
        path "/topology:topology/topology:node/topology:node-id";
      }
    }
    leaf source-port {
      type leafref {
        path "/topology:topology/topology:node/topology:port/topology:port-id";
      }
    }
    leaf target-port {
      type leafref {
        path "/topology:topology/topology:node/topology:port/topology:port-id";
      }
    }
    leaf bandwidth {
      type uint32;
    }
    leaf layer-protocol-name {
      type topology:layer-protocol-name;
    }
}
list connection {
    key "connection-id";
    uses connection;
  }
}
```

Solution: connection.py

```
$ python3 connection.py
```

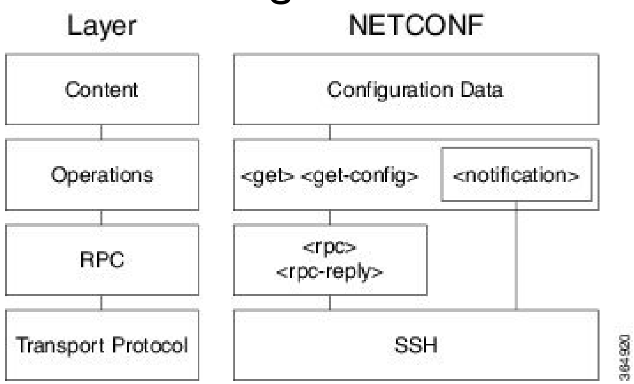
```
from binding_connection import connection
from pyangbind.lib.serialise import pybindIETFXMLEncoder
import pyangbind.lib.pybindJSON as pybindJSON

con = connection()
con1=con.connection.add("con1")
con1.source_node = "node1"
con1.target_node = "node2"
con1.source_port = "node1portA"
con1.target_port = "node2portA"
con1.bandwidth = 1000
con1.layer_protocol_name = "OPTICAL"
print(pybindIETFXMLEncoder.serialise(con))
print(pybindJSON.dumps(con))
```


```
<connection xmlns="urn:connection">
  <connection>
    <connection-id>con1</connection-id>
    <source-node>node1</source-node>
    <target-node>node2</target-node>
    <source-port>node1portA</source-port>
    <target-port>node2portA</target-port>
    <bandwidth>1000</bandwidth>
    <layer-protocol-name>OPTICAL</layer-protocol-name>
  </connection>
</connection>
```

NETCONF

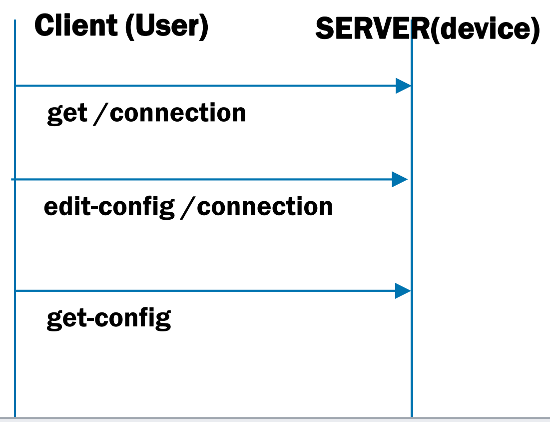
The NETCONF Protocol (I)

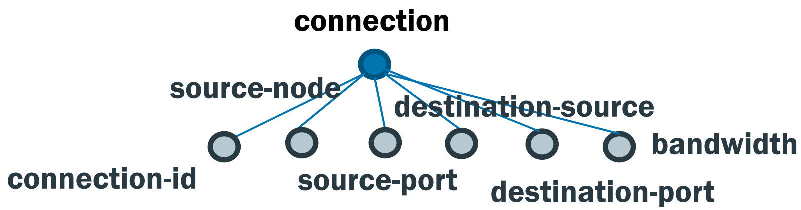
- Offers primitives to view and manipulate data, providing a suitable encoding as defined by the data-model.
  - Data is arranged into one or multiple *configuration datastores* (set of configuration information that is required to get a device from its initial default state into a desired operational state.)
- Enables remote access to a device, and provides the set of rules by which multiple clients may access and modify a datastore within a NETCONF server (e.g., device).
  - NETCONF enabled devices *include a NETCONF server*,
  - Management applications *include a NETCONF client* and device Command Line Interfaces (CLIs) can be a wrapped around a NETCONF client.
- It is based on the exchange of XML-encoded RPC messages over a secure (commonly Secure Shell, SSH) connection.
- NETCONF Layering :
  - Configuration or notification data (Content Layer) that is exchanged between a client and a server,
  - Operations layer (e.g. <get-config>, <edit-config>)
  - Message layer for RPC messages or notifications
  - Secure Transport.



The NETCONF Protocol (2)

- After establishing a session over a secure transport, both entities send a hello message to announce their protocol capabilities, the supported data models, and the server's session identifier.
- When accessing configuration or state data, with NETCONF operations, subtree filter expressions can select subtrees.





Operation	Description
<get>	Retrieve running configuration and device state information
<get-config>	Retrieve all or part of a specified configuration datastore
<edit-config>	Edit a configuration datastore by creating, deleting, merging or replacing content
<copy-config>	Copy an entire configuration datastore to another configuration datastore
<delete-config>	Delete a configuration datastore
<lock>	Lock an entire configuration datastore of a device
<unlock>	Release a configuration datastore lock previously obtained with the <lock> operation
<close-session>	Request graceful termination of a NETCONF session

```
# pyang -f tree connection.yang

module: connection
  +--rw connection* [connection-id]
    +--rw connection-id        string
    +--rw source-node?         ->
/topology:topology/node/node-id
    +--rw target-node?         ->
/topology:topology/node/node-id
    +--rw source-port?         ->
/topology:topology/node/port/port-id
    +--rw target-port?         ->
/topology:topology/node/port/port-id
    +--rw bandwidth?           uint32
    +--rw layer-protocol-name?   topology:layer-
protocol-name
```

Run a Netconf server

- For this example, we will use confd as a netconf server.
- Confd is not OpenSource, but follows a Freemium model, which allows testing and usage.
- Is a powerful server, with lots of options, and it is useful for training purposes.
- Later, we will introduce the development of a netconf server, using open source libraries.

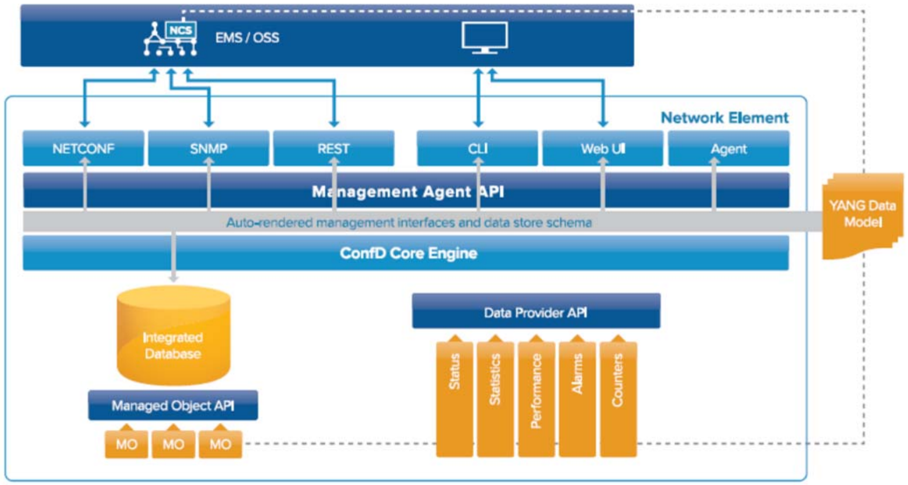


Figure 1: ConfD block diagram

Source:
http://www.tail-f.com/confd-basic/

Using Cisco (Tail-f) ConfD

- Installation

```
$ cd /root/OFC_SC472/netconf
$ unzip confd-basic-6.4.linux.x86_64.zip
$ cd confd-basic-6.4.linux.x86_64/
$ ./confd-basic-6.4.linux.x86_64.installer.bin /root/confd/
```

- Data-Model Compilation

```
$ cd /root/confd/bin/
$ ./confdc -c /root/OFC2019_SC472/yang/topology.yang
```

- Start ConfD

```
$ ./confd –foreground -v –addloadpath .
```

- Use ConfD-client

```
$ ./confd_cli
➢ conf
➢ topology node node1
➢ exit
➢ commit
➢ exit
➢ exit
```

```
$ ./confd_cli
➢ conf
➢ show full-configuration
➢ exit
➢ exit
```

Source:
http://www.tail-f.com/confd-basic/

NETCONF Basic server

- Use Python library: Netconf http://netconf.readthedocs.io/
- Simple server listening on port 830 that handles one RPC:
  - Read and parse as data the file topology.xml
  - Provide it when get-config is requested
- Serve as capability:
  - topology

Basic tutorial:
https://netconf.readthedocs.io/en/master/develop.html#netconf-server

Basic server (simplified)

```
import sys
import time
import logging
import os

from binding_topology import topology

from netconf import nsmap_add, NSMAP
from netconf import server, util
from lxml import etree

logging.basicConfig(level=logging.DEBUG)


nsmap_add("topology", "urn:topology")

class MyServer(object):
    def load_file(self):
        # create configuration
        xml_root = open('topology.xml',
'r').read()
        topo =
pybindIETFXMLDecoder.decode(xml_root,
binding_topology, "topology")
        xml =
pybindIETFXMLEncoder.serialise(topo)
        tree = etree.XML(xml)
        data = util.elm("nc:data")
        data.append(tree)
        self.node_topology = data
```

```
//(...)
def __init__(self, username, password, port):
        host_key_value =
os.path.join(os.path.abspath(os.path.dirname(__file__)),
"server-key")
        auth =
server.SSHUserPassController(username=username,
password=password)
        self.server =
server.NetconfSSHServer(server_ctl=auth,
server_methods=self, port=port, debug=False)
        self.load_file()

def nc_append_capabilities(self, capabilities):
        util.subelm(capabilities, "capability").text =
"urn:ietf:params:netconf:capability:xpath:1.0"
        util.subelm(capabilities, "capability").text =
NSMAP["topology"]

def rpc_get_config(self, session, rpc, source_elm,
filter_or_none):
        return util.filter_results(rpc, self.node_topology,
None)

def close(self):
        self.server.close()
```

```
def main(*margs):
    s = MyServer("admin","admin",
830)

    if sys.stdout.isatty():
        logging.debug("^C to quit
server")

    try:
        while True:
            time.sleep(1)

    except Exception:
        logging.debug("quitting server")

    s.close()
```

Basic client OSS client

- Create a client to CRUD the topology
- Python library: Netconf http://netconf.readthedocs.io/
- Tutorial: https://netconf.readthedocs.io/en/master/develop.html#netconf-client
- First, connect
- Second, print capabilities
- Third, get config
- Fourth, edit basic config

Netconf client

```
from lxml import etree
from netconf.client import NetconfSSHSession

# connexion parameters
host = 'localhost'
port = 2022
username = "admin"
password = "admin"

# connexion to server
session = NetconfSSHSession(host, port, username,
password)

# server capabilities
c = session.capabilities
print(c)

# get config
print("---GET CONFIG---")
config = session.get_config()
xmlstr = etree.tostring(config, encoding='utf8',
xml_declaration=True)
print(xmlstr)
...
```

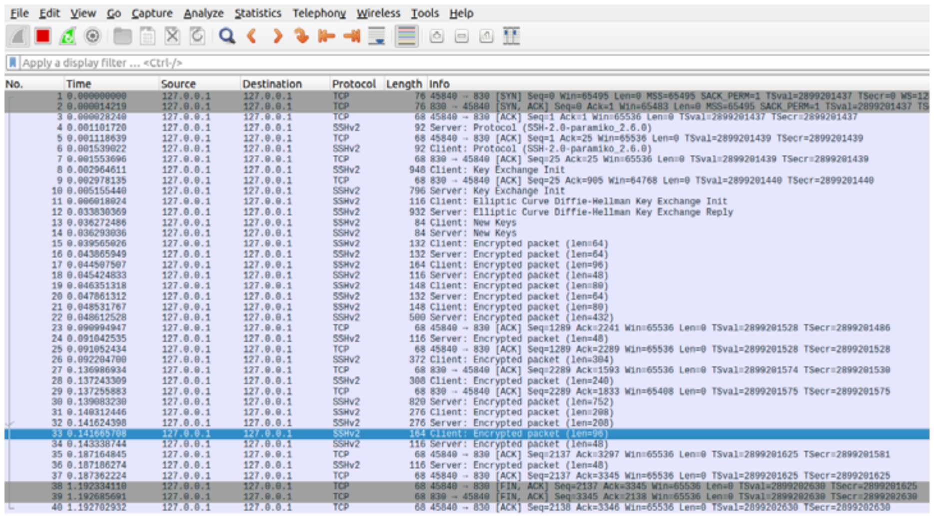
```
...

# edit config
new_config = '''
<config>
   <topology xmlns="urn:topology">
      <node operation="merge"> <!-- modify with delete -->
         <node-id>10.1.7.64</node-id>
         <port>
            <port-id>3</port-id>
         </port>
      </node>
   </topology>
</config>
'''
print("---EDIT CONFIG---")
config = session.edit_config(newconf=new_config)
xmlstr = etree.tostring(config, encoding='utf8',
xml_declaration=True)
print(xmlstr)

# close connexion
session.close()
```

Run NETCONF example

Run server:
$ cd /root/OFC_SC472/netconf
$ python3 serverTopology.py

Run client:
$ cd /root/OFC_SC472/netconf
$ python3 clientTopology.py

- Run Wireshark



NETCONF: edit-config example

- Include connection.yang
- Request to create a new connection (client and server).
- Server adds new connection
- Client list connection

Run server:
```
$ cd /root/OFC_SC472/netconf/connection
$ python3 serverTopologyConnection.py
```

Run client:
```
$ cd /root/OFC_SC472/netconf/connection
$ python3 clientConnection.py
```

CTTC

NETCONF server edit-config: serverTopologyConnection.py

```
def rpc_edit_config(self, session, rpc, target, new_config):
    logging.debug("--EDIT CONFIG--")
    logging.debug(session)

    data_list = new_config.findall(".//xmlns:connection", namespaces={'xmlns': 'urn:connection'})
    for connect in data_list:
        logging.debug("connect: " )
        logging.debug(etree.tostring(connect) )
        logging.debug("CURRENT CONNECTION")
        logging.debug(etree.tostring(self.data[1]) )
        self.data[1].append(connect)
        break
    return util.filter_results(rpc, self.data, None)
```

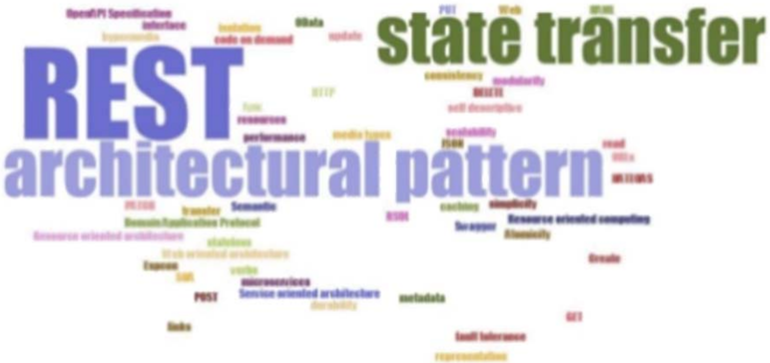
NETCONF client edit-config clientConnection.py

```
# edit config
new_config = '''
<config>
        <connection xmlns="urn:connection" operation="merge">
            <connection-id>connection1</connection-id>
            <source-node>node1</source-node>
            <source-port>node1portA</source-port>
            <target-node>node2</target-node>
            <target-port>node2portA</target-port>
            <bandwidth>10</bandwidth>
            <layer-protocol-name>ETH</layer-protocol-name>
        </connection>
</config>
'''
print("---EDIT CONFIG---")
config = session.edit_config(newconf=new_config)
xmlstr = etree.tostring(config, encoding='utf8', xml_declaration=True)
print(xmlstr)
```

RESTCONF

REST API

- A RESTful application is an application that exposes its state and functionality as a set of resources that the clients can manipulate and conforms to a certain set of principles:
  - All resources are uniquely addressable, usually through URIs; other addressing can also be used, though.
  - All resources can be manipulated through a constrained set of well-known actions, usually CRUD (create, read, update, delete), represented most often through the HTTP's POST, GET, PUT and DELETE;
  - The data for all resources is transferred through any of a constrained number of well-known representations, usually HTML, XML or JSON;
  - The communication between the client and the application is performed over a stateless protocol.



REST vs non-REST API

RESTful API

GET /user/15

```
{
"name" : "John Doe",
"email" : "john.doe@gmail.com"
...
}
```

Non-RESTful API

GET /last_search?page=2

```
{
"products" : [ ... ]
...
}
```

RESTCONF

- RESTCONF
  - RFC 8040
  - RESTful protocol to access YANG defined data
  - Representational State Transfer, i.e. server maintains no session state
  - URIs reflect data hierarchy in a Netconf datastore
  - HTTP as transport
  - Data encaded with either XML or JSON
  - Operations :

RESTCONF	Netconf
GET	<get-config>, <get>
POST	<edit-config> ("create")
PUT	<edit-config> ("replace")
PATCH	<edit-config> ("merge")
DELETE	<edit-config> ("delete")
OPTIONS	(discover supported operations)
HEAD	(get without body)

RESTCONF HTTP tree

- RESTCONF is a REST-like protocol that provides a HTTP-based API to access the data, modeled by YANG. The REST-like operations are used to access the hierarchical data within a datastore. The information modeled in YANG is structured in the following tree:
  - /restconf/data : “Data (configuration/operational) accessible from the client”
  - /restconf/modules : “Set of YANG models supported by the RESTCONF server”
  - /restconf/operations : “Set of operations (**YANG-defined RPCs**) supported by the server”
  - /restconf/streams: “Set of notifications supported by the server”

OpenAPI specs

- Question: How can we define an standardized REST API?

- Open API (formerly known as Swagger) is a popular compact and easy to parse data schema format to describe REST APIs
  - Open API Schemas can be described in two popular web encoding languages – YAML or JSON
- The generated RESTconf OpenAPI specifications provide a mapping from the Yang data schema into OpenAPI JSON format, which can then be used to generate Python and/or Java code for implementation of the API in RestConf
- https://www.openapis.org/
- https://swagger.io

Generate OpenAPI (from YANG to OpenAPI)

- ONF Eagle tool chain:

https://github.com/bartoszm/yang2swagger/releases/tag/1.1.11

- Project is a YANG to Swagger (OpenAPI Specification) generator tool. OpenAPI describes and documents RESTful APIs. The Swagger definition generated with our tool is meant to be compliant with RESTCONF specification. Having the definition you are able to build live documentation services, and generate client or server code using Swagger tools.
- Usage:

```
java -jar swagger-generator-cli-1.0-SNAPSHOT-executable.jar
Argument "module ..." is required
 module ...         : List of YANG module names to generate in swagger output
 -output file       : File to generate, containing the output - defaults to stdout
                      (default: )
 -yang-dir path      : Directory to search for YANG modules - defaults to current
                      directory (default: )
 -api-version string : The current version of your API (default: 1.0)
 -format enum        : The output format (options: YAML, JSON) (default: YAML)
 -content-type string: Content type the API generates / consumes (default: application/yang-data+json)
```

Exercise: Generate topology/connection OpenAPI

- Follow yang2swagger tool calls

```
$ cd /root/OFC_SC472/restconf
$ wget https://github.com/bartoszm/yang2swagger/releases/download/1.1.11/swagger-generator-cli-1.1.11-executable.jar
$ java -jar swagger-generator-cli-1.1.11-executable.jar -yang-dir ../yang/ -output topology.yaml topology
$ java -jar swagger-generator-cli-1.1.11-executable.jar -yang-dir ../yang/ -output connection.yaml connection
```

Understanding topology OpenAPI (I)

- Paths
  - Each path may include CRUD (POST, GET, PUT, DELETE) if config
  - Only GET is allow for State data
  - Each CRUD includes the following details:
    - Summary
    - Parameters (in path or in body)
    - Responses
    - Produces/consumes


```yaml
---
swagger: "2.0"
info:
  description: "topology API generated from yang definitions"
  version: "1.0"
  title: "topology API"
host: "localhost:1234"
consumes:
- "application/yang-data+json"
produces:
- "application/yang-data+json"
paths:
  /data/topology/:
    get:
      tags:
      - "topology"
      description: "returns topology.Topology"
      parameters: []
      responses:
        200:
          description: "topology.Topology"
          schema:
            $ref: "#/definitions/topology.Topology"
        400:
          description: "Internal error"
    post:
      tags:
      - "topology"
      description: "creates topology.Topology"
      parameters:
      - in: "body"
        name: "topology.Topology.body-param"
        description: "topology.Topology to be added to list"
        required: false
        schema:
          $ref: "#/definitions/topology.Topology"
      responses:
        201:
          description: "Object created"
        400:
          description: "Internal error"
        409:
          description: "Object already exists"
    put:
    delete:
```

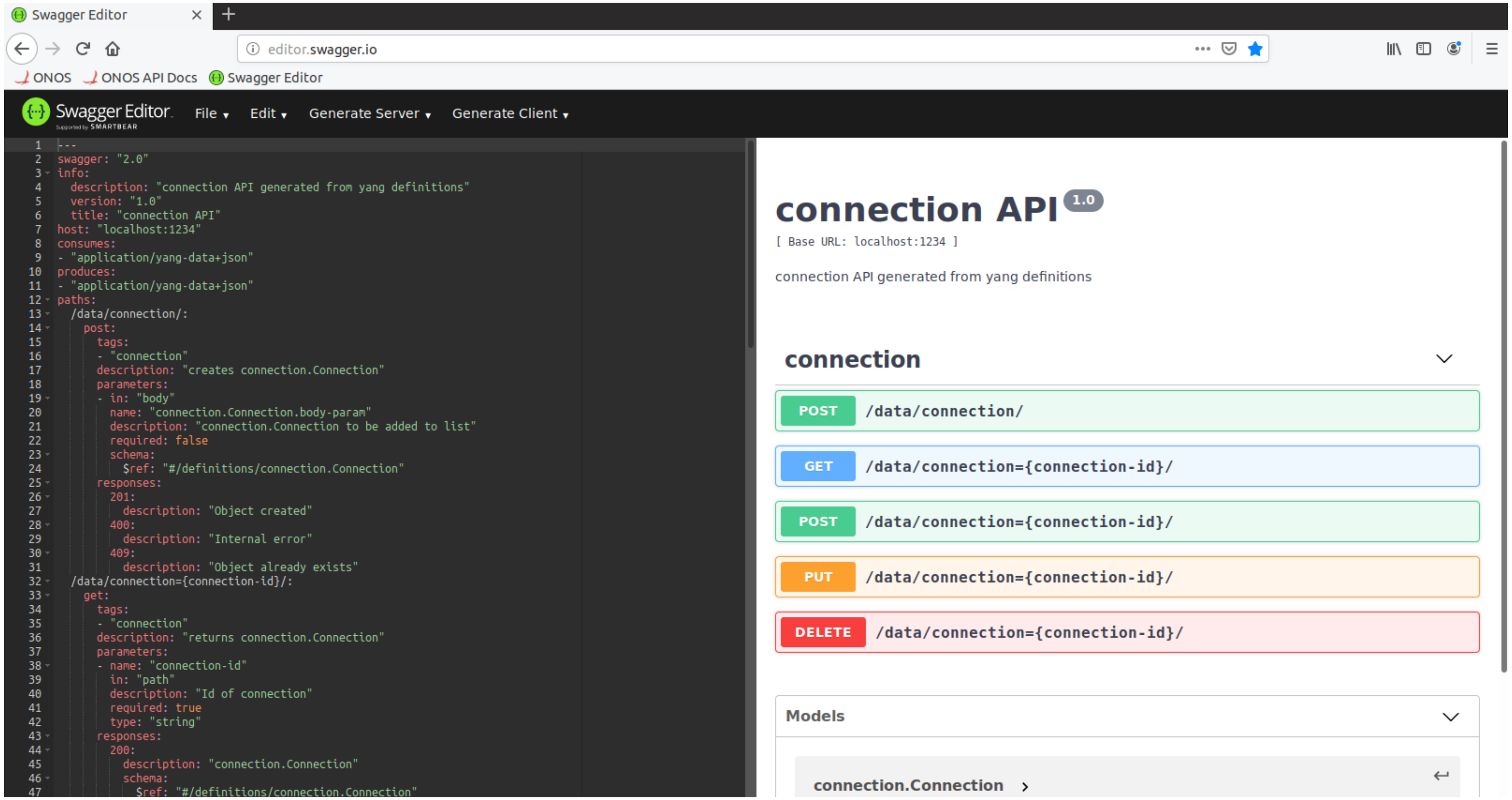
Understanding topology OpenAPI (II)

- Definitions
  - Common Types: Object, Array, String
  - Items are described in properties
  - Other descriptions might be referenced
  - They allow inheritance (allOf)

```yaml
definitions:
  topology.LayerProtocolName:
    type: "string"
    enum:
    - "ETH"
    - "OPTICAL"
  topology.Link:
    type: "object"
    properties:
      target-port:
        type: "string"
        x-path: "/topology/node/port/port-id"
      source-port:
        type: "string"
        x-path: "/topology/node/port/port-id"
      target-node:
        type: "string"
        x-path: "/topology/node/node-id"
      link-id:
        type: "string"
      source-node:
        type: "string"
        x-path: "/topology/node/node-id"
  topology.Node:
    type: "object"
    properties:
      node-id:
        type: "string"
      port:
        type: "array"
        items:
          $ref: "#/definitions/topology.Port"
  topology.Port:
    type: "object"
    properties:
      layer-protocol-name:
        $ref: "#/definitions/topology.LayerProtocolName"
      port-id:
        type: "string"
  topology.Topology:
    type: "object"
    properties:
      link:
        type: "array"
        items:
          $ref: "#/definitions/topology.Link"
      node:
        type: "array"
        items:
          $ref: "#/definitions/topology.Node"
```

Swagger Editor

- Use firefox to open: editor.swagger.io



Generate Server Stub

- Swagger Codegen simplifies your build process by generating server stubs and client SDKs for any API, defined with the OpenAPI specification.


```
$ cd /root/OFC_SC472/restconf
$ wget https://repo1.maven.org/maven2/io/swagger/codegen/v3/swagger-codegen-cli/3.0.11/swagger-codegen-cli-3.0.11.jar -O swagger-codegen-cli.jar
$ java -jar swagger-codegen-cli.jar generate -i connection.yaml -l python-flask -o server/
```

- Run the server:

```
$ cd /root/OFC_SC472/restconf/server
$ pip3 install -r requirements.txt
(Open server/swagger_server/swagger/swagger.yaml and modify all: name: connection_id for name: connection-id)
$ python3 -m swagger_server
```

Source:
https://github.com/swagger-api/swagger-codegen

Create a Connection Server

- Inspect server (__main__.py)

```
app.app.config['JSON_SORT_KEYS']=False
```

- Create a database object, where we can store and access a context json object

```
database.connection={}
```

- Modify default controller behavior

```
data_connection_post(connection_Connection_body_param=None)
data_connectionconnection_id_get(connection_id)
```

- Write backend
- Use curl as client

Connection Server

```
import connexion
import six
import swagger_server.database as database
from swagger_server.models.connection_connection import ConnectionConnection  # noqa: E501
from swagger_server import util

def data_connection_post(connection_Connection_body_param=None):  # noqa: E501
   if connexion.request.is_json:
      connection_Connection_body_param = ConnectionConnection.from_dict(connexion.request.get_json())
connection_Connection_body_param.connection_id=str(database.last_connection_id)
      database.connection[str(database.last_connection_id)] = connection_Connection_body_param
      database.last_connection_id+=1
   return connection_Connection_body_param


def data_connectionconnection_id_delete(connection_id):  # noqa: E501
   del database.connection[connection_id]
   return 'ok'

def data_connectionconnection_id_get(connection_id):  # noqa: E501
   print(database.connection)
   return database.connection[connection_id]
```

CURL AS AN HTTP REST CLIENT

- curl is a command line tool which is used to transfer data over the internet.
- Examples:

```
$ curl -X POST -H "Content-Type: application/yang-data+json" http://127.0.0.1:8080/data/connection/ -d@conn1.json
$ curl -X GET -H "Content-Type: application/yang-data+json" http://127.0.0.1:8080/data/connection=0/
```

conn1.json

```
{
  "source-node" : "node1",
  "target-node" : "node2",
  "source-port" : "node1portA",
  "target-port" : "node2portA",
  "bandwidth" : 10
}
```

Run Connection Server

- Run connection server

```
$ cd /root/OFC_SC472/restconf/connectionserver
$ python3 -m swagger_server
```

- Run curl as client

```
curl -X POST -H "Content-Type: application/yang-data+json" http://127.0.0.1:8080/data/connection/ -d@conn1.json
curl -X GET -H "Content-Type: application/yang-data+json" http://127.0.0.1:8080/data/connection=0/
curl -X DELETE -H "Content-Type: application/yang-data+json" http://127.0.0.1:8080/data/connection=0/
```

ONOS architecture

Applications

Bandwidth on-demand, calendaring, optical restoration
Power balancing, fault management & correlation

Northbound Abstractions

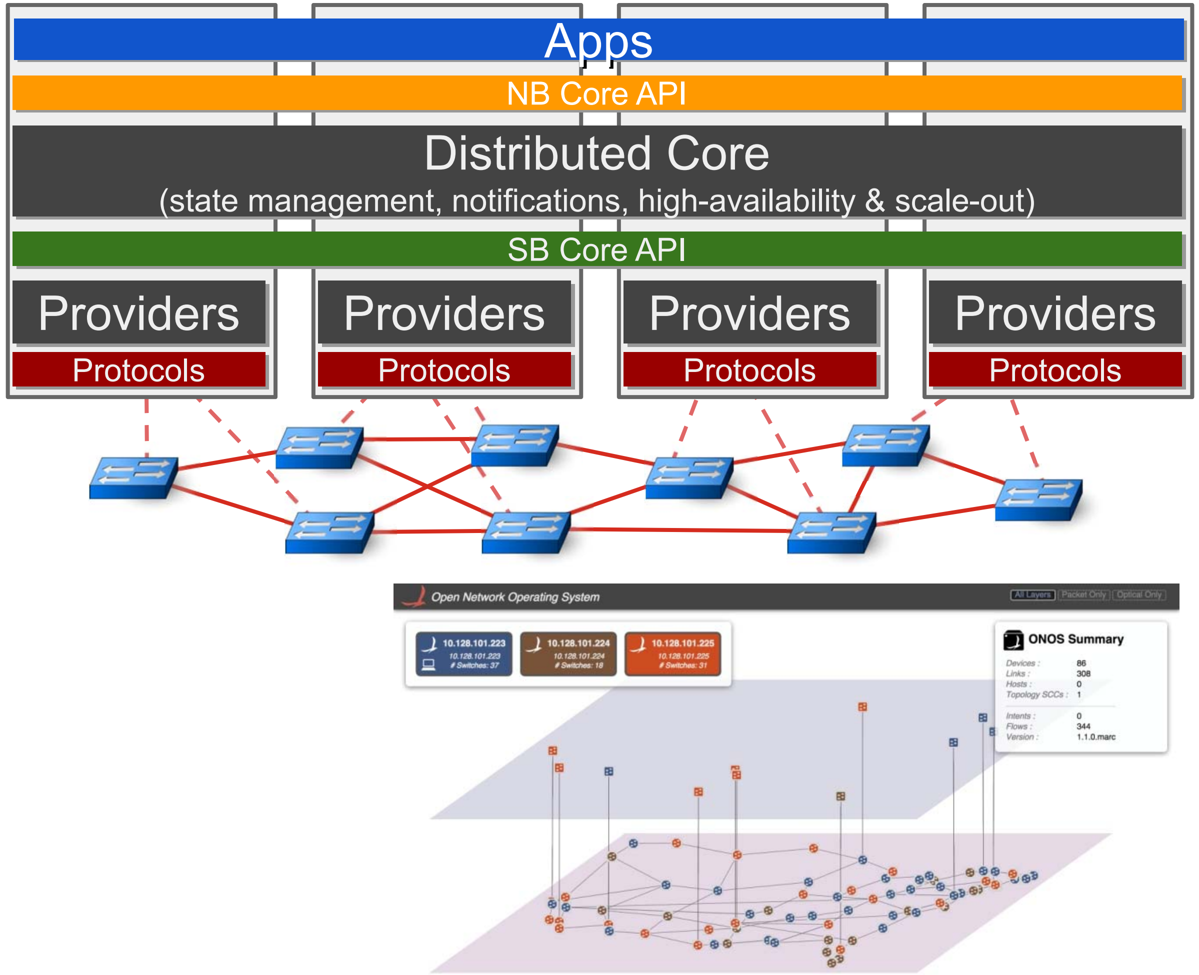
Intent framework
Converged topology graph

ONOS Core: Scale & HA

Modular PCE
Optical information model
Resource manager

Southbound Drivers

OpenFlow, NETCONF,
TL1, PCEP, SNMP, REST
P4Runtime



ONOS NBI

- Run ONOS:

>> cd onos-1.12.0/apache-karaf-3.0.8/bin

>> ./karaf clean

$$ app activate org.onosproject.openflow
$$ app activate org.onosproject.gui

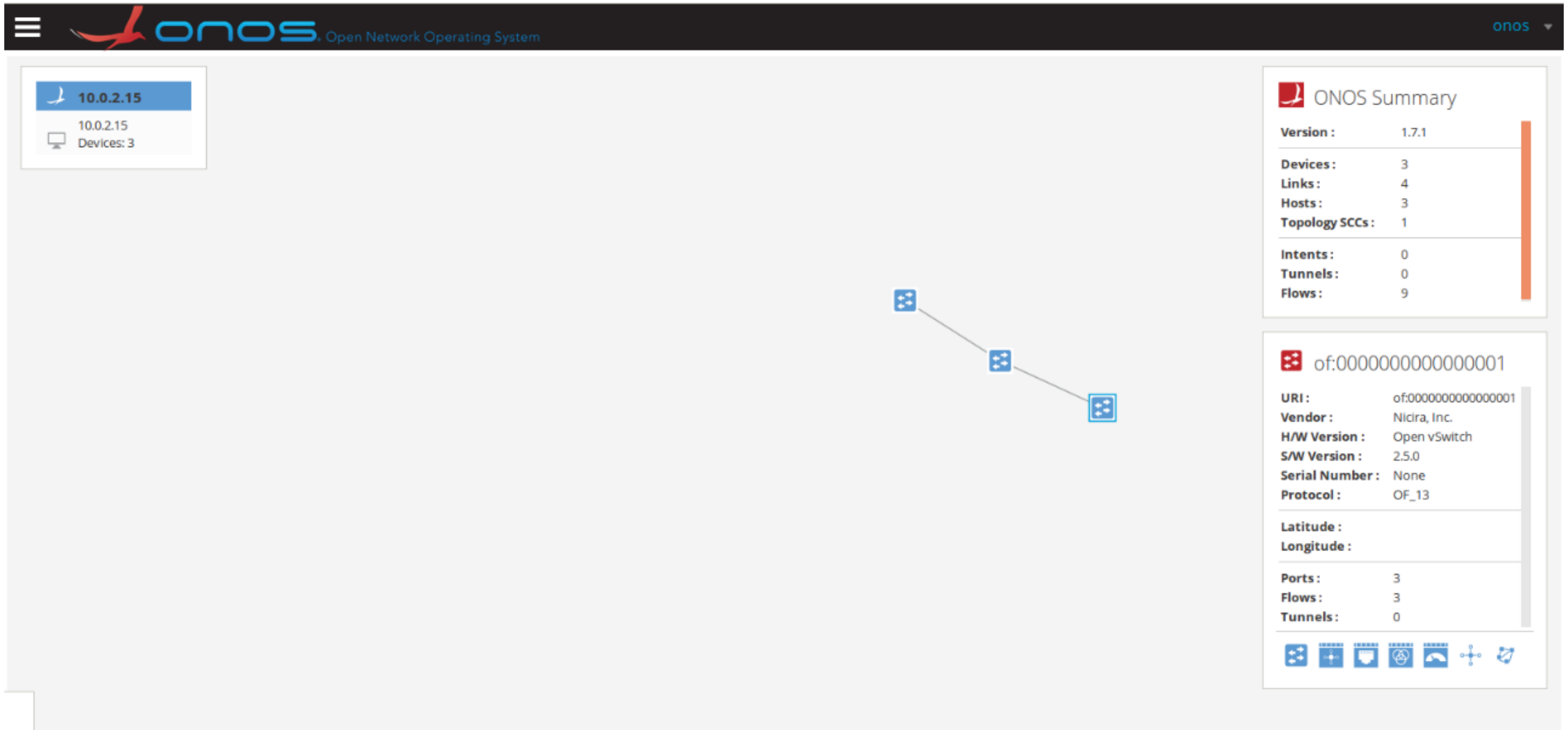
🡐Command to run in ONOS CLI

- Open Firefox:

http://127.0.0.1:8181/onos/ui/index.html
When asked for user/password use onos/rocks

RUN mininet

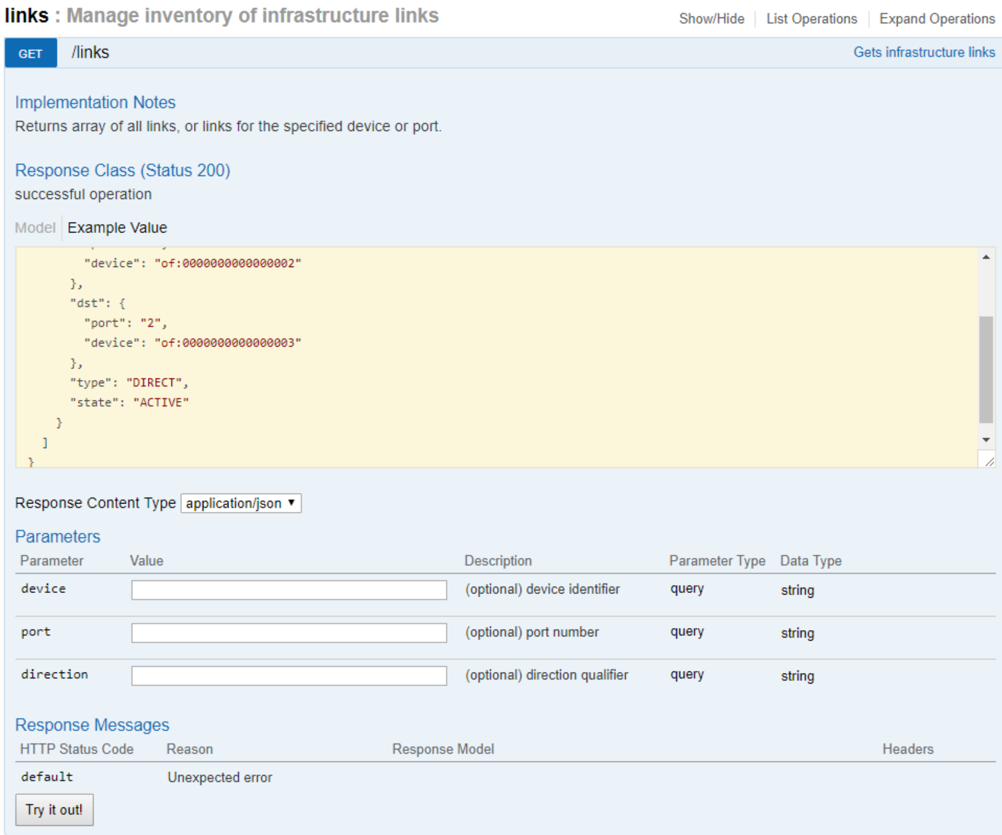
- mn --topo linear,3 --mac --controller=remote,ip=127.0.0.1,port=6653 --switch ovs,protocols=OpenFlow13



ONOS LINKS REST API

- http://localhost:8181/onos/v1/docs/
- curl -X GET -u onos:rocks --header ‘Accept: application/json’ http://localhost:8181/onos/v1/links | python -m json.tool

{ "links": [
{ "src": { "port": "3", "device": "of:0000000000000002" }, "dst": { "port": "2", "device": "of:0000000000000003" }, "type": "DIRECT", "state": "ACTIVE" },
{ "src": { "port": "2", "device": "of:0000000000000002" }, "dst": { "port": "2", "device": "of:0000000000000001" }, "type": "DIRECT", "state": "ACTIVE" },
{ "src": { "port": "2", "device": "of:0000000000000003" }, "dst": { "port": "3", "device": "of:0000000000000002" }, "type": "DIRECT", "state": "ACTIVE" },
{ "src": { "port": "2", "device": "of:0000000000000001" }, "dst": { "port": "2", "device": "of:0000000000000002" }, "type": "DIRECT", "state": "ACTIVE" }
] }



Example using ONOS TOPOLOGY REST API in Python

- cd /root/OFC_SC472/onos_api/
- python3 onos_topology.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

import requests
from requests.auth import HTTPBasicAuth
import json

IP='127.0.0.1'
PORT='8181'
USER='onos'
PASSWORD='rocks'

def retrieveTopology(ip, port, user, password):
    http_json = 'http://' + ip + ':' + port + '/onos/v1/links'
    response = requests.get(http_json, auth=HTTPBasicAuth(user, password))
    topology = response.json()
    return topology

if __name__ == "__main__":

    print "Reading network-topology"
    topo = retrieveTopology(IP, PORT, USER, PASSWORD)
    print json.dumps(topo, indent=4, sort_keys=True)

```

Calling ONOS FLOW REST API with curl

- http://localhost:8181/onos/v1/docs/

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{ \
  "flows": [ \
   { \
     "priority": 40000, \
     "timeout": 0, \
     "isPermanent": true, \
     "deviceId": "of:0000000000000001", \
     "treatment": { \
      "instructions": [ \
       { \
        "type": "OUTPUT", \
        "port": "CONTROLLER" \
       } \
      ] \
     }, \
     "selector": { \
      "criteria": [ \
       { \
        "type": "ETH_TYPE", \
        "ethType": "0x88cc" \
       } \
      ] \
     } \
   } \
  ] \
}' 'http://10.1.7.17:8181/onos/v1/flows?appId=tapi0'
```

1. when device of:000…1

3. output the packet to controller

2. encounter a packet with EthType 0x88cc (=LLDP)

Example using ONOS FLOW REST API in Python

OFC_SC472/onos_api/onos_flows.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import requests
from requests.auth import HTTPBasicAuth
import json

IP='localhost'
PORT='8181'
USER='onos'
PASSWORD='rocks'

URL = 'http://' + IP + ':' + PORT + '/onos/v1/flows/'

def insertFlow( nodeId, priority, inport, outport ):

    flow='{ "priority": '+priority+', "timeout": 0, "isPermanent": true, "deviceId": "'+nodeId+
          '", "treatment": { "instructions": [ { "type": "OUTPUT", "port": "'+outport+
          '" } ] }, "selector": { "criteria": [ { "type": "IN_PORT", "port": "'+inport+'" } ] } }'


    print "Flow: " + flow
    url = URL + nodeId + '?appId=tuto'
    headers = {'content-type': 'application/json'}
    print url
    response = requests.post(url, data=flow,
                    headers=headers, auth=HTTPBasicAuth(USER,
                    PASSWORD))
    print response
    return { 'status':response.status_code, 'content': response.content}

def deleteFlow(nodeId, flow_id):

    url = URL + '' + nodeId + '/' + flow_id
    response = requests.delete(url, auth=HTTPBasicAuth(USER, PASSWORD))
    return {'flow_id':flow_id, 'status':response.status_code, 'content': response.content}


if __name__ == "__main__":

    print "Setting flow"

    res = insertFlow(nodeId="of:0000000000000001", priority="40001", inport="1", outport="2")
    print json.dumps(res, indent=4, sort_keys=True)
```

ONF TRANSPORT API 2.0

Launch/Run TAPI Reference Implementation

- Run in a terminal:

```
$ cd /root/OFC_SC472/tapi/server
$ python3 tapi_server.py
```

- Run in a new terminal:

```
$ cd /root/OFC_SC472/tapi/client
$ curl -X GET -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/
```

TAPI Context, Topology & Connectivity Overview

- All TAPI interaction between an TAPI provider (SDN Controller) and an TAPI Client (Application, Orchestrator or parent SDN Controller) occur within a shared "*Context*"
- TAPI *Context* is defined by a set of *ServiceInterfacePoints* (and some policy)
  - *ServiceInterfacePoints* enable TAPI Client to request TAPI Services between them.
- A TAPI provider may expose 1 or more abstract *Topology* within shared *Context*
  - These topologies may or may-not map 1-to-1 to a provider's internal topology.
- A *Topology* is expressed in terms of *Nodes* and *Links*.
  - *Nodes* aggregate *NodeEdgePoints, Links* connect 2 Nodes & terminate on *NodeEdgePoints*
  - *NodeEdgePoints* may be mapped to 1 or more *ServiceInterfacePoints* at edge of Network
- TAPI Client requests *ConnectivityService* between 2 or more *ServiceInterfacePoints*
- TAPI Provider creates 1 or more *Connections* in response to *ConnectivityService*
  - *ConnectionEndPoints* encapsulate information related to a *Connection* at the ingress/egress points of every *Node* that the *Connection* traverses in a *Topology*
  - Every *ConnectionEndPoint* is supported by a specific "parent" *NodeEdgePoint*
  - Thus with reference to *ConnectivityServices*, a *ServiceInterfacePoint* conceptually represents a pool of "potential" *ConnectionEndPoints* at the edge of the Network

TAPI: Retrieve Context

- GET Context Details

curl -X GET -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/

Response:

```
{   "uuid" : "ctx-ref",
    "service-interface-point" : [
       {……},
       ……
    ],
    "topology" : [
       {……},
       ……
    ],
    "connectivity-service" : [
       {……},
       ……
    ],
    "connection" : [
       {……},
       ……
    ]
}
```

Proper TAPI implementations should use UUID format. An example below: f81d4fae-7edc-11d0-a765-00a0c91e6bf6

TAPI Context is a Container for all ServiceInterfacePoints, Topologies, ConnectivityServices, Connections, etc data.

TAPI: Retrieve List of Service Interface Points

- GET List of Service Interface Points

curl -X GET -H "Content-Type: application/json"
http://127.0.0.1:8080/restconf/config/context/service-interface-point/

Response:


```
{
    [
        “/restconf/config/context/service-interface-point/sip-pe1-uni1/”,
        “/restconf/config/context/service-interface-point/sip-pe1-uni2/”,
        “/restconf/config/context/service-interface-point/sip-pe2-uni1/”,
        “/restconf/config/context/service-interface-point/sip-pe2-uni2/”,
        “/restconf/config/context/service-interface-point/sip-pe3-uni1/”,
        “/restconf/config/context/service-interface-point/sip-pe3-uni2/”
    ]
}
```


Can use the returned URI to make additional retrievals

TAPI: Retrieve Service Interface Point Details

- GET Service Interface Point Details

curl -X GET -H "Content-Type: application/json"
http://127.0.0.1:8080/restconf/config/context/service-interface-point/sip-pe1-uni1/

Response:


```
{   "uuid" : "sip-pe1-uni1",
    "name": [ … ],
    "layer-protocol-name": [ "ETH", "ODU" ],
    "administrative-state": "UNLOCKED",
    "operational-state": "ENABLED",
    "lifecycle-state": "INSTALLED"
    "total-potential-capacity": {
       "total-size": {"value": "10", "unit": "GBPS"},
       "bandwidth-profile": {……}
    }
    "available-capacity": {
       "total-size": {"value": "10", "unit": "GBPS"},
       "bandwidth-profile": {……}
    }
    ……
}
```


Most TAPI objects have layer & state attributes

ServiceInterfacePoint conveys the capabilities of the logical interface point

TAPI: Retrieve List of Topologies

- GET List of Topologies

curl -X GET -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/topology/

Response:


```
{
    [
        "/restconf/config/context/topology/topo-nwk/",
        "/restconf/config/context/topology/topo-pe1/",
        "/restconf/config/context/topology/topo-pe2/",
        "/restconf/config/context/topology/topo-pe3/"
    ]
}
```


Can use the returned URI to make additional retrievals

TAPI: Retrieve Topology Details

- GET Topology Details

curl -X GET -H "Content-Type: application/json"
http://127.0.0.1:8080/restconf/config/context/topology/topo-nwk/

Response:

```
{   "uuid" : "topo-nwk",
    "name": [
       { "value-name": "name",
         "value": "NETWORK_TOPOLOGY"
       }
       ......
    ],
    "node" : [
       {......},
       ......
    ],
    "link" : [
       {......},
       ......
    ]
}
```

Every TAPI object has a name attribute that is defined as a list of name-value pairs.

Topology contains Nodes & Links (by value).

TAPI: Retrieve Node Details - I

- GET Node Details

curl -X GET -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/topology/topo-nwk/node/node-mul-pe-1/

Response:


```
{   "uuid" : "node-mul-pe-1",
    "name": [ … ],
    "layer-protocol-name": [ "ETH", "ODU" ],
    "administrative-state": "UNLOCKED",
    "operational-state": "ENABLED",
    "lifecycle-state": "INSTALLED"
    "encap-topology": "/restconf/config/context/topology/topo-pe1/",
    "owned-node-edge-point": [ ],
    "aggregated-node-edge-point" : [
        "/restconf/config/context/topology/topo-pe1/node/node-eth-pe-1/owned-node-edge-point/nep-pe1-eth-uni1/",
        ……
    ],
    ……
}
```


Node can be single or multi layer

Abstract Node is an abstraction of a Topology

Node can also constrain forwarding across its aggregated NodeEdgePoints (not shown here)

Node represents the potential to forward data between its aggregated NodeEdgePoints

TAPI: Retrieve Node Details - 2

- GET Node Details

curl -X GET -H "Content-Type: application/json"
http://127.0.0.1:8080/restconf/config/context/topology/topo-pe1/node/node-eth-pe-1/

Response:


```
{   “uuid” : “node-eth-pe-1”,
    “name”: [ … ],
    “layer-protocol-name”: [“ETH” ],
    “administrative-state”: “UNLOCKED”,
    “operational-state”: “ENABLED”,
    “lifecycle-state”: “INSTALLED”
    “encap-topology”: “”,
    “owned-node-edge-point”: [
       {……},
       ……
    ],
    “aggregated-node-edge-point” : [
        “/restconf/config/context/topology/topo-pe1/node/node-eth-pe-
1/owned-node-edge-point/nep-pe1-eth-uni1/”,
        ……
    ]
}
```


Switch Node is typically single layer

Switch Node contains/owns a list of NodeEdgePoints

TAPI: NodeEdgePoint Details

- NodeEdgePoint

curl -X GET -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/topology/topo-pe1/node/node-eth-pe-1/owned-node-edge-point/nep-pe1-eth-uni1/

```
{   "uuid" : "nep-pe1-eth-uni1",
    "name": [ … ],
    "layer-protocol-name": "ETH",
    "administrative-state": "UNLOCKED",
    "operational-state": "ENABLED",
    "lifecycle-state": "INSTALLED"
    "termination-state": "LP_CAN_NEVER_TERMINATE",
    "termination-direction": "BIDIRECTIONAL",
    "link-port-direction": "BIDIRECTIONAL",
    "link-port-role": "SYMMETRIC",
    "mapped-service-interface-point" : [
        "/restconf/config/context/service-interface-point/sip-pe1-uni1/",
        ……
    ]
}
```

NodeEdgePoint is single layer

NodeEdgePoint can be mapped to (1 or more) ServiceInterfacePoint to function as a network interface. This attribute is empty for "internal" NodeEdgePoints

TAPI: Retrieve Link Details - I

- GET Link Details

curl -X GET -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/topology/topo-nwk/link/link-pe1-odu4-nni1-pi4-odu4-nni2/


```
{   "uuid" : "link-pe1-odu4-nn1-pi4-odu4-nni1",
    "name": [ … ],
    "layer-protocol-name": ["ODU" ],
    "direction": "BIDIRECTIONAL",
    "resilience-type": {……},
    "total-potential-capacity": {……},
    "available-capacity": {……},
    "cost-characteristic": {……},
    "latency-characteristic": {……},
    ……
    ……
    "node-edge-point" : [
        "/restconf/config/context/topology/topo-pe1/node/node-odu-pe-1/owned-node-edge-point/nep-pe1-odu4-nni1/",
        "/restconf/config/context/topology/topo-nwk/node/node-odu-pi-4/owned-node-edge-point/nep-pi4-odu4-nni2/"
    ]
}
```


Link conveys "transfer-characteristic" information

Link represents adjacency information between 2 NodeEdgePoints

TAPI: Retrieve Link Details - 2

- GET Link Details

curl -X GET -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/topology/topo-pe1/link/link-pe1-eth-pool-pe1-odu2-pool/

```
{   "uuid" : "link-pe1-eth-pool-pe1-odu2-pool",
    "name": [ … ],
    "layer-protocol-name": ["ETH", "ODU"],
    "direction": "BIDIRECTIONAL",
    "resilience-type": {……},
    "total-potential-capacity": {……},
    "available-capacity": {……},
    "cost-characteristic": {……},
    "latency-characteristic": {……},
    ……
    ……
    "node-edge-point" : [
        "/restconf/config/context/topology/topo-pe1/node/node-eth-pe-
1/owned-node-edge-point/nep-pe1-eth-pool/",
        "/restconf/config/context/topology/topo-pe1/node/node-odu-pe-
1/owned-node-edge-point/nep-pe1-odu2-pool/"
    ]
}
```

"Transitional" Link connects NodeEdgePoints from different layers and conveys the layer-transition information

Writing a TAPI Topology client

- Objective:
  - Retrieve and draw Network Topology using TAPI
- Steps:
  - Run TAPI-RI
  - Load topological information
  - Start coding using the following libraries:
    - NetworkX
    - matplotlibt
    - Requests
    - Json

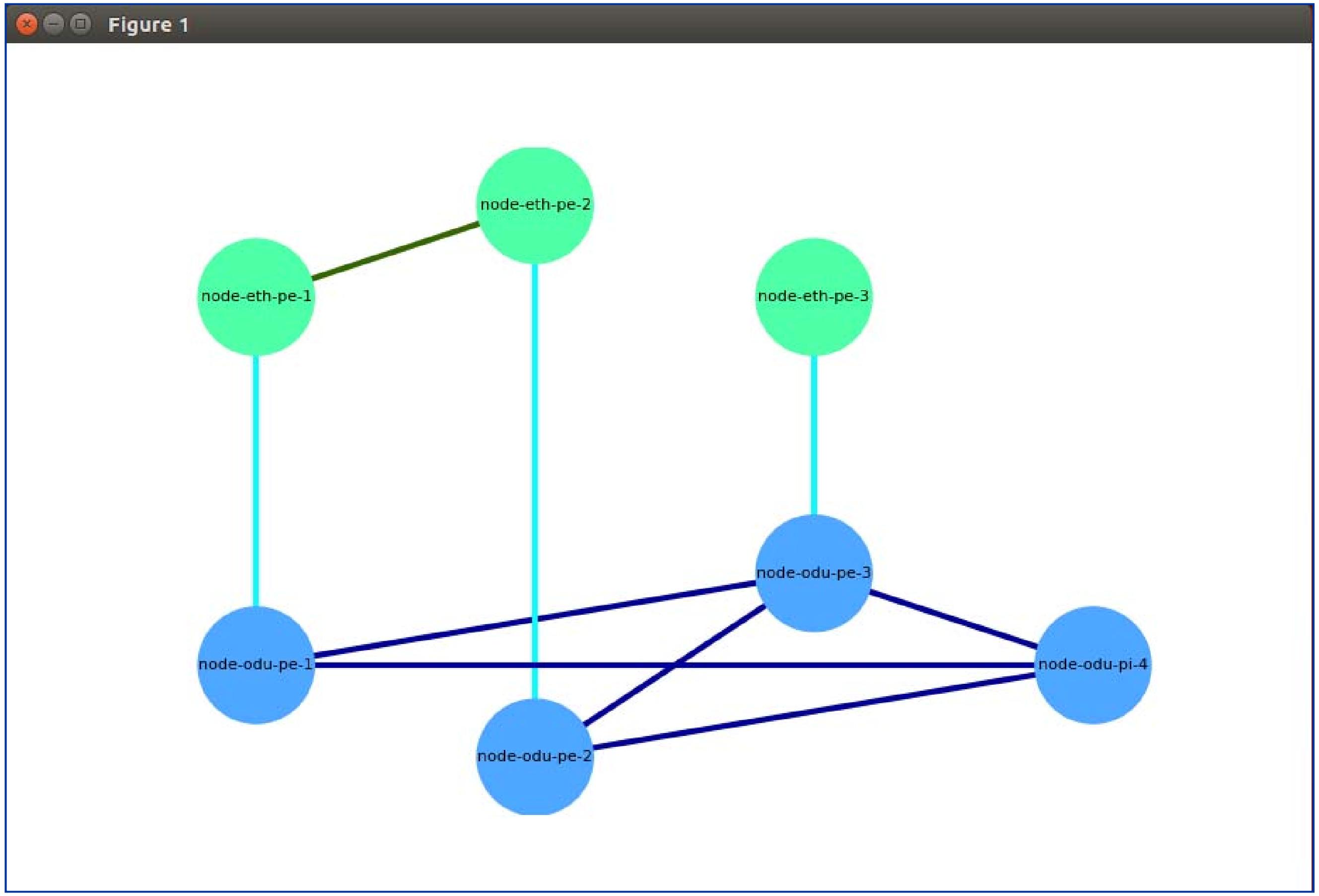
TAPI_APP

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-


import requests
from requests.auth import HTTPBasicAuth
import json
import matplotlib.pyplot as plt
import networkx as nx

IP='127.0.0.1'
PORT='8080'

def retrieveTopology(ip, port, user='', password=''):
    http_json = 'http://' + ip + ':' + port + '/restconf/config/context/topology/top0'
    response = requests.get(http_json, auth=HTTPBasicAuth(user, password))
    topology = response.json()
    return topology

def load_topology ( topology) :
    G=nx.Graph()
    for link in topology['link']:
      node_src = link['node-edge-point'][0].split('restconf/config/context/topology/top0/node/')[1].split('/')[0]
      node_dst = link['node-edge-point'][1].split('restconf/config/context/topology/top0/node/')[1].split('/')[0]
      G.add_edge( node_src, node_dst )
      print 'Link: ' + node_src + ' ' + node_dst
    nx.draw(G)
    plt.show()

if __name__ == "__main__":
    print "Reading network-topology"
    topo = retrieveTopology(IP, PORT)
    print json.dumps(topo, indent=4, sort_keys=True)
    load_topology(topo)
```

Run TAPI Application Client

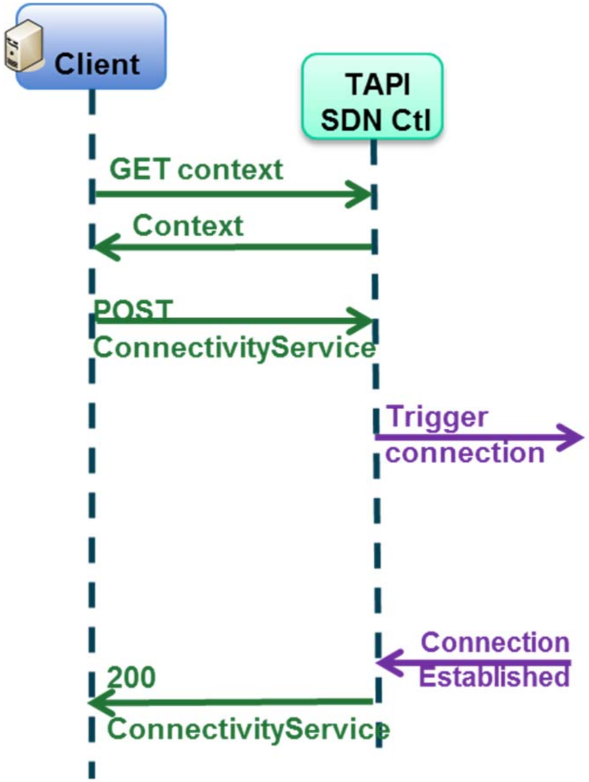
- Run in a terminal:

```
$ cd /root/OFC_SC472/tapi/tapi_app
$ python3 tapi_app.py
```



TAPI: Connectivity Service workflow

```
$ cd /root/OFC_SC472/tapi/client
```



TAPI: Establish Connectivity Service

- curl -X POST -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/connectivity-service/cs1/ -d @cs1.json
- cs1.json:

```
{ "uuid" : "conn-service-1",
  "service-type":"POINT_TO_POINT_CONNECTIVITY",
  "requested-capacity": {"total-size": { "value": "1", "unit": "GBPS" }},
  "end-point":[
    { "local-id": "csep-1",
      "layer-protocol-name": "ETH",
      "direction":"BIDIRECTIONAL",
      "role":"SYMMETRIC",
      "service-interface-point":
          "/restconf/config/context/service-interface-point/sip-pe1-uni1"},
    { "local-id": "csep-2",
     "layer-protocol-name": "ETH",
     "direction":"BIDIRECTIONAL",
     "role":"SYMMETRIC",
     "service-interface-point":
        "/restconf/config/context/service-interface-point/sip-pe2-uni1"}
  ]
}
```

ConnectivityService endpoint information has to specify the ServiceInterfacePoint

TAPI: Created Connection

- GET Connection Details:
- curl -X GET -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/context/connection/cs1/


```
{
    "uuid" : "cs1",
    "connection-end-point": [
       "/restconf/config/topology/top0/node/node1/owned-node-edge-
point/nep11/cep-list/cep11",
        "/restconf/config/topology/top0/node/node1/owned-node-edge-
point/nep12/cep-list/cep11“
  ]
}
```


ConnectivityService has triggered the establishment of a Connection

Node Edge Point is augmented with a list of Connection End Points

Other TAPI models

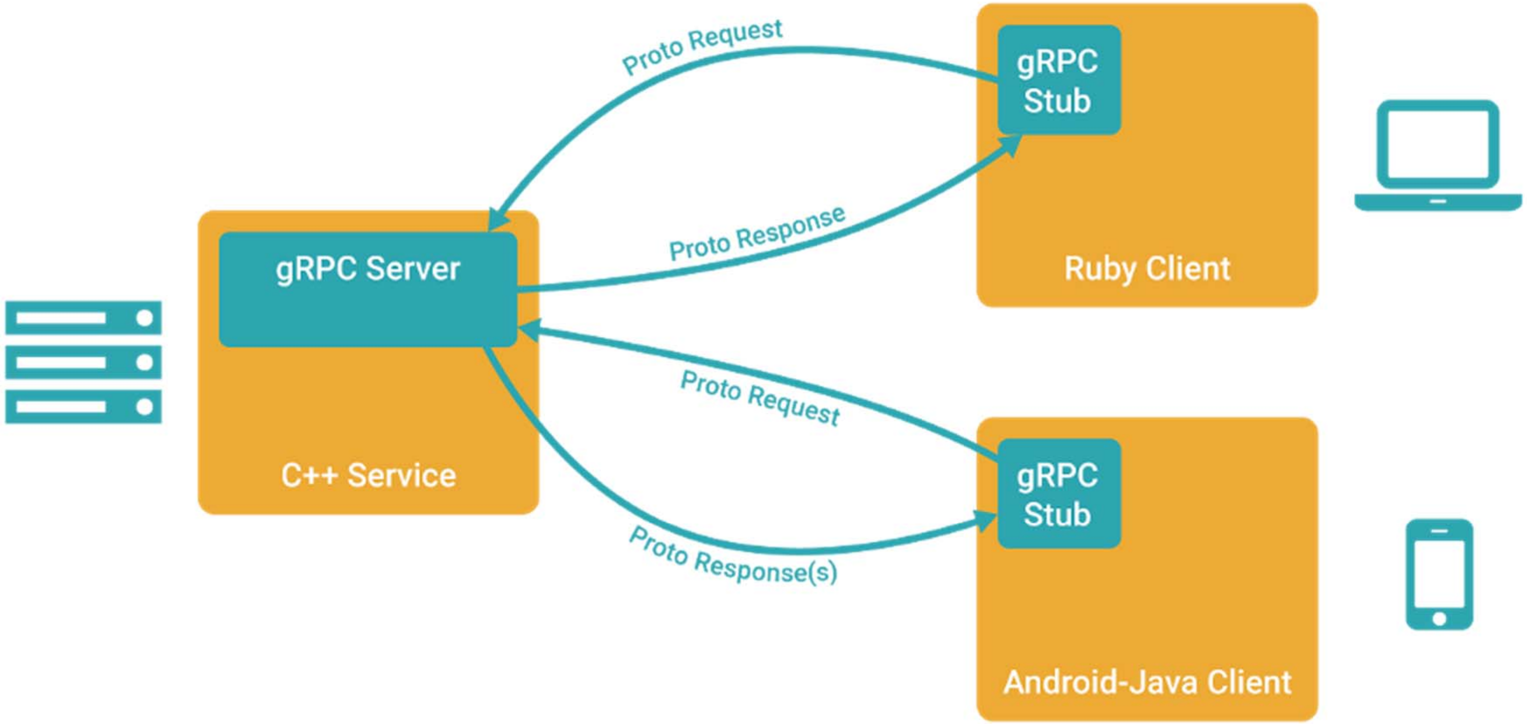
- We have learned tapi-topology and tapi-connectivity, but there are other significant models:
  - Notifications
  - Path Computation
  - Virtual Network
  - OAM
  - Technological augments:
    - Eth
    - ODU
    - OTSI

GRPC

What is gRPC

- gRPC stands for gRPC Remote Procedure Calls
- A high performance, general purpose, feature-rich RPC framework
- Part of Cloud Native Computing Foundation
- HTTP/2 and mobile first
- Open sourced version of Stubby RPC used in Google

gRPC architecture



Source: https://grpc.io/

Protocol Buffers

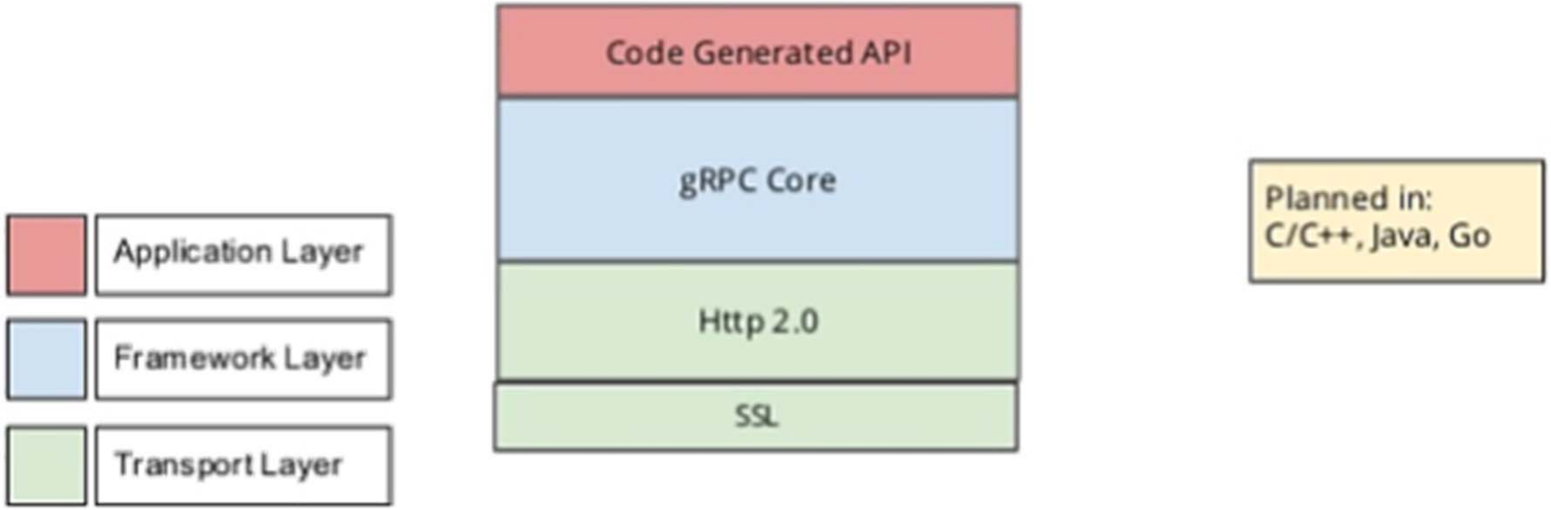
- Interface Definition Language (IDL)
  - Describe once and generate interfaces for any language.
- Data Model
  - Structure of the request and response.
- Wire format
  - Binary format for network transmission.
  - No more parsing text!
  - Compression
  - Streaming
- Compilation:

```
$ protoc -I=. –python_out=out_dir/ example.proto
```

```
syntax = "proto3";
option java_multiple_files = true;
option java_package = "com.grpc.search";
option java_outer_classname = "SearchProto";
option objc_class_prefix = "GGL";
package search;

service Google {
  // Search returns a Search Engine result for the query.
  rpc Search(Request) returns (Result) {}
}
message Request {
  string query = 1;
}
message Result {
  string title = 1;
  string url = 2;
  string snippet = 3;
}
```

gRPC Main Use Cases and architecture

- Efficiently connecting polyglot services in microservices style architecture
- Connecting mobile devices, browser clients to backend services
- Generating efficient client libraries
- Low latency, highly scalable, distributed systems.




```
$ pip3 install grpcio-tools googleapis-common-protos
$ apt install protobuf-compiler
$ python -m grpc_tools.protoc -I. –python_out=. --grpc_python_out=. example.proto
```

Usage of protobufs

- Translate connection.yang to protobuf
- Create a script that writes new connections to a file
- Create a script that lists all stored connections from a file
- You can use the following tutorial

https://developers.google.com/protocol-buffers/docs/pythontutorial

- Warning: Be “careful” with hyphens!

connection.proto

```
//Example of connection
syntax = "proto3";
package connection;

message Connection {
  string connectionId = 1;
  string sourceNode = 2;
  string targetNode = 3;
  string sourcePort = 4;
  string targetPort = 5;
  uint32 bandwidth = 6;

  enum LayerProtocolName {
    ETH = 0;
    OPTICAL = 1;
  }

  LayerProtocolName layerProtocolName = 7;

}

message ConnectionList {
  repeated Connection connection = 1;
}
```

```
$ cd /root/OFC_SC472/grpc
$ python -m grpc_tools.protoc -I=. –python_out=connection/
connection.proto
```

Create Connection

```
#! /usr/bin/env python3
import connection_pb2
import sys

def PromptForConnection(connection):
  connection.connectionId = raw_input("Enter connectionID: ")
  connection.sourceNode = raw_input("Enter sourceNode: ")
  connection.targetNode = raw_input("Enter targetNode: ")
  connection.sourcePort = raw_input("Enter sourcePort: ")
  connection.targetPort = raw_input("Enter targetPort: ")
  connection.bandwidth = int( raw_input("Enter bandwidth: ") )
  type = raw_input("Is this a eth or optical connection? ")
  if type == "eth":
    connection.layerProtocolName = connection_pb2.Connection.ETH
  elif type == "optical":
    connection.layerProtocolName = connection_pb2.Connection.OPTICAL
  else:
    print("Unknown layerProtocolName type; leaving as default value.")

...
```

```
...
if __name__ == '__main__':
  if len(sys.argv) != 2:
    print("Usage:", sys.argv[0], "CONNECTION_FILE")
    sys.exit(-1)

  connectionList = connection_pb2.ConnectionList()

  # Read the existing address book.
  try:
    with open(sys.argv[1], "rb") as f:
      connectionList.ParseFromString(f.read())
  except IOError:
    print(sys.argv[1] + ": File not found.  Creating a new file.")

  # Add an address.
  PromptForConnection(connectionList.connection.add())

  # Write the new address book back to disk.
  with open(sys.argv[1], "wb") as f:
    f.write(connectionList.SerializeToString())
```

```
$ cd /root/OFC_SC472/grpc/connection
$ python3 create.py connection.txt
```

List Connection

```
#! /usr/bin/env python3
from __future__ import print_function
import connection_pb2
import sys


# Iterates though all connections in the ConnectionList and
prints info about them.
def ListConnections(connectionList):
  for connection in connectionList.connection:
    print("connectionID:", connection.connectionId)
    print("  sourceNode:", connection.sourceNode)
    print("  targetNode:", connection.targetNode)
    print("  sourcePort:", connection.sourcePort)
    print("  targetPort:", connection.targetPort)
    print("  bandwidth:", connection.bandwidth)
    if connection.layerProtocolName ==
connection_pb2.Connection.ETH:
      print("  layerProtocolName:ETH")
    elif connection.layerProtocolName ==
connection_pb2.Connection.OPTICAL:
      print("  layerProtocolName:OPTICAL")
...
```

```
...
if __name__ == '__main__':
  if len(sys.argv) != 2:
    print("Usage:", sys.argv[0], "CONNECTION_FILE")
    sys.exit(-1)

  connectionList = connection_pb2.ConnectionList()

  # Read the existing address book.
  with open(sys.argv[1], "rb") as f:
    connectionList.ParseFromString(f.read())

  ListConnections(connectionList)
```

```
$ cd /root/OFC_SC472/grpc/connection
$ python3 list.py connection.txt
```

Create a gRPC client/server

- Example tutorial

https://grpc.io/docs/tutorials/basic/python.html

- Extend connection.proto to connectionService.proto with following service:

```
service ConnectionService {
  rpc CreateConnection (Connection) returns (google.protobuf.Empty) {}
  rpc ListConnection (google.protobuf.Empty) returns (ConnectionList) {}
}
```

```
$ cd /root/OFC_SC472/grpc
$ python -m grpc_tools.protoc -I=. --python_out=connectionService/ –grpc_python_out=connectionService/
connectionService.proto
```

connectionService_server.py

```
from concurrent import futures
import time
import logging
import grpc

import connectionService_pb2
import connectionService_pb2_grpc
from google.protobuf import empty_pb2 as google_dot_protobuf_dot_empty__pb2

_ONE_DAY_IN_SECONDS = 60 * 60 * 24

class connectionService(connectionService_pb2_grpc.ConnectionServiceServicer):
    def __init__(self):
        self.connectionList = connectionService_pb2.ConnectionList()

    def CreateConnection(self, request, context):
        logging.debug("Received Connection " + request.connectionId)
        self.connectionList.connection.extend([request])
        return google_dot_protobuf_dot_empty__pb2.Empty()

    def ListConnection(self, request, context):
        logging.debug("List Connections")
        return self.connectionList

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    connectionService_pb2_grpc.add_ConnectionServiceServicer_to_server(connectionService(), server)
    server.add_insecure_port('[::]:50051')
    logging.debug("Starting server")
    server.start()
    try:
        while True:
            time.sleep(_ONE_DAY_IN_SECONDS)
    except KeyboardInterrupt:
        server.stop(0)

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    serve()
```

connectionService_client.py

```
from __future__ import print_function
import grpc

import connectionService_pb2
import connectionService_pb2_grpc
from google.protobuf import empty_pb2 as google_dot_protobuf_dot_empty__pb2

def createConnection():
    with grpc.insecure_channel('localhost:50051') as channel:
        connection=connectionService_pb2.Connection()
        connection.connectionId = raw_input("Enter connectionID: ")
        connection.sourceNode = raw_input("Enter sourceNode: ")
        connection.targetNode = raw_input("Enter targetNode: ")
        connection.sourcePort = raw_input("Enter sourcePort: ")
        connection.targetPort = raw_input("Enter targetPort: ")
        connection.bandwidth = int( raw_input("Enter bandwidth: ") )
        stub = connectionService_pb2_grpc.ConnectionServiceStub(channel)
        response = stub.CreateConnection(connection)
    print("ConnectionService client received: " + str(response) )

def listConnection():
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = connectionService_pb2_grpc.ConnectionServiceStub(channel)
        response = stub.ListConnection(google_dot_protobuf_dot_empty__pb2.Empty())
    print("ConnectionService client received: " + str(response) )

if __name__ == '__main__':
    createConnection()
    listConnection()
```

Run example

- Run Server

```
$ cd /root/OFC_SC472/grpc/connectionService
$ python3 connectionService_server.py
```

- Run client

```
$ cd /root/OFC_SC472/grpc/connectionService
$ python3 connectionService_client.py
```

gRPC streams

- Create a new function in our Service to return the BER of a connection every 5 seconds.

- Use:

```
rpc GetBer(Connection) returns (stream Ber) {}
```

```
$ cd /root/OFC_SC472/grpc/
$ python -m grpc_tools.protoc -I=. –python_out=connectionServiceWithNotif/ --grpc_python_out=connectionServiceWithNotif/
connectionServiceWithNotif.proto
```

Solution

- Server

```
def GetBer (self, request, context):
    logging.debug("Get Ber")
    while True:
      time.sleep(5)
      ber=connectionServiceWithNotif_pb2.Ber(value=10)
      yield ber
```

```
RUN SERVER
$ cd /root/OFC_SC472/grpc/connectionServiceWithNotif
$ python3 connectionServiceWithNotif_server.py
```

- Client

```
def getBer(stub):
  responses = stub.GetBer(connectionServiceWithNotif_pb2.Connection(connectionId="conn1"))
  for response in responses:
    print("Received Ber %s" % (response.value) )
```

```
RUN CLIENT (in another window)
$ cd /root/OFC_SC472/grpc/connectionServiceWithNotif
$ python3 connectionServiceWithNotif_client.py
```

OPENCONFIG AND GNMI

OpenConfig Projects

OPENCONFIG

Data models

Models for common configuration and operational state across platforms

Streaming telemetry

Scalable, secure, real-time monitoring with modern streaming protocols

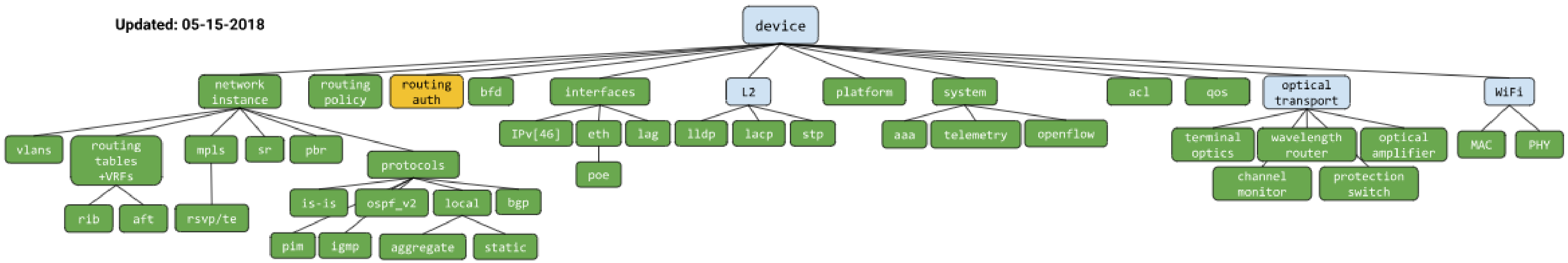
RPCs and tools

Management RPC specs and implementations
Tooling to build config and monitoring stacks

OpenConfig

- Data models for configuration and operational state, written in YANG
- Initial focus: device data for switching, routing, and transport
- Development priorities driven by operator requirements
- Technical engagement with major vendors to deliver native implementations

Updated: 05-15-2018



OpenConfig Data Model Principles

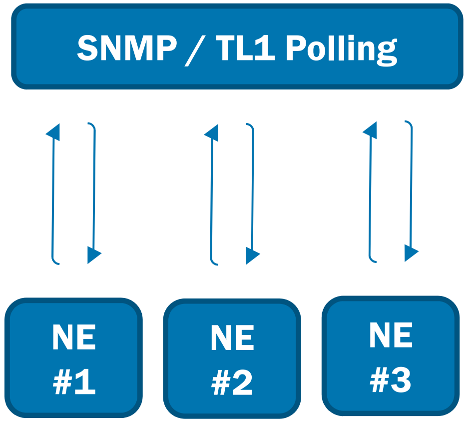
OPENCONFIG

- Modular model definition
- Model structure combines
  - Configuration (intended)
  - Operational data (applied config and derived state)
- Each module subtree declares config and state containers
- Model backward compatibility
  - Driven by use of semantic versioning (xx.yy.zz)
  - Diverges from IETF YANG guidelines (full compatibility)
- String patterns (regex) follow POSIX notation (instead of W3C as defined by IETF)

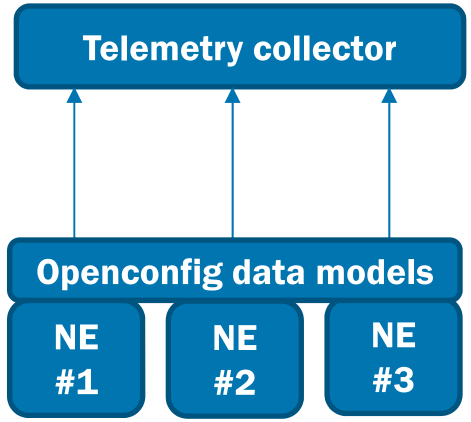
```
module: openconfig-bgp
tree-path /bgp/neighbors/neighbor/transport
    +--rw bgp!
       +--rw neighbors
          +--rw neighbor* [neighbor-address]
             +--rw transport
                +--rw config
                |  +--rw tcp-mss?
                |  +--rw mtu-discovery?
                |  +--rw passive-mode?
                |  +--rw local-address?
                +--ro state
                   +--ro tcp-mss?
                   +--ro mtu-discovery?
                   +--ro passive-mode?
                   +--ro local-address?
                   +--ro local-port?
                   +--ro remote-address?
                   +--ro remote-port?
```

Better visibility with streaming telemetry

- Operational state monitoring is crucial for network health and traffic management. Examples:
  - Counters, power levels, protocol stats, up/down events, inventory, alarms



- O(min) polling
- Resource drain on devices
- Legacy implementation
- Inflexible structure



- Subscribe to desired data based on models
- Streamed directly from devices
- Time-series or event-driven data
- Modern, secure transport

RPCs and gNMI

OPENCONFIG

- gNMI is a protocol for the modification and retrieval of configuration from a target device, as well as the control and generation of telemetry streams from a target device to a data collection system.

https://github.com/openconfig/gnmi

- This gNMI is described using Protobuf:

https://github.com/openconfig/gnmi/blob/master/proto/gnmi/gnmi.proto

- The data can be either enconded in JSON or in Protobuf (Currently in JSON).

Why gNMI?

- provides a single service for state management (streaming telemetry and configuration)
- built on a modern standard, secure transport and open RPC framework with many language bindings
- supports very efficient serialization and data access
  - 3x-10x smaller than XML
- offers an implemented alternative to NETCONF, RESTCONF, …
  - early-release implementations on multiple router and transport platforms
  - reference tools published by OpenConfig

https://datatracker.ietf.org/meeting/98/materials/slides-98-rtgwg-gnmi-intro-draft-openconfig-rtgwg-gnmi-spec-00

gNMI Terminology

- *Telemetry* - refers to streaming data relating to underlying characteristics of the device - either operational state or configuration.
- *Configuration* - elements within the data schema which are read/write and can be manipulated by the client.
- *Target* - the device within the protocol which acts as the owner of the data that is being manipulated or reported on. Typically this will be a network device.
- *Client* - the device or system using the protocol described in this document to query/modify data on the target, or act as a collector for streamed data. Typically this will be a network management system.

gNMI protocol buffer

OPENCONFIG

```
service gNMI {
  rpc Capabilities(CapabilityRequest) returns (CapabilityResponse);
  rpc Get(GetRequest) returns (GetResponse);
  rpc Set(SetRequest) returns (SetResponse);
  rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse);
}
```

```
message GetRequest {
  Path prefix = 1;
  repeated Path path = 2;
  enum DataType {
    ALL = 0;
    CONFIG = 1;
    STATE = 2;
    OPERATIONAL = 3;
  }
  DataType type = 3;
  Encoding encoding = 5;
  repeated ModelData use_models = 6;
  repeated gnmi_ext.Extension extension = 7;
}

message GetResponse {
  repeated Notification notification = 1;
  Error error = 2 [deprecated=true];
  repeated gnmi_ext.Extension extension = 3;
}
```

```
message CapabilityRequest {
  repeated gnmi_ext.Extension extension = 1;
}

message CapabilityResponse {
  repeated ModelData supported_models = 1;
  repeated Encoding supported_encodings = 2;
  string gNMI_version = 3;
  repeated gnmi_ext.Extension extension = 4;
}

message ModelData {
  string name = 1;
  string organization = 2;
  string version = 3;
}
```

gNMI target (server) with topology.yang

- gNxI is A collection of tools for Network Management that use the gNMI and gNOI protocols.

- Set-up server for Capabilities, Set/Get operations based on gNxI:

https://github.com/google/gnxi

- Start at go directory:

```
$ cd /usr/share/gocode/src/
$ export GOPATH=/usr/share/gocode/
```

- Compile modeldata:

```
$ go run github.com/openconfig/ygot/generator/generator.go
   -generate_fakeroot
    -output_file github.com/google/gnxi/gnmi/modeldata/gostruct/generated.go
   -package_name gostruct github.com/rvilalta/OFC_SC472/yang/topology.yang
```

gNMI target with topology.yang

- Write modeldata Package

/usr/share/gocode/src/github.com/google/gnxi/gnmi/modeldata/modeldata.go:

```go
package modeldata

import (
	pb "github.com/openconfig/gnmi/proto/gnmi"
)

const (
	TopologyModel = "topology"
)

var (
	// ModelData is a list of supported models.
	ModelData = []*pb.ModelData{{
		Name :       TopologyModel,
		Organization: "CTTC",
		Version:     "0.0.0",
	},

}
)
```

topology.json

```json
{
  "topology" : {
    "node" : [
      { "node-id" : "A" , "port" : [ { "port-id" : "portA1" } ] },
      { "node-id" : "B" , "port" : [ { "port-id" : "portB1" } ] }
    ]
  }
}
```

- Run target:

```
$ cd /usr/share/gocode/src/github.com/google/gnxi/gnmi_target
$ go run gnmi_target.go -bind_address :10161 -config /root/OFC_SC472/gnmi/topology.json
--notls -alsologtostderr
```

Get Request with gNMI client

- In another window, go to get client directory and run:

```
$ export GOPATH=/usr/share/gocode/
$ cd /usr/share/gocode/src/github.com/google/gnxi/gnmi_get
$ go run gnmi_get.go -notls -xpath "/topology/" -target_addr localhost:10161 -alsologtostderr
```

- Run with querry:

```
$ go run gnmi_get.go -notls -xpath "/topology/node[node-id=A]" -target_addr localhost:10161 -alsologtostderr
```

- Also python gNMI client available:

```
$ cd /usr/share/gocode/src/github.com/google/gnxi/gnmi_cli_py
$ python py_gnmicli.py -n -m get -t localhost -p 10161 -x /topology -u foo -pass bar
```

Wireshark of gNMI

```
82956 5800.9405448… 127.0.0.1    127.0.0.1    HTTP2   171 SETTINGS[0], HEADERS[1]: POST /gnmi.gNMI/Get
82957 5800.9410133… 127.0.0.1    127.0.0.1    GRPC    120 DATA[1] (GRPC) (PROTOBUF)
82958 5800.9410575… 127.0.0.1    127.0.0.1    TCP      68 10161 → 42138 [ACK] Seq=19 Ack=189 Win=65536 Len=0 TSval=2474643897 TSecr=2474643896
82959 5800.9417108… 127.0.0.1    127.0.0.1    GRPC    285 WINDOW_UPDATE[0], PING[0], HEADERS[1]: 200 OK, DATA[1], HEADERS[1] (GRPC) (PROTOBUF)
82960 5800.9425272… 127.0.0.1    127.0.0.1    HTTP2    85 PING[0]
82961 5800.9441456… 127.0.0.1    127.0.0.1    HTTP2    98 WINDOW_UPDATE[0], PING[0]
```

```
› Frame 82957: 120 bytes on wire (960 bits), 120 bytes captured (960 bits) on interface 0
› Linux cooked capture
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
› Transmission Control Protocol, Src Port: 42138, Dst Port: 10161, Seq: 137, Ack: 19, Len: 52
▾ HyperText Transfer Protocol 2
  › Stream: DATA, Stream ID: 1, Length 43
  › GRPC Message: /gnmi.gNMI/Get, Request
  ▾ Protocol Buffers: application/grpc,/gnmi.gNMI/Get,request
    ▾ Field[2]
        .001 0... = Field Number: 2
        .... .010 = Wire Type: Length-delimited (2)
        Value Length: 34
        Value: 1a0a0a08746f706f6c6f67791a140a046e6f6465120c0a07...
    ▾ Field[5], 4 (uint32)
        .010 1... = Field Number: 5
        .... .000 = Wire Type: varint (0)
      ▾ Value: 04
          Uint32: 4

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 68 d2 0b 40 00  40 06 6a 82 7f 00 00 01   E··h··@· @·j·····
0020  7f 00 00 01 a4 9a 27 b1  25 1f d6 88 c0 40 c4 1a   ······'· %····@··
0030  80 18 02 00 fe 5c 00 00  01 01 08 0a 93 80 11 b9   ·····\·· ········
0040  93 80 11 b8 00 00 2b 00  01 00 00 00 01 00 00 00   ······+· ········
0050  00 26 12 22 1a 0a 0a 08  74 6f 70 6f 6c 6f 67 79   ·&·"···· topology
0060  1a 14 0a 04 6e 6f 64 65  12 0c 0a 07 6e 6f 64 65   ····node ····node
0070  2d 69 64 12 01 41 28 04                            -id··A(·
```

```
› Frame 82959: 285 bytes on wire (2280 bits), 285 bytes captured (2280 bits) on interface 0
› Linux cooked capture
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
› Transmission Control Protocol, Src Port: 10161, Dst Port: 42138, Seq: 19, Ack: 189, Len: 217
▾ HyperText Transfer Protocol 2
  › Stream: WINDOW_UPDATE, Stream ID: 0, Length 4
  › Stream: PING, Stream ID: 0, Length 8
  › Stream: HEADERS, Stream ID: 1, Length 14, 200 OK
  › Stream: DATA, Stream ID: 1, Length 122 (partial entity body)
  › Stream: HEADERS, Stream ID: 1, Length 24
  › GRPC Message: /gnmi.gNMI/Get, Response
  ▾ Protocol Buffers: application/grpc,/gnmi.gNMI/Get,response
    ▾ Field[1]
        .000 1... = Field Number: 1
        .... .010 = Wire Type: Length-delimited (2)
        Value Length: 115
        Value: 08d0d69490afbeeadf1522670a221a0a0a08746f706f6c6f...

0000  00 00 00 00 75 0a 73 08  d0 d6 94 90 af be ea df   ····u·s· ········
0010  15 22 67 0a 22 1a 0a 0a  08 74 6f 70 6f 6c 6f 67   ·"g·"··· ·topolog
0020  79 1a 14 0a 04 6e 6f 64  65 12 0c 0a 07 6e 6f 64   y····nod e····nod
0030  65 2d 69 64 12 01 41 1a  41 5a 3f 7b 22 74 6f 70   e-id··A· AZ?{"top
0040  6f 6c 6f 67 79 3a 6e 6f  64 65 2d 69 64 22 3a 22   ology:no de-id":"
0050  41 22 2c 22 74 6f 70 6f  6c 6f 67 79 3a 70 6f 72   A","topo logy:por
0060  74 22 3a 5b 7b 22 70 6f  72 74 2d 69 64 22 3a 22   t":[{"po rt-id":"
0070  70 6f 72 74 41 31 22 7d  5d 7d                     portA1"} ]}
```

CONCLUSION

We are ready for Control and monitoring of Networks

- Motivation
- YANG Data Modelling Language
  - Exercise: Modelling a network
  - Exercise: Using pyang and its plugins
  - Exercise: Pyangbind to write code in python
- Netconf
  - Understanding Netconf protocol
  - Use Confd as a Netconf Server
  - Create a Netconf Client
  - Create a Netconf Server with basic commands
- RESTconf
  - Understanding RESTconf protocol
  - Generate topology/connection OpenAPI
  - Generate connection Server Stub

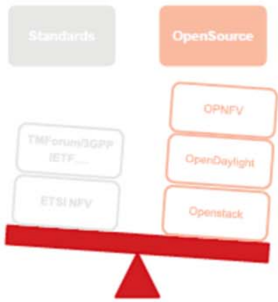
We are ready for Control and monitoring of Networks II

- Using ONOS with RESTconf
  - Introduction to ONOS northbound REST API, Mininet config
  - ONOS client (topology & flows)
- ONF Transport API
  - Understanding TAPI model
  - TAPI Topology client
- gRPC
  - Understanding gRPC and Protocol Buffers
  - Usage of protobufs
  - Create a gRPC client/server
  - gRPC streams
- OpenConfig
  - Data Model Principles
  - RPCs and gNMI

Standards vs Open Source

Standards Outcome/Benefits

- Reference architectures
- Functional requirements
- Interface requirements
- Information models
- Data Models
- Protocols



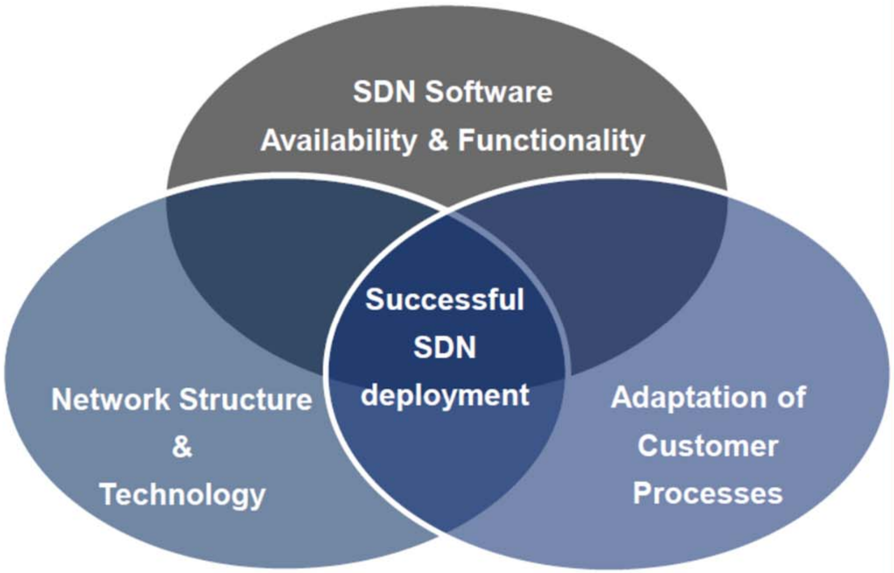
Open Source Outcome/Benefit

- Source code
- Reference implementation
- De-Facto API's
- Inter-operability testing
- Explore new features
- It's all about the community
- Upstream vs Integration

PoC Outcome/Benefits

- Technology Exploration
- Feasibility Check
- Create knowhow
- PR

Transport SDN Benefits and Challenges



- **Benefit**: Completely automated, programmable, integrated and flexible network – leveraging the installed base in an optimized manner.
- **Technical Challenges**:
  - agree on standardized architectures and abstraction/ virtualization models
  - performance of centralized systems & OF
- **Commercialization Challenges**:
  - Open Source business models
  - New business models leveraging SDN
- **Organizational Challenges**:
  - Adapt deep rooted processes across traditional silos & boundaries to leverage SDN flexibility
- **Deployment Challenges**:
  - Carrier grade SDN systems for field deployments
  - Maturity of SDN network technologies for green field deployments as well as integration of legacy networks

References

- RFC6020, YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF), https://tools.ietf.org/html/rfc6020
- RFC6241, Network Configuration Protocol (NETCONF), https://tools.ietf.org/html/rfc6241
- Open ROADM Overview, https://0201.nccdn.net/1_2/000/000/098/a85/Open-ROADM-whitepaper-v1-0.pdf
- RFC8040, RESTCONF Protocol, https://tools.ietf.org/html/rfc8040
- Transport API (TAPI) 2.0 Overview, https://www.opennetworking.org/wp-content/uploads/2017/08/TAPI-2-WP_DRAFT.pdf
- gRPC Basics – Python, https://grpc.io/docs/tutorials/basic/python.html
- OpenConfig FAQ for operators, http://www.openconfig.net/docs/faq-for-operators/

- This SC contains slides from previous OFC 2018 SC449: Hands-on: An introduction to Writing Transport SDN Applications by Ricard Vilalta (CTTC) and Karthik Sethuraman/Yuta Higuchi (NEC) and OFC 2018 SC448: Software Defined Networking for Optical Networks: a Practical Introduction by Ramon Casellas (CTTC).

ricard.vilalta@cttc.es