Software-Defined Networking and Advanced Network Control Programming

3 Control Plane

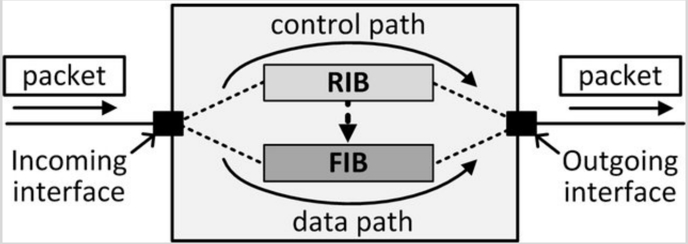
Kai Gao
kaigao@scu.edu.cn

School of Cyber Science and Engineering
Sichuan University

Recap

Network Data Plane

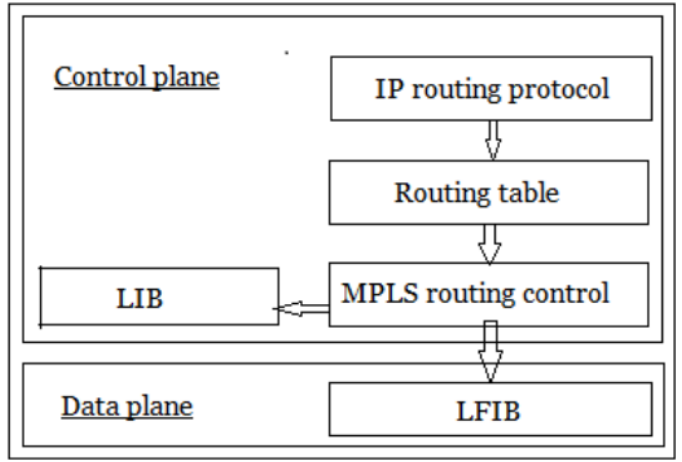
- IP: FIB



campista22challenges

Network Data Plane

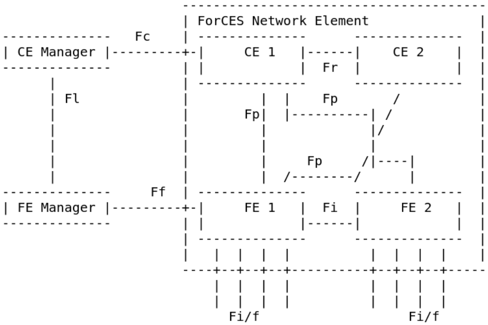
- IP: FIB
- MPLS: LFIB



bhandure2013comparative

Network Data Plane

- IP: FIB
- MPLS: LFIB
- ForCES: FE



Fp: CE-FE interface
Fi: FE-FE interface
Fr: CE-CE interface
Fc: Interface between the CE manager and a CE
Ff: Interface between the FE manager and an FE
Fl: Interface between the CE manager and the FE manager
Fi/f: FE external interface

Figure 1: ForCES Architectural Diagram

rfc5810

Match-Action Paradigm

Architecture	*Match*	*Action*	*Southbound*
IP	destination IP address	forward to egress port, etc.	FIB
MPLS	label	modify label, forward to egress port, etc.	LFIB
ForCES	depending on logical function type	forwarding, QoS, filtering, tunnel, sampling, etc.	ForCES protocol
OpenFlow	multiple protocol header fields	forwarding, QoS, filtering, modify header, etc.	OpenFlow protocol
PoF	header segments & flow metadata	forwarding, simple math, modify header/metadata	OpenFlow
P4	customizable header fields & metadata	customizable actions based on forwarding, simple math, modify header/metadata	P4 runtime

Hardware for Data Plane

Two types of memory (used to realize look-up tables):

- Content Addressable Memory (CAM): used to realize the “look-up” operation
- Random Access Memory (RAM): used to store state (meta data, actions, etc.)

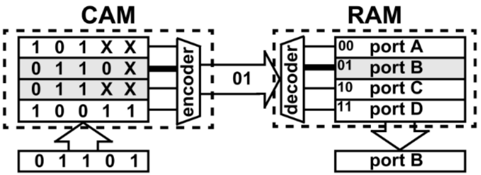
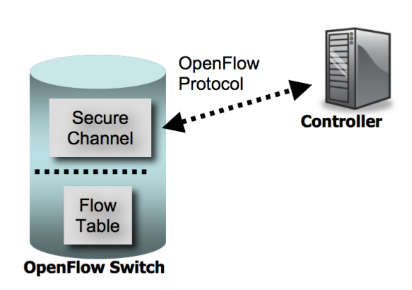


Fig. 3. CAM-based implementation of the routing table of Table I.

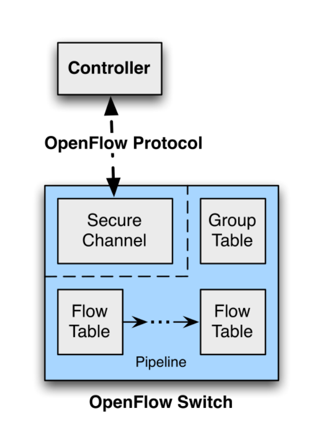
pagiamtzis2006contentaddressable

Components

In OF 1.0.0, a switch has a single flow table (BOTTOM) and multiple flow tables (maximum 256) since OF 1.1.0 (LEFT).



onf2009openflow



onf2011openflow

OpenFlow Tables

Flow table

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Table 1: Main components of a flow entry in a flow table.

OpenFlow Tables

Flow table

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Table 1: Main components of a flow entry in a flow table.

Group table

Group Identifier	Group Type	Counters	Action Buckets

Table 2: Main components of a group entry in the group table.

OpenFlow Tables

Flow table

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Table 1: Main components of a flow entry in a flow table.

Group table

Group Identifier	Group Type	Counters	Action Buckets

Table 2: Main components of a group entry in the group table.

Meter table

Meter Identifier	Meter Bands	Counters

Table 3: Main components of a meter entry in the meter table.

Band Type	Rate	Burst	Counters	Type specific arguments

Table 4: Main components of a meter band in a meter entry.

onf2015openflow

OpenFlow Messages

OpenFlow switches exchange information with the controller with the OpenFlow messages.

Common controller-to-switch messages include:

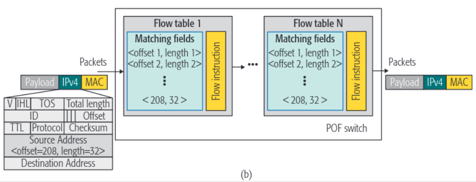
- **Packet-out**: Encapsulate a packet in the message and send to a switch
- **Flow-mod**: Insert/Update/Delete a flow rule in a flow table
- **Barrier**: Setting barriers for messages (to be explained later)

Common switch-to-controller messages include:

- **Packet-in**: Encapsulate a packet in the message and send to the controller
- **Flow-removed**: Notify the controller that a flow rule is removed (because of timetout)
- **Port-status**: Notify the controller that the status of a port has changed (up to down or down to up)

Other SDN Data Plane

Protocol oblivious forwarding



---

li2017protocol

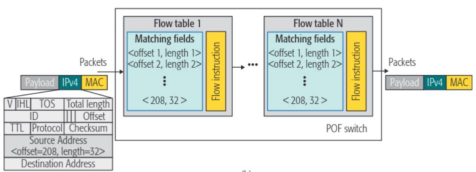
Other SDN Data Plane

Protocol oblivious forwarding

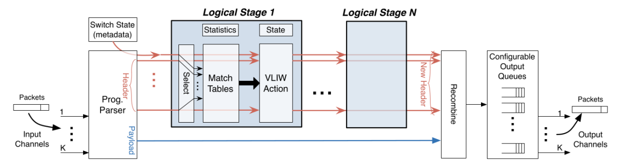
li2017protocol

Reconfigurable Match-action Table (RMT)

bosshart2013forwarding



(b)

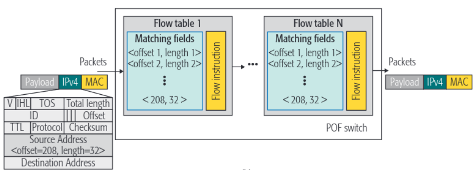


(a) RMT model as a sequence of logical Match-Action stages.

Other SDN Data Plane

Protocol oblivious forwarding

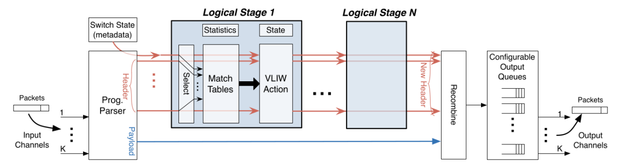
li2017protocol



(b)

Reconfigurable Match-action Table (RMT)

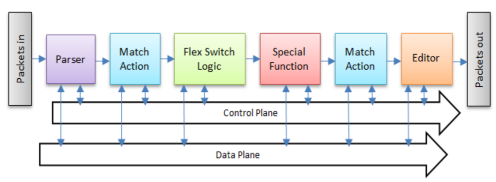
bosshart2013forwarding



(a) RMT model as a sequence of logical Match-Action stages.

Broadcom Programmable Switching ASIC

broadcom2019npl



本期学习目标

- 有哪些具有代表性的 SDN 控制器？有哪些常见的 SDN 控制器架构？

本期学习目标

- 有哪些具有代表性的 SDN 控制器？有哪些常见的 SDN 控制器架构？
- 什么是基于意图的网络控制？有哪些代表性的意图？

本期学习目标

- 有哪些具有代表性的 SDN 控制器？有哪些常见的 SDN 控制器架构？
- 什么是基于意图的网络控制？有哪些代表性的意图？
- 什么是模型驱动的网络？

本期学习目标

- 有哪些具有代表性的 SDN 控制器？有哪些常见的 SDN 控制器架构？
- 什么是基于意图的网络控制？有哪些代表性的意图？
- 什么是模型驱动的网络？
- 如何使用 YANG 数据建模语言定义一个基本数据模型？

本期学习目标

- 有哪些具有代表性的 SDN 控制器？有哪些常见的 SDN 控制器架构？
- 什么是基于意图的网络控制？有哪些代表性的意图？
- 什么是模型驱动的网络？
- 如何使用 YANG 数据建模语言定义一个基本数据模型？
- 如何使用 YANG 数据建模语言扩展一个数据模型？

SDN Control Plane

Summary

In this lecture, we cover the following topics:

- Representative SDN controllers
- Intent-based networking and model-driven networking
- YANG language

Summary

In this lecture, we cover the following topics:

- Representative SDN controllers
- Intent-based networking and model-driven networking
- YANG language

You should

- know the representative SDN controllers and their design choices

Summary

In this lecture, we cover the following topics:

- Representative SDN controllers
- Intent-based networking and model-driven networking
- YANG language

You should

- know the representative SDN controllers and their design choices
- understand the motivations for intent-based networking and model-based networking

Summary

In this lecture, we cover the following topics:

- Representative SDN controllers
- Intent-based networking and model-driven networking
- YANG language

You should

- know the representative SDN controllers and their design choices
- understand the motivations for intent-based networking and model-based networking
- roughly understand what YANG language does and how data trees are constructed

NOX

NOX (2008) is the first SDN controller (SDN 控制器) and propose the idea of network operating system (网络操作系统) in the context of SDN.

> *In the past, the term network operating system referred to operating systems that incorporated networking (e.g., Novell NetWare), but this usage is now obsolete. We are resurrecting the term to denote systems that provide an execution environment for programmatic control of the network.*

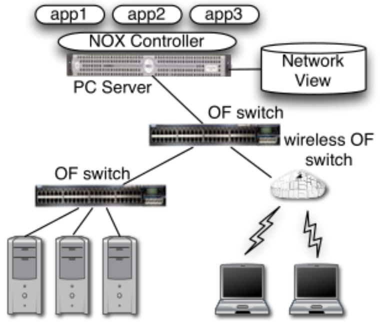


Figure 1: Components of a NOX-based network: OpenFlow (OF) switches, a server running a NOX controller process and a database containing the network view.

gude2008nox

NOX Architecture and Interfaces

System Architecture of NOX:

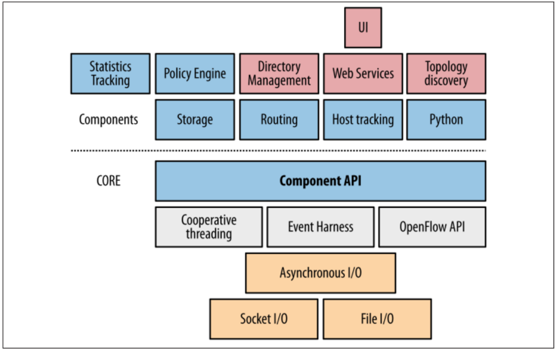


Figure 4-10. NOX architecture

nadeau2013sdn

Ad-hoc native interfaces:

```
# On user authentication, statically setup VLAN tagging
# rules at the user's first hop switch
def setup_user_vlan(dp, user, port, host):
    vlanid = user_to_vlan_function(user)
    # For packets from the user, add a VLAN tag
    attr_out[IN_PORT] = port
    attr_out[DL_SRC] = nox.reverse_resolve(host).mac
    action_out = [(nox.OUTPUT, (0, nox.FLOOD)),
                  (nox.ADD_VLAN, (vlanid))]
    install_datapath_flow(dp, attr_out, action_out)
    # For packets to the user with the VLAN tag, remove it
    attr_in[DL_DST] = nox.reverse_resolve(host).mac
    attr_in[DL_VLAN] = vlanid
    action_in = [(nox.OUTPUT, (0, nox.FLOOD)),
                 (nox.DEL_VLAN)]
    install_datapath_flow(dp, attr_in, action_in)
nox.register_for_user_authentication(setup_user_vlan)
```

Figure 2: An example NOX application written in Python that statically sets VLAN tagging rules on user authentication. A complete application would also add VLAN removal rules at all end-point switches.

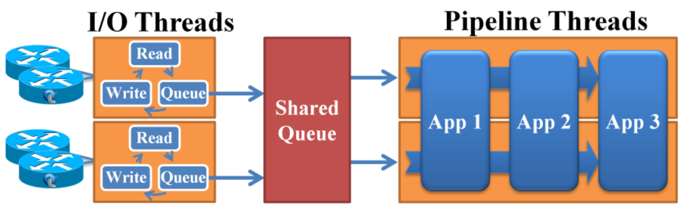
gude2008nox

Beacon

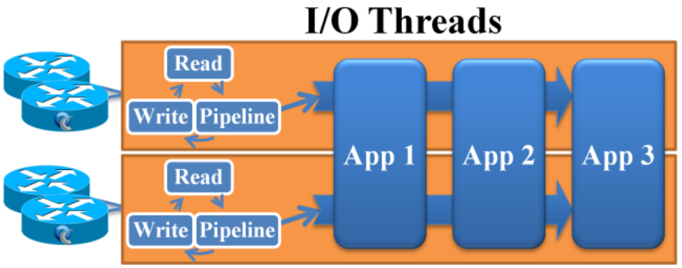
Beacon (2013) improves the controller performance with multi-thread optimizations (多线程优化). Other important features include dynamic service registry.

FloodLight, a widely used open source controller, is a fork of Beacon.

erickson2013beacon



(a) Shared Queue

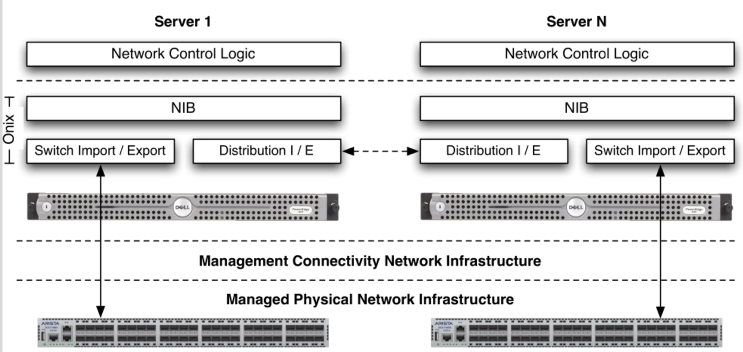


(b) Run-to-completion

Onix

Onix introduces the idea of distributed network information base (NIB, 网络信息表).

It borrows the idea in distributed storage (分布式存储) to allow customized level of consistency (一致性).



koponen2010onix

A Short Note on Consistency

Common consistency levels in networking:

- Strong consistency (强一致性)
- Sequential consistency (顺序一致性)
- Causal consistency (因果一致性)
- Eventual consistency (最终一致性)

Consistency (一致性), availability (可用性) and partition (可分区) are three important properties of a distributed system.

Kandoo

Kandoo is an *hierarchical (层次化)* SDN controller

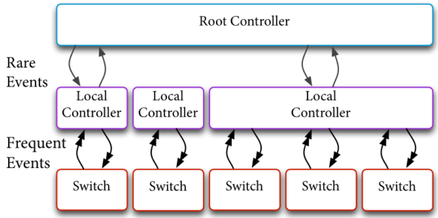


Figure 1: Kandoo's Two Levels of Controllers. Local controllers handle frequent events, while a logically centralized root controller handles rare events.

hassasyeganeh2012kandoo

ONOS

Open Networking Operating System (ONOS) is an open source distributed SDN controller focusing on high performance

It introduces intent-based networking (基于意图的网络)

Fun facts: Releases are named after birds: *woodpecker* is the latest release (2.6.0)

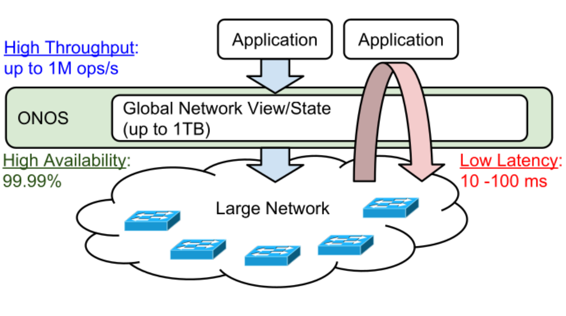


Figure 1: ONOS Requirements

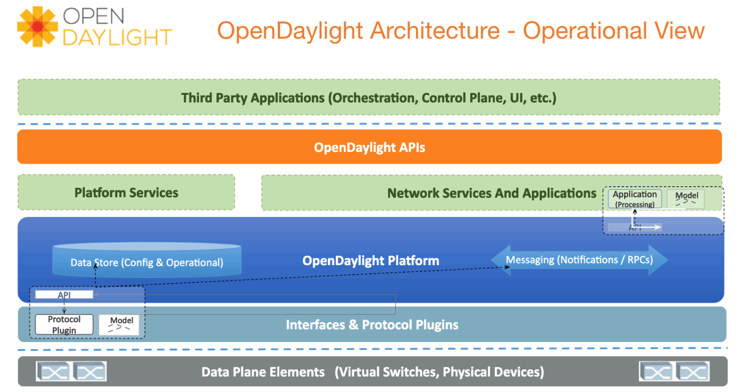
berde2014onos

Open Daylight

Open Daylight is an open source distributed SDN controller and platform widely used in industry and research

It introduces model-driven service abstraction layer (MD-SAL, 模型驱动服务抽象层).

Fun facts: Releases are named after chemical elements: *Silicon* (Si) is the latest release (14.0)



https://www.opendaylight.org/what-we-do/current-release/neon/attachment/opendaylight-architecture

Highlighted Techniques in SDN Controllers

- Centralized (集中式) v.s. distributed (分布式)
- Intent-based network management
- Model-driven network management

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

Distributed SDN controller: the controller software is running on multiple machines

√ Simple to deploy/develop

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

Distributed SDN controller: the controller software is running on multiple machines

- √ Simple to deploy/develop
- √ Fast for small-scale networks

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

Distributed SDN controller: the controller software is running on multiple machines

- √ Simple to deploy/develop
- √ Fast for small-scale networks
- × Not resilient to failures

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

- √ Simple to deploy/develop
- √ Fast for small-scale networks
- × Not resilient to failures
- × Poor scalability

Distributed SDN controller: the controller software is running on multiple machines

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

- √ Simple to deploy/develop
- √ Fast for small-scale networks
- × Not resilient to failures
- × Poor scalability

Distributed SDN controller: the controller software is running on multiple machines

- √ Resilient to controller instance failures

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

- ✓ Simple to deploy/develop
- ✓ Fast for small-scale networks
- × Not resilient to failures
- × Poor scalability

Distributed SDN controller: the controller software is running on multiple machines

- ✓ Resilient to controller instance failures
- ✓ Highly scalable to manage large networks

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

- √ Simple to deploy/develop
- √ Fast for small-scale networks
- × Not resilient to failures
- × Poor scalability

Distributed SDN controller: the controller software is running on multiple machines

- √ Resilient to controller instance failures
- √ Highly scalable to manage large networks
- × Relatively difficult to deploy/develop

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

- √ Simple to deploy/develop
- √ Fast for small-scale networks
- × Not resilient to failures
- × Poor scalability

Distributed SDN controller: the controller software is running on multiple machines

- √ Resilient to controller instance failures
- √ Highly scalable to manage large networks
- × Relatively difficult to deploy/develop
- × Additional cost on state synchronization and message processing latency

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

- √ Simple to deploy/develop
- √ Fast for small-scale networks
- × Not resilient to failures
- × Poor scalability

Distributed SDN controller: the controller software is running on multiple machines

- √ Resilient to controller instance failures
- √ Highly scalable to manage large networks
- × Relatively difficult to deploy/develop
- × Additional cost on state synchronization and message processing latency

Centralized v.s. Distributed

Centralized SDN controller: the controller software is running on a single machine

- √ Simple to deploy/develop
- √ Fast for small-scale networks
- × Not resilient to failures
- × Poor scalability

Distributed SDN controller: the controller software is running on multiple machines

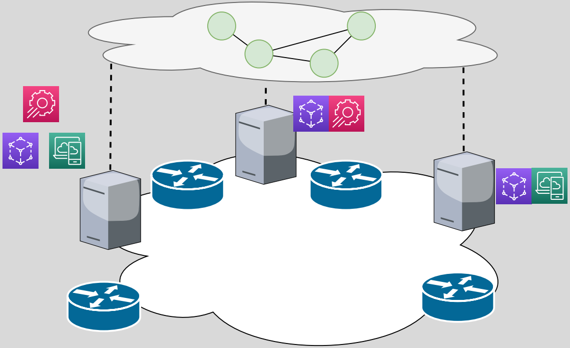
- √ Resilient to controller instance failures
- √ Highly scalable to manage large networks
- × Relatively difficult to deploy/develop
- × Additional cost on state synchronization and message processing latency

In production, SDN controllers are usually distributed.

Distributed Controller: Flat and Hierarchical

There are two types of distributed controller based on whether instances operate on the same view:

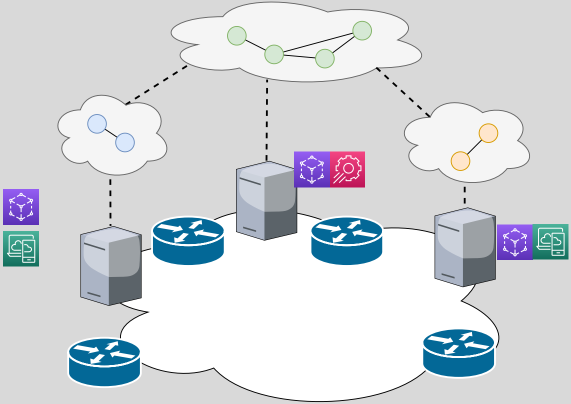
- Flat (扁平化): All controller instances operates on the same network view and scope (applications and services may be deployed on different instances)



Distributed Controller: Flat and Hierarchical

There are two types of distributed controller based on whether instances operate on the same view:

- Flat (扁平化): All controller instances operates on the same network view and scope (applications and services may be deployed on different instances)
- Hierarchical (层次化): Different controllers have different roles, network views and scopes



Different Hierarchical Architectures

There are different types of hierarchical SDN controller architectures:

- Instances of different levels have different functionalities. They *may or may not* have the same network view

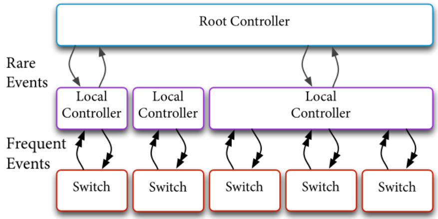


Figure 1: Kandoo's Two Levels of Controllers. Local controllers handle frequent events, while a logically centralized root controller handles rare events.

---

hassasyeganeh2012kandoo

Different Hierarchical Architectures

There are different types of hierarchical SDN controller architectures:

- Instances of different levels have different functionalities. They *may or may not* have the same network view
- Instances of different levels have the same functionality but have different levels of network views

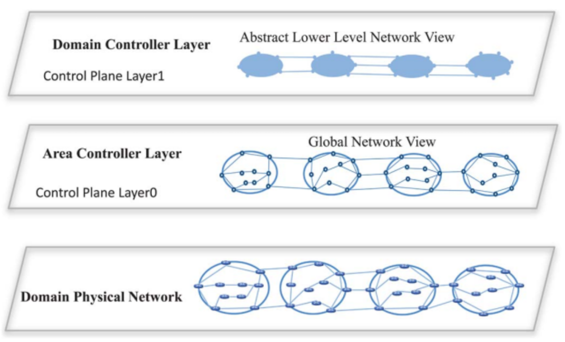


Fig. 2. Network view of the hybrid hierarchical architecture.

fu2015hybrid

Intent-based Networking

The idea of intents (意图) is borrowed from mobile systems (such as Android)

Users specify what they want to do instead of how to do it, and multiple service instances can co-exist to realize the intent.

Intents realize loosely coupled and dynamic service binding (动态服务绑定).

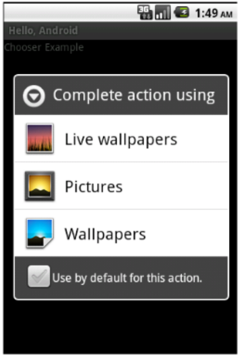


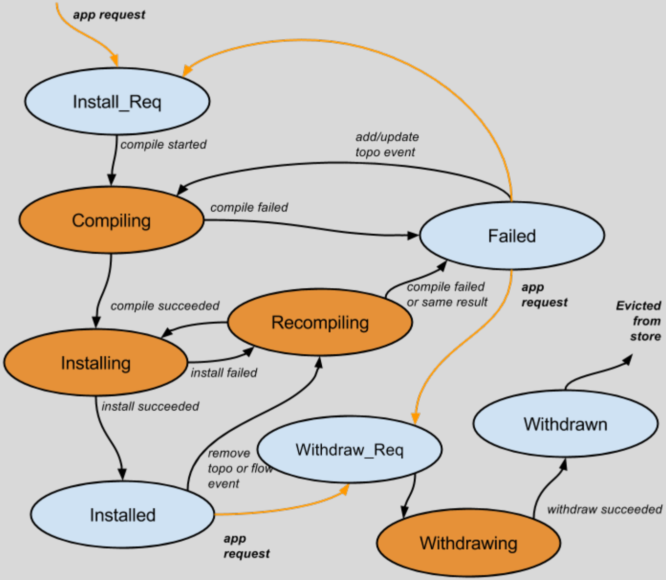
Figure 3: The user is prompted when an implicit Intent resolves to multiple Activities.

---

chin2011analyzing

Intent Life Cycle in ONOS

ONOS maintains a finite state machine (FSM, 有限状态机) to manage the life cycle of intents



https://wiki.onosproject.org/display/ONOS/Intent+Framework

Cascading Compilation

An intent is either directly installed or decomposed as multiple lower-level intents.

```
@Beta
public interface IntentCompiler<T extends Intent> {
    /**
     * Compiles the specified intent into other intents.
     *
     * @param intent      intent to be compiled
     * @param installable previous compilation result; optional
     * @return list of resulting intents
     * @throws IntentException if issues are encountered while compiling the intent
     */
    List<Intent> compile(T intent, List<Intent> installable);

}
```

https://github.com/opennetworkinglab/onos/tree/master/core/api/src/main/java/org/onosproject/net/intent/IntentCompiler.java

Cascading Compilation (cont.)

ONOS uses cascading compilation (级联编译): the root intent is compiled recursively until the compiled intents are installable.

The cascading compilation enforces the atomic property: sub installable intents originated from a single root intent are either all installed or not installed.

```java
List<Intent> installables = new ArrayList<>();
Queue<Intent> compileQueue = new LinkedList<>();
compileQueue.add(intent);

Intent compiling;
while ((compiling = compileQueue.poll()) != null) {
    registerSubclassCompilerIfNeeded(compiling);

    List<Intent> compiled = getCompiler(compiling)
                                .compile(compiling, previousInstallables);

    compiled.forEach(i -> {
        if (i.isInstallable()) {
            installables.add(i);
        } else {
            compileQueue.add(i);
        }
    });
}
return installables;
```

https://github.com/opennetworkinglab/onos/blob/master/core/net/src/main/java/org/onosproject/net/intent/impl/CompilerRegistry.java

Example

There are some pre-defined intents in ONOS

- `HostToHostIntent`: Set up bidirectional connection between two end hosts
- `PointToPointIntent`: Set up connectivity between two end points
- `PathIntent`: Set up the specified path
- `FlowRuleIntent`: Install an OpenFlow rule
- ...

https://github.com/opennetworkinglab/onos/tree/master/core/api/src/main/java/org/onosproject/net/intent

Compiler Example

The PathIntentCompiler compiles a PathIntent into a list of FlowRuleIntents.

```
@Override
public List<Intent> compile(PathIntent intent, List<Intent> installable) {

    List<FlowRule> rules = new LinkedList<>();
    List<DeviceId> devices = new LinkedList<>();
    compile(this, intent, rules, devices);


    return ImmutableList.of(new FlowRuleIntent(appId,
                                               intent.key(),
                                               rules,
                                               intent.resources(),
                                               intent.type(),
                                               intent.resourceGroup()
    ));
}
```

https://github.com/opennetworkinglab/onos/blob/master/core/net/src/main/java/org/onosproject/net/intent/impl/compiler/PathIntentCompiler.java

NEMO

NEMO is Huawei's declarative modeling language (声明式建模语言) to express network intents

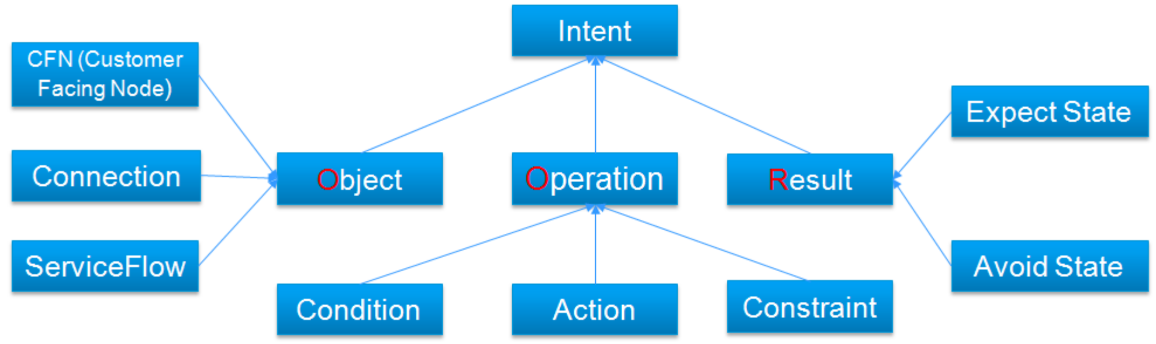
```
SDNRG                                                       Y. Xia, Ed.
Internet-Draft                                            S. Jiang, Ed.
Intended status: Standards Track                           T. Zhou, Ed.
Expires: October 16, 2016                                      S. Hares
                                                            Y.Zhang, Ed.
                                            Huawei Technologies Co., Ltd
                                                          April 14, 2016


                     NEMO (NEtwork MOdeling) Language
                     draft-xia-sdnrg-nemo-language-04

Abstract

   The North-Bound Interface (NBI), located between the control plane
   and the applications, is essential to enable the application
   innovations and nourish the eco-system of SDN.

   While most of the NBIs are provided in the form of API, this document
   proposes the NEtwork MOdeling (NEMO) language which is intent based
   interface with novel language fashion.  Concept, model and syntax are
   introduced in the document.
```

xia2016nemo

NEMO Overview

NEMO allows users to express intents as

- Object: Network components to be managed

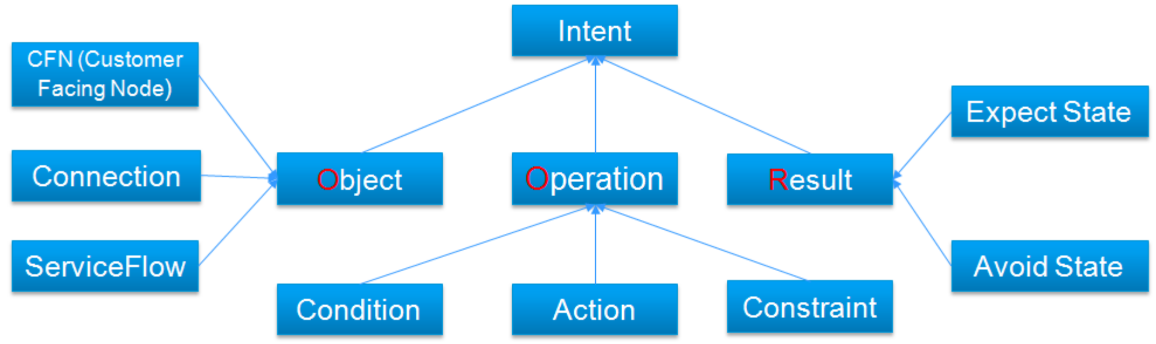


https://wiki.onosproject.org/display/ONOS/NEMO+Language

NEMO Overview

NEMO allows users to express intents as

- Object: Network components to be managed
- Operation: Operating rules for objects

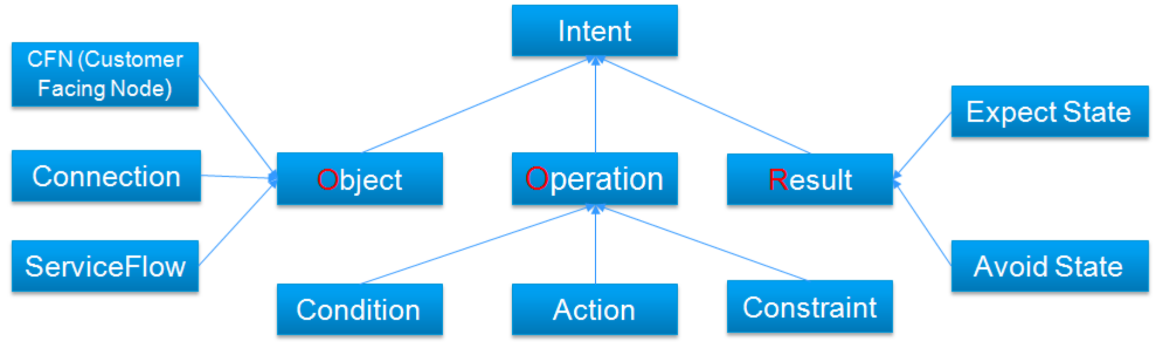


https://wiki.onosproject.org/display/ONOS/NEMO+Language

NEMO Overview

NEMO allows users to express intents as

- Object: Network components to be managed
- Operation: Operating rules for objects
- Results: Target of operations



https://wiki.onosproject.org/display/ONOS/NEMO+Language

NEMO Example

Example: Create a virtual WAN and specify policies for the following topology

```
             Real-time Connection
            +--------------------+
   192.0.2.0/24                198.51.100.0/24
   +---------+                   +-------------+
   | Branch  |-----------------| Headquarter |
   +---------+                   +-------------+
            |                    |
            +--------------------+
             Broadband Connection
```

xia2016nemo

NEMO Example

Example: Create a virtual WAN and specify policies for the following topology

```
             Real-time Connection
            +--------------------+
 192.0.2.0/24                 198.51.100.0/24
 +---------+                  +-------------+
 | Branch  |------------------| Headquarter |
 +---------+                  +-------------+
            |                  |
            +--------------------+
             Broadband Connection
```

Step 1: Create virtual nodes

```
CREATE CFN Branch
       Type l2group
       Property ipv4Prefix : 192.0.2.0/24;

CREATE CFN Headquarter
       Type l2group
       Property ipv4Prefix : 198.51.100.0/24;
```

xia2016nemo

NEMO Example

Example: Create a virtual WAN and specify policies for the following topology

```
               Real-time Connection
              +--------------------+
   192.0.2.0/24                  198.51.100.0/24
   +---------+                  +-------------+
   | Branch  |------------------| Headquarter |
   +---------+                  +-------------+
              |                     |
              +--------------------+
               Broadband Connection
```

Step 2: create virtual connections

```
CREATE Connection broadband_connection
       EndNodes Branch, Headquater
       Property bandwidth : 40000,
                delay : 400;

CREATE Connection realtime_connection
       EndNodes Branch, Headquater
       Property bandwidth : 100,
                delay : 50;
```

---

xia2016nemo

NEMO Example

Example: Create a virtual WAN and specify policies for the following topology

```
                 Real-time Connection
                +--------------------+
    192.0.2.0/24                        198.51.100.0/24
    +---------+                        +-------------+
    | Branch  |------------------------| Headquarter |
    +---------+                        +-------------+
             |                         |
            +--------------------+
                 Broadband Connection
```

xia2016nemo

Step 3: set up flows

```
CREATE ServiceFlow flow_all
       Match src_ip : "192.0.2.0/24",
             dst_ip: "198.51.100.0/24";

CREATE ServiceFlow flow_backup
       Match src_ip : "192.0.2.0/24",
             dst_ip: "198.51.100.0/24",
             port: 55555;
```

NEMO Example

Example: Create a virtual WAN and specify policies for the following topology

```
             Real-time Connection
            +--------------------+
 192.0.2.0/24                   198.51.100.0/24
+---------+                    +-------------+
| Branch  |--------------------| Headquarter |
+---------+                    +-------------+
          |                    |
         +--------------------+
          Broadband Connection
```

Step 4: Apply policies

```
CREATE Operation operation4all
       Target flow_all
       Priority 200
       Action redirect: "realtime_connection";

CREATE Operation operation4backup
       Target flow_backup
       Priority 100
       Condition (time>"19:00:00")
                 || (time<"23:00:00")
       Action redirect: "broadband_connection";
```

xia2016nemo

Model-driven Networking

Why Model-driven?

Before model-driven, SDN controllers are API-driven (e.g., pre-defined, native Java interfaces)

Model-driven Networking

Why Model-driven?

Before model-driven, SDN controllers are API-driven (e.g., pre-defined, native Java interfaces)

× difficult to modularly extend the API

Model-driven Networking

Why Model-driven?

Before model-driven, SDN controllers are API-driven (e.g., pre-defined, native Java interfaces)

- × difficult to modularly extend the API
- × difficult to be adopted for RESTful API

Example

Assume we want to provide an API to read the nodes in a network, with AD-SAL, the API may look like

```
// TopologyService.java
interface TopologyService {
  RpcOutput<List<Node>>
  readNodes(input: RpcInput<List<NodeId>>) {.. .}
}
// Node.Java
class Node {...}
```

However, routers from different vendors or even from different series may have different combinations of sets of properties.

Solution 1: Inheritance

One solution is to extend the `Node` class:

```java
// HuaweiFeature1Node.java
class HuaweiFeature1Node extends Node {
  int feature1;
  ...
}
// HuaweiFeature2Node.java
class HuaweiFeature2Node extends Node {
  int feature2;
  ...
}
// HuaweiFeature12Node.java
class HuaweiFeature12Node extends Node {
  int feature1;
  int feature2;
  ...
}

// CiscoNode.java
class CiscoNode extends Node {...}
```

Drawback:

- Either need too many new classes or a single class contains unnecessary information
- Difficult to be properly serialized/deserialized

Solution 2: Composition

Another solution is to enable new features to be added to the `Node` class:

```
// Node.java
class Node {
  Map<Class<?>, Object> features = ...
  ...
}
// HuaweiFeature1.java
class HuaweiFeature1 {
  int feature1;
  ...
}
// HuaweiFeature2.java
class HuaweiFeature2 {
  int feature2;
  ...
}
```

Drawback:

- Programmers need to read source code/documentation to understand what values are available

```
// UserCode.class
...
  for (Node node: service.readNodes(nodeList))
    if (node.features.get(HuaweiFeature1.class)
      ...
    }
    ...
  }
...
```

Solution 2: Composition

Another solution is to enable new features to be added to the `Node` class:

```
// Node.java
class Node {
  Map<Class<?>, Object> features = ...
  ...
}
// HuaweiFeature1.java
class HuaweiFeature1 {
  int feature1;
  ...
}
// HuaweiFeature2.java
class HuaweiFeature2 {
  int feature2;
  ...
}
```

Drawback:

- Programmers need to read source code/documentation to understand what values are available

```
// UserCode.class
...
  for (Node node: service.readNodes(nodeList))
    if (node.features.get(HuaweiFeature1.class
      ...
    }
    ...
  }
...
```

This is how MD-SAL internally supports the API/data model extension

Model-driven Networking

Model-driven networking is an approach to provide

- Programming flexibility through a common framework and programming model
  - Support API governance
  - Functionally equivalent APIs for different language bindings (多语言绑定)

Model-driven Networking

Model-driven networking is an approach to provide

- Programming flexibility through a common framework and programming model
  - Support API governance
  - Functionally equivalent APIs for different language bindings (多语言绑定)
- Run-time extensibility:
  - Augment existing functionality
  - Load new models (extending controller's functionality)

Model-driven Networking

Model-driven networking is an approach to provide

- Programming flexibility through a common framework and programming model
  - Support API governance
  - Functionally equivalent APIs for different language bindings (多语言绑定)
- Run-time extensibility:
  - Augment existing functionality
  - Load new models (extending controller's functionality)
- Performance & scale

Model-driven Networking

Model-driven networking is an approach to provide

- Programming flexibility through a common framework and programming model
  - Support API governance
  - Functionally equivalent APIs for different language bindings (多语言绑定)
- Run-time extensibility:
  - Augment existing functionality
  - Load new models (extending controller's functionality)
- Performance & scale

Model-driven Networking

Model-driven networking is an approach to provide

- Programming flexibility through a common framework and programming model
  - Support API governance
  - Functionally equivalent APIs for different language bindings (多语言绑定)
- Run-time extensibility:
  - Augment existing functionality
  - Load new models (extending controller's functionality)
- Performance & scale

Similar tools in distributed computing include Protobuf, Apache Thrift, etc.

MD-SAL Example

Consider the node extension case

Define the model

```
container node {
  ... // basic node model
}

augment node {
  // Huawei Feature1
  leaf feature1 { type int32 }
}

augment node {
  // Huawei Feature2
  leaf feature2 { type int32 }
}
```

MD-SAL Example

Consider the node extension case

Define the model

```
container node {
  ... // basic node model
}

augment node {
  // Huawei Feature1
  leaf feature1 { type int32 }
}

augment node {
  // Huawei Feature2
  leaf feature2 { type int32 }
}
```

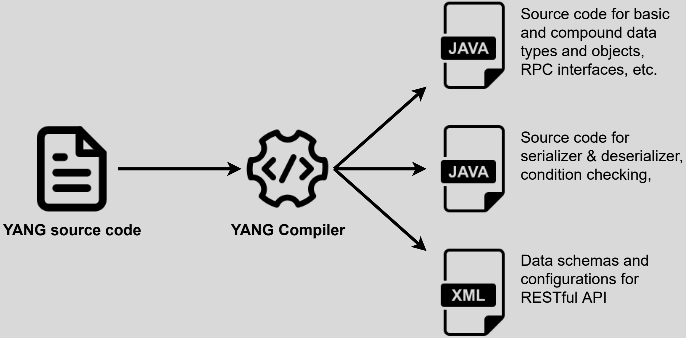
Generate language-bindings (e.g., Java)

```
interface Node {
  ...
  T getAugmentation<T>(Class<T> augmentation);
  void setAugmentation<T>(Class<T> augmentation,
                          T value);
}

interface Feature1Augmentation {
  int getFeature1();
  void setFeature1(int feature1);
}

interface Feature2Augmentation {
  int getFeature2();
  void setFeature2(int feature2);
}
```

MD-SAL

MD-SAL provides an automation tool to handle the complexities of extending existing API or data models

With the model specification (e.g., the YANG modeling language), a compiler automatically generates source code and configuration files

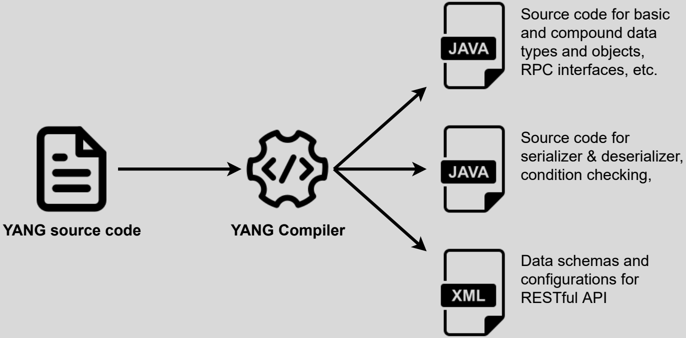


MD-SAL

MD-SAL provides an automation tool to handle the complexities of extending existing API or data models

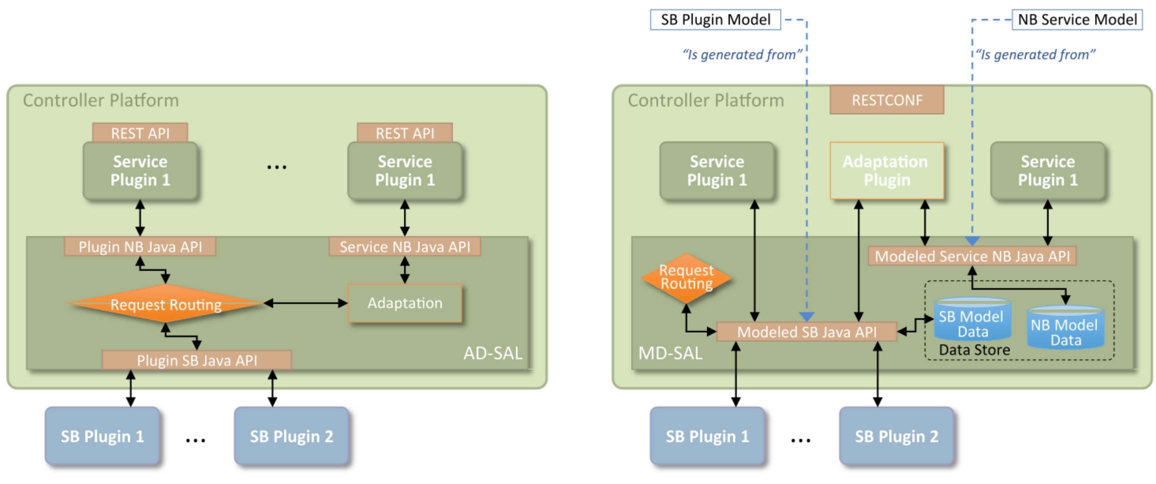
With the model specification (e.g., the YANG modeling language), a compiler automatically generates source code and configuration files

We will introduce YANG later



MD-SAL in Open Daylight

AD-SAL v.s. MD-SAL



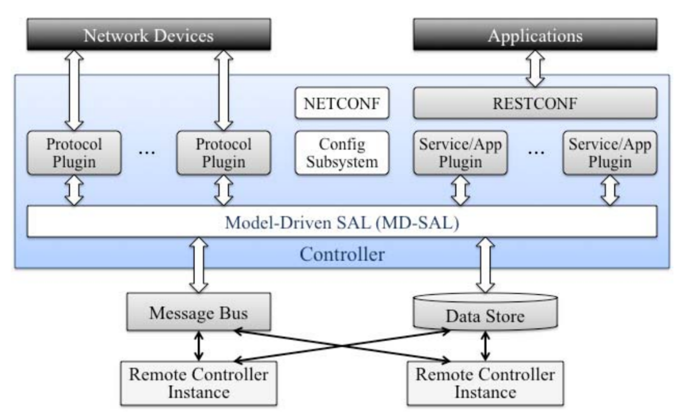
---

medved2014developing

MD-SAL in Open Daylight

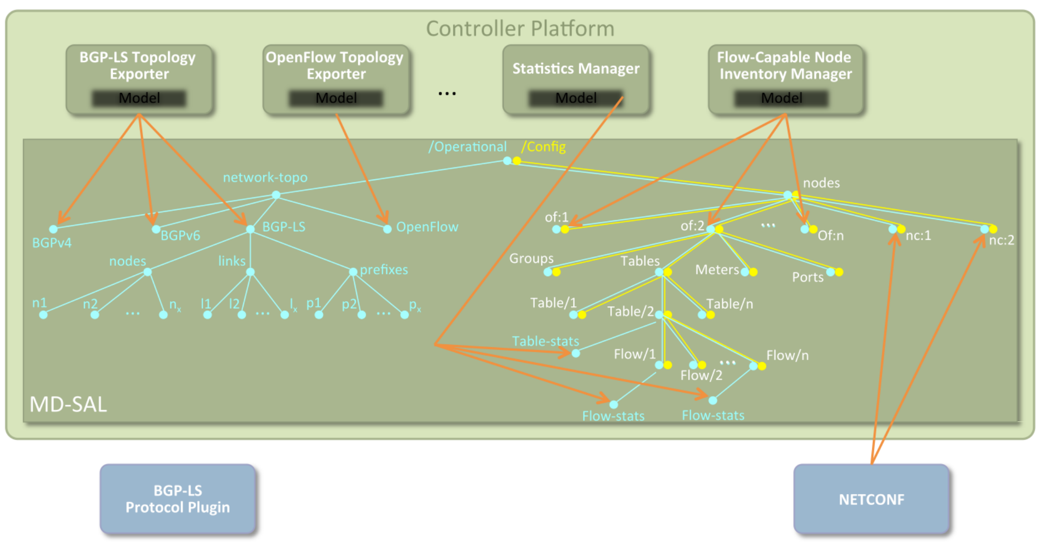
MD-SAL generates API & data store schema for both northbound (北向), southbound (南向), and inter-component communication (组件间通信).

The SAL serves as a communication bus (通信总线) for services in Open Daylight, including the service configuration and management



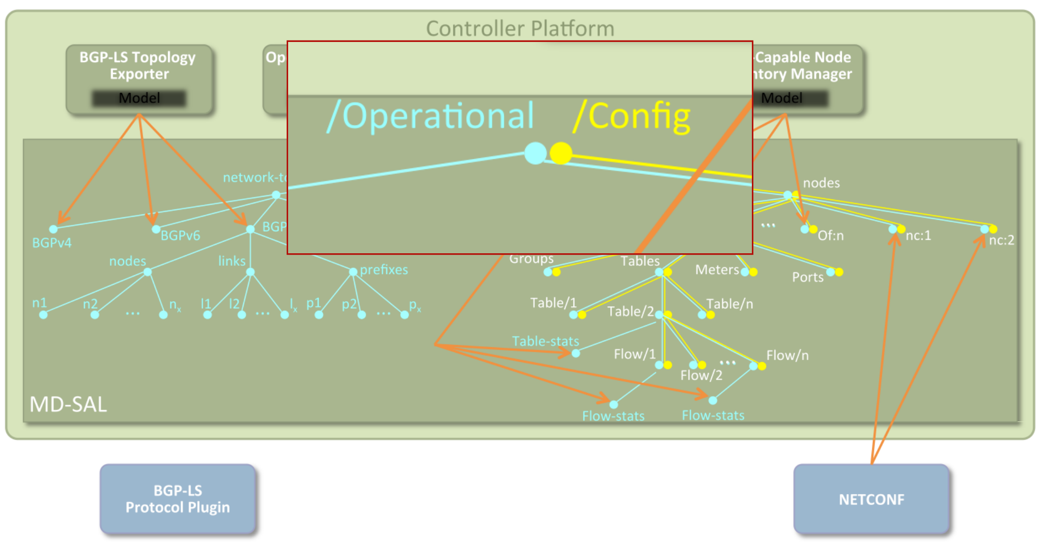
medved2014opendaylight

Example of MD-SAL Data Tree in Open Daylight



medved2014developing

Example of MD-SAL Data Tree in Open Daylight



medved2014developing

Two Types of Data in Open Daylight Data Store

- **config** (配置数据): used to store data that represent configurations, readable/writable through the RESTful API, readable/writable by an internal plugin
- **operational** (运行数据): used to store data that represent ground truth data obtained from plugins or devices, read-only through the RESTful API, readable/writable by an internal plugin

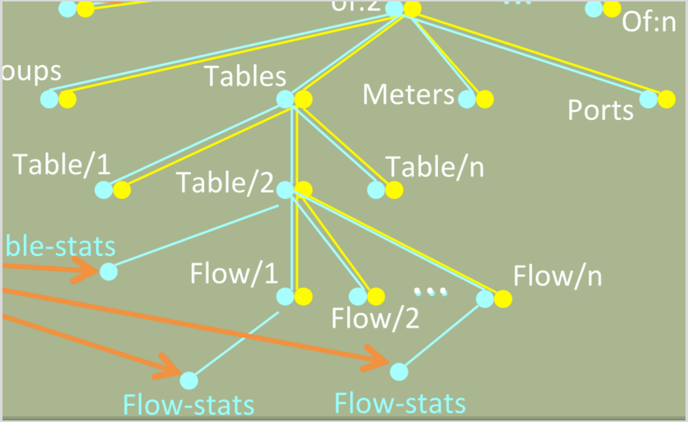
Two Types of Data in Open Daylight Data Store

- **config** (配置数据): used to store data that represent configurations, readable/writable through the RESTful API, readable/writable by an internal plugin
- **operational** (运行数据): used to store data that represent ground truth data obtained from plugins or devices, read-only through the RESTful API, readable/writable by an internal plugin

config represents the state you want the data to be and **operational** represents the state the data really are.

Example of Data Tree Types

Example:

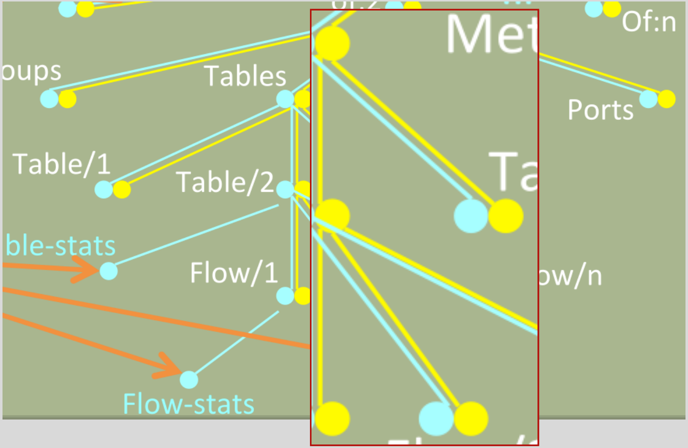


medved2014developing

Example of Data Tree Types

Example:

- Content of an OpenFlow flow entry can be changed, so it is in the `/config` data tree (yellow)

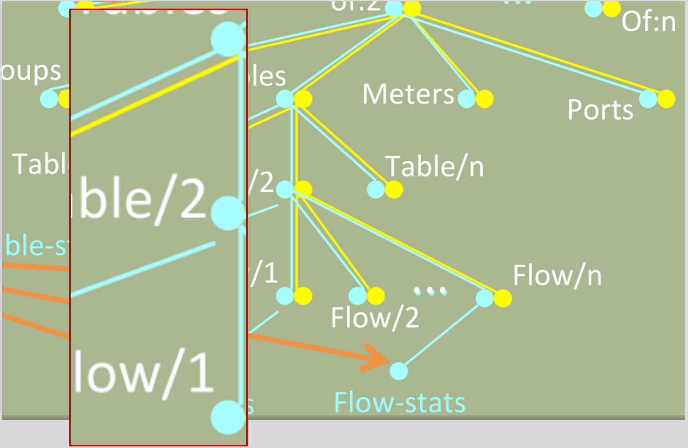


medved2014developing

Example of Data Tree Types

Example:

- Content of an OpenFlow flow entry can be changed, so it is in the `/config` data tree (yellow)
- When an OpenFlow flow entry is installed, it is also in the `/operational` data tree (blue)

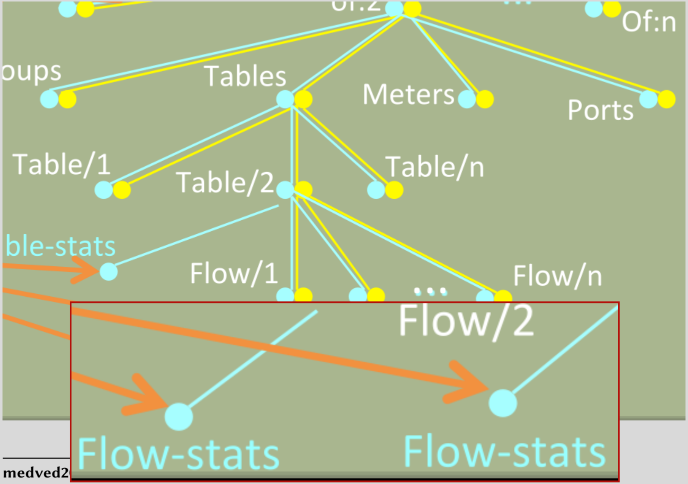


medved2014developing

Example of Data Tree Types

Example:

- Content of an OpenFlow flow entry can be changed, so it is in the `/config` data tree (yellow)
- When an OpenFlow flow entry is installed, it is also in the `/operational` data tree (blue)
- The statistics for the installed flow entry are collected from the OpenFlow switch, they are in the `/operational` data tree (blue)



medved2

Yet Another Next Generation (YANG)

Overview

YANG is *Yet Another Next Generation* modeling language for Network Configuration Protocol. It is first designed for state synchronization between devices and state storage but is also used for service layer abstraction.

```
Internet Engineering Task Force (IETF)                  M. Bjorklund, Ed.
Request for Comments: 6020                                Tail-f Systems
Category: Standards Track                                   October 2010
ISSN: 2070-1721


              YANG - A Data Modeling Language for
          the Network Configuration Protocol (NETCONF)

Abstract

   YANG is a data modeling language used to model configuration and
   state data manipulated by the Network Configuration Protocol
   (NETCONF), NETCONF remote procedure calls, and NETCONF notifications.
```

rfc6020

Module

The top-level structure of YANG is module

A module contains

- statements to express meta information about the module (namespace, prefix, date revision)
- statements to import dependent modules
- statements to define data trees

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
choice	7.9	0..n
contact	7.1.8	0..1
container	7.5	0..n
description	7.19.3	0..1
deviation	7.18.3	0..n
extension	7.17	0..n
feature	7.18.1	0..n
grouping	7.11	0..n
identity	7.16	0..n
import	7.1.5	0..n
include	7.1.6	0..n
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
namespace	7.1.3	1
notification	7.14	0..n
organization	7.1.7	0..1
prefix	7.1.4	1
reference	7.19.4	0..1
revision	7.1.9	0..n
rpc	7.13	0..n
typedef	7.3	0..n
uses	7.12	0..n
yang-version	7.1.2	0..1

rfc6020

Module Specification and Import Example

Define module “example1”

```
module example1 {
  namespace "urn:examples:example1";
  prefix "example1";

  revision "2021-09-15" {
    description "Initial revision.";
  }

  typedef score {
    type uint8 {
      range "0..100";
    }
  }

  ...
}
```

Module Specification and Import Example

Define module “example1”

```
module example1 {
  namespace "urn:examples:example1";
  prefix "example1";

  revision "2021-09-15" {
    description "Initial revision.";
  }

  typedef score {
    type uint8 {
      range "0..100";
    }
  }

  ...
}
```

Define module “example2” which imports “example1” (renamed as “abc”)

```
module example2 {
  namespace "urn:examples:example2";
  prefix "example2";

  revision "2021-09-15" {
    description "Initial revision.";
  }

  import "example1" {
    prefix "abc";
  }

  ...

  typedef score-s {
    type "abc:score" {
      range "90..100";
    }
  }
}
```

Type System

The YANG language is used to specify the data model as a tree structure

Types for leaf and leaf-list nodes:

- built-in types
- types defined by "typedef" statement

Types for non-leaf nodes:

- container
- list

Name	Description
binary	Any binary data
bits	A set of bits or flags
boolean	"true" or "false"
decimal64	64-bit signed decimal number
empty	A leaf that does not have any value
enumeration	Enumerated strings
identityref	A reference to an abstract identity
instance-identifier	References a data tree node
int8	8-bit signed integer
int16	16-bit signed integer
int32	32-bit signed integer
int64	64-bit signed integer
leafref	A reference to a leaf instance
string	Human-readable string
uint8	8-bit unsigned integer
uint16	16-bit unsigned integer
uint32	32-bit unsigned integer
uint64	64-bit unsigned integer
union	Choice of member types

rfc6020

Example of the Type System

We use the following example to illustrate how the data tree is built for different type of nodes

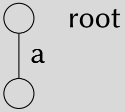
```
container a {
  list b {
    leaf c {
      type string;
    };
    leaf-list d {
      type int32;
    };
  }
}
```

Each module has an implicit root node

Example of the Type System

We use the following example to illustrate how the data tree is built for different type of nodes

```
container a {
  list b {
    leaf c {
      type string;
    };
    leaf-list d {
      type int32;
    };
  }
}
```

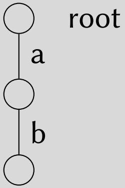


Top-level node is looked up by name

Example of the Type System

We use the following example to illustrate how the data tree is built for different type of nodes

```
container a {
  list b {
    leaf c {
      type string;
    };
    leaf-list d {
      type int32;
    };
  }
}
```

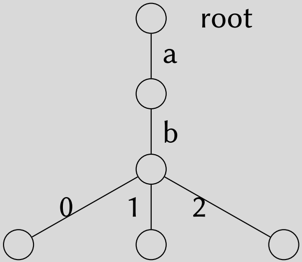


A container node has one instance and looks up the subtree by member name

Example of the Type System

We use the following example to illustrate how the data tree is built for different type of nodes

```
container a {
  list b {
    leaf c {
      type string;
    };
    leaf-list d {
      type int32;
    };
  }
}
```

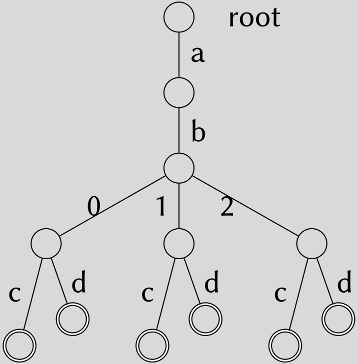


A list node contains multiple instances

Example of the Type System

We use the following example to illustrate how the data tree is built for different type of nodes

```
container a {
  list b {
    leaf c {
      type string;
    };
    leaf-list d {
      type int32;
    };
  }
}
```

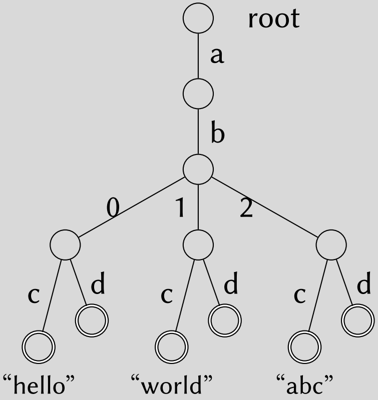


An instance in the list is a data container and looks up subtree by member name

Example of the Type System

We use the following example to illustrate how the data tree is built for different type of nodes

```
container a {
  list b {
    leaf c {
      type string;
    };
    leaf-list d {
      type int32;
    };
  }
}
```

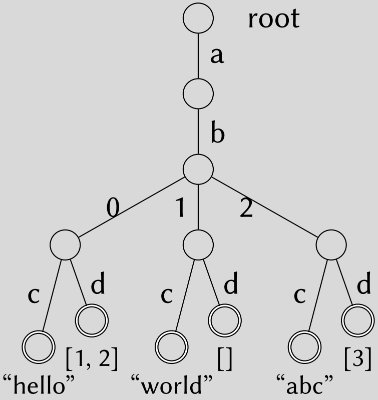


Leaf node *c* has type "string" and the value is a string

Example of the Type System

We use the following example to illustrate how the data tree is built for different type of nodes

```
container a {
  list b {
    leaf c {
      type string;
    };
    leaf-list d {
      type int32;
    };
  }
}
```



Leaf-list node *d* has type “int32” and the value is a list of 32-bit integers

Top-level Types: Data Tree, RPC, & Notifications

Data tree:
Any data tree statement (`container`, `list`, `leaf`, `leaf-list`, etc.) will create a data tree in the module

RPC:
RPC is created with the `rpc` statement

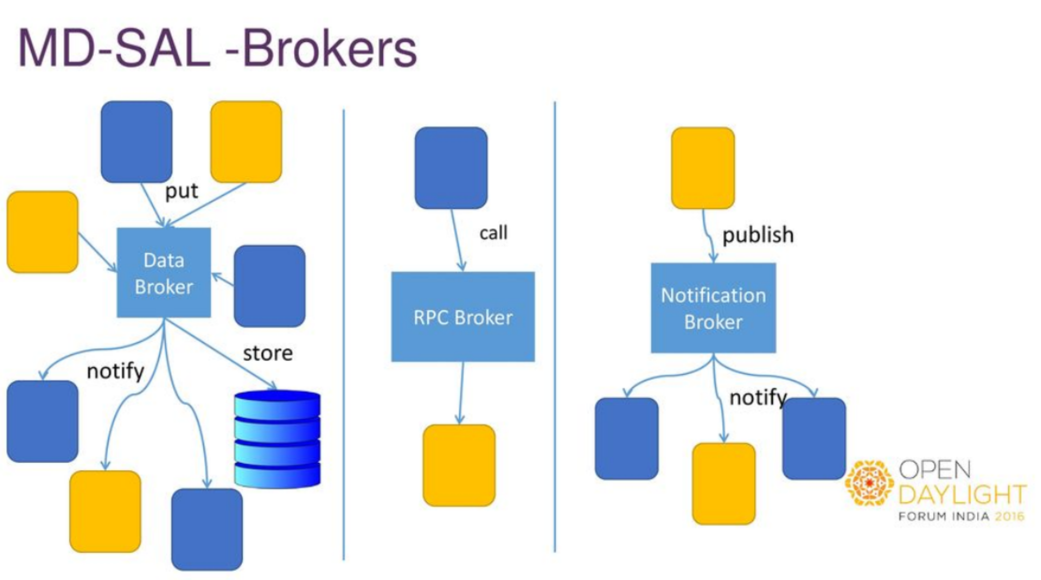
- Data tree in the `input` statement specifies the input format
- Data tree in the `output` statement specifies the output format

Notification:
Notification is created with the `notification` statement

- Data tree in the statement specifies the data format of the notification message

```
module abc {
  ...

  container a {...}

  rpc {
    input {...}
    output {...}
  }

  notification {
    ...
  }
}
```

MD-SAL Top-level Objects



bhagwani2016odl

Config v.s. Operational

In a data tree, one can use the `config` parameter to specify whether it belongs to `config` and `operational` data tree, or `operational` only

Config v.s. Operational

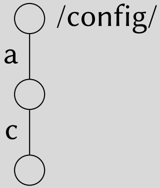
Examples:

```
module config-example {
  container a { // <-- config & operational
    leaf b { // <-- operational only
      type string;
      config false;
    }

    leaf c { // <-- config & operational
      type int32;
    }

    container d { // <-- operational only
      config false;

      leaf e { // <-- invalid
        type int32;
        config true;
      }
    }
  }
}
```

Config v.s. Operational

Examples:

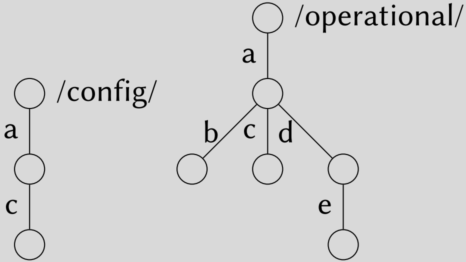
```
module config-example {
  container a { // <-- config & operational
    leaf b { // <-- operational only
      type string;
      config false;
    }

    leaf c { // <-- config & operational
      type int32;
    }

    container d { // <-- operational only
      config false;

      leaf e { // <-- invalid
        type int32;
        config true;
      }
    }
  }
}
```



Config v.s. Operational

Examples:

```
module config-example {
  container a { // <-- config & operational
    leaf b { // <-- operational only
      type string;
      config false;
    }

    leaf c { // <-- config & operational
      type int32;
    }

    container d { // <-- operational only
      config false;

      leaf e { // <-- invalid
        type int32;
        config true;
      }
    }
  }
}
```

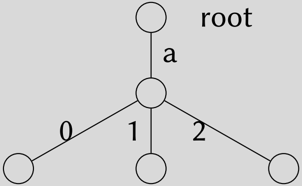


List & List with Keys

YANG uses list with keys to realize maps

List:

```
list a {
  leaf b {
    type int32;
  }
  leaf c {
    type string;
  }
}
```

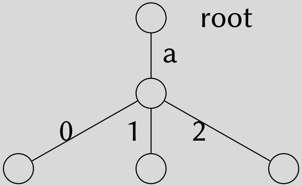


List & List with Keys

YANG uses list with keys to realize maps

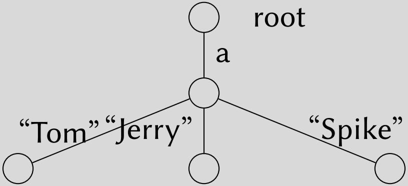
List:

```
list a {
  leaf b {
    type int32;
  }
  leaf c {
    type string;
  }
}
```



List with keys:

```
list a {
  key c; // <----- specify the key field

  leaf b {
    type int32;
  }
  leaf c {
    type string;
  }
}
```

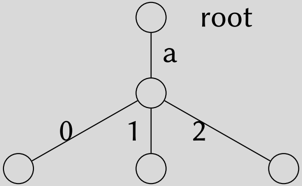


List & List with Keys

YANG uses list with keys to realize maps

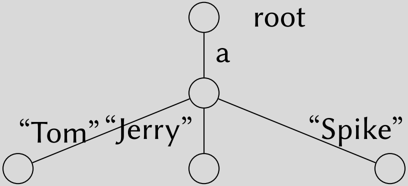
List:

```
list a {
  leaf b {
    type int32;
  }
  leaf c {
    type string;
  }
}
```



List with keys:

```
list a {
  key c; // <----- specify the key field

  leaf b {
    type int32;
  }
  leaf c {
    type string;
  }
}
```



The key always has the same value as the key field

Code Reuse

The grouping statement allows the same subtree structure to be reused

```
module modulea {
  prefix a;

  grouping b {
    leaf c { type string; }
    leaf-list d { type int32; }
  }

  container e {
    uses a:b;
  }
}
```

Code Reuse

The grouping statement allows the same subtree structure to be reused

```
module modulea {
  prefix a;

  grouping b {
    leaf c { type string; }
    leaf-list d { type int32; }
  }

  container e {
    uses a:b;
  }
}
```

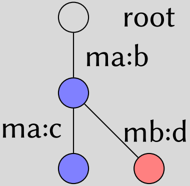
The left YANG model creates the same tree structure as:

```
module modulea {
  prefix a;

  container e {
    leaf c { type string; }
    leaf-list d { type int32; }
  }
}
```

Extension through Augmentation

YANG allows a new module to extend an old module's data/rpc/notification tree

```
// modulea.yang
module modulea {
  prefix ma;
  ...
  container b {
    leaf c { type string; }
  }
}

// moduleb.yang
module moduleb {
  prefix mb;

  import modulea { prefix a; }

  augment /a:b {
    leaf d { type int32; }
  }
}
```



The End

Summary

In this lecture, we cover the following topics:

- Representative SDN controllers
- Intent-based networking and model-driven networking
- YANG language

You should

- know the representative SDN controllers and their design choices

Summary

In this lecture, we cover the following topics:

- Representative SDN controllers
- Intent-based networking and model-driven networking
- YANG language

You should

- know the representative SDN controllers and their design choices
- understand the motivations for intent-based networking and model-based networking

Summary

In this lecture, we cover the following topics:

- Representative SDN controllers
- Intent-based networking and model-driven networking
- YANG language

You should

- know the representative SDN controllers and their design choices
- understand the motivations for intent-based networking and model-based networking
- roughly understand what YANG language does and how data trees are constructed