P4 Developer Day Spring 2019
Advanced Track

Building an SRv6-enabled fabric with P4 and ONOS

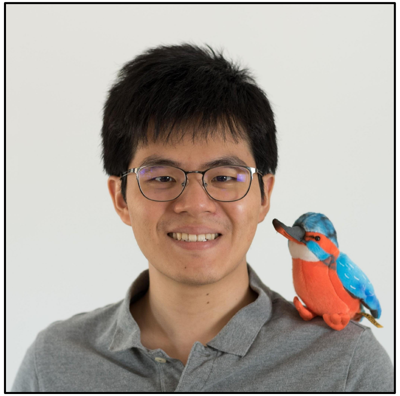
These slides:
http://bit.ly/onos-p4-srv6

Exercises and VM:
http://bit.ly/onos-p4-srv6-repo

Instructors

Carmelo Cascone
ONF

Brian O'Connor
ONF



Yi Tseng 曾毅
ONF

Before we start...

- **Get USB keys with VM from instructors**
  - Or download: http://bit.ly/onos-p4-srv6-repo
- **Copy and import VM into VirtualBox**
  - User: **`sdn`** - Password: **`rocks`**
- **Update ONOS inside VM (requires Internet access)**
  - `cd ~/tutorial`
  - `git pull origin master`
  - `make onos-upgrade`
  - `make app-build`

Why SDN?

All of the exercises could be completed with a traditional, embedded control plane, but SDN:

- Makes programming the devices a data structures problem rather than a distributed protocols problem
- Vastly simplifies the set of protocols “on the wire”
- Makes it easier to reason and verify end-to-end network state
- Means less code (which, hopefully, can be implemented more quickly and with fewer bugs)

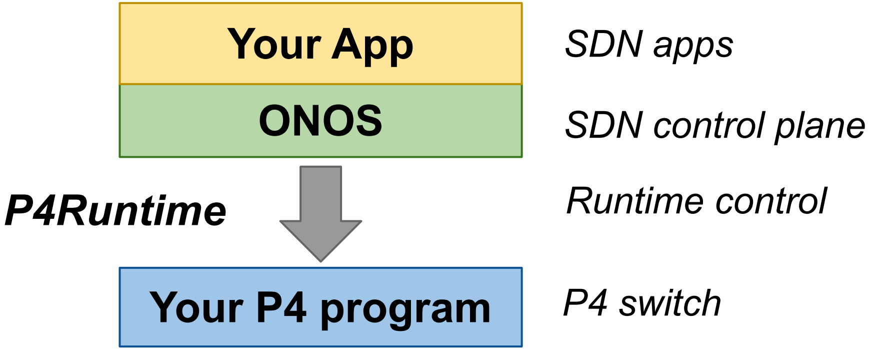
Why SRv6?

- Improves the status-quo for tunnelling (VxLAN, GRE) with similar overhead and superior visibility
- Improves the status-quo for segment routing (MPLS) by enabling non-SR nodes to participate and reusing the RIB
- Enables policies to include data plane processing functions that can either be defined in P4 or offloaded to CPU/FPGA/etc.

SRv6 is a good example of an up and coming protocol that is easy to implement quickly and iterate on in P4.

Goal of this session

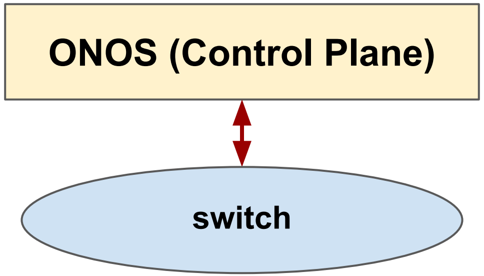
- **Learn the basics of P4, P4Runtime and ONOS**
- **Show you the “big picture” of P4**
    - Acquire enough knowledge to build full-stack network applications
    - Go from a P4 idea to an end-to-end solution
- **Learn the tools to practically experiment with it**



Exercise 1: Packet I/O

Goal: Enable the control plane to do link and host discovery

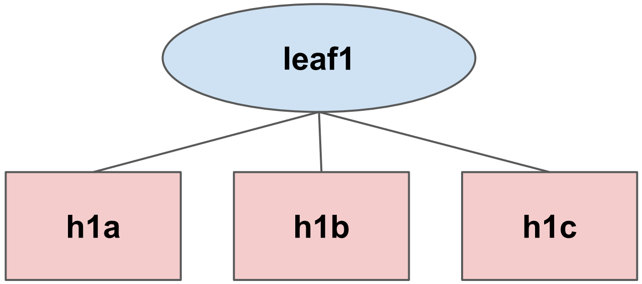
Exercise: Add support for packet-ins from the switch to the control plane and packet-outs from the control plane to the switch



Exercise 2: Bridging

Goal: Enable hosts on the same IPv6 subnet that are connected to the same switch (leaf) to send and receive Ethernet frames to each other

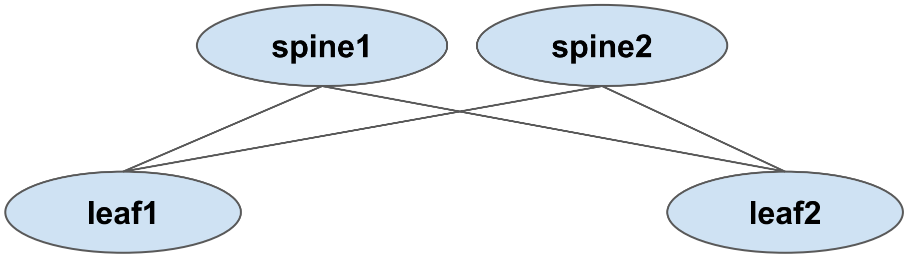
Exercise: Add support for Ethernet bridging in the P4 program, then populate bridging entries using the control plane



Exercise 3: Routing

Goal: Enable hosts connected to different leaves in the leaf-spine topology to send IPv6 packets to each other using multiple paths

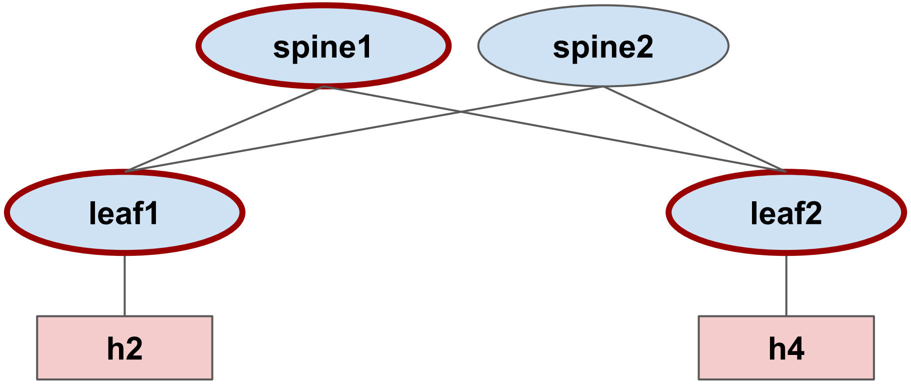
Exercise: Add support for IPv6 routing to the P4 program, then insert static ECMP-based routing rules using static routes from the topology



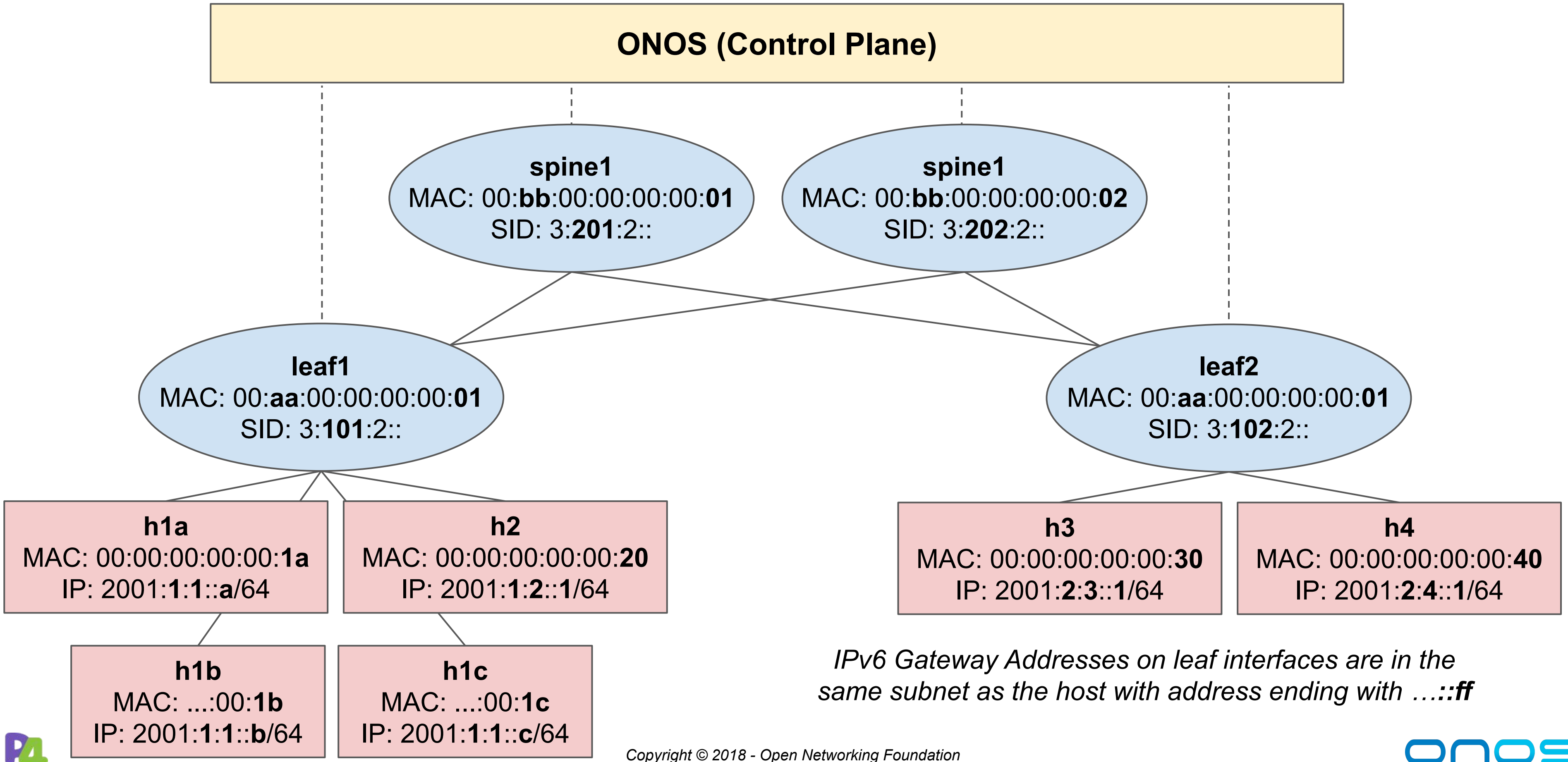
Exercise 4: Segment Routing

Goal: Steer traffic between hosts to use a specific path that is defined at the source node using an SRv6 policy

Exercise: Add support for part of the SRv6 draft standard, then insert SRv6 policies using the ONOS CLI



Tutorial Topology



*IPv6 Gateway Addresses on leaf interfaces are in the same subnet as the host with address ending with …**::ff***

Software tools introduction

Next:

- P4Runtime recap
- Stratum-BMv2
- ONOS
- Packet Test Framework (PTF)

P4Runtime

Runtime control API for P4-defined data planes

P4Runtime v1.0

- **Released on Jan 2019**
- **Open source specification**
  - Started by Google and Barefoot in mid-2016
  - Contributions by many industry professionals
- **Based on continuous implementation feedbacks from Google and ONF**
  - First ONOS demo in Oct 2017

P4Runtime Specification

version 1.0.0

The P4.org API Working Group

2019-01-29

Abstract

P4 is a language for programming the data plane of network devices. The P4Runtime API is a control plane specification for controlling the data plane elements of a device defined or described by a P4 program. This document provides a precise definition of the P4Runtime API. The target audience for this document includes developers who want to write controller applications for P4 devices or switches.

Contents

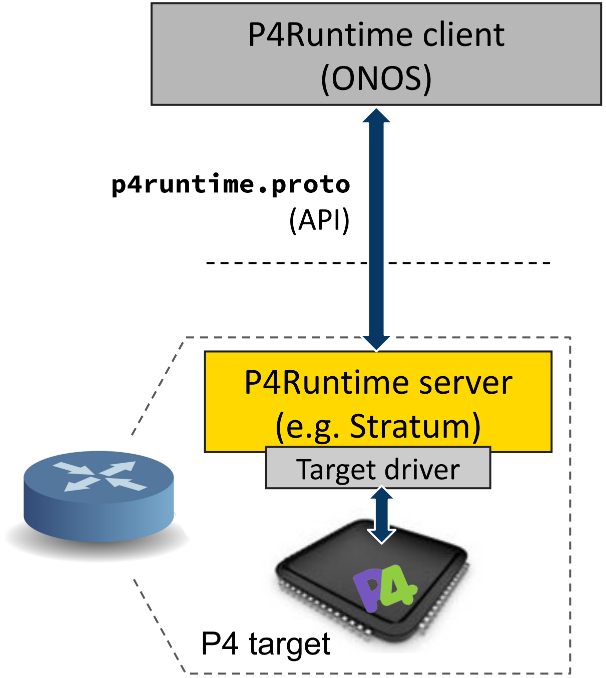
1

https://p4.org/p4-spec/

https://github.com/p4lang/p4runtime

P4Runtime overview

- **Protobuf-based API definition**
  - Efficient wire format
  - Automatically generate code to serialize/deserialize messages for many languages
- **gRPC-based transport**
  - Automatically generate high-performance client/server stubs in many languages
  - Pluggable authentication and security
  - Bi-directional stream channels
- **P4-program independent**
  - Allow pushing new P4 programs to reconfigure the pipeline at runtime
- **Equally good for remote or local control plane**
  - With or without gRPC



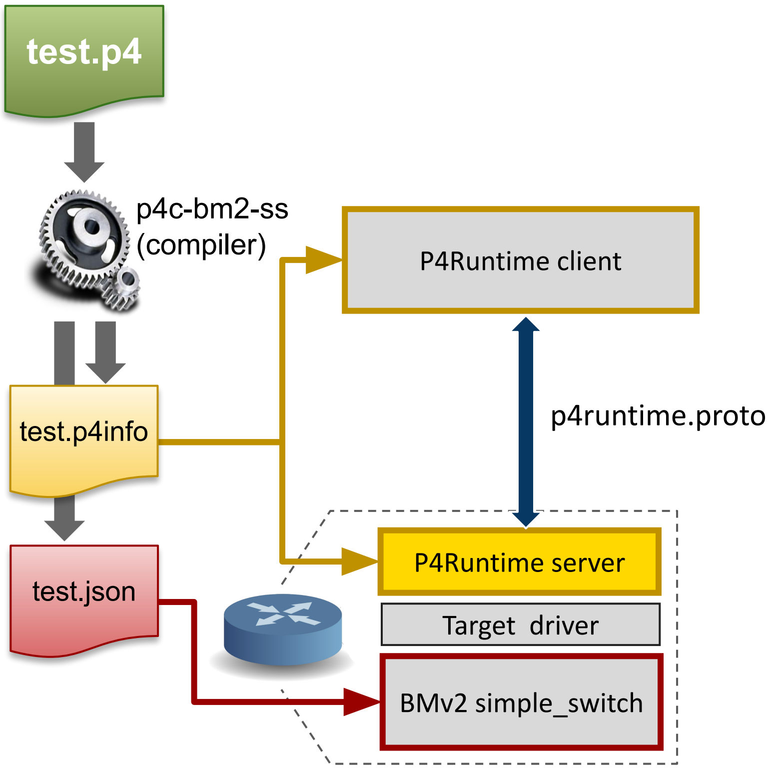
P4Runtime main features

- **Batched read/writes**
  - Table entries, action groups, counters, registers, etc.
- **Master-slave arbitration**
  - For control plane high-availability and fault-tolerance
- **Multiple master controllers via role partitioning**
  - E.g. local control plane for L2, remote one for L3
- **Flexible and efficient packet I/O**
  - OpenFlow-like packet-in/out with arbitrary metadata
  - Digests, i.e. batched notification to controller with subset of packet headers
- **Designed around PSA architecture**
  - But can be extended to others via Protobuf “Any” messages

P4 compiler workflow

P4 compiler generates 2 outputs:

1. **Target-specific binaries**
    - Used to realize switch pipeline
      (e.g. binary config for ASIC, BMv2 JSON, etc.)
2. **P4Info file**
    - "Schema" of pipeline for runtime control
        - Captures P4 program attributes such as tables, actions, parameters, etc.
    - Protobuf-based format
    - Target-independent compiler output
        - Same P4Info for SW switch, ASIC, etc.



Full P4Info protobuf specification:
https://github.com/p4lang/p4runtime/blob/master/proto/p4/config/v1/p4info.proto

P4Info example

basic_router.p4

```
...

action ipv4_forward(bit<48> dstAddr,
                    bit<9> port) {
   eth.dstAddr = dstAddr;
   metadata.egress_spec = port;
   ipv4.ttl = ipv4.ttl - 1;
}

...

table ipv4_lpm {
   key = {
       hdr.ipv4.dstAddr: lpm;
   }
   actions = {
       ipv4_forward;
       ...
   }
   ...
}
```

P4 compiler

basic_router.p4info

```
actions {
  id: 16786453
  name: "ipv4_forward"
  params {
    id: 1
    name: "dstAddr"
    bitwidth: 48
    ...
    id: 2
    name: "port"
    bitwidth: 9
  }
}
...
tables {
  id: 33581985
  name: "ipv4_lpm"
  match_fields {
    id: 1
    name: "hdr.ipv4.dstAddr"
    bitwidth: 32
    match_type: LPM
  }
  action_ref_id: 16786453
}
```

P4Runtime table entry WriteRequest example

basic_router.p4

```
action ipv4_forward(bit<48> dstAddr,
                    bit<9>  port) {
   /* Action implementation */
}
table ipv4_lpm {
   key = {
       hdr.ipv4.dstAddr: lpm;
   }
   actions = {
       ipv4_forward;
       ...
   }
   ...
}
```

Logical view of table entry

```
hdr.ipv4.dstAddr=10.0.1.1/32
     -> ipv4_forward(00:00:00:00:00:10, 7)
```

Control plane generates

WriteRequest message (protobuf text format)

```
device_id: 1
election_id { … }
updates {
  type: INSERT
  entity {
    table_entry {
      table_id: 33581985
      match {
        field_id: 1
        lpm {
          value: "\n\000\001\001"
          prefix_len: 32
        }
      }
      action {
        action_id: 16786453
        params {
          param_id: 1
          value: "\000\000\000\000\000\n"
        }
        params {
          param_id: 2
          value: "\000\007"
```

Stratum

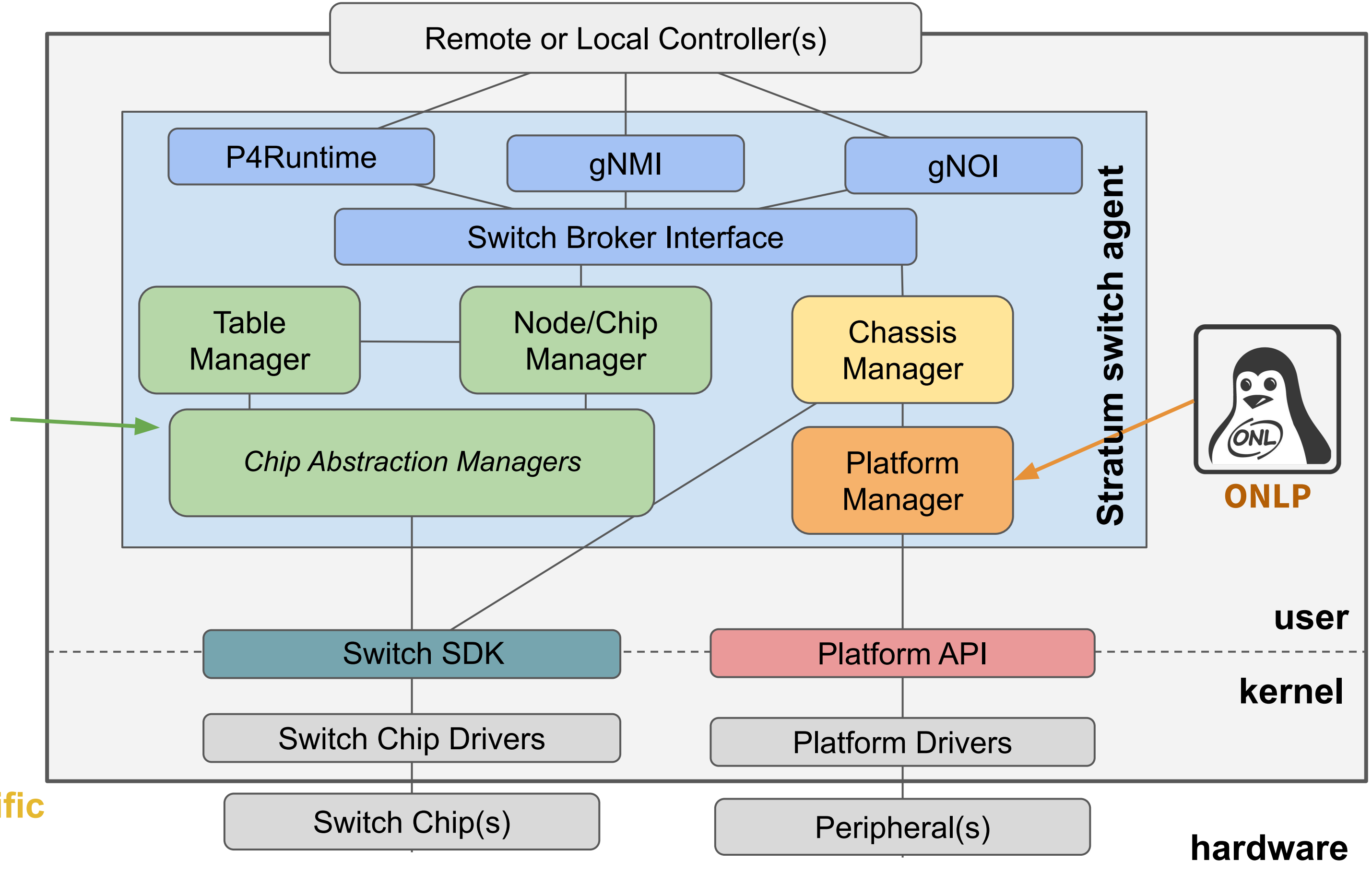
Production-grade reference implementation of P4Runtime server

Stratum overview

Multi-vendor switch implementation of 3 open APIs

- **Control: P4Runtime**
- **Configuration: gNMI with OpenConfig models**
    - Port discovery and configuration, stats collections
    - Power, temp, fans, other peripherals
    - etc.
- **Operations: gNOI (not used in this tutorial)**
    - Device reboot, software upgrade, push certificates, etc.

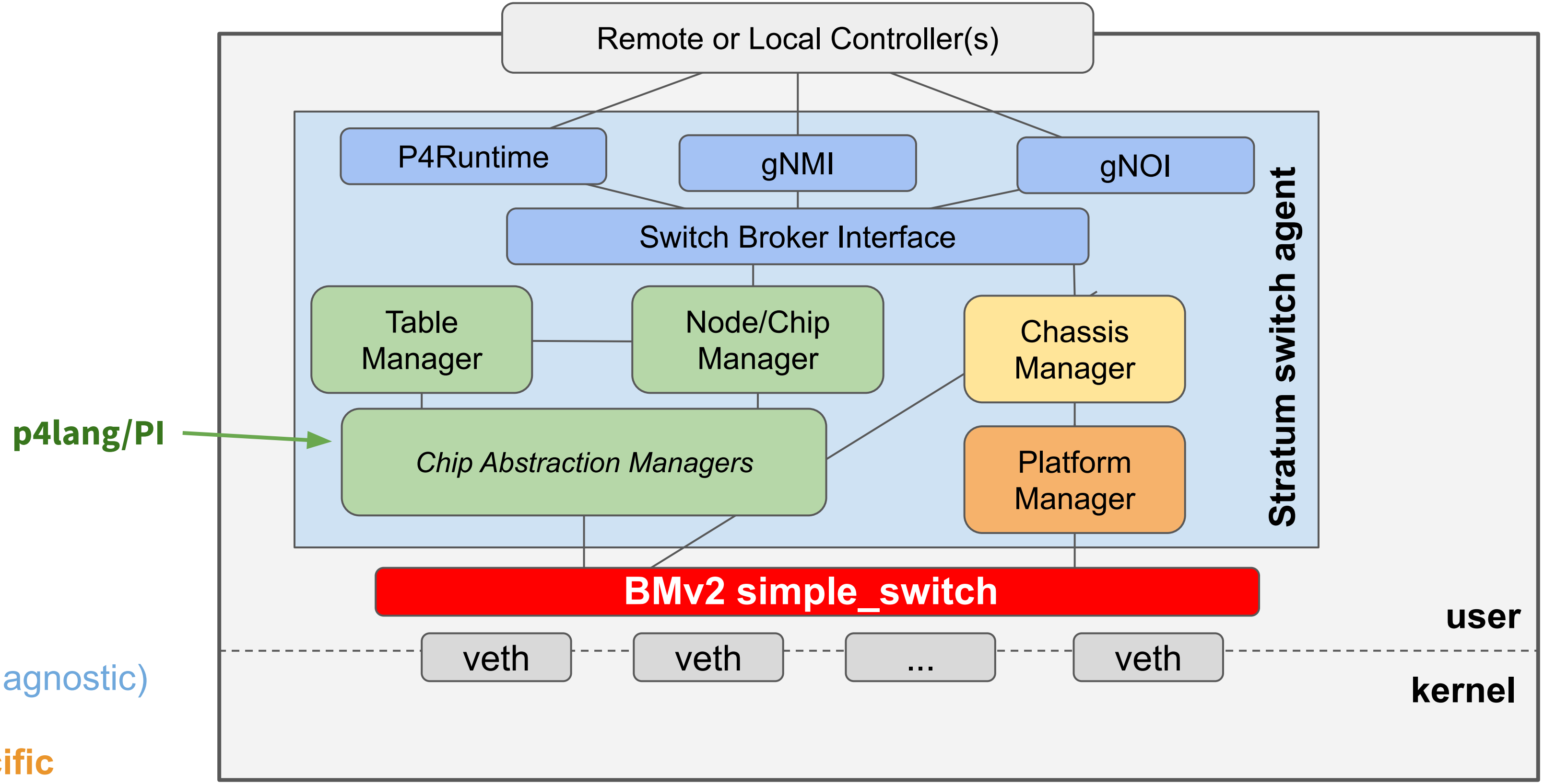
Stratum architecture with HW switches



p4lang/PI and **fpm**-based implementations

Common (HW agnostic)
Chip specific
Platform specific
Chip and Platform specific

Stratum-BMv2



Common (HW agnostic)
Chip specific
Platform specific
Chip and Platform specific

ONOS
A control plane for P4Runtime devices

What is ONOS?

- **Open Network Operating System (ONOS)**
- **Provides the control plane for a software-defined network**
  - Logically centralized remote controller
  - Provides APIs to make it easy to create apps to control a network
- **Runs as a distributed system across many servers**
  - For scalability, high-availability, and performance
- **Focus on service provider for access/edge applications**
  - In production with a major US telecom provider

ONOS releases

4-month release cycles

Avocet (1.0.0)	**2014-12**
...	
Loon (1.11.0)	**2017-08** *(Initial P4Runtime support)*
...	
Raven (2.1.0)	***2019-04*** *(latest release - used today)*

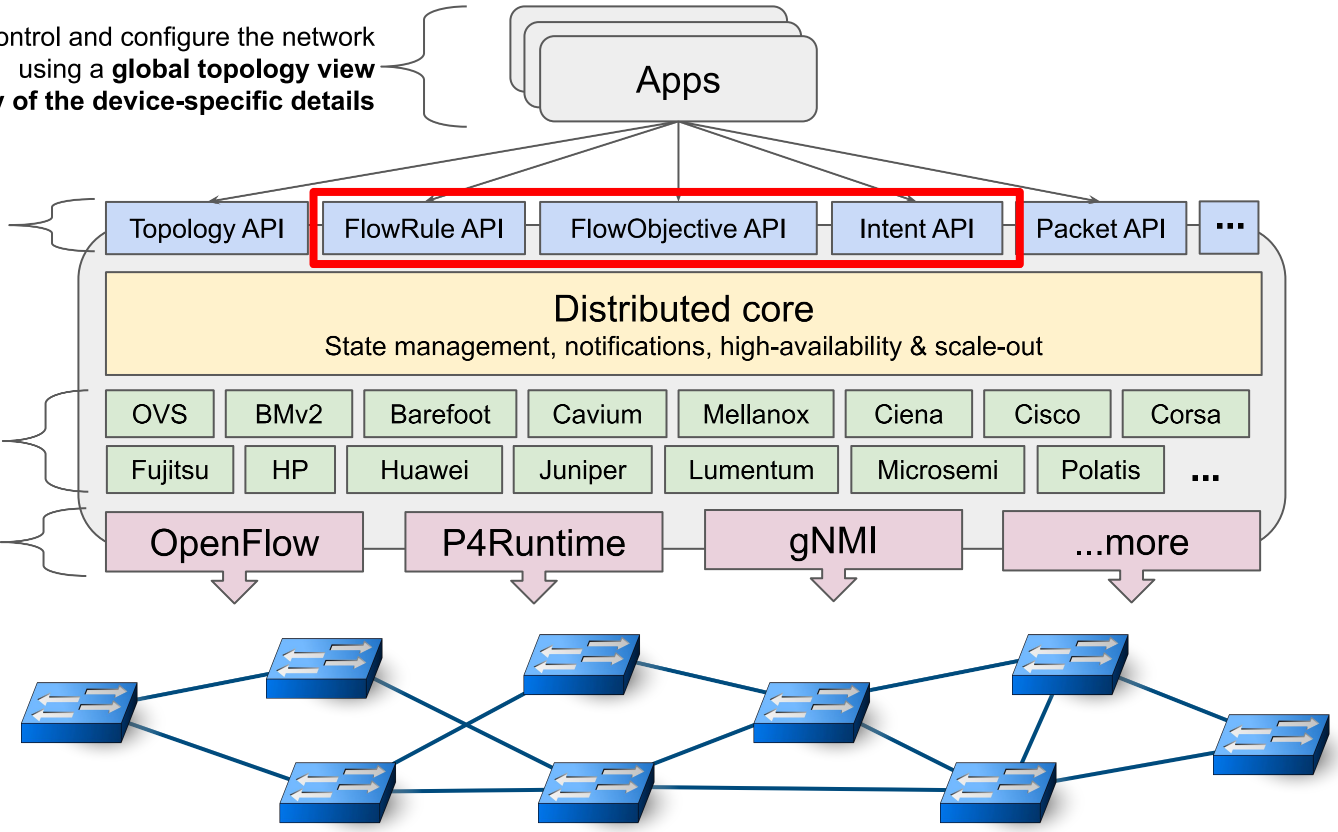
ONOS architecture

Control and configure the network
using a **global topology view**
and **independently of the device-specific details**

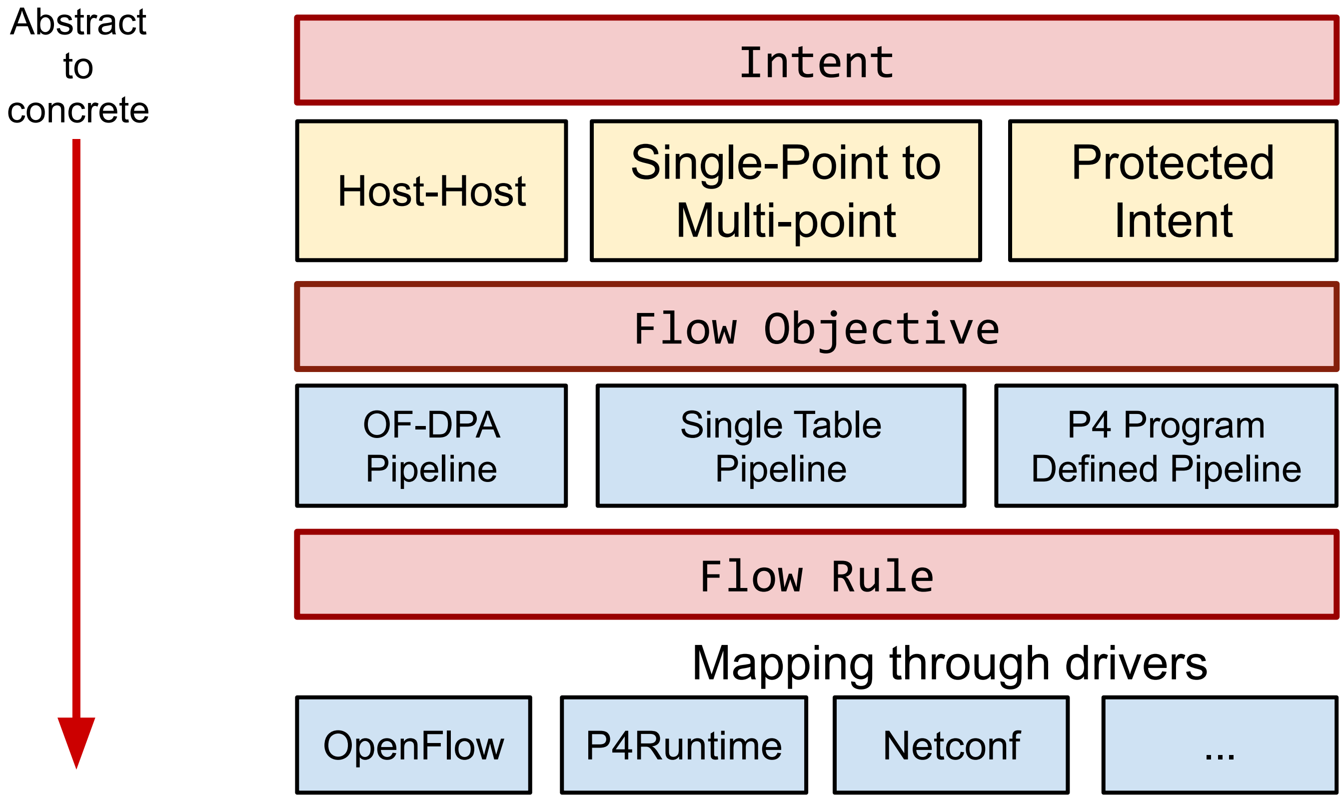
Northbound API
Device/protocol-agnostic
Java, REST, CLI, gRPC

Device driver
Allow device-specific variants
of standard protocols

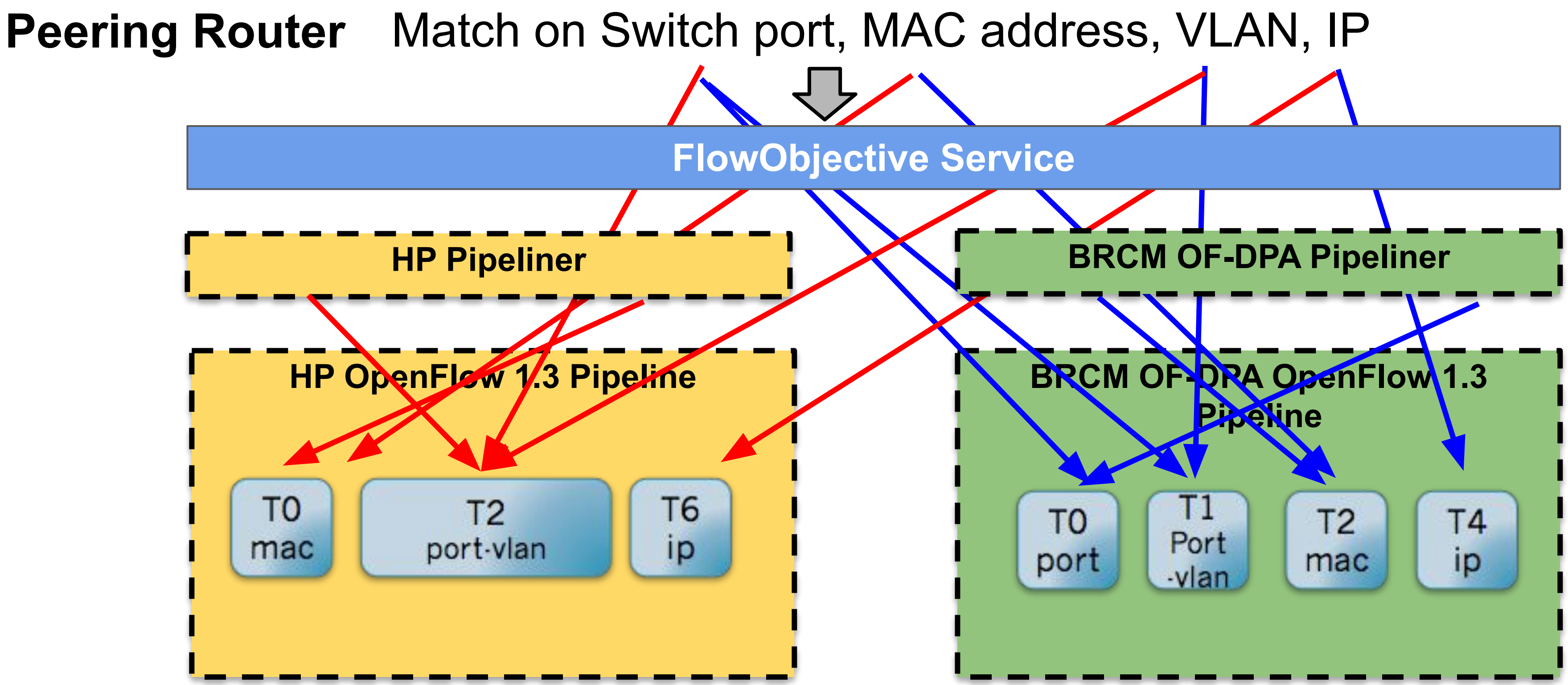
Shared protocol libraries



Network programming API

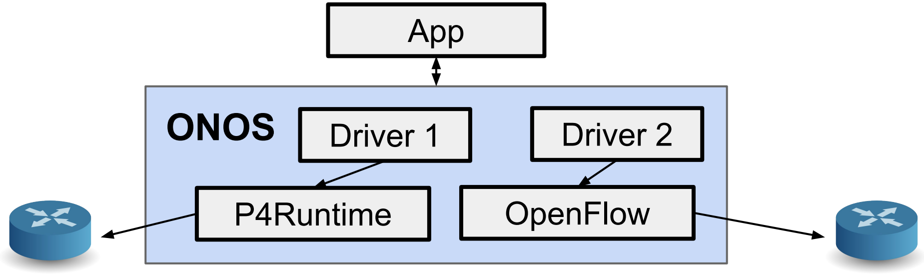


Flow objective example



Driver behaviors in ONOS

- **ONOS defines APIs to interact with device called “behaviors”**
  - `DeviceDescriptionDiscovery` → Read device information and ports
  - `FlowRuleProgrammable` → Write/read flow rules
  - `PortStatisticsDiscovery` → Statistics of device ports (e.g. packet/byte counters)
  - `Pipeliner` → FlowObjective-to-FlowRules mapping logic
  - Etc.
- **Behavior = Java interface**
- **Driver = collection of one or more behavior implementations**
  - Implementations use ONOS protocol libraries to interact with device



ONOS key takeaways

- **Apps are independent from switch control protocols**
  - I.e., same app can work with OpenFlow and P4Runtime devices
- **Different network programming APIs**
  - FlowRule API – pipeline-dependent
  - FlowObjective API – pipeline-independent
    - Drivers translate 1 FlowObjective to many FlowRule
- **FlowObjective API enables application portability**
  - App using FlowObjectives can work with different pipelines
  - For example, switches with different P4 programs

P4 and P4Runtime support in ONOS

P4 and P4Runtime support in ONOS

ONOS originally designed to work with OpenFlow and fixed-function switches.

Extended it to:

1. **Allow ONOS users to bring their own P4 program**
   - For example, today's tutorial
2. **Allow *existing* built-in apps to control *any* P4 pipeline without changing the app**
   - Today: topology and host discovery via packet-in / packet-out
3. **Allow apps to control custom/new protocols as defined in the P4 program**

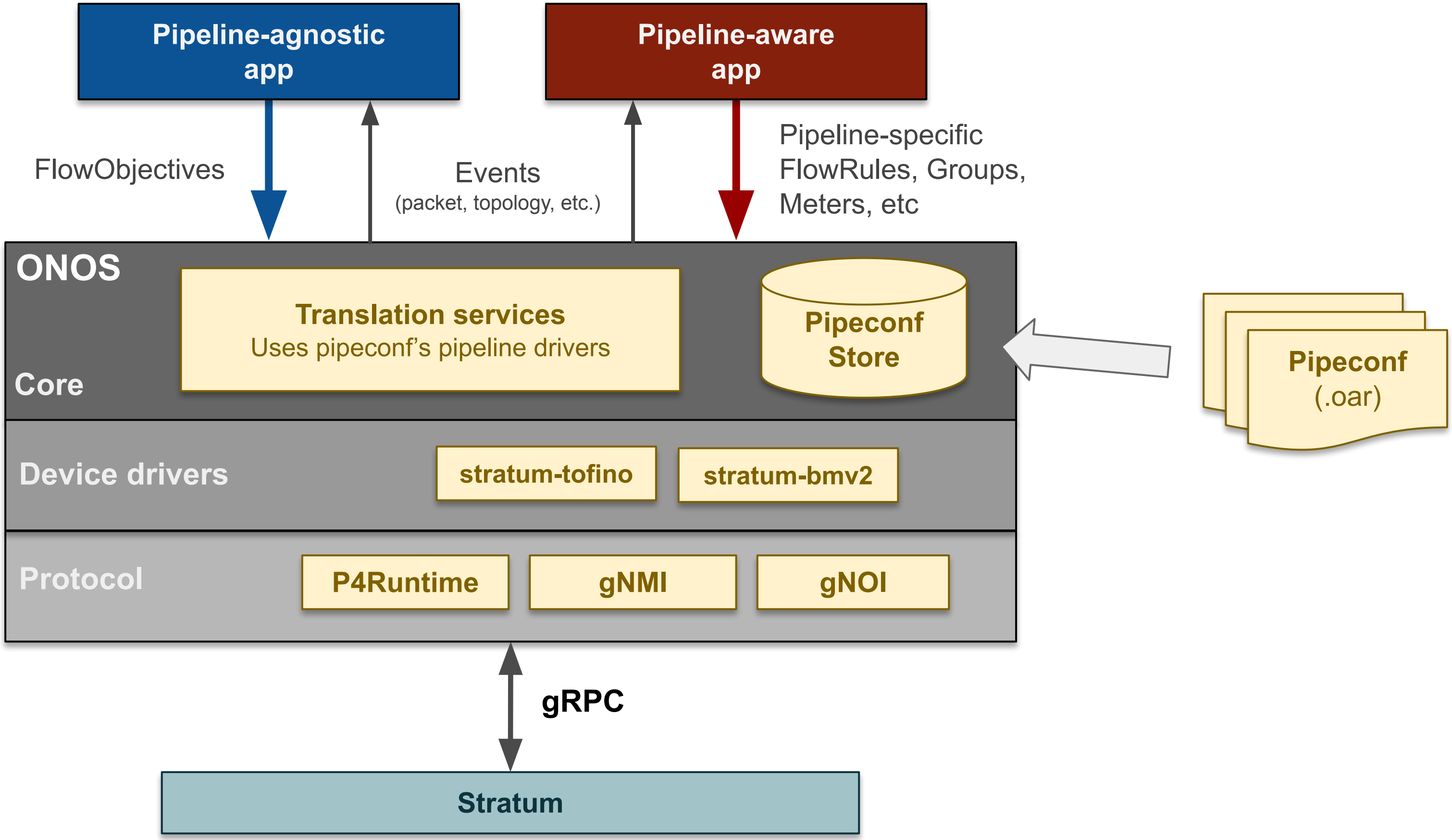
Pipeconf - Bring your own pipeline!

- **Package together everything necessary to let ONOS understand, control, and deploy an arbitrary pipeline**
- **Provided to ONOS as an app**
  - Can use `.oar` binary format for distribution

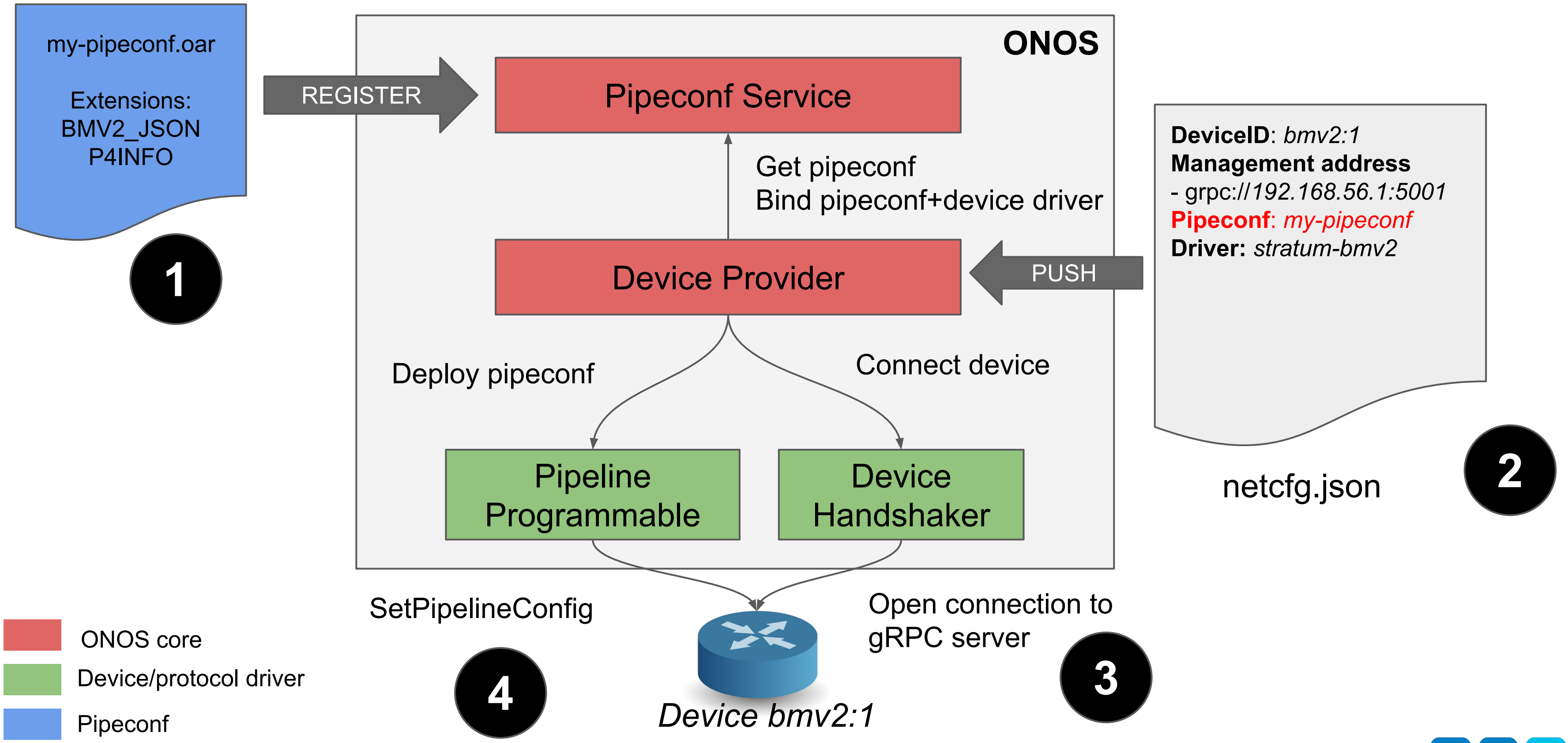
pipeconf.oar

1. **Pipeline model**
   - Description of the pipeline understood by ONOS
   - Automatically derived from P4Info
2. **Target-specific extensions to deploy pipeline to device**
   - E.g. BMv2 JSON, Tofino binary, etc.
3. **Pipeline-specific driver behaviors**
   - E.g. “Pipeliner” implementation: logic to map FlowObjectives to P4 pipeline

Pipeconf support in ONOS



Device discovery and pipeconf deploy



Flow operations

Define flow rules using *same headers/action names as in the P4 program*. E.g match on “`hdr.my_protocol.my_field`”

Pipeconf-based 3 phase translation:

1. Flow Objective → Flow Rule

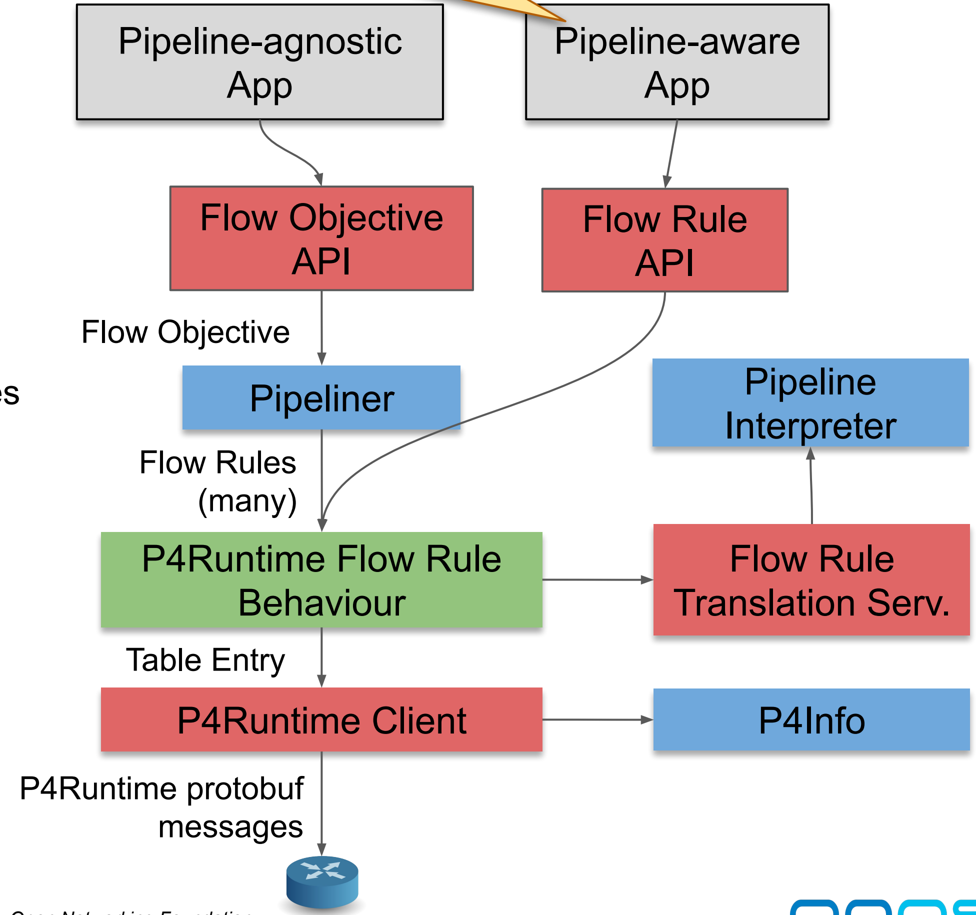
- Maps 1 flow objective to many flow rules

2. Flow Rule → Table entry

- Maps standard headers/actions to P4-defined ones
  E.g. ETH_DST→“`hdr.ethernet.dst_addr`”

3. Table Entry → P4Runtime message

- Maps P4 names to P4Info numeric IDs



ONOS Core
Device/protocol driver
Pipeconf driver

P4Runtime support in ONOS 2.1.0 (Raven)

P4Runtime control entity	ONOS API
Table entry	Flow Rule Service, Flow Objective Service Intent Service
Packet-in/out	Packet Service
Action profile group/members, PRE multicast groups, clone sessions	Group Service
Meter	Meter Service (indirect meters only)
Counters	Flow Rule Service (direct counters) P4Runtime Client (indirect counters)
Pipeline Config	Pipeconf

Unsupported features - community help needed!
Parser value sets, registers, digests

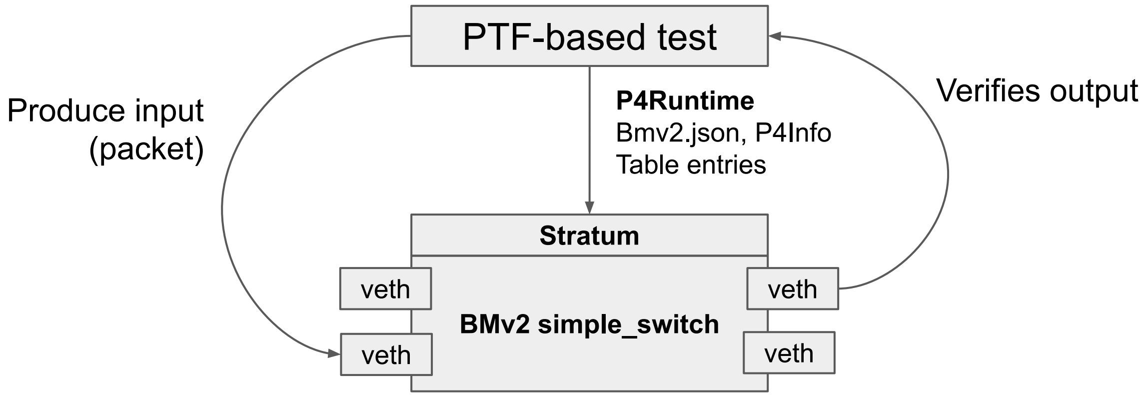
ONOS+P4 workflow recap

- **Write P4 program and compile it**
  - Obtain P4Info and target-specific artifacts (e.g. BMv2 JSON)
- **Create pipeconf**
  - Implement pipeline-specific driver behaviours (Java):
    - Pipeliner (optional - if you need FlowObjective mapping)
    - Pipeline Interpreter (to map ONOS headers/actions to P4 program ones)
    - Other driver behaviors that depend on pipeline
- **Use existing pipeline-agnostic built-in apps**
  - Apps that program the network using FlowObjectives
- **Write new pipeline-aware apps**
  - Apps that use same string names of tables, headers, and actions as in the P4 program

Packet Test Framework (PTF)

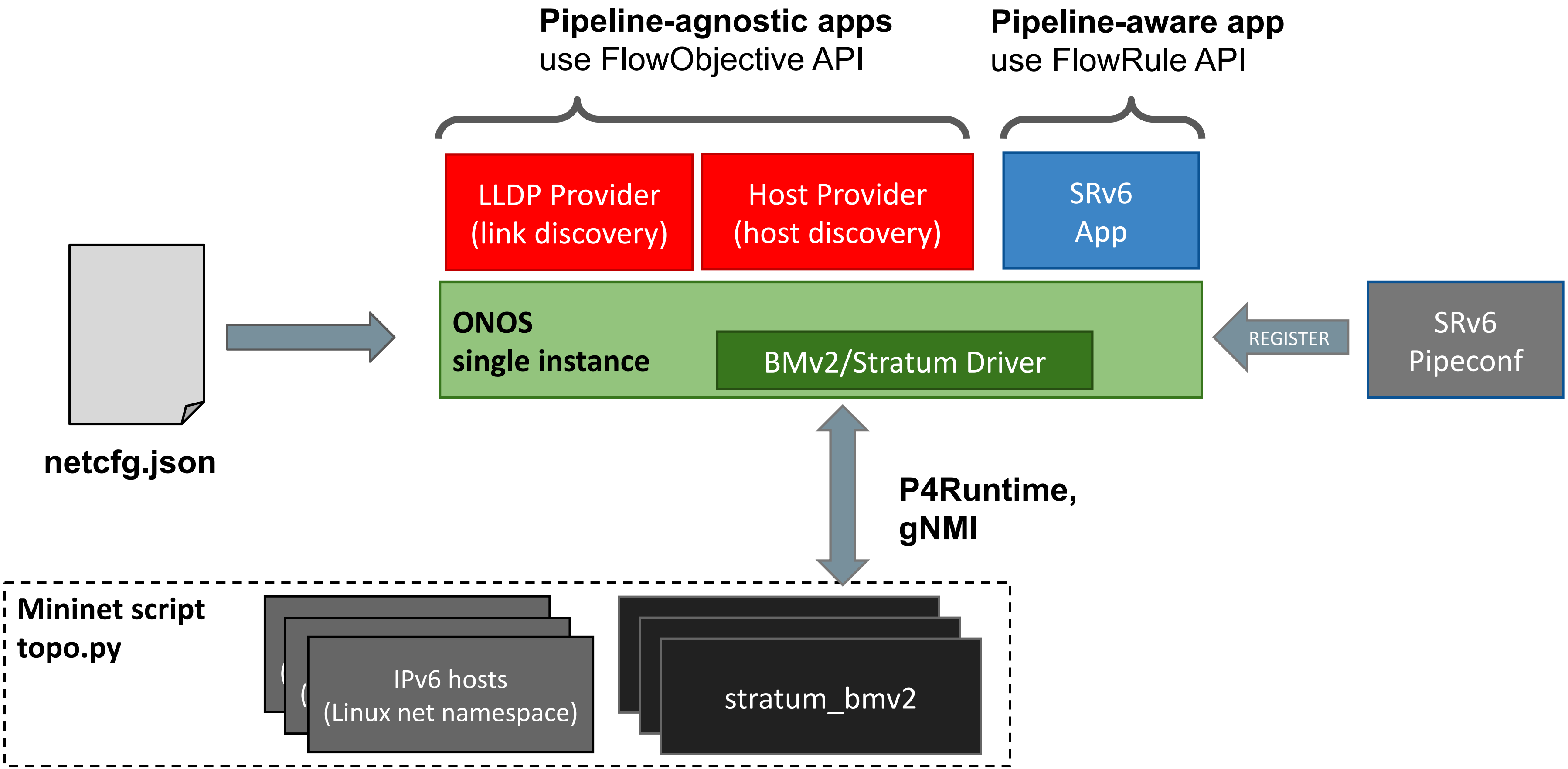
PTF overview

- **Python-based dataplane test framework**
- **Similar to OFTest framework**
  - But focuses on the dataplane and is independent of OpenFlow/P4Runtime
- **P4Runtime lib provided with tutorial starter code**
  - Add/remove table entries, groups, packet-in/out, etc.



Exercise 1

Environment overview



LLDP Provider App

- Provides means to discover network links by injecting LLDP packets in the network

- Reacts to device events (e.g., new switch connection)

- Periodically sends LLDP packets via packet-out for each switch port

- Install packet-in requests (flow objective) on each device
    - Match: ETH_TYPE = LLDP, BDDP
    - Instructions: OUTPUT(CONTROLLER)

Host Provider App

- **Learns location of hosts and IP-MAC mapping by intercepting ARP, NDP and DHCP packets**

- **Reacts to device events (e.g., new switch connection)**

- **Install packet-in requests (flow objective) on each device**
    - Match: ARP, NDP
    - Instructions: OUTPUT(CONTROLLER)

- **Parses sniffed packets to discover hosts**

SRv6 pipeconf

- **ID:** `org.p4.srv6-tutorial`
- **Driver behaviors:**
  - **Pipeliner**
    - Maps FlowObjective from LLDP and Host provider apps
    - Use P4Runtime/v1model clone sessions to send packets to the CPU (packet-in)
  - **Interpreter**
    - Maps packet-in/out to/from ONOS internal representation
    - Maps ONOS known headers (e.g. `ETH_TYPE`) to P4Info-specific ones (e.g. “`hdr.ethernet.type`”)
- **Target-specific extensions**
  - `bmv2.json, p4info.txt`

netcfg.json (devices)

```
{
  "devices": {
    "device:leaf1": {
      "basic": {
        "managementAddress": "grpc://127.0.0.1:50001?device_id=1",
        "driver": "stratum-bmv2",
        "pipeconf": "org.p4.srv6-tutorial"
      },
      "srv6DeviceConfig": {
        "myStationMac": "00:aa:00:00:00:01",
        "mySid": "3:101:2::",
        "isSpine": false
      }
    },
    ...
```

ONOS terminology

- **Criteria**
  - Match fields used in a FlowRule
- **Traffic Treatment**
  - Actions/instructions of a FlowRule
- **Pi* classes**
  - Classes used to describe protocol-independent constructs
  - Equivalent of P4Runtime entities
  - Examples
    - **`PiTableId:`** name of a table as in the P4 program
    - **`PiMatchFieldId:`** name of a match filed in a table
    - **`PiCriterion`**: match fields each one defined by its name and value
    - **`PiAction`**: action defined by its name and list of parameters

Exercise 1: Software tools basics and packet I/O

Goal: Enable ONOS to do link and host discovery using built-in apps

Exercise:

- Modify packet-in/out handling in P4 code
- Run PTF tests
- Modify pipeconf Interpreter (to map packet-in/out)
- Start ONOS; load app with pipeconf; start Mininet
- Verify that link discovery works

Exercise 1: Get Started

Slides: `http://bit.ly/onos-p4-srv6`

Open:

~/tutorial/README.md

~/tutorial/EXERCISE-1.md

Or use GitHub markdown preview:

`http://bit.ly/onos-p4-srv6-repo`

Solution:

`~/tutorial/solution`

Update tutorial repo (requires Internet access)

```
cd ~/tutorial
git pull origin master
make onos-upgrade
make app-build
```

P4 language cheat sheet:
`http://bit.ly/p4-cs`

You can work on your own using the instructions.
Ask for instructors help when needed.

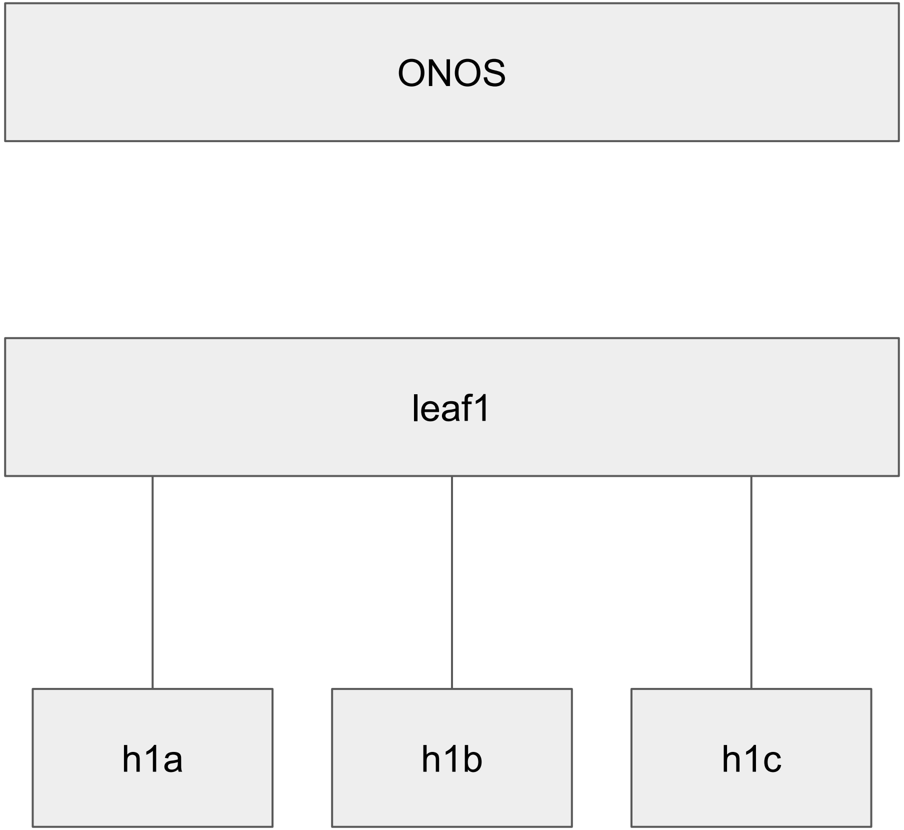
Exercise 2 - Bridging

Exercise 2: Overview

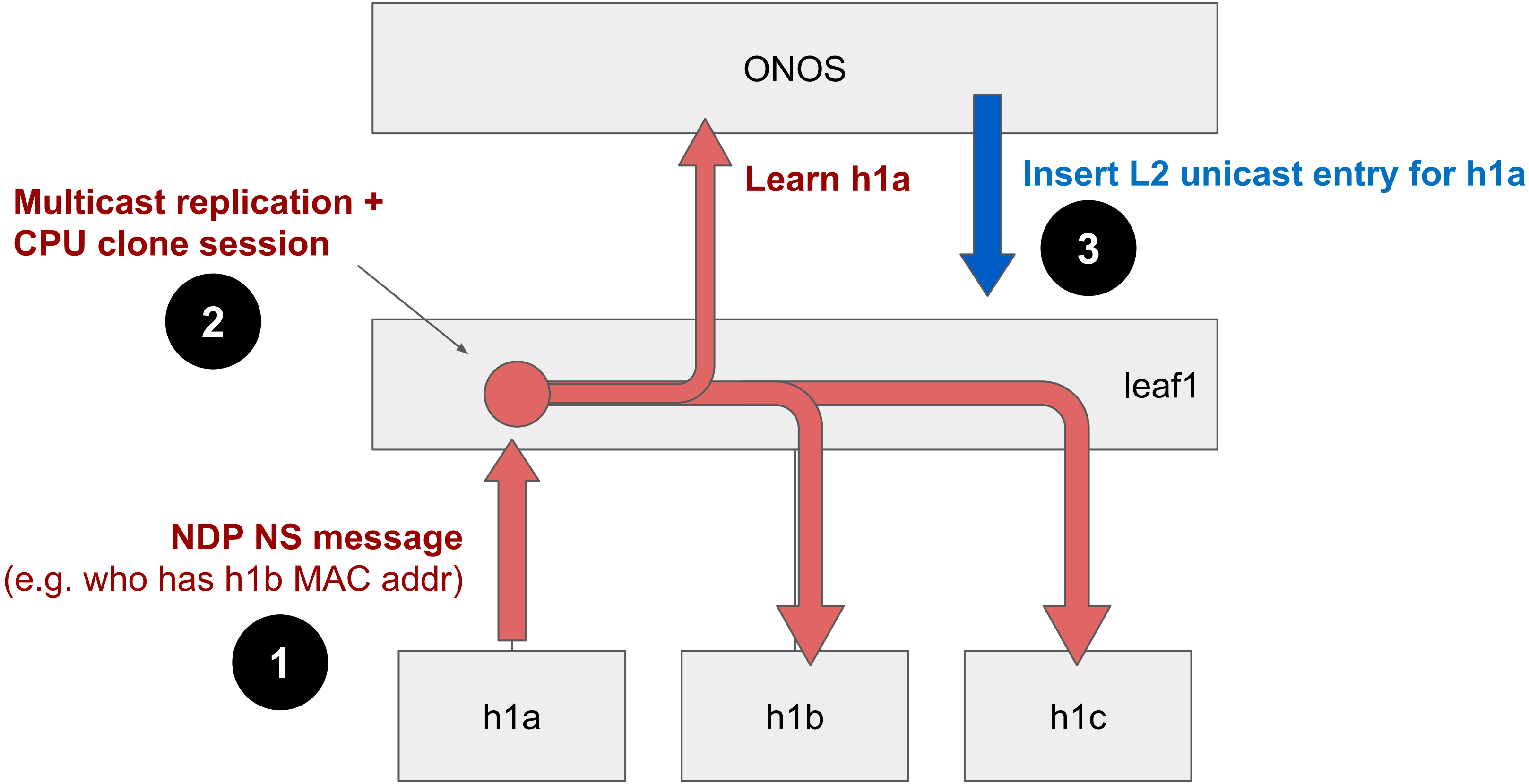
Add basic L2 bridging functionality to leaf switches

- Replicate packets to host-facing interfaces if the destination is multicast or broadcast, such as for NDP Neighbor Advertisement/Solicitation (NA/NS)
- Provide unicast forwarding for “learned” hosts
- ONOS learns about hosts by intercepting NDP NA/NS messages

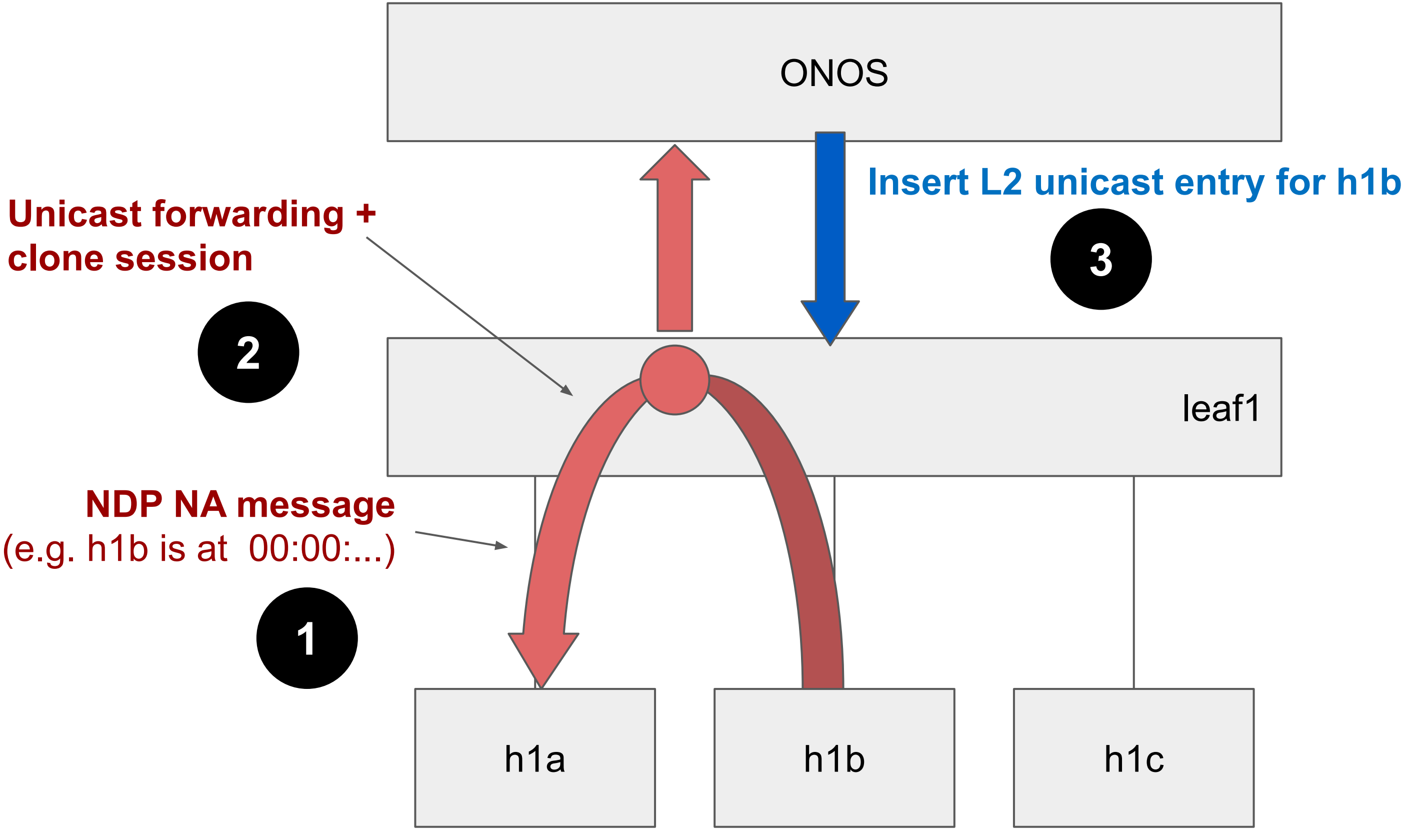
Exercise topology overview



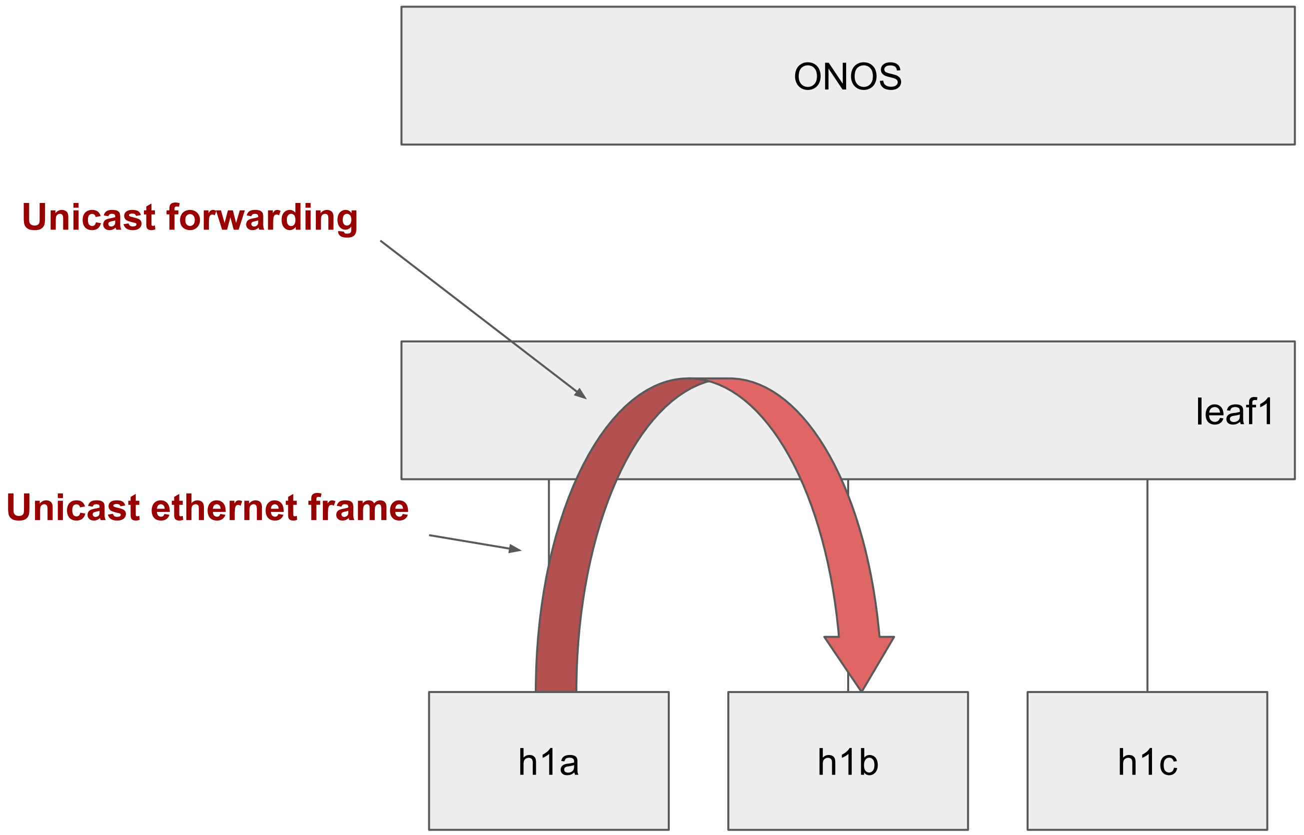
Host discovery (NDP NS)



Host discovery (NDP NA)



Unicast forwarding



Exercise 2 Goal

- **Create P4 table(s) to handle both unicast and broadcast/multicast packets**
  - Note you might need to handle IPv6 multicast addresses 33:33:**:**:**:** with ternary match
- **Modify PTF test case(s) to verify P4 implementation**
  - ptf/tests/bridging.py
- **Modify L2 component in ONOS app**
  - app/src/main/java/org/p4/p4d2/tutorial/L2BridgingComponent.java
- **Test on Mininet**

Exercise 2: Get Started

Slides: `http://bit.ly/onos-p4-srv6`

Open:

~/tutorial/EXERCISE-2.md

Or use GitHub markdown preview:

`http://bit.ly/onos-p4-srv6-repo`

Solution:

`~/tutorial/solution`

Extra Credit Ideas:

- Solve the host discovery race condition
- Remove inactive/old host entries periodically
- Explore the `@name` & `@globalname` annotations to shorten tables, actions, etc.

Update tutorial repo (requires Internet access)

```
cd ~/tutorial
git pull origin master
make onos-upgrade
make app-build
```

P4 language cheat sheet:
`http://bit.ly/p4-cs`

You can work on your own using the instructions.
Ask for instructors help when needed.

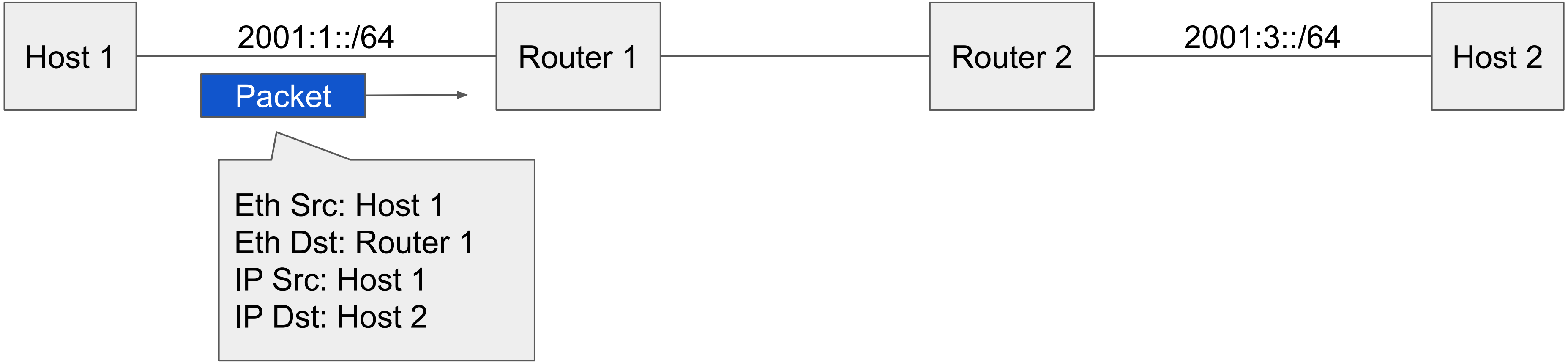
Exercise 3 - IPv6 routing

Exercise 3: Overview

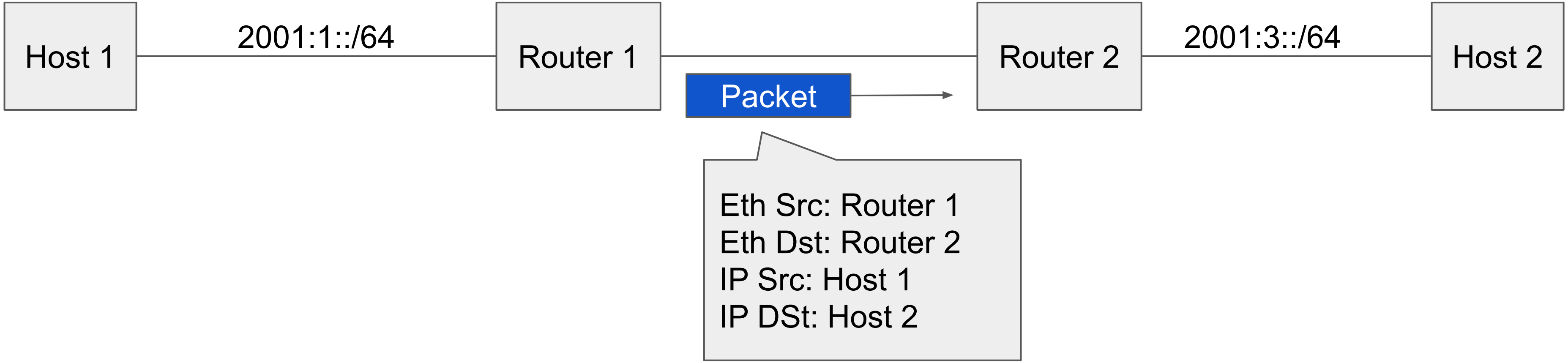
Make the topology behave like a standard IPv6 fabric.

- Leaf switches should reply to NDP NS messages to resolve their “gateway” address
- Process packets through the routing pipeline if the destination mac address is the “gateway” mac address
- Map IPv6 prefixes (LPM) to next hops (routing table)
- Allow mapping to multiple next hops for leaf switches, i.e., use ECMP when forwarding to spines.

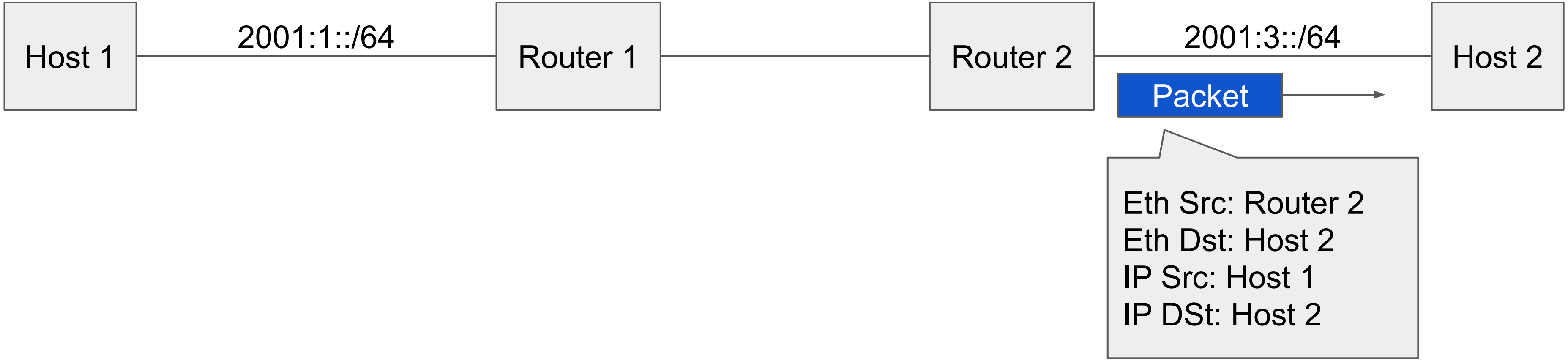
IP unicast routing



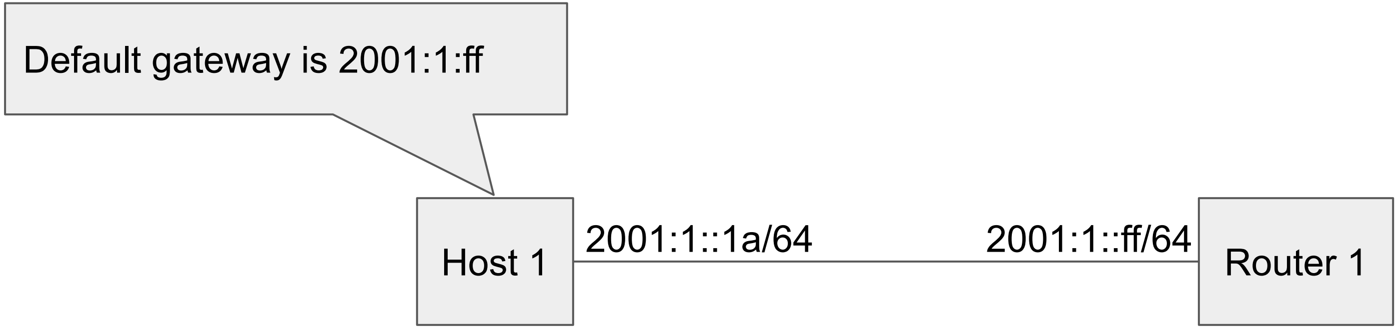
IP unicast routing



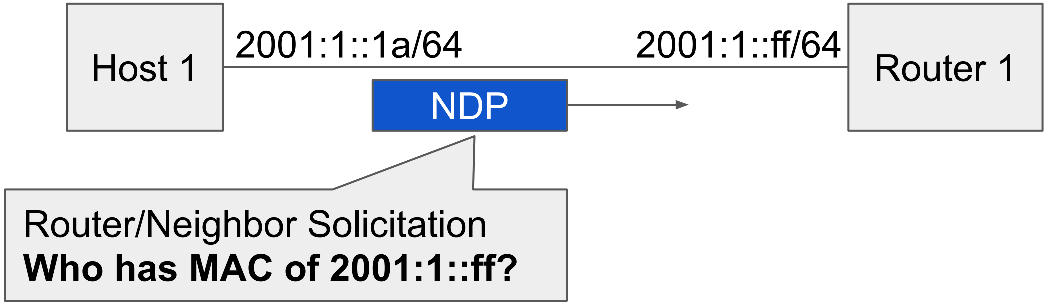
IP unicast routing



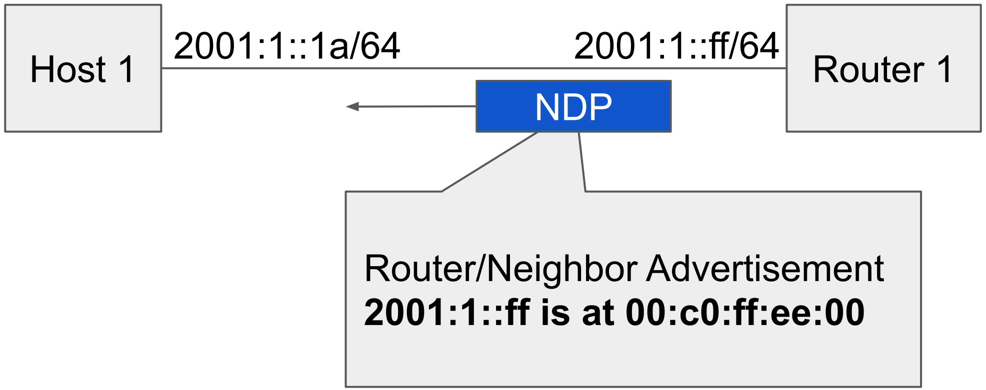
Neighbor Discovery Protocol



Neighbor Discovery Protocol

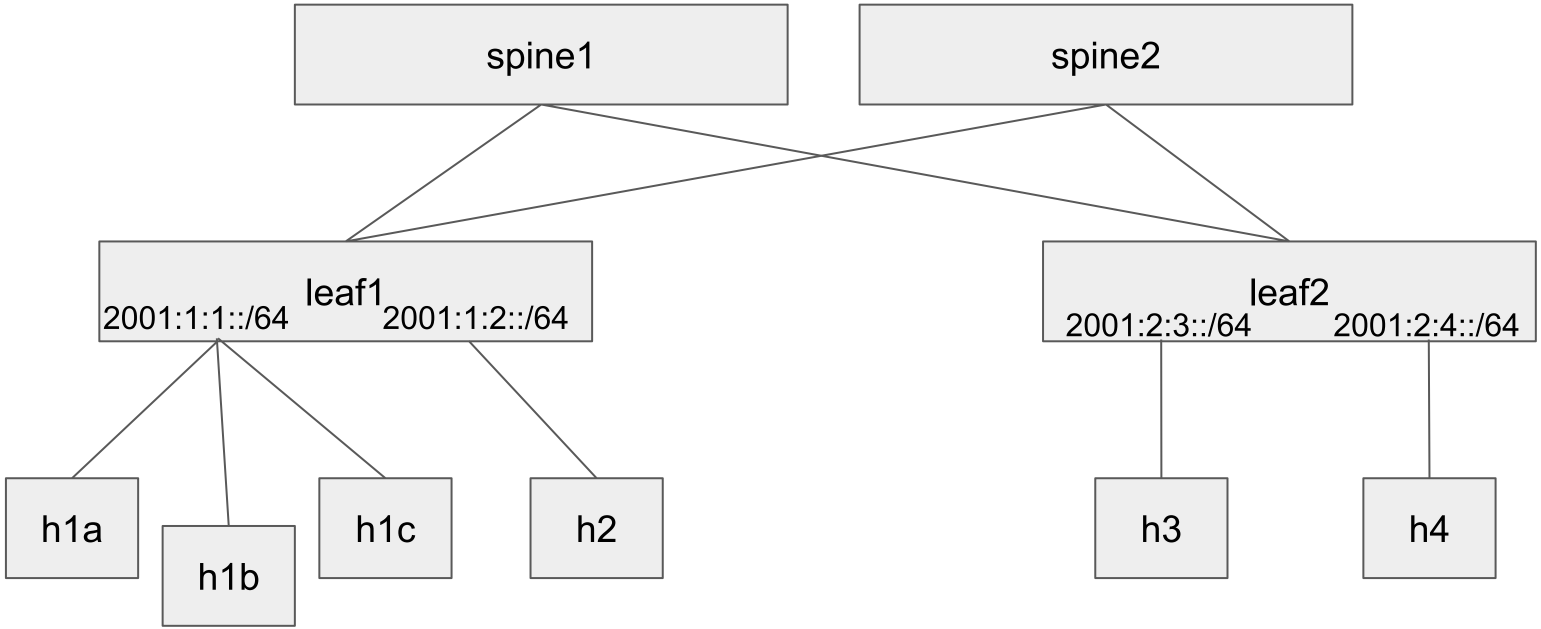


Neighbor Discovery Protocol

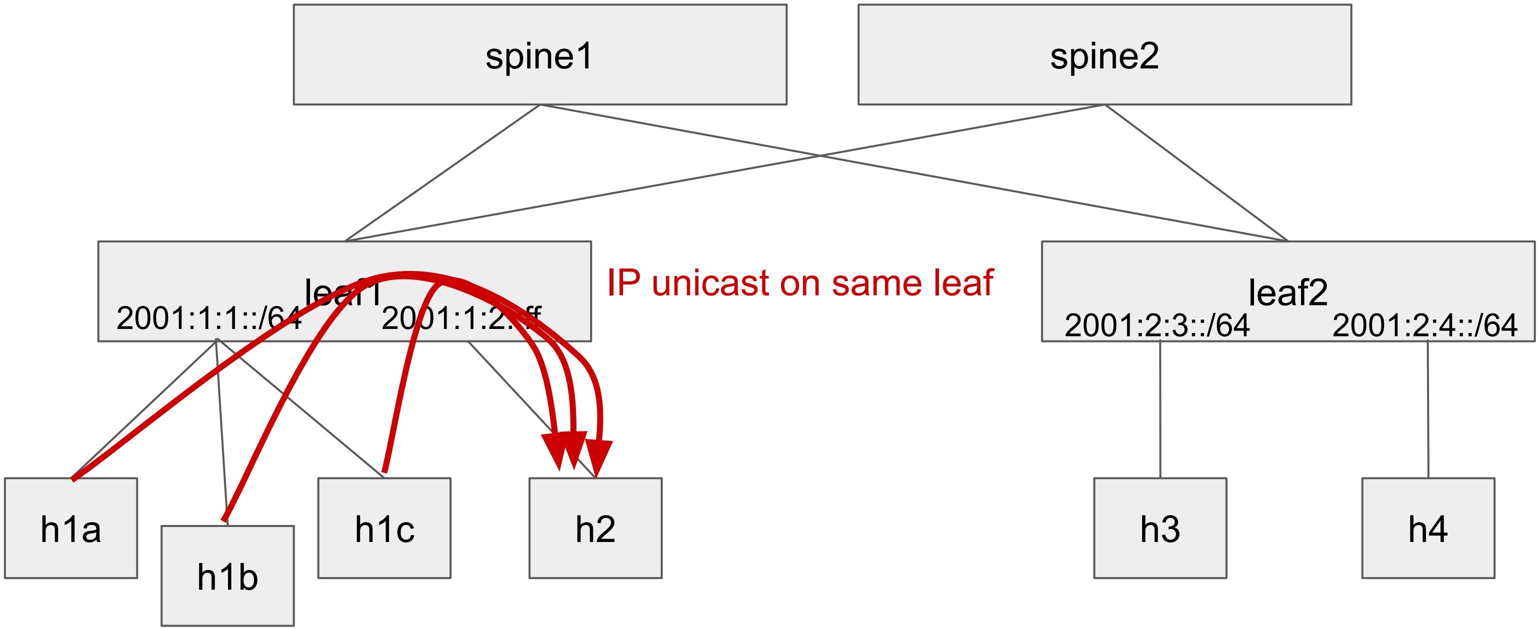


Generated by the P4 program!

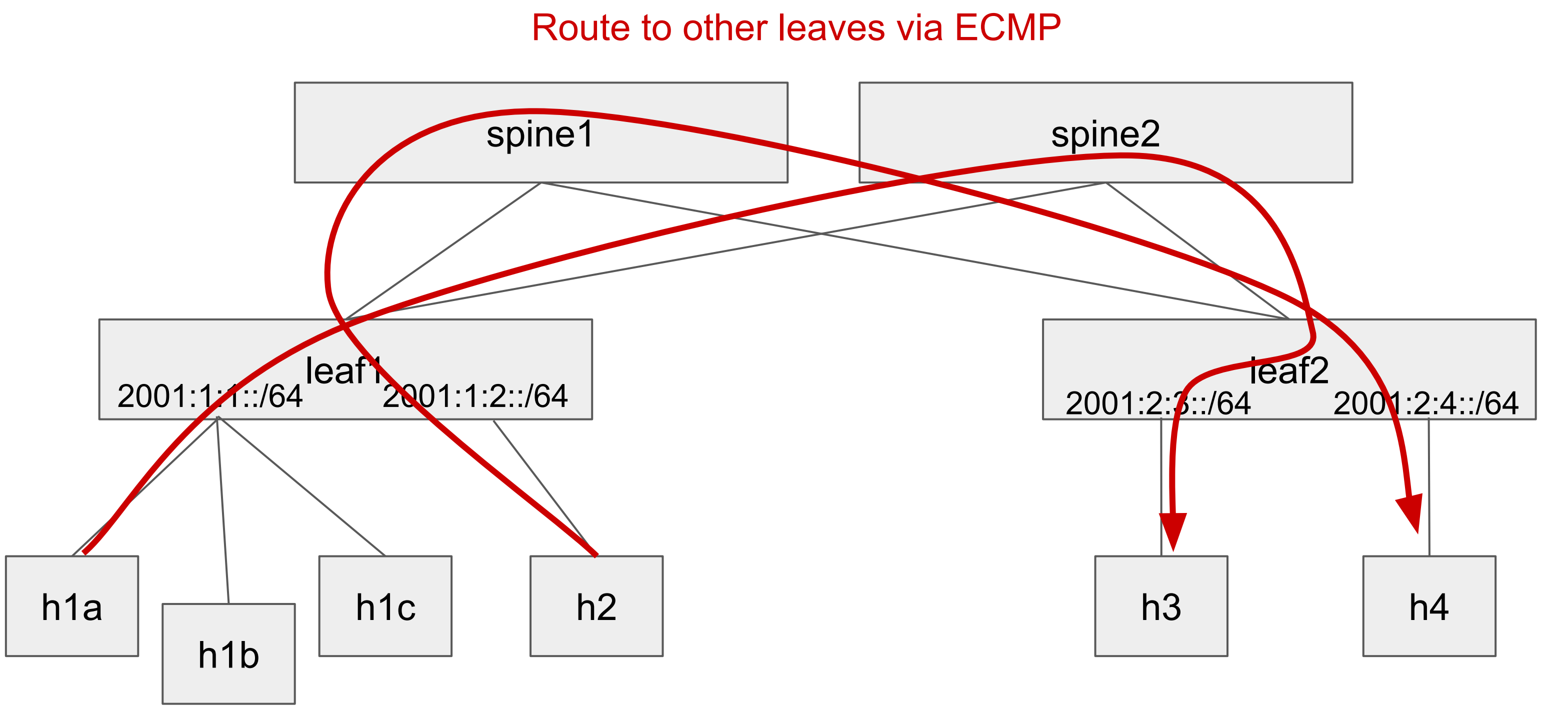
Same-leaf routing



Same-leaf routing



ECMP



Route to other leaves via ECMP
spine1
spine2
leaf1
2001:1:1::/64
2001:1:2::/64
leaf2
2001:2:3::/64
2001:2:4::/64
h1a
h1b
h1c
h2
h3
h4

Exercise 3: Overview

Add tables to P4 program to handle routing of IPv6 packets

Use P4 action selector groups to provide ECMP

Use PTF to verify your P4 code

Modify the ONOS IPv6 routing app

Test on Mininet

Exercise 3: Get Started

Slides: `http://bit.ly/onos-p4-srv6`

Open:

~/tutorial/EXERCISE-3.md

Or use GitHub markdown preview:

`http://bit.ly/onos-p4-srv6-repo`

Solution:

`~/tutorial/solution`

Extra Credit:

- Use ONOS path service to compute paths for routes
- Feed routes from a dynamic protocol agent (e.g. IS-IS, BGP, etc.)

Update tutorial repo (requires Internet access)

```
cd ~/tutorial
git pull origin master
make onos-upgrade
make app-build
```

P4 language cheat sheet:
`http://bit.ly/p4-cs`

You can work on your own using the instructions.
Ask for instructors help when needed.

Exercise 4: Segment Routing v6

Segment Routing Primer

- Segment routing is a source routing method
- Source nodes define a path for the traffic as a list of waypoints (or segments)
- Waypoint (or endpoint) nodes perform basic packet transformation (e.g. popping a label or modifying the destination address), then forward the packet to the next waypoint

Typically, MPLS labels are used to define segments, and the source routing policy is encoded as an MPLS label stack for each packet.

SRv6: What and Why?

- **An SRv6 endpoint is identified using 128 bit address called a Segment Identifier (SID)**
- **SRv6 uses an IPv6 header called the Segment Routing Extension Header (SRH) to encode the list of segments**

SRv6 packets use IPv6 routing tables to forward packets to the next segment, which means there isn't another forwarding database (as is the case for MPLS) and non-SRv6 aware switches can participate in traffic forwarding (between segments).

SRv6 Segment Identifier (SID)

Locator	Function ID	Function Args

Locator: used to route packet to the endpoint (waypoint)

Function ID: specifies type of processing to be performed by the endpoint

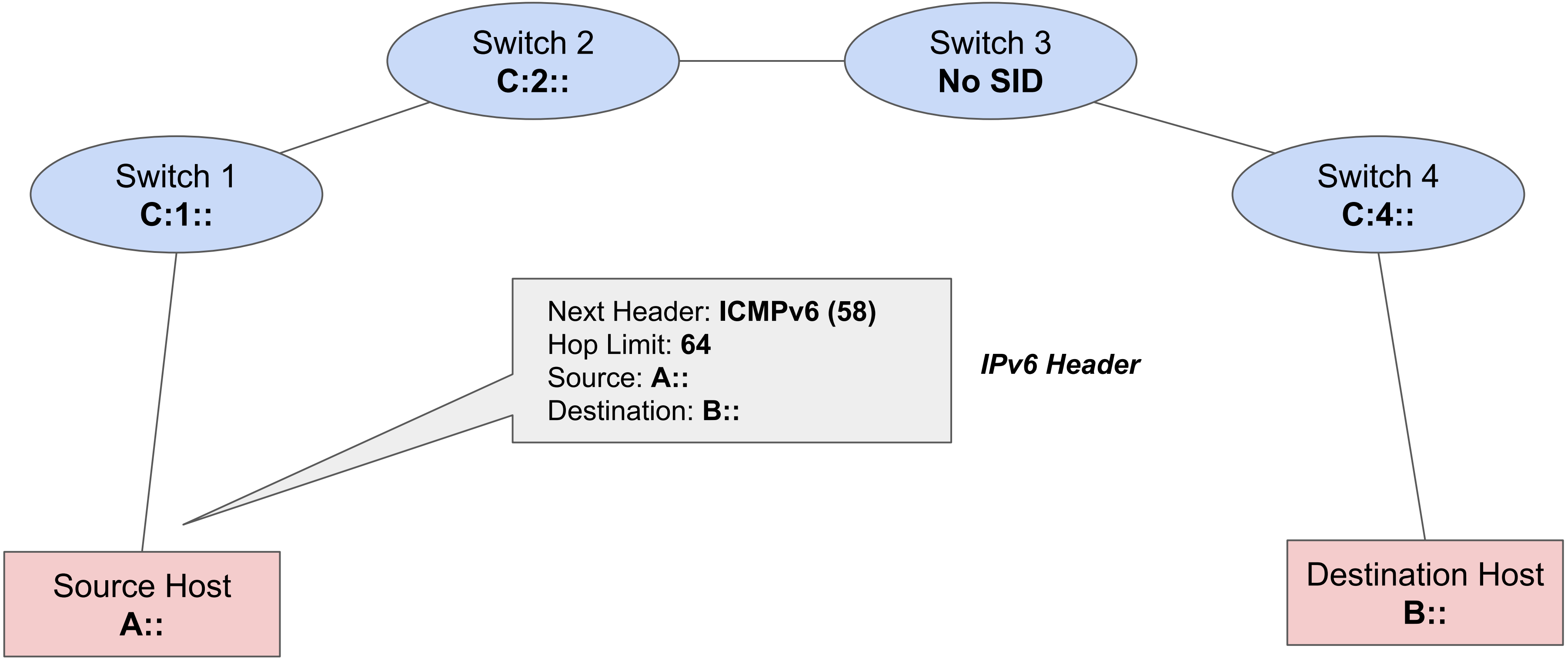
Function Args: (optionally) specified parameters to be interpreted by the function (e.g. VRF ID, customer ID, QoS policy)

The network operator can determine the bit-length for each of these fields in the SID.

SRv6-aware Nodes

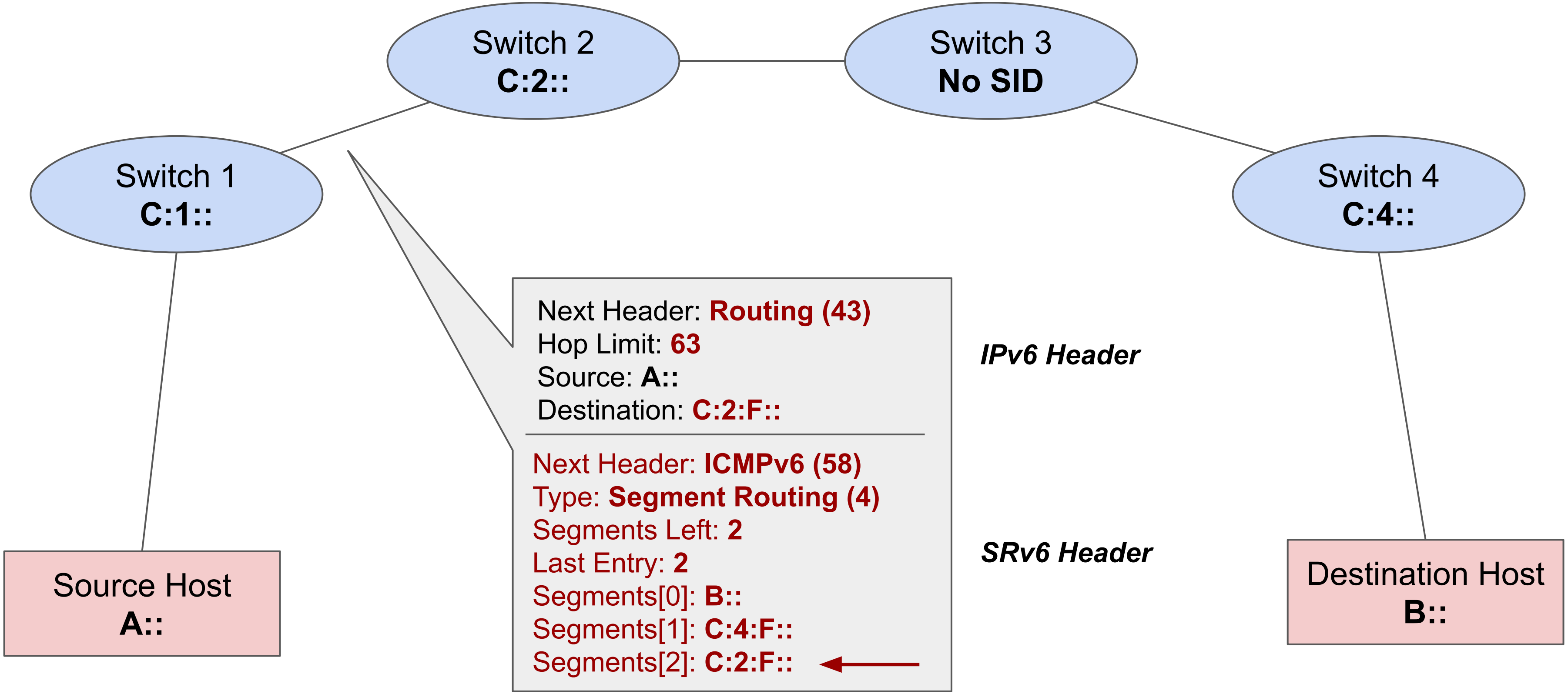
- **Endpoint Node**
  - A participating waypoint in an SRv6 policy that will modify the SRv6 header and perform a specified function
  - Example function: “End”
    - Decrease the segments left field, update the IPv6 destination address, and forward the packet
- **Transit Node**
  - A switch that will participate in traffic forwarding, but is not specified in the segment list
  - By default, perform the “T” behavior: forward the packet normally
  - Example function: “T.Insert”
    - Insert an SRv6 policy, update the IPv6 destination address, and forward the packet

Sending a ping from one host to another



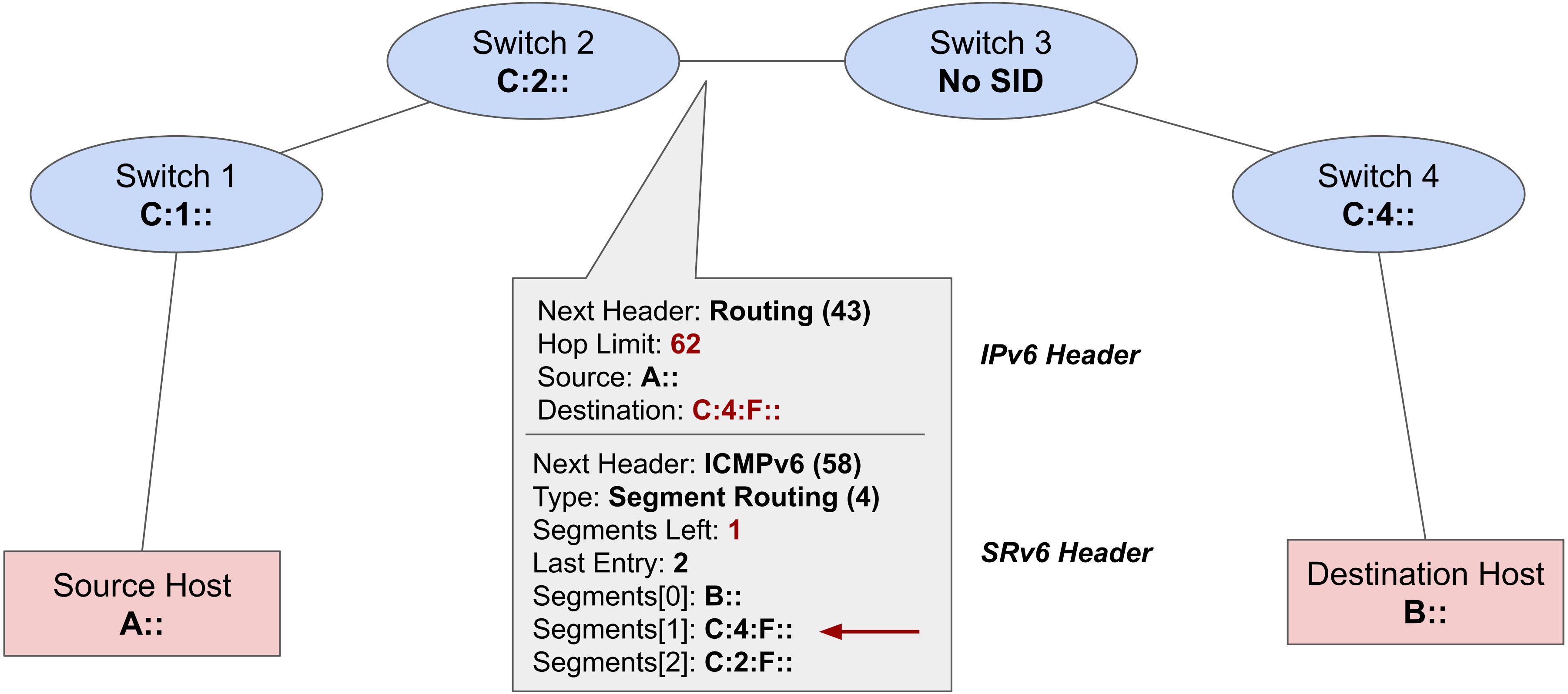
Switch 1 inserts SRv6 policy (T.Insert)

SRv6 Policy: Send traffic to **B::** through **C:2::** (function **F**) and then **C:4::** (function **F**)



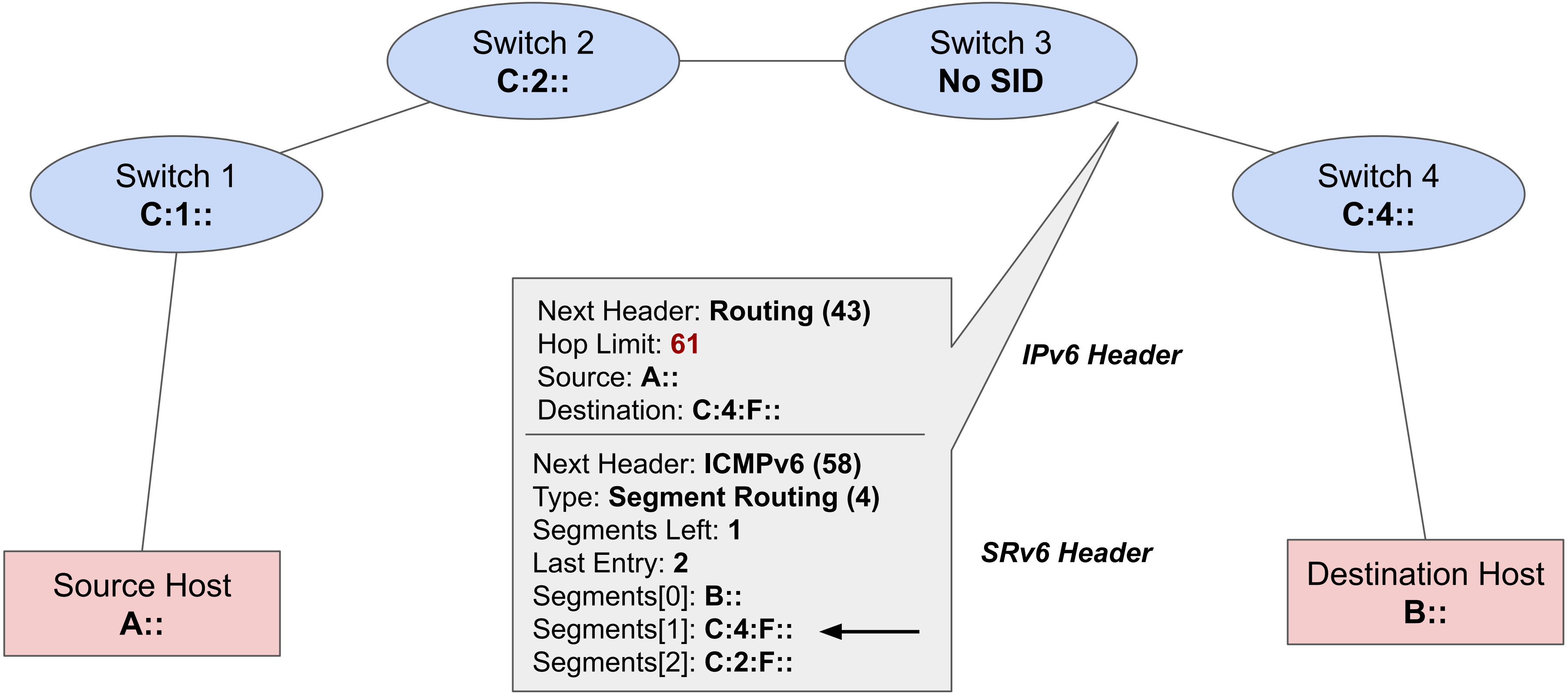
Switch 2 performs End function

Switch 2 (**C:2::**, function **F**) modifies the SRv6 and IPv6 headers, and then forwards the packet



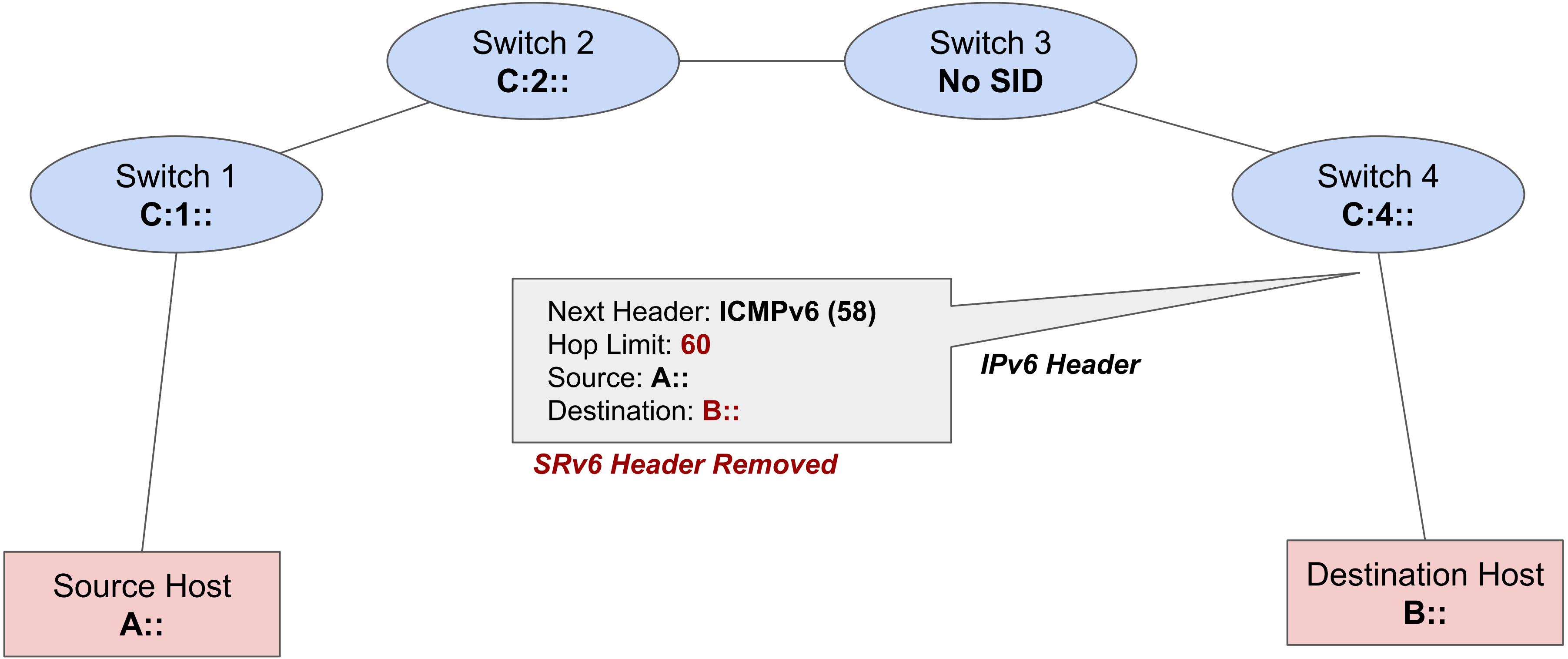
Switch 3 forwards packet normally

Switch 3 simply forwards the packet

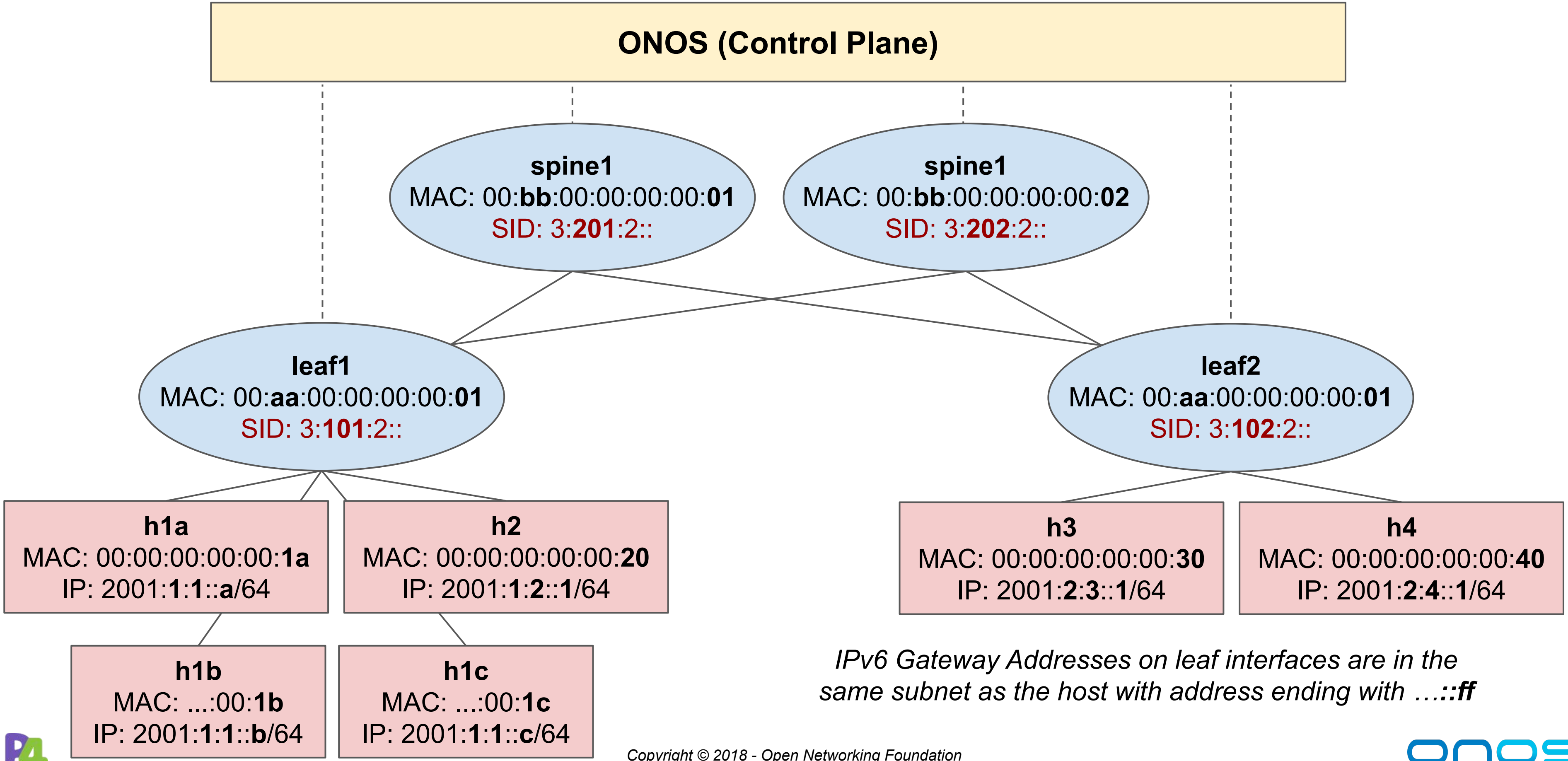


Switch 4 performs End function with PSP

Switch 4 (**C:4::**, function **F**) modifies the SRv6 and IPv6 headers, pops the SRv6 header, and then forwards the packet



Tutorial Topology



*IPv6 Gateway Addresses on leaf interfaces are in the same subnet as the host with address ending with …**::ff***

Exercise 4: Overview

Add support for SRv6 endpoint and transit functionality in the P4 program

Populate the endpoint (srv6_my_sid) table with an entry that matches the switch's SID

Complete the function that creates SRv6 policy rules in the transit table so that you can insert new policies using the ONOS CLI

Verify that traffic is being forwarded via the SRv6 policy using the ONOS UI and Wireshark

Exercise 4: Get Started

Slides: `http://bit.ly/onos-p4-srv6`

Open:

~/tutorial/EXERCISE-4.md

Or use GitHub markdown preview:

`http://bit.ly/onos-p4-srv6-repo`

Solution:

`~/tutorial/solution`

Update tutorial repo (requires Internet access)

```
cd ~/tutorial
git pull origin master
make onos-upgrade
make app-build
```

P4 language cheat sheet:
`http://bit.ly/p4-cs`

You can work on your own using the instructions.
Ask for instructors help when needed.

Summary

What we did:

- Implemented Packet I/O, L2, L3 and SRv6 in P4
- Wrote a unit tests using *PTF*
- Controlled the pipeline using an *ONOS* application
- Tested the pipeline using *Mininet* and *Stratum-BMv2*

Have ideas about improving / extending the tutorial?

Send pull requests!