Chair for Network Architectures and Services
Department of Informatics
TU München – Prof. Carle

Software Defined Networking

2014

Cornelius Diekmann

Contents

- Motivation
- Software Defined Networking – The Idea
- Implementation
  - A Network Operating System
  - OpenFlow
  - The Hardware
- Programming the Controller
- Programming Languages for Software Defined Networks
- Conclusion

Motivation

Mastering Complexity and Extracting Simplicity

A thought experiment

- How would you develop a tool to manage data on an USB stick? On a bare metal machine!

1) Write Input/Output function in assembly
2) Write some operating system kernel in C that calls your assembly routines. Implement a file system.
3) Add an operating system kernel API that allows to start userland processes that can access the file system
4) Write userland tools such as `cat, cp, mv,` to manage data on your USB stick.

Mastering Complexity and Extracting Simplicity

The thought experiment, conclusion.

- Assembly
  - Low level machine language, you must be a of master complexity
- Languages such as C
  - Easier to program the hardware than in Assembly
  - Is translated to Assembly by a compiler
- Operating systems
  - Provide abstraction of hardware, processes, file systems, …
- Languages such as C or Java
  - You can simply write your program
  - The operating system manages (most of) your resources
  - Some environments even manage your memory for you

Mastering Complexity and Extracting Simplicity

Another thought experiment

- How **do** you manage a network on the link layer?

1) Configure all the forwarding tables
2) Configure all the Access Control Lists
3) … set up state in every device ...
4) Check the connectivity, test, test, `ping, traceroute, nmap`, …. debug, fix, test, …

Compared to the first thought experiment, this is like writing everything in Assembly and debugging it by manually inspecting all the memory!

Mastering Complexity and Extracting Simplicity

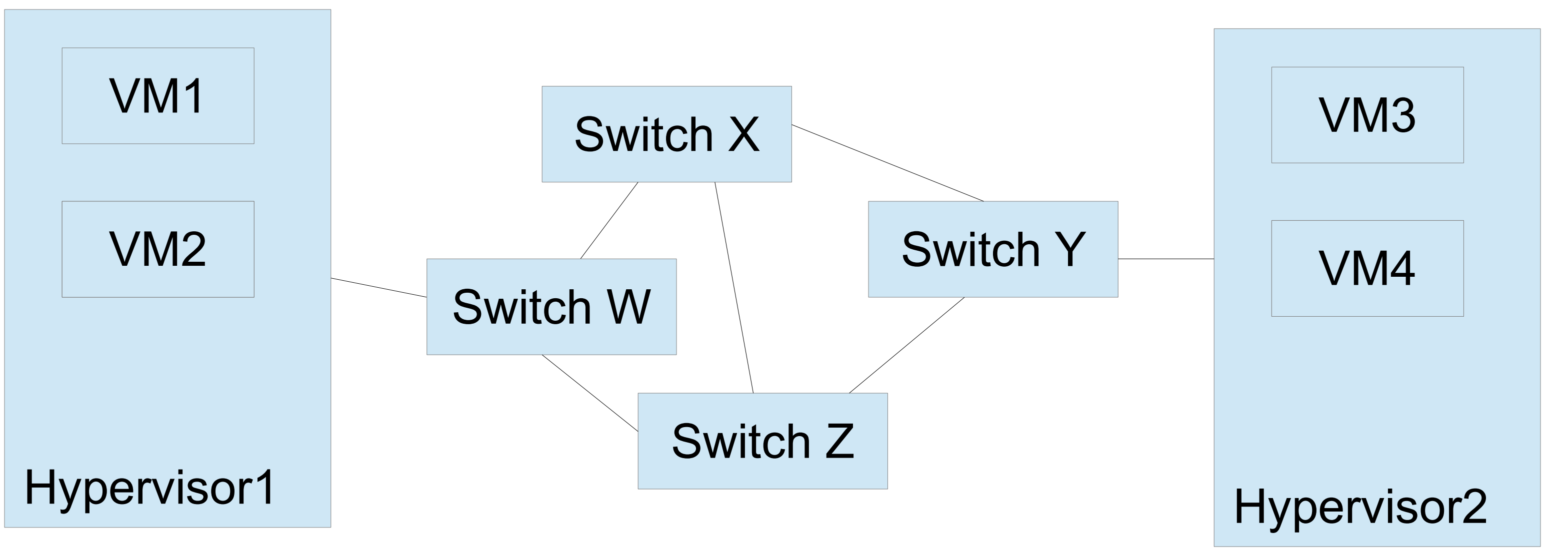
How is link layer connectivity computed?

- E.g. Spanning Tree Protocol (IEEE 802.1D)
- A distributed protocol that deals with distributed state
- May not result in the global optimal solution
- Defines its own protocol format, ...
- Must deal with packet loss, …

- Where are the abstractions?
- How do we influence the resulting connectivity structure?
- Can't we manage it centrally?

A possible SDN use-case (1)

- In your datacenter, you know your traffic flows. It is your datacenter!
- How can you optimize your traffic flows?
  - VM1 to VM3 should flow via W → X → Y
  - VM2 to VM 4 should flow via W → Z → Y



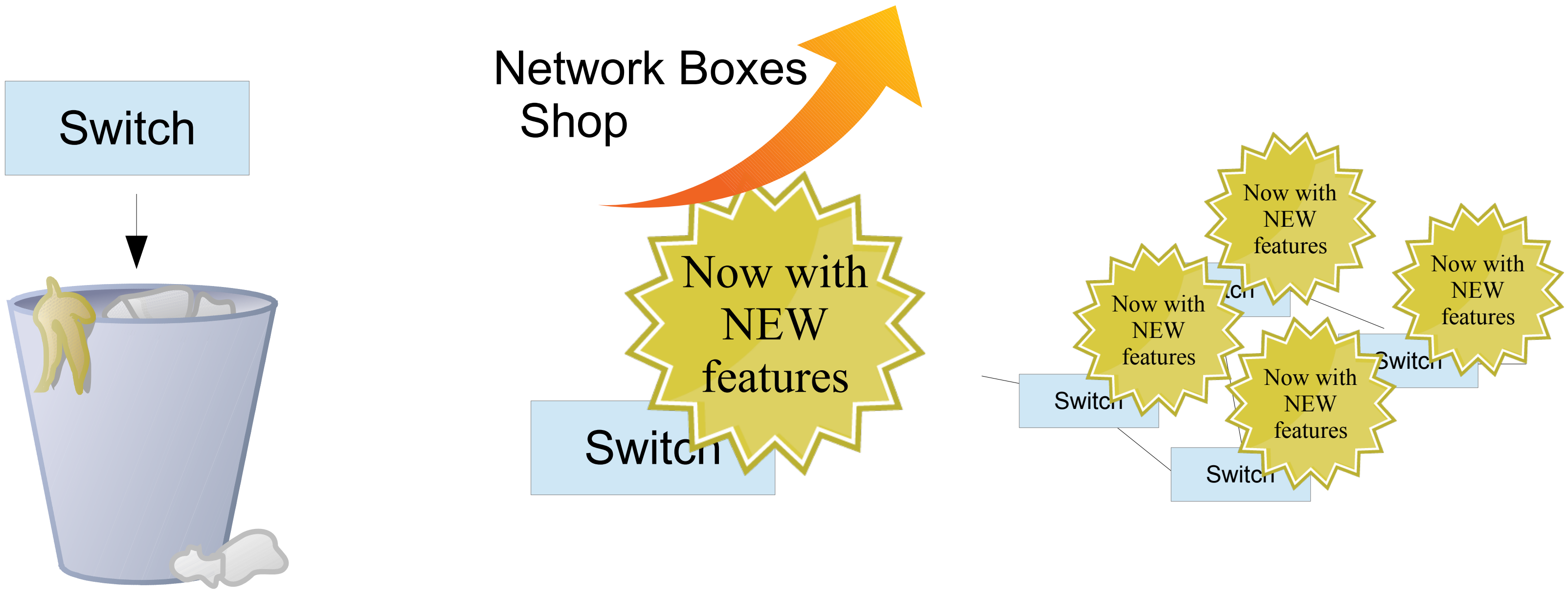
A possible SDN use-case (2)

- You want a load balancer at a switch

Step 1:
throw away

Step 2:
Buy new

Step 3:
Deploy



Step 4:
Repeat until the feature is standardized and understood by all your boxes

Conclusion - The Problem

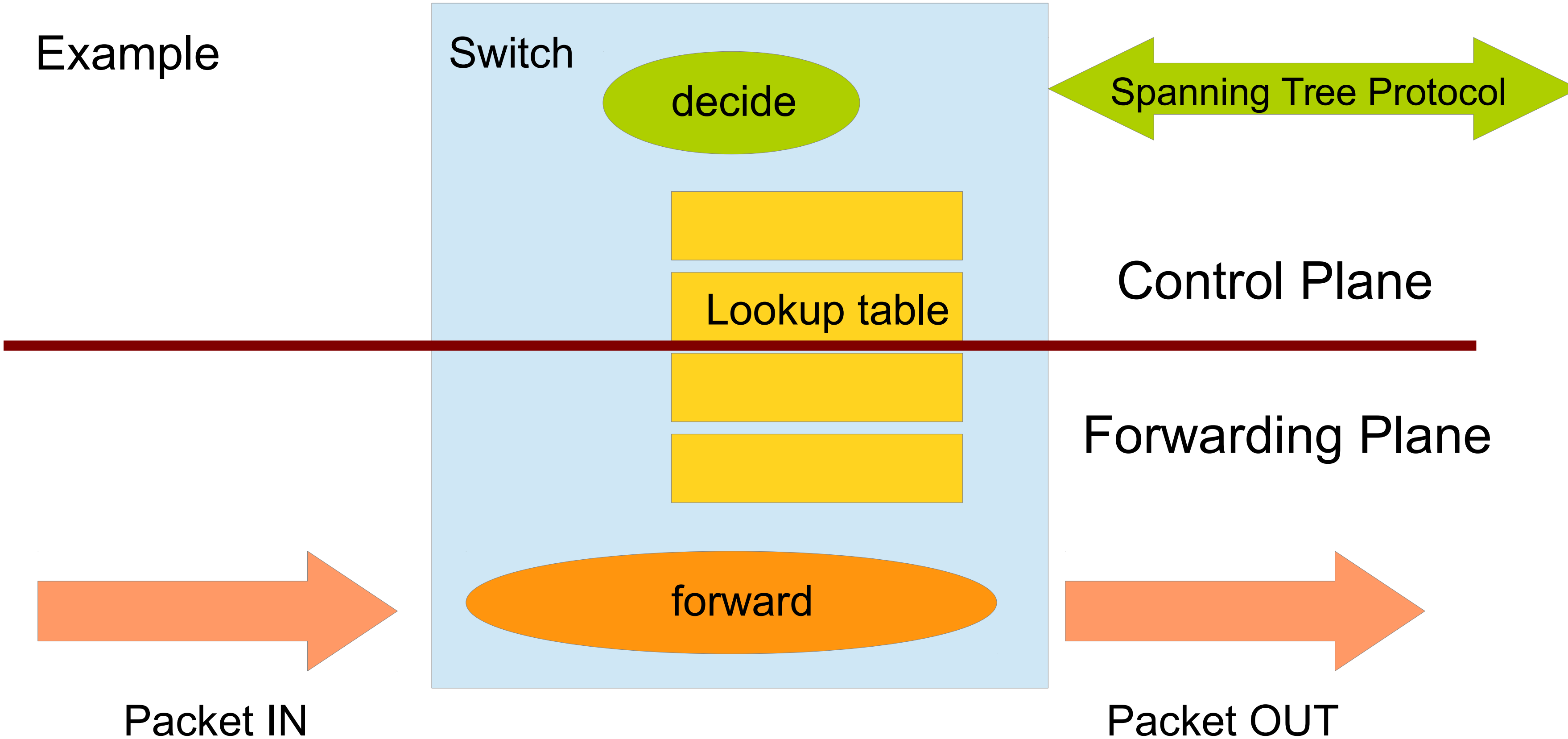
- The use cases
  - A centrally managed network
  - With scenario-specific requirements
    i.e. plugging together some switches is not enough, we have specific requirements that should be implemented by the switches
- The problem:
  - No abstractions or layers
  - No easy-to-use high-level APIs
  - No comfortable way to centrally manage your network
  - Slow innovation
- The idea to solve the problem
  - Software defined networking

Preliminaries

Forwarding Plane and Data Plane

- Forwarding plane: Forwards packets
  - E.g. according to rules
- Control plane: Makes the decision what to do with packets
  - E.g. sets up forwarding plane rules

Example



Software Defined Networking

The Idea

Software Defined Networking (SDN)

- What is SDN?

A network in which the control plane is [...] separate from the forwarding plane

and

A single control plane controls several forwarding devices.

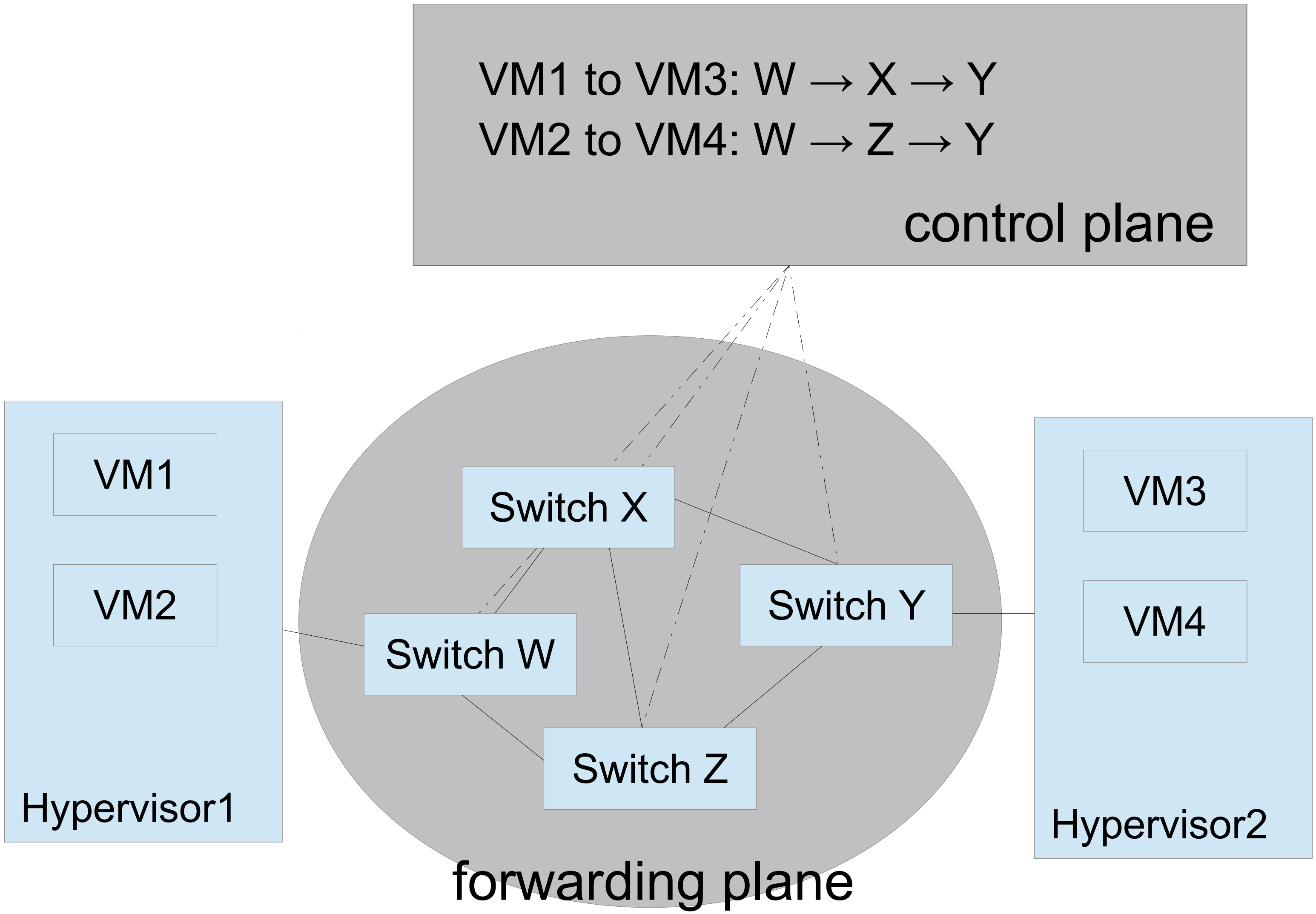
[Keown13]

- Forwarding plane: Forwards packets
  - E.g. according to rules
- Control plane: Makes the decision what to do with packets
  - E.g. sets up forwarding plane rules

Single Control Plane

- *Logically* single control plane
- Not logically distributed
- Can be physically distributed
  - Resilience ...

Example



VM1 to VM3: W → X → Y
VM2 to VM4: W → Z → Y
control plane
VM1
VM2
Hypervisor1
Switch X
Switch W
Switch Y
Switch Z
VM3
VM4
Hypervisor2
forwarding plane

SDN Benefits

- Why the term `Software Defined'?
  - The control plane is just software.
- Abstraction
  - No distributed state, there is a **global network view** centrally at the control plane
  - No need to configure each forwarding plane device manually. Everything can be **managed centrally** at the control plane
  - Simple forwarding plane device configuration. A forwarding plane device model (like a **high-level API**) can be used to configure the devices. No need to develop a separate protocol, deal with packet loss, integrity of transferred data, distributed state, ...

SDN Benefits (1): State Abstraction

- No distributed State
- At a central point with a global view is programmed
  - Complex protocols such as the Spanning Tree Protocol are no longer necessary
  - E.g.: A simple Dijkstra algorithm suffices
    - No more 10k LOC to implement link state routing protocol
  - Globally optimal solutions can be computed
- Complexity is removed from the control plane

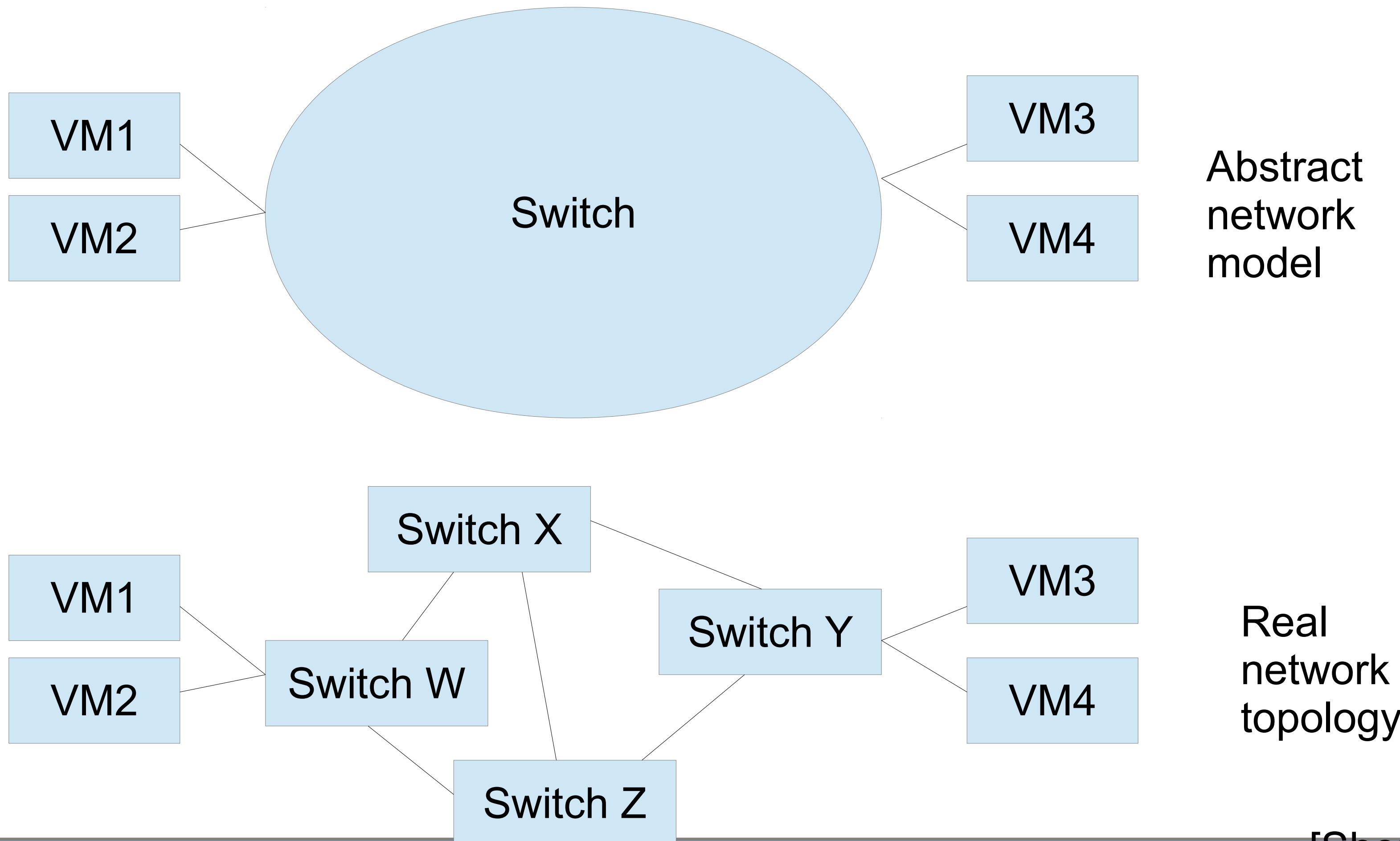
SDN Benefits (2): Specification Abstraction

[Shenker11]

- Control program **should** express desired behavior
- It **should not** be responsible for implementing that behavior on physical network infrastructure
- Natural abstraction: simplified model of network

SDN Benefits: Specification Abstraction - Example

Access Control
Desired behavior: VM1 cannot talk to VM3



[Shenker11]

SDN Benefits (3): Forwarding Abstraction

- Control plane needs **flexible** forwarding model
- Abstraction **should not** constrain control program
  - Should support whatever forwarding behaviors needed
- It **should** hide details of underlying hardware
  - Crucial for evolving beyond vendor-specific solutions

[Shenker11]

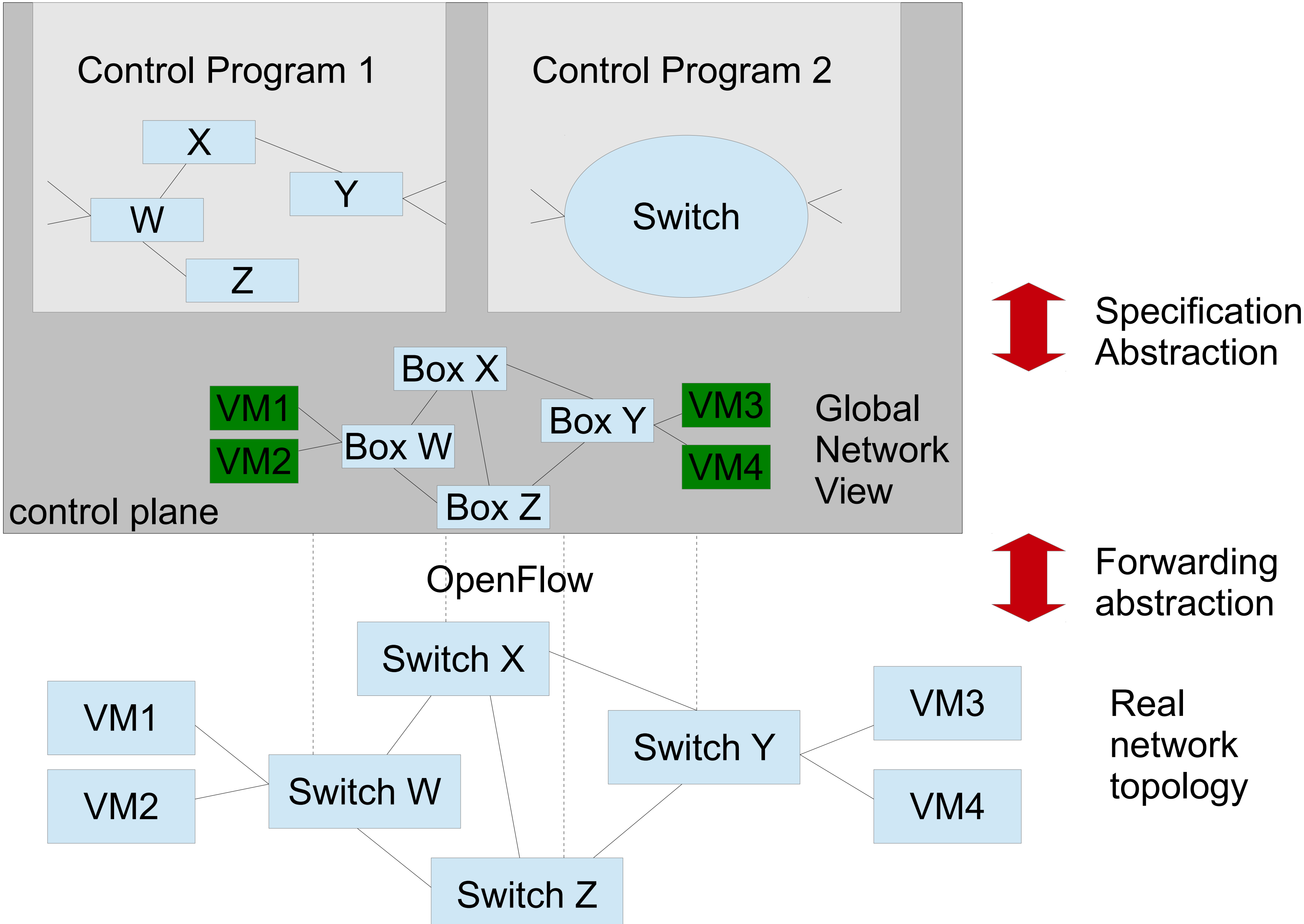
- The same interface for switches from different vendors

- Current standard and realization: OpenFlow

Sum up: Abstractions

- State abstraction
  - Global network view
- Specification abstraction
  - High level API to express desired behavior
- Forwarding abstraction
  - Simple model of forwarding `boxes'

SDN: The Big Picture



SDN: The Big Picture

Bottom-Up

- All forwarding plane devices (switches) are connected to the single control plane
- A common standard (OpenFlow) provides an abstraction such that all forwarding plane devices can be uniformly managed
- Instead of distributed state in the forwarding plane devices, one global network view is available
- Appropriate abstractions (specification abstraction), depending on the desired view, can be utilized to preprocess the global network view
  - E.g. one-big-switch for access control,
  - Complete topology and link speeds for spanning tree calculation
  - Complete topology and link utilization for load balancing
- The control program finally only defines the desired behavior

SDN: The Big Picture

Top-Down

- The control program defines the desired behavior
- The specification abstraction is reversed and maps the control program's output to the global network view / the global state
- A common standard is used to configure the forwarding devices according to the global state

Implementation

A Network Operating System
And
OpenFlow

Disclaimer

We are discussing an idea and scientific prototypes.

There currently exists no network operating system that is as widely used and comparable to common operating systems such as Linux.

Recommended reading (**1 screen page**):
Lee Doyle, *The return of the network operating system (NOS)*, Network World (US), Jan 2013
http://news.idg.no/cw/art.cfm?id=564A6F1B-EF7B-6318-5F43DCBF4BADE856

Recap: What is a `normal' Operating System?

An operating system

- Manages the hardware resources
  - Coordinate access to shared hardware resources.
    E.g. if there is only one printer, print *document1* first, then *document2*, don't try to print them interleaved
  - Manage the hardware
    E.g. put hard disk to sleep if idle, put packets from the NIC to memory, ...
- Ennobles the hardware
  - E.g. you can access /dev/sda as if it were a simple block device.
    You don't have to care about whether it is a SSD, HDD, or raid system
- Provides a standardized API to the hardware resources
  - E.g. you normally don't open /dev/sda, you have a file system to store and access data

What is a Network Operating System?

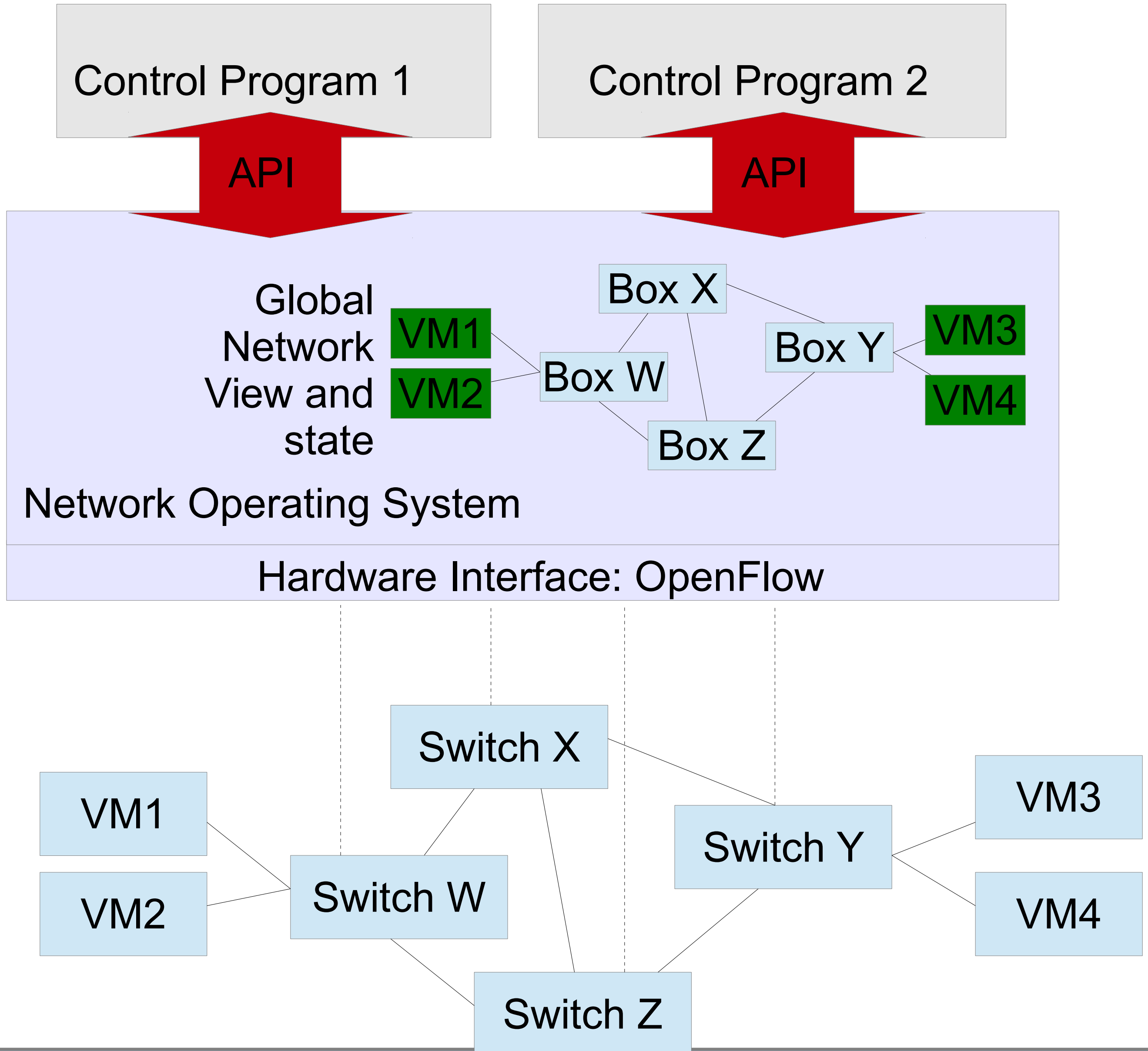
A network operating system

- Manages the hardware resources
  - E.g. all your switches in the network
- Ennobles the hardware
  - The global network view is a central place where all the state is stored and managed. As if there were no distributed state.
- Provides a standardized API to the hardware resources
  - Forwarding abstraction: Simple model of forwarding `boxes'
  - Specification abstraction: High-level API to express desired behavior

Operation of the hardware

- Recap: In a normal operating system
  - A device driver operates the hardware
  - For example via memory mapped areas, I/O instructions, ...
- In a network operating system
  - One needs to deploy the global network view to all the forwarding plane devices
  - The network operating system (control plane) is connected to all forwarding plane devices
  - Via a common protocol, the forwarding plane devices are programmed
  - This protocol is **OpenFlow**

Network Operating System: The Big Picture



Specification Abstraction

Forwarding abstraction

Real network topology

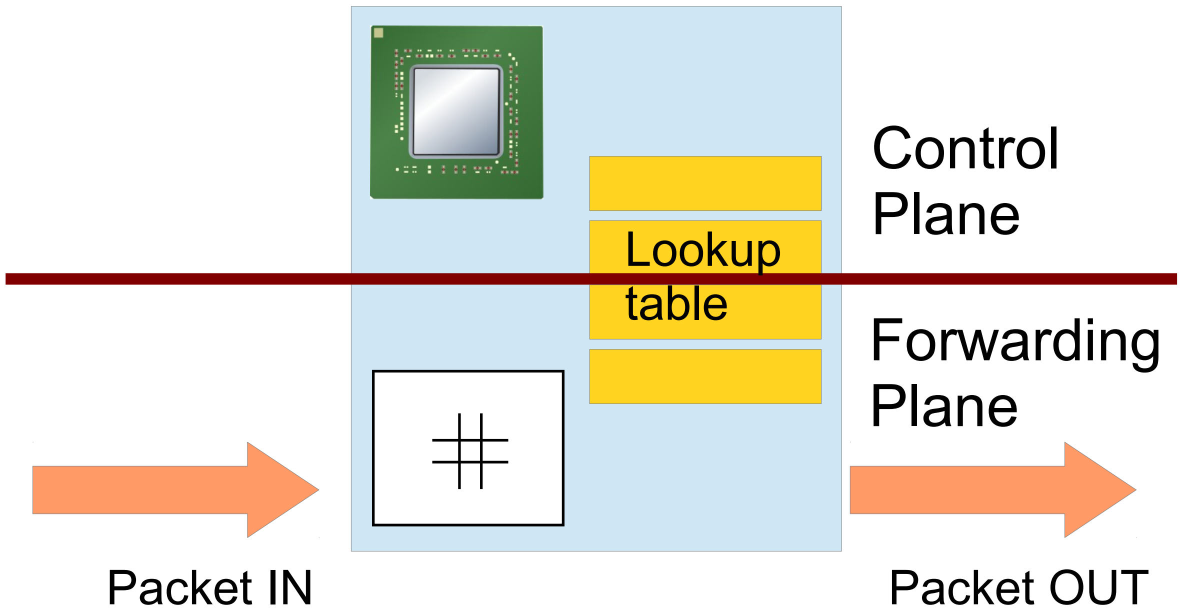
Implementation details

The hardware

Recap: today's common switches

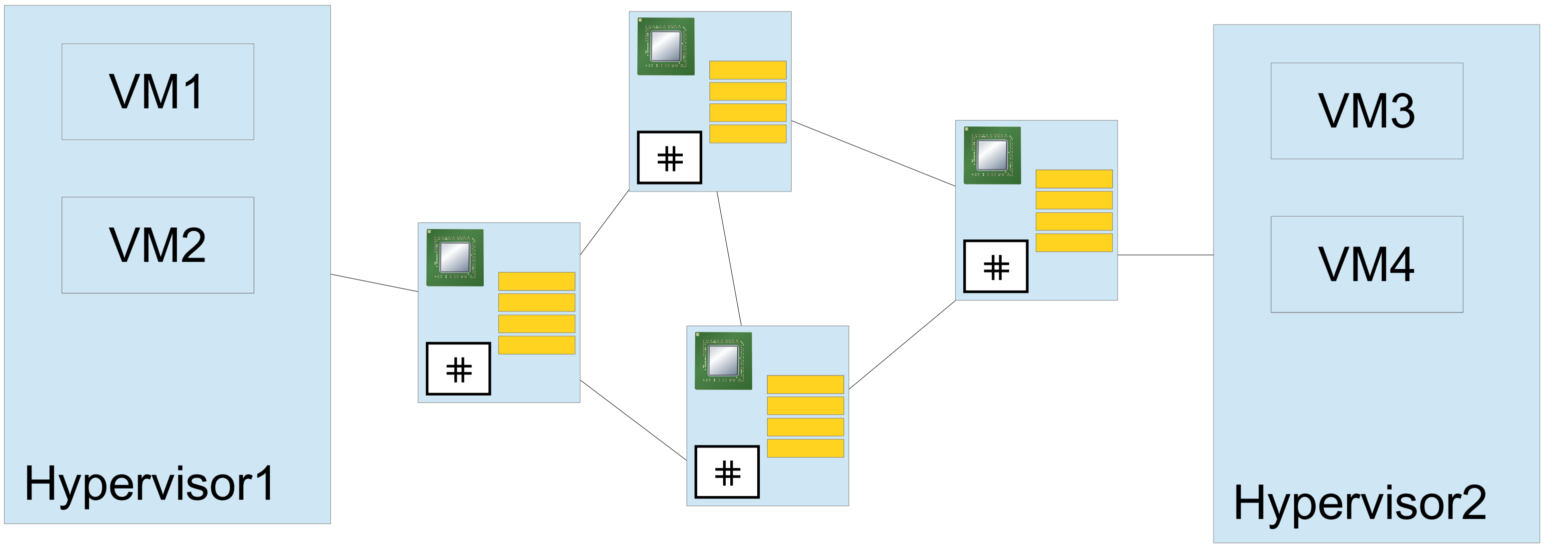
- Forwarding plane
  - Fast ASIC (application-specific integrated circuit)
  - I.e. special forwarding hardware
- Control plane
  - A (more or less) common CPU

Example



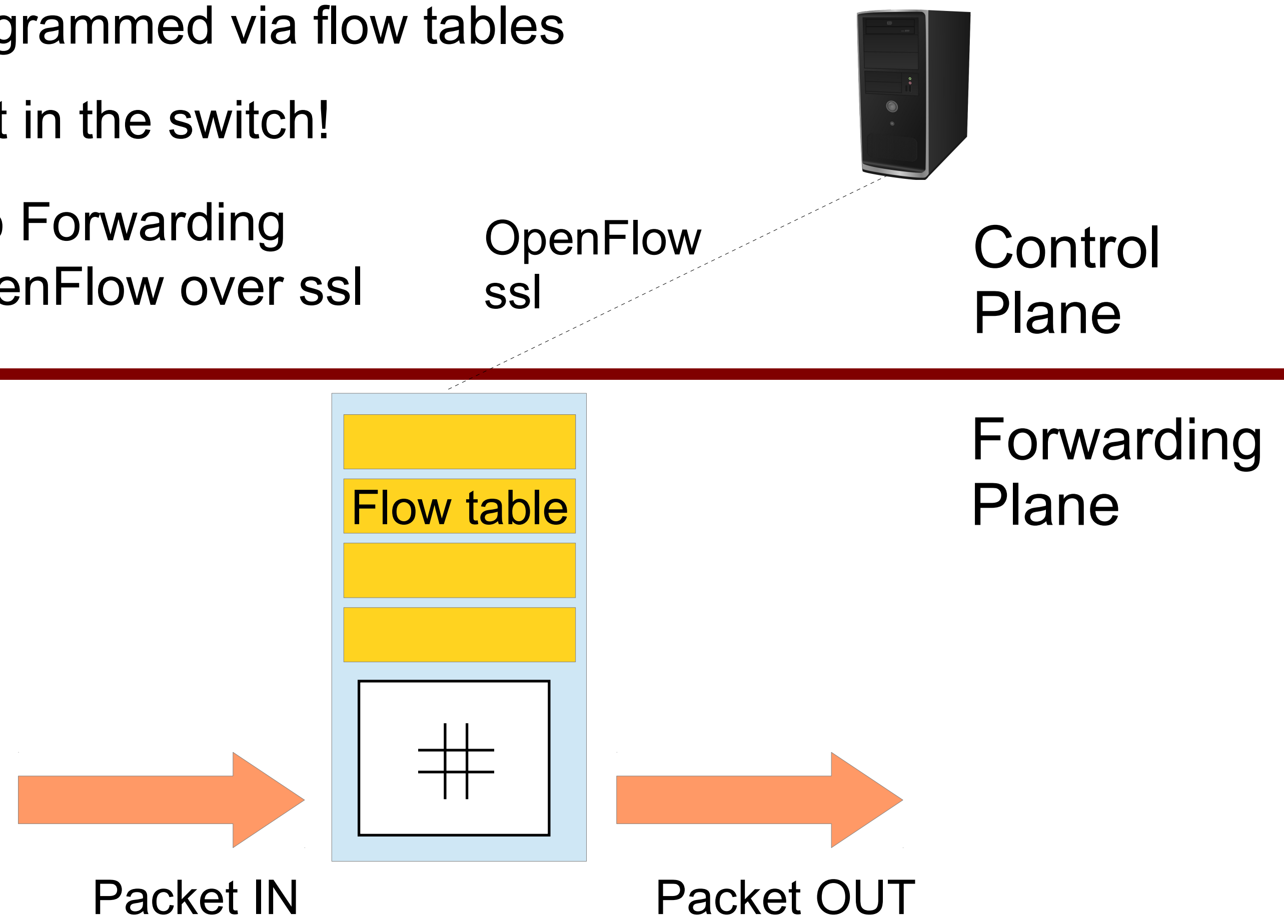
Recap: today's common switches

Forwarding plane and control plane distributed across the network



SDN switches

- Forwarding plane
  - Fast ASIC (application-specific integrated circuit)
  - I.e. special forwarding hardware
  - Behavior programmed via flow tables
- Control plane: Not in the switch!
  - Connected to Forwarding Plane via OpenFlow over ssl



An open switch - Discussion

A switch as open, programmable, forwarding-only platform

- Pros
  - Cheap, simple, fast but stupid devices
  - Allows innovation at software speed
  - Allows experimenting in real-world environments
  - Vendor independence
- Cons (possible vendor point of view)
  - Reveal switch internals
  - Open platforms lower the barrier-to-entry for new competitors
  - Opens the market, price pressure
  - Can sell less added value in their hardware
    Just stupid forwarding devices

An open switch - Discussion

Why not use commodity x86 PCs with Linux as open switches

- Pros
  - Open, available, well-tested
- Cons
  - Low port density.
    Did you recently see a PC with 100+ Ethernet ports?
  - Slow.
    Your memory bus is approximately completely jammed at 10 GB/s

    We can forward 10 GB/s at line speed on a common PC, but not between 100+ ports

→ Special forwarding hardware needed

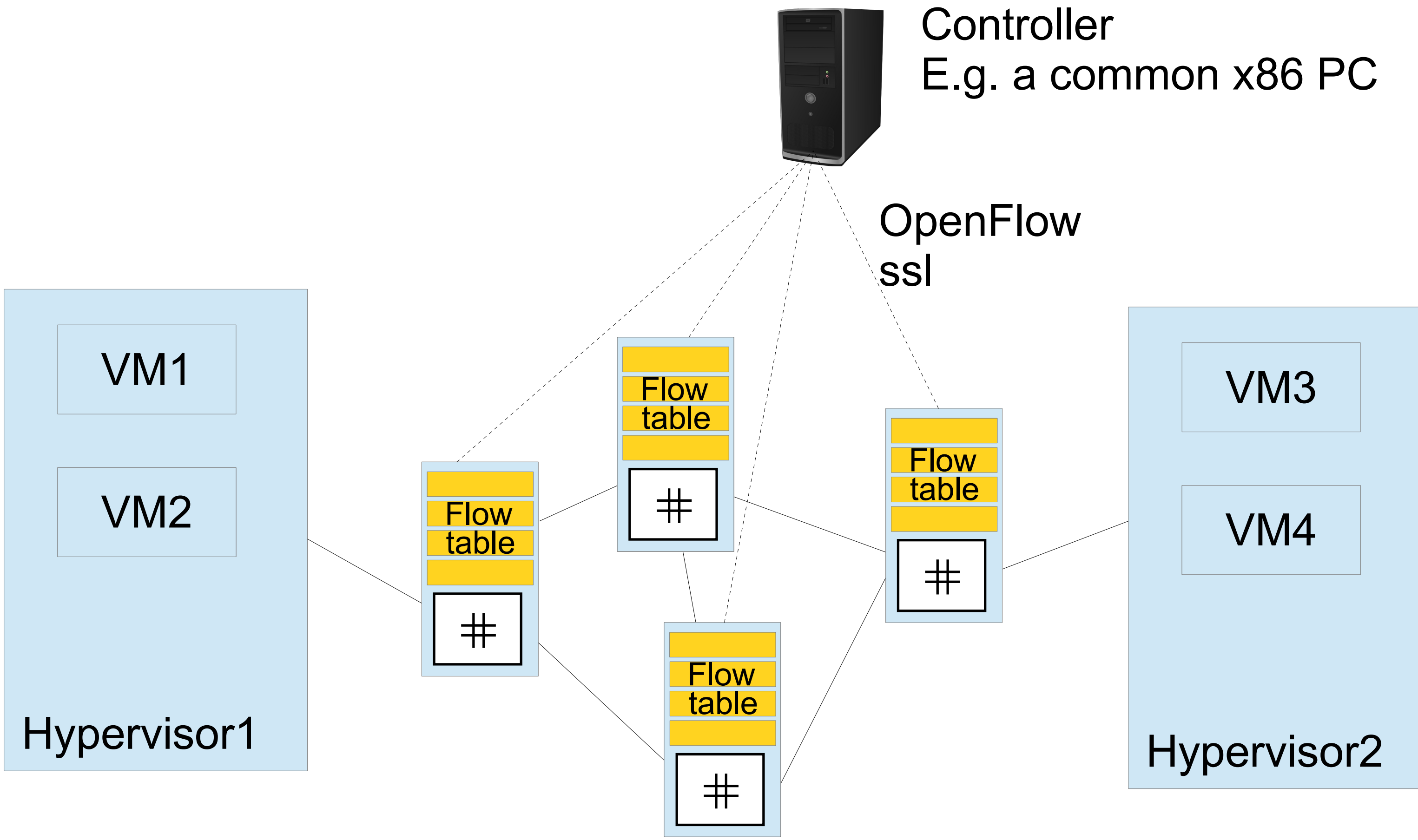
An open switch - Discussion

Why not use a commodity x86 PC as controller

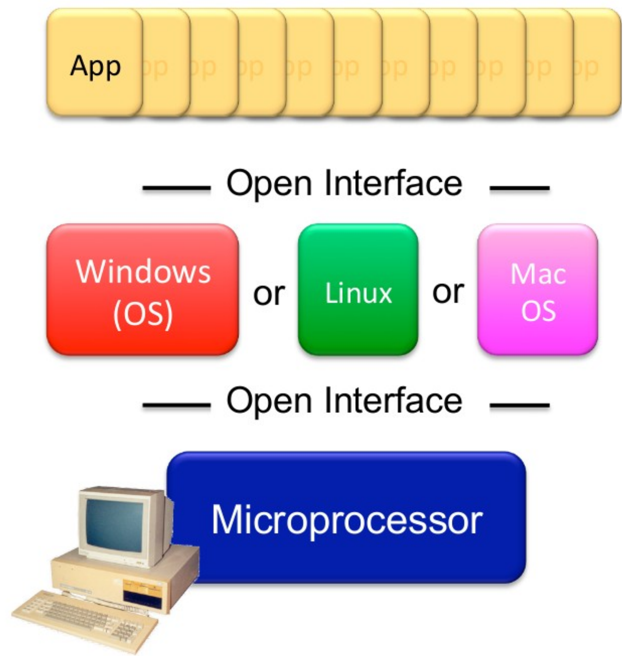
- Pros
  - Open, available, well-tested
- Cons
  - Reliability, but there are means to conquer this

→ You can use a simple x86 PC as your controller!

SDN Hardware: The Big Picture



A comparison: Progress in the Software Industry

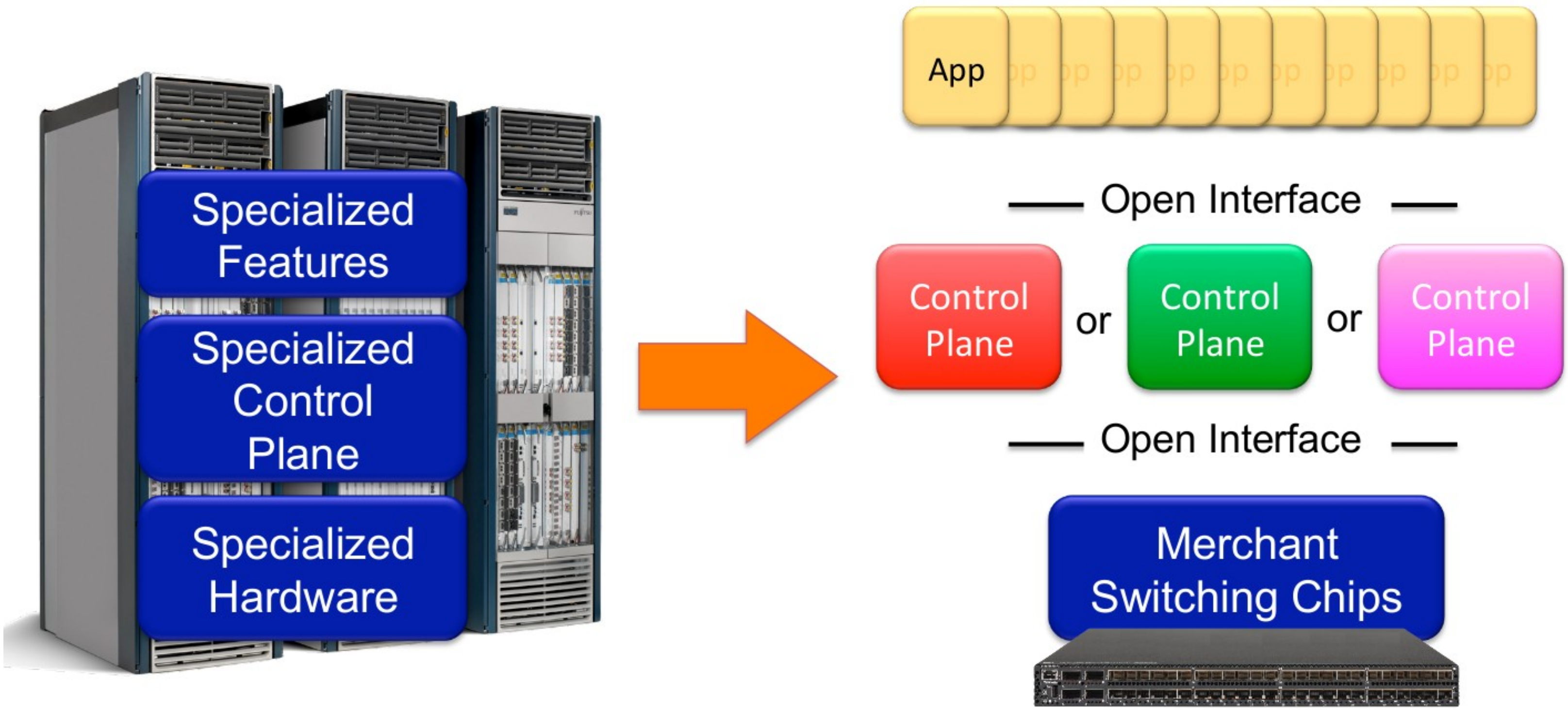


Vertically integrated
Closed, proprietary
Slow innovation
Small industry

Horizontal
Open interfaces
Rapid innovation
Huge industry

Slide borrowed from [Keown13]

The future of Networking?



Vertically integrated
Closed, proprietary
Slow innovation

Horizontal
Open interfaces
Rapid innovation

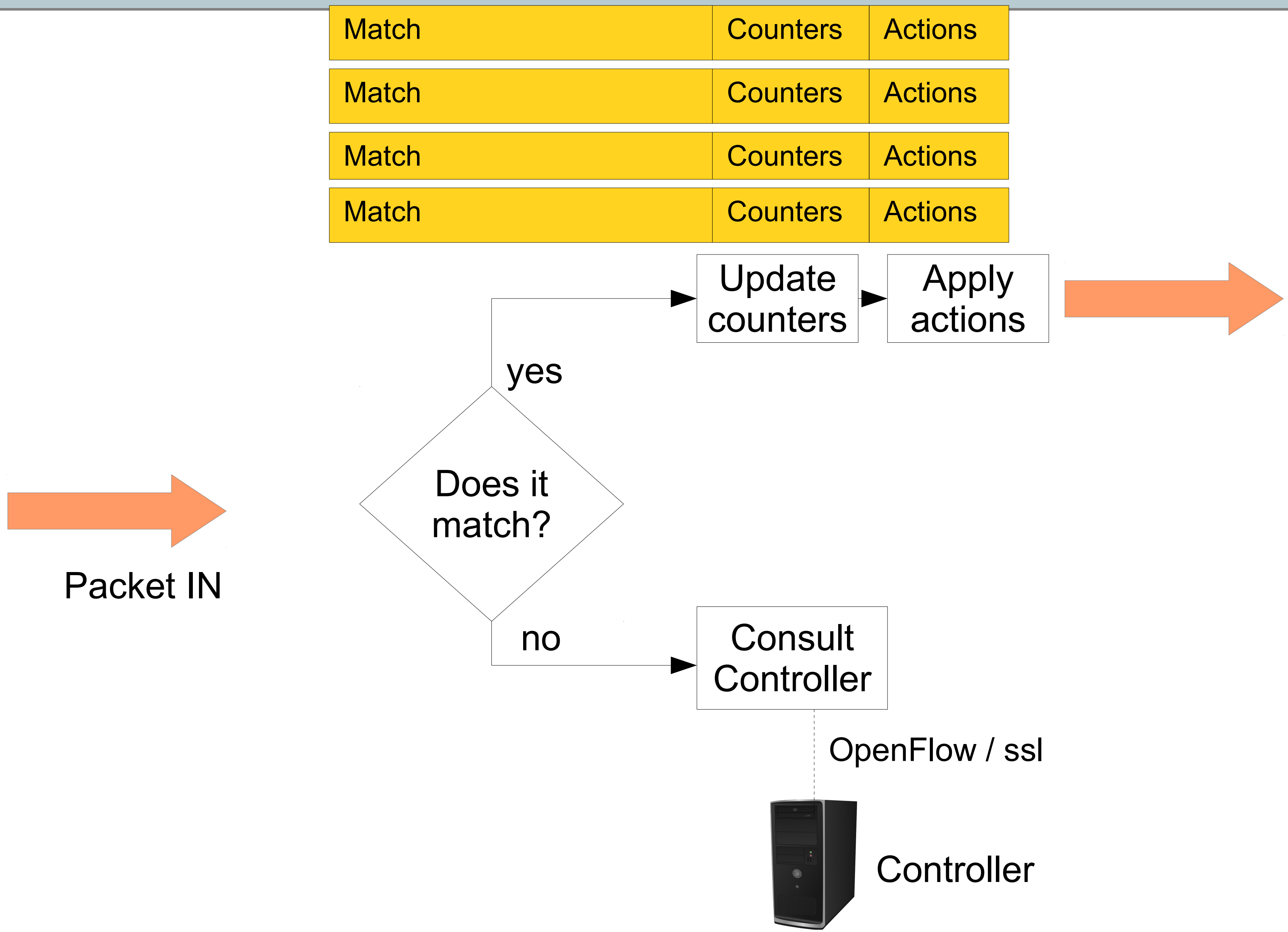
Slide borrowed from [Keown13]

Programming the forwarding plane

OpenFlow

(version 1.0)

OpenFlow Switch Overview



OpenFlow Switches

An OpenFlow Switch consists of three components

- A Flow Table
  - Associates actions with a matching flow table entry
  - E.g. Match(src=1 and dst=2) Action(Forward(Port4))
- A Secure Channel that connects the switch to the controller
  - SSL
- The OpenFlow Protocol
  - An open and standardized way for a controller to program the switch
  - I.e. set up the flow table entries

[OFwp08]

OpenFlow Actions

If a packet matches a flow table entry, the following basic actions can be performed

- Forward
  - Forward packet to a switch's given port(s)
  - Used to move packets through network
- Drop
- Encapsulate
  - Encapsulate packet and send it via the secure channel to the controller
  - The controller decides what to do
  - Sane default setting if packet does not match a flow table entry
  - Controller can install appropriate flow table entry after the first packet of a flow was sent to it

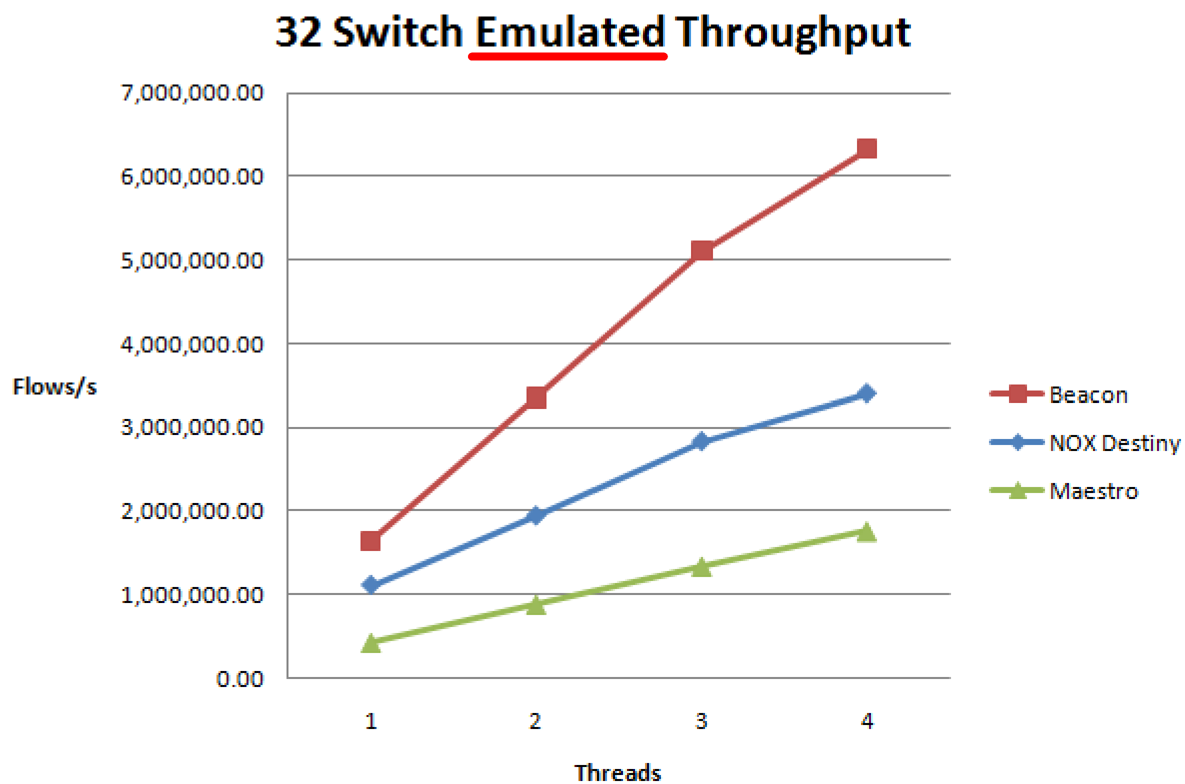
[OFwp08]

The Encapsulate Action

Is the encapsulate action a good choice? Discussion

- Observation
  - There may be many packets in a network but very few flows
  - A flow table entry is installed after the first packet of a flow has been observed
  - Afterwards, all packets that belong to the flow are forwarded by the switch directly
- Possible problems
  - When to delete old flow table entries?
  - How many flow table entries can a switch store? Is it enough?
  - What about attacks?
    Can an attacker send our arbitrary packets that match no flow table entry and thus congest the secure channel or overwhelm the controller?
    Think of port scanning!

OpenFlow Performance



CPU: 1 x Intel Core i7 930 @ 3.33ghz, 4 physical cores, 8 threads

RAM: 9GB

OS: Ubuntu 10.04.1 LTS x86_64

NOX, Beacon, and Maestro are controllers

A Real-World Test

- Analyzing our firewall
- More than 95% of all packets belong to established* connections
- September 2014
- More than 19T of real-world traffic

*
```
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
  16G   19T ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0            state RELATED,ESTABLISHED,UNTRACKED
```

Reminder

Always keep in mind: OpenFlow is just one tiny aspect of SDN and better alternatives may be thought of. But, you can buy OpenFlow switches!

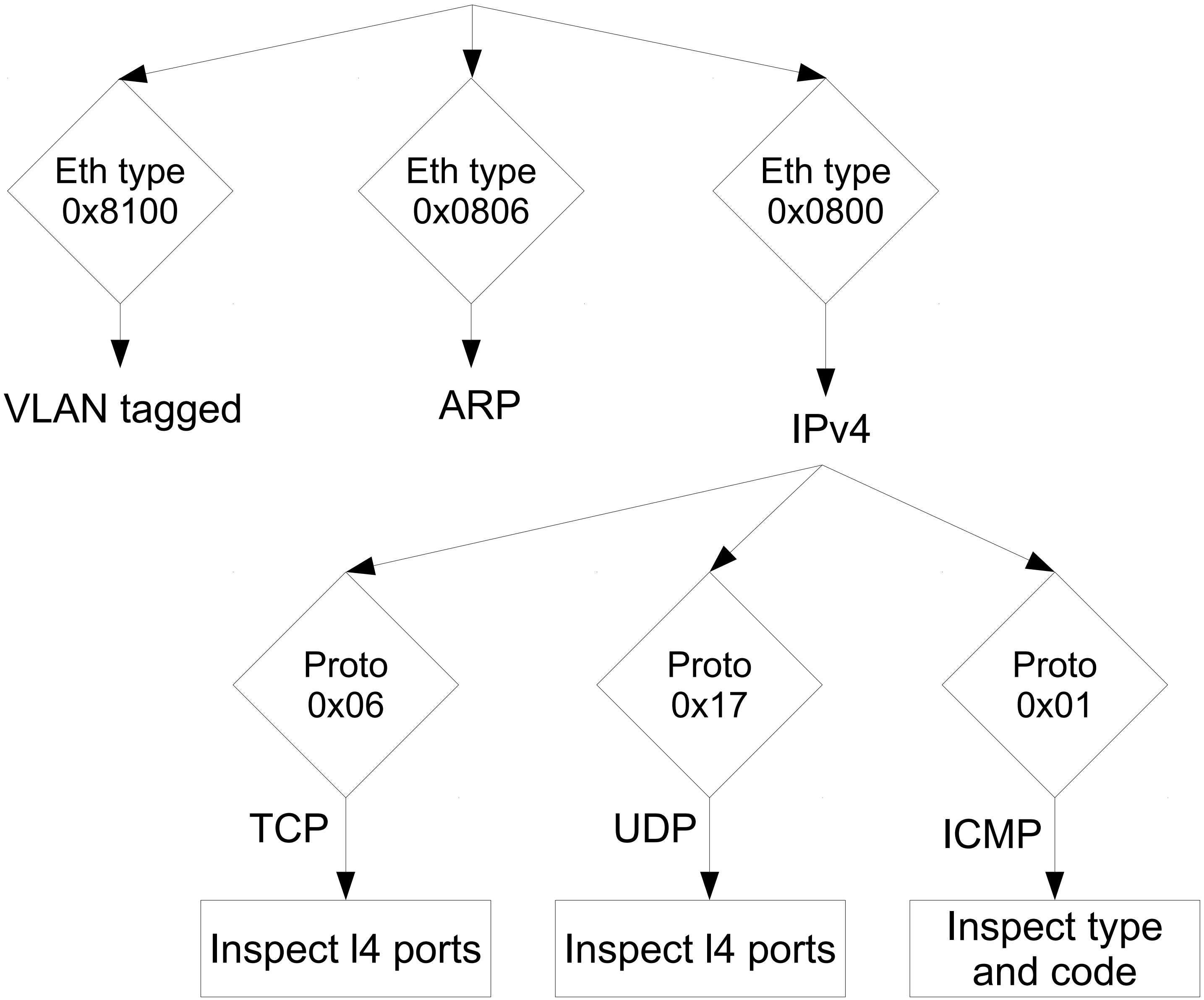
Flow Table Entries

OpenFlow (first generation)

- Match
  - If a packet matches multiple entries, a **priority** decides the match
  - Only one match can apply to a packet but multiple actions can be performed per match
- Counters
  - Statistics: Byte and packet counter per flow
- Actions
  - Zero **or more**

Match										Counters	Actions
In Port	VLAN ID	Ethernet			IPv4			TCP		#pkts #bytes	Fwd Drop Encap ...
		Src addr	Dst addr	Type	Src addr	Dst addr	Proto	Src port	Dst port		

Matching in Detail



based loosely on [of10]

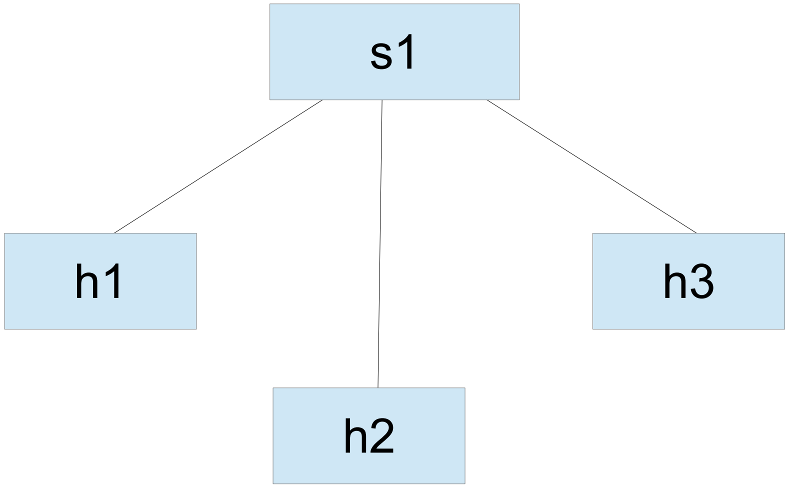
Programming a Controller
Using POX

https://github.com/noxrepo/pox

https://openflow.stanford.edu/display/ONL/POX+Wiki

A Test Network

- `sudo mn --topo single,3 --mac --switch ovsk --controller remote`
- Hosts: h1 h2 h3
- Switch: s1



POX – Switch Identification

- DPID – Data Path IDentifier
- A switch is called a Data Path in the OpenFlow standard
- It has a unique 64 Bit ID
- In the following, dpid uniquely defines a switch in the network

POX – Matches

- in_port
  - Switch port number the packet arrived on
- dl_src
  - Ethernet source address
- dl_dst
  - Ethernet destination address
- dl_vlan
  - VLAN ID
- dl_type
  - Ethertype (e.g. 0x0800 = IPv4)
- nw_proto
  - IP protocol (e.g., 6 = TCP) or lower 8 bits of ARP opcode
- nw_src
  - IP source address
- nw_dst
  - IP destination address
- tp_src
  - TCP/UDP source port
- tp_dst
  - TCP/UDP destination port

- Terminology: data link (dl), network (nw), transport (tp)

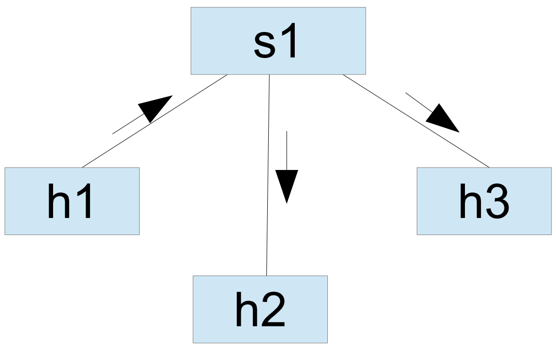
POX – Callbacks

- Called when the switch sends OpenFlow messages to the controller
- Important callbacks
  - `_handle_PacketIn`
    An encapsulated packet is sent from the switch to the controller
  - `_handle_ConnectionUp`
    A (new) switch connected
  - `_handle_ConnectionDown`
    A switch disconnected or restarted or lost OpenFlow connection to controller
  - Other events include: PortStatus, FlowRemoved, ErrorIn, Statistics, ….

A Simple Repeater

Forwards input packets out of all ports

- We react to encapsulated packets
- Step1)
  print debugging output for all received packets

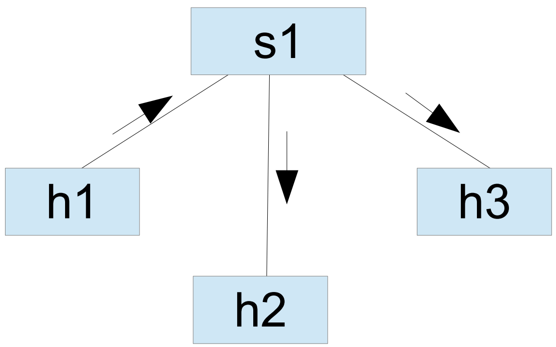



```
def _handle_PacketIn(event):
  dpid = event.dpid
  packet = event.parsed
  print "Switch %s received a packet: %s" % (dpidToStr(dpid), packet.dump())
```

A Simple repeater

Forwards input packets out of all ports

- Step2)
  The controller instructs the switch to FLOOD all incoming packets to all ports
- FLOOD means “send to every port, except the one where the packet was received”

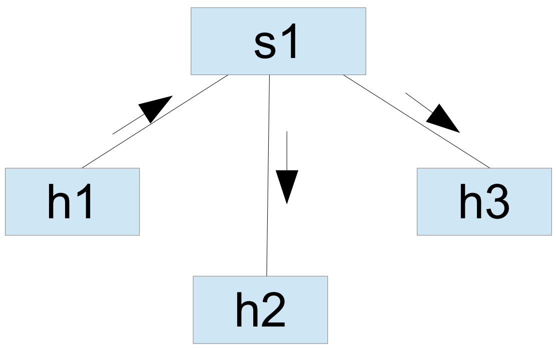



```
def _handle_PacketIn(event):
  dpid = event.dpid
  packet = event.parsed
  print "Switch %s received a packet: %s" % (dpidToStr(dpid), packet.dump())

  msg = of.ofp_packet_out()
  # construct this message from the received event
  msg.data = event.ofp
  msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
  event.connection.send(msg)
```

A Simple Repeater – Test

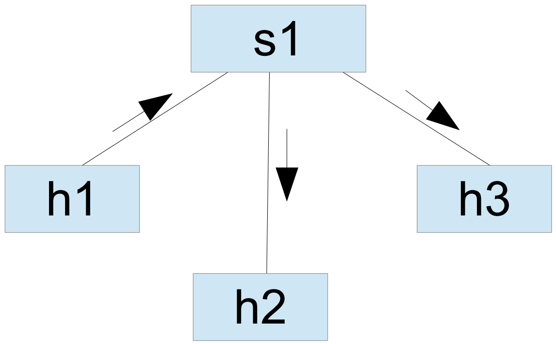
Forwards input packets out of all ports

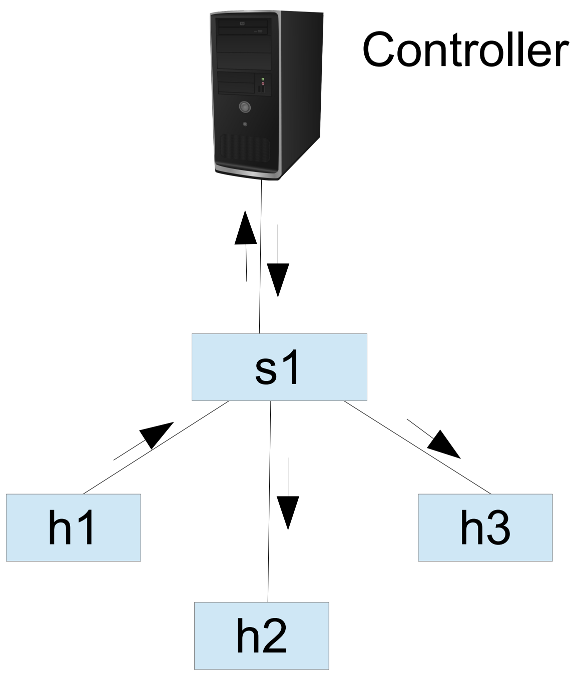
- All hosts are pairwise reachable
- All packets are seen by the controller



A Simple Repeater - Discussion

- This setup is extremely inefficient
- All packets that are received from the switch are forwarded to the controller.
- The controller decides the action
- This is no better than using a common x86 PC as network switch!





An Efficient Repeater

- We install a flow table entry when the switch connects

```
def _handle_ConnectionUp (event):
  print "installing flowtable entries on %s" % dpidToStr(event.dpid)
  msg = of.ofp_flow_mod()
  msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
  event.connection.send(msg)
```

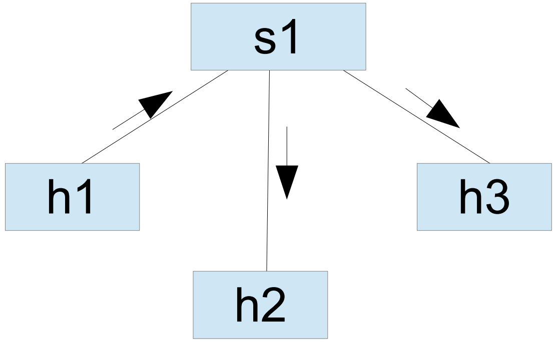
“of.ofp_flow_mod()”

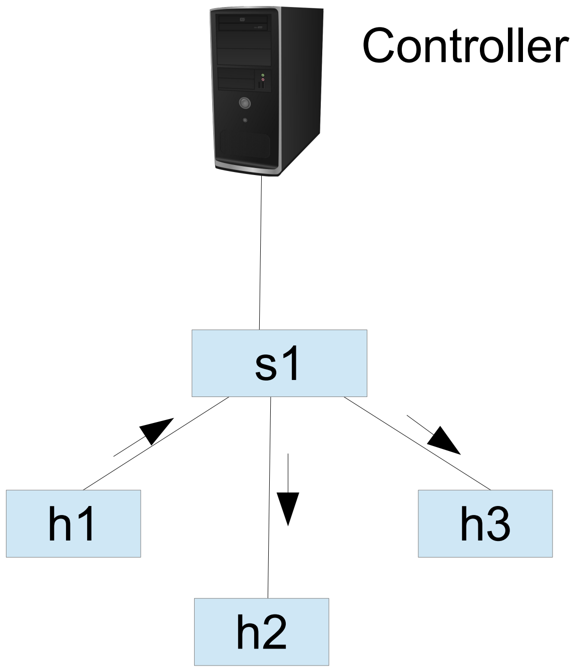
- Construct a new OpenFlow message for the switch
- Default: add new flow table entry
- All unspecified match fields are set to wildcards
- We only specify the actions here

An Efficient Repeater – Test

Forwards input packets out of all ports

- All hosts are pairwise reachable
- No packets are seen by the controller





An Efficient Repeater – Discussion

- Now that we have flow table entries, can we delete `_handle_PacketIn`?
- No!
- Packets may arrive before the flow table entry is installed
- OpenFlow messages do not guarantee any strict order
- Installing flow tables and switch initialization may take a while
- The switch may already send encapsulated packets the controller
- Test: delaying installation of flow table entries by 10 seconds

```
def hubify (event):
  print "installing flowtable entries on %s" % dpidToStr(event.dpid)
  msg = of.ofp_flow_mod()
  msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
  event.connection.send(msg)

def _handle_ConnectionUp (event):
  print "switch connected"
  core.callDelayed(10, hubify, event)
```

- The first packets are processed in software by the controller

An Efficient Repeater - Discussion

- The efficiency is still not very satisfying
- Incoming packets are flooded to all output ports
- This strategy resembles more to hubs than switches

- Next, we will build a learning switch
- Forwarding strategy:
  - When a packet with src MAC address *A* arrives at switch port *n*, we know that packets for device *A* should henceforth only be forwarded to port *n*.

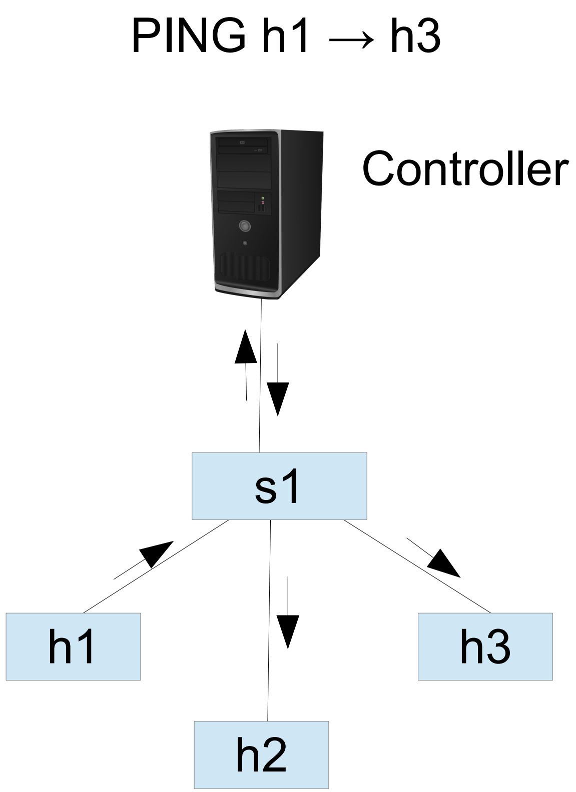
A Simple Learning Switch

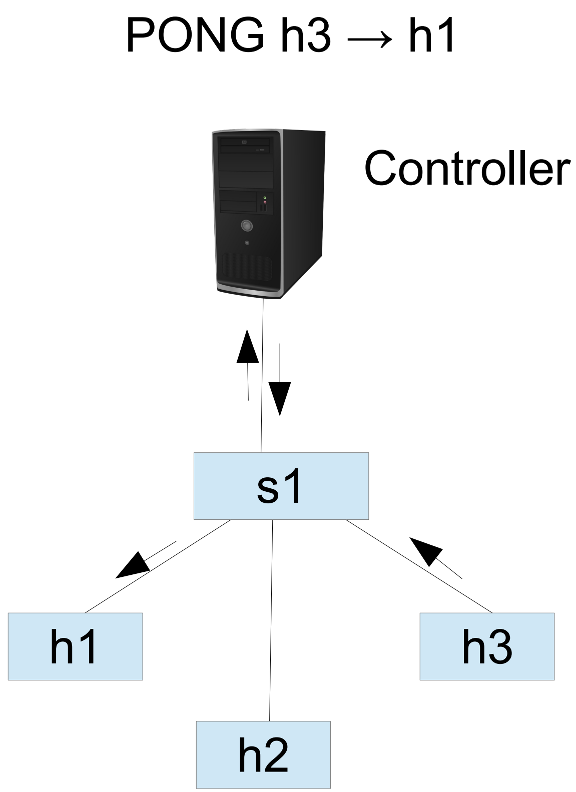
```
#map source_MAC_address -> switch_port
known_ports = dict()

def _handle_PacketIn (event):
  packet = event.parsed

  # Learn
  known_ports[packet.src] = event.port

  dst_port = known_ports.get(packet.dst)

  if dst_port is None:
    # destination port unknown, flood
    action = of.ofp_action_output(port = of.OFPP_FLOOD)
  else:
    print("Learned: %s <-> %s" % (packet.src, packet.dst))
    action = of.ofp_action_output(port = dst_port)

  msg = of.ofp_packet_out()
  msg.data = event.ofp
  msg.actions.append(action)
  event.connection.send(msg)
```


Based on pox/forwarding/l2_pairs.py

A Simple Learning Switch - Discussion

- If the destination is unknown, a packet is flooded
- If the destination is known, the packet is sent only to one port
- All packets traverse the controller





Towards an Efficient Learning Switch

- Need to prevent that every packet is send to the controller

  → Install flow table entries

- Can we install them in `_handle_ConnectionUp`?
  - No, we don't know where the hosts are

Towards an Efficient Learning Switch

- A first attempt
- If we receive a packet
  - The source address is at the input port
  - Install a flow table entry
    - Send everything for `src` to this port

```
print "installing %s -> %s" % (packet.src,  event.port)
msg = of.ofp_flow_mod()
msg.match.dl_dst = packet.src
msg.actions.append(of.ofp_action_output(port = event.port))
event.connection.send(msg)
```

- Where is the problem?

Towards an Efficient Learning Switch

- Example

  1) PING h1 → h3

     We install if `dst=h1 then output:h1.port`

     Flood packet

  2) PONG h3 → h1

     Directly forwarded by flow table entry

- The Problem:
  - We never see the PONG packet at the controller
  - We cannot learn h3's port
  - All further packets from h1 to h3 are send to the controller and flooded

An Efficient Learning Switch

- Solution
  - Only install flow table entries if source and destination port are known
  - This strategy requires about n^2 entries for n hosts

- A helper function
  - Install a rule which outputs packets from `src` MAC address to `dst` MAC address at `port`

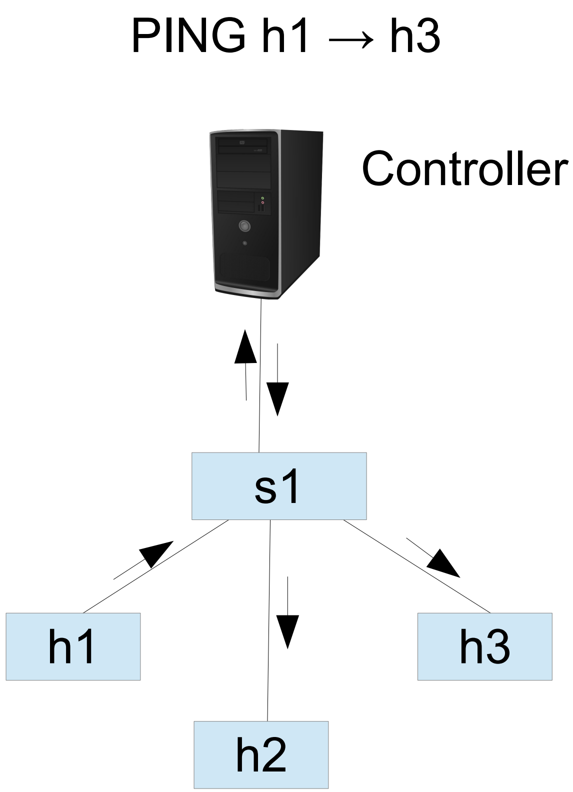
```
def install_l2_rule (conn, src, dst, port):
  msg = of.ofp_flow_mod()
  msg.match.dl_src = src
  msg.match.dl_dst = dst
  msg.actions.append(of.ofp_action_output(port = port))
  conn.send(msg)
```

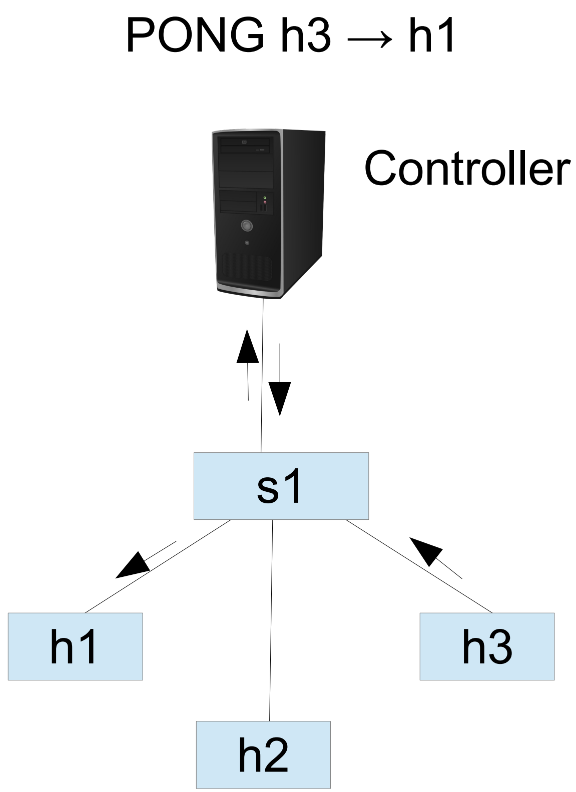
An Efficient Learning Switch

```
#map source_MAC_address -> switch_port
known_ports = dict()

def _handle_PacketIn (event):
  packet = event.parsed

  # Learn
  known_ports[packet.src] = event.port

  dst_port = known_ports.get(packet.dst)

  if dst_port is None:
    # destination port unknown, flood
    action = of.ofp_action_output(port = of.OFPP_FLOOD)
  else:
    print("Learned: %s <-> %s" % (packet.src, packet.dst))
    action = of.ofp_action_output(port = dst_port)
    install_l2_rule(event.connection, packet.src, packet.dst, dst_port)
    install_l2_rule(event.connection, packet.dst, packet.src, event.port)

  msg = of.ofp_packet_out()
  msg.data = event.ofp
  msg.actions.append(action)
  event.connection.send(msg)
```

An Efficient Learning Switch - Discussion

- If the destination is unknown, a packet is flooded
- If the destination is known, the packet is sent only to one port

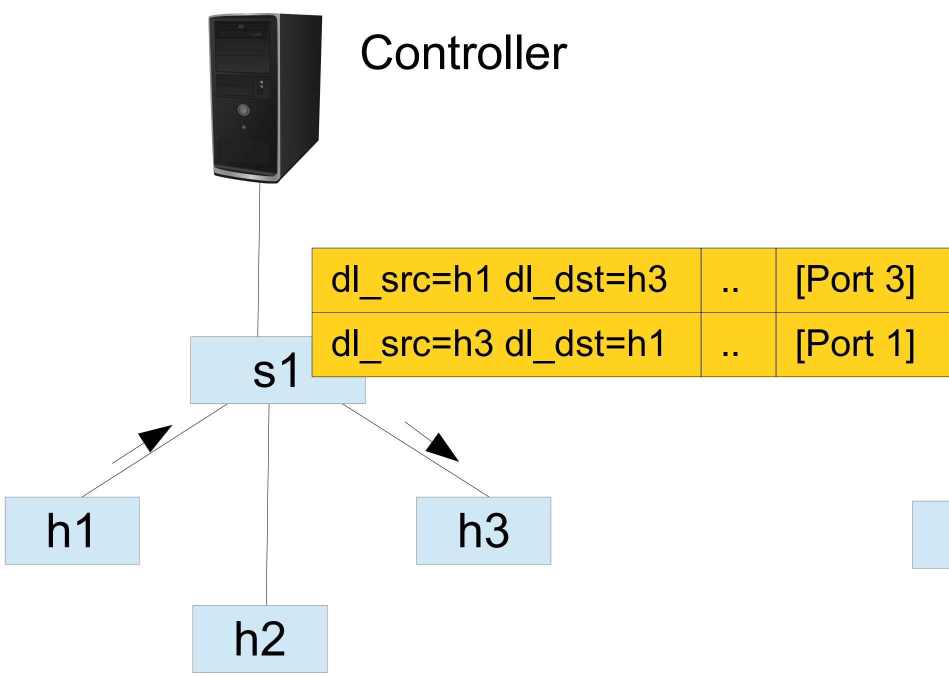




An Efficient Learning Switch - Discussion

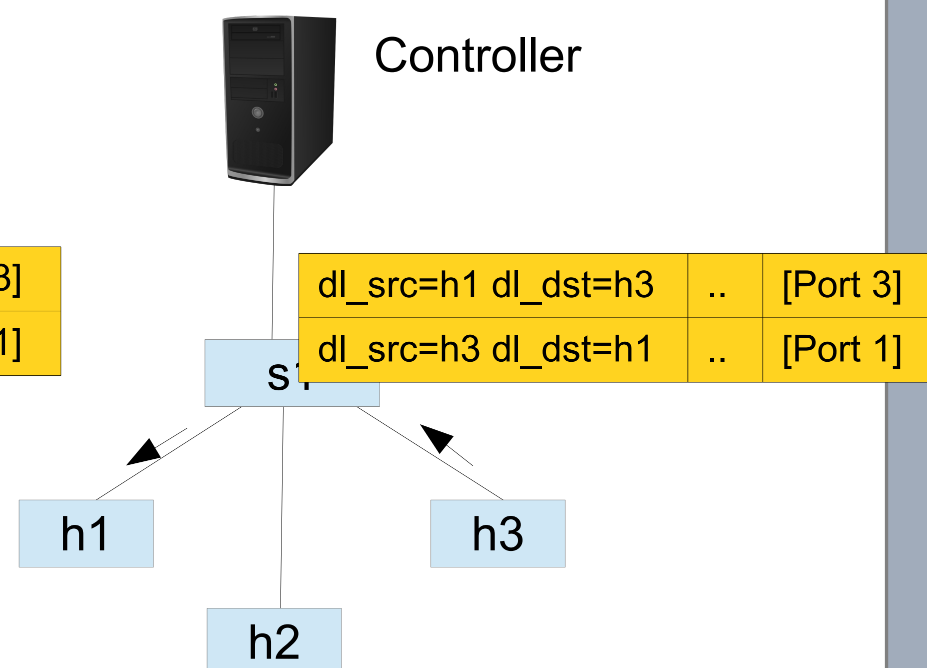
- ...
- Once both destinations are known, a rule is installed

PING h1 → h3



dl_src=h1 dl_dst=h3	..	[Port 3]
dl_src=h3 dl_dst=h1	..	[Port 1]

PONG h3 → h1



dl_src=h1 dl_dst=h3	..	[Port 3]
dl_src=h3 dl_dst=h1	..	[Port 1]

Problems Not Addressed

- Multiple switches
- Loops
- Layer 2 broadcast
- Learned output port = packet's input port?
- Host mobility

Programming Languages for Software Defined Networks

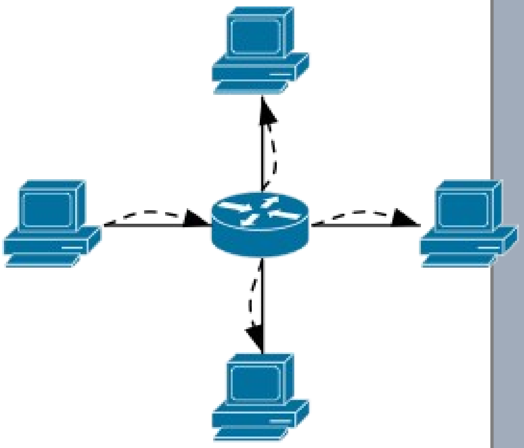
Introduction

Recall our hub example

- The controller code and the flow tables need to be written
- This is comparable to writing your C code and writing the same program in assembly again – by hand
- Programming languages for SDNs (such as frenetic) help out

```
let hub =
     if inPort = 1 then fwd(2,3,4)
  elsif inPort = 2 then fwd(1,3,4)
  elsif inPort = 3 then fwd(1,2,4)
  elsif inPort = 4 then fwd(1,2,3)
in hub
```

- This pseudo code can be compiled to a SDN controller **and** flow table entries



The examples are taken from https://github.com/frenetic-lang/frenetic/wiki/Frenetic-Tutorial

Composition

- Composition of flow table entries is also a non-trivial task
- Recall:
  - A packet matches at most one flow table entry
  - An entry can specify multiple actions
  - If multiple entries apply, a priority decides
  - If multiple entries with the same priority apply, we assume the first one (and only the first one) is applied

Composition: Example

- You want to count all packets with a srcIP *A,* using the packet counter

dl_type=0x800 nw_src=A	..	[count]

- Also, you want to statically forward all packets with dstIP *B* to Port 2

dl_type=0x800 nw_dst=B	..	[Port 2]

- Together: First statistics, then forwarding (sequential composition)

dl_type=0x800 nw_src=A	..	[count]
dl_type=0x800 nw_dst=B	..	[Port 2]

- Problem: all packets with srcIP A and dstIP B are counted and lost afterwards as the first matching rule **only** counts them.

- How do you apply both rules together (parallel composition)?

dl_type=0x800 nw_src=A nw_dst=B	..	[count, Port 2]
dl_type=0x800 nw_src=A	..	[count]
dl_type=0x800 nw_dst=B	..	[Port 2]

Composition

- Using composition operators, policies can be combined
  - Parallel composition `**|**'
  - Sequential Composition `**>>**'

[pyretic13]

- Assume we wrote a controller app that collects statistics and one that does firewalling
- We want to collect the statistics in parallel with applying the firewalling. Afterwards, our learning switch should forward all packets the firewall let through
- Code:

```
( statistics | firewall ) >> learning_switch
```

Composition: Examples

dstip rewriting

Monitor
srcip=5.6.7.8 → count

Route
dstip=10.0.0.1 → fwd(1)
dstip=10.0.0.2 → fwd(2)

Load-balance
srcip=0*,dstip=1.2.3.4 → dstip=10.0.0.1
srcip=1*,dstip=1.2.3.4 → dstip=10.0.0.2

Compiled Prioritized Rule Set for "Monitor | Route"
srcip=5.6.7.8,dstip=10.0.0.1 → count,fwd(1)
srcip=5.6.7.8,dstip=10.0.0.2 → count,fwd(2)
srcip=5.6.7.8 → count
dstip=10.0.0.1 → fwd(1)
dstip=10.0.0.2 → fwd(2)

Compiled Prioritized Rule Set for "Load-balance >> Route"
srcip=0*,dstip=1.2.3.4 → dstip=10.0.0.1,fwd(1)
srcip=1*,dstip=1.2.3.4 → dstip=10.0.0.2,fwd(2)

[pyretic13]

Warning: this example does not test for the Ethernet type

Conclusion

Conclusion

- Software Defined Networking is an idea that focuses on
  - Centralized management
  - Abstractions
  - Innovation at software speed
- Implementation
  - Controller, allows programming the network behavior
    - special programming languages (an active research area) provide easy-to-use means to program it
    - The controller runs on a
  - Network Operating System
    - provides easy-to-use API, keeps central state
    - and manages the hardware (forwarding plane) using
  - OpenFlow
    - an open standard which allows programming the forwarding plane devices

Comparison to Traditional Networks

- Traditional networks (simplified, idealized)
  - Autonomous boxes
  - Works “out of the box”
  - Decentralized
  - No single point of failure
  - General use case
- Software-defined networks (simplified, idealized)
  - Dumb boxes
  - Requires configuration and controller
  - Logically centralized
  - Single point of failure (control plane)
  - Specific scenarios

Bibliography

[Shenker11] Scott Shenker, *The Future of Networking, and the Past of Protocols.* Open Networking Summit 2011 http://www.youtube.com/watch?v=YHeyuD89n1Y

[Keown13] Nick McKeown, *Forwarding Plane Correctness*, Summer School on formal methods and networks, Cornell, 2013 http://www.cs.cornell.edu/conferences/formalnetworks/

[OFwp08] Nick McKeown et al., *OpenFlow: Enabling Innovation in Campus Networks,* ACM SIGCOMM Computer Communication Review, Apr 2008

[of10] Open Networking Foundation, *OpenFlow Switch Specification, Version 1.0.0 (Wire Protocol 0x01),* Dec 2009 https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf

[pyretic13] C. Monsanto et al., *Composing Software-Defined Networks*, 10th USENIX conference on Networked Systems Design and Implementation, 2013