Advanced Computer Networking (ACN)

IN2097 – WiSe 2023–2024

Prof. Dr.-Ing. Georg Carle

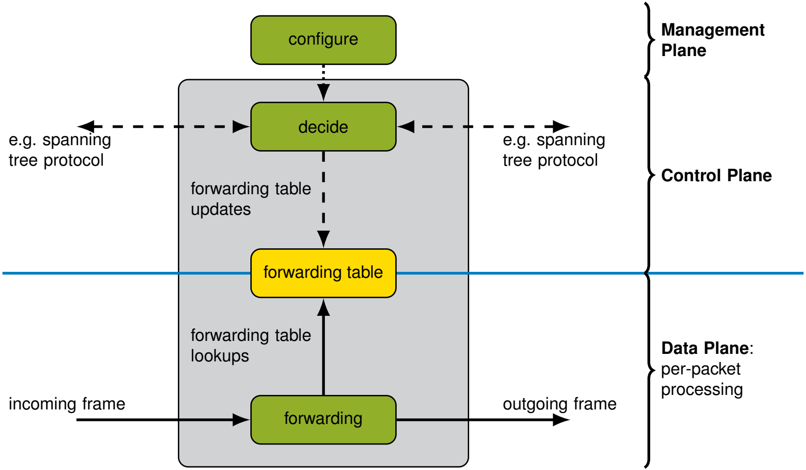
Sebastian Gallenmüller, Max Helm, Benedikt Jaeger,
Marcel Kempf, Patrick Sattler, Johannes Zirngibl

Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

Software Defined Networking

Software Defined Networking

Management Plane, Control Plane, and Data Plane



Introduction
Tasks of the Management Plane, Control Plane, and Data Plane

Management Plane:

- Allows access for administrators to the configuration of the other planes
- Tuning the parameters of the underlying algorithms

Control Plane:

- Has rules about which frames should go where
- Creates lookup tables from those rules
- Provides lookup tables for the data plane

Data Plane (also called Forwarding Plane):

- Uses lookup tables provided by the control plane
- Actually touches / forwards frames

Tasks of the Management Plane, Control Plane, and Data Plane

Management Plane:

- Allows access for administrators to the configuration of the other planes
- Tuning the parameters of the underlying algorithms

Control Plane:

- Has rules about which frames should go where
- Creates lookup tables from those rules
- Provides lookup tables for the data plane

Data Plane (also called Forwarding Plane):

- Uses lookup tables provided by the control plane
- Actually touches / forwards frames

Example: Tasks of the different planes in a router

- Management Plane: configuring link costs
- Control Plane: creating a routing table
- Data Plane: forwarding of frames according to routing table

Standard Telecommunication Architectures

Traditional architectures consist of three planes:

- management plane,
- control plane,
- and data plane.

What is a plane?

A plane is a group of algorithms and network protocols.

These protocols and algorithms

- process different kinds of traffic,
- have different performance requirements,
- are designed using different methodologies,
- are implemented using different programming languages,
- run on different hardware.

Standard Telecommunication Architectures

Problems with the standard approach

Implementations

- depend heavily on hardware platform and chip vendor,
- depend on the specific vendor implementation,
- offer limited access to the source code,
- are updated rarely or slowly (cf. adoption of IPv6),
- are often changing from one vendor to another.

SDN to the rescue

What is SDN?

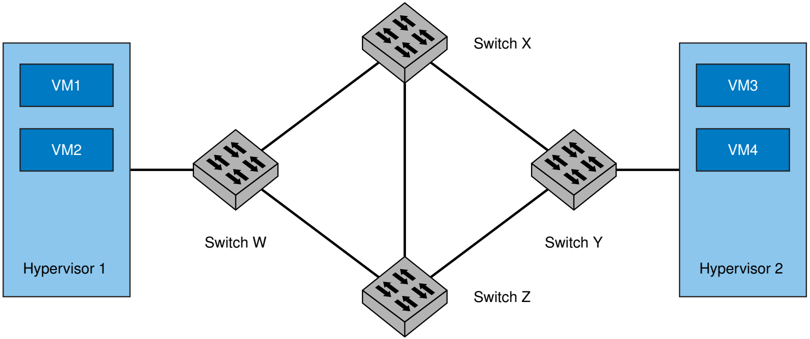
- **S**oftware **D**efined **N**etworking
- Provides a layer of abstraction from the physical network

How does it do that?

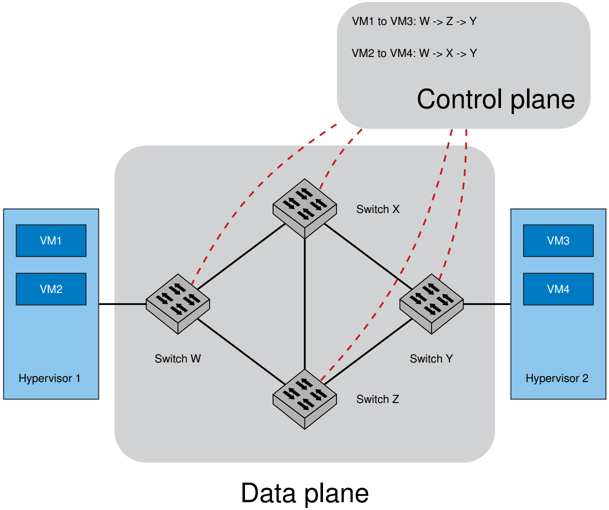
- Historically, devices include both, the control plane and the data plane
- SDN has one central control plane, which manages all the data planes of all the switches

Introduction
Illustration

- 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 ⟶ Z ⟶ Y
  - VM2 to VM4 should flow via W ⟶ X ⟶ Y



Introduction
Illustration



A more formal definition

Two requirements for SDN:

- A network in which the control plane is separate from the data plane
- A single control plane controls several forwarding devices

Both have to be met

SDN Benefits

Why the term “Software Defined”?

- The control plane is just software

Abstraction:

- No distributed state, one central view of the network
- Common model: "one big switch"-abstraction — the entire data plane behaves like a single giant switch
- No individual configuration of devices, one centrally managed control plane
- Important: View centralized, control plane itself may be implemented as a distributed system

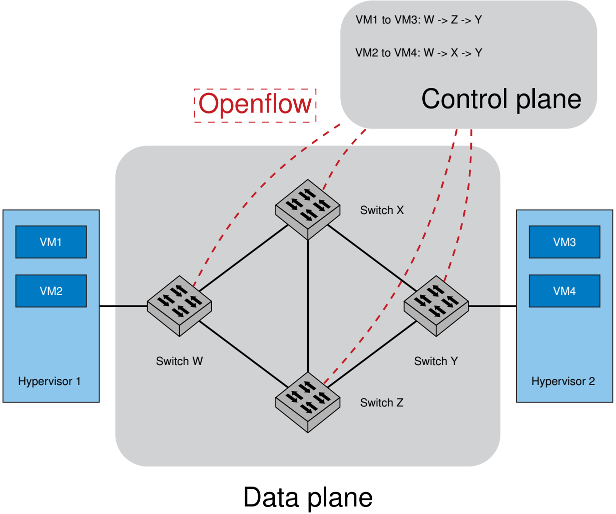
Gain:

- Complex, distributed protocols such as the Spanning Tree Protocol (STP) are no longer necessary
- Simpler algorithms utilizing the central view (e.g., Dijkstra’s algorithm instead of STP)
- Less complexity in the control plane

Software Defined Networking

What is OpenFlow?

- OpenFlow is a protocol configuring the forwarding plane
  - runs on top of TCP/SSL
  - Protocol spoken between control plane and forwarding plane
- Standardized by the Open Networking Foundation (ONF)
- Version 1.0 was released in 2009 [1]
- Latest version 1.6 from 2016 [2]



OpenFlow tables

OpenFlow is based on the match+action principle

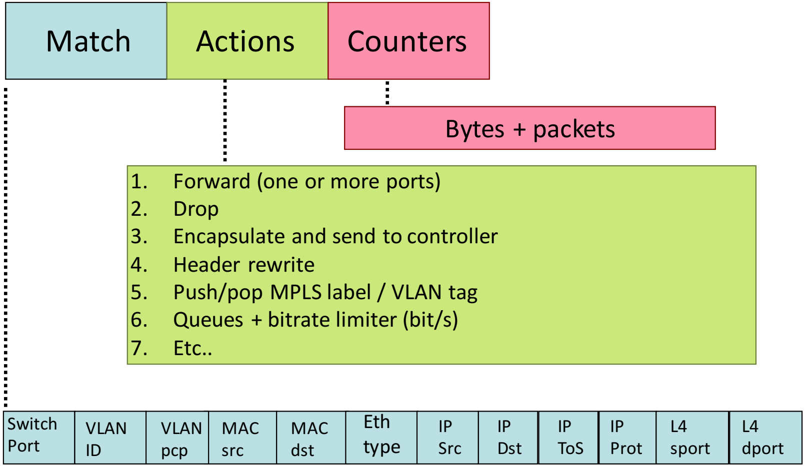


Figure 1: *(Source: cleanslate.stanford.edu)*

Tables examples

Switch Port	MAC Src	MAC Dst	Eth Type	IP Src	IP Dst	IP TOS	IP Prot	L4 Src	L4 Dst	Action
*	*	00:1f:...	*	*	*	*	*	*	*	Port 5

Table 1: Ethernet switch

Switch Port	MAC Src	MAC Dst	Eth Type	IP Src	IP Dst	IP TOS	IP Prot	L4 Src	L4 Dst	Action
*	*	*	*	*	1.2.0.0/16	*	*	*	*	Port 5

Table 2: Router

Switch Port	MAC Src	MAC Dst	Eth Type	IP Src	IP Dst	IP TOS	IP Prot	L4 Src	L4 Dst	Action
*	*	*	*	*	*	*	*	*	22	Drop

Table 3: Firewall

Remark about the term *switch*

Traditional classification

- Switch:
  - Works on Layer 2
  - Simple forwarding of packets
- Router:
  - Works on Layer 3
  - Finding out where to route packets (LPM)

In the context of SDN every "box" is considered a switch

- Clear distinction (e.g. switch, router) no longer possible as functionality is determined by software
- These boxes/switches can even be used as firewall, tunnel gateways

OpenFlow switch

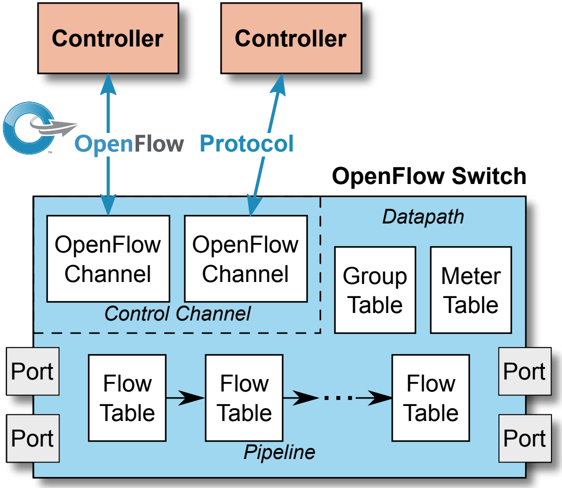


Figure 2: OpenFlow switch *(source: OpenFlow Switch Specification, ONF)*

Open vSwitch

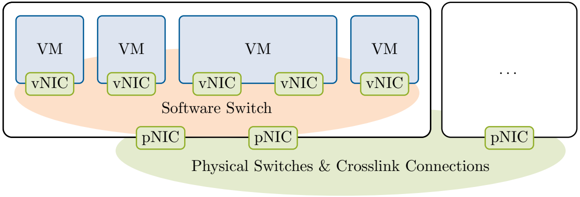
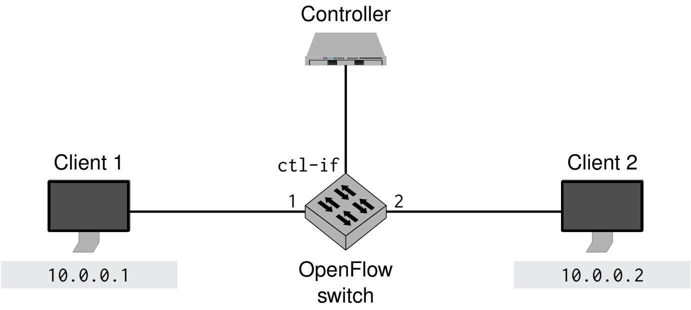


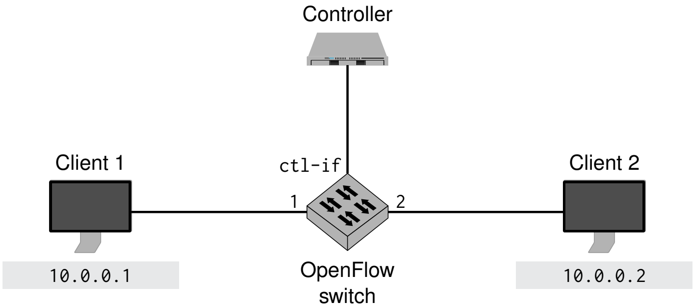
Figure 3: Virtual software switches [3]

- Open vSwitch (OvS) is a (virtual) software switch
- Supports OpenFlow (considered as the de-facto standard implementation of OpenFlow)
- OvS is typically used to connect different VMs on the same host or between different hosts
- OvS can also be used to turn a server with into an OpenFlow switch



Rules installed on the switch

```
ovs-ofctl add-flow ctl-if priority=0,actions=controller
```

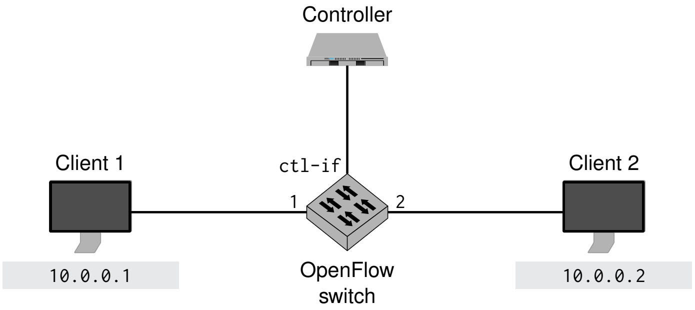


Rules installed on the switch

```
ovs-ofctl add-flow ctl-if priority=0,actions=controller
```

- `add-flow`: "OpenFlow rule" (**Not** a regular network flow!)
- `ctl-if`: Destination for this OpenFlow flow
- `actions=controller`: Send packets matching this rule to the controller
- `priority=0`: 0 is lowest priority

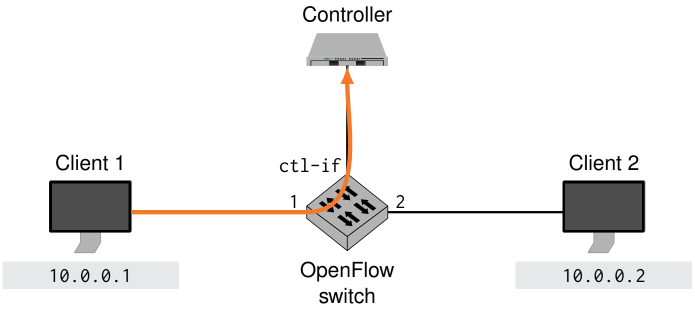
Example



Rules installed on the switch

```
ovs-ofctl add-flow ctl-if priority=0,actions=controller
```

- Packet sent from Client 1 to Client 2

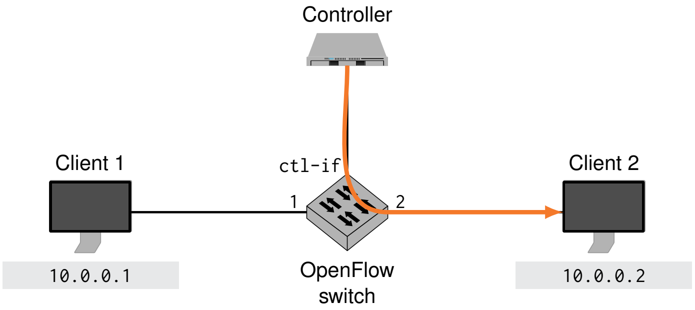


Rules installed on the switch

```
ovs-ofctl add-flow ctl-if priority=0,actions=controller
```

- Packet sent from Client 1 to Client 2
- Packet matches against rule ⟶ Controller

Example

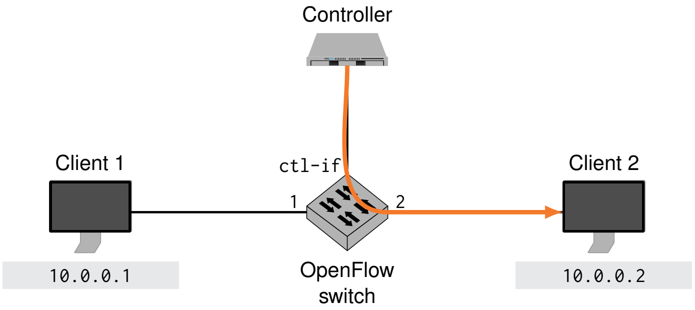


Rules installed on the switch

```
ovs-ofctl add-flow ctl-if priority=0,actions=controller
```

- Packet sent from Client 1 to Client 2
- Packet matches against rule ⟶ Controller
- Controller instructs switch to send packet to destination

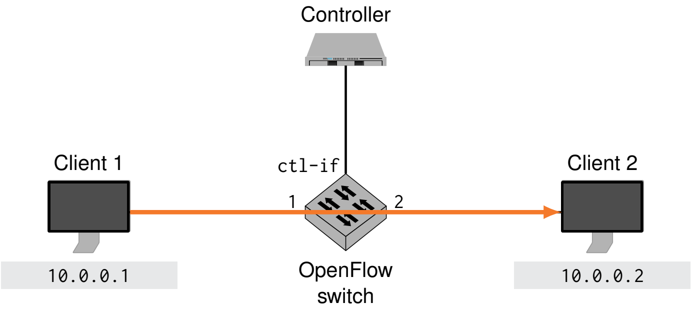
Example



Rules installed on the switch

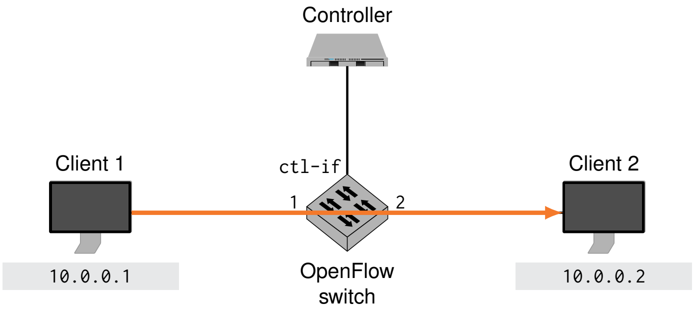
```
ovs-ofctl add-flow ctl-if priority=0,actions=controller
```

- Packet sent from Client 1 to Client 2
- Packet matches against rule ⟶ Controller
- Controller instructs switch to send packet to destination
- **Problem:** sending each packet to the controller, may create a bottleneck / overload the controller



Rules installed on the switch

```
ovs-ofctl add-flow ctl-if priority=0,actions=controller

ovs-ofctl add-flow ctl-if dl_type=0x0800,nw_dst=10.0.0.2, priority=10000,actions=output:2
```



Rules installed on the switch

```
ovs-ofctl add-flow ctl-if priority=0,actions=controller

ovs-ofctl add-flow ctl-if dl_type=0x0800,nw_dst=10.0.0.2, priority=10000,actions=output:2
```

- Controller can also install rule on switch to make forwarding more efficient
- IPv4 packets (matching ethertype `0x0800` destination address `10.0.0.2`) from Client 1 get directly forwarded to Client 2

OpenFlow in the wild

- OpenFlow is not SDN
- OpenFlow with its standardized interface enables SDN deployment
- Very successful in software switches (Open vSwitch)
- There are hardware switches with OpenFlow support
  - Did not make traditional switches obsolete as initially expected
  - Still many proprietary switches today

OpenFlow

- Allows programming the control plane
- Allows modifications in the data plane
- Standard supports only a limited number of protocols
  - To introduce new protocols the standard must be updated
  - Switches must be upgraded to handle the new standard

Software Defined Networking

Network Function Virtualization (NFV)

- Defined by ETSI (European Telecommunications Standards Institute)
- Telco-driven approach for networks initiated in 2012
- Definition of NFV according to ETSI: NFV is a concept *"leveraging standard IT virtualisation technology to consolidate many network equipment types onto industry standard high volume servers, switches and storage, which could be located in Datacentres, Network Nodes and in the end user premises."*

NFV
NFV—simply explained

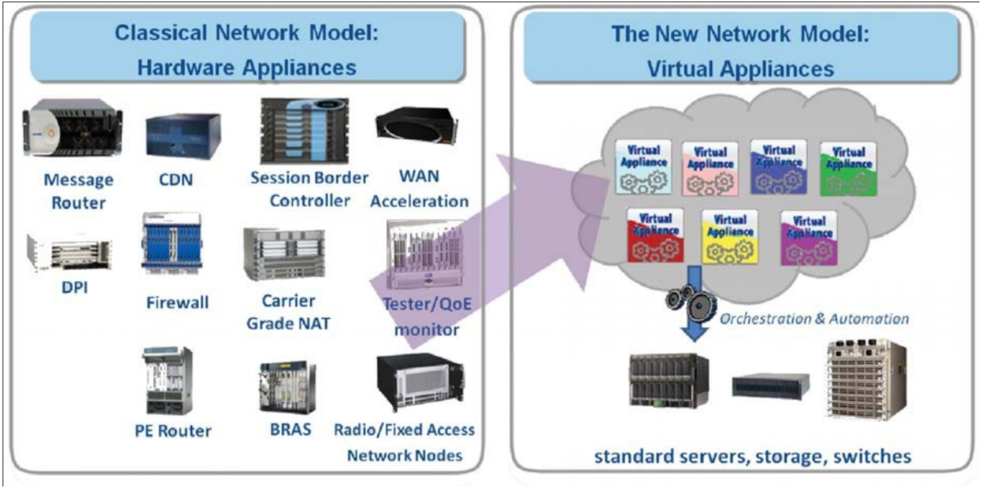


Figure 4: from `https://www.slideshare.net/nearyd/nfv-for-beginners`

NFV
Some Terminology

- **(V)NF: (Virtualized) Network Function**, (virtualized) building block performing a network task
- **NFC: Network Function Chaining**, putting together several network functions to create more complex packet processing chains

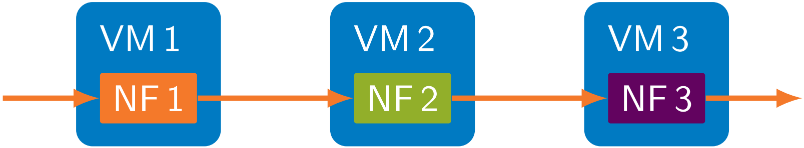


Figure 5: Example of a chain of Virtual Network Functions

NFV
NFV vs. SDN

- "SDN and NFV are complementary but increasingly co-dependent" [4]
- SDN: dynamically control the network
- NFV: manage and orchestrate the virtualization of resources for the provisioning of network functions and their composition into higher-layer network services

NFV architectures I

Traditional approach

- One VM per NF
- Communication between NFs via virtual switch
- \+ Strong isolation between NFs
- \+ Uses traditional OS sockets
- \- High load on virtual switch

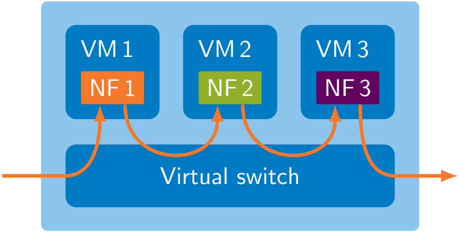


Figure 6: Traditional VM-based NFC

Non-virtualized NFC

- Entire NFC running directly on host system
- Communication between NFs via NF framework (e.g. DPDK), initial entry and last exit via virtual switch
- \+ No costs for virtual switch
- \- NFs need to be rewritten to use NF framework

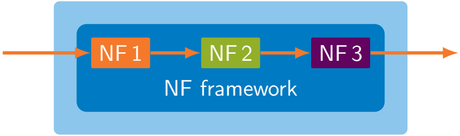


Figure 7: Non-virtualized framework-based NFs

Hybrid solution: virtualized NFC

- One VM for entire NFC
- Communication between NFs via NF framework, initial entry and last exit via virtual switch

\+ Lower load on virtual switch

\+ Isolation between host OS and the NF chain inside the VM

\- NFs need to be rewritten to use NF framework

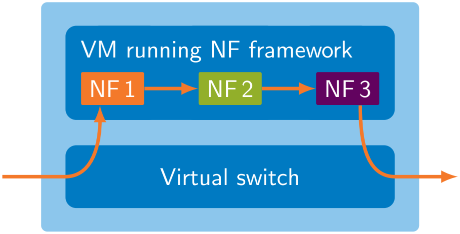


Figure 8: Virtualized framework-based NFs

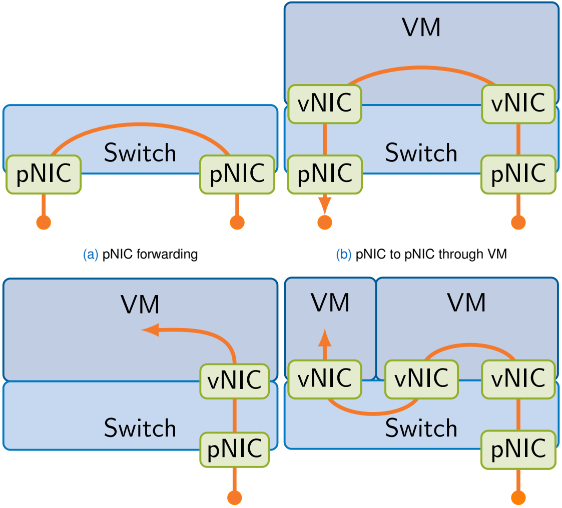
Performance of NFs

- Tradeoff between isolation and performance requirements:
  - Isolation (high to low): Virtual machines, container, no virtualization
  - Performance (low to high): Virtual machines, container, no virtualization

NFV
Performance measurements

Performance of virtual switching solutions [3]

- Investigated 4 different setups involving physical/virtual pNICs/vNICs
- CPU: Intel Xeon E3-1230 V2 CPU (3.3 GHz, base clock)
- pNIC: 10 Gbit/s Intel X540
- SW: GRML Linux kernel v3.7, Open vSwich v2.0, DPDK vSwitch v0.1
- Hypervisor: qemu-kvm 1.1.2
- Worst case measurement scenario: minimum-sized packets 64 B (14.88 Million packets per second (Mpps) @ 10 Gbit/s)



(a) pNIC forwarding (b) pNIC to pNIC through VM

(c) pNIC to vNIC2 (d) vNIC to vNIC

Figure 9: Investigated test setups

Table 4: Single Core Data Plane Performance Comparison

	Million packets per second (Mpps) from pNIC to			
Application	pNIC	vNIC	vNIC to pNIC	vNIC to vNIC
Linux bridge	1.11	0.74	0.20	0.19
IP forwarding	1.58	0.78	0.19	0.16
Open vSwitch (OvS)	1.88	0.85	0.30	0.27
DPDK vSwitch	13.51	2.45	1.10	1.00

- DPDK vSwitch is the DPDK-accelerated version of OvS
- Network IO for VMs is quite expensive

Conclusion

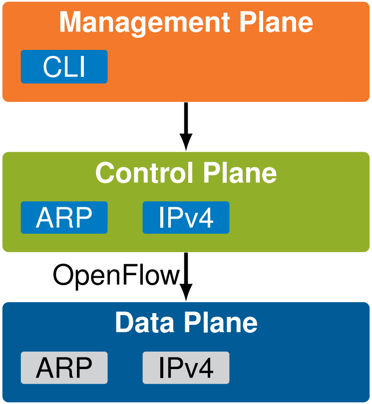
	Traditional approach	Virtualized NFC	Non-virtualized NFC
Performance	+	++	+++
Isolation	+++	++	+
Chaining interface	OS sockets	Framework-based	Framework-based

Table 5: Comparison between different NFC architectures

Possible reasons for choosing different architectures

- Performance requirements
- Integration of legacy NF supporting only socket interface
- Integration of NFs from different vendors
- Stronger isolation requirements for untrusted customer code

OpenFlow versus P4



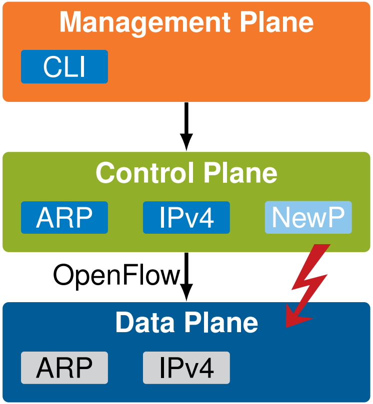
OpenFlow

- OpenFlow allows programmability on the control plane
- OpenFlow offers a standardized interface to configure the data plane
- OpenFlow only supports protocols known by the hardware or software used on the data plane

OpenFlow versus P4

OpenFlow

- OpenFlow allows programmability on the control plane
- OpenFlow offers a standardized interface to configure the data plane
- OpenFlow only supports protocols known by the hardware or software used on the data plane
- Introducing a new protocol (e.g., NewP) fails without support from the data plane



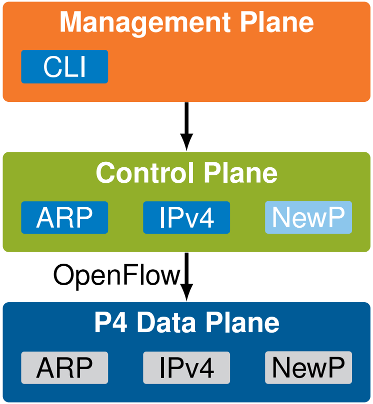
OpenFlow versus P4

OpenFlow

- OpenFlow allows programmability on the control plane
- OpenFlow offers a standardized interface to configure the data plane
- OpenFlow only supports protocols known by the hardware or software used on the data plane

P4 (**P**rogramming **P**rotocol-Independent **P**acket **P**rocessors)

- P4 is a domain specific programming language to program data plane devices
- P4 allows programming switches to support entirely new protocols (e.g., NewP)



OpenFlow versus P4

OpenFlow

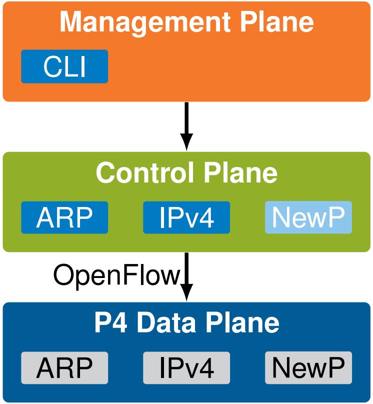
- OpenFlow allows programmability on the control plane
- OpenFlow offers a standardized interface to configure the data plane
- OpenFlow only supports protocols known by the hardware or software used on the data plane

P4 (**P**rogramming **P**rotocol-Independent **P**acket **P**rocessors)

- P4 is a domain specific programming language to program data plane devices
- P4 allows programming switches to support entirely new protocols (e.g., NewP)

OpenFlow vs. P4

- P4 is **not** a successor or a replacement of OpenFlow
- OpenFlow and P4 solve specific tasks on separate planes
- P4 can be used to implement OpenFlow-capable applications for switches



Data plane programmability

Goal: program your own data plane!

Benefits:

- Control and customization: make the device behave exactly as you want, operators can hide internal protocols
- Reliability: include only the features you need
- Efficiency: reduce energy consumption and expand scale by doing only what you need
- Update: Add new features when you want
- Telemetry: See inside the data plane
- Exclusivity: Program your own features without the need for involving a chip vendor
- Rapid Prototyping: enables fast deployment of protocols for prototyping
- Fast Development Cycles: enables software upgrades for protocols

Data plane programmability

Goal: program your own data plane!

Benefits:

- Control and customization: make the device behave exactly as you want, operators can hide internal protocols
- Reliability: include only the features you need
- Efficiency: reduce energy consumption and expand scale by doing only what you need
- Update: Add new features when you want
- Telemetry: See inside the data plane
- Exclusivity: Program your own features without the need for involving a chip vendor
- Rapid Prototyping: enables fast deployment of protocols for prototyping
- Fast Development Cycles: enables software upgrades for protocols

Challenges:

- Performance: data planes need to process millions of packets per second
- Flexibility: Enable the implementation of various protocols
- Hardware independence: keep the description high-level enough

Meet P4

An open source language allowing the specification of packet processing logic

Based on a Match+Action forwarding model

Multiple platforms supported:

- Software-based solution (e.g., using DPDK)
- NPUs - Network Processor Units
- FPGAs - Field Programmable Gate Arrays
- P4-specific ASICs

Software targets

p4c/bmv2

- open source, available at https://p4.org/code/
- "official" P4 reference implementation developed by p4.org
- used for teaching, testing, trying out new features
- no specific hardware required (mininet)
- slow, not optimized for performance

T_4P_4S *(called tapas)*

- open source, available at http://p4.elte.hu/
- compiles P4 for DPDK
- requires DPDK-compatible hardware
- decent performance (>10 Gbit/s)

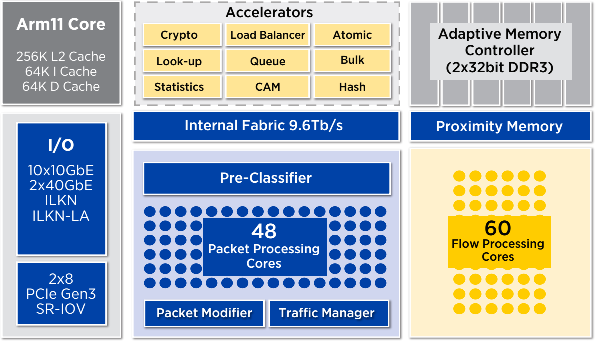
P4TC

- open source, available at https://www.p4tc.dev/
- ongoing effort to bring P4 to the Linux kernel
- based on existing Linux modules (traffic control/TC)
- bringing P4 to end hosts

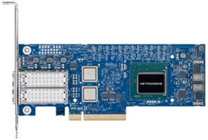
P4 targets
Network Processor Unit (NPU)

Netronome Agilio SmartNIC

- purpose-built processor for packet processing
- specialized hardware accelerators (e.g. hashing, look up)
- highly parallelized architecture (>100 cores)
- supports several programming languages C, P4, eBPF
- up to 2 × 100 Gbit/s interfaces per network card



NFP-4000 architecture [source: netronome.com]

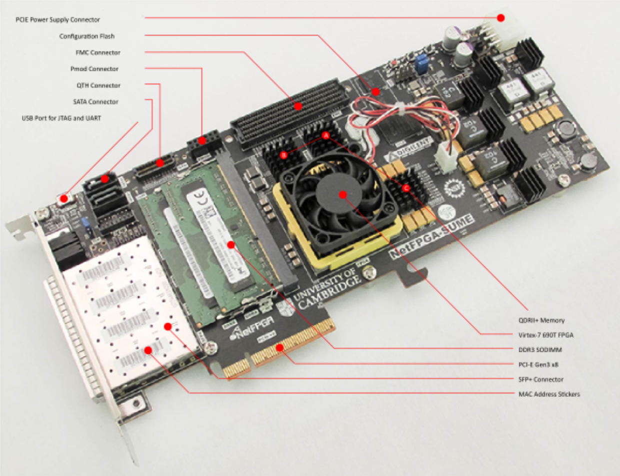


Netronome SmartNIC [source: colfaxdirect.com]

Field Programmable Gate Array (FPGA)

NetFPGA

- fully programmable NIC (down to the physical layer)
- utilizing hardware description languages such as Verilog or VHDL
- Xilinx Virtex 7 FPGA
- up to 4×10 Gbit/s interfaces (via SFP+ transceivers)



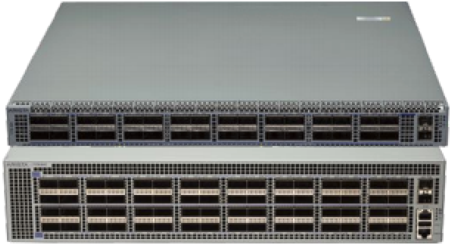
NetFPGA Sume [source: github.com/NetFPGA]

P4-specific ASICs

Barefoot Tofino 2

- Tofino ASIC: specifically designed switching ASIC with native P4 support
- capable of up to 12 Tbit/s throughput (unidirectional)
- *for comparison:* peak traffic at biggest Internet exchange DE-CIX in Frankfurt was 15 Tbit/s in 2023[a]
- up to 64 $\times$ 200 Gbit/s interfaces (via QSFP56 transceivers)

[a] https://www.de-cix.net/en/about-de-cix/media/press-releases/europes-largest-internet-exchange-de-cix-frankfurt-sets-new-traffic-record-15-terabits-per-second, last accessed 2023-01-03



32 / 64-port switch [source: arista.com]

Target comparison

	SW	NPU	FPGA	ASIC
Performance	+	++	++	+++
Flexibility	+++	++	++	+
Ease of use	+++	+	+	+
Costs	0 €	> 500 €	> 1000 €	> 10 000 €

Target comparison

	SW	NPU	FPGA	ASIC
Performance	+	++	++	+++
Flexibility	+++	++	++	+
Ease of use	+++	+	+	+
Costs	0 €	> 500 €	> 1000 €	> 10 000 €

Did P4 achieve its goals?

- Performance: data planes need to process millions of packets per second ➔ accomplished ✓
- Flexibility: Enable the implementation of various protocols ➔ accomplished ✓
- Hardware independence: keep the description high-level enough ➔ development ongoing . . .
  - Basic P4 functionality can be realized on any target
  - Every target offers different additional capabilities not programmed in P4 (e.g. multicast support)
  - These additional functionalities make P4 programs hardware dependent

Organization

P4 open source efforts are centralized on:

- Official website: `https://p4.org`
- Github: `https://github.com/p4lang`

P4 consortium members

P4 versions

Two versions available:

- $P4_{14}$, released in March, 2015
  - unified language for all targets
  - development driven by hardware developers
- $P4_{16}$, released in May, 2017
  - concentrating P4 language on core functionalities
  - development driven by software developers (P4 becoming a more C-like programming language)

Figure 10: P4 logo

Note: the following slides are based on the P4 tutorial from P4.org

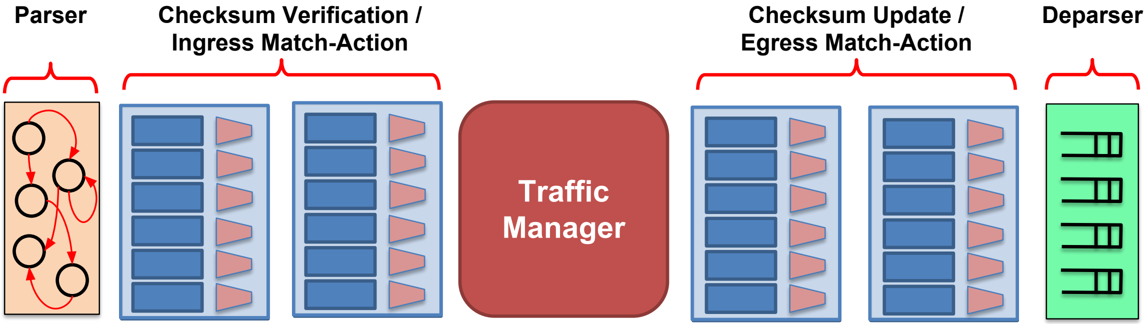


Figure 11: P4 model architecture

P4 Core
Different switch models

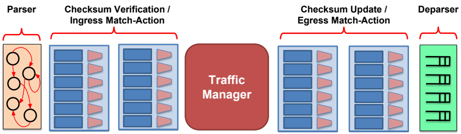


Figure 12: P4 model architecture

Figure 13: P4 model architecture without traffic manager

Figure 14: P4 model architecture without traffic manager and egress stages

- P4 models present the capabilities of a P4-enabled device
- Models typically reflect the different features of different P4 targets

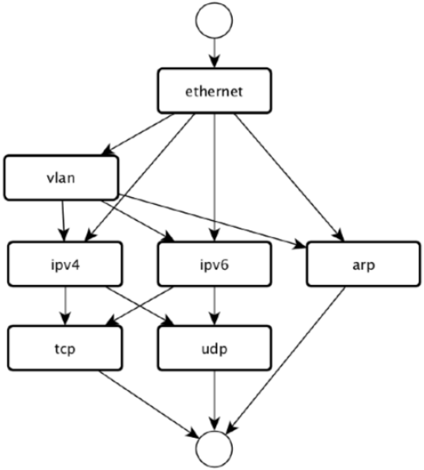
Parser

Parser tasks

- Finite State Machine (FSM)
- Produces a parsed representation of valid headers
- Describes all supported headers
- Describes the order in which headers may appear

Deparser tasks

- Executed before sending a frame
- Assemble the different fields and their order in a frame



Abstract representation of a packet parser [source: open-nfp.org]

Metadata

Tasks

- Data structures associated with every packet
- Standard metadata:
  - Default metadata provided by all P4 targets for every packet
  - e.g. ingress_port
- Intrinsic metadata:
  - Additional target-specific metadata provided for every packet
  - e.g. receive_timestamp
- User-defined metadata
  - Data created by the P4 program during runtime for every packet
  - e.g. new_tunnel_id

Match tables

	name	field	match_kind	match_value	action	action data
[0]	encap	`ingress_port`	`exact`	`port_0`	`encapsulate_act`	`vlantag = 123`
[1]	default				`drop`	

Example table

Tasks

- Each table contains one or more entries
- An entry contains a specific key to match on (field) and a single action (action) to be executed, and additional data (action data)
- The match operation supports different types (match_kind):
  - `exact`: select the entry exactly matching match_value
  - `lpm`: select the entry with the longest prefix matching
  - `ternary`: select with some ignored bits e.g. `match_value` of `10*1` → `1011` or `1001`
- P4 targets may define additional match types, e.g. `range`
- If no entry matches, the mandatory default entry is selected

Actions & extern objects

Tasks

- Similar to C functions without any loops or pointers
- Modification of field values and headers (add or remove)
- Besides the packet/header data, the action also may get additional data from tables
- Primitives for metering, registers, counters, hashes and random numbers

Extern objects

- New in $P4_{16}$
- Externs perform additional tasks which are either not written in or not supported by P4
- Architecture specific:
  - Software/NPU targets: extension via programmed functions (C, Python, . . .)
  - FPGA: extension via VHDL/Verilog-defined functions
  - ASIC: no extension possible

Match-Action Processing

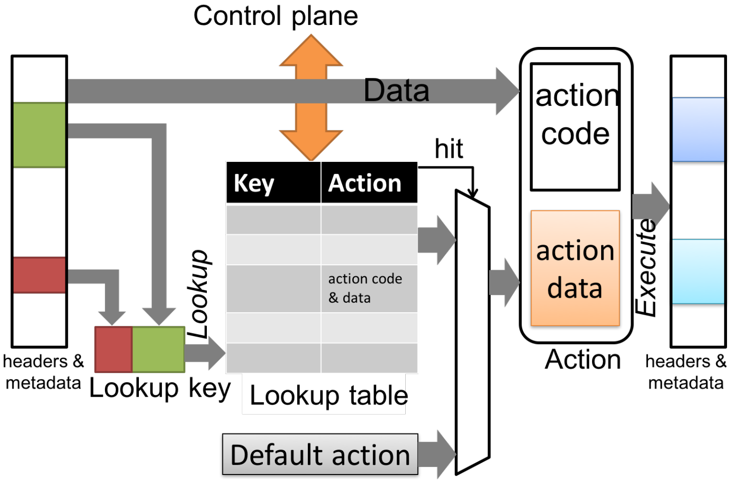


Illustration of P4 match-action process [source: p4.org]

P4 Portable Switch Architecture (PSA)

Goal:

- Reference architecture for P4 switches
- Separate PSA specification available on p4.org
- Architecture describes common capabilities of network switch devices

Common capabilities

- Metadata definitions
- Hashes and checksums (only simple hashes e.g. CRC, no cryptographic hashes such as SHA)
- Counters and meters
- Registers
- Random number generators
- Access to timestamps

Example for non-common capabilities

- Capabilities of the traffic manager, such as packet generation

P4 example: IPv4 router

Disclaimer

- Basic P4 example
- Essential features are missing, no ARP/ICMP/VLAN/IPv6 handling

→ do not use this router for the project ;)

P4 example: IPv4 router
Headers and fields definition

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
```

bit<n> defines unsigned int of length n
typedef introduces a shorter label for field declarations

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   ethpersType;
}
```

header declares a new header. The following operations can be called on a header: **isValid()**, **setValid()**, and **setInvalid()**.

P4 example: IPv4 router
Headers and fields definition

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   ethpersType;
}
```

bit<n> defines unsigned int of length n
typedef introduces a shorter label for field declarations

header declares a new header. The following operations can be called on a header: **isValid()**, **setValid()**, and **setInvalid()**.

What about the frame check sequence?

Headers and fields definition

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
```

bit<n> defines unsigned int of length n
typedef introduces a shorter label for field declarations

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   ethpersType;
}
```

header declares a new header. The following operations can be called on a header: **isValid()**, **setValid()**, and **setInvalid()**.

What about the frame check sequence?
→ Checked and added automatically

P4 example: IPv4 router
Headers and fields definition

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;


header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   ethpersType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<16>   flagsfragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

bit<n> defines unsigned int of length n
typedef introduces a shorter label for field declarations

header declares a new header. The following operations can be called on a header: **isValid()**, **setValid()**, and **setInvalid()**.

What about the frame check sequence?
→ Checked and added automatically

Offset	0–3	4–7	8–15	16–18	19–31
0 B	Version	IHL	TOS	Total Length	
4 B	Identification			Flags	Fragment Offset
8 B	TTL		Protocol	Header Checksum	
12 B	Source Address				
16 B	Destination Address				
20 B	Options / Padding (optional)				

IPv4 header

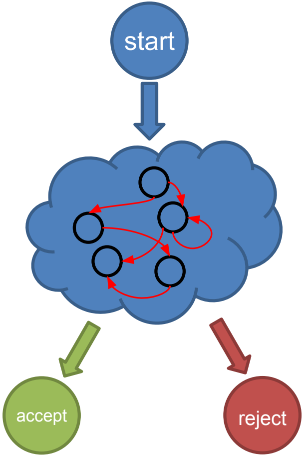
P4 example: IPv4 router
Metadata definition

```
/* Architecture */
struct standard_metadata_t {
    bit<9> ingress_port;
    bit<9> egress_spec;
    bit<9> egress_port;
    bit<32> clone_spec;
    bit<32> instance_type;
    bit<1> drop;
    bit<16> recirculate_port;
    bit<32> packet_length;
    ...
}

/* User program */
struct metadata {
    ...
}

struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
}
```

struct defines a unsorted collection of members

$P4_{16}$ Parsers

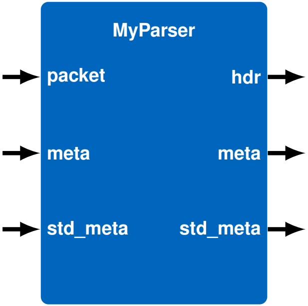
- Parsers map packets to headers and metadata
- Parsers are written as state machines
- Each parser has three predefined stats:
  - `start`
  - `accept`
  - `reject`
- Additional states may be defined by the programmer
- Each state may execute statements and then transition to another state
- Loops are allowed



P4 example: IPv4 router

Parser definition

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t std_meta) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.ethType) {
            TYPE_IPV4: parse_ipv4; // 0x800
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```



select works similar to case statements in Java/C

select ends after successful match (`default` is not executed after successful `TYPE_IPV4` match)

extract set header and its fields to valid

P4 example: IPv4 router

Ingress and table definition

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta) {
    action drop() { mark_to_drop(); }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = { hdr.ipv4.dstAddr: lpm; }
        actions = { ipv4_forward;  drop; NoAction; }
        size = 1024;
        default_action = NoAction();
    }

    apply {
        if (hdr.ipv4.isValid()) { ipv4_lpm.apply(); }
    }
}
```

A **control** block contains the functionality of the program

Control blocks can represent different kinds of processing:

- Match-Action pipelines
- Deparsers
- Additional forms of processing (checksums)

Typically headers and metadata act as interfaces between control blocks

Execution starts with **apply()** statement

IPv4 Table example

	field	match_kind	key	action	action data
[0]	hdr.ipv4.dstAddr	lpm	10.0.1.1/32	ipv4_forward	dstAddr=00:00:00:00:01:01, port=1
[1]	hdr.ipv4.dstAddr	lpm	10.0.1.2/32	drop	
[2]	-	-	-	NoAction	

P4 example: IPv4 router

Checksum verification

```
control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply {
        verify_checksum(
            hdr.ipv4.isValid(), //check validity of header
            { //list of inputs
                hdr.ipv4.version,
                hdr.ipv4.ihl,
                hdr.ipv4.diffserv,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags,
                hdr.ipv4.fragOffset,
                hdr.ipv4.ttl,
                hdr.ipv4.protocol,
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr
            },
            hdr.ipv4.hdrChecksum, //output
            HashAlgorithm.csum16 //hash calculation
        );
    }
}
```

P4 example: IPv4 router

Checksum calculation

```
control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
    apply {
        update_checksum(
            hdr.ipv4.isValid(), //check validity of header
            { //list of inputs
                hdr.ipv4.version,
                hdr.ipv4.ihl,
                hdr.ipv4.diffserv,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags,
                hdr.ipv4.fragOffset,
                hdr.ipv4.ttl,
                hdr.ipv4.protocol,
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr
            },
            hdr.ipv4.hdrChecksum, //output
            HashAlgorithm.csum16 //hash calculation
        );
    }
}
```

P4 example: IPv4 router

Egress, deparser and switch definition

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t std_meta) {
    apply { }
}

// no explicit deparser object => control
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}

Router(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```

Active area of research

P4, like OpenFlow, has attracted a lot of researchers

- Extension of the P4 language itself
- Proposition of new platforms supporting P4
- New protocols and services on top of P4
- Other open programming languages for common network functionalities (e.g., packet scheduling)
- . . .

Theses offered at the chair

- P4 benchmarking
- P4 extensions
- . . .

Software Defined Networking

Acknowledgements

- Slides partially based on work by Cornelius Diekmann

[1] O. N. Foundation, “Openflow switch specification, version 1.0.0,”, 2009.

[2] O. N. Foundation, “Openflow switch specification, version 1.6,”, 2016.

[3] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, “Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis,” in Journal of Network and Systems Management, Jul. 2017. DOI: 10.1007/s10922-017-9417-0.

[4] ESI, Network function virtualisation, last accessed: 2019-11-24, 2012. [Online]. Available: https://www.etsi.org/technologies/nfv.