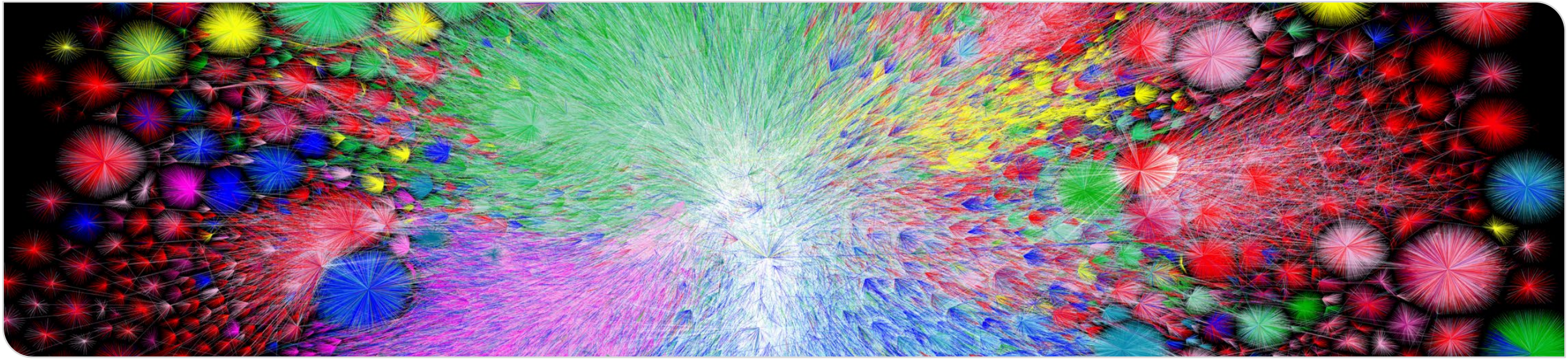
Karlsruher Institut für Technologie

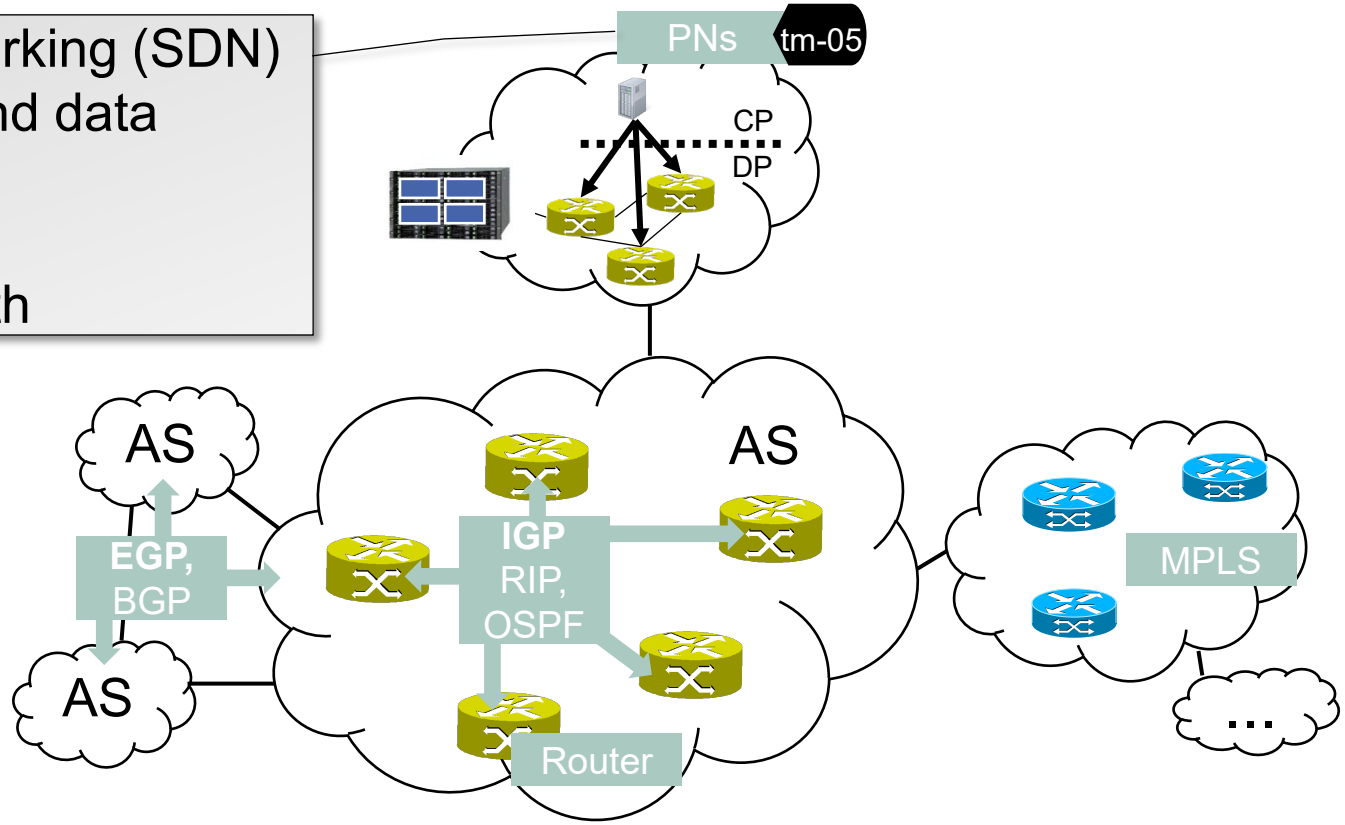
5. Programmable Networks

Prof. Dr. Martina Zitterbart
Institute of Telematics



Karlsruher Institut für Technologie

Software defined networking (SDN)
Separation of control and data plane
Network virtualization
Programmable data path

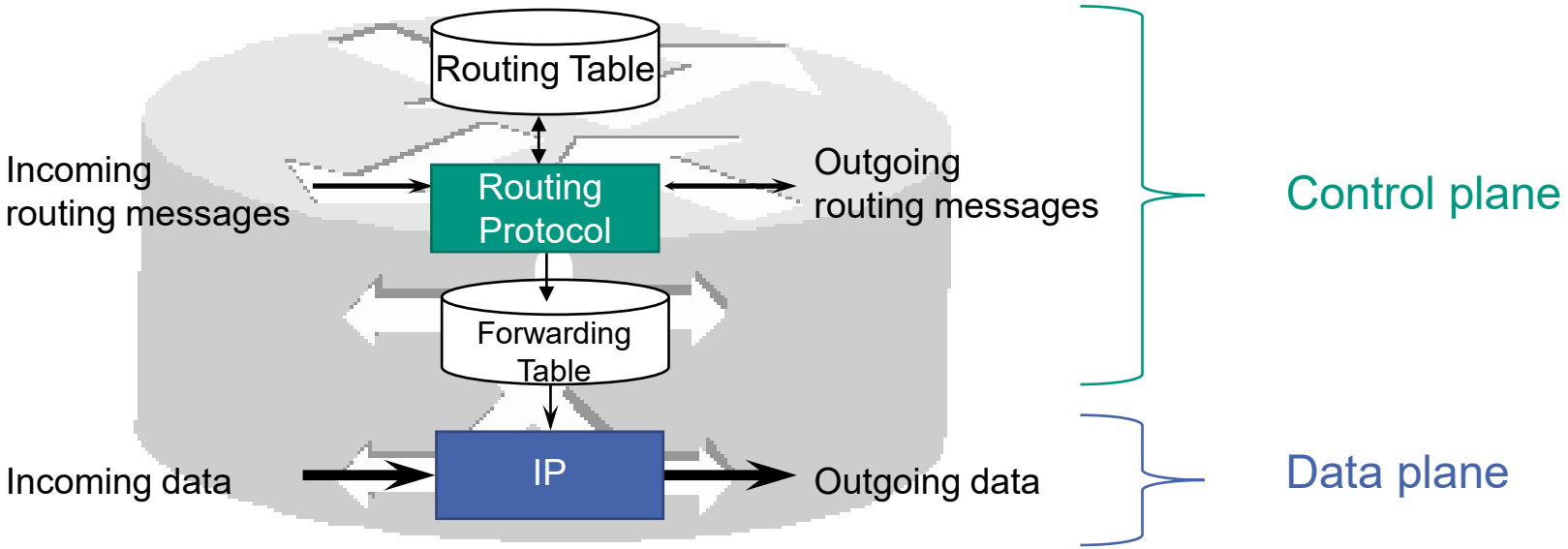


5 Programmable networks

5.1 Motivation

High Level View on Traditional IP Networks

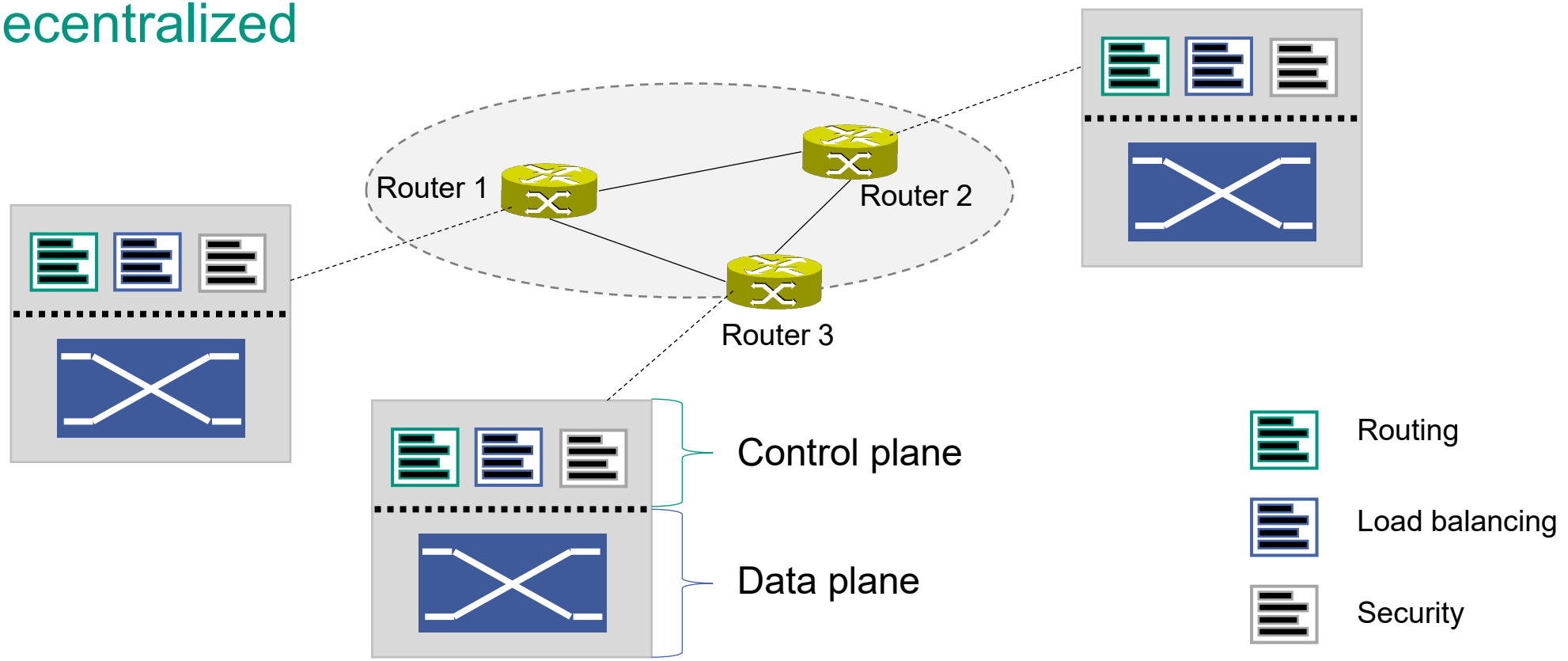
- Recap: abstract view on an IP router



- Control plane
  - Exchange of routing messages for calculation of routes …
  - Additional tasks, such as load balancing, access control, …
- Data plane
  - Forwarding of packets at layer 3

High Level View on Traditional IP Networks

- Every router implements control and data plane functions
  - Control plane: software running on the router
  - Data plane: for high performance networking: networking chips (ASICs)
- Control is decentralized



Observations

- So why did we build IP networks like this?
  - Proven concept for 30+ years
  - High reliability → self-organisation, high redundancy
  - Highly optimized hardware → very fast
- Limitations
  - Manufacturer-specific management interfaces
  - Difficult (and often impossible) to introduce new functions
  - Complex, highly qualified operators required
  - Expensive (at least for core routers)

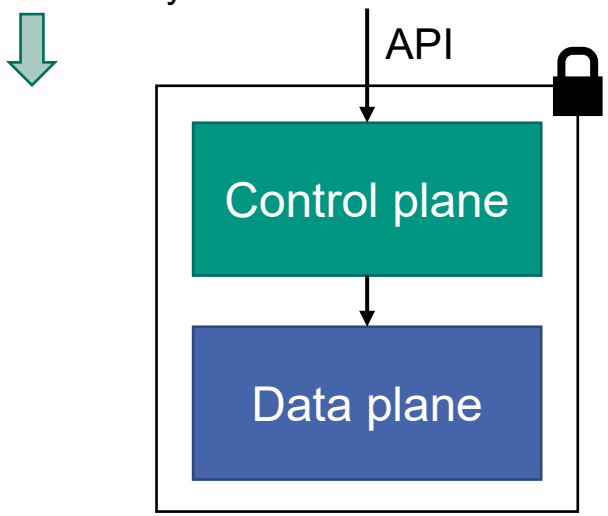
Limited flexibility for network operators

- Goals
  - Increase flexibility
  - Decrease dependencies on hardware and manufactures
  - Decrease cost: commercial off-the-shelf switches (cheaper)

Levels of Programmability

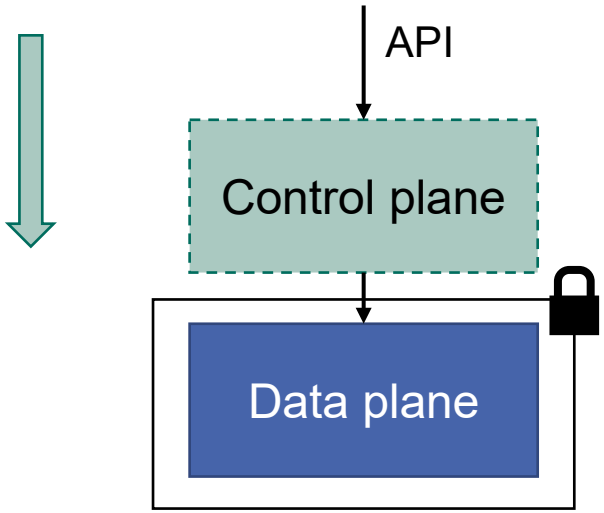
Traditional IP networking

programmability



- API provided by vendor allows some configurability (e.g., routing protocol)
- Algorithms in control and data plane cannot be changed

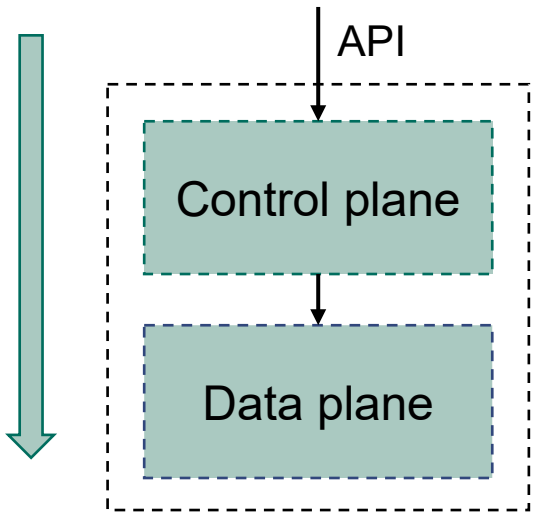
SDN with fixed-function data plane



- Algorithms in control plane can be changed
- Forwarding table in data plane can be programmed through specific interface

Open Flow

SDN with data plane programmability



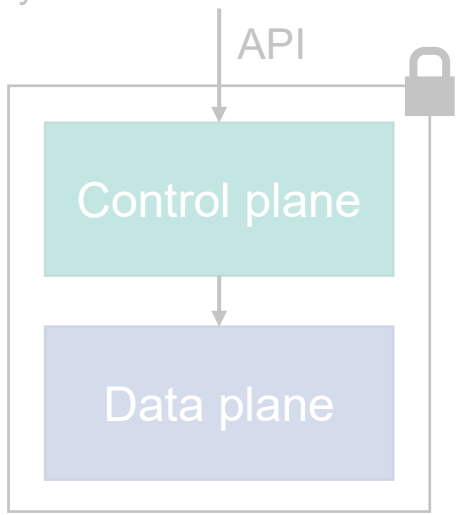
- Algorithms in control and data plane can be changed

P4

Levels of Programmability

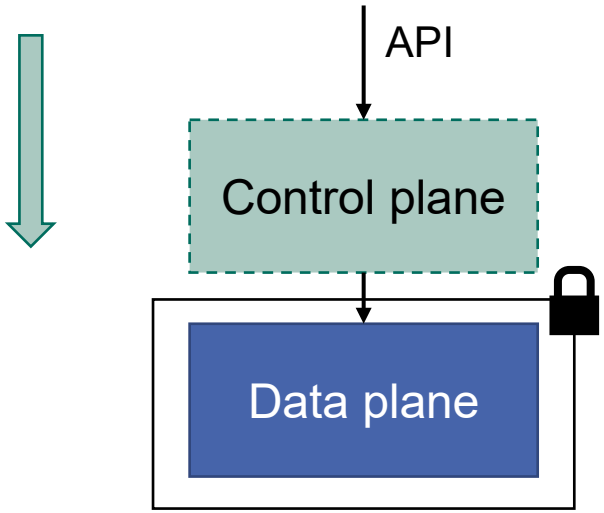
Traditional IP networking

programmability



- API provided by vendor allows some configurability (e.g., routing protocol)
- Algorithms in control and data plane cannot be changed

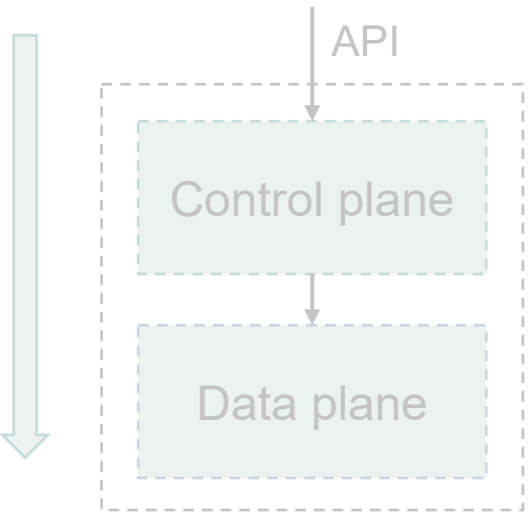
SDN with fixed-function data plane



- Algorithms in control plane can be changed
- Forwarding table in data plane can be programmed through specific interface

Open Flow

SDN with data plane programmability



- Algorithms in control and data plane can be changed

P4

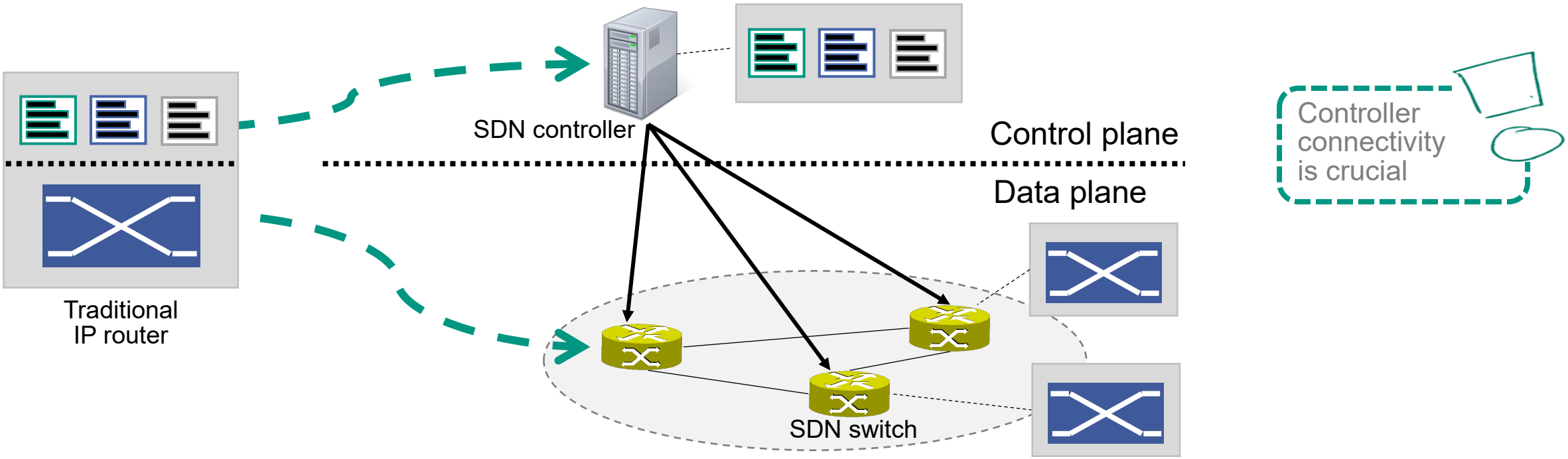
5.2 SDN

5.2.1 Concept

Software-Defined Network (SDN) Architecture

- Separation of control plane and data plane
  - Control functionality resides on a logically centralized SDN controller
    - Controller is executed on commodity hardware

  → Reduces need for specialized routing hardware
  - Data plane consists of simple packet processors (SDN switches)



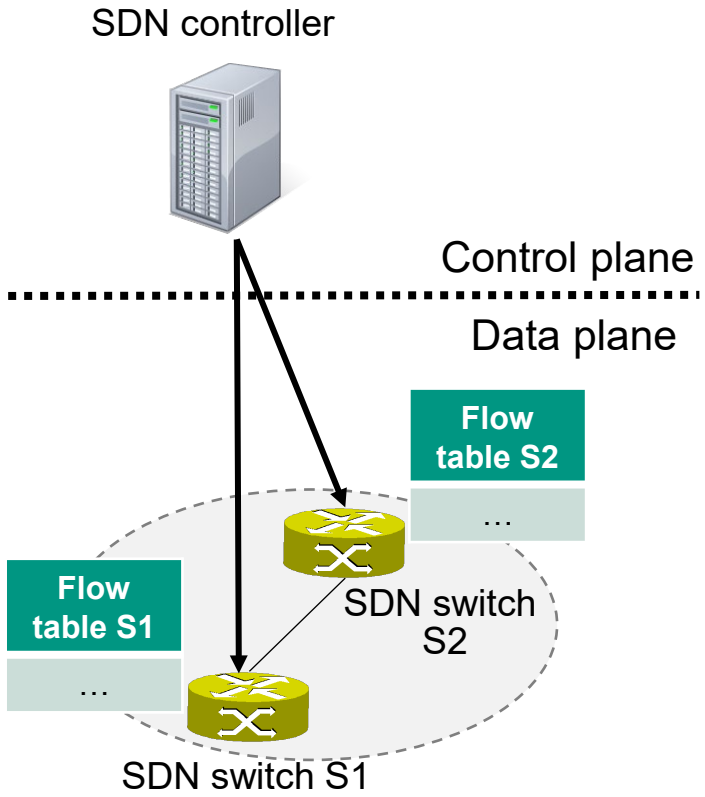
Characteristics of an SDN Architecture

- Separation of control and data plane
  - Data plane
    - simple but fast network switches
    - Switches execute flow rules (match, action) in their flow tables
  - Control plane
    - Servers and software that determine flow table of switches
- Flow-based forwarding
  - Using header fields of layers 2, 3, and 4
  - Flow rules are stored in the flow table
- Network control functions
  - Software runs on servers, not on data plane switches
  - Control plane consists of SDN controller and network control applications
  - Controller has global network view
    - Knows all switches and their configurations
    - Knows network topology
- Network is software-programmable
  - Through network control applications
    - E.g. routing, load balancing
  - SDN controller can execute multiple apps in parallel

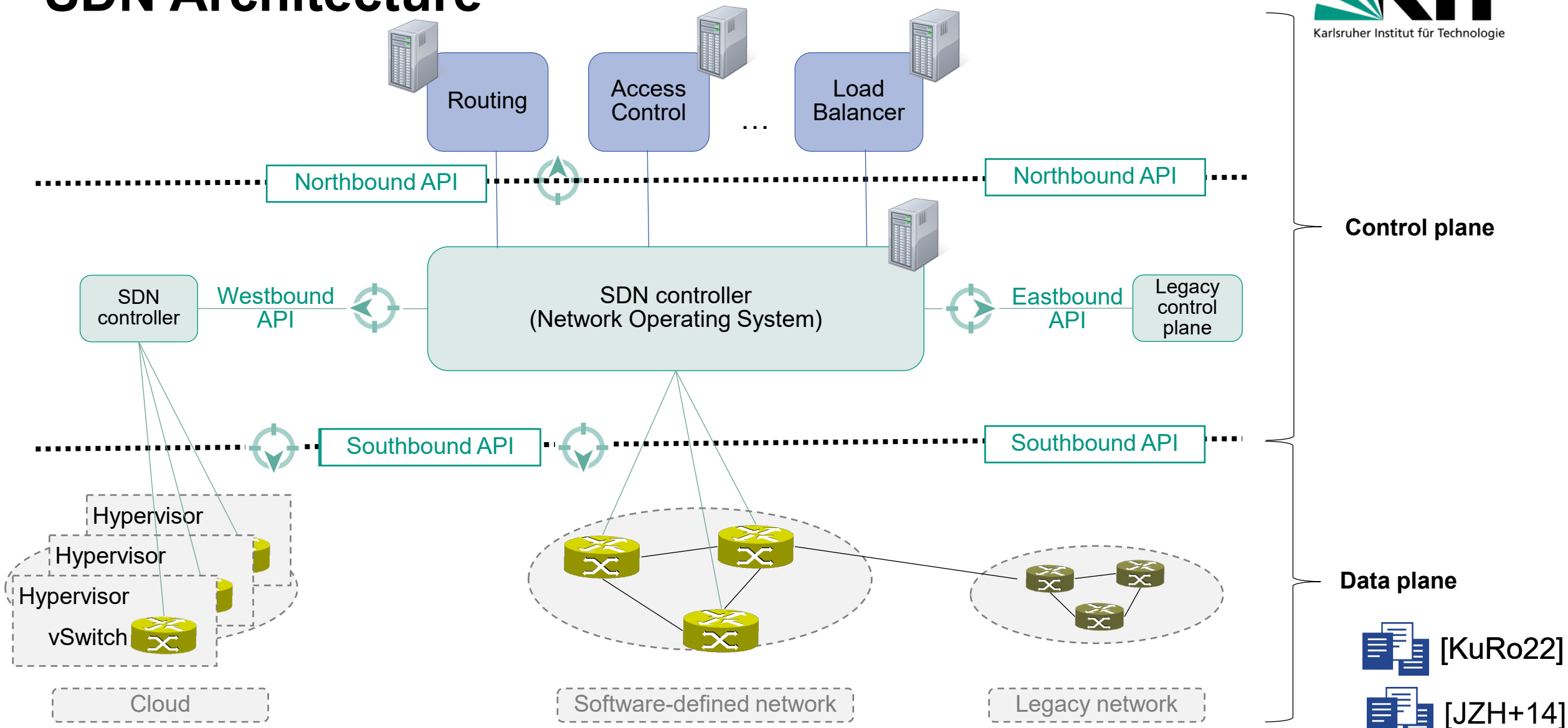
[KuRo22]

Basic Operation

- Control functionality is placed on the SDN controller
  - E.g., routing including routing table
- Forwarding table is placed on SDN switch
  - Called flow table in the context of SDN
  - Each entry includes
    - Set of header fields: determine suited entry in flow table of incoming packet
    - Set of counters
    - Set of actions: determine actions to be taken in case of a match
- Controller programs entries in flow table
  - … according to its control functionality
  - Requires a protocol between controller and switch



SDN Architecture



[KuRo22]

[JZH+14]

SDN Architecture: Planes

- Control Plane
  - Network apps perform network control and management tasks
    - Interact via northbound API with SDN controller
  - SDN controller
    - Control tasks are „outsourced“ from data plane to logically centralized control plane
    - E.g., standard tasks such as topology detection, ARP …
  - More complex tasks can be delegated to application servers
    - E.g., routing decisions, load balancing …

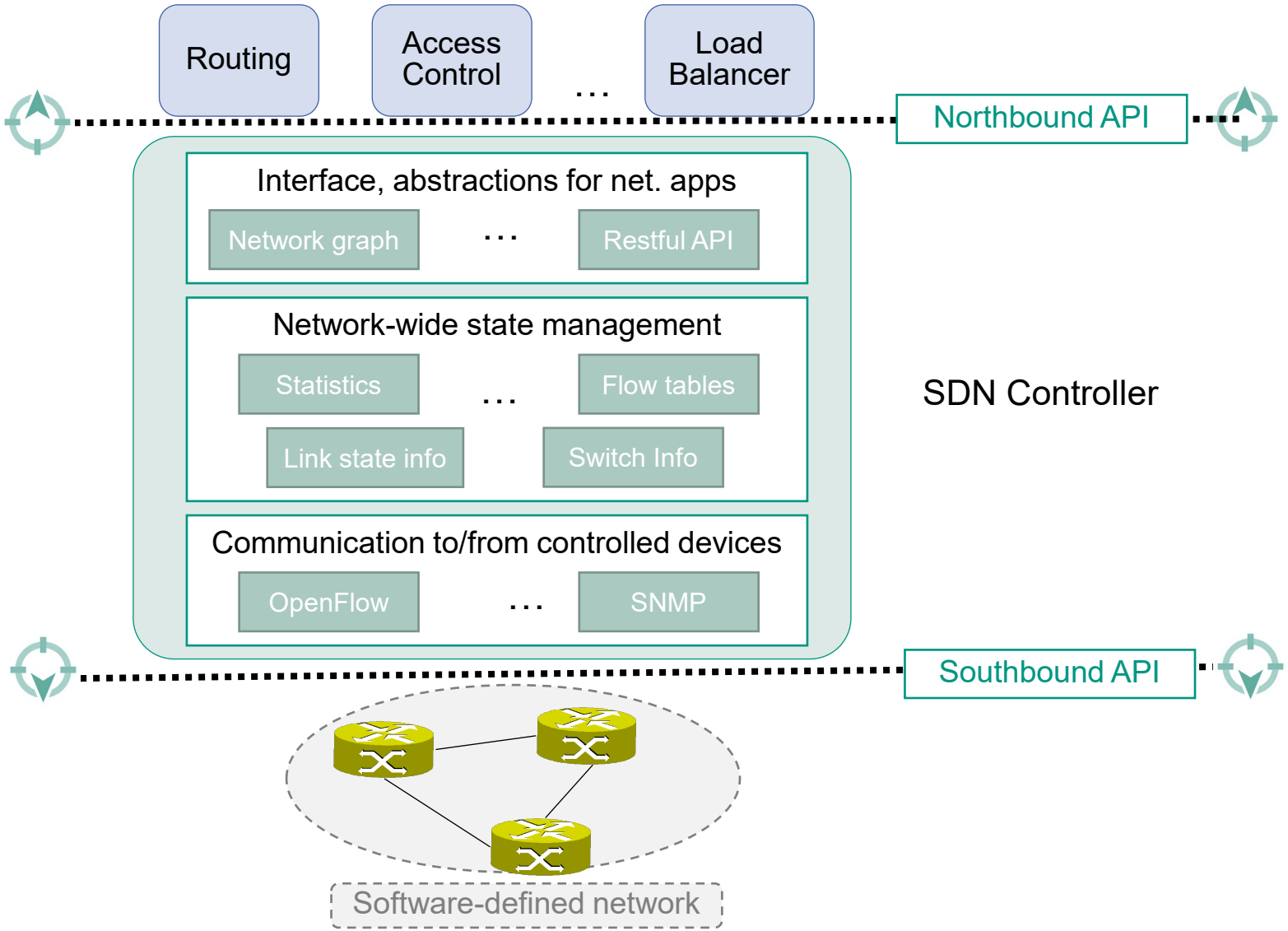
- Data Plane
  - Responsible for packet forwarding / processing
  - SDN switches are relatively simple devices
    - Efficient implementations in hardware (ASIC) or in software (virtual switches)
    - Supports basic operations such as match, forward, drop
  - Interacts via southbound API with control plane

SDN Architecture: Interfaces

- Northbound API: between controller and network apps
  - Exposes control plane functions to apps
  - Abstract from details, apps can operate on consistent network view
- Southbound API: between controller and switches
  - Exposes data plane functions to controller
  - Abstracts from hardware details
- Westbound API: between controllers
  - Synchronization of network state information
  - E.g., coordinated flow setup, exchange of reachability information
- Eastbound API: interface to legacy infrastructures
  - Usually proprietary

No widely accepted standard for north-bound / westbound interfaces currently exists. Controllers provide their own APIs.

Components of an SDN Controller

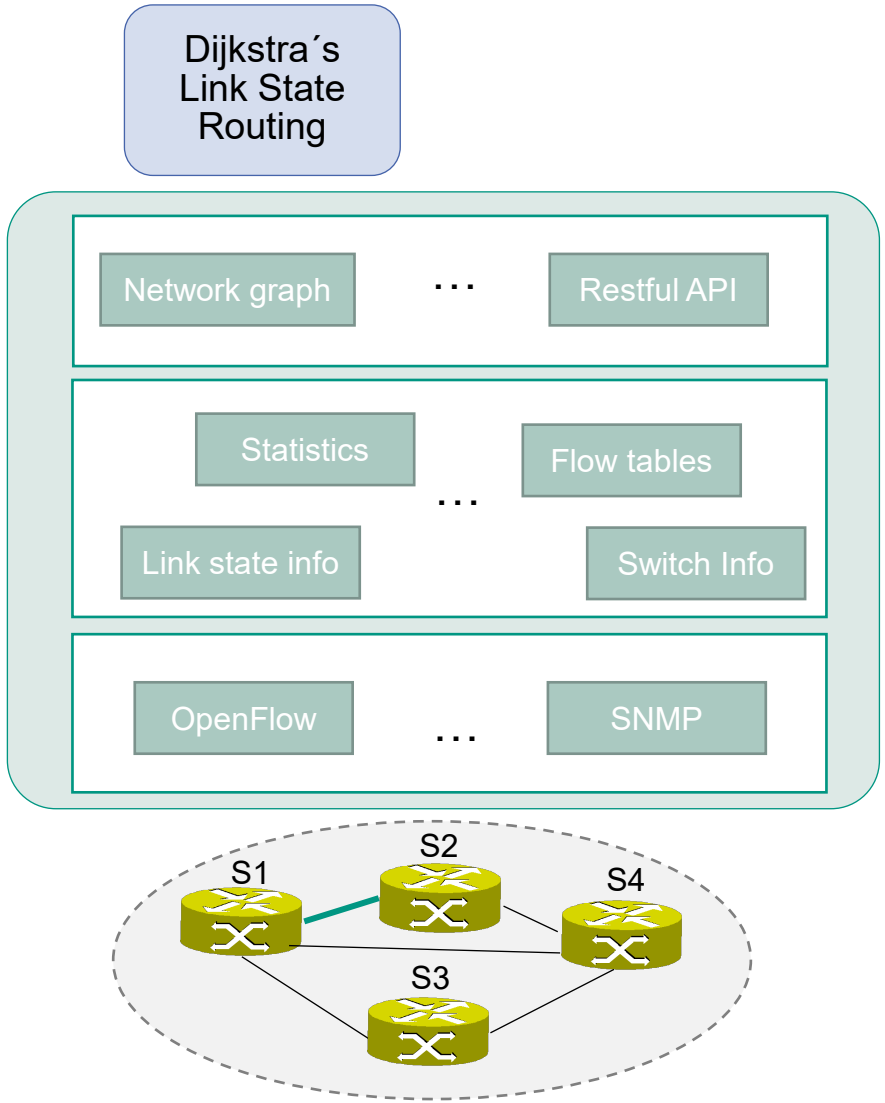


[KuRo22]

Components of an SDN Controller

- Interface to network control applications
  - Allows applications to read/write network state and flow tables within the state-management
- State-management
  - Collect statistics
  - Programm flow table
- Communication layer
  - Communication between SDN Controller and controlled network devices

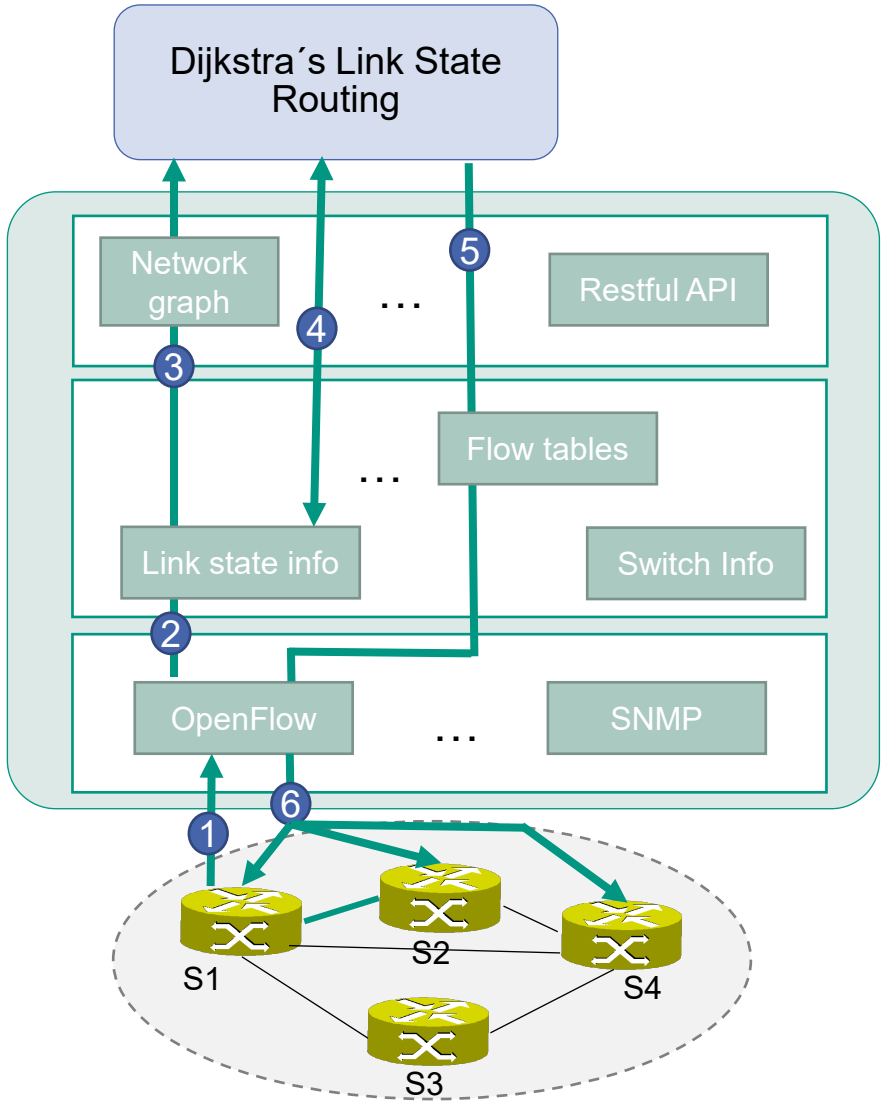
Example: Routing in an SDN Network



- Traditional IP-based routing with OSPF
  - Each router executes Dijkstra algorithm
  - Link State Advertisements (LSAs) flooded among all OSPF routers in network domain
- With SDN
  - Dijkstra algorithm executed as (network)
  - application outside the switches
  - Switches send link state updates to SDN Controller
- Scenario
  - Link between S1 and S2 goes down
  - Switch S3 not affected

[KuRo22]

Example: Routing in an SDN Network



1. S1 notifies SDN Controller
2. Controller receives link state message and informs link state manager which updates link state database
3. Routing app has registered to be notified when link state changes
4. Link state routing app interacts with link state manager to get updated information. Then computes shortest paths
5. Link state routing app interacts with flow table manager, which determines the flow tables to be updated
6. Flow table manager uses OpenFlow to update flow table entries at affected switches

[KuRo22]

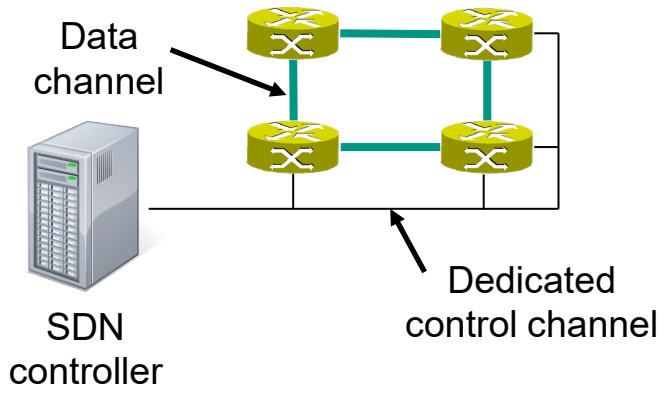
Controller Connectivity

- SDN requires connectivity between controller and switches
  - No connectivity
    → no exchange of control messages
    → no updates of flow tables
    → no further control of the network

- Two different connectivity modes exist
  - Out-of-band
  - In-band

Out-of-band

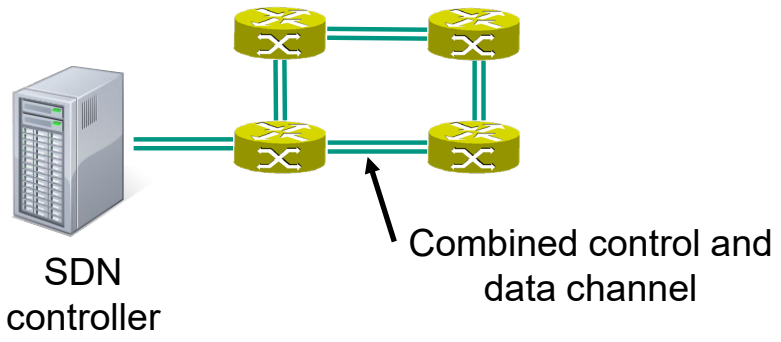
- Dedicated (physical) control channel for messages between controller and switch
  - Cost intensive



- How can operation on both channels be kept synchronous?

In-band

- Control messages use same channel as “normal” traffic (data)

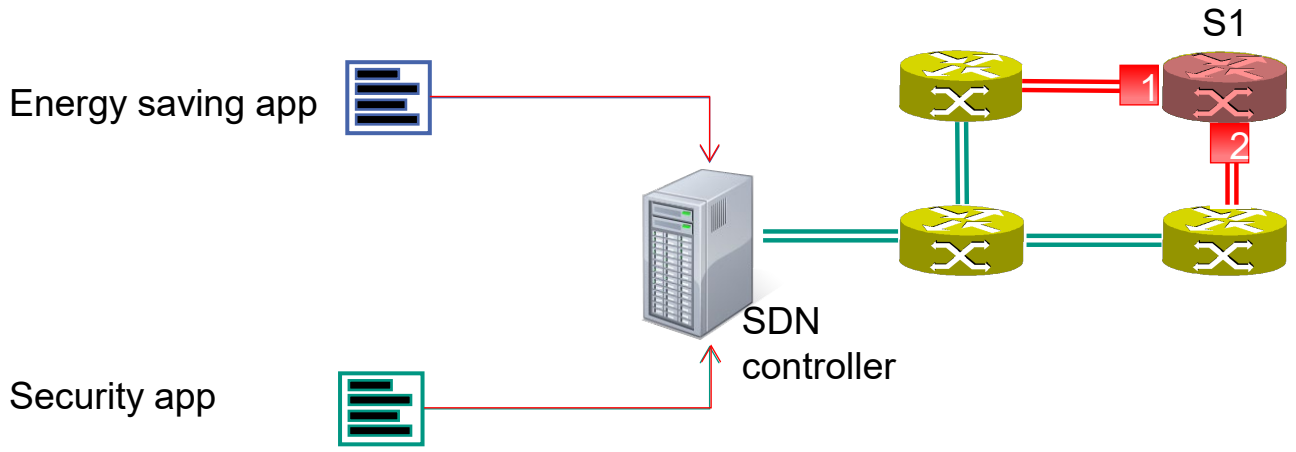


- Multiple applications can configure switch
  - Controller has to validate if rules impact control plane connectivity

In-Band Example

- Energy saving app disables port 1
- Security app disables forwarding on port 2

→Controller looses connectivity and cannot recover



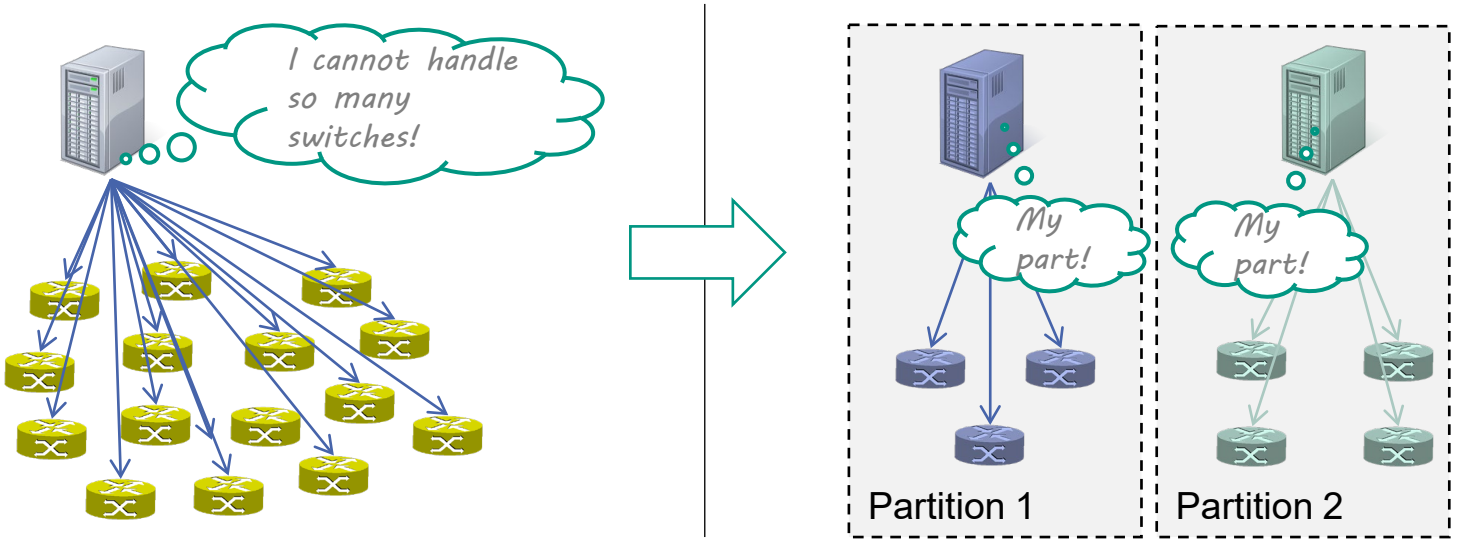
Scalability

- Logically centralized approach requires powerful controllers
  - Size / load of bigger networks can easily overload control plane
- Important parameters with scalability implications
  - Number of remotely controlled switches
  - Number of end systems / flows in the network
  - Number of messages processed by controller (e.g., Packet-in messages)
  - Communication delay between switches and controller
- Possible solution
  - Distributed controllers

[KaDu17]

Distributed Controllers

- Reasons for distributed controllers
  - Scalability: single controller is not powerful enough
  - Reliability: protection against failures / power outage
  - Incremental deployment: network grows over time
  - …



SDN is Widely Deployed: Examples

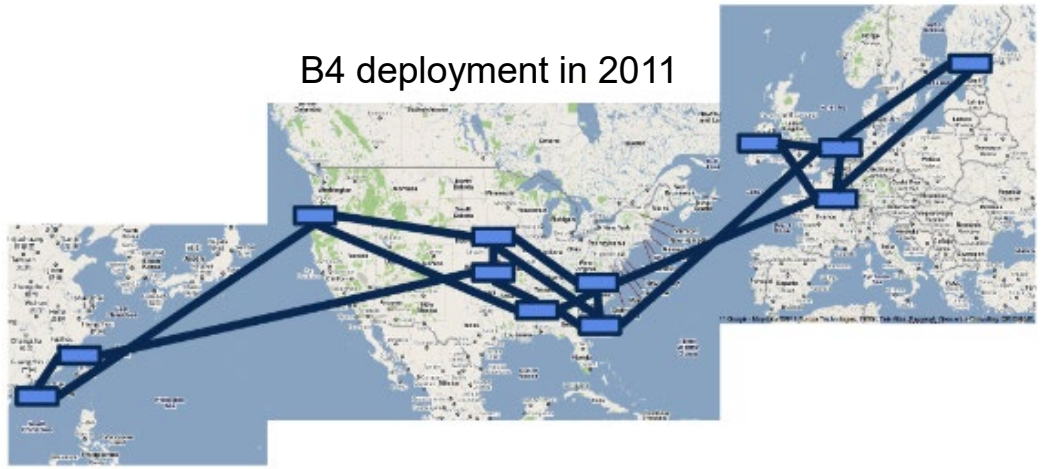
- Google
  - Jupiter: intra-datacenter networking
  - B4: inter-datacenter wide area networking
- Amazon Web Services
  - Virtual private cloud networking
- Meta
  - Terragraph: wireless urban area networking
- VMWare NSX
  - Virtualization and security platform for cloud networking
- NTT Communications
  - Software defined wide area networking in 190+ countries

Google´s Private WAN B4

- Google operates two separate backbone networks
  - B2 carries Internet facing traffic
    - WAN based on traditional protocols
      - BGP, IS-IS, RSVP-TE
  - B4 carries inter-data center traffic
    - More traffic than B2, grows faster than B2

Bandwidth requirements grew by 100x in a five year period

B4 deployment in 2011



- B4 is a large private wide area network
  - B4 interconnects Google´s data centers and server clusters
    - 19 „mega data centers“ in North America, Europe and Asia
    - Large number of clusters located at IXPs and within access ISPs (eyeball networks)

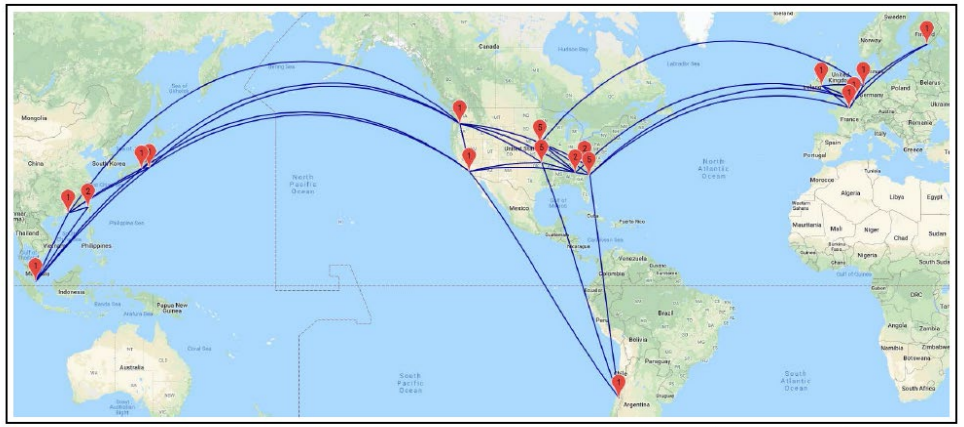


Figure 1: B4 global topology. Each marker indicates a site or multiple sites located in close geographical proximity. B4 consists of 33 sites as of January, 2018.

[Hong18, Jain13, KuRo22]

Google´s Private WAN B4

- B4 is based on SDN
  - Goal: increase efficiency, flexibility, scalability, availability
  - Uses out-of-band network for control traffic
    - Data traffic between data centers flows across different network
- Implements centralized traffic engineering
  - → increased utilization of WAN links 70% on average and near 100% on some links
    - Before that 30-40 % on average
- Also implements two routing protocols
  - BGP between data centers and
  - IS-IS within data center
  - → B4 can operate even if TE does not work
  - TE paths are overlaid on ISIS/BGP routes
    - Higher priority flow rule for TE
- Various improvements over time in order to scale with increasing requirements
  - Hierarchical network topology instead of flat network
  - Hierarchical traffic engineering
  - …

[Hong18, Jain13, KuRo22]

Decentralized SDN Architecture

Current Research

Karlsruher Institut für Technologie

- Recently published paper by Google
- Availability
  - Network operator´s highest priority
  - → avoid complex failures → simplify network

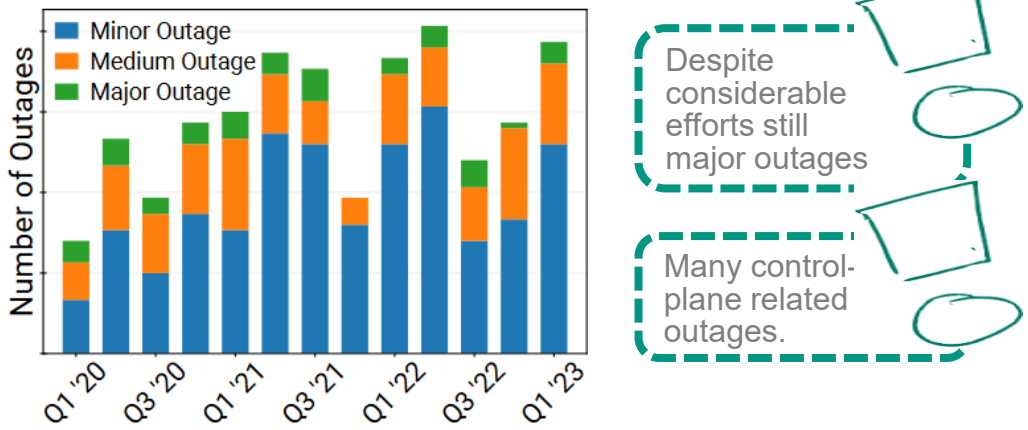


Figure 1: *Historical outage frequency in our WAN. Outages are classified internally based on several factors, including perceived user impact, duration, and blast radius (single or multi-region).*

- Observation
  - SDN control infrastructure introduces significant complexity
    - Out-of-band network in B4 WAN

- Decentralized SDN architecture: dSDN
  - Every router runs an operator-defined dSDN controller
    - Constructs global network view
      - Uses a flooding-based dissemination protocol
      - Establishes common view on topology
    - Locally runs TE algorithm
      - Computes capacity-aware paths
    - Implements strict source routing
      - Transported by lables in the packet

- Back to the roots?
  - Routers now come with (better) possibilities to deploy operator-defined solutions
    - New generation of APIs

Google presents general design. Tested in testbed, but not yet on B4 WAN

[Krent24]

5.2.2 Data Plane

Flows and Flow Table

- Flows
  - Flow is sequence of packets traversing a network that share a set of header field values
  - Flow is identified through match fields, e.g., IP address, port number
- Flow table contains, among others, match fields and actions
  - Matches select appropriate flow table entry
  - Actions are applied to all packets that satisfy a match

Flow table (simplified)

	Match fields	Actions
	Src = 10.0.0.2/32	Drop
	Dst = 10.0.0.0/16	Output 1
	…	…

Drop all packets from source subnet 10.0.0.2/32

Forward all packets to subnet 10.0.0.0/16 via switch port 1

- Flow rule
  - Decision of controller
  - Described in form of match fields, actions, switches

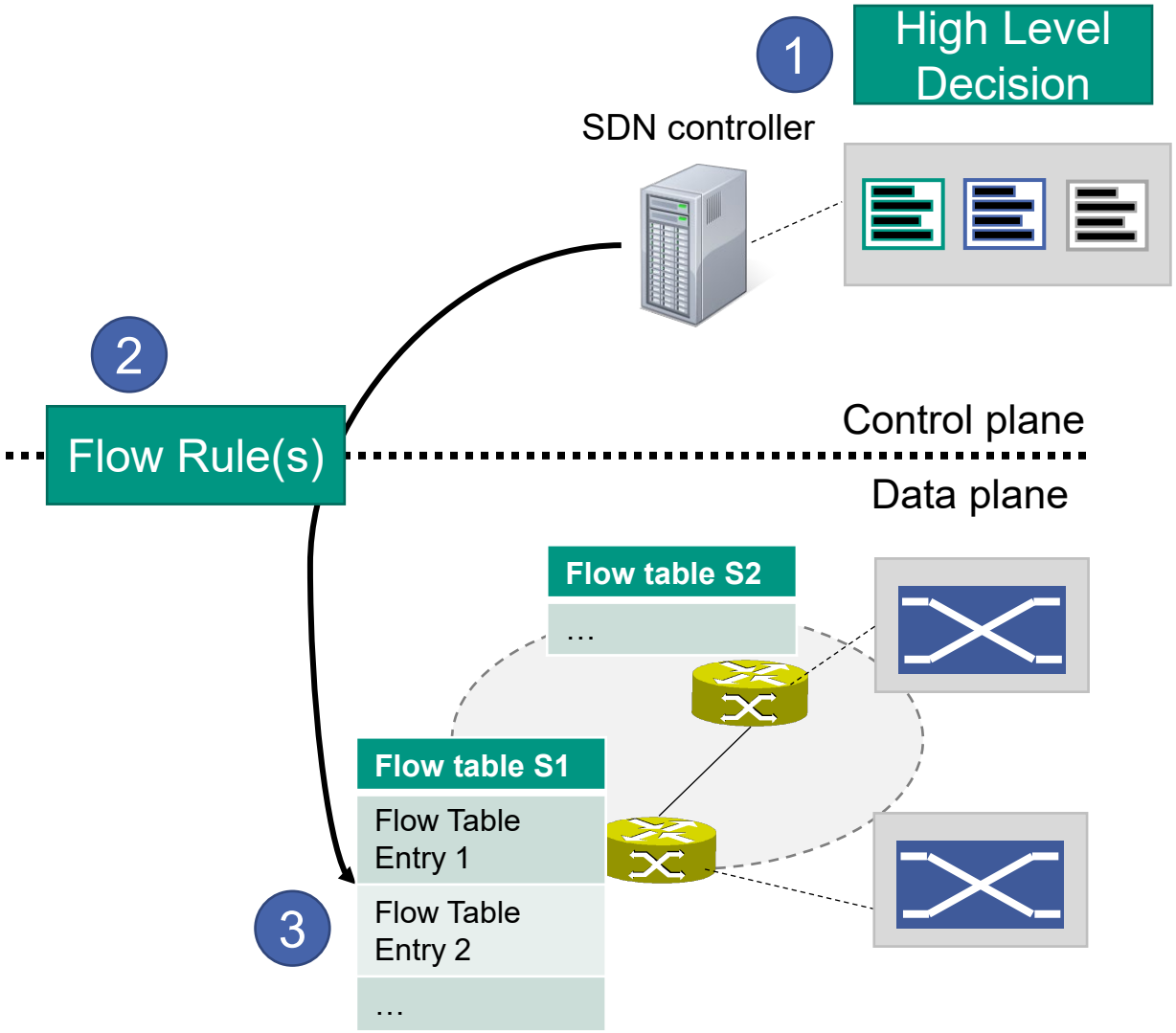
Actions

- Flow table entry provides zero or more actions
  - Determine processing of packet in case of a match
- In case of multiple actions
  - Performed in the order specified in the list
- Some important actions
  - Forwarding
    - To one output port, broadcasted to all ports, multicasted to seleced set of ports
    - To remote controller for this device
  - Dropping
    - If entry does not contain an action the packet should be dropped
  - Modify-field
    - Value in header may be changed before forwarding

5.2.3 Control Plane

Flow Rule and Flow Table Entries

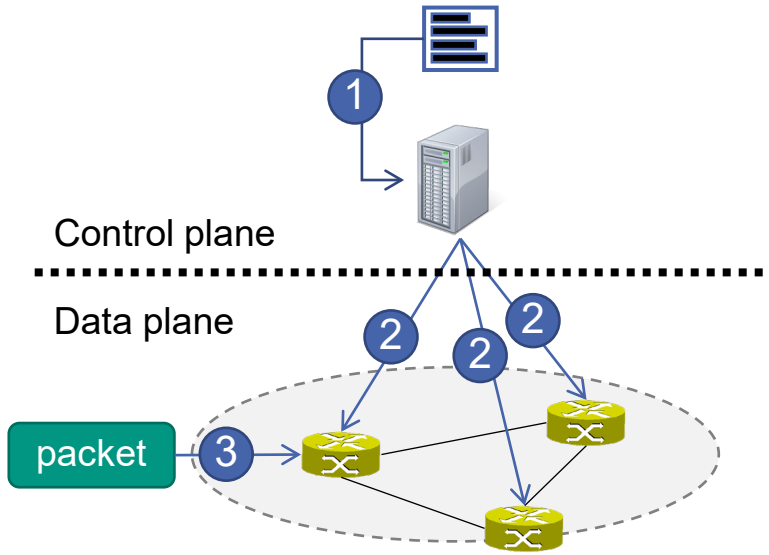
1. Controller (more precise: app executed by controller) makes a high level decision, for example
   a) Traffic for destination X has to be dropped
   b) Connection between end system A and B has to go through switch S4
   c) …
2. High level decision is represented in a certain format, i.e., as a set of flow rules in the form of match fields, actions and switches
3. Flow rules are transmitted (“installed”) to switches with the help of a communication protocol.
   They are stored as flow table entries in flow tables



Flow Programming

- SDN provides two different modes
  - Proactive flow programming
    - Flow rules are programmed before first packet of flow arrives
  - Reactive flow programming
    - Flow rules are programmed in reaction to receipt of first packet of a flow

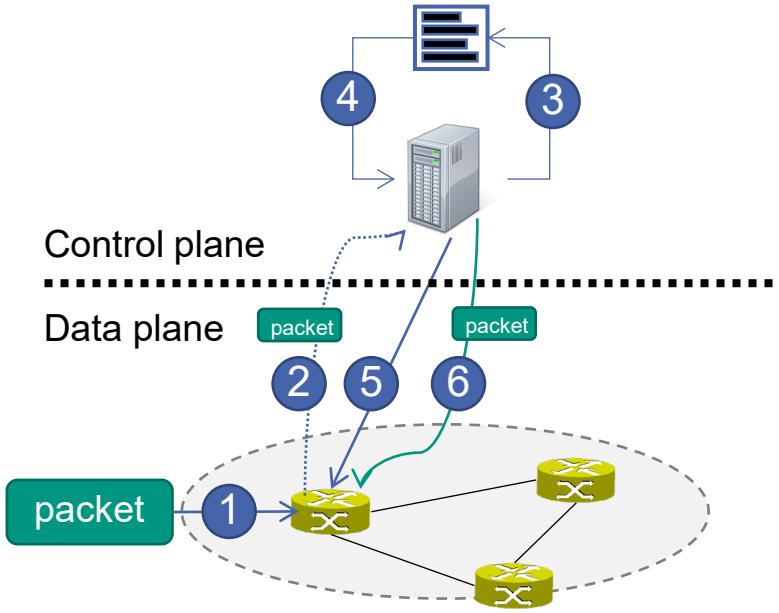
Proactive Flow Programming



"Flow rules are programmed before first packet of flow arrives"

1. Network application (App) makes a decision for the packet beforehand (for all switches)
2. Flow rules are sent to switches and new flow table entries are added to flow tables
3. Lookup in flow table → Match → Packet is NOT forwarded to controller

Reactive Flow Programming



„Flow rules are programmed in reaction to receipt of first packet of a flow“

1. Packet arrives at switch
2. Lookup in flow table → Mismatch → Packet forwarded to controller
3. Controller notifies network application responsible for deciding how to process packet
4. App has come to a decision. Here: a new flow rule has to be installed
5. Flow rule is sent to switch and new flow table entry is added to flow table
6. Packet is re-injected into switch

Three Important Interactions

Flow rule sent to switch which results in new flow table entry

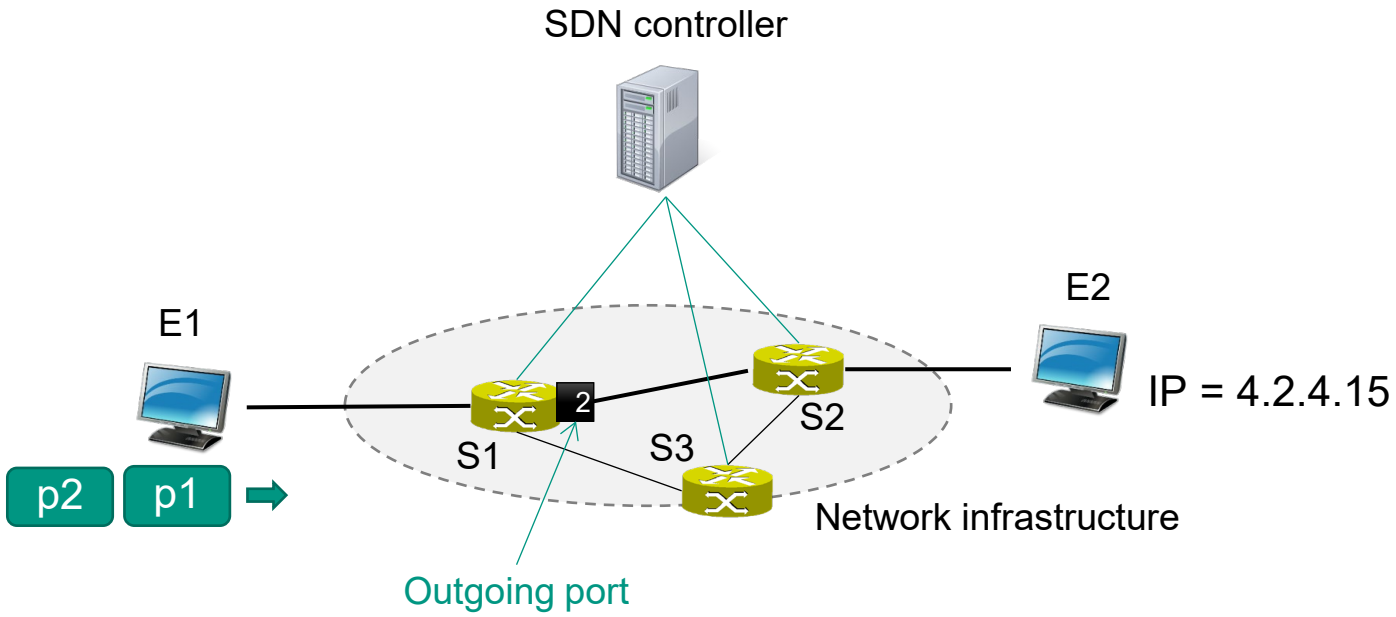
Packet forwarded to controller, e.g., after mismatch in flow table

Packet re-injected to switch, e.g., to not loose a packet that was sent to controller

Used in reactive and proactive case

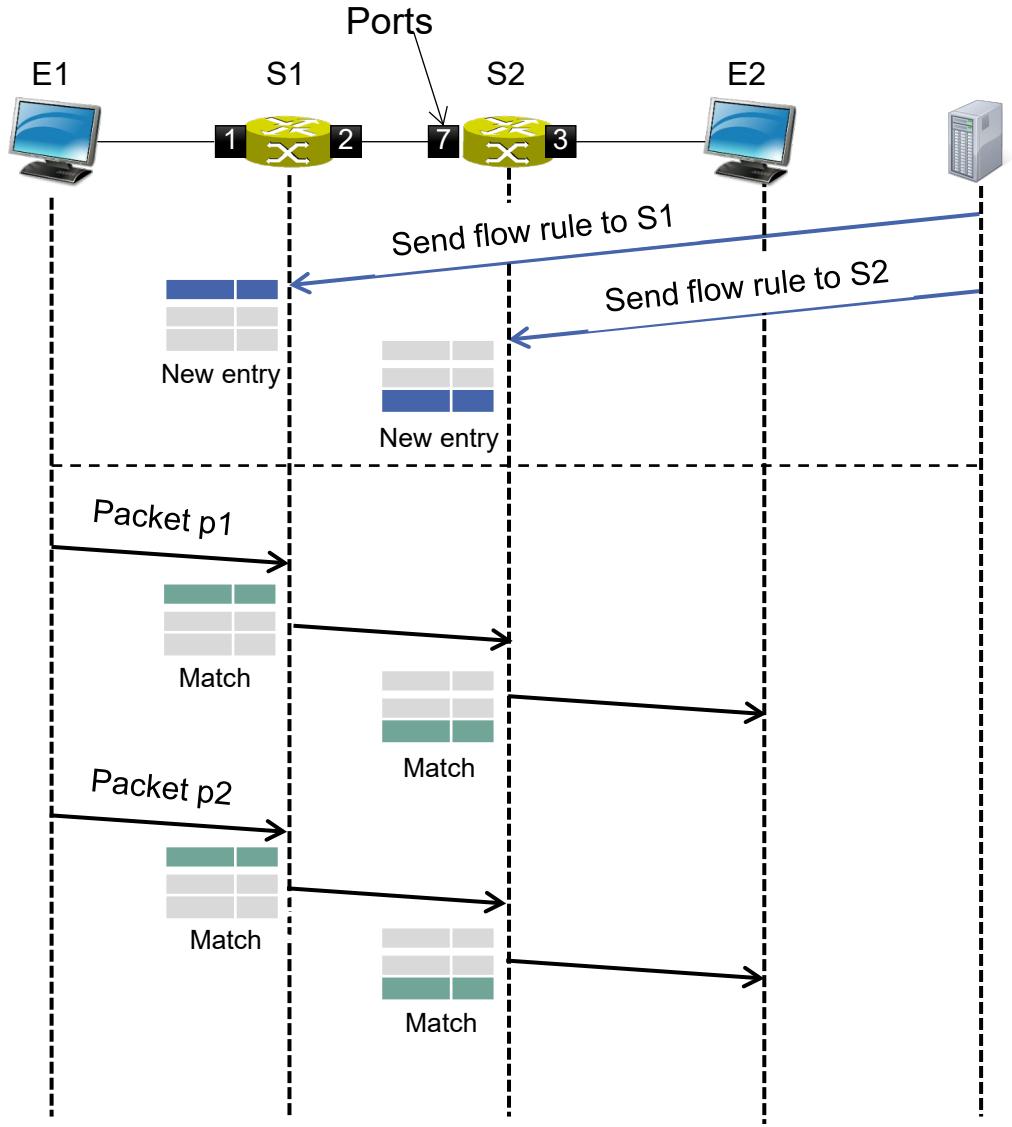
Primarily used in reactive case

Example Scenario



- Goal
  - Forward packets p1 and p2 to end system E2 with IP address 4.2.4.15

Example: Proactive Flow Programming



Controller programs flow table entries proactively into switch S1 and S2 before packet p1 and p2 arrive

Switch S1 and S2 forward packets p1 and p2 based on programmed flow table entries

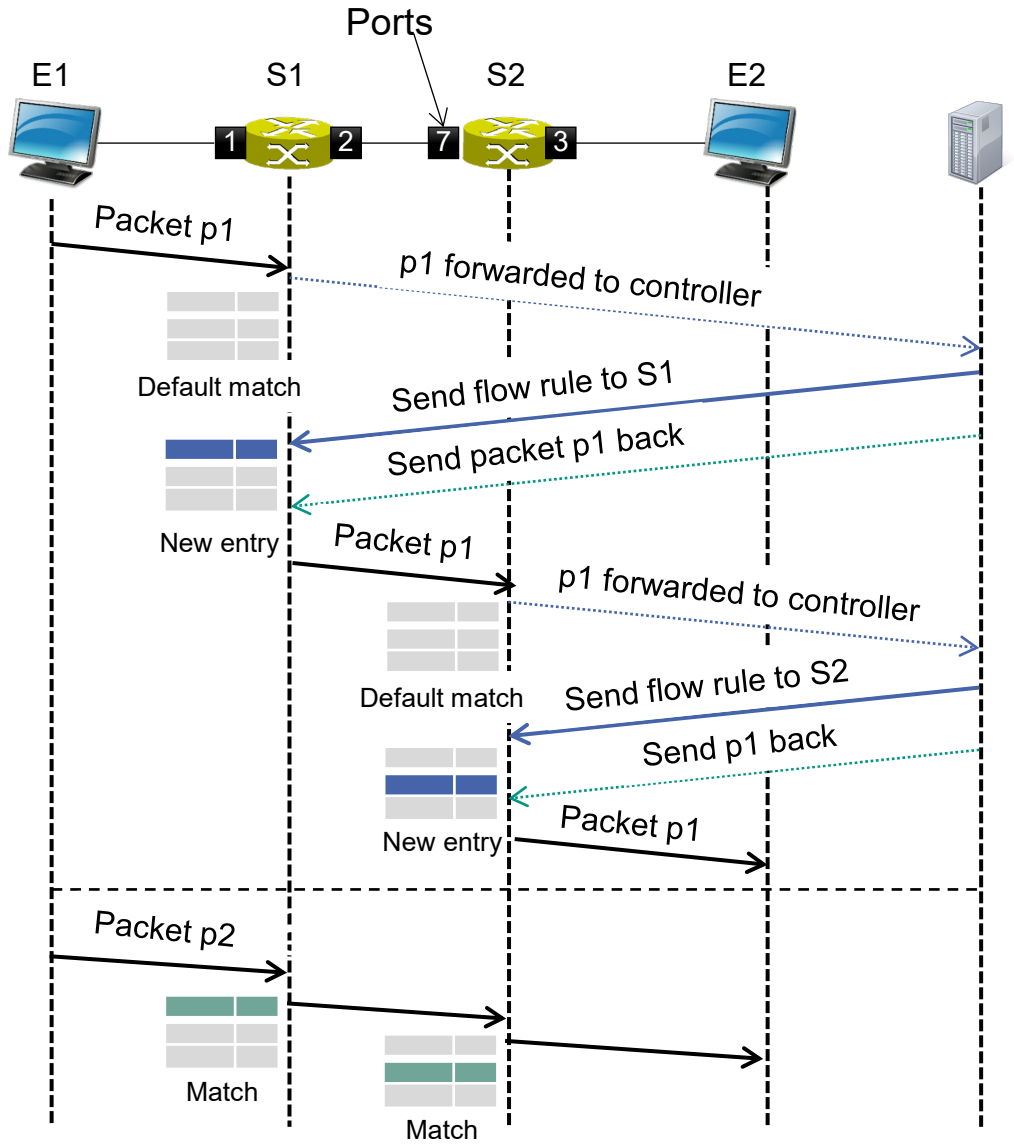
Flow table of S1

Match Fields	Action
IPv4Dst = 4.2.4.15	Output 2
…	…
*	Controller

Example: Proactive Flow Programming

1. Controller generates new flow rule for switch S1
   M: Destination-IP = 4.2.4.15
   A: Forward to port 2
2. Controller sends flow rule <M, A, S1> to switch S1
3. Switch S1 stores rule in local flow table
4. Controller generates new flow rule for switch S2
   M: Destination-IP = 4.2.4.15
   A: Forward to port 3
5. Controller sends flow rule <M, A, S2> to S2
6. Switch S2 stores rule in local flow table
7. Packet p1 arrives at S1 (port 1)
8. Lookup in flow table (S1) → Match → p1 forwarded on port 2 (to switch S2)
9. Packet p1 arrives at S2 (port 7)
10. Lookup in flow table (S2) → Match → p1 forwarded on port 3 (to E2)
13. Packet p2 arrives at switch S1 (port 1)
14. Lookup in flow table (S1) → Match → p2 forwarded on port 2 (to switch 2)
15. Packet p2 arrives at switch S2 (port 7)
16. Lookup in flow table (S2) → Match → p2 forwarded on port 3 (to E2)

Example: Reactive Flow Programming



Controller programs flow table entries reactively into switch S1 and S2

Flow table of S1

Match Fields	Action
IPv4Dst = 4.2.4.15	Output 2
…	…
*	Controller

Switch S1 and S2 forward packet based on programmed flow table entries

Example: Reactive Flow Programming

1. Packet p1 arrives at switch S1 (port 1)
2. Lookup in flow table of S1 → Mismatch, i.e., default match is used → Packet forwarded to controller
3. Controller generates new rule for switch S1
   M: Destination-IP = 4.2.4.15
   A: Forward to port 2
4. Controller sends rule <M, A, S1> to switch S1
5. S1 stores rule in local flow table
6. Controller sends packet p1 back to S1
7. S1 forwards packet p1 on port 2 (to switch S2)
8. Packet p1 arrives at S2 (port 7)
9. Lookup in flow table of S2 → Mismatch, i.e., default match is used → Packet forwarded to controller
10. Controller generates new rule for switch S2
    M: Destination-IP = 4.2.4.15
    A: Forward to port 3
11. Controller sends rule <M, A, S2> to switch S2
12. S2 stores rule in local flow table

Example: Reactive Flow Programming

13. Controller sends packet p1 back to S2
14. S2 forwards p1 on port 3 (to E2)
15. Packet p2 arrives at switch S1 (port 1)
16. Lookup in flow table (S1) → Match → p2 forwarded on port 2 (to switch 2)
17. Packet p2 arrives at switch S2 (port 7)
18. Lookup in flow table (S2) → Match → p2 forwarded on port 3 (to E2)

Reactive vs. Proactive Flow Programming

- Proactive
  - Flow table entries have to be programmed before actual traffic arrives
    - Usually coarse grained pre-defined decisions
    - Not always applicable
  - No additional delays for new connections
  - Loss of controller connectivity does not disrupt traffic

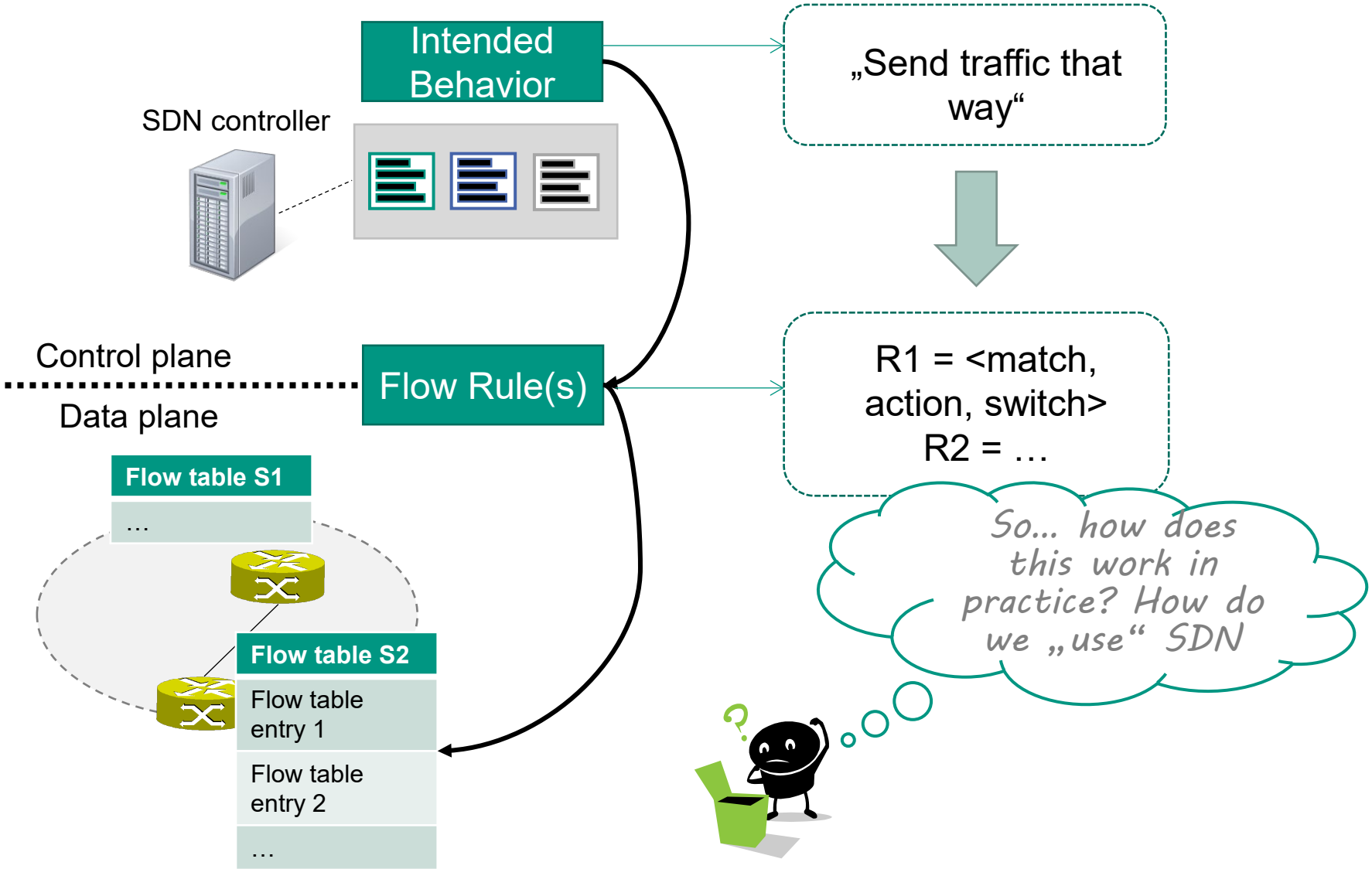
- Reactive
  - Allows fine grained on-demand control
    - Increased visibility of flows that are active in the network
  - Flow setup latency
    - Setup time required for each new flow (see next slide)
    - Overhead can be high compared to flow duration in case of short lived flows
  - New flows cannot be installed if controller connectivity is lost

Flow Setup Latency

- Flow setup latency introduced by remote programming
  - Flow setup: rule(s) generated at SDN controller and installed at switch (e.g., by OpenFlow)
- Consists of
  - Control plane processing delay
  - Propagation delay between switch and controller
  - Data plane processing delay
- Directly affects control latency of the network
  - Can also affect end-to-end delay
  - Can lead to additional buffering at switch or to additional messages sent to controller
- Possible solutions
  - Proactive setup where applicable → avoids setup latency
  - Clever controller placement (difficult problem)

5.2.4 Workflow and Primitives

High Level View



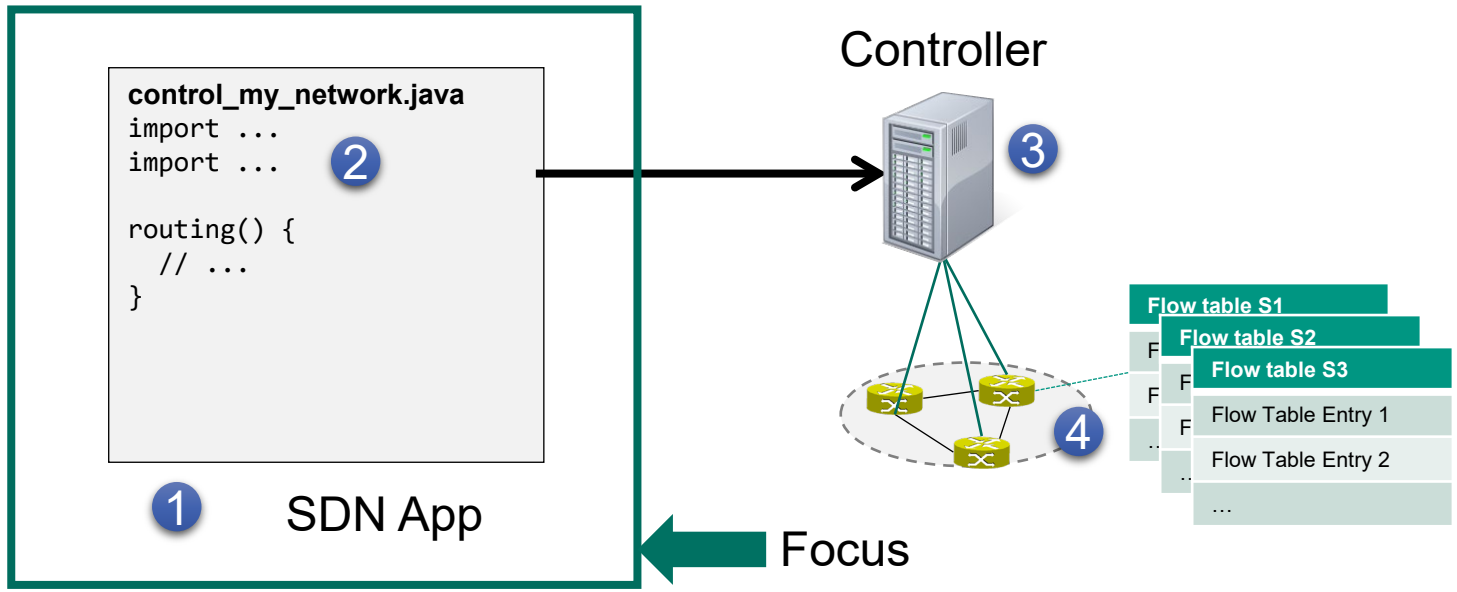
SDN Workflow in Practice

(1) We need a piece of software (app) that realizes the new behavior
- control_my_network.java

(2) We need primitives to assist with creating the app
- import OFMatch, OFAction, …

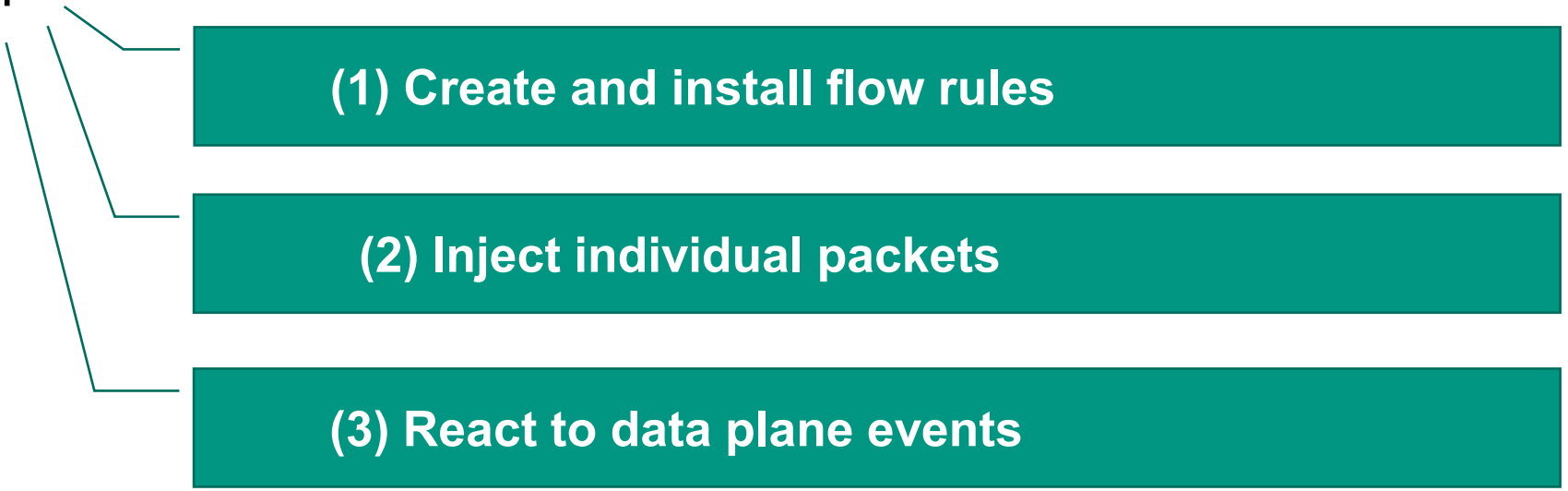
(3) We need a runtime environment that can execute our app
- $ ./myController --runApp control_my_network.java

(4) We need hardware support for SDN in the switches
- Flow table(s)



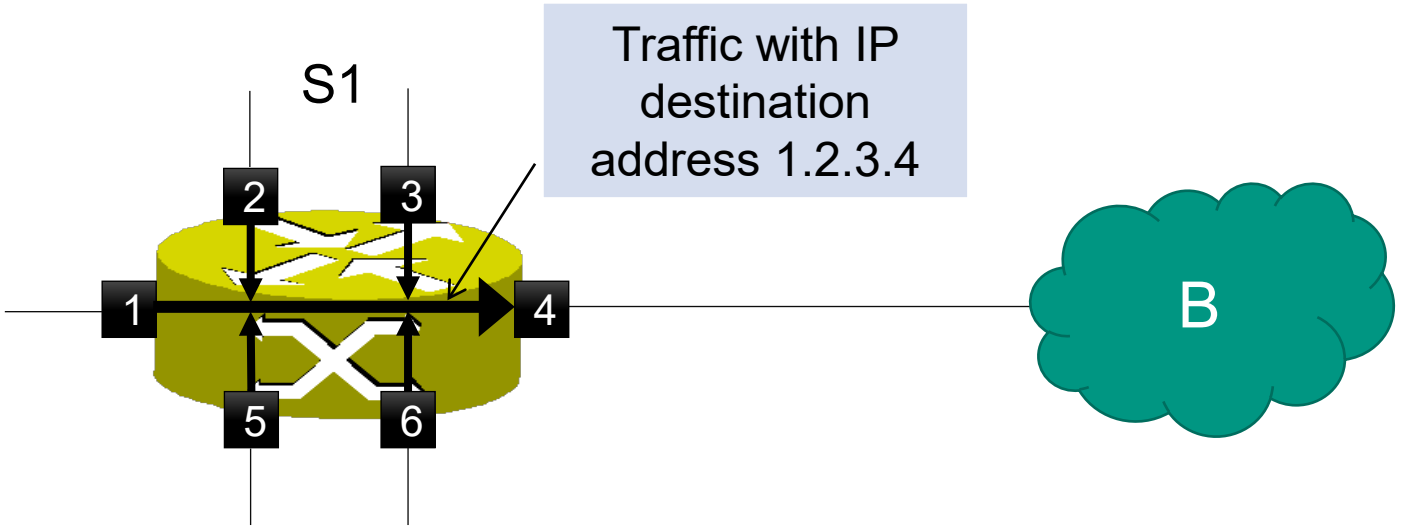
Primitives for SDN Programming

- Overall goal?
  - From intended behavior to low level flow rules
- This requires SDN programming primitives
- Three important areas to cover



Create and Install Flow Rules

- Example: Traffic with IP destination address 1.2.3.4 has to be forwarded to network B by switch S1



- Needed: App that implements the corresponding logic
  - Represent the decision as flow rules
  - Program appropriate flow table entries into the switch

Create and Install Flow Rules

- Example application: static_forwarding.java
  - Creates a new flow rule
  - Sends the flow rule to S1

```
import ...

onConnect(switch) {
  if (switch.id == 1) {
    r = Rule()
    r.MATCH('IP_DST', '1.2.3.4')
    r.ACTION('OUTPUT', 4)
    send_rule(r, switch)
  }
}
```

static_forwarding.java

- Remark: we use a simple pseudo programming language
  - Language used in practice depends on controller
  - Different controllers support different languages
    - Java, Python, C, C++, …

Create and Install Flow Rules

```
import ...

onConnect(switch) {
  if (switch.id == 1) {
    r = Rule()
    r.MATCH('IP_DST', '1.2.3.4')
    r.ACTION('OUTPUT', 4)
    send_rule(r, switch)
  }
}
```

static_forwarding.java

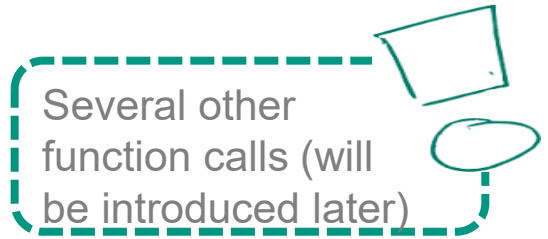
- onConnect → entry point to implement custom logic
  - Called when control connection to switch is established
  - Contains reference to switch as parameter
  - If some logic only applies to certain switches, this distinction has to be made manually (see if clause with match on the switch identifier)

Create and Install Flow Rules

```
import ...

onConnect(switch) {
  if (switch.id == 1) {
    r = Rule()
    r.MATCH('IP_DST', '1.2.3.4')
    r.ACTION('OUTPUT', 4)
    send_rule(r, switch)
  }
}
```

static_forwarding.java

- `Rule` → data structure that holds a single flow rule
  - Follows <match, action, switch> template
  - Can be modified with function calls
  - Two mandatory function calls
    - `r.MATCH(field_name, value)`
    - `r.ACTION(action_name, value)`



Create and Install Flow Rules

```
import ...

onConnect(switch) {
  if (switch.id == 1) {
    r = Rule()
    r.MATCH('IP_DST', '1.2.3.4')
    r.ACTION('OUTPUT', 4)
    send_rule(r, switch)
  }
}
```

static_forwarding.java

- `Rule.MATCH` → select packets based on certain criteria
  - Used here to select all packets with a specific destination IP address
  - Important match fields
    - Ingress port
    - MAC / IPv4 / IPv6 addresses
    - TCP ports
    - VLAN IDs
    - …

Create and Install Flow Rules

```
import ...

onConnect(switch) {
  if (switch.id == 1) {
    r = Rule()
    r.MATCH('IP_DST', '1.2.3.4')
    r.ACTION('OUTPUT', 4)
    send_rule(r, switch)
  }
}
```

static_forwarding.java

- `Rule.ACTION` → specify what happens to matched packets in the switch
  - Most important actions
    - `m.ACTION('OUTPUT', 7)` `// forward on port 7`
    - `m.ACTION('DROP')` `// drop the packet`

Create and Install Flow Rules

```
import ...

onConnect(switch) {
  if (switch.id == 1) {
    r = Rule()
    r.MATCH('IP_DST', '1.2.3.4')
    r.ACTION('OUTPUT', 4)
    send_rule(r, switch)
  }
}
```

static_forwarding.java

- `send_rule` → install flow rule in switch
  - Creates a new flow table entry or overwrites an existing one

Flow table of S1

Match Fields	Action
IP_DST = 1.2.3.4	Output 4
…	…

New flow table entry in S1 after `send_rule()` is executed

Overview: Matches

Usage	Description
`r.MATCH('IP_DST', '1.2.3.4')`	Match on a single IP destination address
`r.MATCH('IP_SRC', '1.2.3.4')` `r.MATCH('IP_DST', '9.8.7.6')`	Match can be used multiple times for more specific matching, e.g., on a pair of source and destination IP address
`r.MATCH('ETHERTYPE', 0x0806)`	Match all packets using a specific protocol inside the ethernet frame (here ARP)
`r.MATCH('MAC_DST', 'H1')`	Symbols may be used for parameters if it makes sense (e.g., H1 instead of 03:4b:3c:12:14:aa)
`r.MATCH('IP_DST', '1.0.0.0/8')`	Address fields usually support longest prefix matching
`r.MATCH('INPORT', 5)`	Matches all packets arriving on port 5
`r.MATCH('*')`	Match on everything, i.e., the most generic match possible

Overview: Actions

Usage	Description
`r.ACTION('DROP')`	Drop packet
`r.ACTION('FLOOD')`	Send packet to all ports („flooding“) except the one where it was received
`r.ACTION('CONTROLLER')`	Send packet to the controller
`r.ACTION('OUTPUT', 14)`	Send packet to port 14
`r.ACTION('GOTO', 4)`	If the switch has more than one flow table, the GOTO command is used to redirect a packet to another table (here flow table 4)
`r.ACTION('SET_FIELD', 'IP_SRC', '1.2.3.5')`	Overwrite IP source address of the packet with 1.2.3.5
`r.ACTION('SET_FIELD', 'TCP_DST', 40)`	Overwrite TCP destination port of the packet with 40

Priorities

- Priorities come into play if there are overlapping flow rules
  - No overlap = all potential packets can only be matched by at most one rule
  - Overlap = at least one packet could be matched by more than one rule
- First rule
  - `rule1.MATCH('IP_DST', '40.2.4.4')`
  - `rule1.ACTION('DROP')`
- Second rule
  - `rule2.MATCH('IP_DST', '40.0.0.0/8')`
  - `rule2.ACTION('OUTPUT', 1)`
- What happens if a packet arrives in a switch where both rules are installed (in the same flow table)?
  - `rule1.PRIORITY > rule2.PRIORITY` ➔ drop
  - `rule2.PRIORITY > rule1.PRIORITY` ➔ forwarded to port 1
  - Overlap has to be resolved → priorities needed!

Priorities

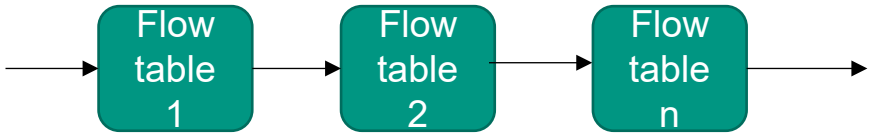
- Pseudo code used in this lecture assumes that all rules are created with same default priority (=1)
- If two rules can overlap, priority has to be changed explicitly
  - Higher values = higher priority

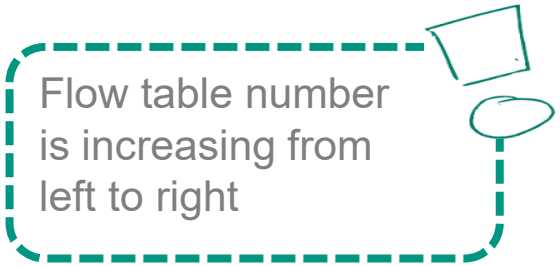
- First rule
  - `rule1.MATCH('IP_DST', '40.2.4.4')`
  - `rule1.ACTION('DROP')`
  - `rule1.PRIORITY(10)`
- Second rule
  - `rule2.MATCH('IP_DST', '40.0.0.0/8')`
  - `rule2.ACTION('OUTPUT', 1)`
  - `// rule2.PRIORITY(1)`

```
rule1.PRIORITY >
rule2.PRIORITY
```

Multiple Flow Tables

- SDN switches can support more than one flow table
  - Common in software switches
  - Expensive in hardware (TCAM limited by available space / energy)

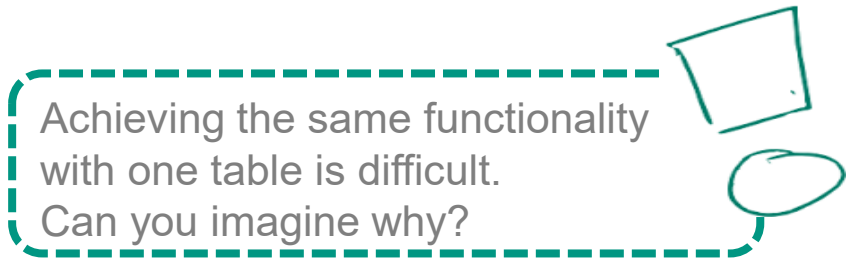




- Using multiple tables has several benefits
  - Can be used to isolate flow rules from different apps
  - Logical separation between different tasks (one table for monitoring, one table for security, …)
  - In some situation: less overall flow table entries

Multiple Flow Tables

- Working with multiple tables is very similar to single table case
  - `r.TABLE(x)` to specify the table for this rule
  - `r.ACTION('GOTO', y)` to specify processing continues in another table
- Can not go to lower flow table number to avoid cycles
  - `GOTO` from table x to table y → y > x
- Example
  - First table: tracking of end systems based on source address
    - `rule1.MATCH('IP_SRC', '5.4.3.2')`
    - `rule1.TABLE(1) // rule1 is applied to table 1`
    - `rule1.ACTION('GOTO', 2) // continue processing in table 2`
  - Second table: forwarding based on destination addresses
    - `rule2.MATCH('IP_DST', '40.27.13.144')`
    - `rule2.TABLE(2) // rule2 is applied to table 2`
    - `Rule2.ACTION('OUTPUT', 24)`



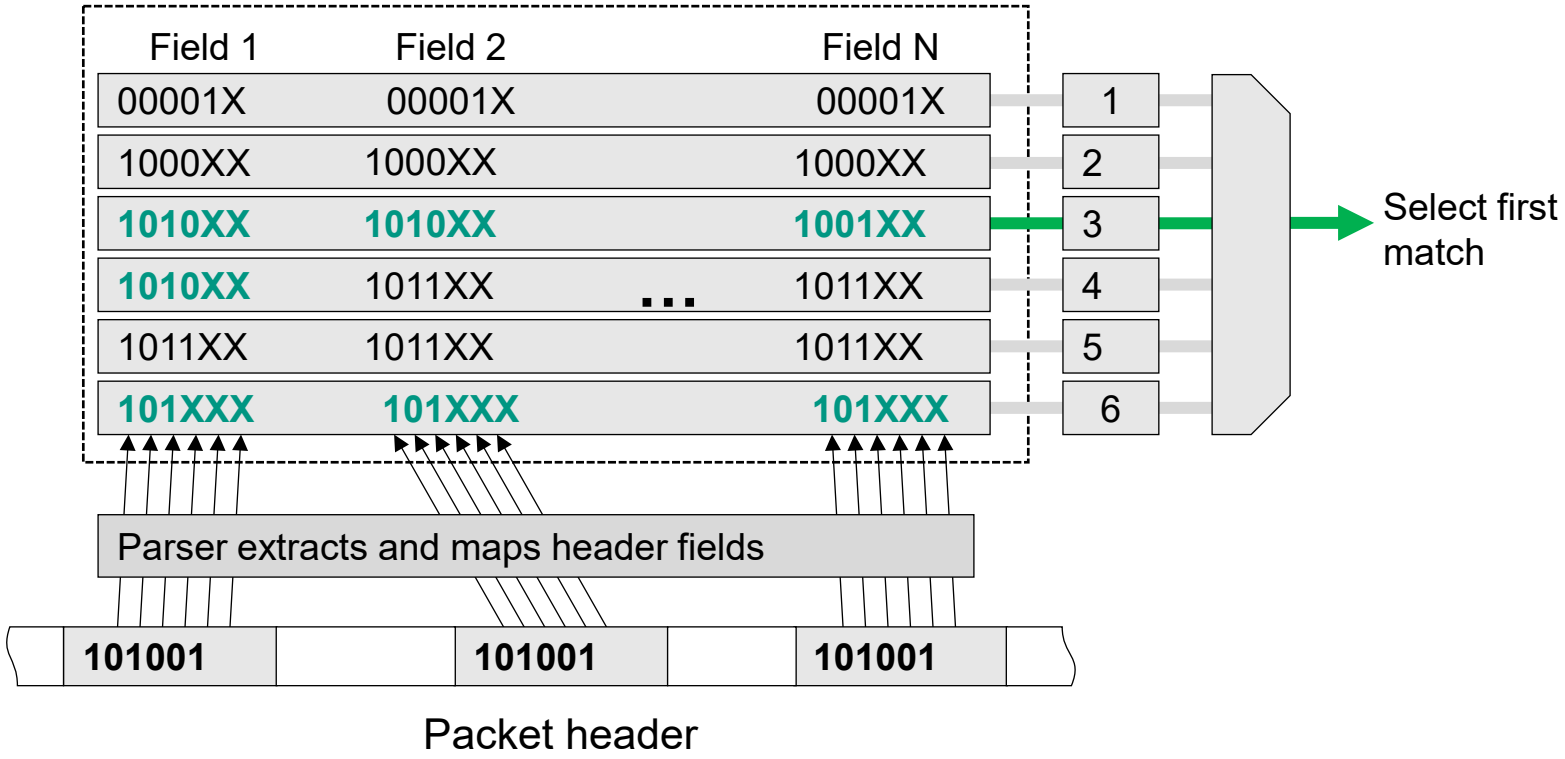
Storing Flow Table Rules in TCAM

- Different ways of structuring TCAM entries possible
  - Single Match Table (SMT)
    - Simple abstraction model
    - One single match table
    - Wasteful memory usage
  - Multiple Match Tables (MMT)
    - Pipeline processing
    - Multiple smaller fixed-size match tables
  - …

[BGK+13]

Single Match Table (SMT)

- Match any header fields against a single match table
  - Match table must store every combination of match fields
  - Requires cartesian product for dependent match values
    - Value of field 1 must be replicated when field 2, 3, etc. differ
  - Wasteful use of limited TCAM space

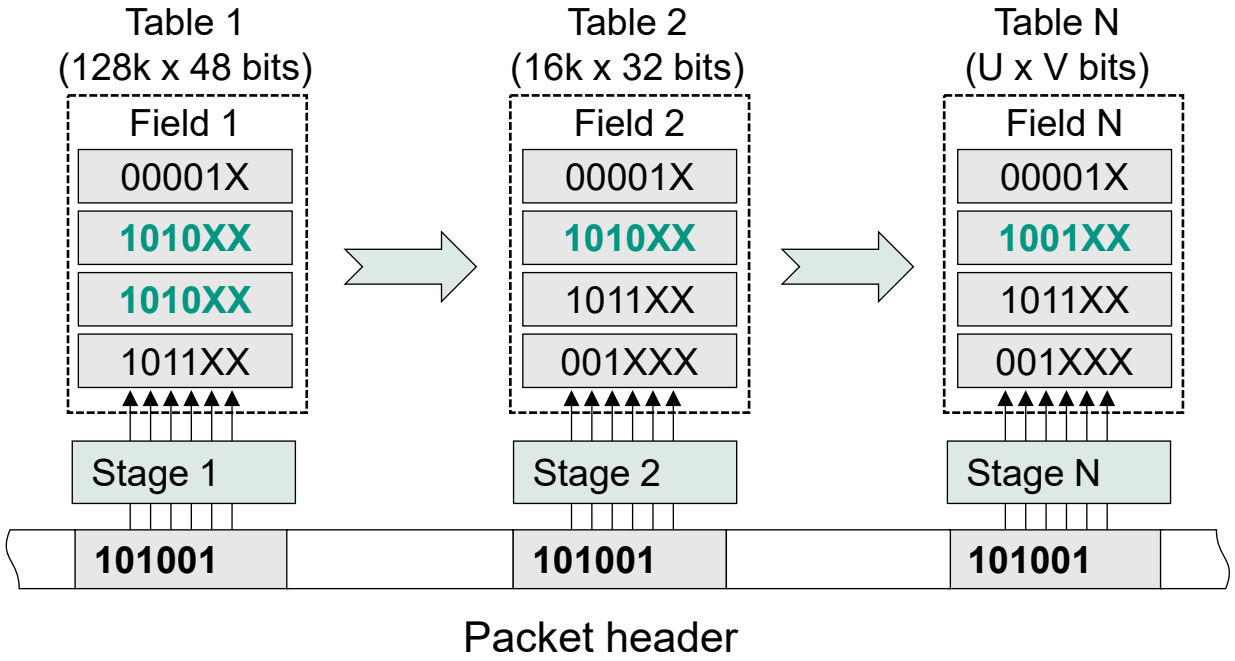


SMT is just a simple abstraction

[BGK+13]

Multiple Match Tables (MMT)

- Multiple smaller match tables matched by subset of header fields
  - Match tables arranged into a pipeline of stages
  - Processing of stage k depends on processing of stage $i < k$

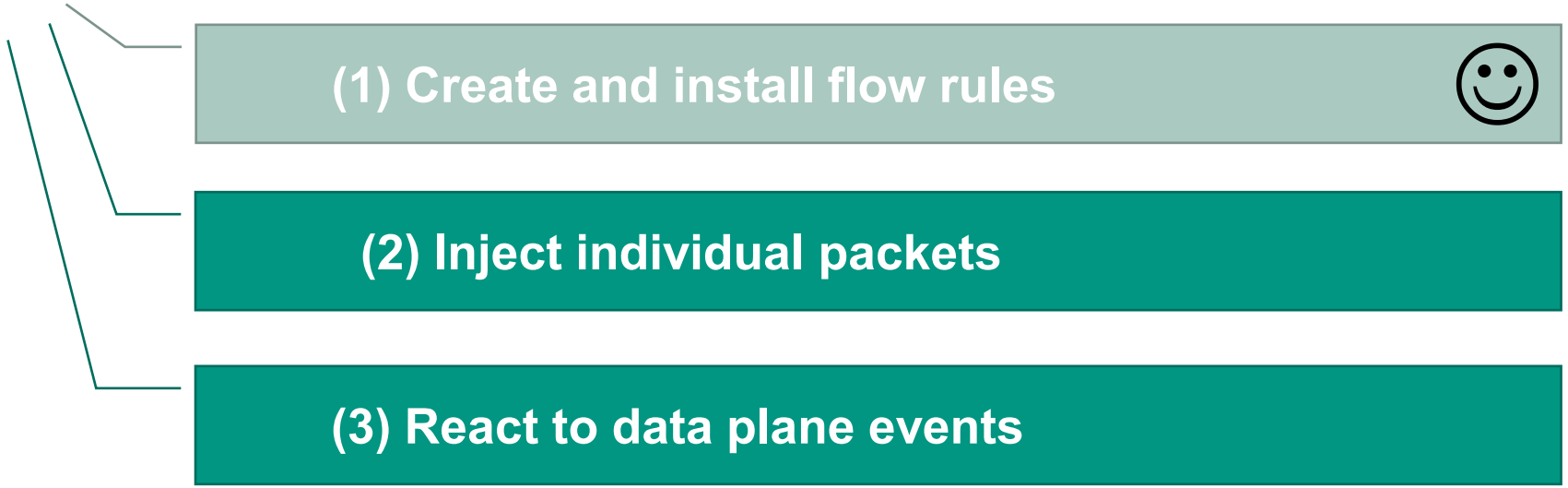


- Challenging to accommodate multiple use cases
  - Table width and depth are determined during chip fabrication
  - MAC or IP addresses more important (48 vs. 32 bit width)?

… for further information see [BGK+13]

Primitives for SDN Programming

- Overall goal?
  - From intended behavior to lower level flow rules
- This requires SDN programming primitives
- Three important areas to cover



Inject Individual Packets

- Missing: a way to handle individual packets from within the app
  - Forward a packet that was sent to the controller
    - Otherwise, this packet is lost
  - Perform topology detection
  - Active monitoring („probe packets“)
  - Answer ARP requests

Inject Individual Packets

- `send_packet(packet, switch, rule)`
  - Injects a single packet into a switch
  - Parameters
    - `packet`: contains the packet that should be injected, i.e., the complete packet including headers and payload
    - `switch`: the switch where the packet is injected (e.g., S1)
    - `rule`: a flow rule that is applied to this packet instead of default flow table processing (optional)
      - Only `rule.ACTION()` is allowed here
      - No matches, no priorities

- This is different from installing flow rules
  - It is used for a single packet only
  - The flow table is not changed
  - Even if the `rule` parameter is present, this does NOT create a new flow table entry

Two Ways to Process Injected Packets

- First option: Inject and process injected packet in the flow table
  - Example
    - `newPacket = createNewPacket()`
    - `send_packet(newPacket, switch)`
  - This option requires installing a flow rule prior to the injection
  - Would create a cycle otherwise

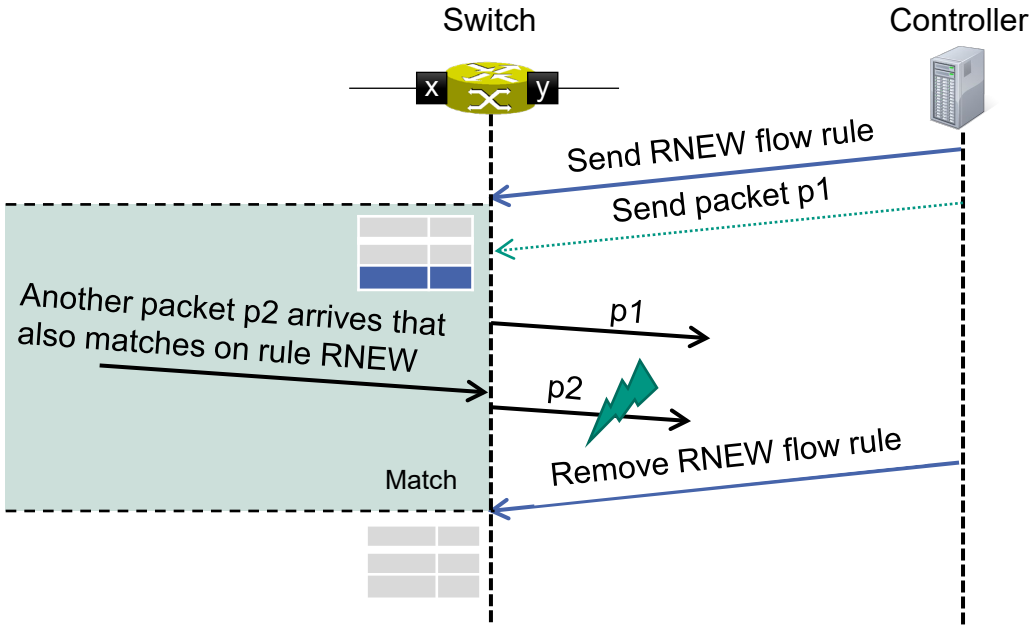
- Second option: Inject and process injected packet with a custom rule
  - Example
    - `newPacket = createNewPacket()`
    - `customRule = Rule()`
    - `customRule.ACTION('OUTPUT', 1)`
    - `send_packet(newPacket, switch, customRule)`
  - This option directly attaches the actions to the injected packet
  - Rule is only used for a single packet
  - Flow table remains unchanged

Why the Second Option?

- Efficiency
  - In certain cases, it can be too expensive to create a flow table entry, e.g., if only a few packets would ever be processed by the entry
  - In this case, it can be more efficient to perform the operations in software (more precisely: on the CPU in the switch)

Why the Second Option?

- Inconsistencies
  - Single packet p1 has to be processed differently from what is currently specified in the flow table, e.g., by a rule called “RNEW”
  - Without the rule option, this could be done as follows



  - Problem occurs if another packet arrives while network is in a inconsistent state (green area)
    - Packet p2 is handled by the wrong rule

ARP Example

- ARP is responsible for IP to MAC resolution
  - Usually involves ARP request and reply packets exchanged between network devices
  - Can be quite costly (broadcasting)

- With SDN, address resolution can be handled centrally
  - ARP application manages an ARP table in software
  - Can answer ARP requests directly, i.e., without contacting the end systems

- Requires that the application can generate ARP reply packets

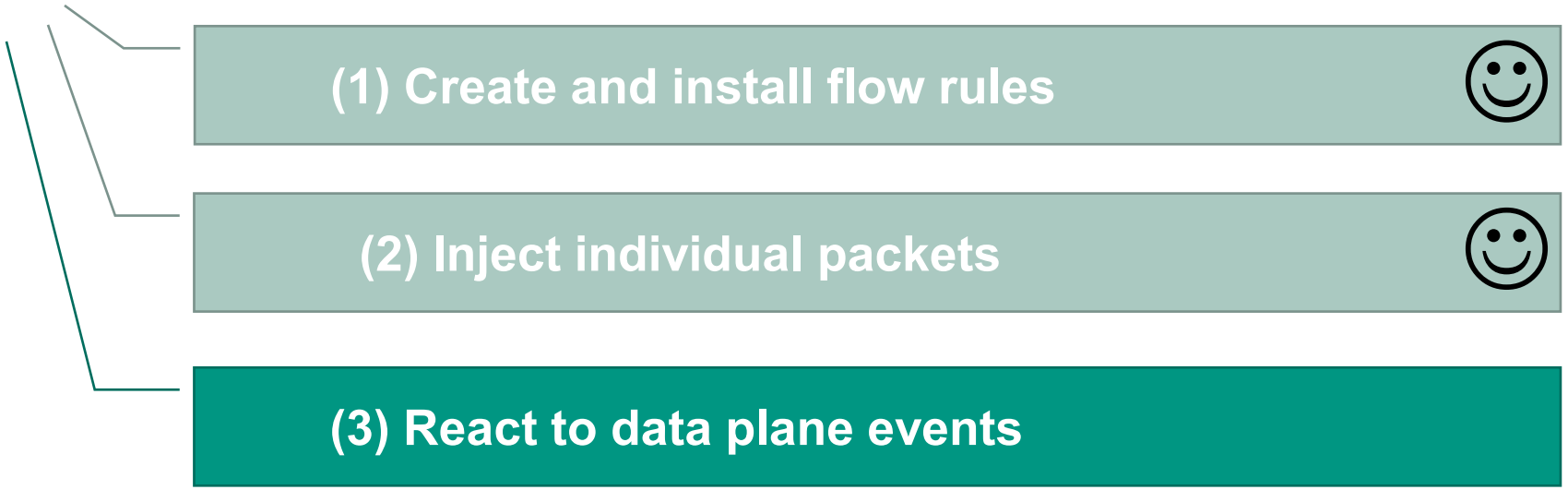
```
import ...
import isARPRequest, createARPReply

onPacketIn(packet, switch, inport) {
  if (isARPRequest(packet) == true) {
    newPacket = createARPReply(packet)
    send_packet(newPacket, switch)
  }
}
```

handle_arp.java

First injection option is applied, i.e., it is assumed that there is already a flow table entry to forward the newly created ARP packet based on its destination MAC address (for example)

Primitives for SDN Programming

- Overall goal?
  - From intended behavior to lower level flow rules
- This requires SDN programming primitives
- Three important areas to cover



React to Data Plane Events

- So far
  - Process of rule creation and installation independent from what happened in the data plane (actual network traffic)
  - Sufficient for proactive use cases
- Not sufficient for reactive use cases
  - Example: „Install a new flow rule only if two end systems A and B talk to each other“
  - Not known who will talk to whom and when this is going to happen
  - Difficult to achieve with a proactive approach
    - How could this be done?
- Required is a way to react to events in the data plane

React to Data Plane Events

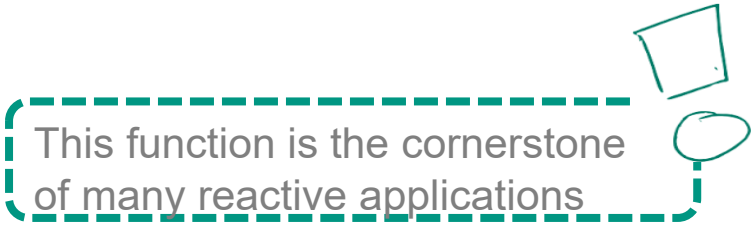
- Controller APIs provide callbacks to react to various control events
  - Switch state changes
  - Link state changes
  - Warnings and errors
  - …

- `onConnect(switch)`
  - Called if a new control connection to `switch` is established
  - Useful for proactive programming
  - Parameters
    - `switch`: a reference to the switch
    - We use simple IDs (e.g., 1 for S1 and 2 for S2) to identify switches via `switch.id`

React to Data Plane Events

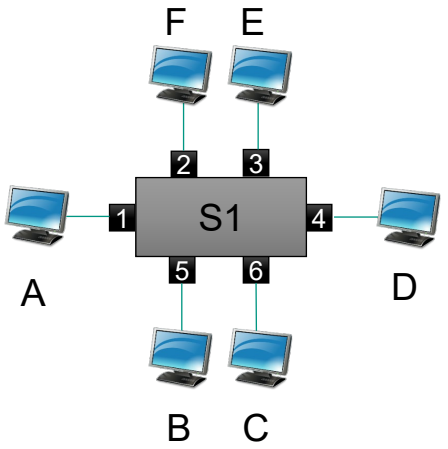
- Dealing with packets sent to the controller requires another callback

- **`onPacketIn(packet, switch, inport)`**
  - Called if the controller receives a packet that was forwarded via `r.ACTION('CONTROLLER')`
  - Parameters
    - `packet`: contains packet that was forwarded and grants access to its header fields
      - `packet.IP_SRC`
      - `packet.IP_DST`
      - `packet.MAC_SRC`
      - `packet.MAC_DST`
      - `packet.TTL`
      - `...`
    - `switch`: the switch the packet was received at (e.g., S1)
    - `inport`: the interface the packet was received at (e.g., port 1)



Example: Reactive Use Case

- Devices are identified by MAC source address (A, B, …)
- Association between port and end system is known (static)
  - A → Port 1
  - B → Port 5
  - …



- Intended behavior: Set up a separate flow for each communication pair (e.g., A talks to B) but ONLY if the end systems do actually talk to each other

- Sketch

  (1) Create a low priority flow rule that sends „all unknown packets“ to the controller
  - `r.MATCH('*') // match on everything`
  - `r.ACTION('CONTROLLER') // send packet to controller`
  - `r.PRIORITY(0) // use lowest priority for this flow rule`

  (2) Use `onPacketIn()` to create and install flow rules on demand

Example: Reactive Use Case

```
import ...
import getPort
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
}

onPacketIn(packet, switch, inport) {
  r1 = Rule()
  r1.MATCH('MAC_SRC', packet.MAC_SRC)
  r1.MATCH('MAC_DST', packet.MAC_DST)
  r1.ACTION('OUTPUT', getPort(packet.MAC_DST))
  send_rule(r1, switch)
  r2 = Rule()
  r2.MATCH('MAC_SRC', packet.MAC_DST)
  r2.MATCH('MAC_DST', packet.MAC_SRC)
  r2.ACTION('OUTPUT', getPort(packet.MAC_SRC))
  send_rule(r2, switch)
  send_packet(packet, switch)
}
```

The association between MAC addresses and ports is static → we can calculate the output port for a given MAC destination address with the `getPort` helper function imported here

Example: Reactive Use Case

```
import ...
import getPort
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
}

onPacketIn(packet, switch, inport) {
  r1 = Rule()
  r1.MATCH('MAC_SRC', packet.MAC_SRC)
  r1.MATCH('MAC_DST', packet.MAC_DST)
  r1.ACTION('OUTPUT', getPort(packet.MAC_DST))
  send_rule(r1, switch)
  r2 = Rule()
  r2.MATCH('MAC_SRC', packet.MAC_DST)
  r2.MATCH('MAC_DST', packet.MAC_SRC)
  r2.ACTION('OUTPUT', getPort(packet.MAC_SRC))
  send_rule(r2, switch)
  send_packet(packet, switch)
}
```

A low priority flow rule is created that will send every packet to the controller. Note that the priority has to be changed explicitly!

Example: Reactive Use Case

```
import ...
import getPort
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
}

onPacketIn(packet, switch, inport) {
  r1 = Rule()
  r1.MATCH('MAC_SRC', packet.MAC_SRC)
  r1.MATCH('MAC_DST', packet.MAC_DST)
  r1.ACTION('OUTPUT', getPort(packet.MAC_DST))
  send_rule(r1, switch)
  r2 = Rule()
  r2.MATCH('MAC_SRC', packet.MAC_DST)
  r2.MATCH('MAC_DST', packet.MAC_SRC)
  r2.ACTION('OUTPUT', getPort(packet.MAC_SRC))
  send_rule(r2, switch)
  send_packet(packet, switch)
}
```

A new flow rule is created and installed. It matches on MAC source and destination address and uses the `getPort` function to calculate the proper output port

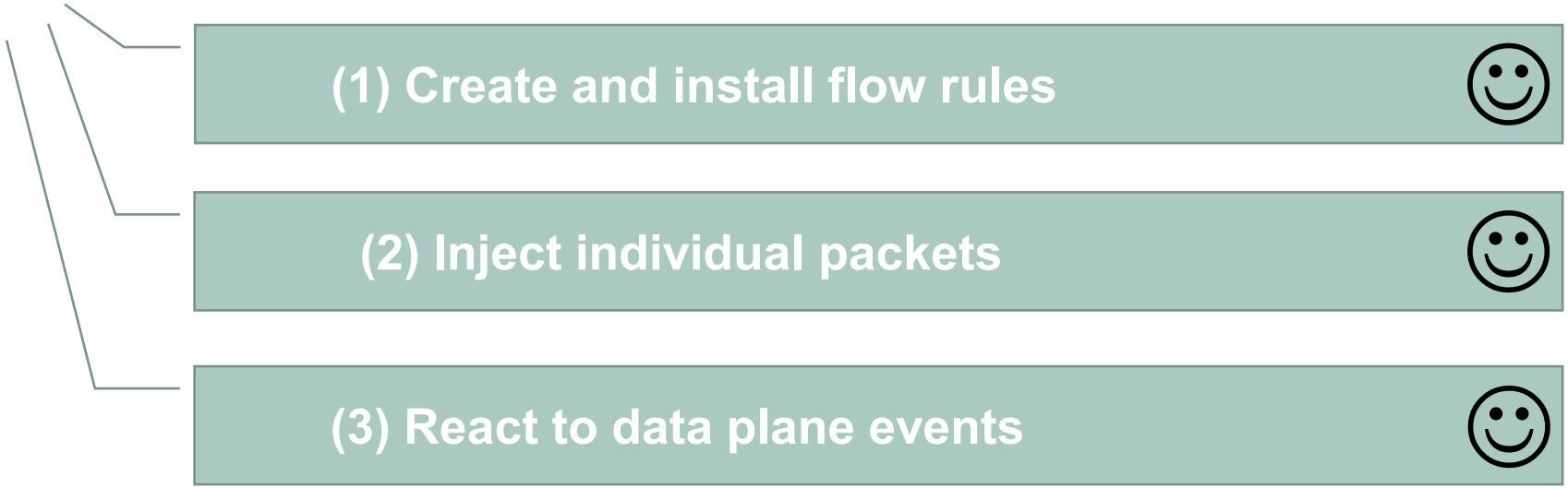
Example: Reactive Use Case

```
import ...
import getPort
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
}

onPacketIn(packet, switch, inport) {
  r1 = Rule()
  r1.MATCH('MAC_SRC', packet.MAC_SRC)
  r1.MATCH('MAC_DST', packet.MAC_DST)
  r1.ACTION('OUTPUT', getPort(packet.MAC_DST))
  send_rule(r1, switch)
  r2 = Rule()
  r2.MATCH('MAC_SRC', packet.MAC_DST)
  r2.MATCH('MAC_DST', packet.MAC_SRC)
  r2.ACTION('OUTPUT', getPort(packet.MAC_SRC))
  send_rule(r2, switch)
  send_packet(packet, switch)
}
```

Same as above, but in reverse direction (return flow). Cannot overlap with the other rule, so default priority can be used.

Primitives for SDN Programming

- Overall goal?
  - From intended behavior to lower level flow rules
- This requires SDN programming primitives
- Three important areas to cover



Summary on Primitives

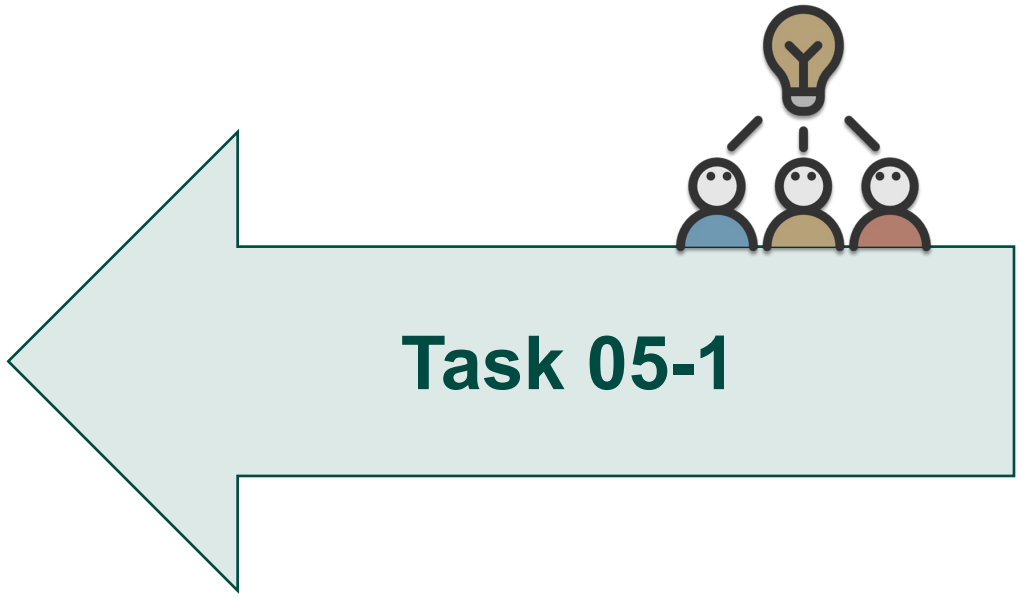
- Entry point primitves: Callbacks to implement custom logic
  - `onConnect(switch)`
    - Called if a new control connection to switch is established
  - `onPacketIn(packet, switch, port)`
    - Called if a packet was forwarded to the controller

- Flow rule creation primitives: Used to define flow rules
  - `Rule.MATCH()`
    - Select packets based on certain header fields
  - `Rule.ACTION()`
    - Specify what happens to a packet in the switch
  - `Rule.PRIORITY()`
    - Specify the priority of the created flow rule
  - `Rule.TABLE()`
    - Specify the flow table the rule should be applied to

Summary on Primitives

- Switch interaction primitives:
  Used to handle flow rule installation and packet injection
  - `send_rule(rule, switch)`
    - Installs a flow rule and creates the associated flow table entry in the switch
  - `send_packet(packet, switch)`
    - Injects a single packet into a switch, process with existing flow table entries
  - `send_packet(packet, switch, rule)`
    - Injects a single packet into a switch, process with custom rule


Task 05-1
Code example


Task 05-1

Homework

Homework 05-01

SDN

- Explain SDN App
- Sequence diagramm of packet processing

5.2.5 Learning Switch Example

Be Careful…

- The introduced primitives are abstract and simple
- The process of programming an SDN app, however, can still be challenging
  - Even for trivial applications there are pitfalls
  - An example for this is given on the next few slides (learning switch)

Self-Learning Switches

- Short recap 

- Goal: learn port-address association of end systems

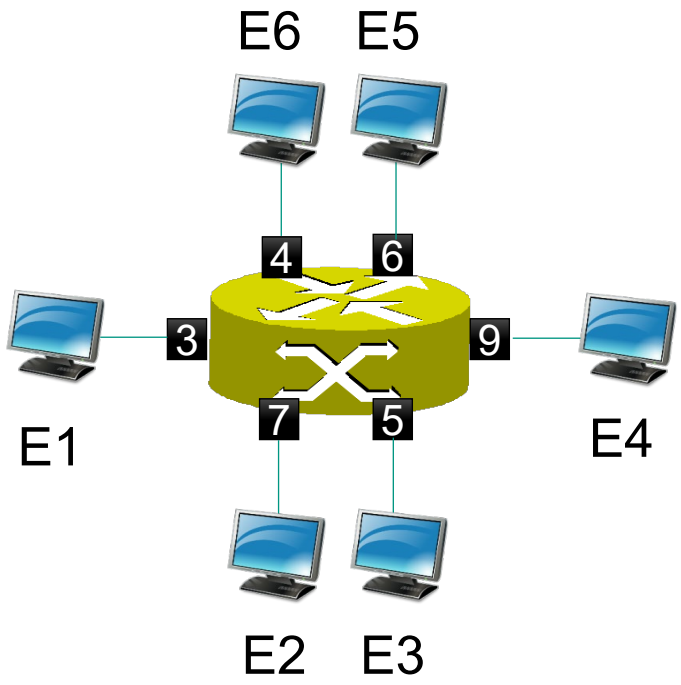
1) Switch receives packet and does not know destination address
   - Floods packets on all active ports
   - Learns "location" of the end system with this destination address
     - Remembers that end system is accessible via this port
     - Entry in table <MAC address, port, lifetime>

2) Switch receives packet and knows destination address
   - Forwards packet via corresponding port

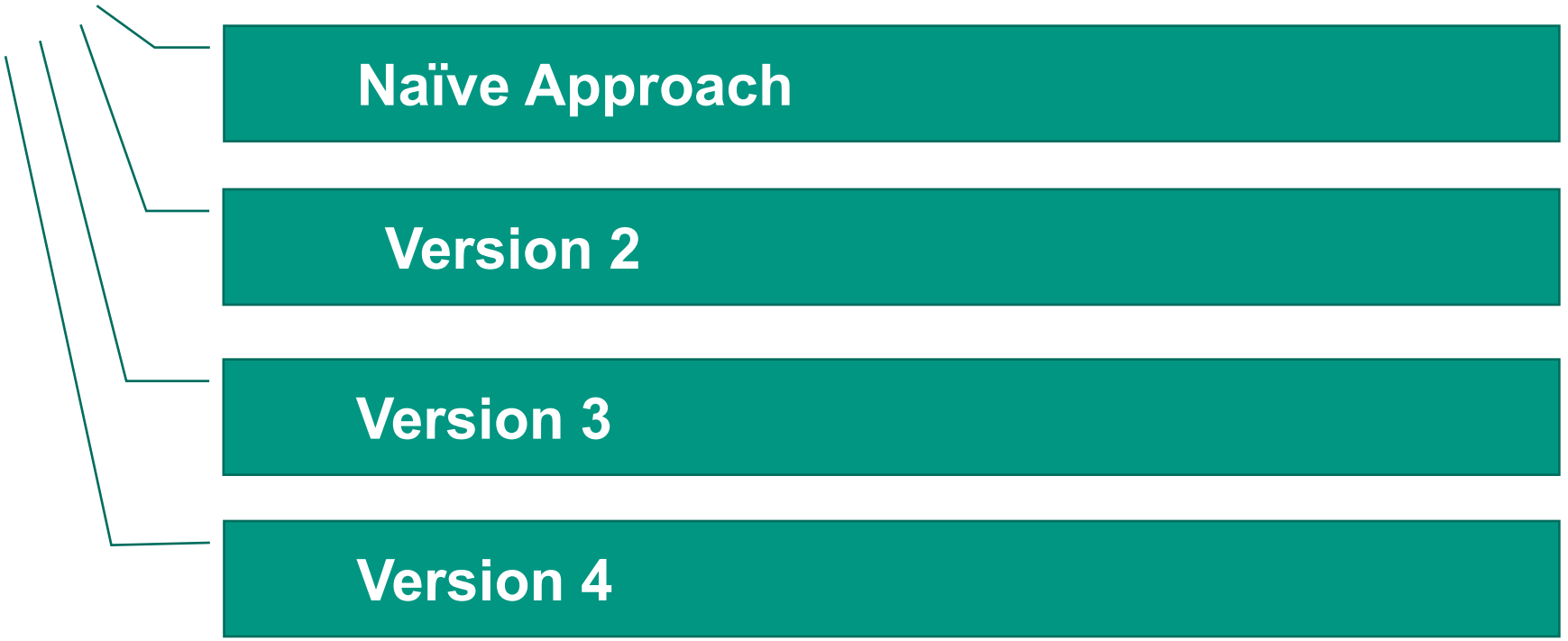
Self-Learning Switches

- We can do the same with SDN: Learning switch app
  - Observe packets by controller
  - Derive locations of end systems
  - Program forwarding rules to allow connectivity between end systems based on MAC addresses and port numbers

- Example topology
  - Learn that E1 is connected via port 3
  - Learn that E2 is connected via port 7
  - …



Learning Switch Example



Naïve Approach

- Send all packets to controller
- Controller looks at INPORT and source MAC address
- Controller creates rules based on these two pieces of information
- Packets with unknown destination addresses are flooded to all ports

Naïve Approach

```
import ...
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
}

onPacketIn(packet, switch, inport) {
  // new packet → no match in flow table
  // install rule for this packet
  r = Rule()
  r.MATCH('MAC_DST', packet.MAC_SRC)
  r.ACTION('OUTPUT', inport)
  send_rule(r, switch)
  // handle this packet → flood
  forwardFlood= Rule()
  forwardFlood.ACTION('FLOOD')
  send_packet(packet, switch, forwardFlood)

}
```

Create a flow table entry that will send all unmatched packets to controller

Naïve Approach

```
import ...
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
}

onPacketIn(packet, switch, inport) {
  // new packet → no match in flow table
  // install rule for this packet
  r = Rule()
  r.MATCH('MAC_DST', packet.MAC_SRC)
  r.ACTION('OUTPUT', inport)
  send_rule(r, switch)
  // handle this packet → flood
  forwardFlood= Rule()
  forwardFlood.ACTION('FLOOD')
  send_packet(packet, switch, forwardFlood)

}
```

Generate and install rule for every packet received via `onPacketIn`

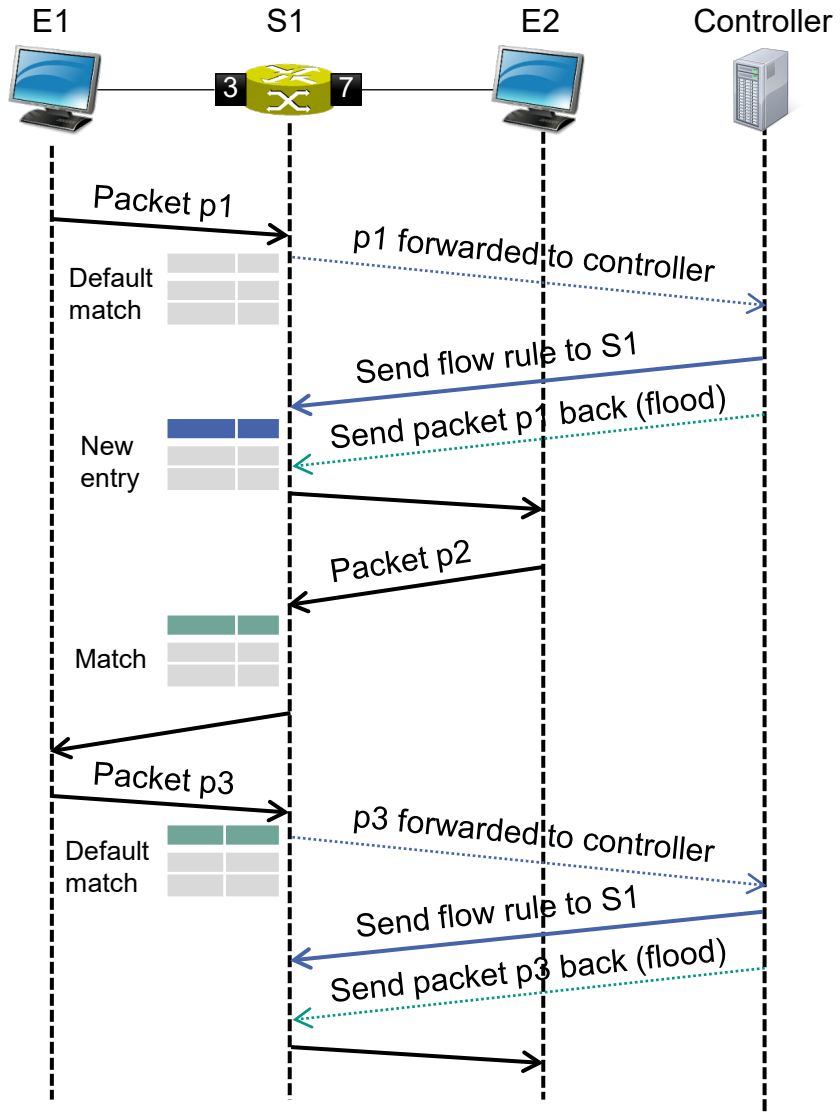
a. We only see packets that are not yet matched by a rule in the flow table
b. We use source address and inport to create the new rule
c. The MAC-to-port association is encoded in the flow table

Naïve Approach

```
import ...
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
}

onPacketIn(packet, switch, inport) {
  // new packet → no match in flow table
  // install rule for this packet
  r = Rule()
  r.MATCH('MAC_DST', packet.MAC_SRC)
  r.ACTION('OUTPUT', inport)
  send_rule(r, switch)
  // handle this packet → flood
  forwardFlood= Rule()
  forwardFlood.ACTION('FLOOD')
  send_packet(packet, switch, forwardFlood)

}
```

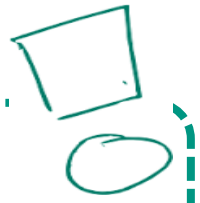
Reinject this packet into the network so it won‘t be lost

Problem



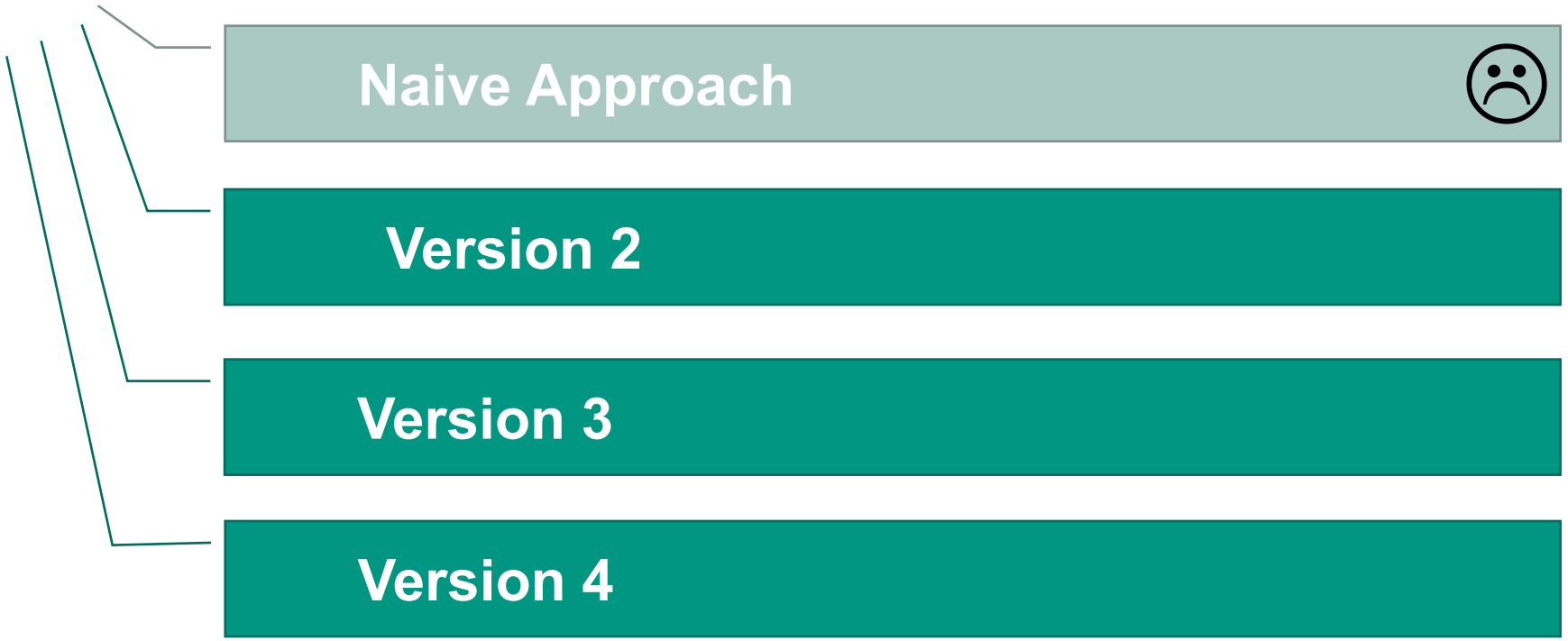
Flow table from S1 after adding new entry for p1

Match Fields	Action
MAC_DST = E1	Output 3
…	…
*	Controller



The controller has no chance to learn that E2 is located behind port 7

Learning Switch Example



Version 2

- Delay rule installation until the destination address was learned (not the source address)
- Avoids installing rules „too early“

Version 2

```
import ...
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
  MAC_TO_PORT[switch] = [] // learn MAC→PORT
}

onPacketIn(packet, switch, inport) {
  MAC_TO_PORT[switch][packet.MAC_SRC] = inport
  out = MAC_TO_PORT[switch][packet.MAC_DST]
  if (out != undefined) {
    // destination port is known, install rule
    r = Rule()
    r.MATCH('MAC_DST', packet.MAC_DST)
    r.ACTION('OUTPUT', out)
    send_rule(r, switch)
    // handle this packet
    forwardSingle = Rule()
    forwardSingle.ACTION('OUTPUT', out)
    send_packet(packet, switch, forwardSingle)
  } else {
    // destination port unknown, flood packet
    customRule = Rule()
    customRule.ACTION('FLOOD')
    send_packet(packet, switch, customRule)
  }
}
```

Use a table in the controller to store the MAC-to-port association

Version 2

```
import ...
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
  MAC_TO_PORT[switch] = [] // learn MAC→PORT
}

onPacketIn(packet, switch, inport) {
  MAC_TO_PORT[switch][packet.MAC_SRC] = inport
  out = MAC_TO_PORT[switch][packet.MAC_DST]
  if (out != undefined) {
    // destination port is known, install rule
    r = Rule()
    r.MATCH('MAC_DST', packet.MAC_DST)
    r.ACTION('OUTPUT', out)
    send_rule(r, switch)
    // handle this packet
    forwardSingle = Rule()
    forwardSingle.ACTION('OUTPUT', out)
    send_packet(packet, switch, forwardSingle)
  } else {
    // destination port unknown, flood packet
    customRule = Rule()
    customRule.ACTION('FLOOD')
    send_packet(packet, switch, customRule)
  }
}
```

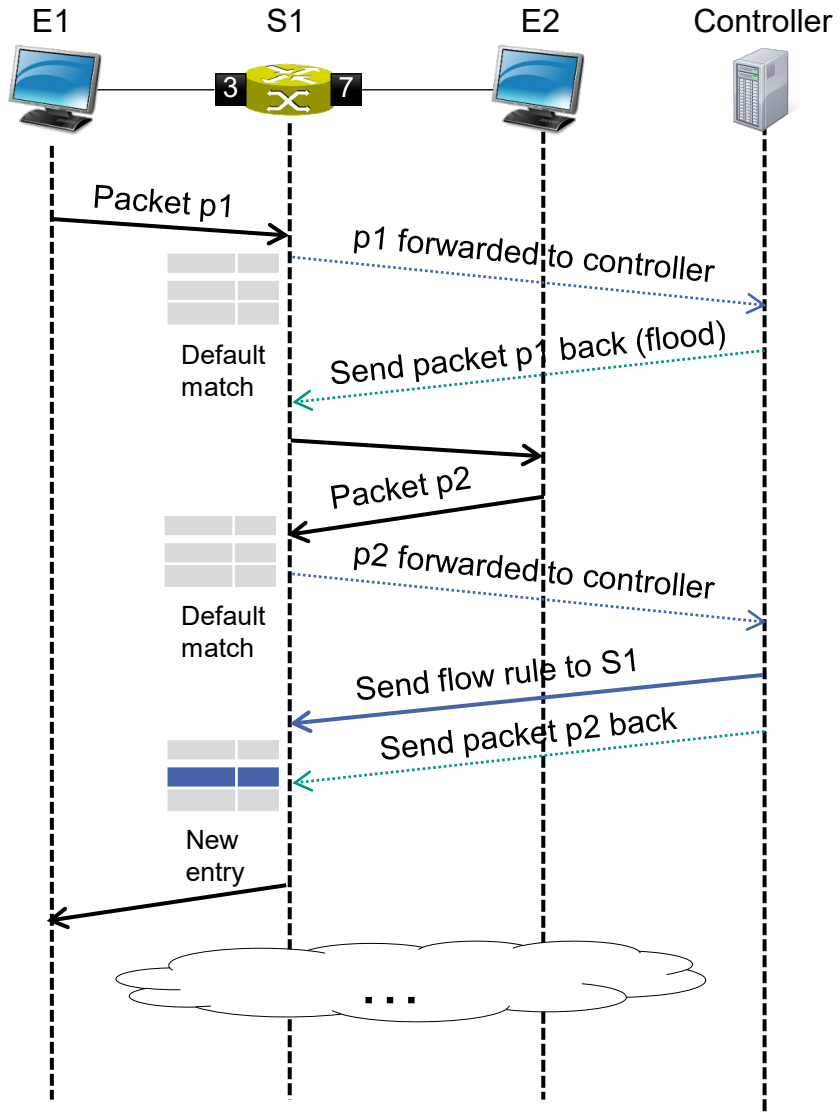
Instead of installing a rule directly (i.e., based on MAC Source), the rule is only installed if the destination MAC is known

Version 2

```
import ...
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
  MAC_TO_PORT[switch] = [] // learn MAC→PORT
}

onPacketIn(packet, switch, inport) {
  MAC_TO_PORT[switch][packet.MAC_SRC] = inport
  out = MAC_TO_PORT[switch][packet.MAC_DST]
  if (out != undefined) {
    // destination port is known, install rule
    r = Rule()
    r.MATCH('MAC_DST', packet.MAC_DST)
    r.ACTION('OUTPUT', out)
    send_rule(r, switch)
    // handle this packet
    forwardSingle = Rule()
    forwardSingle.ACTION('OUTPUT', out)
    send_packet(packet, switch, forwardSingle)
  } else {
    // destination port unknown, flood packet
    customRule = Rule()
    customRule.ACTION('FLOOD')
    send_packet(packet, switch, customRule)
  }
}
```

For known destination addresses, we can now avoid flooding

Version 2

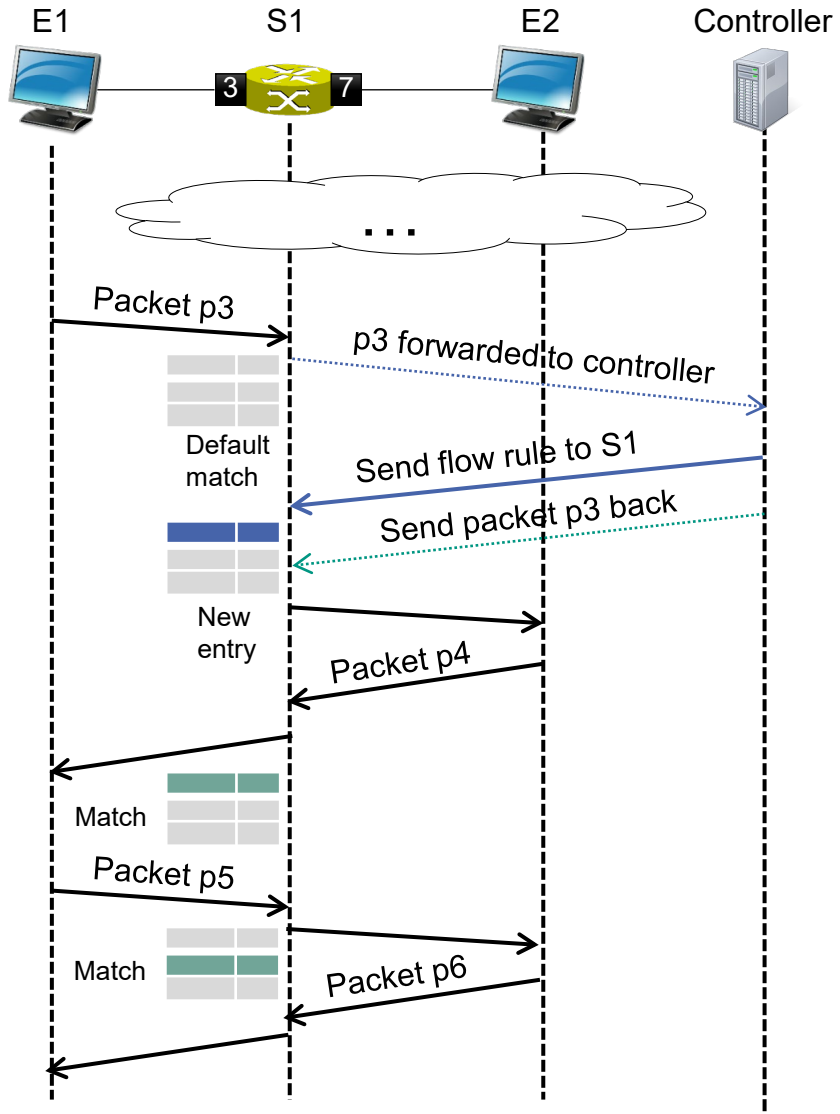


Flow table from S1 after adding new entry for p2

Match Fields	Action
MAC_DST = E1	Output 3
*	Controller

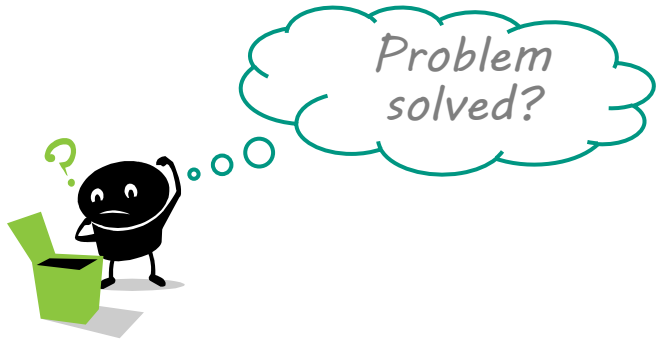
continued on next slide

Version 2

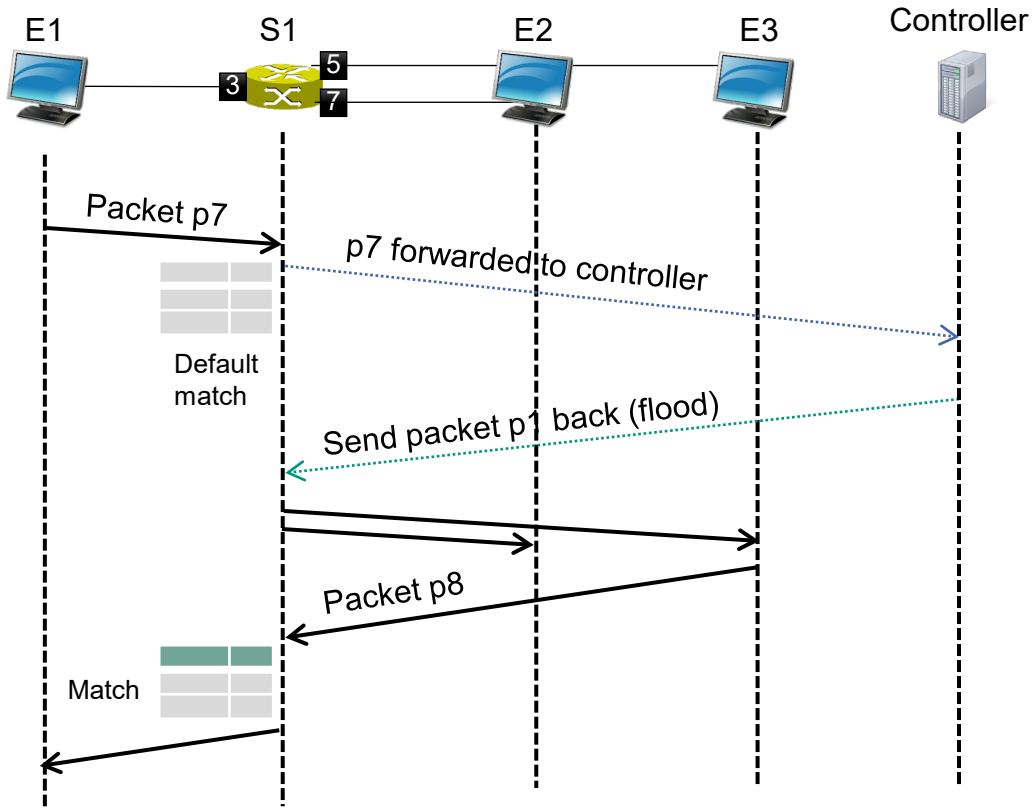


Flow table from S1 after adding new entry for p3

Match Fields	Action
MAC_DST = E1	Output 3
MAC_DST = E2	Output 7
*	Controller



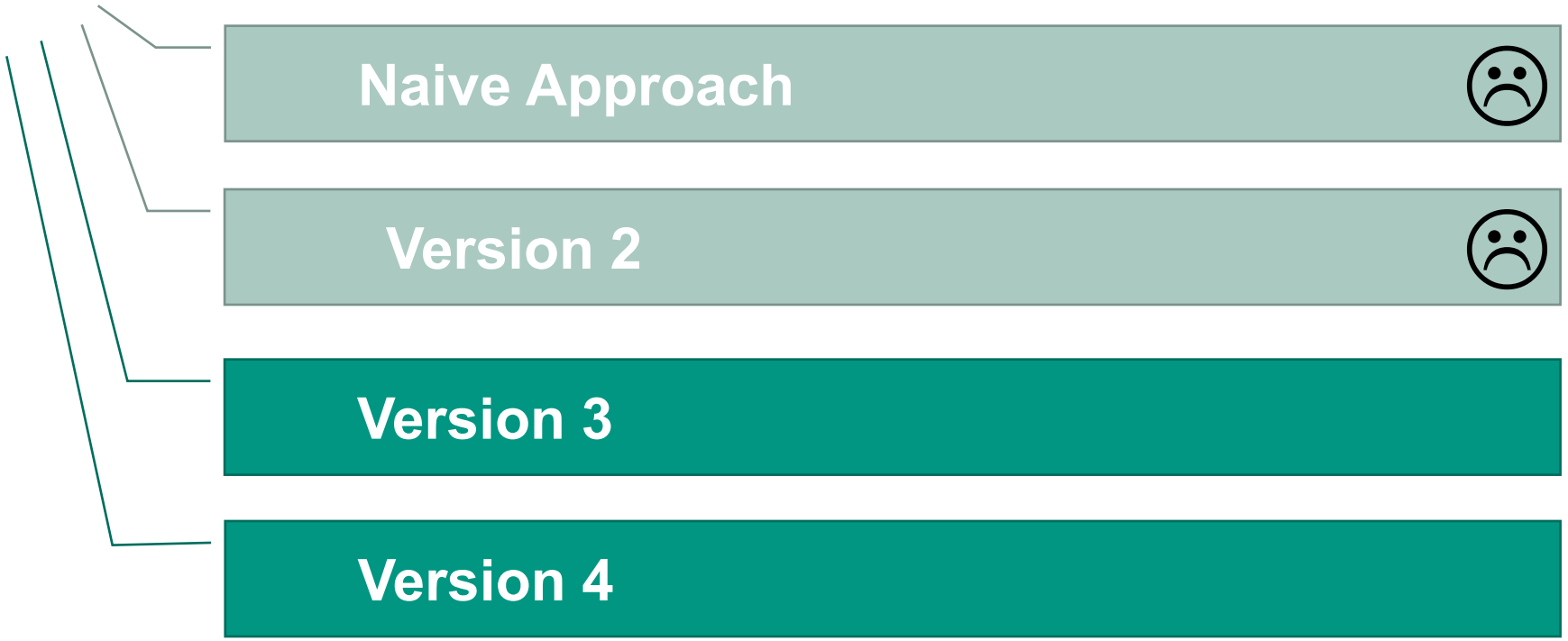
Problem



Match Fields	Action
MAC_DST = E1	Output 3
MAC_DST = E2	Output 7
*	Controller

1. E1 sends packet to E3 (p7).
2. E3 was not yet learned, packet goes to the controller
3. Controller has to flood the packet (E3 unknown)
4. Answer packet from E3 is forwarded directly (matched by first flow table entry)

Learning Switch Example



Version 3

- Only matching on destination address is not specific enough
- Use more specific matches
- Makes sure that all end systems can be learned by controller

Version 3

```
import ...
onConnect(switch) {
  r = Rule()
  r.MATCH('*')
  r.ACTION('CONTROLLER')
  r.PRIORITY(0) // lowest priority
  send_rule(r, switch)
  MAC_TO_PORT[switch] = [] // learn MAC→PORT
}

onPacketIn(packet, switch, inport) {
  MAC_TO_PORT[switch][packet.MAC_SRC] = inport
  out = MAC_TO_PORT[switch][packet.MAC_DST]
  if (out != undefined) {
    // destination port is known, install rule
    r = Rule()
    r.MATCH('MAC_SRC', packet.MAC_SRC)
    r.MATCH('MAC_DST', packet.MAC_DST)
    r.ACTION('OUTPUT', out)
    send_rule(r, switch)
    // handle this packet (skipped here)
  } else {
    // destination port unknown, flood packet
    customRule = Rule()
    customRule.ACTION('FLOOD')
    send_packet(packet, switch, customRule)
  }
}
```

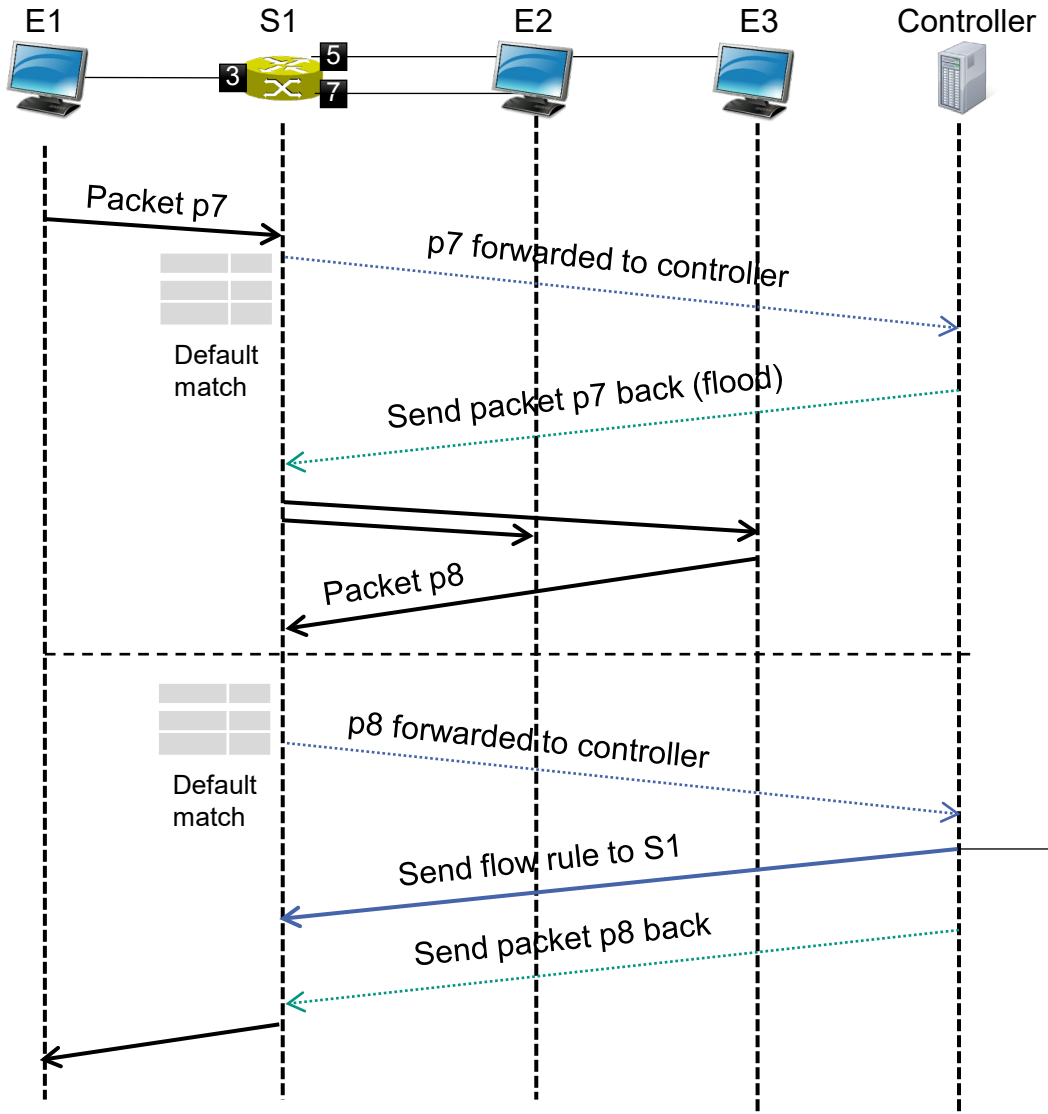
Only part that is modified here is the way how the match is created:

MATCH: source MAC address

MATCH: destination MAC address

ACTION: forward on known port

Version 3



Flow table from S1 after adding new entry for p8

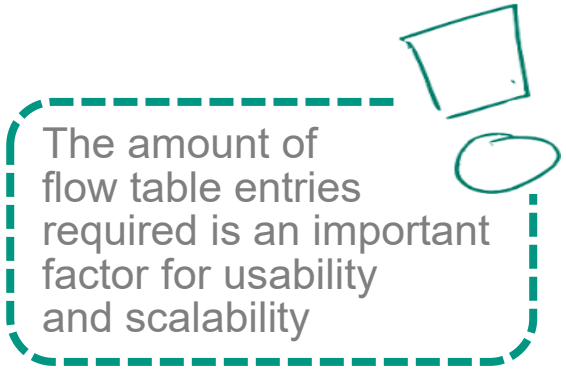
Match Fields	Action
MAC_DST = E1 MAC_SRC = E2	Output 3
MAC_DST = E2 MAC_SRC = E1	Output 7
MAC_DST = E1 MAC_SRC = E3	Output 3
*	Controller

```
MATCH: MAC_SRC = E3
MATCH: MAC_DST = E1
ACTION: forward on port 3
```

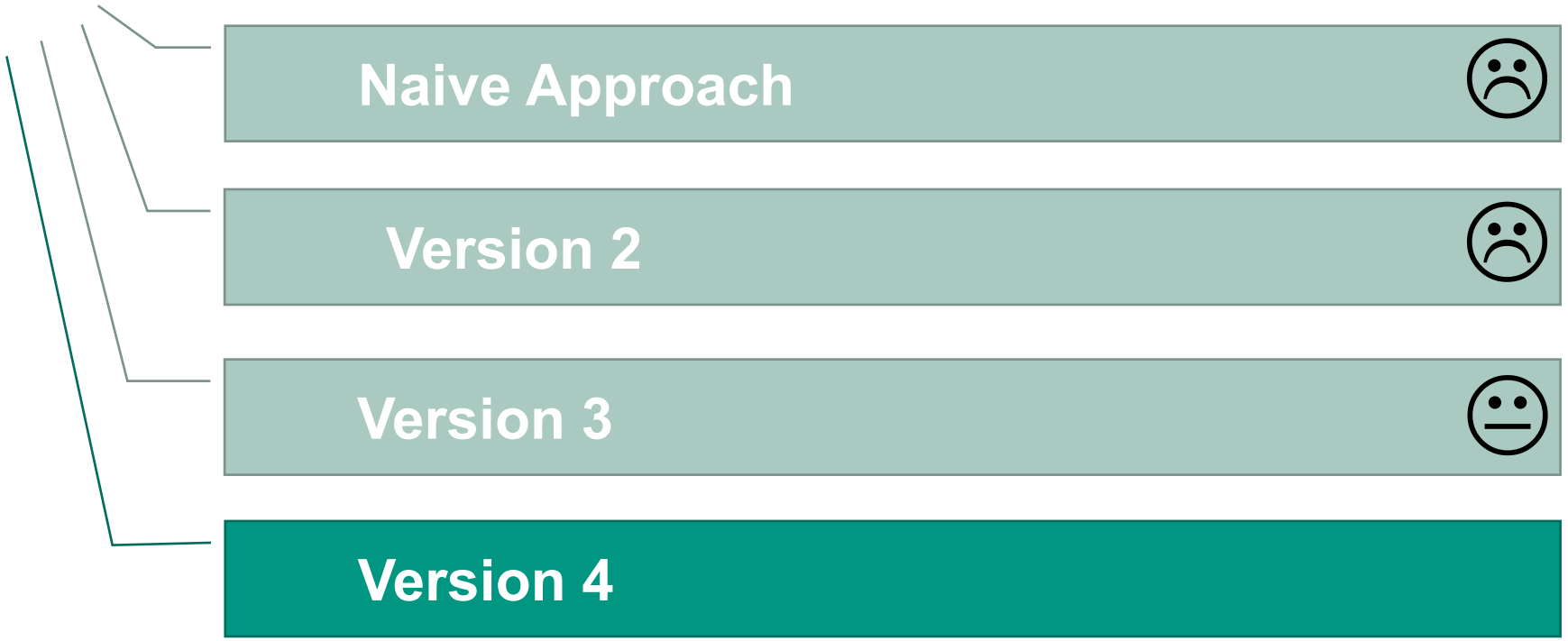
Problem

- Works fine from a functional perspective

- But what about flow table resources?
  - Needs N*N flow entries for N end systems
  - May exceed table capacity

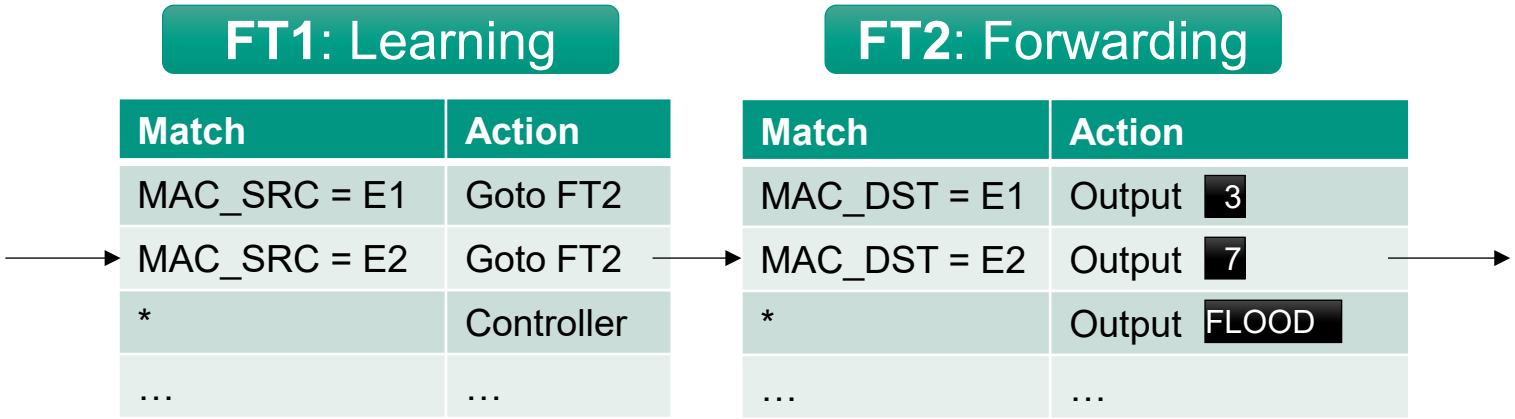


Learning Switch Example



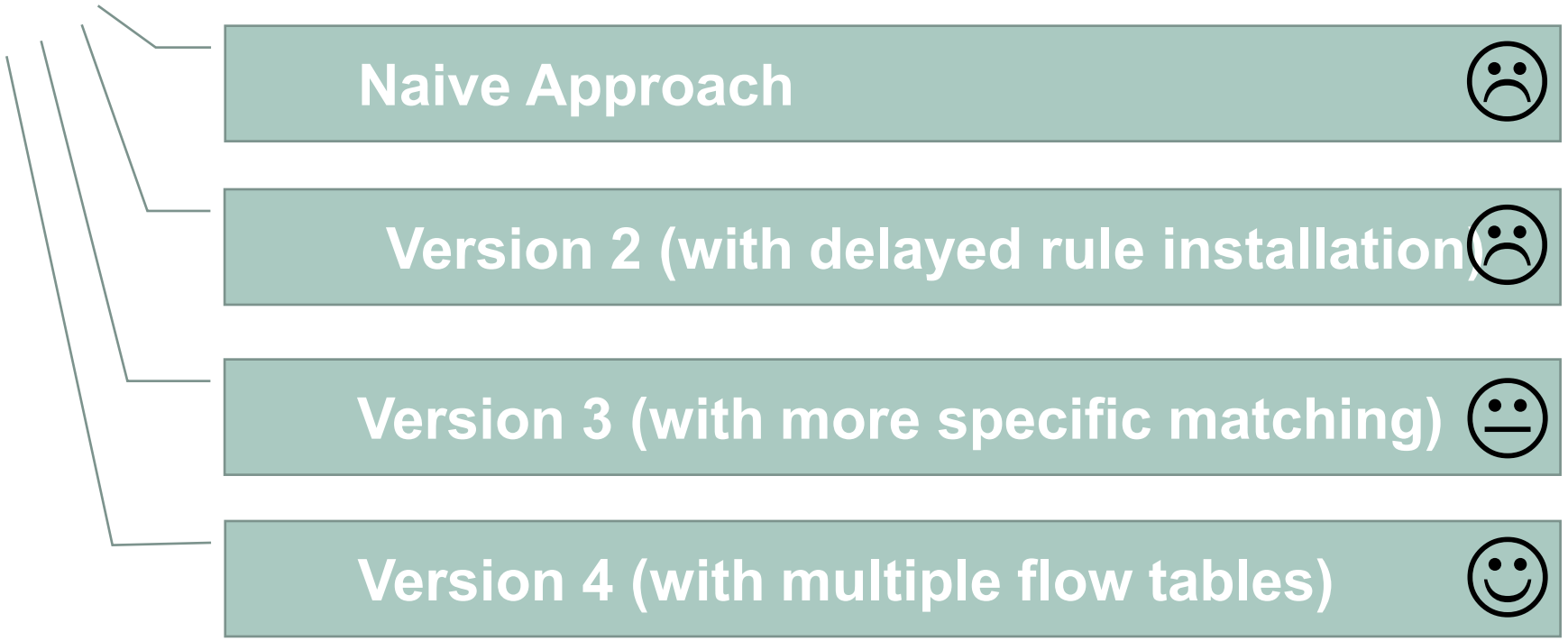
Version 4

- Separate flow tables for learning and forwarding
  - Flow table FT1 matches on source address and forwards to controller, if address was not yet learned
  - Flow table FT2 matches on destination address and forwards packet to destination (if learned) or floods packet (if not learned)
- Only 2*N rules for N end systems
- Problem: Hardware often does not support multiple flow tables due to cost, energy or space constraints



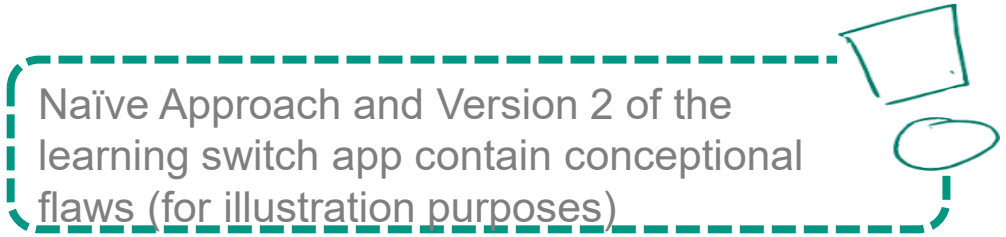
2 flow tables in one switch

Learning Switch Example



Summary

	Problem	Solution
Version 1 (naïve approach)	Flow rule for E1 was created too early so that the reply packet from E2 was not sent to the controller → E2 never learned	(Version 2) Delay flow rule creation until destination address can be resolved by adding a MAC-to-port table in controller
Version 2	Only the destination address was used for matching → does not work for multiple flows (e.g., E1+E2 and E1+E3)	(Version 3) Use more specific matches, e.g., source and destination address together to create separate entries for every flow
Version 3	Works, but requires a high amount of flow table entries (N*N for N end systems)	(Version 4) Use two flow tables, one for learning and one for forwarding

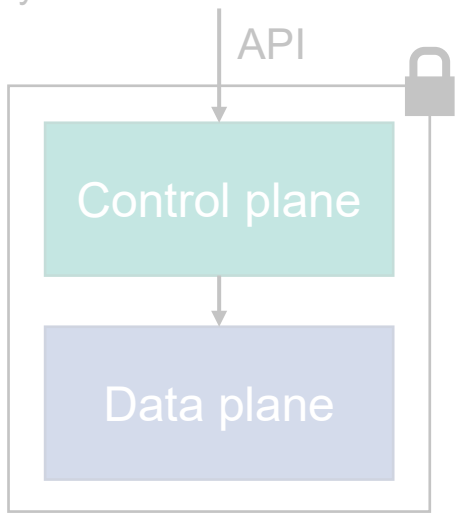


5.2.6 OpenFlow

Levels of Programmability

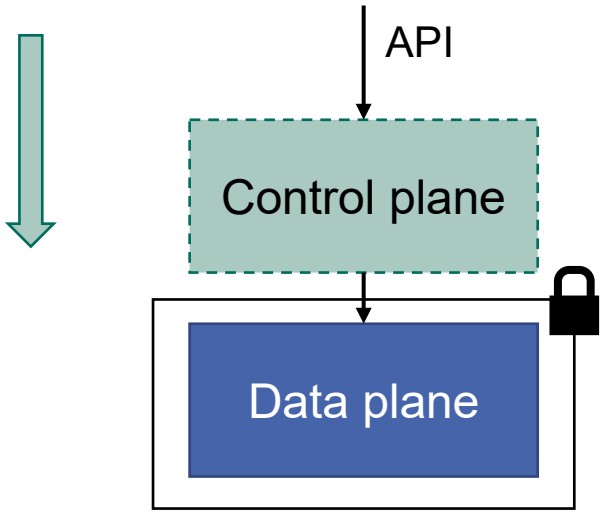
Traditional IP networking

programmability



- API provided by vendor allows some configurability (e.g., routing protocol)
- Algorithms in control and data plane cannot be changed

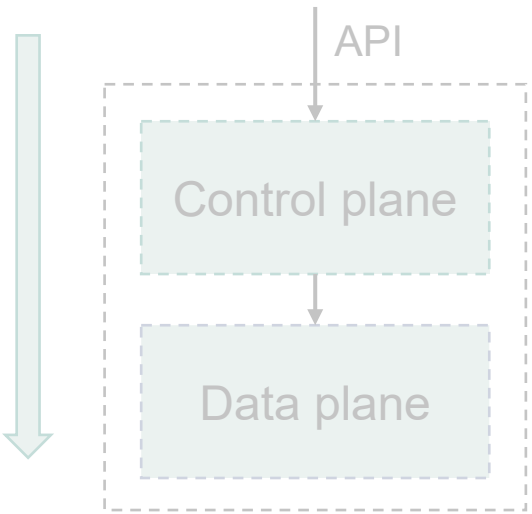
SDN with fixed-function data plane



- Algorithms in control plane can be changed
- Forwarding table in data plane can be programmed through specific interface

Open Flow

SDN with data plane programmability



- Algorithms in control and data plane can be changed

P4

OpenFlow

- A standard for an SDN southbound interface
  - Defines the interaction between controller and switches
  - Defines a logical architecture for SDN switches (flow table, …)
  - Defined by the Open Networking Foundation (ONF)

- Supports all basic structures and primitives seen so far
  - Matches
  - Actions
  - Priorities
  - Multiple flow tables
  - Protocol mechanisms for
    - Creating flow rules
    - Reacting to data plane events
    - Injecting individual packets

Concepts discussed in previous section;
OpenFlow = One possible implementation of these concepts

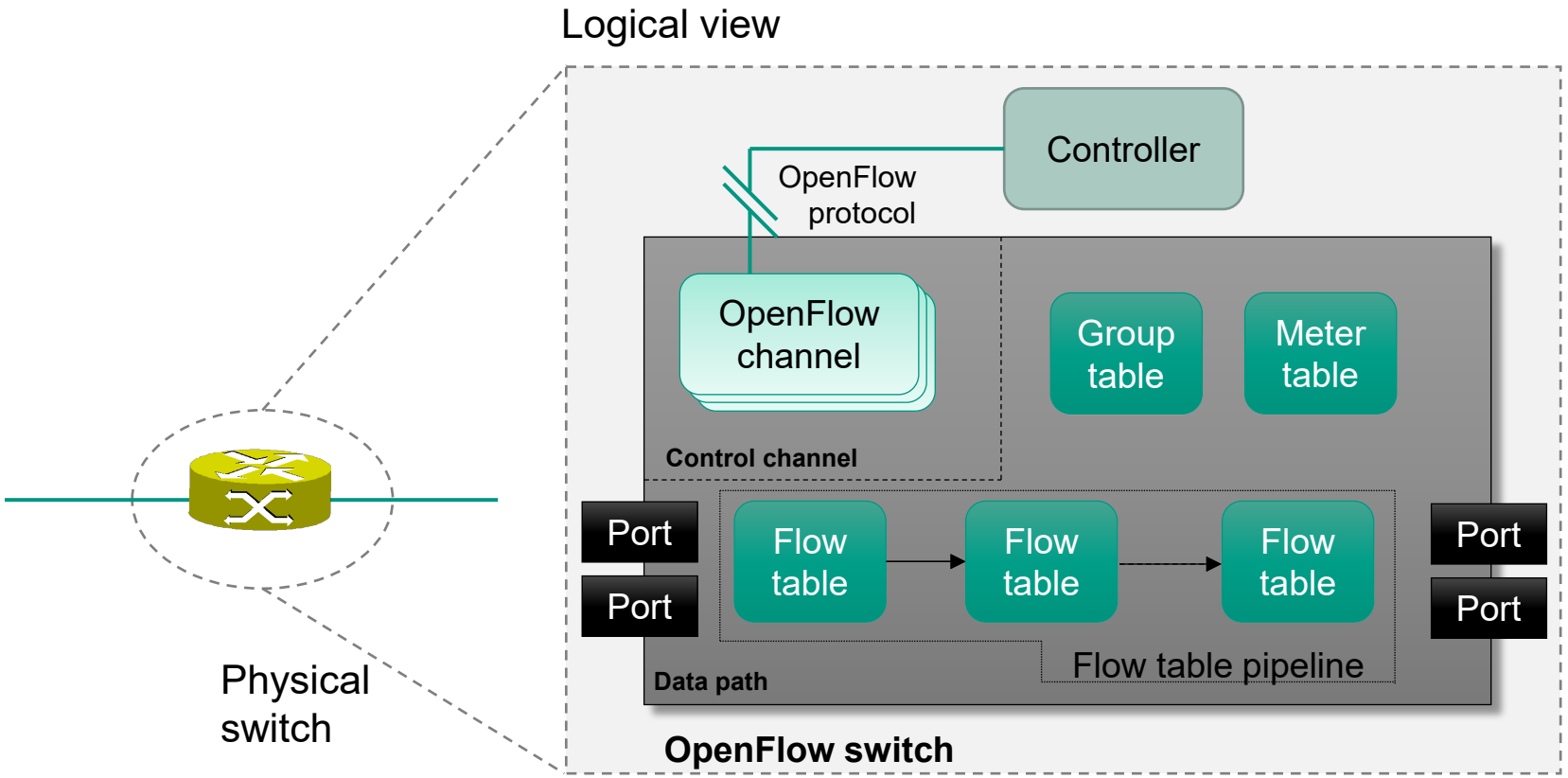
- And more sophisticated features
  - Group table, rate limiting

OpenFlow

OPEN NETWORKING FOUNDATION

OpenFlow Switch Architecture

- Provides a uniform view on SDN-capable switches



Ports

- Represent logical forwarding targets
- Can be selected by the output action
- Physical ports correspond to hardware interfaces

- Reserved ports (special meaning)
  - ALL: Represents all ports eligible to forward a specific packet (= flooding); ingress port is automatically excluded from forwarding
  - IN_PORT: Always references ingress port of a packet (= send packet back the way it came)
  - CONTROLLER: Forwarding a packet on this port sends it to the controller
  - NORMAL: Yields control of the forwarding process to the vendor-specific switch implementation

- Logical ports
  - Provide abstract forwarding targets (vendor-specific)
  - Link aggregation: Multiple interfaces are combined to a single logical port
  - Transparent tunneling: Traffic is forwarded via intermediate switches

Flow Table in OpenFlow

- Next to match (+ priority) and action, there are several other important fields in an OpenFlow flow table

Match fields	Priority	Actions	Counters	Timeouts	Cookie	Flags

- Counters
  - The number of processed packets (counter)
- Timeouts
  - Maximum lifetime of a flow
  - Enables automatic removal of flows
- Cookie
  - Marker value set by an SDN controller
  - Not used during packet processing
  - Simplifies flow management
- Flags
  - Indicate how a flow is managed
  - E.g., notify controller when a flow is automatically removed

Matches in OpenFlow

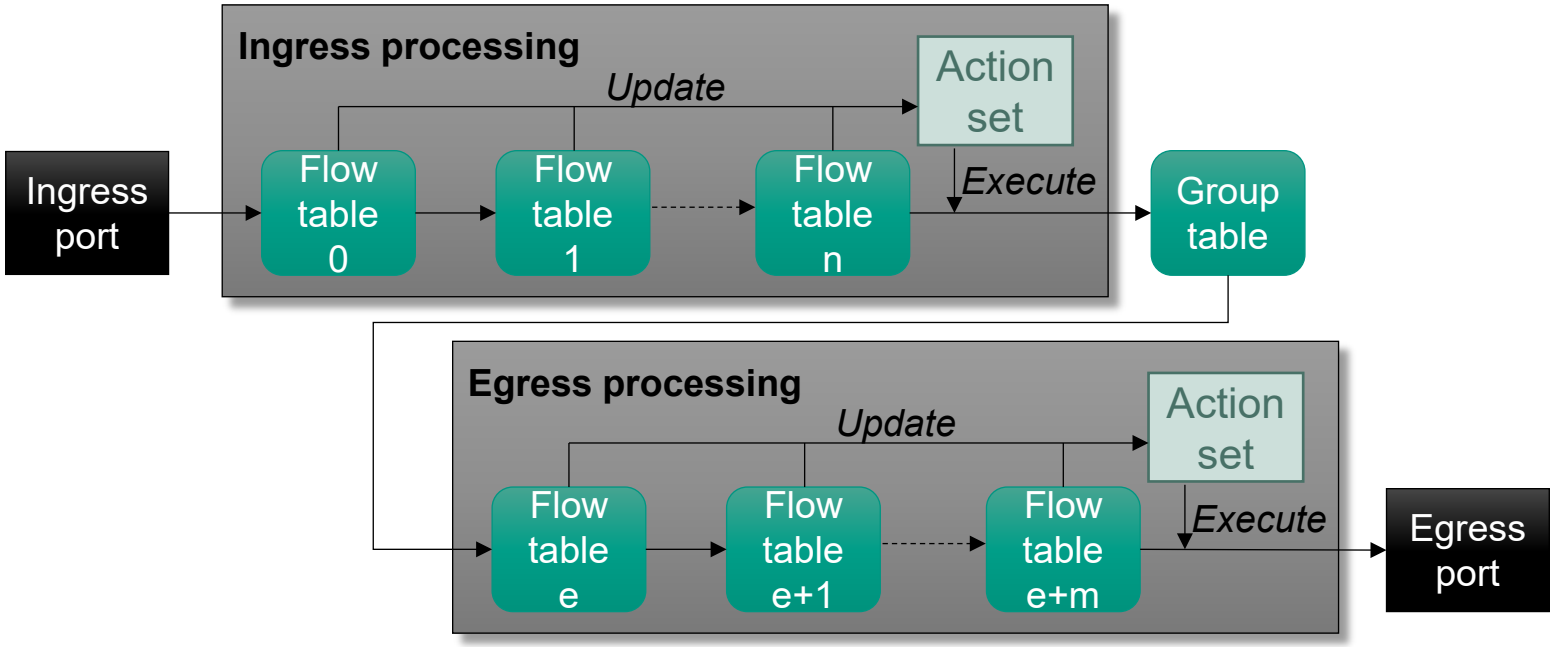
- OpenFlow 1.5.1 supports matching on 44 different fields
  - Only 14 of these are required by the specification

Match fields		
Ingress port	IP DSCP	ARP opcode
Physical ingress port	IP ECN	ARP IPv4 source address
Output port	**IP protocol**	ARP IPv4 target address
Metadata	**IPv4 source address**	ARP source hardware address
Packet type	**IPv4 destination address**	ARP destination hardware address
Ethernet destination address	**IPv6 source address**	SCTP source port
Ethernet source address	**IPv6 destination address**	SCTP destination port
Ethernet frame type	IPv6 flow label	ICMP type
VLAN id	ICMPv6 type	ICMP code
VLAN priority code point	ICMPv6 code	PBB service identifier
MPLS label	IPv6 ND target address	PBB UCA
MPLS traffic class	IPv6 ND source link-layer	**TCP source port**
MPLS bottom of stack	IPv6 ND target link-layer	**TCP destination port**
UDP source port	IPv6 extension header	TCP flags
UDP destination port	Logical port metadata	

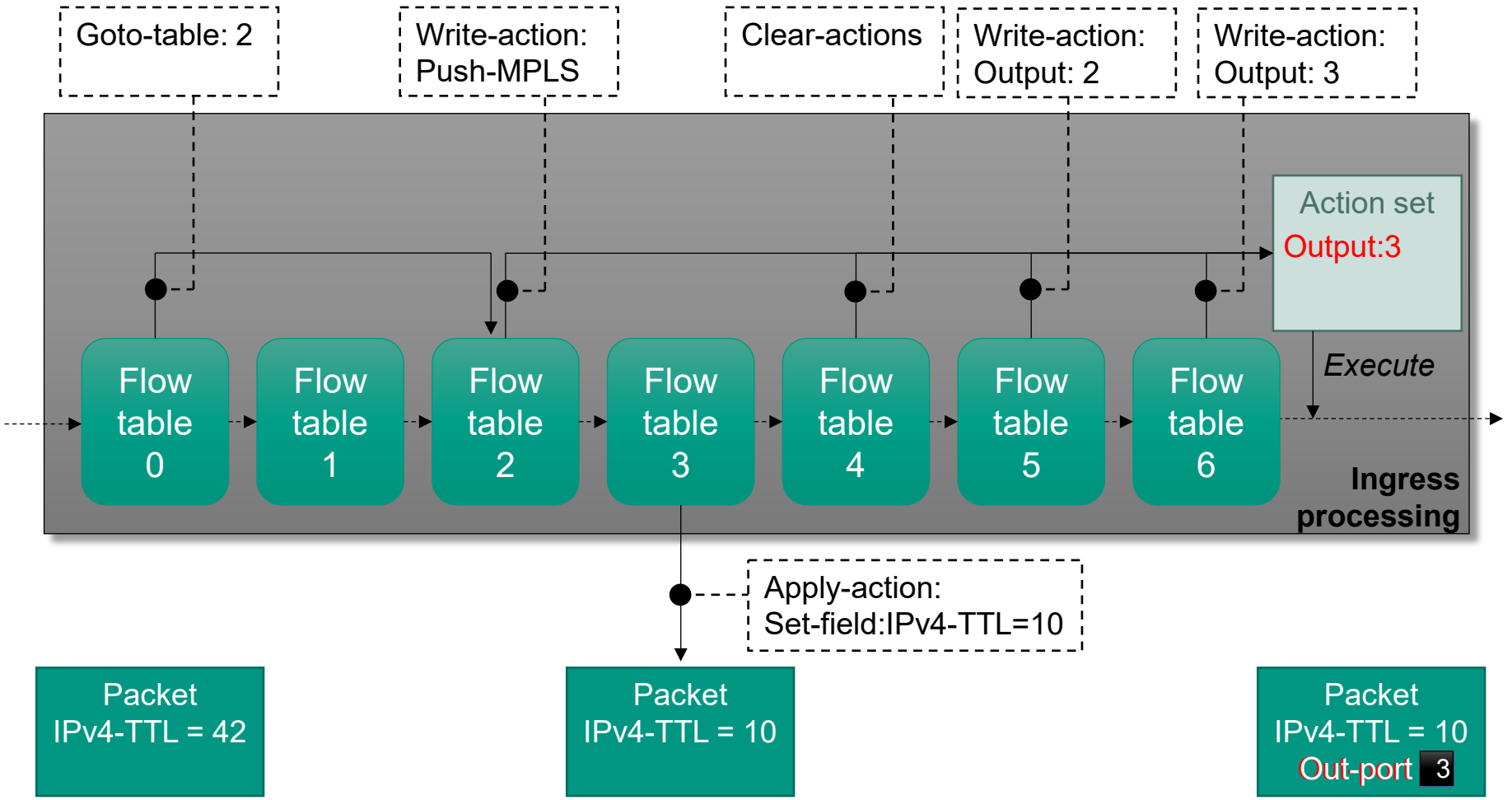
required

Pipeline Processing

- Multiple flow tables can be chained in a flow table pipeline
  - Flow tables are numbered in the order they can be traversed by packets
  - Processing starts at flow table 0
  - Only “forward” traversal is possible → no recursion
  - Actions are accumulated in an action set during pipeline processing
- Pipeline processing is divided into ingress and egress processing

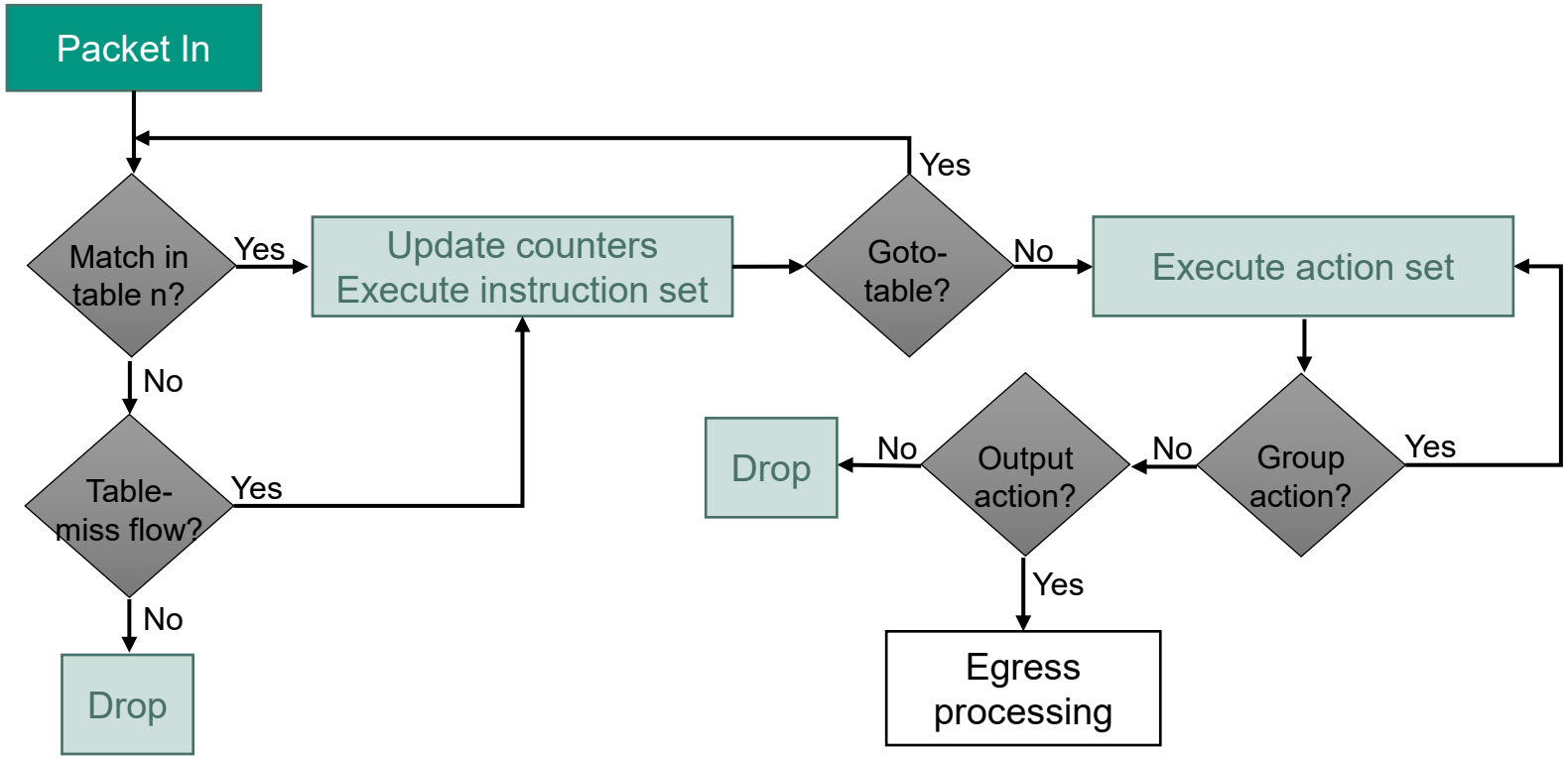


Example: Building an Action Set

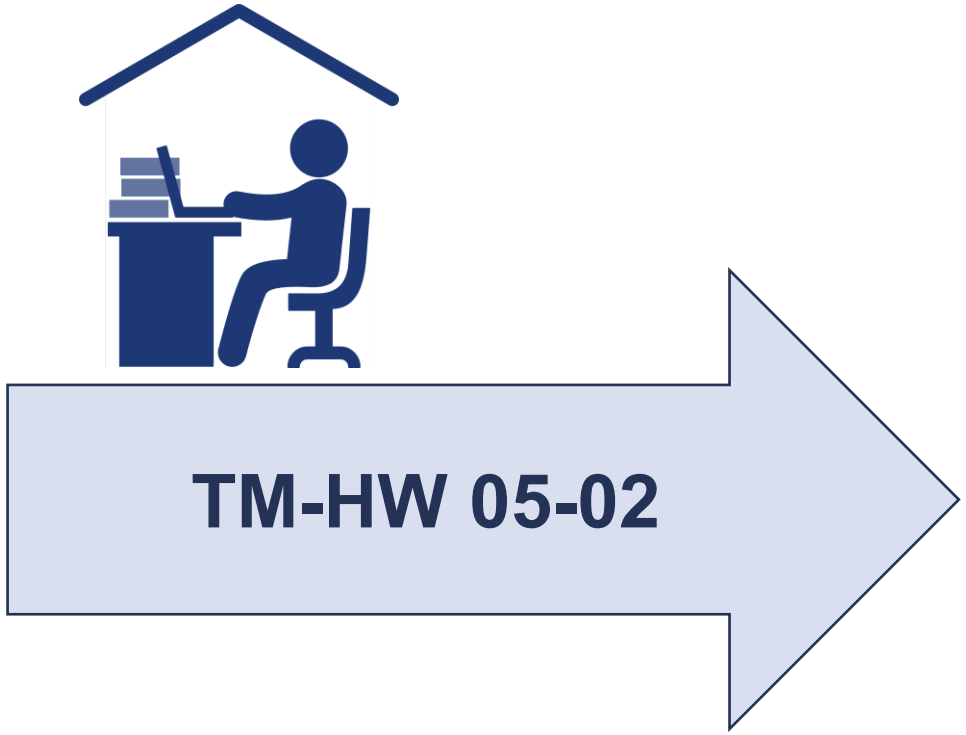


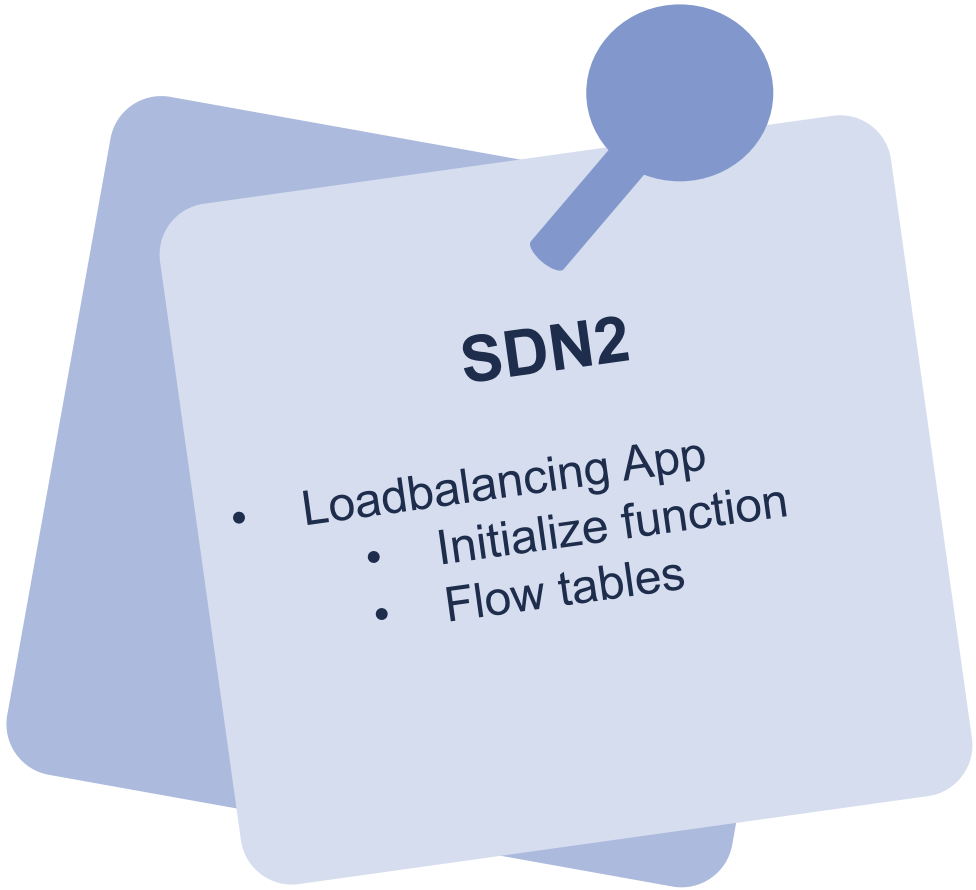
Ingress Processing

- Packet processing starts with ingress processing
  - Starts at flow table 0
    - This may be the only table in an OpenFlow switch
  - Initial action set is empty



Homework


TM-HW 05-02


SDN2
Loadbalancing App
Initialize function
Flow tables

Homework

TM-HW 05-03

SDN – practical part

- Familiarize yourself with SDN Cockpit 2
- Improve demo.py

Homework

TM-HW 05-04

SDN – practical part 2

- Implement VIP policy
- Deal with NVIP traffic

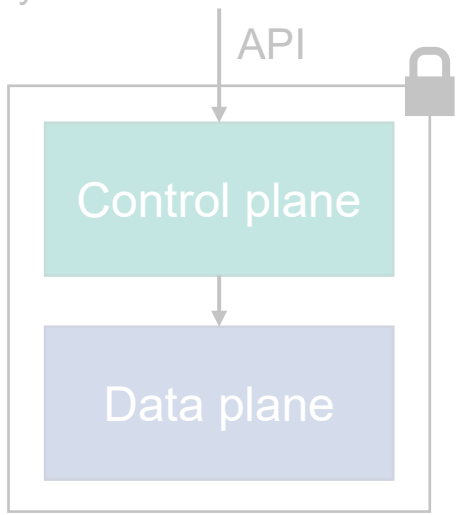
5.3 P4: Data Plane Programming

Some of the slides are adpoted from Prof. Menth, University of Tübingen.

Levels of Programmability

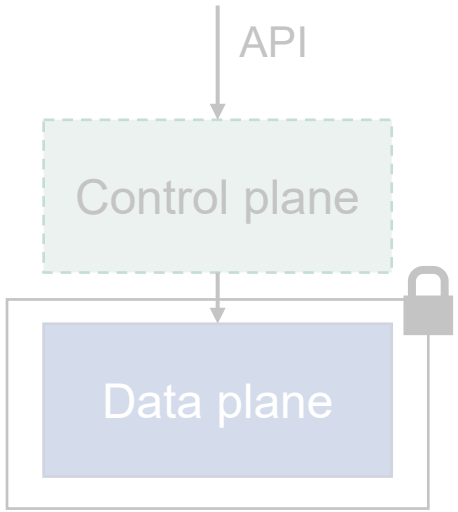
Traditional IP networking

programmability



- API provided by vendor allows some configurability (e.g., routing protocol)
- Algorithms in control and data plane cannot be changed

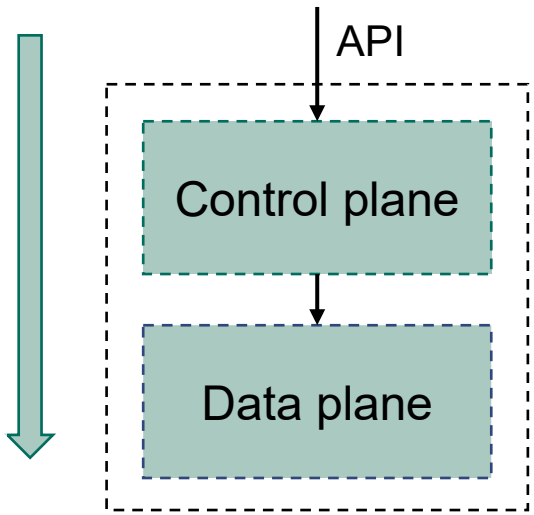
SDN with fixed-function data plane



- Algorithms in control plane can be changed
- Forwarding table in data plane can be programmed through specific interface

Open Flow

SDN with data plane programmability



- Algorithms in control and data plane can be changed

P4

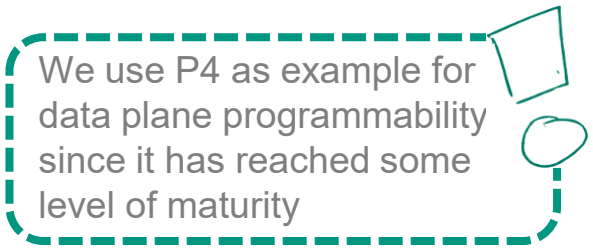
Objectives

- Programm the data plane to your needs

- In high-speed networks
  - Hardware support needed for processing in line-speed
  - Fast path through router/switch commonly implemented on networking processors
    - NPUs: Network Processing Units
    - Traditionally not accessible/programmable by users
      - No APIs or additional support are provided
  - Hardware should be programmable
    - → new developments with P4 and programmable NPUs make this possible

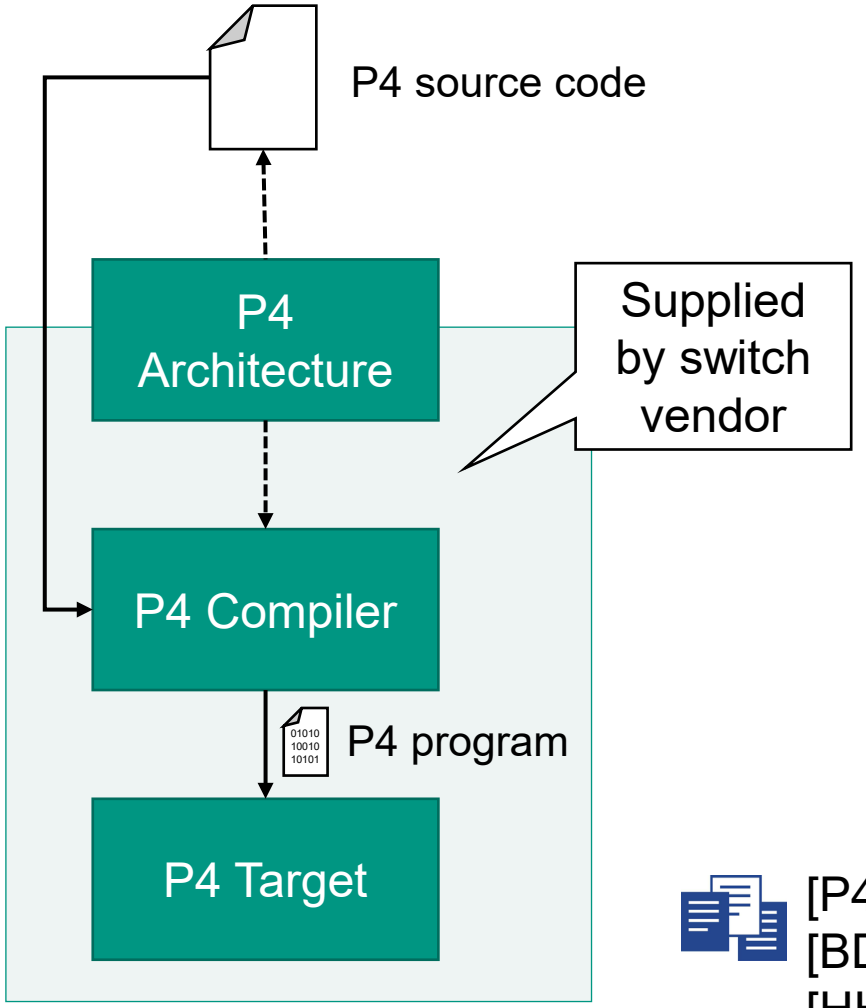
→ P4 (**P**rogramming **p**rotocol-independent **p**acket **p**rocessor)

- Provides high-level programming language to describe data planes
- Abstracts from concrete details of device to be programmed



P4 Overview

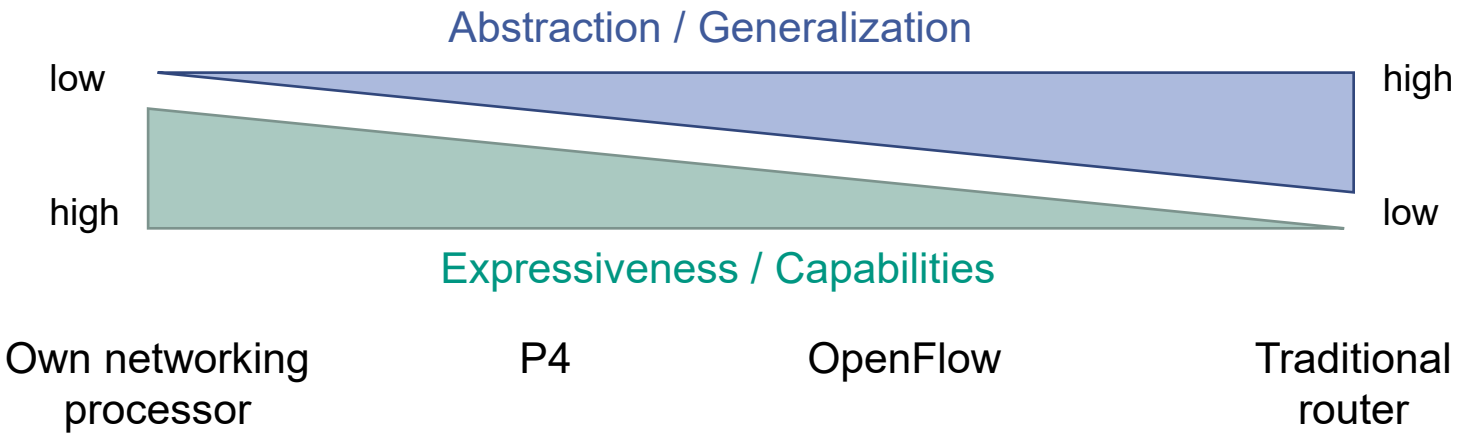
- Protocol independent
  - Does not even include Ethernet or IP
    - Developers have to define these themselves
    - Provides flexibility … not tight to Ethernet and IP
- P4 architecture provides common feature set (interface) across devices
  - Provides hardware abstraction
- Vendor provides compiler for concrete target (device) of architecture
  - Developer does not need to worry about low-level details



[P4.org]
[BDG+14]
[HHM+21]

P4 Advantages

- P4 defines low level (packet processing) operations
  - Fully programmable data plane
  - Entirely new protocols can be programmed
- Limited only by expressiveness of P4 and features of P4 target (device)
- Trade-off: generalized interface (OpenFlow) vs. staying close to hardware (P4)

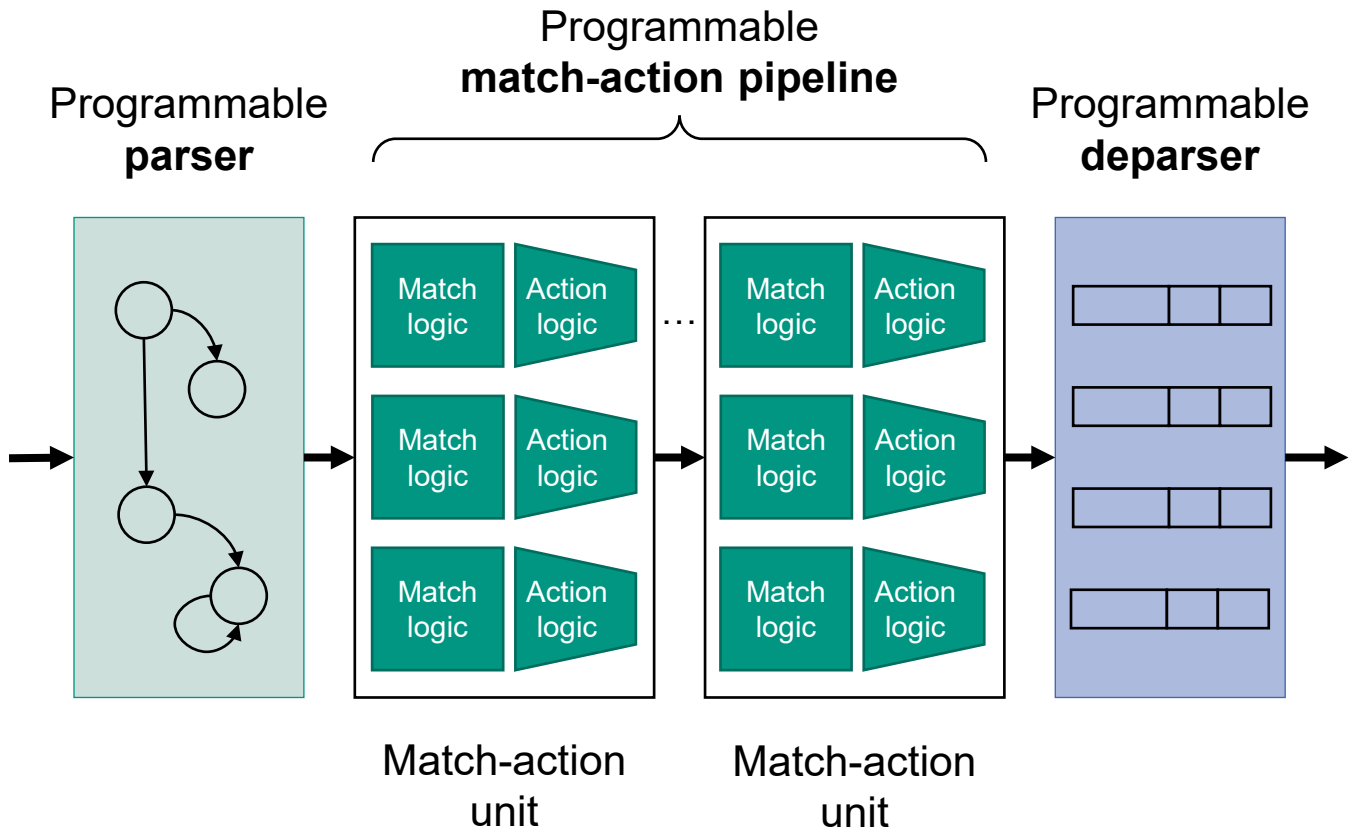


PISA – Example of a P4 Architecture

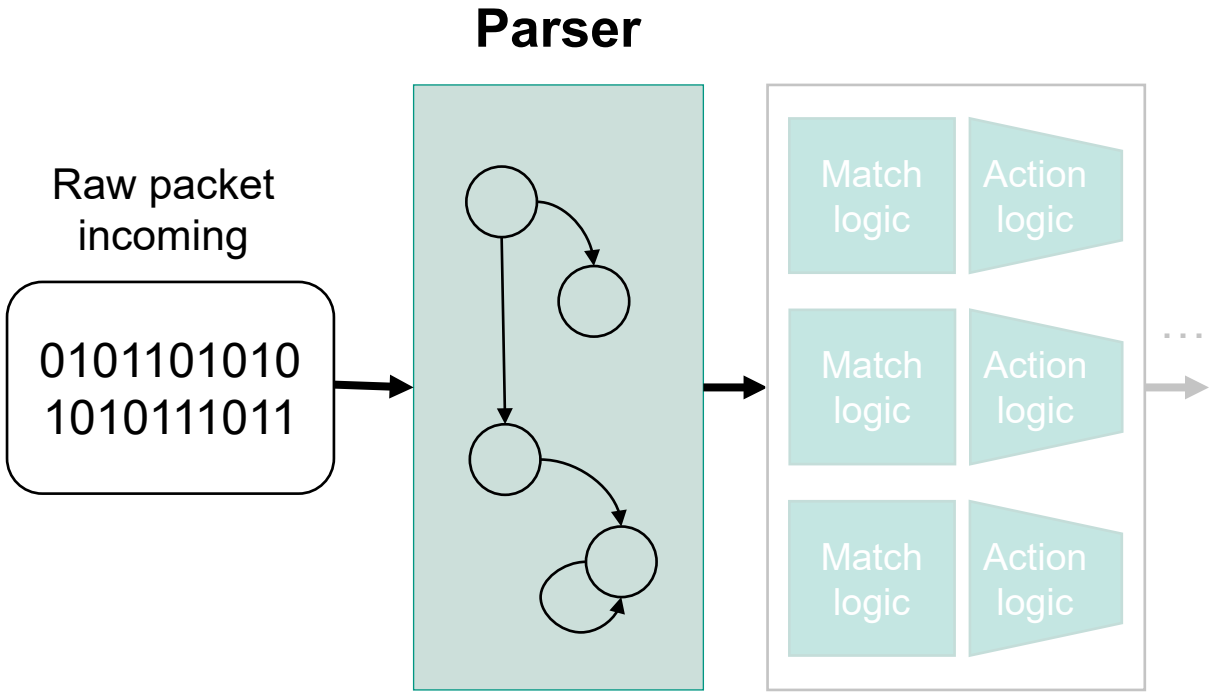
- PISA: **P**rotocol **I**ndependent **S**witch **A**rchitecture
  - Based on concept of programmable Match-Action-Pipeline

Matches high-speed switching architectures

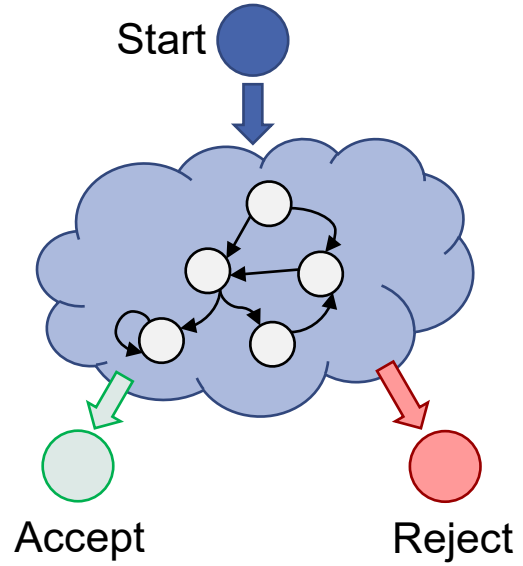
- P4 program consists of
  - Parser
    - Deserializes incoming packet
    - Extracts header fields and metadata
    - Program defines how a packet looks like → protocol independence
  - Match-action pipeline
    - Custom packet processing
    - Match-action logic similar to OpenFlow
  - Deparser
    - Reconstructs headers
    - Serializes outgoing packet



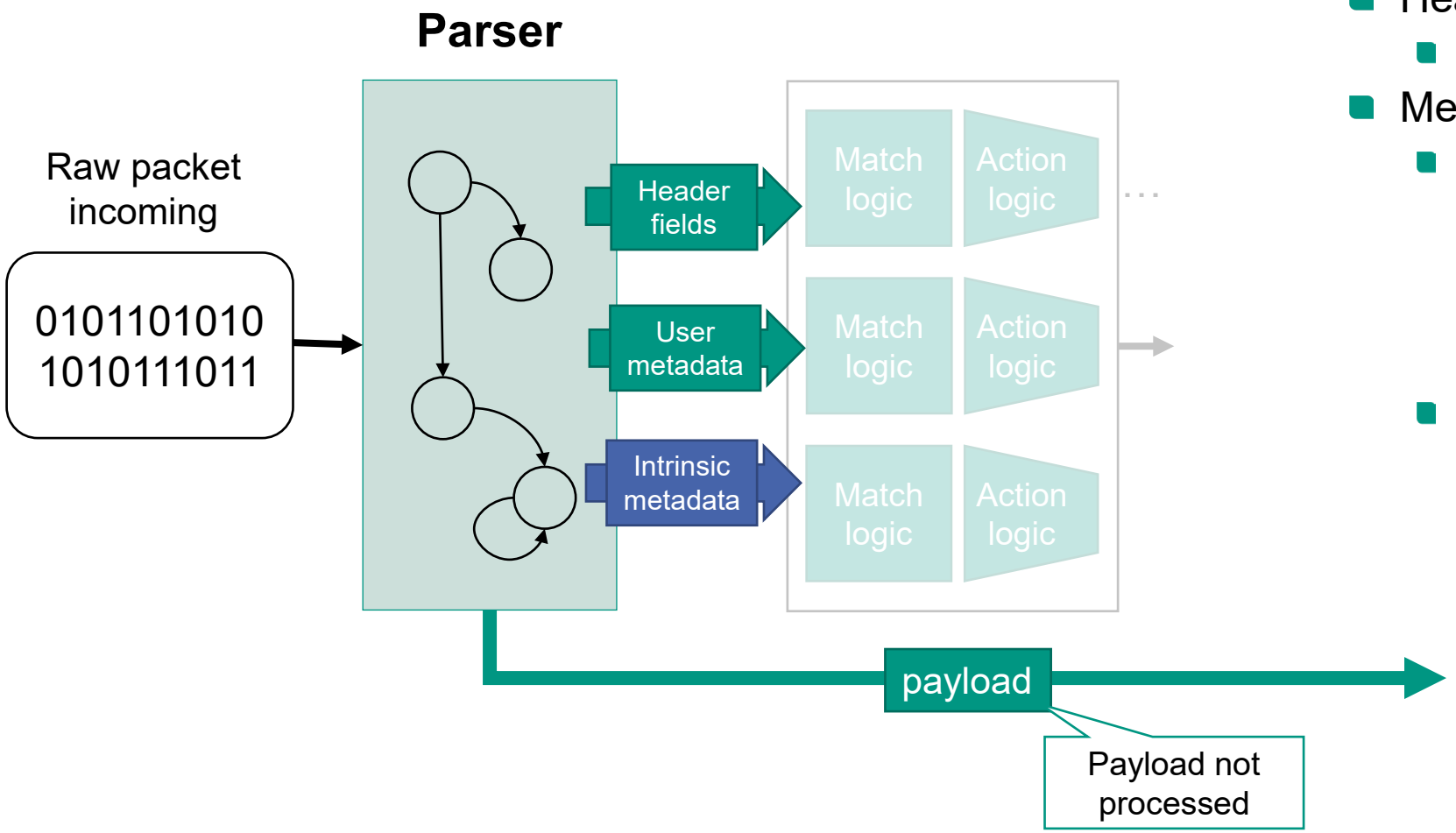
Life of a packet in P4 – Parser



- Described as finite state machine
  - Three predefined states
    - Start
    - Accept
    - Reject
  - Other states may be described by programmer
    - Loops allowed

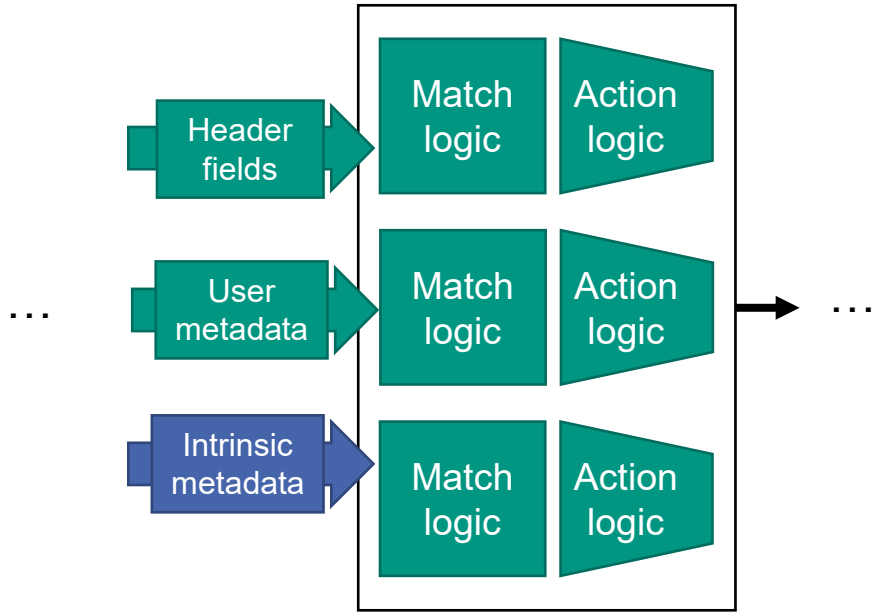


Life of a packet in P4 – Data flow



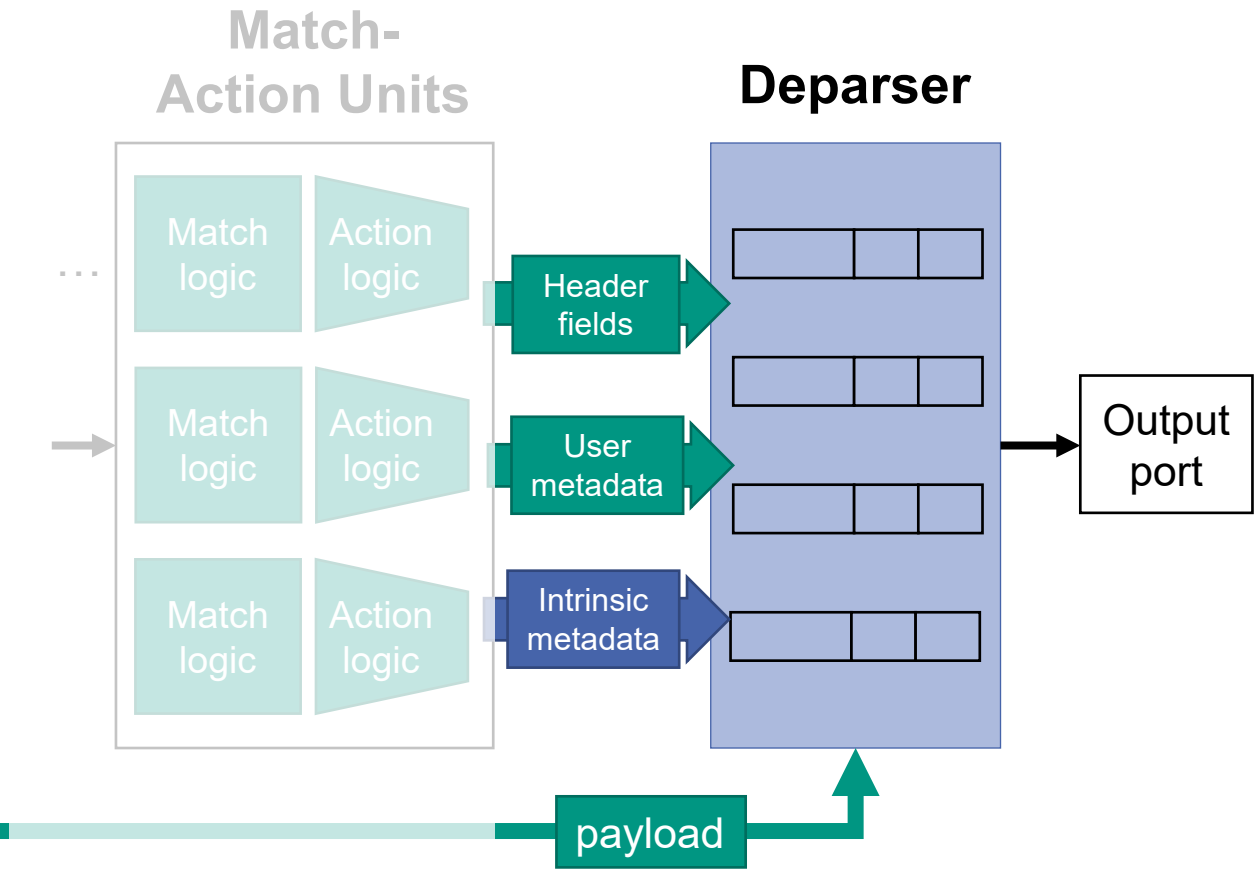
- Two types of fields are derived
  - Header fields
    - → parser
  - Meta data
    - User-defined metadata
      - Defined by user in parser program
      - Data structures associated with each packet
      - e.g., actions (drop, egress port, resubmit), flags
    - Intrinsic metadata
      - Architectural metadata associated with each packet
      - e.g., ingress port, arrival time …

Life of a packet in P4 – Match-Action Units



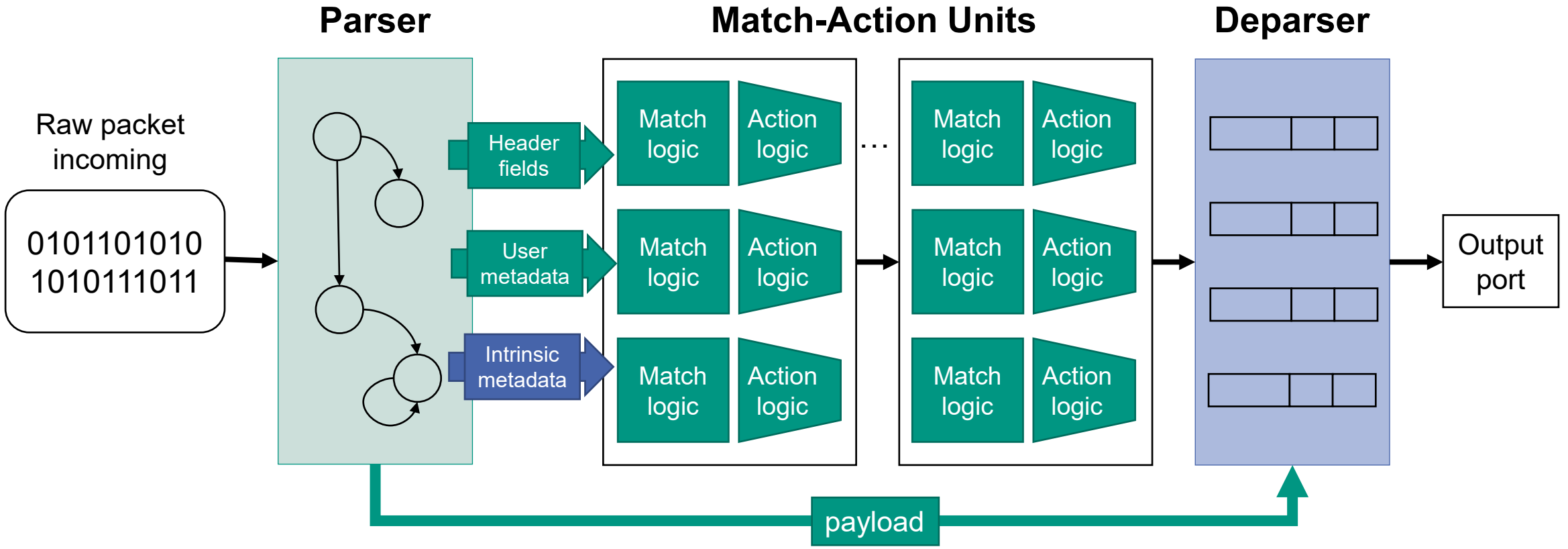
- Control Blocks
  - Define control flow for packet processing
    - Define pipeline: order of match actions
    - Conditional packet processing (if)
  - Similar to „main“ function from other high level programming languages
- Within pipeline
  - Match-action units

Life of a packet in P4 – Deparser



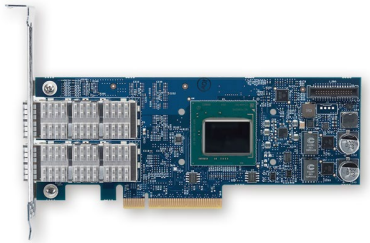
- Tasks of deparser
  - Construct (new) packet header from (modified) header fields
    - Allows completely rewriting packet header
    - Introduce new header for, e.g., encapsulation
    - Recalculate checksums
  - Serialize packet
  - Transmit on output port
    - Egress port previously written by match-action unit to metadata

Life of a packet in P4 – Complete Picture



P4 Targets (Selection)

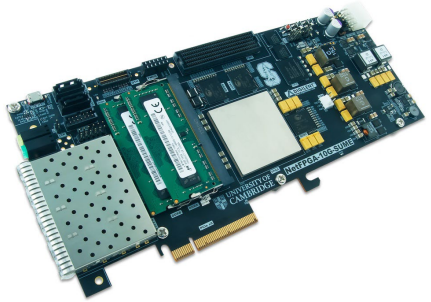
- Software-Targets
  - bmv2 (reference software switch)
  - p4pi (p4 on raspberry pi, for educational purposes)
- Network Interface Cards
  - Netronome Agilio SmartNIC



- Switches
  - With Tofino Barefoot ASIC
  - Vendors: Edgecore, APS Networks

Discontinued. No new P4 ASIC designs…

- FPGAs
  - NetFPGA SUME (discontinued)
  - P4 for AMD Alveo accelerator cards



5.4 eBPF

Packet Processing in Linux

- Previous sections
  - Focus on programmability of switches: SDN and P4
- Observation
  - Lots of networking tasks can be done completely in software
    - Firewalls, load balancing, DDoS mitigation, monitoring
  - Modern general purpose hardware fast enough, e.g., to saturate 1 or even 10 Gbit/s link
    - More than enough for networking tasks close to network edge or on end system
- Problem
  - Normal packet processing in Linux: way too slow
    - Packets sent to user space → context switches and packet copy operations
    - Goal: avoid copy operations
- This section
  - Custom packet processing in kernel space (Linux)
  - Advantages
    - Even more flexibility compared to programmable data plane
    - Cheap commodity hardware

eBPF

- Formerly extended Berkley Packet Filter, now just **eBPF** (not an acronym)
  - Run untrusted code in privileged context (kernel)

- Hook into kernel at various places (not only network related)
  - Event-driven: hooks trigger eBPF program
  - Various hooks: system calls, tracepoints, network events

- Provides programmability of Linux kernel
  - Used for debugging, statistics, security audits, modifying standard behavior, …

- Advantages over kernel modules (linux drivers)
  - Modules must be kept up-to-date in step with new kernel versions
  - Modules may crash and take the whole system down
  - No security boundary between module and kernel (module may compromise kernel)

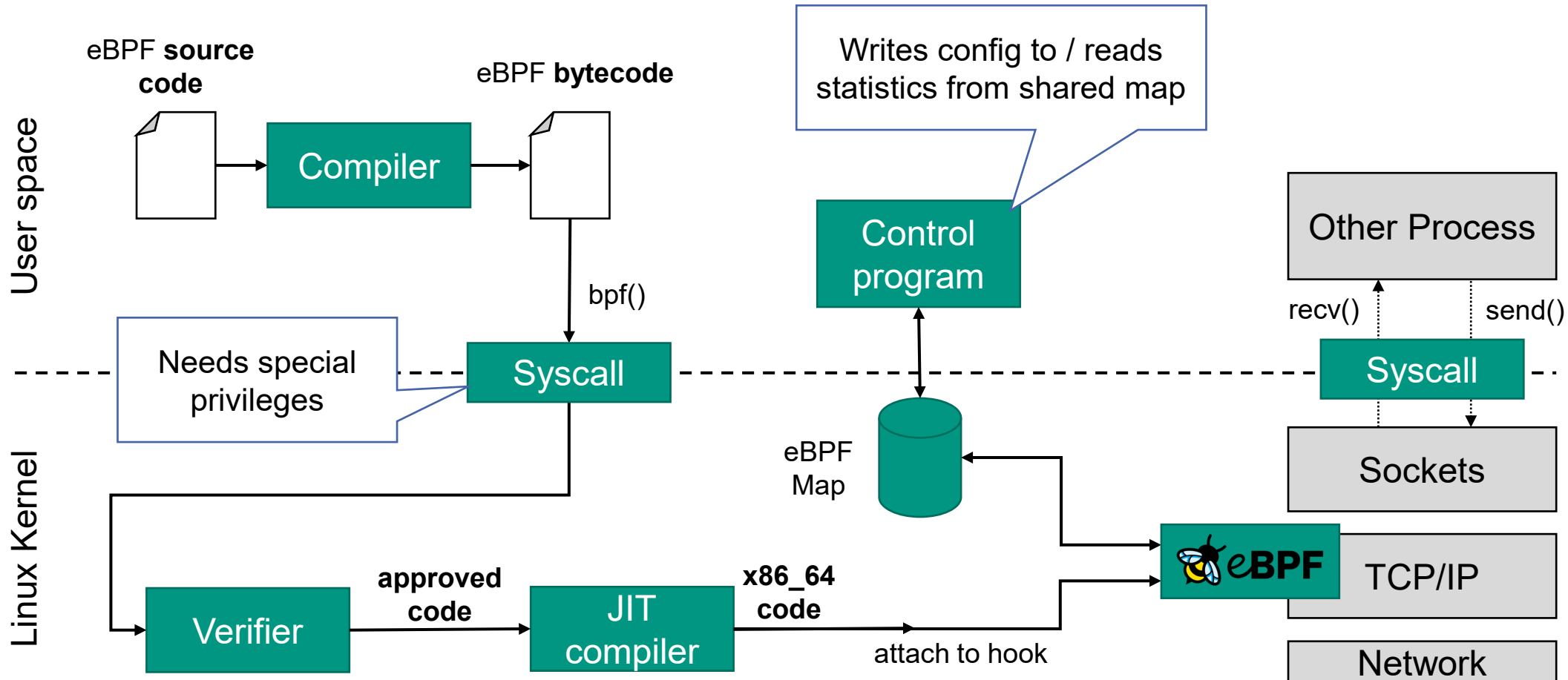
[eBPF.io]

eBPF – Static Verification

- Static verification of the code to ensure eBPF program…
  - …does not crash
  - …has no memory leaks
  - …always terminates
    - → only bounded loops allowed

- eBPF program can only have limited complexity
  - Verifier needs to check every possible execution path

eBPF – How to Run Untrusted Code in Kernel?



[eBPF.io]

eBPF – How to Run Untrusted Code in Kernel?

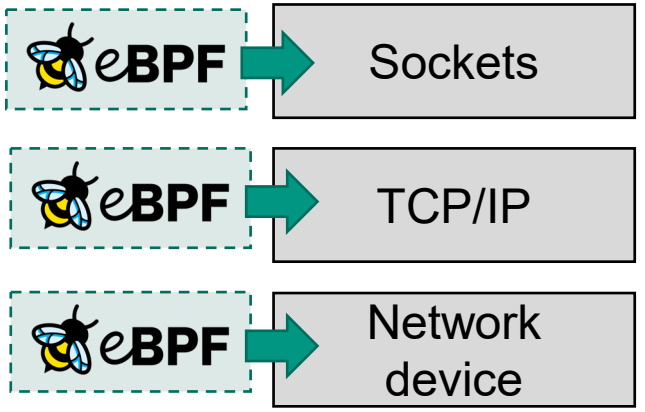
- eBPF source code is compiled to eBPF bytecode
- Bpf() syscall allows privliged processes to load bpf into kernel
- Verifier checks eBPF bytecode for safety
- JIT compiler turns eBPF bytecode into native program (e.g., x86, arm) at runtime

- eBPF programs are registered at specific hooks in kernel, e.g., triggered for each incoming network packet

- Shared data structure (eBPF map) allows communication between eBPF program and control program in user space
  - Control program can, e.g., write configuration or read statistics logged by eBPF program
  - No context switches necessary

Packet Processing with eBPF

- Run custom code securely and safely in kernel ✓
- Ability to hook into linux network stack ✓
  - Hooks at different levels in stack



- Packet processing
  - Extensively modify packet (not only header)
  - Drop, emit to interface, send to application
  - Communication with user application
    - E.g., send some header fields to application
- Written in standard programming language (C, rust, etc.)
- More flexible than programmable hardware
  - P4 architecture dictates some structure (e.g., parser, pipeline, deparser)
- Can be used in virtualization (eBPF in/for linux containers)

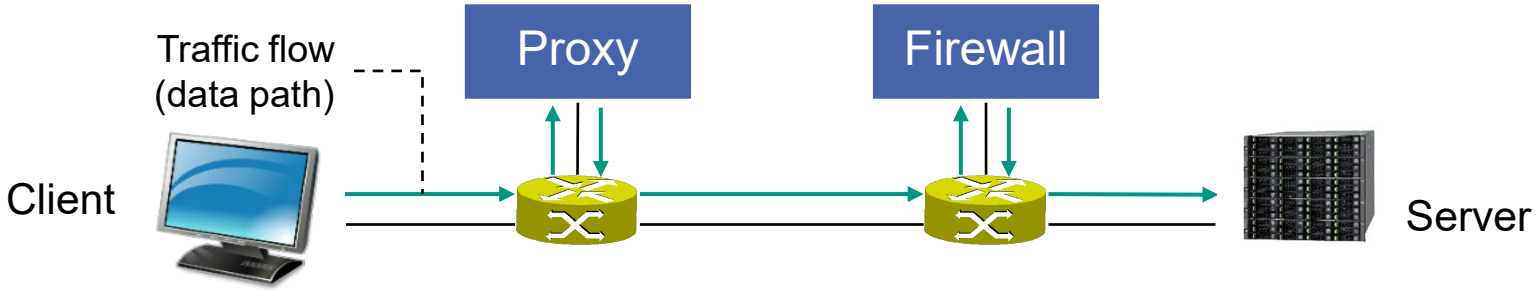
5.5 Network Virtualization

5.5.1 Network Functions

Middleboxes and Network Functions

- Middlebox
  - Device on the data path between a source and destination end system that performs functions other than normal, standard functions of an IP router

[RFC3234]



- Network function (NF)
  - Functionality of a middlebox
  - Executed on the data path

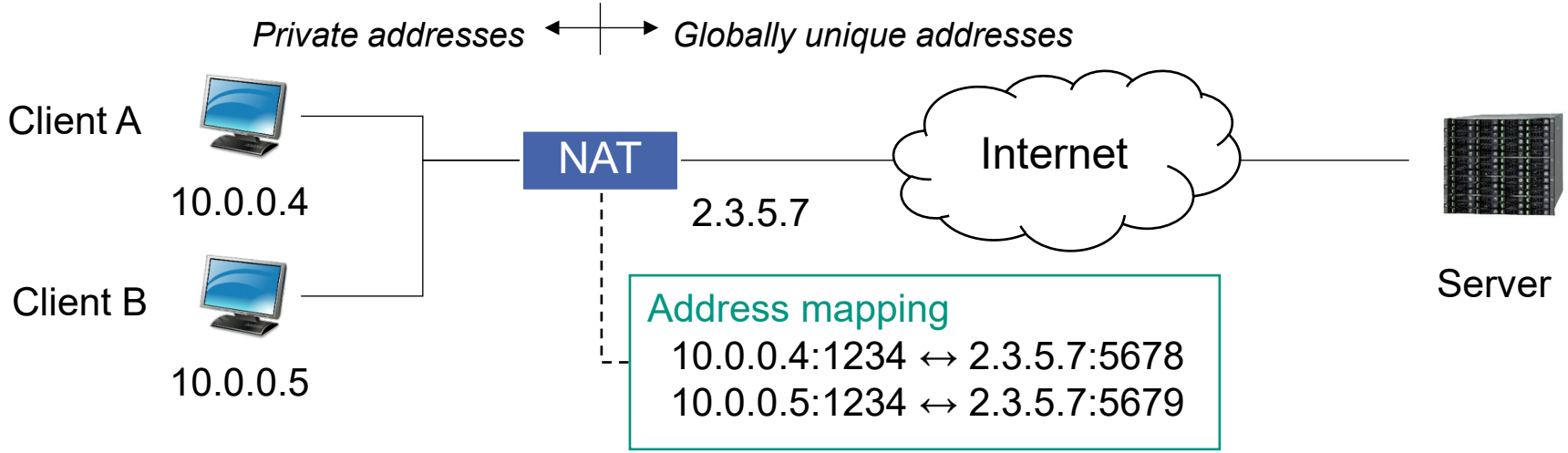
[Raic13]

Example NF: Network Address Translation (NAT)

- Connects network with private addresses to external network with globally unique addresses
  - Problem: private addresses cannot be used for routing in the Internet
  - Switch globally unique and private addresses when packets traverse network boundaries

  → Clients in private address range can share globally unique addresses

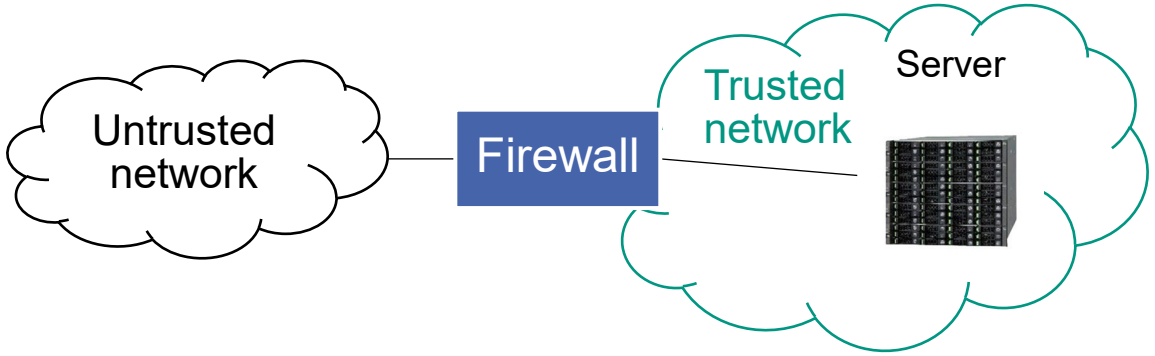
- Example
  - Two clients access same service on a remote server



[RFC3022]

Example NF: Firewall

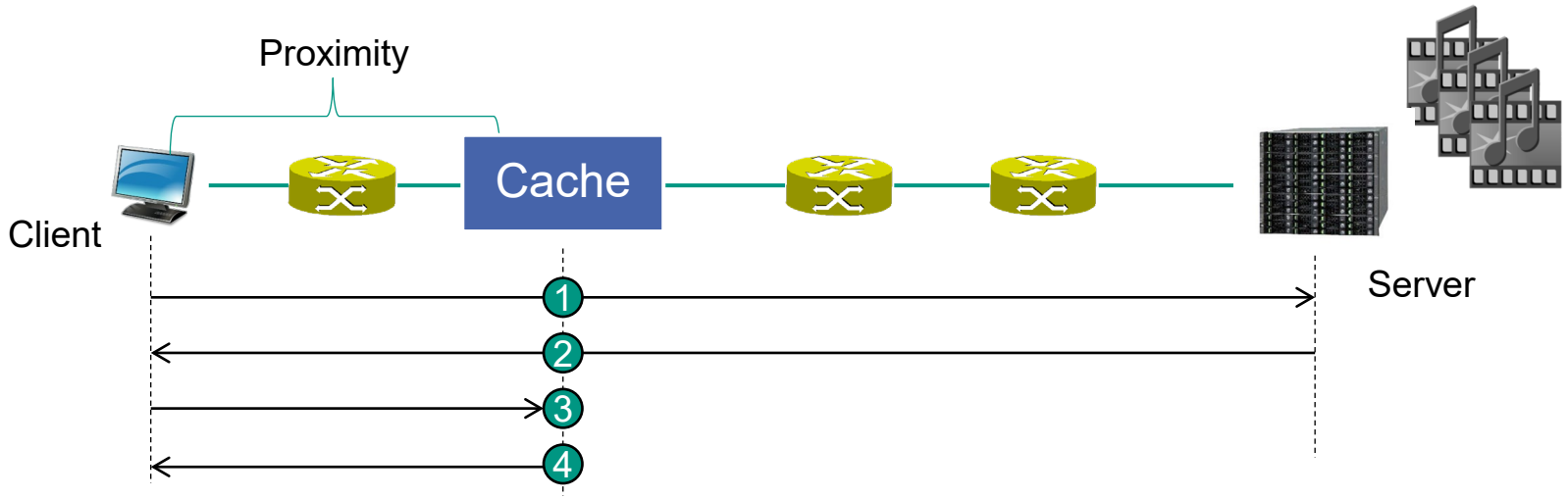
- Monitors and controls incoming and outgoing traffic
  - Establishes barrier between trusted and untrusted networks
  - Forwards or drops packets based on pre-defined rules



- Many variants exist, e.g.,
  - Shallow vs. deep packet inspection
    - Shallow: decisions based on header fields only, e.g., IP and TCP headers
    - Deep: inspects content of higher layer protocols, e.g., detection of malware traffic in application layer protocols
  - Stateful vs. stateless processing
    - Stateless: every packet is inspected independently of other packets
    - Stateful: keeps state between packets, e.g., for every TCP connection to detect invalid sequence numbers

Example NF: Cache

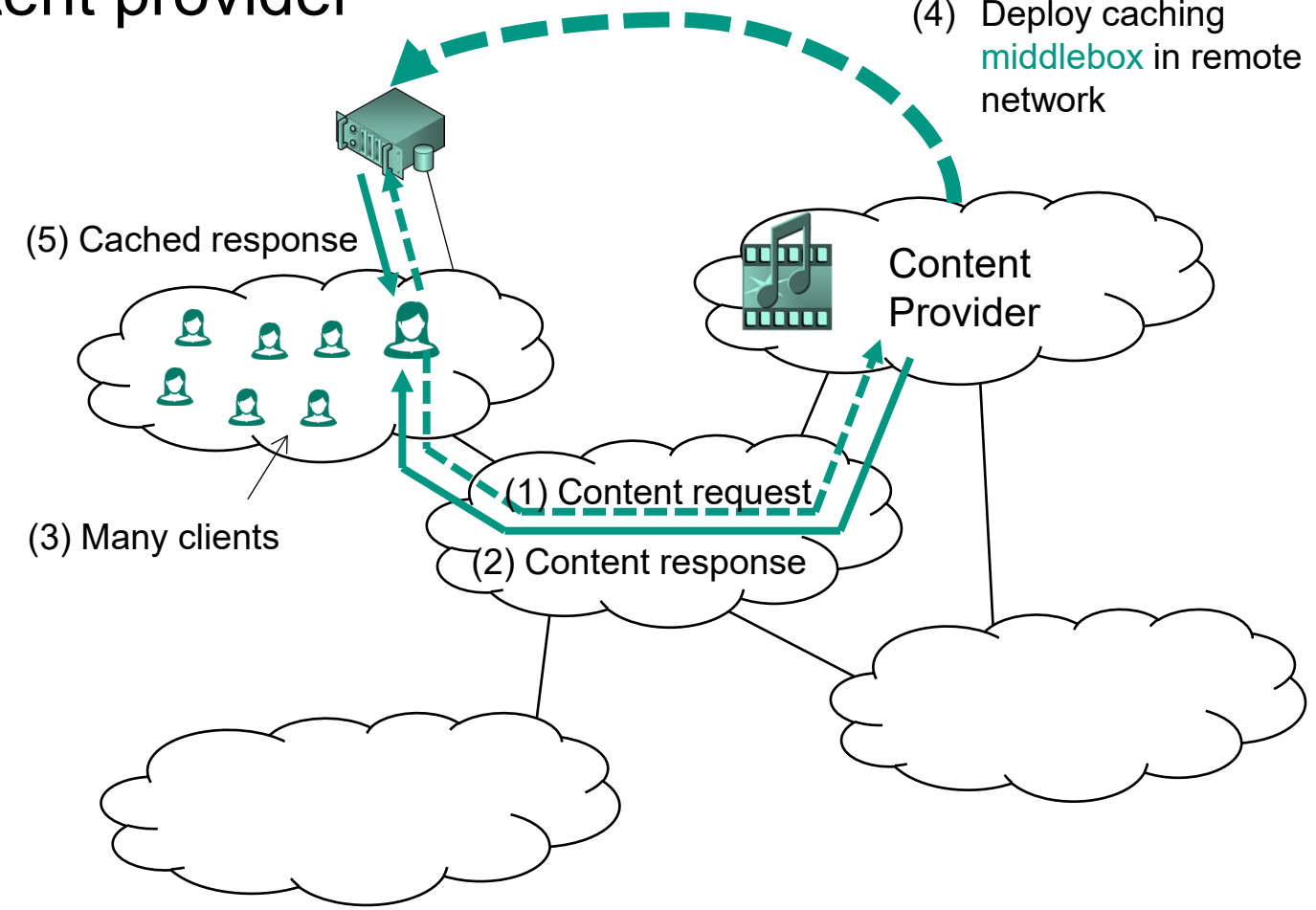
- Caches provide additional storage on the data path
- Temporarily caches content provided by a remote server
- Reduces time for content delivery when placed in proximity of the client
  → Content-delivery networks capitalize on well-placed caches



- Typical sequence
  1) Request content x … not in cache, redirect to server
  2) Store content x in cache
  3) Another request for content x
  4) Responed with cached content

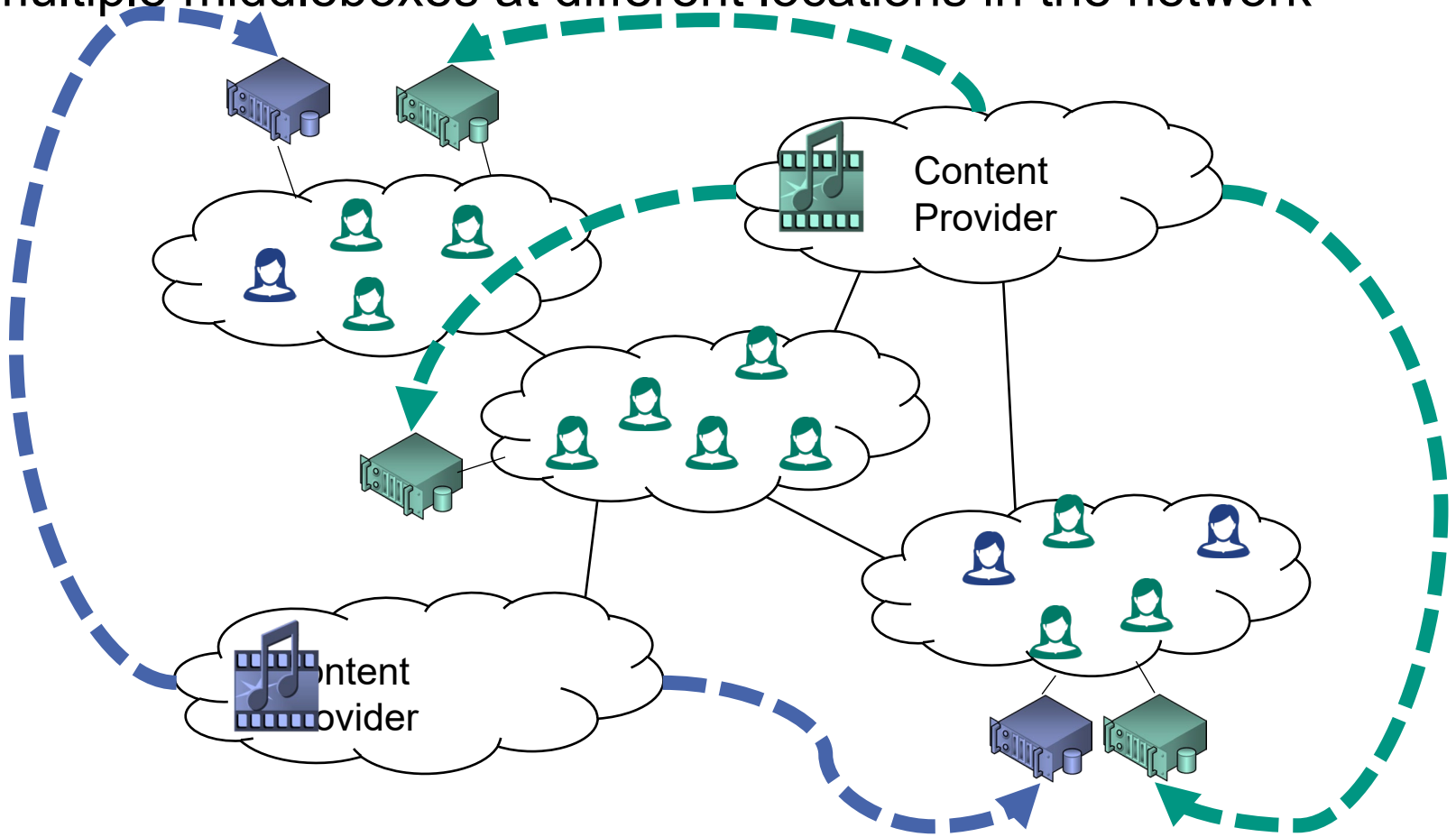
Example: Caching

- Single content provider



Example: Caching

- Multiple content providers
  - Place multiple middleboxes at different locations in the network



Problems?

- Middleboxes are often build as proprietary hardware
  - Fast, but very inflexible
  - Usually closed source → blackbox for infrastructure operator
  - What happens in case of errors? Unplug the cable?

- Static wiring
  - Hard to setup / tear down
  - Hard to move
  - Hard to upgrade → introduce new or bigger boxes

- Network operators have to manage many different vendor-specific boxes
  - Netflix box, Akamai box, …
  - High cost and business barriers

5.5.2 Network Function Virtualization

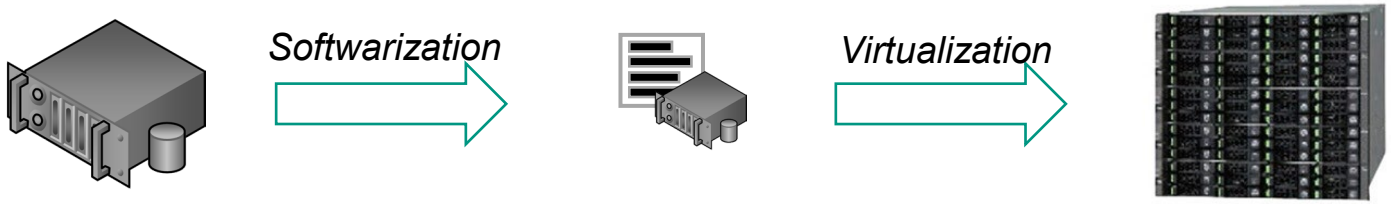
Basic Concept of Network Function Virtualization

- Mimic ideas of cloud computing (application functions)
  - Implement network functions in software
  - Use virtualization technology to decouple network functions from hardware
  - Consolidate functionality on high volume servers, switches and storage

[ETSI14a]

Basic Concept of Network Function Virtualization

- Example: caching functionality



Traditional middlebox with caching functionality

Caching functionality implemented in software

Executed on shared hardware

↓

Easy and fast creation / update of functionality

Can be instantiated at different locations as required

Can be moved across locations

↓

Hardware can be distributed across data centers, network nodes and end user premises

[ETSI14a]

Benefits of Network Function Virtualization

- Resource sharing
  - Single platform for different applications and users
- Agility and flexibility
  - Services can scale to address changing demands
- Rapid deployment and innovation cycles
  - Providers can easily trial and evolve services
- Reduced costs
  - x86 processors are cheaper than proprietary middlebox equipment
  - Flexible provisioning can reduce the need for overprovisioning
  - Reduced need for specially educated personnel

[ETSI12]

SDN and NFV

- Independent but complementary concepts

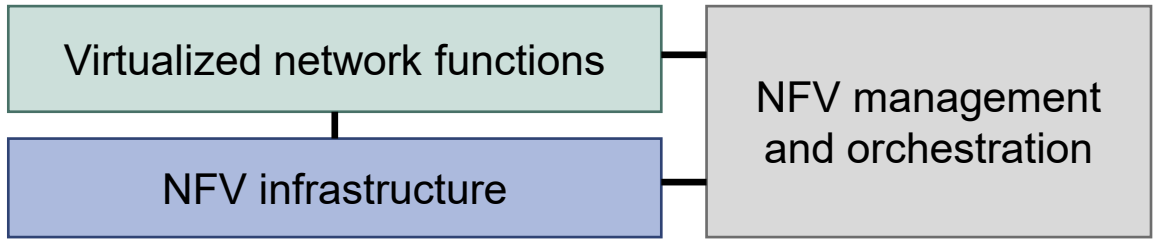
- SDN
  - Decouples data plane from control plane

- NFV
  - Decouples network functions from specific hardware platforms
  - Network functions run on virtual machines (VMs)

  → more efficient and flexible provisioning of such functions

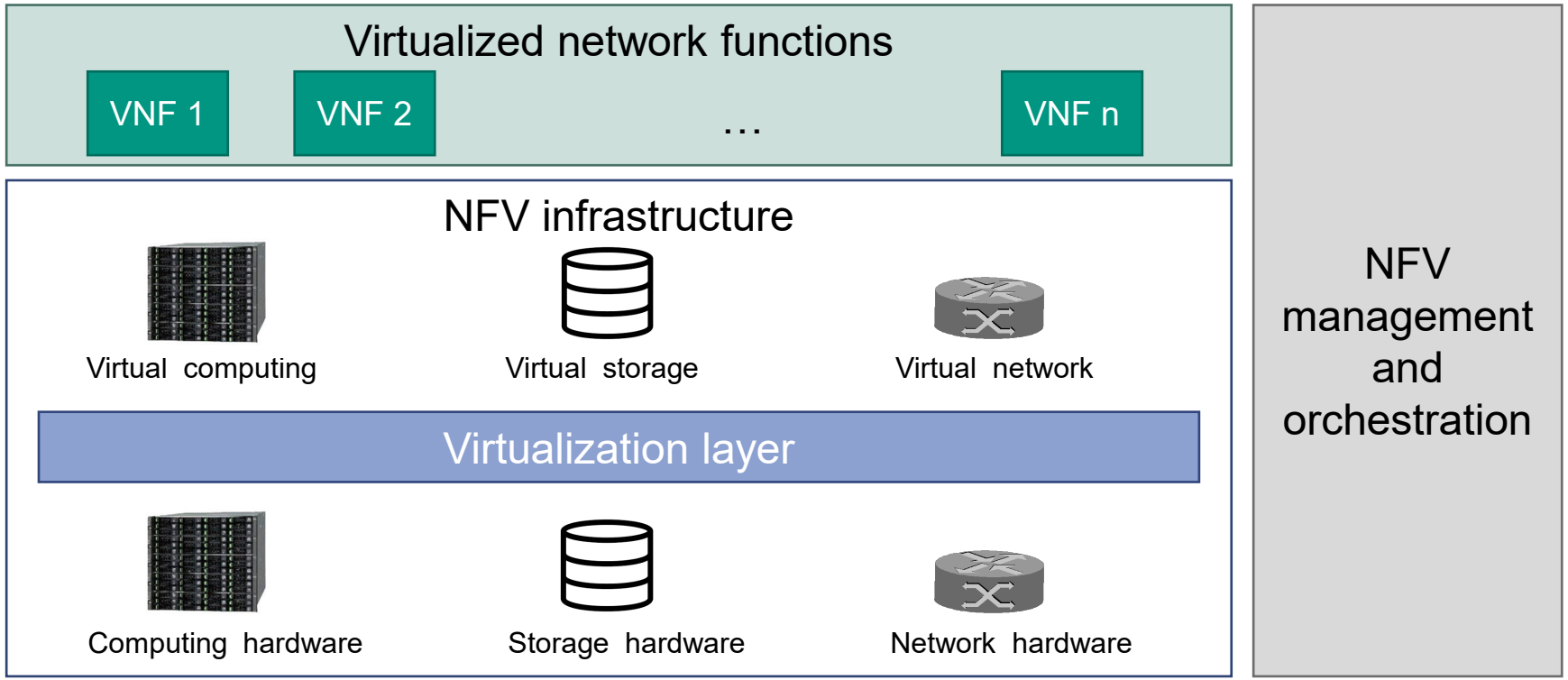
Main Building Blocks of NFV

- Virtualized Network Functions (VNFs)
  - The actual network functions provided in software
  - Independent of its deployment (e.g., hardware)
- NFV Management and Orchestration (MANO)
  - Lifecycle management of VNFs and network services
  - Requests resources for VNFs
- NFV Infrastructure (NFVI)
  - Provides hardware, software and network resources for VNFs
  - Decouples VNFs from underlying hardware



[ETSI14b]

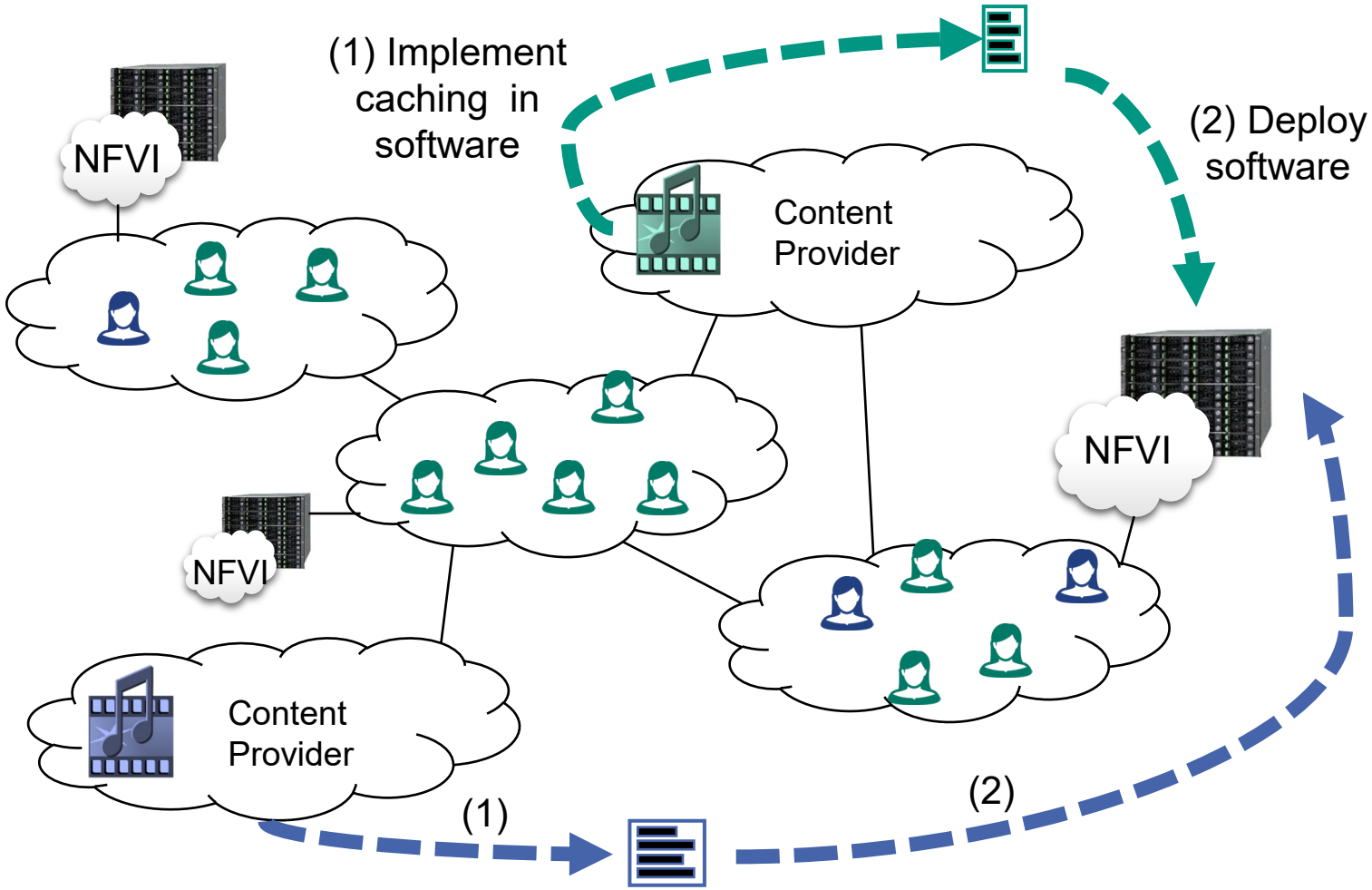
NFV Infrastructure



- Provides physical platform on which VNFs are executed
  - Manages computing, storage and networking resources
  - Usually based on low-cost, standardized hardware
- Virtualization layer
  - Decouples VNFs from underlying hardware
  - Provides virtualized resources to VNFs

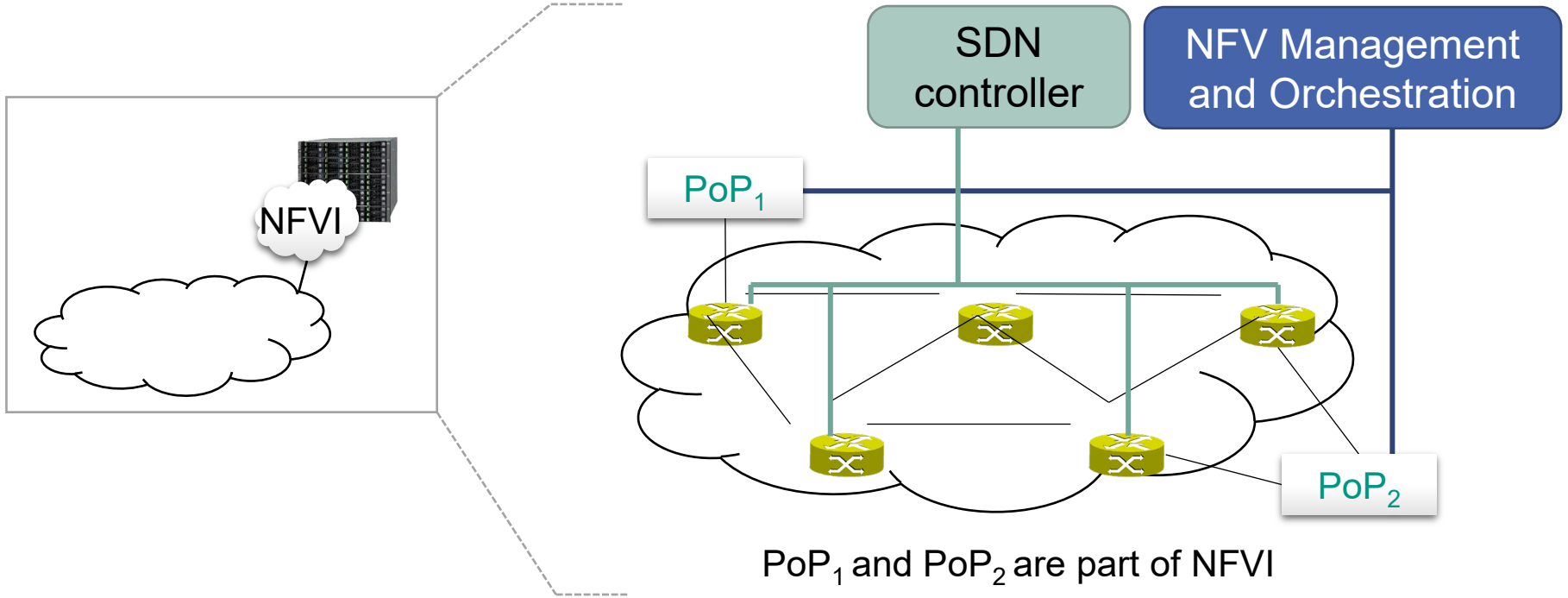
Example: Caching with NFV

- Networks provide NFV Infrastructure (NFVI)



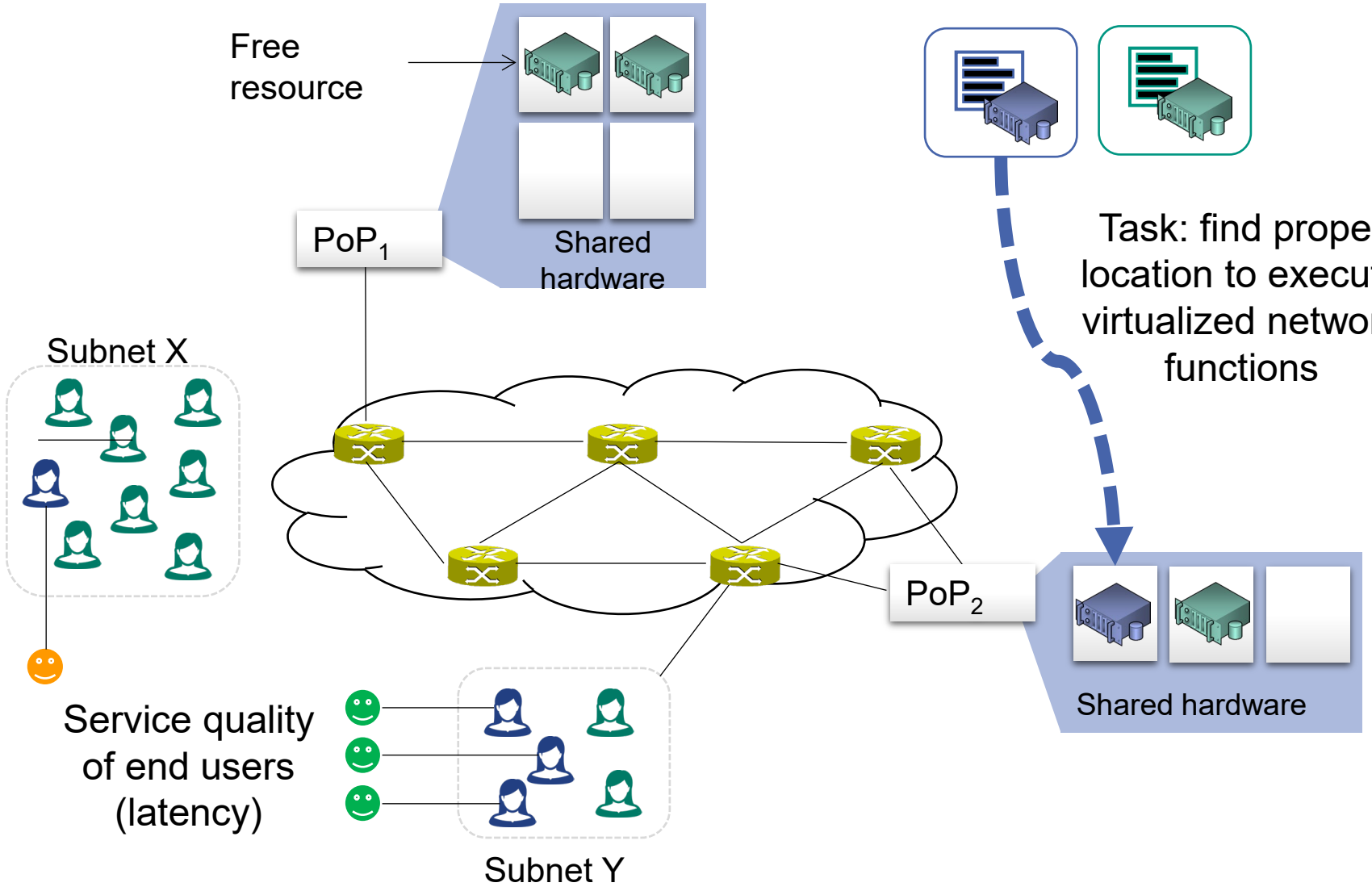
NFV Infrastructure

- Can contain multiple Points of Presence (PoP)
  - Small data centers, located at different points in the infrastructure
- SDN is used to transparently reroute flows to PoPs
  - Could also be done with MPLS or other technologies
  - SDN and NFV complement each other very well



PoP_1 and PoP_2 are part of NFVI

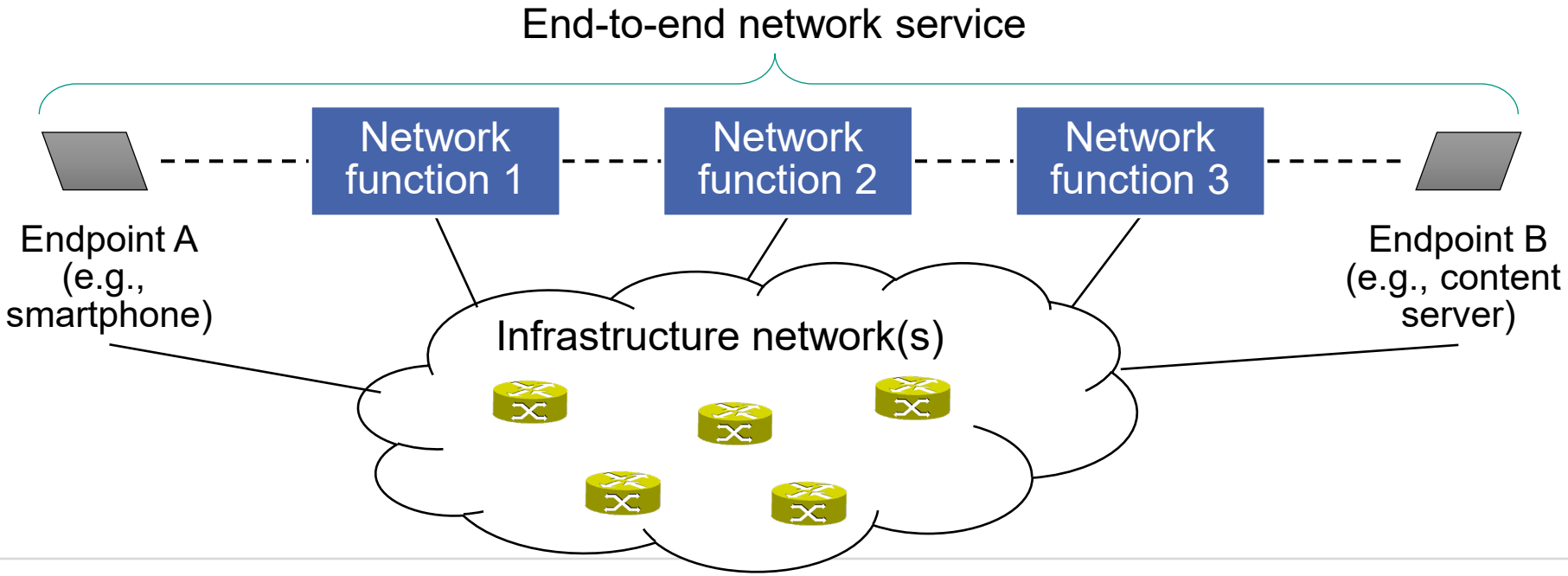
Simple Deployment Example



5.5.3 Service Function Chaining

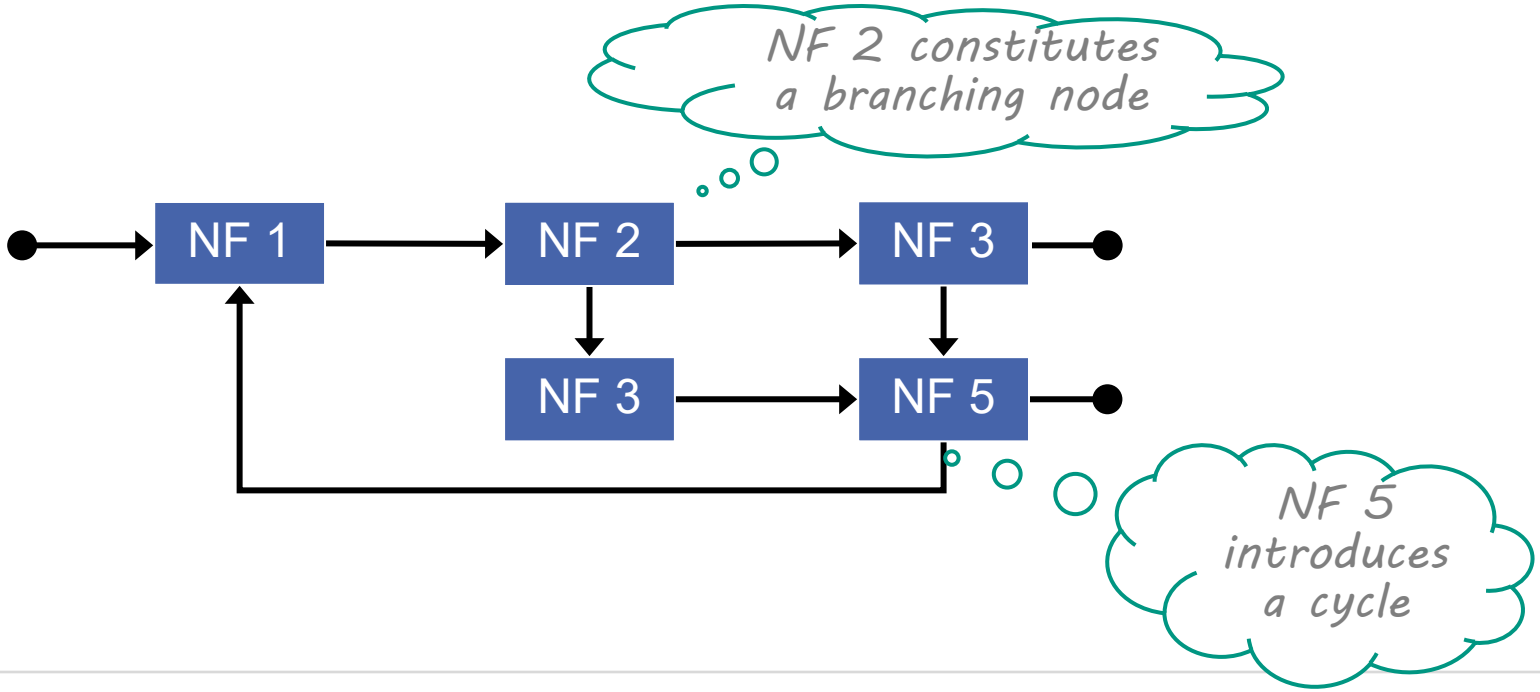
Network Service and Service Function Chain

- (end-to-end) network services may combine multiple network functions
  - Firewall, load balancing …
  - Each network function viewed as distinct node
  - Nodes are interconnected via logical links (dashed lines)
    - Service function chain (SFC) … ordered set of network functions
  - Physical path through infrastructure networks needed



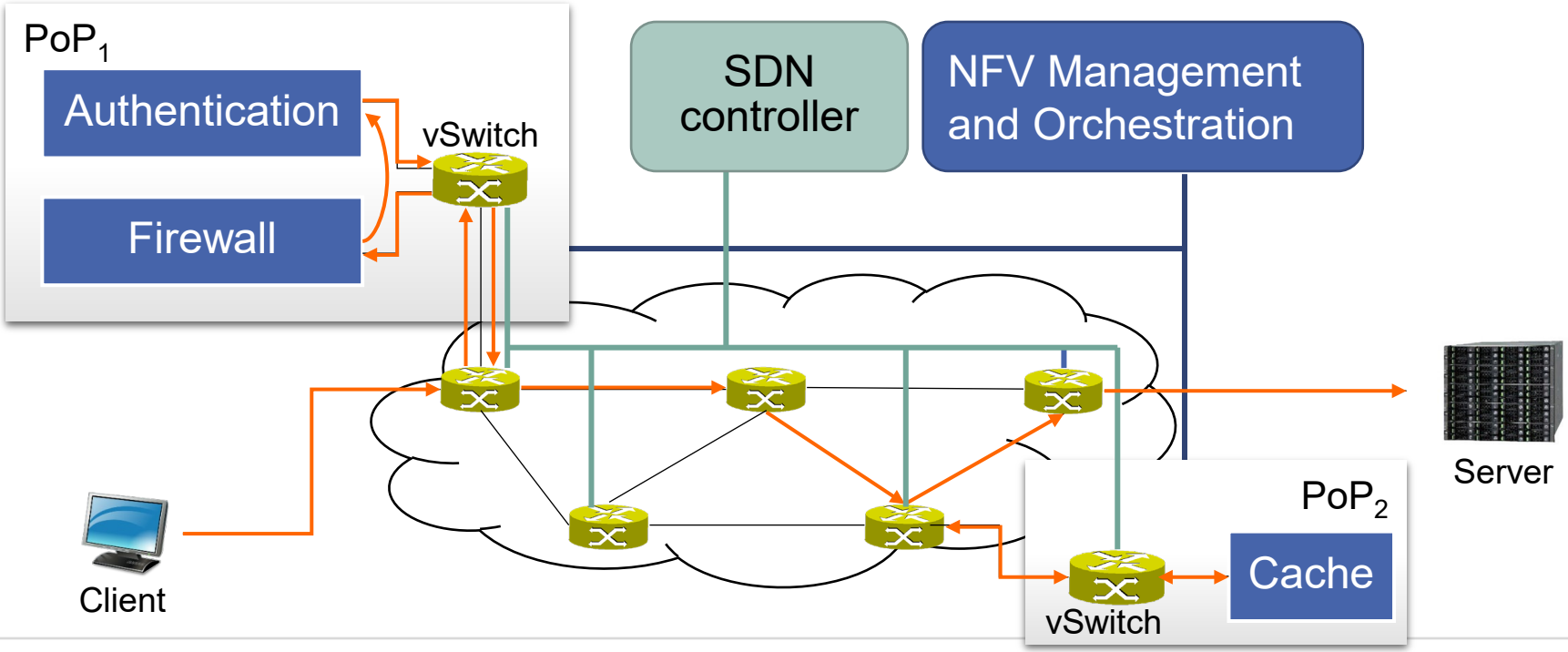
Service Function Chain

- Service function chains can be described as forwarding graphs
  - Nodes represent required network functions
    - Can appear multiple times in a service function chain
    - Can be part of zero, one or many service function chains
      - → Nodes can become branching nodes which redirect traffic to alternate service function chains
  - Forwarding graphs may contain cycles



Example: Advanced Caching Scenario

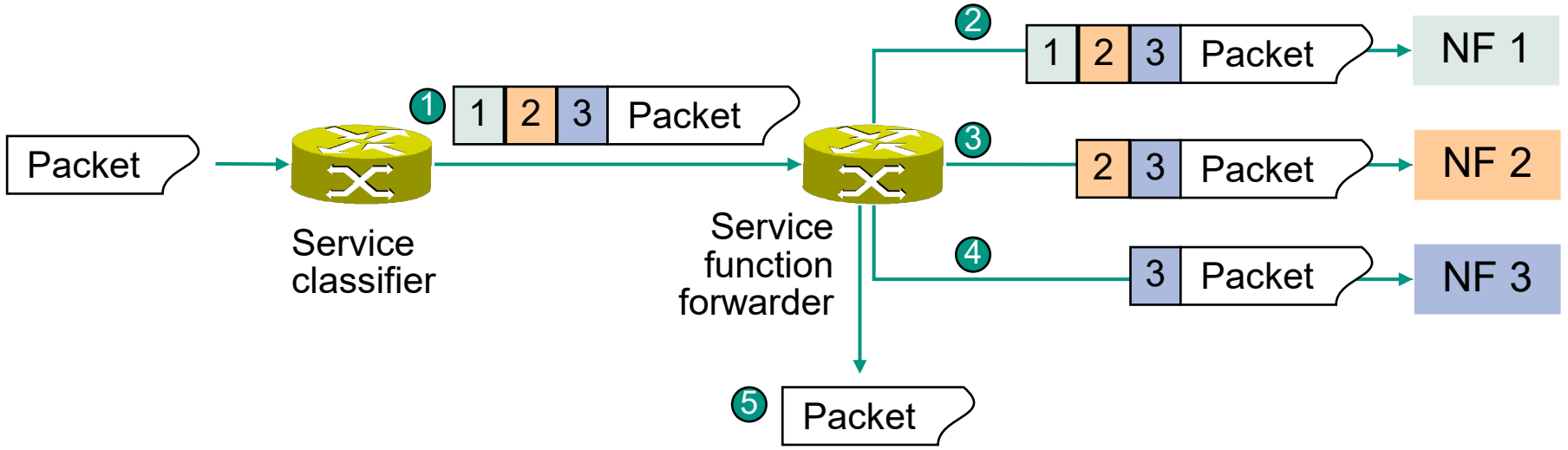
- Place service function chain on data path
  - Firewall → authentication → data path
- Sketch
  1. Required VNFs are instantiated at appropriate PoPs
  2. Service function chain is established (flow table entries in the data plane)
     → Flow table entries enforce correct order of VNF traversal



5.5.3 SFC and MPLS

MPLS-based Service Function Chaining

- Service classifiers select appropriate service function chains
  - Select traffic to be processed in the chain
  - Attach a stack of MPLS labels to packets to determine their path through the chain
- Service function forwarders deliver packets to network functions
  - The service function indicated by the topmost MPLS label is applied
  - The topmost label is removed from the stack afterwards
  - Normal traffic flow resumes when the MPLS stack is empty



PROBLEMS

Karlsruher Institut für Technologie

1) What are the basic principles behind SDN? Explain each of them shortly!
2) What is the „flowspace“ and how does it relate to SDN?
3) Would you consider using microflows or macroflows, if the available flow table space of the switches in your network is heavily limited? What trade-off do you have to keep in mind?
4) Is it possible to combine the use of macroflows and the reactive mode of operation?
5) List and explain the elements of a rule and a flow table entry.
6) Give an example for an SDN app that uses proactive flow programming. Use the SDN pseudo programming language!
7) Give an example for an SDN app that uses reactive flow programming. Use the SDN pseudo programming language!
8) Briefly sketch the layered SDN architecture and explain the function of the different interfaces (southbound, northbound, …)
9) What is OpenFlow? Is there an OpenFlow counterpart in the „non-SDN“ world? For what purpose does OpenFlow need reserved ports?
10) Give two example for controller-to-switch messages
11) What were the main problems with the different learning switch examples? What were the solutions?

12) Is it possible in the SDN pseudo programming language to extract (only) the 10th packet of every TCP flow?
13) Can you think of two (conceptually different) use cases where the group table functionality is required?
14) Consider a simple flow between H1 and H2 (identified by IP source and IP destination) that passes five SDN switches. How can this be programmed with the SDN pseudo programming language from the lecture, if a reactive approach should be used and a static getOutputPort(switch, IP) function is available? (MAC address resolution can be ignored)
15) Sketch the process from question 14 as a sequence diagram.
16) What are the two different modes to exchange control messages between an SDN switch and an SDN controller?

LITERATURE

Literature

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker, P4: programming protocol-independent packet processors. SIGCOMM, 2014, https://doi.org/10.1145/2656877.2656890

[BGK+13] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, M. Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. ACM SIGCOMM 2013, New York, USA

[eBPF.io] eBPF: What is eBPF, https://ebpf.io/what-is-ebpf

[ETSI12] European Telecommunications Standards Institute (ETSI); Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. Issue 1. 2012; https://portal.etsi.org/nfv/nfv_white_paper.pdf

[ETSI14a] European Telecommunications Standards Institute (ETSI); ETSI GS NFV 003 – Network Functions Virtualization (NFV); Terminology for Main Concepts in NFV; Version 1.2.1, Oct 2014

[ETSI14b] European Telecommunications Standards Institute (ETSI); ETSI GS NFV-MAN 001 – Network Functions Virtualization (NFV); Management and Orchestration; Version 1.1.1, Dec 2014

[HHM+21] F. Hauser, M. Haeberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, Michael Menth, "A survey on data plane programming with P4: Fundamentals advances and applied research", arXiv:2101.10632, 2021, [online] Available: https://arxiv.org/abs/2101.10632.

Literature

[Hong18] Hong et al.; B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-DefinedWAN; ACM SIGCOMM 2018

[KaDu17] Murat Karakus, Arjan Durresi, A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN), In Computer Networks, Volume 112, 2017, Pages 279-293,

[KuRo22] J.F. Kurose, K.W. Ross; Computer Networking – A Top-Down Approach; Addison Wesley, 8th Edition, 2022

[Kren24] A. Krentsel et al; A Decentralized SDN Architecture for the WAN, ACM SIGCOMM, 20204

[Jain13] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. 2013. B4: experience with a globally-deployed software defined wan. ACM SIGCOMM 2013

[JZH+14] M. Jarschel, T. Zinner, T. Hossfeld, P. Tran-Gia and W. Kellerer, "Interfaces, attributes, and use cases: A compass for SDN", in IEEE Communications Magazine, vol. 52, no. 6, pp. 210-217, June 2014.

[P4.org] https://p4.org/

[Raic13] C. Raiciu, J. Iyengar, O. Bonaventure; Recent Advantages in Relaible Transport Protocols; in H. HassSI; O. Bonaventure, (Eds.), Recent Advantages in Networking, pp. 59-106, 2013

[RFC3022] P. Srisuresh, K. Egevang; Traditional IP Network Address Translator (Traditional NAT); IETF, RFC 3022, Jan 2001

[RFC3234] B. Carpenter, S. Brim; Middleboxes: Taxonomy and Issues; IETF, RFC 3234, Feb 2002

5.A Appendix

5.3.2 Additional material on OpenFlow, might be helpful for homework

for own studies

Flow Table in OpenFlow

- Flow tables contain match/action-associations
  - Matches select the appropriate flow table entries
    - The combination of priority and match fields is unique within a flow table
  - Actions are applied to all packets that satisfy a match

- Table-miss flows capture all unmatched packets
  - Enables reactive flow programming
  - Corresponding flow table entry has lowest priority
  - Synonym: default flow

Flow table (simplified)

Priority	Match fields	Actions
High	Src = 10.0.0.2/32	Drop
Medium	Dst = 10.0.0.0/16	Output 1
Low	*	Controller

Table-miss flow

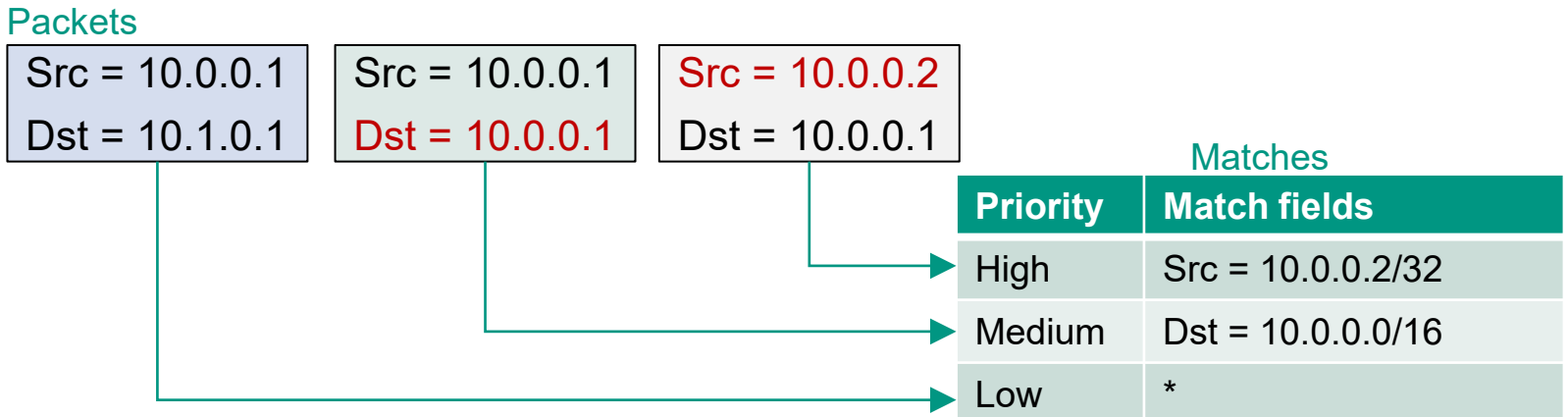
Drop all packets from source 10.0.0.2

Forward all packets from subnet 10.0.0.0/16 via switch port 1

Send everything else to the SDN controller

Matches in OpenFlow

- Matches have priorities
  - Only the entry with the highest priority is selected
  - Disambiguation of similar match fields



- Wildcard matching can be performed using bitmasks
- Empty match fields match all flows

Actions in OpenFlow

- OpenFlow „actions“ are complex
  - Basic functionality is simple: „determine what happens to a packet“
  - In reality, however, OpenFlow makes a distinction between actions, action sets and more general instructions (linked to how the OpenFlow pipeline works)
- Action
  - A concrete command to manipulate packets like „output on port“ or „push MPLS“
- Action set
  - Every packet has its own ActionSet while processed
  - Changes to the packet can be stored in the set / deleted from the set
  - Actual changes are applied when processing ends
  - Set is carried between flow tables (in one switch)
- Instructions
  - Control how packets are processed in the switch
  - Each flow table entry is associated with a set of instructions
    - Change the packet immediately (apply-action)
    - Change the action set
    - Continue processing in another table (goto-table command)

Actions in OpenFlow

- OpenFlow supports various types of actions
  - Output: forwards a packet
  - Set-field: modifies a header field of a packet
  - Push-tag: pushes a new tag onto a packet
    - E.g., VLAN, MPLS
  - Pop-tag: removes a tag from a packet
  - Drop a packet
    - Implicitly defined when no output action is specified

Actions in OpenFlow

- An action set contains at most one action of a specific type
  - Previous instances are overwritten
    - E.g., a later output action overrides a previous one
  - An action set may contain multiple set-field actions
    - Their type depends on the modified field
    - E.g., a set-field action on an IPv4 source address has a different type than a set-field action on an IPv4 destination address
- Execution proceeds in a well-defined order
  - Push-Tag actions precede set-field actions
  - Output actions are always executed last
  - ...
- Modifications to the action set can be performed in to ways
  - write-actions: writing new actions to a set
  - clear-actions: Removing all actions from the set

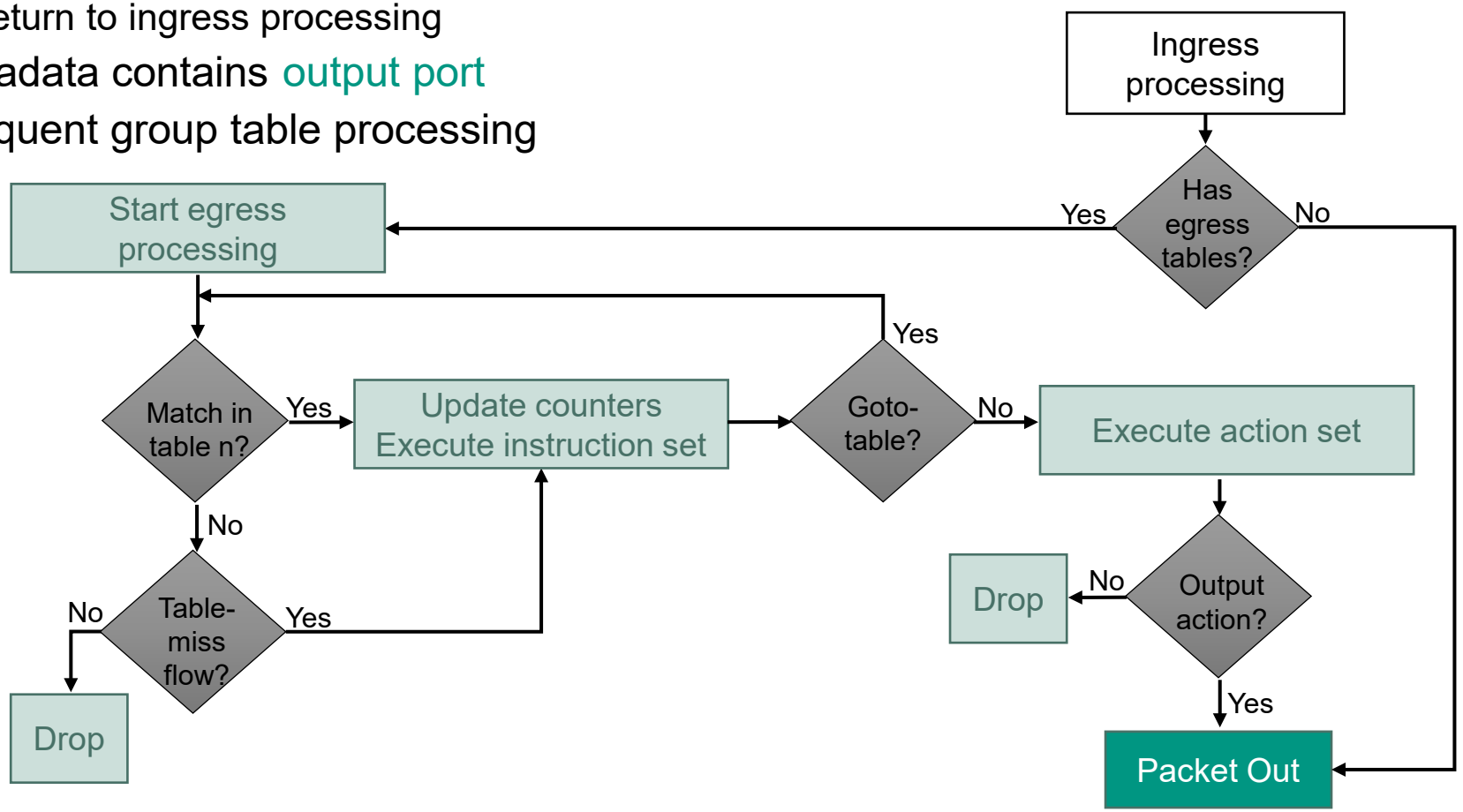
Example: Building an Action Set

1. Flow table 0 applies the instruction goto-table with table 2 as parameter
   → Flow table 1 is skipped
   → Processing resumes at table 2
2. Flow table 2 applies the instruction write-action with a Push-MPLS action as parameter
   → The action is added to the action set
3. Flow table 3 applies the apply-action instruction with a Set-Field: IPv4-TTL=10 action as parameter
   → The action is applied immediately
4. Flow table 4 applies the clear-actions instructions
   → The current action set is cleared is cleared
5. Flow table 5 applies the write-action instruction with an output: 2 action as parameter
   → The action is added to the action set
6. Flow table 6 applies another write-action instruction,
   but this time with an output:3 action as parameter
   → The action overrides the previous output action
7. The current action set is applied
   → The output port is set to port 3

Instructions determine the pipeline processing and control when and which actions are applied.
Actions manipulate packet data and forwarding behavior of packets.

Egress Processing

- Optionally follows ingress or group table processing
  - Egress flow tables must have higher table numbers than ingress tables
    → No return to ingress processing
  - Initial metadata contains output port
  - No subsequent group table processing



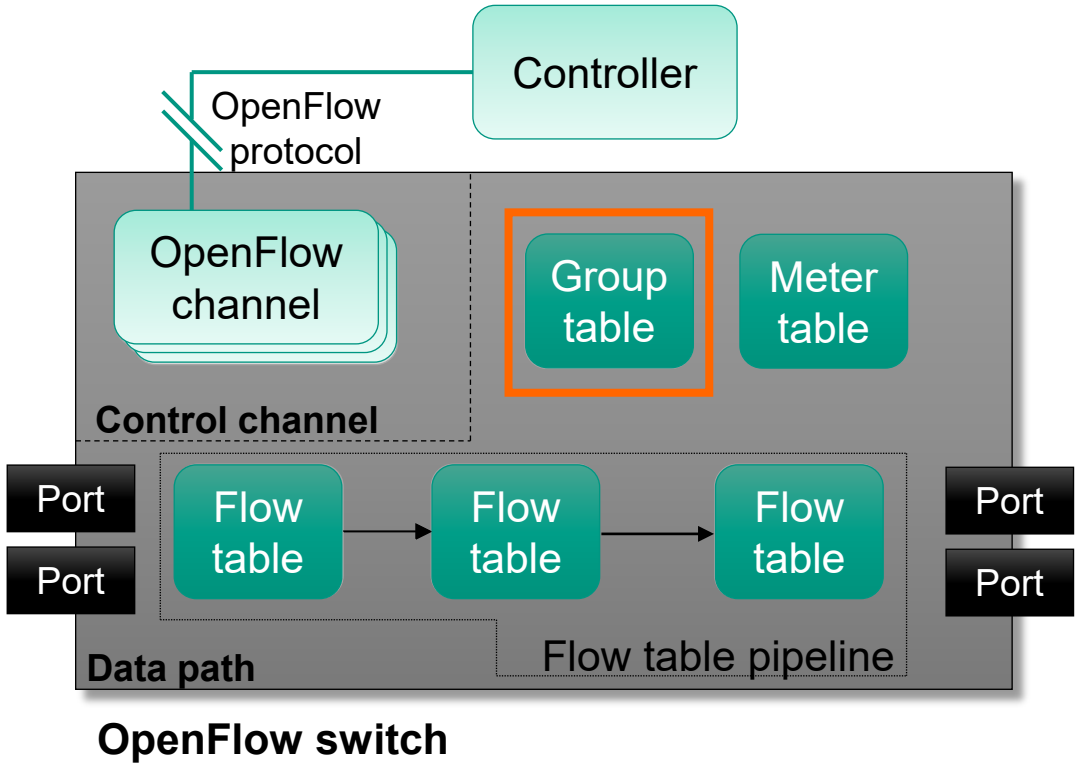
OpenFlow Channel

- The OpenFlow channel connects each switch to a controller
- Provides the southbound API functionality of an OpenFlow switch
  - Management and configuration of switches by controllers
  - Signaling of events from switches to controllers
  - Monitoring of liveliness, error states, statistics, …
  - Experimentation
- Multiple channels to different controllers can be established
- OpenFlow distinguishes between three message types
  - Controller-to-Switch messages
  - Asynchronous messages
  - Symmetric messages

OpenFlow Channel

- Controller-to-Switch Messages
  - Inject controller-generated packets (packet-out message)
  - Modify port properties or switch table entries (modify-state message)
  - Collect runtime information (read-state message)
- Asynchronous messages
  - Packet-in message transfers control of packet to the controller
  - State changes signaled by switches, e.g., flow removal, port status change
- Symmetric messages
  - Handle connection setup and ensure correct operation
    - Hello: exchanged on connection startup (e.g., indicate supported versions)
    - Echo: verify lifelines of controller-switch connections
    - Error: indicate error states of the controller or switch
  - Experimenter messages can offer additional functionality
    - Staging area for future features

Group Tables



Group Tables

Group entry

Group identifier	Group type	Counters	Action buckets

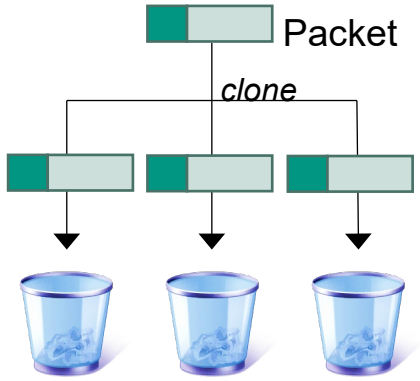
- Group tables represent additional forwarding methods
  - E.g., link selection, fast failover, …
- Group entries can be invoked from other tables via group actions
  - They are referenced by their unique group identifier
  - Flow table entries can perform group actions during ingress processing
- Effect of group processing depends on the group type and its action buckets

Action Buckets

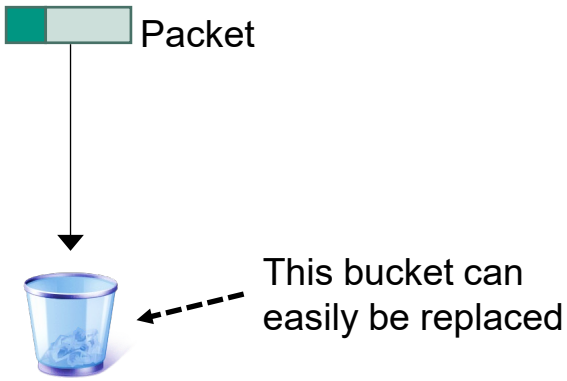
- Each group references zero or more action buckets
  - Not every action bucket of a group has to be executed
  - A group with no action buckets drops a packet
- An action bucket contains a set of actions to execute
  - Just like an action set

Group Types

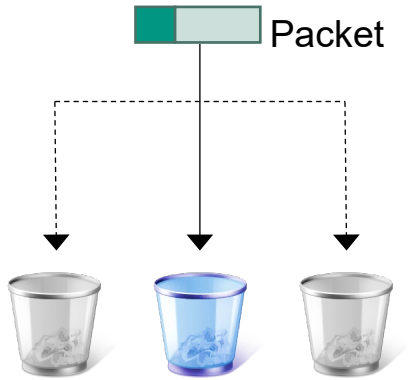
Type = **all**



Type = **indirect**

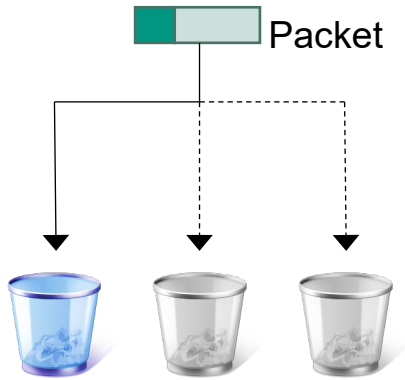


Type = **select**



Random
selection

Type = **fast failover**



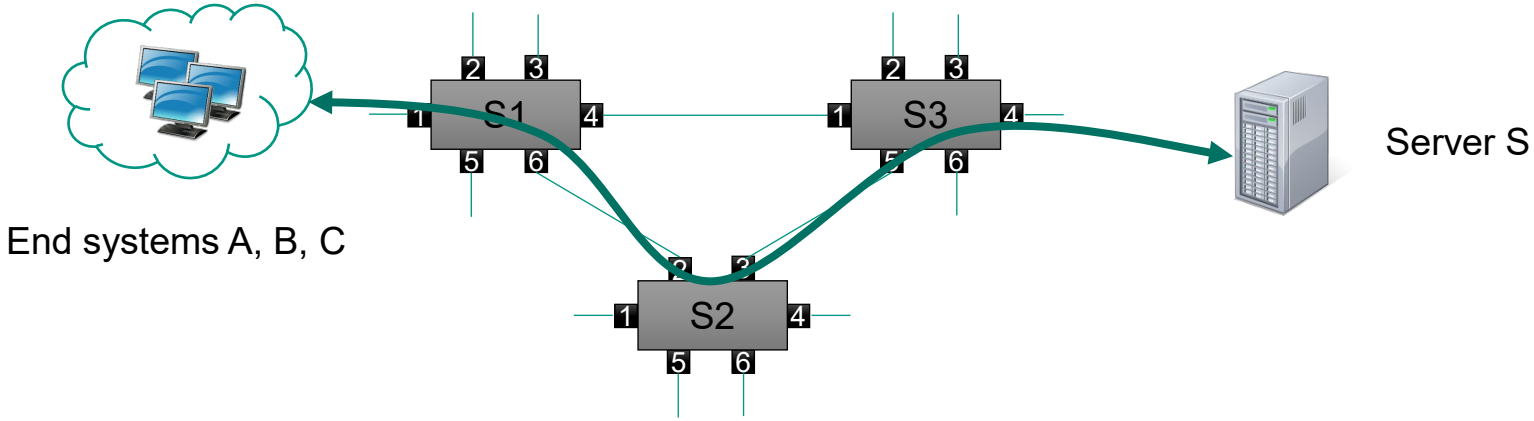
First live
bucket

Group Types

- Group types have specific semantics
  - All: executes all buckets in a group
    - E.g., for broadcast
  - Select: selects one of many buckets of a group
    - E.g., select by round-robin or hashing of packet data
  - Fast failover: executes first live bucket in a group
    - Each bucket is associated with a port that determines its liveliness
      - E.g., a bucket is live when its associated port is up
  - Indirect: executes the single bucket in a group
    - Indirect groups must reference exactly one action bucket
    - Useful to avoid changing multiple flow table entries with common actions

Example: Indirect Group Tables

- Problem
  - Connect a group of end systems (left) to a server (right)
  - Forwarding may change over time → requires modification of flow tables
  - This includes flow table entries in forward and reverse direction



Flow table of S1

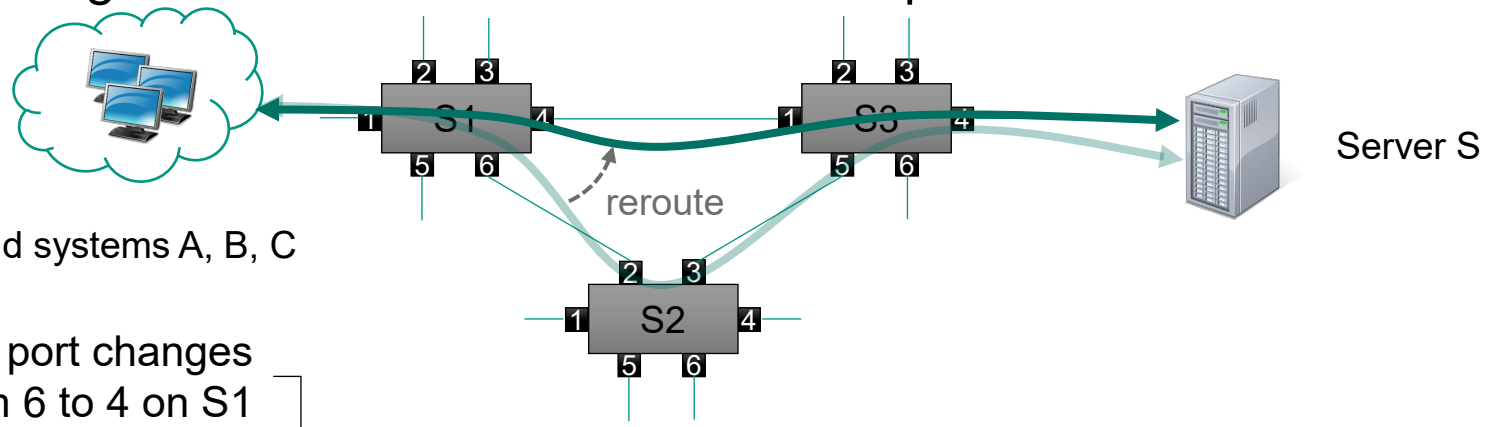
Match	Action
Dst = Server	Output 6
Dst = A	Output 1
Dst = B	Output 1
…	…

Flow table of S3

Match	Action
Dst = Server	Output 4
Dst = A	Output 5
Dst = B	Output 5
…	…

Example: Indirect Group Tables

- Goal: Reroute flows to avoid forwarding via switch S2
  - Output ports specified in flow tables are subject to change
  - SDN controller must send multiple modify-state messages to SDN switches
  - One message for each flow that needs to be updated



Output port changes from 6 to 4 on S1

Flow table of S1

Match	Action
Dst = Server	Output 4
Dst = A	Output 1
Dst = B	Output 1
…	…

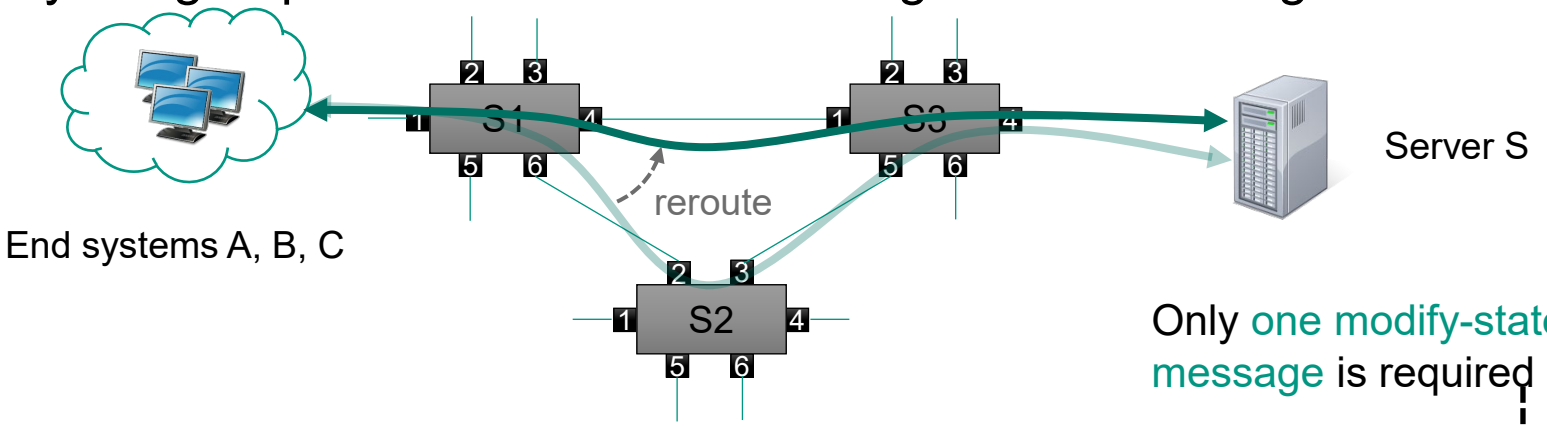
Flow table of S3

Match	Action
Dst = Server	Output 4
Dst = A	Output 1
Dst = B	Output 1
…	…

Output ports of all end systems change from 5 to 1 on S3

Example: Indirect Group Tables

- Optimization
  - Use an indirect group to avoid sending multiple modify-state messages
  - Redirect flows with identical forwarding behavior to that group
  - Modify the groups actions when forwarding behavior changes



Only one modify-state message is required for S3

Flow table of S1

Match	Action
Dst = Server	Output 4
Dst = A	Output 1
Dst = B	Output 1
…	…

Flow table of S3

Match	Action
Dst = Server	Output 4
Dst = A	Group 1
Dst = B	Group 1
…	…

Group table of S3

Group Identifier	Action Buckets
1	Output 1
…	…

Indirection to group 1

Example: Indirect Group Tables

- Using individual flows: proactively program flows in every switch
  - Use one flow for the server and one for each end system

```
import server, hosts, switches

// determines output port for destination host;
// depends on currently active switches
import getOutputPort

onConnect(switch) {
  // program table-miss flow (skipped here)

  // program initial flows
  programFlows(switch)
}

// reroute flows to avoid the specified switch
onRerouteAroundSwitch(switch) {
  // adjust flows in remaining switches
  switches.remove(switch)

  for s in switches {
    programFlows(s)
  }
}
```

```
// program flows in a specified switch
programFlows(switch) {
  // program flows for server
  out = getOutputPort(server, switch)
  r1 = Rule()
  r1.MATCH('IP_DST', server.IP_DST)
  r1.ACTION('OUTPUT', out)
  send_rule(r1, switch)

  // program flows for all hosts
  for host in hosts {
    out = getOutputPort(host, switch)
    r2 = Rule()
    r2.MATCH('IP_DST', host.IP_DST)
    r2.ACTION('OUTPUT', out)
    send_rule(r2, switch)
  }
}
```

reroute_flows.java

Potentially re-programs multiple flows in each switch when flows need to be rerouted

Example: Indirect Group Tables

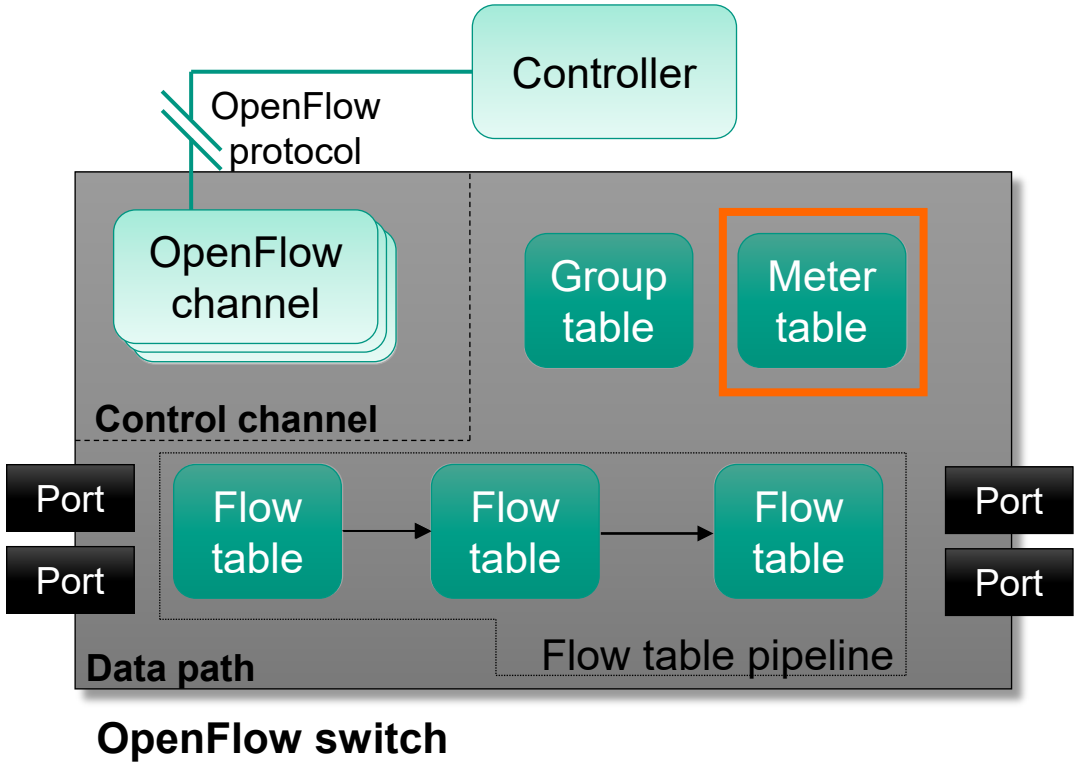
- Using indirection: proactively program flows in every switch
  - Group end systems if they have the same forwarding logic

```
import server, hostGroup, switches

// determines output port for a destination
// host or for a group of hosts with identical
// forwarding behavior;
// depends on currently active switches
import getOutputPort

// creates a new indirect group in the group
// table of a specified switch
import createIndirectGroup

onConnect(switch) {
  // program table-miss flow (skipped here)

  // program initial flows
  programFlows(switch)
}

// reroute flows to avoid the specified switch
onRerouteAroundSwitch(switch) {
  // adjust flows in remaining switches
  switches.remove(switch)

  for s in switches {
    updateGroup(s) // one update per switch
  }
}
```

```
// program flows in a specified switch
programFlows(switch) {
  out = getOutputPort(server, switch)
  r1 = Rule()
  r1.MATCH('IP_DST', server.IP_DST)
  r1.ACTION('OUTPUT', out)
  send_rule(r1, switch)

  // create and program group table
  createIndirectGroup(switch, GROUP_1)
  updateGroup(switch)

  // program indirections for all hosts
  for host in hostGroup {
    r2 = Rule()
    r2.MATCH('IP_DST', host.IP_DST)
    r2.ACTION('GOTO', GROUP_1)
    send_rule(r2, switch)
  }
}

updateGroup(switch) {
  out = getOutputPort(hostGroup, switch)
  r = Rule()
  r.ACTION('OUTPUT', out)
  r.TABLE(GROUP_1)
  send_rule(r, switch)
}
```

indirect_group_table.java

Meter Tables



Meter Table

Meter table entry

Meter identifier	Meter bands	Counters

- Meters measure and control the rate of packets and bytes
  - They are managed in the meter table
  - Each meter has a unique meter identifier
- Meters are invoked from flow table entries through the meter action
  - When invoked, each meter keeps track of the measured rate of packets
- One of several meter bands is triggered when the measured rate exceeds that bands target rate

  → Enabling rate-limiting or QoS policing operations

Meter Bands

- Packet processing by a meter band depends on its band type
  - DSCP remark: implements differentiated services
  - Drop: implements simple rate-limiting
- Rate and burst determine when a band is executed
- Band types may have additional type-specific arguments