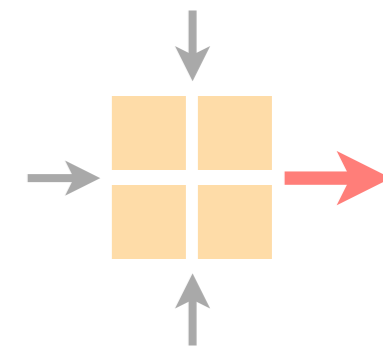


Advanced Topics in Communication Networks

Programming Network Data Planes



Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

24 Sep 2019

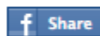
Materials inspired from Jennifer Rexford, Changhoon Kim, and p4.org

Networking is on the verge of a paradigm shift
towards *deep programmability*

Network programmability is attracting tremendous industry interest... (and money)

VMware Acquires Once-Secretive Start-Up Nicira for \$1.26 Billion

JULY 23, 2012 AT 1:25 PM PT



VMware, the software company best known for its virtualization technology that forms the backbones of so-called cloud computing today, said it will pay \$1.26 billion for Nicira, a networking start-up that has sought to do to networks what VMware has done to computers.

The news comes on the same day that VMware was to report quarterly earnings. And while I don't usually cover VMware's

earnings, I may as well mention the results: The company reported revenue for the quarter ended June rose to \$1.12 billion, while earnings on a per-share basis were 68 cents. Analysts had been expecting sales of \$1.12 billion and earnings of 66 cents.

Nicira had been running in stealth mode for quite awhile; [I got to reveal](#) its plans to the world last February.

The deal amounts to a nice payoff for Nicira's investors including Andreessen Horowitz, Lightspeed Venture Partners and NEA, as well as VMware founder Diane Greene and venture capitalist Andy Rachleff.



nicira



With \$600M Invested in SDN Startups, the Ecosystem Builds



Scott Raynovich, June 10, 2014



More than \$600 million has been invested in at least two dozen [software-defined networking \(SDN\)](#) startups so far, according to Rayno Report research. You can expect that to continue to climb. With the SDN ecosystem starting to take hold with a broad range of alliances and distribution partnerships, we're just getting started.

The [Arista IPO](#) will help build visibility for next-generation, software-driven networking. But [Arista](#) is selling its own hardware and is not an SDN pure-play. A new line of [SDN startups](#), with a more radical approach to software-based networking, is building momentum. These newer SDN startups are just getting their gear into customers' hands and starting to build sales channels, so you can expect a long revenue ramp.

This excitement is boosting startup valuations, according to [Rayno Report research](#). There are now at least ten [SDN startups](#) with valuations over \$100 million. As I reported in April, a recent investment in [Cumulus Networks](#) pushed up the valuation of the private company [north of \\$300 million](#), according to industry sources. [Big Switch](#), which did a deal in 2012 valuing it near \$170 million, took money from [Intel](#) in 2013, most likely boosting its valuation to over \$200 million, according to several sources.

Related Articles

[How to Effectively Embed SDN in the Enterprise](#)

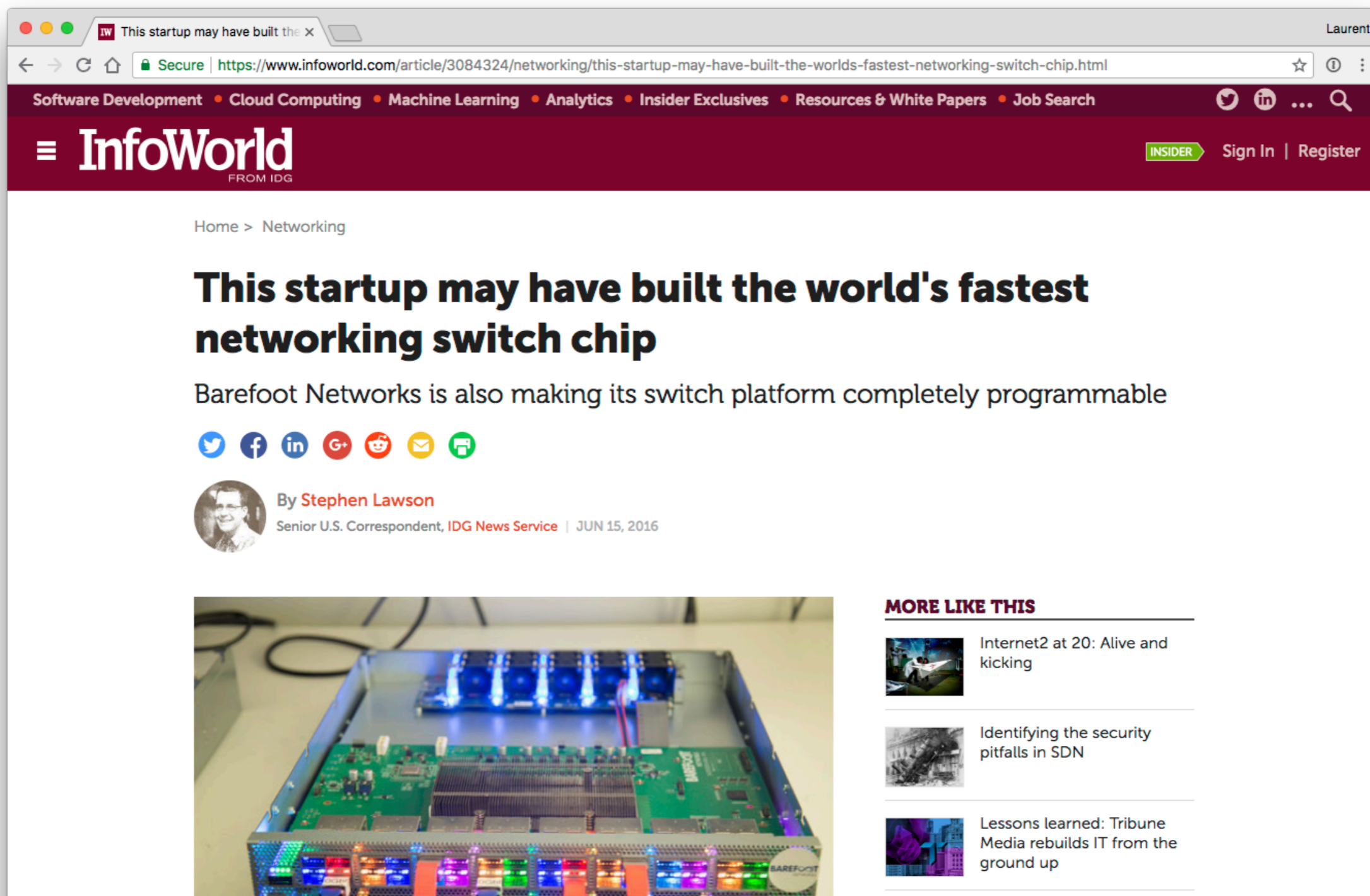
[NFV and SDN: What's the Difference Two Years Later?](#)

[sFlow Creator Peter Phaal On Taming The Wilds Of SDN & Virtual Networking](#)

[Featured Article: Bringing Data-Driven SDN to the Network Edge](#)

[NFV Delivers Pervasive Intelligence for MNOs](#)

Barefoot Networks (Stanford startup) started to produce re-programmable network hardware *in 2013*



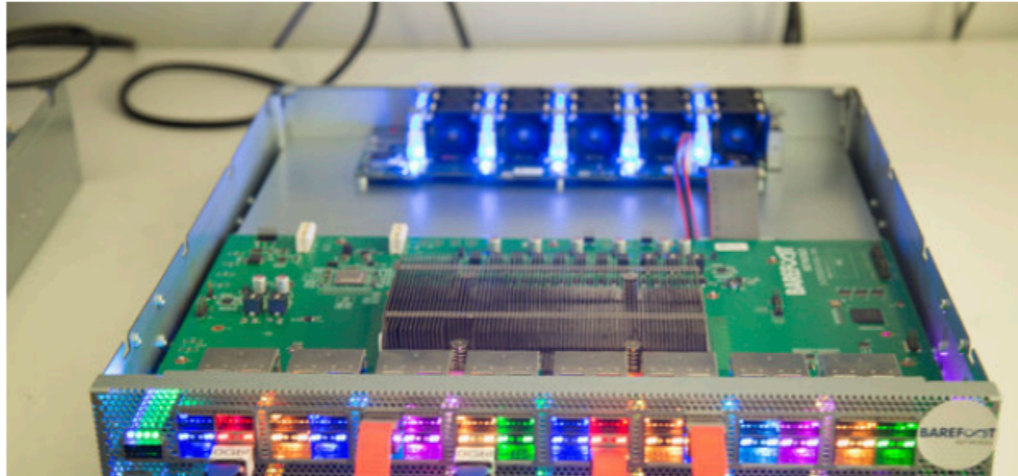
The screenshot shows a web browser window displaying an article on InfoWorld. The browser's address bar shows the URL: <https://www.infoworld.com/article/3084324/networking/this-startup-may-have-built-the-worlds-fastest-networking-switch-chip.html>. The page header includes navigation links for Software Development, Cloud Computing, Machine Learning, Analytics, Insider Exclusives, Resources & White Papers, and Job Search. The InfoWorld logo is prominently displayed, along with a green 'INSIDER' badge and links for Sign In and Register. The article title is 'This startup may have built the world's fastest networking switch chip', and the sub-headline reads 'Barefoot Networks is also making its switch platform completely programmable'. The author is identified as Stephen Lawson, Senior U.S. Correspondent for IDG News Service, with a publication date of June 15, 2016. Below the text is a photograph of a Barefoot Networks switch hardware unit, showing its internal components and a front panel with numerous ports and indicator lights. To the right of the main content, a 'MORE LIKE THIS' section lists three related articles: 'Internet2 at 20: Alive and kicking', 'Identifying the security pitfalls in SDN', and 'Lessons learned: Tribune Media rebuilds IT from the ground up'.

Home > Networking

This startup may have built the world's fastest networking switch chip

Barefoot Networks is also making its switch platform completely programmable

By **Stephen Lawson**
Senior U.S. Correspondent, IDG News Service | JUN 15, 2016



MORE LIKE THIS

- Internet2 at 20: Alive and kicking
- Identifying the security pitfalls in SDN
- Lessons learned: Tribune Media rebuilds IT from the ground up

In *June 2019*,
Barefoot was acquired by... Intel

THE WALL STREET JOURNAL.

Europe Edition | September 22, 2019 | Print Edition | Video

Home World U.S. Politics Economy Business **Tech** Markets Opinion Life & Arts Real Estate WSJ Magazine



SHARE

TECH

Intel Agrees to Acquire Networking Startup Barefoot Networks

Barefoot Networks is backed by Google, Alibaba, Tencent and Goldman Sachs

AA
TEXT



Network programmability is also getting traction in many academic communities

Networking

SIGCOMM

NSDI

HotNets

CoNEXT

Systems

OSDI

SOSP

SOCC

Distributed
Algorithms

PODC

DISC

Security

CCS

NDSS

Usenix
Security

S&P

PL

PLDI

POPL

OOPSLA

>8.7k

of citations of the original
OpenFlow paper (*) in ~10 years

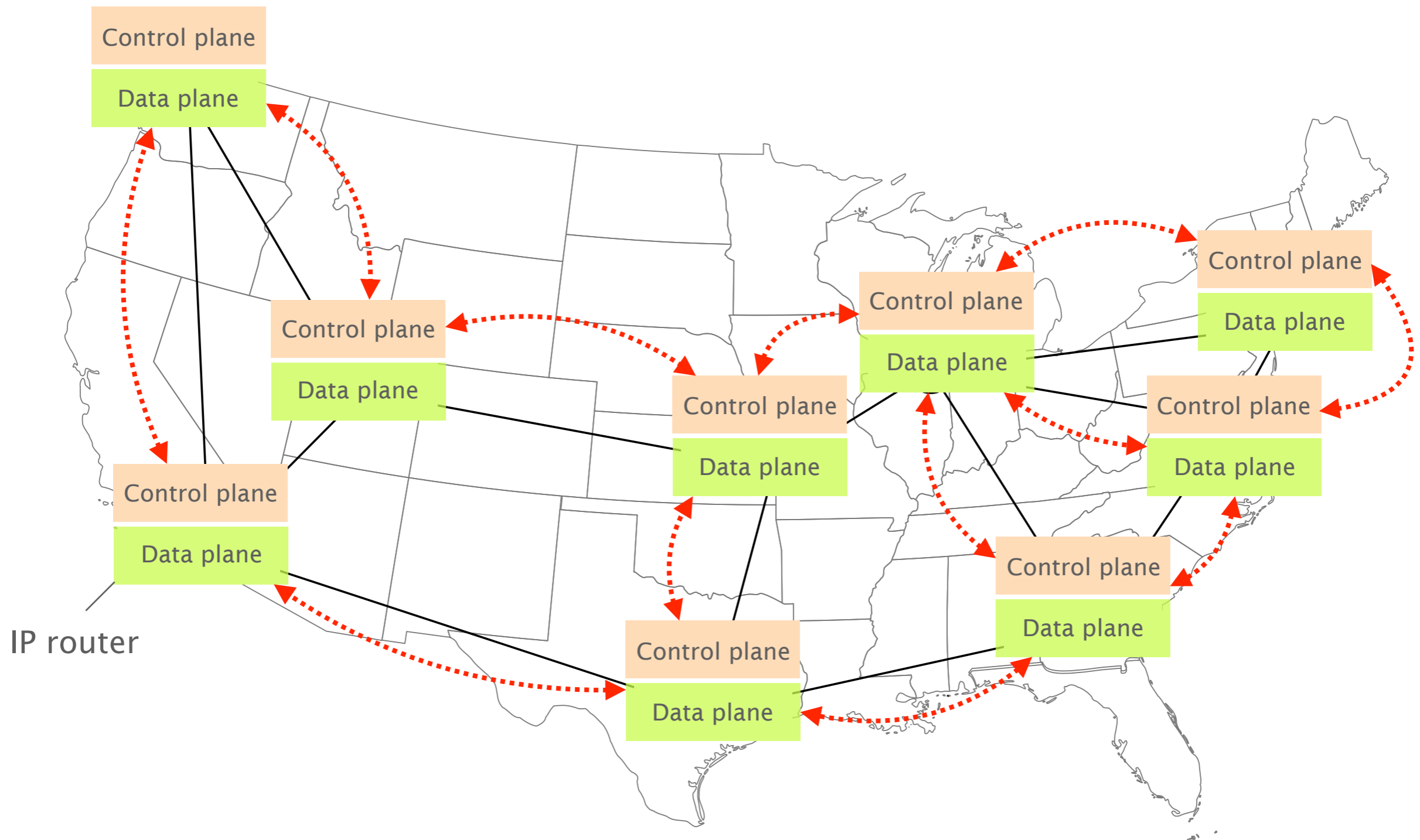
(*) <https://dl.acm.org/citation.cfm?id=1355746>

Why? It's really a story in 3 stages

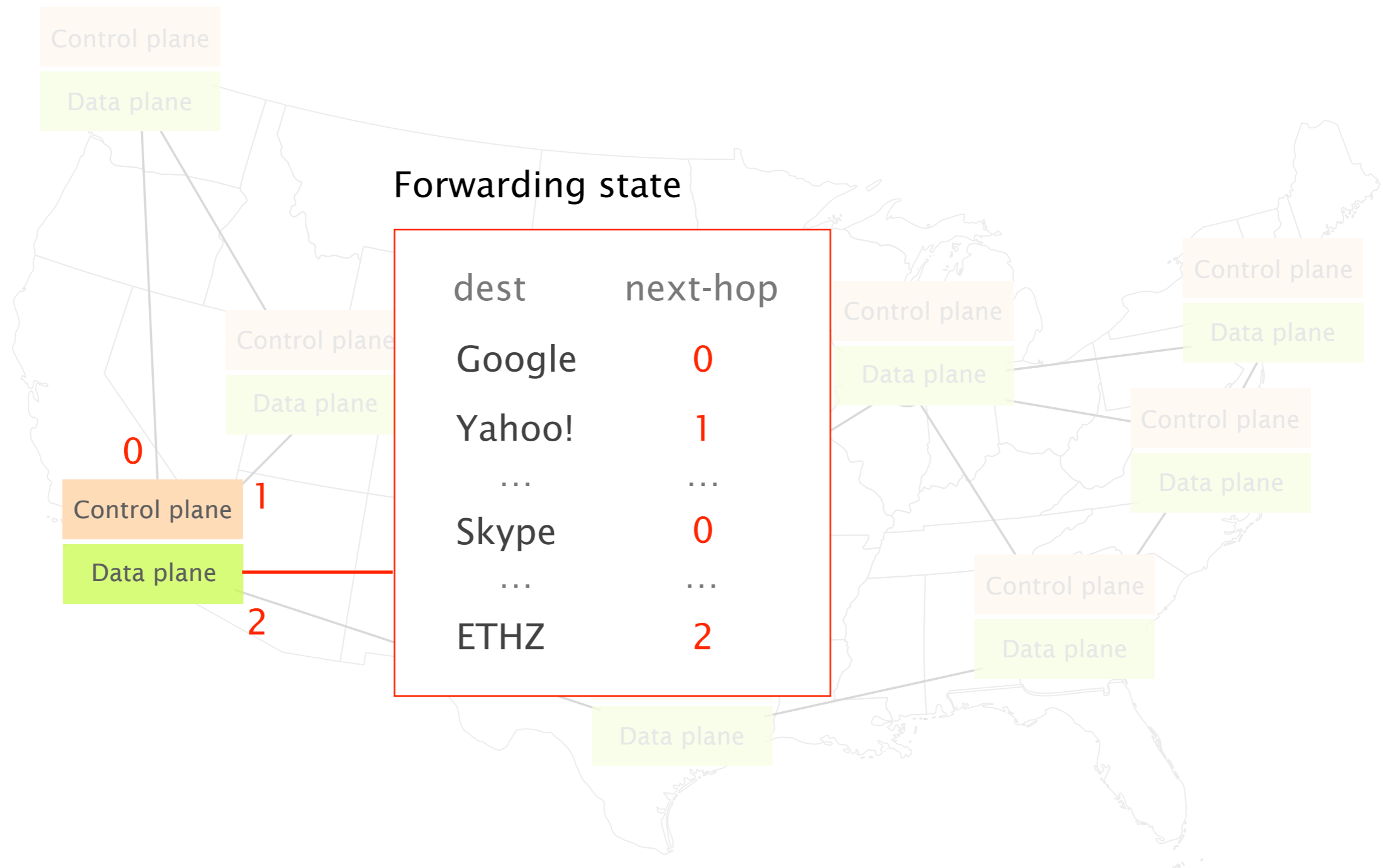
Stage 1

The network management crisis

Networks are large distributed systems running a set of distributed algorithms

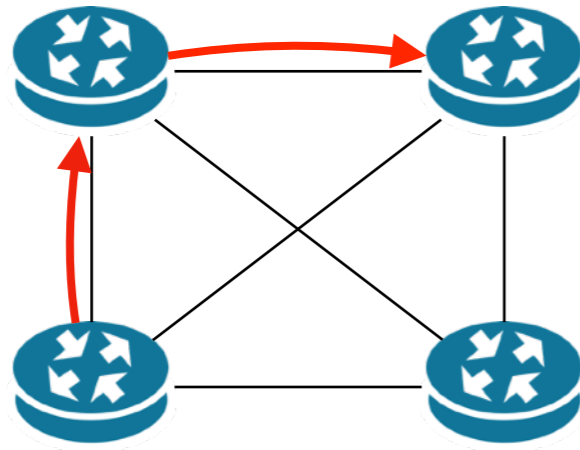


These algorithms produce the forwarding state which drives IP traffic to its destination

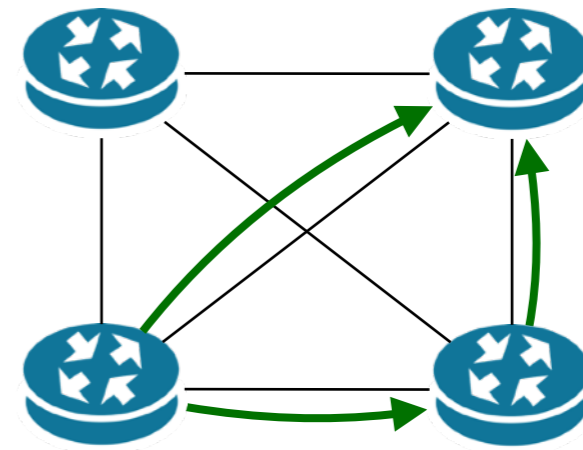


Operators adapt their network forwarding behavior
by configuring each network device individually

Given



and

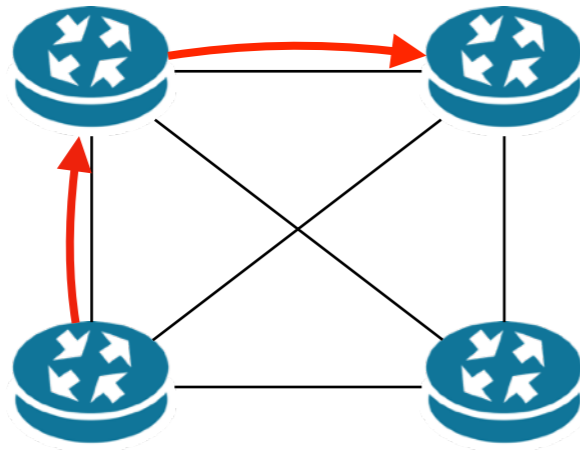


an **existing** network behavior
induced by a low-level configuration C

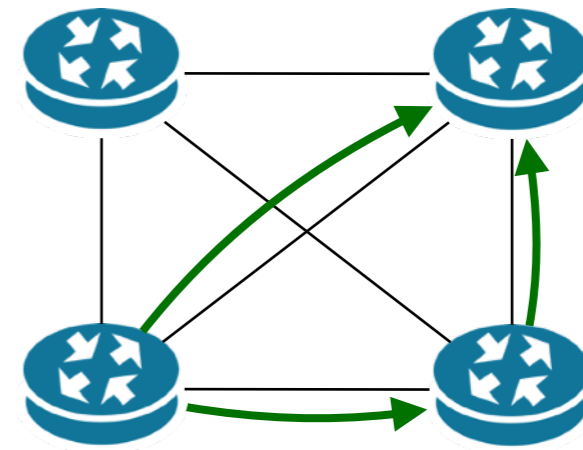
a **desired** network behavior

Adapt C so that the network follows the new behavior

Given



and



an **existing** network behavior
induced by a low-level configuration C

a **desired** network behavior

Adapt C so that the network follows the new behavior

Configuring each element is often done manually, using arcane low-level, vendor-specific “languages”

Cisco IOS

```
!  
ip multicast-routing  
!  
interface Loopback0  
 ip address 120.1.7.7 255.255.255.255  
 ip ospf 1 area 0  
!  
!  
interface Ethernet0/0  
 no ip address  
!  
interface Ethernet0/0.17  
 encapsulation dot1Q 17  
 ip address 125.1.17.7 255.255.255.0  
 ip pim bsr-border  
 ip pim sparse-mode  
!  
!  
router ospf 1  
 router-id 120.1.7.7  
 redistribute bgp 700 subnets  
!  
router bgp 700  
 neighbor 125.1.17.1 remote-as 100  
!  
 address-family ipv4  
  redistribute ospf 1 match internal external 1 external 2  
  neighbor 125.1.17.1 activate  
!  
 address-family ipv4 multicast  
  network 125.1.79.0 mask 255.255.255.0  
  redistribute ospf 1 match internal external 1 external 2
```

Juniper JunOS

```
interfaces {  
  so-0/0/0 {  
    unit 0 {  
      family inet {  
        address 10.12.1.2/24;  
      }  
      family mpls;  
    }  
  }  
  ge-0/1/0 {  
    vlan-tagging;  
    unit 0 {  
      vlan-id 100;  
      family inet {  
        address 10.108.1.1/24;  
      }  
      family mpls;  
    }  
    unit 1 {  
      vlan-id 200;  
      family inet {  
        address 10.208.1.1/24;  
      }  
    }  
  }  
  ...  
}  
protocols {  
  mpls {  
    interface all;  
  }  
  bgp {
```

A single mistyped line is enough to bring down the entire network

Cisco IOS

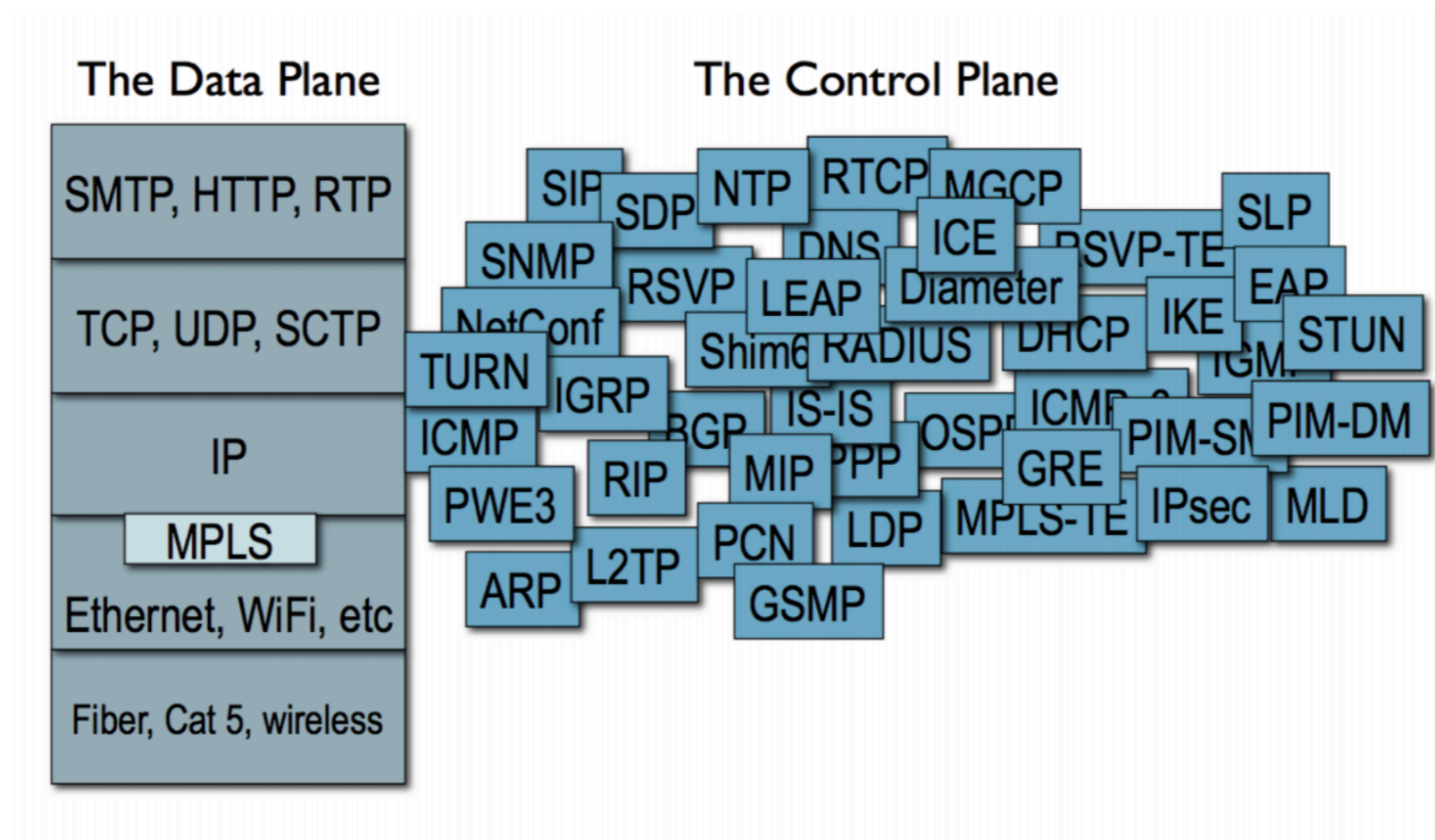
```
!  
ip multicast-routing  
!  
interface Loopback0  
 ip address 120.1.7.7 255.255.255.255  
 ip ospf 1 area 0  
!  
!  
interface Ethernet0/0  
 no ip address  
!  
interface Ethernet0/0.17  
 encapsulation dot1Q 17  
 ip address 125.1.17.7 255.255.255.0  
 ip pim bsr-border  
 ip pim sparse-mode  
!  
!  
router ospf 1  
 router-id 120.1.7.7  
 redistribute bgp 700 subnets  
!  
router bgp 700  
 neighbor 125.1.17.1 remote-as 100  
!  
 address-family ipv4  
  redistribute ospf 1 match internal external 1 external 2  
  neighbor 125.1.17.1 activate  
!  
 address-family ipv4 multicast  
  network 125.179.0 mask 255.255.255.0  
  redistribute ospf 1 match internal external 1 external 2
```

Juniper JunOS

```
interfaces {  
  so-0/0/0 {  
    unit 0 {  
      family inet {  
        address 10.12.1.2/24;  
      }  
      family mpls;  
    }  
  }  
  ge-0/1/0 {  
    vlan-tagging;  
    unit 0 {  
      vlan-id 100;  
      family inet {  
        address 10.108.1.1/24;  
      }  
      family mpls;  
    }  
    unit 1 {  
      vlan-id 200;  
      family inet {  
        address 10.208.1.1/24;  
      }  
    }  
  }  
  ...  
}  
protocols {  
  mpls {  
    interface all;  
  }  
  bgp {
```

Anything else than 700 creates blackholes

It's not only about the problem of configuring...
the level of complexity in networks is staggering



Source

Mark Handley. Re-thinking the control architecture of the internet. Keynote talk. REARCH. December 2009.

Complexity + Low-level Management = **Problems**

Google accidentally broke the internet throughout Japan

A mistake led to internet outages for about half of the country.



Mallory Locklear, @mallorylocklear
08.28.17 in [Internet](#)



Someone in Google fat-thumbbed a Border Gateway Protocol (BGP) advertisement and sent Japanese Internet traffic into a black hole.

[...] the result of which was traffic from Japanese giants like NTT and KDDI was sent to Google on the expectation it would be treated as transit.

The outage in Japan **only lasted a couple of hours**, but was so severe that [...] the country's Internal Affairs and Communications ministries want carriers to report on what went wrong.

JUL 8, 2015 @ 03:36 PM 11,261 VIEWS

United Airlines Blames Router for Grounded Flights

'Configuration Error' Blamed for AWS Outage

By David Ramel ■ 08/12/2015



The summer of network misconfigurations



Amazon's massive AWS outage was caused by human error

One incorrect command and the whole internet suffers.

By Jason Del Rey | @DelRey | Mar 2, 2017, 2:20pm EST

Data Centre ► Networks

Level3 switch config blunder for US-wide VoIP blackout

CenturyLink: 750 calls to 911 missed during Aug. 1 outage caused by human error in Minnesota, North Dakota

By Barry Amundson on Aug 15, 2018 at 4:43 p.m.

affected Comcast, Spectrum, Verizon and AT&T customers

BY: CNN
POSTED: 1:42 PM Nov 6, 2017

Data Centre ► Networks

CloudFlare apologizes for Telia screwing you over

Unhappy about

By Kieren McCarthy in

Facebook struggles to deal with epic outage



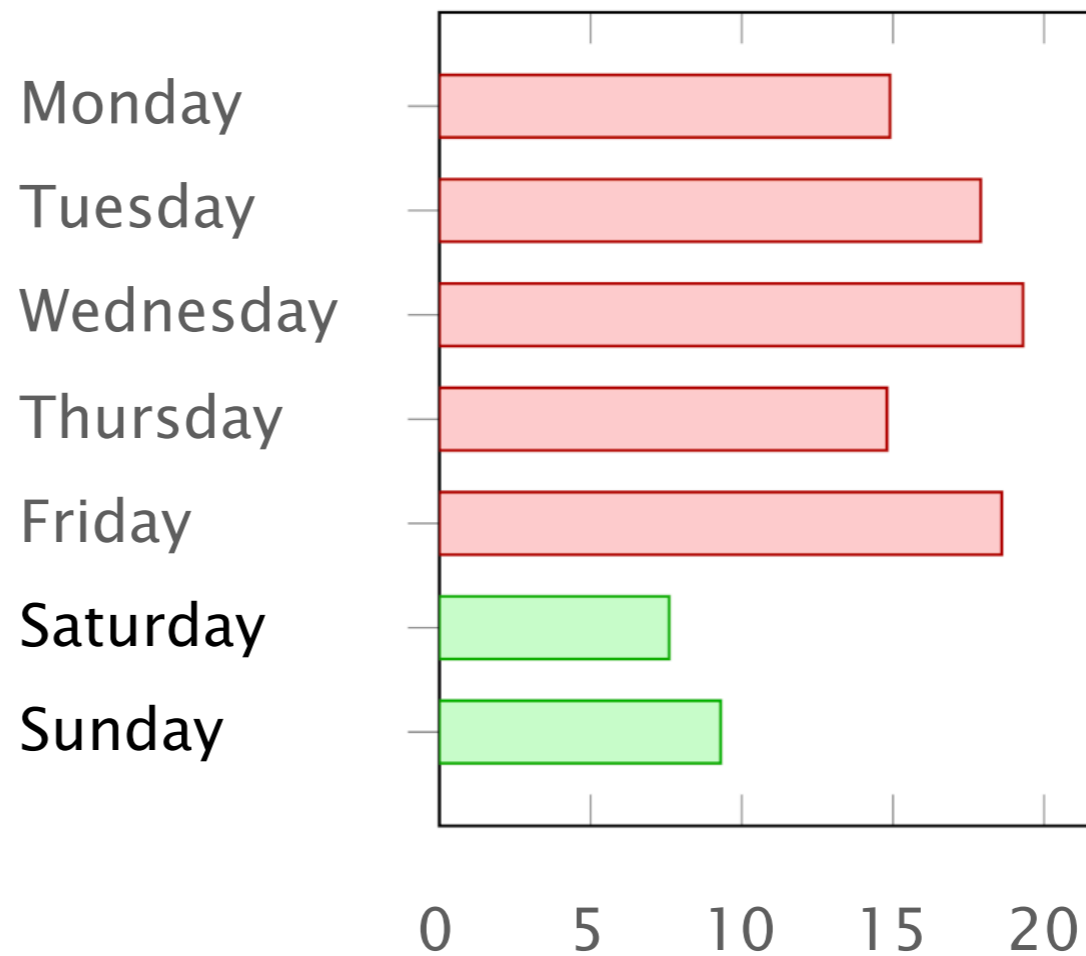
By Donie O'Sullivan and Heather Kelly, CNN Business

Updated 0654 GMT (1454 HKT) March 14, 2019

“Human factors are responsible
for 50% to 80% of network outages”

Juniper Networks, *What's Behind Network Downtime?*, 2008

Ironically, this means that
data networks work better during week-ends...



% of route leaks

source: Job Snijders (NTT)

The Internet Under Crisis Conditions

Learning from September 11

Committee on the Internet Under Crisis Conditions:
Learning from September 11

Computer Science and Telecommunications Board
Division on Engineering and Physical Sciences

NATIONAL RESEARCH COUNCIL
OF THE NATIONAL ACADEMIES

The Internet Under Crisis Conditions

Learning from September 11

Committee on the Internet Under Crisis Conditions:
Learning from September 11

Computer Science and Telecommunications Board
Division on Engineering and Physical Sciences

NATIONAL RESEARCH COUNCIL
OF THE NATIONAL ACADEMIES

Internet advertisements rates
suggest that

The Internet was **more stable
than normal on Sept 11**

The Internet Under Crisis Conditions

Learning from September 11

Committee on the Internet Under Crisis Conditions:
Learning from September 11

Computer Science and Telecommunications Board
Division on Engineering and Physical Sciences

NATIONAL RESEARCH COUNCIL
OF THE NATIONAL ACADEMIES

Internet advertisements rates
suggest that

The Internet was **more stable
than normal on Sept 11**

Information suggests that
operators were **watching the news
instead of making changes**
to their infrastructure

Solving these problems used to be hard because network devices tend to be completely locked down



closed software

closed hardware

Cisco™ device

Stage 2

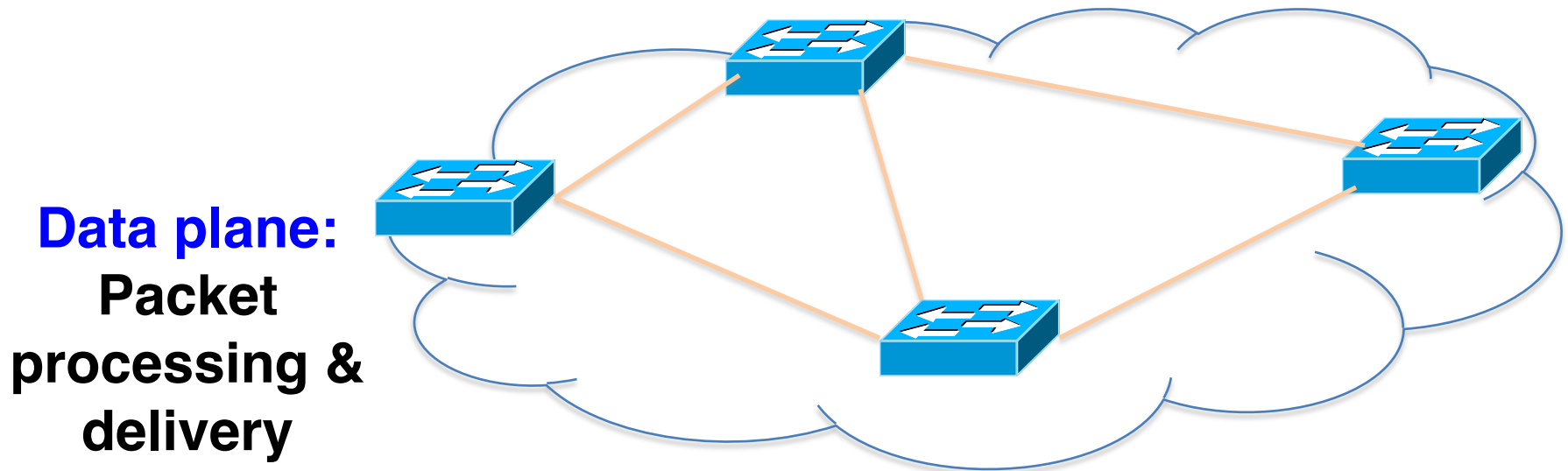
Software-Defined Networking

What is SDN and how does it help?

- SDN is a new approach to networking
 - Not about “architecture”: IP, TCP, etc.
 - But about design of network control (routing, TE,...)
- SDN is predicated around two simple concepts
 - Separates the control-plane from the data-plane
 - Provides open API to directly access the data-plane
- While SDN doesn't do much, it enables *a lot*

Rethinking the “Division of Labor”

Traditional Computer Networks

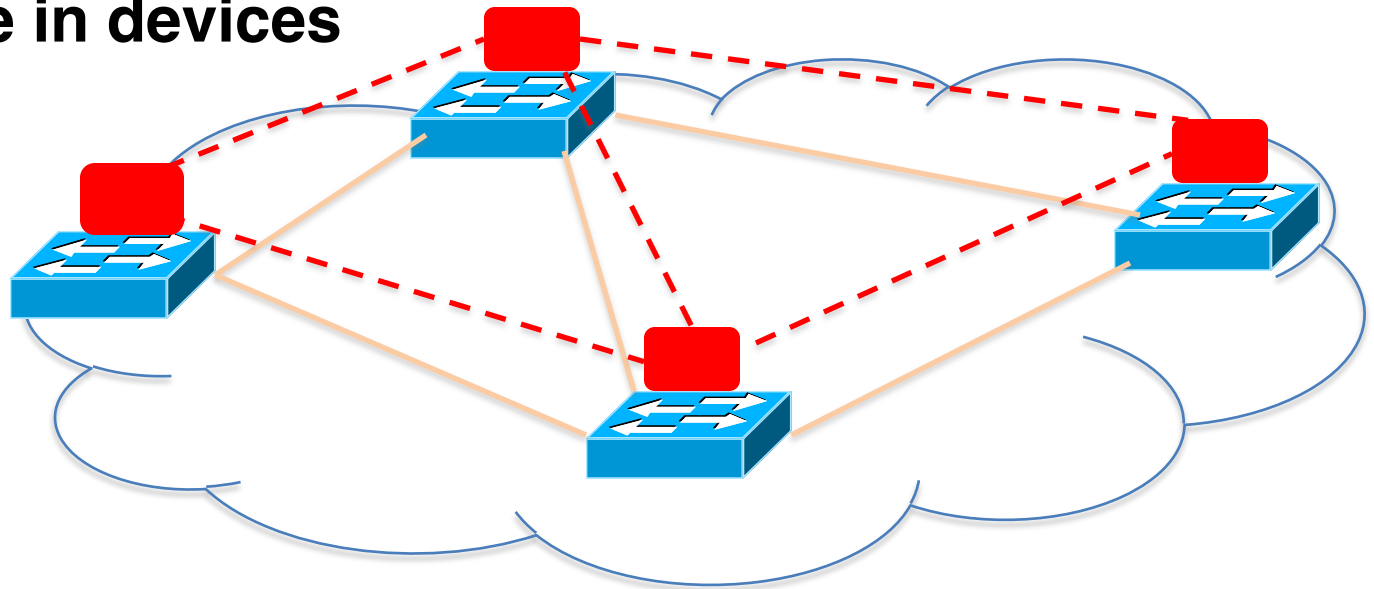


**Forward, filter, buffer, mark,
rate-limit, and measure packets**

Traditional Computer Networks

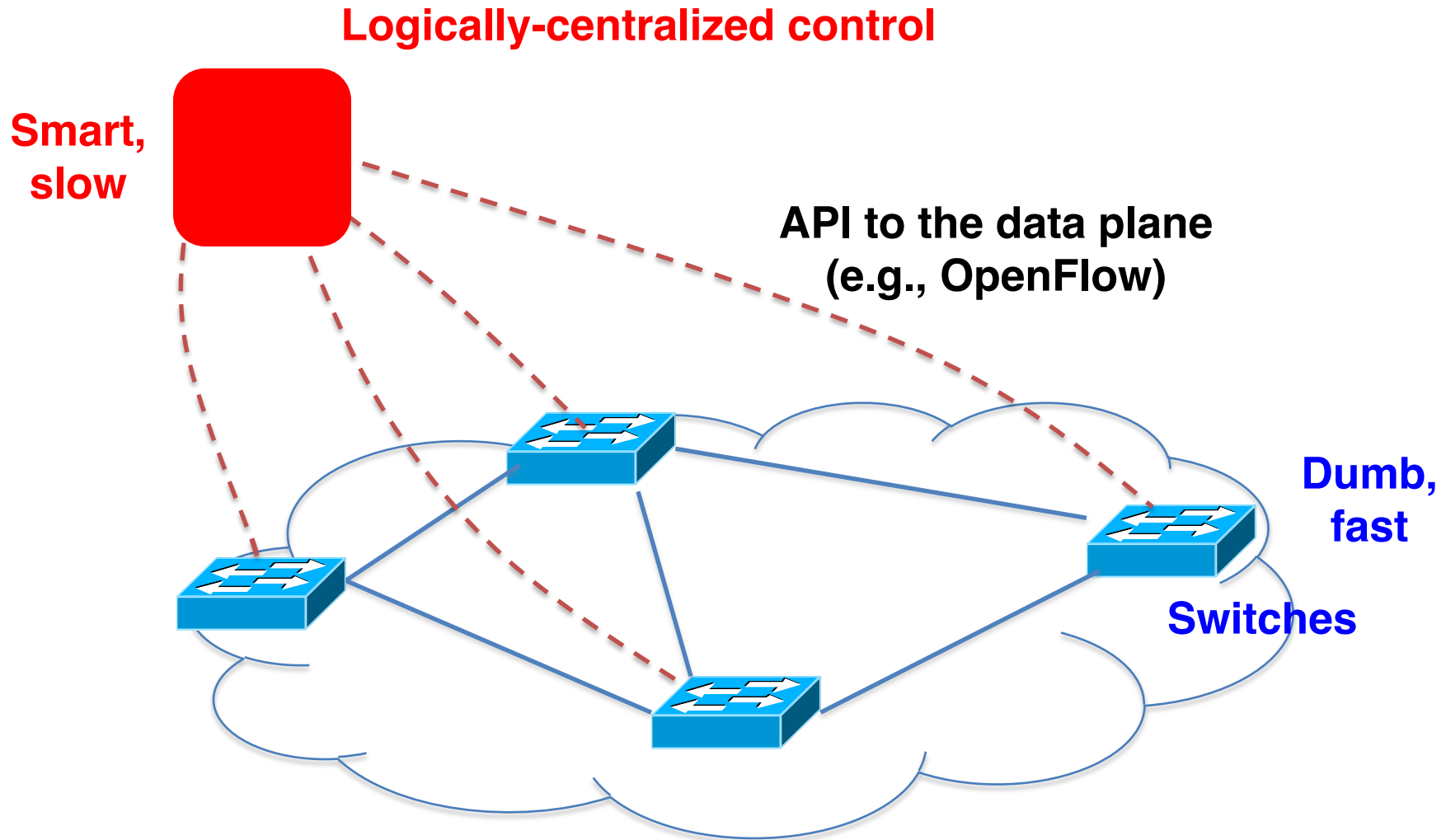
Control plane:

Distributed algorithms,
establish state in devices



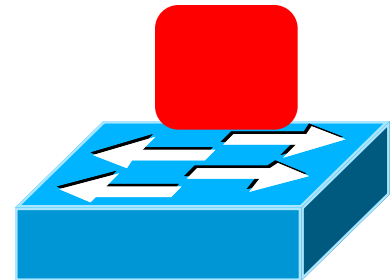
Track topology changes, compute
routes, install forwarding rules

Software Defined Networking (SDN)



SDN advantages

- **Simpler management**
 - No need to “invert” control-plane operations
- **Faster pace of innovation**
 - Less dependence on vendors and standards
- **Easier interoperability**
 - Compatibility only in “wire” protocols
- **Simpler, cheaper equipment**
 - Minimal software



OpenFlow Networks

OpenFlow is an API to a switch flow table

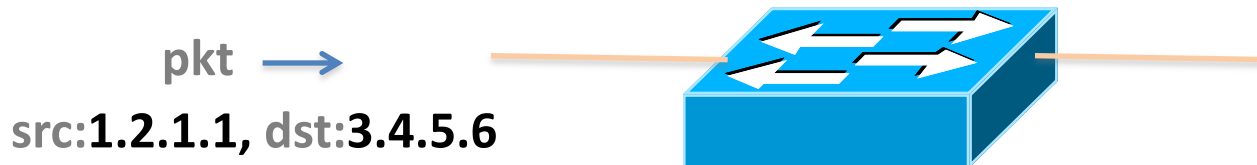
- **Simple packet-handling rules**
 - Pattern: match packet header bits, i.e. flow space
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



```
10. src=1.2.*.* , dest=3.4.5.* → drop
05. src = *.*.*.* , dest=3.4.*.* → forward(2)
01. src=10.1.2.3, dest=*.*.*.* → send to controller
```

OpenFlow is an API to a switch flow table

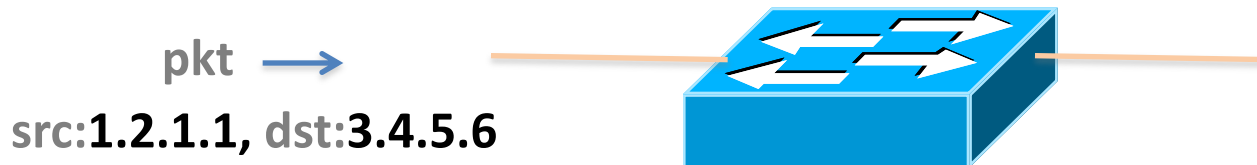
- **Simple packet-handling rules**
 - Pattern: match packet header bits, i.e. flow space
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



```
10. src=1.2.*.* , dest=3.4.5.* → drop
05. src = *.*.*.* , dest=3.4.*.* → forward(2)
01. src=10.1.2.3, dest=*.*.*.* → send to controller
```

OpenFlow is an API to a switch flow table

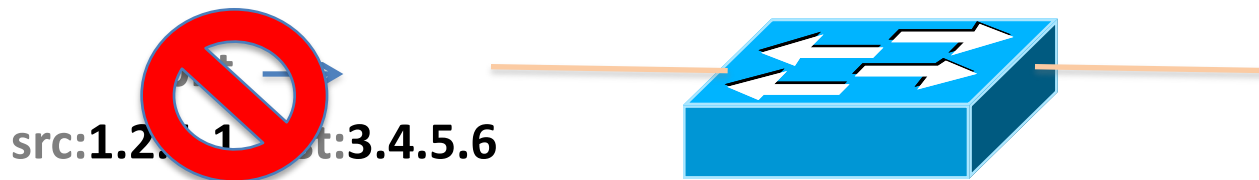
- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flow space
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



10. **src=1.2.*.* , dest=3.4.5.*** → drop
05. src = *.*.*.* , dest=3.4.*.* → forward(2)
01. src=10.1.2.3, dest=*.*.*.* → send to controller

OpenFlow is an API to a switch flow table

- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flow space
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



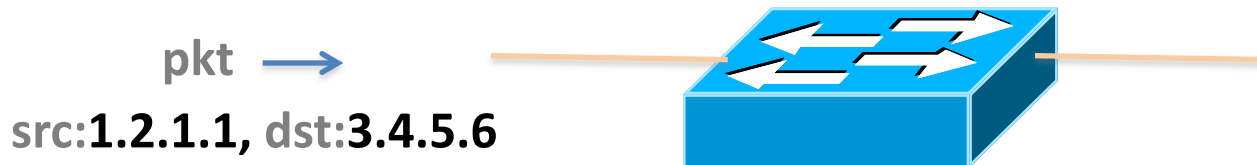
10. src=1.2.*.* , dest=3.4.5.* → drop

05. src = *.*.*.* , dest=3.4.*.* → forward(2)

01. src=10.1.2.3, dest=*.*.* → send to controller

OpenFlow is an API to a switch flow table

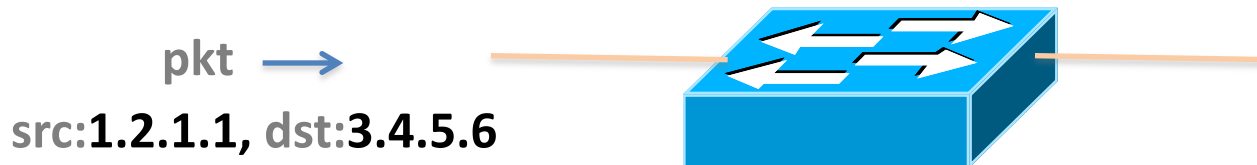
- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flow space
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



```
10. src=1.2.*.* , dest=3.4.5.* → drop
05. src = *.*.*.* , dest=3.4.*.* → forward(2)
01. src=10.1.2.3, dest=*.*.*.* → send to controller
```


OpenFlow is an API to a switch flow table

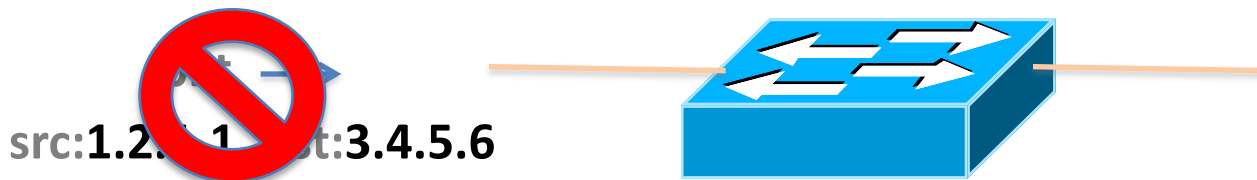
- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flow space
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



10. **src=1.2.*.***, **dest=3.4.5.*** → drop
05. **src = *.*.*.***, **dest=3.4.*.*** → forward(2)
01. **src=10.1.2.3**, **dest=*.*.*.*** → send to controller

OpenFlow is an API to a switch flow table

- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flow space
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



10. src=1.2.*.*, dest=3.4.5.* → **drop**

05. src = *.*.*.*, dest=3.4.*.* → forward(2)

01. src=10.1.2.3, dest=*.*.* → send to controller

OpenFlow switches can emulate different kinds of boxes

- Router

- Match: longest destination IP prefix
- Action: forward out a link

- Switch

- Match: destination MAC address
- Action: forward or flood

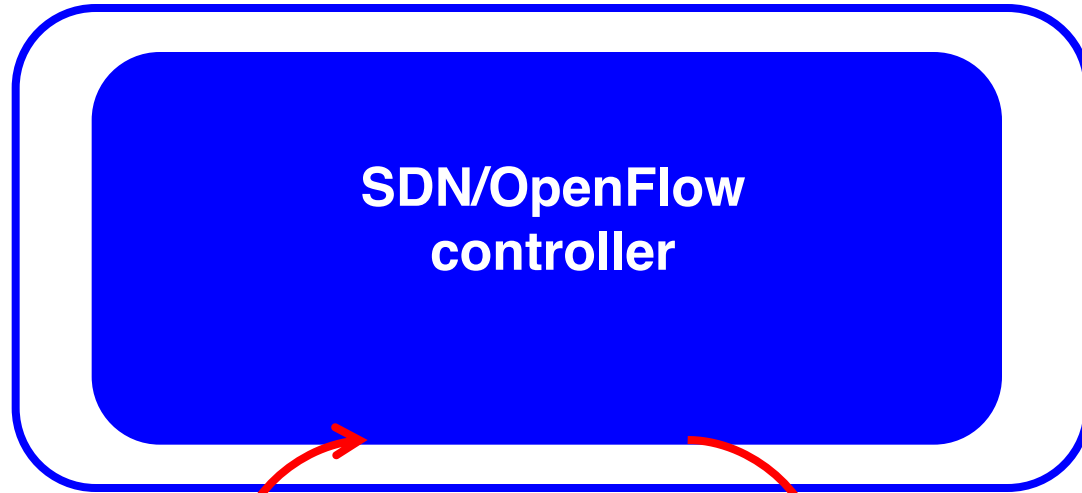
- Firewall

- Match: IP addresses and TCP/UDP port numbers
- Action: permit or deny

- NAT

- Match: IP address and port
- Action: rewrite address and port

Controller: Programmability



Receives events from switches

Topology changes,
Traffic statistics,
Arriving packets

Send commands to switches

(Un)install rules,
Query statistics,
Send packets

Controller: Programmability

```
while (true):  
  read event e:  
    if e == switch up:  
      - update topology  
      - populates switch table  
    ...
```

Receives events from switches

Topology changes,
Traffic statistics,
Arriving packets

Send commands to switches

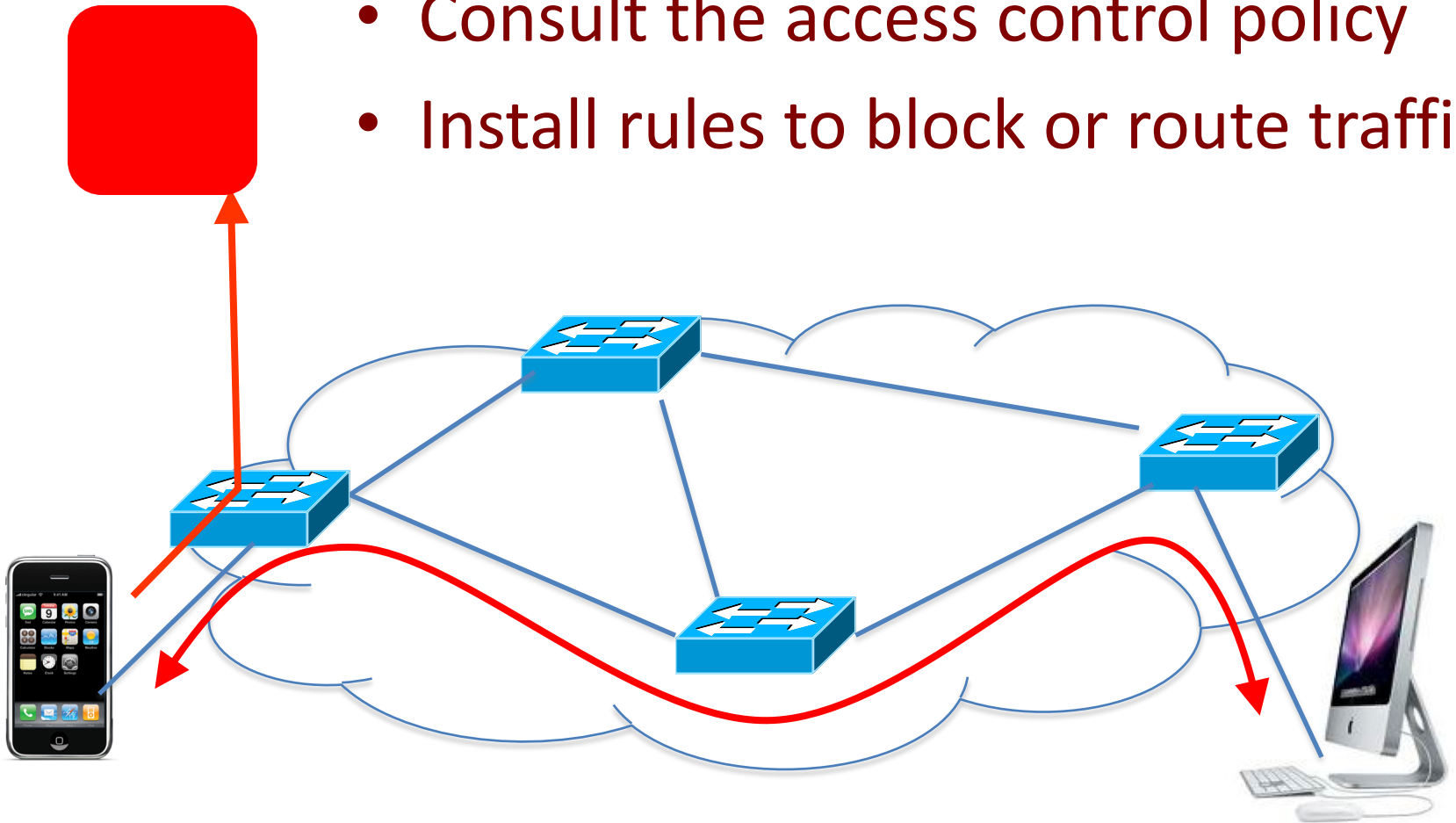
(Un)install rules,
Query statistics,
Send packets

Example OpenFlow Applications

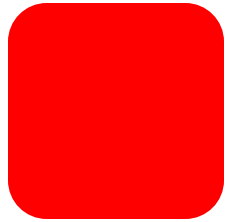
- **Dynamic access control**
- **Seamless mobility/migration**
- **Server load balancing**
- Network virtualization
- Using multiple wireless access points
- Energy-efficient networking
- Adaptive traffic monitoring
- Denial-of-Service attack detection

E.g.: Dynamic Access Control

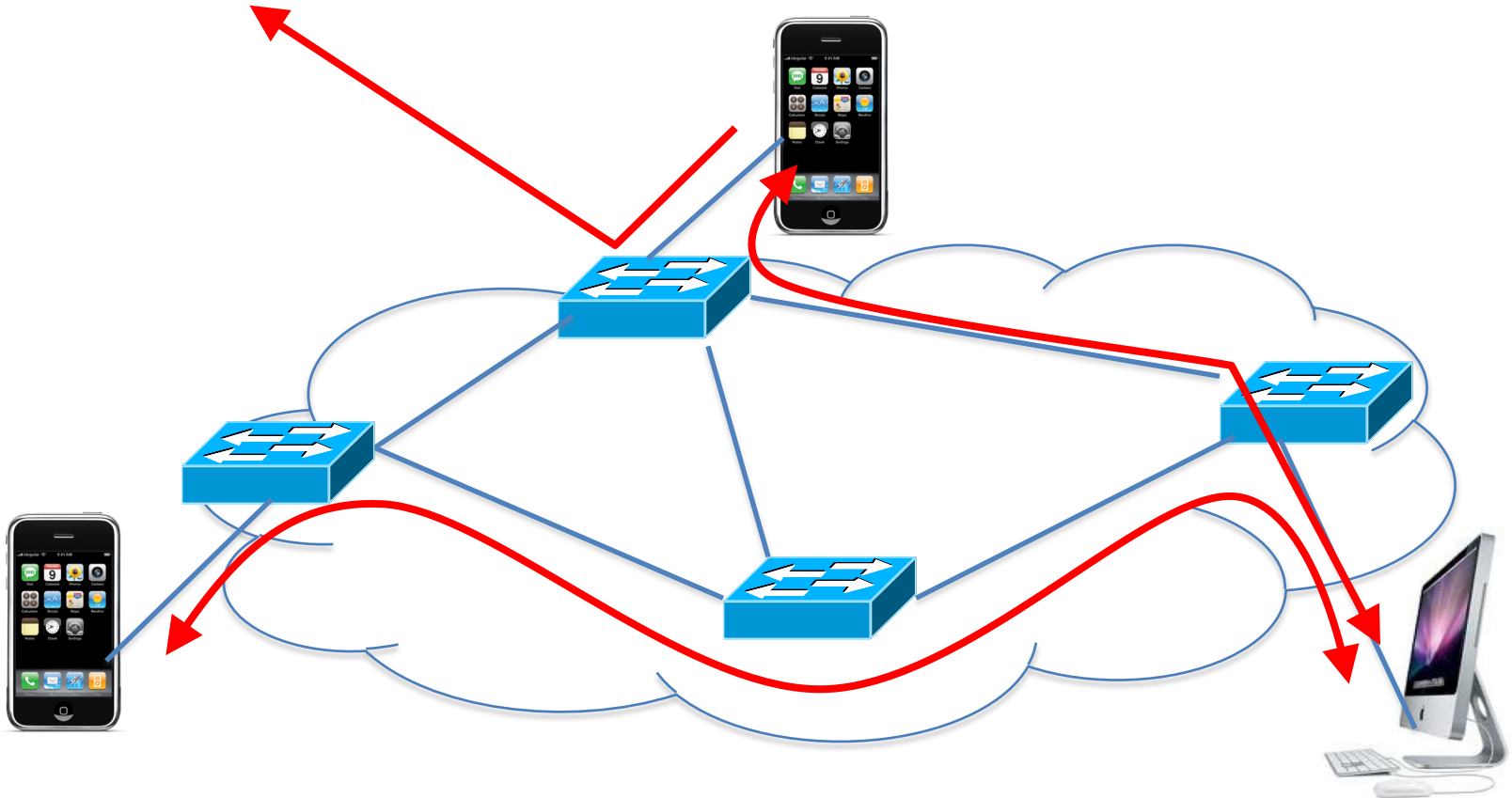
- Inspect first packet of a connection
- Consult the access control policy
- Install rules to block or route traffic



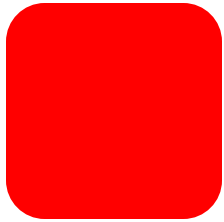
E.g.: Seamless Mobility/Migration



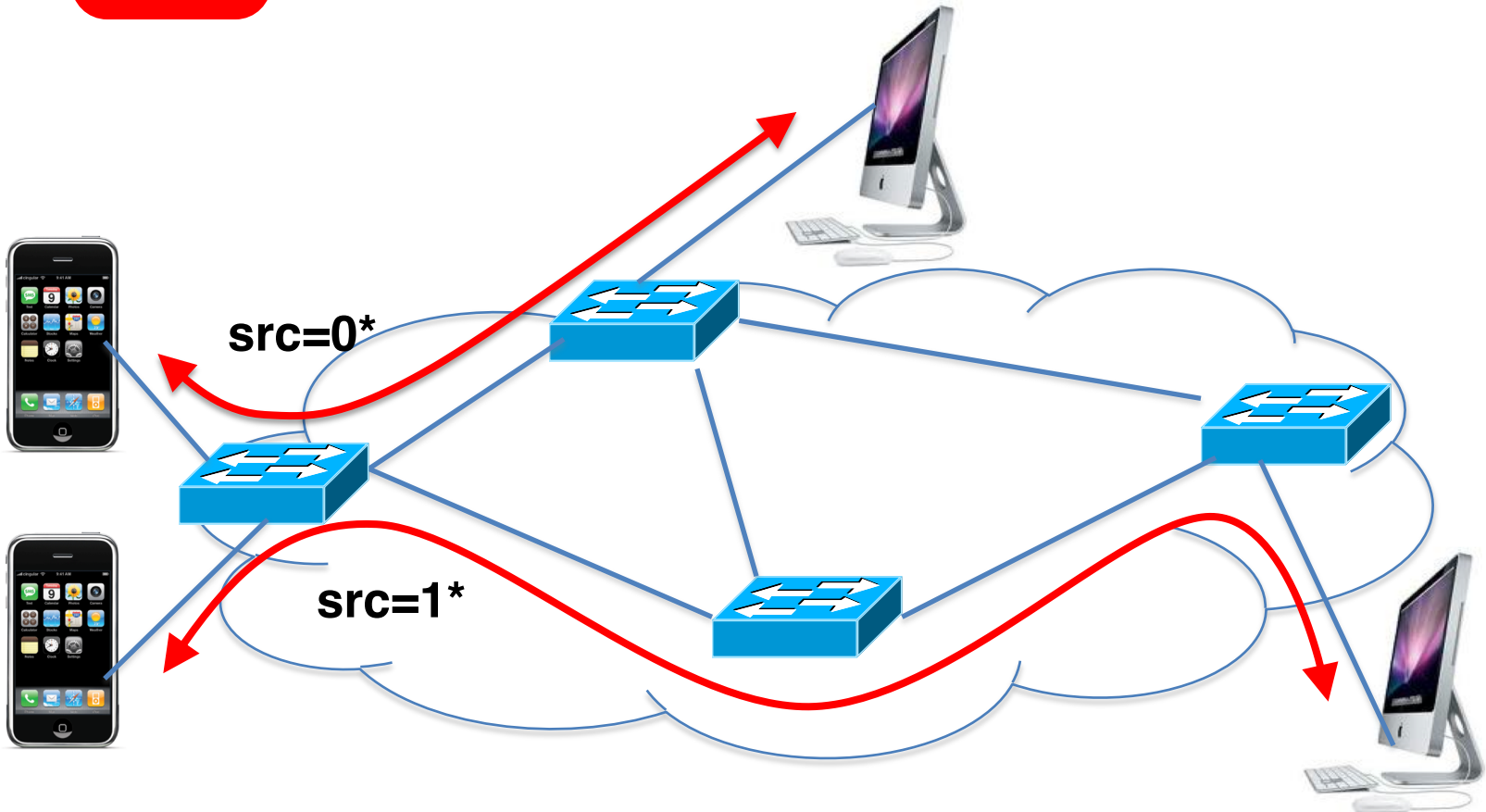
- See host send traffic at new location
- Modify rules to reroute the traffic



E.g.: Server Load Balancing



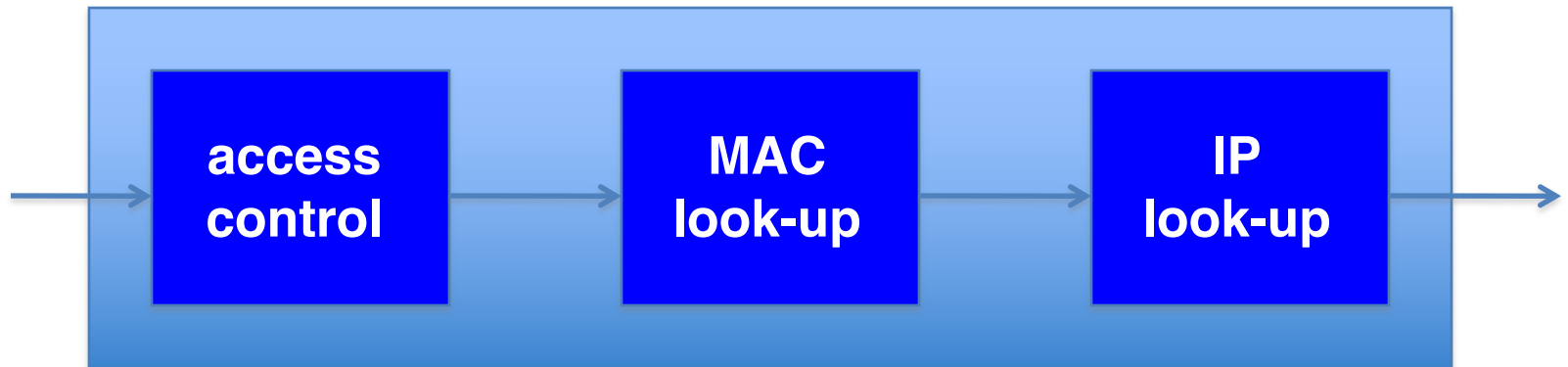
- Pre-install load-balancing policy
- Split traffic based on source IP



Challenges

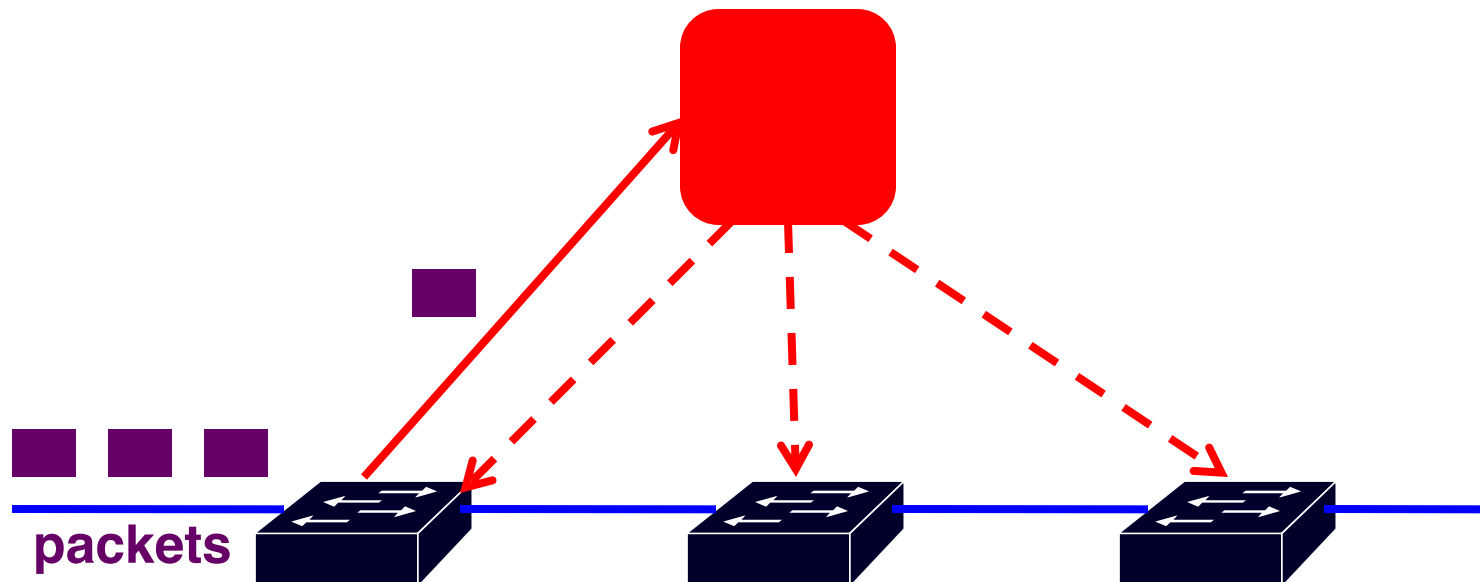
Heterogeneous Switches

- Number of packet-handling rules
- Range of matches and actions
- Multi-stage pipeline of packet processing
- Offload some control-plane functionality (?)

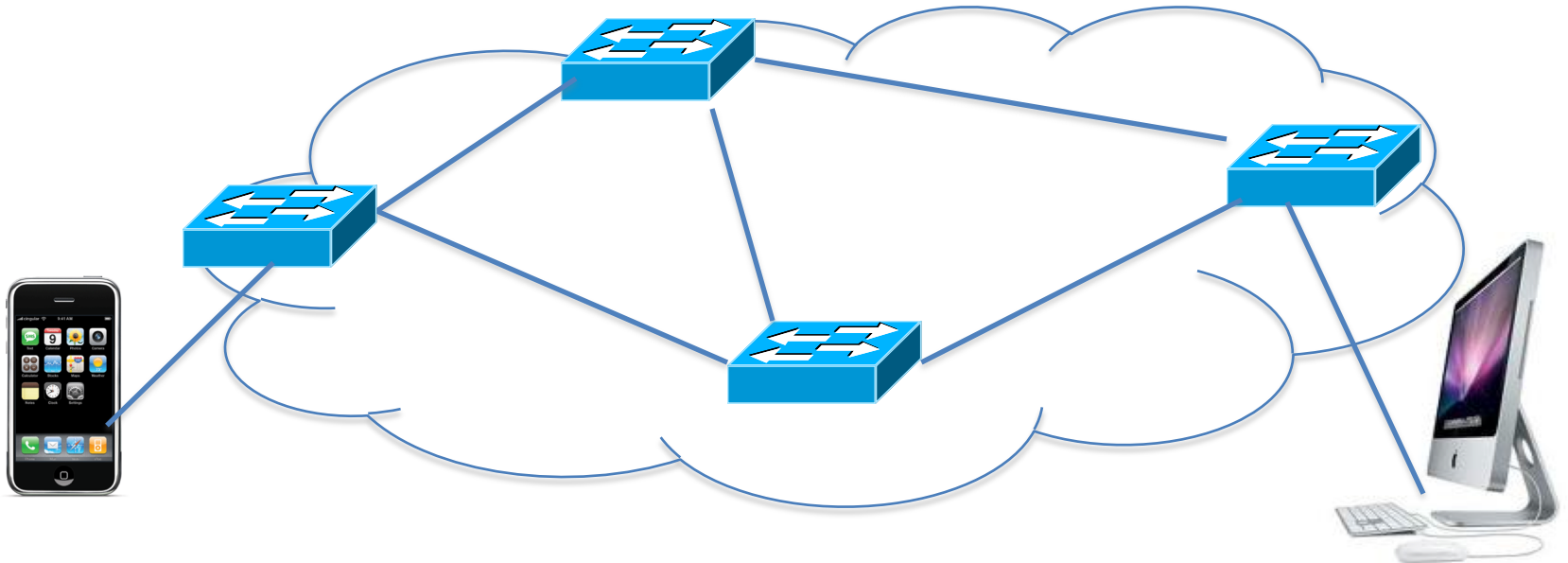
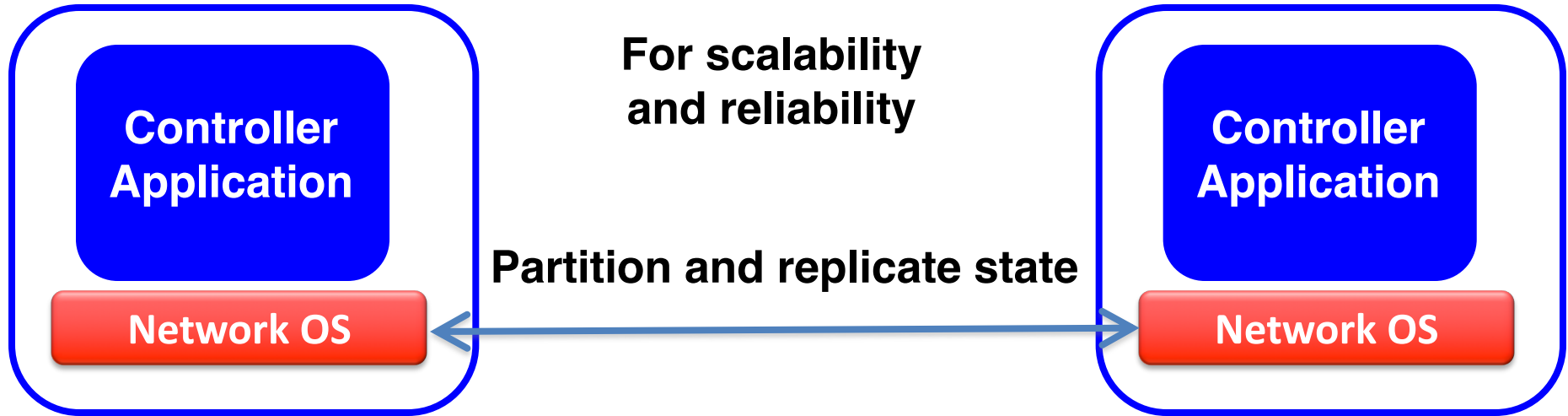


Controller Delay and Overhead

- Controller is much slower than the switch
- Processing packets leads to delay and overhead
- Need to keep most packets in the “fast path”



Distributed Controller

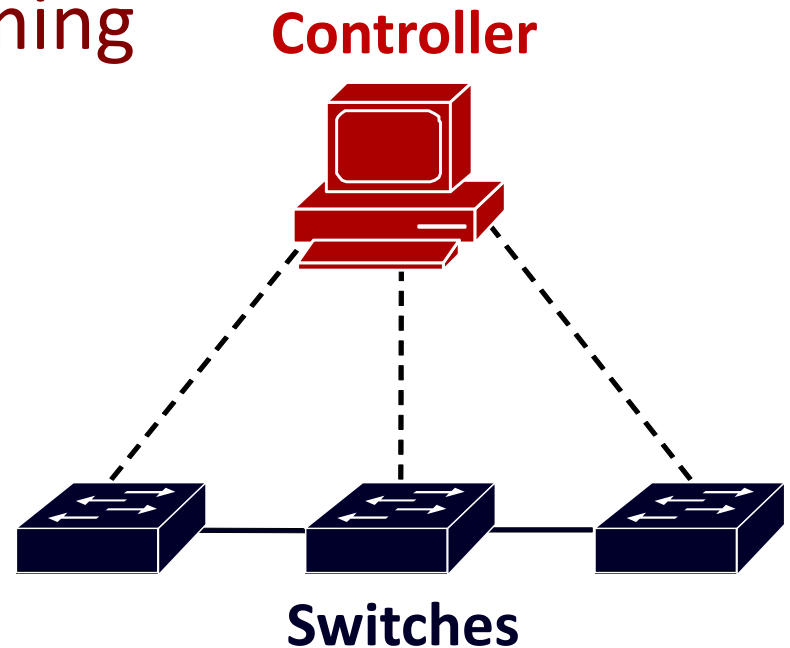


Testing and Debugging

- **OpenFlow makes programming possible**
 - Network-wide view at controller
 - Direct control over data plane
- **Plenty of room for bugs**
 - Still a complex, distributed system
- **Need for testing techniques**
 - Controller applications
 - Controller and switches
 - Rules installed in the switches

Programming Abstractions

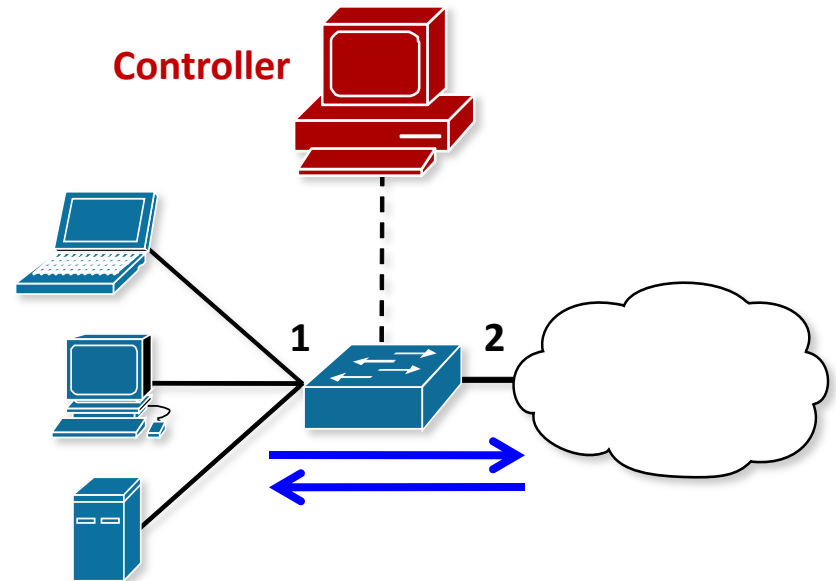
- OpenFlow is a *low-level API*
 - Thin veneer on the underlying hardware
- Makes network programming possible, not easy!



Example: Simple Repeater

Simple Repeater

```
def switch_join(switch):  
    # Repeat Port 1 to Port 2  
    p1 = {in_port:1}  
    a1 = [forward(2)]  
    install(switch, p1, DEFAULT, a1)  
  
    # Repeat Port 2 to Port 1  
    p2 = {in_port:2}  
    a2 = [forward(1)]  
    install(switch, p2, DEFAULT, a2)
```

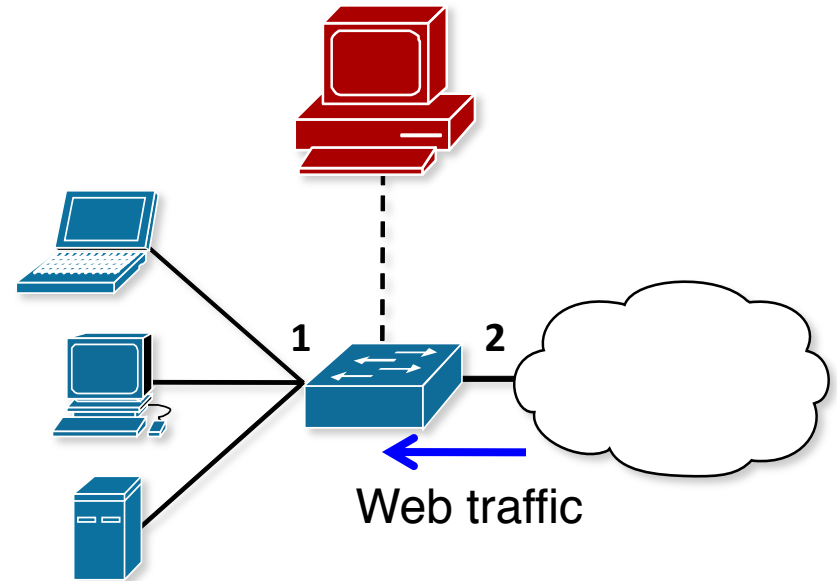


When a switch joins the network, install two forwarding rules.

Example: Web Traffic Monitor

Monitor “port 80” traffic

```
def switch_join(switch):  
    # Web traffic from Internet  
    p = {inport:2,tp_src:80}  
    install(switch, p, DEFAULT, [])  
    query_stats(switch, p)  
  
def stats_in(switch, p, bytes, ...)  
    print bytes  
    sleep(30)  
    query_stats(switch, p)
```



When a switch joins the network, install one monitoring rule.

Composition: Repeater + Monitor

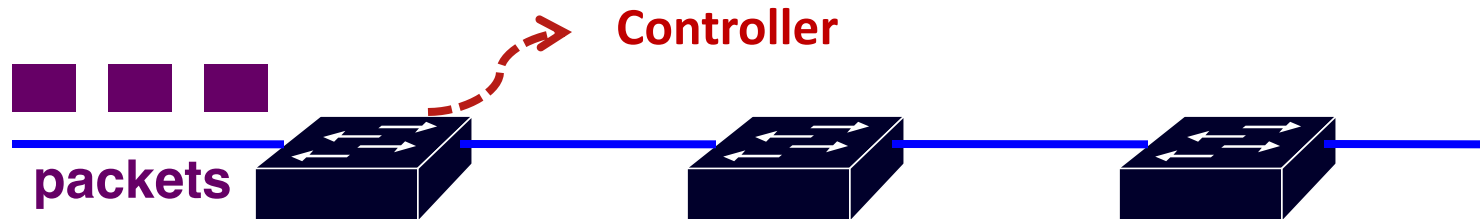
Repeater + **Monitor**

```
def switch_join(switch):  
    pat1 = {inport:1}  
    pat2 = {inport:2}  
    pat2web = {in_port:2, tp_src:80}  
    install(switch, pat1, DEFAULT, None, [forward(2)])  
    install(switch, pat2web, HIGH, None, [forward(1)])  
    install(switch, pat2, DEFAULT, None, [forward(1)])  
    query_stats(switch, pat2web)  
  
def stats_in(switch, xid, pattern, packets, bytes):  
    print bytes  
    sleep(30)  
    query_stats(switch, pattern)
```

Must think about both tasks at the same time.

Asynchrony: Switch-Controller Delays

- Common OpenFlow programming idiom
 - First packet of a flow goes to the controller
 - Controller installs rules to handle remaining packets



- What if more packets arrive before rules installed?
 - Multiple packets of a flow reach the controller
- What if rules along a path installed out of order?
 - Packets reach intermediate switch before rules do

Must think about all possible event orderings.

Better: Increase the level of abstraction

- Separate reading from writing
 - Reading: specify queries on network state
 - Writing: specify forwarding policies
- Compose multiple tasks
 - Write each task once, and combine with others
- Prevent race conditions
 - Automatically apply forwarding policy to extra packets
- See <http://frenetic-lang.org/>

Stage 3

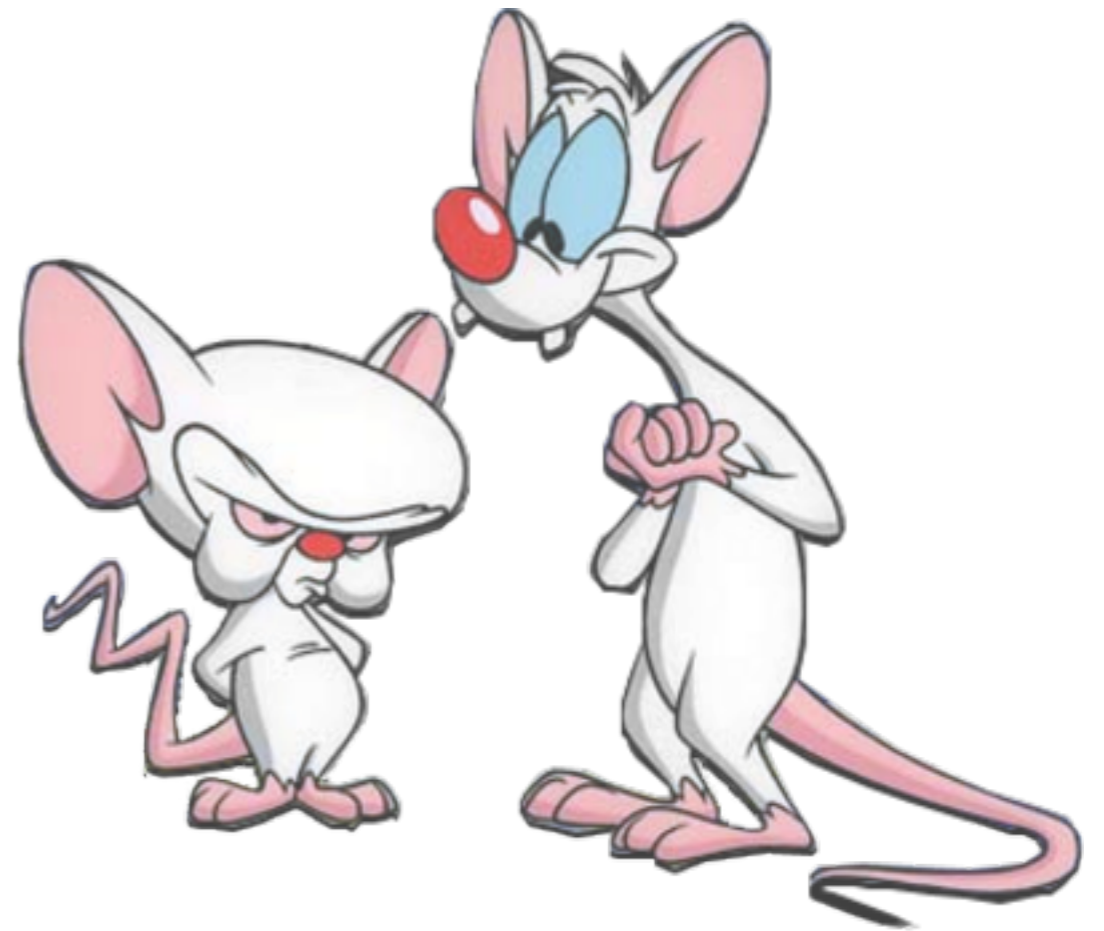
"Deep" Network Programmability

Pinky

Gee, Brain, did OpenFlow take over the world?

The Brain

Well... **no.**



OpenFlow is not all roses

The protocol is too complex (12 fields in OF 1.0 to 41 in 1.5)
switches must support complicated parsers and pipelines

The specification itself keeps getting more complex
extra features make the software agent more complicated

consequences **Switches vendor end up implementing parts of the spec.**
which breaks the abstraction of one API to *rule-them-all*

Enters... Protocol Independent Switch Architecture and P4

0000000-0000004.pdf (page 1 of 8)

P4: Programming Protocol-Independent Packet Processors

Pat Bosshart[†], Dan Daly^{*}, Glen Gibb[†], Martin Izzard[†], Nick McKeown[‡], Jennifer Rexford^{**}, Cole Schlesinger^{**}, Dan Talayco[†], Amin Vahdat[¶], George Varghese[§], David Walker^{**}
[†]Barefoot Networks ^{*}Intel [‡]Stanford University ^{**}Princeton University [¶]Google [§]Microsoft Research

ABSTRACT

P4 is a high-level language for programming protocol-independent packet processors. P4 works in conjunction with SDN control protocols like OpenFlow. In its current form, OpenFlow explicitly specifies protocol headers on which it operates. This set has grown from 12 to 41 fields in a few years, increasing the complexity of the specification while still not providing the flexibility to add new headers. In this paper we propose P4 as a strawman proposal for how OpenFlow should evolve in the future. We have three goals: (1) Reconfigurability in the field: Programmers should be able to change the way switches process packets once they are deployed. (2) Protocol independence: Switches should not be tied to any specific network protocols. (3) Target independence: Programmers should be able to describe packet-processing functionality independently of the specifics of the underlying hardware. As an example, we describe how to use P4 to configure a switch to add a new hierarchical label.

multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.

The proliferation of new header fields shows no signs of stopping. For example, data-center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), for which they resort to deploying software switches that are easier to extend with new functionality. Rather than repeatedly extending the OpenFlow specification, we argue that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new “OpenFlow 2.0” API). Such a general, extensible approach would be simpler, more elegant, and more future-proof than today’s OpenFlow 1.x standard.

SDN Control Plane

Enters... Protocol Independent Switch Architecture and P4

The image shows a screenshot of a PDF document viewer. The window title is '0000000-0000004.pdf (page 1 of 8)'. The document content is as follows:

P4: Programming Protocol-Independent Packet Processors

Pat Bosshart[†], Dan Daly^{*}, Glen Gibb[†], Martin Izzard[†], Nick McKeown[‡], Jennifer Rexford^{**},
Cole Schlesinger^{**}, Dan Talayco[†], Amin Vahdat[¶], George Varghese[§], David Walker^{**}
[†]Barefoot Networks ^{*}Intel [‡]Stanford University ^{**}Princeton University [¶]Google [§]Microsoft Research

ABSTRACT

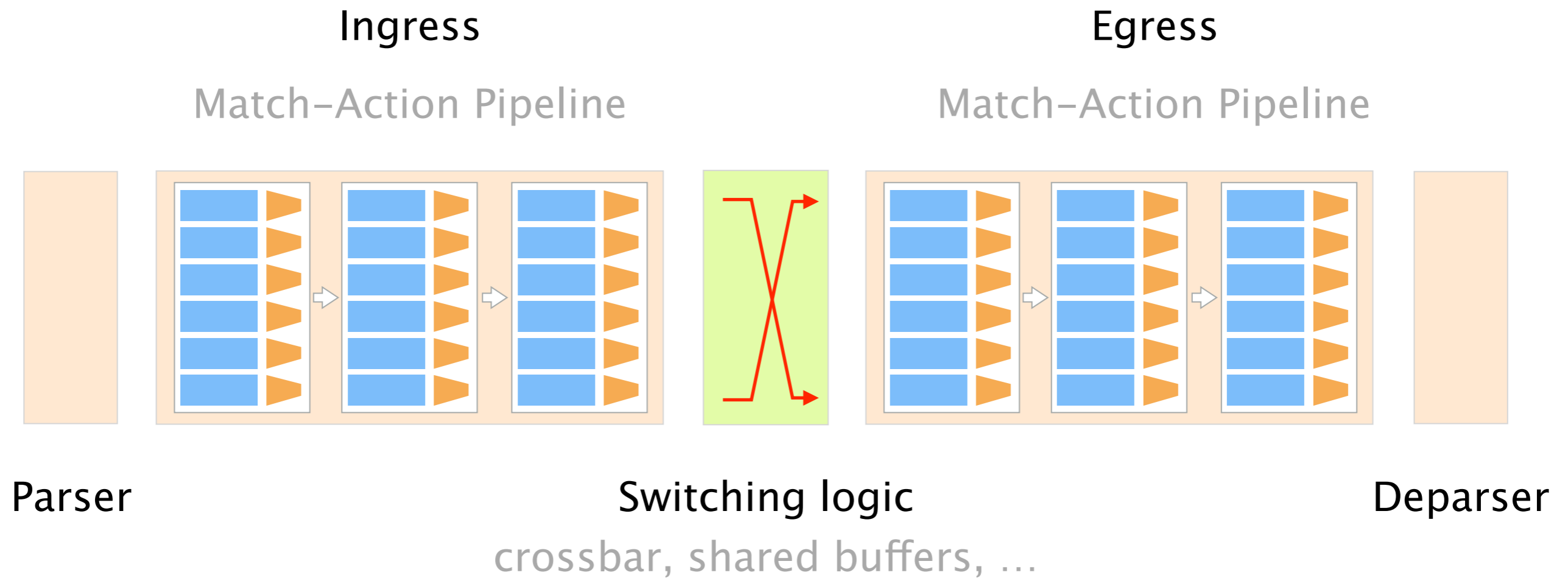
P4 is a high-level language for programming protocol-independent packet processors. P4 works in conjunction with SDN control protocols like OpenFlow. In its current form, OpenFlow explicitly specifies protocol headers on which it operates. This set has grown from 12 to 41 fields in a few years, increasing the complexity of the specification while still not providing the flexibility to add new headers. In this paper we propose P4 as a strawman proposal for how OpenFlow should evolve in the future. We have three goals: (1) Reconfigurability in the field: Programmers should be able to change the way switches process packets once they are deployed. (2) Protocol independence: Switches should not be tied to any specific network protocols. (3) Target independence: Programmers should be able to describe packet-processing functionality independently of the specifics of the underlying hardware. As an example, we describe how to use P4 to configure a switch to add a new hierarchical label.

multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.

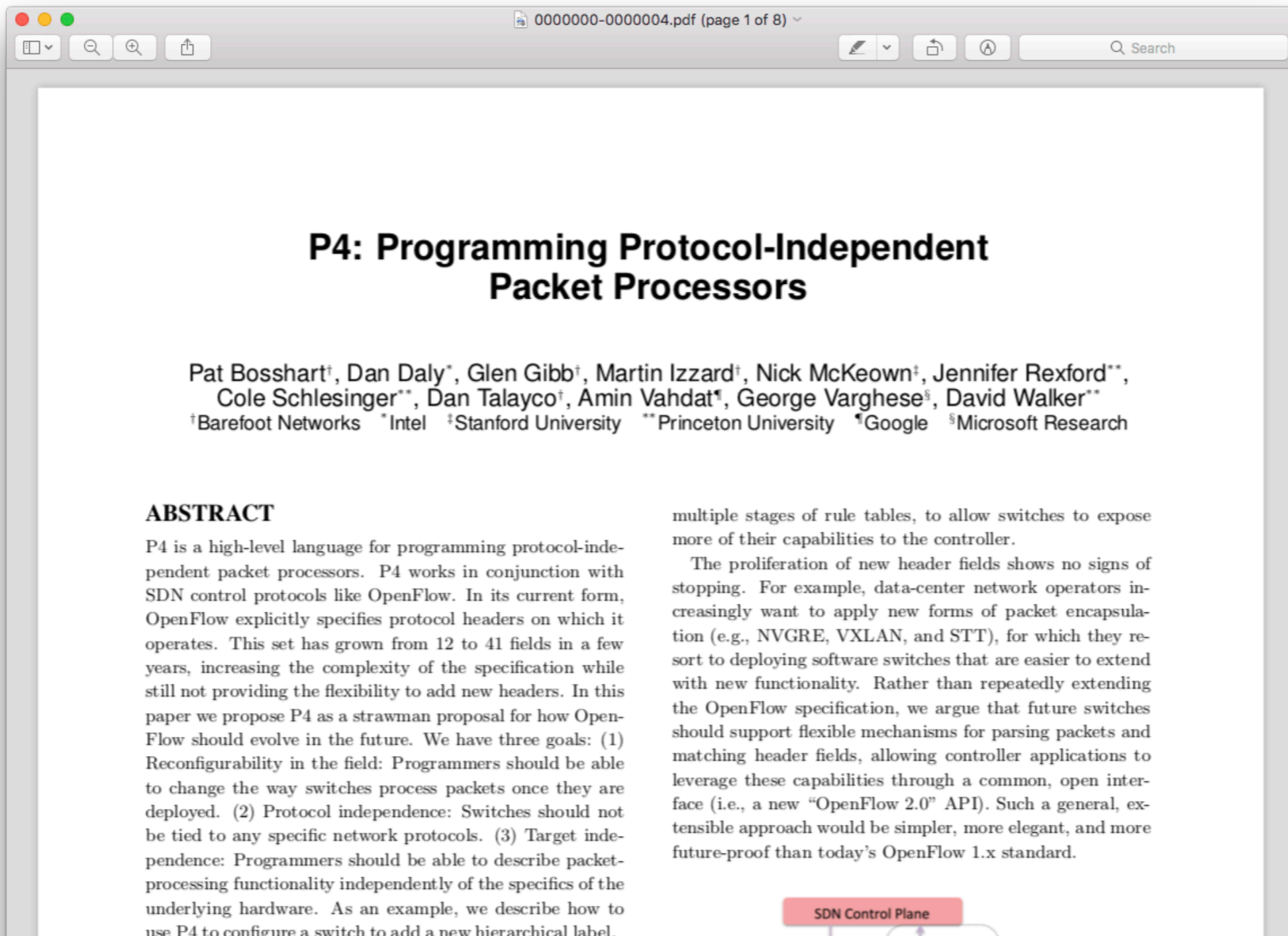
The proliferation of new header fields shows no signs of stopping. For example, data-center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), for which they resort to deploying software switches that are easier to extend with new functionality. Rather than repeatedly extending the OpenFlow specification, we argue that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new “OpenFlow 2.0” API). Such a general, extensible approach would be simpler, more elegant, and more future-proof than today’s OpenFlow 1.x standard.

SDN Control Plane

Protocol Independent Switch Architecture (PISA) for high-speed programmable packet forwarding



Enters... Protocol Independent Switch Architecture and P4



The image shows a screenshot of a PDF document viewer. The title bar at the top indicates the file is '0000000-0000004.pdf (page 1 of 8)'. The document content is centered and features the title 'P4: Programming Protocol-Independent Packet Processors' in a large, bold font. Below the title, the authors' names are listed: Pat Bosshart[†], Dan Daly^{*}, Glen Gibb[‡], Martin Izzard[‡], Nick McKeown[‡], Jennifer Rexford^{**}, Cole Schlesinger^{**}, Dan Talayco[†], Amin Vahdat[¶], George Varghese[§], and David Walker^{**}. Below the names, their affiliations are listed: [†]Barefoot Networks, ^{*}Intel, [‡]Stanford University, ^{**}Princeton University, [¶]Google, and [§]Microsoft Research. The document is divided into two columns. The left column contains the 'ABSTRACT' section, which begins with 'P4 is a high-level language for programming protocol-independent packet processors. P4 works in conjunction with SDN control protocols like OpenFlow. In its current form, OpenFlow explicitly specifies protocol headers on which it operates. This set has grown from 12 to 41 fields in a few years, increasing the complexity of the specification while still not providing the flexibility to add new headers. In this paper we propose P4 as a strawman proposal for how OpenFlow should evolve in the future. We have three goals: (1) Reconfigurability in the field: Programmers should be able to change the way switches process packets once they are deployed. (2) Protocol independence: Switches should not be tied to any specific network protocols. (3) Target independence: Programmers should be able to describe packet-processing functionality independently of the specifics of the underlying hardware. As an example, we describe how to use P4 to configure a switch to add a new hierarchical label.' The right column contains a paragraph that starts with 'multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.' followed by another paragraph: 'The proliferation of new header fields shows no signs of stopping. For example, data-center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), for which they resort to deploying software switches that are easier to extend with new functionality. Rather than repeatedly extending the OpenFlow specification, we argue that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new "OpenFlow 2.0" API). Such a general, extensible approach would be simpler, more elegant, and more future-proof than today's OpenFlow 1.x standard.'

ABSTRACT

P4 is a high-level language for programming protocol-independent packet processors. P4 works in conjunction with SDN control protocols like OpenFlow. In its current form, OpenFlow explicitly specifies protocol headers on which it operates. This set has grown from 12 to 41 fields in a few years, increasing the complexity of the specification while still not providing the flexibility to add new headers. In this paper we propose P4 as a strawman proposal for how OpenFlow should evolve in the future. We have three goals: (1) Reconfigurability in the field: Programmers should be able to change the way switches process packets once they are deployed. (2) Protocol independence: Switches should not be tied to any specific network protocols. (3) Target independence: Programmers should be able to describe packet-processing functionality independently of the specifics of the underlying hardware. As an example, we describe how to use P4 to configure a switch to add a new hierarchical label.

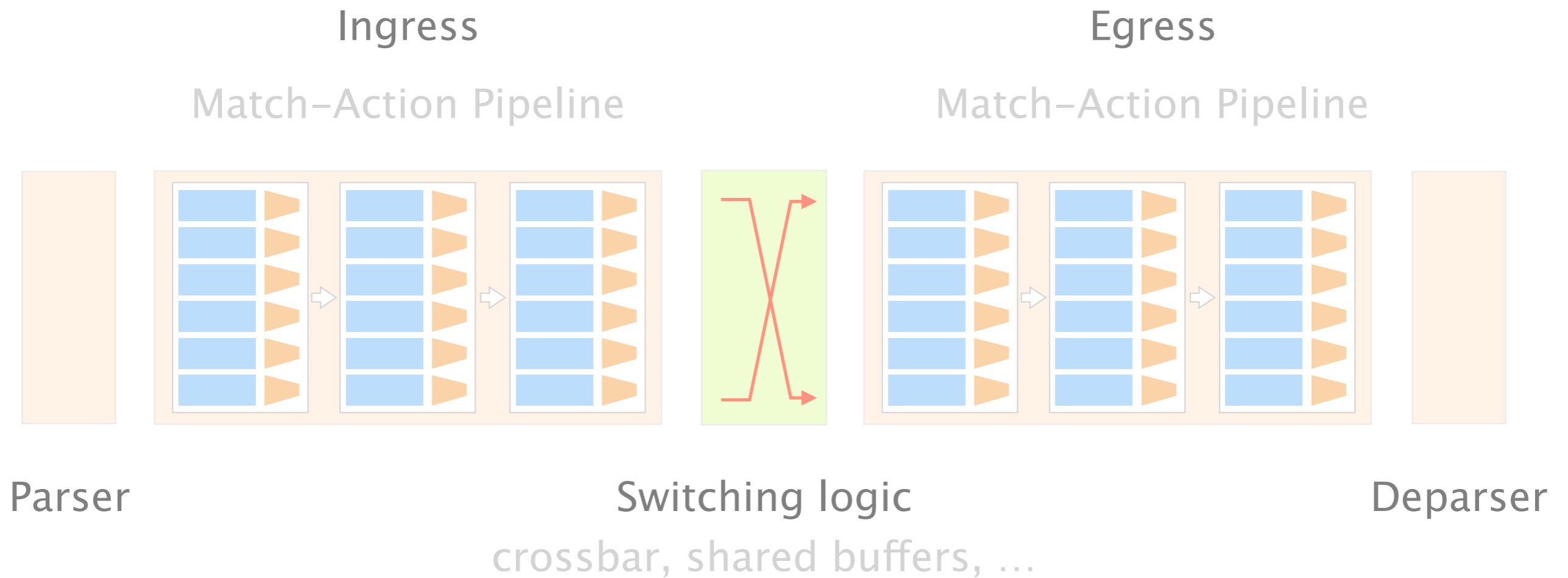
multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.

The proliferation of new header fields shows no signs of stopping. For example, data-center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), for which they resort to deploying software switches that are easier to extend with new functionality. Rather than repeatedly extending the OpenFlow specification, we argue that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new "OpenFlow 2.0" API). Such a general, extensible approach would be simpler, more elegant, and more future-proof than today's OpenFlow 1.x standard.

SDN Control Plane

By default,

PISA doesn't do anything, it's just an "architecture"

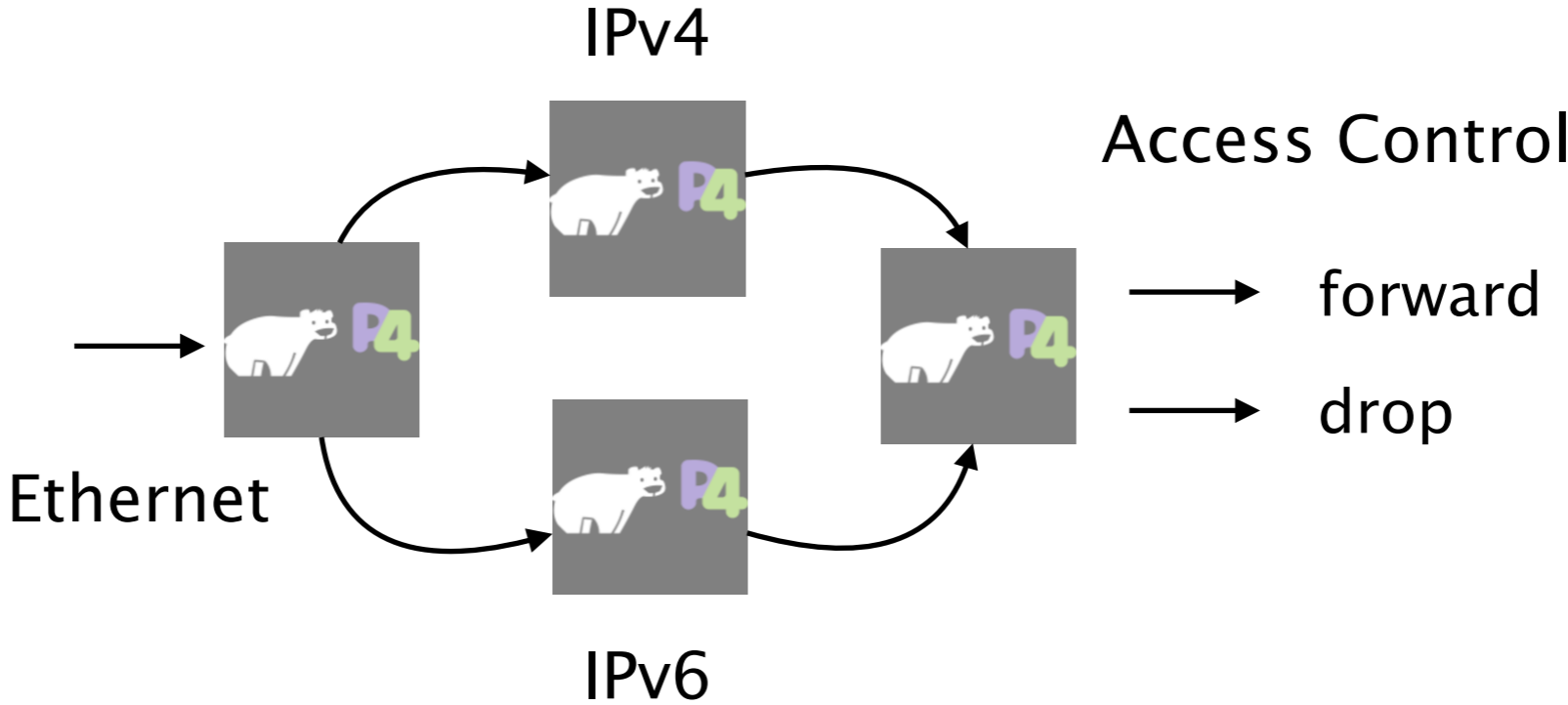


P4 is a domain-specific language which describes how a PISA architecture should process packets



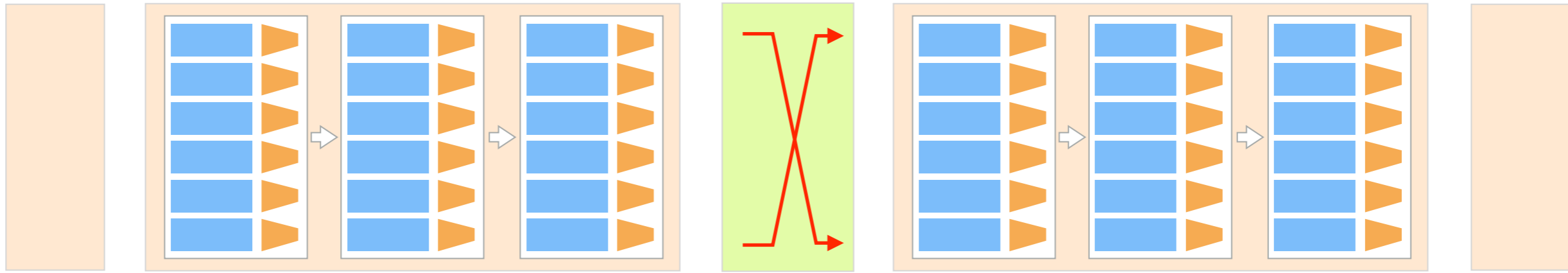
<https://p4.org>

Logical behavior



Compiler

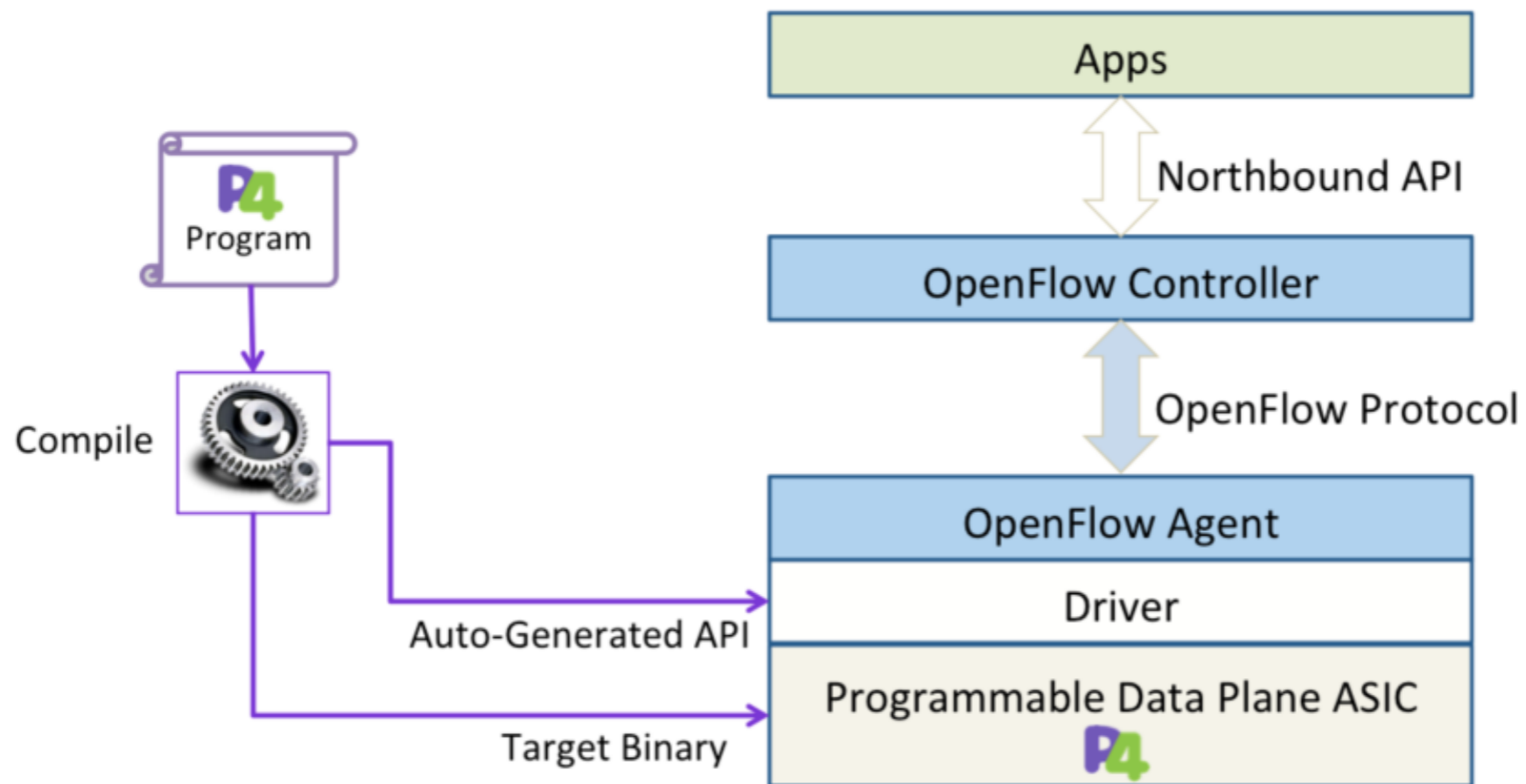
PISA backend



PISA + P4 is strictly more general OpenFlow



P4 & OpenFlow



Course goals

This course will introduce you to the emerging area of network programmability

Learn the principles of network programmability at the control-plane *and* at the data-plane level

Get fluent in P4 programming

the go-to language for programming data planes

Get insights into hard, research-level problems

and how programmability can help solving them

Course organization

The course is gonna be divided in **two blocks**

Lectures/Exercices

~7 weeks

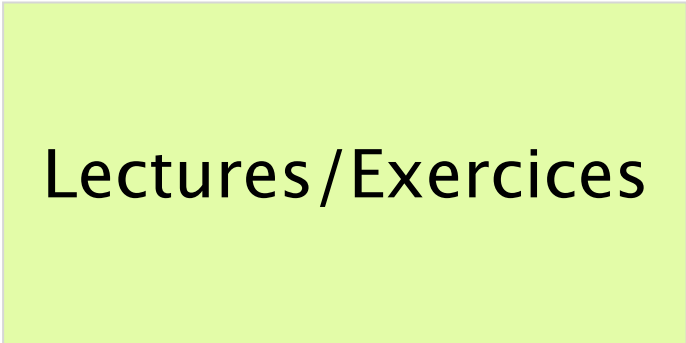
how to program in P4

Group project

>= 7 weeks

in teams of 2—3

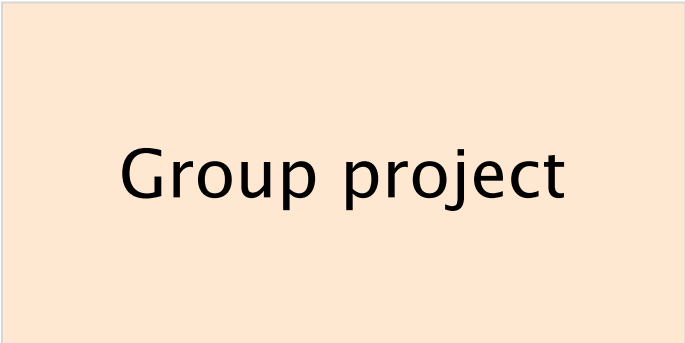
The course is gonna be divided in **two blocks**



Lectures/Exercices

~7 weeks

how to program in P4



Group project

>= 7 weeks

in teams of 2—3

There will be 2h of lectures & 2h of exercises

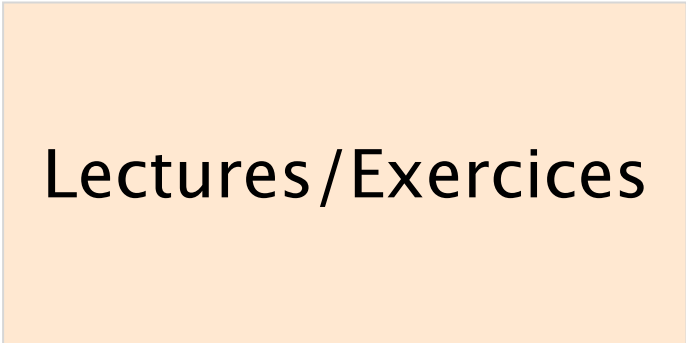
Tue 13—15 Lecture

Tue 15—17 Practical exercises with P4

Exercises are not graded *but* will help at the exam

Both will take place in ML H 44

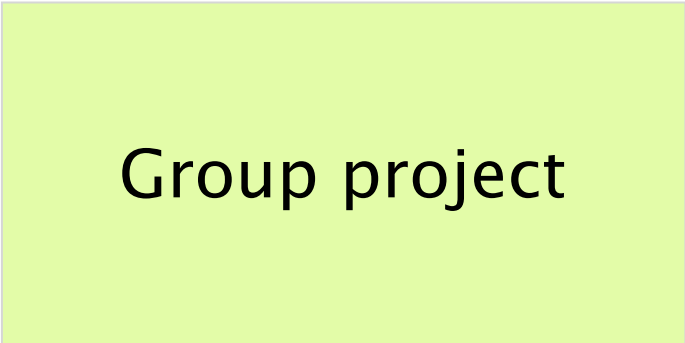
The course is gonna be divided in **two blocks**



Lectures/Exercices

~7 weeks

how to program in P4



Group project

>= 7 weeks

ideally, teams of 3

In the project, you'll develop and evaluate
a **fully-working network application in P4**

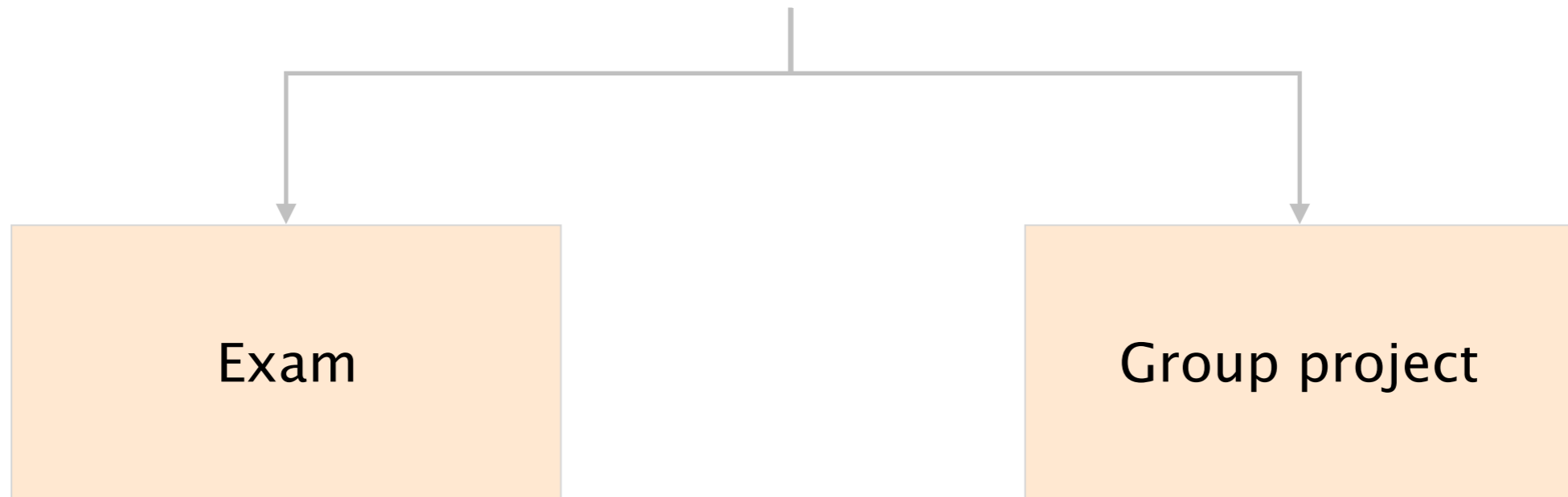
Your can choose your application

Reproduce research papers or implement new ones

We'll provide feedback and assist you throughout
during the lecture/exercise slot and online

Grade will depend on the code, report and presentation
presentations during the last week of the lecture

Your final grade

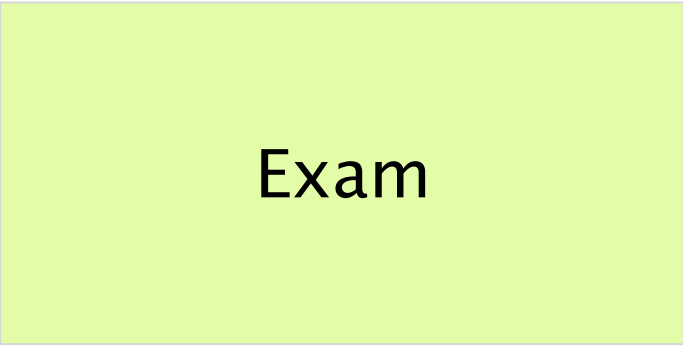


50%

oral

50%

code, report, and presentation



Exam

50%

oral

Examples

Design a P4 application
for solving problem $\langle X \rangle$

Optimize program $\langle X \rangle$

Is program $\langle X \rangle$ correct?

... **important** to do the exercises

Your dream team for the semester



Edgar
(head TA)



Roland



Thomas



Maria



Albert



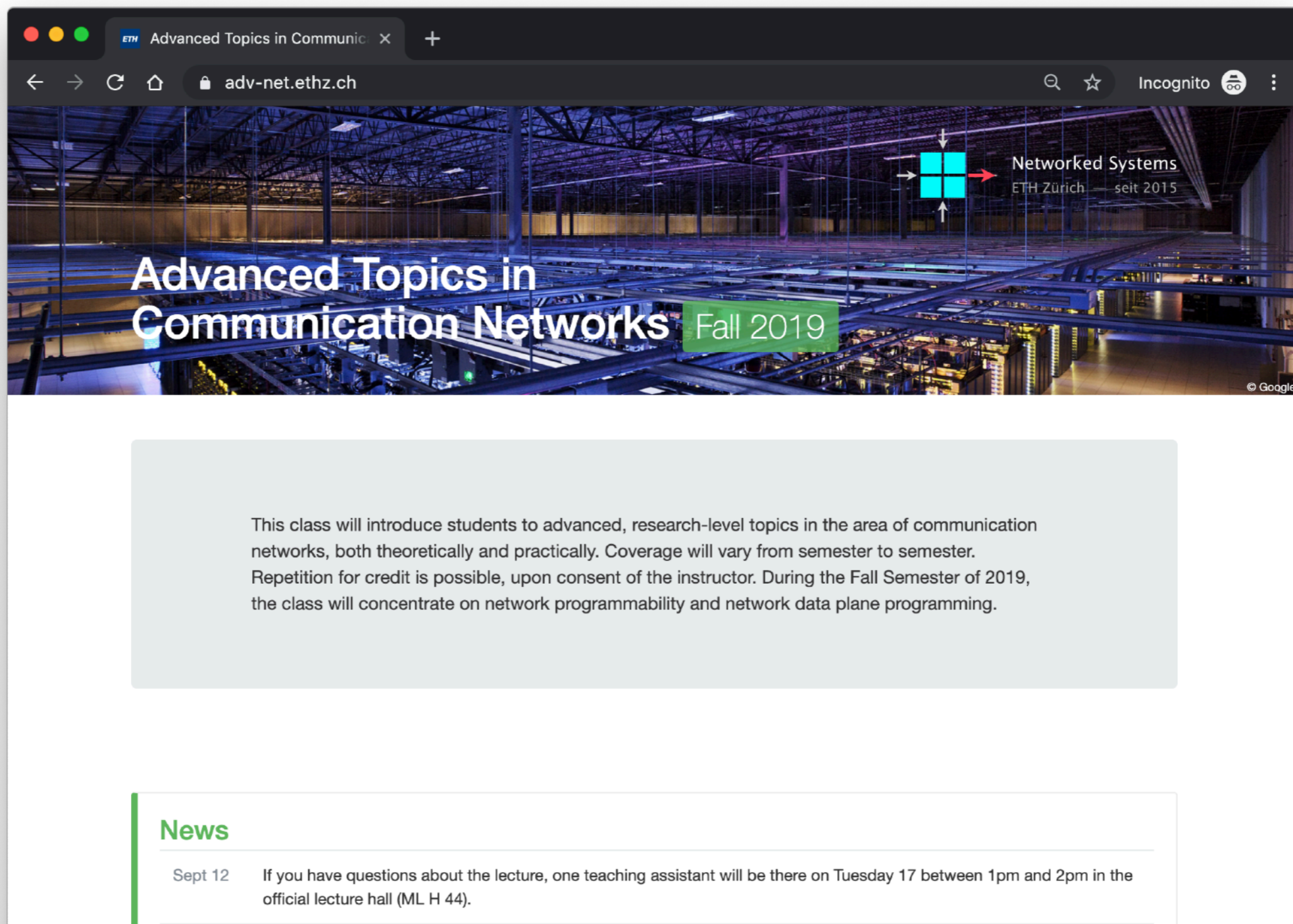
Alexander



Damien

Our website: <https://adv-net.ethz.ch>
check it out regularly

slides, pointers to exercises, readings, ...



ETH Advanced Topics in Communicat... x +

adv-net.ethz.ch Incognito

Networked Systems
ETH Zürich — seit 2015

Advanced Topics in Communication Networks

Fall 2019

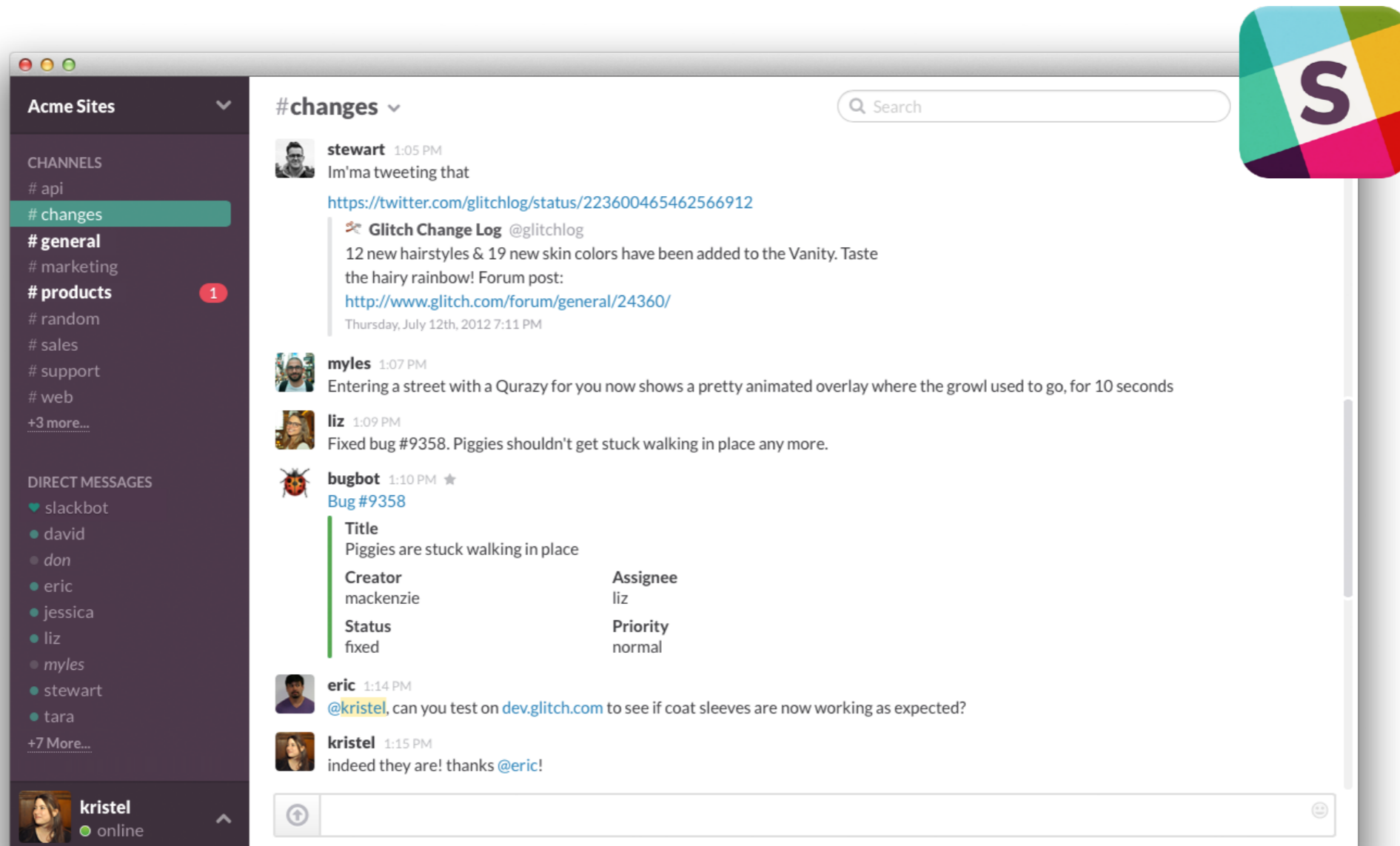
© Google

This class will introduce students to advanced, research-level topics in the area of communication networks, both theoretically and practically. Coverage will vary from semester to semester. Repetition for credit is possible, upon consent of the instructor. During the Fall Semester of 2019, the class will concentrate on network programmability and network data plane programming.

News

Sept 12 If you have questions about the lecture, one teaching assistant will be there on Tuesday 17 between 1pm and 2pm in the official lecture hall (ML H 44).

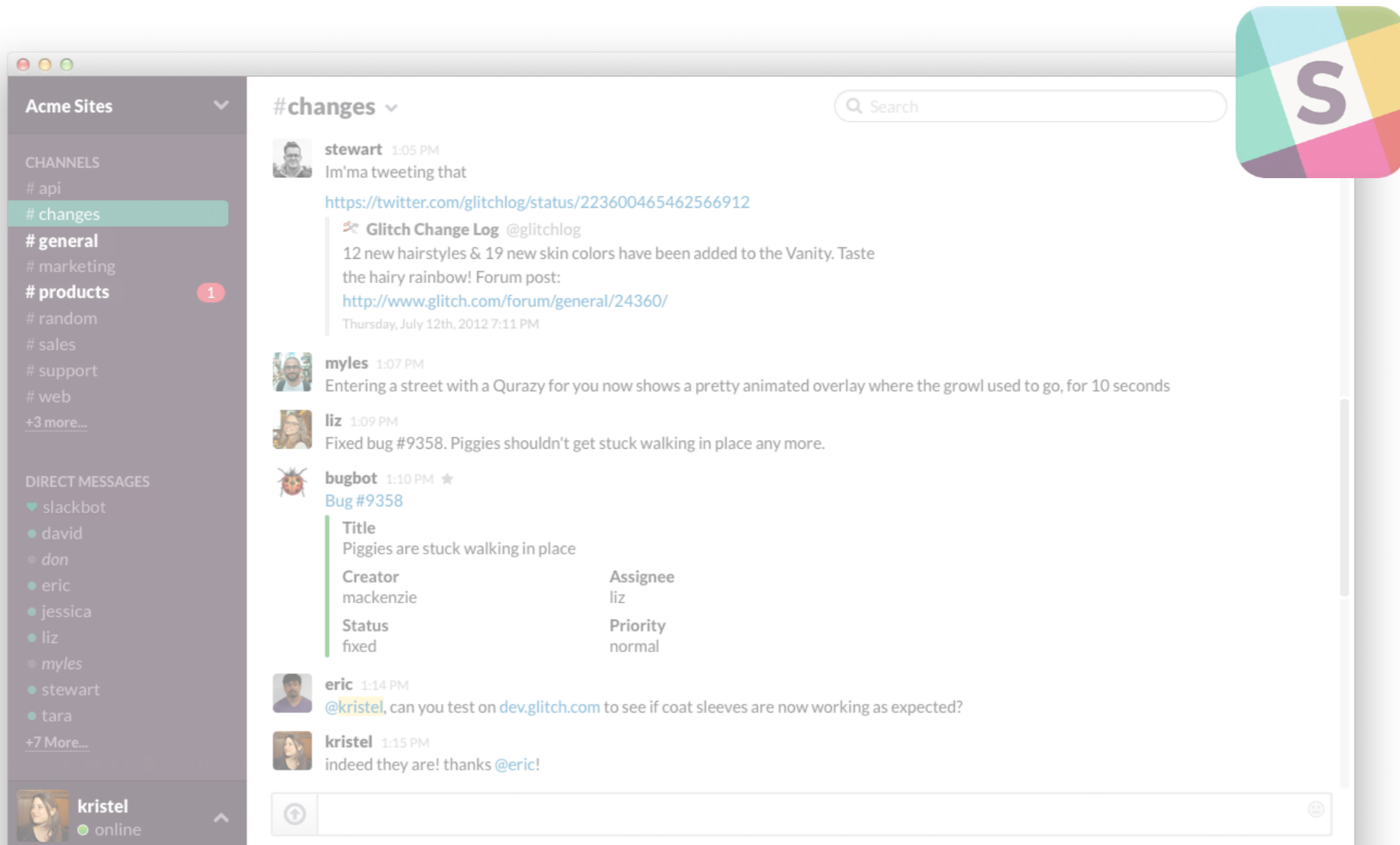
We'll use Slack (chat client) to discuss about the course, exercises, and projects



Web, smartphone and desktop clients available

Register today using your real name

> <https://adv-net19.slack.com/signup>



Web, smartphone and desktop clients available

Should I take this course?

It depends...

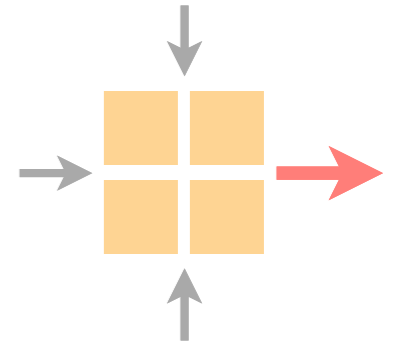
You shouldn't take the course if...

- you *hate* programming
- you don't want to work during the semester
- you expect 10+ years of exam history

Besides that, if you like networking... *go for it!*

Advanced Topics in Communication Networks

Programming Network Data Planes



Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

24 Sep 2019

A quick look at how **we** use



in our research

We used P4 to...

Enable programmable
packet scheduling
at Tbps

SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues

Albert Gran Alcoz
ETH Zürich

Alexander Dietmüller
ETH Zürich

Laurent Vanbever
ETH Zürich

Abstract

Push-In First-Out (PIFO) queues are hardware primitives which enable programmable packet scheduling by providing the abstraction of a priority queue at line rate. However, implementing them at scale is not easy: just hardware designs (not implementations) exist, which support only about 1k flows.

In this paper, we introduce SP-PIFO, a programmable packet scheduler which closely approximates the behavior of PIFO queues using strict-priority queues—at line rate, at scale, and on existing devices. The key insight behind SP-PIFO is to dynamically adapt the mapping between packet ranks and available strict-priority queues to minimize the scheduling errors with respect to an ideal PIFO. We present a mathematical formulation of the problem and derive an adaptation technique which closely approximates the optimal queue mapping without any traffic knowledge.

We fully implement SP-PIFO in P4 and evaluate it on real workloads. We show that SP-PIFO: (i) closely matches PIFO, with as little as 8 priority queues; (ii) scales to large amount of flows and ranks; and (iii) quickly adapts to traffic variations. We also show that SP-PIFO runs at line rate on existing hardware (Barefoot Tofino), with a negligible memory footprint.

1 Introduction

Until recently, packet scheduling was one of the last bastions standing in the way of complete data-plane programmability. Indeed, unlike forwarding whose behavior can be adapted thanks to languages such as P4 [7] and reprogrammable hardware [2], scheduling behavior is mostly set in stone with hardware implementations that can, at best, be configured.

To enable programmable packet scheduling, the main challenge was to find an appropriate abstraction which is flexible enough to express a wide variety of scheduling algorithms and yet can be implemented efficiently in hardware [22]. In [23], Sivaraman et al. proposed to use Push-In First-Out (PIFO) queues as such an abstraction. PIFO queues allow enqueued packets to be pushed in arbitrary positions (according to the packets rank) while being drained from the head.

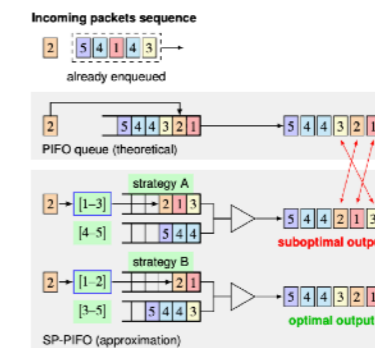


Figure 1: SP-PIFO approximates the behavior of PIFO queues by adapting how packet ranks are mapped to priority queues.

While PIFO queues enable programmable scheduling, implementing them in hardware is hard due to the need to arbitrarily sort packets at line rate. [23] described a possible hardware design (not implementation) supporting PIFO on top of Broadcom Trident II [1]. While promising, realizing this design in an ASIC is likely to take years [6], not including deployment. Even ignoring deployment considerations, the design of [23] is limited as it only supports ~1000 flows and relies on the assumption that the packet ranks increase monotonically within each flow, which is not always the case.

Our work In this paper, we ask whether it is possible to approximate PIFO queues at scale, in existing programmable data planes. We answer positively and present SP-PIFO, an adaptive scheduling algorithm that closely approximates PIFO behaviors on top of widely-available Strict-Priority (SP) queues. The key insight behind SP-PIFO is to dynamically adapt the mapping between packet ranks and SP queues in order to minimize the amount of scheduling mistakes relative to a hypothetical ideal PIFO implementation.

We used P4 to...

Converge upon
remote Internet failures
in less than 1 sec

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach*, Edgar Costa Molero*, Maria Apostolaki*
Alberto Dainotti†, Stefano Vissicchio‡, Laurent Vanbever*

*ETH Zurich, †CAIDA / UC San Diego, ‡University College London

Abstract

We present Blink, a data-driven system that leverages TCP-induced signals to detect failures directly in the data plane. The key intuition behind Blink is that a TCP flow exhibits a predictable behavior upon disruption: retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. Blink efficiently analyzes TCP flows to: (i) select which ones to track; (ii) reliably and quickly detect major traffic disruptions; and (iii) recover connectivity—all this, completely in the data plane.

We present an implementation of Blink in P4 together with an extensive evaluation on real and synthetic traffic traces. Our results indicate that Blink: (i) achieves sub-second rerouting for large fractions of Internet traffic; and (ii) prevents unnecessary traffic shifts even in the presence of noise. We further show the feasibility of Blink by running it on an actual Tofino switch.

1 Introduction

Thanks to widely deployed fast-convergence frameworks such as IPFFR [35], Loop-Free Alternate [7] or MPLS Fast Reroute [29], sub-second and ISP-wide convergence upon link or node failure is now the norm [6, 15]. At a high-level, these fast-convergence frameworks share two common ingredients: (i) *fast detection* by leveraging hardware-generated signals (e.g., Loss-of-Light or unanswered hardware keepalive [23]); and (ii) *quick activation* by promptly activating pre-computed backup state upon failure instead of recomputing the paths on-the-fly.

Problem: Convergence upon remote failures is still slow. These frameworks help ISPs to retrieve connectivity upon *internal* (or peering) failures but are of no use when it comes to restoring connectivity upon *remote* failures. Unfortunately, remote failures are both frequent and slow to repair, with average convergence times above 30 s [19, 24, 28]. These failures indeed trigger a *control-plane-driven* convergence through the propagation of BGP updates on a per-router and per-prefix

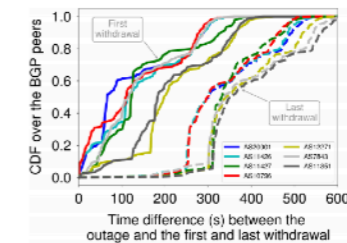


Figure 1: It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

basis. To reduce convergence time, SWIFT [19] predicts the entire extent of a remote failure from a few received BGP updates, leveraging the fact that such updates are correlated (e.g., they share the same AS-PATH). The fundamental problem with SWIFT though, is that it can take $O(\text{minutes})$ for the *first* BGP update to propagate after the corresponding data-plane failure.

We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [1]. We consider as outage time t_0 , the time at which traffic originated by TWC ASes observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [27] and RIPE RIS [2], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

We used P4 to...

Secure Bitcoin
by relaying blocks
in hardware

SABRE: Protecting Bitcoin against Routing Attacks

Maria Apostolaki
ETH Zurich
apmaria@ethz.ch

Gian Marti
ETH Zurich
gimarti@student.ethz.ch

Jan Müller
ETH Zurich
jan.m.muller@me.com

Laurent Vanbever
ETH Zurich
lvanbever@ethz.ch

Abstract—Nowadays Internet routing attacks remain practically effective as existing countermeasures either fail to provide protection guarantees or are not easily deployable. Blockchain systems are particularly vulnerable to such attacks as they rely on Internet-wide communications to reach consensus. In particular, Bitcoin—the most widely-used cryptocurrency—can be split in half by any AS-level adversary using BGP hijacking.

In this paper, we present SABRE, a secure and scalable Bitcoin relay network which relays blocks worldwide through a set of connections that are resilient to routing attacks. SABRE runs alongside the existing peer-to-peer network and is easily deployable. As a critical system, SABRE design is highly resilient and can efficiently handle high bandwidth loads, including Denial of Service attacks.

We built SABRE around two key technical insights. First, we leverage fundamental properties of inter-domain routing (BGP) policies to host relay nodes: (i) in networks that are inherently protected against routing attacks; and (ii) on paths that are economically-preferred by the majority of Bitcoin clients. These properties are generic and can be used to protect other Blockchain-based systems. Second, we leverage the fact that relaying blocks is communication-heavy, not computation-heavy. This enables us to offload most of the relay operations to programmable network hardware (using the P4 programming language). Thanks to this hardware/software co-design, SABRE nodes operate seamlessly under high load while mitigating the effects of malicious clients.

We present a complete implementation of SABRE together with an extensive evaluation. Our results demonstrate that SABRE is effective at securing Bitcoin against routing attacks, even with deployments of as few as 6 nodes.

I. INTRODUCTION

Cryptocurrencies, and Bitcoin in particular, are vulnerable to routing attacks in which network-level attackers (i.e., malicious Autonomous Systems or ASes) manipulate routing (BGP) advertisements to divert their connections. Once on-path, the AS-level attacker can disrupt the consensus algorithm by partitioning the peer-to-peer network. Recent studies [17] have shown that these attacks are practical and disruptive. Specifically, any AS-level attacker can isolate ~50% of the Bitcoin mining power by hijacking less than 100 prefixes [17]. Such attacks can lead to significant revenue loss for miners and enable exploits such as double spending.

Problem Protecting against such partitioning attacks is challenging. On the one hand, local (and easily deployable) countermeasures [17] fail to provide strong protection guarantees. These countermeasures include having Bitcoin clients monitor their connections (e.g., for increased or abnormal delays) or having them select their peers based on routing information. On the other hand, Internet-wide countermeasures are extremely hard to deploy. For example, systematically hosting Bitcoin clients in /24 prefixes (to prevent more-specific prefix attacks) requires the unlikely cooperation of all Internet Service Providers hosting Bitcoin clients in addition to a considerable increase to the size of the Internet routing tables. Worse yet, even heavy protocol modification such as encrypting all Bitcoin traffic would not be enough to guarantee Bitcoin safety as AS-level attackers would still be able to distinguish (and drop) Bitcoin traffic using transport headers.

SABRE: A Secure Relay Network for Bitcoin In this paper, we present SABRE, a secure relay network which runs alongside the existing Bitcoin network and which can protect the vast majority of the Bitcoin clients against routing attacks. Unlike existing countermeasures, SABRE secures Bitcoin against routing attacks in a way which: (i) provides strong security guarantees to any connected client by enabling it to learn and propagate blocks; (ii) is partially deployable; and (iii) provides security benefits early-on in the deployment, with as little as two relay nodes. We built SABRE based on two key insights.

Insight #1: Hosting relays in inherently safe locations Our first insight is to host SABRE relay nodes in locations that: (i) prevent attackers from diverting relay-to-relay connections, so as to secure SABRE internal connectivity; and (ii) are attractive (from a routing viewpoint) to many Bitcoin clients, so as to protect client connections to the relay network. We do so by leveraging a fundamental characteristic of BGP policies, namely, that connections established between two ASes which directly peer with each other and which have no customers cannot be diverted by routing attacks. In SABRE, only such ASes are considered for relay locations.

Using real routing data, we show that such safe locations are plentiful in the current Internet with 2000 ASes being eligible. These ASes include large cloud providers, content delivery networks, and Internet eXchange Points which already provide hosting services today and therefore have an incentive to host SABRE nodes. We also show that SABRE deployments with 6 nodes are already enough to protect 80% of the clients from 96% of the AS-level adversaries (assuming worst case scenario for SABRE).

We used P4 to...

Speed-up
network computations
by running them in hardware

Hardware-Accelerated Network Control Planes

Edgar Costa Molero
ETH Zürich
cedgar@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@cs.ucl.ac.uk

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

ABSTRACT

One design principle of modern network architecture seems to be set in stone: a software-based control plane drives a hardware- or software-based data plane. We argue that it is time to revisit this principle after the advent of programmable switch ASICs which can run complex logic at line rate.

We explore the possibility and benefits of accelerating the control plane by offloading some of its tasks directly to the network hardware. We show that programmable data planes are indeed powerful enough to run key control plane tasks including: failure detection and notification, connectivity retrieval, and even policy-based routing protocols. We implement in P4 a prototype of such a “hardware-accelerated” control plane, and illustrate its benefits in a case study.

Despite such benefits, we acknowledge that offloading tasks to hardware is not a silver bullet. We discuss its tradeoffs and limitations, and outline future research directions towards hardware-software codesign of network control planes.

1 INTRODUCTION

As the “brain” of the network, the control plane is one of its most important assets. Among other things, the control plane is responsible for *sensing* the status of the network (e.g., which links are up or which links are overloaded), *computing* the best paths along which to guide traffic, and *updating* the underlying data plane accordingly. To do so, the control plane is composed of many dynamic and interacting processes (e.g., routing, management and accounting protocols) whose operation must scale to large networks. In contrast, the data plane is “only” responsible for forwarding traffic according to the control plane decisions, albeit as fast as possible.

These fundamental differences lead to very different design philosophies. Given the relative simplicity of the data plane and the “need for speed”, it is typically entirely implemented in hardware. That said, software-based implementations of data planes are also commonly found (e.g., OpenVSwitch [30]) together with hybrid software-hardware ones (e.g., CacheFlow [20]). In short, data plane implementations

cover the entire implementation spectrum, from pure software to pure hardware. In contrast, there is *much* less diversity in control plane implementations. The sheer complexity of the control plane tasks (e.g., performing routing computations) together with the need to update them relatively frequently (e.g., to support new protocols and features) indeed calls for software-based implementations, with only a few key tasks (e.g., detecting physical failures, activating backup forwarding state) being (sometimes) offloaded to hardware [13, 22].

Yet, we argue that a number of recent developments are creating both the *need* and *opportunity* for rethinking basic design and implementation choices of network control planes.

Need There is a growing need for faster, more scalable, and yet more powerful control planes. Nowadays, even beefed-up and highly-optimized software control planes can only process thousands of (BGP) control plane messages per second [23], and can take *minutes* to converge upon large failures [17, 36]. Parallelizing only marginally helps: for instance, the BGP specification [31] mandates to lock all Adj-RIBs-In before proceeding with the best-path calculation, essentially preventing the parallel execution of best path computations. A concrete risk is that convergence time will keep increasing with the network size and the number of Internet destinations. At the same time, recent research has repeatedly shown the performance benefits of controlling networks with extremely tight control loops, among others to handle congestion (e.g., [7, 21, 29]).

Opportunity Modern reprogrammable switches (e.g., [1]) can perform complex stateful computations on billions of packets per second [19]. Running (pieces of) the control plane at such speeds would lead to almost “instantaneous” convergence, leaving the propagation time of the messages as the primary bottleneck. Besides speed, offloading control plane tasks to hardware would also help by making them traffic-aware. For instance, it enables to update forwarding entries consistently with real-time traffic volumes rather than in a random order.

Research questions Given the opportunity and the need, we argue that it is time to revisit the control plane’s design and implementation by considering the problem of offloading parts of it to hardware. This redesign opens the door to multiple research questions including: *Which pieces of the control plane should be offloaded? What are the benefits?* and *How can we overcome the fundamental hardware limitations?* These fundamental limitations come mainly from the very limited instruction set (e.g., no floating point) and the memory available (i.e., around tens of megabytes [19]) of programmable network hardware. We start to answer these questions in this paper and make two contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA

© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6120-0/18/11...\$15.00
<https://doi.org/10.1145/3286062.3286080>

We used P4 to...

Obfuscate network topologies from attackers

NetHide: Secure and Practical Network Topology Obfuscation

Roland Meier*, Petar Tsankov*, Vincent Lenders[◇], Laurent Vanbever*, Martin Vechev*

* *ETH Zürich* [◇] *armasuisse*

nethide.ethz.ch

Abstract

Simple path tracing tools such as `traceroute` allow malicious users to infer network topologies remotely and use that knowledge to craft advanced denial-of-service (DoS) attacks such as Link-Flooding Attacks (LFAs). Yet, despite the risk, most network operators still allow path tracing as it is an essential network debugging tool.

In this paper, we present NetHide, a network topology obfuscation framework that mitigates LFAs while preserving the practicality of path tracing tools. The key idea behind NetHide is to formulate network obfuscation as a multi-objective optimization problem that allows for a flexible tradeoff between security (encoded as hard constraints) and usability (encoded as soft constraints). While solving this problem exactly is hard, we show that NetHide can obfuscate topologies at scale by only considering a subset of the candidate solutions and without reducing obfuscation quality. In practice, NetHide obfuscates the topology by intercepting and modifying path tracing probes directly in the data plane. We show that this process can be done at line-rate, in a stateless fashion, by leveraging the latest generation of programmable network devices.

We fully implemented NetHide and evaluated it on realistic topologies. Our results show that NetHide is able to obfuscate large topologies (> 150 nodes) while preserving near-perfect debugging capabilities. In particular, we show that operators can still precisely trace back > 90% of link failures despite obfuscation.

1 Introduction

Botnet-driven Distributed Denial-of-Service (DDoS) attacks constitute one of today's major Internet threats [1, 2, 5, 10]. Such attacks can be divided in two categories depending on whether they target end-hosts and services (volume-based attacks) or the network infrastructure itself (link-flooding attacks, LFAs).

Volume-based attacks are the simplest and work by sending massive amounts of data to selected targets. Recent examples include the 1.2 Tbps DDoS attack against Dyn's DNS service [6] in October 2016 and the 1.35 Tbps DDoS attack against GitHub in February 2018 [8]. While impressive, these attacks can be mitigated today by diverting the incoming traffic through large CDN infrastructures [23]. As an illustration, CloudFlare's infrastructure can now mitigate volume-based attacks reaching Terabits per second [18].

Link-flooding attacks (LFAs) [26, 38] are more sophisticated and work by having a botnet generate low-rate flows between pairs of bots or towards public services such that all of these flows cross a given set of network links or nodes, degrading (or even preventing) the connectivity for *all* services using them. LFAs are much harder to detect as: (i) traffic volumes are relatively small (10 Gbps or 40 Gbps attacks are enough to kill most Internet links [31]); and (ii) attack flows are indistinguishable from legitimate traffic. Representative examples include the Spamhaus attack which flooded selected Internet eXchange Point (IXP) links in Europe and Asia [4, 7, 12].

Unlike volume-based attacks, performing an LFA requires the attacker to know the topology *and* the forwarding behavior of the targeted network. Without this knowledge, an attacker can only "guess" which flows share a common link, considerably reducing the attack's efficiency. As an illustration, our simulations indicate that congesting an *arbitrary* link without knowing the topology requires 5 times more flows, while congesting a *specific* link is order of magnitudes more difficult.

Nowadays, attackers can easily acquire topology knowledge by running path tracing tools such as `traceroute` [17]. In fact, previous studies have shown that entire topologies can be precisely mapped with `traceroute` provided enough vantage points are used [37], a requirement easily met by using large-scale measurement platforms (e.g., RIPE Atlas [16]).

We used P4 to...

Classify traffic
using ML-based models
at Tbps

arXiv:1909.05680v1 [cs.NI] 12 Sep 2019

***p*Forest: In-Network Inference with Random Forests**

Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, Laurent Vanbever
ETH Zürich

Abstract

The concept of “self-driving networks” has recently emerged as a possible solution to manage the ever-growing complexity of modern network infrastructures. In a self-driving network, network devices adapt their decisions in real-time by observing network traffic and by performing in-line inference according to machine learning models. The recent advent of programmable data planes gives us a unique opportunity to implement this vision. One open question though is whether these devices are powerful enough to run such complex tasks?

We answer positively by presenting *p*Forest, a system for performing *in-network* inference according to supervised machine learning models on top of programmable data planes. The key challenge is to design classification models that fit the constraints of programmable data planes (e.g., no floating points, no loops, and limited memory) while providing high accuracy. *p*Forest addresses this challenge in three phases: (i) it optimizes the features selection according to the capabilities of programmable network devices; (ii) it trains random forest models tailored for different phases of a flow; and (iii) it applies these models in real time, on a per-packet basis.

We fully implemented *p*Forest in Python (training), and in P4₁₆ (inference). Our evaluation shows that *p*Forest can classify traffic at line rate for hundreds of thousands of flows, with an accuracy that is on-par with software-based solutions. We further show the practicality of *p*Forest by deploying it on existing hardware devices (Barefoot Tofino).

1 Introduction

What if networks could “self-manage” instead of having operators painstakingly specifying their behavior? Behind this vision—perhaps a bit futuristic—lies the concept of *Self-Driving Networks* [24, 25, 36, 37]. In a self-driving network, network devices measure, analyze, and adapt to the network conditions in real-time, without requiring off-path analysis.

Akin to self-driving cars, the idea of having networks “driving themselves” is appealing in terms of performance, re-

liability, and security. As an illustration, a self-driving network could optimize application performance (e.g., maximize bitrate, minimize rebuffering) by: (i) observing lower-level metrics (e.g., delay, throughput); and (ii) using a predictive model of the application behavior to decide the best action to take (e.g., increase the flow priority). Similarly, self-driving networks could swiftly detect network problems by observing, say TCP retransmissions, and reroute traffic upon detecting statistical anomalies [31]. Self-driving networks could also improve security by classifying traffic – even if it is encrypted – or by detecting subtle DDoS attacks.

All these applications require network devices to support two key building blocks: (i) the ability to derive precise measurements; and (ii) the ability to perform complex inference—both, directly in the data plane. While this might seem unrealistic, the advent of fully programmable data planes (e.g., Barefoot Tofino [2], Netronome NICs [50]) offers us an opportunity to implement such features. The question is though: *Are programmable data planes powerful enough?*

***p*Forest** We answer this question positively by describing *p*Forest, a system which enables programmable data planes to perform real-time inference, accurately and at scale, according to supervised machine learning models. *p*Forest takes as input a labeled dataset (e.g., an annotated traffic trace) and automatically trains a P4-based [21] online classifier that can run directly in the data plane (on existing hardware) and infer labels on live traffic. Despite being performed in the data plane, *p*Forest inference is accurate—as accurate as if it was done in software using state-of-the-art machine learning frameworks [11]. As an online classifier, *p*Forest further optimizes for the classification “speed”, i.e. it classifies flows as early as possible (after few packets).

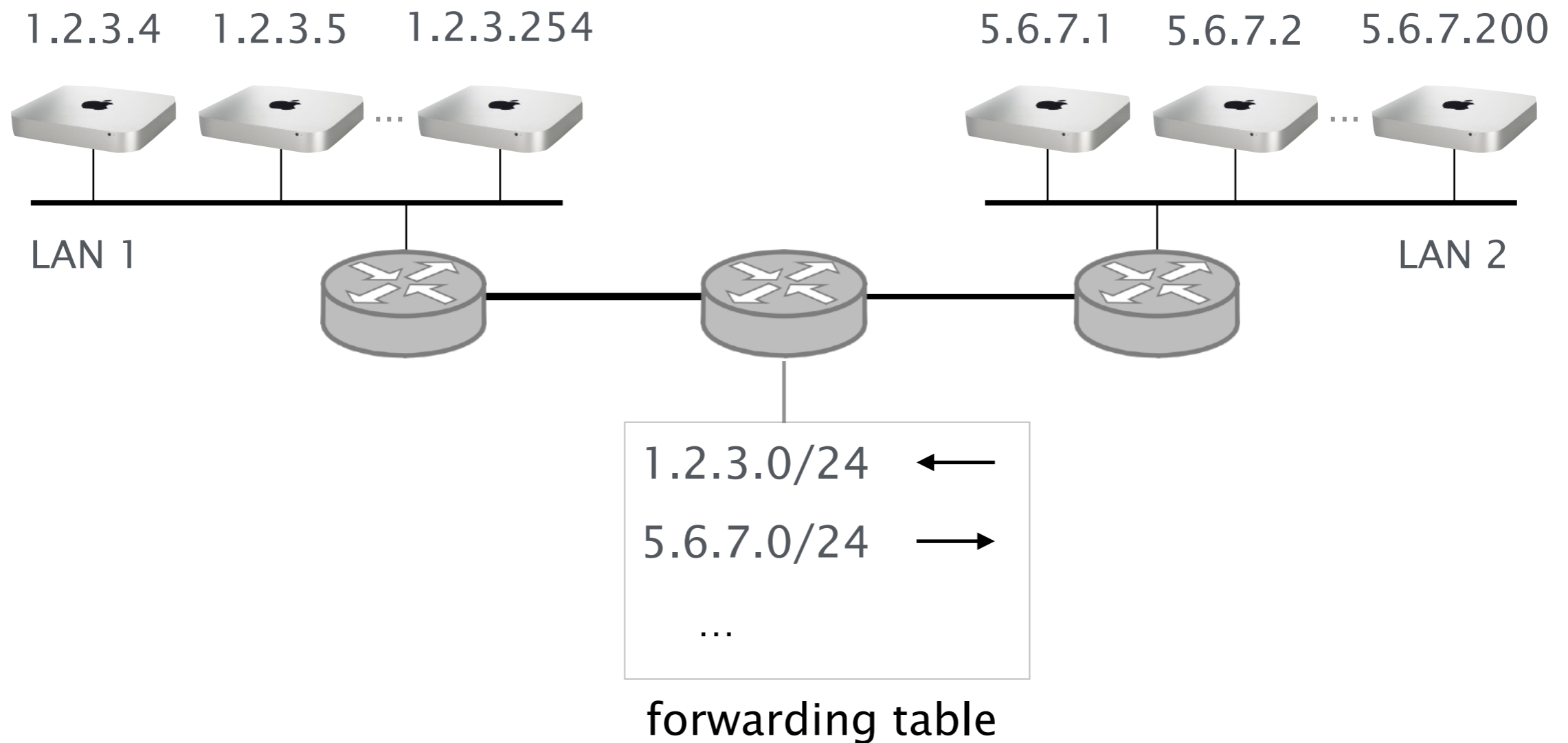
We stress that *p*Forest is a general framework that enables to perform in-network inference. As such, it does *not* remove the need to obtain a representative training dataset. As for any machine learning model, poor input data will result in poor performance. We consider the problem of building a representative dataset as orthogonal to this paper.

Your first



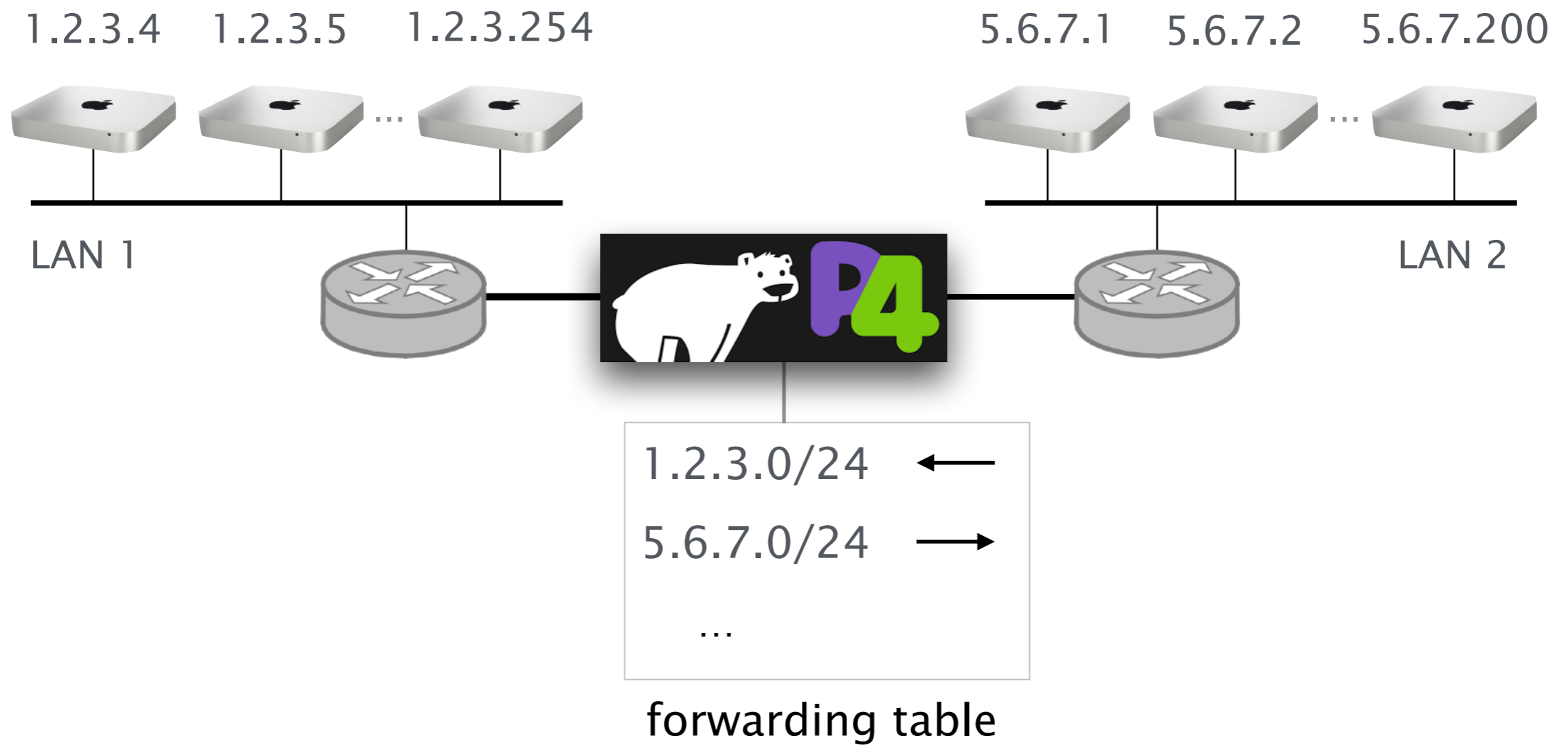
program

IP forwarding in a traditional router



IP forwarding

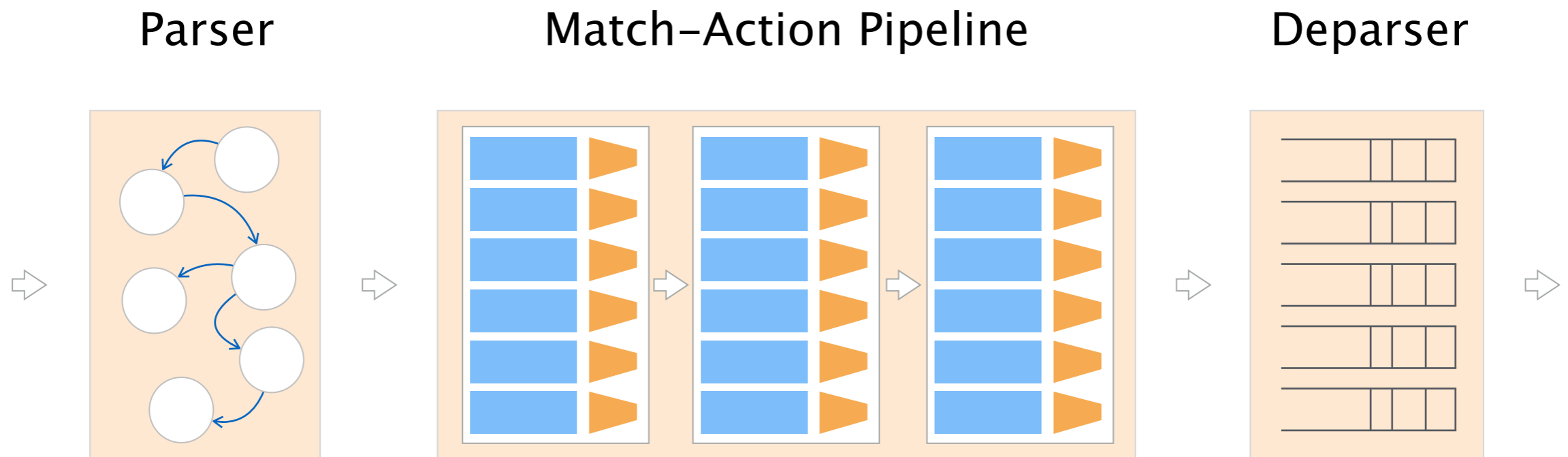
in P4?



An IP router has
four main ingredients

- Lookup a forwarding table
- Update the destination MAC
- Decrement TTL
- Forward packets to output ports

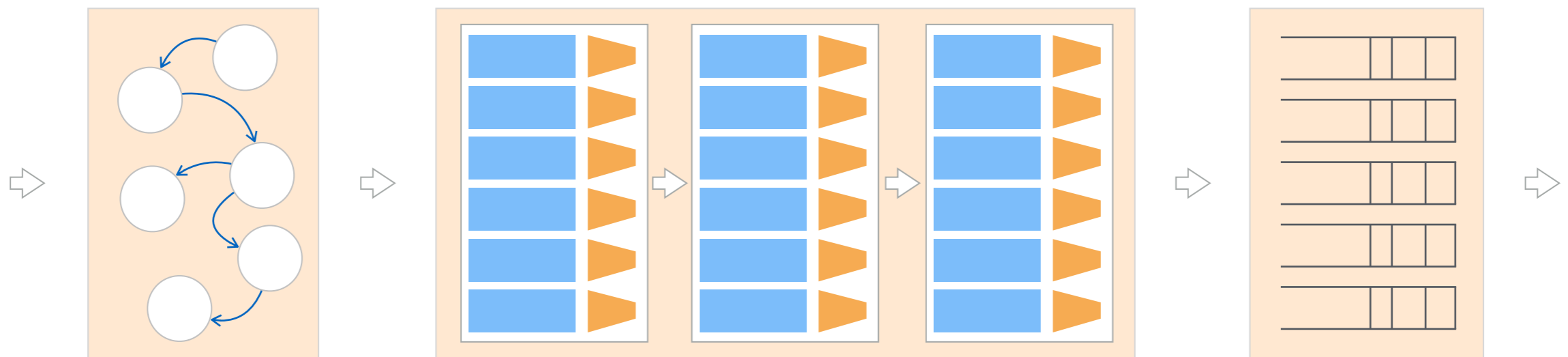
A P4 program consists of three basic parts



Parser

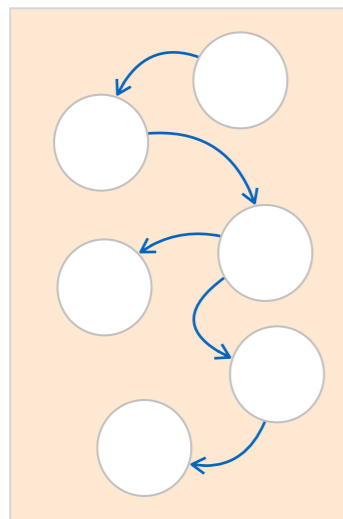
Match-Action Pipeline

Deparser

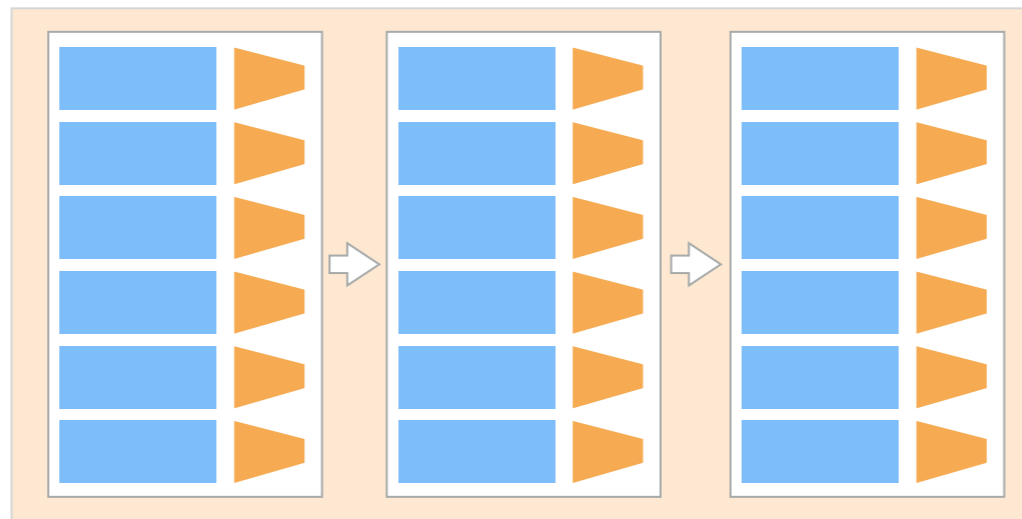


Programmer declares the headers
that should be recognized
and their order in the packet

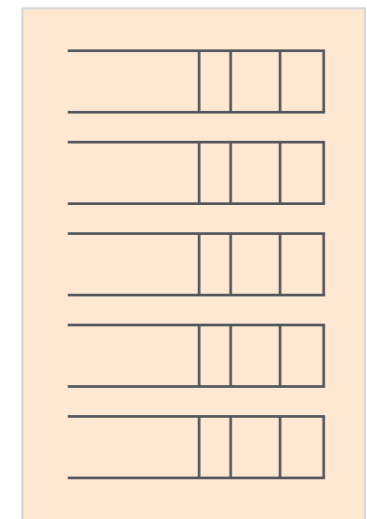
Parser



Match-Action Pipeline

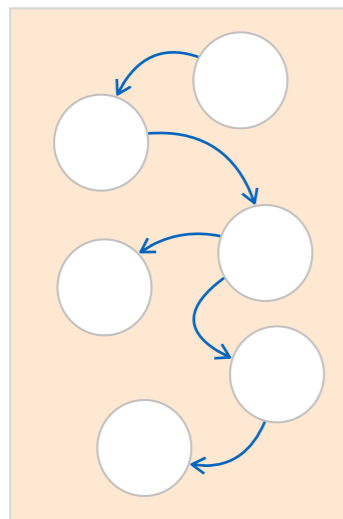


Deparser

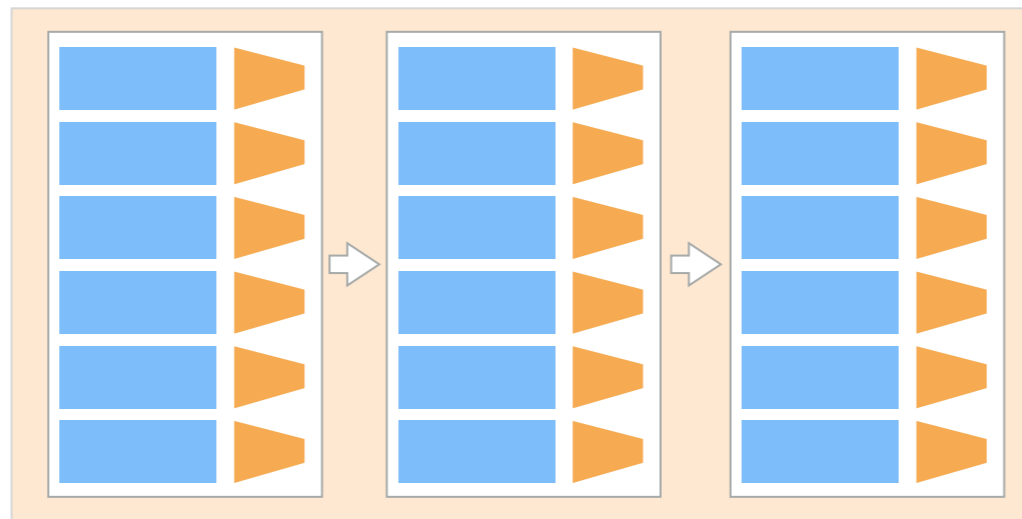


Programmer defines the tables
and the processing logic

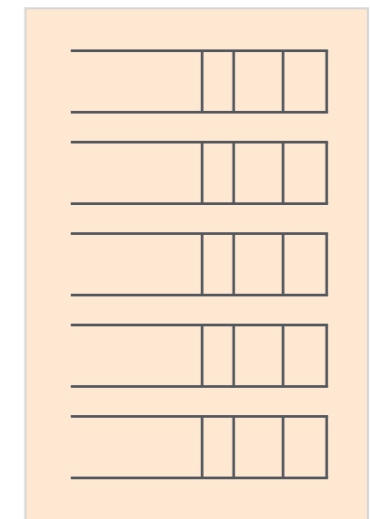
Parser



Match-Action Pipeline

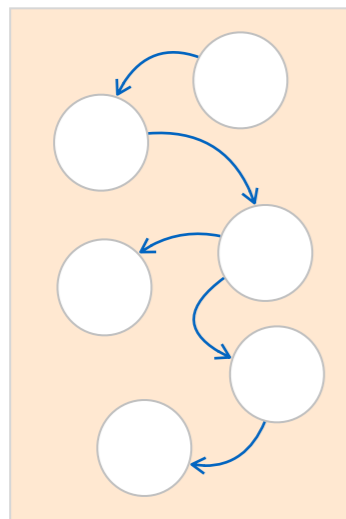


Deparser

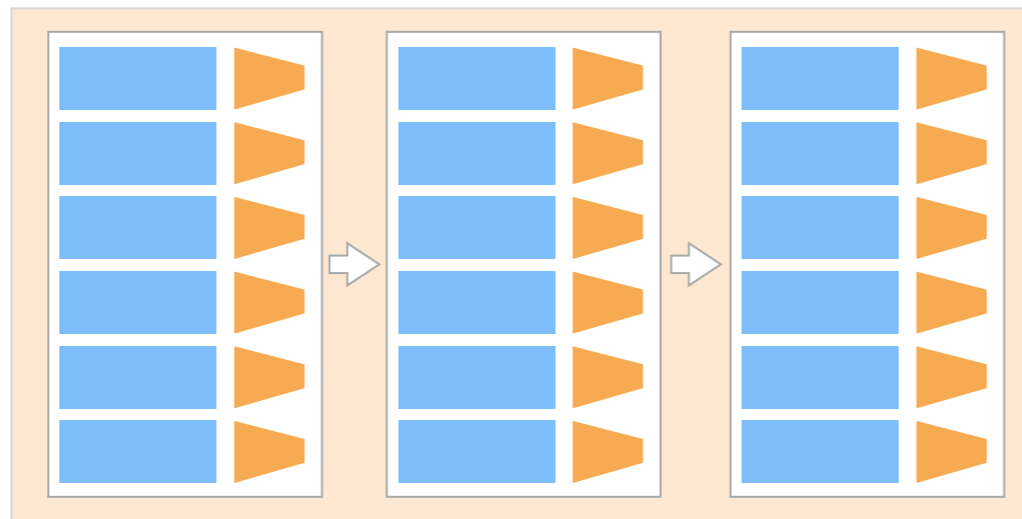


Programmer declares
how the output packet
will look on the wire

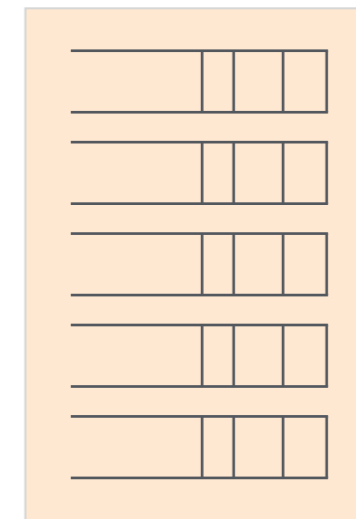
Parser



Match-Action Pipeline



Deparser



```
v1Switch(  
  MyParser(),  
  MyVerifyChecksum(),  
  MyIngress(),  
  MyEgress(),  
  MyComputeChecksum(),  
  MyDeparser()  
) main;
```




```
#include <core.p4>
#include <v1model.p4>
```

Libraries

```
const bit<16> TYPE_IPV4 = 0x800;
typedef bit<32> ip4Addr_t;
header ipv4_t {...}
struct headers {...}
```

Declarations

```
parser MyParser(...) {
    state start {...}
    state parse_ethernet {...}
    state parse_ipv4 {...}
}
```

Parse packet headers

```
control MyIngress(...) {
    action ipv4_forward(...) {...}
    table ipv4_lpm {...}
    apply {
        if (...) {...}
    }
}
```

Control flow
to modify packet

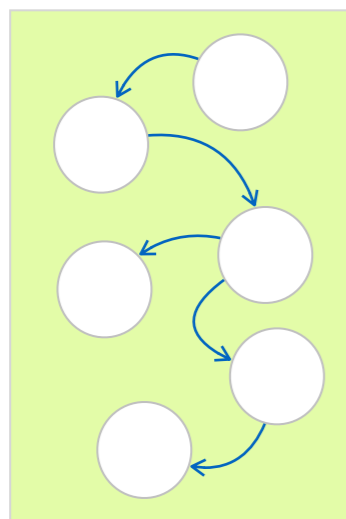
```
control MyDeparser(...) {...}
```

Assemble
modified packet

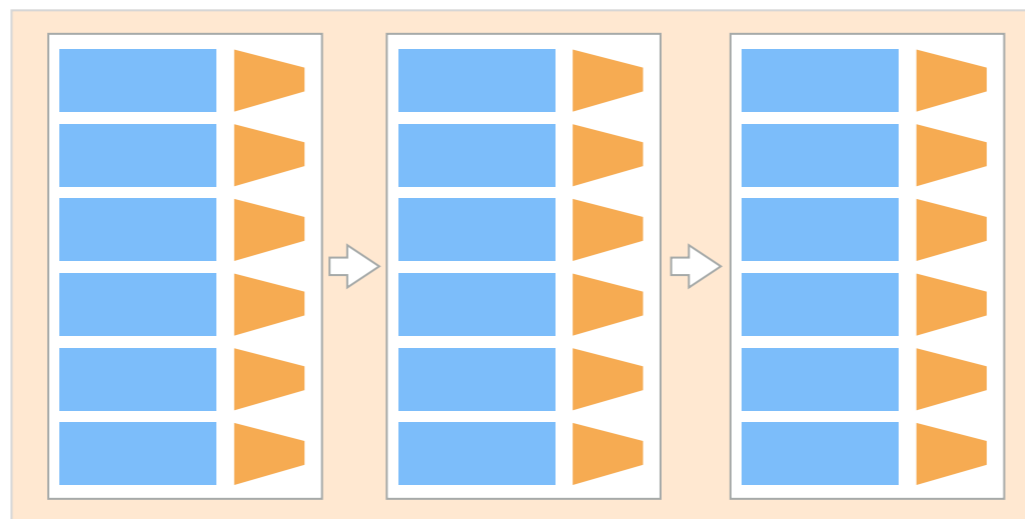
```
V1Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```

“main()”

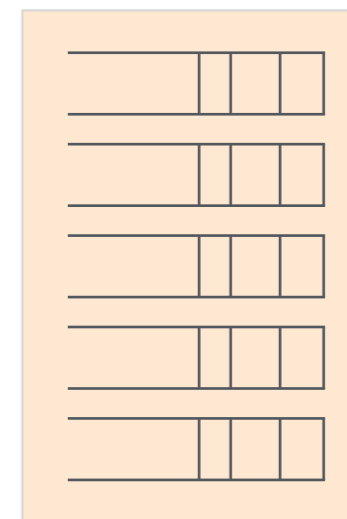
Parser



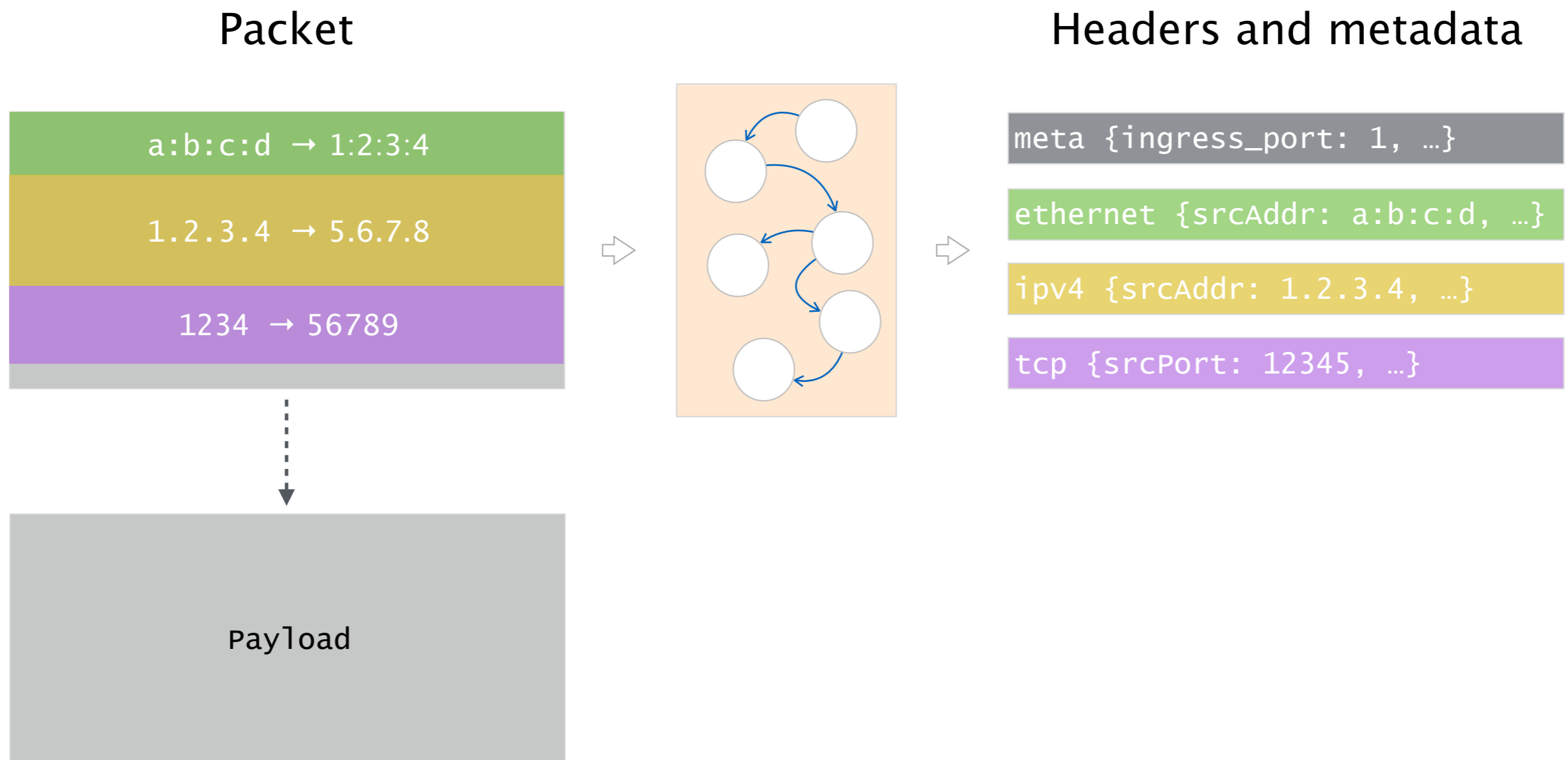
Match-Action Pipeline



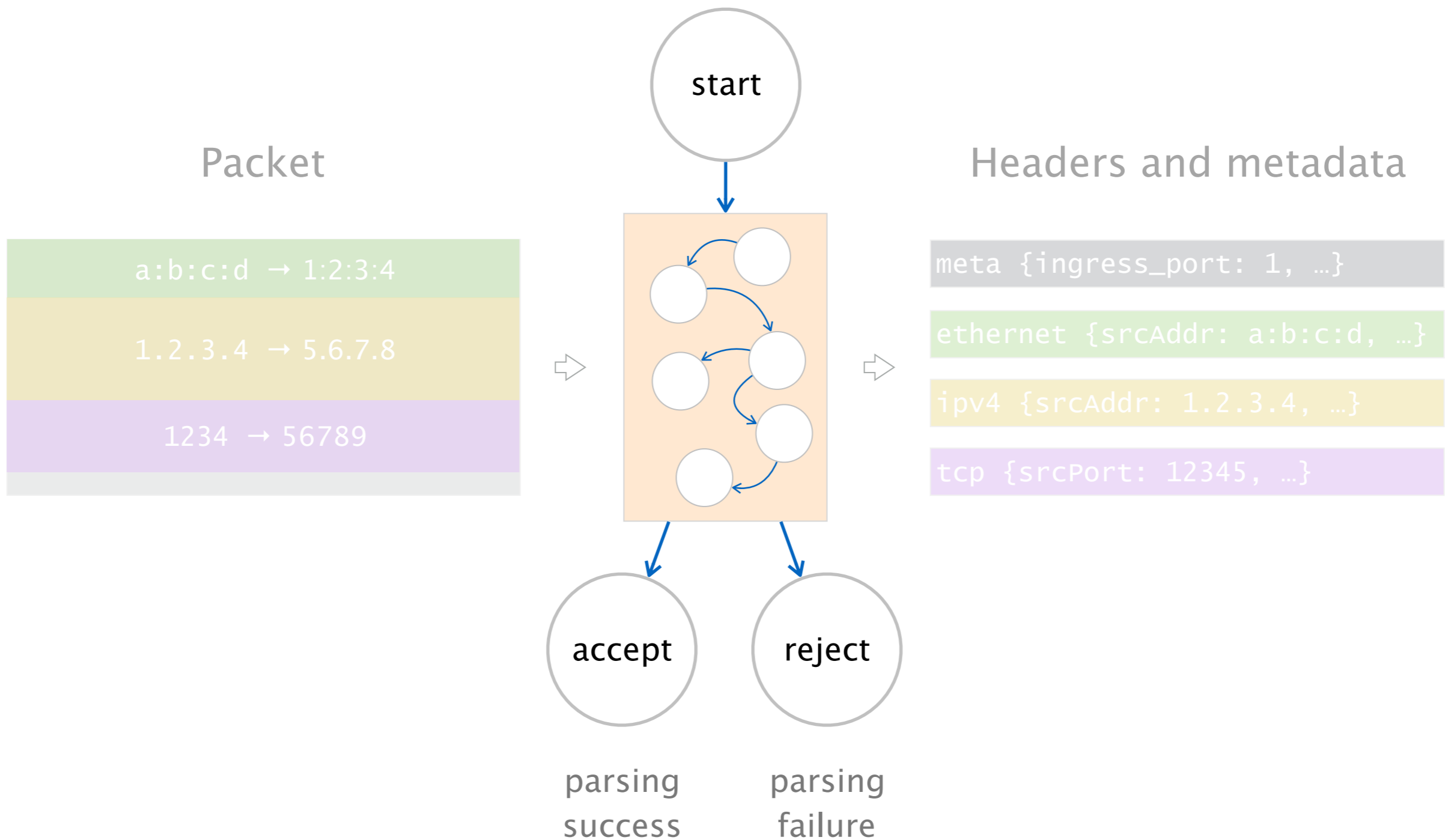
Deparser

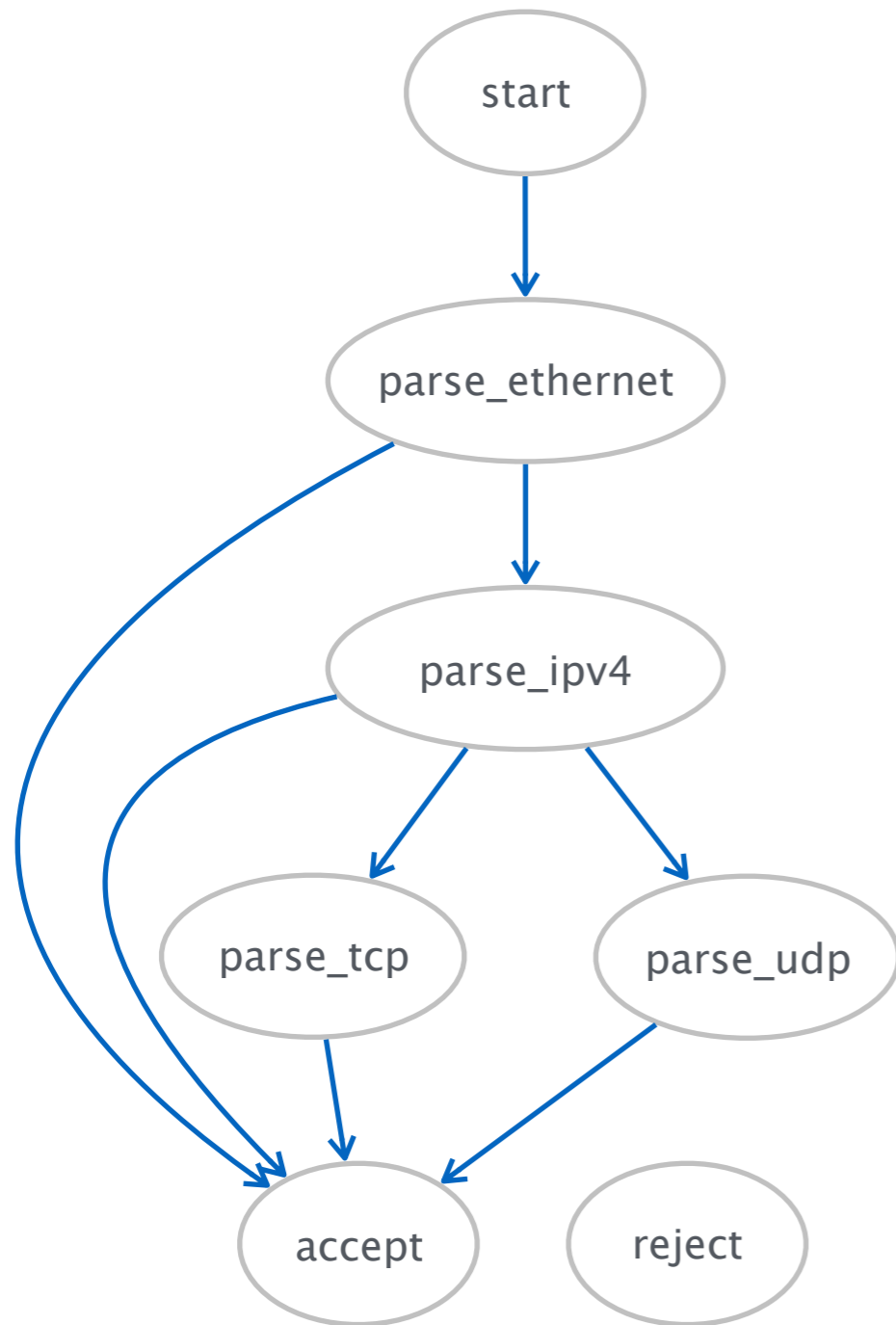


The parser uses a state machine to map packets into headers and metadata



The parser has three predefined states: start, accept and reject





```

parser MyParser(...) {
  state start {
    transition parse_ethernet;
  }

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

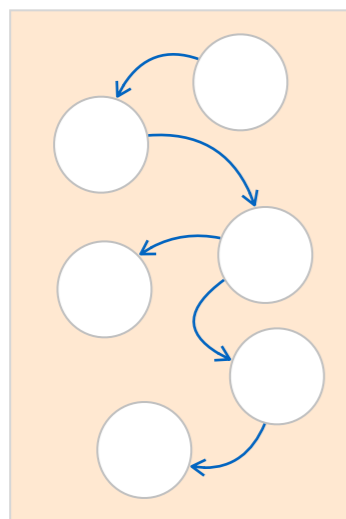
  state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol) {
      6: parse_tcp;
      17: parse_udp;
      default: accept;
    }
  }

  state parse_tcp {
    packet.extract(hdr.tcp);
    transition accept;
  }

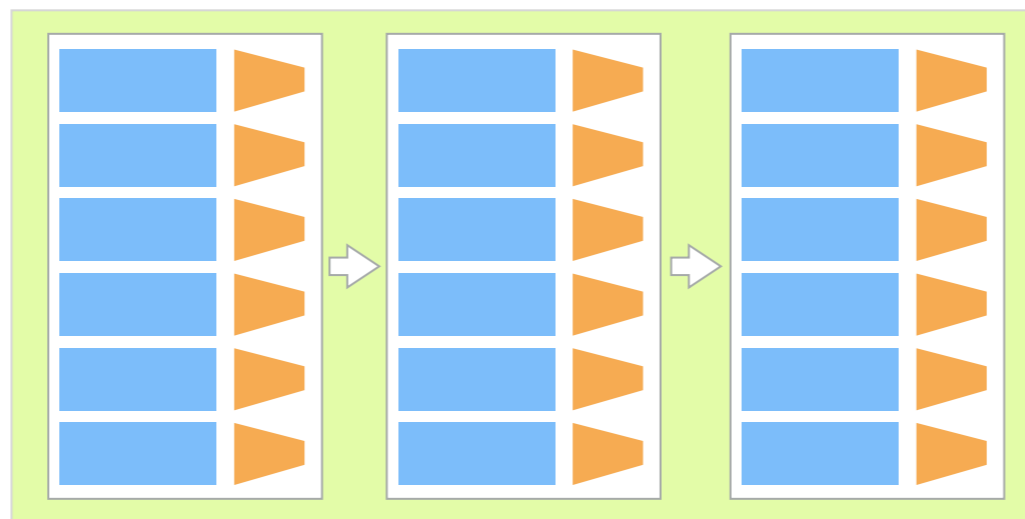
  state parse_udp {
    packet.extract(hdr.udp);
    transition accept;
  }
}

```

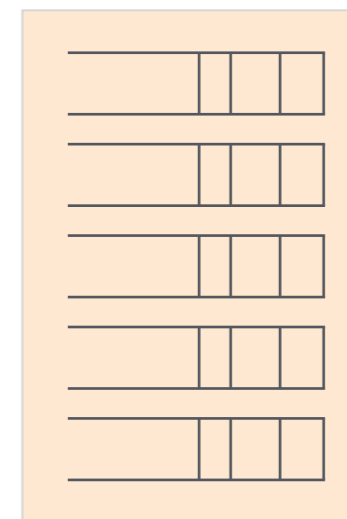
Parser



Match-Action Pipeline



Deparser



Basic building blocks of P4 programs

Control

Control flow

describes how packets should be processed

Actions

fragments manipulating headers fields/metadata

Tables

map user-defined keys with actions

Control

Control flow

describes how packets should be processed

Actions

fragments manipulating headers fields/metadata

Tables

map user-defined keys with actions

Control flow expresses an imperative program which describes how packets are processed

Headers and metadata from parser

```
control MyIngress(inout headers hdr,  
                 inout metadata meta,  
                 inout standard_metadata_t std_meta) {
```

```
    bit<9> port;
```

Variable declaration

```
    apply {
```

```
        port = 1;  
        std_meta.egress_spec = port;  
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  
        hdr.ethernet.dstAddr = 0x2;  
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
```

```
    }  
}
```

Control flow

Control

Control flow

describes how packets should be processed

Actions

fragments manipulating headers fields/metadata

Tables

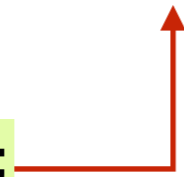
map user-defined keys with actions

Actions allow to re-use code similar to functions in C

```
control MyIngress(inout headers hdr,  
                  inout metadata meta,  
                  inout standard_metadata_t std_meta) {
```

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

```
    apply {  
        ipv4_forward(0x123, 1);  
    }  
}
```



Control

Control flow

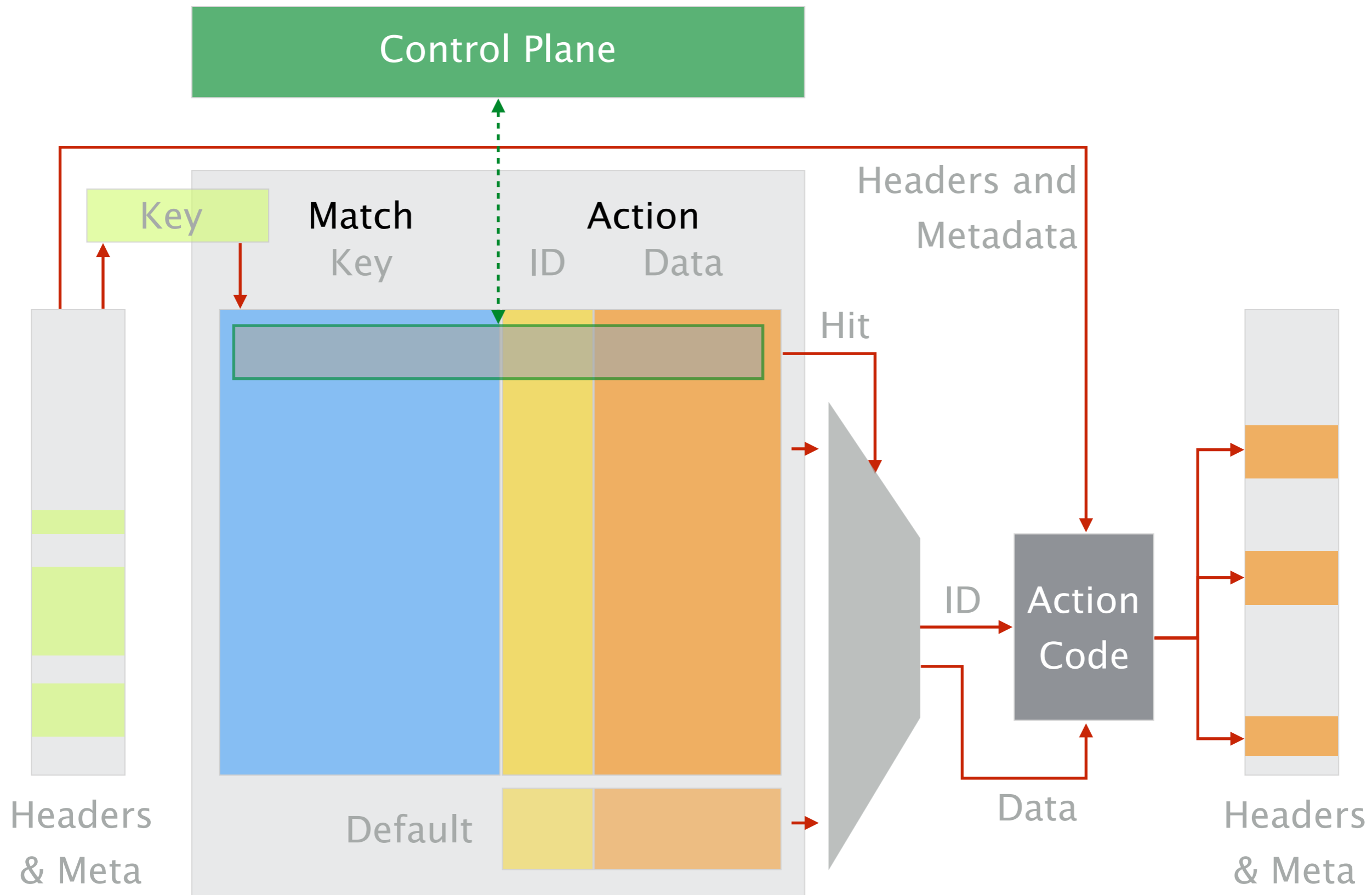
describes how packets should be processed

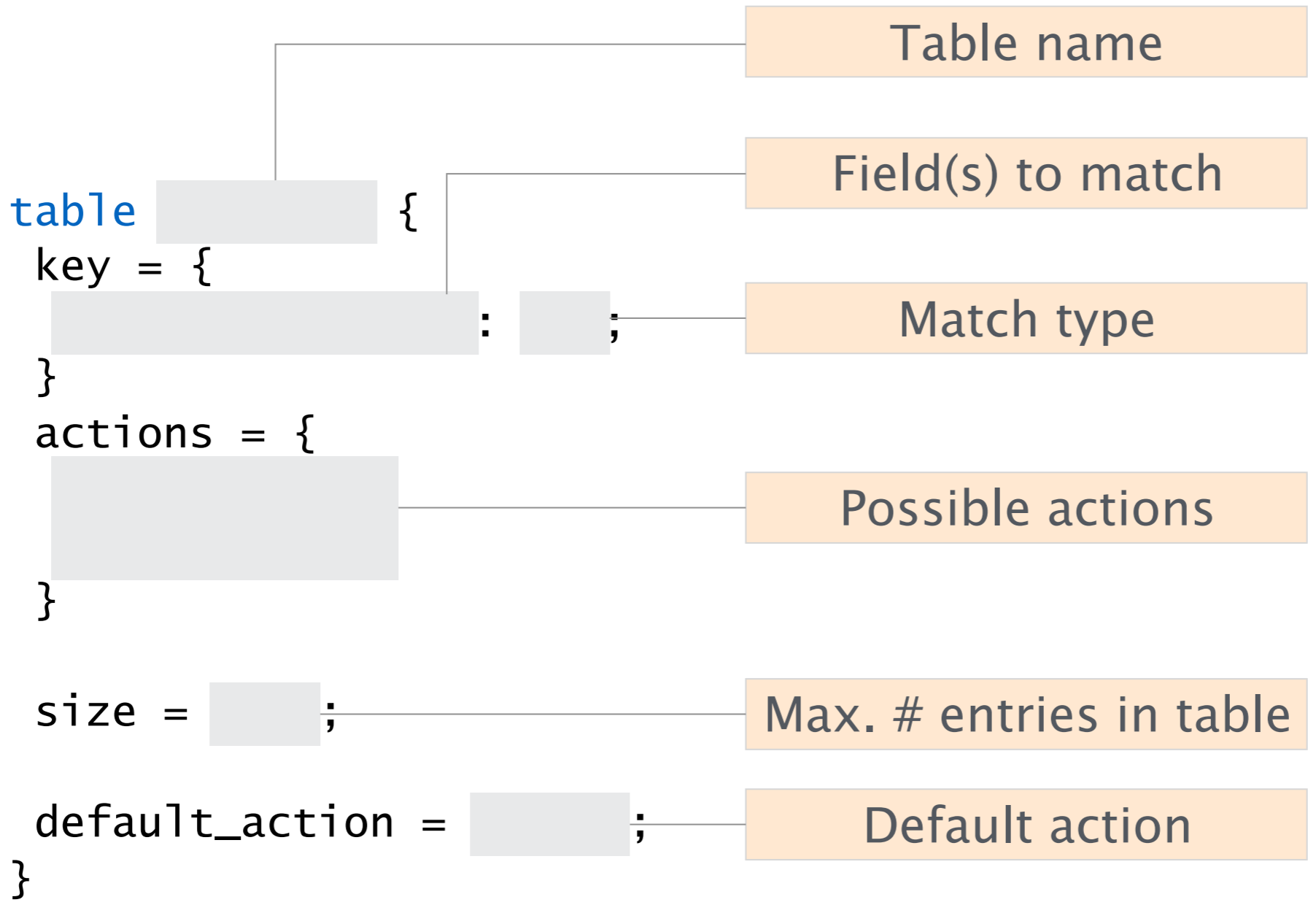
Actions

fragments manipulating headers fields/metadata

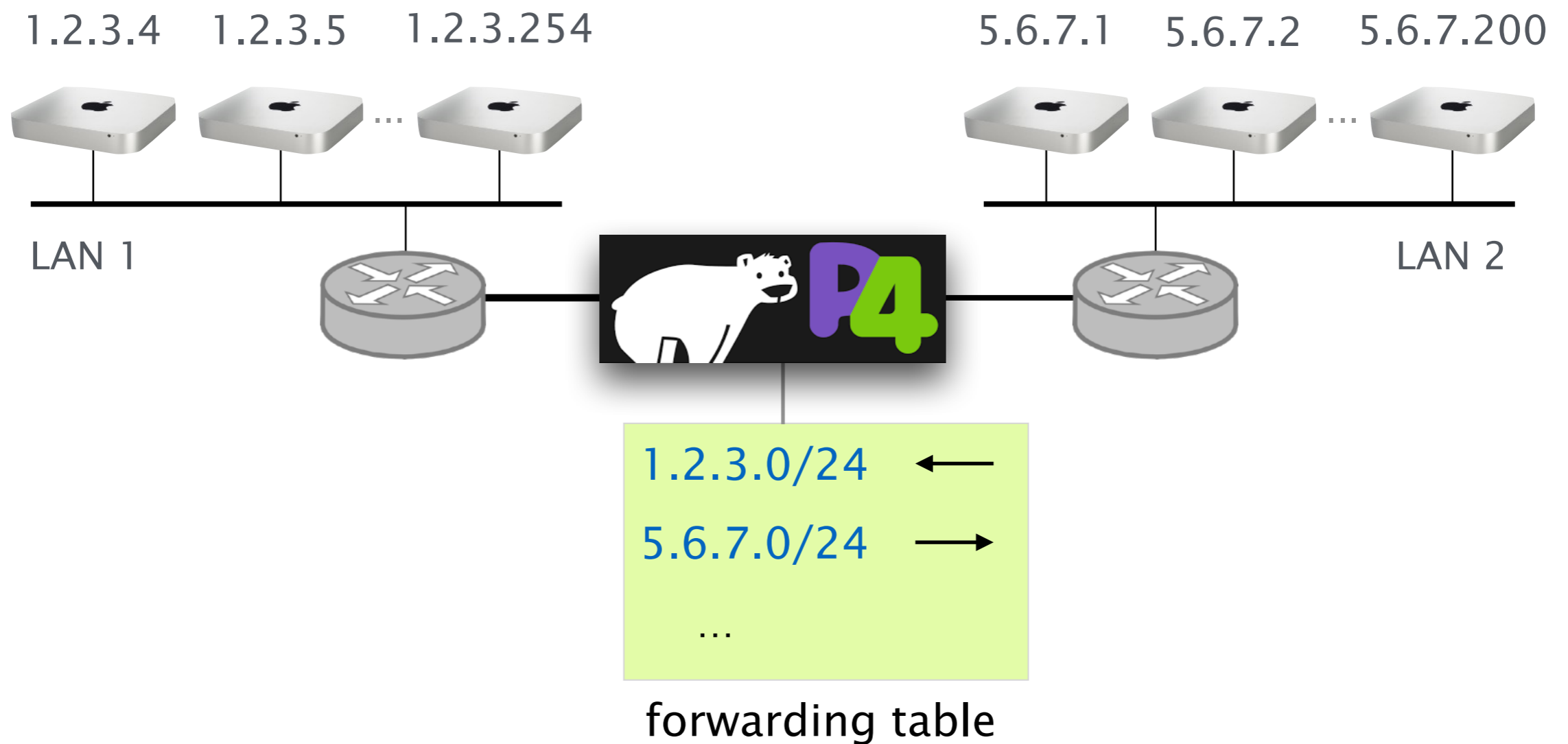
Tables

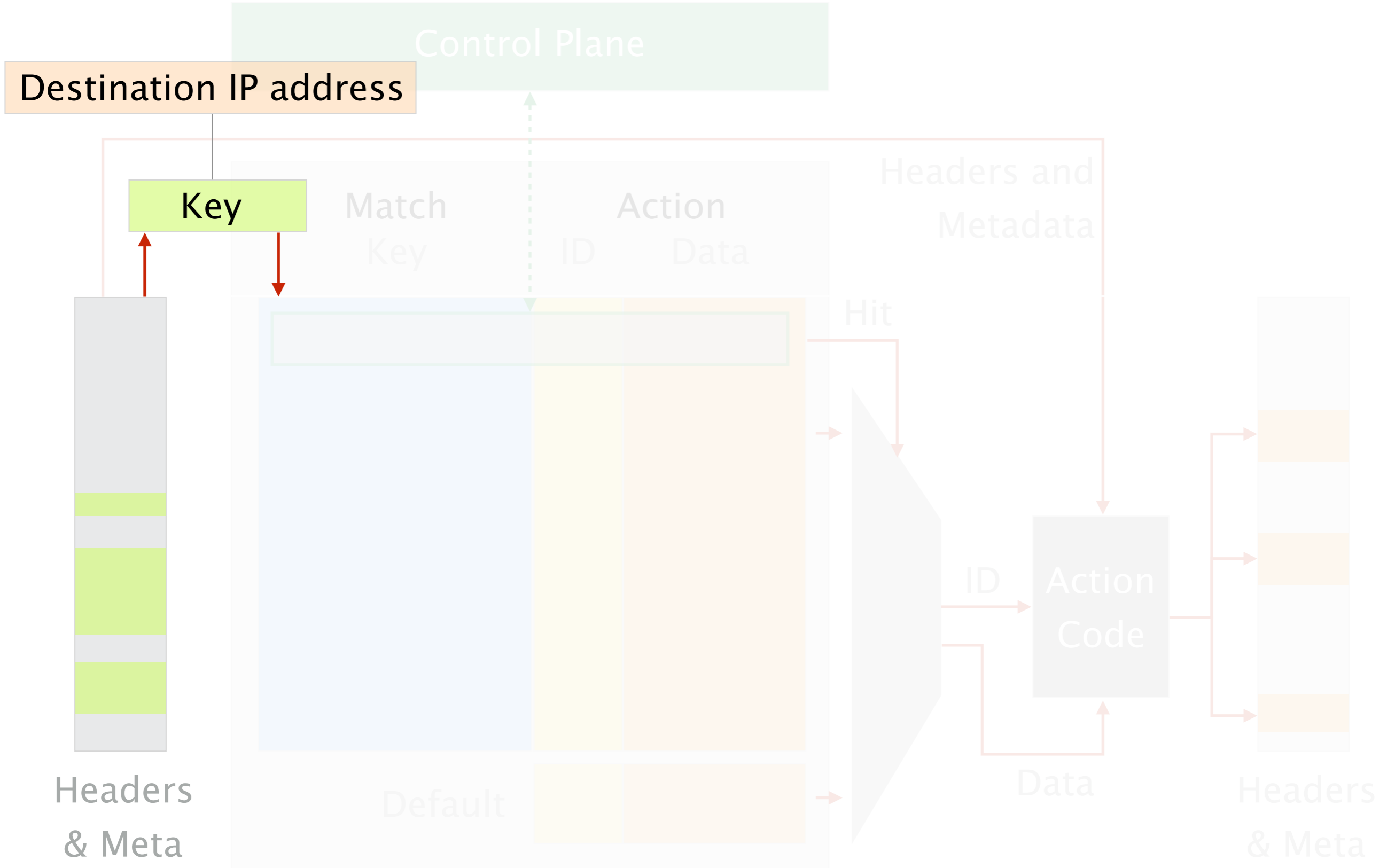
map user-defined keys with actions

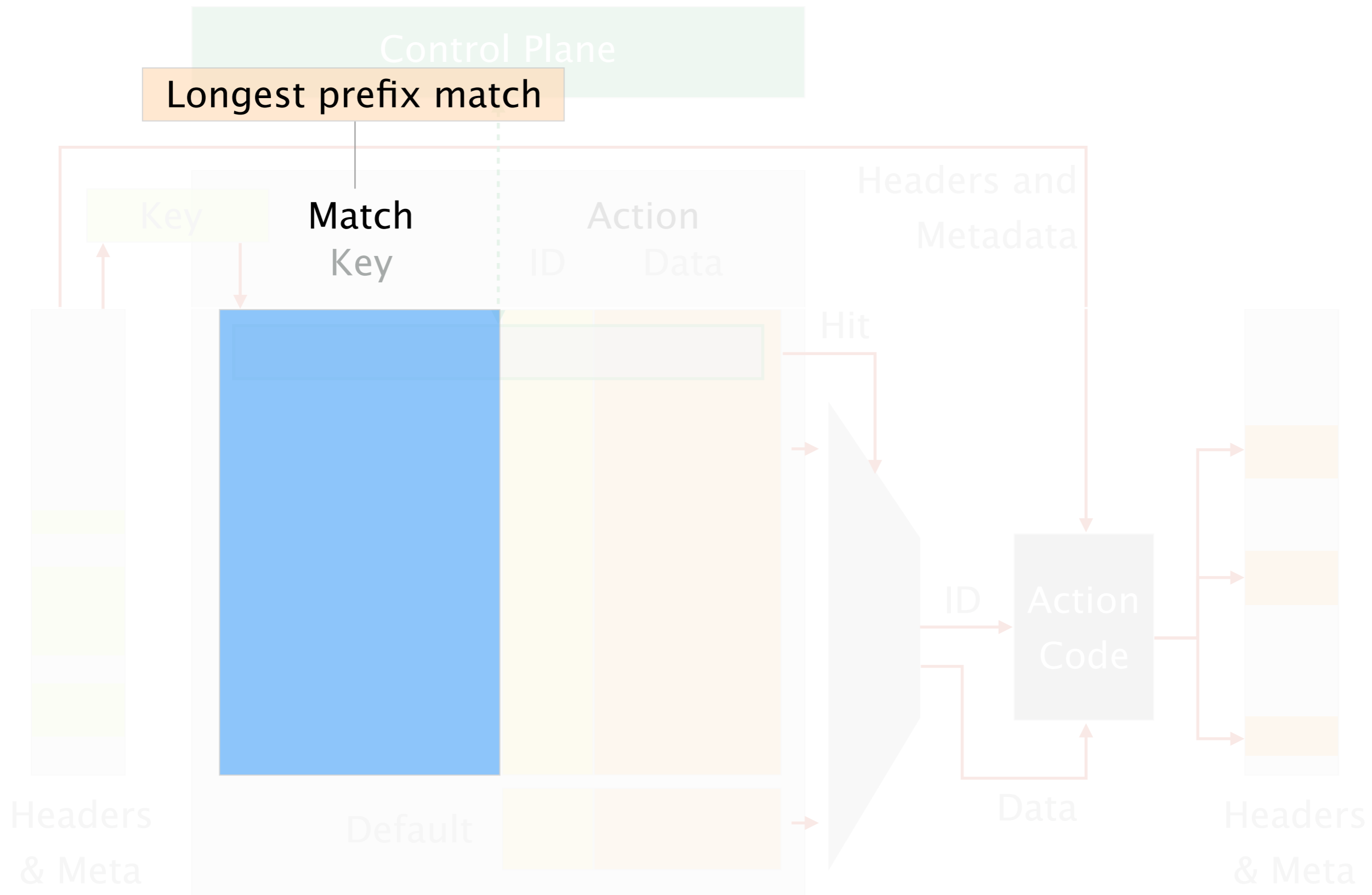




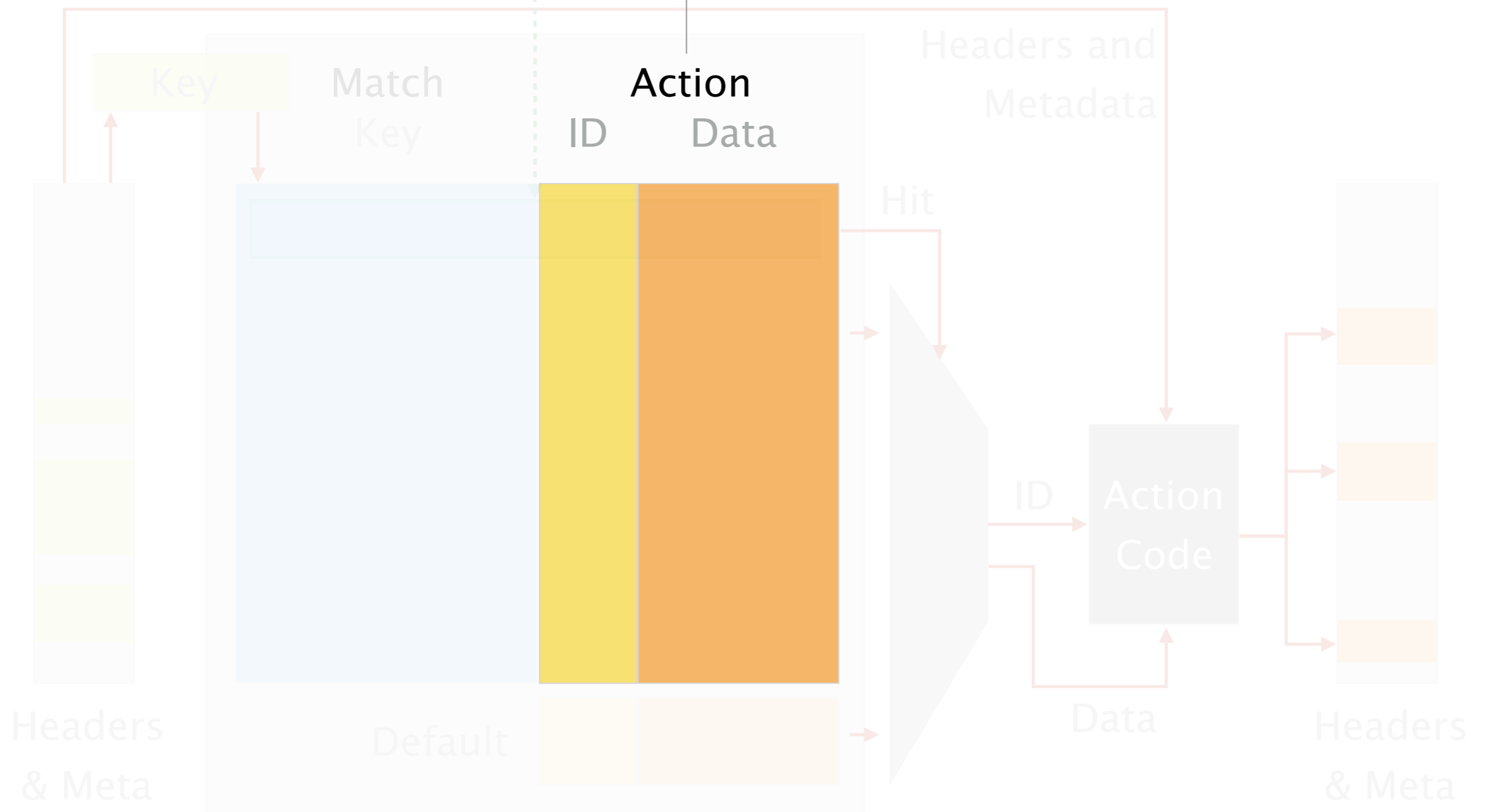
Example: IP forwarding table

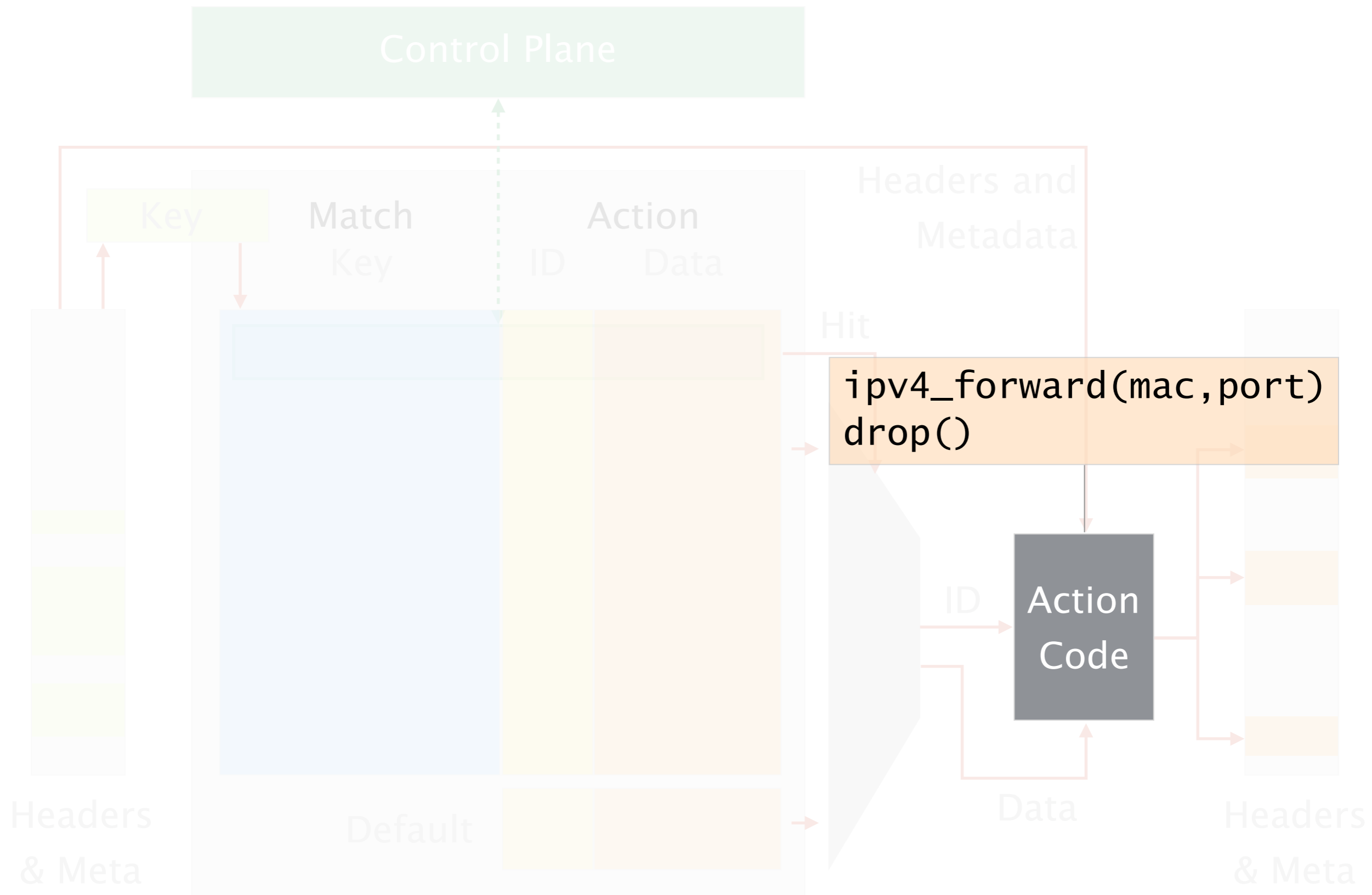


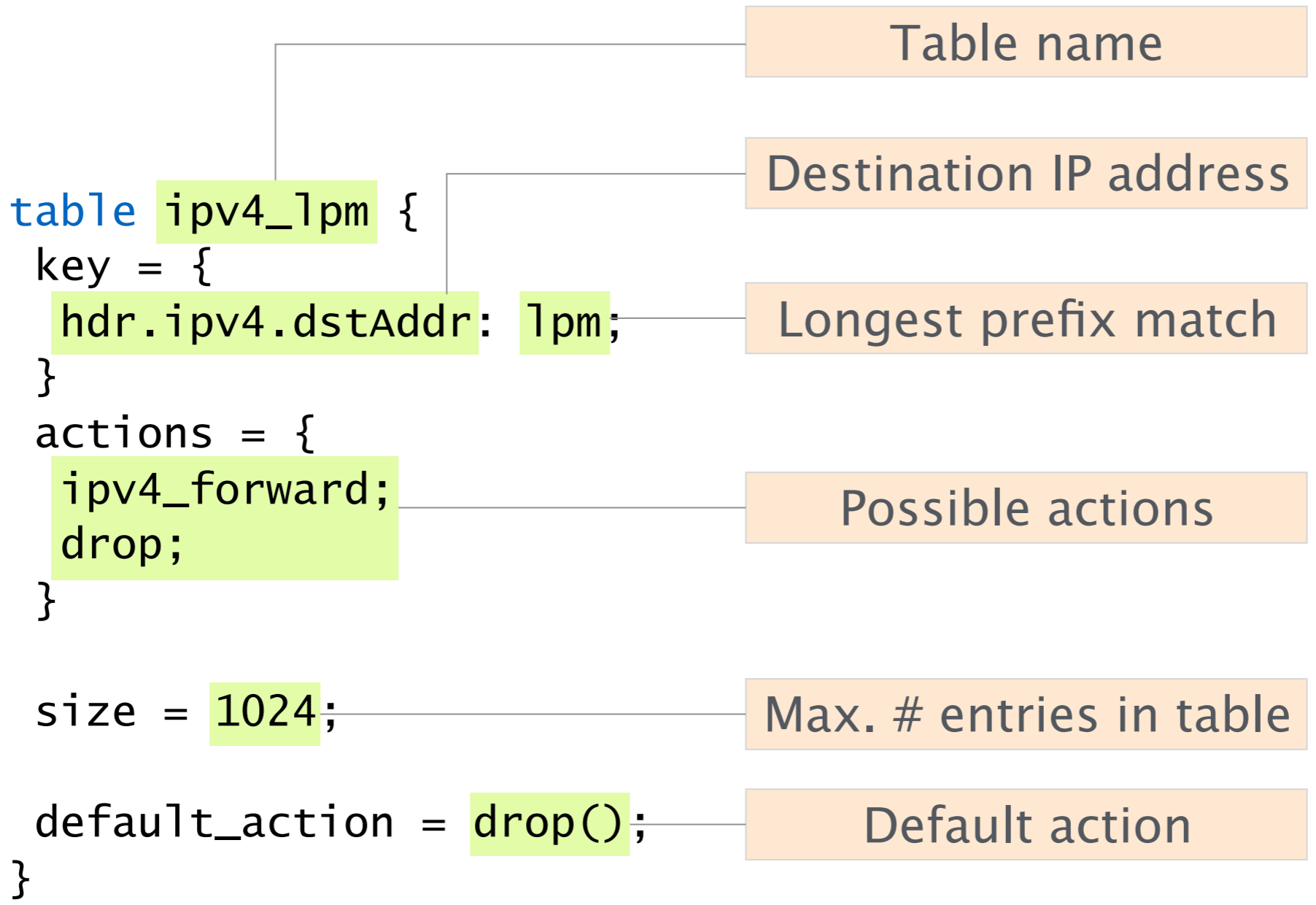




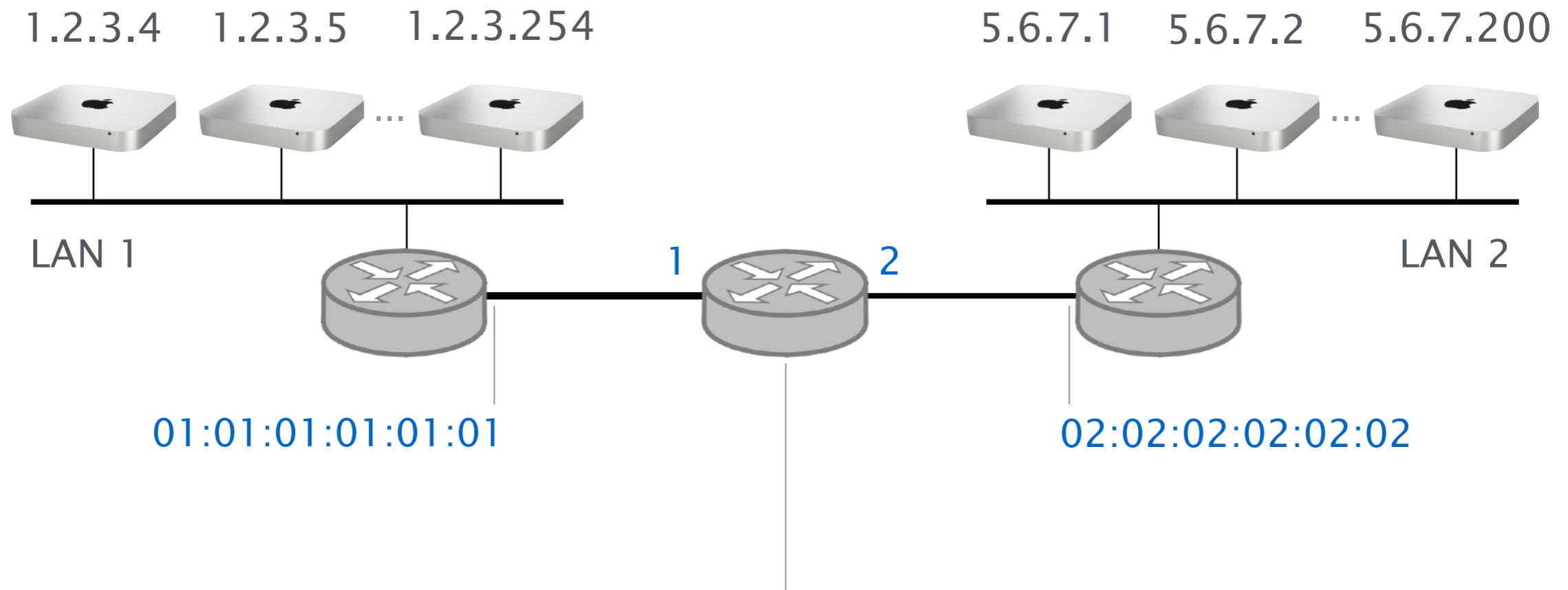
1: `ipv4_forward(mac, port)`
2: `drop()`





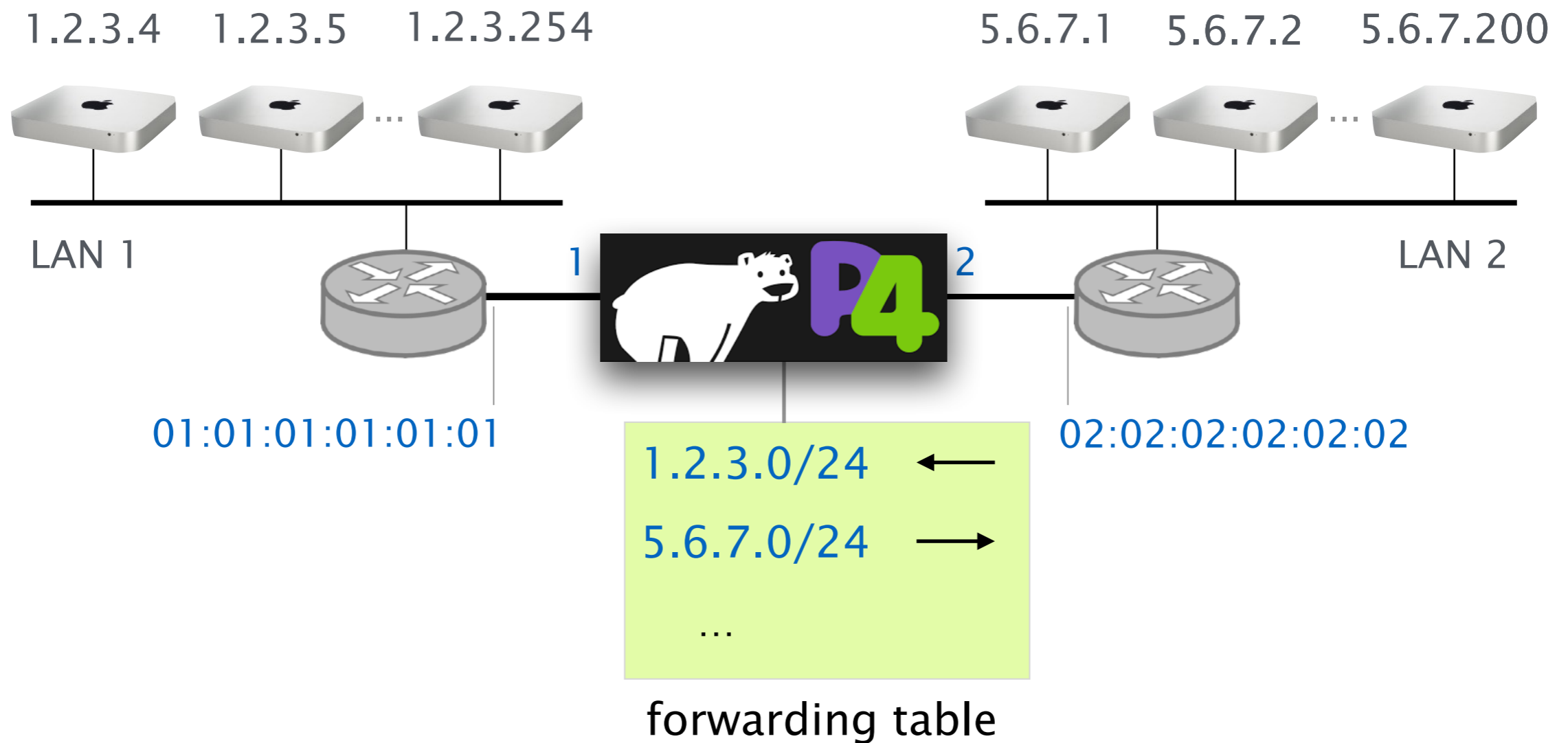


Example: IP forwarding table



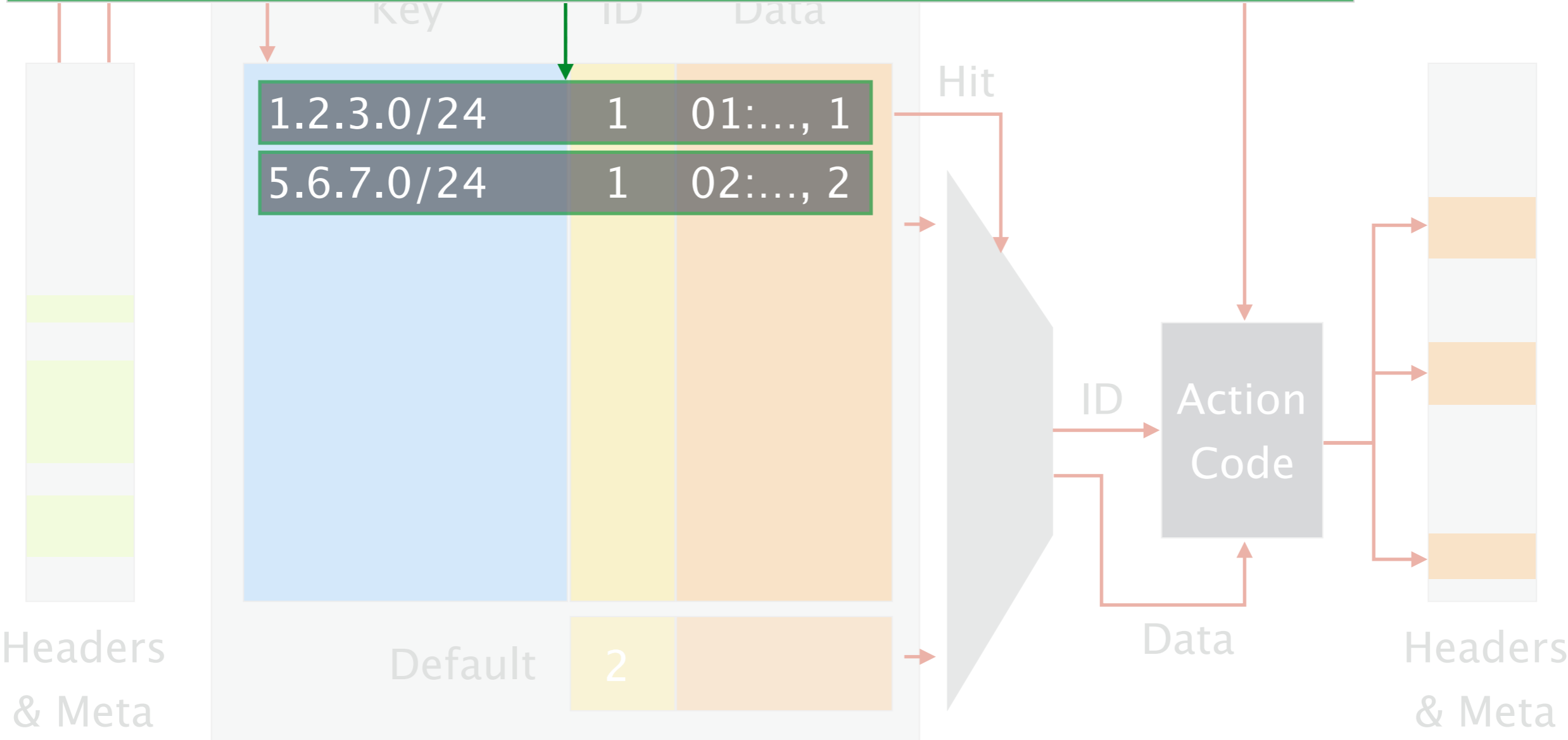
```
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {  
    standard_metadata.egress_spec = port;  
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;  
    hdr.ethernet.dstAddr = dstAddr;  
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  
}
```

Example: IP forwarding table

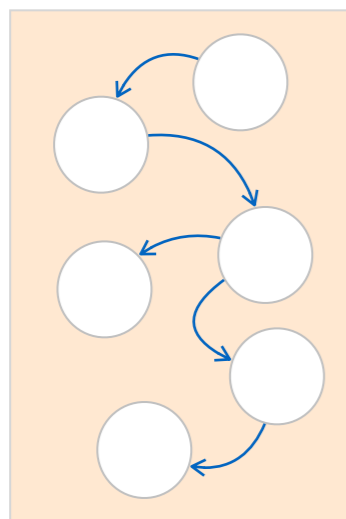


Control Plane

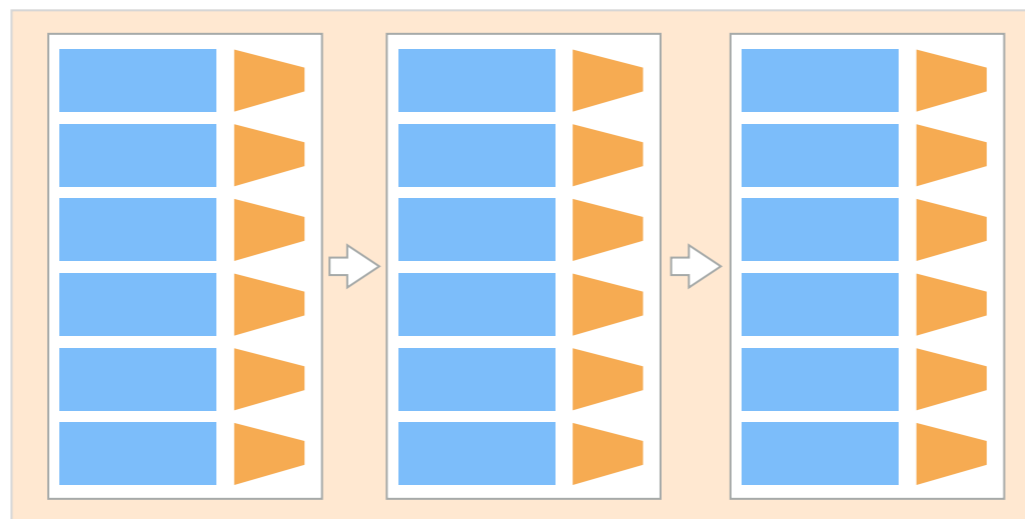
```
table_add ipv4_lpm ipv4_forward 1.2.3.0/24 => 01:01:01:01:01:01 1  
table_add ipv4_lpm ipv4_forward 5.6.7.0/24 => 02:02:02:02:02:02 2
```



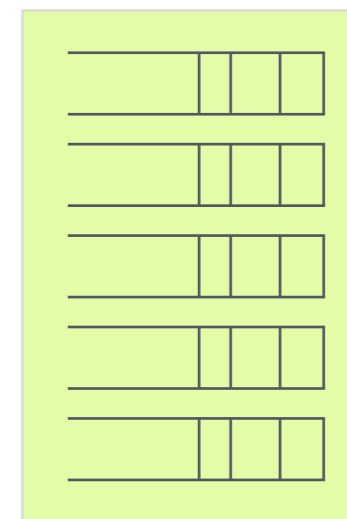
Parser



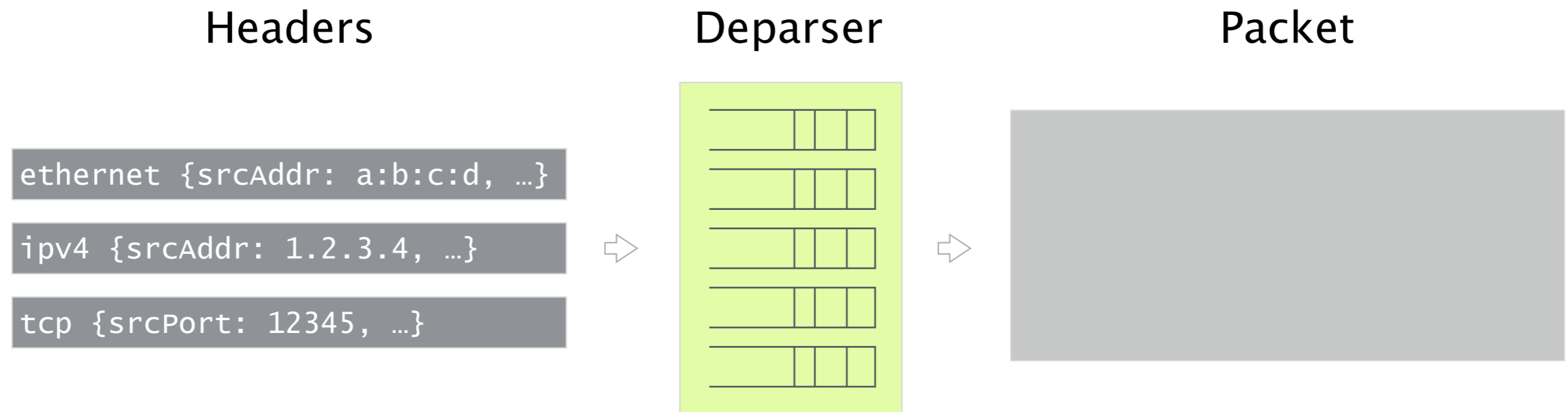
Match-Action Pipeline



Deparser



The Deparser assembles the headers back into a well-formed packet



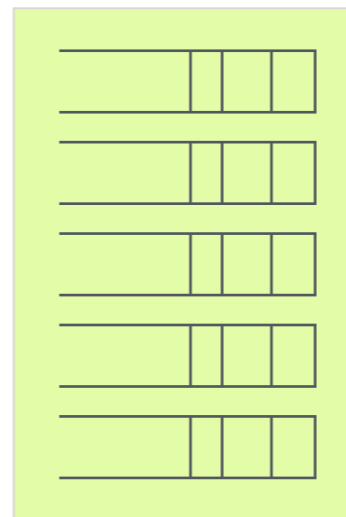
Headers

```
ethernet {srcAddr: a:b:c:d, ...}
```

```
ipv4 {srcAddr: 1.2.3.4, ...}
```

```
tcp {srcPort: 12345, ...}
```

Deparser



Packet

```
a:b:c:d → 1:2:3:4
```

```
control MyDeparser(packet_out packet, in headers hdr) {  
  apply {  
    packet.emit(hdr.ethernet);  
  }  
}
```

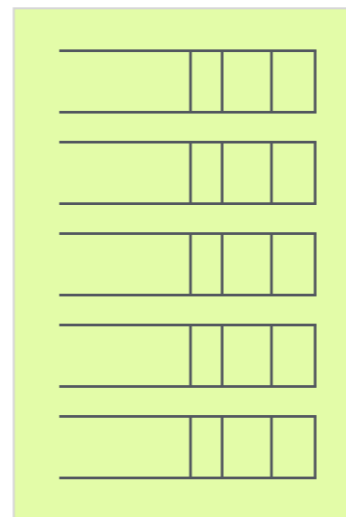
Headers

```
ethernet {srcAddr: a:b:c:d, ...}
```

```
ipv4 {srcAddr: 1.2.3.4, ...}
```

```
tcp {srcPort: 12345, ...}
```

Deparser



Packet

```
a:b:c:d → 1:2:3:4
```

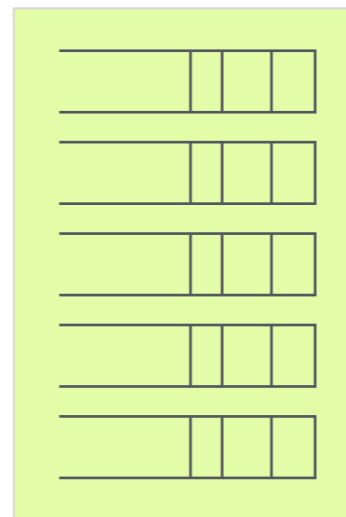
```
1.2.3.4 → 5.6.7.8
```

```
control MyDeparser(packet_out packet, in headers hdr) {  
  apply {  
    packet.emit(hdr.ethernet);  
    packet.emit(hdr.ipv4);  
  }  
}
```

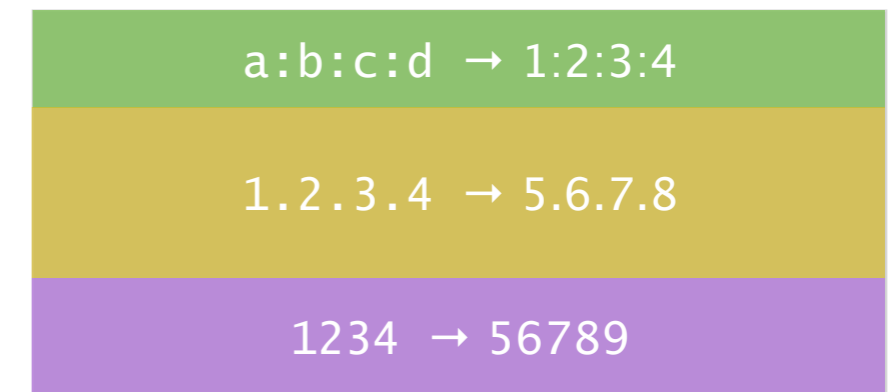
Headers

```
ethernet {srcAddr: a:b:c:d, ...}  
ipv4 {srcAddr: 1.2.3.4, ...}  
tcp {srcPort: 12345, ...}
```

Deparser



Packet

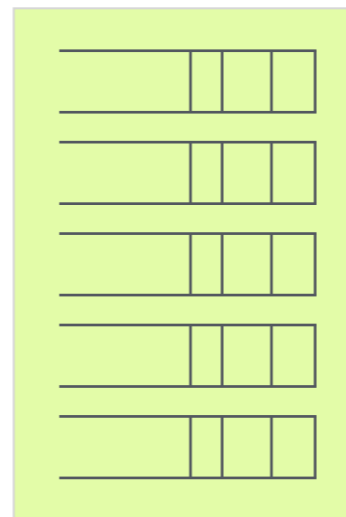


```
control MyDeparser(packet_out packet, in headers hdr) {  
  apply {  
    packet.emit(hdr.ethernet);  
    packet.emit(hdr.ipv4);  
    packet.emit(hdr.tcp);  
  }  
}
```

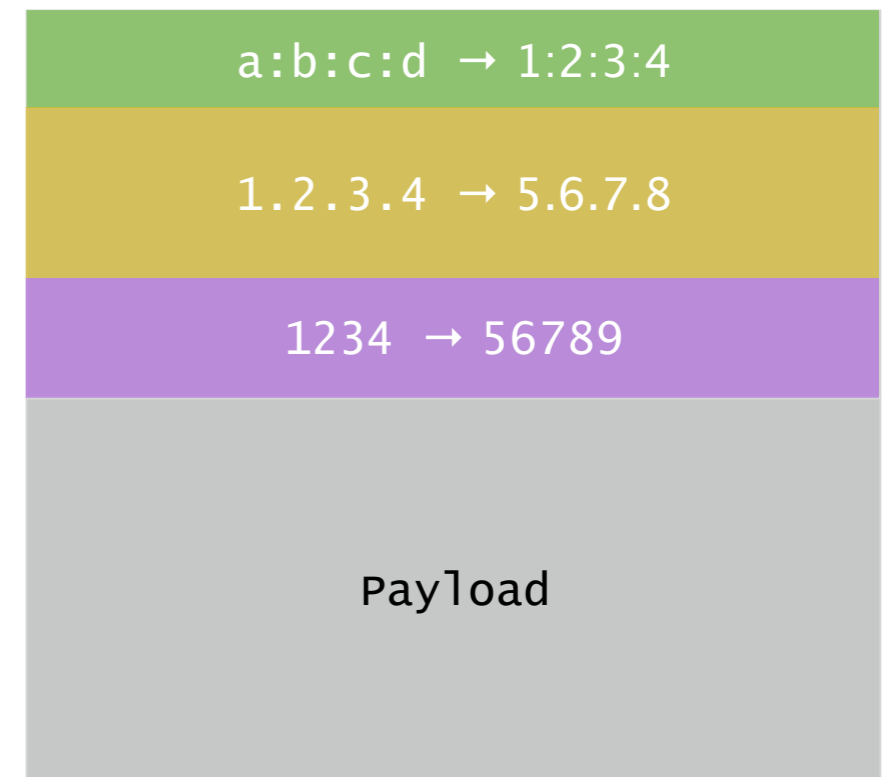
Headers

ethernet {srcAddr: a:b:c:d, ...}
ipv4 {srcAddr: 1.2.3.4, ...}
tcp {srcPort: 12345, ...}

Deparser

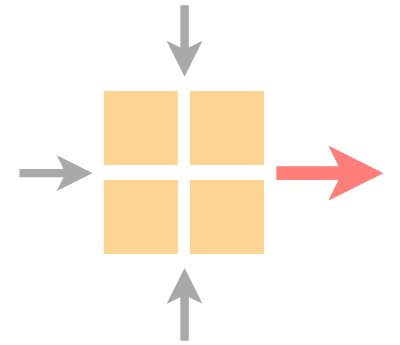


Packet



Advanced Topics in Communication Networks

Programming Network Data Planes



Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

24 Sep 2019