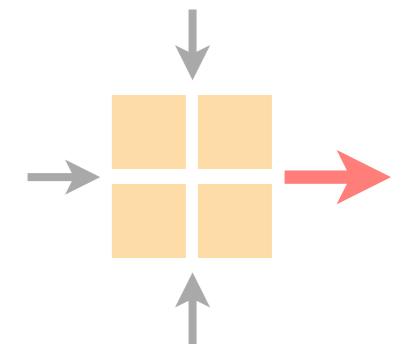


Advanced Topics in Communication Networks

Programming Network Data Planes



Laurent Vanbever
nsg.ee.ethz.ch

ETH Zürich
Sep 20 2018

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

[Tweet](#) | [Share](#) | [+1](#) | [Share](#) | [Print](#)

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.



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



Scott Raynovich, June 10, 2014

[Twitter](#) [LinkedIn](#) [Facebook](#) [Google+](#) [Email](#)



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](#)

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

Laurent

Software Development • Cloud Computing • Machine Learning • Analytics • Insider Exclusives • Resources & White Papers • Job Search

InfoWorld FROM IDG

INSIDER Sign In | Register

Home > Networking

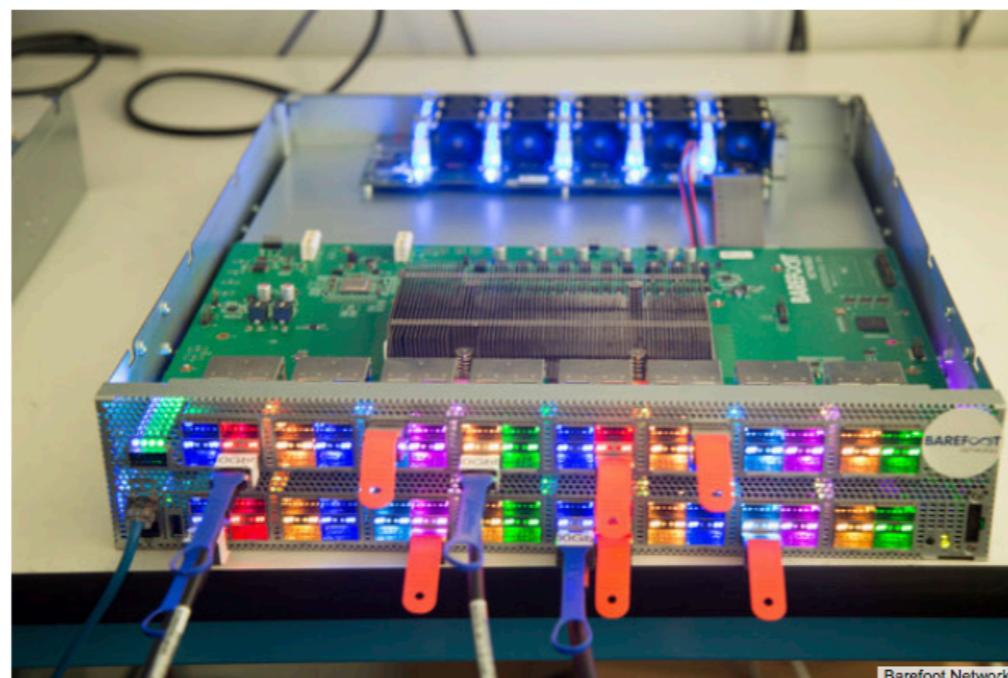
This startup may have built the world's fastest networking switch chip

Barefoot Networks is also making its switch platform completely programmable



By [Stephen Lawson](#)

Senior U.S. Correspondent, IDG News Service | JUN 15, 2016



Networking has undergone radical changes in the past few years, and two startup launches this week show the revolution isn't over yet.

Barefoot Networks is making what it calls a fully programmable switch platform. It came out of stealth mode on Tuesday, the same day 128 Technology emerged claiming a new approach to routing. Both say they're rethinking principles that haven't changed since the 1990s.

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



VIDEO
Highlights from Interop 2015

Network programmability is getting traction in many academic communities

Networking	Systems	Distributed Algorithms	Security	PL
SIGCOMM	OSDI	PODC	CCS	PLDI
NSDI	SOSP	DISC	NDSS	POPL
HotNets	SOCC		Usenix Security	OOPSLA
CoNEXT			S&P	

>7.7k

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

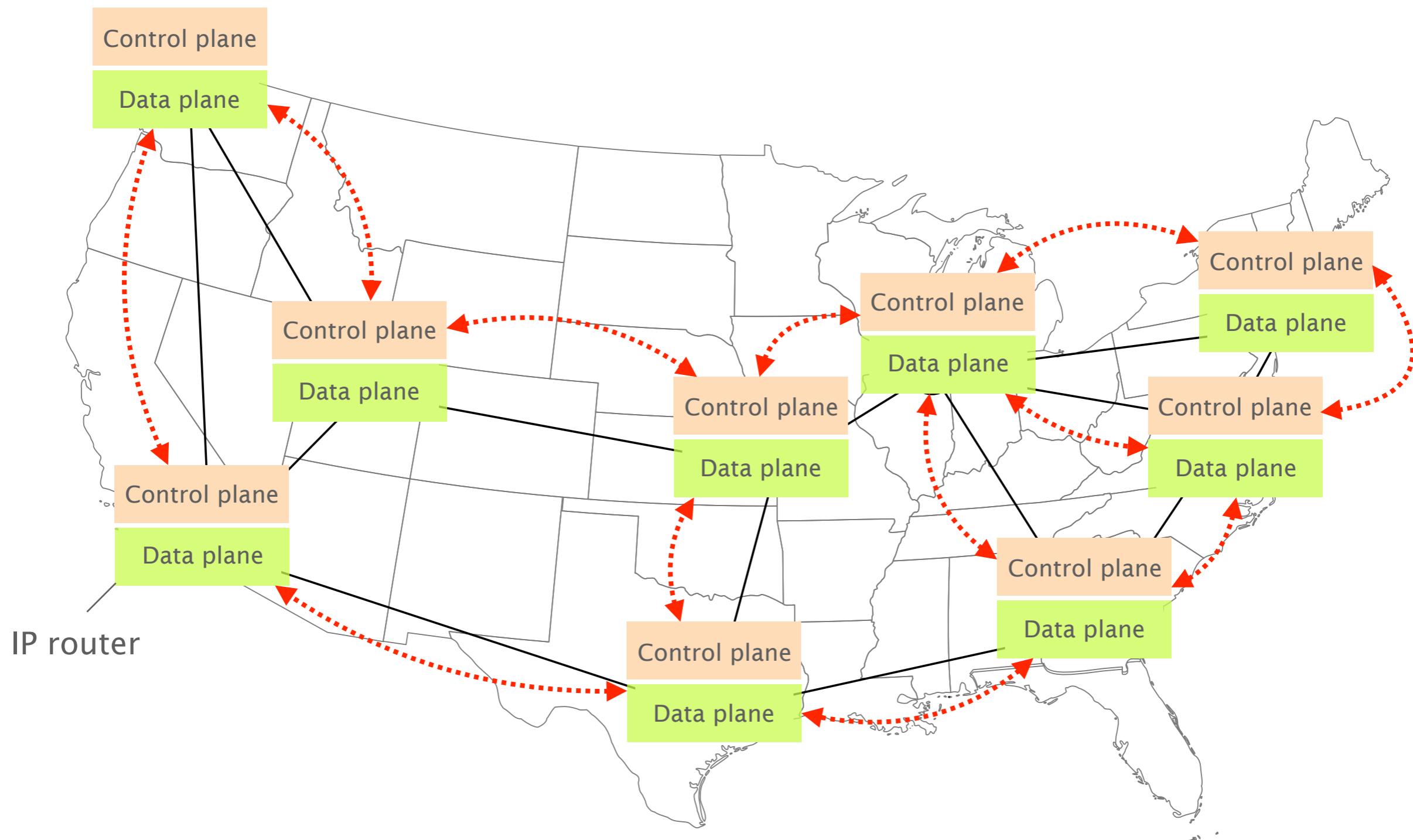
(*) <https://dl.acm.org/citation.cfm?id=1355746>

Why? It's really a story in 3 stages

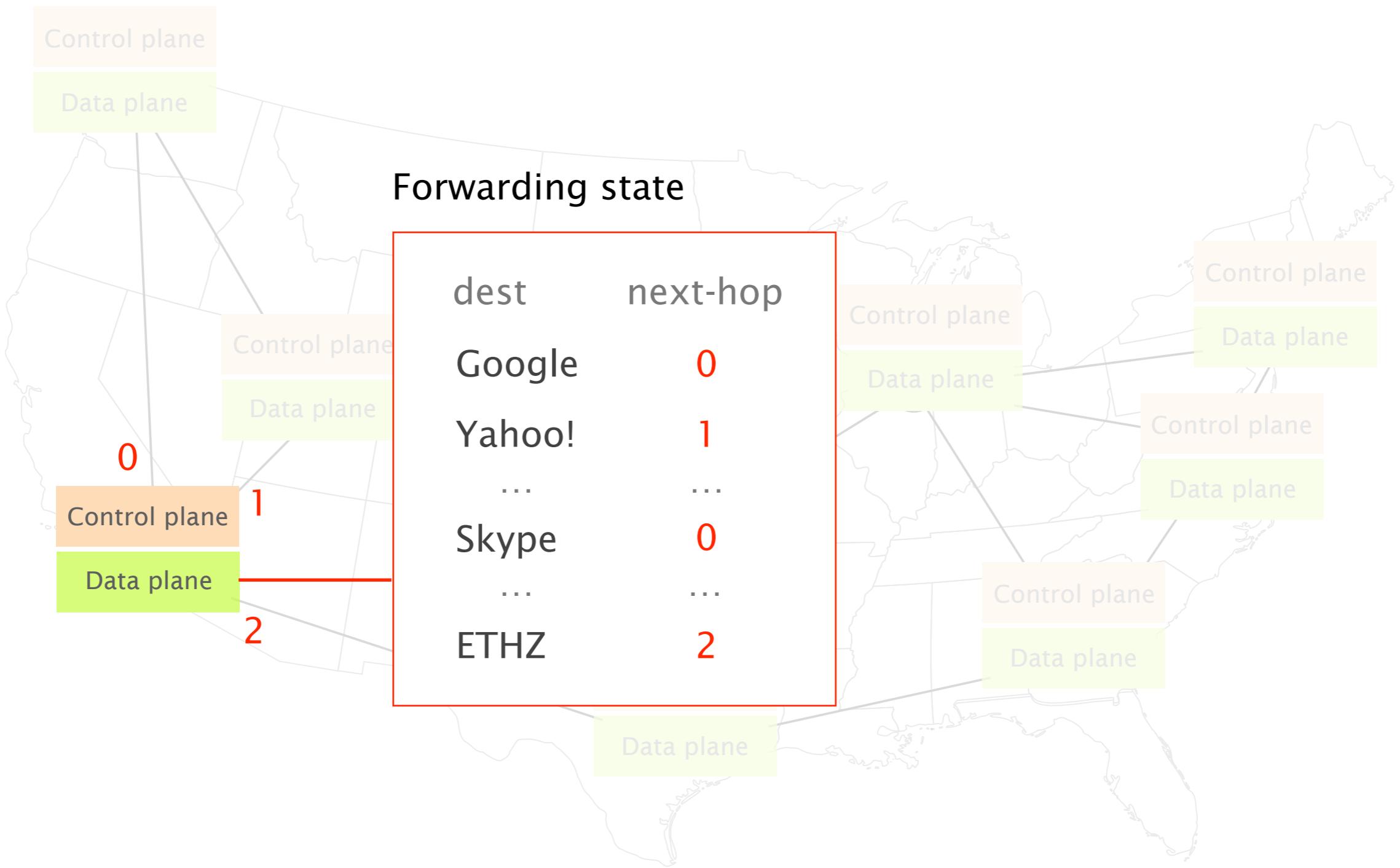
Stage 1

The network management crisis

Networks are large distributed systems
running a set of distributed algorithms

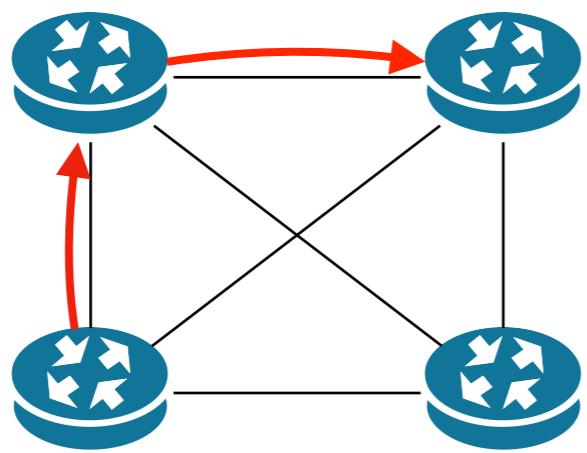


These algorithms produce the forwarding state which drives IP traffic to its destination

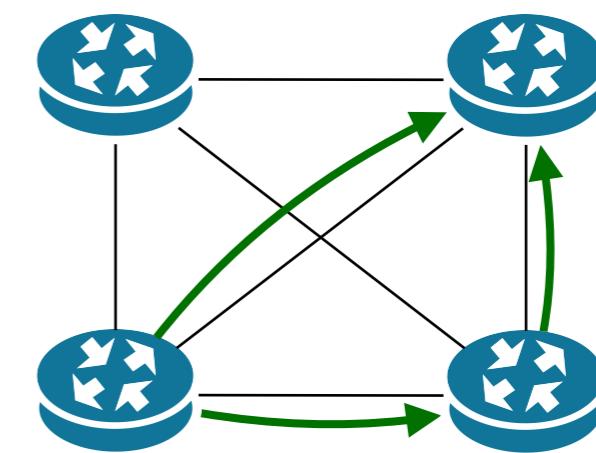


Operators adapt their network forwarding behavior
by configuring each network device individually

Given



and



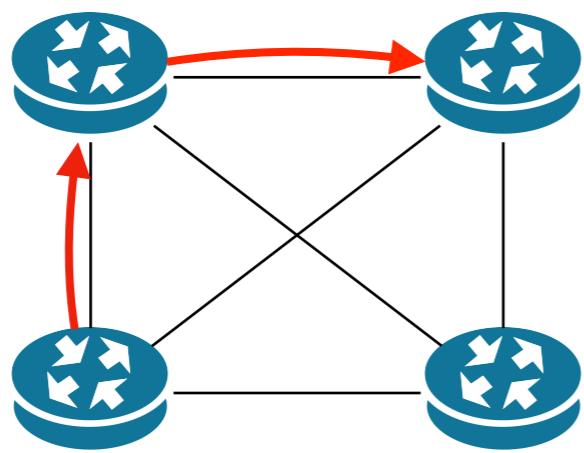
an **existing** network behavior

induced by a low-level configuration C

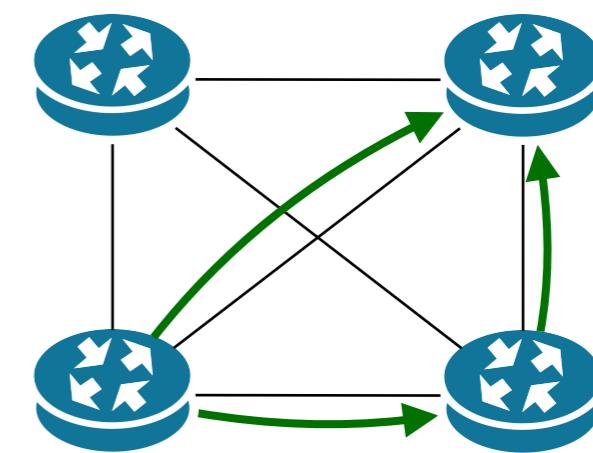
a **desired** network behavior

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

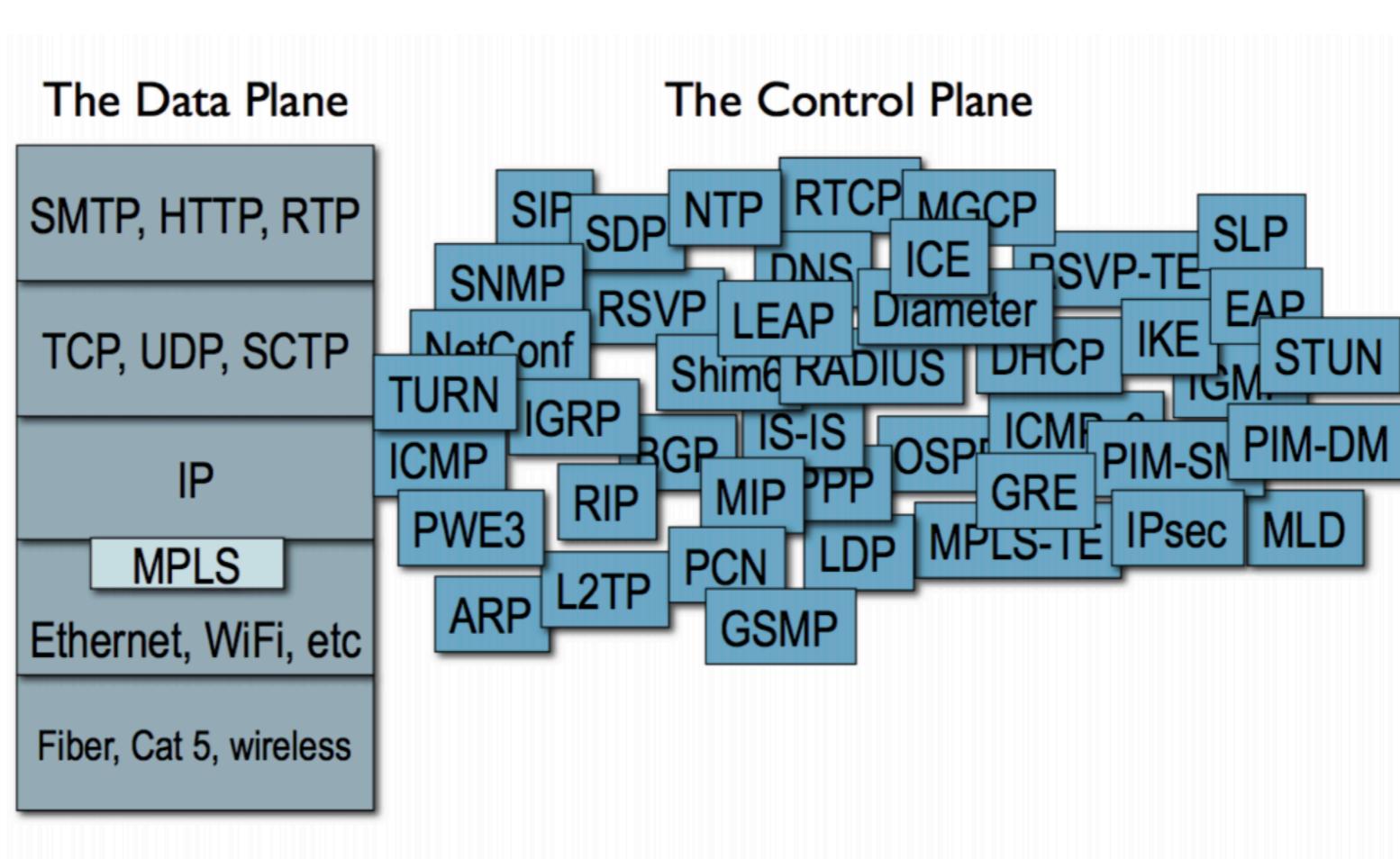
Cisco IOS

```
!
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 —— Anything else than 700 creates blackholes
!
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 {
```

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

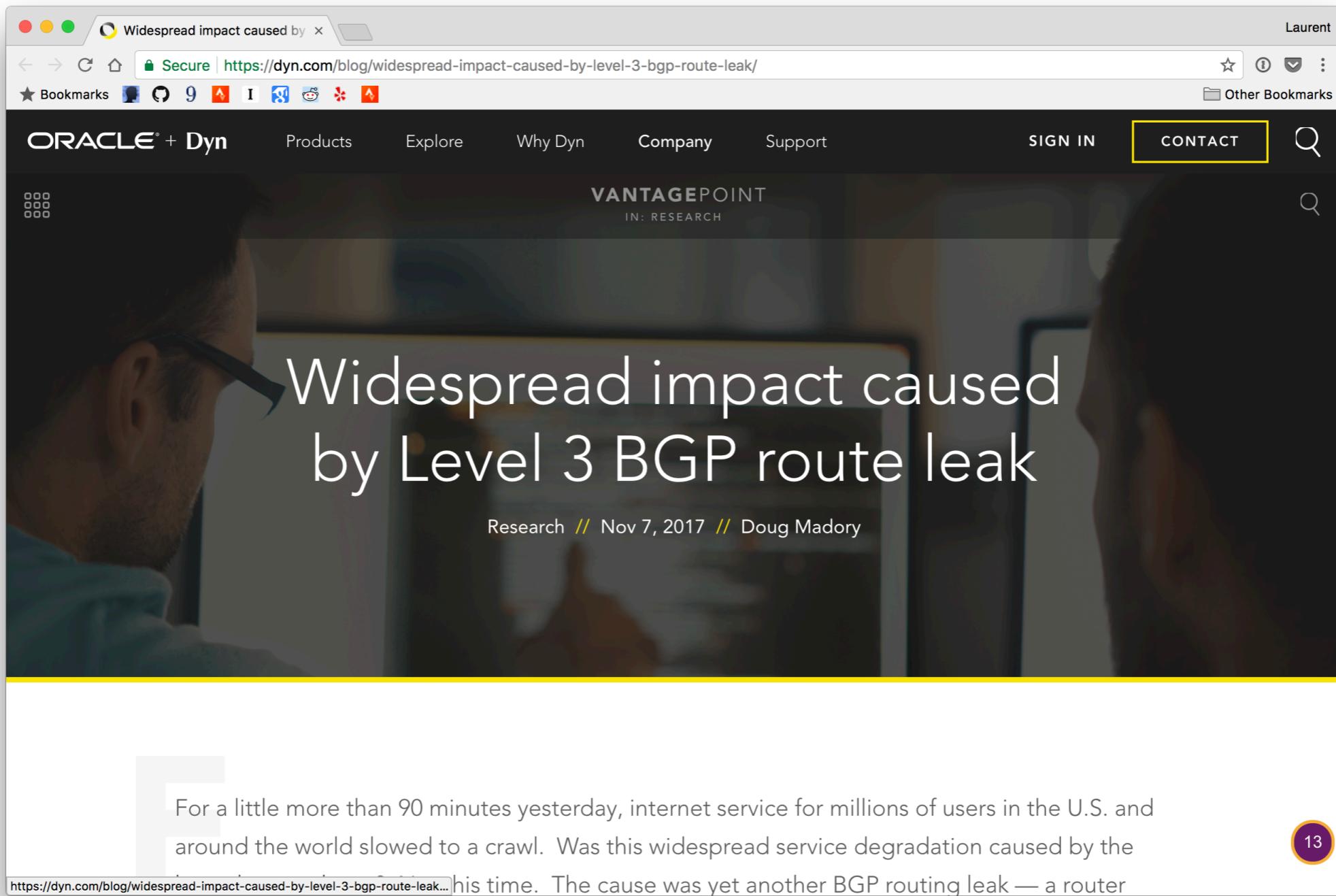


Source

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

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

November 2017



<https://dyn.com/blog/widespread-impact-caused-by-level-3-bgp-route-leak/>

For a little more than 90 minutes [...],

Internet service for millions of users in the U.S.
and around the world slowed to a crawl.

The cause was yet another BGP routing leak,
a **router misconfiguration** directing Internet traffic
from its intended path to somewhere else.

August 2017

The screenshot shows a web browser window displaying an article from The Register. The title of the article is "Google routing blunder sent Japan's Internet dark on Friday". The article discusses a routing mistake by Google that caused Japanese internet traffic to disappear. The Register's logo is at the top, and there are social media sharing options. To the right, there is a sidebar titled "Most read" with several news items.

Data Centre > Networks

Google routing blunder sent Japan's Internet dark on Friday

Another big BGP blunder

By Richard Chirgwin 27 Aug 2017 at 22:35

Last Friday, someone in Google fat-thumbed a border gateway protocol (BGP) advertisement and sent Japanese Internet traffic into a black hole.

The trouble began when The Chocolate Factory "leaked" a big route table to Verizon, 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.

Since Google doesn't provide transit services, as BGP Mon explains, that traffic either filled a link beyond its capacity, or hit an access control list, and disappeared.

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

BGP Mon dissects [what went wrong here](#), reporting that more than

Most read

-  Helicopter crashes after manoeuvres to 'avoid... DJI Phantom drone'
-  That terrifying 'unfixable' Microsoft Skype security flaw: THE TRUTH
-  Stephen Elop and the fall of Nokia revisited
-  BBC presenter loses appeal, must pay £420k in IR35 crackdown
-  Microsoft's Windows 10 Workstation adds killer feature: No Candy Crush

https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/

Someone in Google fat-thumbed 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.



Traders work on the floor of the New York Stock Exchange (NYSE) in July 2015.
(Photo by Spencer Platt/Getty Images)

DOWNTIME

UPDATED: “Configuration Issue” Halts Trading on NYSE

The article has been updated with the time trading resumed.

A second update identified the cause of the outage as a “configuration issue.”

A third update added information about a software update that created the configuration issue.

NYSE network operators identified
the culprit of the **3.5 hour** outage,
blaming the incident on a
“network configuration issue”

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

United Airlines Blames Router for Grounded Flights

**Alexandra Talty,** CONTRIBUTOR*I cover personal finance and travel.*[FOLLOW ON FORBES \(110\)](#)

Opinions expressed by Forbes Contributors are their own.

[FULL BIO ▾](#)

After a computer problem caused nearly two hours of grounded flights for United Airlines this morning and ongoing delays throughout the day, the airline announced the culprit: a [faulty router](#).

Spokeswoman Jennifer Dohm said that the router problem caused “degraded network connectivity,” which affected various applications.

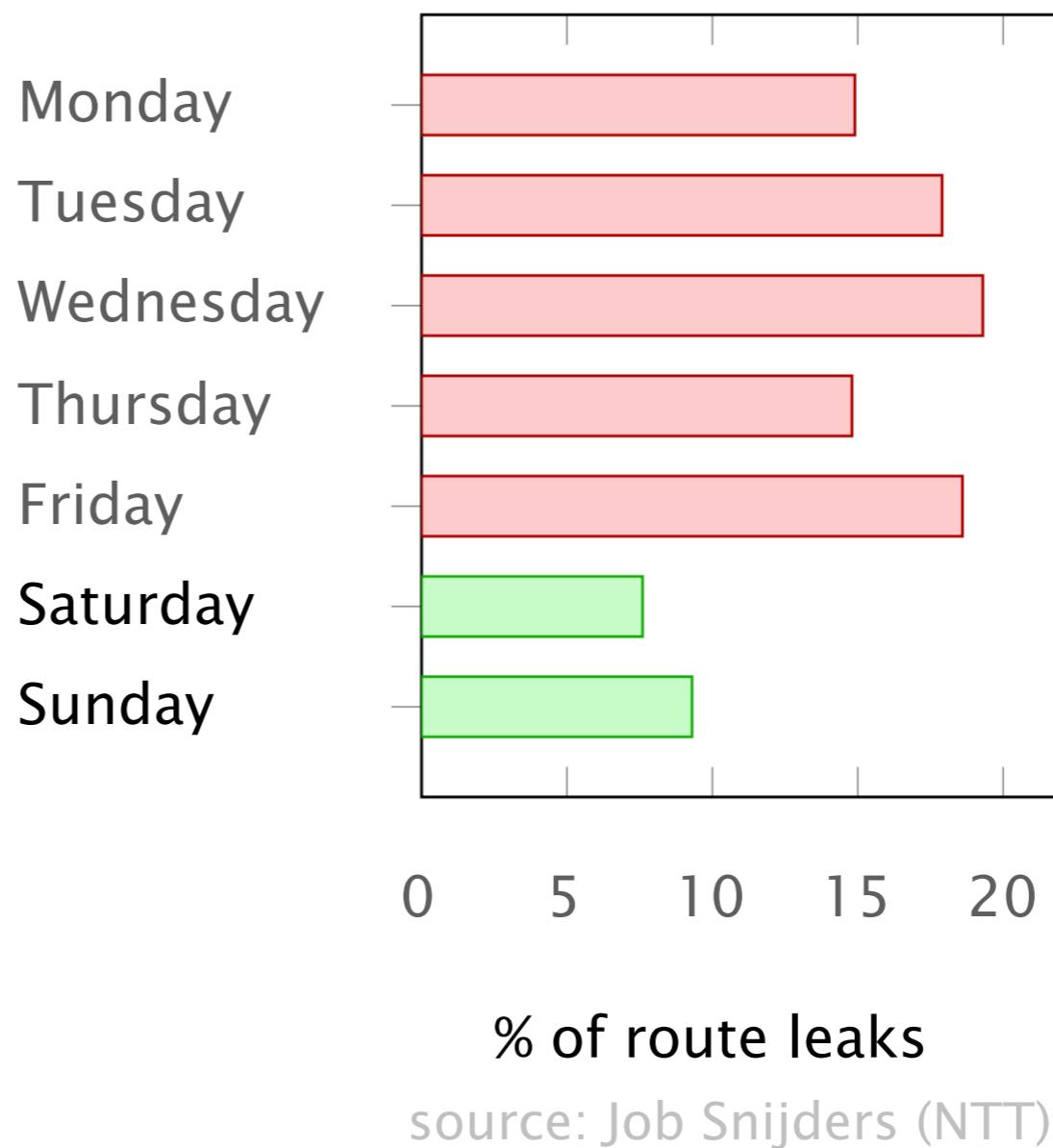
A computer glitch in the airline’s reservations system caused the Federal Aviation Administration to impose a groundstop at 8:26 a.m. E.T. Planes that were in the air continued to operate, but all planes on the ground were held. There were reports of agents writing tickets by hand. The ground stop was lifted around 9:47 a.m. ET.

<http://bit.ly/2sBJ2jf>

“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...



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

Internet advertisements rates suggest that
The Internet was **more stable than normal on Sept 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

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

“Cost per network outage
can be as high as 750 000\$”

Smart Management for Robust Carrier Network Health
and Reduced TCO!, NANOG54, 2012

Solving this problem is hard because
network devices are completely locked down



closed software

closed hardware

Cisco™ device

Stage 2

Software-Defined Networking

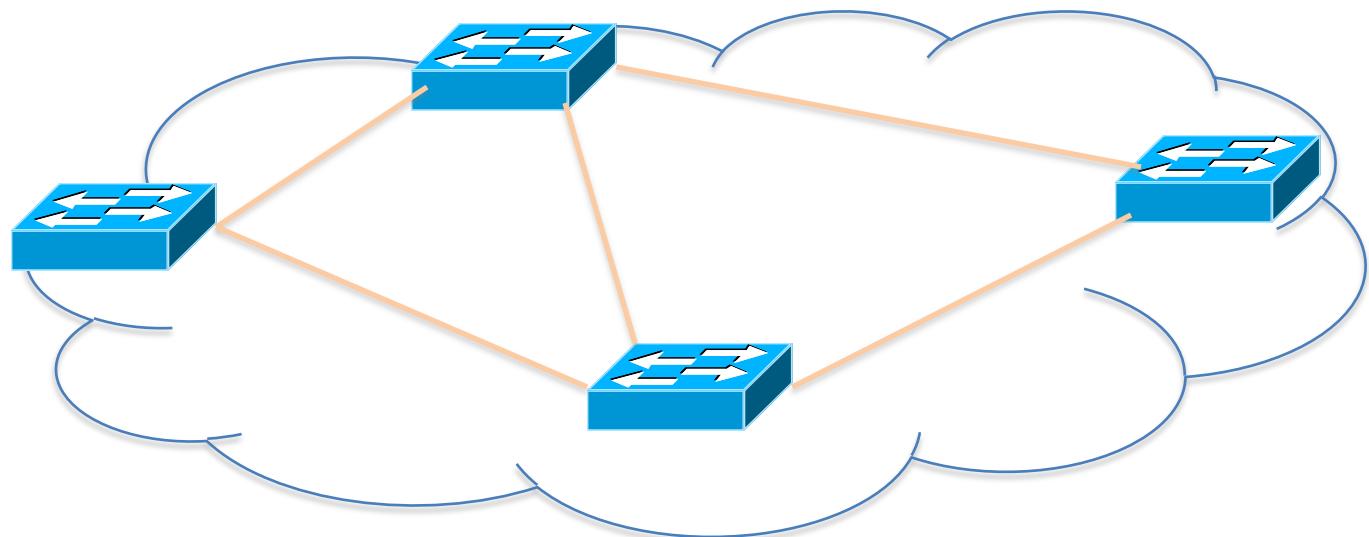
What is SDN and how does it help?

- 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

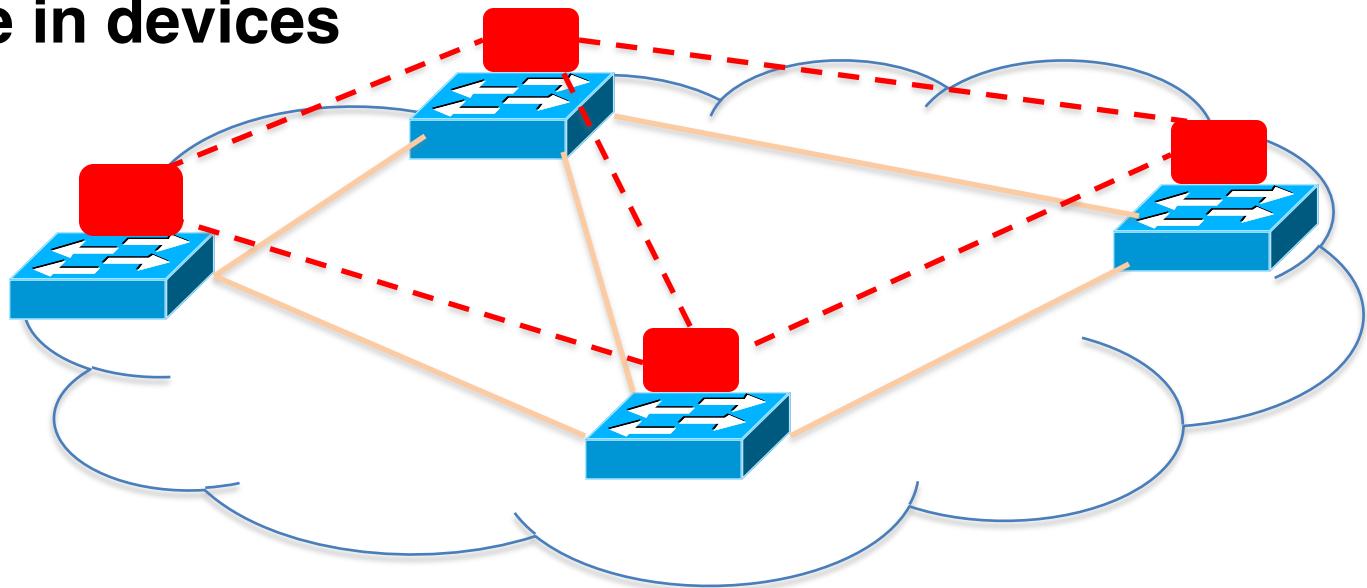
Data plane:
Packet
processing &
delivery



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

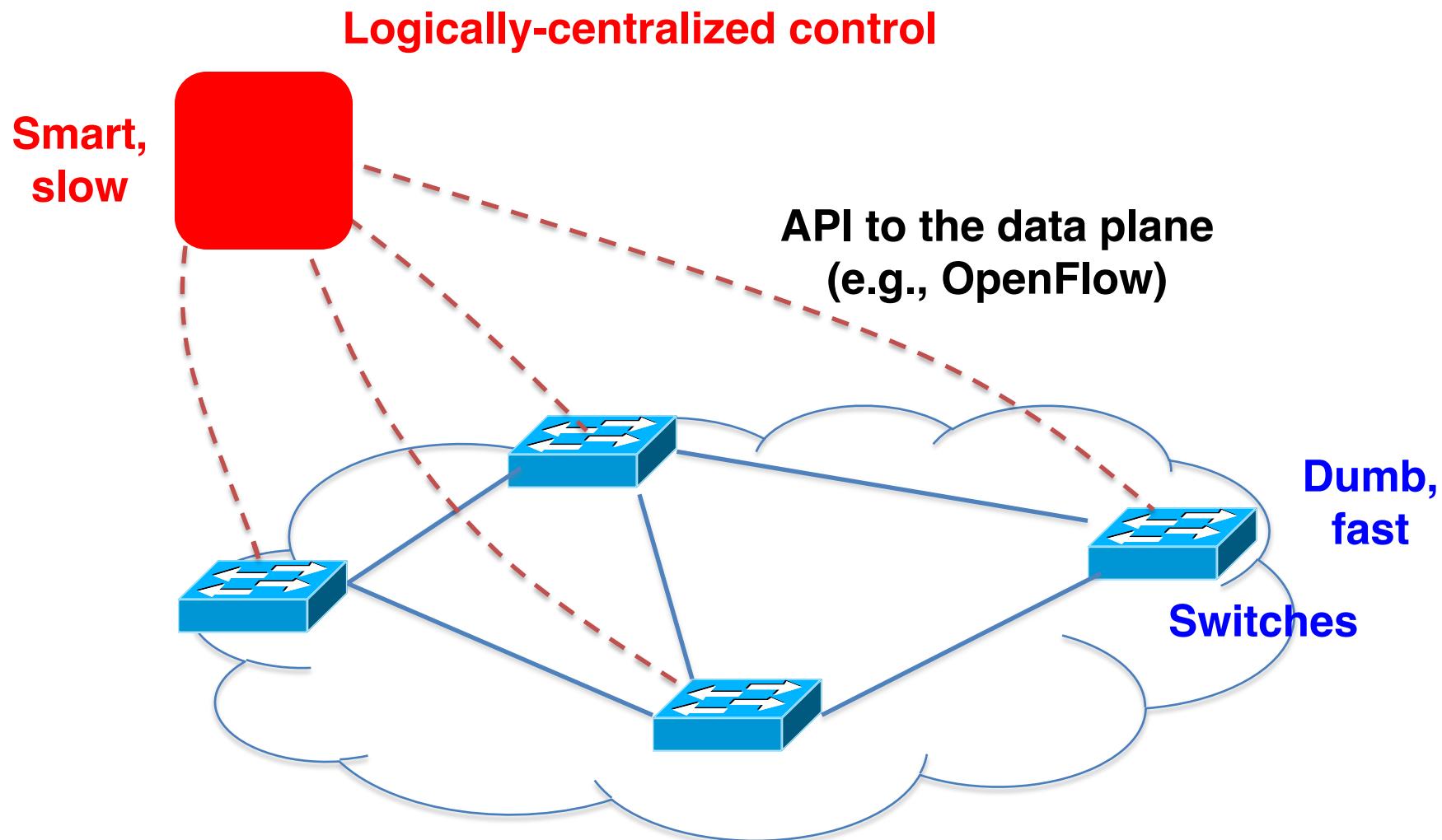
Traditional Computer Networks

Control plane:
Distributed algorithms,
establish state in devices



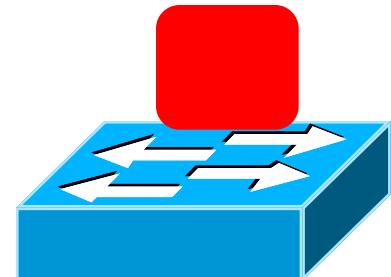
**Track topology changes, compute
routes, install forwarding rules**

Software Defined Networking (SDN)



SDN advantages

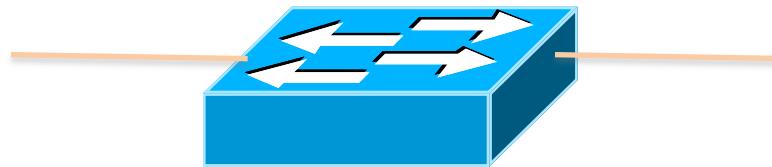
- Simpler management
 - No need to “invert” control-plane operations
- Faster pace of innovation
 - Less dependence on vendors and standards
- Easier interoperability
 - Compatibility only in “wire” protocols
- Simpler, cheaper equipment
 - Minimal software



OpenFlow Networks

OpenFlow is an API to a switch flow table

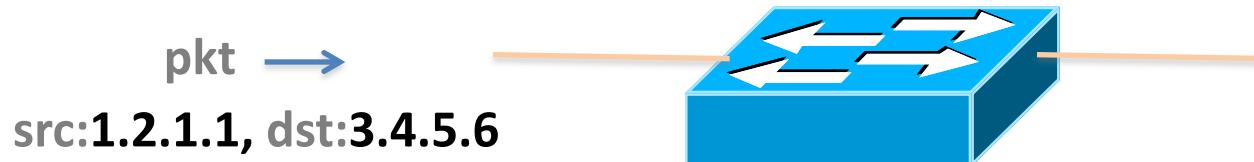
- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flowspace
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



10. $\text{src}=1.2.*.*$, $\text{dest}=3.4.5.* \rightarrow \text{drop}$
05. $\text{src} = *.*.*.*$, $\text{dest}=3.4.*.* \rightarrow \text{forward}(2)$
01. $\text{src}=10.1.2.3$, $\text{dest} = *.*.*.* \rightarrow \text{send to controller}$

OpenFlow is an API to a switch flow table

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



10. $\text{src}=1.2.*.*$, $\text{dest}=3.4.5.* \rightarrow \text{drop}$
05. $\text{src} = *.*.*.*$, $\text{dest}=3.4.*.* \rightarrow \text{forward}(2)$
01. $\text{src}=10.1.2.3$, $\text{dest} = *.*.*.* \rightarrow \text{send to controller}$

OpenFlow is an API to a switch flow table

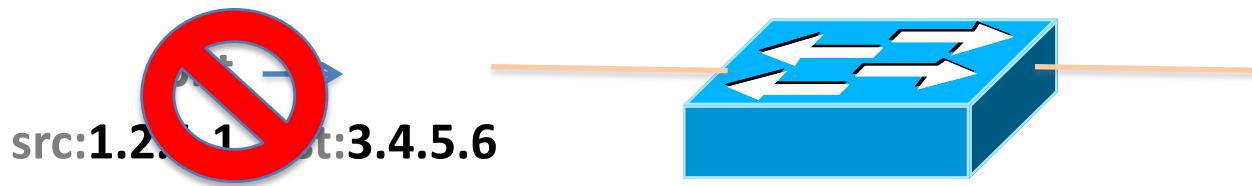
- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flowspace
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



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

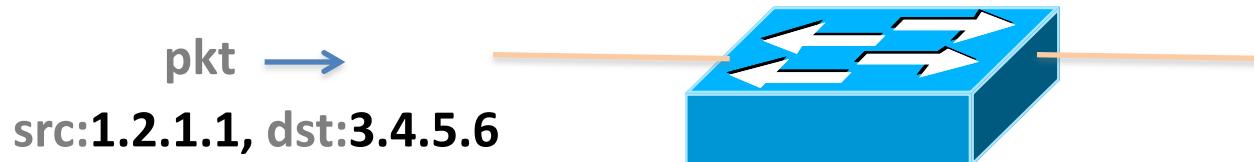
- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flowspace
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



10. $\text{src}=1.2.*.*$, $\text{dest}=3.4.5.* \rightarrow \text{drop}$
05. $\text{src} = *.*.*.*$, $\text{dest}=3.4.*.* \rightarrow \text{forward}(2)$
01. $\text{src}=10.1.2.3$, $\text{dest} = *.*.*.* \rightarrow \text{send to controller}$

OpenFlow is an API to a switch flow table

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



10. $\text{src}=1.2.*.*$, $\text{dest}=3.4.5.* \rightarrow \text{drop}$
05. $\text{src} = *.*.*.*$, $\text{dest}=3.4.*.* \rightarrow \text{forward}(2)$
01. $\text{src}=10.1.2.3$, $\text{dest} = *.*.*.* \rightarrow \text{send to controller}$

OpenFlow is an API to a switch flow table

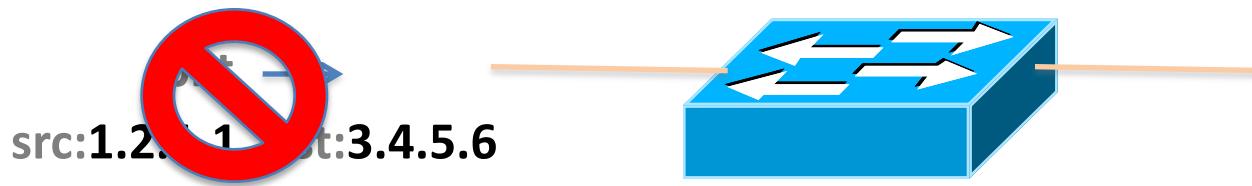
- Simple packet-handling rules
 - Pattern: match packet header bits, i.e. flowspace
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets



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. flowspace
 - Actions: drop, forward, modify, send to controller
 - Priority: disambiguate overlapping patterns
 - Counters: #bytes and #packets

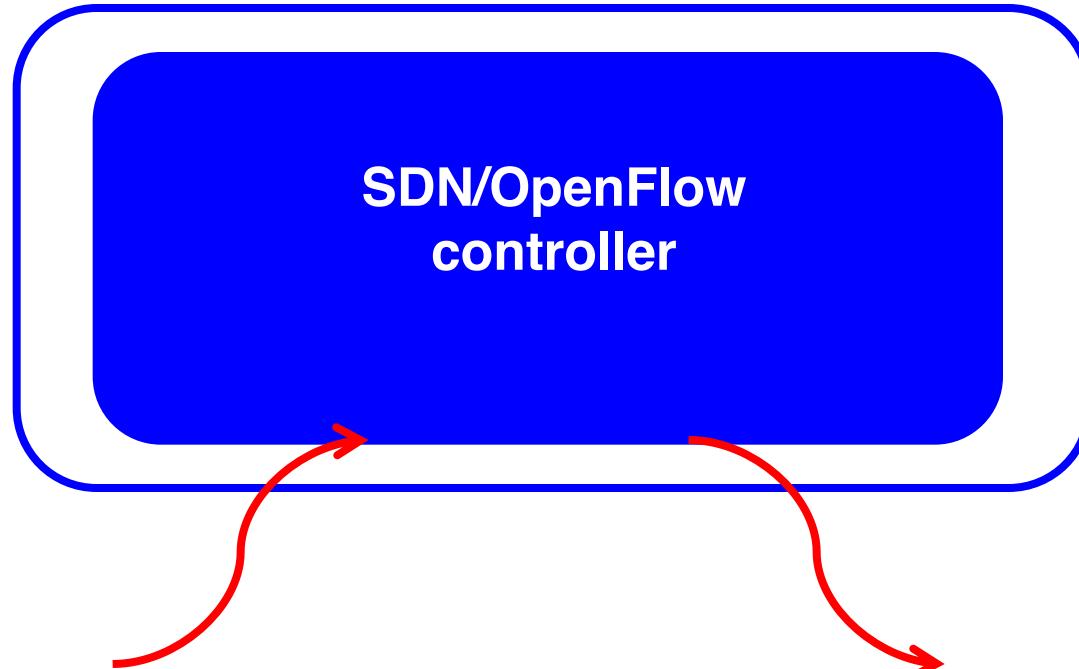


10. $\text{src}=1.2.*.*$, $\text{dest}=3.4.5.* \rightarrow \text{drop}$
05. $\text{src} = *.*.*.*$, $\text{dest}=3.4.*.* \rightarrow \text{forward}(2)$
01. $\text{src}=10.1.2.3$, $\text{dest} = *.*.*.* \rightarrow \text{send to controller}$

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
```

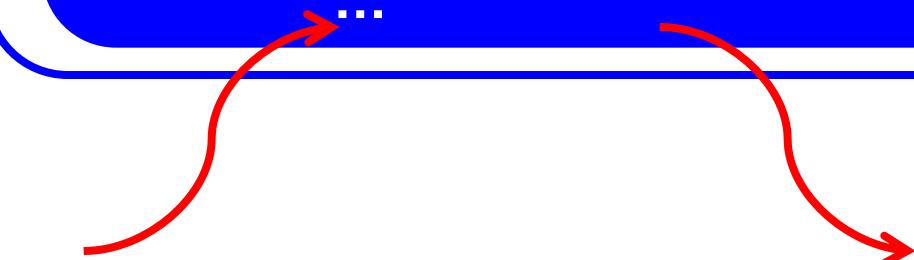
...

Receives events from switches

Topology changes,
Traffic statistics,
Arriving packets

Send commands to switches

(Un)install rules,
Query statistics,
Send packets

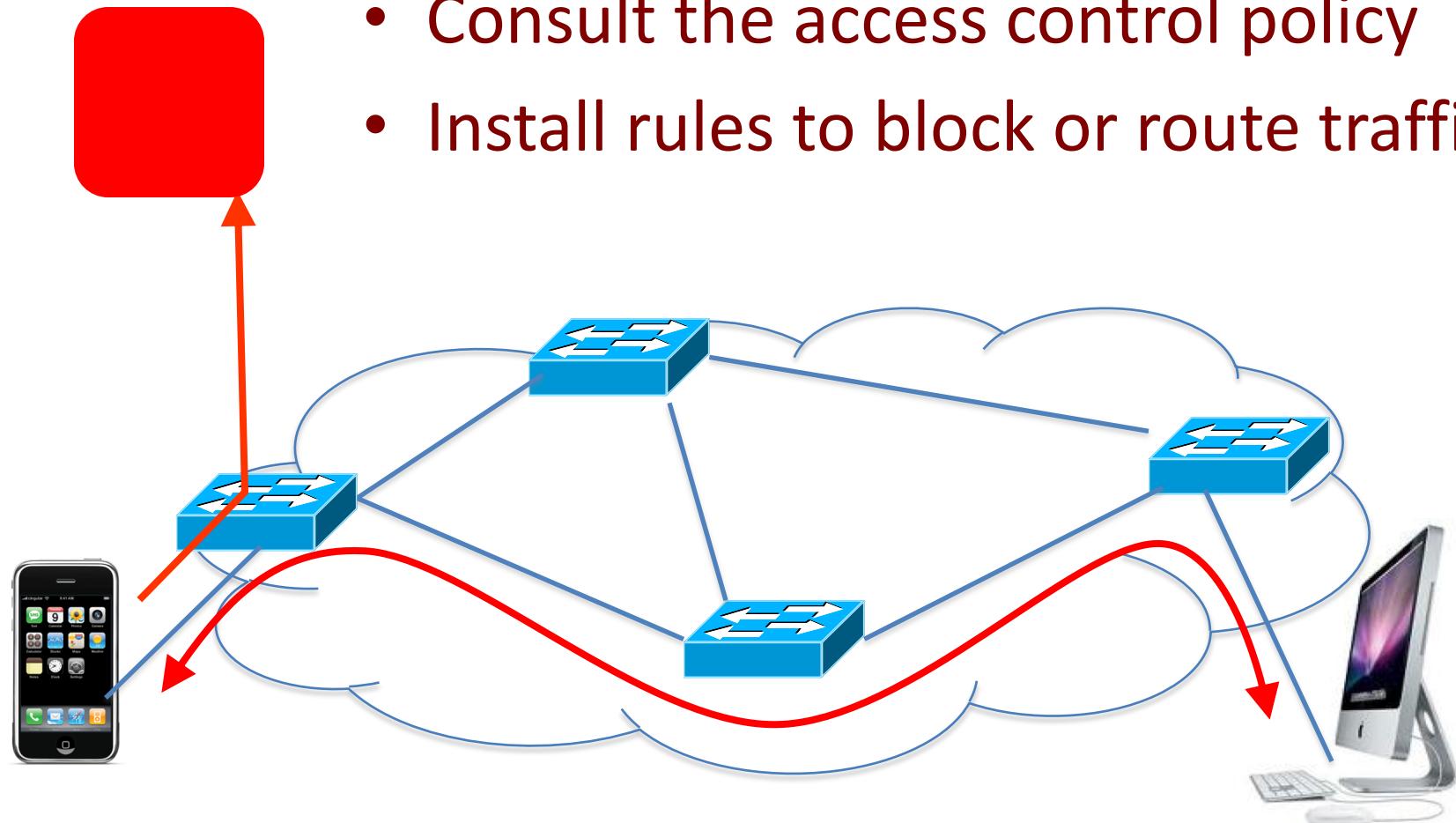


Example OpenFlow Applications

- **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

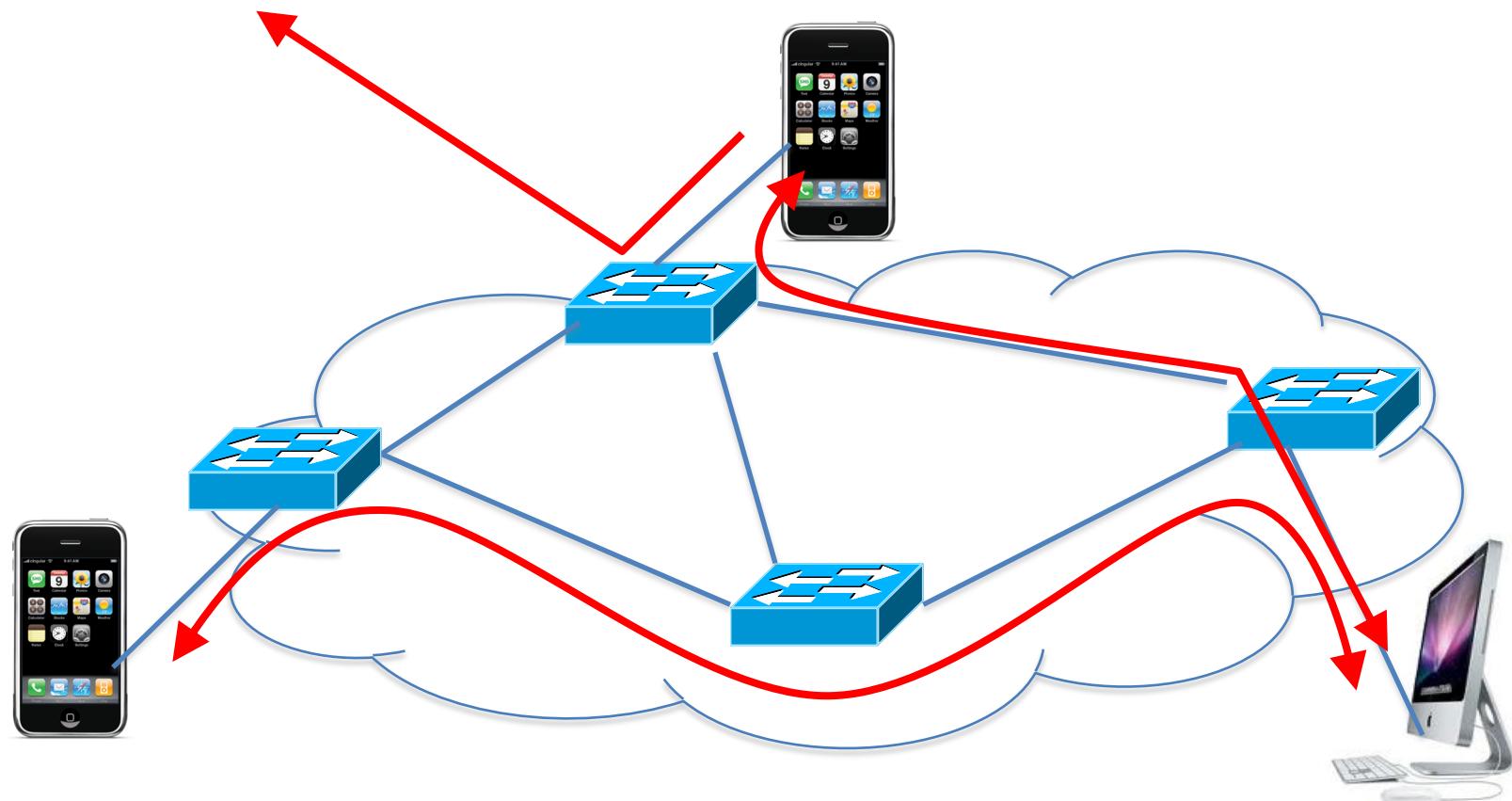
E.g.: Dynamic Access Control

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



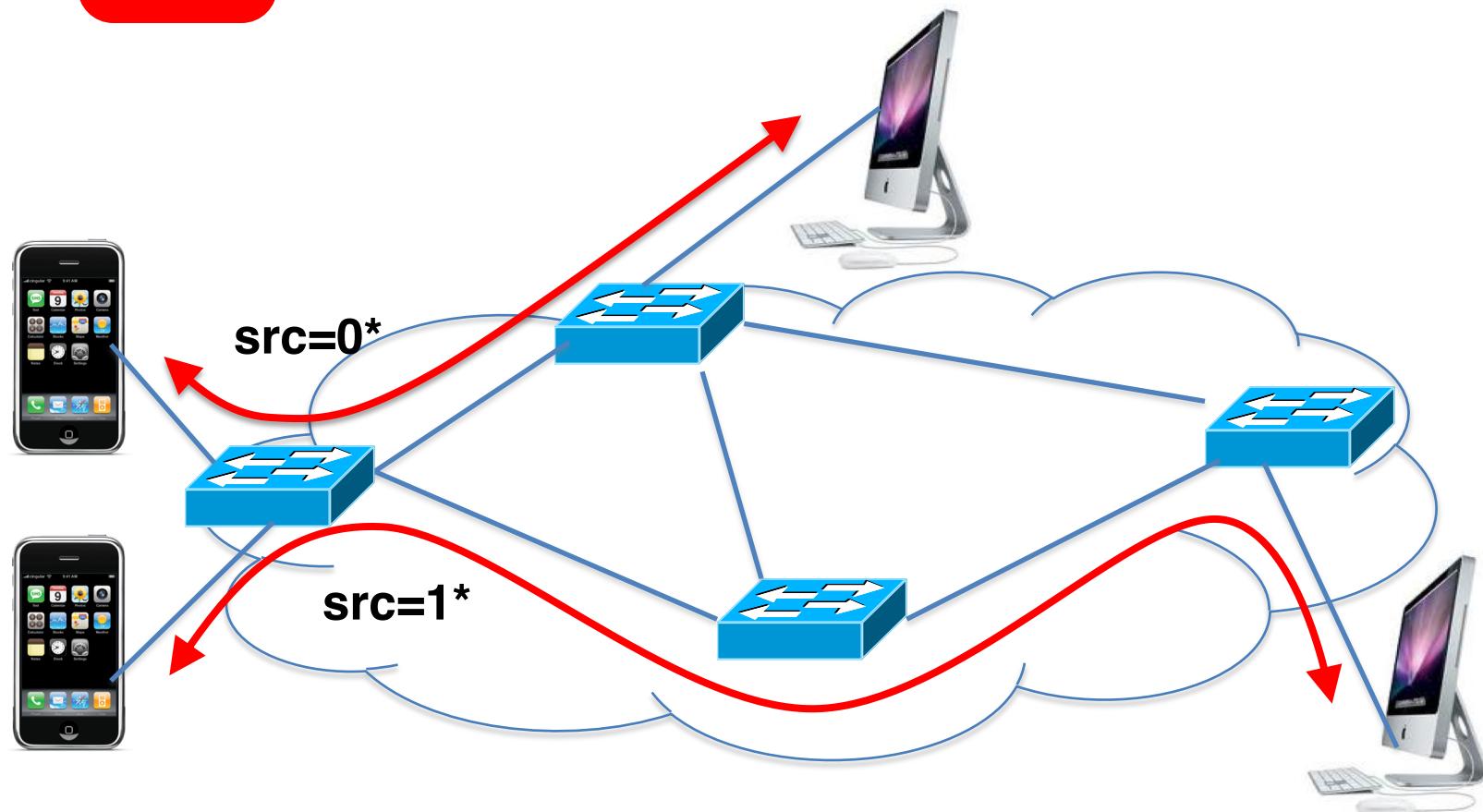
E.g.: Seamless Mobility/Migration

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



E.g.: Server Load Balancing

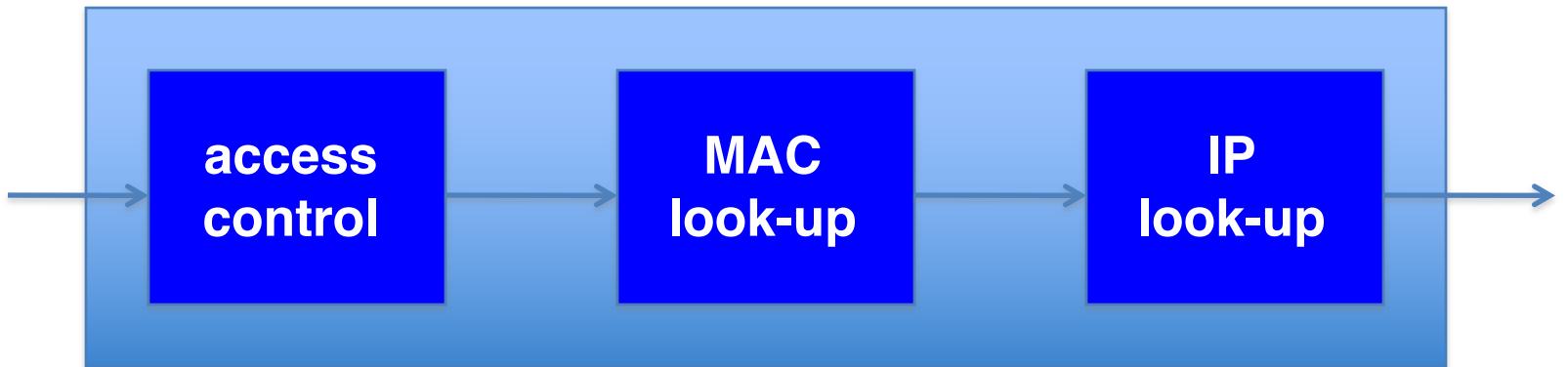
- Pre-install load-balancing policy
- Split traffic based on source IP



Challenges

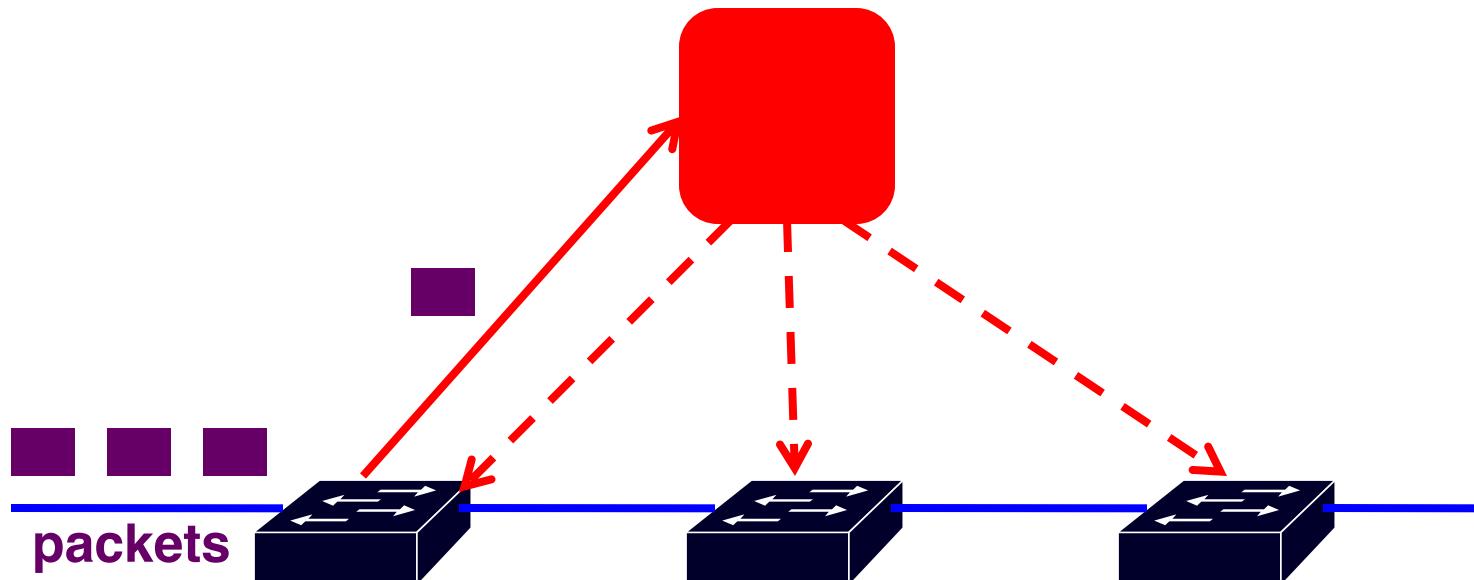
Heterogeneous Switches

- 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

Controller Application

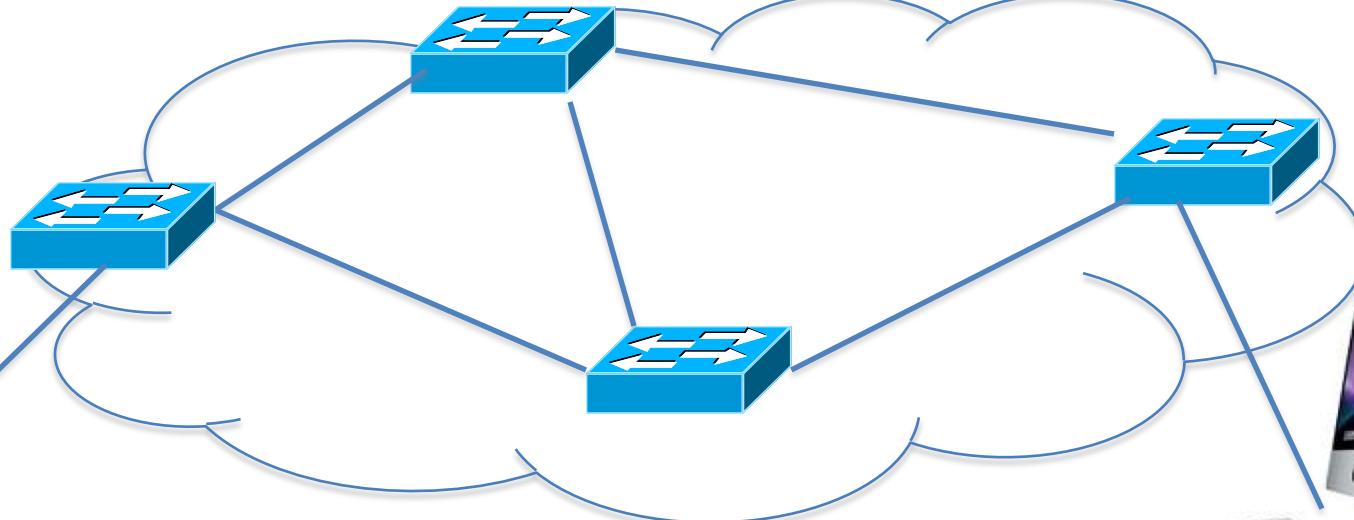
Network OS

For scalability
and reliability

Controller Application

Network OS

Partition and replicate state

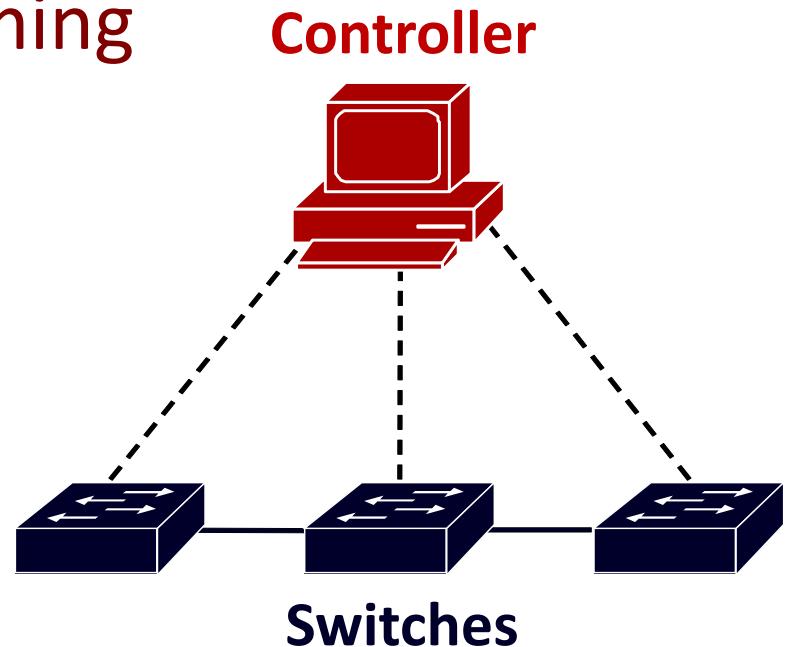


Testing and Debugging

- 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!

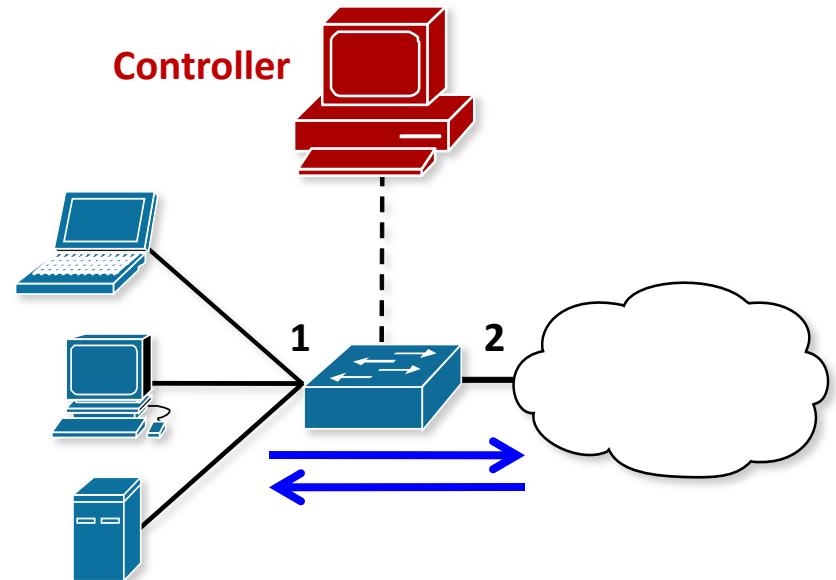


Example: Simple Repeater

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)
```



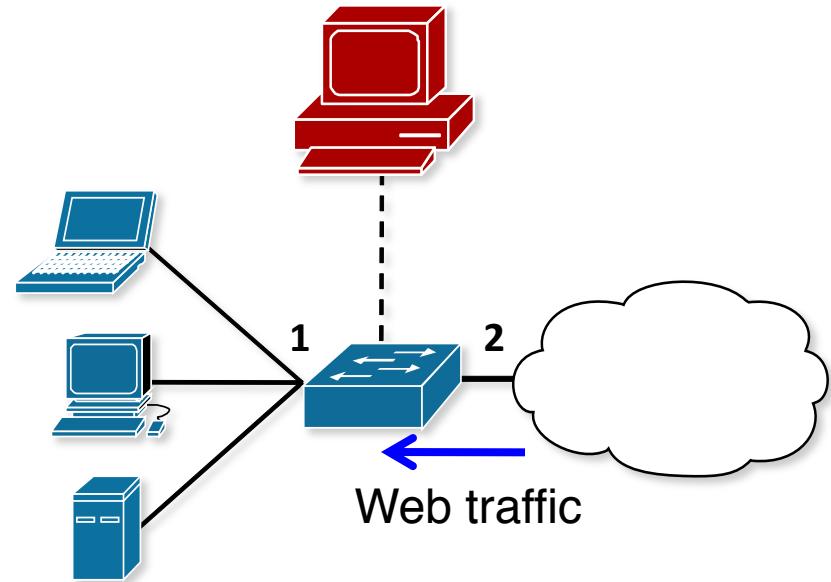
When a switch joins the network, install two forwarding rules.

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

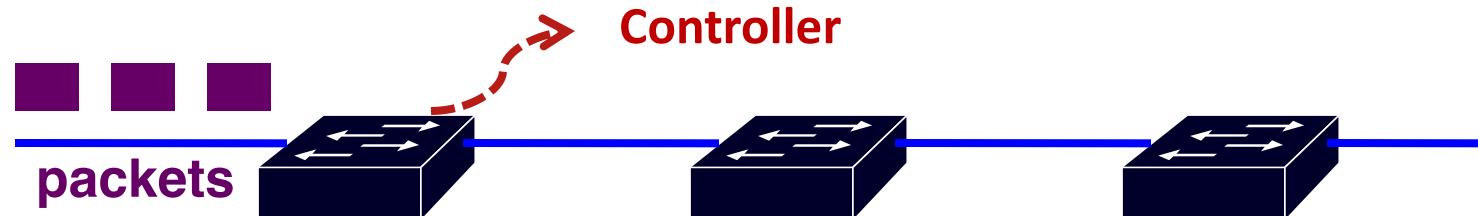
```
def switch_join(switch):
    pat1 = {inport:1}
    pat2 = {inport:2}
    pat2web = {in_port:2, tp_src:80}
    install(switch, pat1, DEFAULT, None, [forward(2)])
    install(switch, pat2web, HIGH, None, [forward(1)])
    install(switch, pat2, DEFAULT, None, [forward(1)])
    query_stats(switch, pat2web)

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

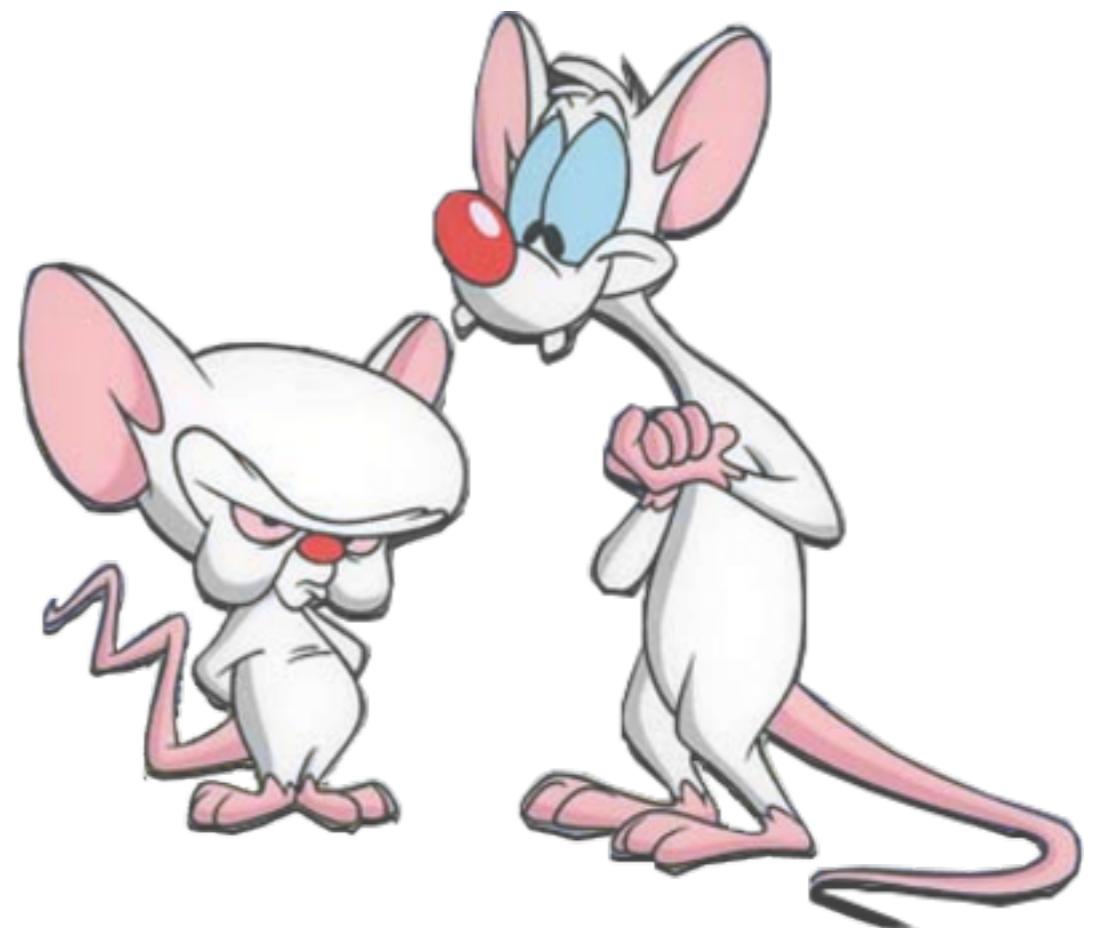
Deep Network Programmability

Pinky

Gee, Brain, did OpenFlow take over the world?

The Brain

Well... **no.**



OpenFlow is not all roses

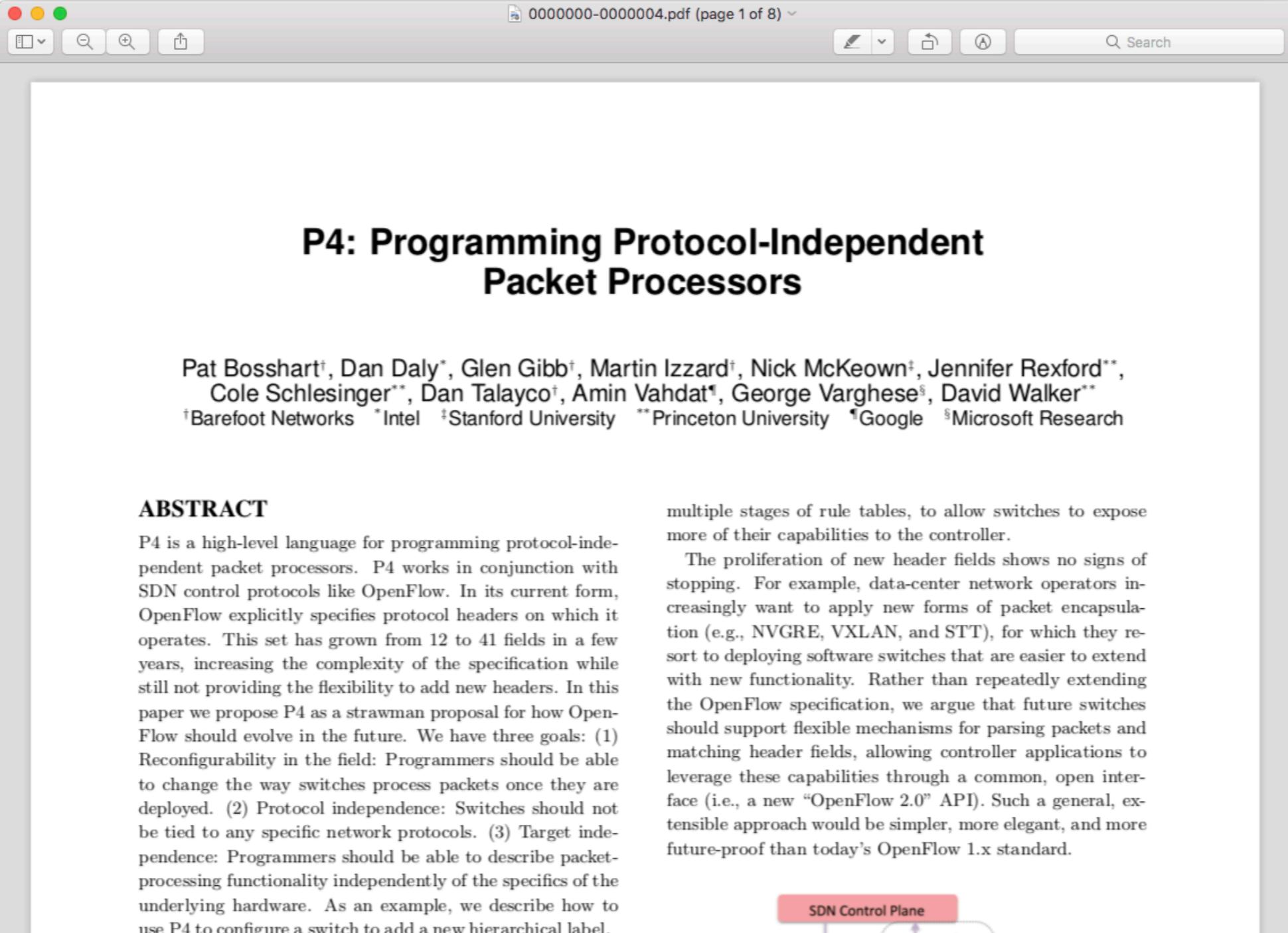
The protocol is too complex (12 fields in OF 1.0 to 41 in 1.5)

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



The screenshot shows a PDF document titled "0000000-0000004.pdf (page 1 of 8)". The title of the paper is "P4: Programming Protocol-Independent Packet Processors". The authors listed are Pat Bosshart[†], Dan Daly^{*}, Glen Gibb[†], Martin Izzard[†], Nick McKeown[‡], Jennifer Rexford^{**}, Cole Schlesinger^{**}, Dan Talayco[†], Amin Vahdat[†], George Varghese[§], David Walker^{**}. The affiliations are: [†]Barefoot Networks ^{*}Intel [‡]Stanford University ^{**}Princeton University [§]Google [§]Microsoft Research.

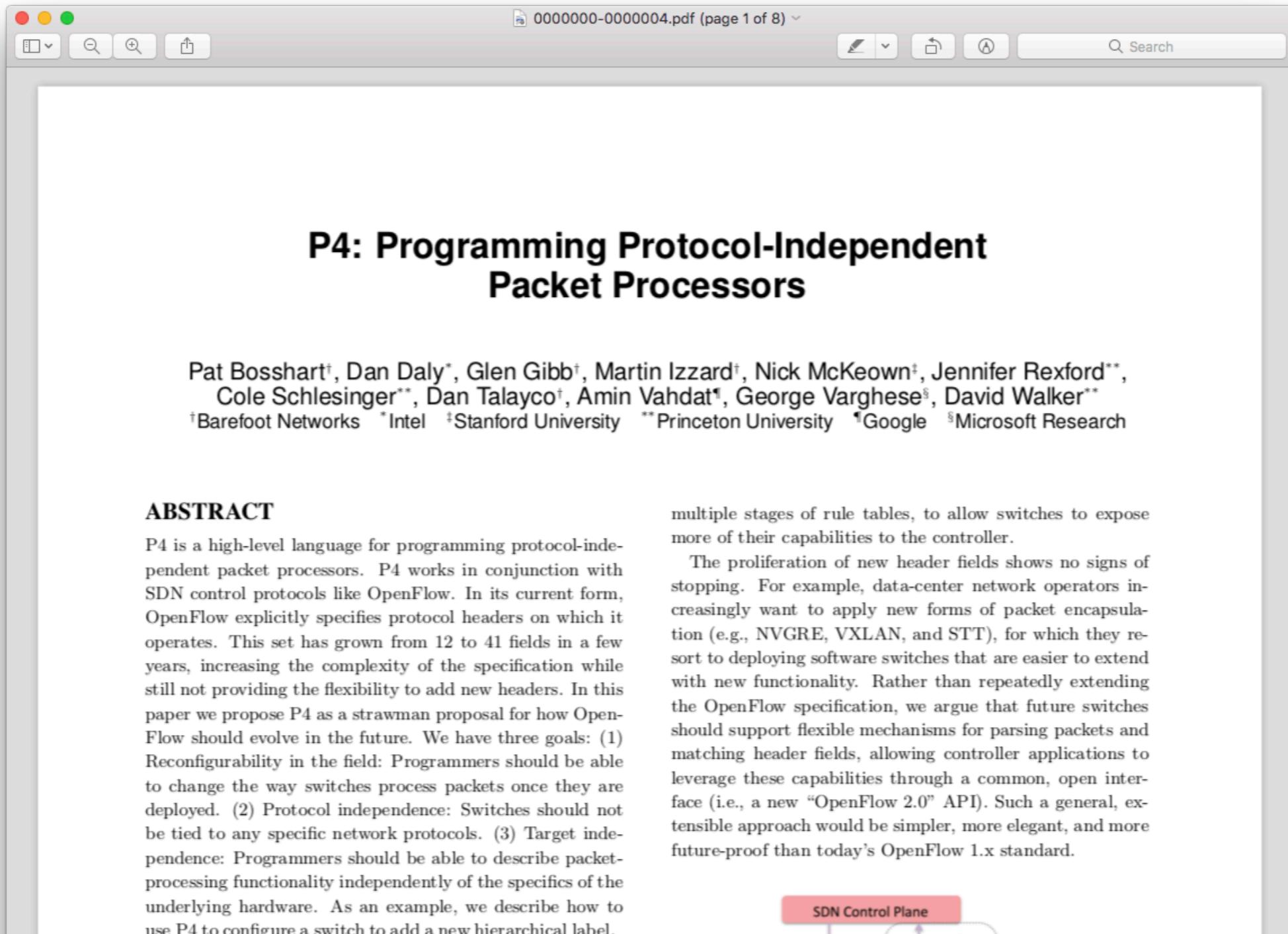
ABSTRACT

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.

Enters... Protocol Independent Switch Architecture and P4



The screenshot shows a PDF document titled "0000000-0000004.pdf (page 1 of 8)". The title of the paper is "P4: Programming Protocol-Independent Packet Processors". The authors listed are Pat Bosshart[†], Dan Daly^{*}, Glen Gibb[†], Martin Izzard[†], Nick McKeown[‡], Jennifer Rexford^{**}, Cole Schlesinger^{**}, Dan Talayco[†], Amin Vahdat[†], George Varghese[§], David Walker^{**}. The affiliations are: [†]Barefoot Networks ^{*}Intel [‡]Stanford University ^{**}Princeton University [§]Google [§]Microsoft Research.

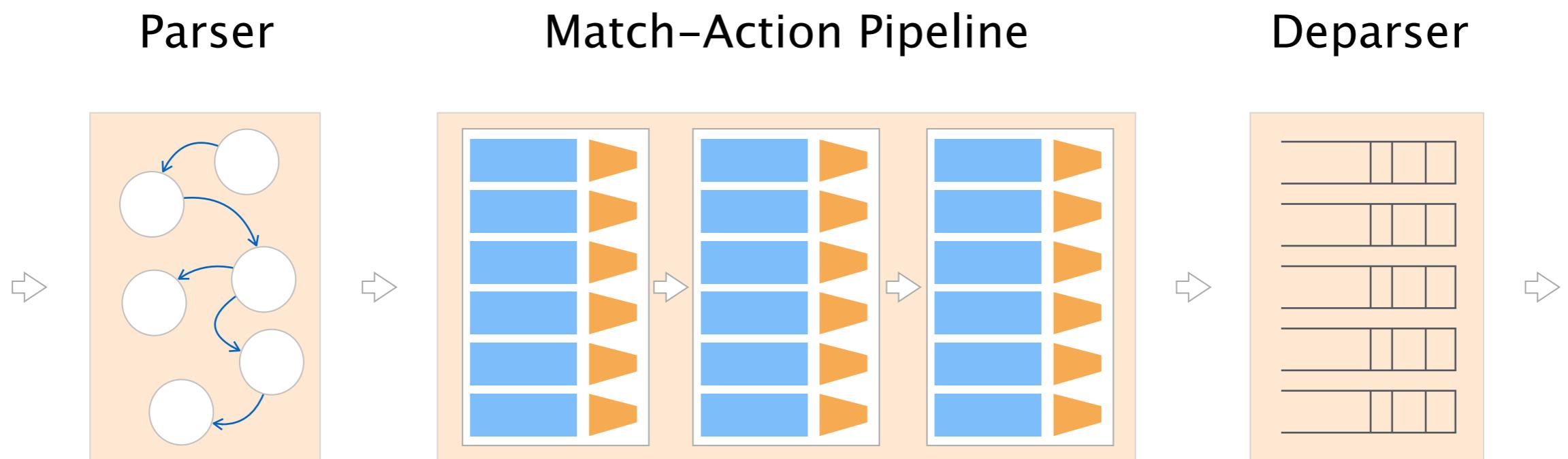
ABSTRACT

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.

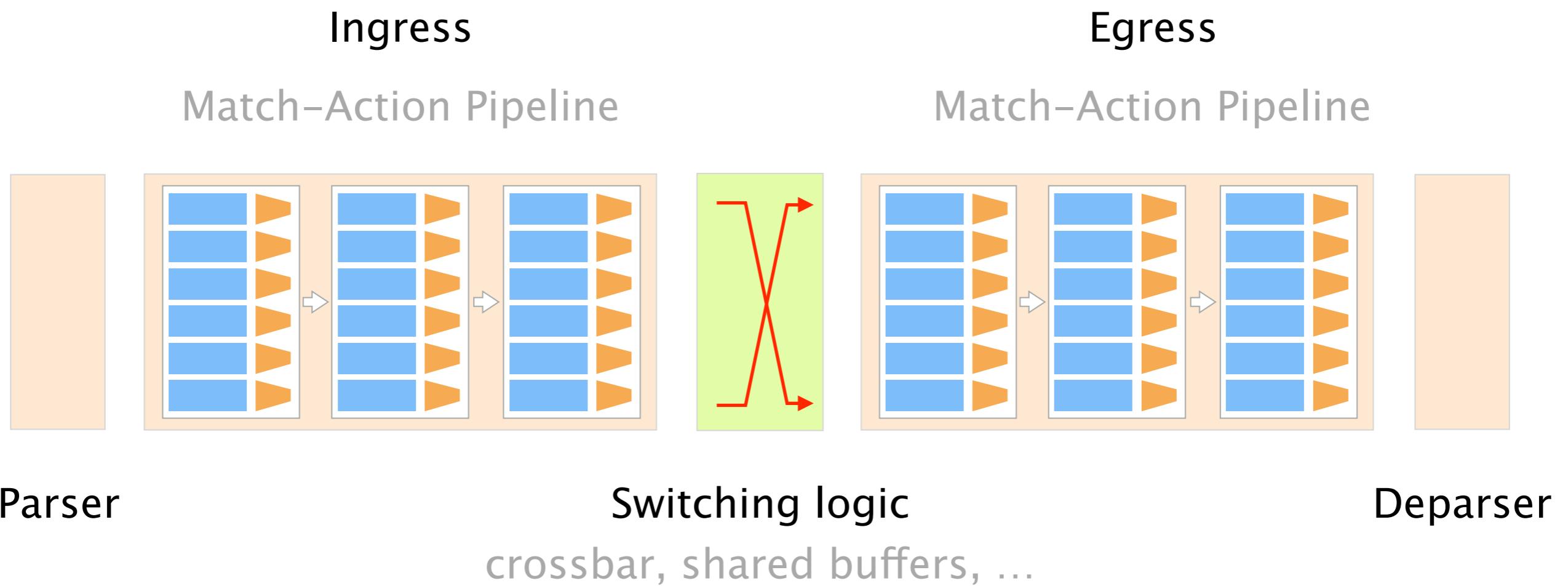
multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.

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.

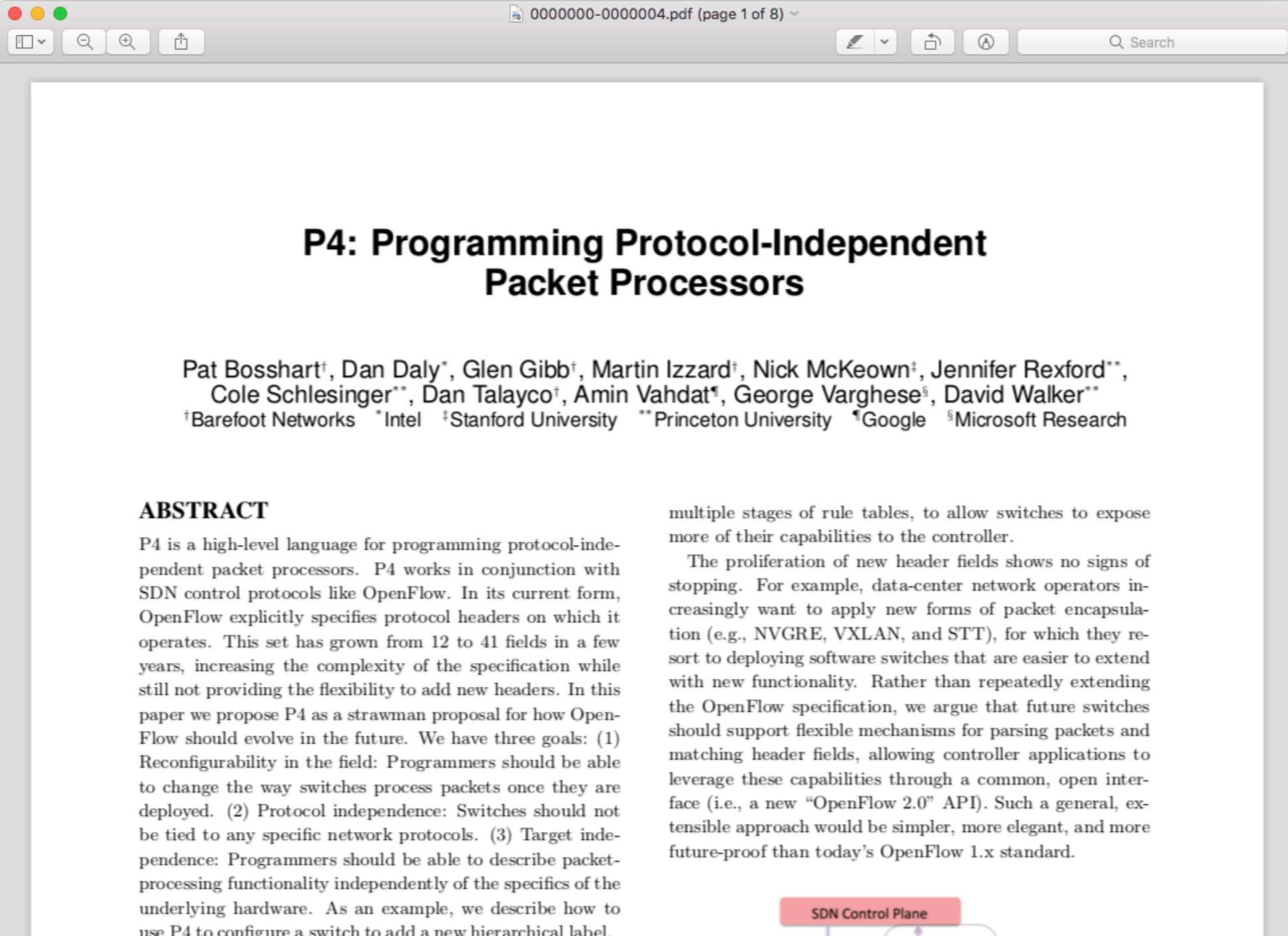
Protocol Independent Switch Architecture (PISA) for high-speed programmable packet forwarding



A slightly more accurate architecture



Enters... Protocol Independent Switch Architecture and P4



The screenshot shows a PDF document titled "0000000-0000004.pdf (page 1 of 8)". The title of the paper is "P4: Programming Protocol-Independent Packet Processors". The authors listed are Pat Bosshart[†], Dan Daly^{*}, Glen Gibb[†], Martin Izzard[†], Nick McKeown[‡], Jennifer Rexford^{**}, Cole Schlesinger^{**}, Dan Talayco[†], Amin Vahdat[†], George Varghese[§], David Walker^{**}. The affiliations are: [†]Barefoot Networks ^{*}Intel [‡]Stanford University ^{**}Princeton University [§]Google [§]Microsoft Research.

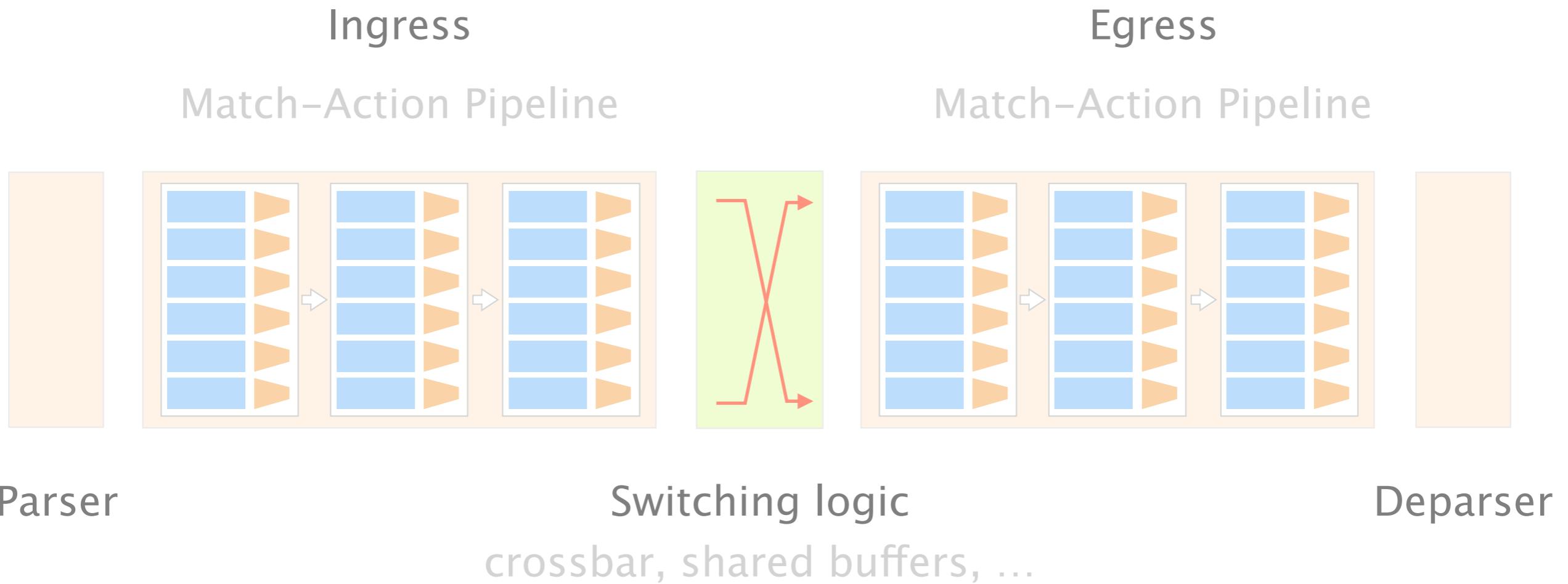
ABSTRACT

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.

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

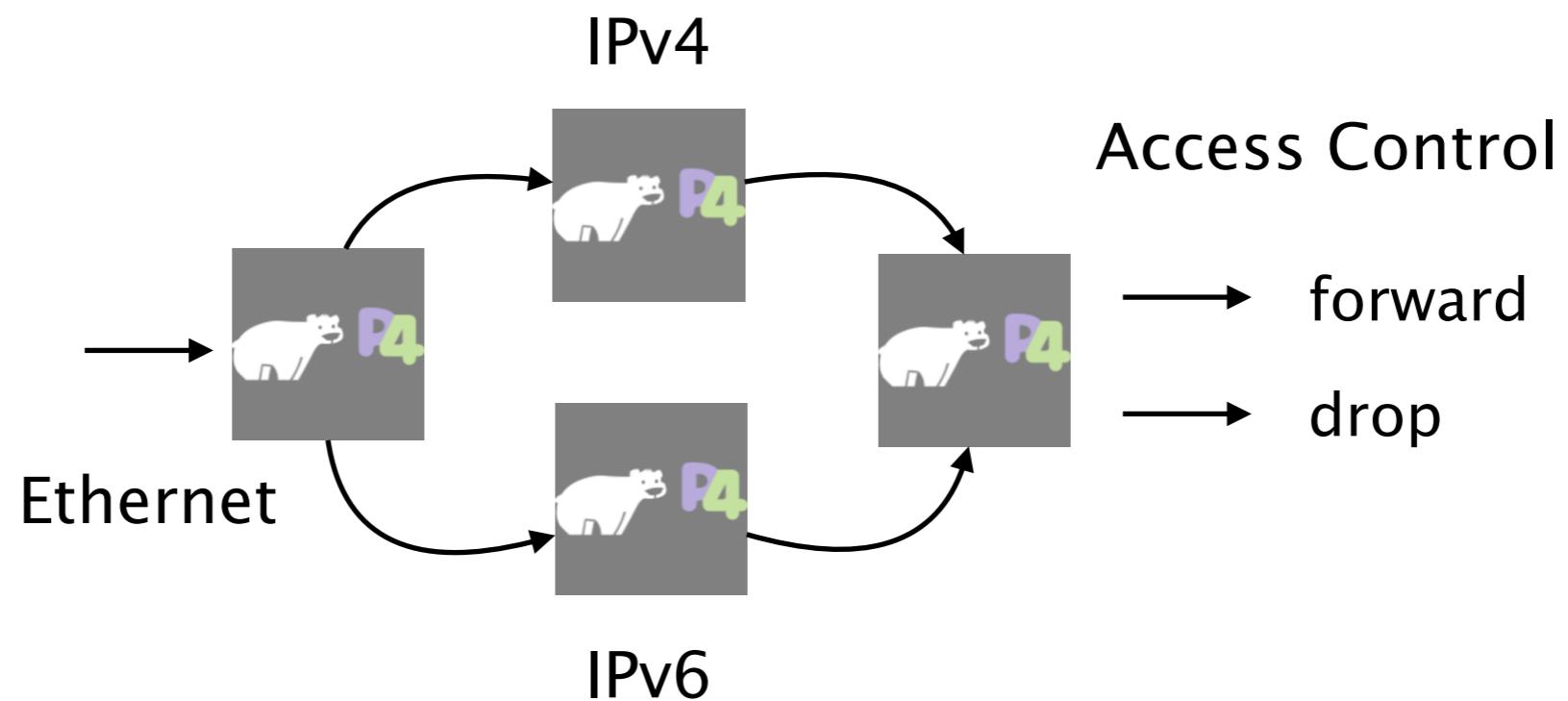


P4 is a domain-specific language which describes how a PISA architecture should process packets

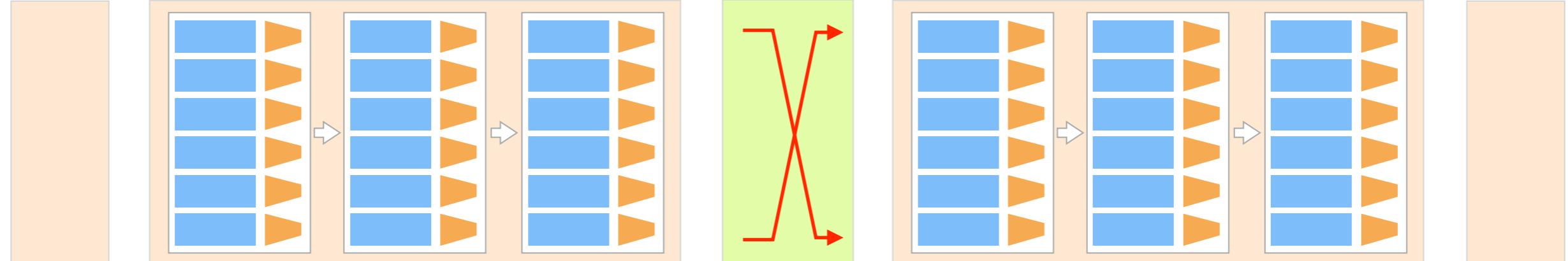


<https://p4.org>

Logical behavior



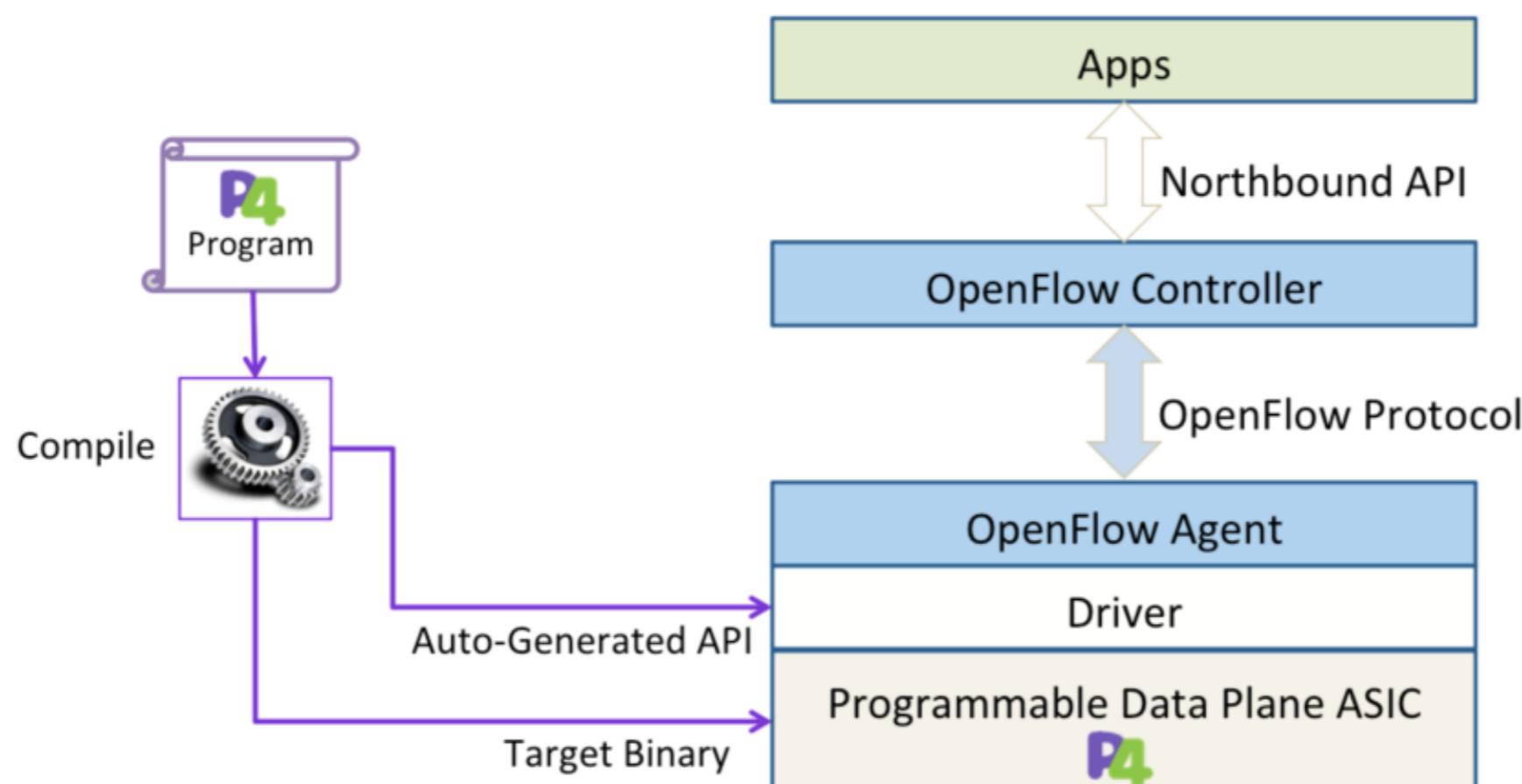
PISA backend



PISA + P4 is strictly more general OpenFlow



P4 & OpenFlow



Copyright © 2016 P4 Language Consortium.

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 7-weeks blocks

Lectures/Exercises

Group project

~7 weeks

how to program in P4

≥ 7 weeks

in teams of 2—3

The course is gonna be divided in
two 7-weeks blocks

Lectures/Exercises

Group project

~7 weeks

how to program in P4

≥ 7 weeks

in teams of 2—3

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

Thu 8—10 Lecture (for 7 weeks)

Thu 10—12 Practical exercises with P4
Exercises are not graded *but* will help at the exam

For now, both will take place in LFW B 3

The course is gonna be divided in
two 7-weeks blocks

Lectures/Exercises

Group project

~7 weeks

how to program in P4

≥ 7 weeks

in teams of 2—3

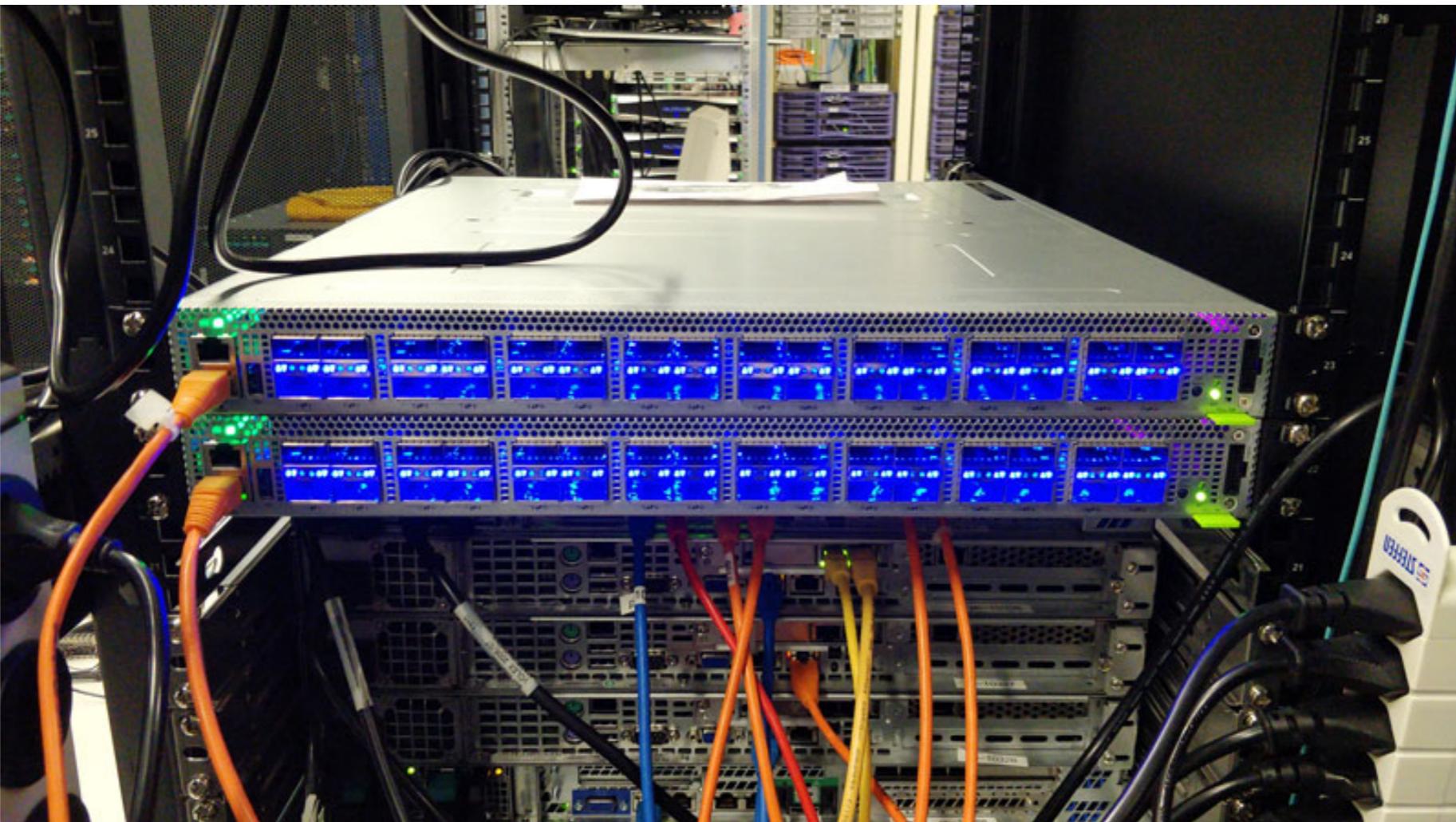
For the project, we'll ask you to develop your own network application

Your can choose your application
we'll provide feedback and a list of default choices

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

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

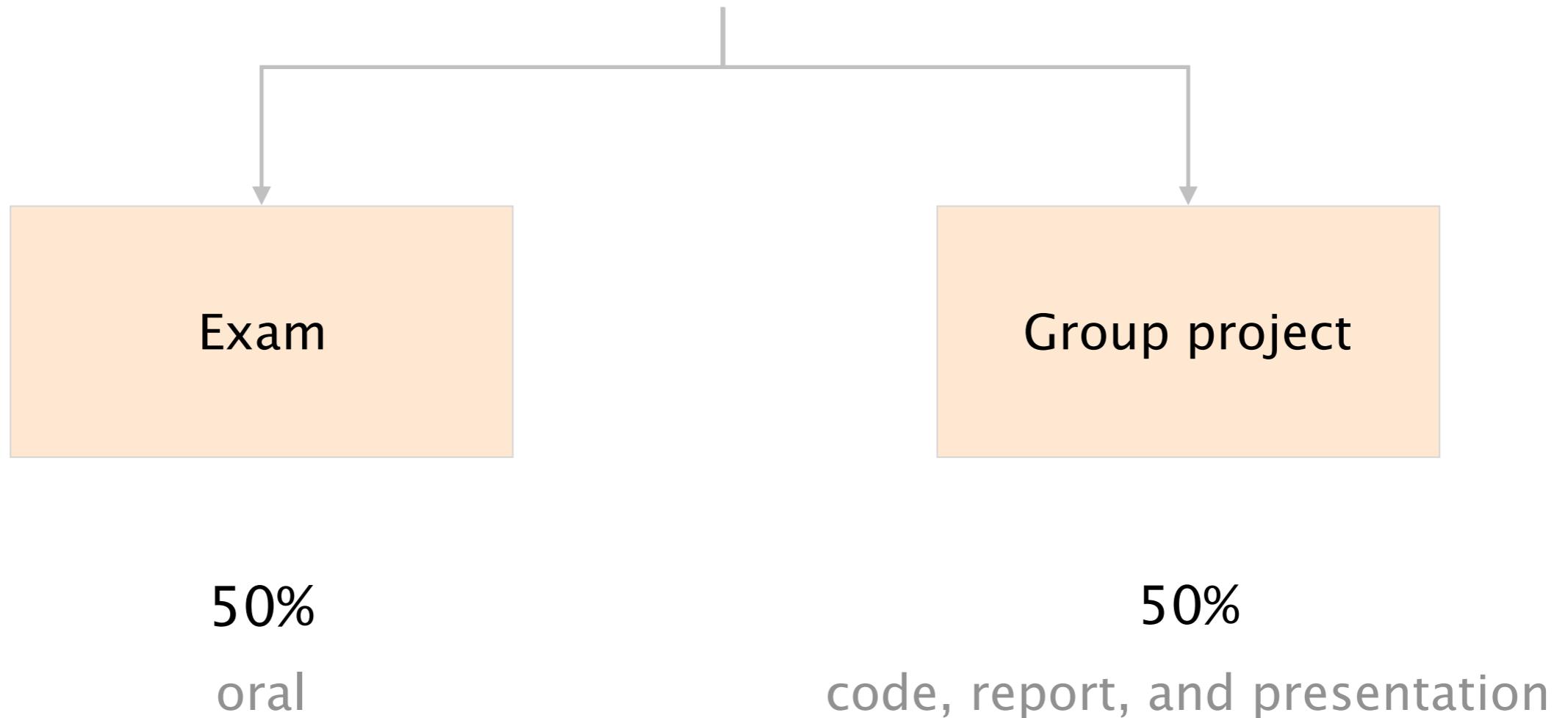
You'll have the opportunity to port your application on real hardware (not mandatory... if you're motivated :-))



Barefoot Tofino Wedge 100BF-32X

3.2 Tbps

Your final grade



Exam

50%

oral

Examples

Design a P4 application
for solving problem <X>

Optimize program <X>

Is program <X> correct?

... important to do the exercises

Your dream team for the semester



Edgar



Roland



Thomas



Maria

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

Check for slides, pointers to exercises, readings, ...

The screenshot shows the homepage of the website <https://adv-net.ethz.ch/>. The header features a background image of a server room with network equipment. On the right, there's a logo for "Networked Systems ETH Zürich seit 2015". The main title "Advanced Topics in Communication Networks" is displayed prominently in white text over a green bar. Below the title, a text box contains a brief description of the class's purpose and focus. The page is divided into three main sections: "Lectures", "Exercises", and "Project", each with a corresponding image and a brief description. At the bottom, there is a "Tentative timeline" section showing a horizontal timeline from September 23 to December 9, with colored bars indicating the schedule for Lectures, Exercises, and Project.

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 2018, the class will concentrate on network programmability and network data plane programming.

Lectures
Weekly lectures in the first part of the semester
(more details coming soon)

Exercises
Ungraded theoretical and practical exercises as well as paper readings (more details coming soon)

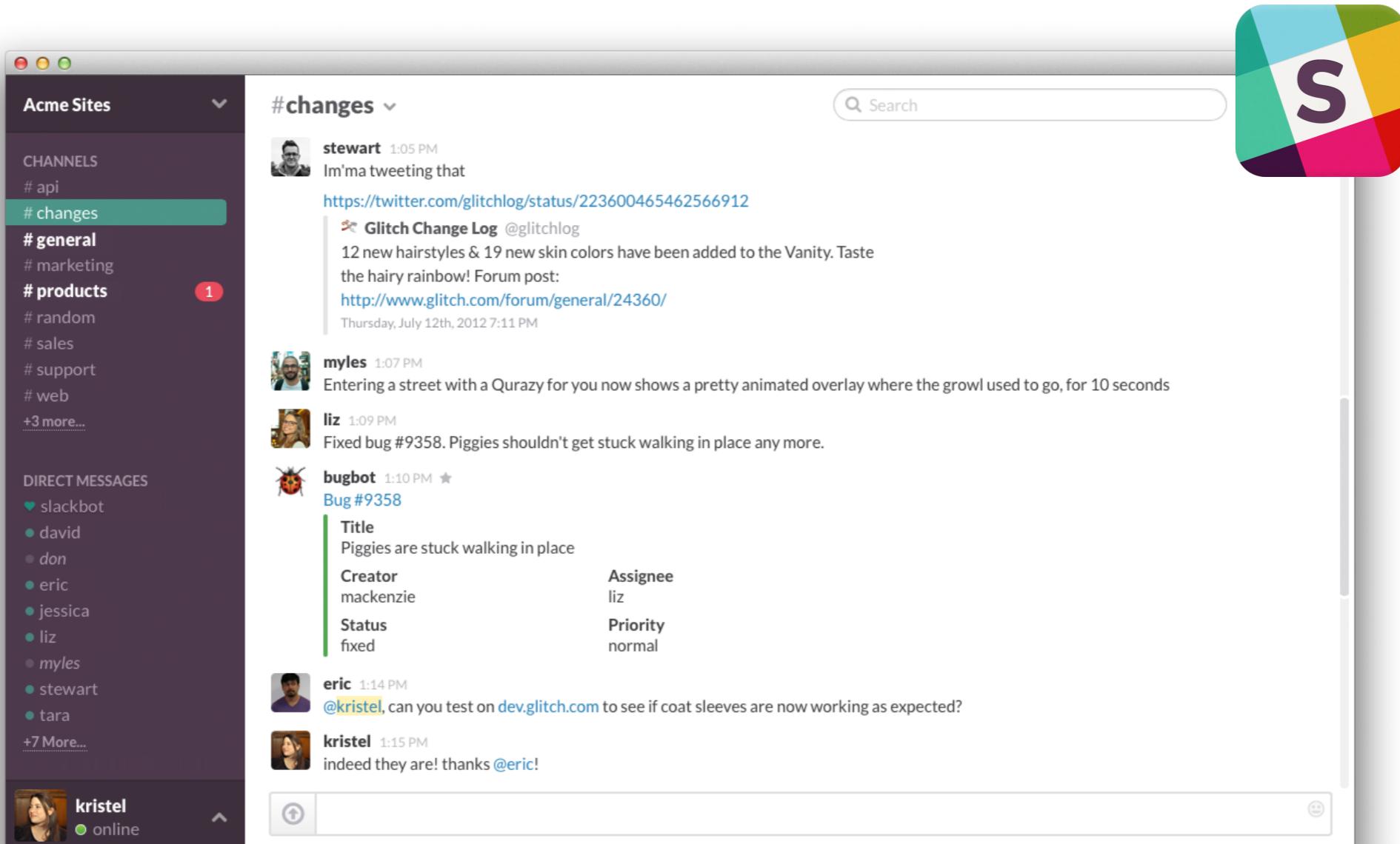
Project
Graded practical project performed in groups
(more details coming soon)

Tentative timeline

Lectures	Weekly lectures (2h per week)
Exercises	Weekly theoretical and practical exercises (2h per week)
Project	Practical project (in groups)

Sep 23 Sep 30 Oct 7 Oct 14 Oct 21 Oct 28 Nov 4 Nov 11 Nov 18 Nov 25 Dec 2 Dec 9

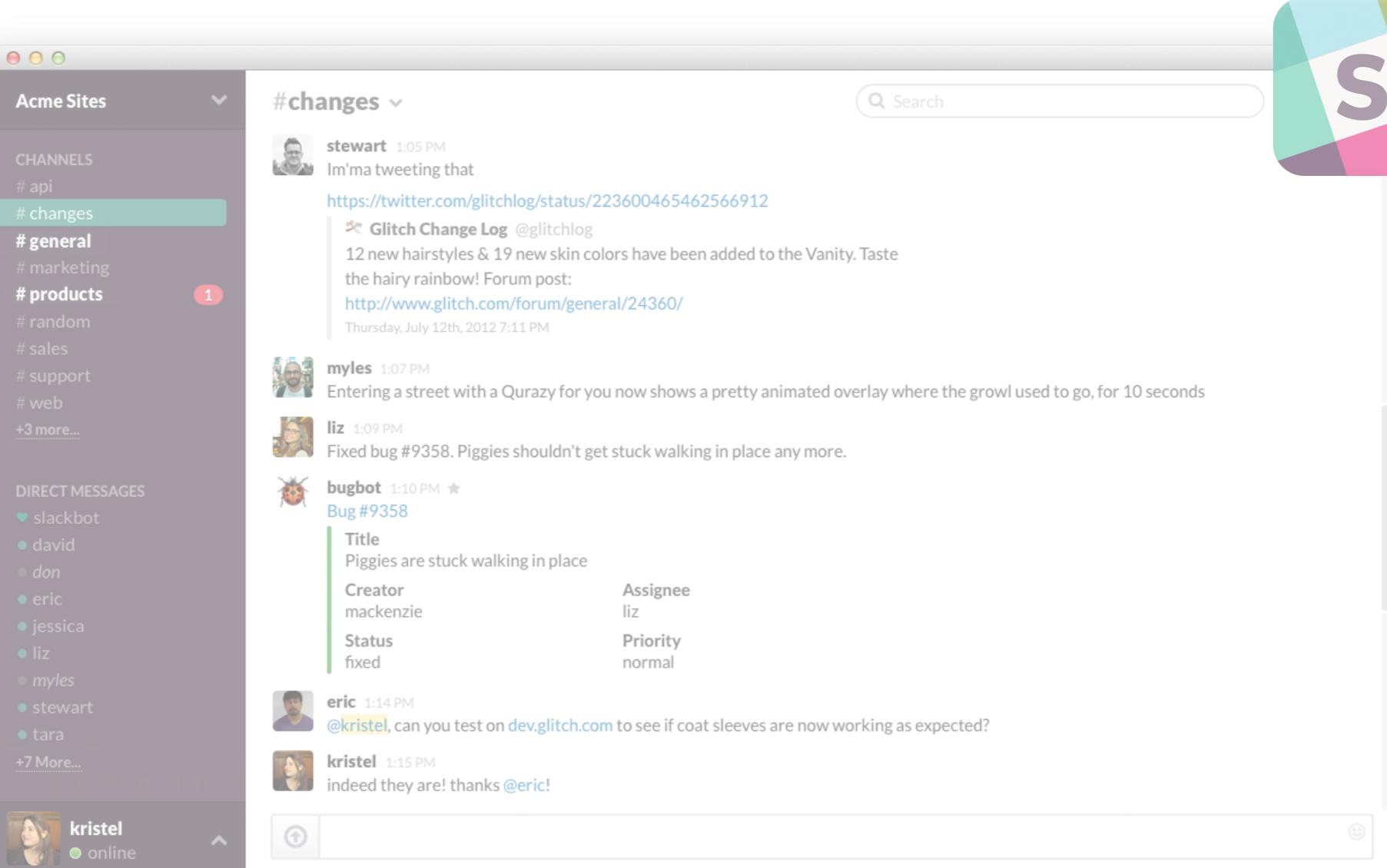
We'll use Slack (chat client) to discuss about the course, exercises, and projects



Web, smartphone and desktop clients available

Register today using your *real* name

> <https://adv-net18.slack.com/signup>



The screenshot displays the Slack application interface. On the left, a sidebar lists channels and direct messages. The main area shows a channel named '#changes'. A message from user 'stewart' at 1:05 PM says 'Im'ma tweeting that' with a link to a Twitter post. Another message from 'stewart' at 1:05 PM is a link to a 'Glitch Change Log' forum post. A message from 'myles' at 1:07 PM discusses a Qurazy overlay. A message from 'liz' at 1:09 PM fixes bug #9358. A message from 'bugbot' at 1:10 PM provides details about bug #9358, including a title, creator, status, assignee, and priority. A message from 'eric' at 1:14 PM asks for testing on dev.glitch.com. A message from 'kristel' at 1:15 PM responds that the issue is fixed. At the bottom, there's a message input field.

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!**

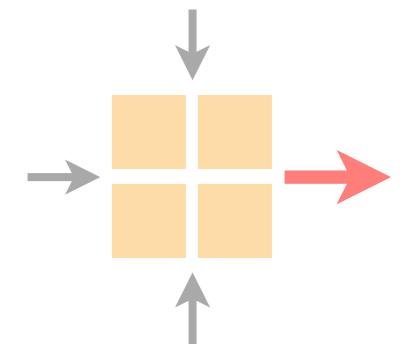
All of the assignments (and the course) will be new, meaning you will act as guinea pigs...



We'll try to take your feedback into account... so shoot!

Advanced Topics in Communication Networks

Programming Network Data Planes



Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

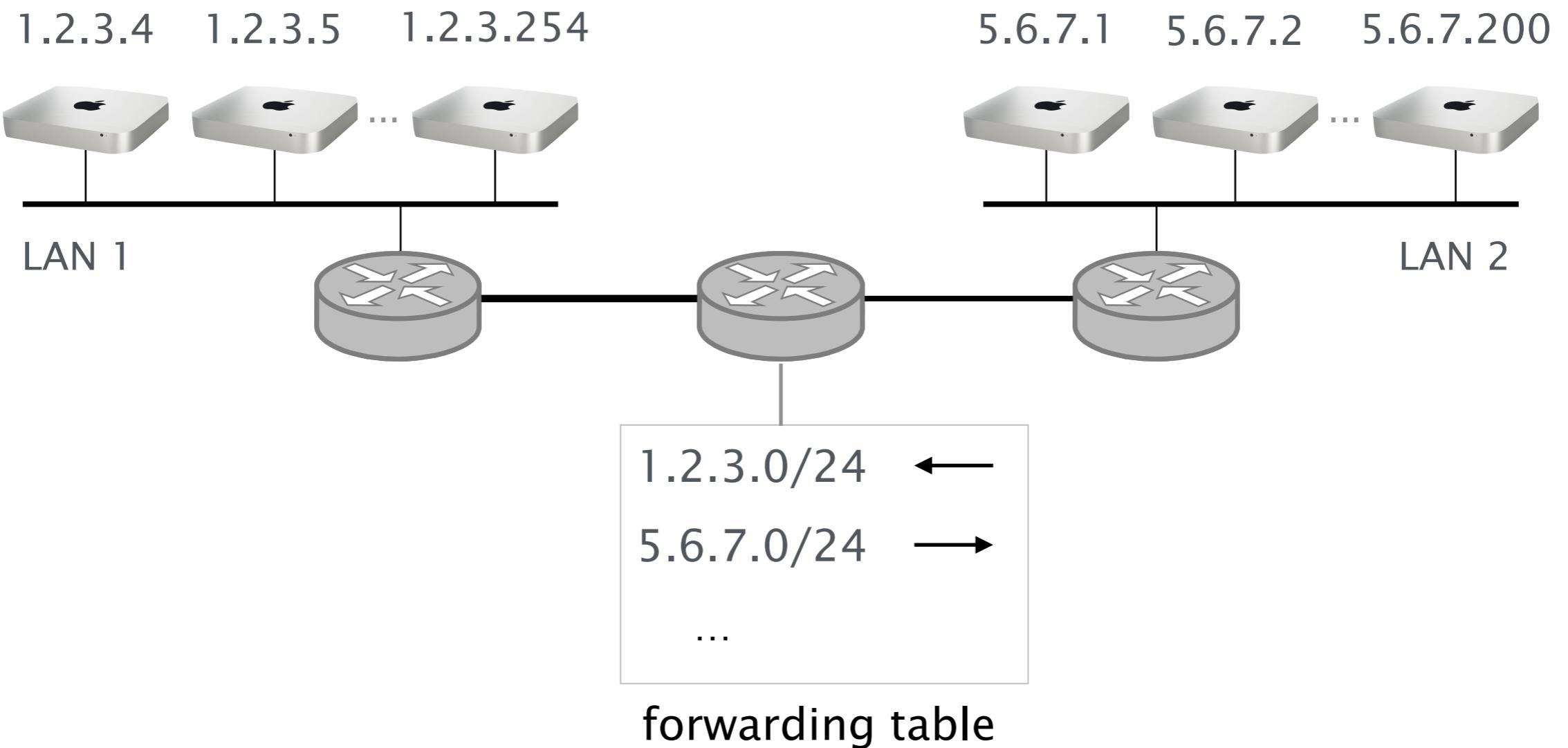
Sep 20 2018

Let's look at one



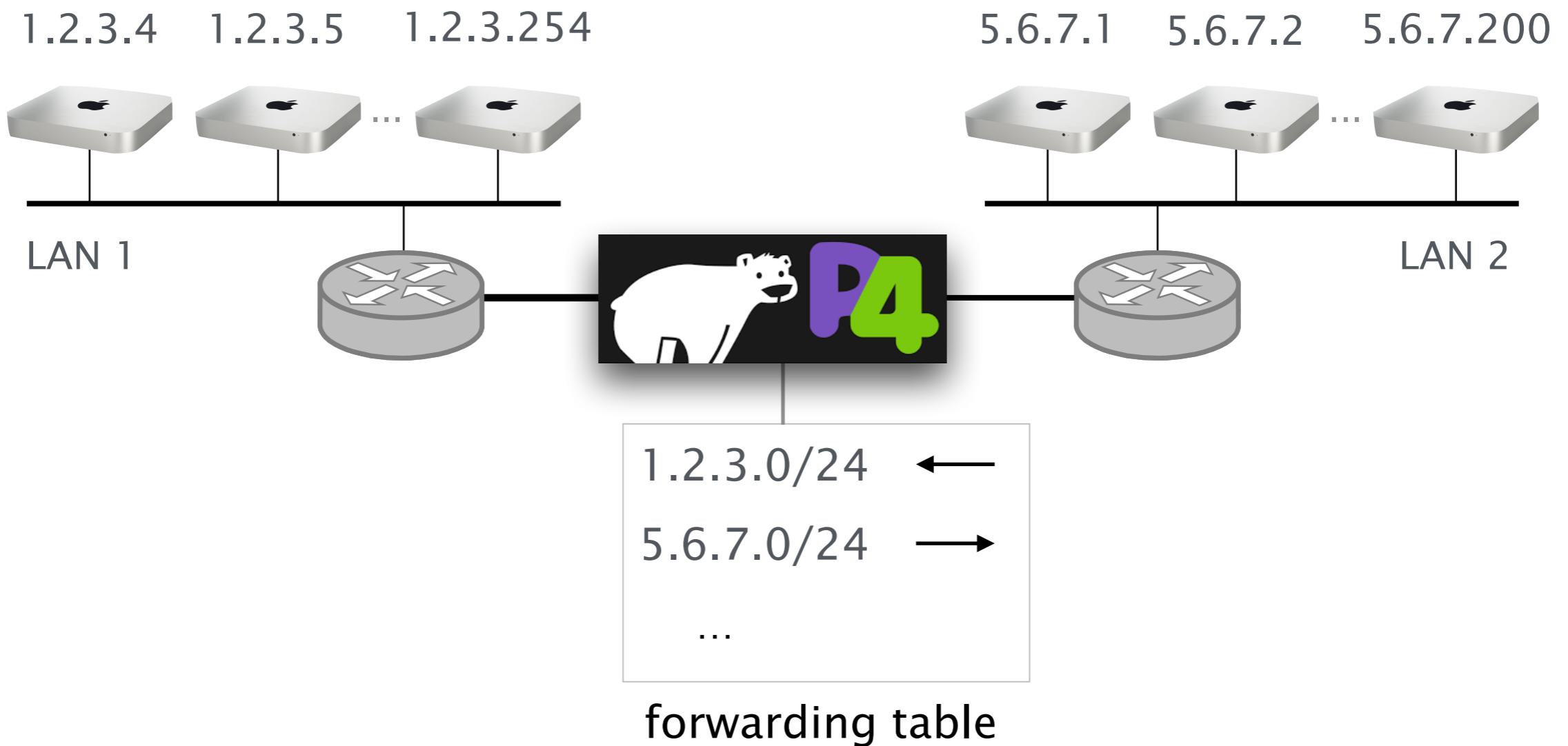
example

IP forwarding in a traditional router



IP forwarding

in a P4?

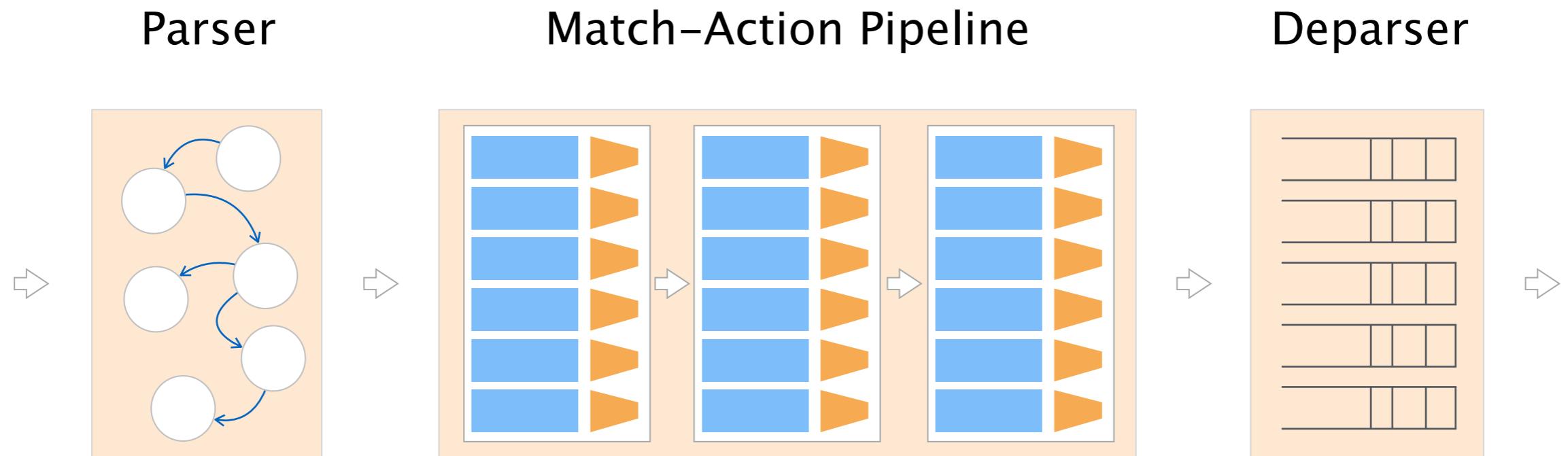


How can we do this in P4?

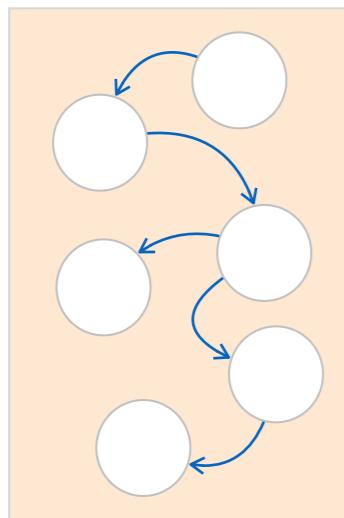
IP forwarding

- Forwarding table lookup
- Update destination MAC
- Decrement TTL
- Send packet to output port

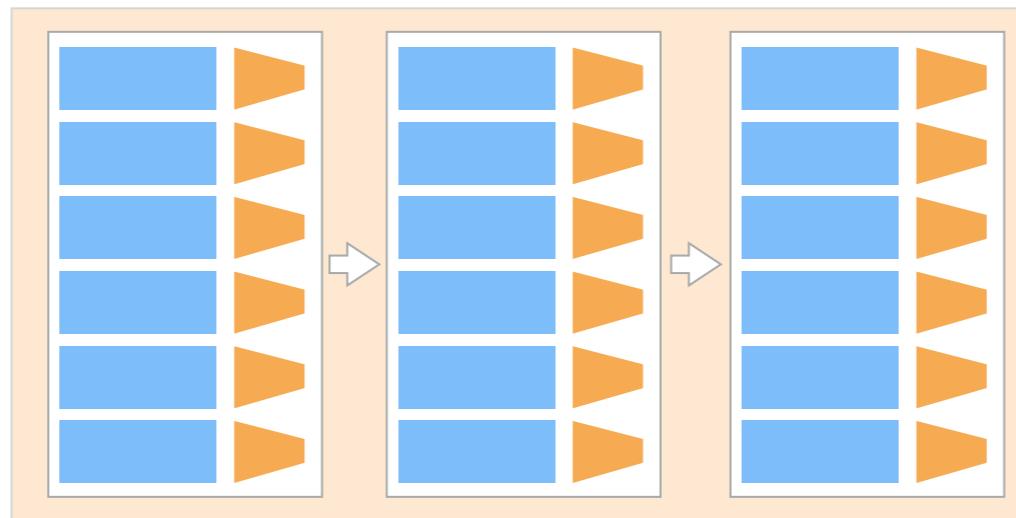
A P4 program consists of three basic parts



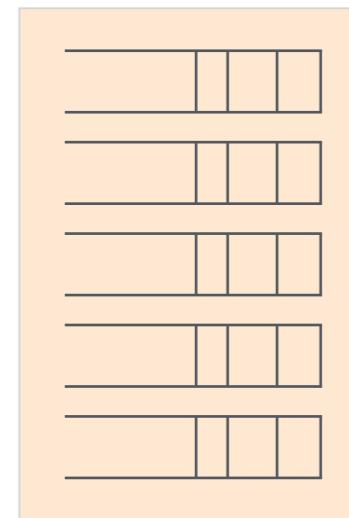
Parser



Match–Action Pipeline

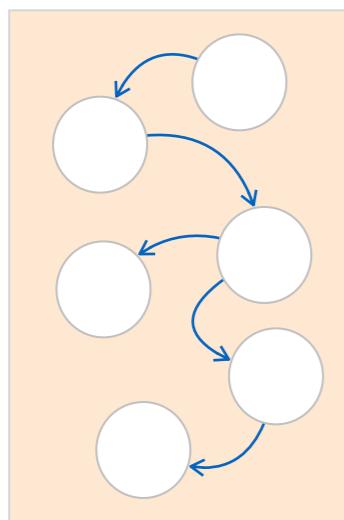


Deparser

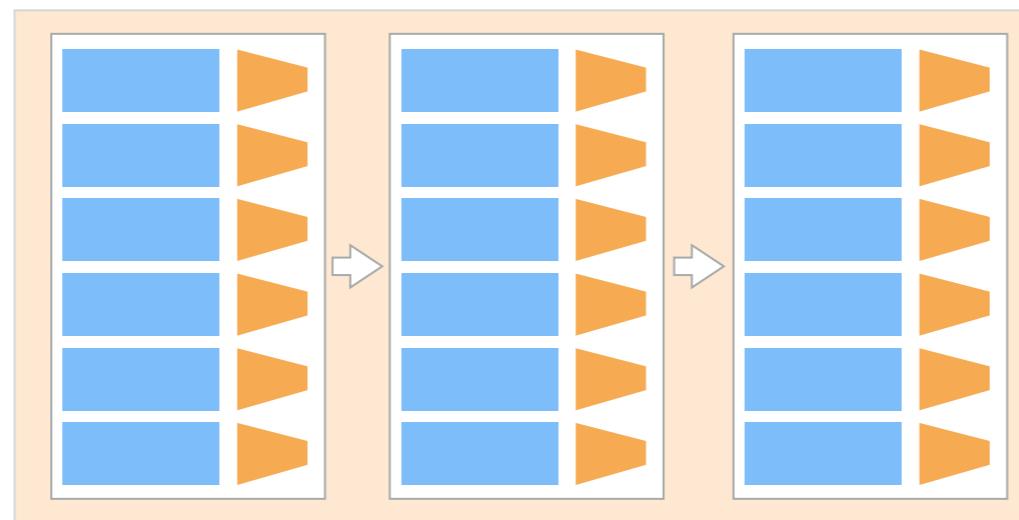


Programmer declares the headers
that should be recognized
and their order in the packet

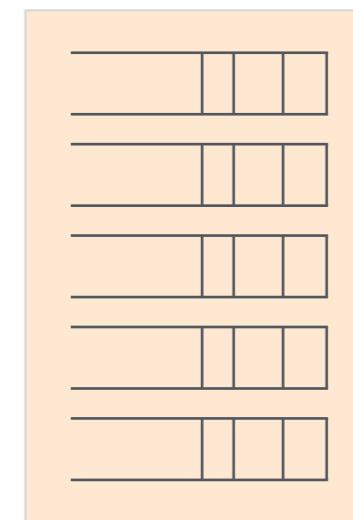
Parser



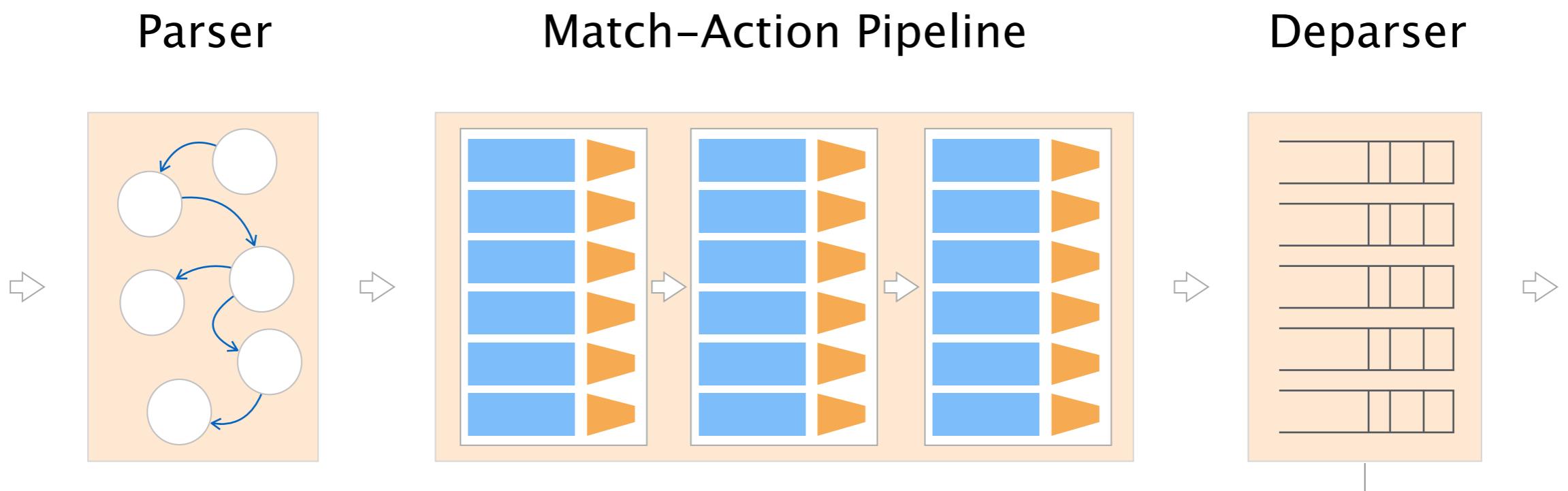
Match–Action Pipeline



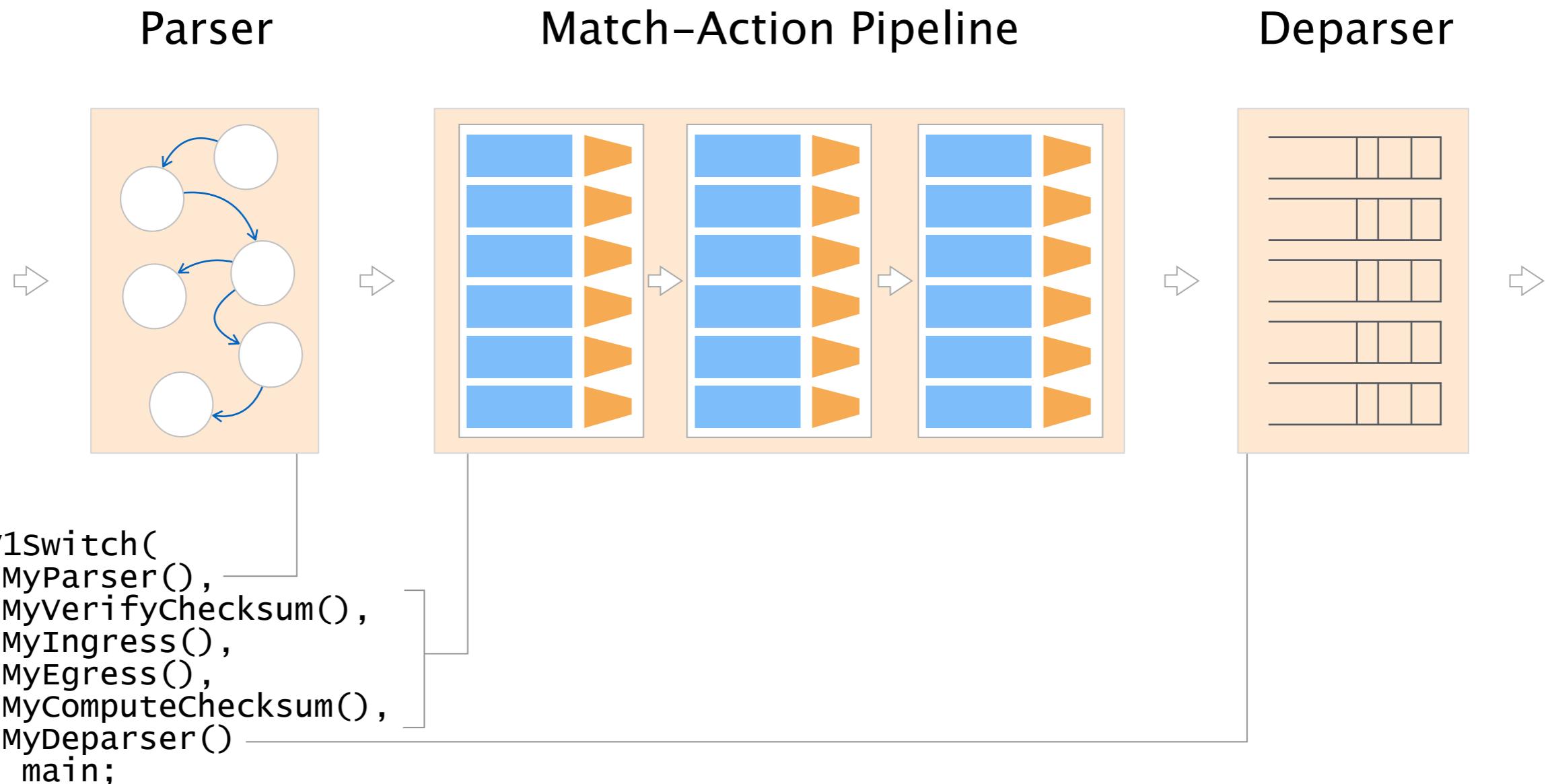
Deparser



Programmer defines the tables
and the exact processing algorithm



Programmer declares
how the output packet
will look on the wire



```
#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_etherent {...}
    state parse_ip4 {...}
}
```

Parse packet headers

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

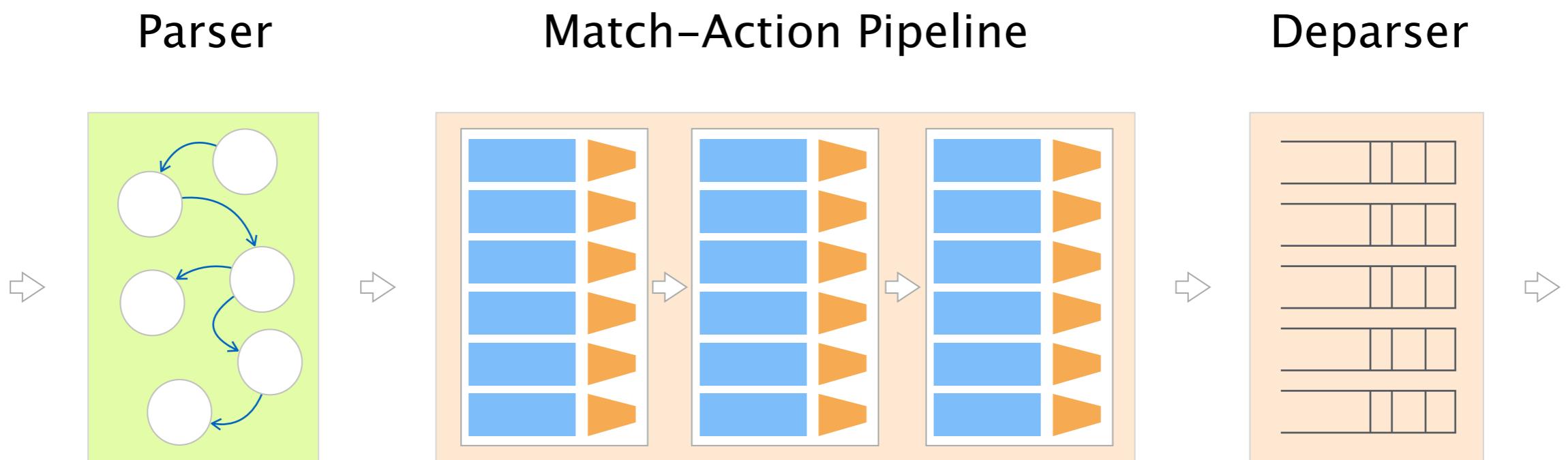
Control flow
to modify packet

```
control MyDeparser(...)
```

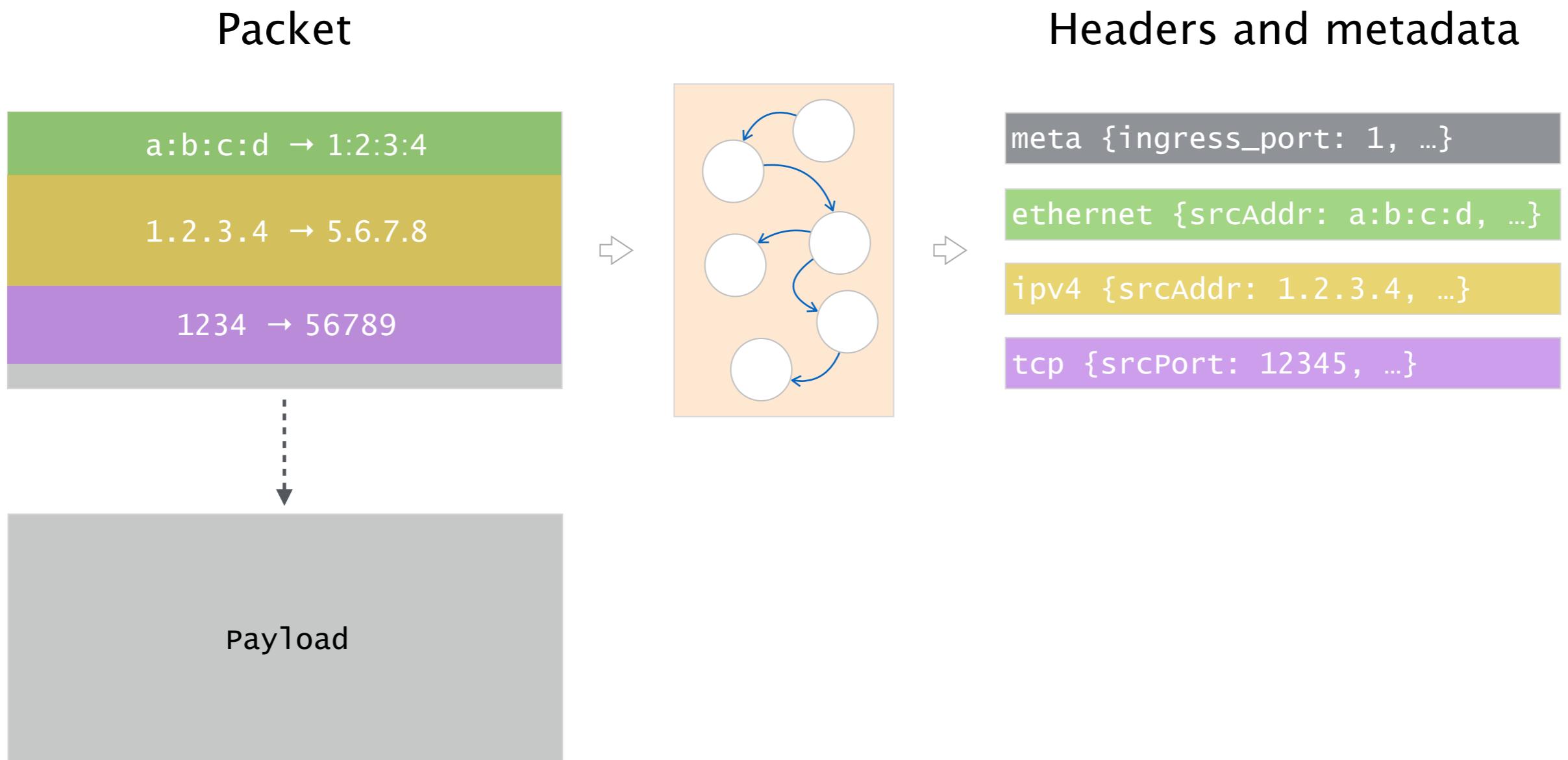
Assemble
modified packet

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

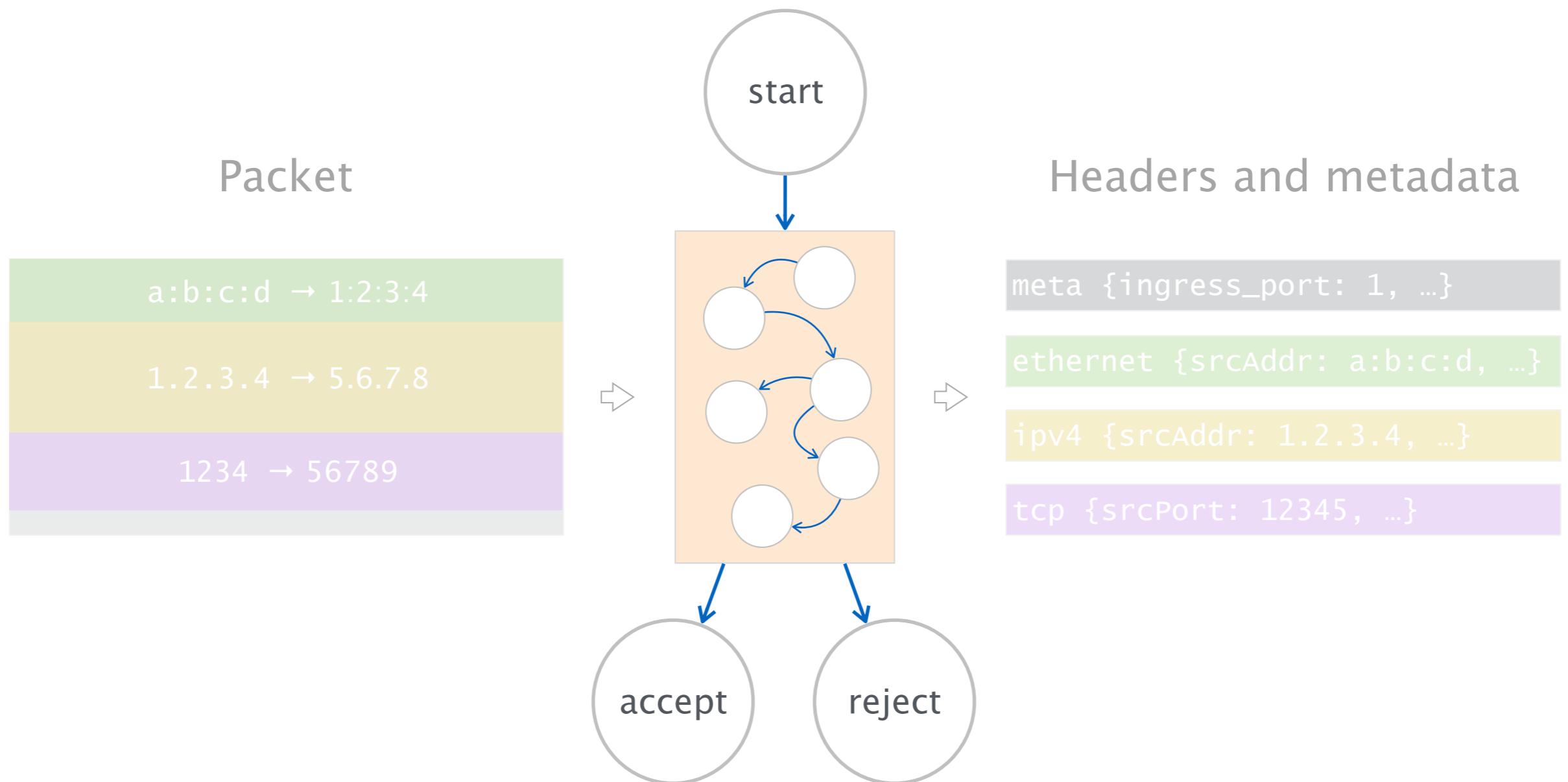
“main()”

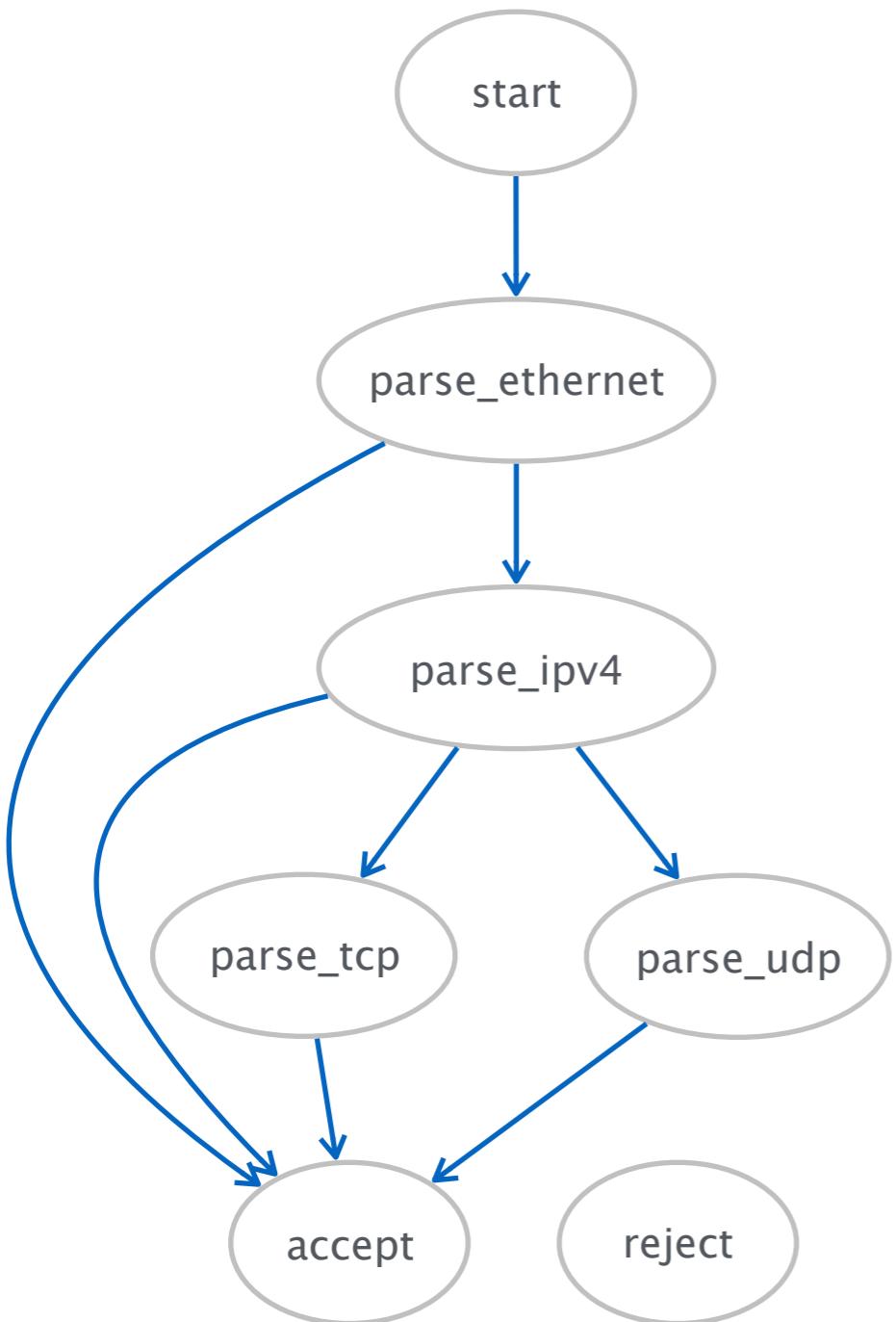


The parser uses a state machine
to map packets into headers and metadata



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;
        }
    }

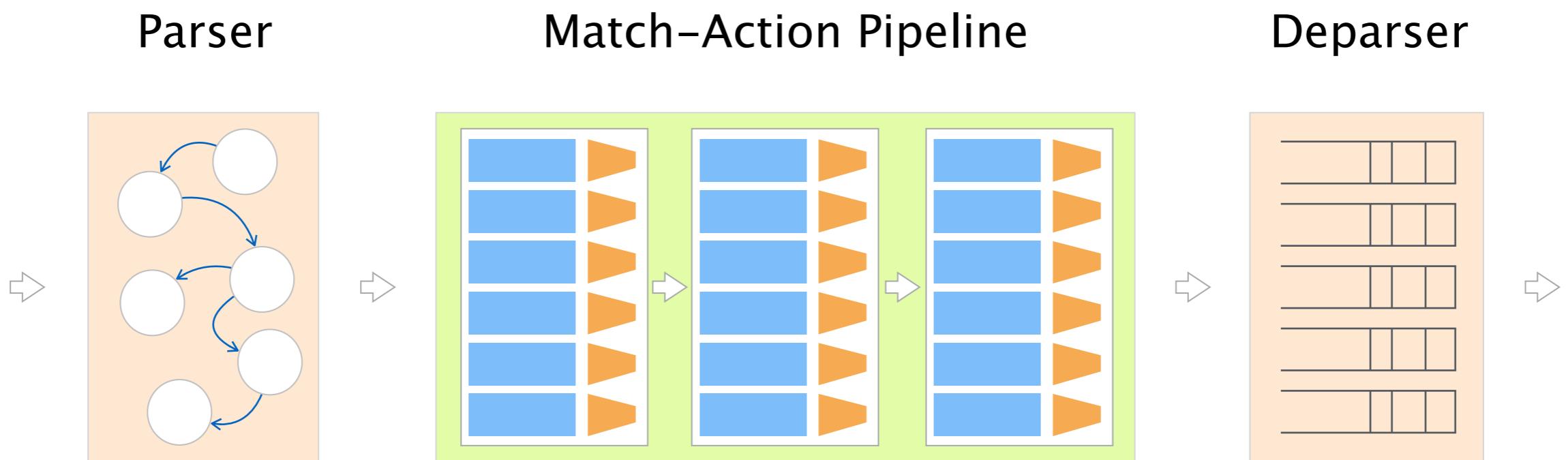
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            6: parse_tcp;
            17: parse_udp;
            default: accept;
        }
    }

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

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

}

```



Control

Similar to functions in C

- declare variables
- create tables
- describe control flow
- ...

Basic building blocks of P4 programs

Control

Control flow

similar to C but without loops

Actions

similar to functions in C

Tables

match a key and return an action

Control

Control flow

similar to C but without loops

Actions

similar to functions in C

Tables

match a key and return an action

Controls can apply changes to packets

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

similar to C but without loops

Actions

similar to functions in C

Tables

match a key and return an action

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

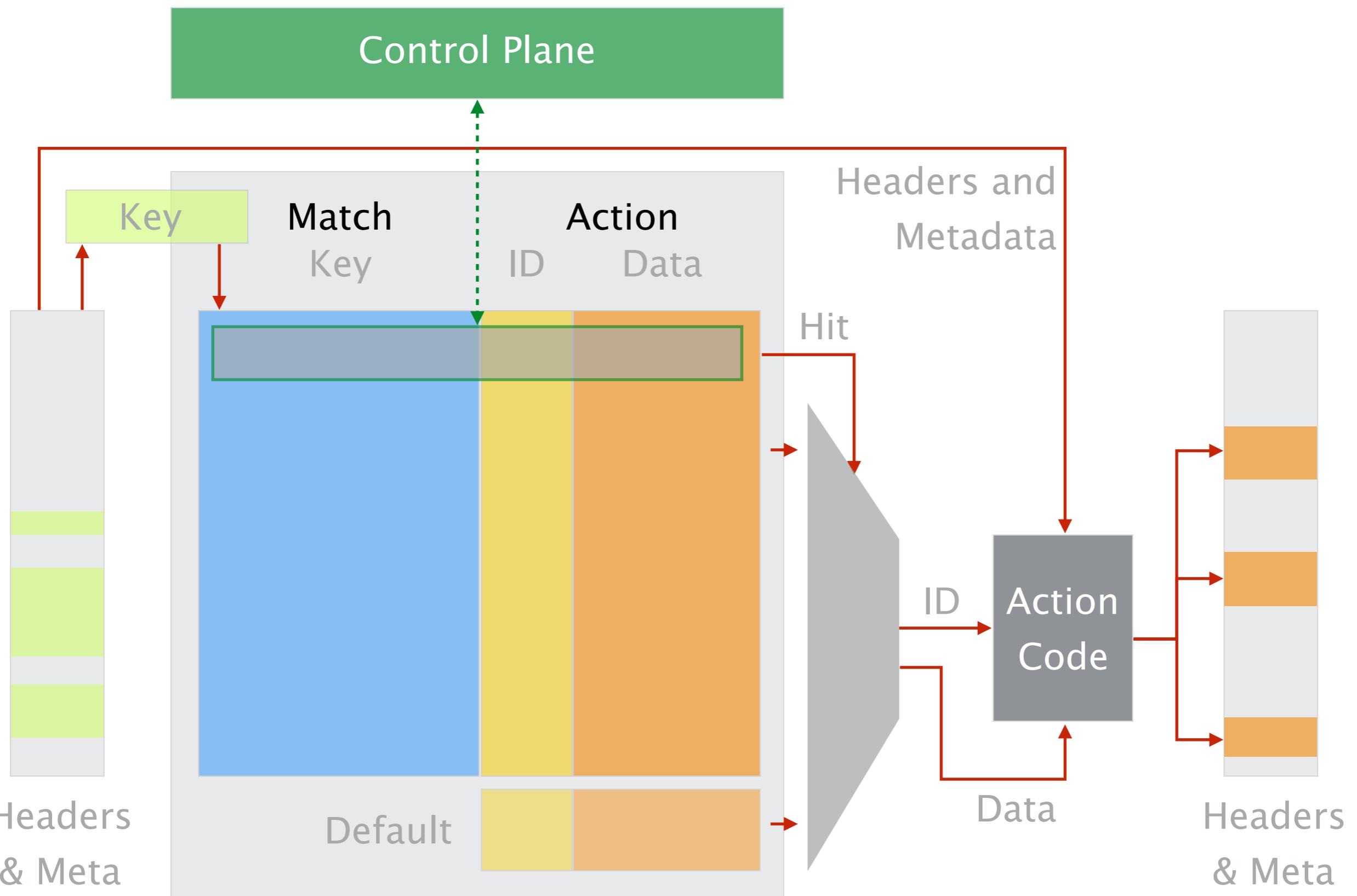
similar to C but without loops

Actions

similar to functions in C

Tables

match a key and return an action

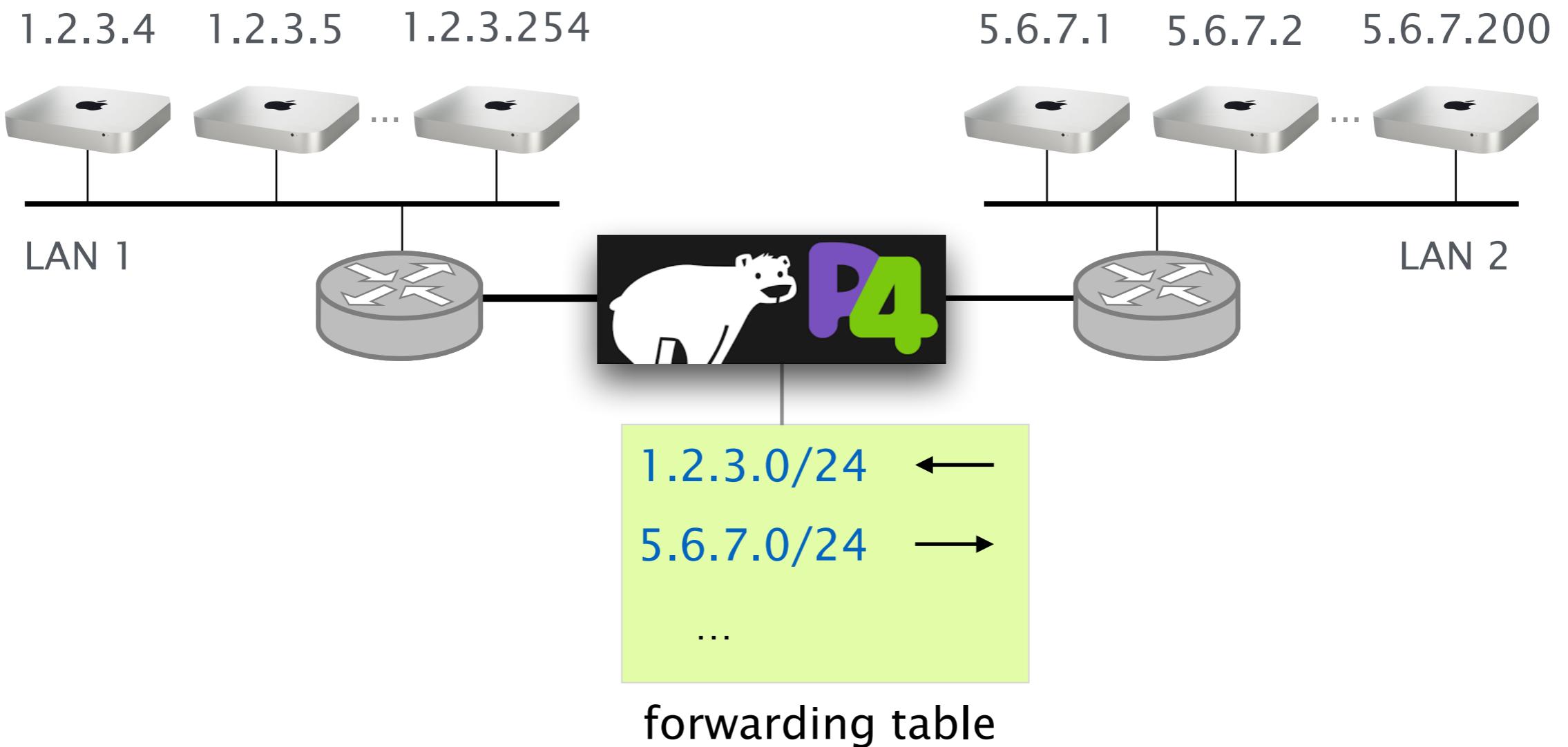


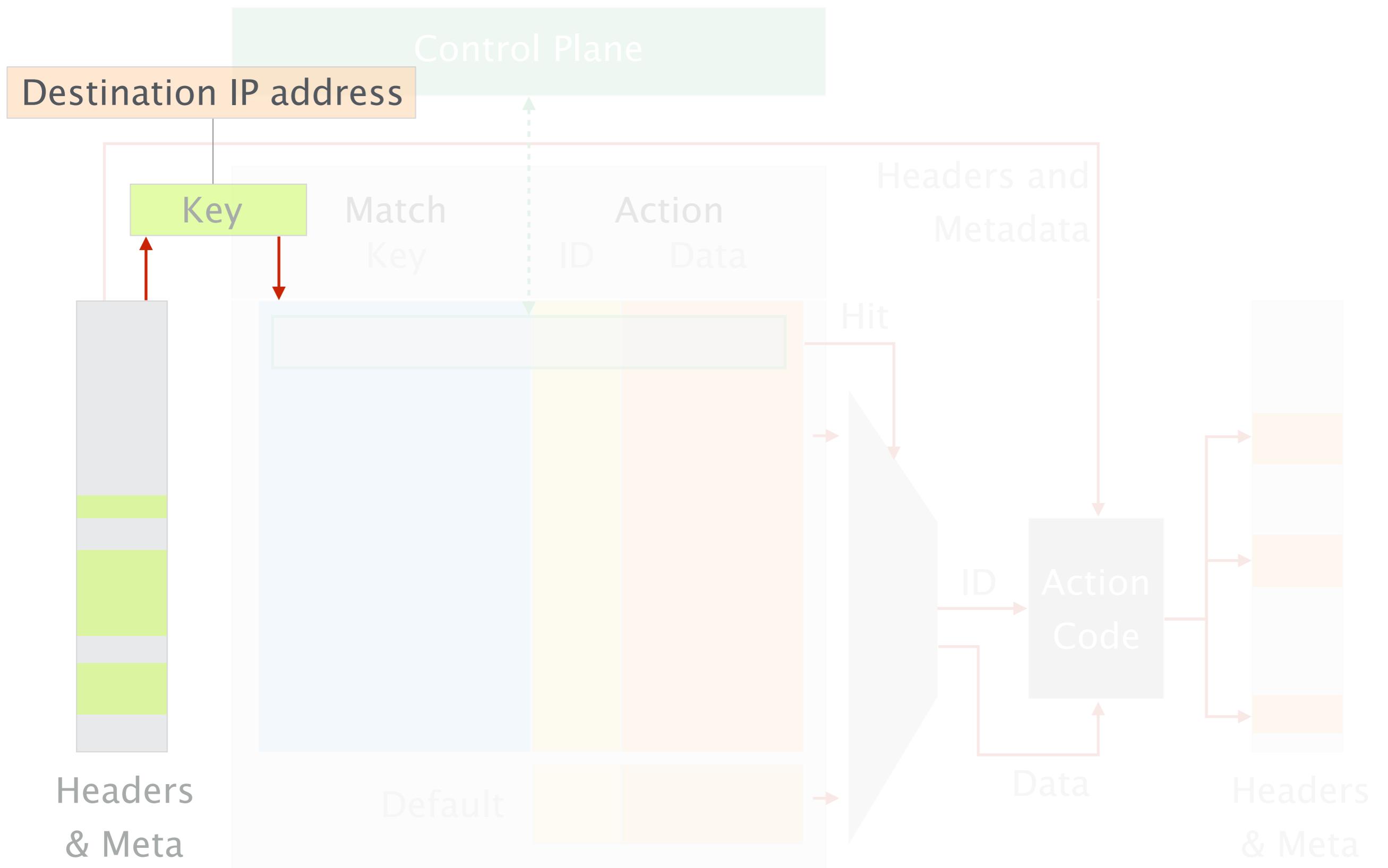
```
table [ ] {  
    key = { [ ] : [ ]; } [ ] ;  
    actions = { [ ] } [ ] ;  
    size = [ ] ;  
    default_action = [ ] ; } [ ]
```

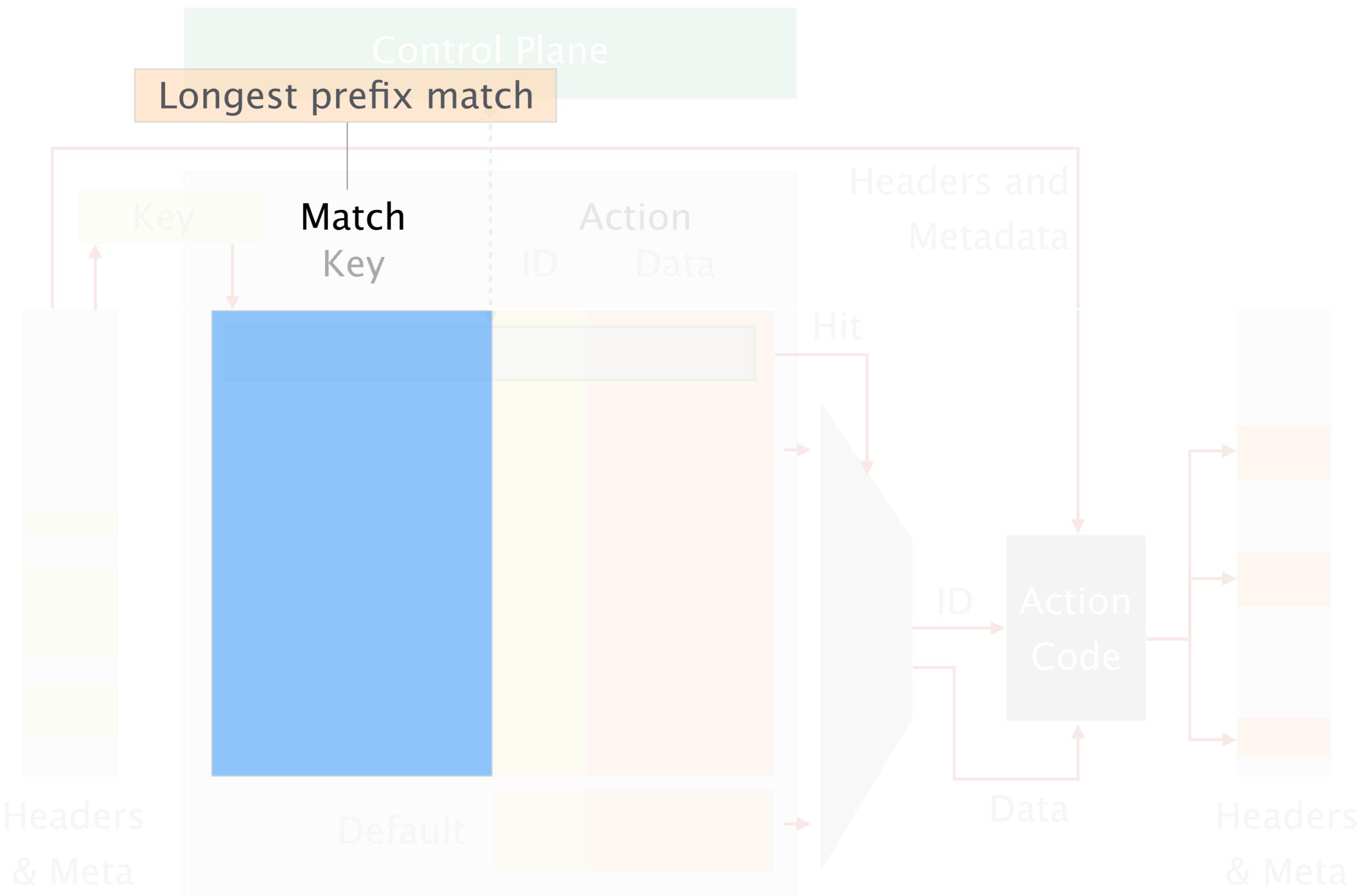
The diagram illustrates the structure of a table configuration. It highlights several parts with gray boxes and connects them to orange callout boxes labeled:

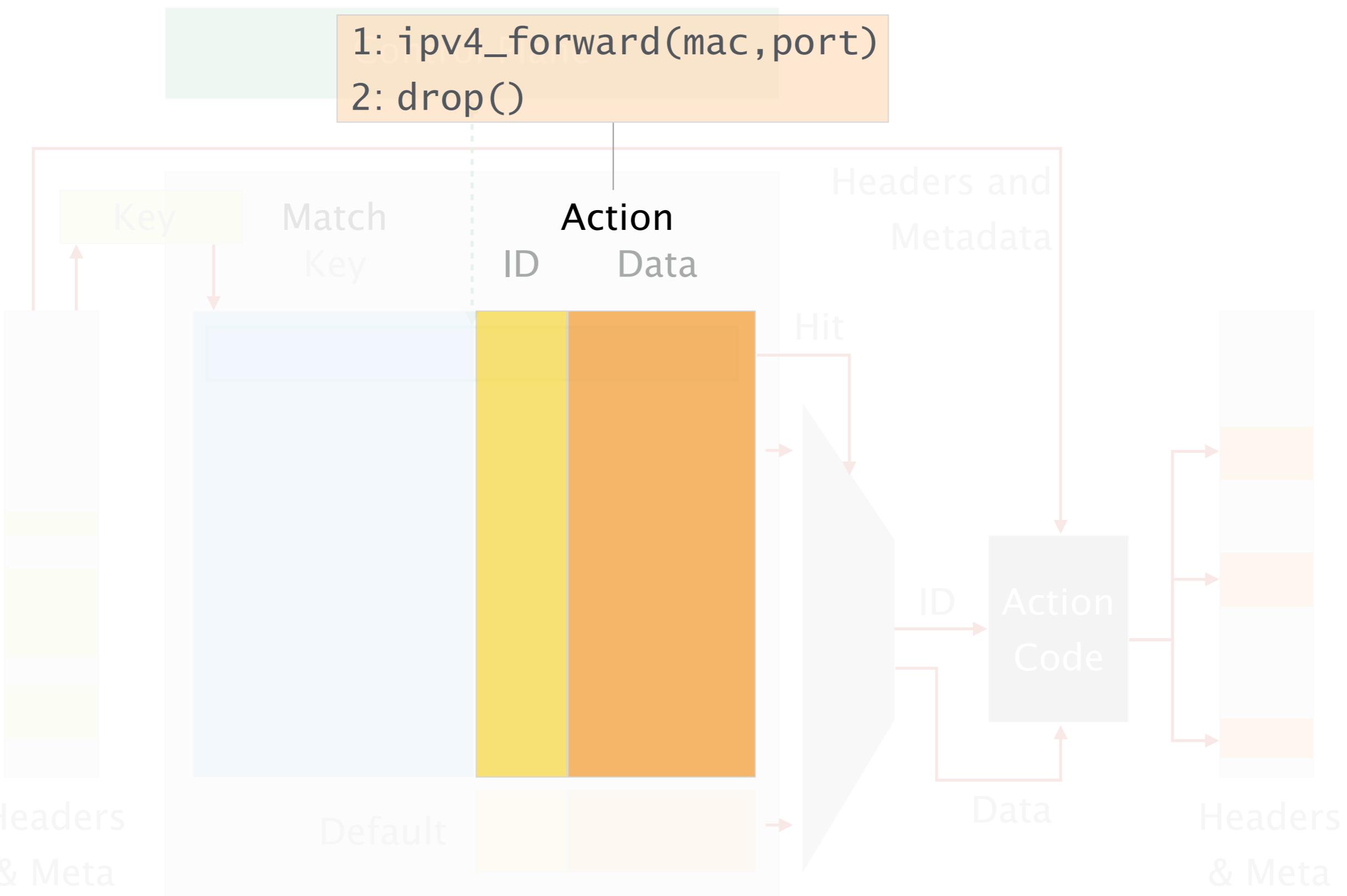
- Table name: Points to the first part of the table definition.
- Field(s) to match: Points to the key field definition.
- Match type: Points to the colon and semicolon in the key field definition.
- Possible actions: Points to the actions field definition.
- Max. # entries in table: Points to the size field definition.
- Default action: Points to the default_action field definition.

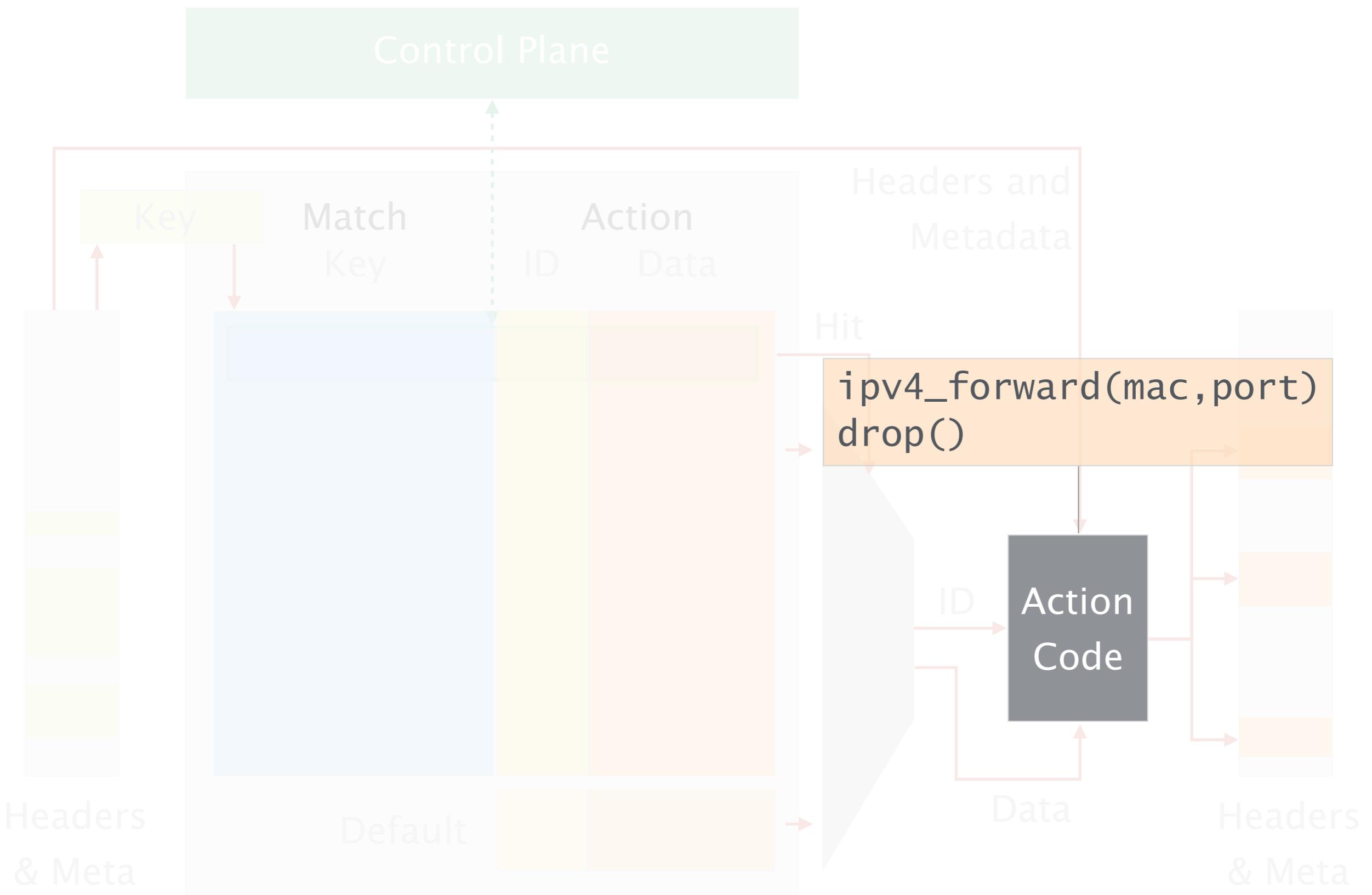
Example: IP forwarding table











```
table ipv4_1pm {  
    key = {  
        hdr.ipv4.dstAddr: 1pm;  
    }  
    actions = {  
        ipv4_forward;  
        drop;  
    }  
  
    size = 1024;  
  
    default_action = drop();  
}
```

Table name

Destination IP address

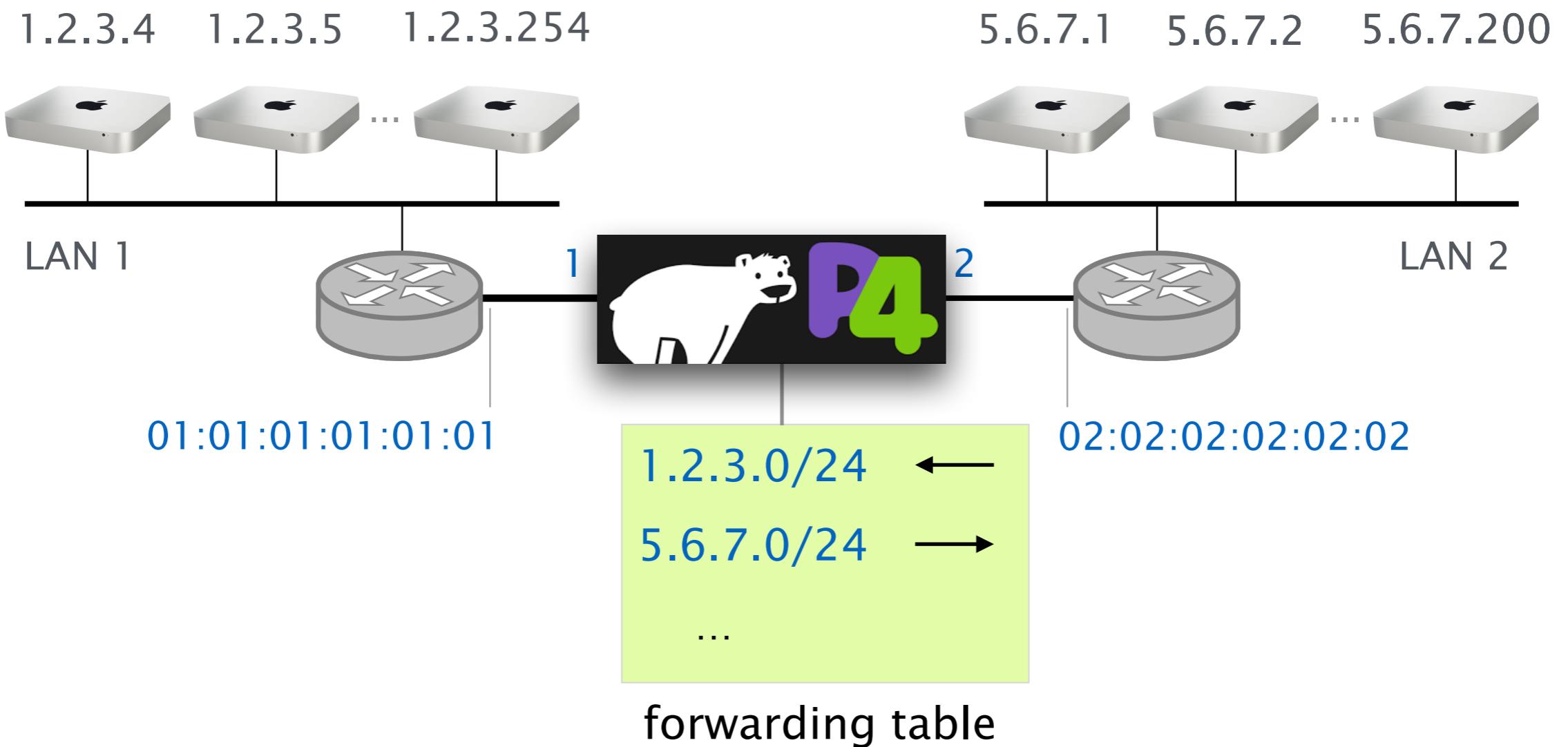
Longest prefix match

Possible actions

Max. # entries in table

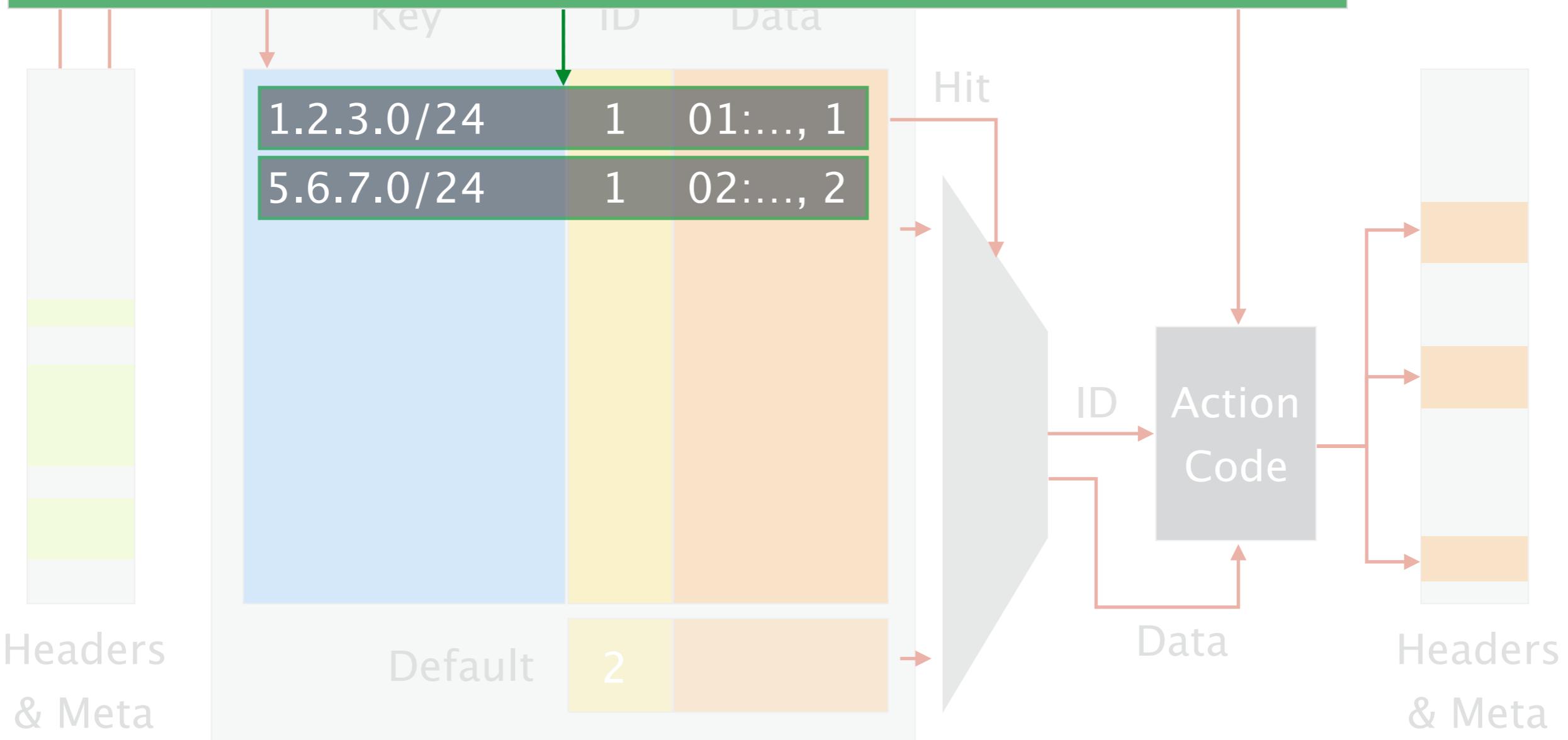
Default action

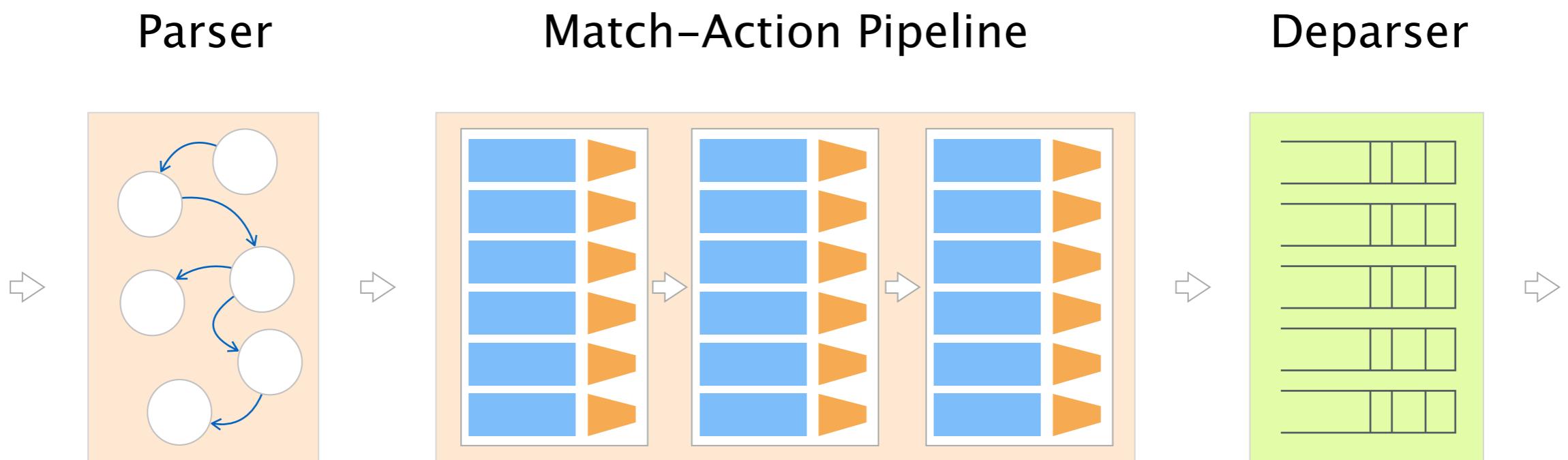
Example: IP forwarding table



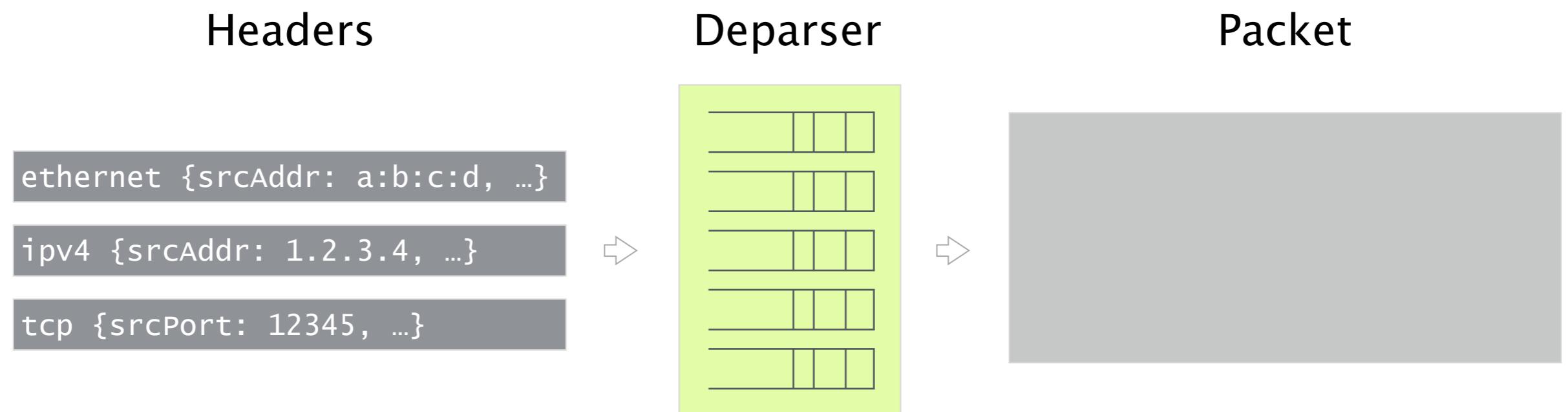
Control Plane

```
table_add ipv4_1pm ipv4_forward 1.2.3.0/24 => 01:01:01:01:01:01 1  
table_add ipv4_1pm ipv4_forward 5.6.7.0/24 => 02:02:02:02:02:02 2
```





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



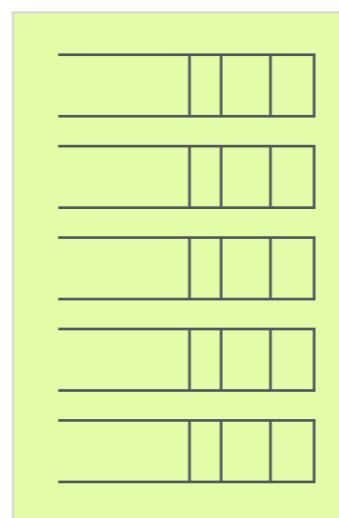
Headers

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

```
ip4 {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);
    }
}
```

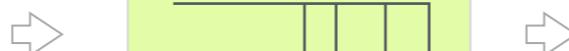
Headers

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

```
ip4 {srcAddr: 1.2.3.4, ...}
```

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

Deparser



Packet

```
a:b:c:d → 1:2:3:4
```

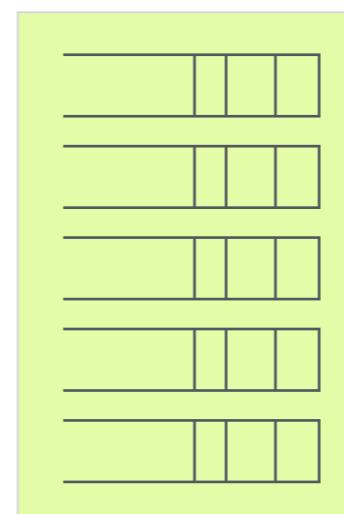
```
1.2.3.4 → 5.6.7.8
```

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.etheren);
        packet.emit(hdr.ip4);
    }
}
```

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  
1234 → 56789
```

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

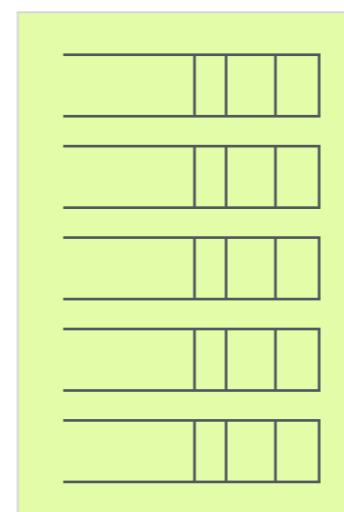
Headers

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

ipv4 {srcAddr: 1.2.3.4, ...}

tcp {srcPort: 12345, ...}

Deparser



Packet

a:b:c:d → 1:2:3:4

1.2.3.4 → 5.6.7.8

1234 → 56789

Payload