Advanced Topics in Communication Networks

# Internet Routing and Forwarding

Laurent Vanbever

nsg.ee.ethz.ch

22 Sep 2020

Lecture starts at **14:15**

Materials inspired from Olivier Bonaventure, David Andersen, Jennifer Rexford, and p4.org

Last week on

# Advanced Topics in Communication Networks

In this lecture, you'll learn

how to optimize the

- performance
- flexibility
- reliability

of large-scale network infrastructures

In this lecture, you'll learn

how to optimize the

- performance
- flexibility
- reliability

of large-scale network infrastructures

Why should *you* care, **as a user?**

best effort service

IP routers do their best to deliver packets to their destination without making any promises

For many applications and users
doing "its best" is just not good enough anymore

In this lecture, you'll learn

how to optimize the

- performance
- flexibility
- reliability

of large-scale network infrastructures

Why should *you* care, as a **network operator?**

IP networks carry always more critical services

# IP networks carry always more traffic

always faster user speeds

x

always more users

x

always more content consumed

Techniques

Performance

Flexibility

Reliability

Traffic Engineering

Load Balancing

Quality of Service

Multicast

Virtual Private Networks

Fast Convergence

Besides learning about the techniques per-se,

you'll learn how to operate and implement them


operate

your own IP network

in virtual labs


implement

your own forwarding logic

in programmable networks

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

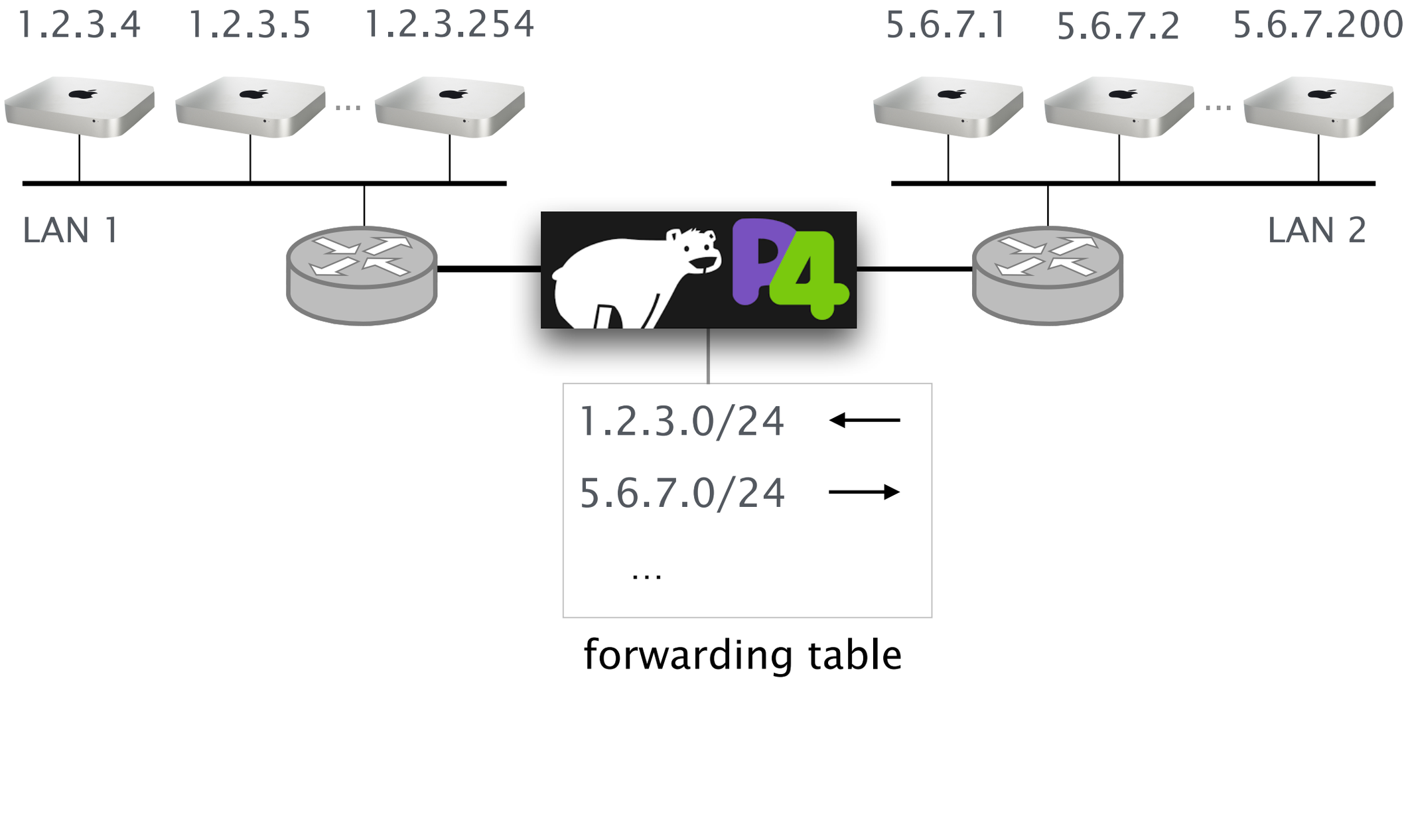
https://p4.org

# Your first


# program

# IP forwarding

## in P4?



This week on

# Advanced Topics in Communication Networks

We will dive into the P4 ecosystem and look at Multiprotocol Label Switching

| P4 environment | P4 language | label switching |
| --- | --- | --- |
| What is needed to program in P4? | Deeper-dive into the language constructs | the basics |

P4 environment

P4 language

label switching

What is needed to program in P4?

# Quick historical recap

July

Initial paper

P4$_{14}$ v1.0.1

v1.0.2

v1.0.3

v1.0.4

December

P4$_{16}$ specification (draft)


P4$_{14}$ specification

September

P4$_{16}$ specification

May

P4$_{14}$ v1.0.1

v1.0.2

v1.0.3

v1.0.4

July

Initial paper

December

P4$_{16}$ specification (draft)

2014

2015

2016

2017

2018

P4$_{14}$ specification

September

P4$_{16}$ specification

focus of the lecture

# P4$_{16}$ introduces the concept of an architecture

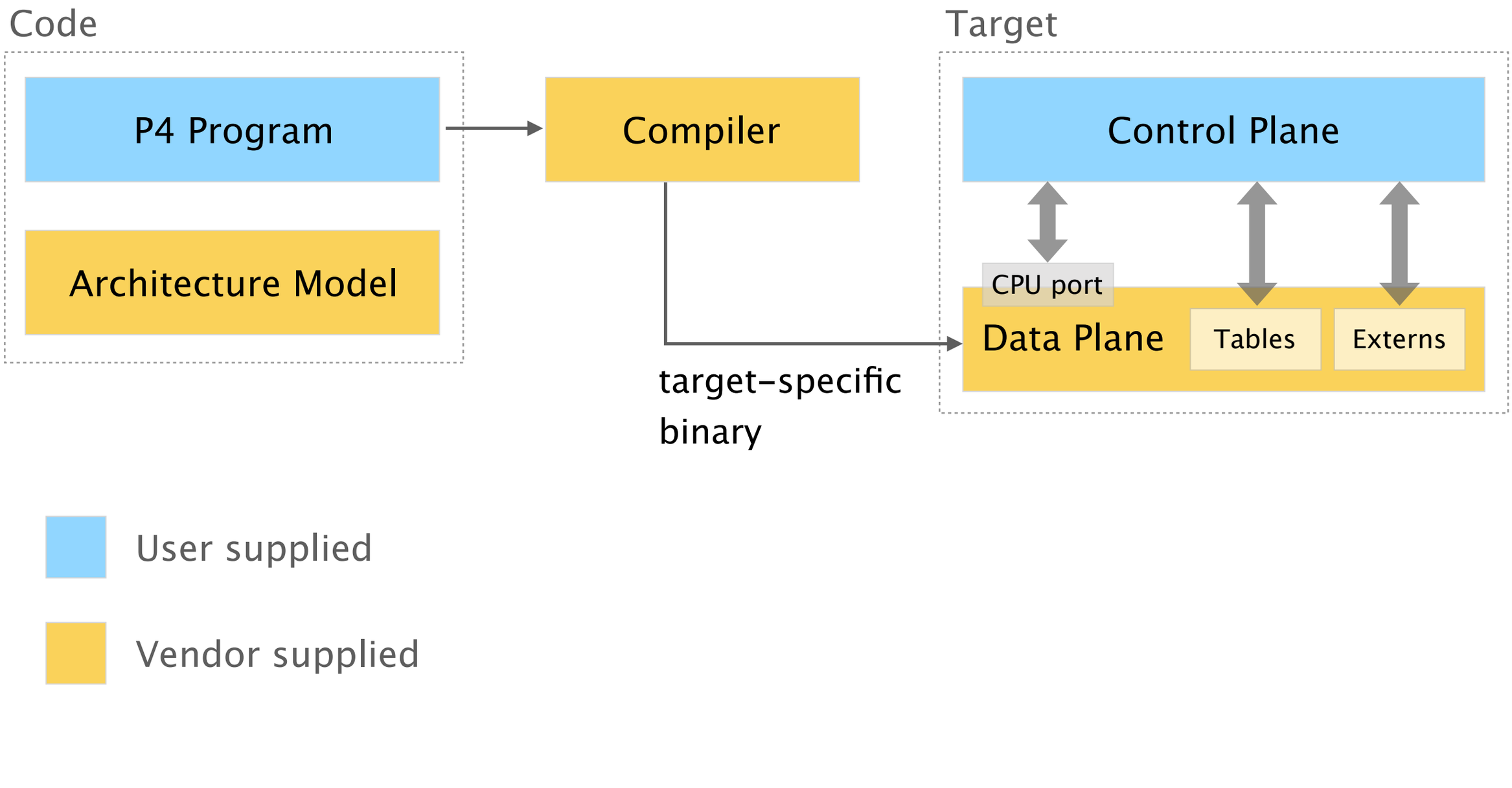
P4 Target

a model of a specific hardware implementation

P4 Architecture

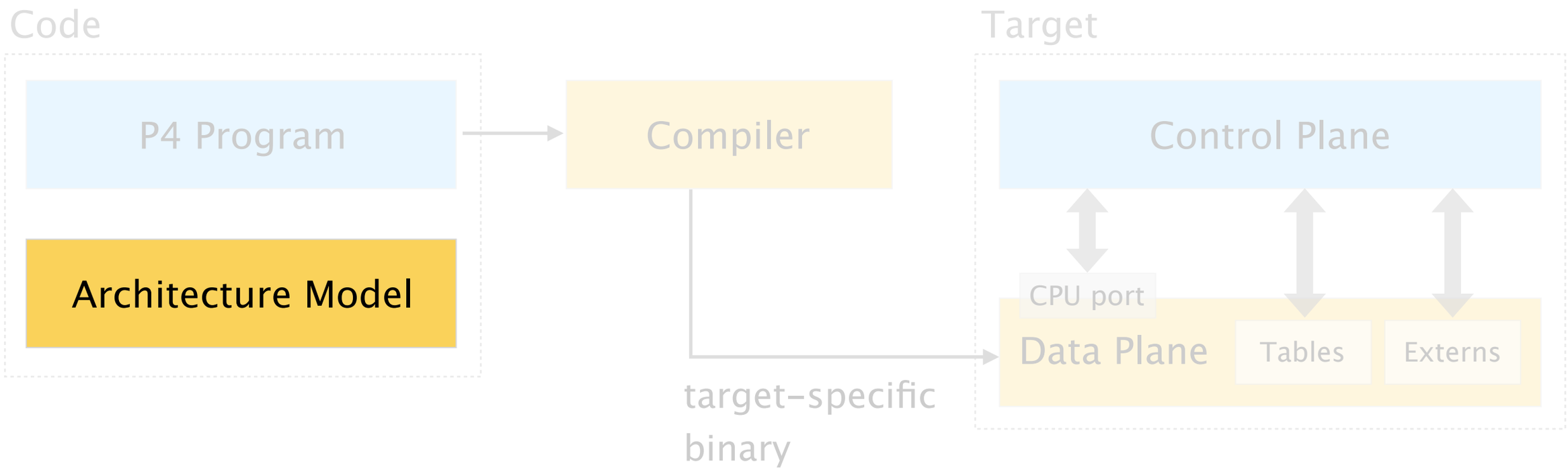
an API to program a target

# Programming a P4 target involves a few key elements




User supplied

Vendor supplied

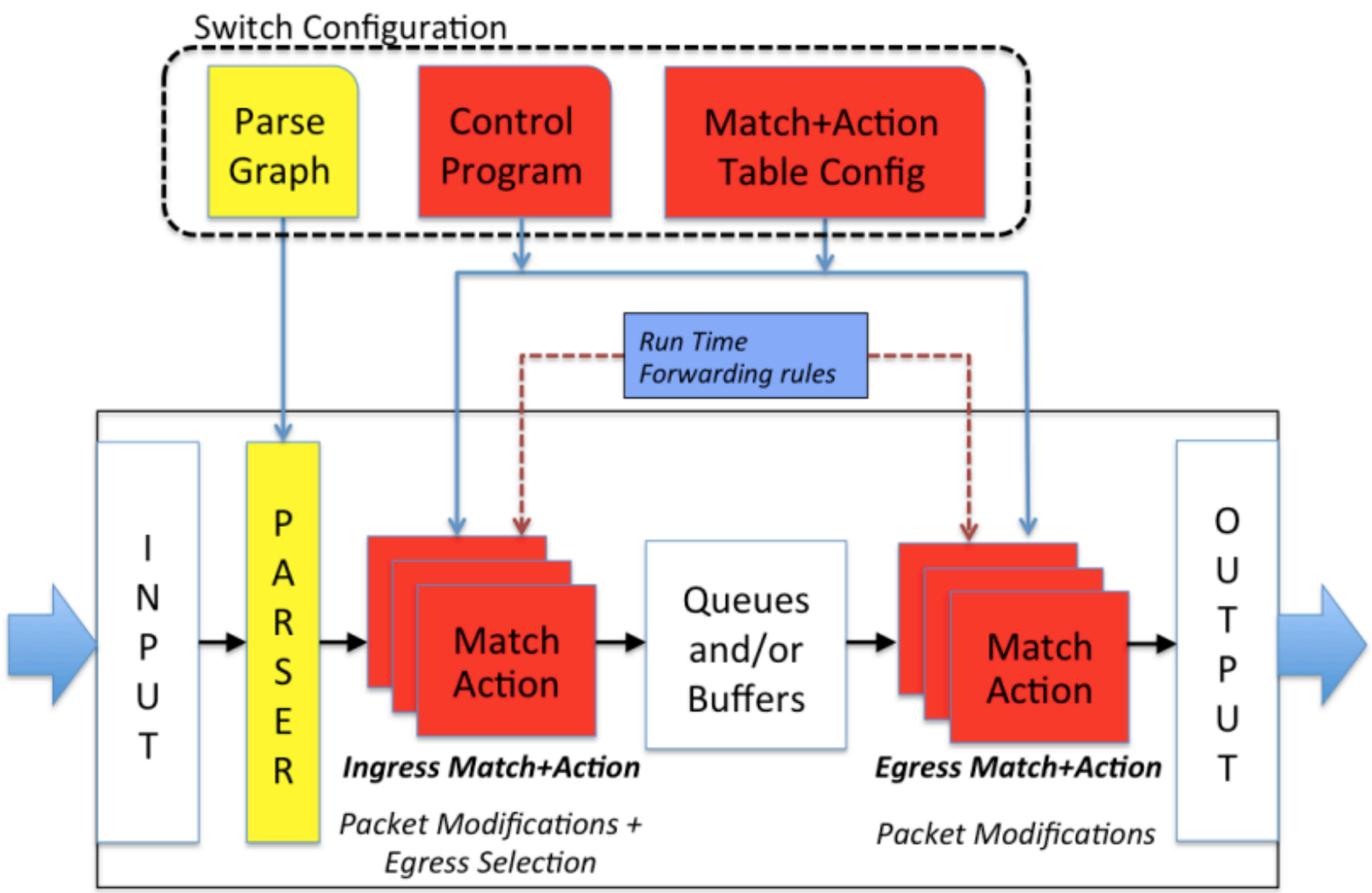



User supplied

Vendor supplied

# We'll rely on a simple P4$_{16}$ switch architecture (v1model) which is roughly equivalent to "PISA"

v1model/
simple switch




source https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

Each architecture defines the metadata it supports, including both standard and intrinsic ones

v1model

```
struct standard_metadata_t {
    bit<9>  ingress_port;
    bit<9>  egress_spec;
    bit<9>  egress_port;
    bit<32> clone_spec;
    bit<32> instance_type;
    bit<1>  drop;
    bit<16> recirculate_port;
    bit<32> packet_length;
    bit<32> enq_timestamp;
    bit<19> enq_qdepth;
    bit<32> deq_timedelta;
    bit<19> deq_qdepth;
    error parser_error;
    bit<48> ingress_global_timestamp;
    bit<48> egress_global_timestamp;
    bit<32> lf_field_list;
    bit<16> mcast_grp;
    bit<32> resubmit_flag;
    bit<16> egress_rid;
    bit<1>  checksum_error;
    bit<32> recirculate_flag;
}
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Each architecture also defines a list of "externs",
i.e. blackbox functions whose interface is known

Most targets contain specialized components

which cannot be expressed in P4 (e.g. complex computations)

At the same time, $P4_{16}$ should be target-independent

In $P4_{14}$ almost 1/3 of the constructs were target-dependent

Think of externs as Java interfaces

only the signature is known, not the implementation

v1model

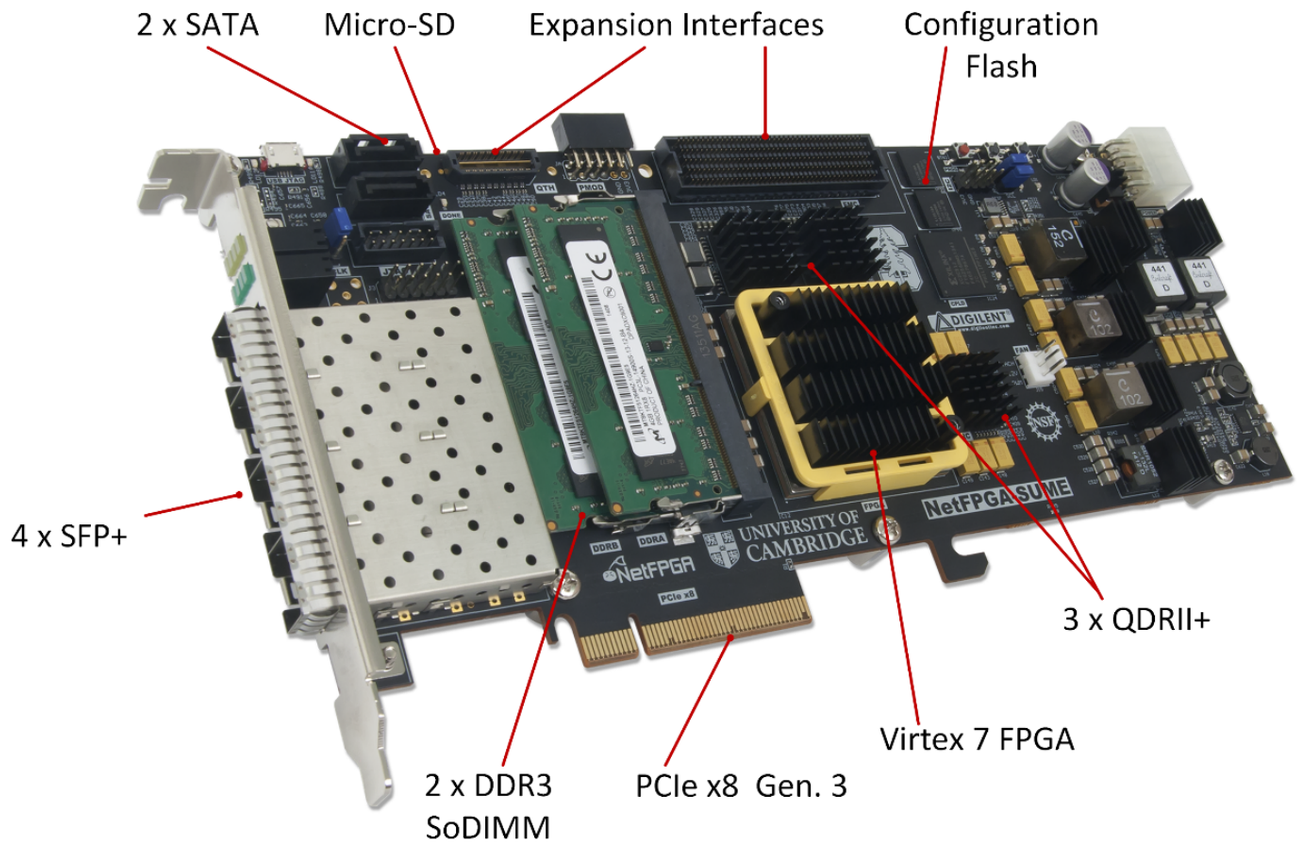
```
extern register<T> {
    register(bit<32> size);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}

extern void random<T>(out T result, in T lo, in T hi);

extern void hash<O, T, D, M>(out O result,
    in HashAlgorithm algo, in T base, in D data, in M max);

extern void update_checksum<T, O>(in bool condition,
    in T data, inout O checksum, HashAlgorithm algo);
```

+ many others (see below)

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

≠ architectures → ≠ metadata & ≠ externs

## NetFPGA-SUME








more info http://isfpga.org/fpga2018/slides/FPGA-2018-P4-tutorial.pdf

# Standard Metadata in SimpleSumeSwitch Architecture

```
/* standard sume switch metadata */
struct sume_metadata_t {
    bit<16> dma_q_size;
    bit<16> nf3_q_size;
    bit<16> nf2_q_size;
    bit<16> nf1_q_size;
    bit<16> nf0_q_size;
    bit<8> send_dig_to_cpu; // send digest_data to CPU
    bit<8> dst_port; // one-hot encoded
    bit<8> src_port; // one-hot encoded
    bit<16> pkt_len; // unsigned int
}
```

- `*_q_size` – size of each output queue, measured in terms of 32-byte words, when packet starts being processed by the P4 program
- `src_port`/`dst_port` – one-hot encoded, easy to do multicast
- `user_metadata`/`digest_data` – structs defined by the user





more info http://isfpga.org/fpga2018/slides/FPGA-2018-P4-tutorial.pdf

P4 environment

P4 language

label switching

Deeper dive into
the language constructs (*)

(*) full info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Recap

```
#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()”

But first, the basics:

data types, operations, and statements

P4$_{16}$ is a statically-typed language with

base types and operators to derive composed ones

| | |
|---|---|
| `bool` | Boolean value |
| `bit<W>` | Bit-string of width W |
| `int<W>` | Signed integer of width W |
| `varbit<W>` | Bit-string of dynamic length ≤W |
| `match_kind` | describes ways to match table keys |
| `error` | used to signal errors |
| `void` | no values, used in few restricted circumstances |
| `float` | not supported |
| `string` | not supported |

P4$_{16}$ is a statically-typed language with base types and operators to derive composed ones

Header

```
header Ethernet_h {
 bit<48> dstAddr;
 bit<48> srcAddr;
 bit<16> etherType;
}
```

```
Ethernet_h
  ethernetHeader;
```

corresponding declaration

Think of a header as a struct in C containing the different fields plus a hidden "validity" field

```
header Ethernet_h {
 bit<48> dstAddr;
 bit<48> srcAddr;
 bit<16> etherType;
}
```

Parsing a packet using `extract()` fills in the fields of the header from a network packet

A successful `extract()` sets to true the validity bit of the extracted header

P4$_{16}$ is a statically-typed language with base types and operators to derive composed ones

Header

```
header Ethernet_h {
 bit<48> dstAddr;
 bit<48> srcAddr;
 bit<16> etherType;
}
```

Header stack

```
header Mpls_h {
 bit<20> label;
 bit<3>  tc;
 bit     bos;
 bit<8>  ttl;
}

Mpls_h[10] mpls;
```

Array of up to 10 MPLS headers

Header union

```
header_union IP_h {
 IPv4_h v4;
 IPv6_h v6;
}
```

Either IPv4 or IPv6 header is present

only one alternative

P4$_{16}$ is a statically-typed language with
base types and operators to derive composed ones

Struct

Unordered collection
of named members

```
struct standard_metadata_t {
  bit<9> ingress_port;
  bit<9> egress_spec;
  bit<9> egress_port;
  …
}
```

Tuple

Unordered collection
of unnamed members

```
tuple<bit<32>, bool> x;
x = { 10, false };
```

P4$_{16}$ is a statically-typed language with
base types and operators to derive composed ones

- enum `enum Priority {High, Low}`
- type specialization `typedef bit<48> macAddr_t;`
- extern …
- parser …
- control …
- package …

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

P4 operations are similar to C operations and vary depending on the types (unsigned/signed ints, …)

- arithmetic operations `+, -, *`
- logical operations `~, &, |, ^, >>, <<`
- non-standard operations `[m:l]` Bit-slicing

  `++` Bit concatenation

✘ *no* division and modulo (can be approximated)

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

# Constants, variable declarations and instantiations are pretty much the same as in C too

Variable

```
bit<8> x = 123;
```

```
typedef bit<8> MyType;
MyType x;
x = 123;
```

Constant

```
const bit<8> X = 123;
```

```
typedef bit<8> MyType;
const MyType x = 123;
```

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Variables have local scope and their values is not maintained across subsequent invocations

important

variables *cannot* be used to maintain state between different network packets

instead

to maintain state

you can only use two stateful constructs

- tables — modified by control plane
- extern objects — modified by control plane & data plane

more on this next week

# P4 statements are pretty classical too

## Restrictions apply depending on the statement location

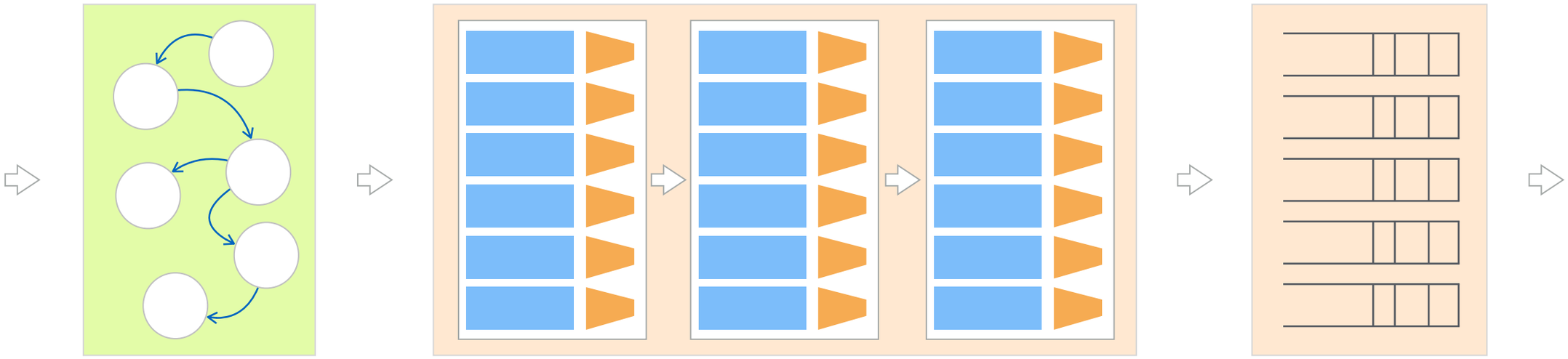
| | | |
|---|---|---|
| `return` | terminates the execution of the action or control containing it | |
| `exit` | terminates the execution of all the blocks currently executing | |
| Conditions | `if (x==123) {…} else {…}` | not in parsers |
| Switch | `switch (t.apply().action_run) { action1: { ...} action2: { ...} }` | only in control blocks |

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Parser

Match-Action Pipeline

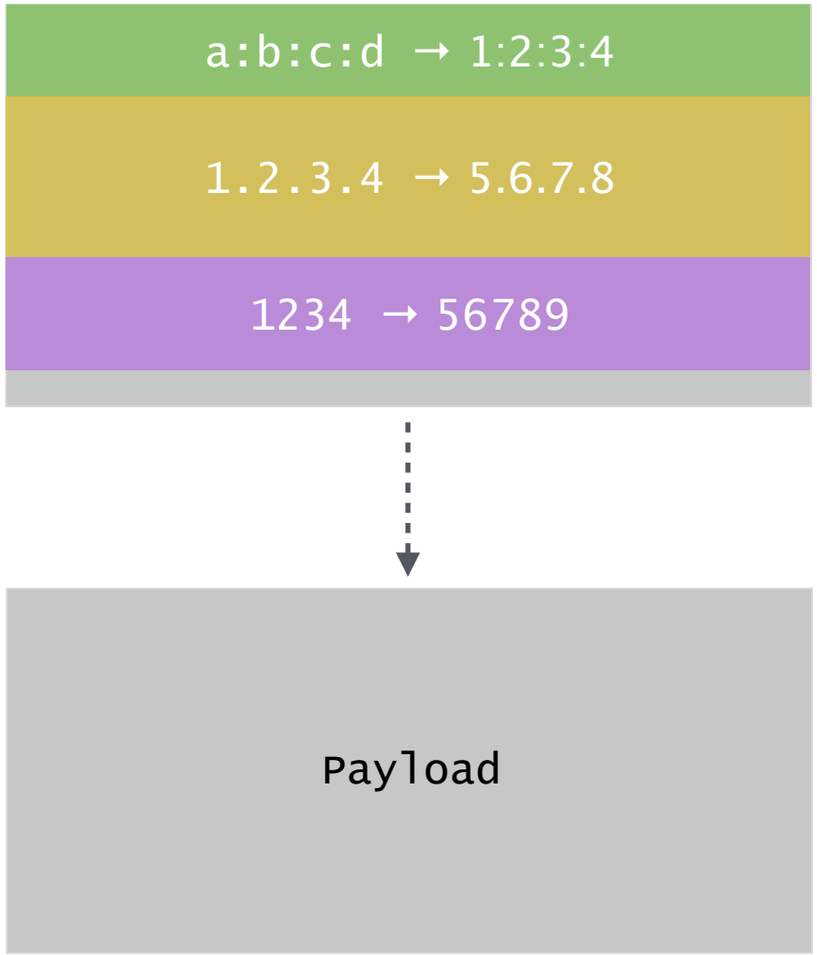
Deparser

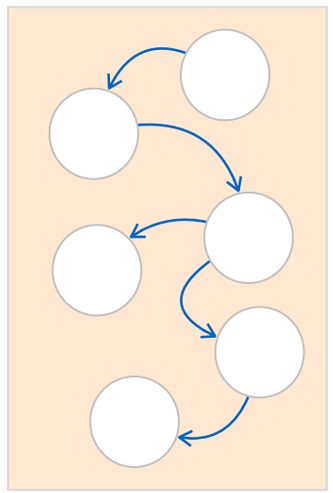


Recap

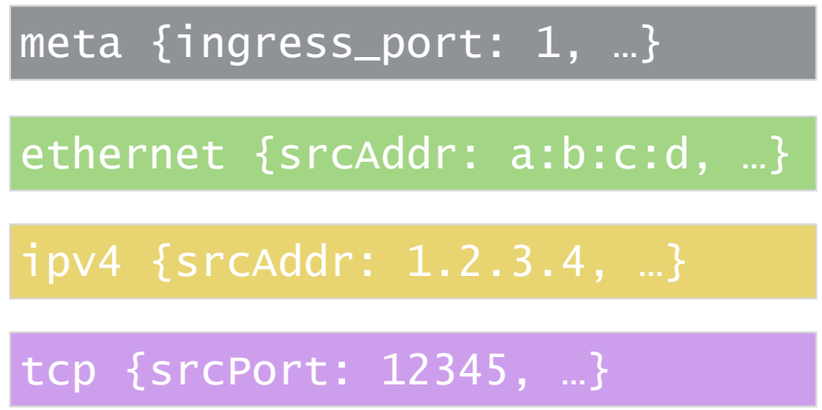
# The parser uses a state machine to map packets into headers and metadata

Packet

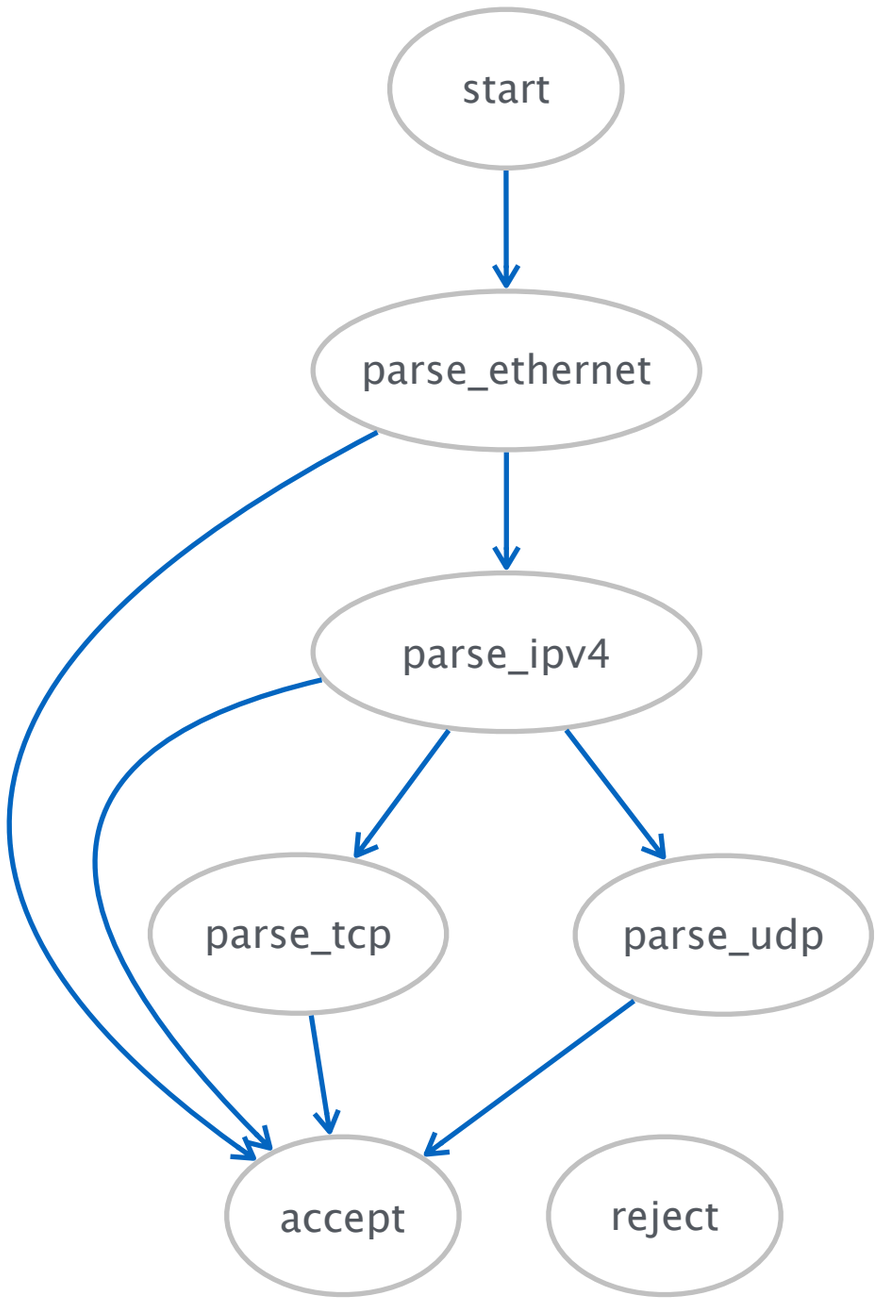


Headers and metadata



Recap




```
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;
 }

}
```

The last statement in a state is an (optional) transition, which transfers control to another state (inc. accept/reject)

```
state start {
 transition parse_ethernet;
}
```

Go directly to parse_ethernet

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

Next state depends on etherType

Defining (and parsing) custom headers allow you to implement your own protocols

# P4 parser supports both fixed and variable-width header extraction

```
header IPv4_no_options_h {
 ...
 bit<32>  srcAddr;
 bit<32>  dstAddr;
}

header IPv4_options_h {
 varbit<320> options;
}

...
parser MyParser(…) {
 ...
 state parse_ipv4 {
  packet.extract(headers.ipv4);
  transition select (headers.ipv4.ihl) {
   5: dispatch_on_protocol;
   default: parse_ipv4_options;
  }

 state parse_ipv4_options {
  packet.extract(headers.ipv4options,
       (headers.ipv4.ihl - 5) << 2);
  transition dispatch_on_protocol;
 }
}
```

Fixed width fields

Variable width field

ihl determines length of options field

# Parsing a header stack requires the parser to loop

the only “loops” that are possible in P4

# The parser contains more advanced concepts

## check them out!

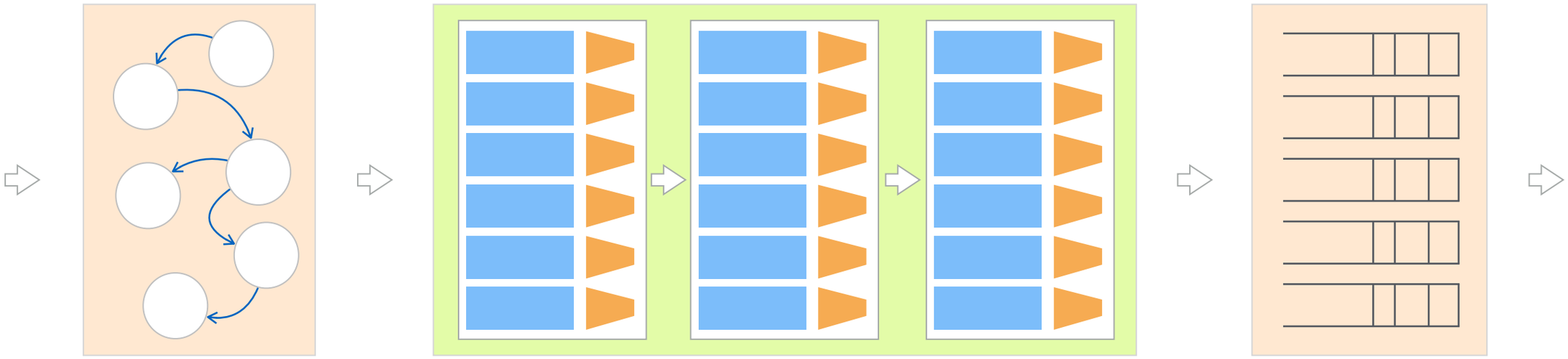
- verify — error handling in the parser
- lookahead — access bits that are not parsed yet
- sub-parsers — like subroutines

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Parser

Match-Action Pipeline

Deparser



Recap

Control

Tables

match a key and return an action

Actions

similar to functions in C

Control flow

similar to C but without loops

Control

Tables

match a key and return an action

Actions

similar to functions in C

Control flow

similar to C but without loops

# Tables can match on one or multiple keys in different ways

```
table Fwd {
    key = {
        hdr.ipv4.dstAddr : ternary;
        hdr.ipv4.version : exact;
    }
    ...
}
```

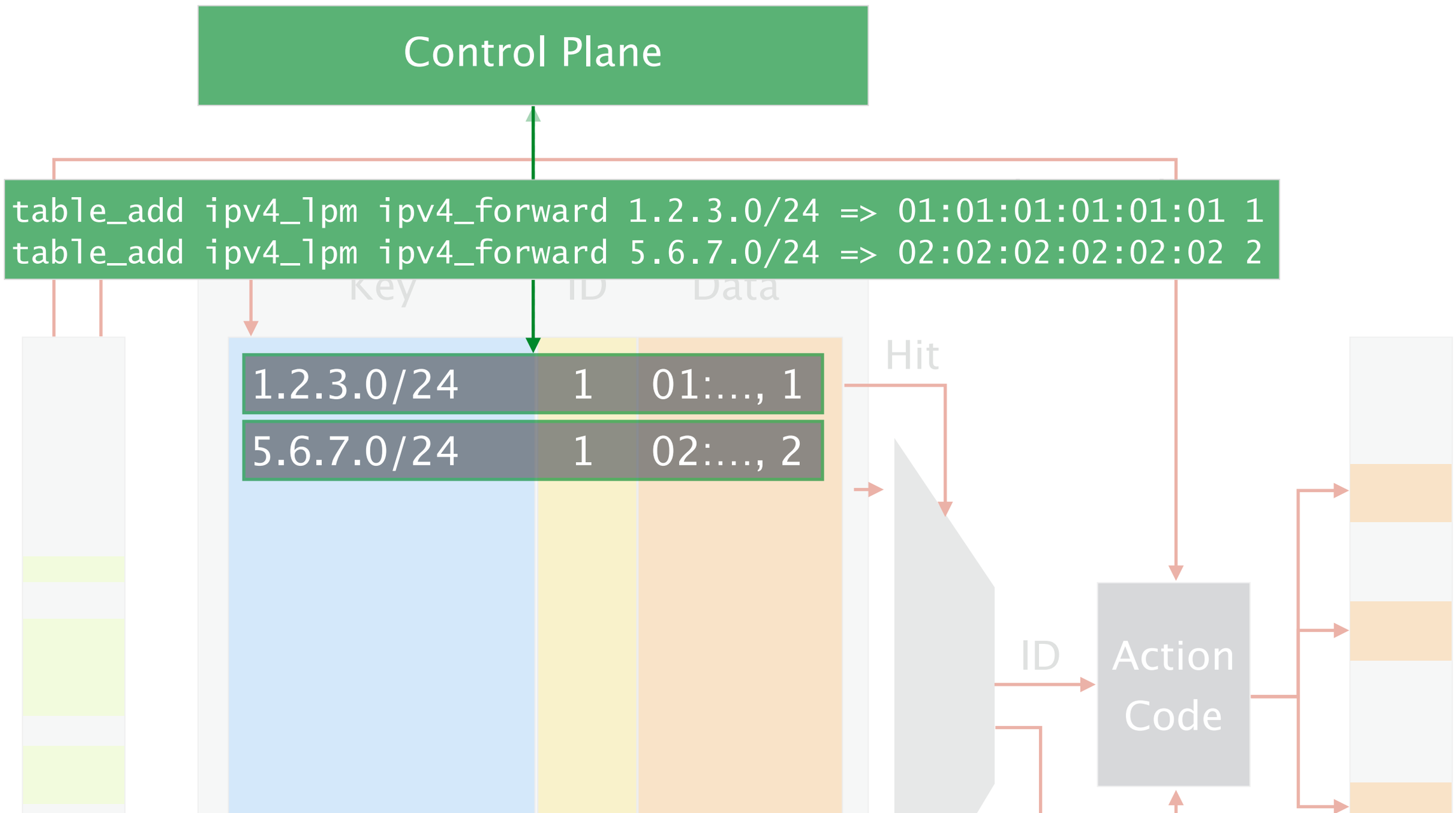
Fields to match

Match kind

# Match types are specified in the P4 core library and in the architectures

| | | |
|---|---|---|
| `exact` | exact comparison<br>`0x01020304` | core.p4 |
| `ternary` | compare with mask<br>`0x01020304 & 0x0F0F0F0F` | |
| `lpm` | longest prefix match<br>`0x01020304/24` | |
| `range` | check if in range<br>`0x01020304 - 0x010203FF` | v1model.p4 |
| `...` | | |
| `...` | | other architecture |

# Table entries are added through the control plane



Control

Tables

match a key and return an action

Actions

similar to functions in C

Control flow

similar to C but without loops

Actions are blocks of statements that possibly modify the packets

Actions usually take directional parameters indicating how the corresponding value is treated within the block

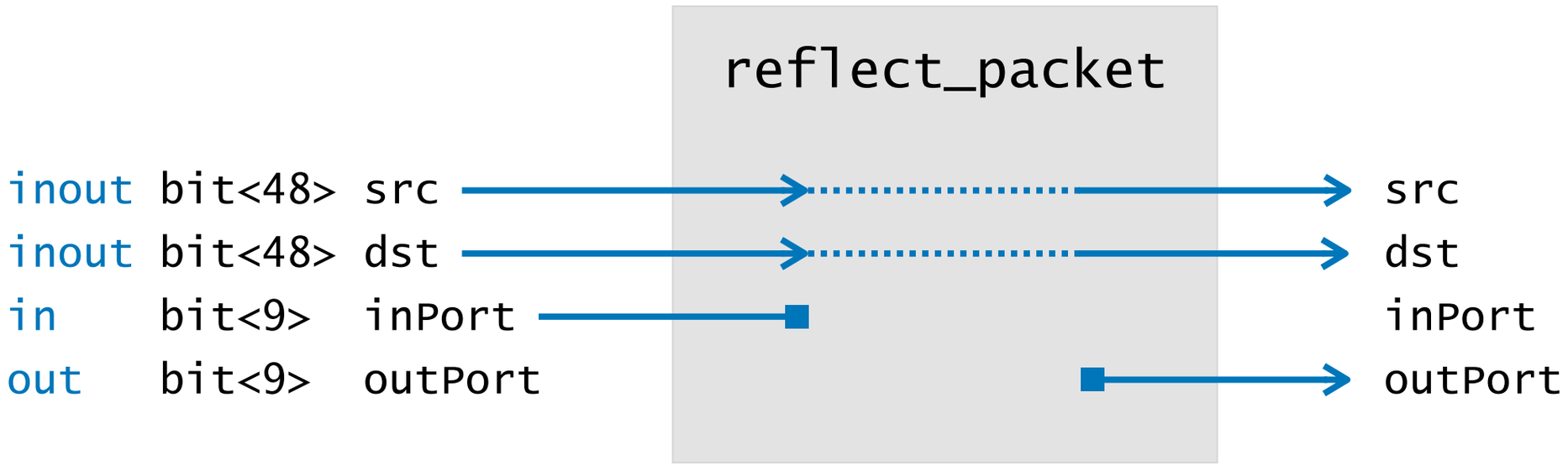
# Directions can be of three types

| | |
|---|---|
| `in` | read only inside the action<br>like parameters to a function |
| `out` | uninitialized, write inside the action<br>like return values |
| `inout` | combination of `in` and `out`<br>like “call by reference” |

# Let's reconsider a known example

```
action reflect_packet( inout bit<48> src,
                       inout bit<48> dst,
                       in bit<9> inPort,
                       out bit<9> outPort
                     ) {
 bit<48> tmp = src;
 src = dst;
 dst = tmp;
 outPort = inPort;
}


reflect_packet( hdr.ethernet.srcAddr,
                hdr.ethernet.dstAddr,
                standard_metadata.ingress_port,
                standard_metadata.egress_spec
              );
```

Parameter
with direction


reflect_packet
inout bit<48> src
inout bit<48> dst
in bit<9> inPort
out bit<9> outPort
src
dst
inPort
outPort

Actions parameters resulting from a table lookup do not take a direction as they come from the control plane

Parameter
without direction

```
action set_egress_port(bit<9> port) {
 standard_metadata.egress_spec = port;
}
```

Control

Tables

match a key and return an action

Actions

similar to functions in C

Control flow

similar to C but without loops

# Interacting with tables from the control flow

- Applying a table

```
ipv4_lpm.apply()
```

- Checking if there was a hit

```
if (ipv4_lpm.apply().hit) {...}
else {...}
```

- Check which action was executed

```
switch (ipv4_lpm.apply().action_run) {
    ipv4_forward: { ... }
}
```

# Validating and computing checksums

```
extern void verify_checksum<T, O>( in bool condition,
                                   in T data,
                                   inout O checksum,
                                   HashAlgorithm algo
                                 );


extern void update_checksum<T, O>( in bool condition,
                                   in T data,
                                   inout O checksum,
                                   HashAlgorithm algo
                                 );
```

v1model.p4

# Re-computing checksums

## (e.g. after modifying the IP header)

```
control MyComputeChecksum(...) {
 apply {
  update_checksum(
   hdr.ipv4.isValid(),
   { hdr.ipv4.version,
     hdr.ipv4.ihl,
     hdr.ipv4.diffserv,
     hdr.ipv4.totalLen,
     hdr.ipv4.identification,
     hdr.ipv4.flags,
     hdr.ipv4.fragOffset,
     hdr.ipv4.ttl,
     hdr.ipv4.protocol,
     hdr.ipv4.srcAddr,
     hdr.ipv4.dstAddr },
     hdr.ipv4.hdrChecksum,
   HashAlgorithm.csum16);
 }
}
```

pre-condition

fields list

checksum field

algorithm

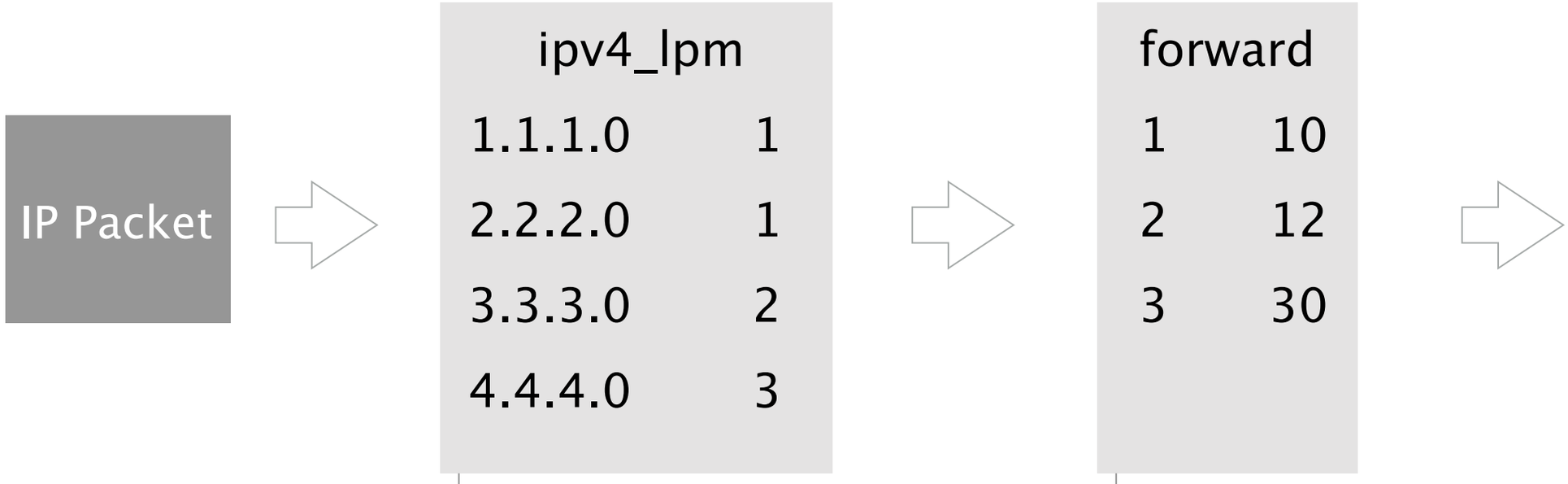
# Control flows contain more advanced concepts

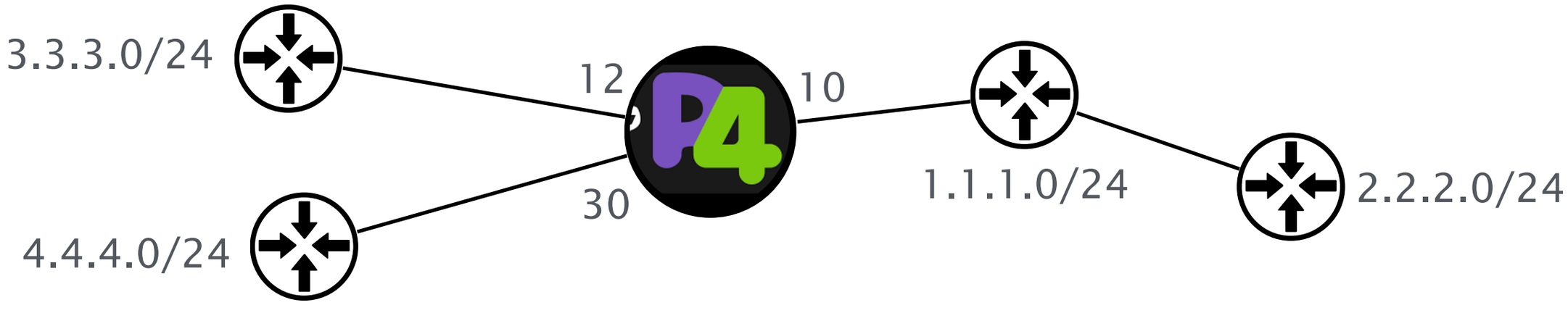
check them out!

- cloning packets — create a clone of a packet
- sending packets to control plane — using dedicated Ethernet port, or target-specific mechanisms (e.g. digests)
- recirculating — send packet through pipeline multiple times

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

# Example: L3 forwarding with multiple tables



# Example 1: L3 forwarding with multiple tables

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

```
table forward {
 key = {
  meta.nhop_index: exact;
 }
 actions = {
  _forward;
  NoAction;
 }

 size = 64;
 default_action = NoAction();
}
```

# Applying multiple tables in sequence

## and checking whether there was a hit

```
control MyIngress(...) {
 action drop() {...}
 action set_nhop_index(...}
 action _forward(...}
 table ipv4_lpm {...}
 table forward {...}
 apply {
  if (hdr.ipv4.isValid()){
   if (ipv4_lpm.apply().hit) {
    forward.apply();
   }
  }
 }
}
```

Check if IPv4 packet

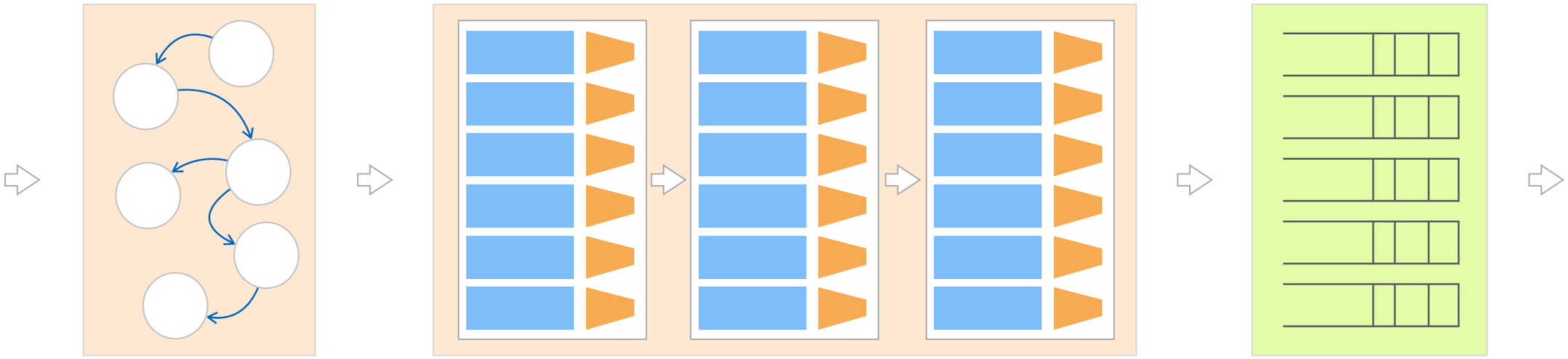
Apply `ipv4_lpm` table and check if there was a hit

apply `forward` table

Parser

Match-Action Pipeline

Deparser

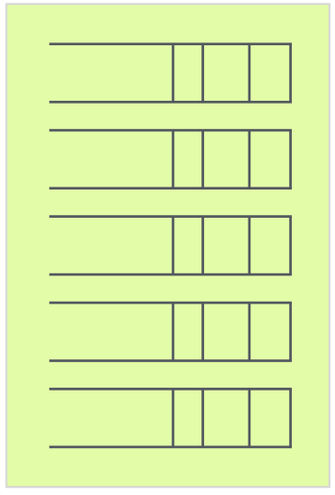


Recap

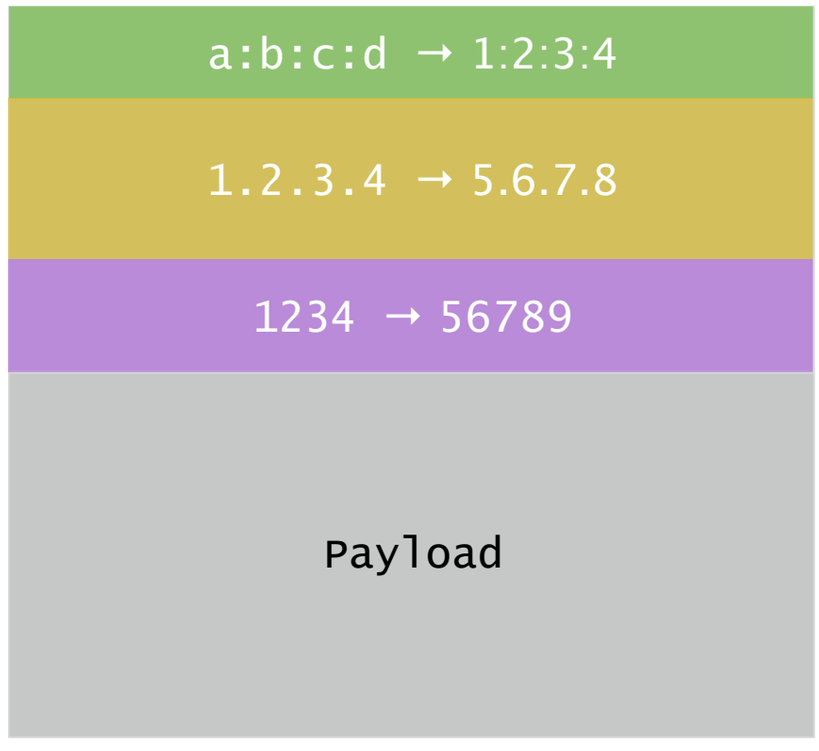
Headers

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

Deparser



Packet




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

P4 environment

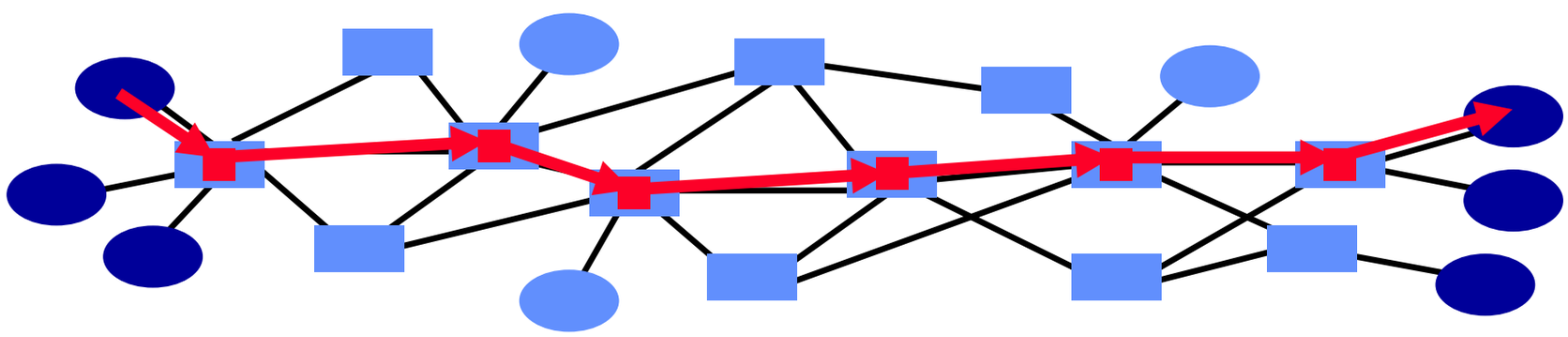
P4 language

label switching

the basics

# Packet Switching

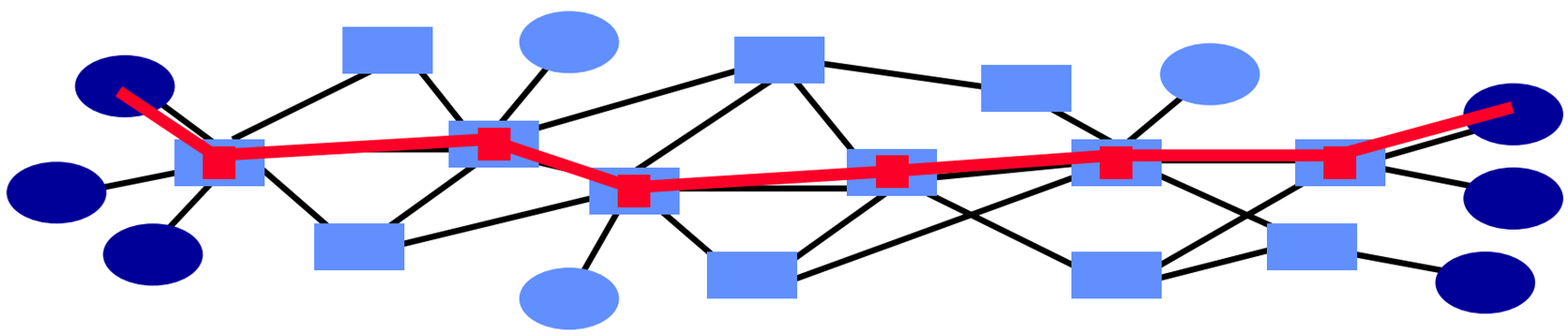
- Source sends information as self-contained packets that have an address.
  - » Source may have to break up single message in multiple
- Each packet travels independently to the destination host.
  - » Routers and switches use the address in the packet to determine how to forward the packets
- Destination recreates the message.
- Analogy: a letter in surface mail.



source: David Andersen, CMU

# Circuit Switching

- **Source first establishes a connection (circuit) to the destination.**
  - » **Each router or switch along the way may reserve some bandwidth for the data flow**
- **Source sends the data over the circuit.**
  - » **No need to include the destination address with the data since the routers know the path**
- **The connection is torn down.**
- **Example: telephone network.**



source: David Andersen, CMU

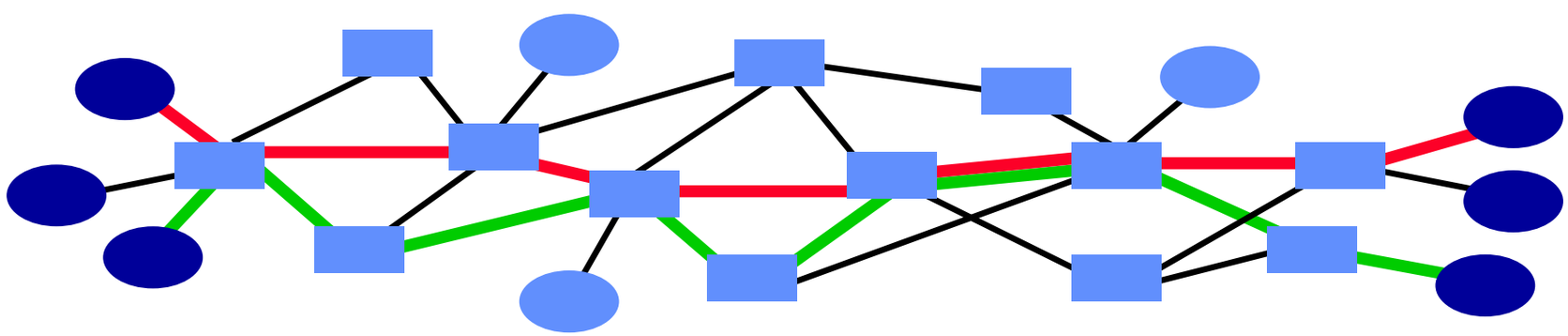
# Circuit Switching Discussion

- **Traditional circuits: on each hop, the circuit has a dedicated wire or slice of bandwidth.**
  - » **Physical connection - clearly no need to include addresses with the data**
- **Advantages, relative to packet switching:**
  - » **Implies guaranteed bandwidth, predictable performance**
  - » **Simple switch design: only remembers connection information, no longest-prefix destination address look up**
- **Disadvantages:**
  - » **Inefficient for bursty traffic (wastes bandwidth)**
  - » **Delay associated with establishing a circuit**
- **Can we get the advantages without (all) the disadvantages?**

source: David Andersen, CMU

# Virtual Circuits

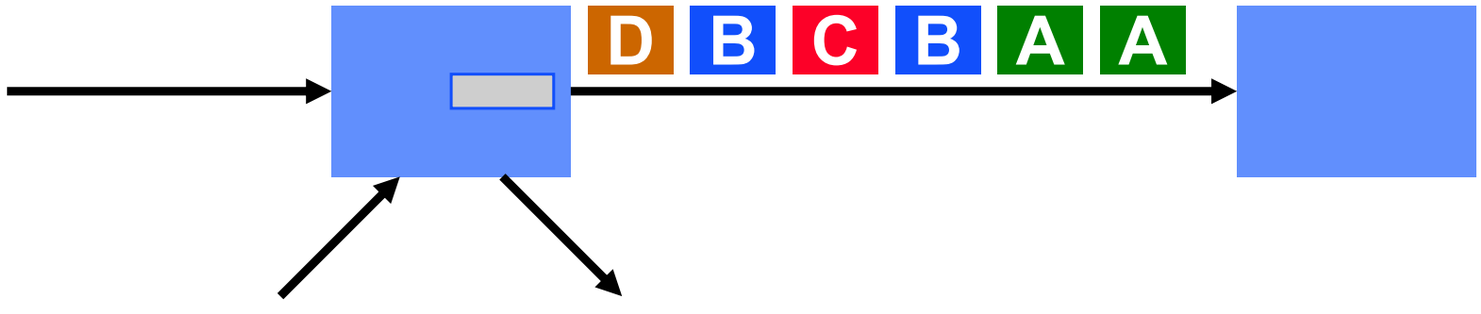
- **Each wire carries many "virtual" circuits.**
  - » **Forwarding based on virtual circuit (VC) identifier**
    - – **IP header: src, dst, etc.**
    - – **Virtual circuit header: just "VC"**
  - » **A path through the network is determined for each VC when the VC is established**
  - » **Use statistical multiplexing for efficiency**
- **Can support wide range of quality of service.**
  - » **No guarantees: best effort service**
  - » **Weak guarantees: delay < 300 msec, …**
  - » **Strong guarantees: e.g. equivalent of physical circuit**



source: David Andersen, CMU

# Packet Switching and Virtual Circuits: Similarities

- **"Store and forward" communication based on an address.**
  - » **Address is either the destination address or a VC identifier**
- **Must have buffer space to temporarily store packets.**
  - » **E.g. multiple packets for some destination arrive simultaneously**
- **Multiplexing on a link is similar to time sharing.**
  - » **No reservations: multiplexing is statistical, i.e. packets are interleaved without a fixed pattern**
  - » **Reservations: some flows are guaranteed to get a certain number of "slots"**




source: David Andersen, CMU

# Virtual Circuits Versus Packet Switching

- **Circuit switching:**
  - » Uses short connection identifiers to forward packets
  - » Switches know about the connections so they can more easily implement features such as quality of service
  - » Virtual circuits form basis for traffic engineering: VC identifies long-lived stream of data that can be scheduled
- **Packet switching:**
  - » Use full destination addresses for forwarding packets
  - » Can send data right away: no need to establish a connection first
  - » Switches are stateless: easier to recover from failures
  - » Adding QoS is hard
  - » Traffic engineering is hard: too many packets!

source: David Andersen, CMU

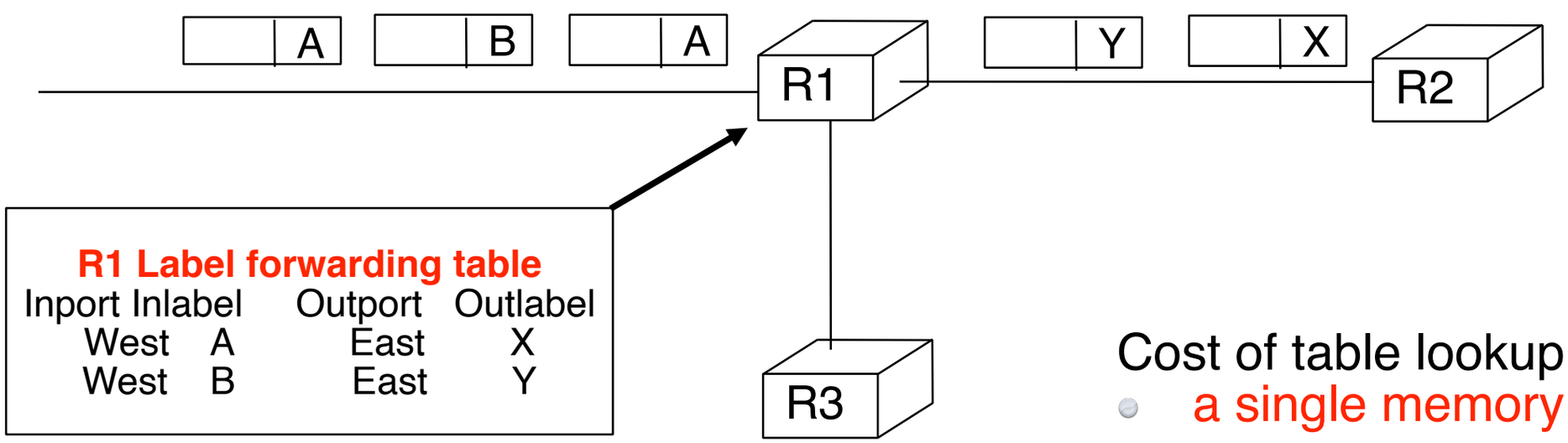
# Multiprotocol Label Switching

"IP meets virtual circuits"

| | |
|---|---|
| core idea | store the label in-between layer 2 and layer 3<br>MPLS is often referred to as a layer 2.5 protocol |
| usages | Traffic engineering, QoS, Fast reroute, VPNs, … |

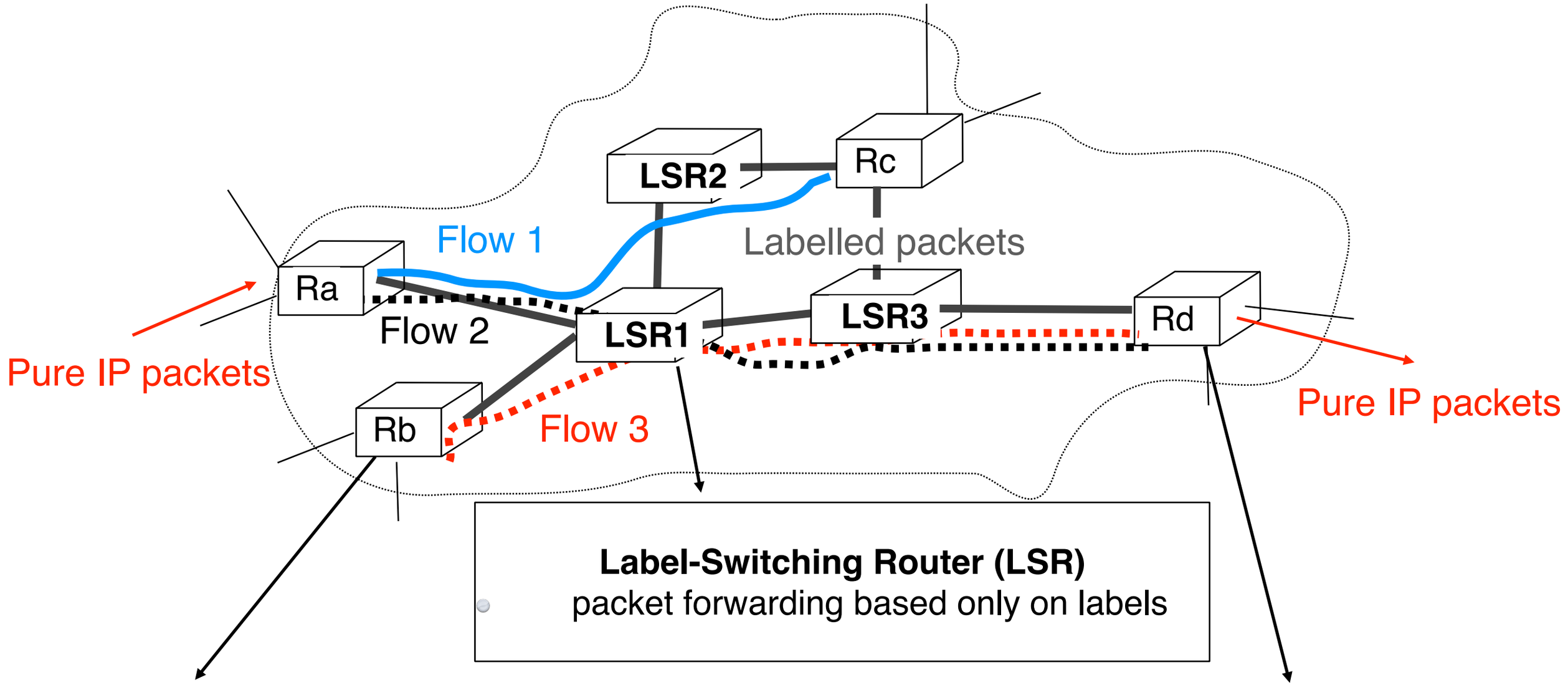
# Label swapping:
# A simple forwarding mechanism

- On packet arrival, the router analyses
  - Packet label
  - Input port
- Using its label forwarding table, the router then decides
  - Output port
  - Packet label




source (slides 80–116): Olivier Bonaventure, UCLouvain

# Integrating label swapping and IP




**Label-Switching Router (LSR)**
- packet forwarding based only on labels

**Label Edge Router (LER)**
- inserts labels on packets sent through backbone

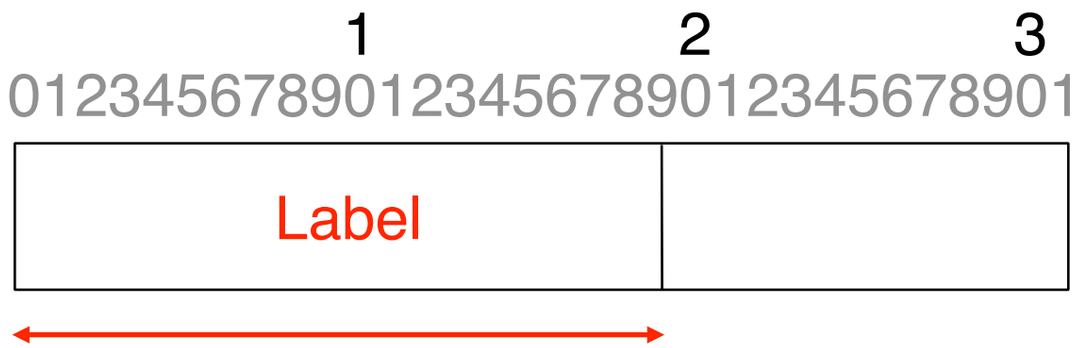
**Egress LSR**
- removes labels before sending packets outside MPLS network

# Integrating label swapping and IP (2)

- We need to solve three problems
  1. What do we use as packet label?
  2. What is the behaviour of a core LSR?
  3. What is the behaviour of an edge LSR?

# Labelled IP packets

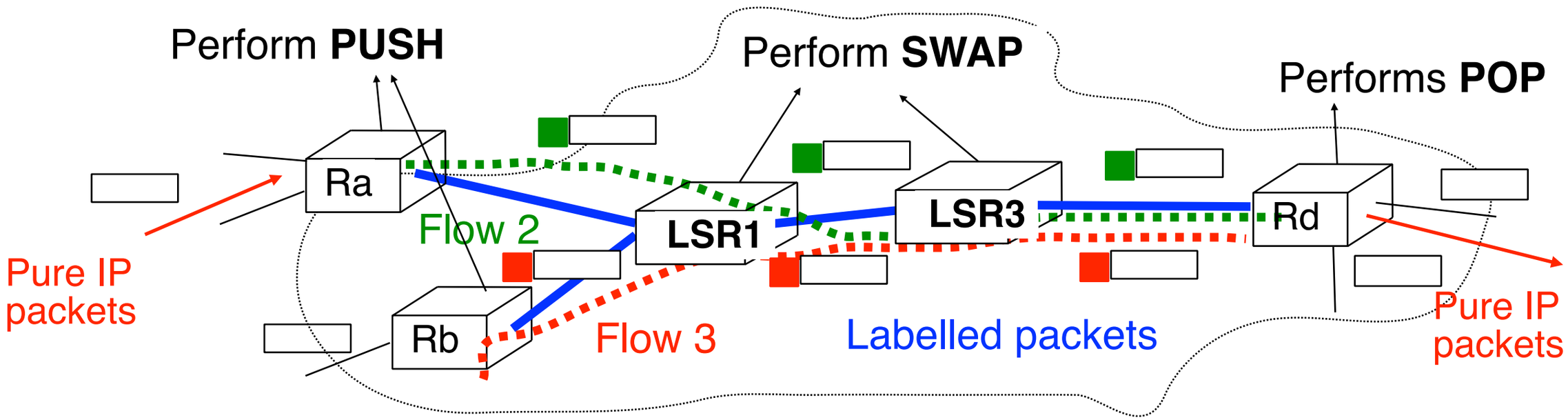
- Insert special 32 bits headers in front of the IP header



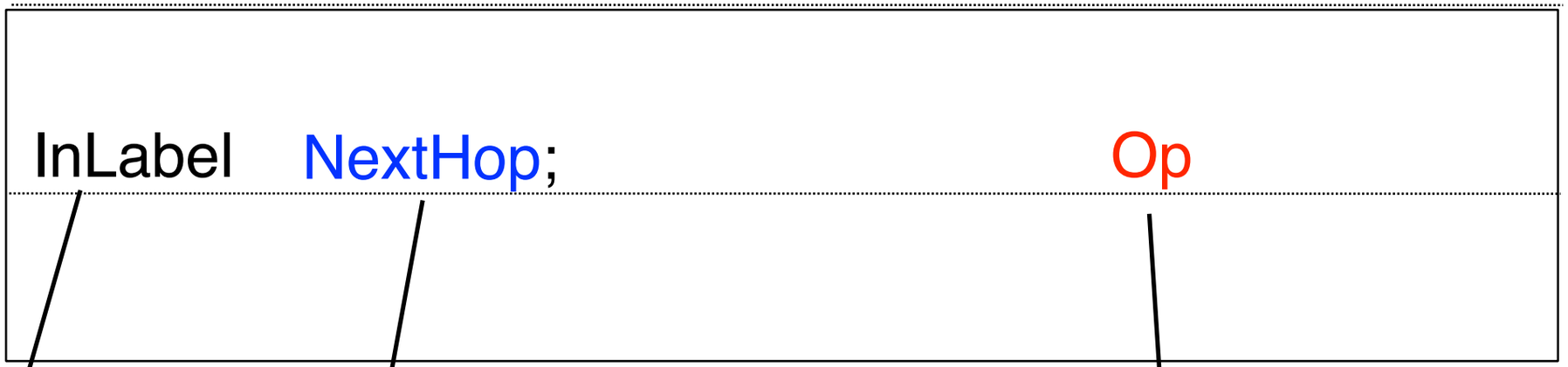

20 bits: 1048576 different labels

# MPLS data plane allows for three operations on a labelled packet

1. PUSH
   - insert a label in front of a received packet
2. SWAP
   - change the value of the label of a received labelled packet
3. POP
   - remove the label in front of a received labelled packet



# Content of the Label forwarding table



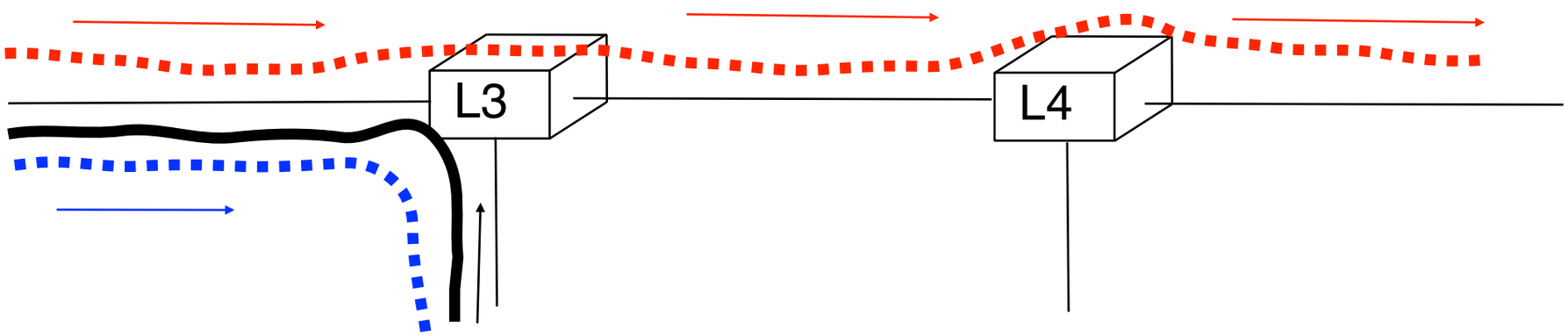

- Label of the incoming packet
- Next-hop for the packet
  - outgoing interface
    - packet sent to another LSR
  - LSR itself
    - packet destination is LSR
    - packet needs to be routed as a normal IP packet after pop operation
- Operation to be performed on the label
  - swap label
  - push new label (on top)
  - pop label

# Label Switch Routers manage their labels on a per-interface or per-device basis

- A LSR may manage its labels with 2 methods
  - Per-interface label space
    - one label space for each interface
    - $2^{20}$ distinct labels for each interface of the LSR

**L3's per-interface Label forwarding table**

| Inlabel | Outport | Operation |
|---|---|---|
| West:L3 | South | SWAP(L0) |
| West:L4 | East | SWAP(L3) |
| South:L3 | West | SWAP(L5) |



# Label Switch Routers manage their labels on a per-interface or per-device basis

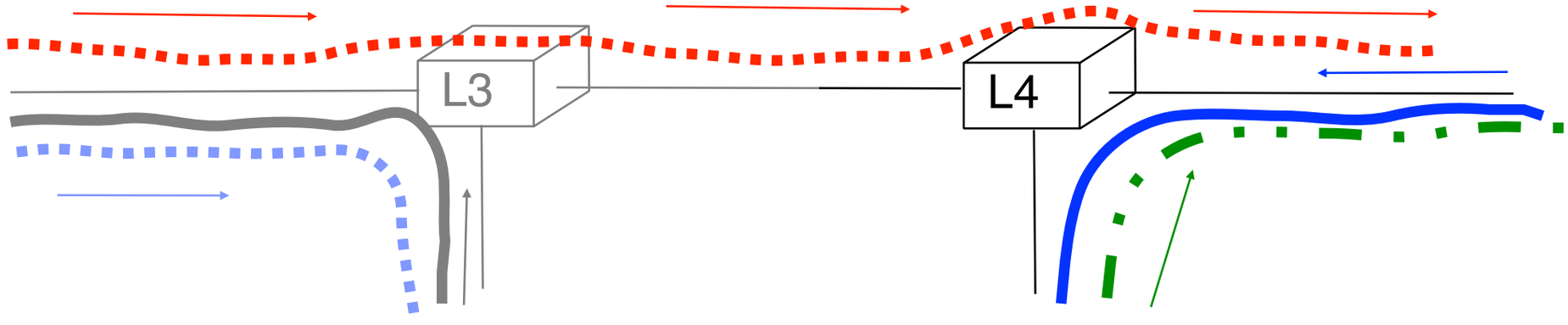
- A LSR may manage its labels with 2 methods
  - Per-interface label space
    - one label space for each interface
    - $2^{20}$ distinct labels for each interface of the LSR

**L3's per-interface Label forwarding table**

| Inlabel | Outport | Operation |
|---|---|---|
| West:L3 | South | SWAP(L0) |
| West:L4 | East | SWAP(L3) |
| South:L3 | West | SWAP(L5) |

**L4's per-LSR Label forwarding table**

| Inlabel | Outport | Operation |
|---|---|---|
| L0 | East | SWAP(L0) |
| L4 | South | SWAP(L3) |
| L3 | East | SWAP(L5) |

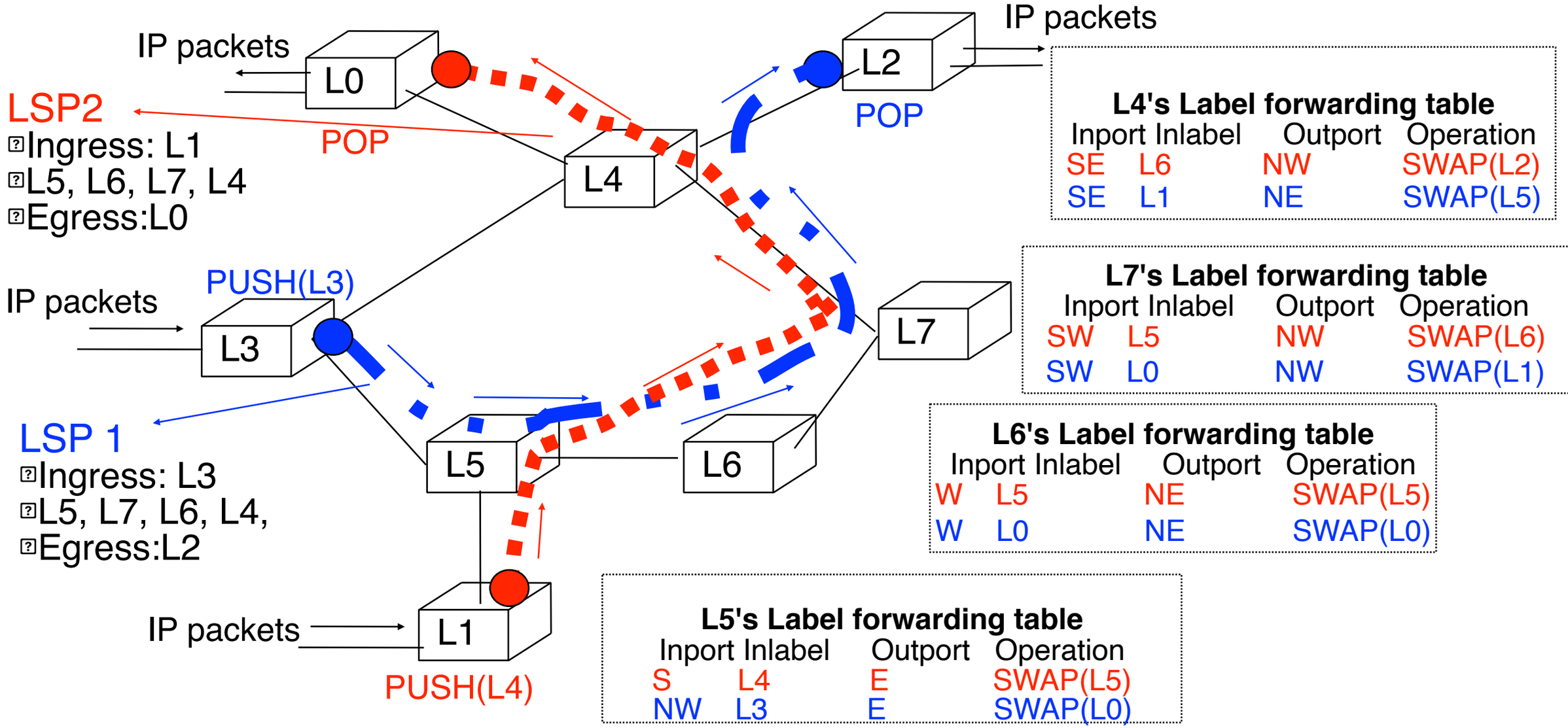



- Per-LSR label space
  - a single label space for all interfaces
  - $2^{20}$ distinct labels for the LSR

# A Label Switched Path (LSP) is a unidirectional tunnel between a pair of routers

- LSP
  - a path followed by a labelled packet over several hops starting at an LER and ending at an egress LSR
  - an LSP is required for any MPLS forwarding to occur

# LSP: Label switched path




**L4's Label forwarding table**

| Inport | Inlabel | Outport | Operation |
|---|---|---|---|
| SE | L6 | NW | SWAP(L2) |
| SE | L1 | NE | SWAP(L5) |

**L7's Label forwarding table**

| Inport | Inlabel | Outport | Operation |
|---|---|---|---|
| SW | L5 | NW | SWAP(L6) |
| SW | L0 | NW | SWAP(L1) |

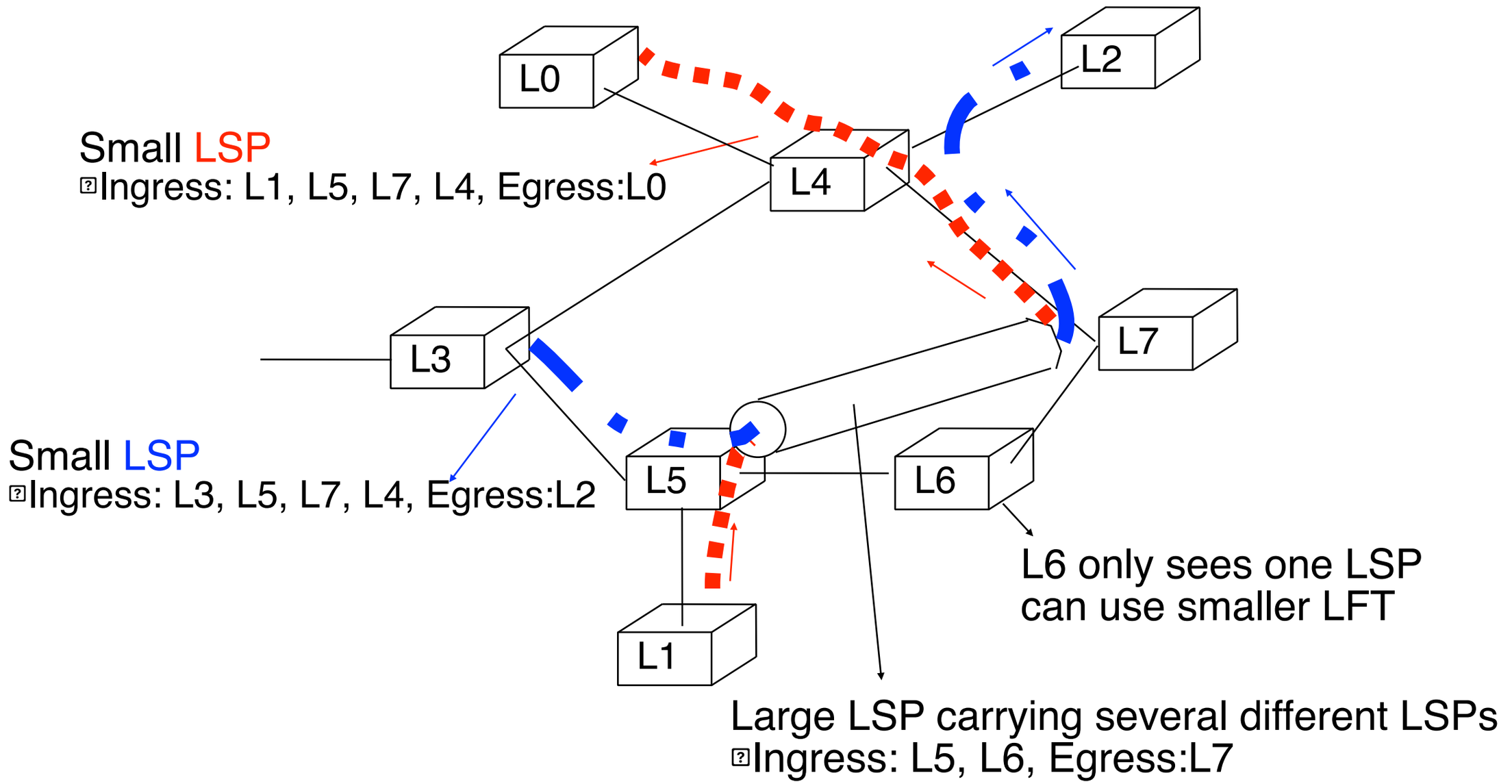
**L6's Label forwarding table**

| Inport | Inlabel | Outport | Operation |
|---|---|---|---|
| W | L5 | NE | SWAP(L5) |
| W | L0 | NE | SWAP(L0) |

**L5's Label forwarding table**

| Inport | Inlabel | Outport | Operation |
|---|---|---|---|
| S | L4 | E | SWAP(L5) |
| NW | L3 | E | SWAP(L0) |

# Label stacking helps to scale by introducing a LSP hierarchy

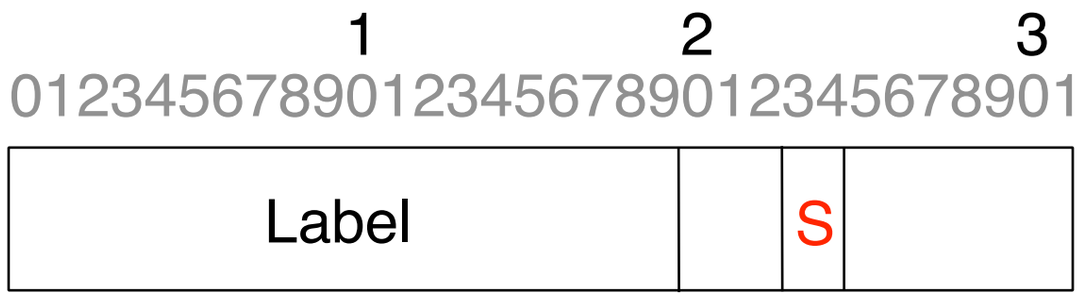



- Scalability of LSPs
  - it should be possible to place many small LSPs inside on large LSPs

# Label stacking helps to scale by introducing a LSP hierarchy

- How to support hierarchy of LSPs ?
  - it should be possible to place small LSPs inside large LSPs
  - ideally, there should be no predefined limit on the number of levels supported

- Solution adopted by MPLS
  - each labelled packet can carry a stack of labels

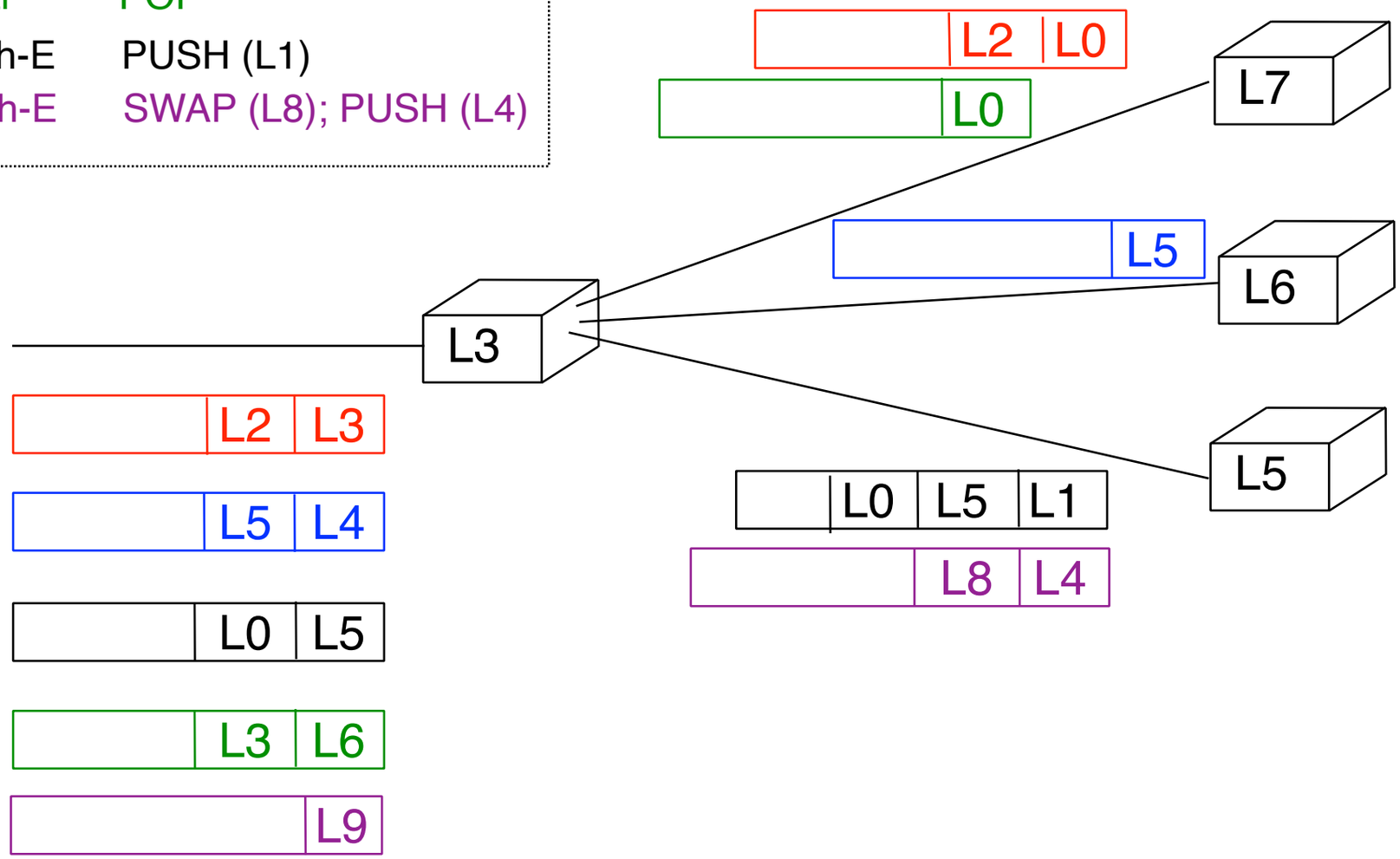



- label at the top of the stack appears first in packet
  - S=1 if the label is at the bottom of the stack
  - S=0 if the label is not at the bottom of the stack

# Content of the Label forwarding table (more) (2)

**L3's Label forwarding table**

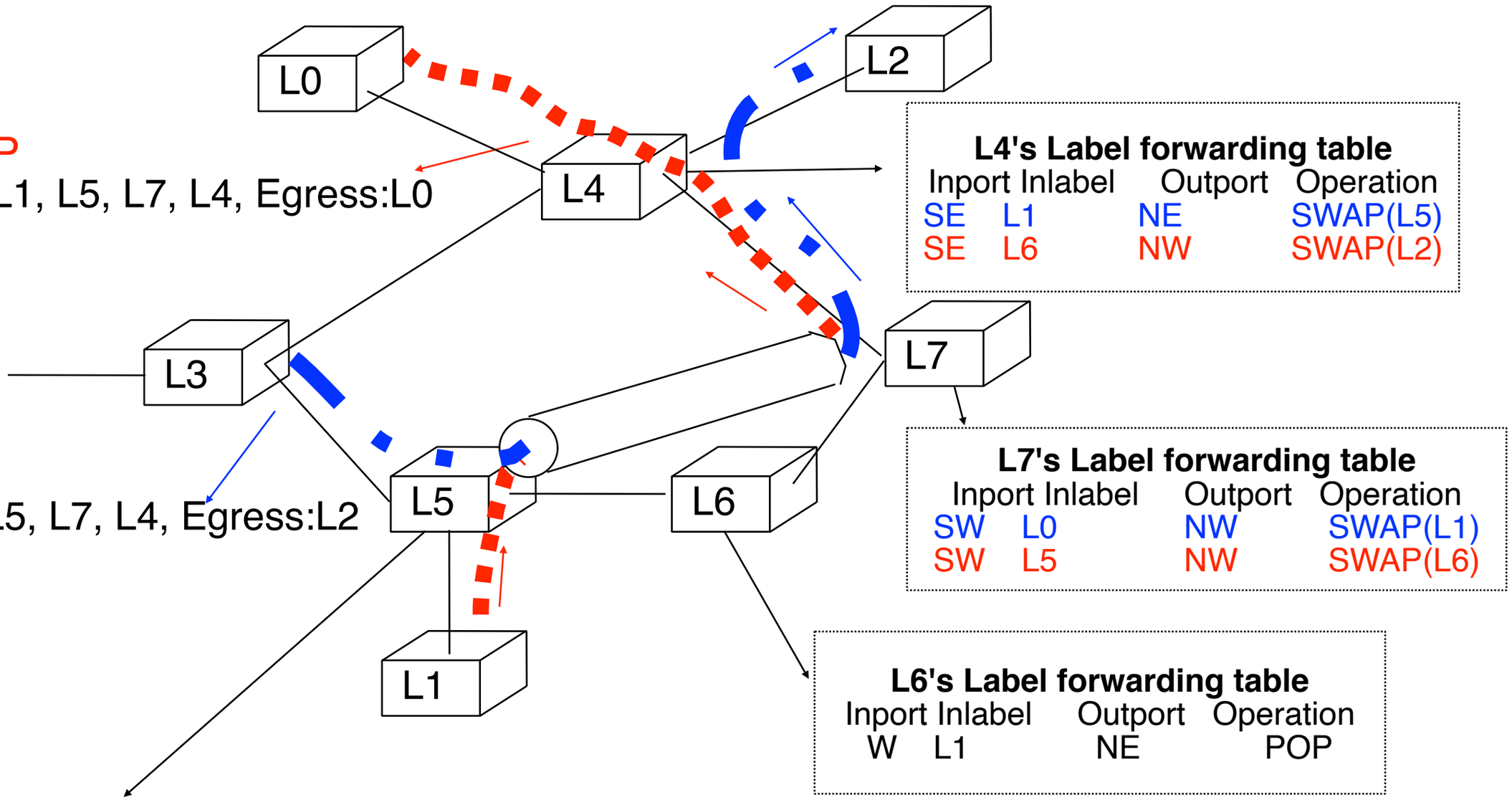
| Inlabel | Nexthop | Operation |
|---|---|---|
| L3 | North-E | SWAP(L0) |
| L4 | East | POP |
| L6 | Local | POP |
| L5 | South-E | PUSH (L1) |
| L9 | South-E | SWAP (L8); PUSH (L4) |



# MPLS and label stacks

Small LSP
- Ingress: L1, L5, L7, L4, Egress:L0

Small LSP
- Ingress: L3, L5, L7, L4, Egress:L2




**L4's Label forwarding table**

| Inport | Inlabel | Outport | Operation |
|---|---|---|---|
| SE | L1 | NE | SWAP(L5) |
| SE | L6 | NW | SWAP(L2) |

**L7's Label forwarding table**

| Inport | Inlabel | Outport | Operation |
|---|---|---|---|
| SW | L0 | NW | SWAP(L1) |
| SW | L5 | NW | SWAP(L6) |

**L6's Label forwarding table**

| Inport | Inlabel | Outport | Operation |
|---|---|---|---|
| W | L1 | NE | POP |

**L5's Label forwarding table**

| Inport | Inlabel | Outport | Operation |
|---|---|---|---|
| NW | L3 | E | SWAP(L0); PUSH(L1) |
| S | L4 | E | SWAP(L5); PUSH(L1) |

L6 only sees **one** LSP! (Label L1)

# How does ingress LSR determine the label to be used to forward a received packet?

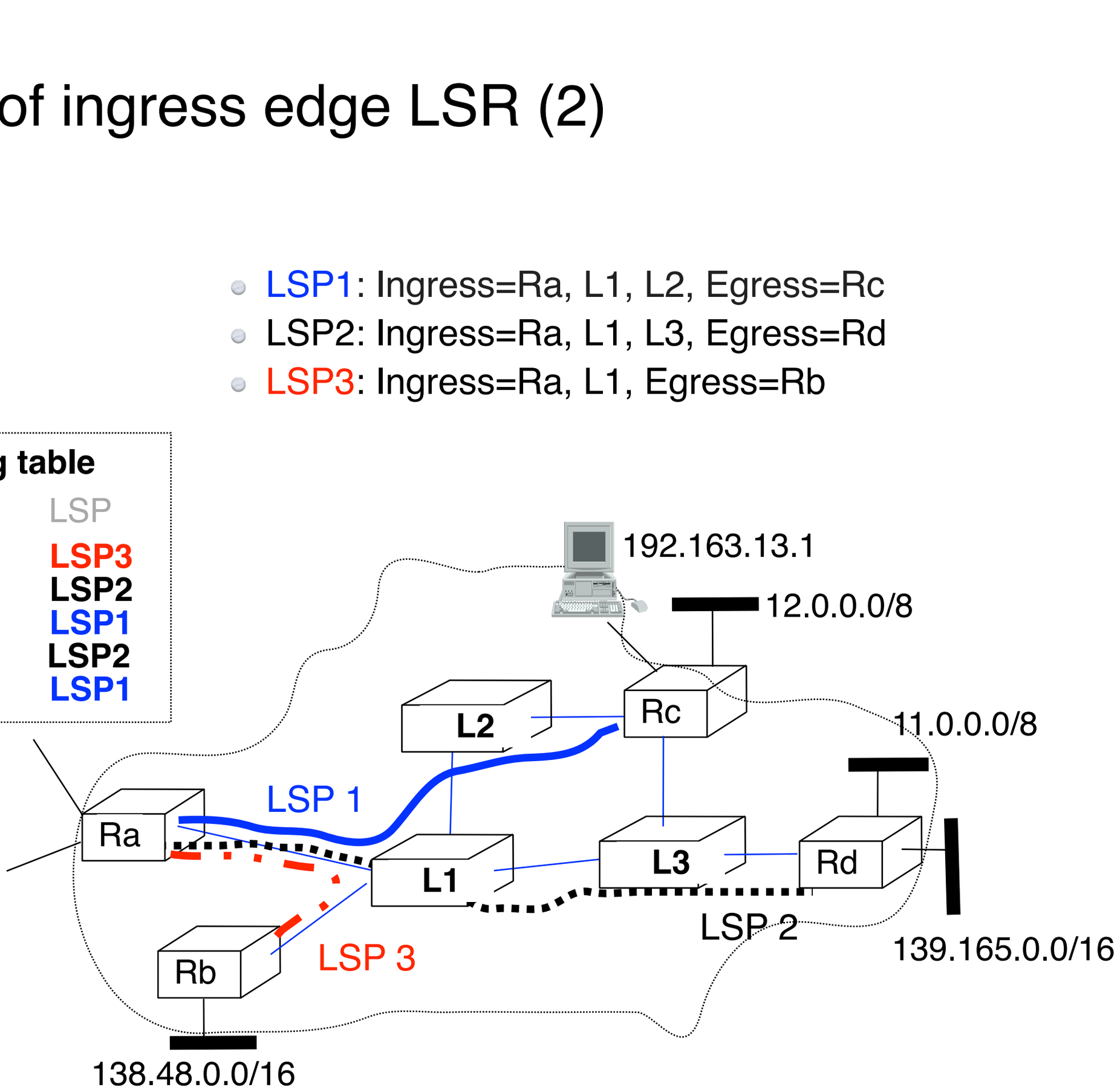
- Principle
  1. Divide the set of all possible packets into several Forwarding Equivalence Classes (FEC)
     - *A FEC is a group of IP packets that are forwarded in the same manner (e.g. over the same path, with the same forwarding treatment)*
     - Examples
       - All packets sent to the same destination prefix
       - All packets sent to the same BGP next hop
  2. Associate the same label to all the packets that belong to the same FEC

# Behaviour of ingress edge LSR (2)

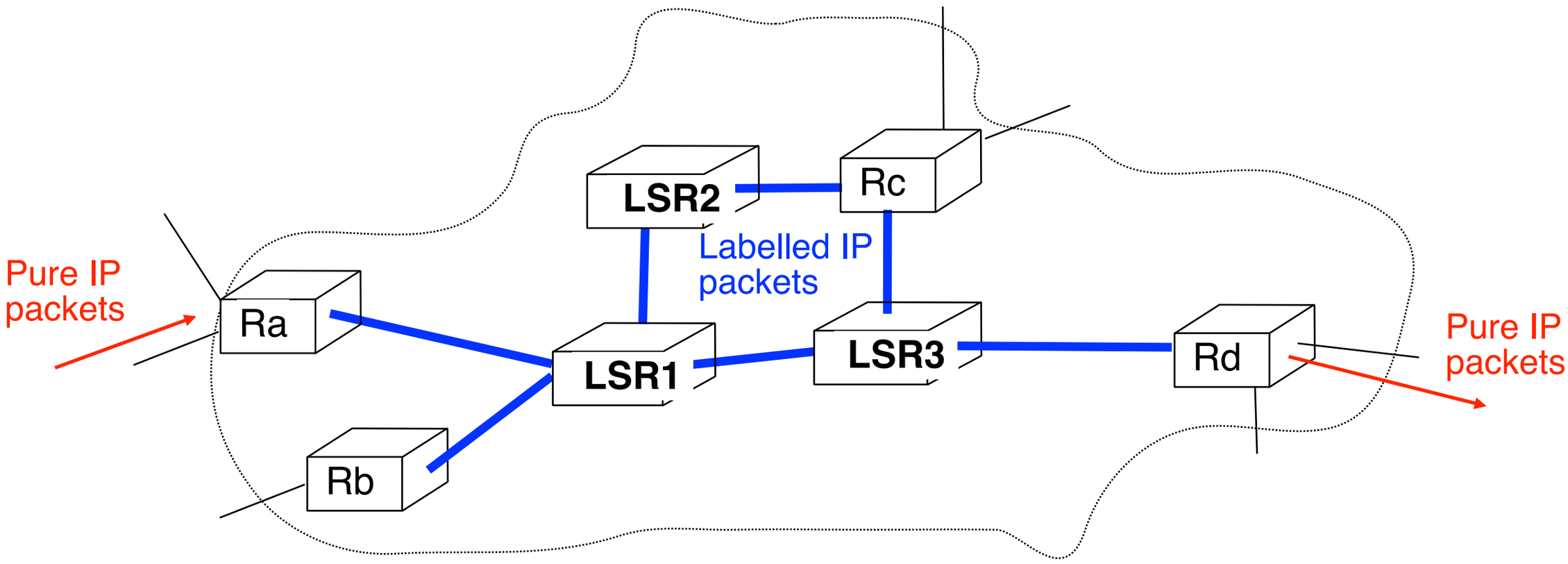
- LSP1: Ingress=Ra, L1, L2, Egress=Rc
- LSP2: Ingress=Ra, L1, L3, Egress=Rd
- LSP3: Ingress=Ra, L1, Egress=Rb

**Ra's Mapping table**

| IP dest prefix | LSP |
|---|---|
| 138.48.0.0/16 | **LSP3** |
| 139.165.0.0/16 | **LSP2** |
| 12.0.0.0/8 | **LSP1** |
| 11.0.0.0/8 | **LSP2** |
| 192.163.13.1 | **LSP1** |



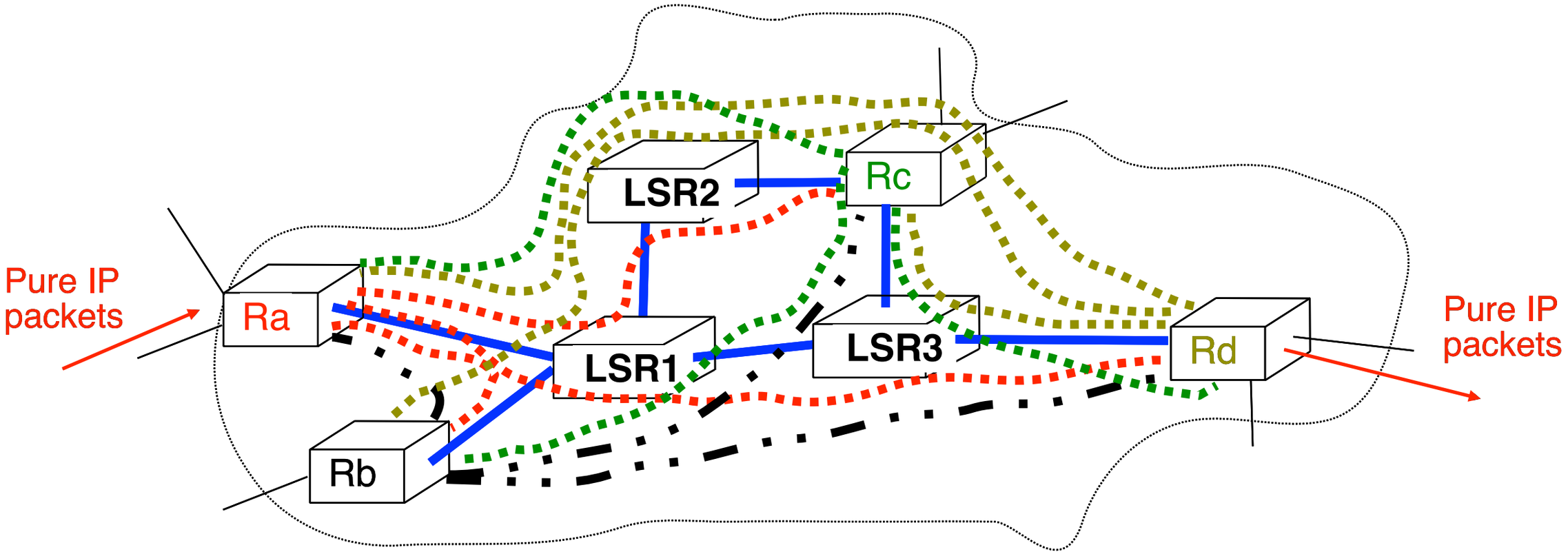
# Destination-based packet forwarding




- How to provide transit service when
  - Edge LSRs are able to attach and remove labels
  - Edge LSR and Core LSRs run IP routing protocols and maintain IP routing tables
  - Core LSR can *only* forward labelled packets
    - Core LSR cannot route IP packets efficiently!

# Destination-based packet forwarding (2)

- Manual solution
  - Create full mesh of LSPs between all edge LSRs



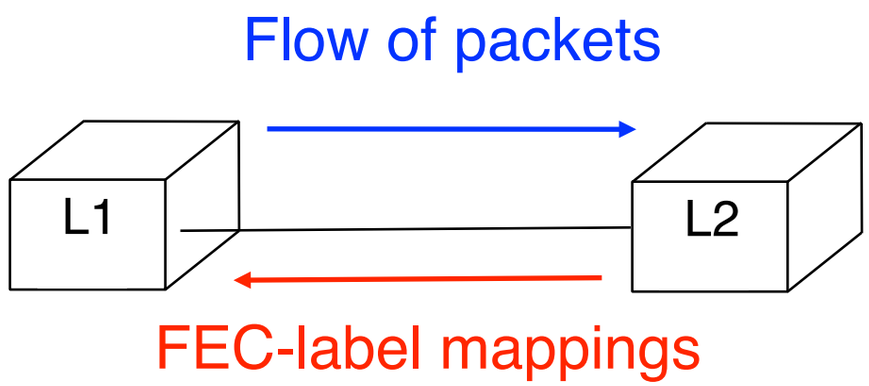

- Problems to be solved
  - N edge LSRs -> N*(N-1) unidirectional LSPs
    - How to automate LSP establishment?
    - How to reduce the number of required LSPs?

# Distributing labels

- How to fill the label forwarding tables of all LSRs in a given network ?
  - Use a dedicated protocol to distribute FEC-label mappings
    - LDP: Label Distribution Protocol
    - RSVP-TE: Resource Reservation Protocol—Traffic Engineering
  - Piggyback FEC-label mappings inside messages sent by existing routing protocols
    - possible if routing protocol is extensible
      - BGP can be easily modified to associate label with route

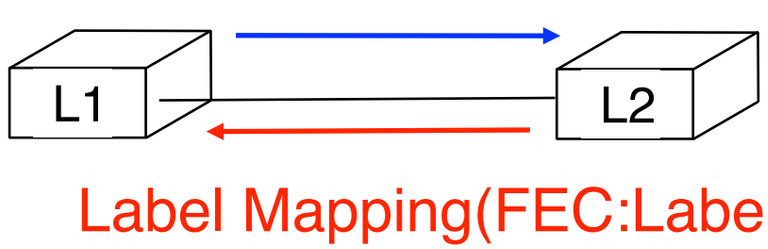
# Distributing labels

- Who determines the FEC-label mapping?
  - Packets are sent by upstream LSR towards downstream LSR
  - FEC-label mappings are sent by downstream LSR towards upstream LSR



# Label distribution modes

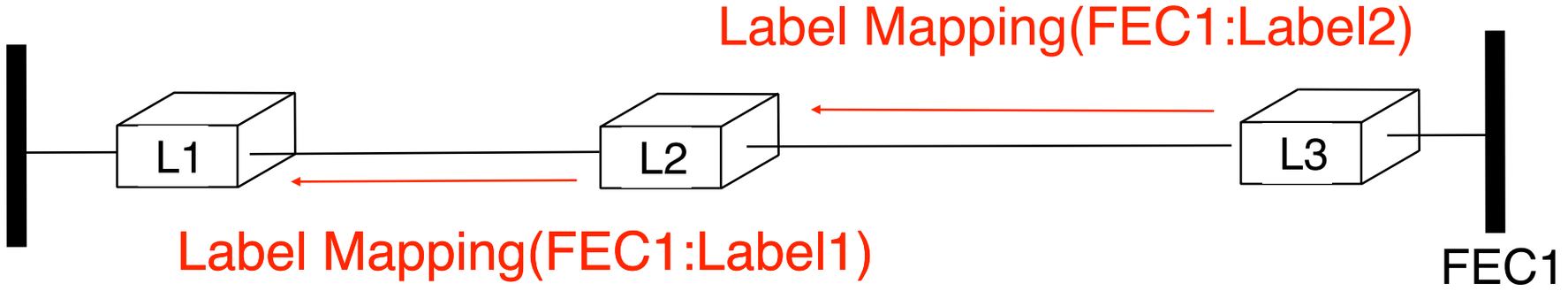
- When to distribute the FEC-label mapping?
  - Downstream on-demand label distribution
    - upstream LSR requests and downstream allocates label




  - advantages
    - the FEC-label mappings are only distributed when needed
    - LSR only need to store the FEC-label mappings that are in use
  - drawbacks
    - when a next-hop fails, some time may elapse while a new FEC-label mapping is being requested from the new next-hop

# Label distribution modes (2)

- When to distribute the FEC-label mapping ?
  - Unsolicited downstream label distribution
    - downstream LSR announces independently FEC-label mappings to upstream LSR




  - advantages
    - Each LSR can obtain several labels for each FEC
    - in case of failure, LSR can quickly switch from one label to another
  - drawbacks
    - Labels may not be distributed at the best time

# LDP: Label Distribution Protocol

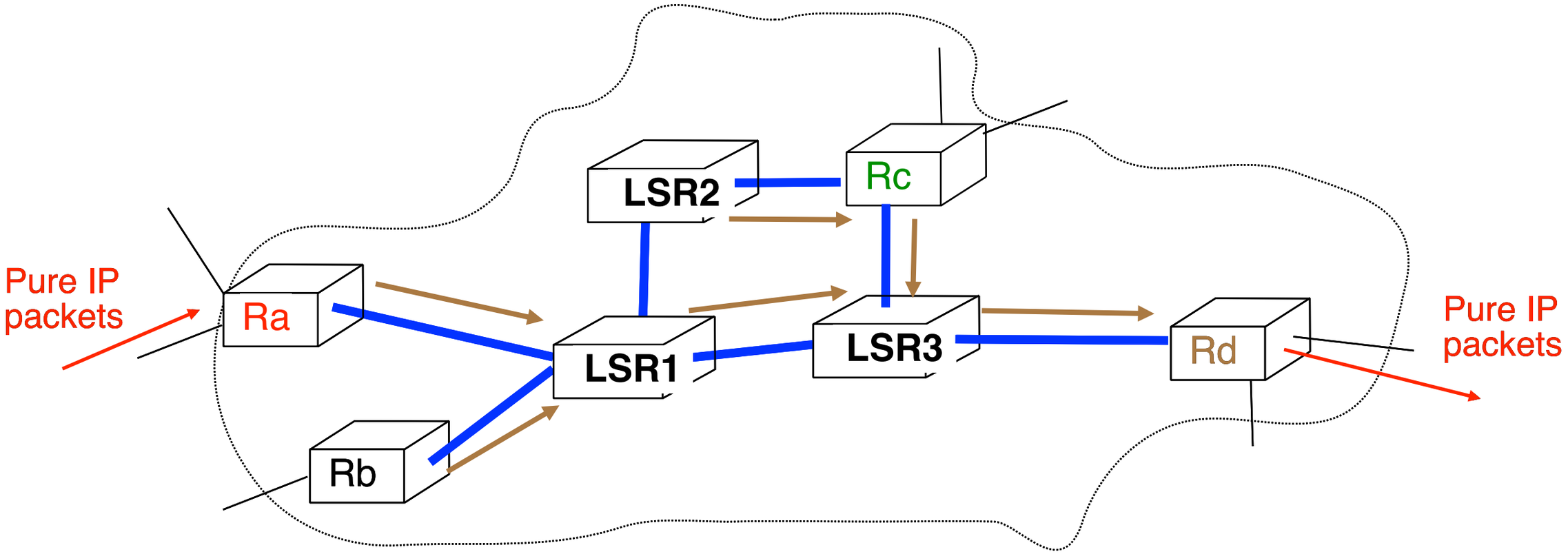
- Designed to distribute FEC-labels mapping on a hop-by-hop basis inside network when labels cannot be distributed by routing protocols
- Neighbour discovery over UDP
- Distribution of FEC-label mappings over TCP
  - several modes of distribution supported

# LDP messages

- Initialisation
  - used during LDP session establishment to announce and negotiate options
- Keepalive
  - sent periodically in absence of other messages
- Label mapping
  - used by LSR to announce a FEC-label mapping
- Label withdrawal
  - used by LSR to withdraw a previous FEC-label mapping
- Label Release
  - used by LSR to indicate that it will not use a  previously received FEC-label mapping
- Label Request
  - used by LSR to request a label for a specific FEC

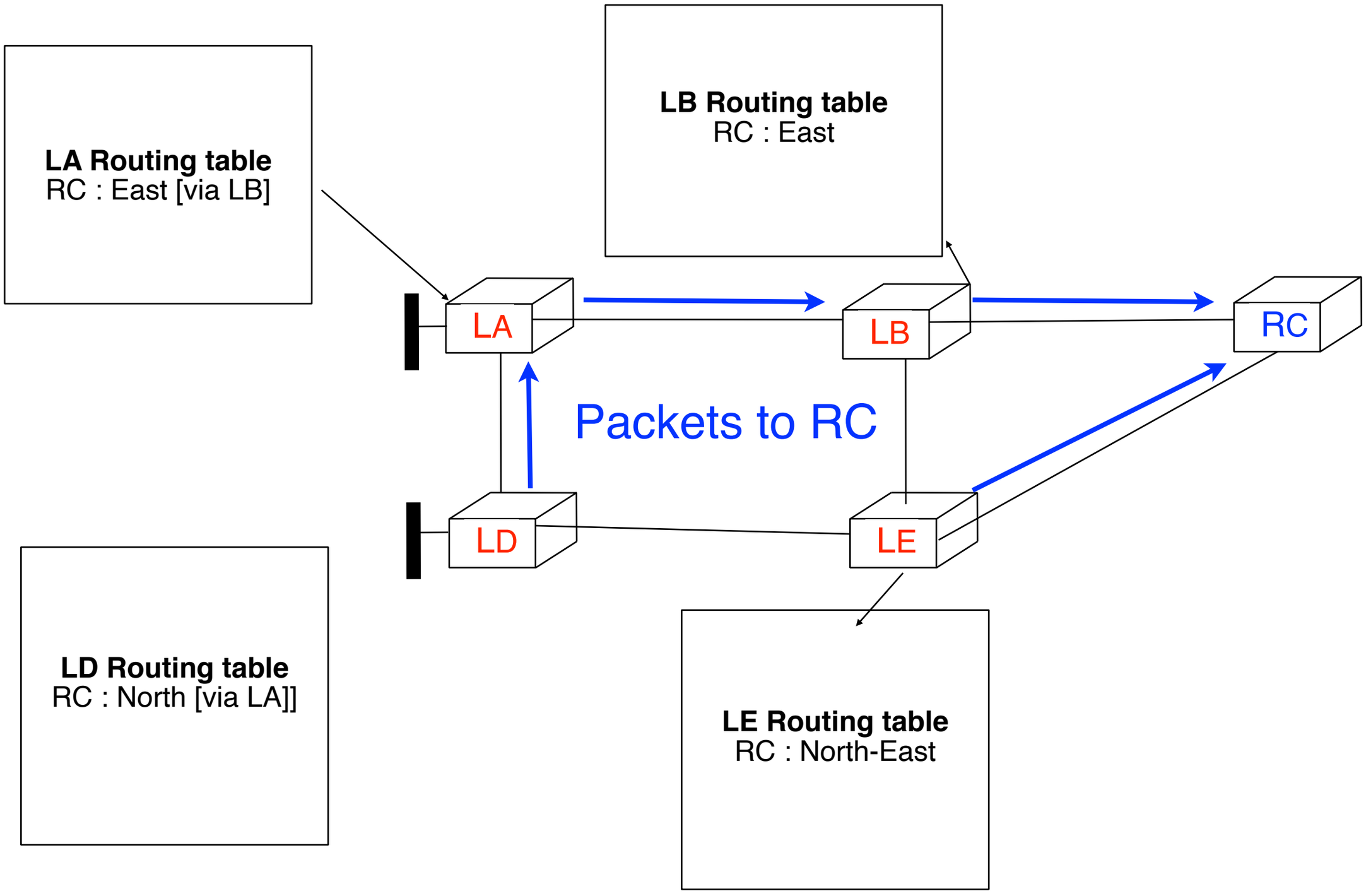
# Destination-based forwarding… with MPLS

- Principle
  - Labelled packets should follow the same path through the network as if they were pure IP packets

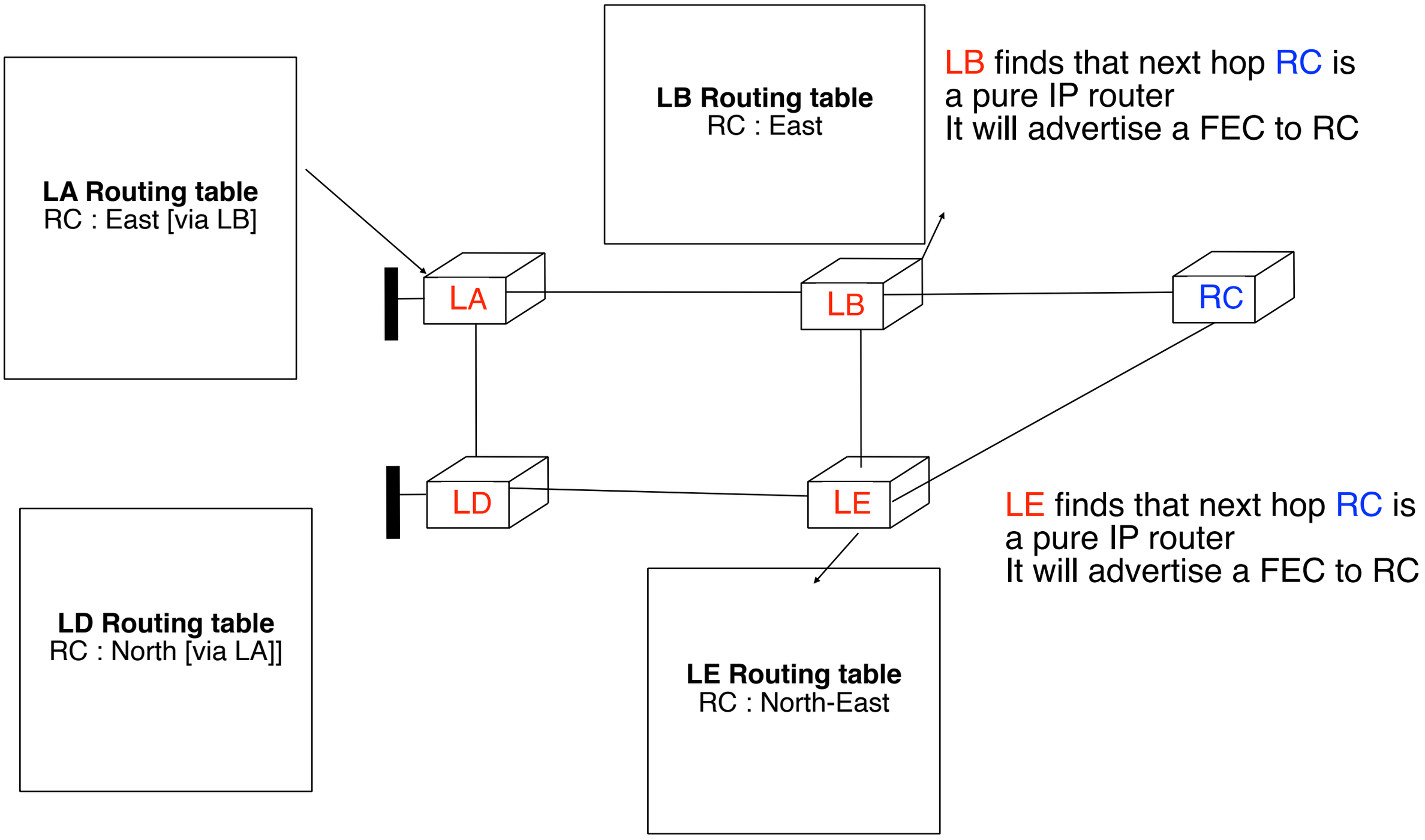



  - create a tree shaped LSP rooted on each egress LSR
    - similar to the way IP routing would forward packets
    - one tree per egress LSR, reduces total number of LSPs
  - distribute the labels to build those trees

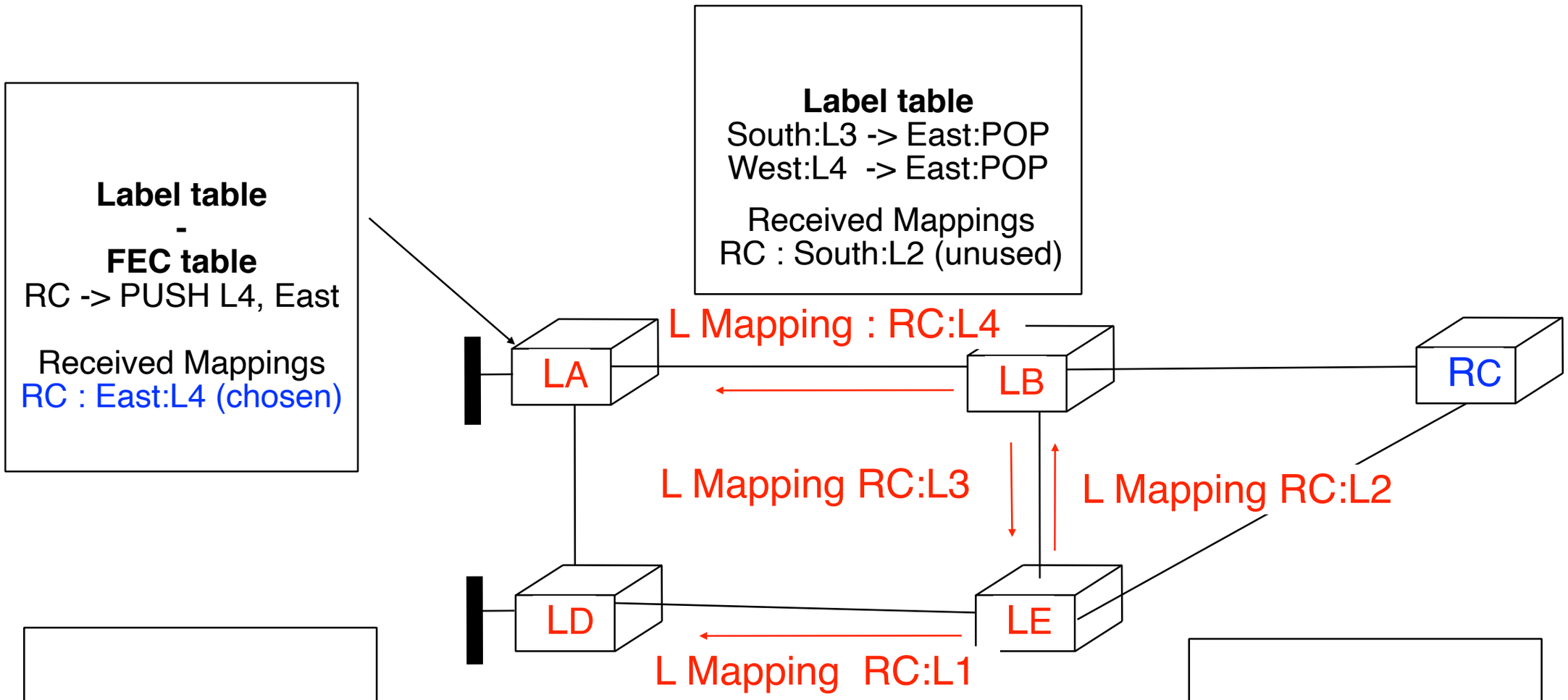
# How does LDP create the LSPs?



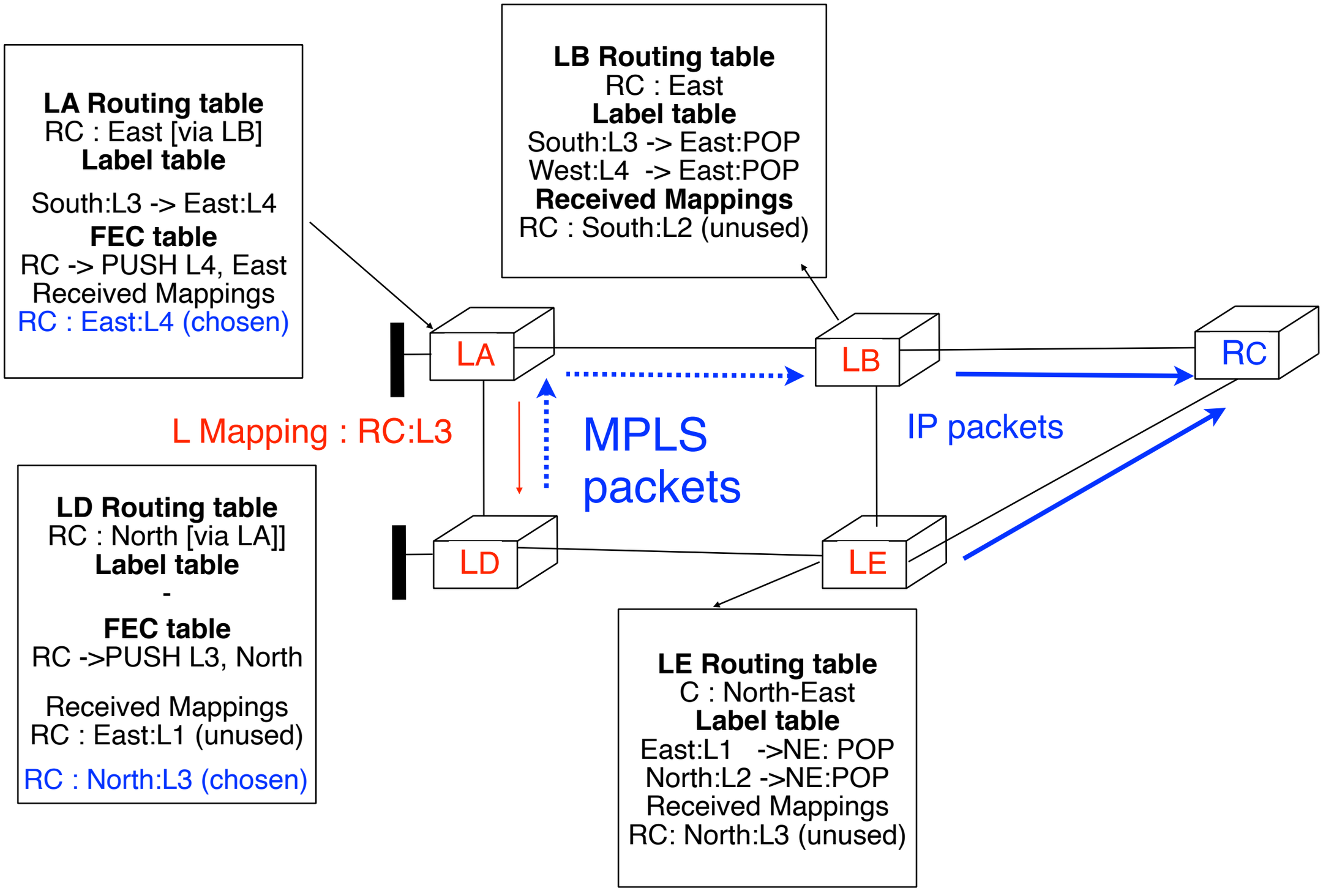
# First, choose a destination for which a LSP will be established



# Then advertise the mappings at LB and LE



# Then advertise the mappings at LA



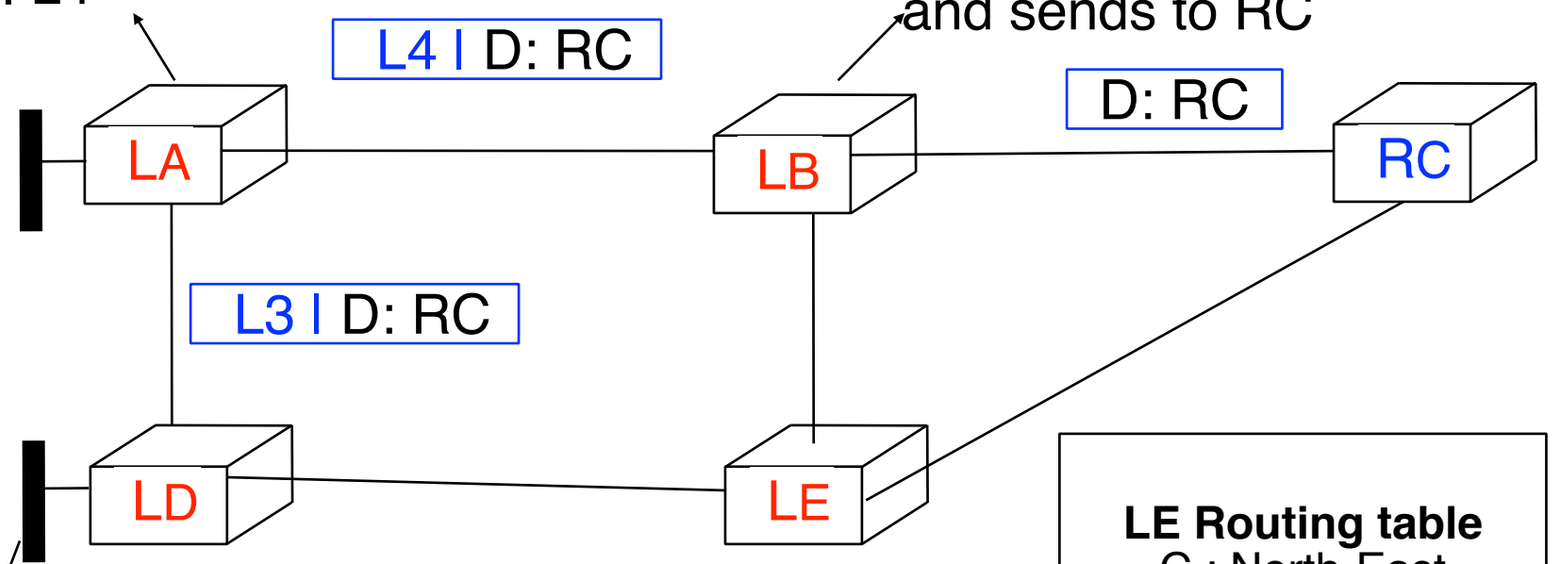
# Actual packet flow

**LA Routing table**
RC : East [via LB]
**Label table**
South:L3 -> East:L4
**FEC table**
RC -> PUSH L4, East
Received Mappings
RC : East:L4 (chosen)

**LB Routing table**
RC : East
**Label table**
South:L3 -> East:POP
West:L4 -> East:POP
**Received Mappings**
RC : South:L2 (unused)

LA swaps label L3 with label L4

LB pops label L4 and sends to RC




**LD Routing table**
RC : North [via LA]]
**Label table**
-
**FEC table**
RC ->PUSH L3, North
Received Mappings
RC : East:L1 (unused)
RC : North:L3 (chosen)

LD pushes label L3 to packets sent towards RC

D: RC

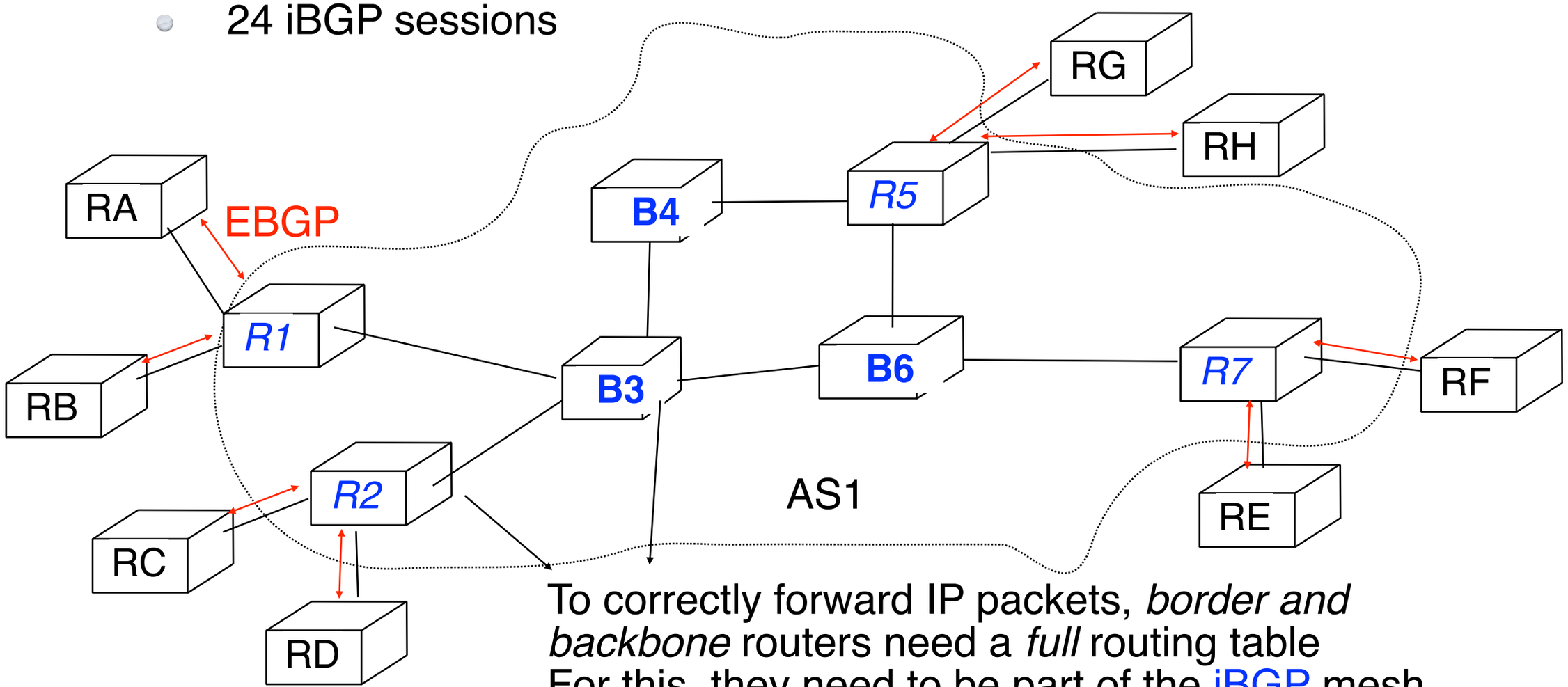
**LE Routing table**
C : North-East
**Label table**
East:L1 ->NE: POP
North:L2 ->NE:POP
Received Mappings
RC: North:L3 (unused)

# How do network operators use LDP in practice?

- Most common deployment is to create a full mesh of LSPs among all LSRs so that each LSR is able to send MPLS packets to any LSR
  - LSR usually advertises a label for its loopback address

# MPLS in large, IP-based ISP networks

- eBGP on border routers
  - current full BGP Internet routing table
    - ~840k active routes
- iBGP full-mesh
  - 4 border, 3 core routers
    - 24 iBGP sessions

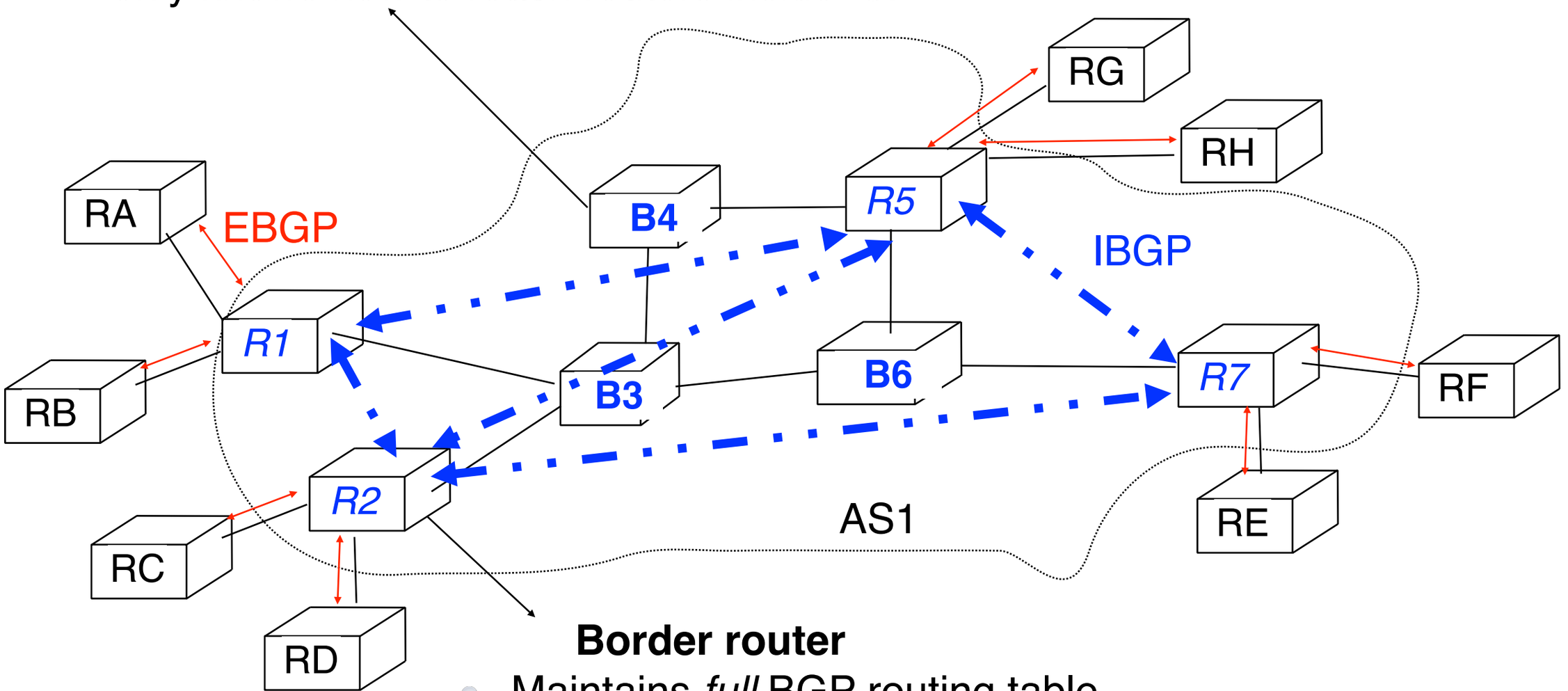



To correctly forward IP packets, *border and backbone* routers need a *full* routing table
For this, they need to be part of the iBGP mesh

# MPLS enables to build *BGP-free* backbones

**Backbone router**

- Maintains *internal* routing table of ISP network
  - only knows how to reach routers *inside* ISP

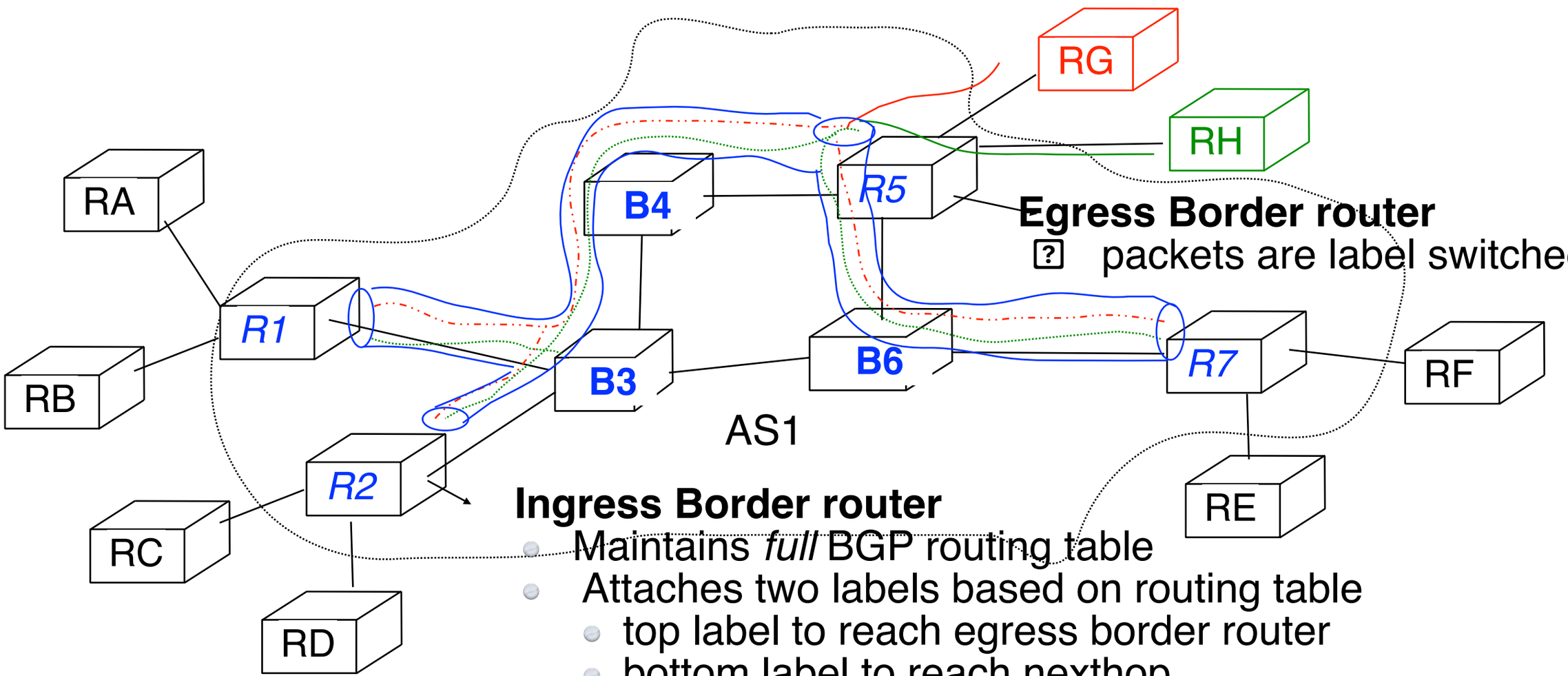



**Border router**

- Maintains *full* BGP routing table
- AS path towards destination
- IP address of next-hop to reach destination

# MPLS enables to build *BGP-free* backbones

- Principle: Use a hierarchy of labels
  - top label is used to reach egress border router (blue LSP)
  - second label is used to reach eBGP peer (red/green LSP)




**Egress Border router**

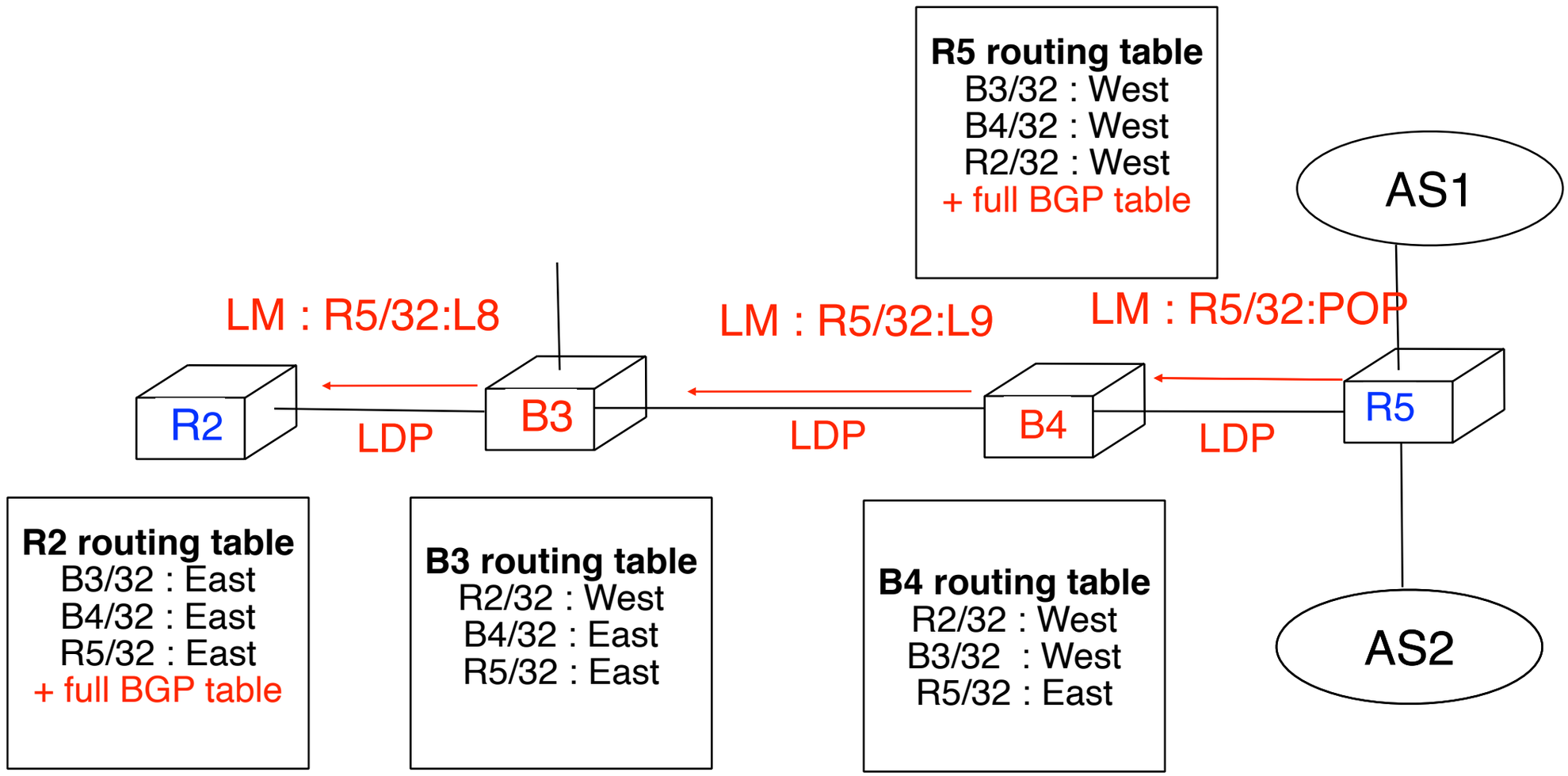
- packets are label switched

**Ingress Border router**

- Maintains *full* BGP routing table
- Attaches two labels based on routing table
  - top label to reach egress border router
  - bottom label to reach nexthop

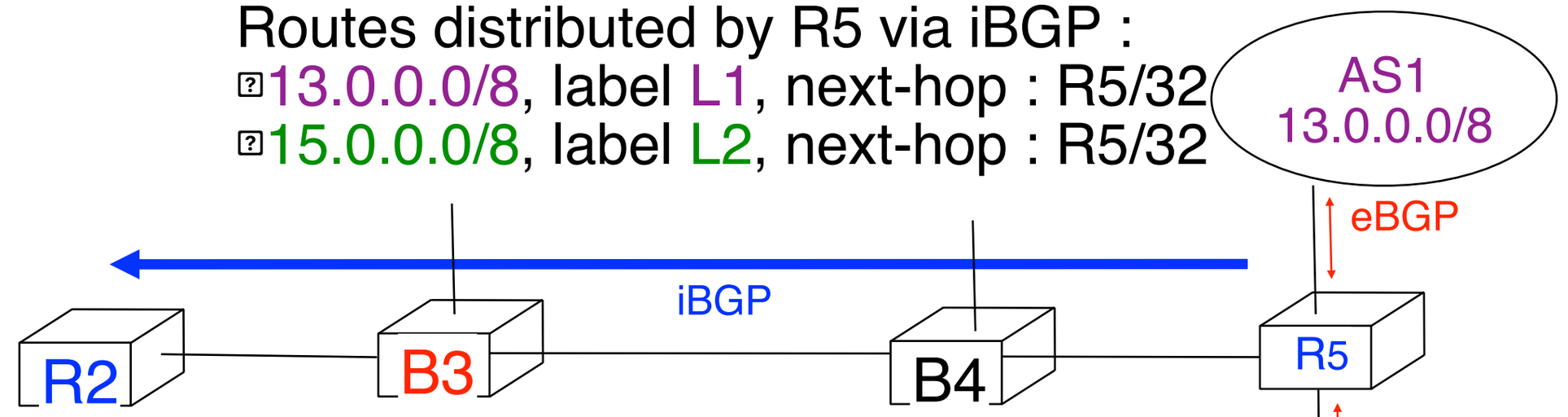
# How do we distribute these two labels?

- LDP allows to distribute the label to reach the egress BGP router



# How do we distribute these two labels?

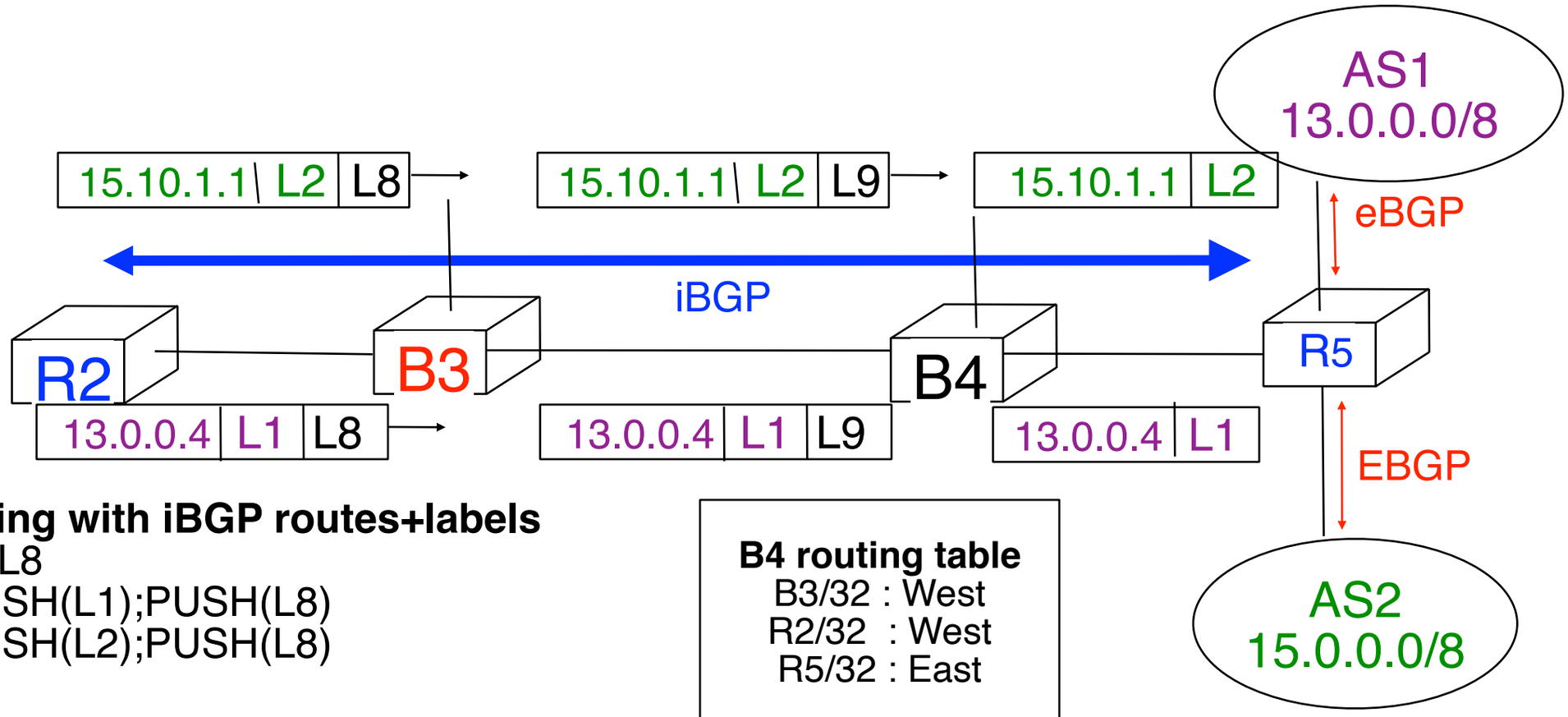
- BGP allows to distribute the label associated to each prefix




**R2 FEC mapping with iBGP routes+labels**
R5/32 : PUSH L8
13.0.0.0/8 : PUSH(L1);PUSH(L8)
15.0.0.0/8 : PUSH(L2);PUSH(L8)

**R5 routing table**
B4/32 : East
B3/32 : East
R2/32 : East
13.0.0.0/8 : North
15.0.0.0/8 : South

# How do we distribute these two labels?




**R2 FEC mapping with iBGP routes+labels**
R2/32 : PUSH L8
13.0.0.0/8 : PUSH(L1);PUSH(L8)
15.0.0.0/8 : PUSH(L2);PUSH(L8)

**B4 routing table**
B3/32 : West
R2/32 : West
R5/32 : East

**Label table**
West:L9 -> East:POP

**R2 routing table**
B4/32 : East
B4/32 : East
R5/32 : East

**LDP FEC mapping**
R2/32 : PUSH L8

**B4 routing table**
R2/32 : West
B4/32 : East
R5/32 : East

**Label table**
West:L8 -> East:L9

**R5 routing table**
B4/32 : East
B3/32 : East
R2/32 : East
13.0.0.0/8 : North
15.0.0.0/8 : South

**R5 Label table**
West:L3 -> Local:POP
L1 -> North:POP
L2 -> South:POP

Advanced Topics in Communication Networks

# Internet Routing and Forwarding

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

22 Sep 2020