DVAD41 - Introduction to Data Plane Programming

Webinar 4 - ECN and Advanced P4 concepts

Announcement

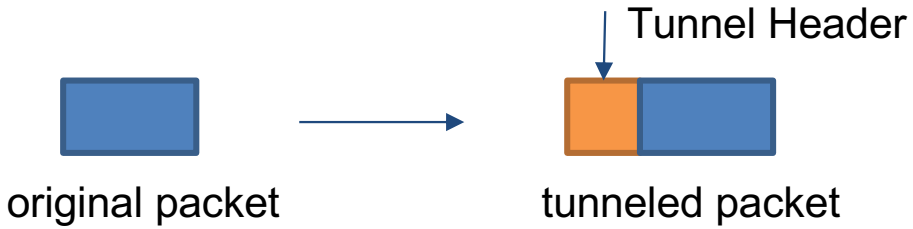
- **There will be an assignment in the week after the module 1 in DVAD40 course.**
  - Unlock: Mar 15, 12am
  - Handin: Mar 22, 11:55pm
- **If you need separate assignment for DVAD41 course module, please let me know.**

Exercise 2: Recap

- **basic_tunnel**

Basic Tunneling

- **Tunneling main feature for**
  - Data Center networks
  - Mobile Core Networks (e.g. Evolved Packet Core - EPC)
  - Network Virtualization (e.g. VXLAN, GRE, ...)
  - Mobility Management (e.g. Mobile IP)
  - Overlay Routing
  - ...
- **How can we implement tunneling?**
  - encapsulate a packet into another one by prepending a new header



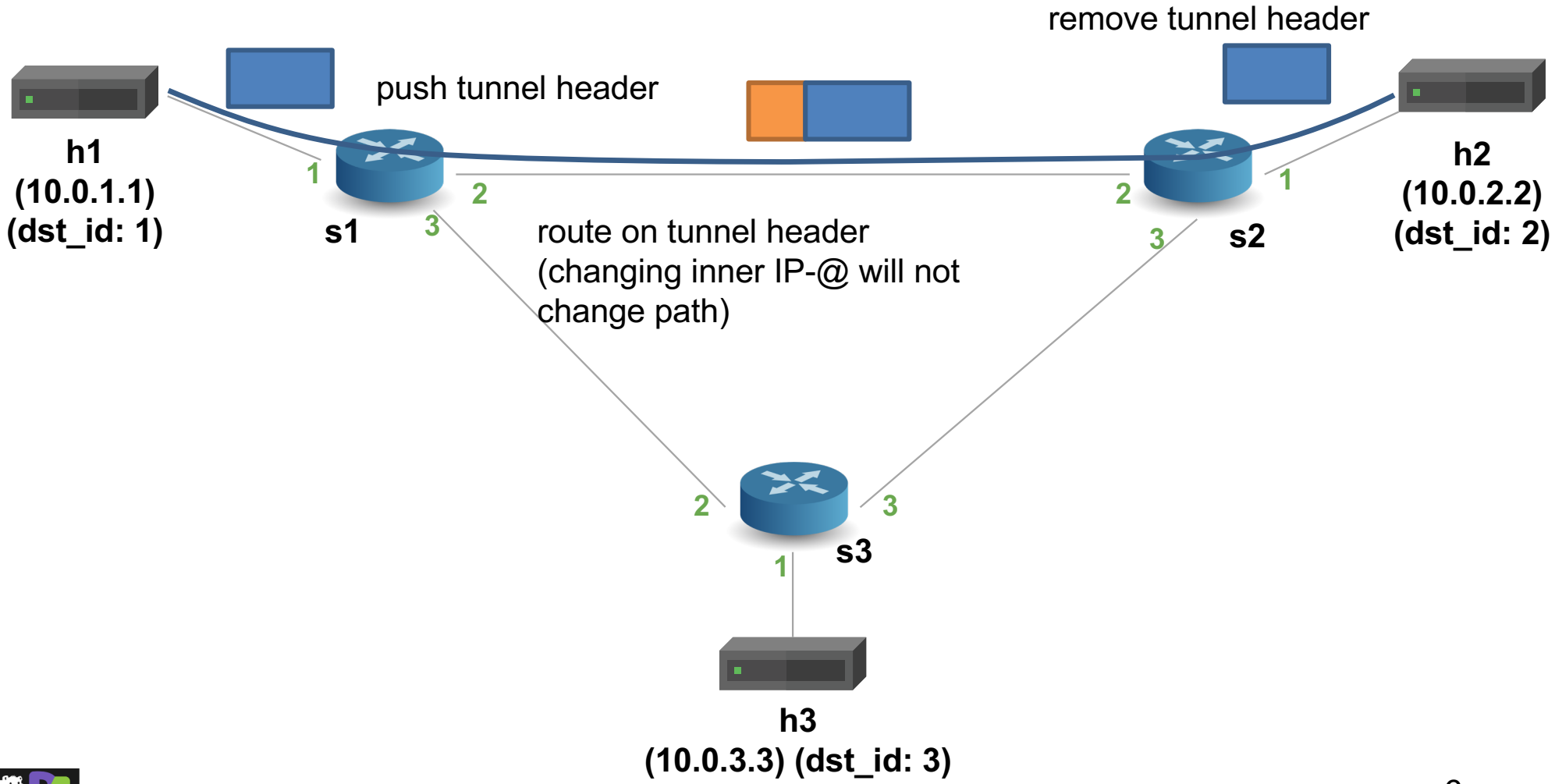
Basic Tunneling

- ToDo: Add support for basic tunneling to the basic IP router in P4

- Define a new header type (`myTunnel`) to encapsulate the IP packet
- `myTunnel` header includes:
  - `proto_id` : type of packet being encapsulated
  - `dst_id` : ID of destination host

- Modify the switch to do routing using the `myTunnel` header

Basic Forwarding: Topology



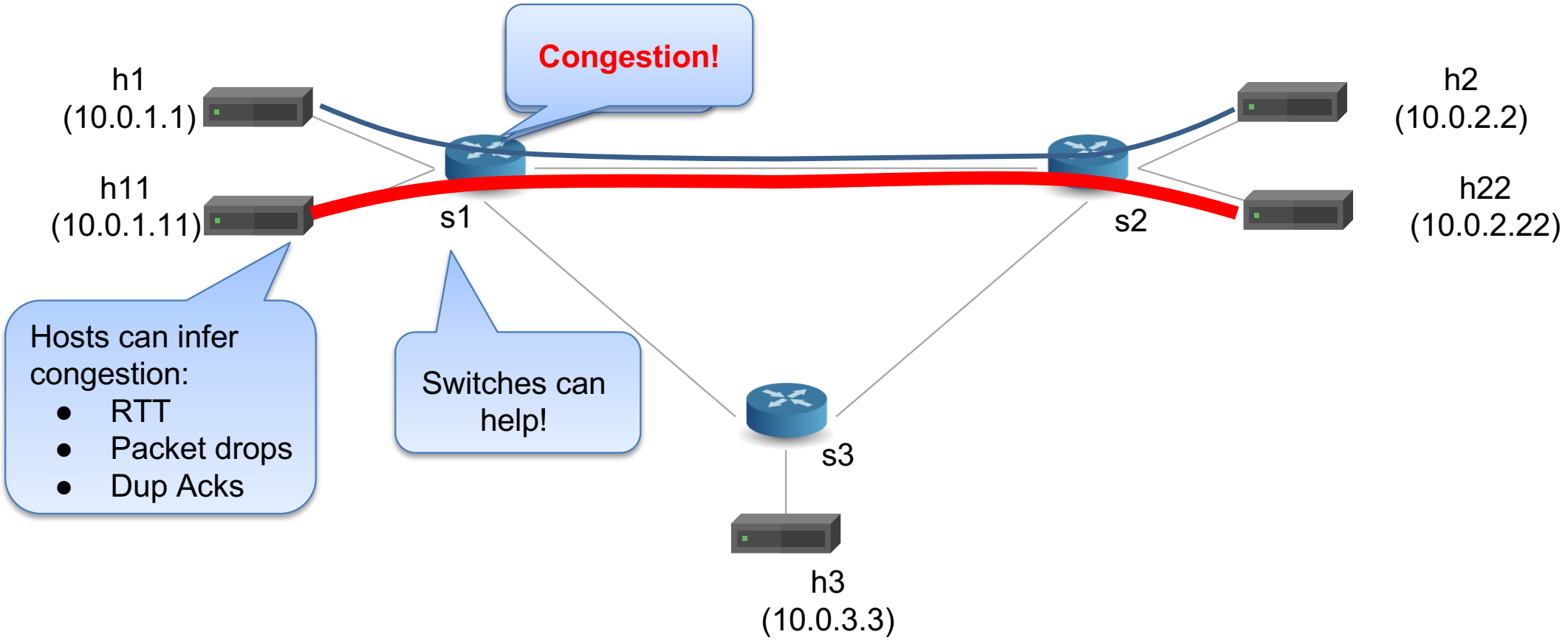
Basic Tunneling TODO List

- **Define `myTunnel_t` header type and add to `headers` struct**
- **Update parser based on ethertype (0x1212: tunnel)**
- **Define `myTunnel_forward` action**
- **Define `myTunnel_exact` table**
- **Update table application logic in `MyIngress` `apply` statement**
- **Update deparser**
- **Adding forwarding rules**
  - myTunnel_forward rule to forward packets on the tunnel header

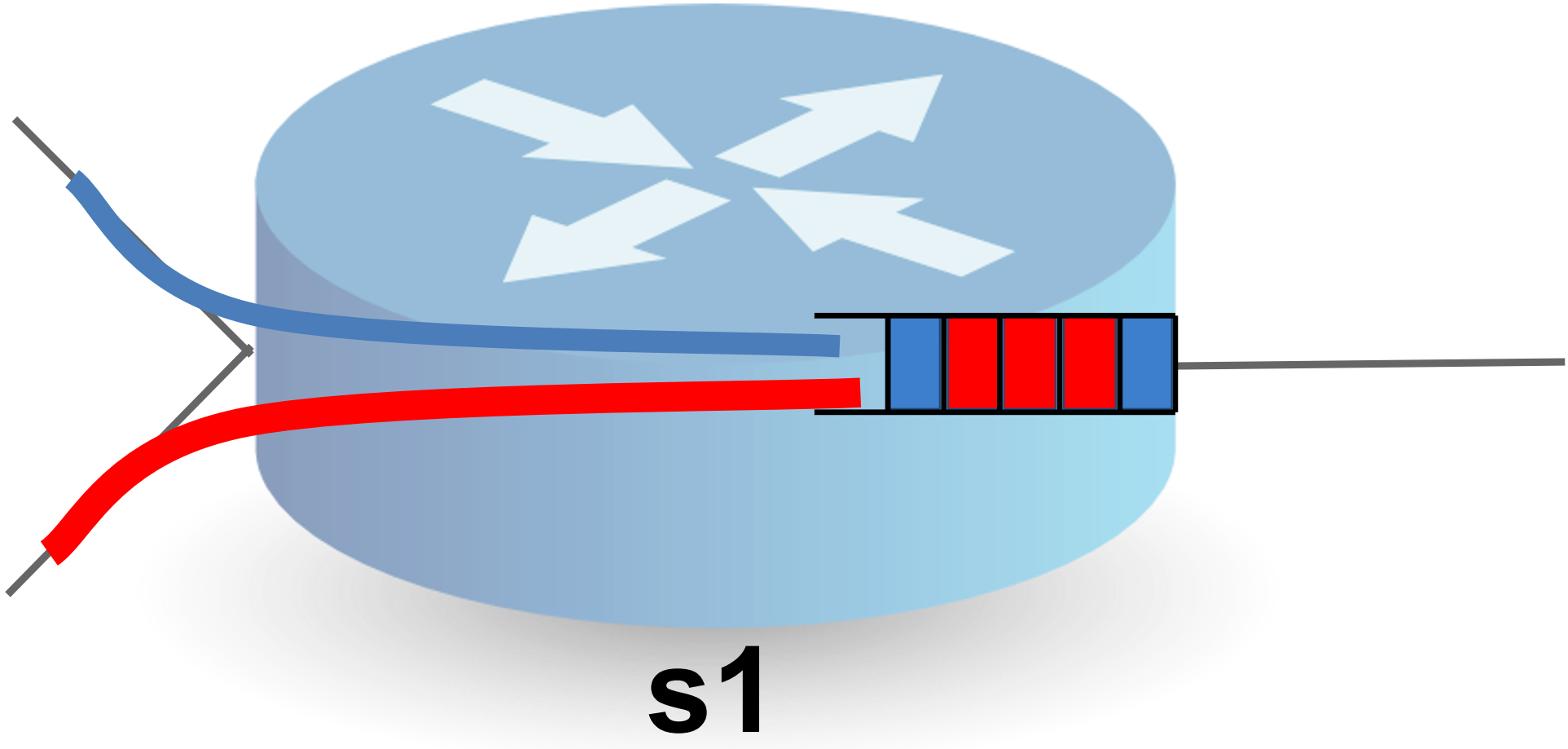
Exercise 3: Monitoring & Debugging

Explicit Congestion Notification - ECN

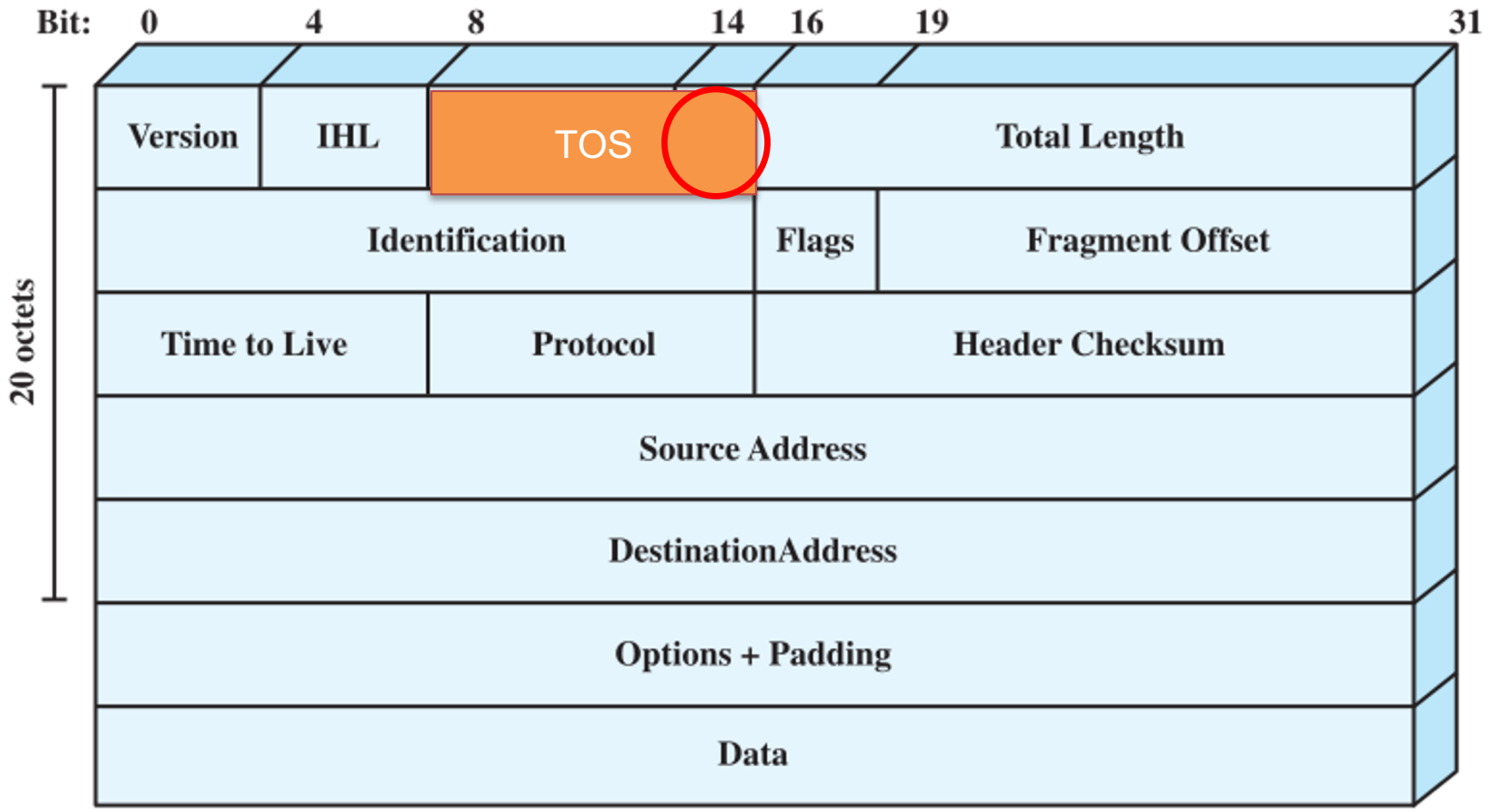
Monitoring & Debugging



Monitoring & Debugging



Explicit Congestion Notification



Explicit Congestion Notification

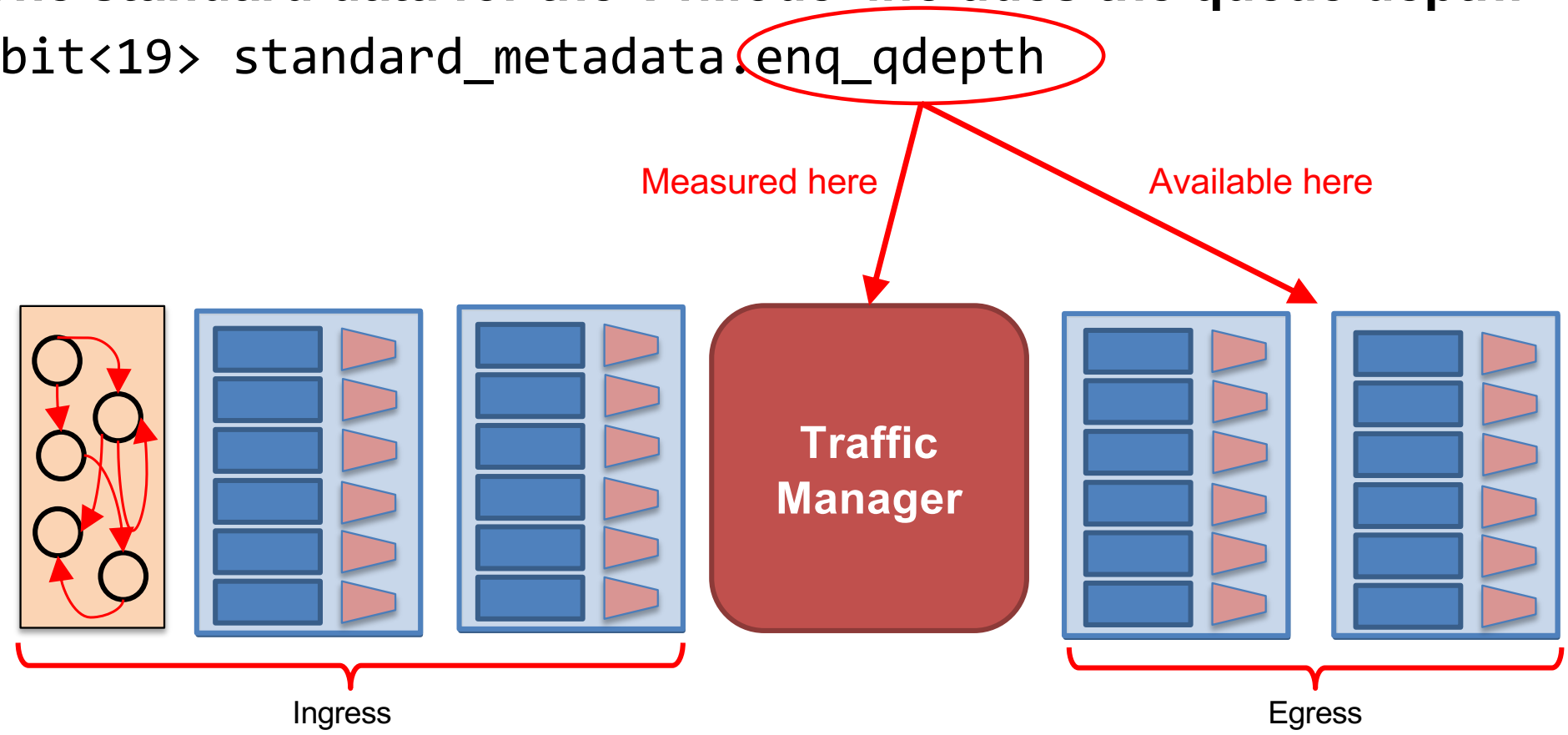
- **Explicit Congestion Notification**
  - `00`: Non ECN-Capable Transport, Non-ECT
  - `10`: ECN Capable Transport, ECT(0)
  - `01`: ECN Capable Transport, ECT(1)
  - `11`: Congestion Encountered, CE

- **For packets originating from ECT, ECN-capable switches set the CE bit upon congestion**
  - E.g., observed queue depth > threshold
  - more details: IETF RFC 3168
    - https://tools.ietf.org/html/rfc3168

Explicit Congestion Notification in P4

- **The standard data for the V1Model includes the queue depth:**

```
bit<19> standard_metadata.enq_qdepth
```



ECN marking

- **ToDo: Add support for ECN marking to the basic IP router**

- **Desired behavior:**
  - If an end-host supports ECN, it puts the value of 1 or 2 in the ipv4.ecn field.
  - For such packets, each switch may change the value to 3 if the queue size is larger than a threshold.
  - The receiver copies the value to sender, and the sender can lower the rate.

ECN marking in P4

- **modify `ipv4_t` to split TOS into DiffServ and ECN**
- **update checksum accordingly**
- **In egress, compare queue length with ECN_THRESHOLD**
  - if queue is larger, set ECN bits to 3 (bin 11) (congestion encountered)
  - do this only if end-host supports by having set original ECN to 1 or 2
- **Define an action to drop a packet which calls mark_to_drop();**
- **Define an egress control block that checks the ECN and `standard_metadata.enq_qdepth` and sets the `ipv4.ecn` accordingly**
- **test your solution by redirecting the receive.py to a log file to check the TOS**

Advanced P4 Constructs

Data Types, Externs, Registers, Meters, etc.

Different Data Types in P4

`bool`	Boolean values
`bit<W>`	Bit-strings of width W
`int<W>`	Signed integer of width W
`varbit<W>`	Bit-string with dynamic length (max W)
~~`float`~~	no support
~~`string`~~	no support

Operators to define composed types in P4

Header

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

We have shown already struct

Header Union

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

either IPv4 or IPv6 header

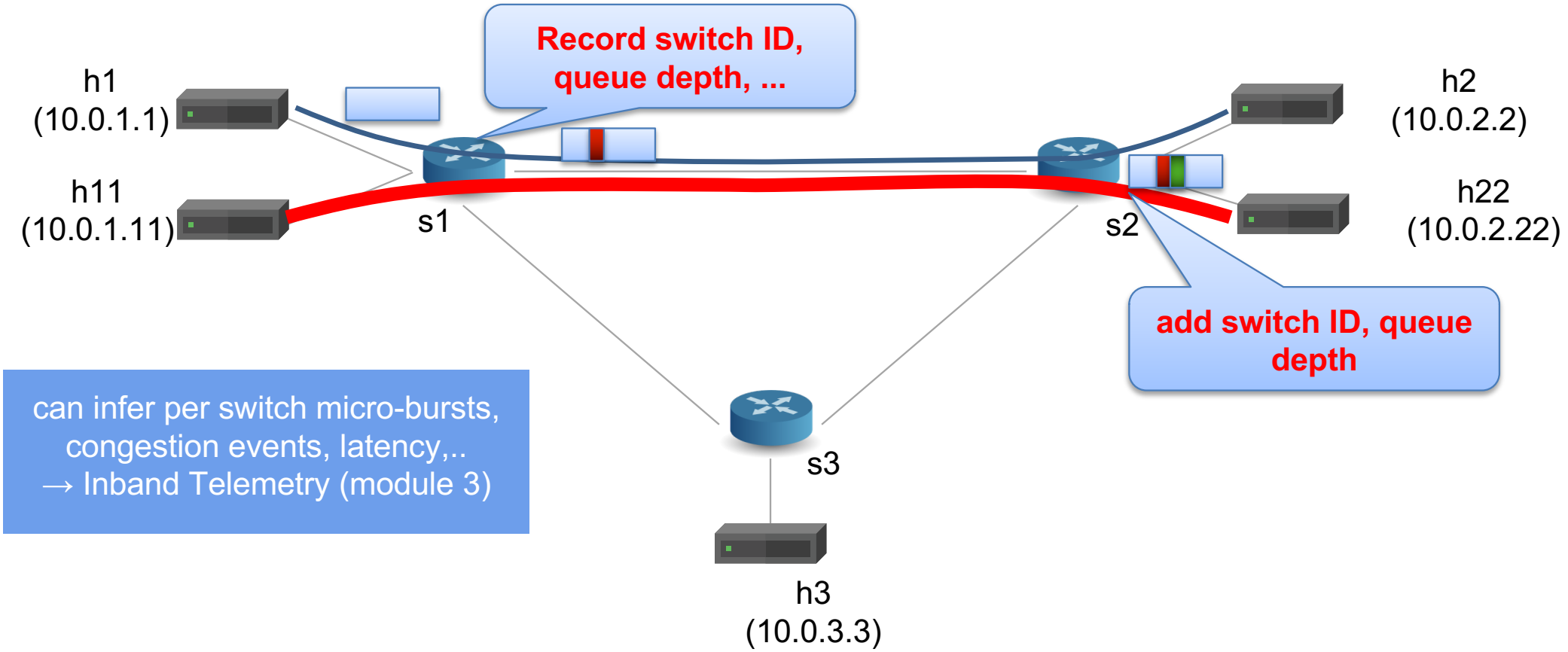
Header Stack

```
header mpls_h {
        bit<20>         label;
        bit<3>          tc;
        bit             BoS;
        bit<8>          ttl;
}
```

mpls_h[8] mpls;

Array of up to 8 MPLS headers

Example Use of Header Stacks - MultiRoute Inspect



can infer per switch micro-bursts, congestion events, latency,..
→ Inband Telemetry (module 3)

Multi-Route Inspect: Packet Format

```
header  mri_t  {
        bit<16>     count;
}
header  switch_t  {
    switchID_t      swid;
    qdepth_t        qdepth;
}
struct  headers  {
        ethernet_t       ethernet;
        ipv4_t           ipv4;
        ipv4_option_t    ipv4_option;
        mri_t            mri;
        switch_t[MAX_HOPS]  swtraces;
}
```

- **Header validity operations:**
  - `hdr.setValid()`: add_header
  - `hdr.setInvalid()`: remove_header
  - `hdr.isValid()`:test validity
- **Header Stacks**
  - `hdr[CNT] stk;`
- **Header Stacks in Parsers**
  - `stk.next`
  - `stk.last`
  - `stk.lastIndex`
- **Header Stacks in Controls**
  - `stk[i]`
  - `stk.size`
  - `stk.push_front(int count)`
  - `stk.pop_front(int count)`

State Management in P4

- **Stateless Objects**
  - Variables (metadata), headers,..do not maintain state across packets
- **Stateful Objects**
  - Tables
  - Externs in P4-14: Counters, Meters, ...keep state across different packets

Object	Data Plane Interface		Control Plane Can	
	Read State	Modify/Write State	Read	Modify/Write
Table	apply()	---	Yes	Yes
Parser Value Set	get()	---	Yes	Yes
Counter	---	count()	Yes	Yes*
Meter	execute ()		Configuration Only	Configuration Only
Register	read()	write()	Yes	Yes

WriteRequest with the MODIFY update type

P4 Registers

- Store arbitrary data (single values or arrays of N entries)
- Definition:

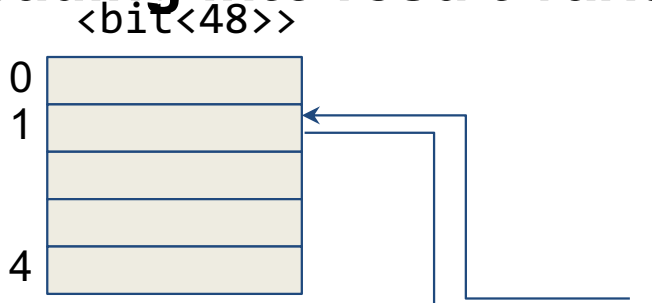
```
register<Type>(N) reg;
```

- writing:

```
reg.write(n, val) //0 <= n <N
```

- reading into result variable:

```
reg.read(result,n)
```




```
register <bit<48>>(5) hello;

hello.write(1,0xff);
hello.read(res, 1);
```

Example: Inter packet gap detection

```
register<bit<32>>(8192) flowlet_lasttimeseen;

action flowlet_gap(out bit<32> delta, bit<32> flow_id)
{
    bit<32> last_pkt_seen;

    /* Get the time the previous packet was seen for same flow */
    flowlet_lasttimeseen.read(last_pkt_seen, flow_id);

    /* Calculate the time interval */
    delta = standard_metadata.ingress_global_timestamp - last_pkt_seen;

    /* Update the register with the new timestamp */
    flowlet_lasttimeseen.write(flow_id,
        standard_metadata.ingress_global_timestamp);
    ...
}
```

Caveat: concurrent read and write needs to be synchronized if required

What is this code doing?

P4 Counters

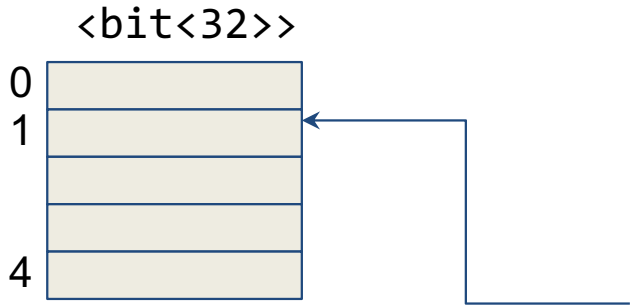
- used to count packets, bytes or both, formed in arrays
- Definition:

```
counter(N, Type) my_count;
```

- Updating:

```
my_count.count(n)
```

```
enum CounterType {
    packets,
    bytes,
    packets_and_bytes
}
```




```
counter (5,CounterType.packets_and_bytes) hello;
```

```
hello.count(1); //increases counter at position 1
with current packet size information
```

Example: Count incoming packet and bytes per port

```
control MyIngress(...) {

    counter(64, CounterType.packets_and_bytes) c;

    apply { //ingress port number as index
        c.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

Interaction from Control Plane

```
RuntimeCmd: counter_read MyIngress.c 1   //will then return
MyIngress.c[1] = BmCounterValue(packets=1, bytes=658)

//Note: cannot access counter information from within data plane
```

Example: counter use

```
control MyIngress(...) {
    counter(64, CounterType.packets) c;

  action tally() {
    c.count((bit<32>) standard_metadata.ingress_port);  }

  table monitor {
    key = {
      hdr.ipv4.srcAddr: lpm;   }
    actions = { tally; NoAction; } }

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

Question: What this P4 code does?

```
{
  "table": "MyIngress.monitor",
  "match": {
    "hdr.ipv4.srcAddr": ["10.0.1.1", 32]
  },
  "action_name": "MyIngress.tally",
  "action_params": { }
}
```

Example: using direct counters

```
control MyIngress(...) {
    direct_counter(CounterType.packets) c;

  action tally() {
    c.count();  }

  table monitor {
    key = {
      hdr.ipv4.srcAddr: lpm;   }
    actions = { tally; NoAction; }
    counters = c;
    size =1024}
  apply {
    ...
    if(hdr.ipv4.isValid()) {
      ...
      monitor.apply();
    }
```

- Direct counters are attached to tables
- Each table entry has a counter that counts upon match

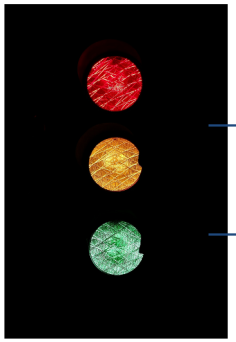
Question: What this P4 code does?

```
{
  "table": "MyIngress.monitor",
  "match": {
    "hdr.ipv4.srcAddr": ["10.0.1.1", 32]
  },
  "action_name": "MyIngress.tally",
  "action_params": { }
}
```

P4 Meters

- used to measure packet rates, can be formed in arrays
- Definition: `meter(N, Type) my_meter;`
- Applying: `my_meter.execute(n)`

```
enum MeterType {
    packets,
    bytes
}
enum MeterColor {
    RED, GREEN, YELLOW }
```



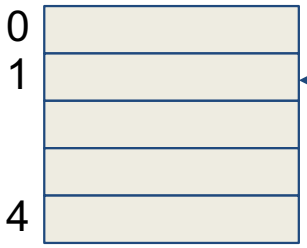
more info https://tools.ietf.org/html/rfc2698

exceeds Peak Information Rate (PIR) per sec.

exceeds Committed Information Rate (CIR) but still below PIR per sec.

does not exce

<bit<32>>




```
meter (5,MeterType.packets) hello;
hello.execute(1);
```

Rates set by control plane per instance

```
meter_set_rates hello 1 0.0001:1 0.0005:1
.0001=> allow 100packets/sec , if the each
packet size is equal to 1000 bytes, then the
obtained throughput can be 100 packets/sec *
1000 (bytes) *8 = 800 kbps
```

Example: Rate-limitig with meters

```
control MyIngress(...){
 meter(16384, MeterType.packets) acl_meter;

action color_my_packets(bit<32> index) {
       acl_meter.execute_meter((bit<32>)index, meta.meta_tag);
}

table m_read {
       key = { hdr.ethernet.srcAddr: exact; }
       actions = { color_my_packets; NoAction; }
       …
       }

table m_filter {
       key = { meta.meta_tag: exact; }
       actions = { drop; NoAction; }
       ...
}
apply {
       m_read.apply();
       m_filter.apply();
}
}
```

- can also use direct meters
- direct meters are attached to table entry

```
// This is v1 model code
// https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4
// meta.meta_tag now holds the color
```

depending on the meter tag, treat packets differently

Question: What this P4 code does?

Example: Rate-limitig with direct meters

```
control MyIngress(...){
 direct_meter(16384, MeterType.packets) acl_meter;

action color_my_packets(bit<32> index) {
       acl_meter.read(meta.meta_tag);
}

table m_read {
       key = { hdr.ethernet.srcAddr: exact; }
       actions = { color_my_packets; NoAction; }
       meters = acl_meter;
       …
       }

table m_filter {
       key = { meta.meta_tag: exact; }
       actions = { drop; NoAction; }
       ...
}
apply {
       m_read.apply();
       m_filter.apply();
}
}
```

```
// This is v1 model code
// https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4
// meta.meta_tag now holds the color
```

```
@ControlPlaneAPI
  {
    reset(in MeterColor_t color);
    setParams(in S index, in MeterConfig config);
    getParams(in S index, out MeterConfig config);
  }
  */
```

attach acl_meter to table m_read

Summary

- Several Stateful constructs to record and update per flow/packet state
- Useful for many things,
  - e.g. congestion tracking, stateful forwarding,....
- Will see some of these concepts applied in next module
  - e.g. congestion aware load-balancing

Module Summary - P4

- **Clearly defined semantics**
  - You can describe what your data plane program is doing
- **Expressive**
  - Supports a wide range of architectures through standard methodology
- **High-level, Target-independent**
  - Uses conventional constructs
  - Compiler manages the resources and deals with the hardware
- **Type-safe**
  - Enforces good software design practices and eliminates “stupid” bugs
- **Agility**
  - High-speed networking devices become as flexible as any software
- **Insight**
  - Freely mixing packet headers and intermediate results

Module Summary - P4 things we covered

- **The P4 world**
  - Protocol Independent Packet Processing
  - Language/Architecture separation
  - If you can interface with it, you can use it
- **Key data types**
- **Constructs for packet parsing**
  - state-machine type
- **Constructs for packet processing**
  - Actions, tables, controls
- **Packet Deparsing**
- **Architectures and Programs**

Module Summary - P4 things we did not cover

- **Enforcing Modularity**
  - Instantiating and invoking parsers or controls
- **Variable Length field processing**
  - parsing and deparsing of TLVs
- **Architecture definition constructs**
  - How to create such template definitions
- **Advanced features**
  - learning, multicast, cloning, resubmitting
  - header unions
- **Control Plane Interface**

Next Module

- **Datacenter Load-balancing**
  - What is a datacenter
  - What is data center networking
  - Traffic characteristics in a data center
  - Load-balancing techniques for data center networking
  - P4 based Load-balancing
- **When?**
  - March 15th, 17:00 CET