[S05] OpenFlow

Redes Software
Múltiples Grados

Curso 2024-2025

Antonio de la Oliva – Universidad Carlos III de Madrid (aoliva@it.uc3m.es)

Pedro Aranda – Universidad Carlos III de Madrid (paranda@it.uc3m.es)

Outline

- OpenFlow Switch Architecture
- Implementation of OpenFlow Switches
- OpenFlow 1.5 Specification

Implementation of OpenFlow Switches

Implementing OF switches

- OpenFlow specification does not mandate an implementation option
- It only defines common set of behaviours
- Each manufacturer is free to implement OF on its way
  - Enabling innovation and differentiation
  - While preserving interoperability

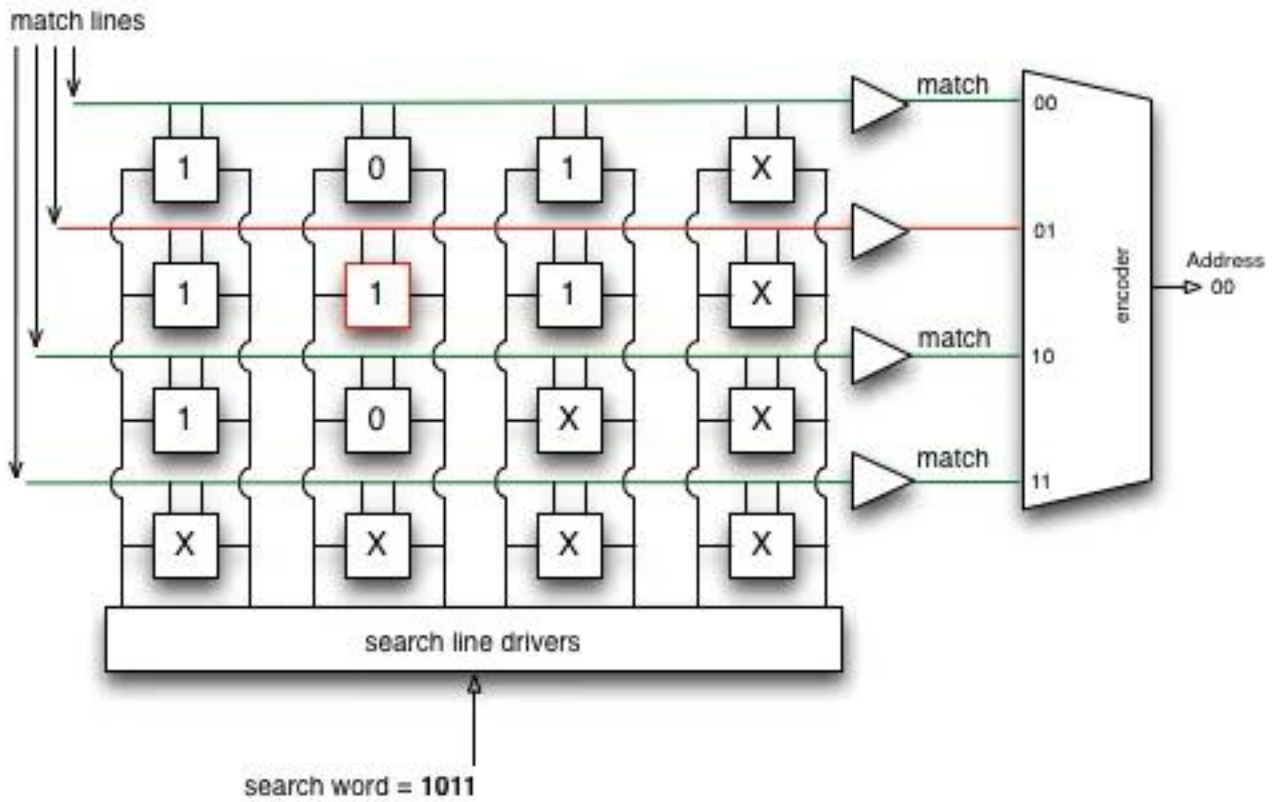
Implementing OF-TCAMS

- One of the key points of OpenFlow was that it was easily implemented in hardware
- OF operation based on matches and actions associated
  - It makes use of a generic memory platform which is the base of all switches and routing devices→Ternary Content Addressable Memory (TCAM)
  - TCAM is an evolution of CAM
  - Content Addressable Memory (CAM) allows you to input a search ‘word’ and search the entire memory in a single operation returning one (or more) matches.

How TCAM works

- CAM a special type of RAM which combines both storage cells and the comparison logic-gates.
- However, CAM isn't well suited to prefix lookups, as it can only perform exact matches.
- The Ternary in TCAM refers to the three states which can exist in a TCAM cell. The 'Ternary' in TCAM refers to the third state which can exist in a TCAM cell. Regular (Binary) CAM stores two states '1' and '0'. TCAMs can store a 'don't-care' bit and require twice the number of logic gates to handle this more complex state and 3-way decision.
- TCAM returns the first match, not the longest, ordering is required

How TCAM works



TCAM and OF

- Incoming frame header fields are looked up in the TCAM, which allows partial matches with the usage of wildcards. Matches in the TCAM point to an entry in the RAM, which controls various operations to be performed in the incoming frame header fields as well as extracting and propagating these fields to subsequent stages.
- TCAM tables can run out of space

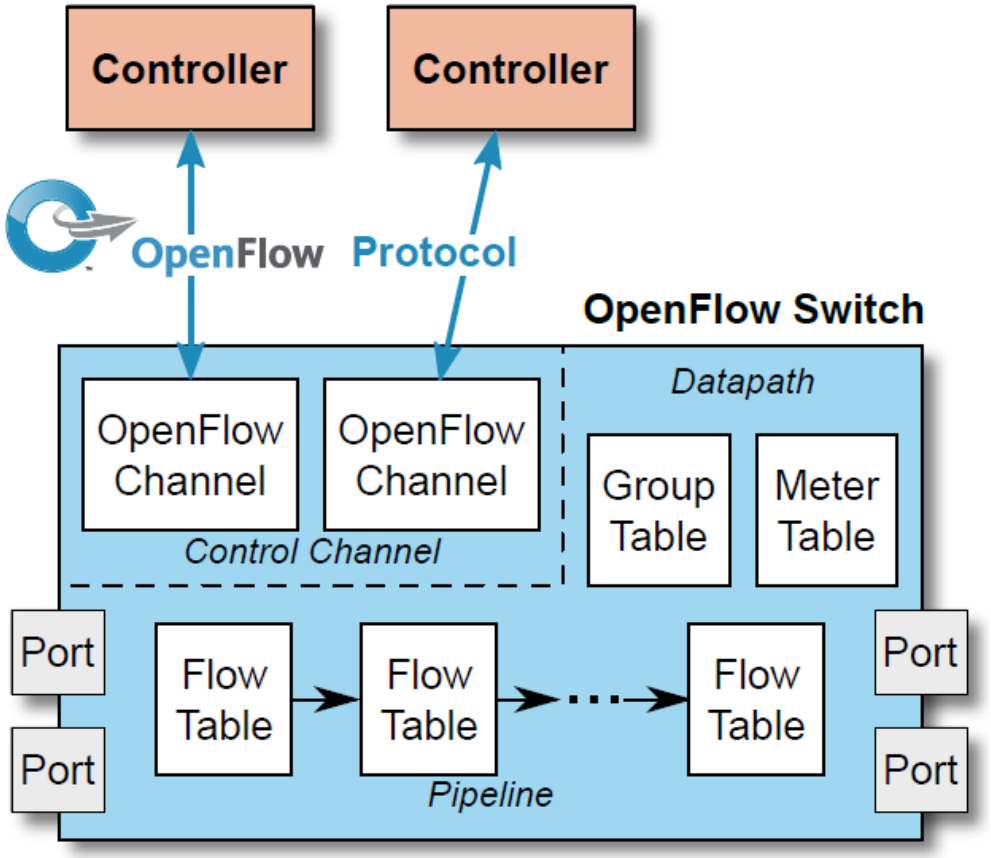
OpenFlow Switch Architecture

Check your hardware

- Very important while working in OF
- Great difference between OF 1.0 and 1.3 and on.
- Check the capabilities of your hardware
  - OF version and what it is supported
  - A lot of optional capabilities
    - Vendors diversify their products on capabilities
  - Understand the caveats of your hardware implementation

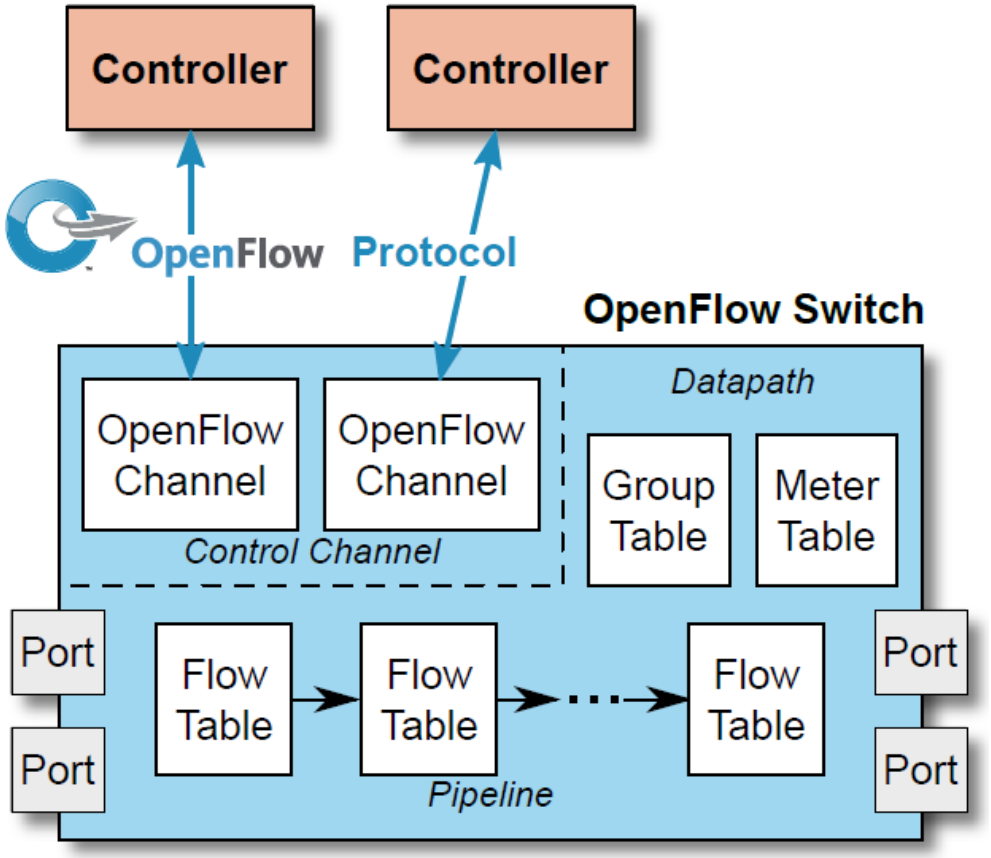
OpenFlow – Overview

- An OpenFlow Logical Switch consists of one or more *flow tables* and a *group table*, which perform packet lookups and forwarding, and one or more *OpenFlow channels* to an external controller
- The controller can add, update, and delete flow entries in flow tables, both reactively and proactively
- Matching starts at the first flow table and may continue to additional flow tables of the pipeline
- Instructions associated with each flow entry either contain actions or modify pipeline processing



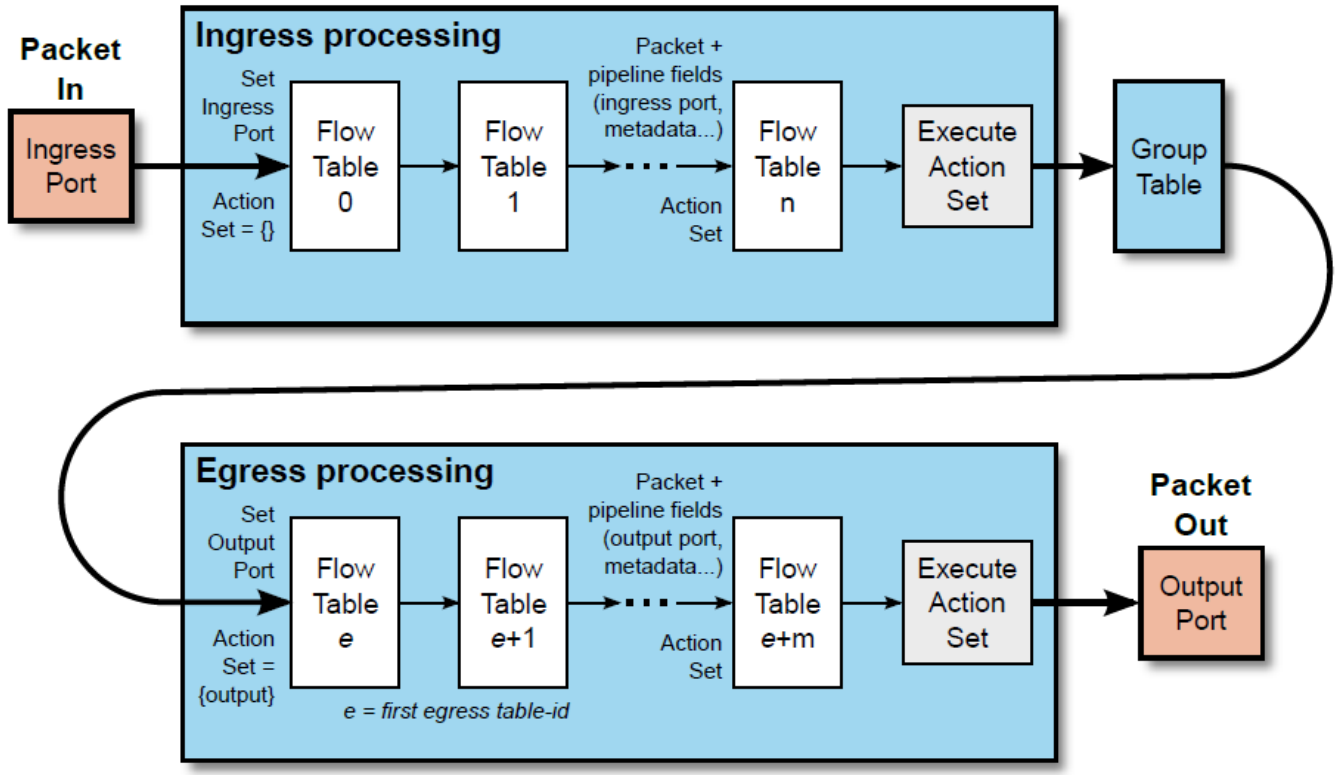
OpenFlow – Overview

- Flow entries may forward to a physical, logical, or reserved port
- Actions associated with flow entries may direct packets to a group for additional processing
- The group table contains group entries; each group entry contains a list of action buckets with specific semantics dependent on group type (e.g., multicast, flooding, multipath).
- Switch designers are free to implement the internals in any way convenient, provided that correct match and instruction semantics are preserved



OpenFlow 1.5 – Pipeline processing

- The OpenFlow pipeline is an abstraction that is mapped to the switch hardware
- A flow table entry is identified by its match fields and priority
- A flow entry instruction may contain actions to be performed on the packet at some point of the pipeline
- Packet match fields used for table lookups depend on the packet type

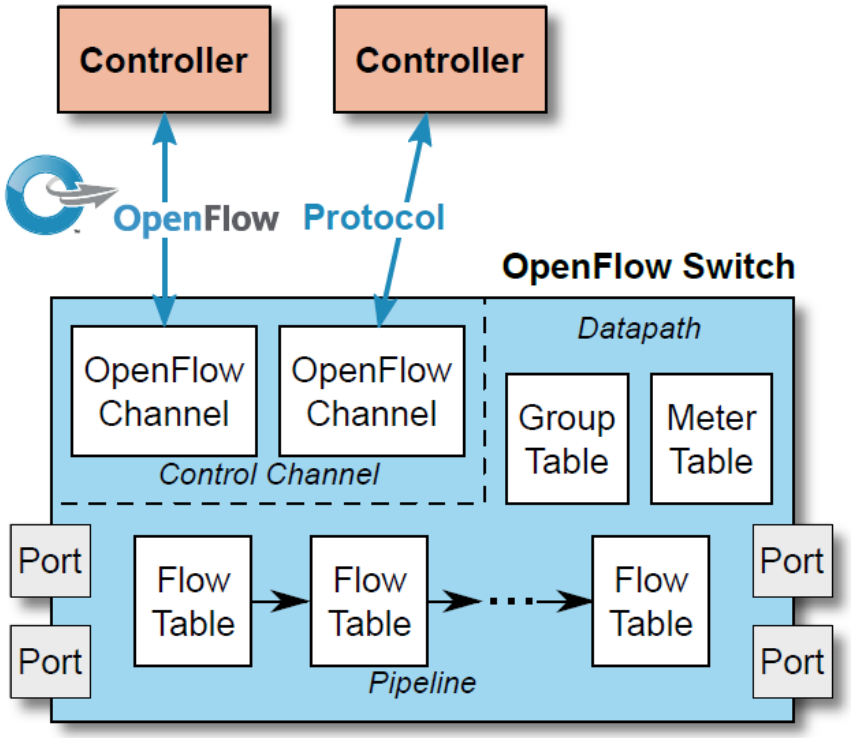


- Every flow table must support a table-miss flow entry to process table misses
- *Ingress processing* happens when the packet enters the OpenFlow switch
- *Egress processing* is the processing that happens after the determination of the output port (output port context)

OpenFlow 1.5 Specification

OpenFlow Switch Architecture

- Now we are going to focus on each of the elements of the OF 1.5 architecture
  - Ports
  - Flow Tables
  - OpenFlow Channel
  - Group Table
  - Meter Table
  - OpenFlow Protocol



OpenFlow 1.5 – Ports

- OpenFlow ports are the network interfaces for passing packets between OpenFlow processing and the rest of the network
- The set of OpenFlow ports may not be identical to the set of network interfaces provided by the switch hardware, some network interfaces may be disabled for OpenFlow, and the OpenFlow switch may define additional OpenFlow ports.
- OpenFlow packets are received on an ingress port and processed by the OpenFlow pipeline which may forward them to an output port
  - OpenFlow physical ports are switch ports that correspond to a hardware interface. OpenFlow switch may be virtualised. In those cases, an OpenFlow physical port may represent a virtual slice of the hardware interface
  - OpenFlow logical ports are switch defined ports that don't correspond directly to a hardware interface of the switch. They are higher level abstractions that may be defined in the switch using non-OpenFlow methods (e.g. link aggregation, tunnels) that must be transparent to OpenFlow processing
  - OpenFlow reserved ports specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as "normal" switch processing

OpenFlow 1.5 – Ports

- OpenFlow logical ports can be used to insert a network service or complex processing in the switch
  - Packet recirculation via logical ports is optional, and OpenFlow supports multiple types of port recirculation
    - A packet is sent on a logical port and returns back via the same logical port (unidirectional processing)
    - A packet is sent on a logical port and returns back via another logical port. This could be used to represent tunnel endpoints (port pairs) or bidirectional packet processing
  - A switch should protect itself from packet infinite loops when using port recirculation

OpenFlow 1.5 – Ports

- Reserved Ports
  - Required:
    - ALL
    - CONTROLLER
    - TABLE
    - IN_PORT
    - ANY
    - UNSET
  - Optional
    - LOCAL
    - NORMAL
    - FLOOD

OF-only and OF-Hybrid Switches

- Two classes of OF switches
  - OF-Only: Only support OF pipeline processing
    - Do not support NORMAL and FLOOD reserved ports
  - OF-Hybrid: Can use ports for OF and ports for normal switching
    - classification mechanism outside of Open- Flow that routes traffic to *either* the OpenFlow pipeline *or* the normal pipeline
      - E.g., VLAN, input port

What does a Port look like

- Extracted from OF SW

```
>>dpctl show tcp:192.168.20.3:6633
features_reply (xid=0xd03029cb): ver:0x1, dpid:2320e7afd5
n_tables:2, n_buffers:256
features: capabilities:0xc7, actions:0xeff
 1(wlan0): addr:00:1c:10:44:32:ca, config: 0, state:0
 2(eth0.0): addr:00:1c:10:44:32:c8, config: 0, state:0
     current:    100MB-FD COPPER AUTO_NEG
     advertised: 10MB-HD 10MB-FD 100MB-HD 100MB-FD AUTO_NEG AUTO_PAUSE AUTO_PAUSE_ASYM
     supported:  10MB-HD 10MB-FD 100MB-HD 100MB-FD AUTO_NEG
 3(eth0.1): addr:00:1c:10:44:32:c8, config: 0, state:0
     current:    100MB-FD COPPER AUTO_NEG
     advertised: 10MB-HD 10MB-FD 100MB-HD 100MB-FD AUTO_NEG AUTO_PAUSE AUTO_PAUSE_ASYM
     supported:  10MB-HD 10MB-FD 100MB-HD 100MB-FD AUTO_NEG
 4(eth0.2): addr:00:1c:10:44:32:c8, config: 0, state:0
     current:    100MB-FD COPPER AUTO_NEG
     advertised: 10MB-HD 10MB-FD 100MB-HD 100MB-FD AUTO_NEG AUTO_PAUSE AUTO_PAUSE_ASYM
     supported:  10MB-HD 10MB-FD 100MB-HD 100MB-FD AUTO_NEG
 5(eth0.3): addr:00:1c:10:44:32:c8, config: 0, state:0
     current:    100MB-FD COPPER AUTO_NEG
     advertised: 10MB-HD 10MB-FD 100MB-HD 100MB-FD AUTO_NEG AUTO_PAUSE AUTO_PAUSE_ASYM
     supported:  10MB-HD 10MB-FD 100MB-HD 100MB-FD AUTO_NEG
get_config_reply (xid=0x3f4af6c): miss_send_len=128
```

OpenFlow support of IEEE 802 Interfaces

- Current specification (OF 1.5) defines three kind of port properties: Ethernet, Optical and Experimental (!wlan).

```
/* Optical port description property. */
struct ofp_port_desc_prop_optical {
        uint16_t type; /* OFPPDPT_3OPTICAL. */
        uint16_t length; /* Length in bytes of this property. */
        uint8_t pad[4]; /* Align to 64 bits. */
        uint32_t supported; /* Features supported by the port. */
        uint32_t tx_min_freq_lmda; /* Minimum TX Frequency/Wavelength */
        uint32_t tx_max_freq_lmda; /* Maximum TX Frequency/Wavelength */
        uint32_t tx_grid_freq_lmda; /* TX Grid Spacing Frequency/Wavelength */
        uint32_t rx_min_freq_lmda; /* Minimum RX Frequency/Wavelength */
        uint32_t rx_max_freq_lmda; /* Maximum RX Frequency/Wavelength */
        uint32_t rx_grid_freq_lmda; /* RX Grid Spacing Frequency/Wavelength */
        uint16_t tx_pwr_min; /* Minimum TX power */
        uint16_t tx_pwr_max; /* Maximum TX power */
};
```

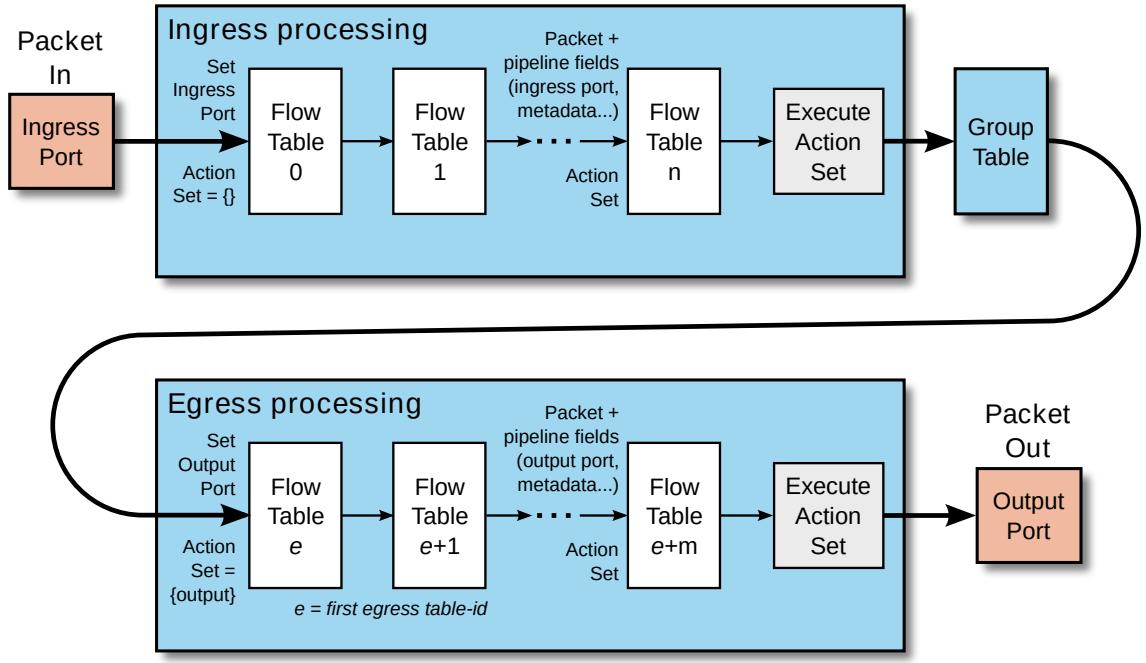
- Optical properties recently added (OF 1.4)
- It allows the controller to configure physical properties of the port, such as Tx power or wavelength used
- It can also monitor this properties
- Other actions such as turning iface on/off are also supported

Port Changes

- A controller can change/modify or add a port at any moment
  - This can be done using OF-Config (we will see it)
- The switch may modify port status due to some events, such as link down
  - Need to report to the Controller
- Port modification/addition or removal does not change the content of flow tables, ports are still referenced
  - Packet forwarded to a removed port are discarded
  - This will lead to problems if a new port is added and the port number is reused

Flow Tables

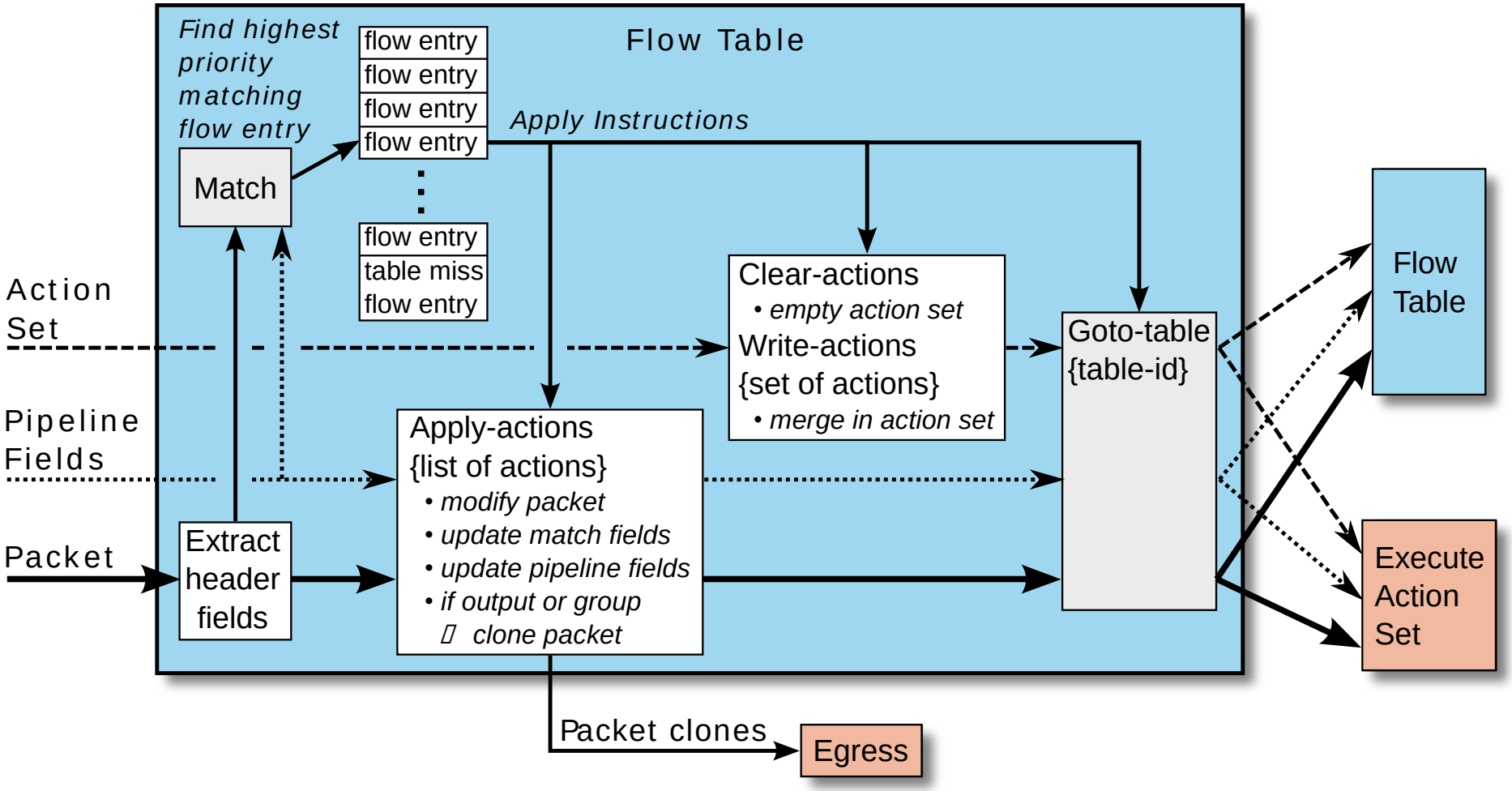
- Flow tables are divided in ingress and egress
- Minimum configuration is 1 ingress table
- All tables have a monotony increasing ID
- Forwarding can only be done to a table with a higher ID
- Group Table is processed after the last of the ingress tables



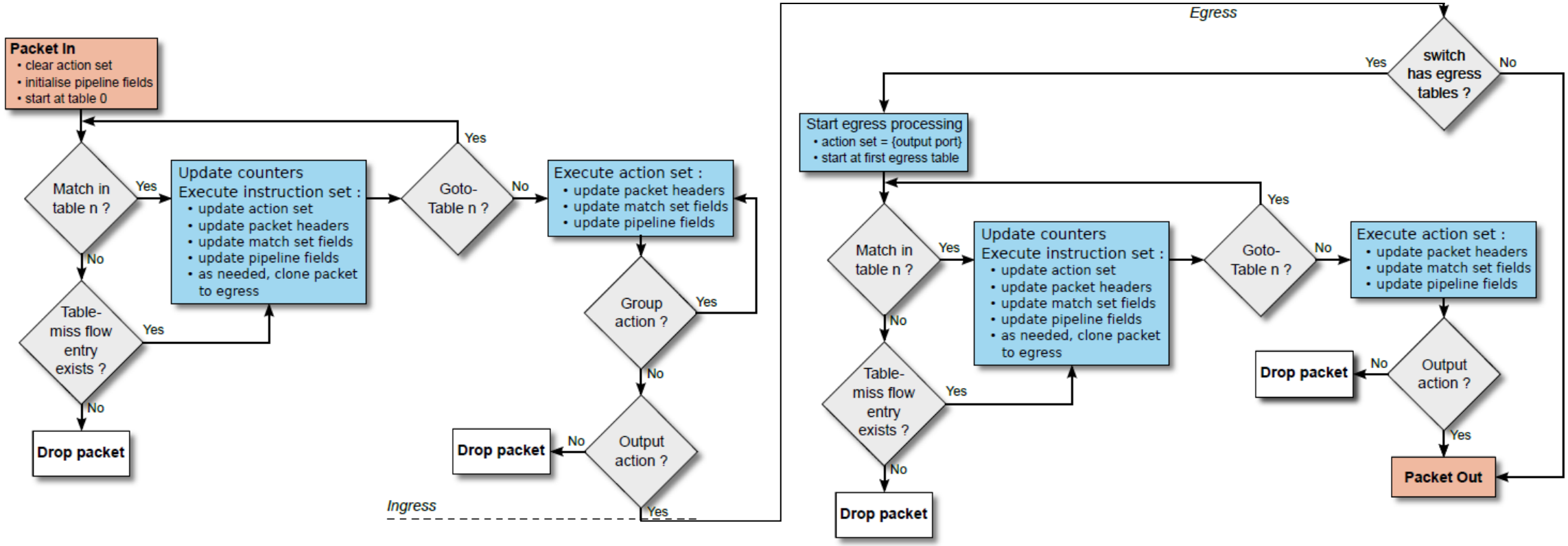
Flow Tables

- All processing starts at table 0, first of the ingress
- If the result of processing a flow is to forward it to an output port, egress processing is performed in the context of the output port
  - Metadata property (output port)
  - If no egress table is configured the packet is processed by the port itself, usually forwarding it

Matching and instruction execution



OpenFlow 1.5 – Pipeline processing



Pipeline processing

- A packet is matched against the flow entries in the first table.
- If a flow entry is found, the instruction set associated is executed.
- This may redirect the flow to another table, where the process is repeated again.
- If a flow entry does not redirect the flow to another flow table, the pipeline processing is stopped, the actions are executed and the packet is usually forwarded.

Pipeline processing

- When a packet is not matched to any flow entry it is a TABLE MISS
  - Instructions in the TABLE MISS entry indicate what to do with the packet
    - Drop it
    - Send it to another table
    - Send it to the controller through a specific message (packet-in)
  - If no TABLE MISS entry is found, the packet is dropped

Pipeline implementation

- The OpenFlow pipeline is an abstraction that is mapped to the actual hardware of the switch
- The hardware typically will not correspond to the OpenFlow pipeline

Flow Entries

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

- A flow entry consist of:
  - match fields: to match against packets. These consist of the ingress port and packet headers, and optionally other pipeline fields such as metadata specified by a previous table.
  - priority: matching precedence of the flow entry.
  - counters: updated when packets are matched.
  - instructions: to modify the action set or pipeline processing.
  - timeouts: maximum amount of time or idle time before flow is expired by the switch.
  - cookie: opaque data value chosen by the controller. May be used by the controller to filter flow entries affected by flow statistics, flow modification and flow deletion requests. Not used when processing packets.
  - flags: flags alter the way flow entries are managed, for example the flag OFPFF_SEND_FLOW_REM triggers flow removed messages for that flow entry.

Flow Entries

- A flow table entry is identified by its match fields and priority: the match fields and priority taken together identify a unique flow entry in a specific flow table.
- The flow entry that wildcards all fields (all fields omitted) and has priority equal to 0 is called the table-miss flow entry.
- A flow entry instruction may contain actions to be performed on the packet
  - E.g., *set-field* action may specify some header fields to rewrite.
- Not all actions are supported by all switches and all tables, mechanisms for capability discovery exist.

Instructions

- Instructions result in changes to the packet, action set and/or pipeline processing
- Instructions are not actions, define the actions to be added to the action set
- Required instructions:
  - Write-Actions *action(s)*
  - Goto-Table *next-table-id*
- Optional instructions:
  - Apply-Actions *action(s)*
  - Clear-Actions
  - Write-Metadata *metadata / mask*
  - Stat-Trigger *stat thresholds*

Instructions

- The instruction set associated with a flow entry contains a maximum of one instruction of each type
- *Clear-Actions* instruction is executed before the *Write-Actions* instruction, the *Apply-Actions* is executed before the *Write-Metadata* and *Goto-Table* is executed last
- A switch must reject a flow entry if it is unable to execute the instructions or part of the instructions associated with the flow entry.
  - the switch must return the error message associated with the issue

Action list

- Required
  - Output *port no*
  - Group *group id*
  - Drop
- Optional
  - Set-Queue *queue id*
  - Meter *meter id*
  - Push-Tag/Pop-Tag *ethertype*
  - Set-Field *field type value*
  - Copy-Field *src field type dst field type*
  - Change-TTL *ttl*

Action execution

- Two ways of executing actions
  - Delayed execution: Action set
  - Immediate execution: Apply Action instruction (and Packet-out)
- Action set
  - Associated with each packet, initially is empty
  - Modified by *Write-Action* instruction or a *Clear-Action* instruction associated with a particular match.
  - Carried between flow tables.
  - Executed when pipeline processing stops (no Goto-Table instruction)

Action Execution

- Action set
  - Only one action of each type is allowed
  - One Set-field action for each field type
  - Copy-field has race problems
  - If an action of a type is already present, it is overwritten
  - If multiple actions of the same type are required, use Apply Action instruction (e.g., multiple VLAN)
  - Egress Action Set: output and group actions are not allowed (output port already set by output action at ingress)

Action Execution

- Action set
  - Strict execution order defined
    - copy TTL inwards: apply copy TTL inward actions to the packet
    - pop: apply all tag pop actions to the packet
    - push-MPLS: apply MPLS tag push action to the packet
    - push-PBB: apply PBB tag push action to the packet
    - push-VLAN: apply VLAN tag push action to the packet
    - copy TTL outwards: apply copy TTL outwards action to the packet
    - decrement TTL: apply decrement TTL action to the packet
    - set: apply all set-field actions to the packet
    - qos: apply all QoS actions, such as meter and set queue to the packet
    - group: if a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list
    - output: if no group action is specified, forward the packet on the port specified by the output action

Action Execution

- Action set:
  - The *output* action in the action set is executed last.
  - If output and group are present, output is ignored.
  - If the *output* action references the *ALL* reserved port, a clone (copy) of the packet start egress processing for each relevant port
- Apply-Action instruction (and Packet-out)
  - Include a list of actions
  - The actions of a list of actions are executed in the order specified by the list, and are applied immediately to the packet
  - Actions are executed on the packet in sequence, the effect of those actions is cumulative

Action Execution

- Apply-Action instruction (Packet-out)
  - After the execution of the list of actions in an *Apply-Actions* instruction, pipeline execution continues on the modified packet. The action set of the packet is unchanged by the execution of the list of actions
  - Can be used to apply multiple actions to the packet immediately, and continue its processing

Actions

- Output *port no*. The Output action forwards a packet to a specified OpenFlow port where it starts egress processing.
- Group *group id*. Process the packet through the specified group.
- Drop. There is no explicit action to represent drops. Instead, packets whose action sets have no output action and no group action must be dropped.
- Set-Queue *queue id*. The set-queue action sets the queue id for a packet. When the packet is forwarded to a port using the output action, the queue id determines which queue attached to this port is used for scheduling and forwarding the packet. Forwarding behavior is dictated by the configuration of the queue and is used to provide basic Quality-of-Service (QoS) support.

Actions

- Meter *meter id*. Direct packet to the specified meter. As the result of the metering, the packet may be dropped (depending on meter configuration and state). If the switch supports meters, it must support this action as the first action in an *list of actions*, for backward compatibility with earlier versions of the specification.

Actions

- Push-Tag/Pop-Tag *ethertype*. Switches may support the ability to push/pop tags.
  - Newly pushed tags should *always* be inserted as the outermost tag in the outermost valid location for that tag
  - When multiple push actions are added to the action set of the packet, they apply to the packet in the order defined by the action set rules, first MPLS, then PBB, then VLAN
  - When a Push operation is performed the fields in the new header are copied from the existing header. If some field does not exist, it is initialized to 0.
  - Do not forget to use set-field action to initialize header fields on Push operations

Actions

- Push/PoP Tags

Action	Associated Data	Description
Push VLAN header	Ethertype	Push a new VLAN header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8100 and 0x88a8 should be used.
Pop VLAN header	-	Pop the outer-most VLAN header from the packet.
Push MPLS header	Ethertype	Push a new MPLS shim header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8847 and 0x8848 should be used.
Pop MPLS header	Ethertype	Pop the outer-most MPLS tag or shim header from the packet. The Ethertype is used as the Ethertype for the resulting packet (Ethertype for the MPLS payload).
Push PBB header	Ethertype	Push a new PBB service instance header (I-TAG TCI) onto the packet (see 7.2.6.6). The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x88E7 should be used.
Pop PBB header	-	Pop the outer-most PBB service instance header (I-TAG TCI) from the packet (see 7.2.6.6).

Actions

- Set-Field *field type value*. The various Set-Field actions are identified by their field type and modify the values of respective header fields in the packet.
- Copy-Field *src field type dst field type*. Copy data between any header or pipeline fields.
  - It is typically used to copy data from a header field to a packet register pipeline field or from a packet register pipeline field to a header field, and in some cases from a header field to another header field.

Actions

- Change-TTL *ttl*. The various Change-TTL actions modify the values of the IPv4 TTL, IPv6 Hop Limit or MPLS TTL in the packet.
  - Change-TTL actions should *always* be applied to the outermost-possible header.

Counters

- Counters are maintained for each flow table, flow entry, port, queue, group, group bucket, meter and meter band.
- OpenFlow-compliant counters may be implemented in software and maintained by polling hardware counters with more limited ranges.
- Duration refers to the amount of time the flow entry, a port, a group, a queue or a meter has been installed in the switch, and must be tracked with second precision.
- The Receive Errors field is the total of all receive and collision errors
- Counters are unsigned and wrap around with no overflow indicator. If a specific numeric counter is not available in the switch, its value must be set to the maximum field value

Group Table

- A group table consists of group entries.
- Each group entry is identified by its group identifier and contains:
  - group identifier: a 32 bit unsigned integer uniquely identifying the group on the OpenFlow switch.
  - group type: to determine group semantics.
  - counters: updated when packets are processed by a group.
  - action buckets: an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters. The actions in a bucket are always applied as an action set.
- Groups are used to provide higher flexibility for Forwarding

Group Types

- Required:
  - Indirect: Execute the only bucket existing. It is equivalent to an all group with one bucket. Used for example to provide next hops in routing tables.
  - All: Execute all buckets in the group. This group is used for multicast or broadcast forwarding. The packet is effectively cloned for each bucket; one packet is processed for each bucket of the group.
- Optional:
  - Select: Execute one bucket in the group. Packets are processed by a single bucket in the group, based on a switch-computed selection algorithm
  - Fast failover: Execute the first live bucket. Each action bucket is associated with a specific port and/or group that controls its liveness.

Meter Table

- A meter table consists of meter entries, defining per-flow meters.
- Per-flow meters enable OpenFlow to implement various simple QoS operations, such as rate-limiting, and can be combined with per-port queues
- Meters are attached directly to flow entries (as opposed to queues which are attached to ports).
- A flow is sent to the meter before the Action Set is executed, so the packet can be dropped before output

Meters

- A meter is assigned to a flow through the Meter optional Action
- A meter is defined by:
  - meter identifier: a 32 bit unsigned integer uniquely identifying the meter
  - meter bands: an unordered list of meter bands, where each meter band specifies the rate of the band and the way to process the packet
  - counters: updated when packets are processed by a meter

Meter Band

- Each meter may have one or more meter bands.
- Each band specifies the rate at which the band applies and the way packets should be processed.
- The meter applies the meter band with the highest configured rate that is lower than the current measured rate.
- If the current rate is lower than any specified meter band rate, no meter band is applied.

Meter Bands

- Each band is defined by:
  - band type: defines how packet are processed
  - rate: used by the meter to select the meter band, defines the lowest rate at which the band can apply
  - burst: defines the granularity of the meter band
  - counters: updated when packets are processed by a meter band
  - type specific arguments: some band types have optional arguments

Band Type	Rate	Burst	Counters	Type specific arguments

Meter Bands

```
/* Common header for all meter bands */
struct ofp_meter_band_header {
    uint16_t        type;     /* One of OFPMBT_*. */
    uint16_t        len;      /* Length in bytes of this band. */
    uint32_t        rate;     /* Rate for this band. */
    uint32_t        burst_size; /* Size of bursts. */
};
```

Band types

- Two band types are defined and are optional:
  - drop: drop (discard) the packet. Can be used to define a rate limiter band.
  - dscp remark: increase the drop precedence of the DSCP field in the IP header of the packet. Can be used to define a simple DiffServ policer.

Differences between ingress and egress processing

- One main difference
- The Action Set on Egress includes an output action with destination the output port defined at ingress processing
- Ingress tables cannot direct flows to egress tables through Goto-Table instruction, only through output action
- Egress tables cannot direct flows to ingress tables
- No output action nor group action can be used can be introduced in the action set, although changes of port can be done through Apply-action instruction

Matching

- OpenFlow 1.2 introduced the Extensible Matching of headers (OXM)
- Prior releases used a static fixed structure to specify matches
- This prevented flexible expression of matches and inclusion of new matches (e.g., no IPv6 support in OF 1.0)
- List of features for OpenFlow Extensible Match :
  - Flexible and compact TLV structure called OXM
  - Enable flexible expression of match, and flexible bitmasking
  - Pre-requisite system to insure consistency of match
  - Give every match field a unique type, remove overloading
  - Modify VLAN matching to be more flexible
  - Add vendor classes and experimenter matches
  - Allow switches to override match requirements

Required Basic Matches

Field		Description
OXM_OF_IN_PORT	*Required in ingress*	Ingress port. This may be a physical or logical port.
OXM_OF_ACTSET_OUTPUT	*Required in egress*	Egress port from action set.
OXM_OF_ETH_DST	*Required*	Ethernet destination address. Can use arbitrary bitmask
OXM_OF_ETH_SRC	*Required*	Ethernet source address. Can use arbitrary bitmask
OXM_OF_ETH_TYPE	*Required*	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_IP_PROTO	*Required*	IPv4 or IPv6 protocol number
OXM_OF_IPV4_SRC	*Required*	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	*Required*	IPv4 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_SRC	*Required*	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	*Required*	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_TCP_SRC	*Required*	TCP source port
OXM_OF_TCP_DST	*Required*	TCP destination port
OXM_OF_UDP_SRC	*Required*	UDP source port
OXM_OF_UDP_DST	*Required*	UDP destination port

Basic matches

```
/* OXM Flow match field types for OpenFlow basic class. */
enum oxm_ofb_match_fields {
    OFPXMT_OFB_IN_PORT        = 0,  /* Switch input port. */
    OFPXMT_OFB_IN_PHY_PORT    = 1,  /* Switch physical input port. */
    OFPXMT_OFB_METADATA       = 2,  /* Metadata passed between tables. */
    OFPXMT_OFB_ETH_DST        = 3,  /* Ethernet destination address. */
    OFPXMT_OFB_ETH_SRC        = 4,  /* Ethernet source address. */
    OFPXMT_OFB_ETH_TYPE       = 5,  /* Ethernet frame type. */
    OFPXMT_OFB_VLAN_VID       = 6,  /* VLAN id. */
    OFPXMT_OFB_VLAN_PCP       = 7,  /* VLAN priority. */
    OFPXMT_OFB_IP_DSCP        = 8,  /* IP DSCP (6 bits in ToS field). */
    OFPXMT_OFB_IP_ECN         = 9,  /* IP ECN (2 bits in ToS field). */
    OFPXMT_OFB_IP_PROTO       = 10, /* IP protocol. */
    OFPXMT_OFB_IPV4_SRC       = 11, /* IPv4 source address. */
    OFPXMT_OFB_IPV4_DST       = 12, /* IPv4 destination address. */
    OFPXMT_OFB_TCP_SRC        = 13, /* TCP source port. */
    OFPXMT_OFB_TCP_DST        = 14, /* TCP destination port. */
    OFPXMT_OFB_UDP_SRC        = 15, /* UDP source port. */
    OFPXMT_OFB_UDP_DST        = 16, /* UDP destination port. */
    OFPXMT_OFB_SCTP_SRC       = 17, /* SCTP source port. */
    OFPXMT_OFB_SCTP_DST       = 18, /* SCTP destination port. */
    OFPXMT_OFB_ICMPV4_TYPE    = 19, /* ICMP type. */
    OFPXMT_OFB_ICMPV4_CODE    = 20, /* ICMP code. */
    OFPXMT_OFB_ARP_OP         = 21, /* ARP opcode. */
    OFPXMT_OFB_ARP_SPA        = 22, /* ARP source IPv4 address. */
    OFPXMT_OFB_ARP_TPA        = 23, /* ARP target IPv4 address. */
    OFPXMT_OFB_ARP_SHA        = 24, /* ARP source hardware address. */
```

Basic matches

```
OFPXMT_OFB_ARP_THA        = 25, /* ARP target hardware address. */
OFPXMT_OFB_IPV6_SRC       = 26, /* IPv6 source address. */
OFPXMT_OFB_IPV6_DST       = 27, /* IPv6 destination address. */
OFPXMT_OFB_IPV6_FLABEL    = 28, /* IPv6 Flow Label */
OFPXMT_OFB_ICMPV6_TYPE    = 29, /* ICMPv6 type. */
OFPXMT_OFB_ICMPV6_CODE    = 30, /* ICMPv6 code. */
OFPXMT_OFB_IPV6_ND_TARGET = 31, /* Target address for ND. */
OFPXMT_OFB_IPV6_ND_SLL    = 32, /* Source link-layer for ND. */
OFPXMT_OFB_IPV6_ND_TLL    = 33, /* Target link-layer for ND. */
OFPXMT_OFB_MPLS_LABEL     = 34, /* MPLS label. */
OFPXMT_OFB_MPLS_TC        = 35, /* MPLS TC. */
OFPXMT_OFP_MPLS_BOS       = 36, /* MPLS BoS bit. */
OFPXMT_OFB_PBB_ISID       = 37, /* PBB I-SID. */
OFPXMT_OFB_TUNNEL_ID      = 38, /* Logical Port Metadata. */
OFPXMT_OFB_IPV6_EXTHDR    = 39, /* IPv6 Extension Header pseudo-field */
OFPXMT_OFB_PBB_UCA        = 41, /* PBB UCA header field. */
OFPXMT_OFB_TCP_FLAGS      = 42, /* TCP flags. */
OFPXMT_OFB_ACTSET_OUTPUT  = 43, /* Output port from action set metadata. */
OFPXMT_OFB_PACKET_TYPE    = 44, /* Packet type value. */
```

Packet type matches

- Most of matches on headers start with a packet type match indicating the valid header fields to match

namespace	**ns_type**	**Match description**	**Packet-in and packet-out format**
0	0	Ethernet packet (default type).	Ethernet header and Ethernet payload.
1	0x800	IPv4 packet (with no header in front).	IPv4 header and IPv4 payload.
1	0x86dd	IPv6 packet (with no header in front).	IPv6 header and IPv6 payload.
0	1	No packet (for example circuit switch).	Empty.
0	0xFFFF	Experimenter defined.	Experimenter defined.

OpenFlow Controller Connection

The OpenFlow Channel

- The OpenFlow channel is used to exchange OpenFlow message between an OpenFlow switch and an OpenFlow controller.
- A typical OpenFlow controller manages multiple OpenFlow channels, each one to a different OpenFlow switch.
- An OpenFlow switch may have one OpenFlow channel to a single controller, or multiple channels for reliability, each to a different controller or to the same controller

The OpenFlow Channel

- Two ways of setting up the connection
  - Out of band: connection through a separated dedicated network.
    - For out-of-band connections, the switch must make sure that traffic to and from the OpenFlow channel is not run through the OpenFlow pipeline.
  - In band: uses the network managed by the OF switch
    - For in-band connections, the switch must set up the proper set of flow entries for the connection in the OpenFlow pipeline.
- The only requirement is that it should provide TCP/IP connectivity.

The OpenFlow Channel

- The OpenFlow channel is usually instantiated as a single network connection between the switch and the controller, using TLS or plain TCP.
- Alternatively, the OpenFlow channel may be composed of multiple network connections to exploit parallelism.
- The OpenFlow switch must be able to create an OpenFlow channel by initiating a connection to an OpenFlow controller.
- Some switch implementations may optionally allow an OpenFlow controller to connect to the OpenFlow switch, in this case the switch usually should restrict itself to secured connections to prevent unauthorised connections.

The OpenFlow Channel

- Connection URI:
  - *protocol*:*name-or-address*:*port*
  - *protocol*:*name-or-address*
- Acceptable values for *protocol* are tls or tcp for main connections.
- Acceptable values for auxiliary connections are tls, dtls, tcp, or udp.
- Port by default is 6653

The OpenFlow Channel

- Initializing the connection
  - Exchange of HELLO messages
  - Negotiation of OF version (based on bitmap carried by HELLO messages)
  - This establishes the connection, next step: Feature request→DataPath ID of the switch, main identifier
- Maintainance of the connection
  - Uses TCP timeouts and TLS session timeouts
  - Also uses Echo Request/Reply messages

The OpenFlow Channel

- OF Channel interruption:
  - When communication with all controllers fails, the switch goes into:
    - fail secure mode: the only change to switch behavior is that packets and messages destined to the controllers are dropped. Flow entries should continue to expire according to their timeouts.
    - fail standalone mode: the switch processes all packets using the OFPP_NORMAL reserved port. The switch is free to use flow tables in any way it wishes, the switch may delete, add or modify any flow entry. This mode is usually only available on Hybrid switches

The OpenFlow Channel

- When OF Channel reestablishment:
- flow entries present in the flow tables at that time are preserved and normal OpenFlow operation resumes.
- the controller has then the option of:
  - reading all flow entries with a *flow-stats* request
  - to re-synchronise its state with the switch state
  - deleting all flow entries with a *flow-mod* request, to start from a clean state on the switch

The OpenFlow Channel

- Using TLS:
  - The switch and controller mutually authenticate by exchanging certificates signed by a site-specific private key
  - One certificate for authenticating the controller (controller certificate)
  - One certification for authenticating to the controller (switch certificate)
- Plain TCP can be used for non- encrypted communication or to implement an alternate encryption configuration, such as using IPsec or VPN

Messages supported by the OpenFlow Channel

Messages

- The OpenFlow switch protocol supports three message types:
  - *Controller-to-switch*: initiated by the controller and used to directly manage or inspect the state of the switch
  - *Asynchronous:* initiated by the switch and used to update the controller of network events and changes to the switch state
  - *Symmetric:* initiated by either the switch or the controller and sent without solicitation

Controller to switch messages

- Packet-out: These are used by the controller to send packets out of a specified port on the switch, and to forward packets received via Packet-in messages. Packet-out messages must contain a full packet or a buffer ID referencing a packet stored in the switch. The message must also contain a list of actions to be applied in the order they are specified; an empty list of actions drops the packet.
  - OFPT_PACKET_OUT

OFPT_PACKET_OUT

```
/* Send packet (controller -> datapath). */
struct ofp_packet_out {
    struct ofp_header header;
    uint32_t buffer_id;           /* ID assigned by datapath (OFP_NO_BUFFER
                                     if none). */
    uint16_t actions_len;         /* Size of action array in bytes. */
    uint8_t pad[2];               /* Align to 64 bits. */
    struct ofp_match match;       /* Packet pipeline fields. Variable size. */
    /* The variable size and padded match is followed by the list of actions. */
    /* struct ofp_action_header actions[0]; *//* Action list - 0 or more. */
    /* The variable size action list is optionally followed by packet data.
     * This data is only present and meaningful if buffer_id == -1. */
    /* uint8_t data[0]; */        /* Packet data.  The length is inferred
                                     from the length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_packet_out) == 24);
```

Asynchronous Messages

- Packet-in: Transfer the control of a packet to the controller. For all packets forwarded to the CONTROLLER reserved port using a flow entry or the table-miss flow entry, a packet-in event is always sent to controllers
  - OFPT_PACKET_IN

```
/* Packet received on port (datapath -> controller). */
struct ofp_packet_in {
    struct ofp_header header;
    uint32_t buffer_id;     /* ID assigned by datapath. */
    uint16_t total_len;     /* Full length of frame. */
    uint8_t reason;         /* Reason packet is being sent (one of OFPR_*) */
    uint8_t table_id;       /* ID of the table that was looked up */
    uint64_t cookie;        /* Cookie of the flow entry that was looked up. */
    struct ofp_match match; /* Packet metadata. Variable size. */
    /* The variable size and padded match is always followed by:
     *   - Exactly 2 all-zero padding bytes, then
     *   - An Ethernet frame whose length is inferred from header.length.
     * The padding bytes preceding the Ethernet frame ensure that the IP
     * header (if any) following the Ethernet header is 32-bit aligned.
     */
    //uint8_t pad[2];       /* Align to 64 bit + 16 bit */
    //uint8_t data[0];      /* Ethernet frame */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 32);
```

Asynchronous Messages

```
/* Packet received on port (datapath -> controller). */
struct ofp_packet_in {
    struct ofp_header header;
    uint32_t buffer_id;     /* ID assigned by datapath. */
    uint16_t total_len;     /* Full length of frame. */
    uint8_t reason;         /* Reason packet is being sent (one of OFPR_*) */
    uint8_t table_id;       /* ID of the table that was looked up */
    uint64_t cookie;        /* Cookie of the flow entry that was looked up. */
    struct ofp_match match; /* Packet metadata. Variable size. */
    /* The variable size and padded match is always followed by:
     *   - Exactly 2 all-zero padding bytes, then
     *   - An Ethernet frame whose length is inferred from header.length.
     * The padding bytes preceding the Ethernet frame ensure that the IP
     * header (if any) following the Ethernet header is 32-bit aligned.
     */
    //uint8_t pad[2];       /* Align to 64 bit + 16 bit */
    //uint8_t data[0];      /* Ethernet frame */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 32);
```