SDN & NFV
OpenFlow and ForCES
IETF-93

Presented by

- **Yaakov Stein** RAD (yaakov_s@rad.com)
- **Evangelos Haleplidis** U of Patras (ehalep@ece.upatras.gr)

Why SDN and NFV ?

Before explaining ***what*** SDN and NFV are
we need to explain ***why*** SDN and NFV are

Its all started with two related trends ...

1. The blurring of the distinction
between ***computation*** and ***communications***
revealing a fundamental disconnect
between ***software*** and ***networking***

2. The decrease in profitability
of ***traditional communications service providers***
along with the increase in profitability
of **Cloud and Over The Top service providers**

The 1st led directly to SDN
and the 2nd to NFV
but today both are intertwined

1. *Computation* and *communications*

Once there was little overlap
between *communications* (telephone, radio, TV)
and *computation* (computers)

Actually communications devices always ran complex algorithms
but these are hidden from the user

But this dichotomy has become blurred

Most home computers are not used for *computation* at all
rather for entertainment and communications (email, chat, VoIP)

Cellular telephones have become computers

The differentiation can still be seen in the terms *algorithm* and *protocol*
Protocol design is fundamentally harder
since there are two interacting entities (the *interoperability* problem)

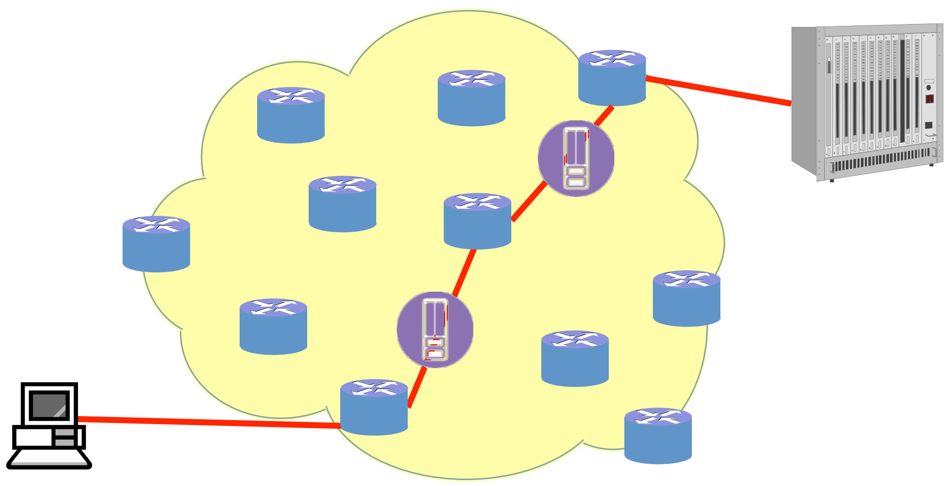
SDN academics claim that packet forwarding is a computation problem
and protocols as we know them should be avoided

1. Rich communications services

Traditional communications services are pure *connectivity* services
 transport data from A to B
  with constraints (e.g., minimum bandwidth, maximal delay)
  with maximal efficiency (minimum cost, maximized revenue)

Modern communications services are richer
 combining connectivity and network functionalities
  e.g., firewall, NAT, load balancing, CDN, parental control, ...

Such services further blur the computation/communications distinction
  and make service deployment optimization more challenging



1. *Software* and *networking speed*

Today, developing a new *iOS/Android* app takes hours to days
but developing a new communications service takes months to years

Even adding new instances of well-known services
is a time consuming process for conventional networks

When a new service types requires new protocols, the timeline is

- protocol standardization (often in more than one SDO)
- hardware development
- interop testing
- vendor marketing campaigns and operator acquisition cycles
- staff training
- deployment

how long has it been since the first IPv6 RFC ?

This leads to a *fundamental disconnect*
between software and networking development timescales

An important goal of SDN and NFV is
to create new network functionalities at the *speed of software*

2. Today's communications world

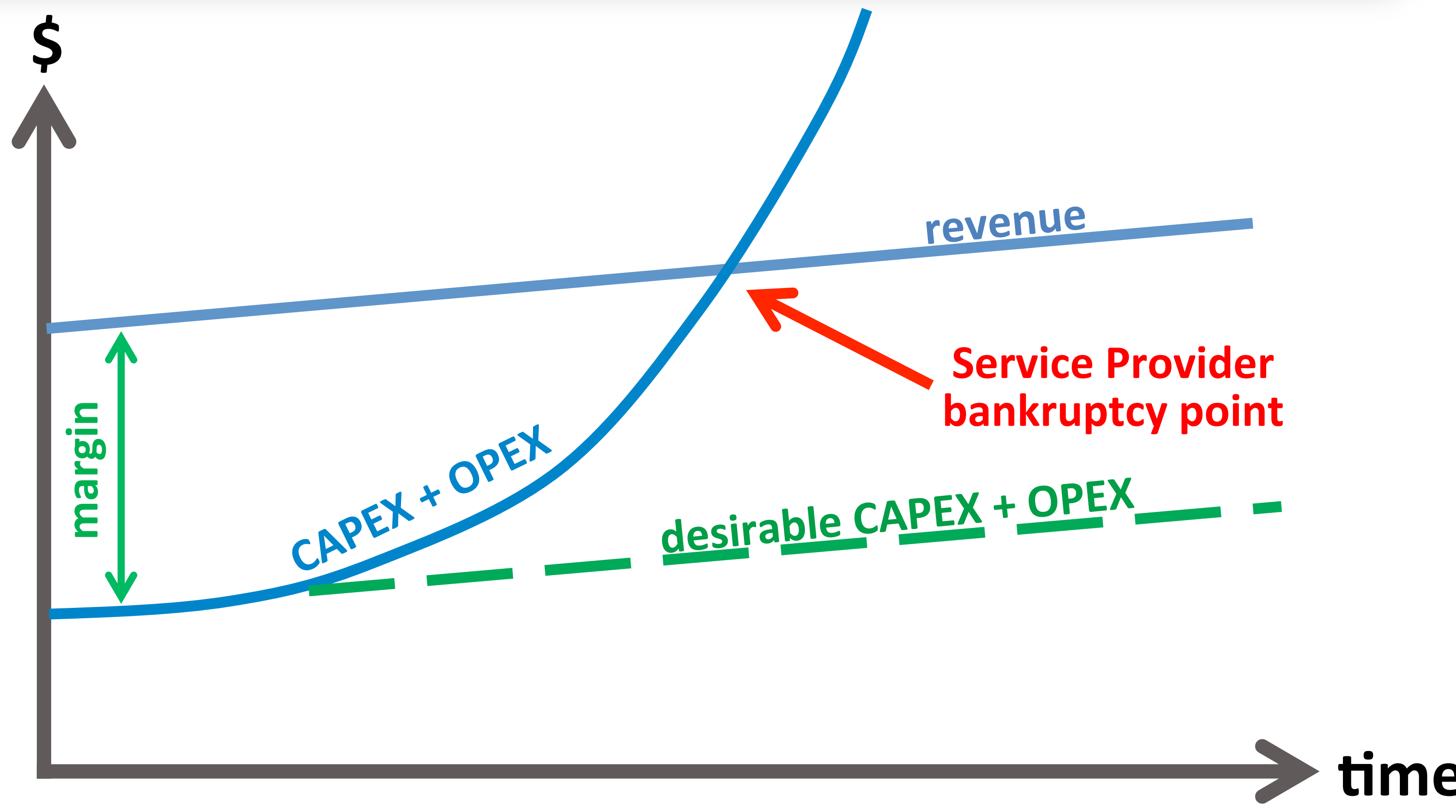
Today's infrastructures are composed of many different Network Elements (NEs)

- sensors, smartphones, notebooks, laptops, desk computers, servers,
- DSL modems, Fiber transceivers,
- SONET/SDH ADMs, OTN switches, ROADMs,
- Ethernet switches, IP routers, MPLS LSRs, BRAS, SGSN/GGSN,
- NATs, Firewalls, IDS, CDN, WAN aceleration, DPI,
- VoIP gateways, IP-PBXes, video streamers,
- performance monitoring probes , performance enhancement middleboxes,
- etc., etc., etc.

New and ever more complex NEs are being invented all the time,
and while equipment vendors like it that way
Service Providers find it hard to shelve and power them all !

In addition, while service innovation is accelerating
the increasing sophistication of new services
the requirement for backward compatibility
and the increasing number of different SDOs, consortia, and industry groups
which means that
it has become very hard to experiment with new networking ideas
NEs are taking longer to standardize, design, acquire, and learn how to operate
NEs are becoming more complex and expensive to maintain

2. The service provider crisis



This is a *qualitative* picture of the service provider's world
Revenue is at best increasing with number of users
Expenses are proportional to bandwidth – doubling every 9 months
This situation obviously can not continue forever !

Two complementary solutions

Software Defined Networks (SDN)

SDN advocates replacing standardized networking protocols
with centralized software applications
that configure all the NEs in the network

Advantages:

- easy to experiment with new ideas
- control software development is much faster than protocol standardization
- centralized control enables stronger optimization
- functionality may be speedily deployed, relocated, and upgraded

Network Functions Virtualization (NFV)

NFV advocates replacing hardware network elements
with software running on COTS computers
that may be housed in POPs and/or datacenters

Advantages:

- COTS server price and availability scales with end-user equipment
- functionality can be located where-ever most effective or inexpensive
- functionalities may be speedily combined, deployed, relocated, and upgraded

SDN

Abstractions

SDN was triggered by the development of networking technologies
not keeping up with the speed of software application development

Computer science theorists theorized
that this derived from not having the required **abstractions**

In CS an *abstraction* is a representation
that reveals semantics needed *at a given level*
while hiding implementation details
thus allowing a programmer to focus on necessary concepts
without getting bogged down in unnecessary details

Programming is fast because programmers exploit abstractions

Example:
It is very slow to code directly in assembly language (with few abstractions, e.g. opcode mnemonics)
It is a bit faster to coding in a low-level language like C (additional abstractions : variables, structures)
It is much faster coding in high-level imperative language like Python
It is much faster yet coding in a declarative language (coding has been abstracted away)
It is fastest coding in a domain-specific language (only contains the needed abstractions)
In contrast, in protocol design we return to *bit level* descriptions every time

Packet forwarding abstraction

The first abstraction relates to how network elements forward packets

At a high enough level of abstraction
all network elements perform the same task

Abstraction 1 *Packet forwarding as a computational problem*
The function of any network element (NE) is to
- receive a packet
- observe packet fields
- apply algorithms (classification, decision logic)
- optionally edit the packet
- forward or discard the packet

For example

- An Ethernet switch observes MAC DA and VLAN tags, performs exact match, forwards the packet
- A router observes IP DA, performs LPM, updates TTL, forwards packet
- A firewall observes multiple fields, performs regular expression match, optionally discards packet

We can replace all of these NEs with a configurable ***whitebox switch***

Network state and graph algorithms

How does a whitebox switch learn its required functionality ?

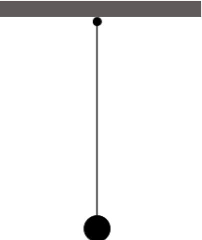
Forwarding decisions are optimal
  when they are based on full global knowledge of the network

With full knowledge of topology and constraints
  the path computation problem can be solved by a graph algorithm

While it may sometimes be possible to perform path computation (e.g., Dijkstra)
  in a distributed manner
It makes more sense to perform them centrally

Abstraction 2 *Routing as a computational problem*
  Replace distributed routing protocols with graph algorithms
    performed at a central location

Note with SDN, the pendulum that swung
  from the completely centralized PSTN
  to the completely distributed Internet
swings back to completely centralized control



Configuring the whitebox switch

How does a whitebox switch acquire the information needed to forward
that has been computed by an omniscient entity at a central location ?

Abstraction 3 *Configuration*
Whitebox switches are directly *configured* by an *SDN controller*

Conventional network elements have two parts:

1. smart but slow CPUs that create a Forwarding Information Base
2. fast but dumb switch fabrics that use the FIB

Whitebox switches only need the dumb part, thus

- eliminating distributed protocols
- not requiring intelligence

The API from the SDN controller down to the whitebox switches
is conventionally called the *southbound API* (e.g., OpenFlow, ForCES)

Note that this SB API is in fact a *protocol*
but is a simple configuration protocol
not a distributed routing protocol

Separation of data and control

You will often hear stated that the *defining attribute* of *SDN* is the separation of the *data* and *control* planes

This separation was not invented recently by SDN academics

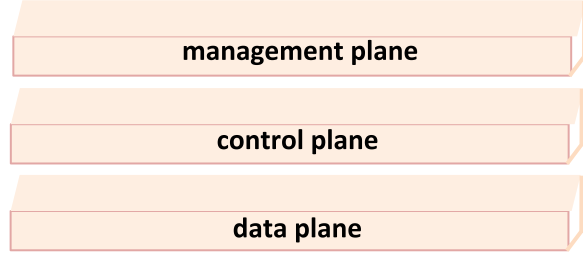
Since the 1980s all well-designed communications systems have enforced logical separation of 3 planes :

- data plane (forwarding)
- control plane (e.g., routing)
- management plane (e.g., policy, commissioning, billing)

What SDN really does is to

1) insist on ***physical*** separation of data and control

2) erase the difference between control and management planes



Control or management

What happened to the management plane ?

Traditionally the distinction between control and management was that :

- management had a human in the loop
- while the control plane was automatic

With the introduction of more sophisticated software
the human could often be removed from the loop

The difference that remains is that

- the management plane is *slow* and *centralized*
- the control plane is *fast* and *distributed*

So, another way of looking at SDN
is to say that it merges
the control plane
into a single centralized management plane

SDN vs. distributed routing

Distributed routing protocols are limited to

- finding simple connectivity
- minimizing number of hops (or other additive cost functions)

but find it hard to perform more sophisticated operations, such as

- guaranteeing isolation (privacy)
- optimizing paths under constraints
- setting up non-overlapping backup paths (the Suurballe problem)
- integrating networking functionalities (e.g., NAT, firewall) into paths

This is why MPLS created the **P**ath **C**omputation **E**lement architecture

An SDN controller is omniscient (the *God box*)
and holds the entire network description as a graph
on which arbitrary optimization calculations can be performed

But centralization comes at a price

- the controller is a single point of failure
  (more generally different CAP-theorem trade-offs are involved)
- the architecture is limited to a single network
- additional (overhead) bandwidth is required
- additional set-up delay may be incurred

Flows

It would be too slow for a whitebox switch
to query the centralized SDN controller
for every packet received

So we identify packets as belonging to **flows**

Abstraction 4 *Flows* (as in Open**Flow**)
Packets are handled solely based on the flow to which they belong

Flows are thus just like **F**orwarding **E**quivalence **C**lasses

Thus a flow may be determined by

- an IP prefix in an IP network
- a label in an MPLS network
- VLANs in VLAN cross-connect networks

The granularity of a flow depends on the application

Control plane abstraction

In the standard SDN architecture, the SDN controller is omniscient
but does not itself *program* the network
since that would limit development of new network functionalities

With software we create building blocks with defined APIs
which are then used, and perhaps inherited and extended, by programmers

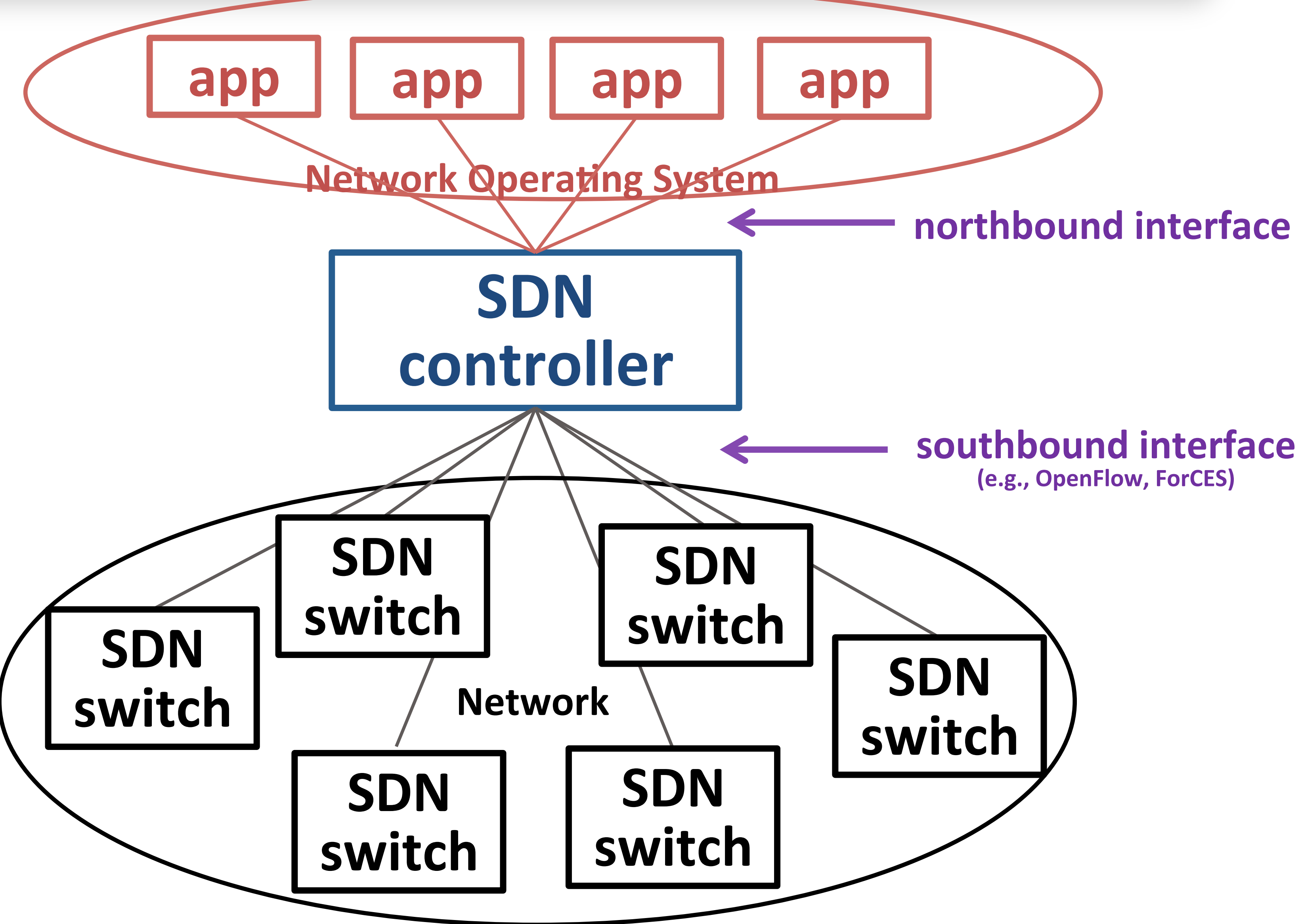
With networking, each network *application* has a tailored-made control plane
with its own element discovery, state distribution, failure recovery, etc.

Note the subtle change of terminology we have just introduced
instead of calling switching, routing, load balancing, etc. network *functions*
we call them network *applications* (similar to software *apps*)

Abstraction 5 Northbound *APIs instead of protocols*
Replace control plane protocols with well-defined APIs to network applications

This abstraction hide details of the network from the network application
revealing high-level concepts, such as requesting connectivity between A and B
but hiding details unimportant to the application
such as details of switches through which the path A → B passes

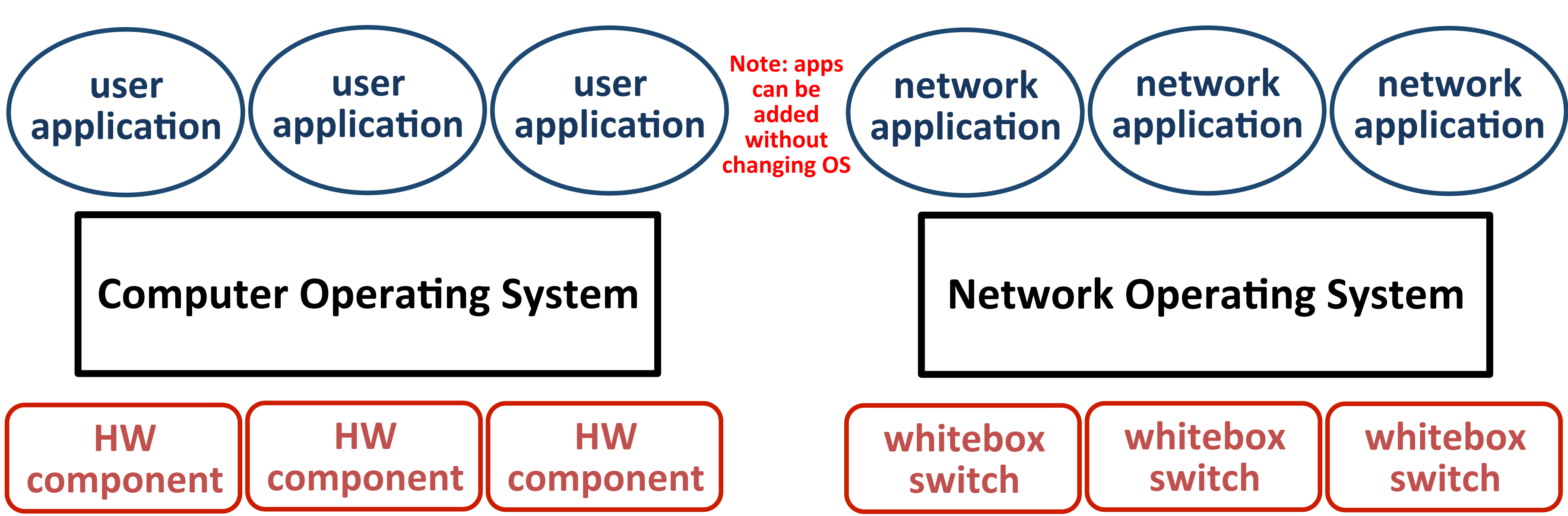
SDN overall architecture



Network Operating System

For example, a computer operating system
- sits between user programs and the physical computer hardware
- reveals high level functions (e.g., allocating a block of memory or writing to disk)
- hides hardware-specific details (e.g., memory chips and disk drives)

We can think of SDN as a **Network Operating System**



SDN overlay model

We have been discussing the purist SDN model
 where SDN builds an entire network using whiteboxes

For non-greenfield cases this model requires
 upgrading (downgrading?) hardware to whitebox switches

An alternative model builds an **SDN overlay network**

The overlay tunnels traffic through the physical network
 running SDN on top of switches that do not explicitly support SDN

Of course you may now need to administer two separate networks

Organizations working on SDN

The IRTF's SDNRG

- see RFC 7426

The Open Networking Forum (ONF)

- responsible for OpenFlow and related work
- promoting SDN principles

ITU-T SG13

- working on architectural issues

and many open source communities, including :

OpenDaylight

ON.Lab

Open Source SDN (OSSDN)

many other controllers

NFV

Virtualization of computation

In the field of computation, there has been a major trend towards ***virtualization***

Virtualization here means the creation of a **virtual machine** (VM)
that acts like an independent physical computer

A **VM** is software that emulates hardware (e.g., an x86 CPU)
over which one can run software as if it is running on a physical computer

The VM runs on a *host* machine
and creates a *guest* machine (e.g., an x86 environment)

A single host computer may host many fully independent guest VMs
and each VM may run different Operating Systems and/or applications

For example

- a datacenter may have many racks of server cards
- each server card may have many (host) CPUs
- each CPU may run many (guest) VMs

A **hypervisor** is software that enables *creation* and *monitoring* of VMs

Network Functions Virtualization

CPUs are not the only hardware device that can be virtualized

Many (but not all) NEs can be replaced by software running on a CPU or VM

This would enable

- using standard COTS hardware (whitebox servers)
  - reducing CAPEX and OPEX
- fully implementing functionality in software
  - reducing development and deployment cycle times, opening up the R&D market
- consolidating equipment types
  - reducing power consumption
- optionally concentrating network functions in datacenters or POPs
  - obtaining further economies of scale. Enabling rapid scale-up and scale-down

For example, switches, routers, NATs, firewalls, IDS, etc.
are all good candidates for virtualization
as long as the data rates are not too high

Physical layer functions (e.g., Software Defined Radio) are not ideal candidates

High data-rate (core) NEs will probably remain in dedicated hardware

Potential VNFs

Potential **V**irtualized **N**etwork **F**unctions

- **forwarding elements**: Ethernet switch, router, Broadband Network Gateway, NAT
- **virtual CPE:** demarcation + network functions + VASes
- **mobile network nodes**: HLR/HSS, MME, SGSN, GGSN/PDN-GW, RNC, NodeB, eNodeB
- **residential nodes**: home router and set-top box functions
- **gateways:** IPSec/SSL VPN gateways, IPv4-IPv6 conversion, tunneling encapsulations
- **traffic analysis**: DPI, QoE measurement
- **QoS**: service assurance, SLA monitoring, test and diagnostics
- **NGN signalling**: SBCs, IMS
- **converged and network-wide functions**: AAA servers, policy control, charging platforms
- **application-level optimization**: CDN, cache server, load balancer, application accelerator
- **security functions**: firewall, virus scanner, IDS/IPS, spam protection

Function relocation

Once a network functionality has been virtualized
it is relatively easy to relocate it

By relocation we mean
placing a function somewhere other than its conventional location
e.g., at **P**oints **o**f **P**resence and **D**ata **C**enters

Many (mistakenly) believe that the main reason for NFV
is to move networking functions to data centers
where one can benefit from economies of scale

Some telecomm functionalities need to reside at their conventional location

- Loopback testing
- E2E performance monitoring

but many don't

- routing and path computation
- billing/charging
- traffic management
- DoS attack blocking

Note: even nonvirtualized functions *can* be relocated

Example of relocation with SDN

SDN is, in fact, a specific example of function relocation

In conventional IP networks routers perform 2 functions

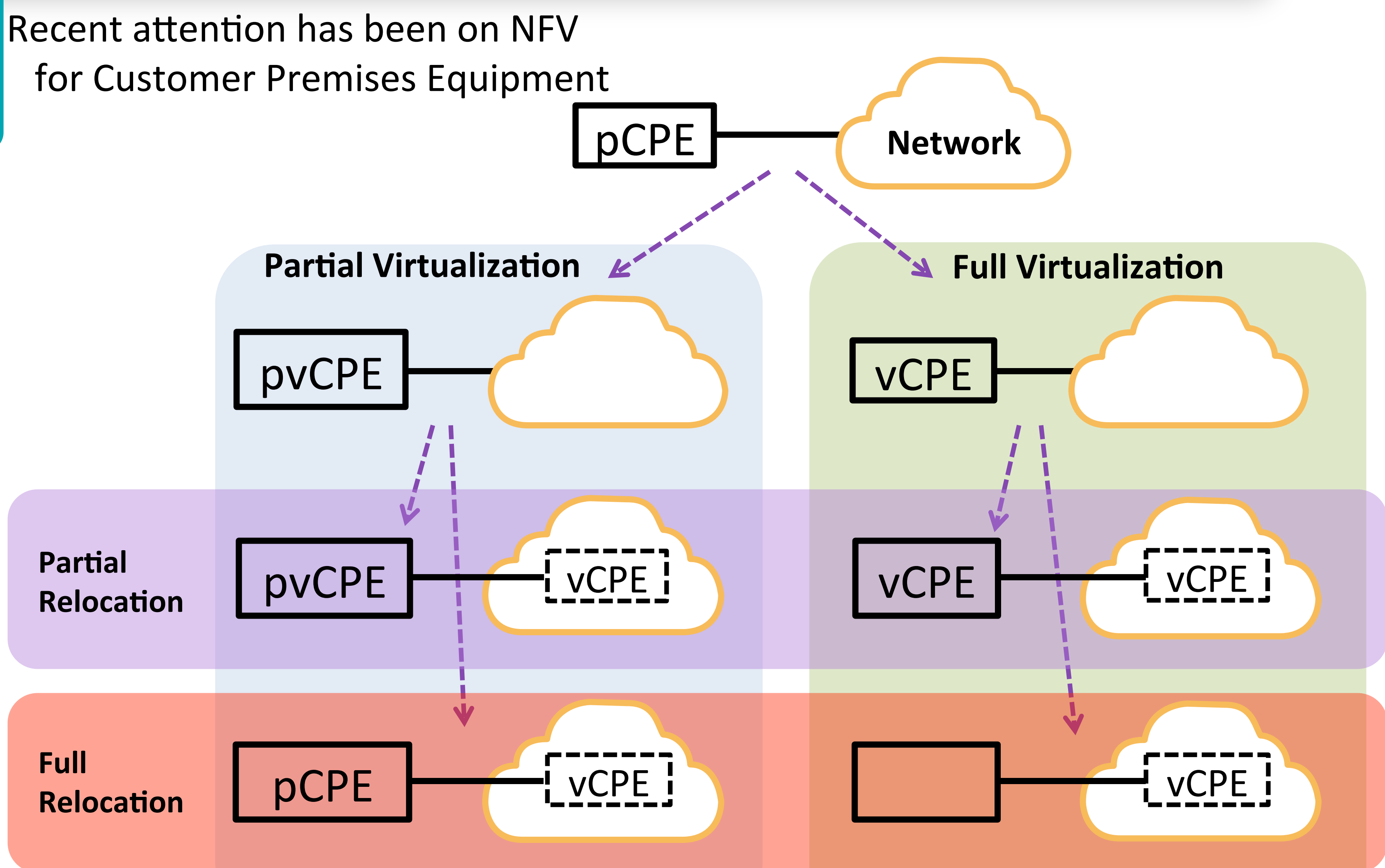
- forwarding
  - observing the packet header
  - consulting the **F**orwarding **I**nformation **B**ase
  - forwarding the packet
- routing
  - communicating with neighboring routers to discover topology (routing protocols)
  - runs routing algorithms (e.g., Dijkstra)
  - populating the FIB used in packet forwarding

SDN enables moving the routing algorithms to a centralized location

- replace the router with a simpler but configurable whitebox switch
- install a centralized SDN controller
  - runs the routing algorithms (internally – w/o on-the-wire protocols)
  - configures the NEs by populating the FIB

Virtualization and Relocation of CPE

Recent attention has been on NFV
for Customer Premises Equipment



Distributed NFV

The idea of optimally placing virtualized network functions in the network
from edge (CPE) through aggregation through PoPs and HQs to datacenters
is called **Distributed-NFV** (DNFV)

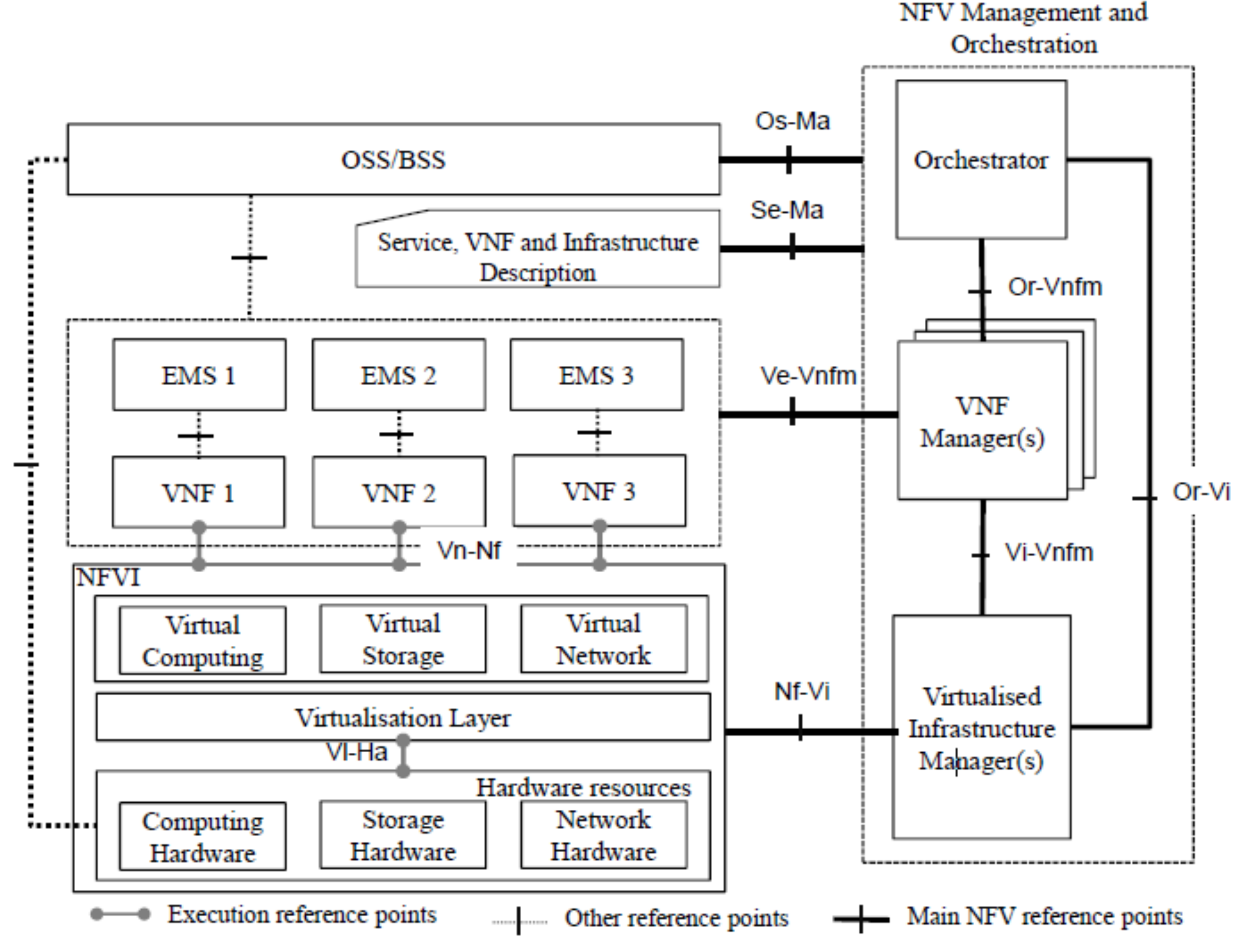
Optimal location of a functionality needs to take into consideration:

- resource availability (computational power, storage, bandwidth)
- *real-estate* availability and costs
- energy and cooling
- management and maintenance
- other economies of scale
- security and privacy
- regulatory issues

For example, consider moving a DPI engine from where it is needed
this requires sending the packets to be inspected to a remote DPI engine

If bandwidth is unavailable or expensive or excessive delay is added
then DPI must not be relocated
even if computational resources are less expensive elsewhere!

ETSI NFV-ISG architecture



MANO ? VIM ? VNFM? NFVO?

Traditional NEs have NMS (EMS) and perhaps are supported by an OSS

NFV has *in addition* the MANO (Management and Orchestration) containing :

- an orchestrator
- VNFM(s) (VNF Manager)
- VIM(s) (Virtual Infrastructure Manager)
- lots of reference points (*interfaces*) !

The VIM (usually OpenStack) manages NFVI resources in one NFVI domain

- life-cycle of virtual resources (e.g., set-up, maintenance, tear-down of VMs)
- inventory of VMs
- FM and PM of hardware and software resources
- exposes APIs to other managers

The VNFM manages VNFs in one VNF domain

- life-cycle of VNFs (e.g., set-up, maintenance, tear-down of VNF instances)
- inventory of VNFs
- FM and PM of VNFs

The NFVO is responsible for resource and service orchestration

- controls NFVI resources everywhere via VIMs
- creates end-to-end services via VNFMs

Organizations working on NFV

ETSI NFV Industry Specification Group (NFV-ISG)

- architecture and MANO
- Proofs of Concept

ETSI Mobile Edge Computing Industry Specification Group (MEC ISG)

- NFV for mobile backhaul networks

Broadband Forum (BBF)

- vCPE for residence and business applications

and many open source communities, including :

Open Platform for NFV (OPNFV)

- open source platform for accelerating NFV deployment

OpenStack – the most popular VIM

Open vSwitch – an open source switch supporting OpenFlow

DPDK, ODP – tools for making NFV more efficient

OpenFlow

What is OpenFlow ?

OpenFlow is an SDN southbound interface –
i.e., a protocol from an SDN controller to an SDN switch (*whitebox*)
that enables configuring forwarding behavior

What makes OpenFlow different from similar protocols is its *switch model*
it assumes that the SDN switch is based on TCAM matcher(s)
so flows are identified by exact match with wildcards on header field
supported header fields include:

- Ethernet - DA, SA, EtherType, VLAN
- MPLS – top label and BoS bit
- IP (v4 or v6) – DA, SA, protocol, DSCP, ECN
- TCP/UDP ports

OpenFlow grew out of *Ethane* and is now developed by the ONF
it has gone through several major versions
the latest is 1.5.0

OpenFlow

The OpenFlow specifications describe

- the southbound protocol between OF controller and OF switches
- the operation of the OF switch

The OpenFlow specifications do not define

- the northbound interface from OF controller to applications
- how to boot the network
- how an E2E path is set up by touching multiple OF switches
- how to configure or maintain an OF switch (which can be done by of-config)

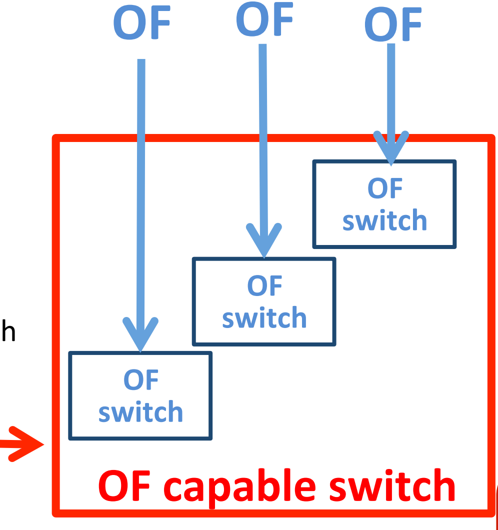
The **OF-CONFIG** specification defines

a configuration and management protocol between *OF configuration point* and *OF capable switch*

- configures which OpenFlow controller(s) to use
- configures queues and ports
- remotely changes port status (e.g., up/down)
- configures certificates
- switch capability discovery
- configuration of tunnel types (IP-in-GRE, VxLAN)

NB for Open vSwitch OVSDB (RFC 7047) can also be used

OF-CONFIG



OF matching

The basic entity in OpenFlow is the ***flow***

A flow is a sequence of packets

that are forwarded through the network in the same way

Packets are classified as belonging to flows

based on **match fields** (switch ingress port, packet headers, metadata)

detailed in a **flow table** (list of match criteria)

Only a finite set of match fields is presently defined

and an even smaller set that must be supported

The matching operation is *exact match*

with certain fields allowing *bit-masking*

Since OF 1.1 the matching proceeds in a **pipeline**

Note: this limited type of matching is too primitive

to support a complete NFV solution

(it is even too primitive to support IP forwarding, let alone NAT, firewall ,or IDS!)

However, the assumption is that DPI is performed by the network application

and all the relevant packets will be easy to match

OF flow table

The flow table is populated by the controller

The incoming packet is matched by comparing to match fields

For simplicity, matching is exact match to a static set of fields

If matched, actions are performed and counters are updated

Entries have priorities and the highest priority match succeeds

Actions include editing, metering, and forwarding

OpenFlow 1.3 basic match fields

- **Switch input port**
- Physical input port
- Metadata

- **Ethernet DA**
- **Ethernet SA**
- **EtherType**
- VLAN id
- VLAN priority

- IP DSCP
- IP ECN
- **IP protocol**
- **IPv4 SA**
- **IPv4 DA**
- IPv6 SA
- IPv6 DA

- **TCP source port**
- **TCP destination port**
- **UDP source port**
- **UDP destination port**
- SCTP source port
- SCTP destination port

- ICMP type
- ICMP code
- ARP opcode
- ARP source IPv4 address
- ARP target IPv4 address
- ARP source HW address
- ARP target HW address

- IPv6 Flow Label
- ICMPv6 type
- ICMPv6 code
- Target address for IPv6 ND
- Source link-layer for ND
- Target link-layer for ND
- IPv6 Extension Header pseudo-field

- MPLS label
- MPLS BoS bit

- PBB I-SID

- Logical Port Metadata (GRE, MPLS, VxLAN)

bold match fields MUST be supported

OpenFlow Switch Operation

There are two different kinds of OpenFlow compliant switches

- OF-only   all forwarding is based on OpenFlow
- OF-hybrid   supports conventional and OpenFlow forwarding

Hybrid switches will use some mechanism (e.g., VLAN ID) to differentiate
between packets to be forwarded by conventional processing
and those that are handled by OF

The switch first has to classify an incoming packet as

- conventional forwarding
- OF protocol packet from controller
- packet to be sent to flow table(s)

OF forwarding is accomplished by a flow table or since 1.1 by flow table**s**
An OpenFlow compliant switch must contain at least one flow table

OF also collects PM statistics (counters)
and has basic rate-limiting (metering) capabilities

An OF switch can not usually react by itself to network events
but there is a *group* mechanism that can be used for limited reactions

Matching fields

An OF flow table can match multiple fields

So a single table may require

ingress port = P *and*
source MAC address = SM *and* destination MAC address = DM *and*
VLAN ID = VID *and* EtherType = ET *and*
source IP address = SI *and* destination IP address = DI *and*
IP protocol number = P *and*
source TCP port = ST *and* destination TCP port = DT

This kind of exact match of many fields is expensive in software
but can readily implemented via TCAMs

OF 1.0 had only a single flow table
which led to overly limited hardware implementations
since practical TCAMs are limited to several thousand entries

OF 1.1 introduced **multiple tables** for scalability

OF 1.1+ flow tables

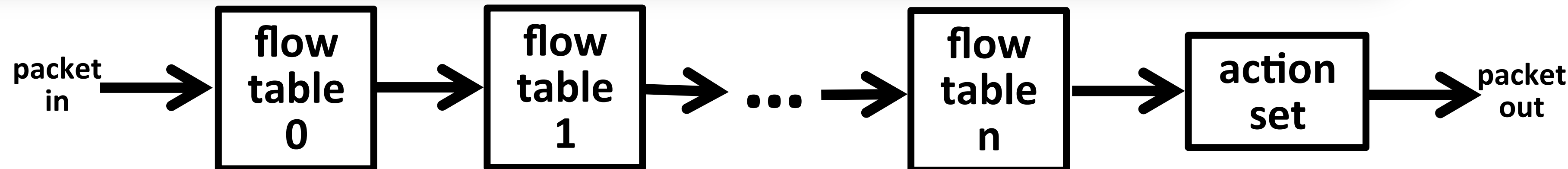


Table matching

- each flow table is ordered by priority
- highest priority match is used (match can be made "negative" using drop action)
- matching is exact match with certain fields allowing bit masking
- table may specify ANY to wildcard the field
- fields matched may have been modified in a previous step

Although the pipeline was introduced mainly for scalability
it gives the matching syntax more expressibility to (although no additional semantics)
In addition to the verbose
if (field1=value1) AND (field2=value2) then …
if (field1=value3) AND (field2=value4) then …
it is now possible to accommodate
if (field1=value1) then if (field2=value2) then …
else if (field2=value4) then …

Unmatched packets

What happens when no match is found in the flow table ?

A flow table *may* contain a flow miss entry
to catch unmatched packets
The flow miss entry must be inserted by the controller just like any other entry
and is defined as wildcard on all fields, and lowest priority

The flow miss entry may be configured to :

- discard packet
- forward to a subsequent table
- forward (OF-encapsulated) packet to controller
- use "normal" (conventional) forwarding (for OF-hybrid switches)

If there is no flow miss entry
the packet is by default discarded
but this behavior may be changed via of-config

OF switch ports

The ports of an OpenFlow switch can be physical or logical

The following ports are defined :

- physical ports (connected to switch hardware interface)
- logical ports connected to tunnels (tunnel ID and physical port are reported to controller)
- ALL output port (packet sent to all ports except input and blocked ports)
- CONTROLLER packet from or to controller
- TABLE represents start of pipeline
- IN_PORT output port which represents the packet's input port
- ANY wildcard port
- LOCAL optional – switch local stack for connection over network
- NORMAL optional port sends packet for conventional processing (hybrid switches only)
- FLOOD output port sends packet for conventional flooding

Instructions

Each flow entry contains an **instruction set** to be executed upon match

Instructions include:

- Metering : rate limit the flow (may result in packet being dropped)
- Apply-Actions : causes actions in *action list* to be executed immediately
  (may result in packet modification)
- Write-Actions / Clear-Actions : changes *action set* associated with packet
  which are performed when pipeline processing is over
- Write-Metadata : writes metadata into metadata field associated with packet
- Goto-Table : indicates the next flow table in the pipeline
  if the match was found in flow table k
  then goto-table m must obey m > k

Actions

OF enables performing actions on packets

- **output** packet to a specified port
- **drop** packet (if no actions are specified)
- apply **group** bucket actions (to be explained later)
- overwrite packet header fields
- copy or decrement TTL value
- push or pop push MPLS label or VLAN tag
- set QoS queue (into which the packet will be placed before forwarding)

mandatory to support (output, drop, group)

optional to support (overwrite, copy/decrement TTL, push/pop, set QoS queue)

Action lists are performed immediately upon match

- actions are accumulatively performed in the order specified in the list
- particular action types may be performed multiple times
- further pipeline processing is on the modified packet

Action sets are performed at the end of pipeline processing

- actions are performed in the order specified in OF specification
- actions can only be performed once

Meters

OF is not very strong in QoS features
  but does have a metering mechanism

A flow entry can specify a **meter**, and the meter measures and limits the
  aggregate rate of all flows to which it is attached

The meter can be used directly for simple rate-limiting (by discarding)
  or can be combined with DSCSP remarking for DiffServ mapping

Each meter can have several **meter bands**
  if the meter rate surpasses a meter band, the configured action takes place
where possible actions are

- drop
- increase DSCP drop precedence

OpenFlow statistics

OF switches maintain **counters** for every
- flow table
- flow entry
- port
- queue
- group
- group bucket
- meter
- meter band

Counters are unsigned integers and wrap around without overflow indication

Counters may count received/transmitted packets, bytes, or durations

See table 5 of the OF specification for the list of mandatory and optional counters

Flow removal and expiry

Flows may be explicitly deleted by the controller at any time

However, flows may be preconfigured with finite lifetimes
and are automatically removed upon expiry

Each flow entry has two timeouts

- hard_timeout : if non-zero, the flow times out after X seconds
- idle_timeout : if non-zero, the flow times out
after not receiving a packet for X seconds

When a flow is removed for any reason,
there is flag which requires the switch to inform the controller

- that the flow has been removed
- the reason for its removal (expiry/delete)
- the lifetime of the flow
- statistics of the flow

Groups

Groups enable performing some set of actions on multiple flows
thus common actions can be modified once, instead of per flow

Groups also enable additional functionalities, such as

- replicating packets for multicast
- load balancing
- protection switch

Group operations are defined in group table

ID	type	counters	action buckets

Group tables provide functionality not available in flow table

While flow tables enable dropping or forwarding to one port
group tables enable (via group *type*) forwarding to :

- a random port from a group of ports (load-balancing)
- the first live port in a group of ports (for failover)
- all ports in a group of ports (packet replicated for multicasting)

Action buckets are triggered by type:

- **All** execute all buckets in group
- **Indirect** execute one defined bucket
- **Select** (optional) execute a bucket (via round-robin, or hash algorithm)
- **Fast failover** (optional) execute the first live bucket

Slicings

Network slicing

A network can be divided into isolated *slices*

each with different behavior

each controlled by different controller

Thus the same switches can treat different packets in completely different ways

(for example, L2 switch some packets, L3 route others)

Bandwidth slicing

OpenFlow supports multiple queues per output port

in order to provide some minimum data bandwidth per flow

This is also called *slicing* since it provides a *slice* of the bandwidth to each queue

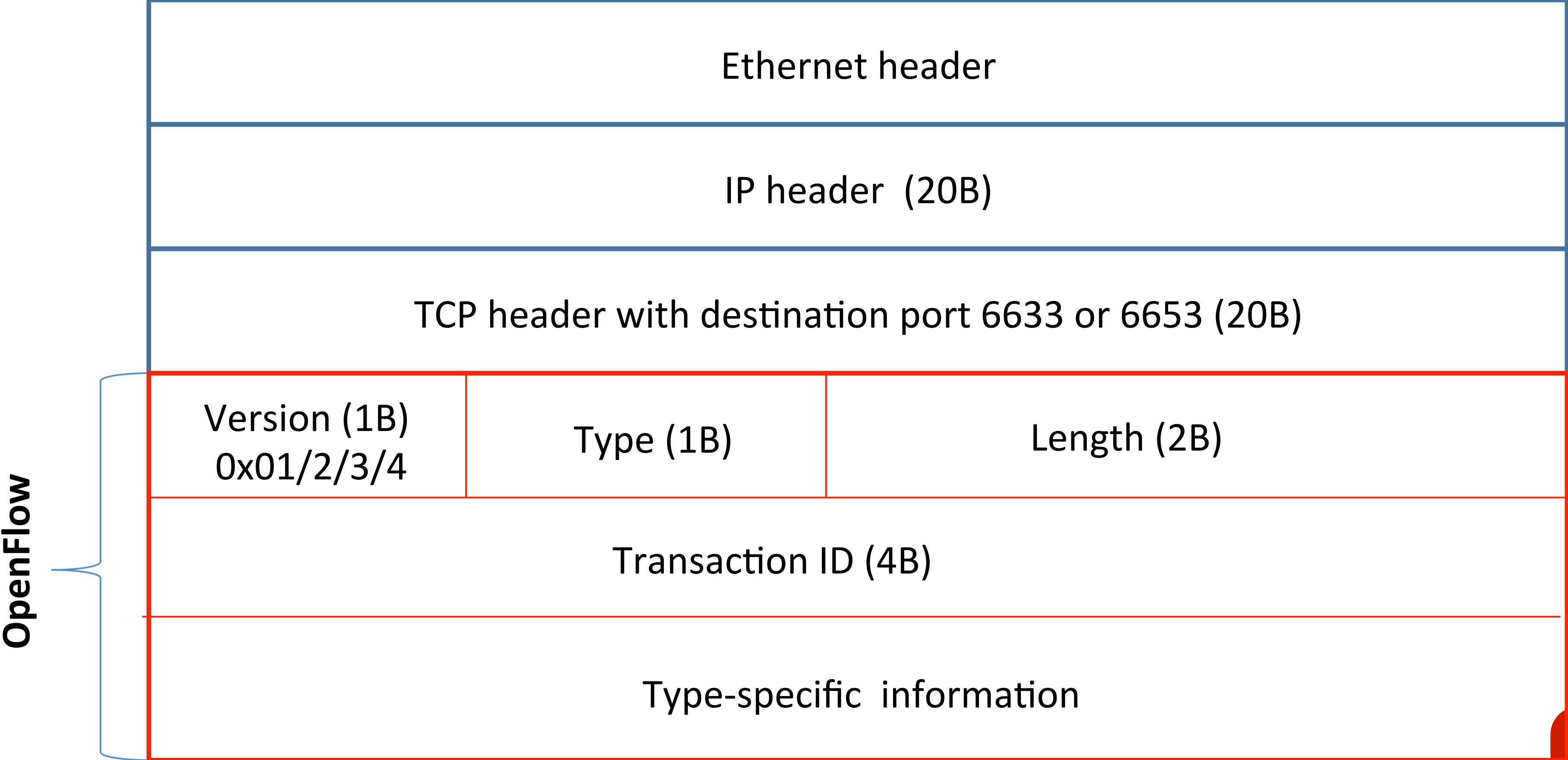
Queues may be configured to have :

- given length
- minimal/maximal bandwidth
- other properties

OpenFlow protocol packet format

OF runs over TCP (optionally SSL for secure operation) using port 6633 and is specified by C **structs**

OF is a very low-level specification (assembly-language-like)



OpenFlow messages

The OF protocol was built to be *minimal* and *powerful*

There are 3 types of OpenFlow messages :

OF controller to switch

- populates flow tables which SDN switch uses to forward
- request statistics

OF switch to controller (asynchronous messages)

- packet/byte counters for defined flows
- sends packets not matching a defined flow

Symmetric messages

- hellos (startup)
- echoes (heartbeats, measure control path latency)
- experimental messages for extensions

OpenFlow message types

Symmetric messages
0 HELLO
1 ERROR
2 ECHO_REQUEST
3 ECHO_REPLY
4 EXPERIMENTER

Switch configuration
5 FEATURES_REQUEST
6 FEATURES_REPLY
7 GET_CONFIG_REQUEST
8 GET_CONFIG_REPLY
9 SET_CONFIG

Asynchronous messages
10 PACKET_IN = 10
11 FLOW_REMOVED = 11
12 PORT_STATUS = 12

Controller command messages
13 PACKET_OUT
14 FLOW_MOD
15 GROUP_MOD
16 PORT_MOD
17 TABLE_MOD

Multipart messages
18 MULTIPART_REQUEST
19 MULTIPART_REPLY

Barrier messages
20 BARRIER_REQUEST
21 BARRIER_REPLY

Queue Configuration messages
22 QUEUE_GET_CONFIG_REQUEST
23 QUEUE_GET_CONFIG_REPLY

Controller role change request messages
24 ROLE_REQUEST
25 ROLE_REPLY

Asynchronous message configuration
26 GET_ASYNC_REQUEST
27 GET_ASYNC_REPLY
28 SET_ASYNC

Meters and rate limiters configuration
29 METER_MOD

Interestingly, OF uses a protocol version and TLVs for extensibility
These are 2 generic control plane mechanisms,
of the type that SDN claims don't exist …

Session setup and maintenance

An OF switch may contain default flow entries to use
before connecting with a controller

The switch will boot into a special failure mode

An OF switch is usually pre-configured with the IP address of a controller

An OF switch may establish communication with multiple controllers in order to improve reliability or scalability; the hand-over is managed by the controllers.

OF is best run over a secure connection (TLS/SSL),
but *can* be run over unprotected TCP

Hello messages are exchanged between switch and controller upon startup
hellos contain version number and optionally other data

Echo_Request and **Echo_reply** are used to verify connection liveliness
and optionally to measure its latency or bandwidth

Experimenter messages are for experimentation with new OF features

If a session is interrupted by connection failure
the OF switch continues operation with the current configuration

Upon re-establishing connection the controller may delete all flow entries

Bootstrapping

How does the OF controller communicate with OF switches
before OF has set up the network ?

The OF specification explicitly avoids this question
- one may assume conventional IP forwarding to pre-exist
- one can use spanning tree algorithm with controller as root,
  once switch discovers controller it sends topology information

How are flows initially configured ?

The specification allows two methods
- proactive (push) flows are set up without first receiving packets
- reactively (pull) flows are only set up after a packet has been received

A network may mix the two methods

Service Providers may prefer proactive configuration
while enterprises may prefer reactive

Barrier message

An OF switch does not explicitly acknowledge message receipt or execution

OF switches may arbitrarily reorder message execution
in order to maximize performance

When the order in which the switch executes messages is important
or an explicit acknowledgement is required
the controller can send a **Barrier_Request** message

Upon receiving a barrier request
the switch must finish processing all previously received messages
before executing any new messages

Once all old messages have been executed
the switch sends a **Barrier_Reply** message back to the controller

ForCES

ForCES History

- FORwarding & Control Element Separation.
  - IETF working group
    - Established in 2001
      - Era of Network Processing Forum (NPF)
      - Need for open and standardized programmable interfaces for off-the-shelf network processor devices[1]
    - Concluded in 2015
  - Set of:
    - Protocols
    - Model

[1]https://datatracker.ietf.org/doc/charter-ietf-forces/03/

ForCES History – Major milestones

Date	RFC	I/PS	Milestone
July 2001			Working group established
Dec 2003	RFC3654	I	Requirements RFC
Apr 2004	RFC3746	I	Framework RFC
Jul 2009	(RFC6053)		1st interoperability test
Mar 2010	RFC5810	PS	ForCES Protocol
Mar 2010	RFC5811	PS	SCTP-TML
Mar 2010	RFC5812	PS	ForCES Model
Feb 2011	(RFC6984)		2nd interoperability test
Jun 2013	RFC6956	PS	LFB library (Data model)
May 2013			Re-chartered
Oct 2014	RFC7391	PS	ForCES Protocol Extension
Nov 2014	RFC7408	PS	ForCES Model Extension
Mar 2015			Working group concluded

ForCES terminology

Protocol (RFC5810)

The ForCES protocol is a master-slave protocol in which FEs are slaves and CEs are masters. Includes both the management of the communication channel and the control messages.

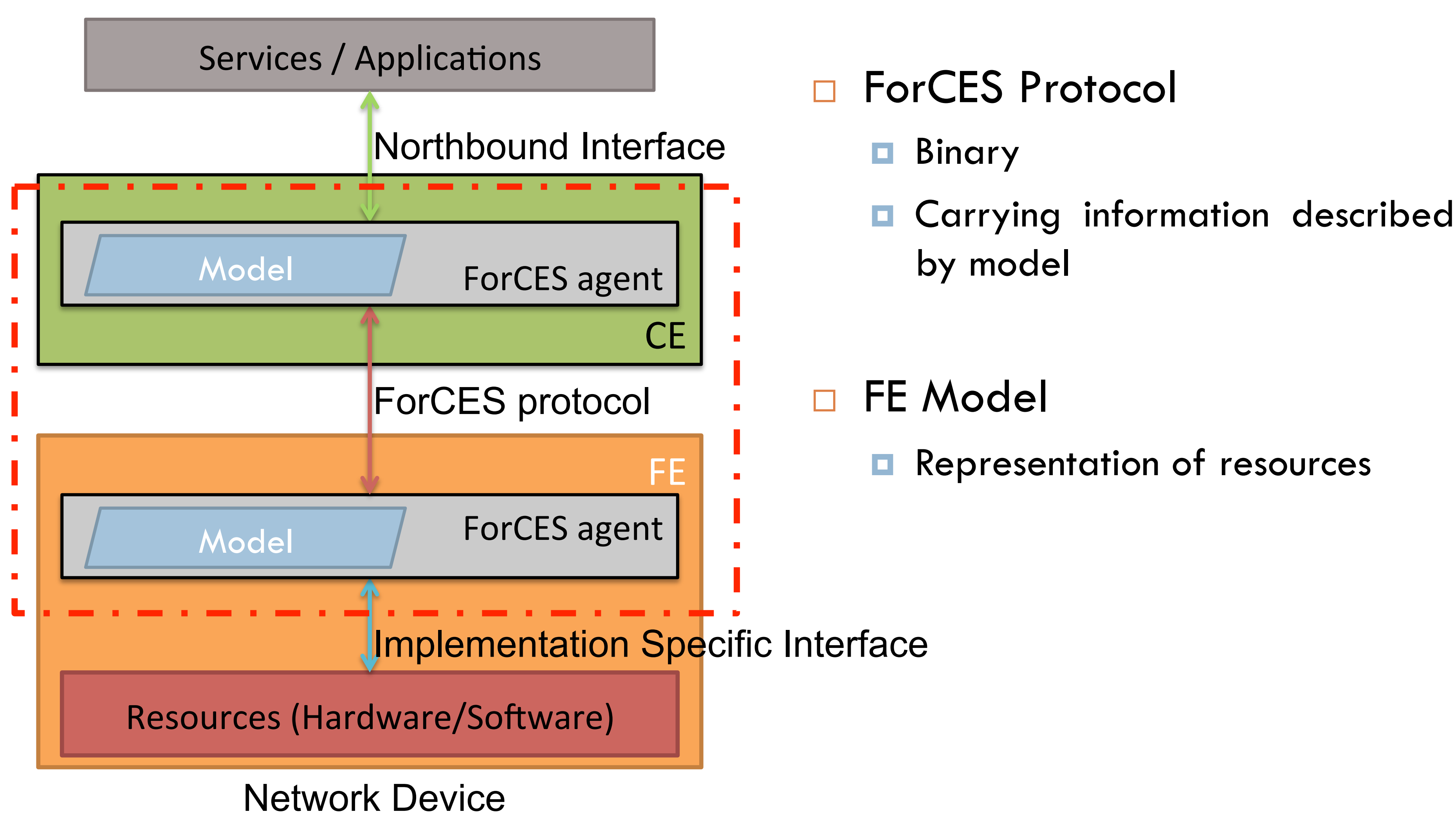
FE Model (RFC5812)

The FE model provides the basis to define the information elements exchanged between the CE and the FE in the ForCES protocol.

The FE model is primarily an information model[2], but includes aspects of a data model.

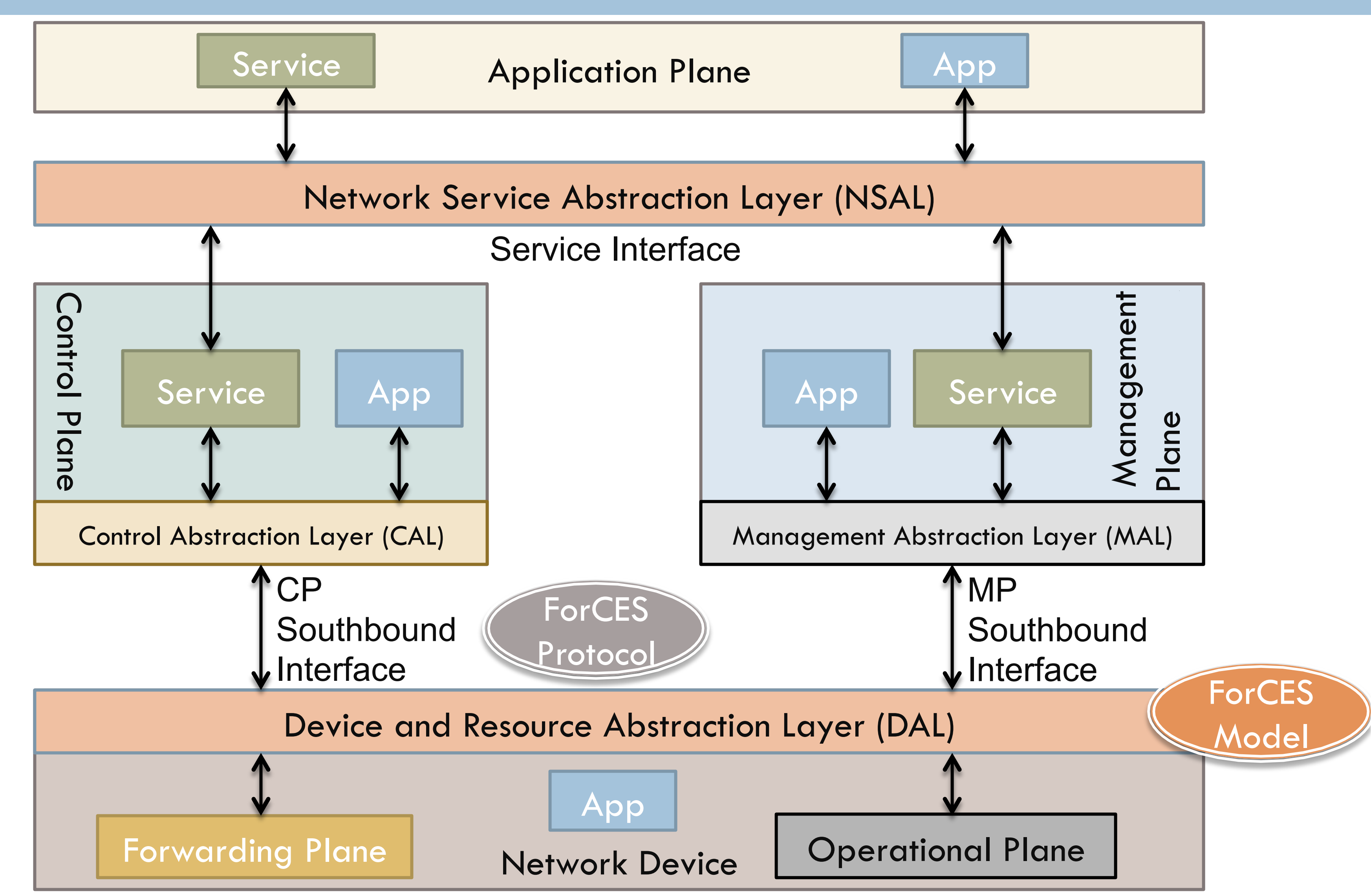
[2]https://tools.ietf.org/html/rfc3444

Conceptual view

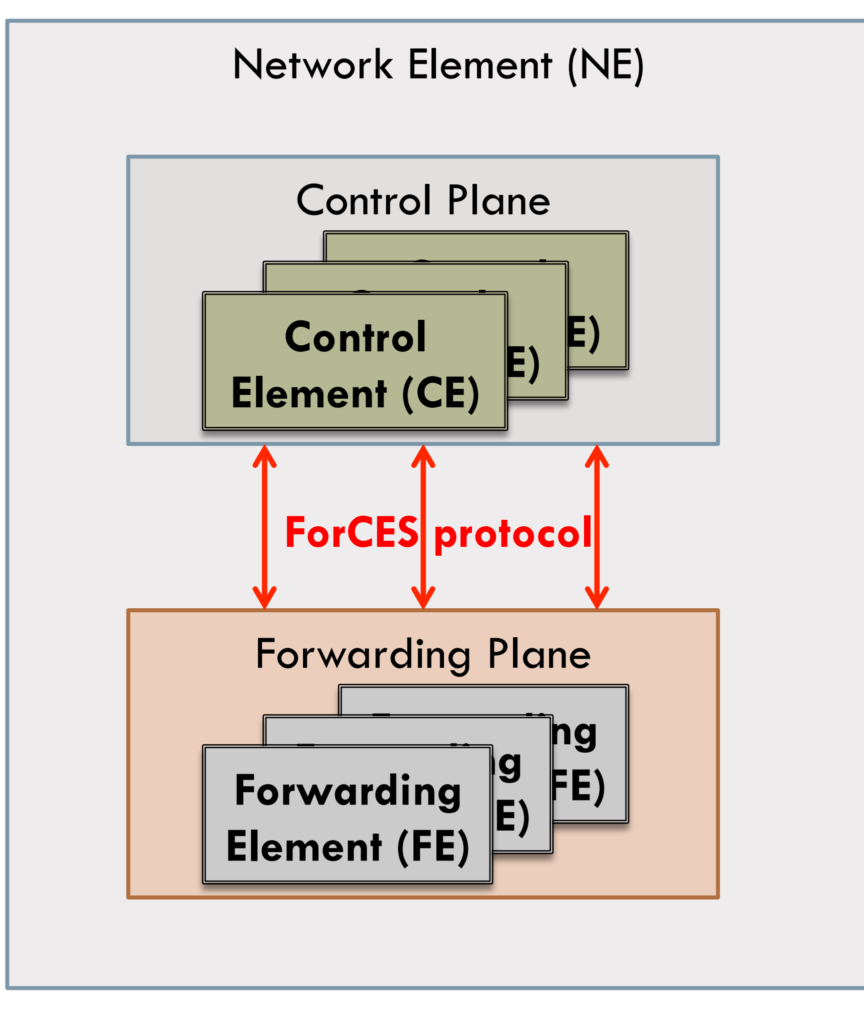


- ForCES Protocol
  - Binary
  - Carrying information described by model
- FE Model
  - Representation of resources

ForCES juxtaposition on SDN (RFC7426)



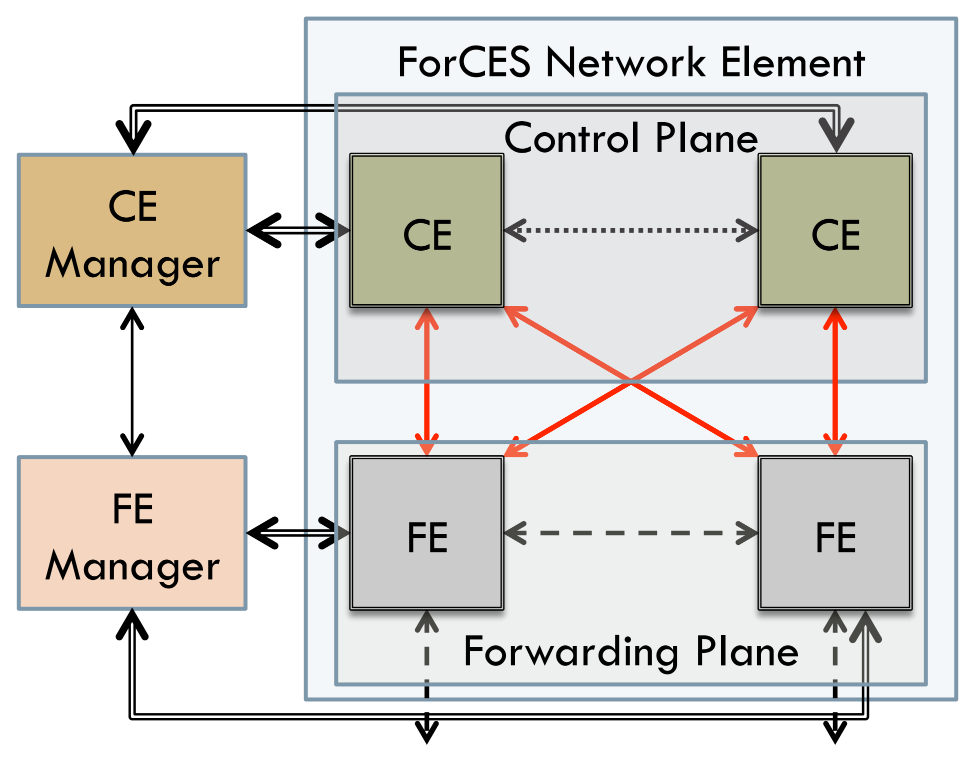
ForCES Framework (RFC3746)



- Network Element (NE)
  - Packet Processing Entity
  - Constitutes of CEs & FEs
  - Multiple CEs to FEs for HA
  - CEs/FEs Physical or Virtual
- NE components distributed
  - Local (within one box)
  - Geographical distributed (LAN/WAN/Internet)

ForCES Framework (RFC3746)

- Managers (CE/FE)
  - Bootstrap and subsidiary mechanisms.
  - CE/FE discovery
  - Determine which CEs/FEs will communicate.
  - FE manager (part) in charter
    - Subsidiary mechanism[3]
  - Could be:
    - A protocol (proprietary/open)
      - E.g. ForCES[3]
    - Simple text file



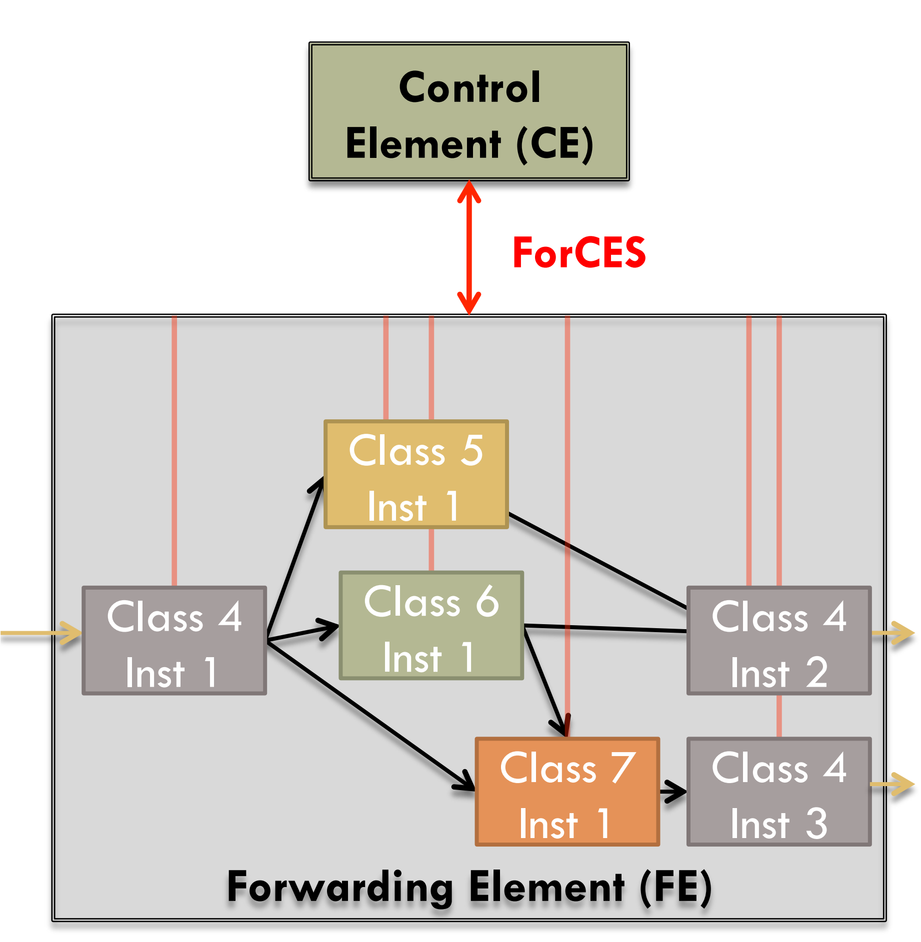
[3]https://www.ietf.org/id/draft-ietf-forces-lfb-subsidiary-management-01.txt (to be published)

ForCES FE Model (RFC5812)

- ForCES FE Model
  - NEs composed of logically separate packet processing elements
  - Model FEs using Logical Functional Blocks (LFBs).
    - Fine (or coarse as needed) grained operations
    - Hardware/Software
    - Physical/Virtual
  - FE – directed graph of LFB class instances
    - Graph can be dynamic if supported by implementation
  - Includes both Capability & State Model
  - XML schema
    - Rules on how to describe LFB model libraries in XML.

ForCES FE Model (Meta Model) ← LFB Model (Model) ← Resource (Implementation)

LFB Model (RFC5812)



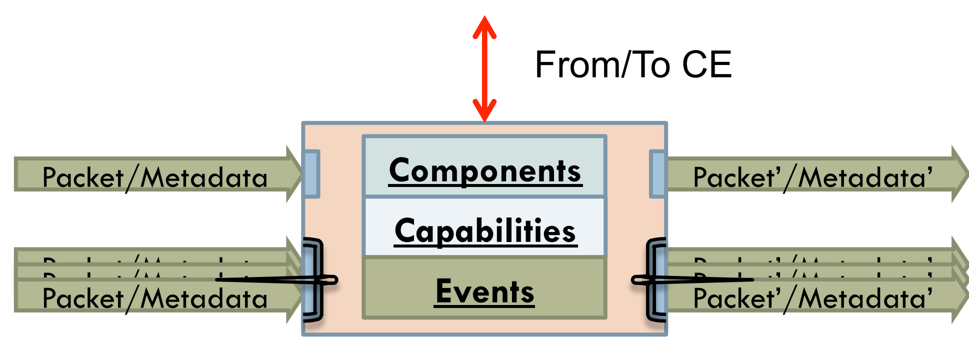
- Written in XML
- Object-oriented approach
  - Model defines LFB Classes
  - Instantiate LFB Instances
- Features
  - LFB Class Versioning
  - LFB Class Inheritance
  - Backwards/Forwards compatibility
- Point of interoperability between implementations
  - Similar to SNMP MIBs

ForCES Model – Core LFBs

- Core LFBs (FE Management as a whole)
  - FEObject LFB (Class ID 1 – RFC5812)
    - Regards capabilities and state of FE e.g.:
      - Instantiated LFBs (Can be used to instantiate new LFBs runtime)
      - LFB Topology (Can be used to change topology runtime)
      - Supported LFBs
  - FEProtocol LFB (Class ID 2 – RFC5810)
    - Regards protocol functionality e.g.:
      - All CEs
      - Heartbeat policies
      - HA policies/capabilities

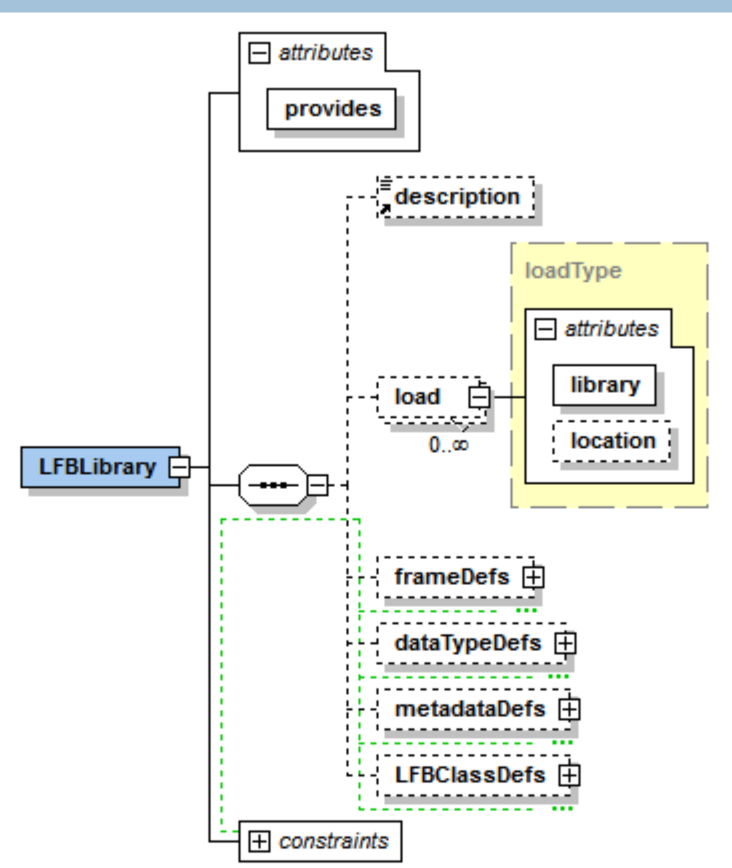
ForCES Model – LFBs (RFC5812)

- Logical Functional Block Class
  - Fine-grained or coarse grained per need.
  - Abstractions:
    - Input / Output Ports
      - **Frame Expected/Frame Produced**
        - Packet
        - Metadata
      - Singleton/Group
    - Components
    - Capabilities
    - Events



ForCES Model – LFB Library (RFC5812)

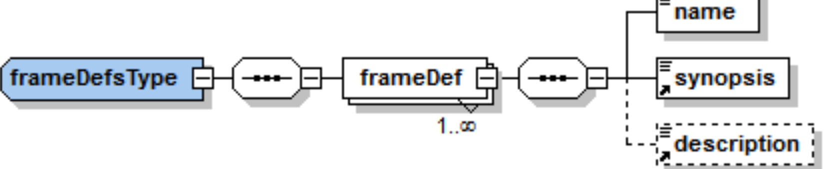
- Sequence of top-level elements
- Optional list (ordered)
  - Description (Description)
  - Load (Imports)
  - Frame Definitions
  - Data Type Definitions
  - Metadata Type Definitions
  - LFB class Definition



ForCES Model – Frames

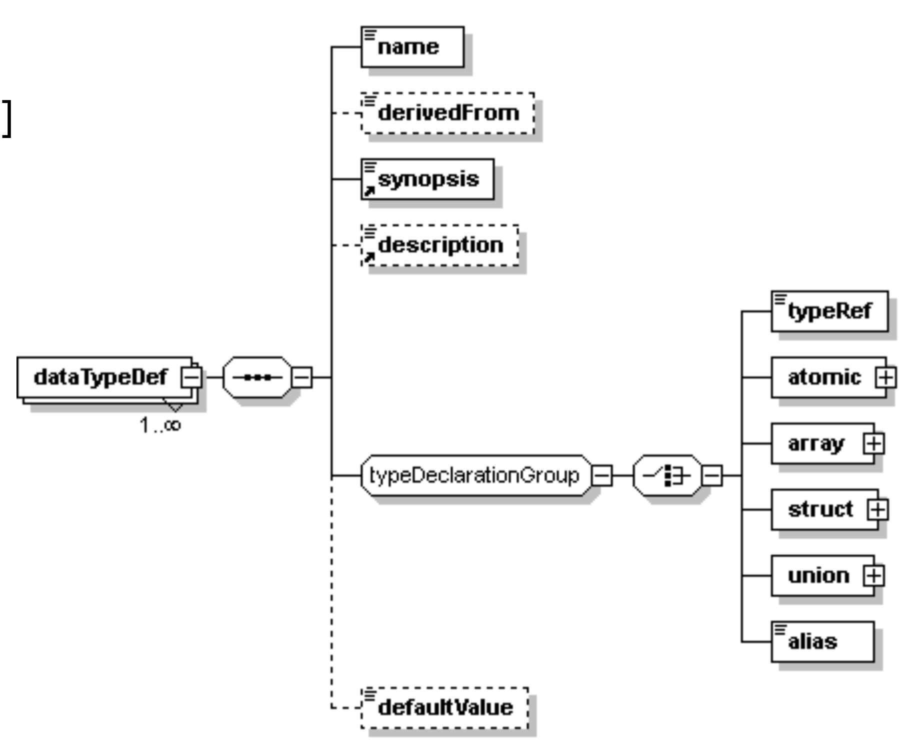
- Frames or Packets Definitions expected/produced from/at LFB ports
- Example:

```
<frameDef>
  <name>IPv4</name>
  <synopsis>An IPv4 packet</synopsis>
</frameDef>
```



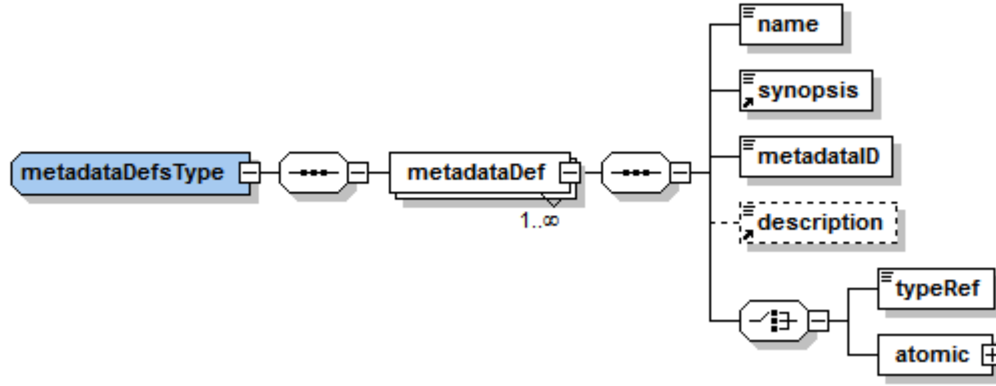
ForCES Model – DataTypes (RFC5812)

- Datatype definition
  - C-like base datatypes
    - Atomic
      - char, uchar, byte[N]
      - String, String[N], octetstring[N]
      - (u)int16, (u)int32, (u)int64
      - float32, float64
      - Boolean
    - Compound
      - Struct
      - Arrays
    - Alias (Symbolic Links)
    - Augmentations
  - Building blocks for **custom-defined datatypes.**



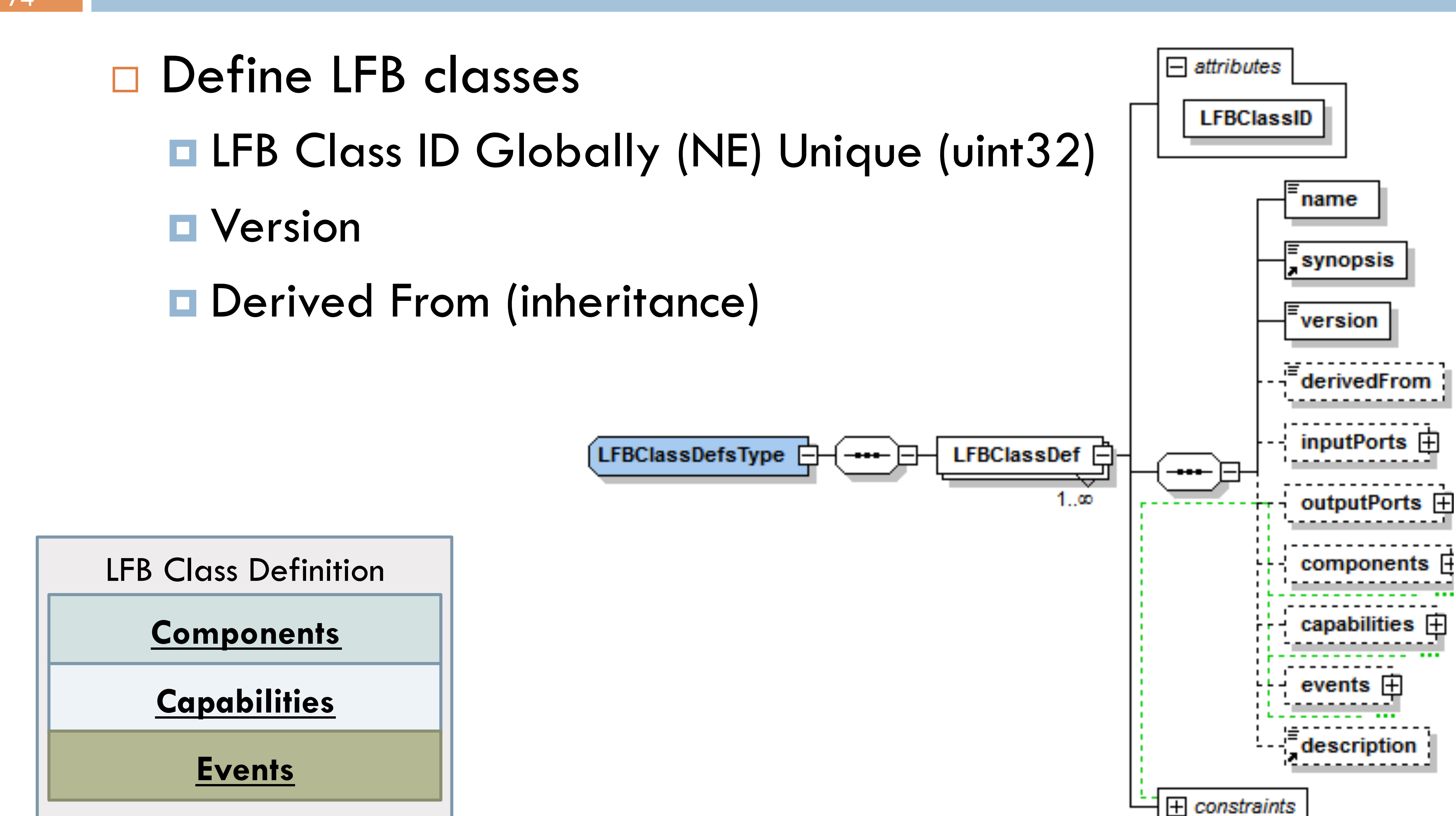
ForCES Model – Metadata

- Metadata Type definition
  - Data produced by LFBs to assist other LFB's processing
  - E.g. PHYPortID.
- Atomic (RFC5812)/Compound (RFC7408) Data type
- Metadata ID MUST be LFB library unique



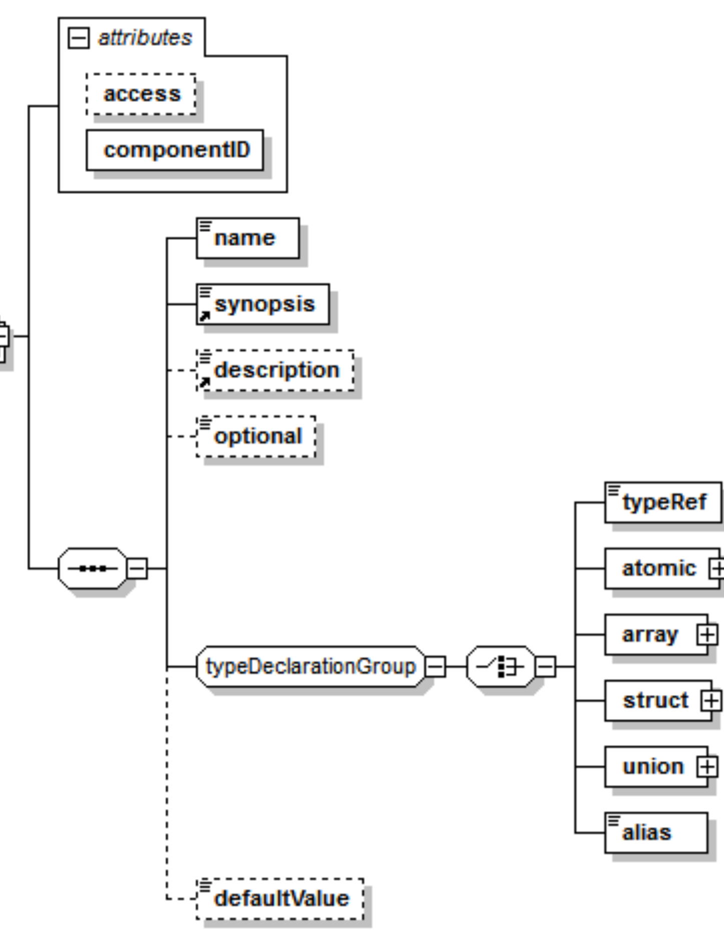
ForCES Model – LFB Class

- Define LFB classes
  - LFB Class ID Globally (NE) Unique (uint32)
  - Version
  - Derived From (inheritance)



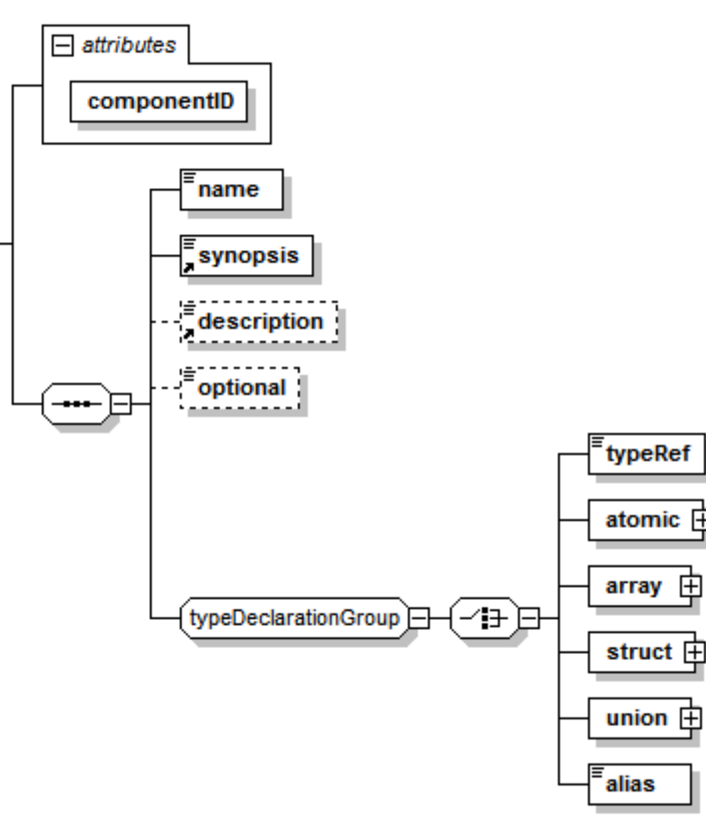
ForCES Model – Components

- Operational Parameters visible to CE
- Component ID
  - Unique per component level
- Access control
  - Read-only
  - Read-write
  - Read-reset
  - Trigger-only
  - Write-only
- Datatype Reference

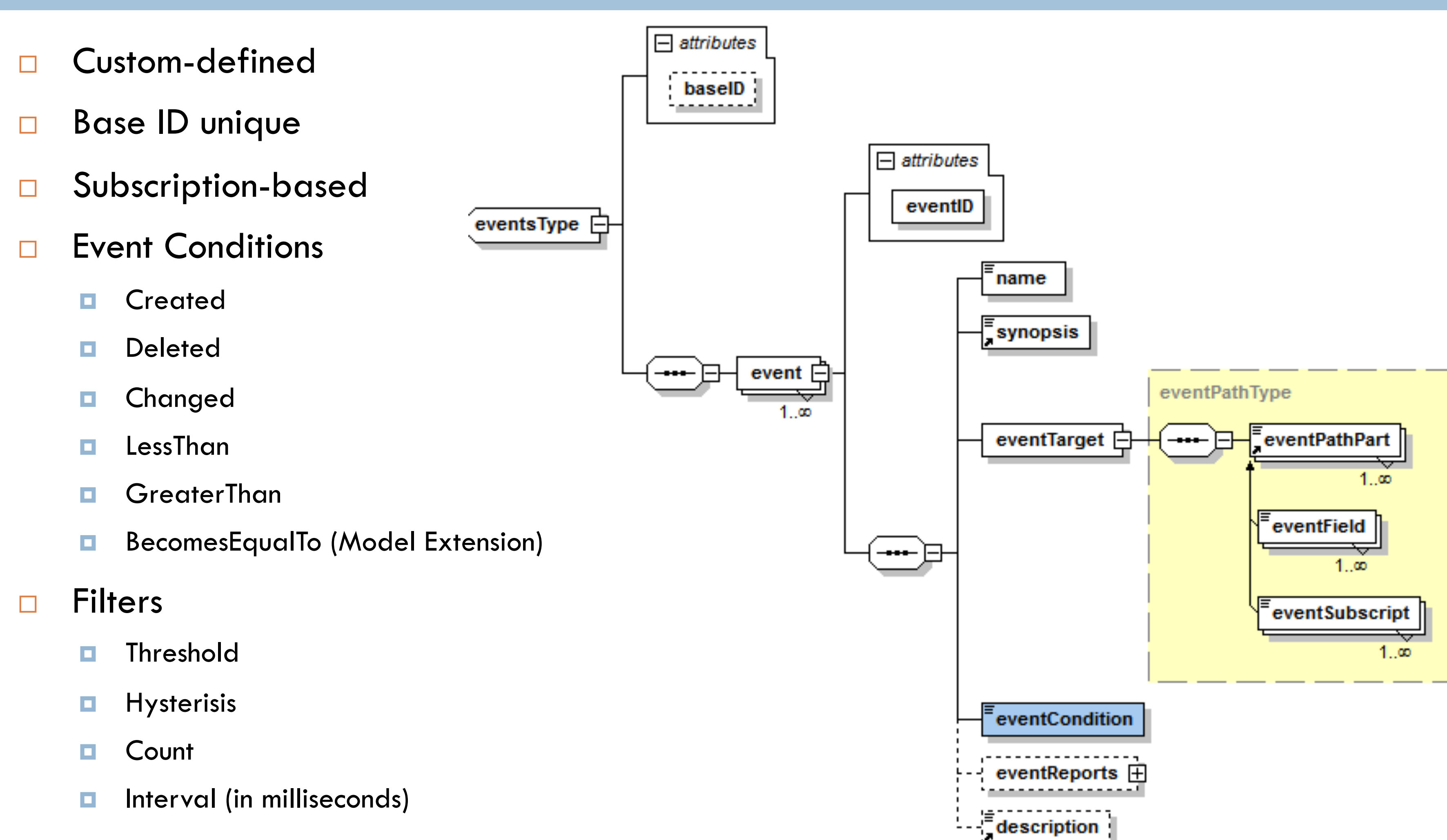


ForCES Model – Capabilities

- Limitations or Capabilities advertised by LFB
  - E.g. HA capabilities
  - E.g. Port number Limit
- Similar to Components
  - Read-Only
- Component ID unique



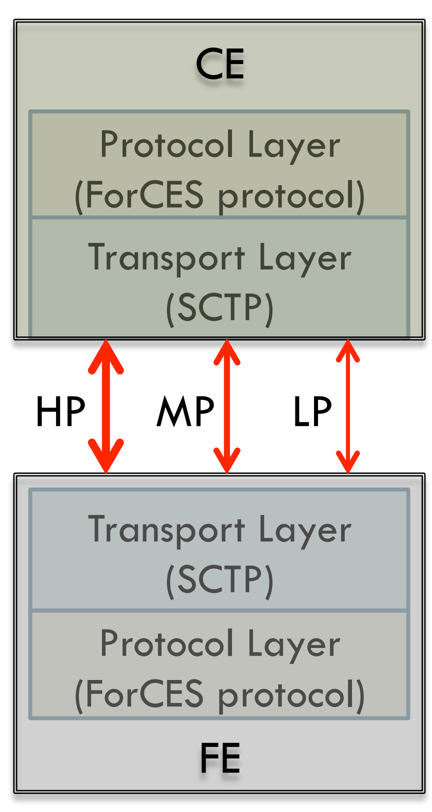
ForCES Model – Events



ForCES Model – LFB Example

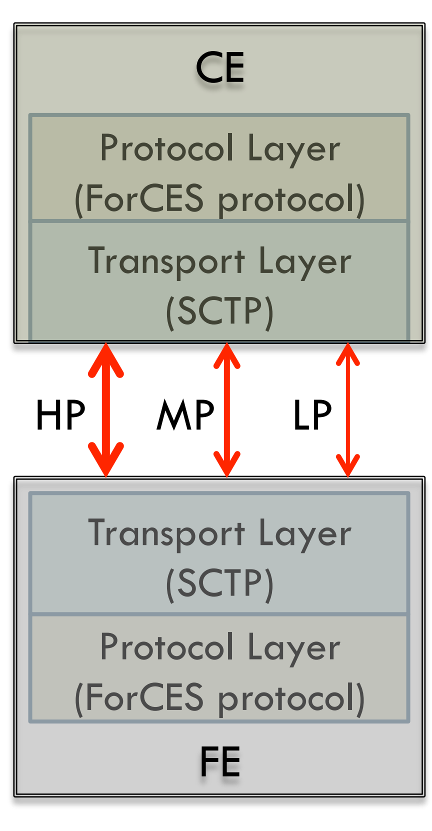

```
<dataTypeDefs>
  <dataTypeDef>
    <name>PacketCounter</name>
    <synopsis>Counts Packets</synopsis>
    <typeRef>uint32</typeRef>
  </dataTypeDef>
</dataTypeDefs>
<LFBClassDefs>
    <LFBClassDef LFBClassID="1000">
    <name>Monitor</name>
    <synopsis>A monitor LFB for packets</synopsis>
    <version>1.0</version>
    <components>
      <component componentID="1" access="read-only">
        <name>GoodPackets</name>
        <synopsis>Good packets</synopsis>
        <typeRef>PacketCounter</typeRef>
      </component>
      <component componentID="2" access="read-only">
        <name>BadPackets</name>
        <synopsis>Bad packets</synopsis>
        <typeRef>PacketCounter</typeRef>
      </component>
    </components>
    <capabilities>
      <capability componentID="3">
        <name>PacketCheck</name>
        <synopsis>Type of checks</synopsis>
        <struct>
          <component componentID="1">
            <name>CRC</name>
            <synopsis>Checks for CRC</synopsis>
            <typeRef>boolean</typeRef>
          </component>
          <component componentID="2">
            <name>BadFrame</name>
            <synopsis>Checks for bad frames</synopsis>
            <typeRef>boolean</typeRef>
          </component>
        </struct>
      </capability>
    </capabilities>
  <events baseID="4">
      <event eventID="1">
        <name>CheckBadPackets</name>
        <synopsis>Checks for bad packets</synopsis>
        <eventTarget>
          <eventField>BadPackets</eventField>
        </eventTarget>
      <eventGreaterThan>1000</eventGreaterThan>
        <eventReports>
          <eventReport>
            <eventField>BadPackets</eventField>
          </eventReport>
        </eventReports>
      </event>
    </events>
  </LFBClassDef>
```

ForCES Protocol (RFC5810)



- Protocol & Transport Layer
- ForCES
  - Base ForCES semantics and encapsulation (RFC 5810)
  - Two phases:
    - Pre-association
    - Post-association
- Transport depends on underlying media. One is standardized (RFC 5811) – others expected to be
  - Standardized TML: SCTP with strict priority schedule
    - High Priority (HP): Strictly reliable channel
    - Medium Priority (MP): Semi-reliable
    - Low Priority (LP): Unreliable channel

ForCES Protocol (cont.)

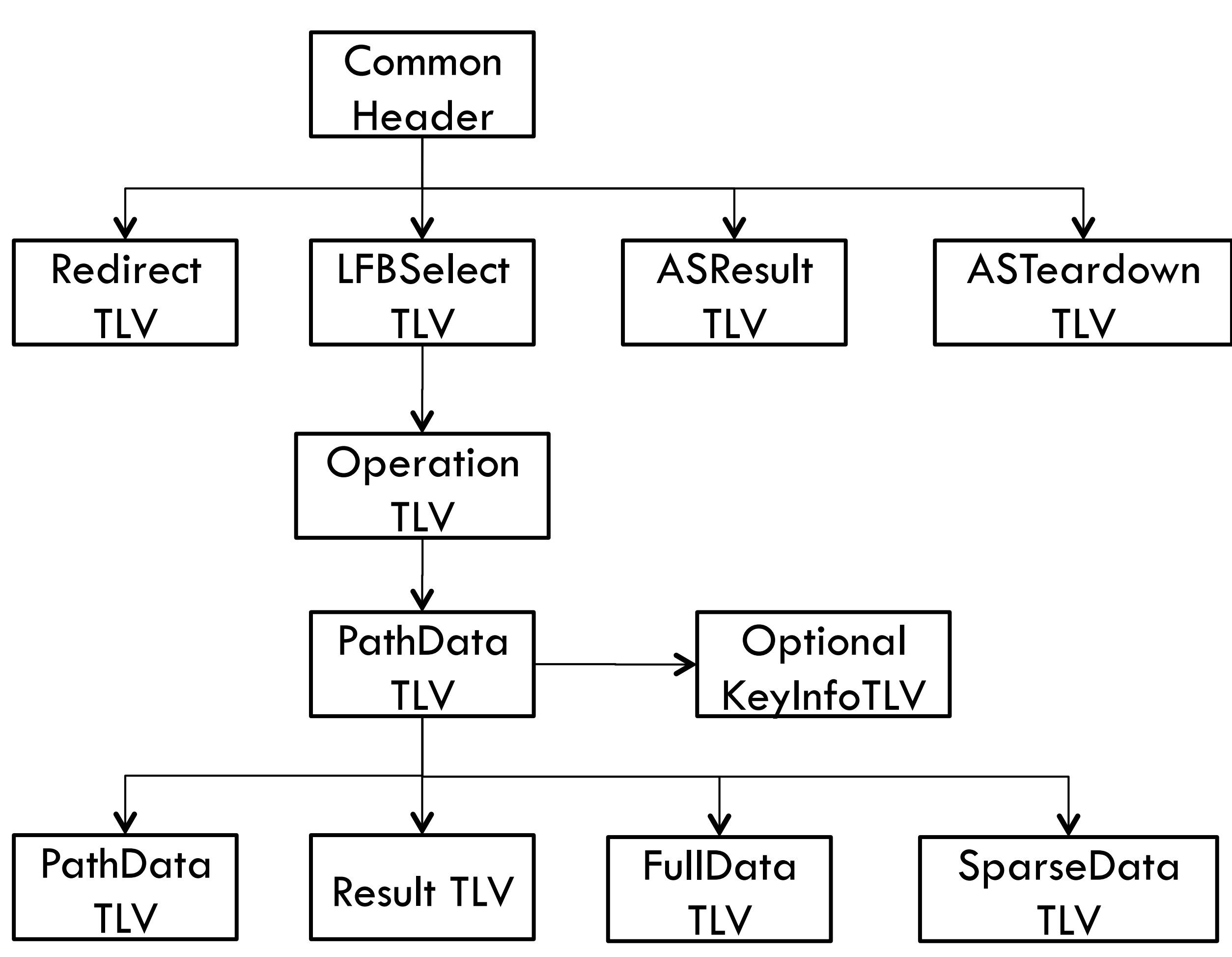


- Simple Commands (Verbs) (Model elements are nouns)
  - Set/Get/Del
  - Set/Get Properties (for properties & events)
- Message Acknowledgment
  - Always/Never/On Failure/On success
- Transactional capability (2 Phase Commit)
- Various Execution modes
  - Execute all or none
  - Execute till failure
  - Execute on failure
- Scalability
  - Batching
  - Command pipeline
- Security
  - IPSec
- Traffic Sensitive Heartbeating
- High Availability
  - Hot/Cold Standby

ForCES Protocol – Addressing

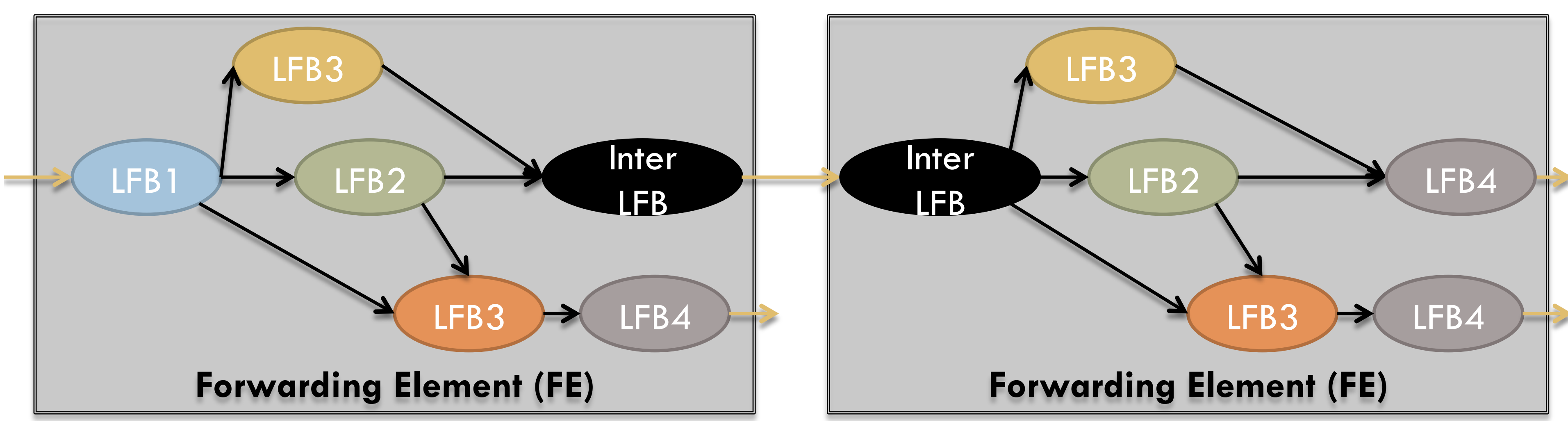
- Addressing scheme similar to SNMP MIBs (OIDs)
- FEs in an NE uniquely distinguished by a 32-bit ID (FEID)
  - FEID within FE Protocol LFB (Assigned by FE Manager)
- LFB Class IDs unique 32-bit ID (IANA assigned)
  - LFB Instance ID unique per FE
- Components/Capabilities/Struct Components/Events have 32-bit IDs
- Arrays – Each row with a 32-bit row ID index
  - Supports Key content addressable
- Path: **Verb+/FEID/LFB Class/LFB Instance/Path to component**
  - **E.g. GET /FE 3/Port/1/PortTable/ifindex10**
  - **Wire: 7 /3/4/1/1/10**

ForCES Protocol – Message Construction

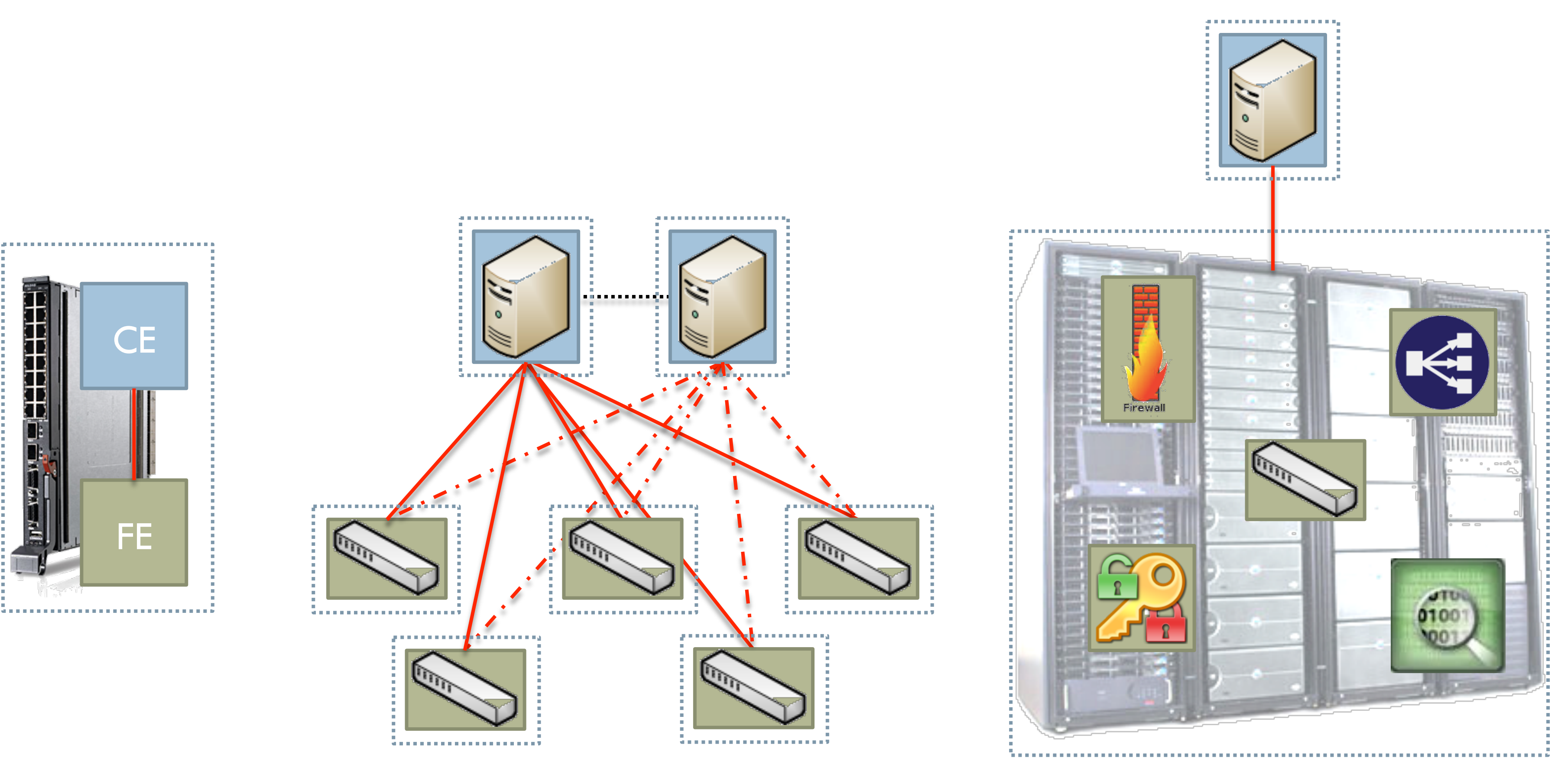


Newest additions

- Subsidiary mechanism LFB
  - LFB to handle management functions on FEs
    - Load/Unload new LFBs
    - Setup CE connectivity
- InterFE LFB
  - LFB to chain functionality of LFBs across multiple FEs

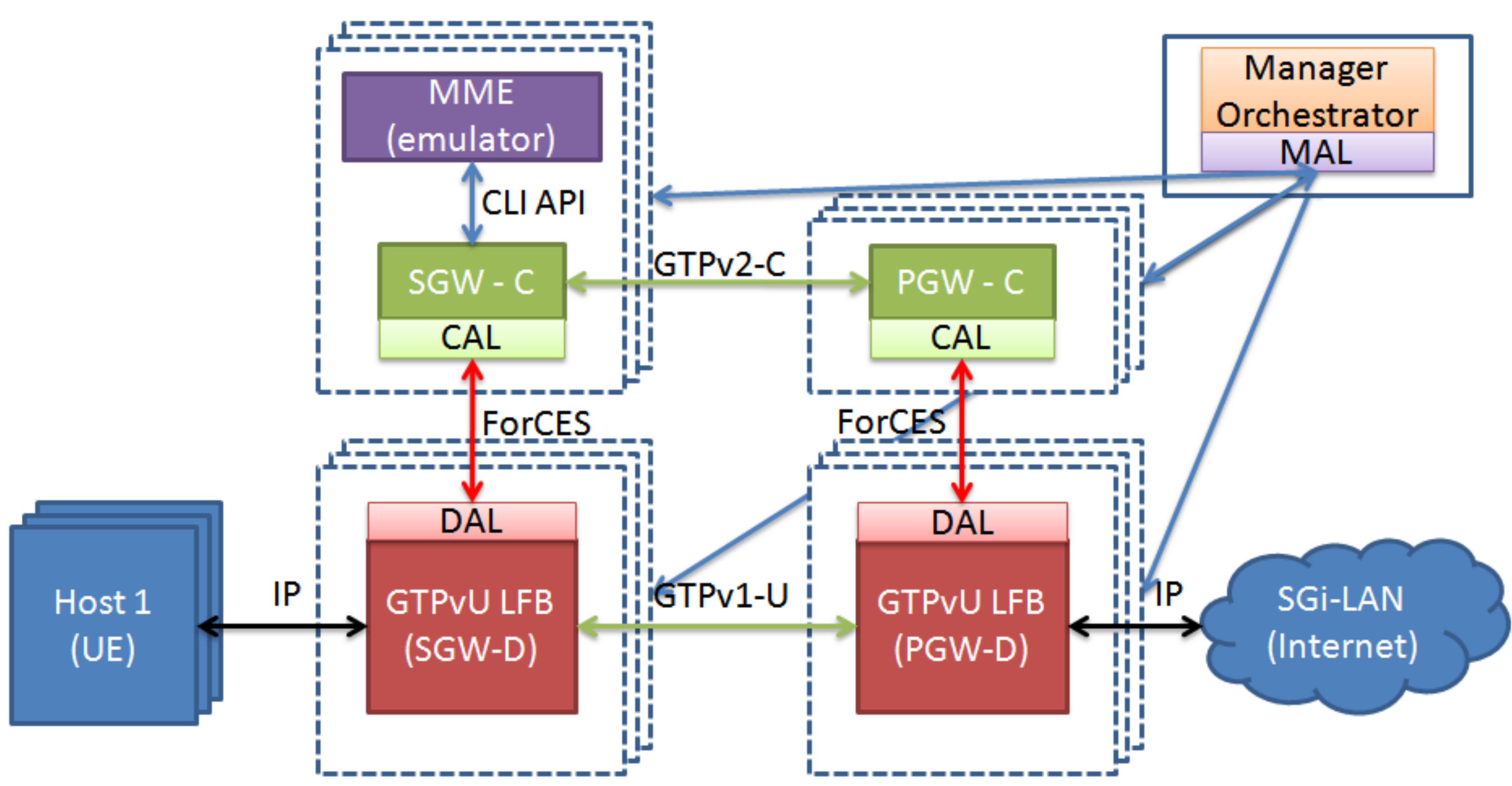


Usage Examples



Usage Examples (Data Center)

Usage examples (SDN/NFV)



Usage examples (Common model)

App
App
App
App

Network Manager
ForCES Abstraction Layer

ForCES Layer
ForCES
VNF
FE
ForCES
VNF
FE
VM
Hypervisor
LFB
Network Devices

ForCES Layer
ForCES
VNF
FE
Network Devices

ForCES Layer
ForCES Layer
VNF
Forwarding Element (FE) (Hardware)
Forwarding Element (FE) (VM)
Network Devices

ForCES Layer
ForCES
VNF
FE
Network Devices

ForCES (for configuration)

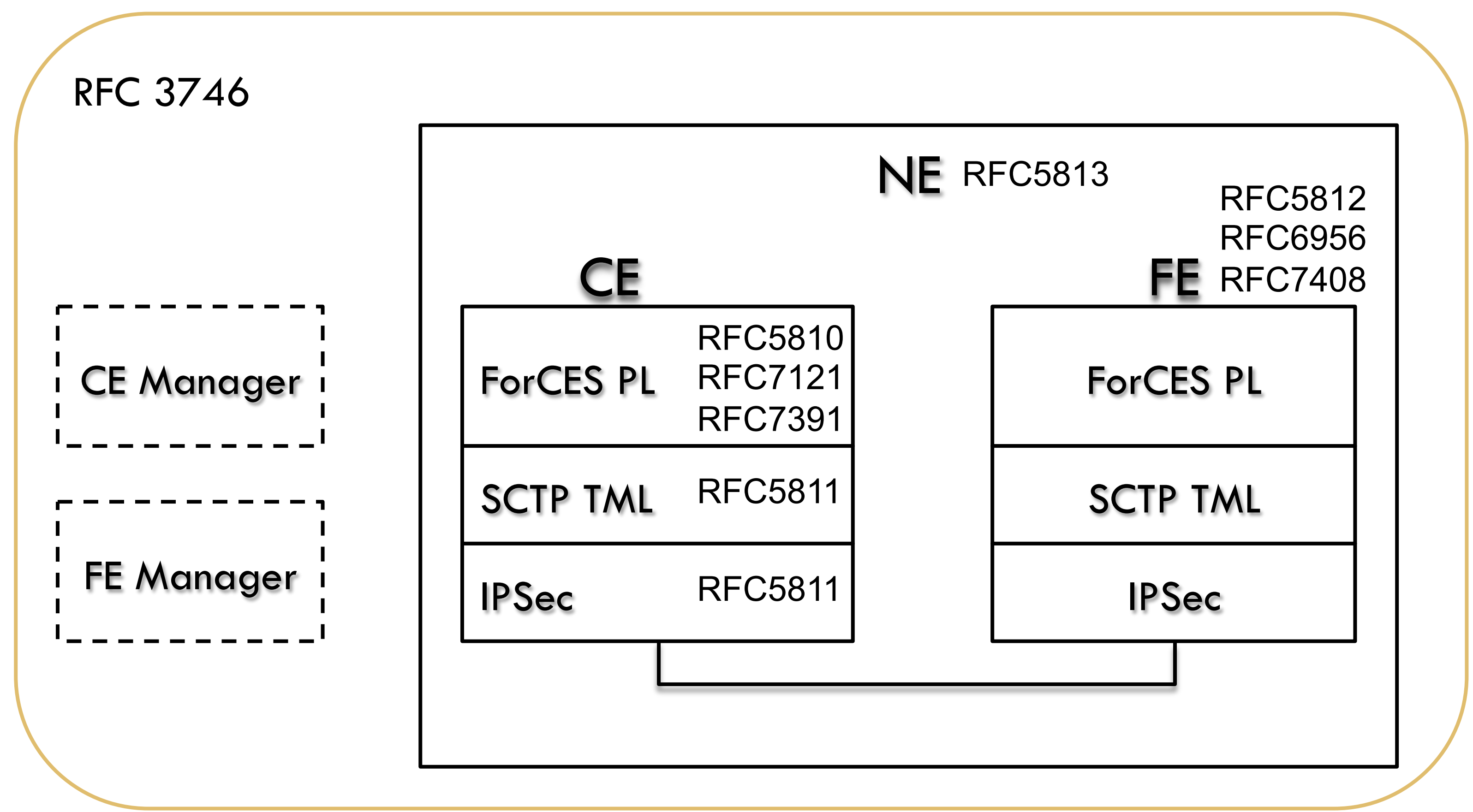
ForCES (for management)

Open API (with ForCES semantics)

Summary & Conclusion

- ForCES has a potential to be used where separation is required.
- Besides datapath management
  - Wired
    - Device management (Up/Down)
    - Change device functionality (if device is capable)
  - Wireless
    - Channel selection
    - SSID management
    - Adjust RF parameters
    - Access Control
  - LTE
    - Management of devices (from base stations to backbone) from a central location

Thank you for listening



ForCES RFC Roadmap