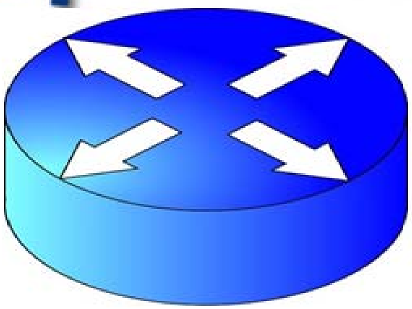
Software Defined Networking

Network Operating Systems & SDN Languages

David Hausheer

Department of Electrical Engineering
and Information Technology
Technische Universität Darmstadt

E-Mail: hausheer@ps.tu-darmstadt.de
http://www.ps.tu-darmstadt.de/teaching/sdn



*Based on original slides by Bernhard Ager, Stephan Neuhaus (ETH Zürich)

Lecture Overview

- Network Operating Systems

- Network Programming Languages

Network Operating Systems

Why SDN?

Based on a talk by Scott Shenker

- **Why** is SDN the right choice for the future?
  - Obviously efficiency, scalability, security, functionality, versatility, ...
  - **Well, not directly ...**

- **What** are the fundamental aspects of SDN?
  - Obviously OpenFlow ...
  - **Actually, not at all**

The role of abstractions in networking

- Networking currently built on a weak foundation
  - Lack of fundamental abstractions

- Network control plane needs three abstractions

- Abstractions solve other architecture problems
  - Not discussed here, see Scott Shenker's original talk: http://www.youtube.com/watch?v=YHeyuD89n1Y or http://www.slideshare.net/martin_casado/sdnabstractions (longer version)

Weak Intellectual Foundations

- OS courses teach fundamental principles
  - Synchronization primitives, e.g., mutex, semaphore
  - Files, file systems, threads, ...
  - Processes, memory separation, isolation, ...
  - Privileges, roles, permissions, ...

- Networking courses teach a bag of protocols
  - Design guidelines instead of principles

Weak practical foundations

- Computation and storage have been virtualized
  - Infrastructure more flexible and more manageable

- Networks notoriously hard to manage
  - Network admins large share of sysadmin staff
  - Anecdotally: „18 layers of virtualization“

Weak evolutionary foundations

- Ongoing innovation in system software
  - New languages, operating systems, etc.
- Networks stuck in the past
  - Routing algorithms change very slowly
  - Network management extremely primitive

Why are networking foundations weak?

- Networks used to be simple
  - Basic IP over Ethernet simple and easy to manage
- New control requirements have led to complexity
  - ACLs, VLANs, traffic engineering, middle boxes, DPI
- It still works
  - because of our ability to master complexity
- This ability is both a blessing and a curse

The evolution of software design

1. Machine languages: no abstractions
   - Dealing with register use, memory layout, ...

2. Higher-level languages and operating systems
   - File system, virtual memory, malloc/free, arrays, ...

3. Modern languages
   - Object oriented, garbage collection, iterators, exceptions, higher-level data structures, ...

Abstractions simplify programming:

Easier to write, maintain, reason about programs

Why are abstractions/interfaces useful?

- Interfaces are instantiations of abstractions

- Interfaces shields implementation details
  - Implementation freedom on both sides
  - Leads to modularity and exchangeability

- What role do abstractions play in networking?

Layers: The main network abstraction

- Layers provide nice **data plane** abstractions
  - IP’s best effort delivery
  - TCP’s reliable byte stream

- Aside: Good abstraction, terrible interface
  - Implementation details not hidden away

- However: no **control plane** abstractions

No abstractions → Increased complexity

- Each control requirement: new mechanism
  - TRILL, LISP, ...

- We are good at designing mechanisms
  - So we never tried to make our live easier
  - And networks grow more and more complex

- But this cannot work forever
  - We have to find ways to **extract simplicity** instead of continuing to **master complexity**

How do we build a control plane, today?

- Define a new protocol from scratch
  - E.g., a routing protocol
- Or, reconfigure an existing mechanism
  - E.g., traffic engineering
- Or, leave it to manual configuration
  - E.g., access control, middleboxes, home routers

Design constraints

- Operate within the confines of a given data path
  - Must live with capabilities of IP
- Operate without communication guarantees
  - Distributed system with arbitrary delays and loss
- Compute configuration of each physical device
  - Switch, router, middlebox: FIB, ACL, ...

This is insanity!

Analogy in programming

- What if programmers had to
  - Specify where each bit was stored
  - Explicitly deal with all internal communication errors
  - With a limited expressibility programming language
- Programmers would redefine problem:
  - Define higher level abstractions for memory
  - Build on reliable communication primitives
  - Use a more general language
- Divide the problem into tractable pieces

Why not for network control?

Personal side note

- SDN research today often stuck with old habits
  - Defining new protocols to solve old problems
  - Or, adapting old solutions to new technology
  - Leads to even more complexity, not to high impact
- Only very few research groups interested in finding good (new?) abstractions (or in understanding fundamental limitations)
- Why is that so?
  - Easier? More straight-forward? Fast publications?
  - „SDN problem“ not well enough understood?

Abstractions should separate 3 problems

- Constrained forwarding model
- Distributed state
- Detailed configuration

(Actually, this is the minimum set)

Forwarding abstraction

- Flexible forwarding model
  - **Behaviour specified by control program**
- Abstract away forwarding hardware
  - For evolving beyond vendor-specific solutions
- Flexibility and vendor-neutrality both valuable
  - One architecturally, the other economically

Specification abstraction

- Control program should express desired behaviour
- Control program should **not** be responsible for implementing that behaviour
- Natural abstraction: **simplified model** of network (aka virtualization)
  - Only enough detail to **specify** goals

State distribution abstraction

- Control programs should not have to deal with problems caused by distributed state
  - Complicated, source of errors
  - Abstraction should hide state distribution details

- Proposed abstraction: global network view

- Control program works on global view
  - Input: global view, e.g., network graph
  - Output: configuration of each device

Network Operating System: NOS

- Distributed system that creates network view
  - Runs on servers in the network

- Communicates with forwarding elements
  - Get state from forwarding elements
  - Send control directives to forwarding elements
  - Utilizes forwarding abstraction

- Control program works on view of network
  - Doesn’t have to be a distributed system
  - Computes configuration

NOX: Towards an OS for Networks

TECHNISCHE UNIVERSITÄT DARMSTADT

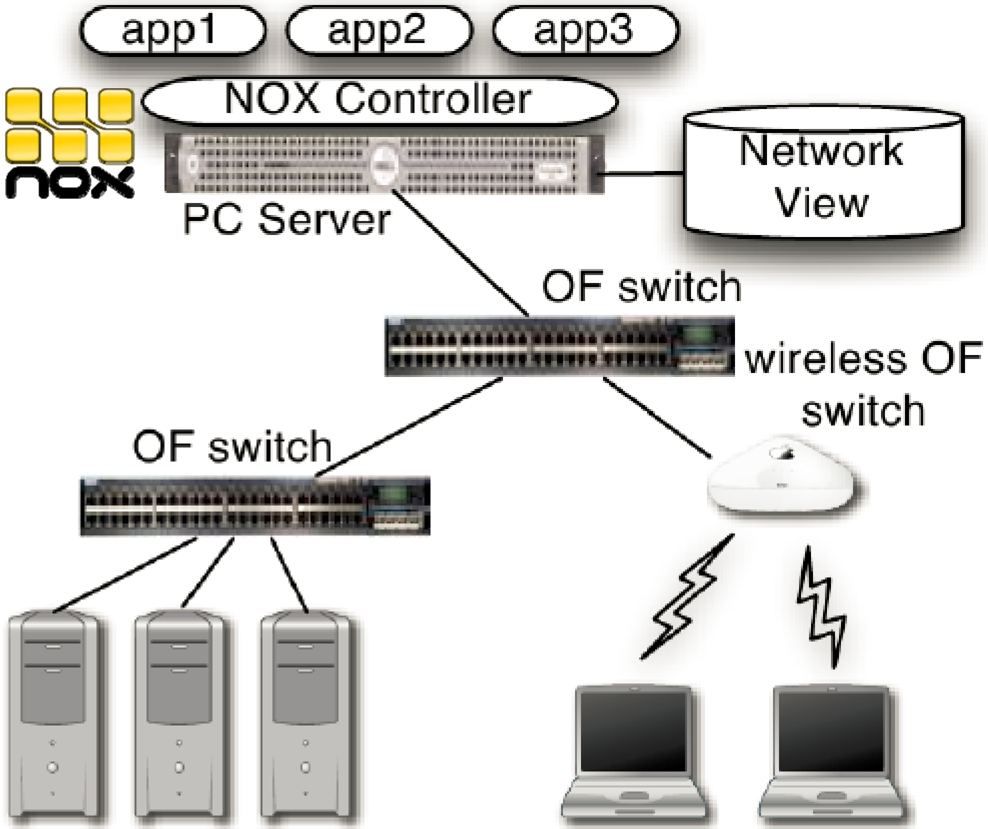
Natasha Gude, Teemu Koponen et al, SIGCOMM CCR, 2008.

Talk by Martin Casado, „A Network Operating System for OpenFlow", SDN Workshop 2009

- The first take on a NOS
- Targeted to implement the abstractions described before
- Implemented in C++, Python runtime on top
- In the meanwhile: Replaced by POX

http://www.noxrepo.org/

NOX components



NOX design overview

- Granularity: Scalability vs. flexibility
  - Aware of switch-level topology, user locations, middleboxes, services, ...
  - Forwarding management on flow level

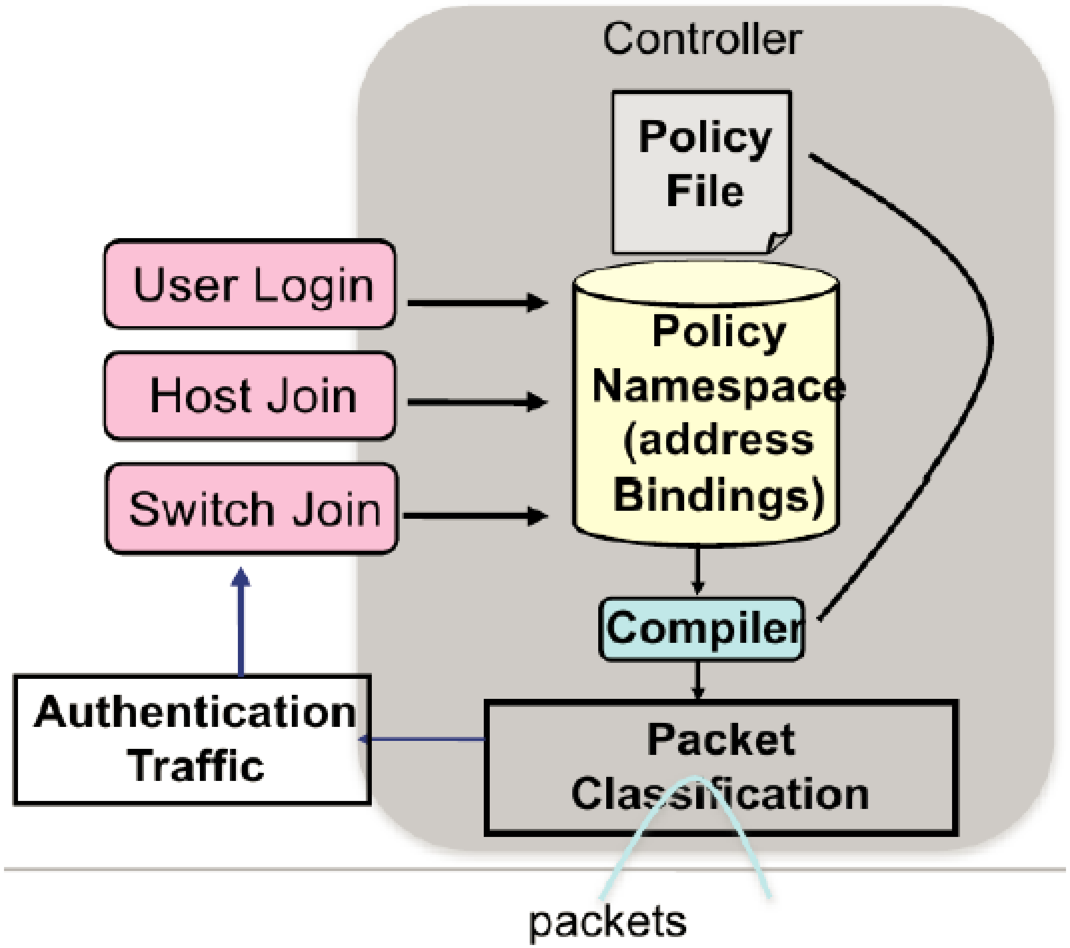
- Switch abstraction and operation
  - Adopts OpenFlow model

- Scalability:
  - Leverage parallelism for flow arrivals
  - Maintain centralized network view: indexed hash tables, distributed access with local caching

NOX programming interface

- Events
  - Event handlers handled according to priority
- Network view and namespace
  - Writes to network view should be limited
  - Monitoring DNS allows mapping host names to flows
- Control: OpenFlow
- Higher-level services
  - System libraries for routing, packet classification, standard services (DHCP and DNS), network filtering

NOX usage example: Policy lookup



- Enables
  - User-based VLAN tagging
    - User-based traffic isolation
  - Ethane
    - Network-wide access control

NOX doesn't solve all the problems

TECHNISCHE UNIVERSITÄT DARMSTADT

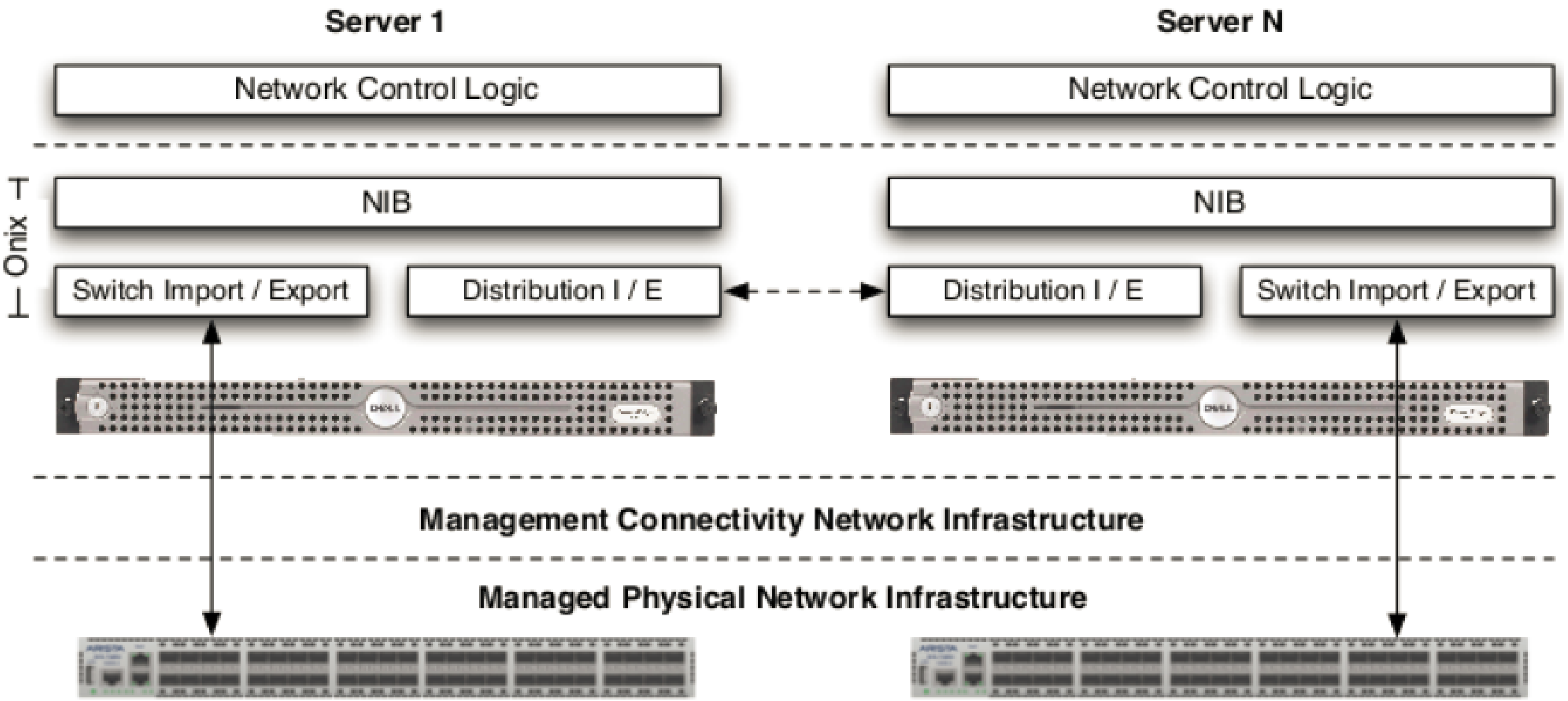
- Reliability and robustness?
- Managing state
  - Strong consistency?
  - Persistence?
  - Scalability?
- Generality
  - What if the switch doesn't speak OpenFlow?

ONIX

Onix: A Distributed Control Platform for Large-scale Production Networks. T. Koponen , et al. OSDI 2010.

- Network operating system (closed-source)
- Design goals: Generality, Scalability, Reliability, Simplicity, Performance
- Used in the Google backbone
- Speculation: asset for Nicira deal (1.26 Billion $)

ONIX components



- Similar to NOX, but distributed by design

ONIX contributions

- Centralized logic on distributed system
  - Scalability and robustness

- "Useful" tools to application developers
  - Simple, general API
  - Flexible state distribution mechanisms
  - Flexible scaling and reliability mechanisms

- Production quality system

Onix API

- Program on a network graph
  - Nodes are physical network entities
- Represented by Network Information Base (NIB)
  - External changes imported to NIB
  - Local changes exported from it to affected elements

Onix data distribution

- Different requirements depending on application
  - Strong consistency vs. eventual consistency
- Different storage options
  - Replicated transactional (SQL) storage
  - Distributed hash table (DHT)
- Storage requirements specified at startup
  - But during run-time only interact with NIB

Onix scaling

- Multiple dimensions for partitioning
  - By task
  - By subsetting NIB
  - By subsetting switches

- By aggregation
  - Partition network
  - Only distribute “averaged” NIB information to other partitions

Use cases

- Ethane
- Distributed Virtual Switch
  - One virtualized combining host-switches and physical switches in a data center
- Multi-tenant data center
  - Provides isolation
- Scale-out IP router
  - Scales with number of physical switches
  - Manage routing with, e.g., Quagga

ONIX does not solve all problems

Most advanced (as of yet) NOS, but still

- Not clear how to write applications
  - That work on the same header space
  - That need to perform non-local decisions

- Not clear if NIB abstraction is abstract enough

Network OS conclusion

- Abstractions are essential to simplify management of complex systems
  - But networking just “not there” yet
- Network operating systems are a first step in the right direction
  - But cannot solve all problems
- Line between NOS and controller is not sharp
  - Floodlight, Opendaylight, and others call themselves controllers, but provide more services than, e.g., NOX
  - Terms often used interchangably

Network Programming Languages

Why Languages?

- SDN talks at the level of flows
- Novel applications talk on different level
  - Cache video streams to a caching server nearby
  - Anonymise certain flows on entry, deanonymise on exit
- Policies talk on different level
  - “Gold” customers can use the service 5h/month
  - If there are more than 100 connection requests per second to the web server network-wide, drop some
- Need a way to map one to the other

Language Components

- Syntax (how something is written down)
  - In C, you write `for (int i = 0; i < n; i++)`
  - In Python, you write `for i in range(0, n):`
- Semantics (what it means what you've written down)
  - Loop means that the variable `i` gets assigned the values `0, 1, …, n-1` in order and the loop body executed.
- Paradigms (things you get for free)
  - In object-oriented languages: classes, inheritance, …
  - In functional languages: lambdas, higher-order functions
  - Influence what you can write and how
    - Array assignment in C: `for (i = 0; i < n; i++) b[i] = a[i];`
    - Array assignment in PL/I: `b = a;`

Importance of Paradigms

- All “real” programming languages these days are “Turing-complete”
  - What you can do in one language, you can do in any other
  - No language inherently more powerful than another
- Still, some things are easier in some languages than in others
  - Text management in FORTRAN?
  - Linear algebra in Perl?
  - Inheritance in C? (Look at the Linux kernel)
- Can all be done, but it’s cumbersome

SDN Languages vs. Ordinary Languages

- SDN languages are special-purpose languages
  - Won't talk much about integers
  - Talks more about network addresses
  - Builds SDN-specific abstractions

- Different paradigms
  - C++, Python: inheritance, virtual functions
  - SDN: traffic handling rules, rule composition, topology abstraction/constraints, concurrency, ...

- SDN languages *very* different from ordinary ones

Deep vs. Wide

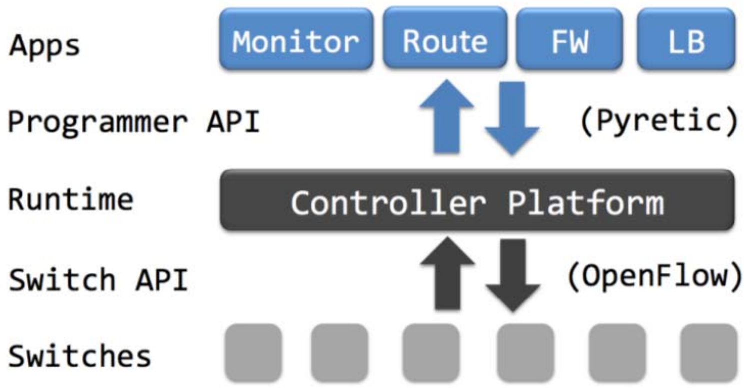
- There are just a few SDN languages

- Still, no time to discuss them all
  - Deep (few languages, much detail)?
  - Wide (many languages, little detail)?

- We choose “deep” and focus on one language:
  - Frenetic/Pyretic (Main publication: Monsanto et al., Composing Software-Defined Networks, NSDI ‘13)
  - Project website: http://frenetic-lang.org/pyretic/
  - Python-based language (you can try it!)

Key Concept: Modularity (1)

- Applications usually perform more than one task
  - Routing
  - Monitoring
  - Load balancing
- Code that affects one part must not affect others
- With “low-level” APIs speaking only of flow rules, leads to large applications where everything is connected with everything else
- Difficult to develop, deploy, maintain
- Silver bullet: modularity



Picture source: Reich et al.: Modular SDN Programming with Pyretic. ;login Magazine, 38(5):128-134, 2013.

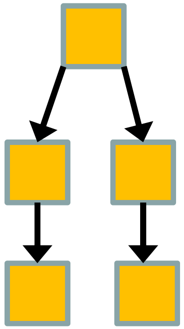
Key Concept: Modularity (2)

- Modularity is *not* the slicing up of the network
  - Slicing enables *different* apps to work on *different* parts of the network without influencing each other

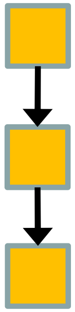
- We want to enable building a *single app* out of *reusable components* all processing the *same traffic*
  - Components specify traffic handling *partially*
  - App = components
    + how components interact (*composition*)
    + on what traffic (topology abstraction)

Types of Composition

- Parallel
  - Part of the app acts on destination address (routing)
  - Other part acts on source address (monitoring)
  - Components have the illusion of each acting on its own private copy of the traffic



- Sequential
  - Access-control policy drops unwanted traffic
  - Routing acts on the remainder of traffic
  - Components get traffic from previous component



Topology Abstraction

- *Network objects* allow each module to work on its own abstract view of the network
- Can be constrained to limit what traffic each module can see and do, e.g.,:
  - Subgraph of actual topology
  - Giant switch comprising the whole network
- Supports:
  - Multiple nesting levels, physical and virtual switches
  - “Many-to-one”: many switches appear as one virtual
  - “One-to-many”: one switch appears as several virtual
- Not explained in detail here

Pyretic Basic Concepts: Packets (1)

- Extensible packet model
- Packet not a fixed structure, unlike, e.g., TCP
- *Dictionary* that maps field names to values
  - Field name “srcip” mapped to source IP address
  - Field name “dstport” mapped to destination port
  - If `p` is a packet and `f` a field name, you write `p[f]`
- Some fields are about *packet location*
  - Switch name, port, direction (in or out)
- In addition to standard fields, also *virtual* fields
- Virtual fields can hold arbitrary data structures

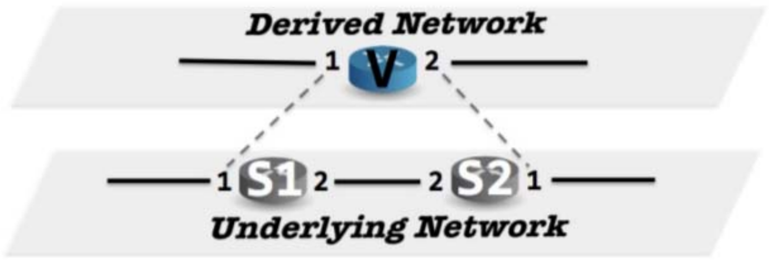
Pyretic Basic Concepts: Packets (2)

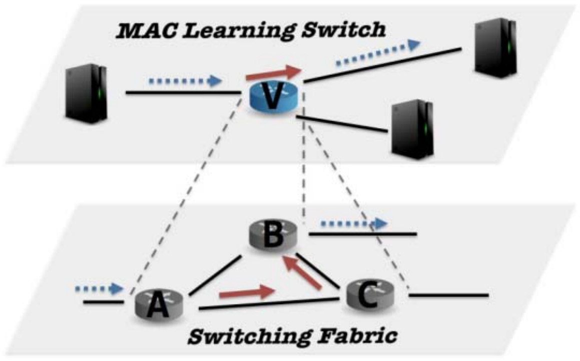
- Every field can hold a *stack* of values
  - E.g.: Source IP field can be [192.168.34.5, 10.0.0.2]
  - Top of stack is current value (192.168.34.5 in the example)
  - Operations: *Push* new values onto stack, *pop* values off
  - Some similarities to concept of VLAN tags or MPLS labels
    - Indeed, they might be used by run-time system
    - Difference: stacking can be applied to all header fields
- Very useful for certain applications, e.g.:
  - Anonymise packet while packet is inside network
    - Must remember original source/destination addresses
    - Push anonymised values for src/dst address on ingress
    - Pop original values on egress

Pyretic Basic Concepts: Packets (3)

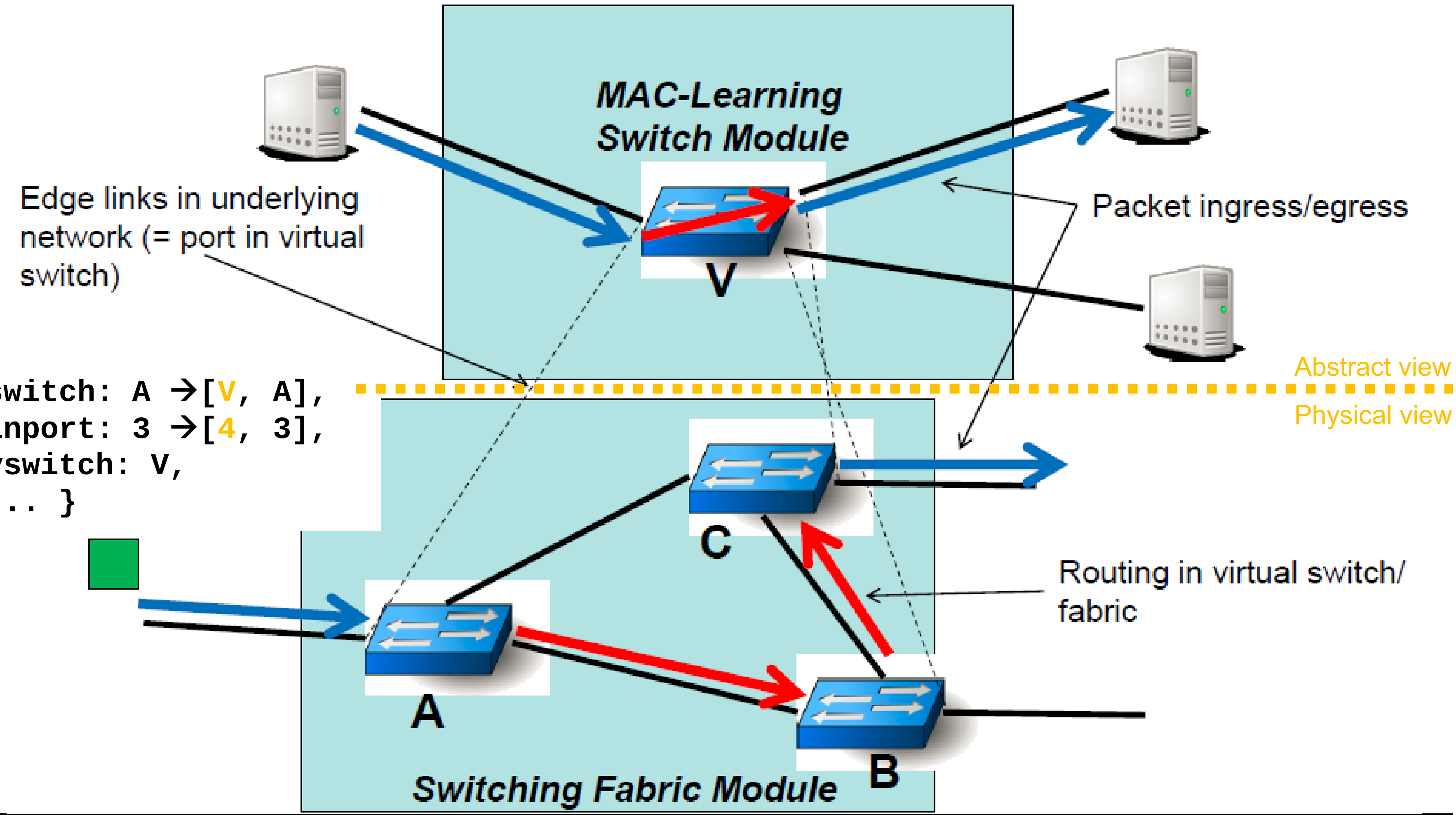
- Stacks are useful for virtual switches:
  - Packet enters virtual switch: push location of virtual switch
  - Packet leaves switch: pop location

- Presents the illusion of a packet travelling through *multiple* levels of abstract networks





Example: Many-to-one Switch




```
{ switch: A →[V, A],
  inport: 3 →[4, 3],
  vswitch: V,
  ... }
```

Stacking Applies to Everything

- All header fields
  - Switch information
  - MAC addresses
  - IP addresses
  - Port numbers
  - Virtual fields
  - ...
  - Only constraint (invariant): standard OpenFlow headers must not ever be empty
- Mapping extended packet model to OpenFlow-supported model: Done by Pyretic run-time system

Policies

- Policies say what is to be done with a packet
- Typical policies are flood, drop, etc.
- Two types of policies: constant (static) and changing (dynamic)
  - Static policies are a “snapshot” of a network’s global forwarding behaviour
  - Can’t make many useful network applications with just a single static (unchanging) policy
  - Need sequence of static policies (dynamic policies)

Static Policy Syntax and Semantics (Reference Slide)

// Primitive actions

A ::= drop | passthrough | fwd(*port*) | flood
| push(*h*=*v*) | pop(*h*) | move(*h1*=*h2*)

// Predicates

P ::= all_packets | no_packets | match(*h*=*v*)
| ingress | egress | *P* & *P* | (*P* | *P*) | ~*P*

// Query Policies

Q ::= packets(*limit*, [*h*]) | counts(*every*, [*h*])

// Policies

C ::= *A* | *Q* | *P*[*C*] | (*C* | *C*) | *C* >> *C* | if_(*P*, *C*, *C*)

Primitive Actions

- Receive a packet with location information as input and returns a set of located packets as output.
  - Example: Input is **`{ switch: A, inport: 3 }`**, output could be **`{{ switch: A, outport: 4 }}`**.
- **`drop`** produces the empty set (no packet is output)
- **`passthrough`** produces the input packet
- **`fwd`**(*port*) changes *outport*
- **`flood`** floods packet using minimum spanning tree
- **`push`**, **`pop`**, **`move`** change packet value stacks
  - **`move(h1=h2)`**: pop top value of h2 and push to h1

Predicates

- Needed to define conditions on packets
- If *C* is a policy and *P* is a predicate, then *P*[*C*] means to apply *C* to all packets for which *P* is true
- **`all_packets`**, **`no_packets`** return true or false, resp.
- **`ingress`**, **`egress`** return true if packet enters (leaves)
- **`match`**(*h*=*v*) return true if header *h* has value *v*
- *P* & *P*, *P* | *P*, ~*P*: composition of predicates:
  - Conjunction: *P1* & *P2* is true if both *P1* and *P2* are true
  - Disjunction: *P1* | *P2* is true if at least one of *P1, P2* are true
  - Negation: ~*P* is true if *P* is false

Query Policies

- Direct information from phys. network to controller
- Packets aren't moved to phys. port on phys. switch
- Rather, they are put into buckets on controller
  - **`counts`**: packet goes to **`counts`** bucket
  - **`packets`**: packet goes to **`packets`** bucket
  - Applications are informed about arrival of packets
  - **`packets`**: entire packets, **`counts`**: packet counts
  - **`packets(1, ['srcip'])`** passes each packet with new source address
  - **`counts(every, ['srcip'])`** calls listeners every **`every`** seconds with number of times each source IP has been seen

Policy Composition

- Policies can be simple actions (flood etc)
- Or query policies (not discussed here)
- Or conditional policies P[C]
  - Applies policy *C* if the packet satisfies predicate *P*
- Composed so that they either act in parallel
  - *C1* **|** *C2* (*C1* and *C2* together)
- or in sequence
  - *C1* **>>** *C2* (first *C1*, then *C2*)
- Or conditionally
  - **`if_(`***P***,** *C1***,** *C2***`)`** (if P, then *C1*, else *C2*)

Policies by Example (1)

- Broadcast every packet entering the network
  - `flood`
- Broadcast all packets entering switch s2 on port 3
  - `match(switch=s2,inport=3)[flood]`
- Drop all HTTP packets on switch s3
  - `match(switch=s2,dstport=80)[drop]`
- Forward packets on s2 from port 2 to port 3
  - `match(switch=s2,inport=2)[fwd(3)]`
- Drop all packets matching some predicate *P*
  - **if_(*P*, drop, passthrough)**

Policies By Example (2)

- Forward packets from port 2 to port 3 if they fulfill some predicate *P*
  - **`(match(switch=s2,inport=2) & P)[fwd(3)]`**
  - **`if_(P, passthrough, drop)`**
    `>> match(switch=s2,inport=2)[fwd(3)]`

- The combination of **`if_`**, **`drop`**, and **`passthrough`** is very convenient for composing policies!

Examples (1): Hub

```
from pyretic.lib import *

def main():
     return flood
```

Examples (2): Monitoring

```
from pyretic.lib import *

def printer(pkt):
    print pkt

def dpi():
    q = packets(None, [])
    q.when(printer)
    return match(srcip='1.2.3.4')[q]

def main():
    return dpi() | flood
```

Print packet to terminal

Query that takes all and unlimited number of packets

Register printer as listener for q

Match srcip and pass on to q

Parallel execution

Examples (2), Detail

```
def dpi():
    q = packets(None, [])
```

- Process ALL the packets!

```
    q.when(printer)
```

- When I get a packet, print it!

```
    return match(srcip=‘1.2.3.4’)[q]
```

- Print all packets matching source IP 1.2.3.4

Examples (3): Dynamic Policies MAC-learning Switch

```
From pyretic.lib import *

def learn(self):
   def update(pkt):
      self.P =
         if_(match(dstmac=pkt['srcmac'],
                   switch=pkt['switch']),
            fwd(pkt['inport']),
            self.P)
   q = packets(1,['srcmac','switch'])
   q.when(update)
   self.P = flood | q

def main():
   return dynamic(learn)()
```

Update dynamic policy: extend by conditional forward

The first time you see new source MAC and switch, update the policy

Initial definition of dynamic policy

Example Pyretic Applications

- ARP responder
- Firewalls
- Gateways
- Load balancers
- Monitoring
- Big switch
- Spanning tree