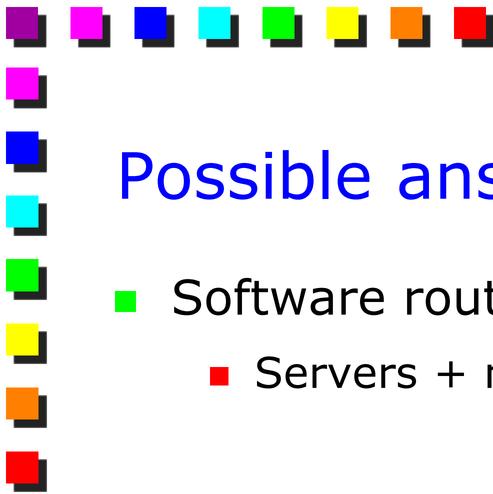
OpenFlow

Fulvio Risso

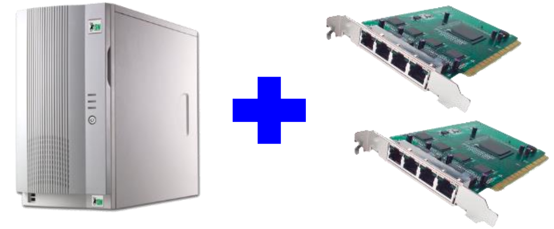
Politecnico di Torino

- Many research problem still exist on the network
  - Mobility, security, computing/storage virtualization, etc.
  - No way to test possible solutions on realistic networks: we need to build a test network with sufficient scale and realism
- This is at the foundation of the original OpenFlow question:
  - **How can researchers test out new ideas in a real network, given that current network devices are closed to third-party software?**
- Which can be translated into:
  - How can we build router/switch that allow to play with the new protocols we like?
  - How can we use real traffic in the testbed environment?



Possible answers toward more programmability

- Software routers: limited throughput
  - Servers + multiport NICs



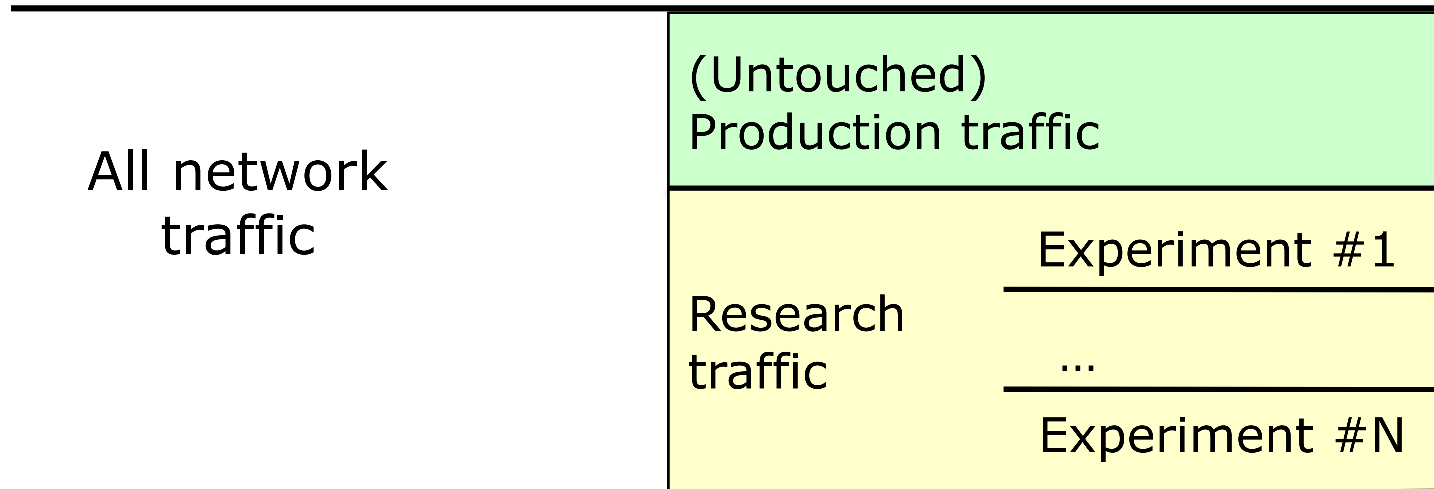
- Custom (programmable) hardware
  - Huge effort required (available only to big vendors, not to researchers)
  - Difficult to program, resource constrained, expensive

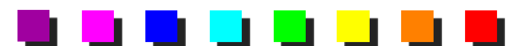
- Modify existing equipment
  - Not an option (in 2000+), closed software, no available to third parties

- None of them represents a suitable answer

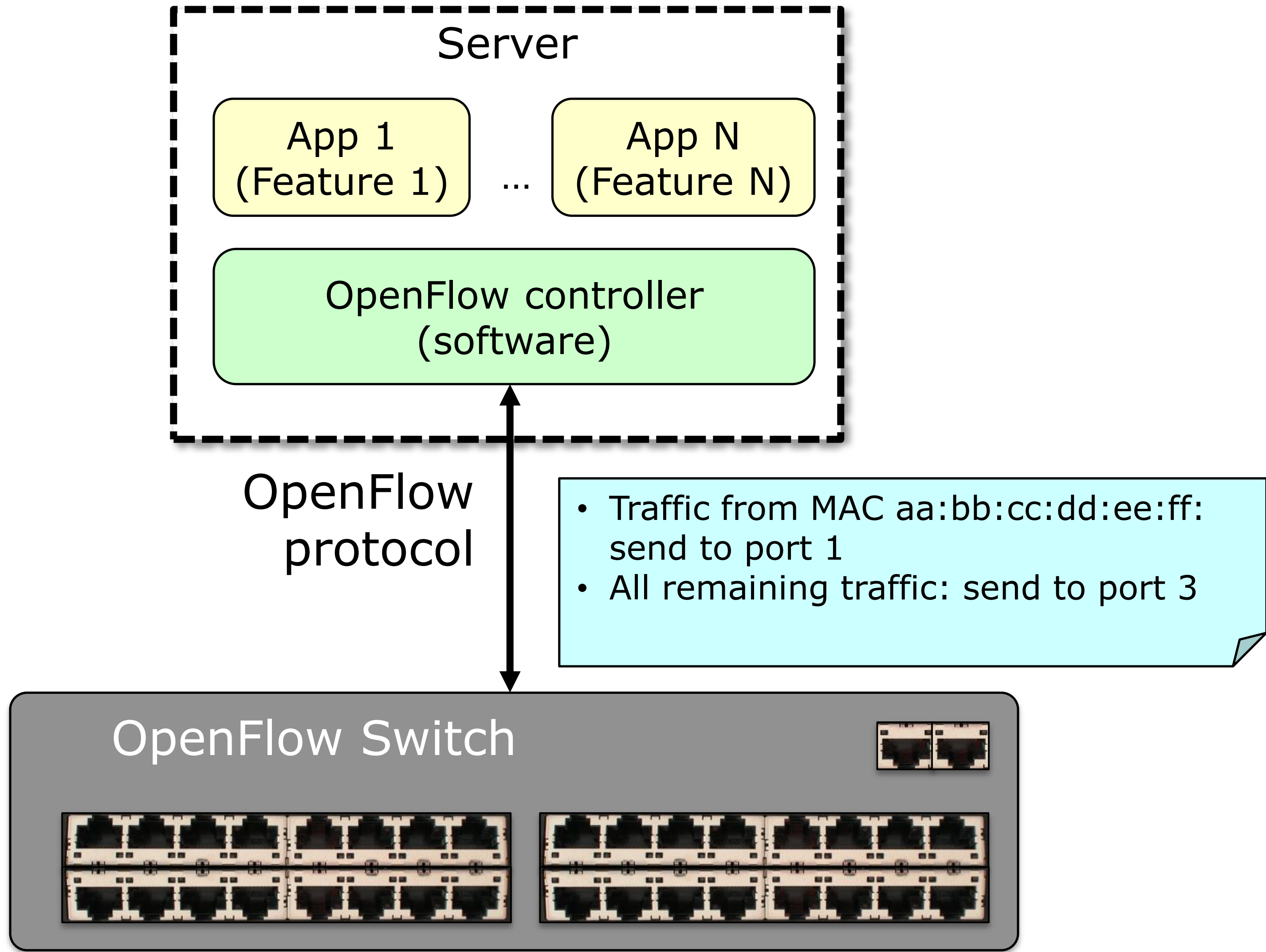
OpenFlow: a possible answer

- **The only test network large enough to evaluate future Internet technologies at scale, is the Internet itself.**
- This requires a new concept in our networks: **Slicing**
- OpenFlow launched around 2008 by McKeown, Peterson et al.
  - Ancestors (e.g., Ethane) published even earlier

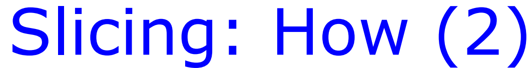




Slicing: How (1)



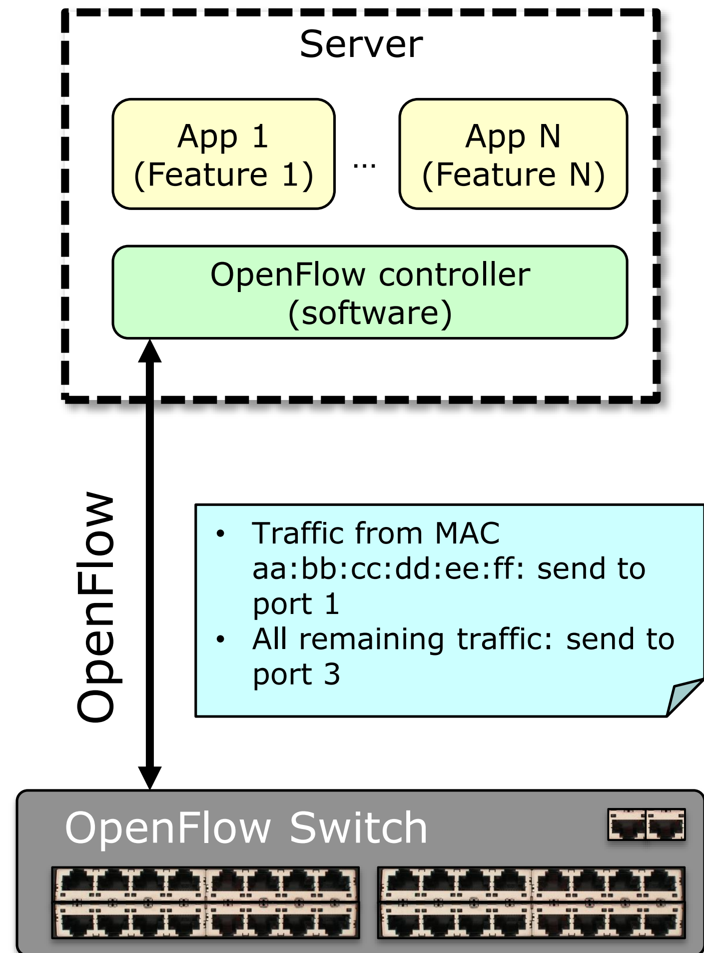
Slicing: How (2)



- An OpenFlow controller, running on an external server, is introduced in the architecture
- This can configure "flowrules", which tell the switch how to forward packets
- Did we solve the problem of having the same network used for production and experimentation traffic?
  - **YES**: now we can run experiments on real networks
  - **NO**: most likely, real users are still attached to the production network, so the traffic generated on the experimentation slide may not be so realistic

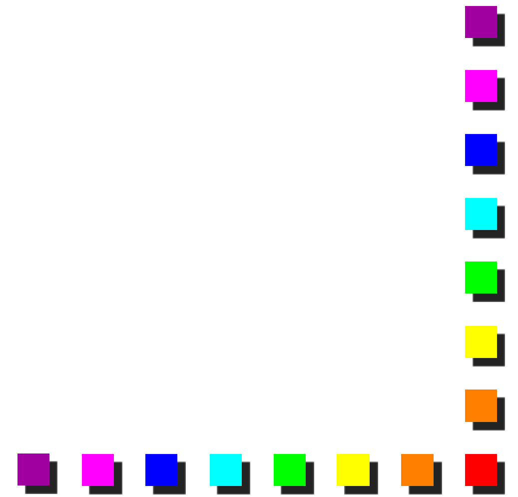
What is OpenFlow?

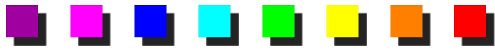
- OpenFlow is a **protocol**
- In a nutshell
  - Sends forwarding rules to the switch
  - Receives from the switch the packets that may not match any rule*
  - Can inject packets in the switch (e.g., the previous ones)
  - Asks for switch statistics (e.g., counters) and gets data back



* This is not 100% correct. More details later.

OpenFlow Basics



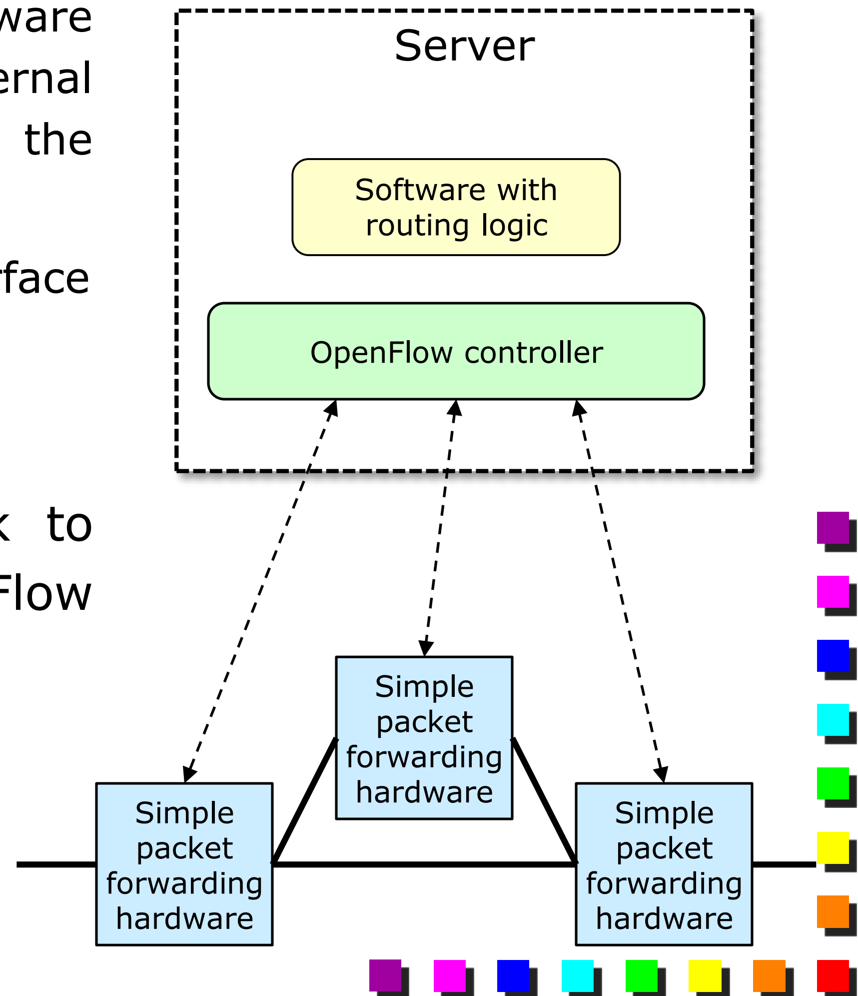


Four pillars behind OpenFlow

- (1) Separate control path from data path
- (2) Simple data path (“plumbing”)
- (3) Centralized control
  - At least logically
- (4) Context-based control path
  - Hence, also forwarding can become context-based

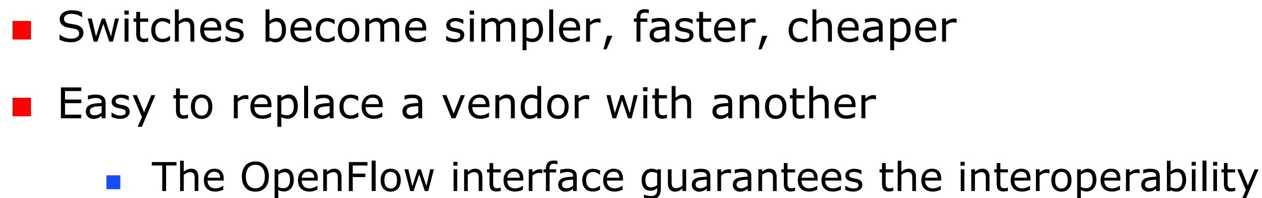
(1) Separate control from data path

- Arbitrary routing logic
  - Implemented in a software application that runs on an external controller that is separated by the data plane
  - Controller exports an open interface
- Simple forwarding switches
  - In principle, a lookup table
- Applications and switches talk to each other thanks to the OpenFlow protocol
  - Also known as **southbound interface**

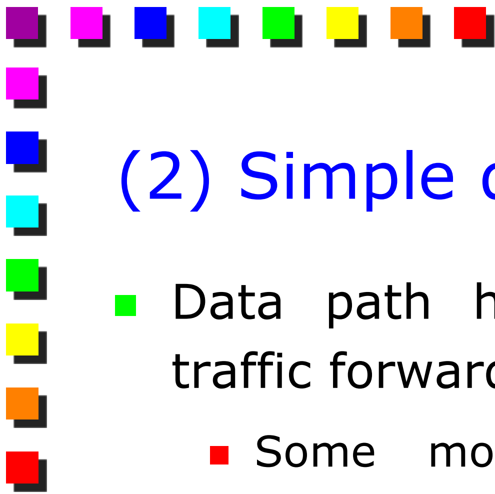


Advantages

- Control plane (routing logic)
  - Easy to change: new applications can be executed, replacing the original routing logic with some more sophisticated algorithms
  - Examples
    - Application 1: routing based on the IP paradigm (longest prefix match)
    - Application 2: sliced network sliced in two portion; pure IP routing on the first, label-switch–like routing in the second
    - Application 3: separate research from production network

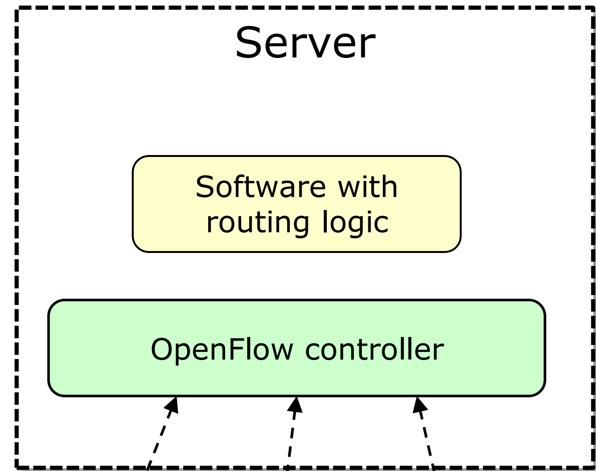


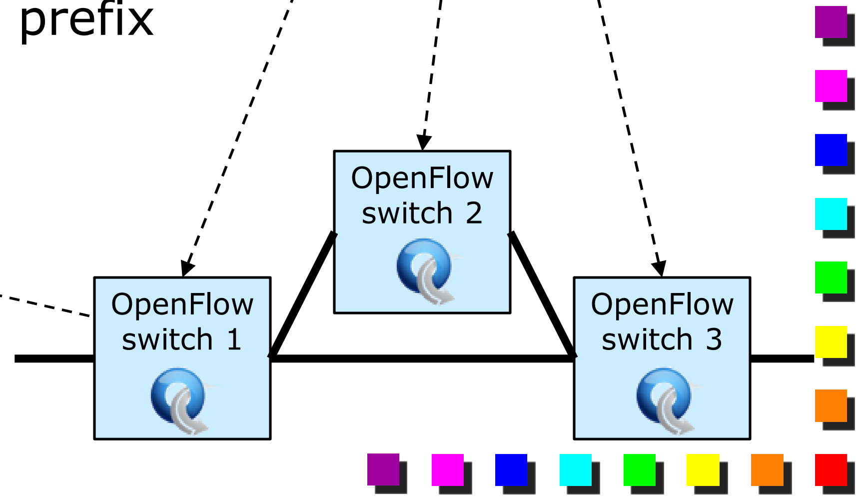
- Forwarding plane
  - Switches become simpler, faster, cheaper
  - Easy to replace a vendor with another
    - The OpenFlow interface guarantees the interoperability



(2) Simple data path ("plumbing")

- Data path has to perform **mainly** traffic forwarding
  - Some more powerful actions are defined (see later)
- Data path is mainly a way to create "network pipes" (hence "plumbing")
- Match/action couples can be used to create a bridge (matching MAC addresses), a router (longest prefix match), etc.



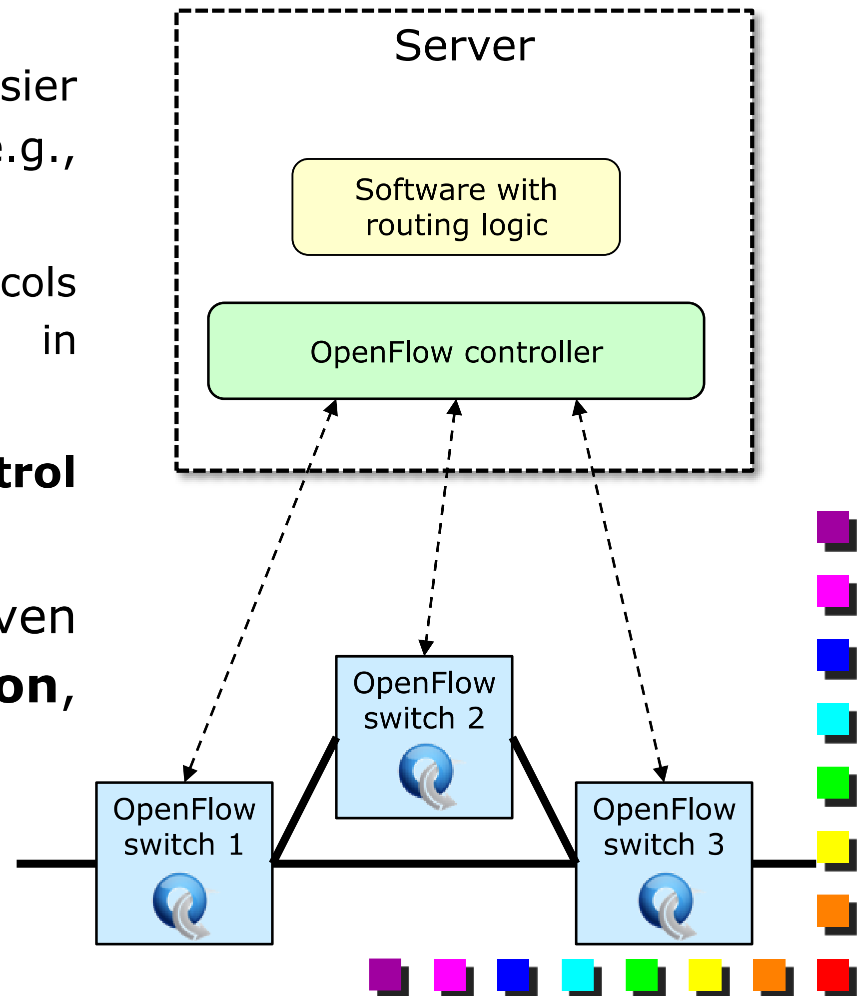


Forwarding table for openflow switch 1

```
If header = x, send to port 4
If header = y, send to ports 5,6
```

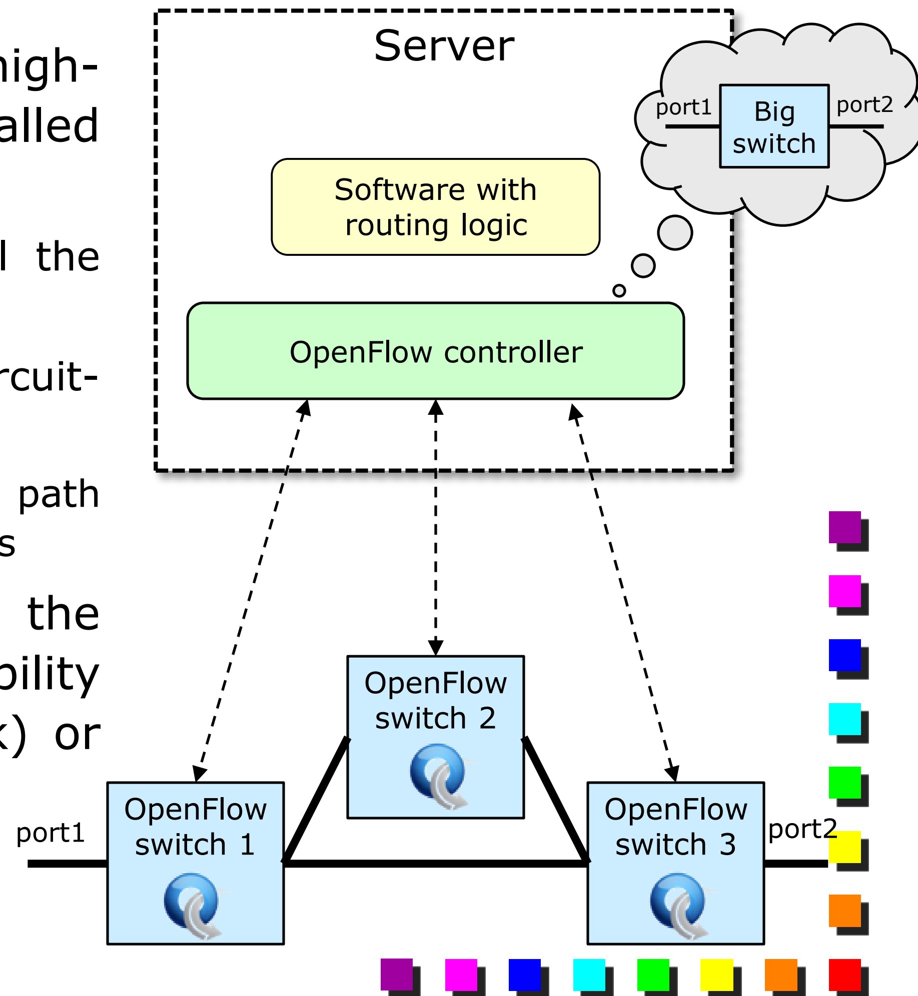
(3) Centralized control

- The software has a view of the **entire** network
  - Centralized control is usually easier than distributed control (e.g., current routing protocols)
    - Current routing protocols operate on each node in isolation
  - The software can easily **control** the **entire** network
- The OpenFlow controller can even export a **big switch abstraction**, if needed

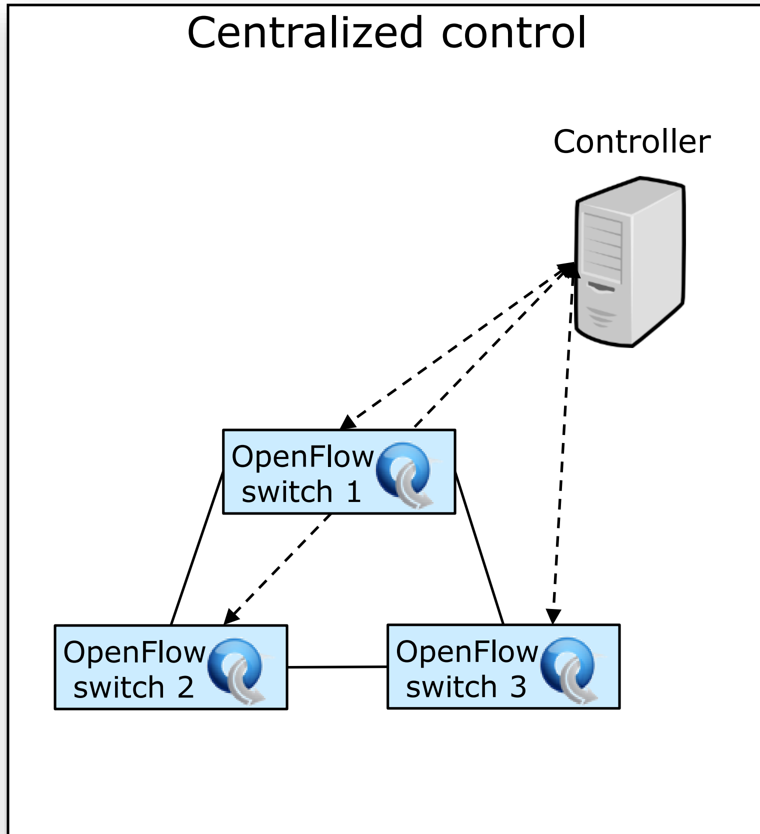


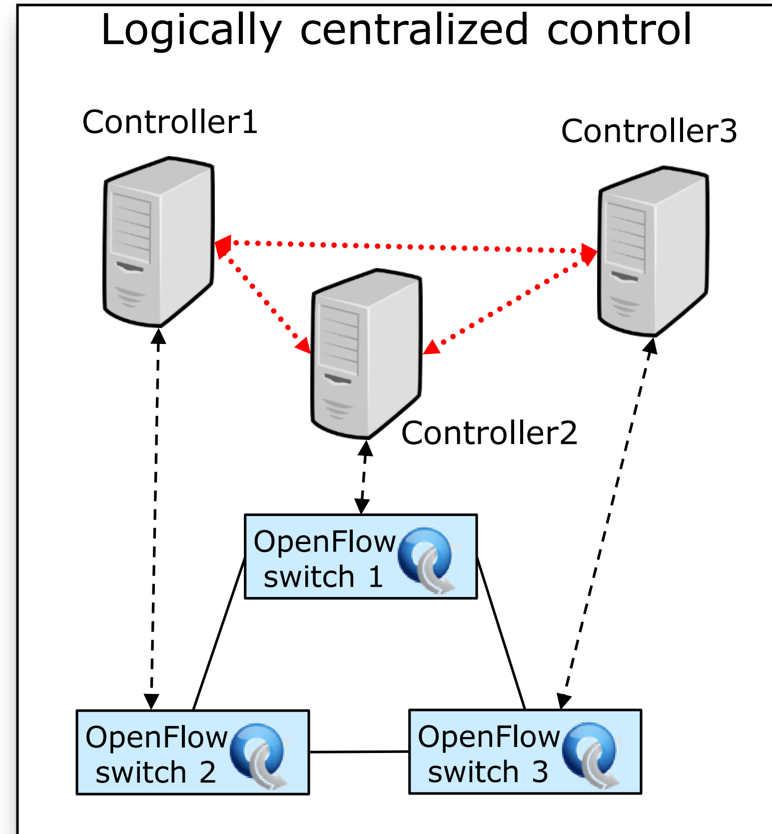
The "big switch" abstraction

- The big switch abstraction hides the internal details of the network
- This allow the software to set high-level commands (often called "intents"), such as:
  - Set a given configuration to all the **edge** ports
  - Create a direct path (e.g., a circuit-like) **from port A to port B**
    - OF controller will set up the path e.g., through a chains of VLANs
- The software can either use the **normal** view (that offer the visibility of the **internals** of the network) or **big switch** view



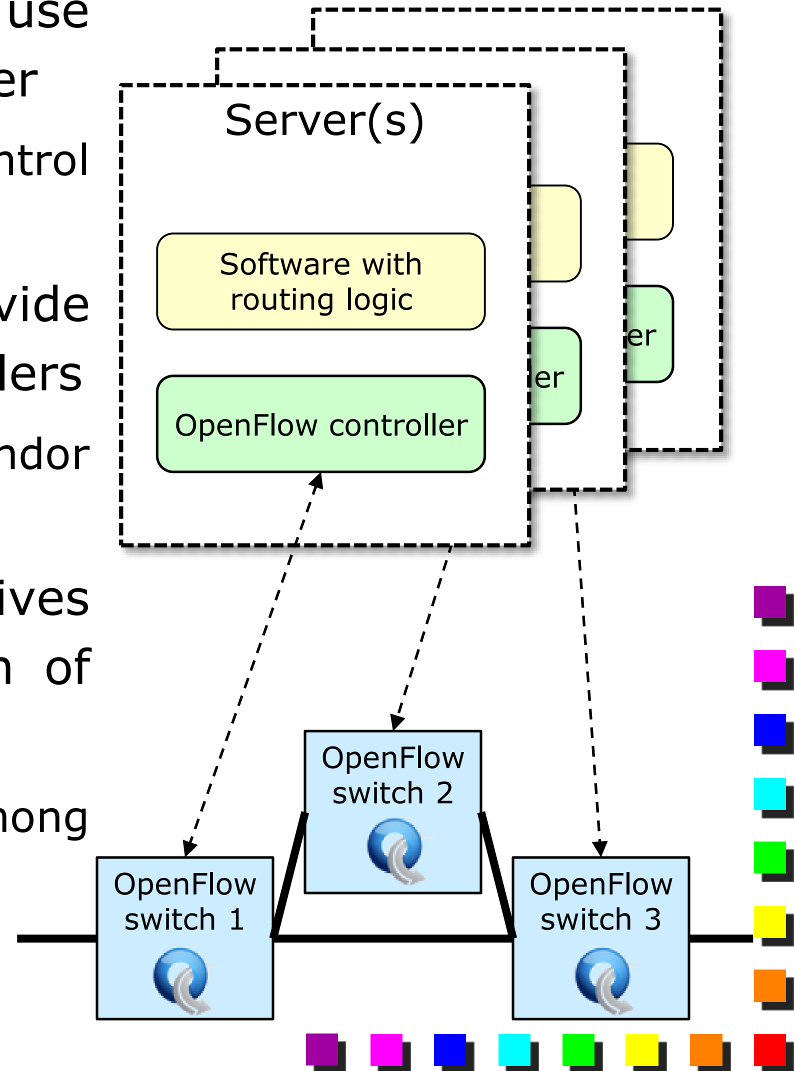
Centralized vs. Logically Centralized Control





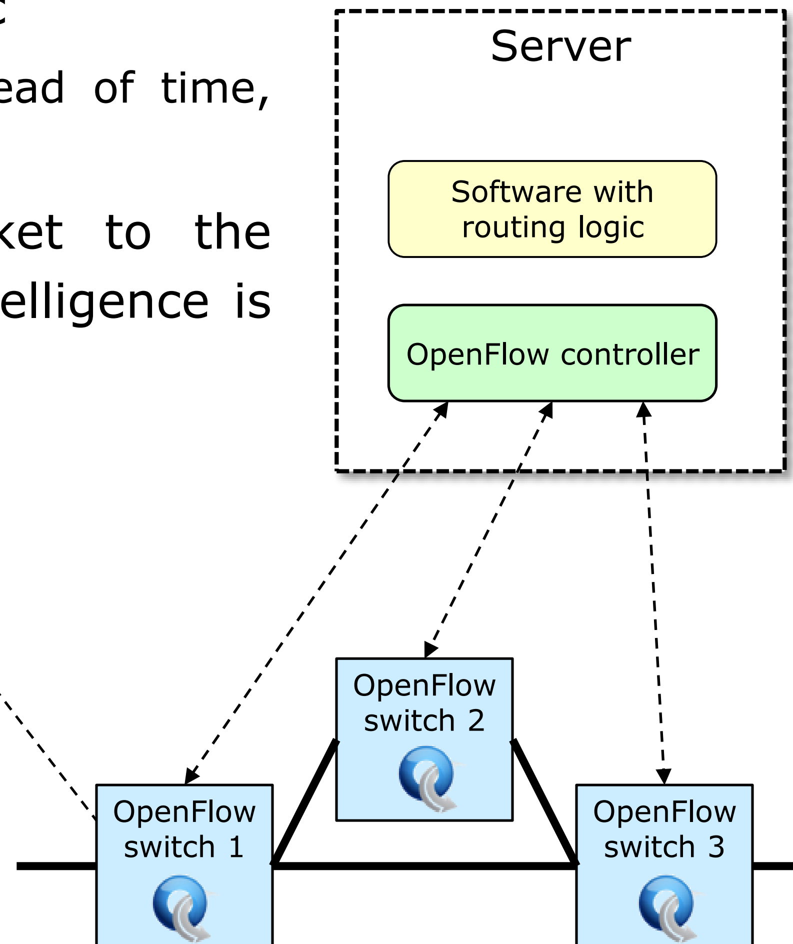
Centralized vs. Logically Centralized Control

- OpenFlow does not mandates the use of a physically centralized controller
  - In fact, a logically centralized control is possible as well
- However, OpenFlow does not provide primitives to **synchronize** controllers
  - Implementation-dependent (vendor lock-in)
- Controller should export primitives that facilitate the implementation of **distributed** applications
  - E.g., **data synchronization** among different instances



(4) Context-based control path

- The control path can take decisions at run-time, based on the actual traffic
  - In IP, decisions are taken ahead of time, based on the network topology
- The switch can send a packet to the controller when some more intelligence is needed to handle that traffic

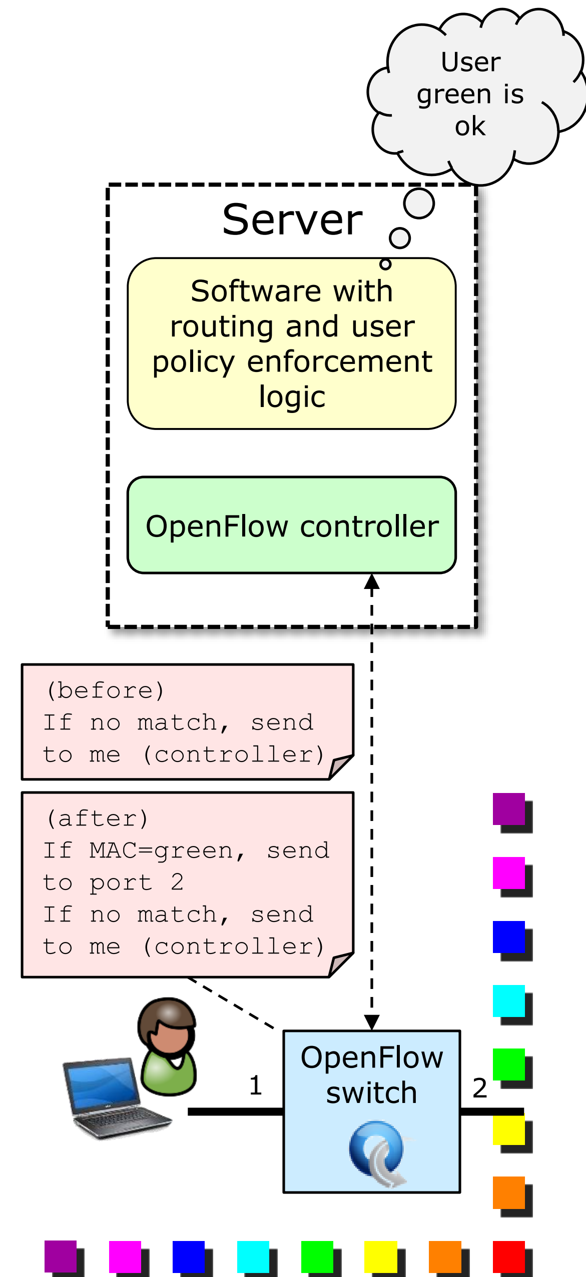


Forwarding table for openflow switch 1

```
If header = x, send to port 4
If header = y, send to ports 5,6
If header = z, send to me (controller)
If no match, send to me (controller)
```

Context-based control path

- When the switch sends a packet to the controller, the controller examines the packet and it may send it back with the proper forwarding decision, and may add new forwarding rules in the switch
  - E.g., dynamically recognizing new hosts connected to the switch and configuring the proper forwarding rules
- Hence, the controller
  - Can customize the forwarding rules based on the traffic itself
  - Can implement directly part of the data plane (when packet is sent back to the switch)
- The switch may act as a "cache" for the forwarding decisions

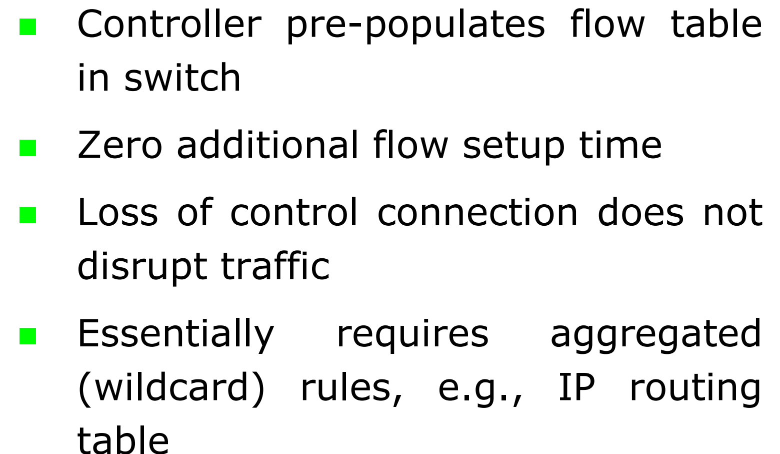


OF: Reactive vs. Proactive

- OpenFlow supports both models, depending on how we configure the rules on the switch

Reactive

- Some packet(s) are sent to the controller and trigger the insertion of new flow entries
- Efficient use of flow table
  - Only needed forwarding rules are configured
  - Switch look like a “cache” for forwarding rules
- When this happens, the traffic will experience a small additional delay due to the flow setup time
- If control connection lost, switch has limited utility



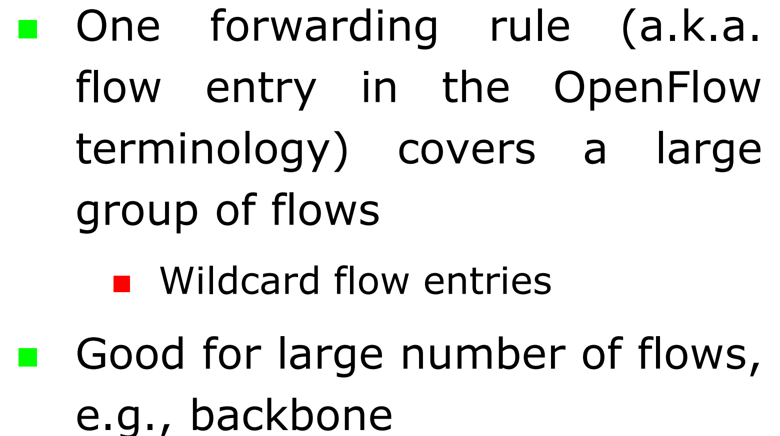
Proactive

- Controller pre-populates flow table in switch
- Zero additional flow setup time
- Loss of control connection does not disrupt traffic
- Essentially requires aggregated (wildcard) rules, e.g., IP routing table

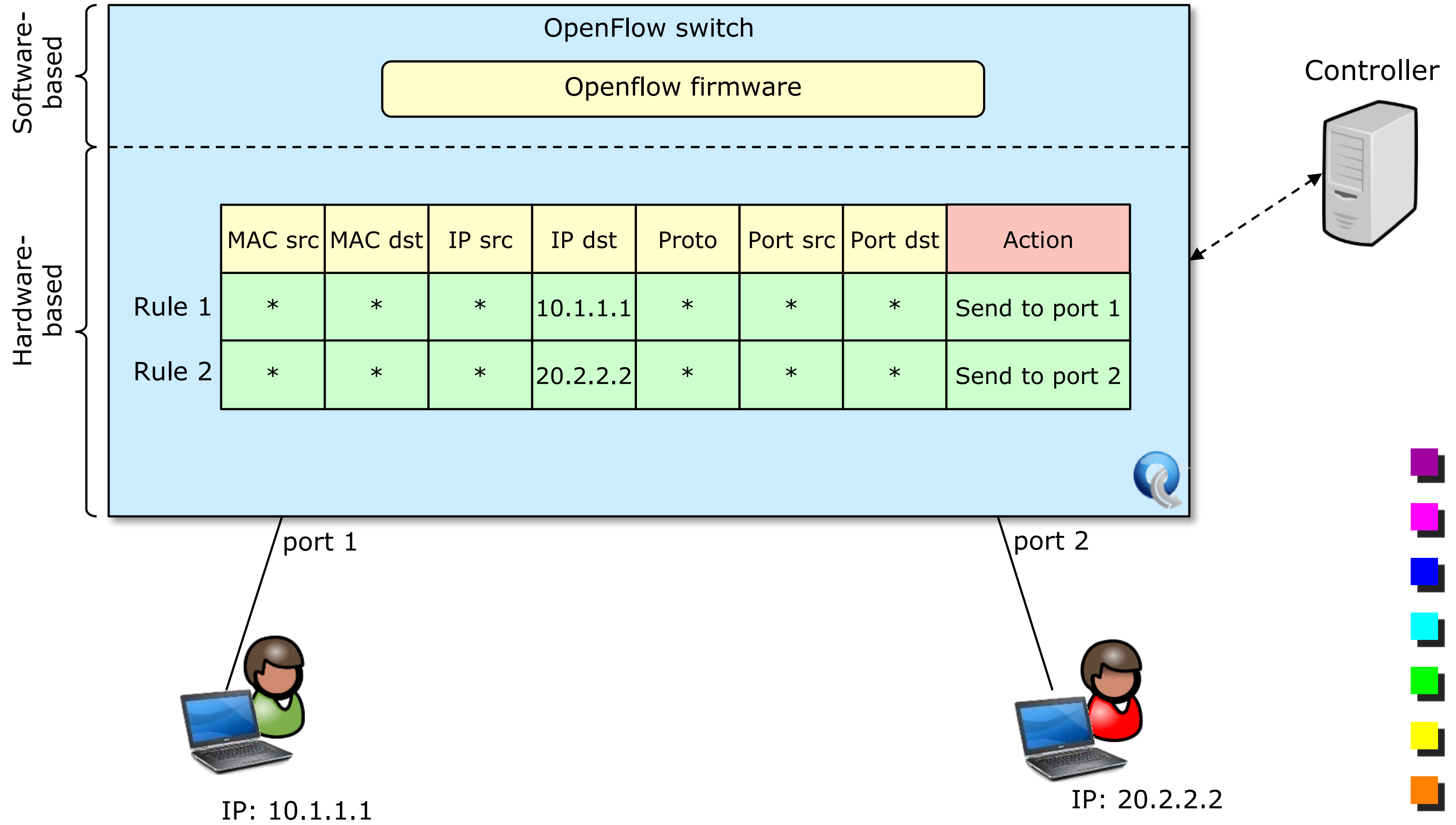
- OpenFlow supports both models; the latter is more appropriate when proactive control is used

Fine-grained

- Matching is done at a very fine granular level
  - E.g., the forwarding rule of each flow is individually set up by controller
- Exact-match flow entries
- Flow table contains one entry per flow
- Good for fine grain control, e.g. campus networks, datacenters

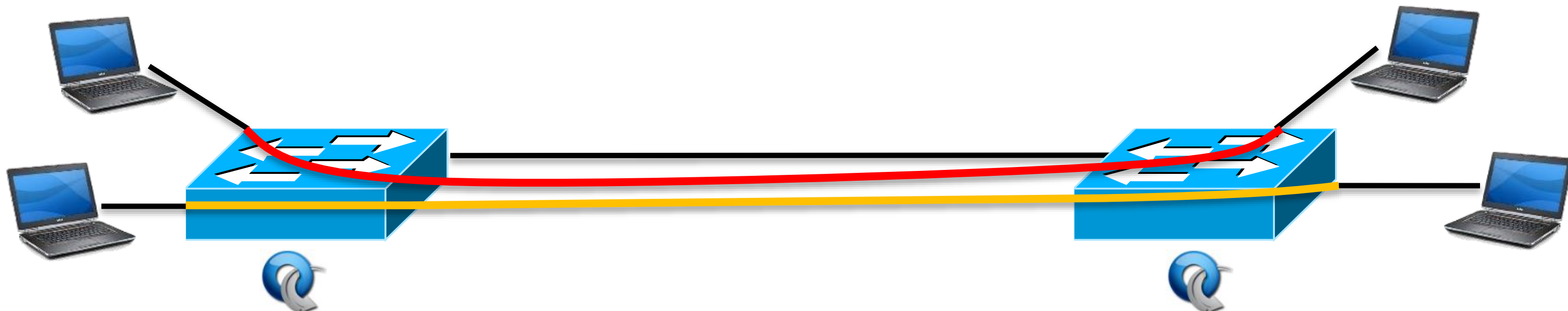


OpenFlow flow table example



OpenFlow terminology

- A **Flow Rule** is a traffic flow abstraction that:
  - Uses OF fields (actions and matches)
  - Can connect two arbitrary OF switch ports
  - May cross multiple OF switches (more than one OF Flowmod)
  - Is isolated from the other Flowrules
  - A FlowRule can be a flow table entry in a single switch
- **Flowmod**
  - A flowmod (OpenFlow flow modification) is an OF message used to instantiate, modify, delete a **flow rule** on a **single** switch



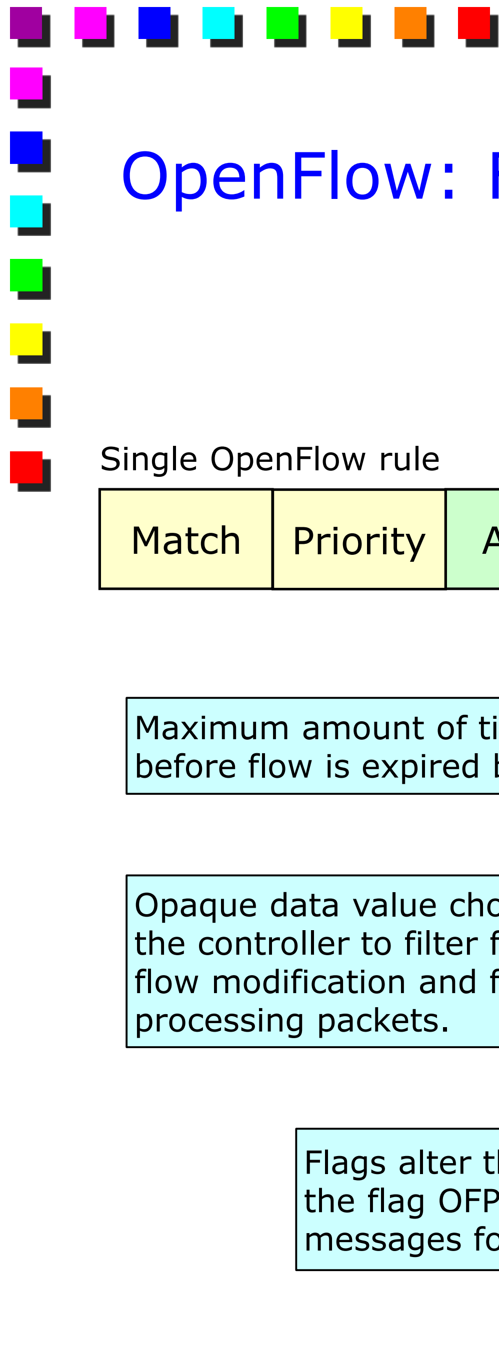
OpenFlow: Flow Table Entry (1)

- Plumbing primitive mainly based on the **<match, action>** tuple

Single OpenFlow rule

Match	Priority	Action(s)	Counters	Timeouts	Cookies	Flags

- Counters: Statistics (packet/byte counters), updated in case of match.
- Action(s): List of actions that are applied to the packet in case of matching, which modify the action set or pipeline processing.
- Priority: Matching precedence of the flow entry.
- Match: List of fields (with netmask) used to match 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.



OpenFlow: Flow Table Entry (2)

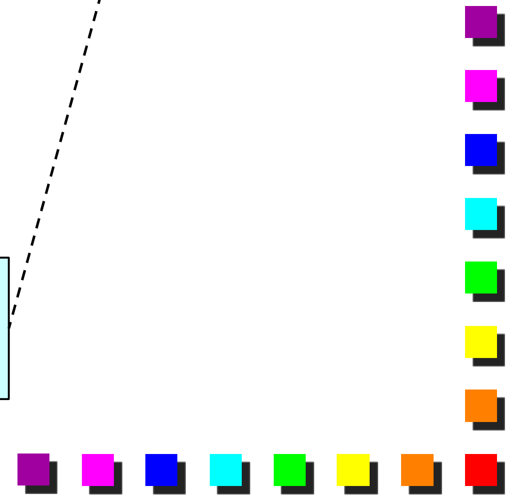
Single OpenFlow rule

Match	Priority	Action(s)	Counters	Timeouts	Cookies	Flags

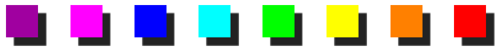
Maximum amount of time or idle time before flow is expired by the switch.

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 alter the way flow entries are managed, for example the flag OFPFF_SEND_FLOW_REM triggers flow removed messages for that flow entry.



- Match arbitrary bits in headers
  - Match on several L2-L4 headers, allowing any flow granularity
  - L7 fields not currently supported
  - Wildcards supported as well
- Examples
  - Input port of the switch
  - MAC source and destination address, VLAN tag
  - IP source and destination address, TCP/UDP ports, …
- Number of supported fields changes across different OpenFlow versions



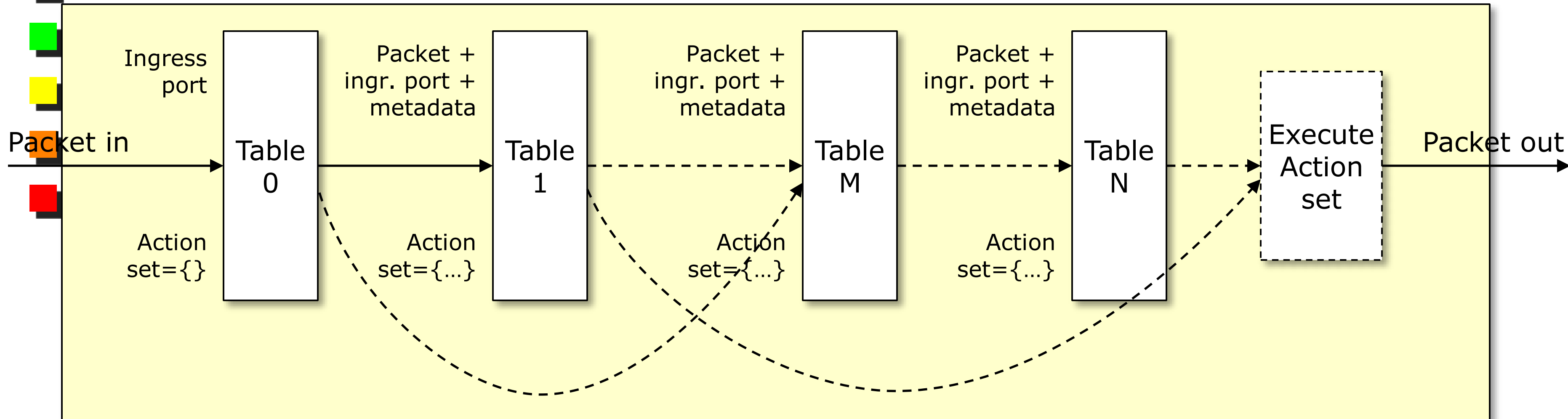
OpenFlow Flow Table Entry: Priority

- The matching process starts with the flowrules at the highest priority
- If no matches are found, we start analyzing the rules at the (highest-1) priority
- The process continues until a match is found
- 16 bit value

OpenFlow Flow Table Entry: action(s)

- Most common actions
  - Forward
    - Forward to port(s)
    - Encapsulate and send to controller
    - Send to normal processing pipeline (e.g., L2 bridging process)
  - Drop
  - Overwrite header field
    - E.g., replace source IP 10.0.0.1 with 20.0.0.2
  - Push or pop fields
    - E.g., add / remove a VLAN tag
    - The same for GRE keys, MPLS labels…
  - Forward at specific bit-rate
- Multiple actions can be supported in the same match

OpenFlow 1.2/1.3



- Multiple tables, not just a simple pipeline
- *Each* flow in table M can be redirected to another table N (M<N) (i.e., no loops in the pipeline)
- Some actions (e.g., send to controller, or drop) can be applied immediately after each table; other are added to the action set
- At the end of the pipeline, the action set is executed in a predefined order (not in the order the action set was created)
- Is it a good idea? More complicated than OF 1.0, while it does not solve the problem of creating a suitable abstraction of the network device

Deployment and discussion

- Typical deployment:
  - Traditional SDN networks: flow-based, centralized controller, reactive control
  - OpenFlow in NFV services: often operated on (a) aggregated traffic and (b) with a mixture of proactive/reactive control
  - Nothing prevents from using this protocol in a different way (e.g., distributed controllers, proactive control, etc)
- Where is being deployed
  - Limited deployment on real networks
  - Very much used in virtualized environment (software switches)

OpenFlow goals (now and back in 2008/9)

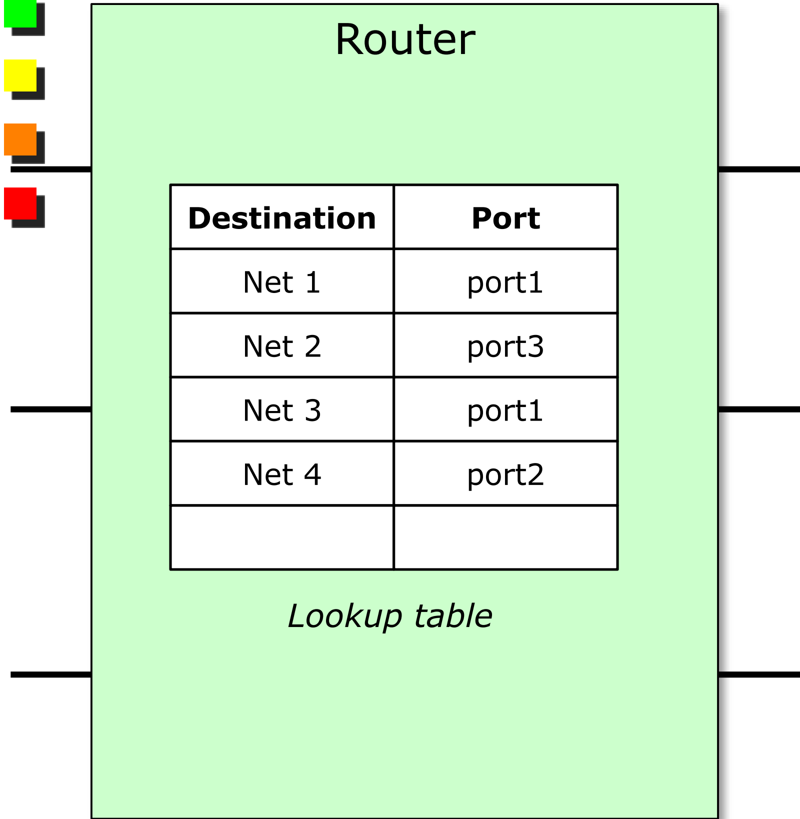
- OpenFlow started as a way to create **your own network paths**, through a software interface
  - OpenFlow 1.0, mainly forwarding
- Soon, people recognized that forwarding was **one** of the issues
  - The backbone of the network is in fact mainly forwarding packets
  - However, the edge has to deal with much more applications
    - NAT, firewall, PPPoe encapsulation/decapsulation, access control, etc
- OpenFlow started to introduce new primitives to support them
  - Modify fields, multiple tables, etc.
- Is it the right way to go?

Openflow limitations

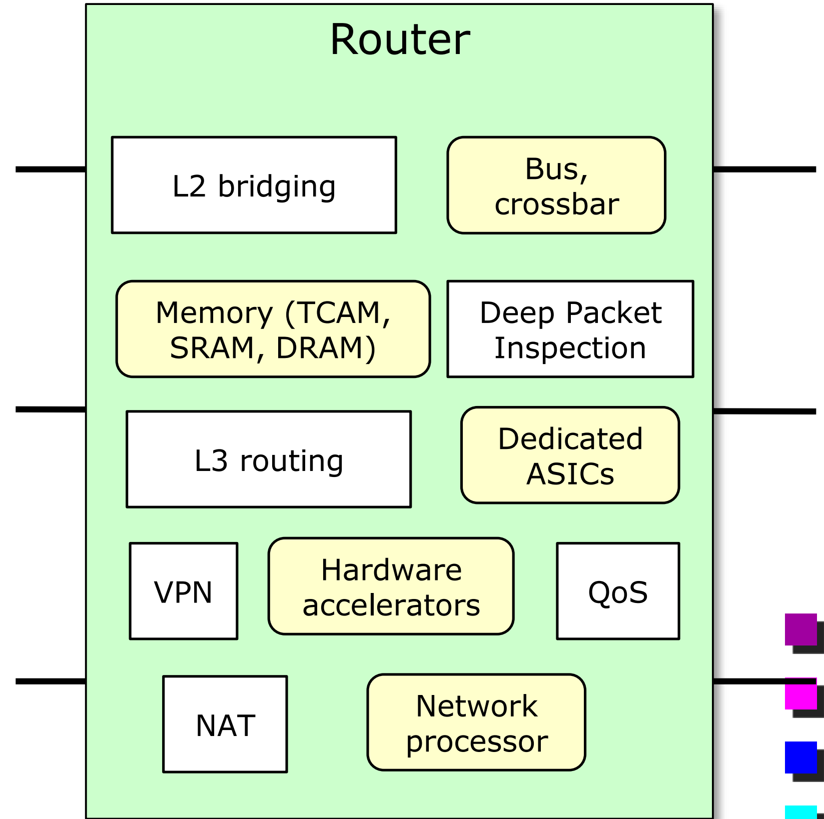
- Good for separation of control and data planes
  - For instance, this is something already existed for years (the separation between supervisor/linecards is not that much different)

- Not good enough for full data plane abstraction
  - Too simple switch model
  - Only tables
  - What about specialized hardware (encryption, content matching, etc?)
  - What about complex applications?

OpenFlow and computing (2)



The view of a router from a **Network** perspective



The view of a router from a **Computing** perspective

OpenFlow and computing (2)

- Hard to define a suitable southbound API that will accommodate all the computing needs
  - At that point, OpenFlow should support generic code (e.g., x86 instructions) in order to implement all the possible instructions
  - What about "simple forwarding elements"?
- Current status so far
  - Added new features, actions, tables: very hard to offer a unified programming model to them
    - Also because different tables, in the hardware, have different capabilities (e.g., a table T1 can support only MAC addresses, etc, and the software must be properly mapped on it)
  - New proposals: P4 and eBPF

P4

- P4 is a programming language designed to allow programming of packet forwarding dataplanes
- In contrast to a general purpose language such as C or python, P4 is a domain-specific language with a number of constructs optimized around network data forwarding
- P4 is an open-source, permissively-licensed language and is maintained by a non-profit organization called the P4 Language Consortium
- Priginally described in a 2014 paper titled “Programming Protocol-Independent Packet Processors” hence “P4”

(Source: Wikipedia)

P4: concepts

- P4 is based on the concept of **match-action pipelines**.
- Conceptually, forwarding network packets or frames can be broken down into a series of table lookups and corresponding header manipulations
- In P4 these manipulations are known as “actions” and generally consist of things such as copying byte fields from one location to another based on the lookup results on learned forwarding state.
- P4 addresses only the data plane of a packet forwarding device, it does not specify the control plane nor any exact protocol for communicating state between the control and data planes.
- Instead, P4 uses the concept of tables to represent forwarding plane state. An interface between the control plane and the various P4 tables must be provided to allow the control plane to inject/modify state in the program.

(Source: Wikipedia)

eBPF

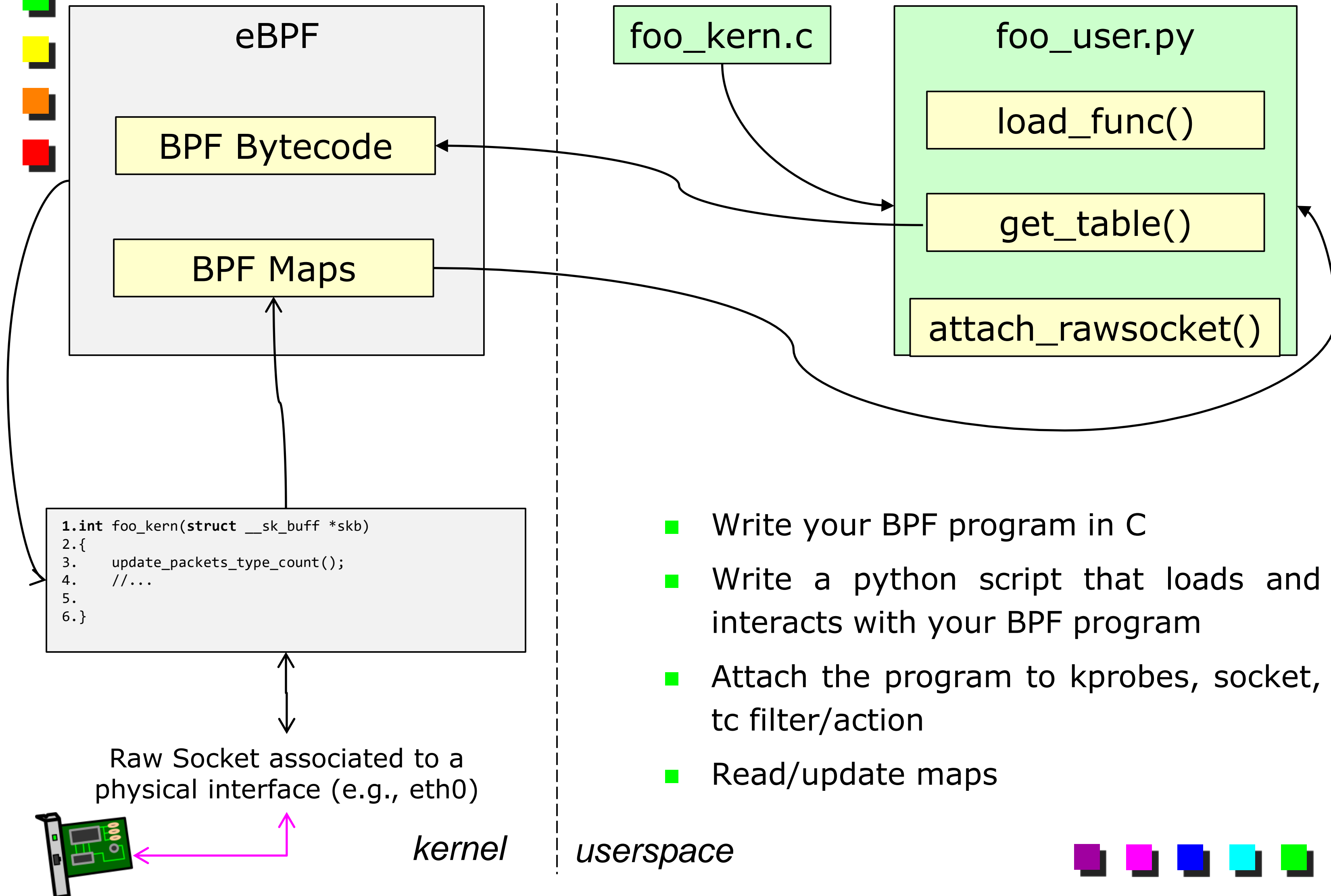
- BPF (classic) - Berkeley Packet Filter
  - Initially used as socket filter by packet capture tool tcpdump (via libpcap)
  - Introduced in Linux in 1997 in kernel version 2.1.75
- Use cases:
  - Mainly socket filters (drop or trim packet and pass to user space)
  - Used by tcpdump/libpcap, wireshark, nmap, dhcp ..
- bBPF improve and extend existing BPF infrastructure
  - Programs can be written in C and translated into eBPF.
  - New set of patches added to the Linux kernel since 3.15 (2014)
- Use Cases:
  - Networking (packet filtering , network traffic control, etc ...)
  - Tracing (analytics, monitoring, debugging)

BPF vs eBPF - Extended BPF

- Idea: improve and extend existing BPF infrastructure.
- Programs can be written in C and translated into eBPF.
- New set of patches introduced in the Linux kernel since 3.15 (June 8th, 2014).
- Current Linux kernel version 4.3.99 (November 2015).

- Use Cases:
- Networking (packet filtrering , network traffic control, etc ...)
- Tracing (analytics, monitoring, debugging)

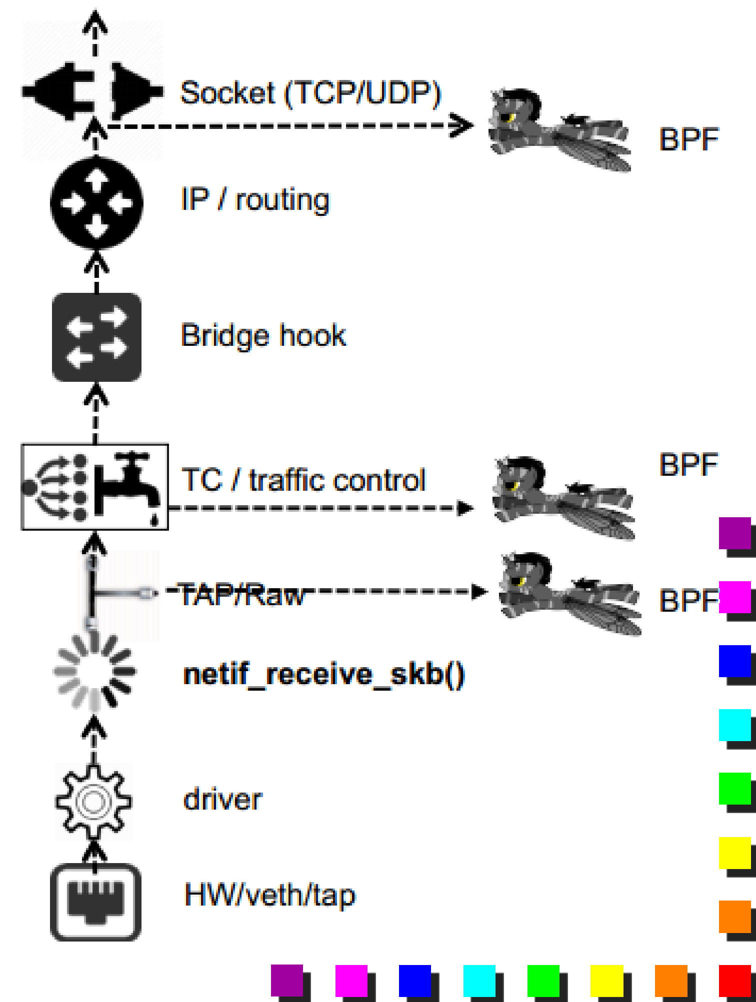
BCC - eBPF Compiler Collection



- Write your BPF program in C
- Write a python script that loads and interacts with your BPF program
- Attach the program to kprobes, socket, tc filter/action
- Read/update maps

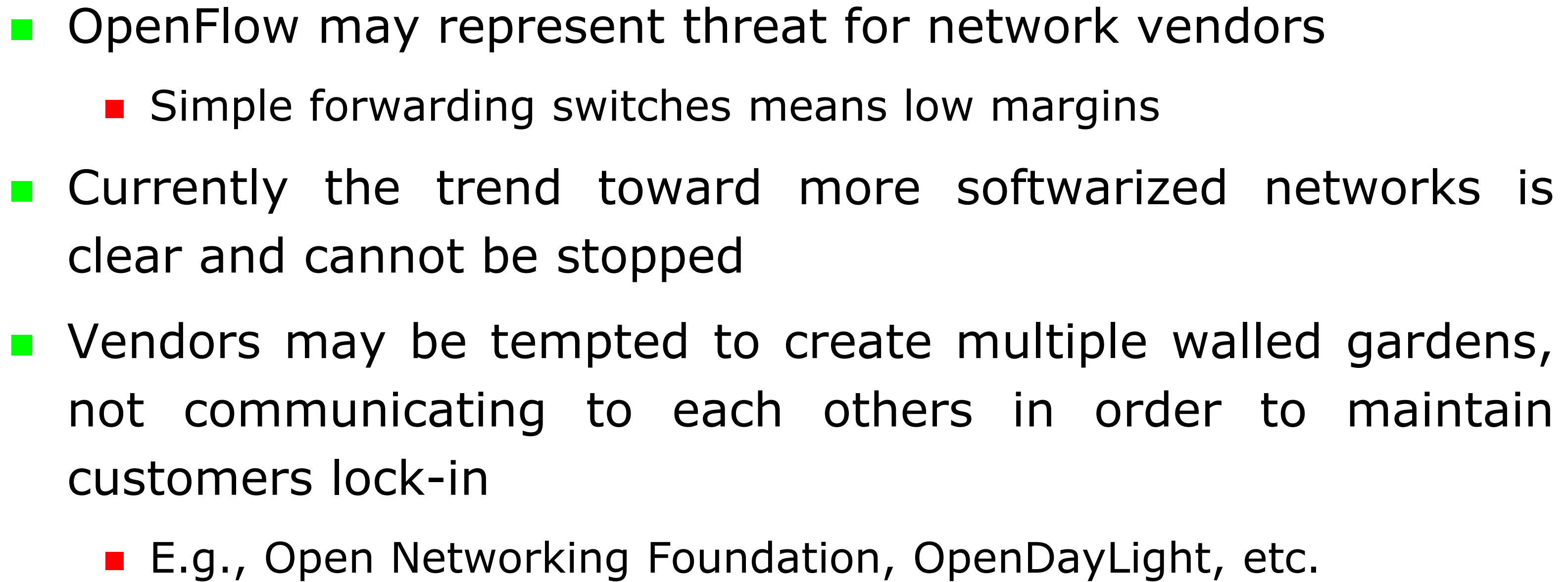
eBPF & Networking in the Linux networking stack

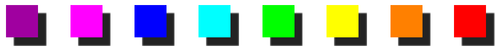
- BPF programs can attach to sockets or the traffic control (TC) subsystem, kprobes, syscalls, tracepoints ...
- sockets: STREAM (L4/TCP), DATAGRAM (L4/UDP) or RAW (TC)
- This allows to hook at different levels of the Linux networking stack, providing the ability to act on traffic that has or hasn't been processed already by other pieces of the stack
- Opens up the possibility to implement network functions at different layers of the stack



- SDN is a broader concept, and in particular
  - SDN != OpenFlow
  - SDN is *what* do to, OpenFlow is *how* to do
- OpenFlow represents a way to implement the **southbound** interface of SDN
- SDN is about *network programmability*
  - Contrast device programmability
- One of the hard problems with SDN is finding the appropriate abstractions and APIs

OpenFlow economics


OpenFlow may represent threat for network vendors
Simple forwarding switches means low margins
Currently the trend toward more softwarized networks is clear and cannot be stopped
Vendors may be tempted to create multiple walled gardens, not communicating to each others in order to maintain customers lock-in
E.g., Open Networking Foundation, OpenDayLight, etc.



Conclusions

- OpenFlow is here to stay
- So far, more important in case of software environments (virtualized networks) than real (hardware) networks
- NFV may be a good complement to OpenFlow