From dumb to smarter switches in software defined networks: an overview of data plane evolution

Antonio Capone, Carmelo Cascone

Politecnico di Milano – ANTLab
Ecole Polytechnique de Montreal
Acknowledgements

This tutorial has been supported by the EU project BEBA

The authors wish to thank:
- prof. Giuseppe Bianchi and Marco Bonola (Univ. of Rome, Tor Vergata)
- Luca Pollini and Davide Sanvito (Politecnico di Milano)
Agenda

1) **Setting the scene:** a brief intro to SDN and OpenFlow

2) **Switches cannot remain dumb:** Starting the process of data plane evolution

3) **Not too much not too little:** OpenState and statefull data planes

4) **Applied smartness:** statefull applications (with hands-on activities)
Setting the scene: a brief intro to SDN and OpenFlow

The future has already arrived. It's just not evenly distributed yet. [William Gibson]
Classic network paradigm

Distributed network functions

State distribution mechanism (protocols)

Router/switch/appliance

A. Capone & C. Cascone: SDN tutorial
Vertically integrated

Protocols guarantee interoperability...
But what’s the drawback?

A. Capone & C. Cascone: SDN tutorial
Way too many standards?

Source: IETF
**How Standards Proliferate:**
(see: A/C chargers, character encodings, instant messaging, etc)

**Situation:**
There are 14 competing standards.

14?! Ridiculous! We need to develop one universal standard that covers everyone's use cases. Yeah!

**Soon:**

**Situation:**
There are 15 competing standards.
Vendors dominated?

Source: IETF

A. Capone & C. Cascone: SDN tutorial
Non-standard management

• **Configuration interfaces** vary across:
  – Different vendors
  – Different devices of same vendor
  – Different firmware versions of same device!

• **SNMP fail**
  – Proliferation of non-standard MIBs
  – Partially implemented standard MIBS
  – IETF recently published a recommendation to stop producing writable MIB modules
Evolution of the infrastructure

Server

- x86, Linux, hypervisors, cloud, open-source, etc.

Storage

- Scale out, flash, thin provisioning, object storage, etc.

Networking

- 7200+ RFCs
- Mainframe hardware
- Software integration
- Extremely expensive
- Protocols development needs too long + adoption times

A. Capone & C. Cascone: SDN tutorial
The (new) paradigm

Traditional networking

Software-Defined Networking

Switch

Programmable switch
SDN architecture

Network OS

Network control API

HW open interface

Simple forwarding
HW

Simple forwarding
HW

Simple forwarding
HW

Simple forwarding
HW

App

App

App
From protocols to API

• HW forwarding abstraction
  – low-level primitives to describe packet forwarding

• Control plane API
  – Network topology abstraction
  – High-level access to switch programming
  – Common libraries
    • Host tracking
    • Shortest-path
    • Etc..
Success keys

• Low-level HW open interface (but not too low level ...)

• Good, extensible and possibly open-source Network OS (too many?)

• Open market for third-party network application developers (development ecosystem is crucial)
  – Network app store
Several attempts ...

Active Networks, IETF ForCES, ...
Active networking

• “The goal for active networking is to have programmable open nodes, with the ability to deploy programs dynamically into node engines.”

Capsule model

Programmable router model
ForCES Architecture

- **ForCES - Forwarding and Control Element Separation**
- IETF ForCES working group was established in 2001 and closed in 2014
- RFC3746: “ForCES Framework” defines
  - CE : Control Element
  - FE : Forwarding Element
    - CE may be required to control hundreds of FEs
ForCES Architecture - FE

FE Model

- ForCES Protocol
  - To provide a universal standardized control interface for FEs
- LFB – Logical Functional Block
  - e.g., Classifier LFB, IPv4 LPF LFB, IPv6 LPF LFB, Scheduler LFB
- Datapath
  - Can configure dynamically LFB topology for supporting various over IP services
... but one winner

OpenFlow
OpenFlow

• Stanford, 2008
• Clean Slate research program
• “With what we know today, if we were to start again with a clean slate, how would we design a global communications infrastructure?”

Is it really a clean slate approach?
OpenFlow

- OpenFlow is actually a **pragmatic approach** to SDN based on a simple HW abstraction that can be implemented with current HW commercial platforms.
OpenFlow
What is OpenFlow

• Switch abstraction
  – Match/action **flow table**
  – Flow counters
  – It doesn’t describe how this should be implemented in switches (**vendor neutral !!!**)

• Application layer protocol
  – Binary wire protocol, messages to program the flow table

• Transport protocol
  – TCP, TLS
Flow table

1. Forward (one or more ports)
2. Drop
3. Encapsulate and send to controller
4. Header rewrite
5. Push/pop MPLS label / VLAN tag
6. Queues + bitrate limiter (bit/s)
7. Etc..

Switch Port | VLAN ID | VLAN pcp | MAC src | MAC dst | Eth type | IP Src | IP Dst | IP ToS | IP Prot | L4 sport | L4 dport
--- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | ---

Slide courtesy: Rob Sherwood

A. Capone & C. Cascone: SDN tutorial
Switch abstraction

Software

OpenFlow client

Hardware (e.g. TCAM) or software

Flow table (aka Forwarding Information Base)
## Example

<table>
<thead>
<tr>
<th>Description</th>
<th>Port</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>VLAN ID</th>
<th>IP Src</th>
<th>IP Dest</th>
<th>TCP sport</th>
<th>TCP dport</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 switching</td>
<td>*</td>
<td>*</td>
<td>00:1f..</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Port6</td>
</tr>
<tr>
<td>L3 routing</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>5.6.*..</td>
<td>*</td>
<td>*</td>
<td>Port6</td>
</tr>
<tr>
<td>Micro-flow handling</td>
<td>3</td>
<td>00:20..</td>
<td>00:1f..</td>
<td>0x800</td>
<td>Vlan1</td>
<td>1.2.3.4</td>
<td>5.6.7.8</td>
<td>4</td>
<td>17264</td>
<td>Port4</td>
</tr>
<tr>
<td>Firewall</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>22</td>
<td>Drop</td>
</tr>
<tr>
<td>VLAN switching</td>
<td>*</td>
<td>*</td>
<td>00:1f..</td>
<td>*</td>
<td>Vlan1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Port6, port7, port8</td>
</tr>
</tbody>
</table>
Reactive vs Proactive

• **Reactive**
  – Start with flow table empty
  – First packet of a flow sent to controller
  – Controller install flow entries
  – Good for stateful forwarding:
    • L2 switching, dynamic firewall, resource management

• **Proactive**
  – Flow entries installed at switch boot
  – Good for stateless forwarding:
    • L3 routing, static firewall, etc..
OpenFlow 1.0 recap

But how do we implement a flow table?
HW vs SW switches

- Soft switching based on general CPU
- Switching based on network processors
- Switching based on dedicated chips

- 1 Tb/s capacity is the target switching capacity for top level devices
- The differences in performance between HW and SW switches are expected to remain
- But the application domain of soft switches will expand in scenarios where extremely high rates are not required
HW vs SW switches

• OF has been designed having in mind current specialized HW architecture for switches
• Even if it has greatly stimulated the evolution of soft switches and extended their use
• OF doesn’t describe how the forwarding abstraction should be implemented in switches (vendor neutral !!!)

– but what does this really mean? (…)

A. Capone & C. Cascone: SDN tutorial
Forwarding Abstraction OF1.0

SMT - Single Match Table

TCAM
Ternary Content Addressable Memory
Limitations of SMT abstraction

• SMT is a simple and powerful abstraction, but the option to implement it using a single TCAM is not practical
• A very big and wide TCAM would be necessary in most cases
  – Wide: all header fields
  – Big: all possible combinations of values relevant

– the distance between abstraction and implementations is at the basis of the OF success
– but also an incentive to try to make the abstraction evolve and become more flexible (…)

A. Capone & C. Cascone: SDN tutorial
Other (less known) sides of OF

• Before moving to the evolution of the forwarding abstraction it is worth pointing out some other characteristics of OF

• OF focuses only on the control of the data plane

• However its relation with network management and configuration is within an area which is not completely black-and-white
Control plane vs Management plane?

• Another important practical issue is the relation of OF and the controller with the network management platforms
• In principle controllers could take over the management functions (mainly device configuration and monitoring)
• This however requires more complex functions and a number of legacy management services that are still crucial for networks
• If we take a look at a popular SDN controller (OpenDayLight) architecture, it is easy to understand things are more complex than they appear
Network OS / controller example:
OpenDaylight (Linux Foundation)
OF-CONFIG

- ONF defined its own configuration protocol based on NETCONF and xml data models.

![Diagram of OF-CONFIG and its components showing OpenFlow Capable Switch, OpenFlow, OF Logical Switch, and Configuration Point using IETF Netconf & XML data models.](image-url)
SDN Layers and Architecture Terminology

- IETF is trying to clean up terminology and architecture and reconcile it with real schemes.

Source: IRTF Software-Defined Networking Research Group (SDNRG)
**Switches cannot remain dumb:**
Starting the process of data plane evolution

One man alone can be pretty dumb sometimes, but for real bona fide stupidity, there ain't nothin' can beat teamwork.
[Edward Abbey]
Models can be perfect and clean, reality is dirty!

- The match/action model can ideally be used to program any network behavior and to get rid of protocol limitations at any level.
- But unfortunately, with OF:
  - Matches can be done only on a set of predefined header fields (Ethernet, IPv4, MPLS, VLAN tag, etc.)
  - Actions are limited to a rather small set
  - Header manipulation (like adding label/tags, rewriting of fields, etc.) is limited to standard schemes
- As a result, OF is not really protocol independent and standards (including OF standards) are still necessary.
Where do OF limitations come from?

- TCAMs are typically expensive components that are used by manufacturers only when strictly necessary.
- Less expensive memory components based on predefined search keys are often used for most of the common functions of a switch.
- OF success depends on its “vendor neutral” approach where implementations issues are completely opaque (including reuse of standard modules for e.g. MAC and IP forwarding).
- Specialized ASICs are typically complex with a number of hard limitations on table types, sizes, and match depth.
Multiple Match Tables (MMT)

• Single Match tables are costly: all possible combinations of header values in a single long table
• Solution: **Multiple Match Tables (MMT)**
• MMTs are the HAL of OF 1.1

![Flowchart diagram](diagram.png)
MMT and implementations

- MMT introduced in OF 1.1 are actually much closer to real switch implementation in specialized chips
Switch pipeline

• Existing switch chips implement a small (4–8) number of tables whose widths, depths, and execution order are set when the chip is fabricated

• Optimization of the pipeline can lead to very different results depending on the context:
  – A chip used for a core router may require a very large 32-bit IP longest matching table and a small 128 bit ACL match table;
  – A chip used for an L2 bridge may wish to have a 48-bit destination MAC address match table and a second 48-bit source MAC address learning table;
  – an enterprise router may wish to have a smaller 32-bit IP prefix table and a much larger ACL table as well as some MAC address match tables.

Group Tables (OF 1.1)

- Packets of the same flow are applied the same actions unless the table entry is modified by the controller.
- Not good for some common and important cases (e.g. multicast, multipath load balancing, failure reaction, etc.).
- **Solution:** **Group tables**
  - Goto table “group table n”
  - List of buckets of actions
  - All or some of the buckets are executed depending on the type.
- **Types** of Group tables
  - All (multicast)
  - Select (multipath)
  - Fast-failover (protection switching)
Group Tables (OF 1.1)

- **Fast failover**
- Note that this is the first “stateful” behavior in the data plane introduced in OF!!!
OF 1.2

• **Extensible match** (Type Length Value)
• Support for IPv6, new match fields:
  – source address, destination address, protocol number, traffic class, ICMPv6 type, ICMPv6 code, IPv6 neighbor discovery header fields, and IPv6 flow labels
• Experimenter extensions
• Full VLAN and MPLS support
• **Multiple controllers**
OF 1.3

• Initial traffic shaping and QoS support
  – **Meters**: tables (accessed as usual with “goto table”) for collecting statistics on traffic flows and applying rate-limiters

<table>
<thead>
<tr>
<th>Meter Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meter identifier</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
• More extensible wire protocol
• **Synchronized tables**
  – tables with synchronized flow entries
• **Bundles**
  – similar to transactional updates in DB
• Support for optical ports
OF 1.5

Egress tables
OF 1.5

- Packet type aware pipeline
- Extensible flow entry statistics
- TCP flags matching
OF future extensions


• The discussion on flow states
  – The capability to store / access flow metadata that persists for lifetime of flow (not just current packet)
  – Potential to enable a variety of new capabilities:
    • Fragment handling without reassembly
    • Relation between bidirectional flows (e.g., RDI)
    • Autonomous flow learning + flow state tracking
    • MAC learning
    • TCP proxy
  – Hierarchies of flows
    • e.g. FTP control / data, all belonging to a user, etc.
Also abstraction “involutions” (?): Typed tables

- “A step back to ensure wider applicability”
- A third way between reactive and proactive
- Pre-run-time description of switch-level “behavioral abstraction” (tell to the switch which types of flowmods will be instantiated at run time)
- Limit types supported according to HW type

Typed tables patterns: Forwarding Elements (F:E.)

- OpenFlow 1.0
- Layer 3 IPv4
- Statefull Generic Tunnel
- Stateless Generic Tunnel
- 802.1D Forwarding
- Constrained OpenFlow 1.1

ONF Forwarding Abstractions WG
Breaking the limitation of specialized pipelines: Reconfigurable Match Table

• OF introduced the MMT model but does not mandate the width, depth, or even the number of tables
• OF allows the introduction of new match fields through a user-defined field facility

• Recent proposal by McKeown, Varghese et al. [BOS13]: RMT
• Goal: reconfigurability of multiple tables with arbitrary width

Reconfigurable Match Tables (RMT)

- Field definitions can be altered and new fields added
- Number, topology, widths, and depths of match tables can be specified, subject only to an overall resource limit on the number of matched bits
- New actions may be defined
- Arbitrarily modified packets can be placed in specified queues, for output at any subset of ports, with a queuing discipline specified for each queue.

Reconfigurable Match Tables (RMT)

Programmable parser

Header re-write with very long instruction word (VLIW)

Configurable queues

Pipeline flexibility: FlexPipe™ (Intel)

- Flexible pipeline and table solutions already available from some vendors
All the way down to programmability: P4

• OpenFlow 2.0 proposal by McKeown, Rexford et al. [BOS14]

• 3 goals:

  – **Protocol independence**
    • Configurable packet parser
    • Match/actions not tied to any specific network protocols

  – **Target independence**
    • Compiler handles mapping independently to switch details
      (e.g. TCAM vs RAM, softswitch, RMT, FlexPipe, etc.)

  – **Reconfigurability**
    • Change parsing and processing in the field

All the way down to programmability: P4

- **Configuration** achieved with a **programming language** describing parsing and control
- Flow tables **populated** with “Classic” OpenFlow
- **Compiler** maps parser and tables to specialized hardware or software
All the way down to programmability: P4

P4 configuration abstractions:

• **Header**
  – Ordered list of fields with name and width

• **Parser**
  – State machine traversing the packet to extract header fields
    (If ethertype == 0x800 then...)

• **Action functions**
  – Custom functions made of primitive actions
    (add_field, remove_header, copy, set, increment, checksum, etc...)

• **Typed tables**
  – What fields are matched in what way
    (exact match, longest prefix, range, etc...)

• **Control flow**
  – Table application order, conditionals, functions

• **Stateful memories**
  – Counters, meters and registers which persist across packets
Example:

**Header formats:**

```java
header ethernet {
    fields {
        dst_addr : 48; // width in bits
        src_addr : 48;
        ethertype : 16;
    }
}

header vlan {
    fields {
        pcp : 3;
        cfi : 1;
        vid : 12;
        ethertype : 16;
    }
}
```

**Typed tables and functions:**

```java
table mTag_table {
    reads {
        ethernet.dst_addr : exact;
        vlan.vid : exact;
    }
    actions {
        // At runtime, entries are programmed with params
        // for the mTag action. See below.
        add_mTag;
    }
}
```

**Parser state function:**

```java
parser start {
    ethernet;
}

parser ethernet {
    switch(ethertype) {
        case 0x8100: vlan;
        case 0x9100: vlan;
        case 0x800: ipv4;
        // Other cases
    }
}
```

**Action function:**

```java
action add_mTag(up1, up2, down1, down2, egr_spec) {
    add_header(mTag);
    // Copy VLAN ethertype to mTag
    copy_field(d(mTag.ethertype, vlan.ethertype);
    // Set VLAN ethertype to signal mTag
    set_field(d(vlan.ethertype, 0xaaaa);
    set_field(d(mTag.up1, up1);
    set_field(d(mTag.up2, up2);
    set_field(d(mTag.down1, down1);
    set_field(d(mTag.down2, down2);
    // Set the destination egress port as well
    set_field(d(metad ata.egress_spec, egr_spec);
}
```
Protocol Oblivious Forwarding (POF)

• Removing/neglecting constraints on general HW can lead to extreme flexibility of a clean slate approach (not in the OF evolution track)
• POF proposal by Huawei [SON13]
• POF makes the forwarding plane totally protocol-oblivious
• The POF FE has no need to understand the packet format
• Arbitrary field definition:

```c
field {
    type;
    offset;
    length;
}
```

Example:

```
MAC
0  6  12

dst | src | type
---|-----|-----
dst: {0, 0, 48};
src: {0, 48, 48};
type: {0, 96, 16};
```

Protocol Oblivious Forwarding (POF)

- POF FE execute instruction of its controller to:
  - extract and assemble the search keys from the packet header,
  - conduct the table lookups,
  - execute the associated instructions (in the form of executable code written in FIS or compiled from FIS).

Protocol Oblivious Forwarding (POF)

Example: forwarding based on a “new” protocol IPvX with 64 bits addresses

Tiny programs in the packets

• Taking programmability to the extreme ...
• Remember “active networks” ...

Deeply Programmable Networks (FLARE)

- Fully programmable control plane
- Fully programmable data plane
- Flexible and extensible API for both planes
- Experimental implementation

Aki Nakao, FLARE Project, NakaoLab, The university of Tokyo.
Not too much not too little:
OpenState and stateful data planes

Too clever is dumb.
[Ogden Nash]
Looking for the “right” abstraction

• Programmability and real world viability
  – High levels of (deep) programmability in the data and control planes since ages
    • Active Networks
    • IETF ForCES

• Keywords for success:
  – Pragmatism
  – Compromise
  – “Right” mix of programmability: right level of abstraction
    • Many wonderful programmable platforms buried in the “lab world”
Remember: OF meant to be a compromise
[original quotes: from OF 2008 paper]

- **Best approach**: “persuade commercial name-brand equipment vendors to provide an open, programmable, virtualized platform on their switches and routers”
  - Plainly speaking: *open the box!! No way...*

- **Viable approach**: “compromise on generality and seek a degree of switch flexibility that is
  - High performance and low cost
  - Capable of supporting a broad range of research
  - **Consistent with vendors’ need for closed platforms.**
OF **forces** separation of data and control

- **SMART!**
  - Logically-centralized control
- **DUMB!**
  - Events from switches
    - Topology changes,
    - Traffic statistics,
    - Arriving packets
  - Commands to switches
    - (Un)install rules,
    - Query statistics,
    - Send packets
Centralized Control: pros and cons

• PROS:
  – Central view of the network as a whole
    • Network states, etc
  – One-click network config/deploy
    • Platform agnosting switch API is key - e.g. OpenFlow forwarding abstraction

• CONS:
  – Control latency!!!
    • $O(\text{second})$
      1s = 300M packets lost @ 100 gbps
  – Signalling overhead

Great idea for network-wide states and «big picture» decisions

Poor idea for local states/decision, (way!) better handled locally (less delay, less load)
Distributed controllers
the «common» way to address such cons

Proprietary controller extensions?
Back to Babel?

A non-solution!
still slow path latency!!
«true» fast path solution: update forwarding rules in 1 packet time – 3 ns @ 40B x 100 Gbps

3 ns = 60cm signal propagation...
Our vision

Events from switches & central rule updates
Restricted to those of interest for GLOBAL decisions

Decision on how network should operate remains at the controller (SDN vision)
But “execution” of forwarding plane updates can be locally delegated
Inject “switch control programs”
Change forwarding behavior as specified by “this program” IF (local) events occur

Local processing: Ultra low Latency:
o(nanosec) versus o(sec)

Local states: lower signalling
What is missing in the picture

Behavioral Description
src=1.2.*, dest=3.4.5.* → drop
src = *, dest=3.4.*.* → forward(2)
src=10.1.2.3, dest=*.*.*.* → send to controller

OF forwarding abstraction insufficient!!
Platform-agnostic stateful processing: how to?

Any vendor, any size, any HW/SW platform...
Stateless vs. Stateful data plane

Stateless model (e.g. OpenFlow)

- Controller: Global + local states
  - Event notifications
  - Control enforcing

Switch: Stateless

- SMART!

Stateful model

- Controller: Global states
  - Control delegation
  - Auto-adaption

Switch: Local states

- SMART!
Easier said than done

• We need a switch architecture and API which is...
  
  – **High performance**: control tasks executed at wire-speed (packet-based events)
  
  – **Platform-independent**: consistent with vendors’ needs for closed platforms
  
  – **Low cost and immediately viable**: based on commodity HW

Apparently, far beyond OpenFlow switches...

Our (perhaps surprising?) finding: much closer to OF than expected!!
Our findings at a glance

- Any control program that can be described by a Mealy (Finite State) Machine is already (!) compliant with OF1.3

- MM + Bidirectional flow state handling requires minimal hardware extensions to OF1.1+

- Proof of concept HW and SW implementation
Our approach: **OpenState**

easier understood via a running example: port knocking


Remember OF match/action API

### Programmable logic
- Matching Rule
- Action

<table>
<thead>
<tr>
<th>Switch Port</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>VLAN ID</th>
<th>IP Src</th>
<th>IP Dst</th>
<th>IP Prot</th>
<th>TCP sport</th>
<th>TCP dport</th>
</tr>
</thead>
</table>

1. FORWARD TO PORT
2. ENCAPSULATE & FORWARD
3. DROP
4. ...

### Pre-implemented matching engine

### Vendor-implemented

Multiple flow tables since OF version 1.1

---

A. Capone & C. Cascone: SDN tutorial
Background: Port Knocking firewall
knock «code»: 5123, 6234, 7345, 8456

| IPsrc=1.2.3.4 | Port=5123 | Drop(); 1.2.3.4 → 1° knock OK |
| IPsrc=1.2.3.4 | Port=6234 | Drop(); 1.2.3.4 → 2° knock OK |
| IPsrc=1.2.3.4 | Port=7345 | Drop(); 1.2.3.4 → 3° knock OK |
| IPsrc=1.2.3.4 | Port=8456 | Drop(); 1.2.3.4 → OPEN port SSH |
| IPsrc=1.2.3.4 | Port=22  | Forward()                         |
Port Knocking @ controller

Encapsulate & forward ALL packets of ALL flows

When knock sequence finalized, add entry <Ipsrc, port=22; forward>

Lots of overhead!!
Needed as no «knock» state handled in switch
«Abstract» description for port knocking: Mealy Machine

- **Stage 1**: Port\(!=6234\) Drop()
- **Stage 2**: Port\(!=7345\) Drop()
- **Stage 3**: Port\(!=8456\) Drop()
- **Default**: Drop()
Can transform in a flow table? Yes:

MATCH: <state, port> → ACTION: <drop/forward, state_transition>
Plus a state lookup/update

Match fields

<table>
<thead>
<tr>
<th>state</th>
<th>event</th>
<th>action</th>
<th>Next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>Port=5123</td>
<td>drop</td>
<td>STAGE-1</td>
</tr>
<tr>
<td>STAGE-1</td>
<td>Port=6234</td>
<td>drop</td>
<td>STAGE-2</td>
</tr>
<tr>
<td>STAGE-2</td>
<td>Port=7345</td>
<td>drop</td>
<td>STAGE-3</td>
</tr>
<tr>
<td>STAGE-3</td>
<td>Port=8456</td>
<td>drop</td>
<td>OPEN</td>
</tr>
<tr>
<td>OPEN</td>
<td>Port=22</td>
<td>forward</td>
<td>OPEN</td>
</tr>
<tr>
<td>OPEN</td>
<td>Port=*</td>
<td>drop</td>
<td>OPEN</td>
</tr>
<tr>
<td>*</td>
<td>Port=*</td>
<td>drop</td>
<td>DEFAULT</td>
</tr>
</tbody>
</table>

Ipsrc: ??

State DB
Putting all together

1) State lookup

<table>
<thead>
<tr>
<th>Flow key</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPsrc= ... ...</td>
<td>... ... ...</td>
</tr>
<tr>
<td>IPsrc= 1.2.3.4</td>
<td>Write: OPEN</td>
</tr>
<tr>
<td>IPsrc= 5.6.7.8</td>
<td>OPEN</td>
</tr>
<tr>
<td>IPsrc= ... ...</td>
<td>... ... ...</td>
</tr>
<tr>
<td>IPsrc= no match</td>
<td>DEFAULT</td>
</tr>
</tbody>
</table>

2) XFSM state transition

<table>
<thead>
<tr>
<th>Match fields</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>event</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>Port=5123</td>
</tr>
<tr>
<td>STAGE-1</td>
<td>Port=6234</td>
</tr>
<tr>
<td>STAGE-2</td>
<td>Port=7345</td>
</tr>
<tr>
<td>STAGE-3</td>
<td>Port=8456</td>
</tr>
<tr>
<td>OPEN</td>
<td>Port=22</td>
</tr>
<tr>
<td>OPEN</td>
<td>Port=*</td>
</tr>
<tr>
<td>*</td>
<td>Port=*</td>
</tr>
</tbody>
</table>

3) State update

1 «program» XFSM table for all flows
(same knocking sequence)
N states, one per (active) flow

A. Capone & C. Cascone: SDN tutorial

84
Proof of concept

• SW implementation:
  – Trivial modifications to SoftSwitch
  – Public domain

• HW implementation:
  – 5 clock (2 SRAM read + 2 TCAM + 1 SRAM write)
  – 10 Gbps just requires 156 MHz clock TCAM, trivial
  – Optimization in progress (pipelining) for 100 Gbps.
Cross-flow state handling

- Yes but what about MAC learning, multi-port protocols (e.g., FTP), bidirectional flow handling, etc?

<table>
<thead>
<tr>
<th>MACdst</th>
<th>MACsrc</th>
<th>lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>State Table</td>
</tr>
<tr>
<td>Flow key</td>
<td>state</td>
<td></td>
</tr>
<tr>
<td>48 bit MAC addr</td>
<td>Port #</td>
<td></td>
</tr>
</tbody>
</table>

XFSM Table

<table>
<thead>
<tr>
<th>state</th>
<th>event</th>
<th>action</th>
<th>Next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port#</td>
<td>*</td>
<td>forward</td>
<td>In-port</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MACdst</th>
<th>MACsrc</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>State Table</td>
</tr>
<tr>
<td>Flow key</td>
<td>state</td>
<td></td>
</tr>
<tr>
<td>48 bit MAC addr</td>
<td>Port #</td>
<td></td>
</tr>
</tbody>
</table>

DIFFERENT lookup/update scope

<table>
<thead>
<tr>
<th>Field 1</th>
<th>Field 2</th>
<th>Field N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flowkey selector</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read/write signal
Current challenge
Prove programmability of complex functions

DONE:
• Port Knocking
• MAC learning
• Label/address advertisement learning
• Reverse Path Forwarding
• Flow-consistent Load Balancing
• DDoS multi-stage flow marking
• ...

All (!) otherwise not possible without explicit controller’s involvement or custom distrib. control...

CHALLENGE:
⇒ IDS/DPI
⇒ TCP flow processing
⇒ Monitoring
⇒ ...

Need new «actions»
Need extra logic (full XFSM)

Our challenge: towards an open «flow processor»?
Aftermath

• Control intelligence in devices seems possible
  – Via Platform-independent abstraction
  – Retaining high speed & scalability
  – As «small» OpenFlow extension (?!)

• TCAM as «State Machine processor»
  – Now Mealy Machines
  – Currently working on full XFSM extension

• Rethinking control-data plane SDN separation?
  – Control = Decide! Not decide+enforce!
Applied smartness: stateful applications

There are science and the applications of science, bound together as the fruit of the tree which bears it. [Louis Pasteur]
Forwarding Consistency

• Ensure consistency in forwarding decisions for packets of a same transport layer flow

Example: LAG @Internet Exchange Point

Example: Server Load Balancer
Fault Tolerance

• OpenFlow

**Fast failover:** local reroute based on port states (Group table)

But what if a local reroute is not available???
Switch: ofsoftswitch13; Controller: Ryu
Hands-on session

Carmelo Cascone
Forwarding Consistency

- Ensure consistency in forwarding decisions for packets of a same transport layer flow

Example: LAG @Internet Exchange Point

Example: Server Load Balancer
Forwarding Consistency

**One to Many:**
Intra-flow state handling

**Many to One:**
Cross-flow state handling

**Many to Many:**
Inter-stage cross-flow state handling
Forwarding Consistency: **One to Many**

The first packet of a TCP connection coming from the input port is sent to one of many possible output ports.

All the next packets of the same TCP connection must be forwarded on the same selected port.
Forwarding Consistency: One to Many

OpenFlow solution: the controller is in charge of states management.

First packet of each new TCP connection is sent to the controller in order to:
- select an output port (e.g. randomly) and forward the packet
- install a flow entry for subsequent packets
Forwarding Consistency: **One to Many**

**OpenState** solution:
the switch itself handles connection’s state.

A flow is identified by \([\text{IP}_{\text{SRC}},\text{IP}_{\text{DST}},\text{TCP}_{\text{SRC}},\text{TCP}_{\text{DST}}]\)
FLOW STATE: output port

First packet of each new TCP connection is sent to the **Group Table** in order to:
• select an output port randomly
• store the selected port in the State Table for subsequent packets

**Controller is not involved!**
Forwarding Consistency: **One to Many**

- In this case the state is set in the group table
- Random Group Entry

**Lookup Scope:** IP\(_{\text{src}}\), IP\(_{\text{dst}}\), TCP\(_{\text{src}}\),TCP\(_{\text{dst}}\)

**Update Scope:** IP\(_{\text{src}}\), IP\(_{\text{dst}}\), TCP\(_{\text{src}}\),TCP\(_{\text{dst}}\)

1) **State lookup**

<table>
<thead>
<tr>
<th>Flow key</th>
<th>IP(_{\text{src}})=10.0.0.1</th>
<th>IP(_{\text{dst}})=10.0.0.2</th>
<th>TCP(_{\text{src}})=2500</th>
<th>TCP(_{\text{dst}})=80</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>DEFAULT</td>
</tr>
</tbody>
</table>

2) **XFSM state transition**

<table>
<thead>
<tr>
<th>Match fields</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>event</td>
</tr>
<tr>
<td>DEF.</td>
<td>In-port=1</td>
</tr>
<tr>
<td>1</td>
<td>In-port=1</td>
</tr>
<tr>
<td>2</td>
<td>In-port=1</td>
</tr>
<tr>
<td>3</td>
<td>In-port=1</td>
</tr>
<tr>
<td>*</td>
<td>In-port=2</td>
</tr>
<tr>
<td>*</td>
<td>In-port=3</td>
</tr>
<tr>
<td>*</td>
<td>In-port=4</td>
</tr>
</tbody>
</table>

3) **Group Table**

<table>
<thead>
<tr>
<th>Group Entry</th>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>action</td>
</tr>
<tr>
<td>Entry 1</td>
<td>Forward(2)</td>
</tr>
<tr>
<td></td>
<td>Forward(3)</td>
</tr>
<tr>
<td></td>
<td>Forward(4)</td>
</tr>
</tbody>
</table>

4) **State update**

- write

A. Capone & C. Cascone: SDN tutorial
Forwarding Consistency: Many to One

Forwarding consistency must be ensured according to packets received in the reverse direction.

The first packet of a TCP connection coming from one of the many input ports is forwarded on the only output port. All packets of the reverse flow of the same TCP connection must be forwarded on the same ingress port.
Forwarding Consistency: Many to One

**OpenFlow** solution:
the controller is in charge of states management.

First packet of each new TCP connection is sent to the controller in order to:
- forward the packet
- install a flow entry for reverse flow’s packets
Forwarding Consistency: Many to One

**OpenState** solution:

the switch itself handles connection’s state.

A flow is identified by [IP_SRC,IP_DST,TCP_SRC,TCP_DST]

FLOW STATE: input port

First packet of each new TCP connection is forwarded and the input port is stored to forward response packets

Cross-flow state
Controller is not involved!
Forwarding Consistency: Many to One

Communication Host -> Server

Lookup Scope: IP_src, IP_dst, TCP_src, TCP_dst
Update Scope: IP_dst, IP_src, TCP_dst, TCP_src

1) State lookup

State Table

<table>
<thead>
<tr>
<th>Flow key</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_src=10.0.0.1</td>
<td>IP_dst=10.0.0.100</td>
</tr>
<tr>
<td>IP_src=10.0.0.2</td>
<td>IP_dst=10.0.0.1</td>
</tr>
<tr>
<td>IP_src=10.0.0.100</td>
<td>IP_dst=10.0.0.1</td>
</tr>
</tbody>
</table>

2) XFSM state transition

XFSM Table

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>Action</th>
<th>Next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>IP_src=10.0.0.1</td>
<td>IP_dst=10.0.0.100</td>
<td>TCP_src=2500</td>
</tr>
<tr>
<td>1</td>
<td>In-port=4</td>
<td>Forward(1)</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>In-port=4</td>
<td>Forward(2)</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>In-port=4</td>
<td>Forward(3)</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>In-port=1</td>
<td>Forward(4)</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>In-port=2</td>
<td>Forward(4)</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>In-port=3</td>
<td>Forward(4)</td>
<td>3</td>
</tr>
</tbody>
</table>

3) State update

DIFFERENT lookup/update scope
Forwarding Consistency: Many to One

Communication Server -> Host

Lookup Scope: IP_src, IP_dst, TCP_src, TCP_dst
Update Scope: IP_dst, IP_src, TCP_dst, TCP_src

1) State lookup

```
<table>
<thead>
<tr>
<th>IP_src</th>
<th>IP_dst</th>
<th>TCP_src</th>
<th>TCP_dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.100</td>
<td>10.0.0.1</td>
<td>2500</td>
<td>80</td>
</tr>
</tbody>
</table>
```

State Table

```
<table>
<thead>
<tr>
<th>Flow key</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_src=10.0.0.200</td>
<td>IP_dst=10.0.0.2</td>
</tr>
<tr>
<td>IP_src=10.0.0.100</td>
<td>IP_dst=10.0.0.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
```

2) XFSM state transition

```
<table>
<thead>
<tr>
<th>State Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
```

XFSM Table

```
<table>
<thead>
<tr>
<th>Match fields</th>
<th>Actions</th>
<th>Next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>event</td>
<td>action</td>
</tr>
<tr>
<td>1</td>
<td>In-port=4</td>
<td>Forward(1)</td>
</tr>
<tr>
<td>2</td>
<td>In-port=4</td>
<td>Forward(2)</td>
</tr>
<tr>
<td>3</td>
<td>In-port=4</td>
<td>Forward(3)</td>
</tr>
<tr>
<td>*</td>
<td>In-port=1</td>
<td>Forward(4)</td>
</tr>
<tr>
<td>*</td>
<td>In-port=2</td>
<td>Forward(4)</td>
</tr>
<tr>
<td>*</td>
<td>In-port=3</td>
<td>Forward(4)</td>
</tr>
</tbody>
</table>
```

A. Capone & C. Cascone: SDN tutorial
Forwarding Consistency: Many to Many

Combining the first two, we want here to load balance on the output ports while doing reverse path forwarding on the input port.

The first packet of a TCP connection coming from one of the many input ports is forwarded to one of many possible output ports.

All the next packets of the same TCP connection must be forwarded on the same selected output port, while all packets of the reverse flow of the same TCP connection must be forwarded on the same ingress port.
Forwarding Consistency: Many to Many

**OpenState** solution

A flow is identified by [IP_SRC,IP_DST,TCP_SRC,TCP_DST]

**Two states** are needed for each bidirectional flow:
FLOW STATE 1: output port
FLOW STATE 2: input port

For each first packet of each new TCP connection:
• packet is forwarded to a random output port
• the selected output port is stored in the State Table 0
• the input port is stored in the State Table 1
Forwading Consistency: Many to Many

Communication Host -> Server

Lookup Scope: IP_src, IP_dst, TCP_src,TCP_dst
Update Scope: IP_src, IP_dst, TCP_src,TCP_dst

1) State lookup

Flow key

State Table (Stage 0)

<table>
<thead>
<tr>
<th>Flow key</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_src=10.0.0.1 IP_dst=10.0.0.100 TCP_src=2500 TCP_dst=80</td>
<td>6</td>
</tr>
<tr>
<td>IP_src=10.0.0.1 IP_dst=10.0.0.100 TCP_src=2500 TCP_dst=80</td>
<td>5</td>
</tr>
<tr>
<td>IP_src=10.0.0.2 IP_dst=10.0.0.100 TCP_src=3000 TCP_dst=80</td>
<td>4</td>
</tr>
<tr>
<td>* * * *</td>
<td>DEFAULT</td>
</tr>
</tbody>
</table>

2) XFSM state transition

XFSM Table (Stage 0)

<table>
<thead>
<tr>
<th>Match fields</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>event</td>
</tr>
<tr>
<td>DEF.</td>
<td>In-port=1</td>
</tr>
<tr>
<td>4</td>
<td>In-port=1</td>
</tr>
<tr>
<td>5</td>
<td>In-port=1</td>
</tr>
<tr>
<td>6</td>
<td>In-port=1</td>
</tr>
<tr>
<td>DEF.</td>
<td>In-port=2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>*</td>
<td>In-port=4</td>
</tr>
<tr>
<td>*</td>
<td>In-port=5</td>
</tr>
<tr>
<td>*</td>
<td>In-port=6</td>
</tr>
</tbody>
</table>

3) State update (Stage 0)

Group Table

<table>
<thead>
<tr>
<th>Group Entry</th>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>action</td>
</tr>
<tr>
<td>Entry 1</td>
<td>Forward(4)</td>
</tr>
<tr>
<td></td>
<td>Forward(5)</td>
</tr>
<tr>
<td></td>
<td>Forward(6)</td>
</tr>
</tbody>
</table>
### Forwarding Consistency: Many to Many

**Communication Host -> Server**

- **Lookup Scope:** IP\_src, IP\_dst, TCP\_src, TCP\_dst
- **Update Scope:** IP\_dst, IP\_src, TCP\_dst, TCP\_src

1) State lookup

2) XFSM state transition

3) State update (Stage 1)

**State Table (Stage 0)**

<table>
<thead>
<tr>
<th>Flow key</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_src=10.0.0.1</td>
<td>6</td>
</tr>
<tr>
<td>IP_dst=10.0.0.1</td>
<td>5</td>
</tr>
<tr>
<td>IP_dst=10.0.0.100</td>
<td>4</td>
</tr>
</tbody>
</table>

**State Table (Stage 1)**

<table>
<thead>
<tr>
<th>Flow key</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_src=10.0.0.1</td>
<td>3</td>
</tr>
<tr>
<td>IP_src=10.0.0.1</td>
<td>2</td>
</tr>
<tr>
<td>IP_src=10.0.0.1</td>
<td>1</td>
</tr>
</tbody>
</table>

**DIFFERENT lookup/update scope**

**Match fields**

<table>
<thead>
<tr>
<th>state</th>
<th>event</th>
<th>action</th>
<th>Next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF.</td>
<td>In-port=1</td>
<td>Group(1)</td>
<td>(1, stg_1)</td>
</tr>
<tr>
<td>4</td>
<td>In-port=1</td>
<td>Forward(4)</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>In-port=1</td>
<td>Forward(5)</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>In-port=1</td>
<td>Forward(6)</td>
<td>-</td>
</tr>
<tr>
<td>DEF.</td>
<td>In-port=2</td>
<td>Group(1)</td>
<td>(2, stg_1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>*</td>
<td>In-port=4</td>
<td>Table(1)</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>In-port=5</td>
<td>Table(1)</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>In-port=6</td>
<td>Table(1)</td>
<td>-</td>
</tr>
</tbody>
</table>
### Forwarding Consistency: Many to Many

**Communication Server -> Host**

#### Lookup Scope: IP_src, IP_dst, TCP_src, TCP_dst

#### Update Scope: IP_dst, IP_src, TCP_dst, TCP_src

**1) State lookup Stage 0**

<table>
<thead>
<tr>
<th>IP_dst</th>
<th>IP_src</th>
<th>TCP_dst</th>
<th>TCP_src</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.100</td>
<td>10.0.0.1</td>
<td>TCP_dst</td>
<td>TCP_src</td>
</tr>
</tbody>
</table>

**2) XFSM state transition Stage 0**

- **DEFAULT**
- **IP dst=10.0.0.100**
- **IP src=10.0.0.1**
- **TCP_src=2500**
- **TCP_dst=80**

**3) State lookup Stage 1**

**4) XFSM state transition Stage 1**

<table>
<thead>
<tr>
<th>IP_dst</th>
<th>IP_src</th>
<th>TCP_dst</th>
<th>TCP_src</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.100</td>
<td>10.0.0.1</td>
<td>TCP_dst</td>
<td>TCP_src</td>
</tr>
</tbody>
</table>

**Table (Stage 0)**

<table>
<thead>
<tr>
<th>Flow key</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_src=10.0.0.100</td>
<td>IP_dst=10.0.0.100</td>
</tr>
<tr>
<td>IP_src=10.0.0.1</td>
<td>IP_dst=10.0.0.100</td>
</tr>
<tr>
<td>IP_src=10.0.0.1</td>
<td>IP_dst=10.0.0.100</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

**Table (Stage 1)**

<table>
<thead>
<tr>
<th>Flow key</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_src=10.0.0.100</td>
<td>IP_dst=10.0.0.100</td>
</tr>
</tbody>
</table>

**State Table (Stage 0)**

<table>
<thead>
<tr>
<th>Match fields</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>event</td>
</tr>
<tr>
<td>1</td>
<td>In-port=4</td>
</tr>
<tr>
<td>1</td>
<td>In-port=5</td>
</tr>
<tr>
<td>1</td>
<td>In-port=6</td>
</tr>
<tr>
<td>2</td>
<td>In-port=1</td>
</tr>
</tbody>
</table>

**State Table (Stage 1)**

<table>
<thead>
<tr>
<th>Flow key</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_src=10.0.0.100</td>
<td>IP_dst=10.0.0.3</td>
</tr>
<tr>
<td>IP_src=10.0.0.100</td>
<td>IP_dst=10.0.0.2</td>
</tr>
<tr>
<td>IP_src=10.0.0.100</td>
<td>IP_dst=10.0.0.1</td>
</tr>
</tbody>
</table>

**XFSM Table (Stage 0)**

<table>
<thead>
<tr>
<th>Match fields</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>event</td>
</tr>
<tr>
<td>DEF.</td>
<td>In-port=1</td>
</tr>
<tr>
<td>4</td>
<td>In-port=1</td>
</tr>
<tr>
<td>5</td>
<td>In-port=1</td>
</tr>
<tr>
<td>6</td>
<td>In-port=1</td>
</tr>
<tr>
<td>DEF.</td>
<td>In-port=2</td>
</tr>
<tr>
<td>*</td>
<td>In-port=4</td>
</tr>
<tr>
<td>*</td>
<td>In-port=5</td>
</tr>
<tr>
<td>*</td>
<td>In-port=6</td>
</tr>
</tbody>
</table>

**XFSM Table (Stage 1)**

- **DEFAULT**
- **IP dst=10.0.0.100**
- **IP src=10.0.0.1**
- **TCP_src=2500**
- **TCP_dst=80**

A. Capone & C. Cascone: SDN tutorial
Forwarding Consistency: Example Results

Results will show the average value of the time required by 1000 TCP SYN packets to cross the switches at increasing rate.

Switch: ofsoftswitch13; Controller: Ryu
Fault Tolerance

- Ensure the network failure resiliency, quickly readapting the routing after a failure
- Fundamental function in any network (telco operators, data centers)
- Weak support in OpenFlow
Fault Tolerance

• OpenFlow

Fast failover: local reroute based on port states (Group table)

But what if a local reroute in not available ???
Fault Tolerance

- OpenFlow

Obviously it is always possible to rely on the controller to:
- forward the packet on the backup path
- install flow entries for the backup path
Fault Tolerance in OpenState

With **OpenState** the switch itself can react to the fault

Proposed solution:

- Faults are signaled using the same data packets
- Packets are tagged and sent back
- Packets are sent back until matched against a node able to respond to that fault
Fault Tolerance

OpenState

• A DETOUR is enabled based on the specific failure without constraints
• Backup paths can be pre-computed and installed by the controller (traffic engineering and quality/congestion control)
• The controller is entitled to restore the primary path once the fault has been resolved

Fault Tolerance: Fault Reaction Example

OpenState

Redirect Node:

FLOW STATE = DEF 20
TAG = STATE = 20

Detect Node:

GLOBAL REGISTERS = 00
TAG = FAULT_ID = 20
Fault Tolerance: Example on larger network

- **Primary node**: Node 19
- **Detect node**: Node 13
- **Forward-back node**: Node 17
- **Redirect node**: Node 16
- **Detour node**: Node 15
Fault Tolerance: Example Results

OpenState vs. OpenFlow

Ideal case:
0ms failure detection delay

Realistic case:
Switch embedded failure detection mechanism

<table>
<thead>
<tr>
<th>Ping rate (req/s)</th>
<th>OpenFlow (Ideal)</th>
<th>OpenFlow (Realistic)</th>
<th>OpenState (Ideal)</th>
<th>OpenState (Realistic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thanks

Antonio Capone (antonio.capone@polimi.it)
Carmelo Cascone (carmelo.cascone@polimi.it)

OpenState: openstate-sdn.org

EU project BEBA – http://www.beba-project.eu

This slide-set is available on OpenState web site!