L6: Network Hardware and Software

ECE 544: Computer Networks II
Spring 2018
Seskar at winlab dot rutgers dot edu
Network Hardware Basics
Standards

- Availability of interoperable equipment from multiple vendors
- Prevents a “Tower of Babel” situation
  - Equipment from different vendors will interoperate if it complies with the standard
  - Alliances and certification bodies assure interoperability
    - Wi-Fi for 802.11 WiMax for 802.16
- Lowers costs to consumers
  - Both through competition and economies of scale
- Promotes advancement of technology
  - Vendors constantly strive for competitive advantage through improved technology and features
IEEE 802 Standards

Maintained by IEEE 802 LAN/MAN Standards Committee (LMSC):

- 802.1 Overview, Architecture, Internetworking and Management
- 802.2 Logical Link Control
- 802.3 Ethernet (CSMA/CD PHY and MAC)
- 802.5 Token Ring PHY and MAC
- **802.11 Wireless LAN**
  - 802.12 Demand Priority Access
- **802.15 Wireless PAN**
- **802.16 Broadband Wireless Access**
  - 802.17 Resilient Packet Ring
- **802.18 Radio Regulatory**
- 802.19 Coexistence
- **802.20 Mobile Broadband Wireless Access**
- 802.21 Media Independent Handoff
- **802.22 Wireless Regional Area Network**
IEEE 802 Naming Conventions

- Standalone documents either get no letter (IEEE 802.3) or gets a capital letter (IEEE 802.1D)

- Document that supplements a standalone document gets a lower-case letter (IEEE 802.11b)

- Letters are assigned in sequential order (a,B,C,d,e ...) and uniquely identify both the Working Group Task Force and the actual document

- Approved standards have IEEE in front while drafts have P only designation followed by the draft number (P802.1p/D9)
802.1

- 802.1B Management
- 802.1D MAC Bridges
- 802.1E System Load Protocol
- 802.1F Common Definitions for Management
- 802.1G Remote MAC Bridging
- 802.1H Bridging of Ethernet
- 802.1Q Virtual Bridged LANs
## Terminology

**Fast Ethernet Symbol Stream**

**Ethernet Frame**

**IP Packet**

**TCP Segment**

<table>
<thead>
<tr>
<th><strong>PHY Layer Header</strong></th>
<th><strong>Ethernet Header</strong></th>
<th><strong>IP Header</strong></th>
<th><strong>TCP Header</strong></th>
<th><strong>Data</strong></th>
<th><strong>Ethernet Trailer</strong></th>
<th><strong>PHY Layer Trailer</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>PDU</strong></th>
<th><strong>Interconn. Device</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Application</em></td>
<td><strong>Gateway</strong></td>
</tr>
<tr>
<td><em>Presentation</em></td>
<td></td>
</tr>
<tr>
<td><em>Session</em></td>
<td></td>
</tr>
<tr>
<td><em>Transport</em></td>
<td>Segment or Message</td>
</tr>
<tr>
<td><em>Network</em></td>
<td>Packet</td>
</tr>
<tr>
<td><em>Data Link</em></td>
<td>Frame</td>
</tr>
<tr>
<td><em>Physical</em></td>
<td>Symbol Stream</td>
</tr>
</tbody>
</table>

Source: Seifert “The switch Book”
Router Hardware History

- Mid 1980s (early days):
  - Shared LANs interconnected by bridges
  - Two port software based routers

- Late 1980s – early 1990s (rapid expansion for router market)
  - Slower than bridges but have much more functions
  - “Route when you can, bridge when you must”

- Early – mid 1990s (routers as necessary evils)
  - Hardware based bridges (switches) with wire-speed performance
  - “Switch when you can, route when you must”

- Late 1990s
  - Hardware based routers become practical
  - Wire-speed routing
  - Perception that all traffic can be switched
Devices

- Repeaters/Hubs
- Bridges (Layer 2 Switches?)
- Routers (Layer 3 Switches?)
  - Core
  - Edge
- Firewalls, Network Address Translators (Layer 4 Switches?)
- Gateways, Load Balancers (Layer 7 Switches?)
Switch/Router Hardware

- Port 1 (Line Card)
- Port 2
- Port N (Line Card)
- Media Interface
- Link Protocol Controller
- Header Processing
- Switch Fabric
- Processor(s)
Requirements

■ Distributed Data Plane
  ■ Packet Processing: Examine L2-L7 protocol information (Determine QoS, VPN ID, policy, etc.)
  ■ Packet Forwarding: Make appropriate routing, switching, and queuing decisions
  ■ Performance: At least sum of external BW

■ Distributed Control Plane
  ■ Up to $10^6$ entries in various tables (forwarding addresses, routing information etc.)
  ■ Performance: on the order of 100 MIPS
Switch Fabric

- Connects inputs and outputs

- Fabric types:
  - **Shared Bus** – shared backplane with or without DMA (first and second generation routers) – arbitration problem
  - **Shared Memory** – single common memory is shared between multiple input outputs (typically used in low-cost devices) – memory bandwidth problem
  - **Shared Interconnect** (Crossbar) – switching fabric provides parallel paths. Scheduler is centralized; routing tables are kept in the line cards (third generation routers) – multicast problem
## Queuing

<table>
<thead>
<tr>
<th>Input queue</th>
<th>Output queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros:</strong></td>
<td><strong>Pros:</strong></td>
</tr>
<tr>
<td>- Simple algorithms</td>
<td>- Simple algorithms</td>
</tr>
<tr>
<td>- Single congestion point</td>
<td>- Single congestion point</td>
</tr>
<tr>
<td><strong>Cons:</strong></td>
<td><strong>Cons:</strong></td>
</tr>
<tr>
<td>- Must implement flow control</td>
<td>- N inputs may send to the same output; requires N times speedup</td>
</tr>
<tr>
<td>- Low utilization due to HoL Blocking</td>
<td></td>
</tr>
</tbody>
</table>
Internally non-blocking switch
Connect all inputs simultaneously as long as:

\[ O_i \neq O_j \text{ for each } P_i \text{ (in the same time slot)} \]

Implementation mechanisms: speedup, sorting, etc.
Output Blocking
Output Blocking

Requires output conflict resolution
Per time slot conflict resolution (FCFS, LCFS, etc.)
Implementation mechanisms: polling, input sorting & purging
with or without ack, etc.
Head of Line Blocking

Easing HoL Blocking:

a.) Allow packets behind to contend
b.) Allow multi-packet delivery
Modern routers

Combine input buffering with virtual output queues (separate input queue per output) and use output buffering

- Solves blocking problem
- Resolves contention and simplifies scheduling
- Can achieve utilization of 1
- Scales to > 1 Tbps

Crossbar switch for distributed forwarding

Increases complexity (as well as introduces multiple congestion points)
Switched LAN

- Modern switches - LAN Segmentation taken to the extreme (microsegmentation):
  - No access contention
  - No collisions (full-duplex)
  - Dedicated bandwidth for each station
  - No distance reduction per segment
- Best case capacity (non-blocking switch)

\[
\text{Capacity} = \sum_{\text{port}=1}^{n} \text{DataRate}_{\text{port}}
\]
## Cut-Trough vs. Store-and-Forward

<table>
<thead>
<tr>
<th>Store-and-forward</th>
<th>Cut-trough</th>
</tr>
</thead>
<tbody>
<tr>
<td>The frame (packet, message) is received completely before decision is made</td>
<td>Table lookup and forwarding decision is made as soon as “Destination” has been received</td>
</tr>
</tbody>
</table>

- **Absolute latency** (not that much affected by the lookup process)
- **Problem with output port availability**
- **Cut-trough is generally not possible for multicast or unknown destination** (all output ports have to be available simultaneously)
## Performance Requirements

<table>
<thead>
<tr>
<th>Rate</th>
<th>Overhead</th>
<th>Peak packet rate</th>
<th>Time per packet [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>small</td>
</tr>
<tr>
<td>10Base-T</td>
<td>10.00</td>
<td>38 [Bytes]</td>
<td>19.5</td>
</tr>
<tr>
<td>100Base-T</td>
<td>100.00</td>
<td>38 [Bytes]</td>
<td>195.3</td>
</tr>
<tr>
<td>OC-3</td>
<td>155.52</td>
<td>6.912 [Mbps]</td>
<td>303.8</td>
</tr>
<tr>
<td>OC-12</td>
<td>622.08</td>
<td>20.736 [Mbps]</td>
<td>1214.8</td>
</tr>
<tr>
<td>1000Base-T</td>
<td>1000.00</td>
<td>38 [Bytes]</td>
<td>1953.1</td>
</tr>
<tr>
<td>OC-48 (2.5G)</td>
<td>2488.32</td>
<td>82.944 [Mbps]</td>
<td>4860.0</td>
</tr>
<tr>
<td>OC-192 (10G)</td>
<td>9953.28</td>
<td>331.776 [Mbps]</td>
<td>19440.0</td>
</tr>
<tr>
<td>OC-768 (40G)</td>
<td>39,813.12</td>
<td>1327.104 [Mbps]</td>
<td>77760.0</td>
</tr>
</tbody>
</table>

(Optical Carrier (OC-1) is SONET line with payload of 50.112 Mbps and overhead of 1,728 Mbps)

- In general header inspection and packet forwarding require complex look-ups on a per packet basis resulting in up to 500 instructions per packet
- At 40Gbps processing requirements are > 100 MPPS
Software Based “Switching”

1. NIC receives the frame, stores it in a buffer, and signals the device driver to pick it up.
2. Device driver will transfer the packet to the IP stack.
3. IP stack will check the header and options for correctness and then check the address; local packets are passed up to higher layers (or socket abstraction); non-local packets are passed to IP forwarder.
4. IP forwarder will decrement the TTL, and consult the routing table as to where to send the packet (lookup). Tables are constructed by a separate routing daemon (OSPF, BGP, or RIP).
5. Finally packet will be sent to appropriate device driver for delivery. Device driver will perform fragmentation if needed before actually delivering the packet to NIC.

Things that define performance:
- Interrupt latency
- Bus bandwidth
- Speed of the CPU
- Memory bandwidth
Click Router

- Linux kernel/user space implementation based on peer-to-peer packet transfer between processing blocks.
- The router is assembled from packet processing modules called elements. Configuration consist of selecting elements and connecting them into a directed graph. (e.g. IP router has sixteen elements on its forwarding path)
- Supports both push and pull (queue element as a connection between opposite ports)
- Maximum loss-free forwarding rate for IP routing is 357,000 64-byte packets per second (4 year old number)
Fast Path

- Software based “switching” is amendable to changes but slow
- Hardware based “switching” has very high processing rate but is inflexible
- The switching architecture is optimized for functions needed for majority of packets – **FAST PATH.** For unicast IP:
  - Packet parsing and validation
  - Routing table lookup
  - ARP mapping
  - Update TTL and header checksum
- The rest of the features are implemented in software
  - Rarely used protocol options
  - Fragmentation (if ports are not the same)
  - Housekeeping
  - SNMP, ICMP, Exception conditions
Cisco 6513 Switch

- The 13-slot chassis based switch
- Variety of modules:
  - Supervisor engines
  - Ethernet modules (Fast Ethernet, Gigabit Ethernet, 10 Gigabit Ethernet)
  - Flex WAN modules
  - Shared Port Adaptors/SPA Interface Processors
  - Multi-Gigabit services modules (content services, firewall, intrusion detection, IP Security [IPSec], VPN, network analysis, and Secure Sockets Layer [SSL] acceleration)

<table>
<thead>
<tr>
<th>Ethernet</th>
<th>Max Density (ports)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast</td>
<td>1152</td>
</tr>
<tr>
<td>Gig</td>
<td>577</td>
</tr>
<tr>
<td>10 Gig</td>
<td>20</td>
</tr>
</tbody>
</table>

**Performance (SUP720)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth per slot [Gbps]</td>
<td>40</td>
</tr>
<tr>
<td>Total bandwidth [Gbps]</td>
<td>720</td>
</tr>
<tr>
<td>Routes supported</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Packets per second [Mpps]</td>
<td>407</td>
</tr>
<tr>
<td>MAC addresses supported</td>
<td>64K</td>
</tr>
</tbody>
</table>
Virtual LANs

Allows separation of *logical* and *physical* connectivity

- multiple logical connections are sharing single physical connection

**VLAN Concepts**

- Tagging
  - Implicit
  - Explicit
- Awareness
- Association rules
- Frame distribution
### Virtual LANs (contd.)

<table>
<thead>
<tr>
<th>Association Rule</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>- Software patch panel</td>
</tr>
<tr>
<td></td>
<td>- Switch segmentation</td>
</tr>
<tr>
<td></td>
<td>- Security and BW preservation</td>
</tr>
<tr>
<td>MAC address</td>
<td>Station mobility</td>
</tr>
<tr>
<td>IP subnet</td>
<td>IP mobility</td>
</tr>
<tr>
<td>Protocol</td>
<td>Protocol based access control</td>
</tr>
<tr>
<td>Application</td>
<td>- Groupware environment</td>
</tr>
<tr>
<td></td>
<td>- Fine-grained BW preservation</td>
</tr>
</tbody>
</table>
### 802.1Q Tagging

#### Priority – 802.1p guys needed “home”

#### Canonical Format Indicator (CFI) – indicates endianness of the frame
- 0 -> addresses are in canonical format (Little Endian) and no source routing information in the VLAN tag
- 1 -> addresses are in non-canonical format (Big Endian) and tag is extended with source routing information

#### VLAN Identifier – up to 4096 VLANs
- 0xFFF - RFU
- 0x000 – used to indicate priority without VLAN
802.1Q Switch Flow

Ingress

Port 1 (input)
Acceptable Frame Filter
Ingress Rules
Ingress Filter
Forward Decision
Ingress Rules
Ingress Filter
Forward Decision
Acceptable Frame Filter

Port 2 (input)

Port n (input)
Acceptable Frame Filter
Ingress Rules
Ingress Filter
Forward Decision
Ingress Rules
Ingress Filter
Forward Decision
Acceptable Frame Filter

Progress

Switching Fabric

Egress

Port 1 (outp.)
Egress Rules
Egress Filter

Port 2 (outp.)

Port n (outp.)
Egress Rules
Egress Filter

Filtering DB
Link Aggregation (802.3ad)

Unify multiple connections in a single link:
- Increased capacity
- Incremental capacity
- Higher availability

Requirements:
- Multiple interfaces with same MAC and data rate
- Mapping (distribution function to physical links)
- Configuration control (Link Aggregation Sublayer)

Switch 1
- Frame Distributor
  - Aggregator Transmit Queue
  - Interface 1
  - Interface 2
  - Interface 3
  - Interface 4

Switch 2
- Frame Collector
  - Aggregator Receiver Queue
  - Interface 1
  - Interface 2
  - Interface 3
  - Interface 4

Aggregated Links

Timeline
802.3ad

- Applies only to Ethernet
- Burden is on *Distributor* rather than *Collector*
- Link Aggregation Control Protocol (LACP)
  - Concept of *Actor* and *Partner*
  - Port modes:
    - Active mode – port emits messages on a periodic basis (fast rate = 1 sec, slow rate – 30 sec)
    - Passive mode – port will not speak unless spoken to
Network Management and Monitoring
Simple Network Management Protocol

- SNMP – introduced in 1988 as a protocol for management of IP devices
  - RFC 1157 - Simple Network Management Protocol (SNMP)
  - RFC 1213 - The Management Information Base II (MIB II)
  - RFC 1902 - The Structure for Management Information (SMI)
- A simple set of operations and information that enables operator to set/get state of a device
SNMP (contd.)

- Managers and agents
- UDP based protocol
- Ports 161 and 162
- Community
- SMI - Managed Object with attributes:
  - Name
  - Type and syntax
  - Encoding
Management Information Base

Logical grouping of managed objects as they pertain to a management task

Example MIBs
- MIB-II (RFC 1213)
- ATM (RFC 2515)
- DNS Server (RFC 1611)
- 802.11MIB
- CHRM-SYS.mib
  (private Cisco mib for general system variables)

Example:
MIB managed object: **1.3.6.1.2.1.1.6** (iso.org.dod.internet.management.mib-2.system.sysLocation)
SNMP Operations

- Simple request/response protocol.

- **SNMP Operations:**
  - **Retrieving Data:** `get`, `getnext`, `getbulk`, `getresponse`
  - **Altering Variables:** `set`
  - **Receiving Unsolicited Messages:** `trap`, `notification`, `inform`, `report`

- Each operation has a standard Protocol Data Unit (PDU) format

- Most implementations have command line operation equivalents

SNMPv2 and SNMPv3 only
### Management/Monitoring Tools

**Commercial SNMP based**
- **NMS**
  - HP’s OpenView Network Node Manager
  - Castle Rock’s SNMPc Enterprise Edition
- **Agent**
  - HP OpenView Agent
  - OS SNMP Agent
  - Cisco IOS
  - SystemEDGE

**Open Source SNMP based**
- **NMS**
  - Nagios
  - Multi Router Traffic Grapher (MRTG)
  - OpenNMS
  - Netdisco
- **Agent**
  - Net-SNMP
- **Other tools**
  - RRDTool
  - Ganglia
OpenFlow
OpenFlow is an API

- Control how packets are forwarded
- Implementable on COTS hardware
- Make deployed networks programmable
  - not just configurable
- Makes innovation easier
OpenFlow

OpenFlow Switch

Secure Channel

Flow Table

PC

OpenFlow Controller

OpenFlow Switch specification

OpenFlow Protocol

SSL
OpenFlow: Flow Table Entry

“Type 0” OpenFlow Switch

Rule
Action
Stats

- Packet + byte counters

1. Forward packet to port(s)
2. Encapsulate and forward to controller
3. Drop packet
4. Send to normal processing pipeline

<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>

## OF Examples

### Switching

<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>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>00:1f..</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>port6</td>
</tr>
</tbody>
</table>

### Flow Switching

<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>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>port3</td>
<td>00:20..</td>
<td>00:1f..</td>
<td>0800</td>
<td>vlan1</td>
<td>1.2.3.4</td>
<td>5.6.7.8</td>
<td>4</td>
<td>17264</td>
<td>80</td>
<td>port6</td>
</tr>
</tbody>
</table>

### Firewall

<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>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>22</td>
</tr>
</tbody>
</table>
OF Examples (cont’d)

Routing

<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>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>5.6.7.8</td>
<td>*</td>
<td>port6</td>
</tr>
</tbody>
</table>

VLAN Switching

<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>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>00:1f..</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>port6, port7, port9</td>
</tr>
</tbody>
</table>
OpenFlow Table: Basic Actions

- **All**: To all interfaces except incoming interface.
- **Controller**: Encapsulate and send to controller.
- **Local**: Send to its local networking stack.
- **Table**: Perform actions in the next flow table (table chaining or multiple table instructions).
- **In_port**: Send back to input port.
- **Normal**: Forward using traditional Ethernet.
- **Flood**: Send along minimum spanning tree except the incoming interface.
Centralized vs Distributed Control

Centralized Control

Distributed Control
Flow Routing vs. Aggregation

Flow-Based
- Every 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

Aggregated
- One flow entry covers large groups of flows
- Wildcard flow entries
- Flow table contains one entry per category of flows
- Good for large number of flows, e.g. backbone
Reactive vs. Proactive

**Reactive**
- First packet of flow triggers controller to insert flow entries
- Efficient use of flow table
- Every flow incurs small additional flow setup time
- If control connection is 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 very large number and/or aggregated (wildcard) rules
# Additional Feature to Rules and Stats

<table>
<thead>
<tr>
<th>OpenFlow Version</th>
<th>Match fields</th>
<th>Statistics</th>
<th># Matches</th>
<th># Instructions</th>
<th># Actions</th>
<th># Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>v 1.0</td>
<td>Ingress Port</td>
<td>Per table statistics</td>
<td>18</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Ethernet: src, dst, type, VLAN</td>
<td>Per flow statistics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IPv4: src, dst, proto, ToS</td>
<td>Per port statistics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TCP/UDP: src port, dst port</td>
<td>Per queue statistics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v 1.1</td>
<td>Metadata, SCTP, VLAN tagging</td>
<td>Group statistics</td>
<td>23</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MPLS: label, traffic class</td>
<td>Action bucket statistics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v 1.2</td>
<td>OpenFlow Extensible Match (OXM)</td>
<td></td>
<td>14</td>
<td>18</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>IPv6: src, dst, flow label, ICMPv6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v 1.3</td>
<td>PBB, IPv6 Extension Headers</td>
<td>Per-flow meter</td>
<td>14</td>
<td>26</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Per-flow meter band</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v 1.4</td>
<td>—</td>
<td>Optical port properties</td>
<td>14</td>
<td>27</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
OpenFlow (SDN) Controllers

- **POX**: (Python) Out of Date.
- **IRIS**: (Java) Scalability and High Availability
- **MUL**: (C) MũL, is an openflow (SDN) controller. It has a C based multi-threaded infrastructure at its core.
- **NOX**: (C++/Python) NOX was the first OpenFlow controller.
- **Jaxon**: (Java) Jaxon is a NOX-dependent Java-based OpenFlow Controller.
- **Trema**: (C/Ruby) Trema is a full-stack framework for developing OpenFlow controllers in Ruby and C.
- **Beacon**: (Java) Beacon supports both event-based and threaded operation.
- **Floodlight**: (Java) It was forked from the Beacon controller, originally developed by David Erickson at Stanford.
- **OpenDaylight**: (Java) ODL is a modular open platform for customizing and automating networks of any size and scale
- And many more.
OpenVswitch
OpenFlow Based Software Switch

- Open Source Virtual Switch
- Software only implementation of OpenFlow
- Can Run as a stand alone hypervisor switch or as a distributed switch across multiple physical servers.
- Default switch in XenServer 6.0, Xen Cloud Platform and supports Proxmox VE, VirtualBox, Xen KVM.
- Integrated into many cloud management systems including OpenStack, openQRM, OpenNebula, and oVirt.
- Distributed with Ubuntu, Debian, Fedora Linux, FreeBSD, Windows, etc.
- More info at OpenVSwitch doc. repository
Mininet

- Mininet is a system for rapidly prototyping large networks on a single machine
- Lightweight OS-level virtualization:
  - Isolated network namespace
  - Constrained CPU usage on isolated namespace
- CLI and Python APIs
- Can:
  - Create custom topologies
  - Run real programs
  - Custom packet forwarding using OpenFlow

```
net = Mininet()  # net is a Mininet() object
h1 = net.addHost( 'h1' )  # h1 is a Host() object
h2 = net.addHost( 'h2' )  # h2 is a Host()
s1 = net.addSwitch( 's1' )  # s1 is a Switch() object
c0 = net.addController( 'c0' )  # c0 is a Controller()
net.addLink( h1, s1 )  # creates a Link() object
net.addLink( h2, s1, bw=10, delay='50ms' )
net.start()  # start net
CLI( net )
net.stop()  # stop net
```

Can be controlled with external OpenFlow controller
Mininet API reference and examples
Case Study: ORBIT Network
“Big” Picture
Network Programming
Linux Protocol Stack Layering

- BSD sockets layer – provides standard UNIX sockets API for inter-process communication (not necessarily only across network).
- INET layer – manages communication end-points for the IP based protocols (TCP and UDP).
- Device driver – manages physical device.
Socket Address Structures

**Generic socket address structure**

```c
struct sockaddr {
    uint8_t sa_len;
    sa_family_t sa_family;
    char sa_data[14];
};
```

- Always passed by reference
- Socket functions take pointer to the generic socket address structure
- Protocol dependant address conversion functions
- Linux sockaddr don't have a length field, only a family

**IPv4**

- `sockaddr_in()`
- Length: 16 bytes
- `AF_INET`
- 16 bit port#
- 32-bit IPv4 address

**IPv6**

- `sockaddr_in6()`
- Length: 24 bytes
- `AF_INET6`
- 16 bytes
- 128-bit IPv6 address
- 32-bit Flow label

**Datalink**

- `sockaddr_dl()`
- Length: variable
- `AF_LINK`
- Interface index
- Type
- Interface name and link-layer address
- Name len.
- Addr. Len.
- Sel. len

**Unix**

- `sockaddr_un()`
- Length: variable
- `AF_LOCAL`
- Pathname (up to 1024 bytes)

### Definitions

```c
#define AF_UNSPEC 0
#define AF_UNIX 1      /* Unix domain sockets */
#define AF_LOCAL 1      /* POSIX name for AF_UNIX */
#define AF_INET 2 ...   /* ATM PVCs */
#define AF_X25 9       /* Reserved for X.25 project */
#define AF_INET6 10     /* IP version 6 */
```
Support Functions

**Byte Ordering Functions**

- Endianness - order in which integer values are stored as bytes in computer memory (byte order)

<table>
<thead>
<tr>
<th></th>
<th>Little endian</th>
<th>Big endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr A</td>
<td>Low-byte</td>
<td>High-byte</td>
</tr>
<tr>
<td>addr A+1</td>
<td>High-byte</td>
<td>Low-byte</td>
</tr>
</tbody>
</table>

- Internet protocol use big-endian byte ordering for multibyte integers (*network byte order*)

“l” – long (32 bit), “s” – short (16-bit)

On systems with same host and network byte order these are null macros

**Byte Manipulating Functions**

```c
#include <strings.h>

void bzero(void *s, size_t n);
void bcopy(const void *src, void *dest, size_t n);
int bcmp(const void *s1, const void *s2, size_t n);
```

**Address Conversion Functions**

- Convert between ASCII string and network byte ordered binary values

```c
#include <netinet/in.h>
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
in_addr_t inet_addr(const char *cp);
int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src, char *dst, socklen_t cnt);
```
# Socket Functions

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

- Creates an endpoint for communication and returns a (socket) descriptor.

- Functions `setsockopt` and `getsockopt` can be used to control the socket operation.

<table>
<thead>
<tr>
<th>Family (domain)</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_UNIX, PF_LOCAL</td>
<td>Local communication</td>
</tr>
<tr>
<td>PF_INET</td>
<td>IPv4 Internet protocols</td>
</tr>
<tr>
<td>PF_INET6</td>
<td>IPv6 Internet protocols</td>
</tr>
<tr>
<td>PF_IPX</td>
<td>IPX - Novell protocols</td>
</tr>
<tr>
<td>PF_BLUETOOTH</td>
<td>Bluetooth sockets</td>
</tr>
<tr>
<td>PF_X25</td>
<td>ITU-T X.25 / ISO-8208 protocol</td>
</tr>
<tr>
<td>PF_AX25</td>
<td>Amateur radio AX.25 protocol</td>
</tr>
<tr>
<td>PF_ATM PVC</td>
<td>Access to raw ATM PVCs</td>
</tr>
<tr>
<td>PF_APPLETALK</td>
<td>Appletalk</td>
</tr>
<tr>
<td>PF_PACKET</td>
<td>Low level packet interface</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCK_STREAM</td>
<td>Sequenced, reliable, two-way, connection-based byte streams (TCP)</td>
</tr>
<tr>
<td>SOCK_DGRAM</td>
<td>Connectionless, unreliable fixed maximum length datagrams (UDP)</td>
</tr>
<tr>
<td>SOCK_RAW</td>
<td>Raw network protocol access</td>
</tr>
<tr>
<td>SOCK_RDM</td>
<td>Reliable datagram layer that does not guarantee ordering</td>
</tr>
<tr>
<td>SOCK_SEQPACKET</td>
<td>Sequenced, reliable, two-way connection-based datagram layer</td>
</tr>
<tr>
<td>SOCK_PACKET</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>
Socket Functions (contd.)

#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);

- Assigns local address to the sockfd socket (also referred to as "passing a name to a socket")
  - Used by servers to "bind their well known port"; on clients, can be used to a specific IP address for multi-homed hosts.
  - Return 0 on success; -1 and errno on failure.

#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);

- Connects the sockfd socket to the address specified by serv_addr
  - Clients: kernel will chose source address and ephemeral port
  - For connection-based protocol it will attempt to connect; for connectionless sets the default destination address.
  - Return 0 on success; -1 and errno on failure.
Socket Functions (contd.)

- **Indicated willingness to accept incoming connections**
  - `backlog` – maximum queue length for established sockets waiting to be accepted (the ones for which three-way handshake has completed)
  - Return 0 on success; -1 and errno on failure.

- **Extracts the first from the queue of pending connections, creates a new connected socket, and returns a new file descriptor referring to that socket.**
  - Only for connection-based protocols; blocks until connection is available
  - `addr` is filled in with the address of the peer
  - Returns `socketfd` on success; -1 and errno on failure.
Socket Functions (contd.)

- Closes the socket descriptor.
  - Returns \textit{socketfd} on success; -1 and \text{errno} on failure.

- Used to receive and send data
  - If no data, the receive call waits (unless nonblocking); otherwise it returns any data available, up to \textit{len}.
  - For send, socket has to be in connected state (endpoint has to be set); error is also returned for message too long to pass atomically through the underlying protocol.
  - All functions return the length of the data on success; -1 and \text{errno} on failure.

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
```

```c
#include <sys/types.h>
#include <sys/socket.h>

int close(int fd);
```
Simple TCP Client-Server Example

**ECHO Server:**
- Client sends a line of text
- Server reads that line end echoes it back
- Client reads the line and displays it on its stdout

TCP SERVER
- `socket()`
- `bind()`
- `listen()`
- `accept()`

TCP CLIENT
- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`

Data (request)
- `read()`
- `write()`

Data (reply)
- `read()`
- `write()`

End-of-file notification
- `close()`
/*  
* Simple echo client  
*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <strings.h>  
#include <errno.h>  
#include <resolv.h>  
#include <sys/socket.h>  
#include <arpa/inet.h>  

#define SERVER "127.0.0.1"  
#define ECHO_PORT 7+1000  
#define MAXBUF 1024  

int main()  
{  
    int sockfd;  
    struct sockaddr_in servaddr;  
    char buffer[MAXBUF];  
    
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )  
    {  
        perror("socket"); exit(errno);  
    }  
    
    /* Server address structure */  
    bzero(&servaddr, sizeof(servaddr));  
    servaddr.sin_family = AF_INET;  
    servaddr.sin_port = htons(ECHO_PORT);  
    if ( inet_aton(SERVER, &servaddr.sin_addr) == 0 ) // Address  
        {  
            perror(SERVER); exit(errno);  
        }  
    
    /* Connect to server */  
    if ( connect(sockfd, (struct sockaddr*)&servaddr,  
                sizeof(servaddr)) != 0 ) // Address  
        {  
            perror("connect"); exit(errno);  
        }  
    
    sprintf(buffer,"Test message");  
    send(sockfd, buffer, sizeof(buffer), 0);  
    bzero(buffer, MAXBUF);  
    recv(sockfd, buffer, sizeof(buffer), 0);  
    printf("%s
", buffer);  
    close(sockfd);  
    return 0;  
}
/* Simple echo server. */
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <resolv.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define ECHO_PORT 7 + 1000
#define MAXBUF 1024
#define LISTENQ 20

int main(int argc, char *argv[])
{
  int sockfd;
  struct sockaddr_in servaddr;
  char buffer[MAXBUF];

  /* Create a TCP socket */
  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
    /* Show why we failed */
    perror("socket");
    exit(errno);
  }

  /* Set our address and port */
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = INADDR_ANY; // On any interface
  servaddr.sin_port = htons(ECHO_PORT); // Our port

  /* Let's bind to the port/address */
  if ( bind(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) != 0 )
    {
    perror("bind");
    exit(errno);
  }

  /* We are the server */
  if ( listen(sockfd, LISTENQ) != 0 )
    {
    perror("listen");
    exit(errno);
  }

  for (; ; )
    {
    int clientfd;
    struct sockaddr_in clientaddr;
    int addrlen = sizeof(clientaddr);

    /* Accept a client connection */
    clientfd = accept(sockfd, (struct sockaddr*)&clientaddr, &addrlen);
    printf("%s:%d connected\n", inet_ntoa(clientaddr.sin_addr),
           ntohs(clientaddr.sin_port));
    send(clientfd, buffer, recv(clientfd, buffer, MAXBUF, 0), 0);
    close(clientfd);
  }

  close(sockfd);
  return 0;
}