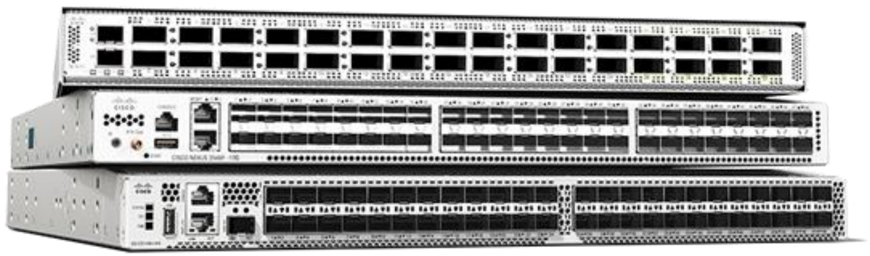
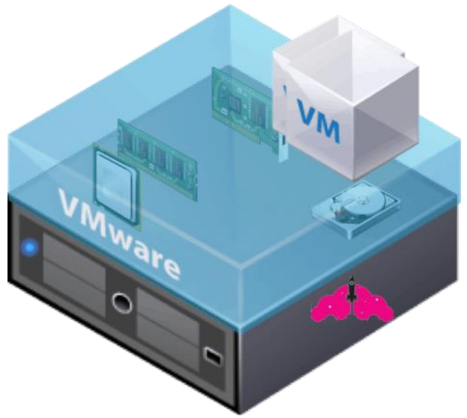
Problem / Overview

Course: Networking Principles in Practice – Linux Networking
Module: Linux Networking Intro

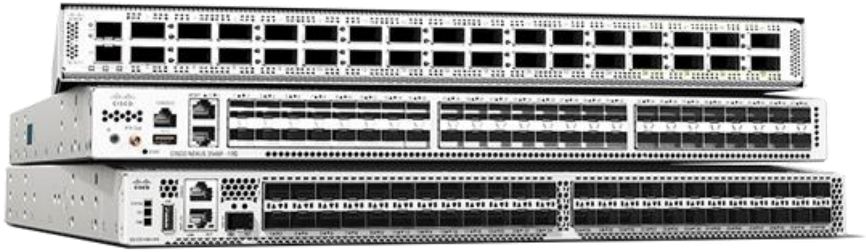
University of Colorado Boulder

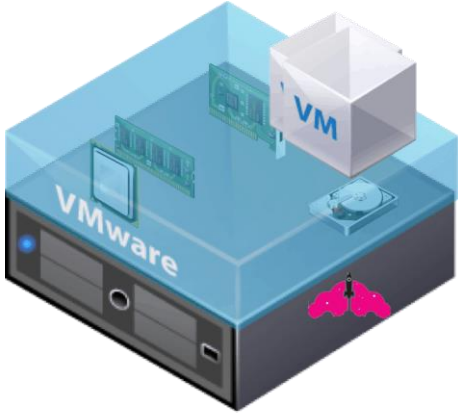
Network Appliances Come in a Variety of Forms





Network Appliances From a Variety of Vendors





cisco

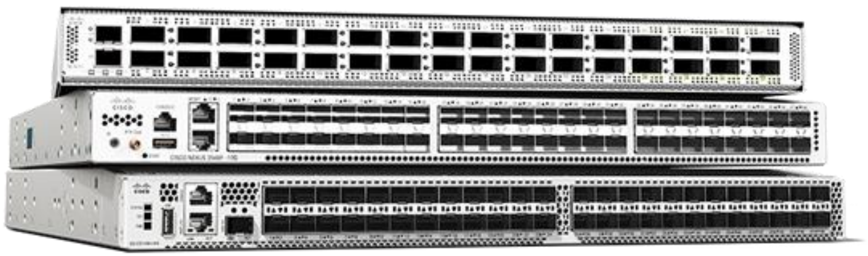
paloalto® NETWORKS

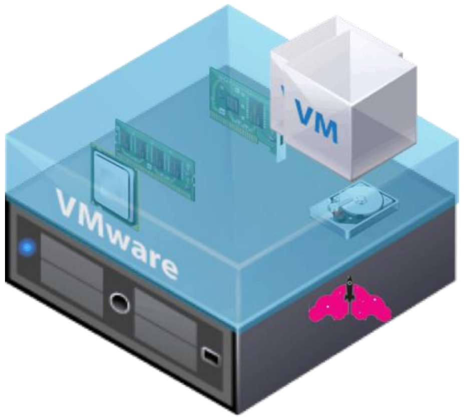
JUNIPer® NETWORKS

ARISTA

Network Appliances
For a Variety of Uses

Data center networks
Core Internet router
Wireless Gateway
Perimeter Firewall
Cloud Routers
…





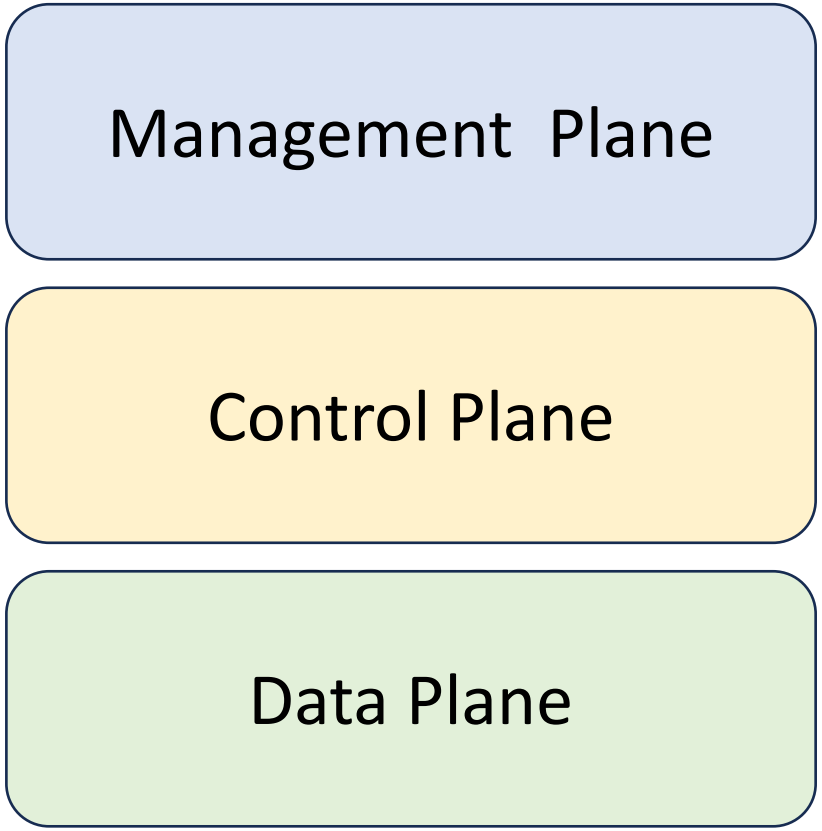
cisco

paloalto NETWORKS

JUNIPer NETWORKS

ARISTA

Network Appliance Planes



Network Appliance Planes

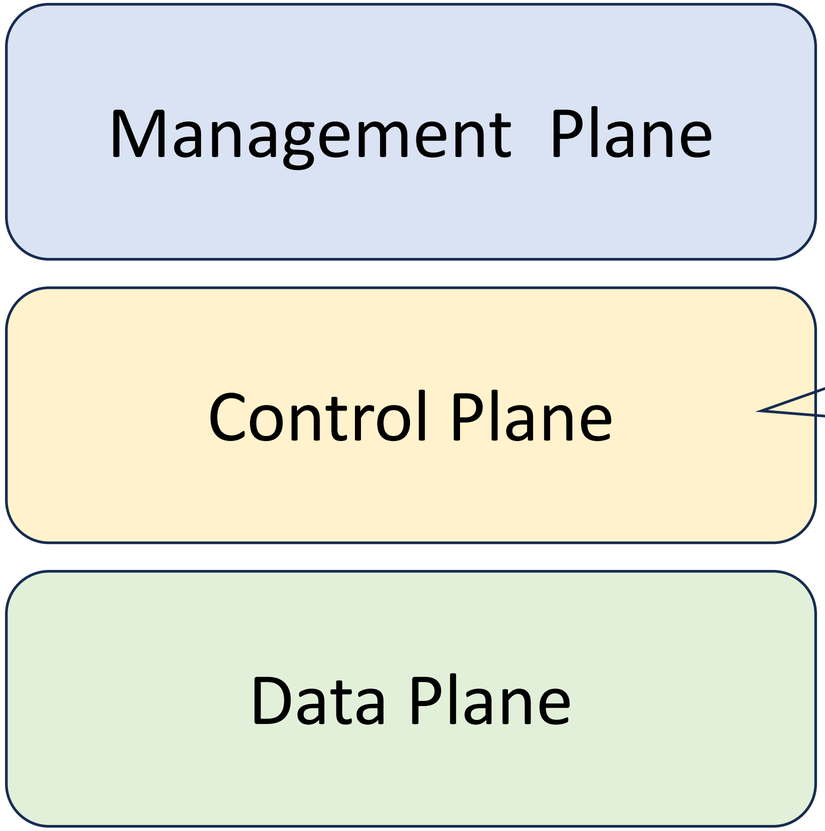
Management Plane

Control Plane

Data Plane

Data plane:
- Processes packets
- Needs to be fast
- Example functions: NAT, Forwarding, filtering

Network Appliance Planes



Control plane:

- Runs protocols to determine how the data plane should process traffic
- Example functions: BGP, OSPF, DHCP

Network Appliance Planes

Management Plane

Control Plane

Data Plane

Management plane:

- Means for how a user or automation software configures and monitors the appliance
- Examples: Command Line Interface, Terraform, SNMP

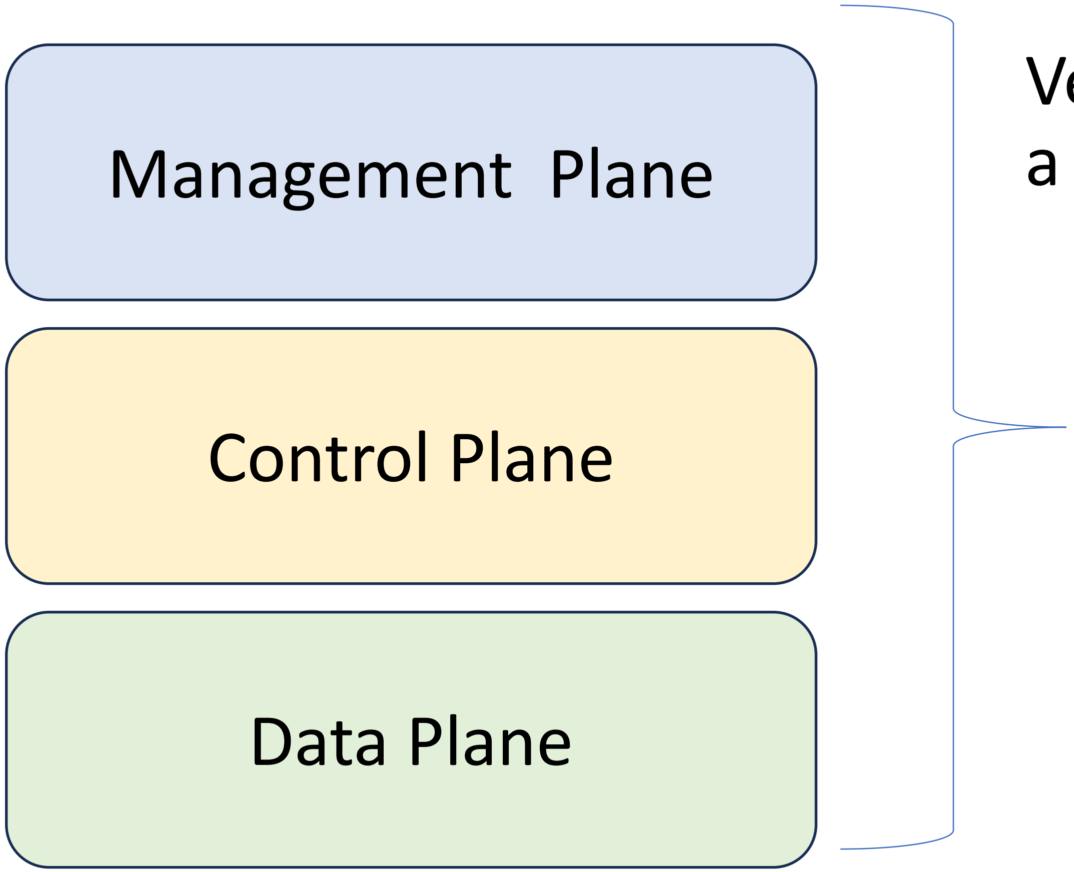
Motivation 1 – Linux Covers the Whole Stack

Management Plane
(Linux Utilities)

Control Plane
(Linux Ecosystem)

Data Plane
(Linux Kernel)

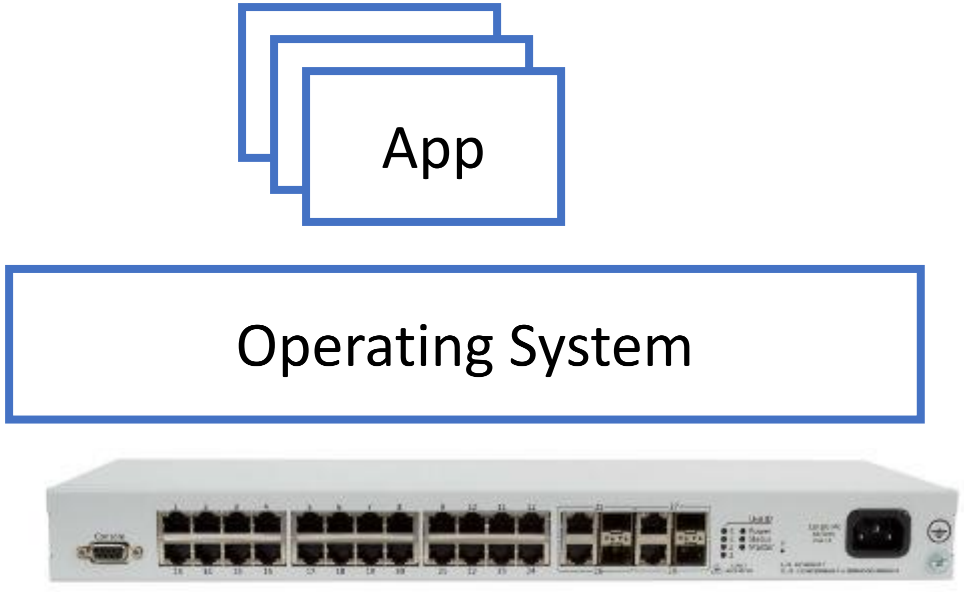
Legacy – Tight Coupling



Vendor provides everything in a vertically integrated solution

Recent Trend - Disaggregation

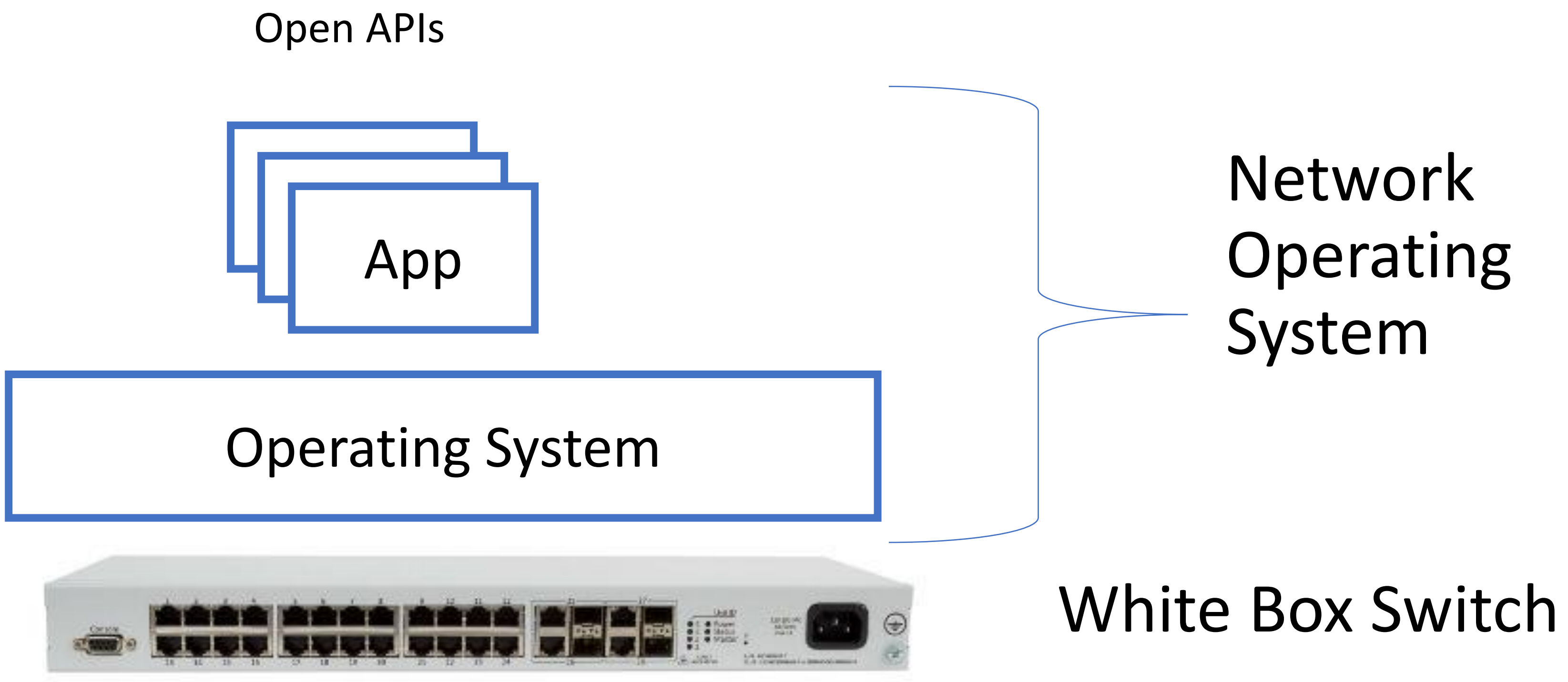
Open APIs



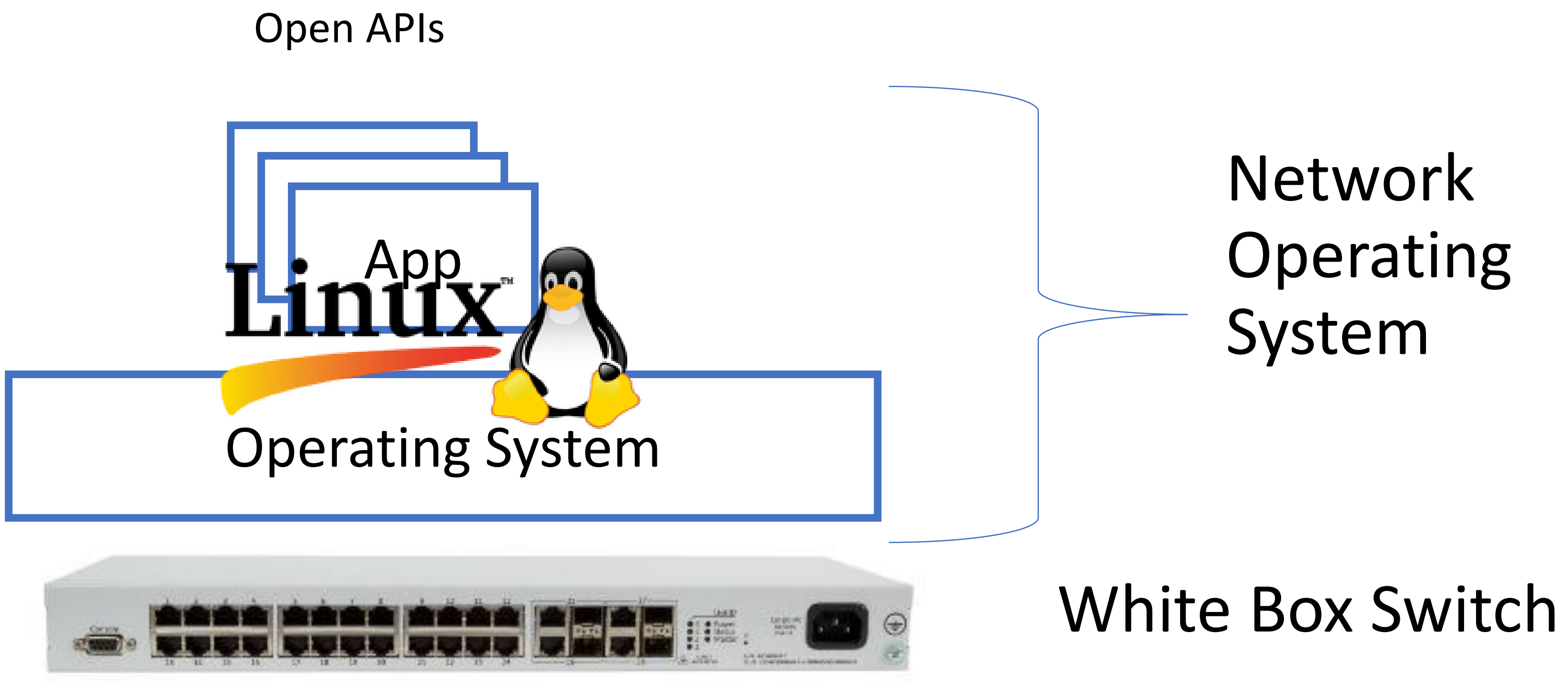
Recent Trend - Disaggregation

Benefits:

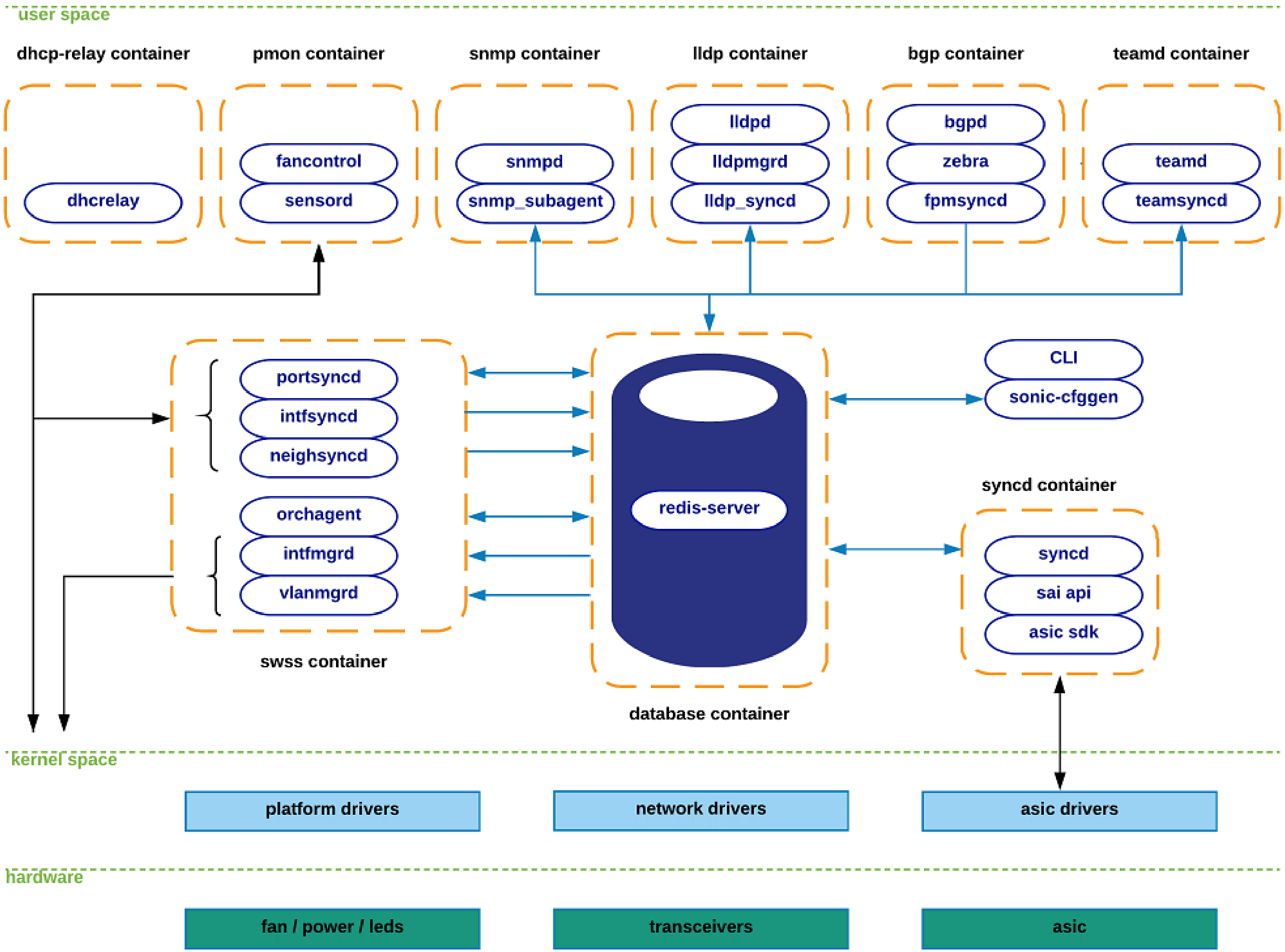
- More choice
- Extensibility



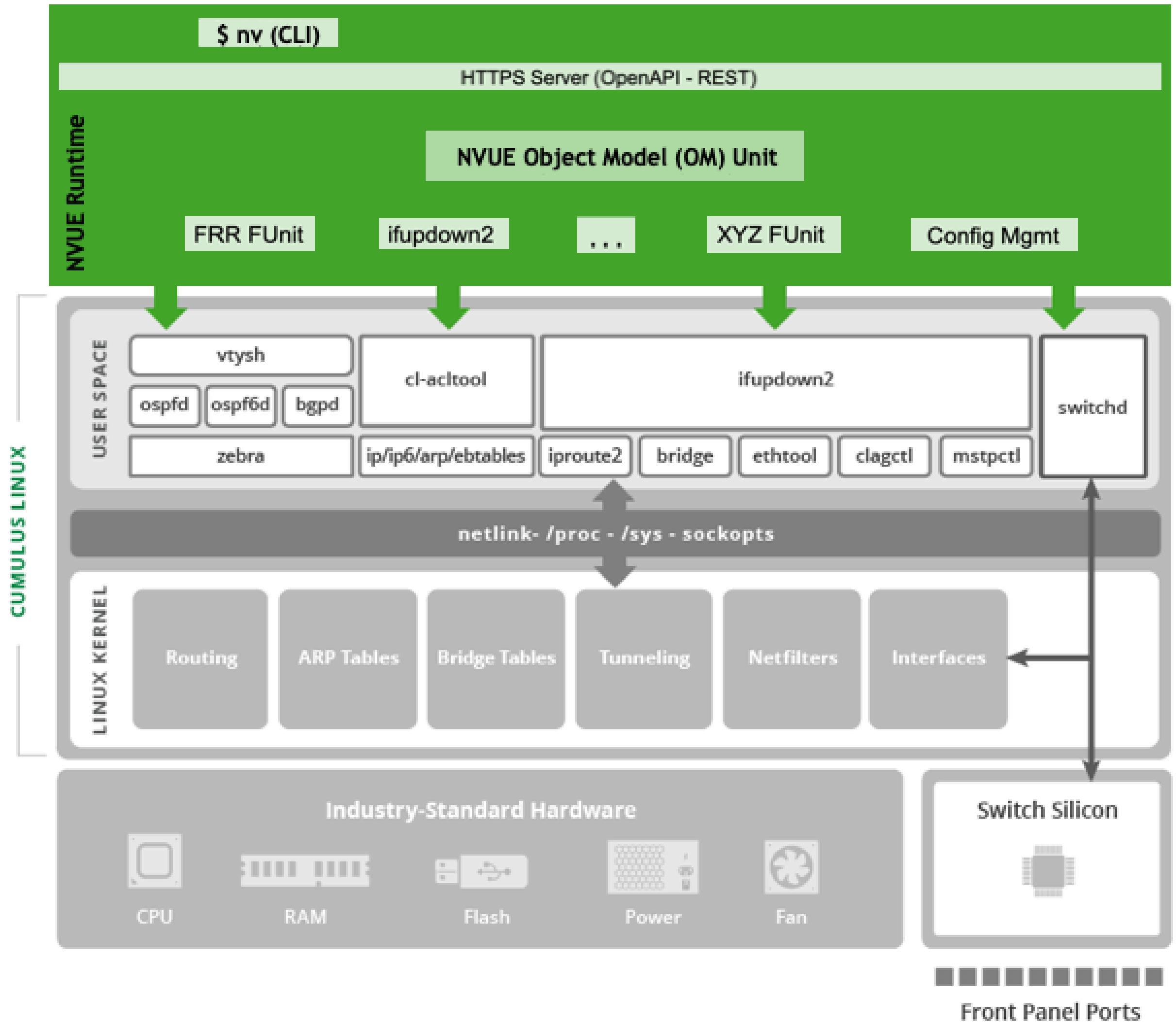
Motivation 2 – Linux as the NOS



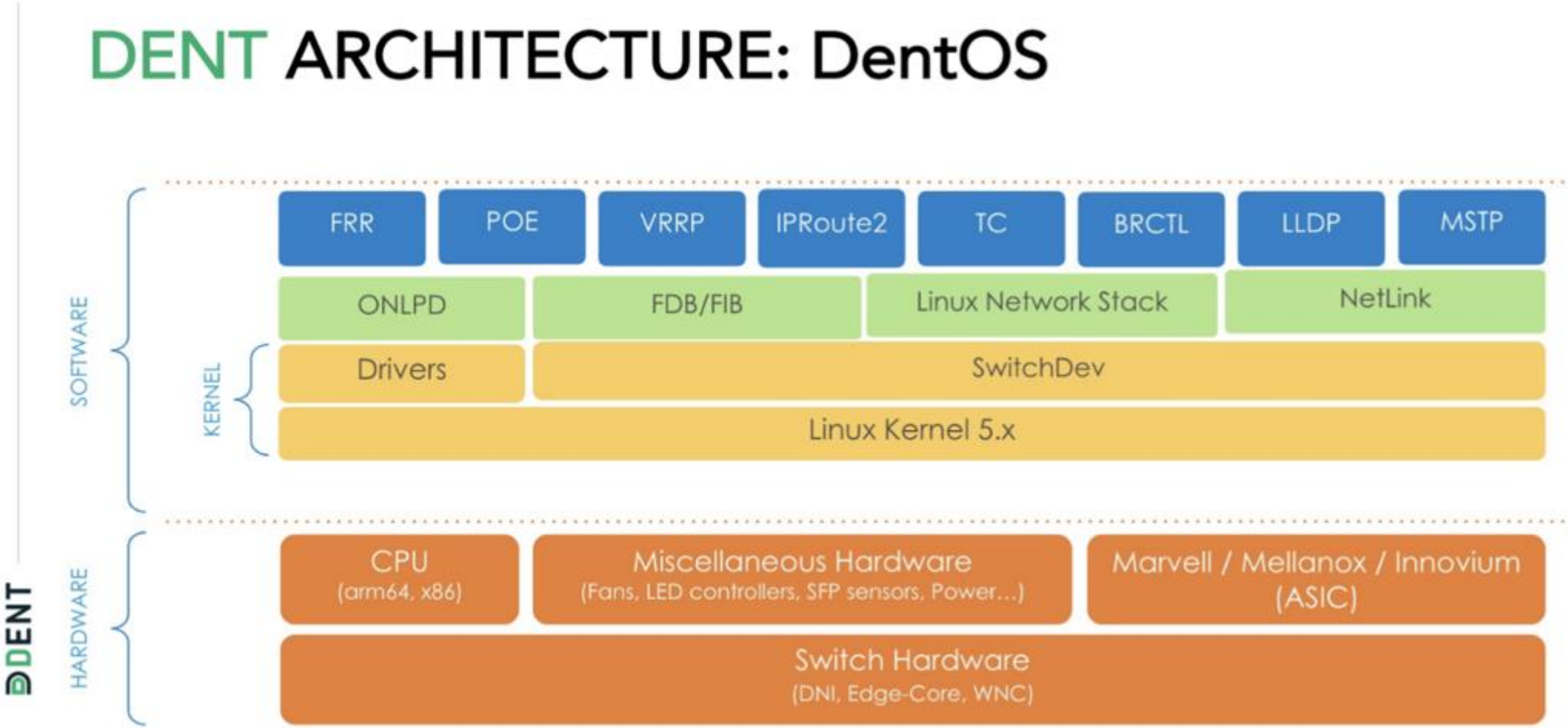
Example 1 - SONiC



Example 2 - Cumulus



Example 3 – Dent OS



Upcoming Lessons

- Linux data plane overview
- Lab environment
- Linux utilities for troubleshooting
- Linux device management utility

University of Colorado **Boulder**

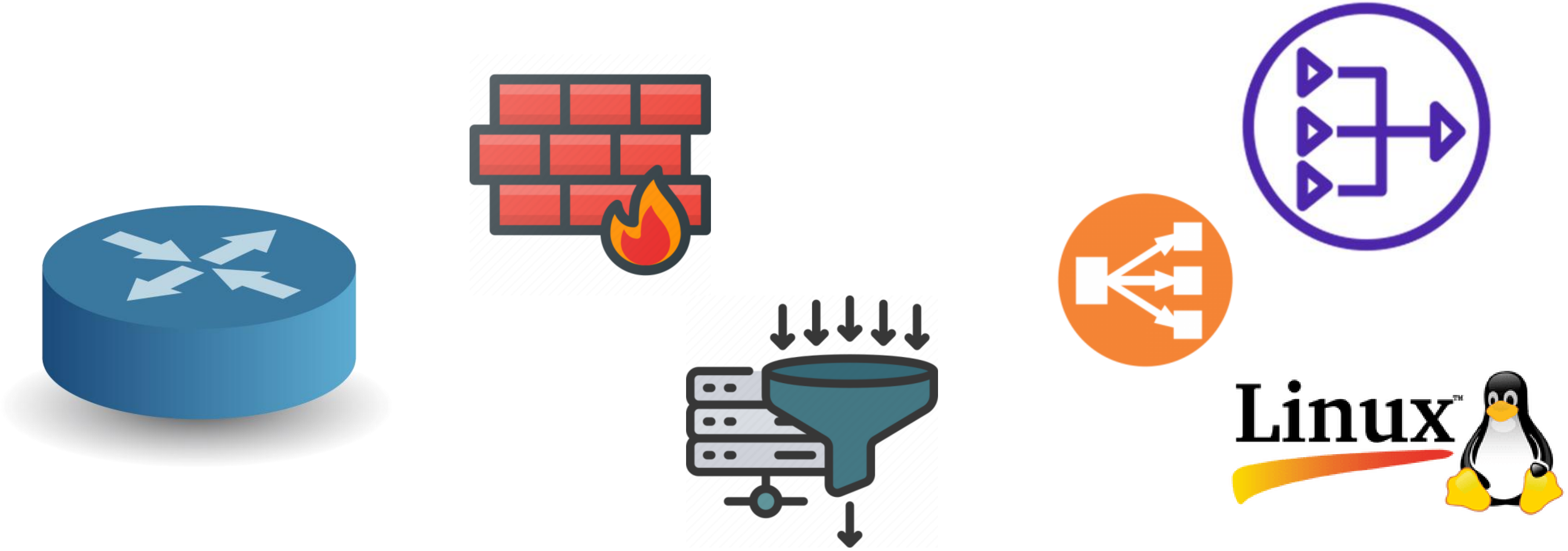
Linux Data Plane

Course: Networking Principles in Practice – Linux Networking
Module: Linux Networking Intro

University of Colorado Boulder

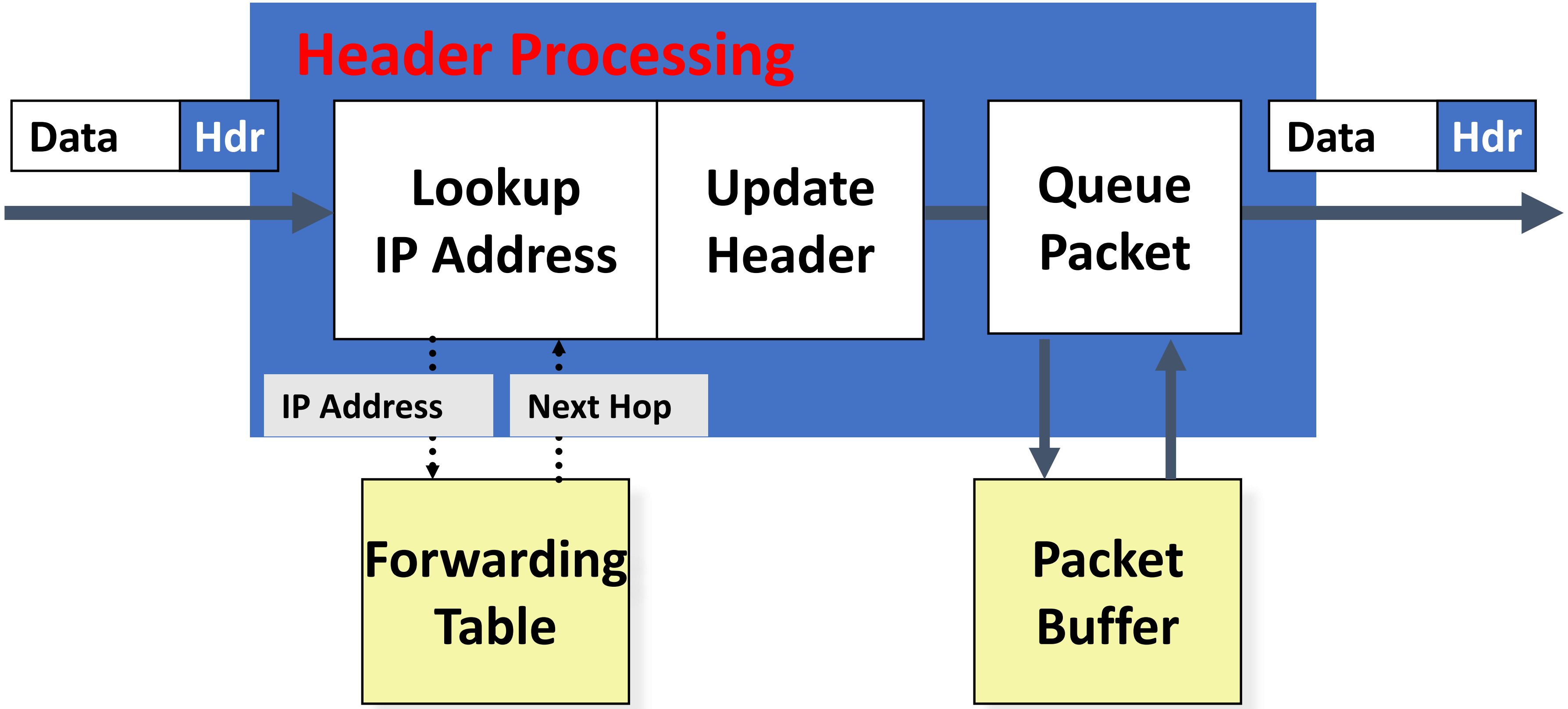
Linux Networking – Rich Functionality

Forwarding, NATing, filtering, bridging, load balancing, traffic shaping



Generic Router Architecture

- Key take aways: Pipeline of functions, tables driving processing



Slide from Princeton COS 461 (Mike Freedman, Jen Rexford, Nick Feamster)

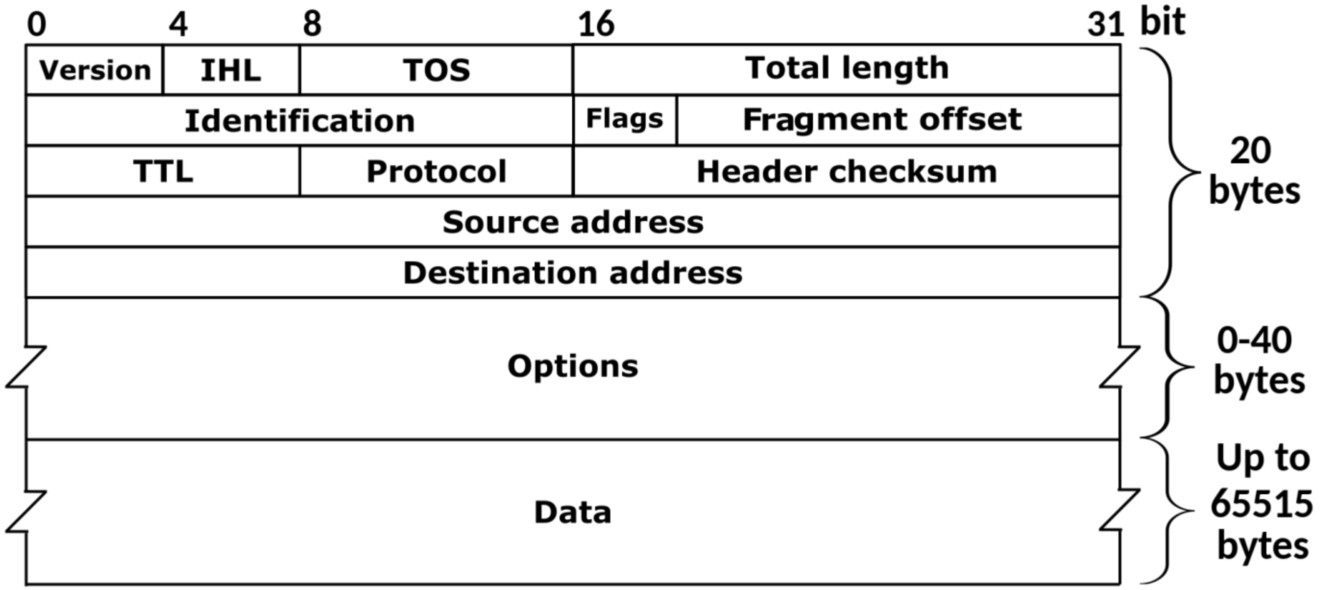
Protocol Headers

Ethernet

Dest	Source	Type	data	FCS
48 bits	48 bits	16 bits		32 bits

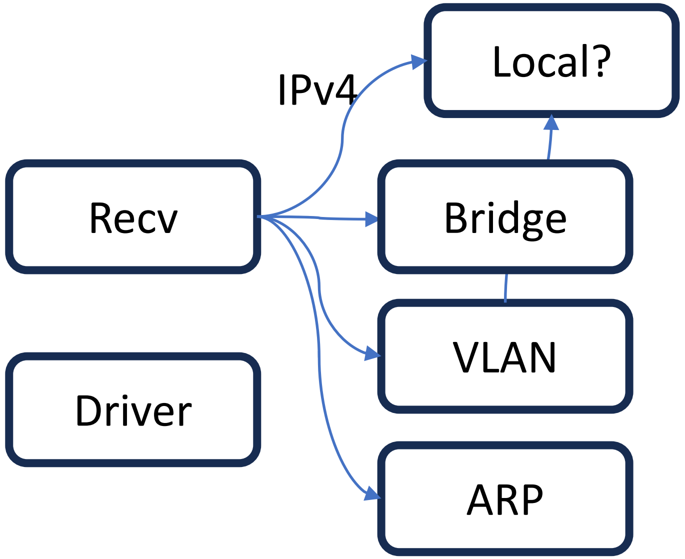
Types: IPv4, IPv6, ARP, VLAN, etc.

IPv4

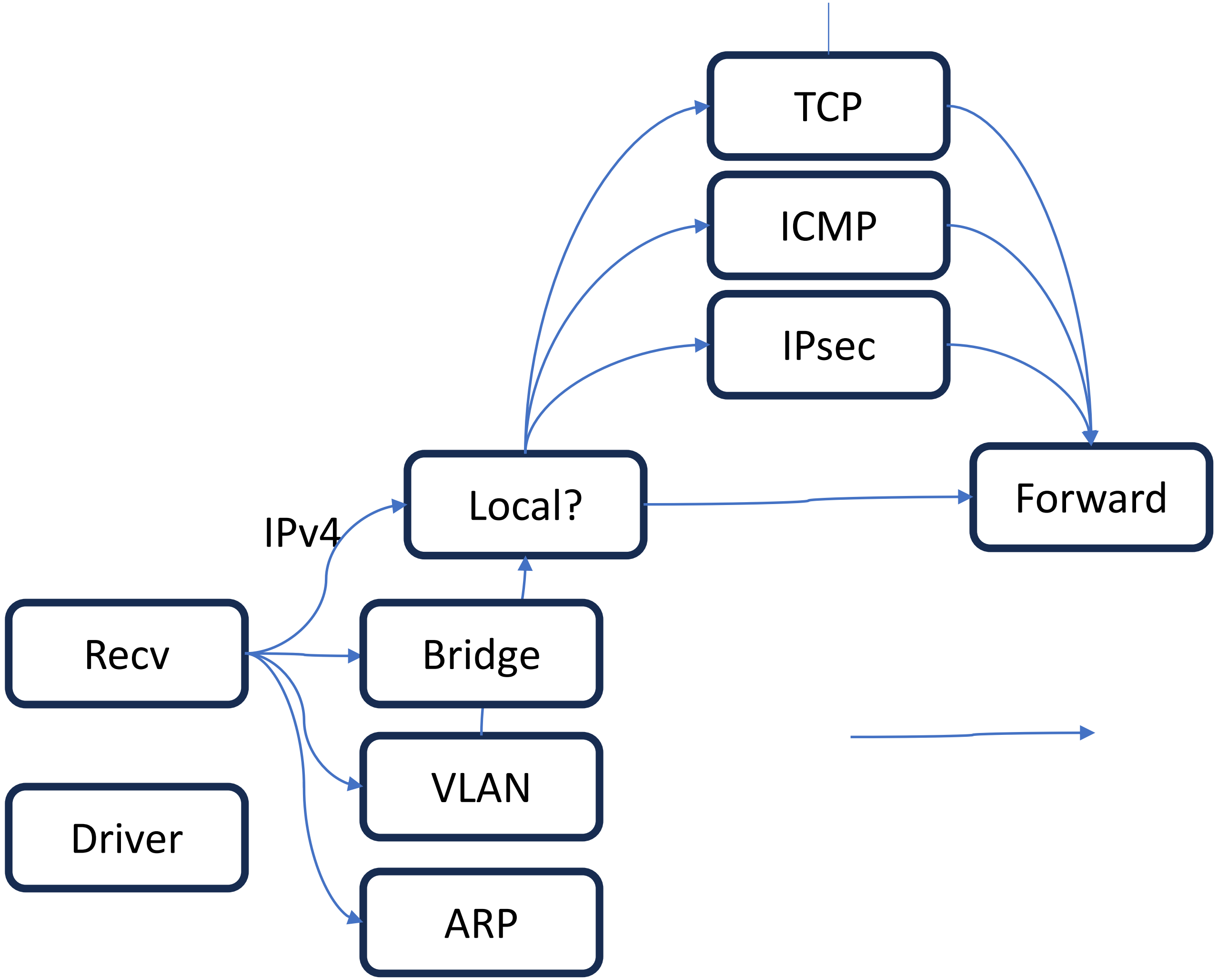


Types: ICMP, TCP, UDP, GRE, ESP

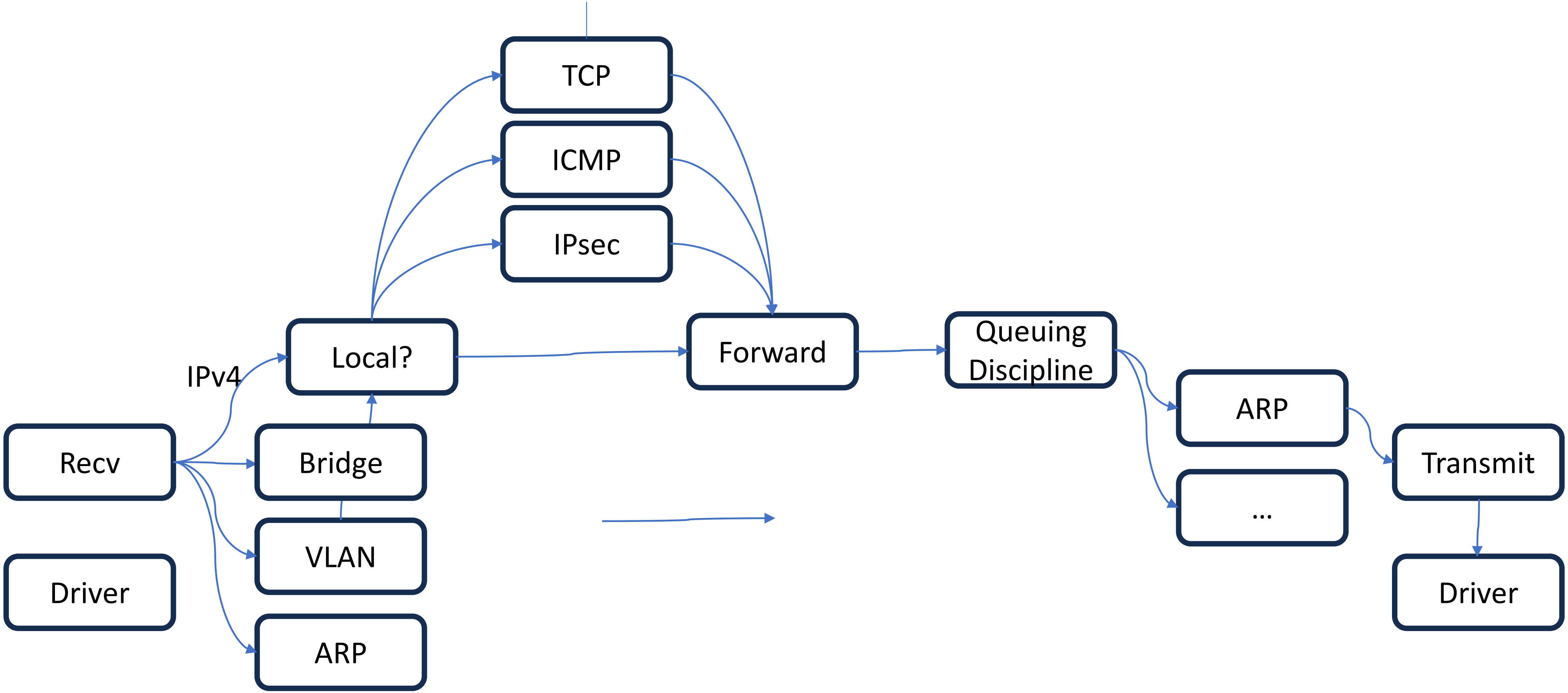
Simplified Linux Data Plane



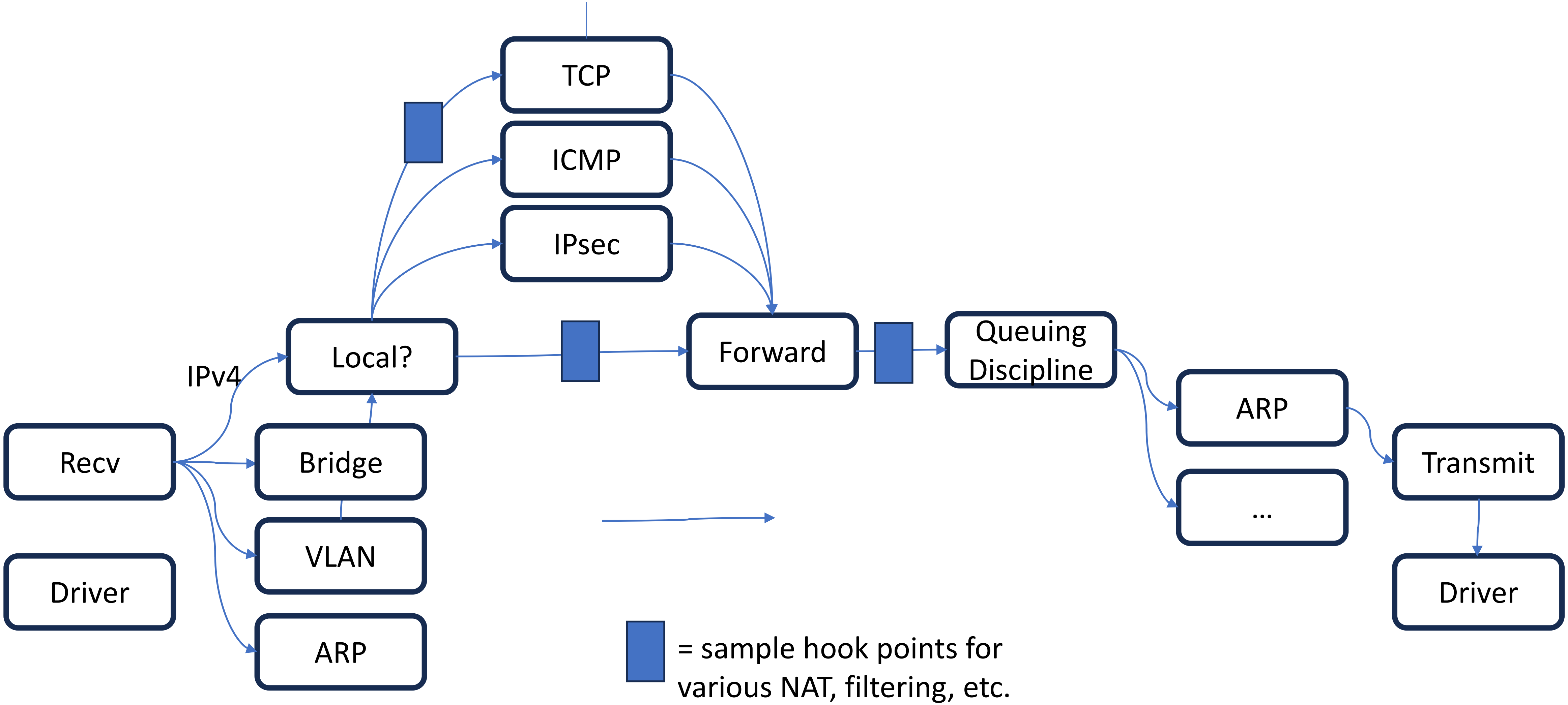
Simplified Linux Data Plane



Simplified Linux Data Plane



Simplified Linux Data Plane



Take Away

There is a rich set of functionality provided by Linux

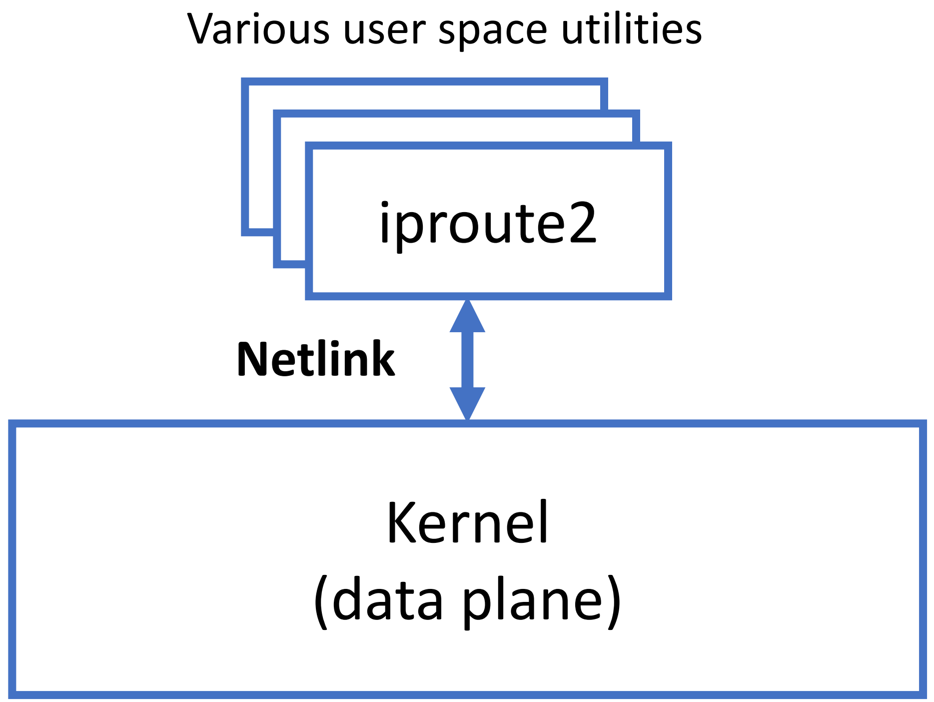
Very powerful, if you know how to use it

Linux Networking Utilities (sample)

- iproute2 – interfaces, bridges, forwarding, …
- iptables – classification, NAT, filtering
- tc (traffic control) – queueing discipline
- ipvsadm – load balancing

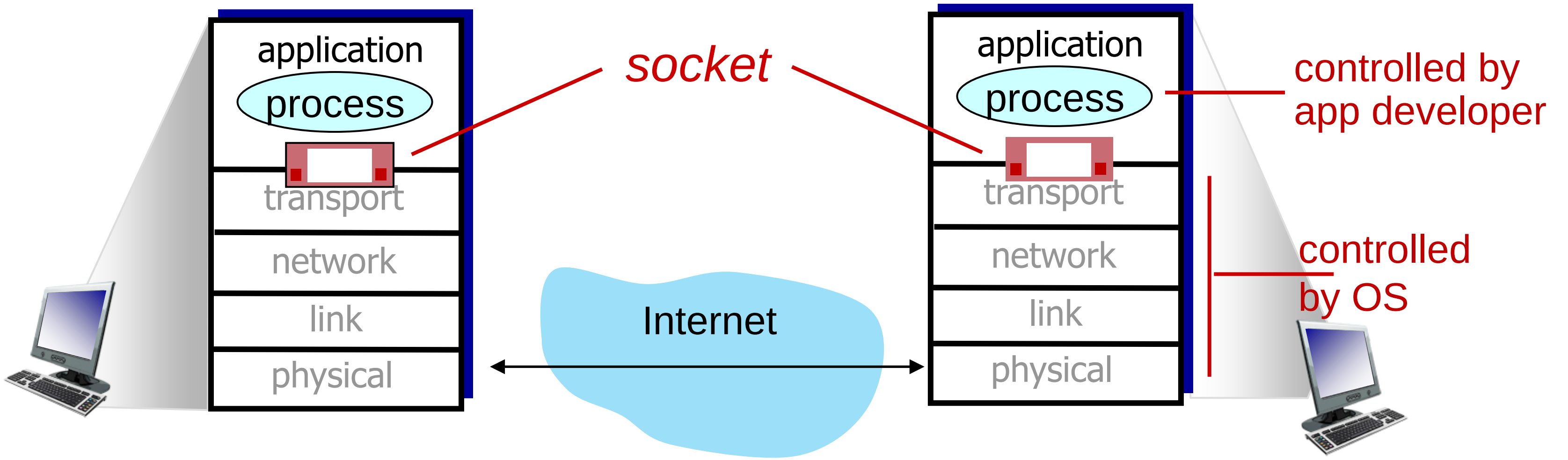
Netlink

- How do these utilities tell the kernel how it should process the traffic?

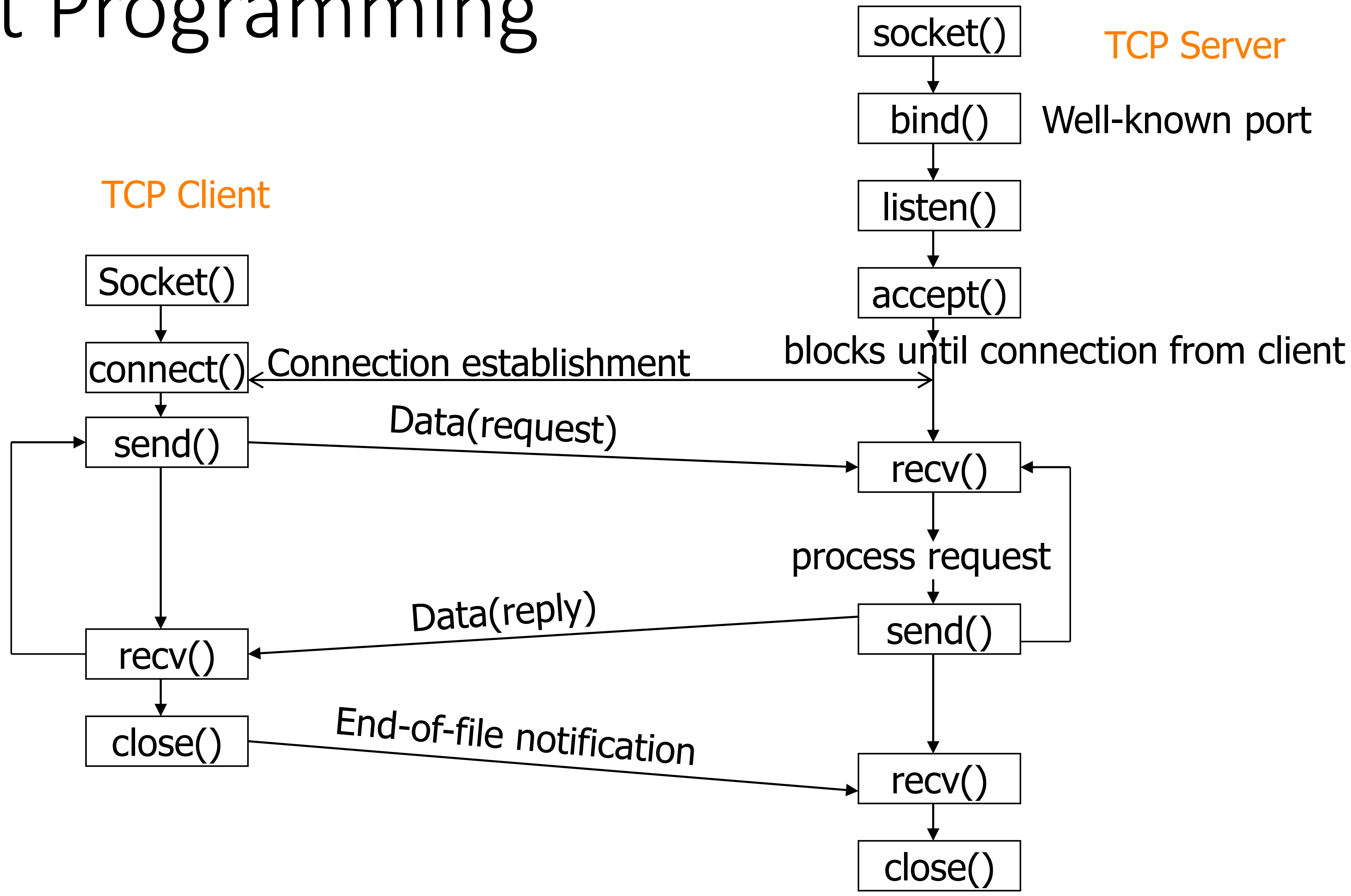


Berkeley Sockets

- Originated in 1983 and has been the standard since
- A socket is an abstract representation for the local endpoint of a network communication path.
- App. puts data into socket, other app. gets data from the socket.

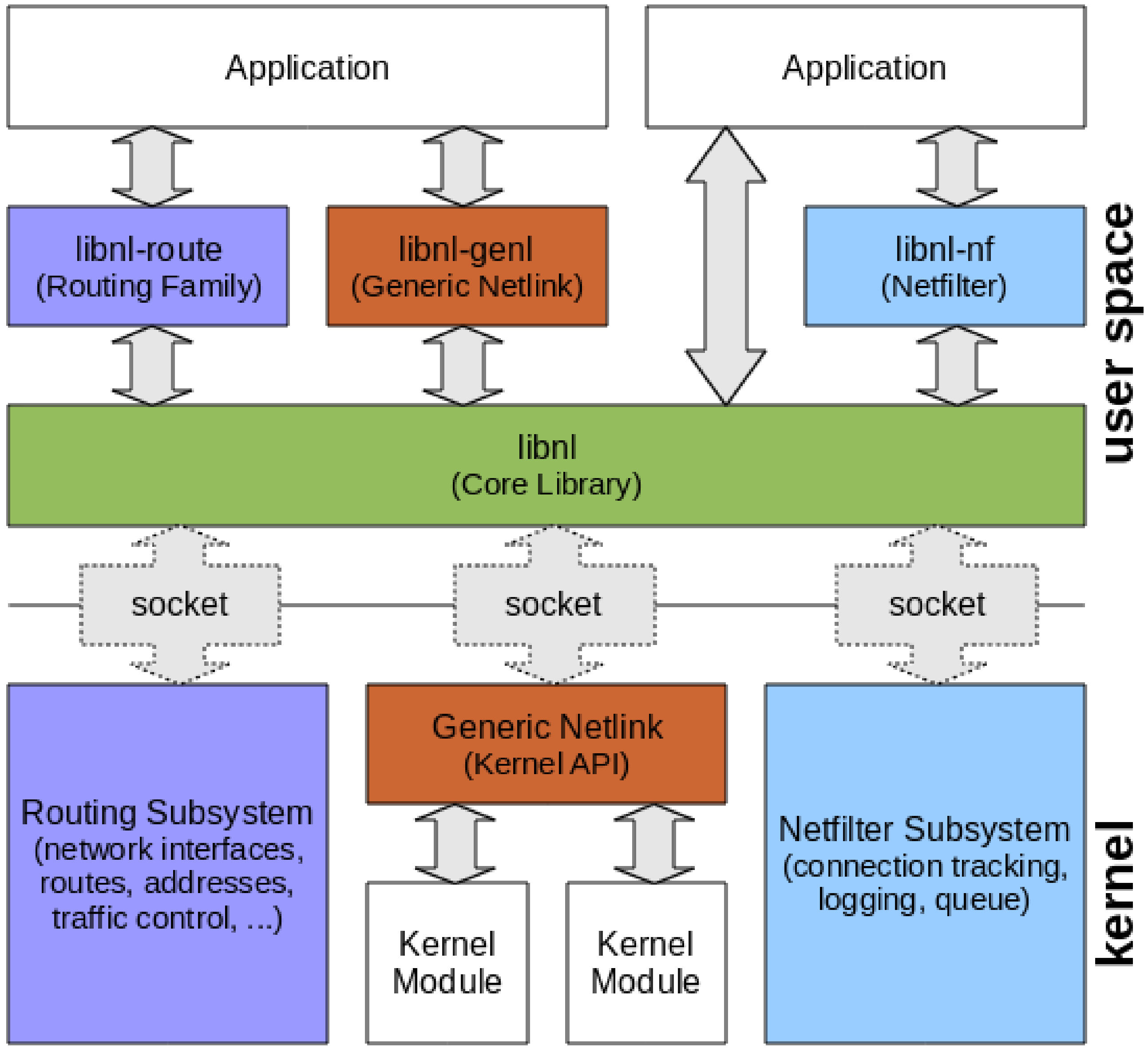


Socket Programming



Netlink Sockets

- Each application is a client
- Servers run in kernel
- Supports multicast
- Libraries hide details



https://www.infradead.org/~tgr/libnl/

Creating the Socket (Addressing)

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

Domain = AF_NETLINK,
Type = SOCK_RAW,
Protocol = family (see man page)
Example families: NETLINK_ROUTE, NETLINK_NETFILTER

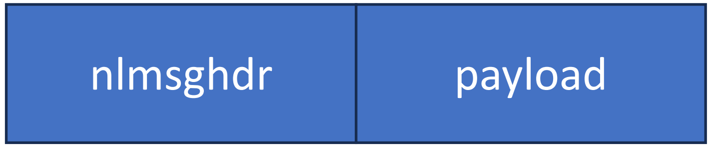
```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

```
struct sockaddr_nl {
    sa_family_t nl_family;    /* AF_NETLINK */
    unsigned short nl_pad;    /* zero */
    __u32 nl_pid;             /* process pid */
    __u32 nl_groups;          /* mcast groups bit mask */
} nladdr;
```

Each family has set of multicast groups:
Examples: RTMGRP_LINK, RTMGRP_IPV4_IFADDR

Sending Messages

```
int send(int sockfd, const void *msg, int len, int flags);
```




```
struct nlmsghdr {
__u32 nlmsg_len; /* Length of message including header */
__u16 nlmsg_type; /* Message content */
__u16 nlmsg_flags; /* Additional flags */
__u32 nlmsg_seq; /* Sequence number */
__u32 nlmsg_pid; /* Sending process port ID */
};

nlmsg->nlmsg_type = RTM_NEWLINK;
nlmsg->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
```

```
struct rtmsg {
unsigned char       rtm_family;
unsigned char       rtm_dst_len;
unsigned char       rtm_src_len;
unsigned char       rtm_tos;

unsigned char       rtm_table;
unsigned char       rtm_protocol;
unsigned char       rtm_scope;
unsigned char       rtm_type;
unsigned            rtm_flags;
};
```

Summary

- Data plane – functions with tables organized in a pipeline
  - API for each function / table exposed through Netlink sockets

- Utilities – provide abstractions for users to configure various aspects of the Linux data plane
  - Uses the APIs provided by the data plane through Netlink sockets

University of Colorado **Boulder**

Lab Environment

Course: Networking Principles in Practice – Linux Networking
Module: Linux Networking Intro

University of Colorado Boulder

Goal

- Create a topology of (linux) network devices and end hosts
- Configure the network devices
- Inject traffic
- Troubleshoot

Vagrant Overview

- Vagrant enables the creation and configuration of lightweight, reproducible, and portable development environments.

- We will provide the needed Vagrantfile
  - Specifies desired Linux distribution
  - Includes configuration scripts to install all needed software
- https://docs.vagrantup.com
- Assumes VirtualBox, but can use other virtualization providers (with modification to Vagrantfile)

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  # Every Vagrant development environment requires a box. You can search for
  # boxes at https://vagrantcloud.com/search.
  config.vm.box = "ubuntu/jammy64"

  config.vm.provider "virtualbox" do |vb|
  #   # Display the VirtualBox GUI when booting the machine
  #   vb.gui = true
     vb.cpus = 1
     vb.memory = "1024"
  end

  config.vm.provision "shell", inline: <<-SHELL
     apt update
     apt install net-tools
     curl -fsSL https://get.docker.com -o get-docker.sh
     sh get-docker.sh --version 24.0.5
     usermod -aG docker vagrant
     newgrp docker
     bash -c "$(curl -sL https://get.containerlab.dev)" -- -v 0.44.0
  SHELL
end
```

Specifies Linux Distribution

Allocate 1 CPU and 1GB RAM to this VM

Install net-tools, docker, containerlab

Other options available (e.g., set up networking, etc.)

Run “vagrant up” in directory with Vagrantfile

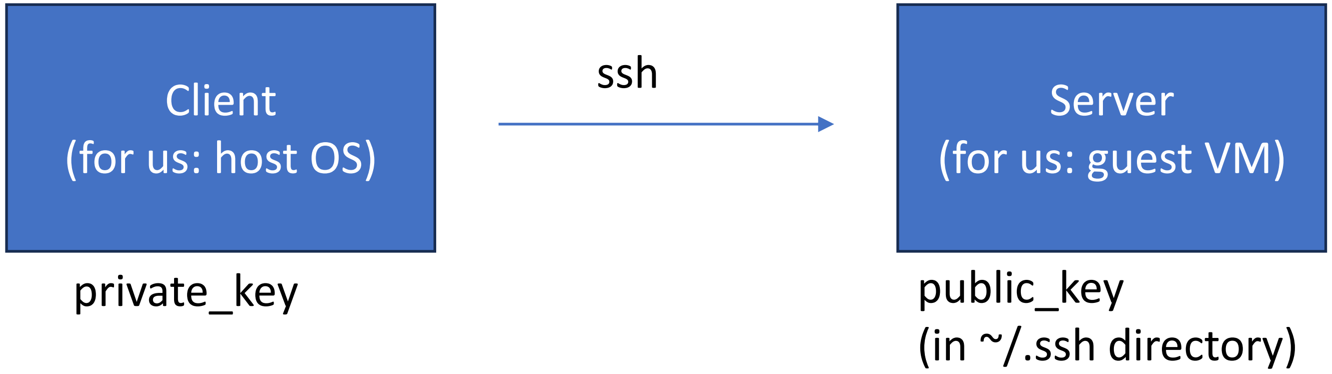
Command Prompt - vagrant

```
c:\vagrant\npp-linux-lab1>dir
 Volume in drive C is OSDisk
 Volume Serial Number is A684-6EE8

 Directory of c:\vagrant\npp-linux-lab1

09/19/2023  09:45 AM    <DIR>          .
09/19/2023  09:44 AM    <DIR>          ..
08/19/2023  11:43 AM    <DIR>          .vagrant
09/19/2023  09:21 AM             3,989 Vagrantfile
               1 File(s)          3,989 bytes
               3 Dir(s)  803,492,823,040 bytes free

c:\vagrant\npp-linux-lab1>vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Checking if box 'ubuntu/jammy64' version '20230819.0.0' is up to date...
==> default: A newer version of the box 'ubuntu/jammy64' for provider 'virtualbox' is
==> default: available! You currently have version '20230819.0.0'. The latest is version
==> default: '20230914.0.0'. Run `vagrant box update` to update.
```

Shutting down the VM

- vagrant suspend – saves the state, then turns off VM. Next time, can run "vagrant resume" to start the VM and continue from same state
- vagrant halt – shuts down the VM
- vagrant status – checks the status of all VMs

ssh overview



"The **Secure Shell Protocol** (**SSH**) is a cryptographic network protocol for operating network services securely over an unsecured network. Its most notable applications are remote login and command-line execution." <Wikipedia>

X11 forwarding enables running remote applications with graphical user interfaces and interact with them using the on local display and I/O devices. We'll mostly use command line (remote shell)

Run “vagrant ssh-config” for ssh config

Command Prompt

```
c:\vagrant\npp-linux-lab1>vagrant ssh-config
Host default
  HostName 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile c:/vagrant/npp-linux-lab1/.vagrant/machines/default/virtualbox/private_key
  IdentitiesOnly yes
  LogLevel FATAL
  PubkeyAcceptedKeyTypes +ssh-rsa
  HostKeyAlgorithms +ssh-rsa


c:\vagrant\npp-linux-lab1>
```

Hostname and port number

ssh private key generated

Configure ssh client

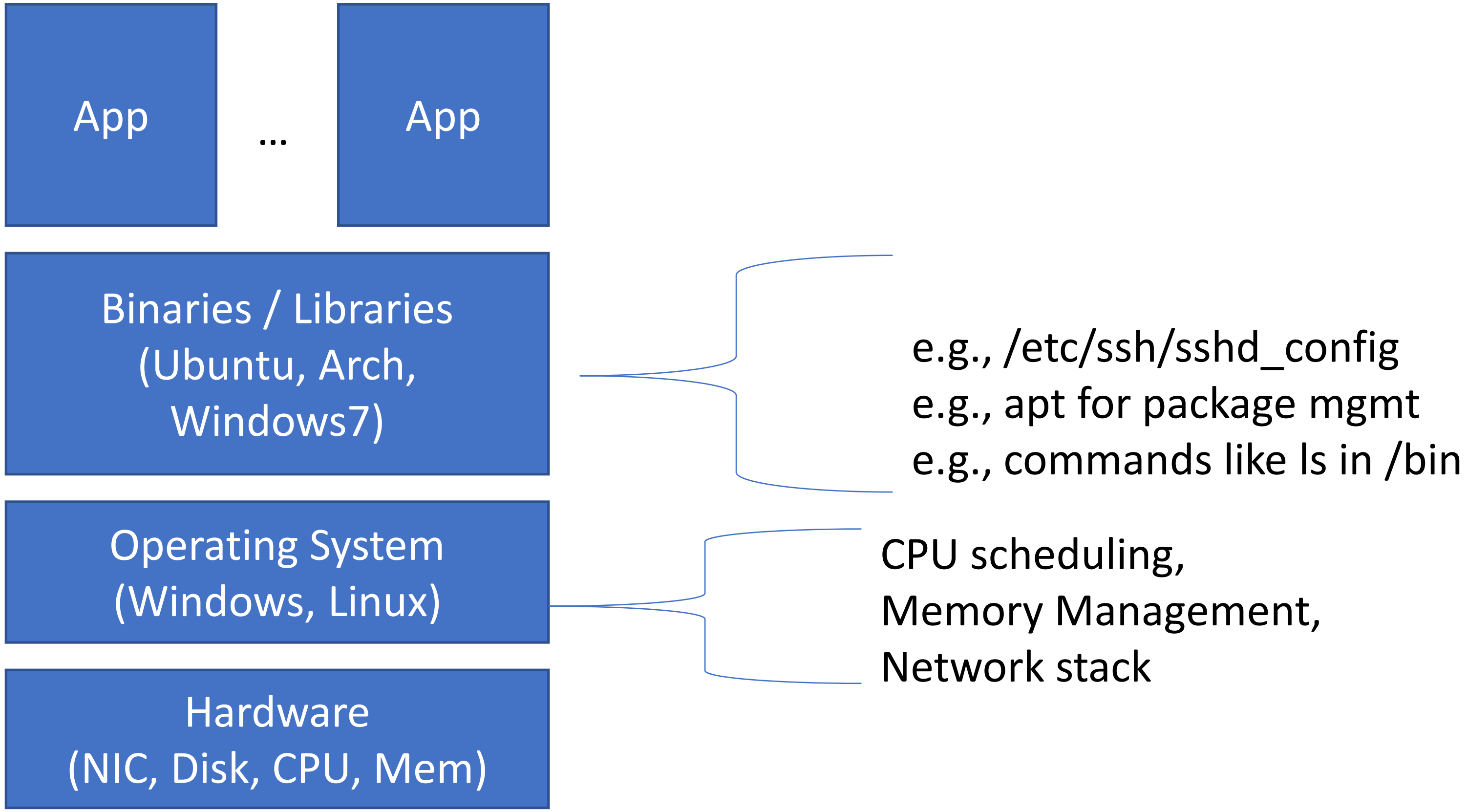
- I’m using Mobaxterm https://mobaxterm.mobatek.net/
- Configure with:
  - Hostname/IP address: 127.0.0.1
  - Port number: 2222
  - Username: vagrant
  - Private Key: <path/to/private_key>
  - Note: you’ll want x11 forwarding on

(switch to live demo)

Aside – Containers (and Docker)

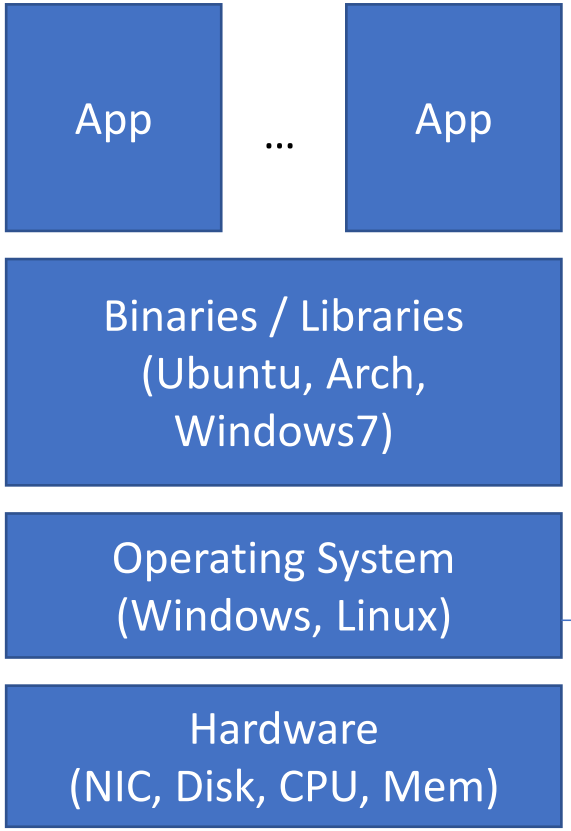
Bare Metal

Run applications directly on servers



Bare Metal

Run applications directly on servers



e.g., /etc/ssh/sshd_config
e.g., apt for package mgmt
e.g., commands like ls in /bin

CPU scheduling,
Memory Management,
Network stack

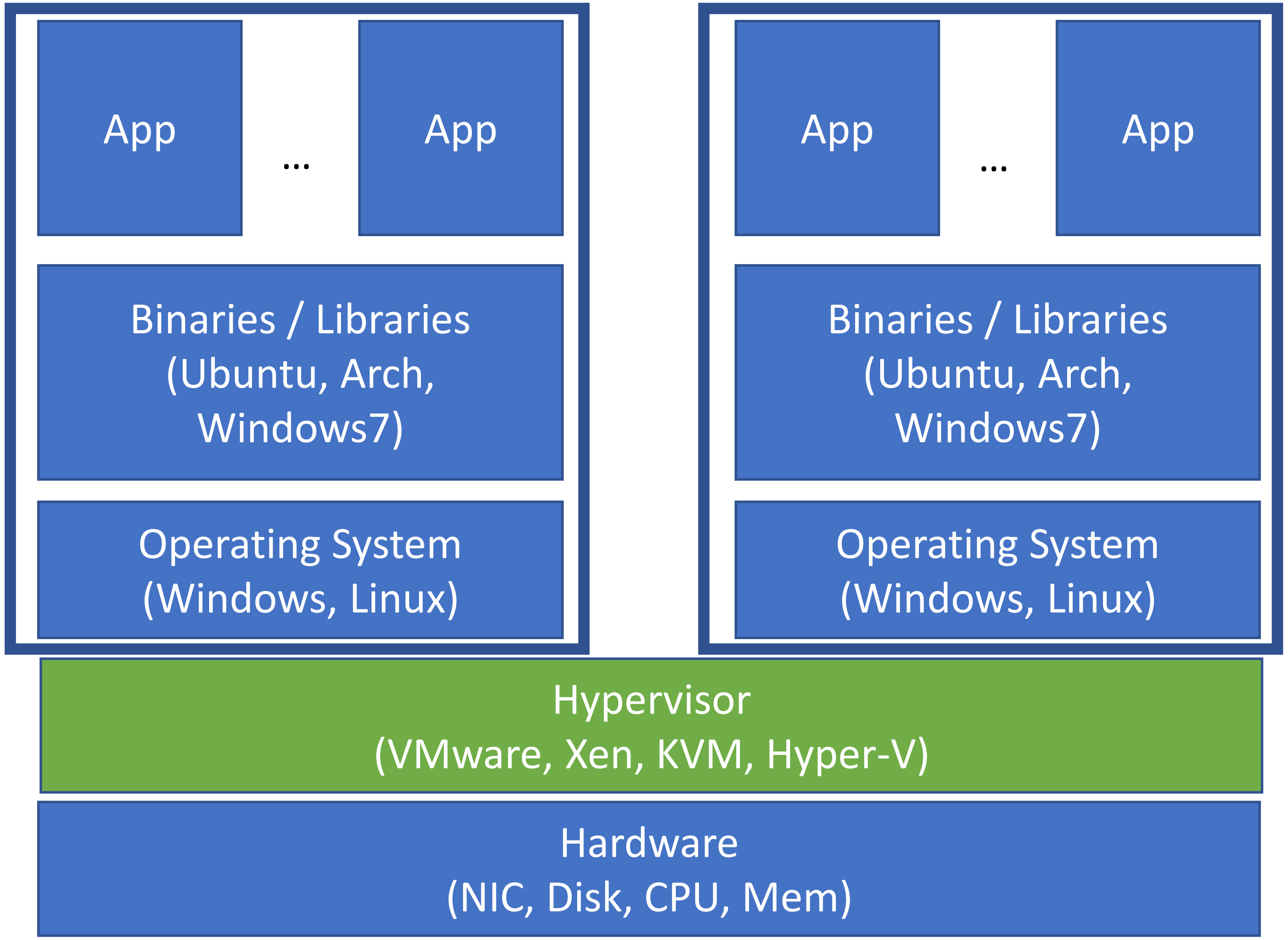
Challenges:

- Resource wastage (using only part of a server), so want to run multiple applications on the same server

- Applications might need different Operating Systems
- Applications might have variable workloads
- Applications might have conflicting dependencies (e.g., App1 needs python3, App2 needs python2) - sometimes complex to resolve

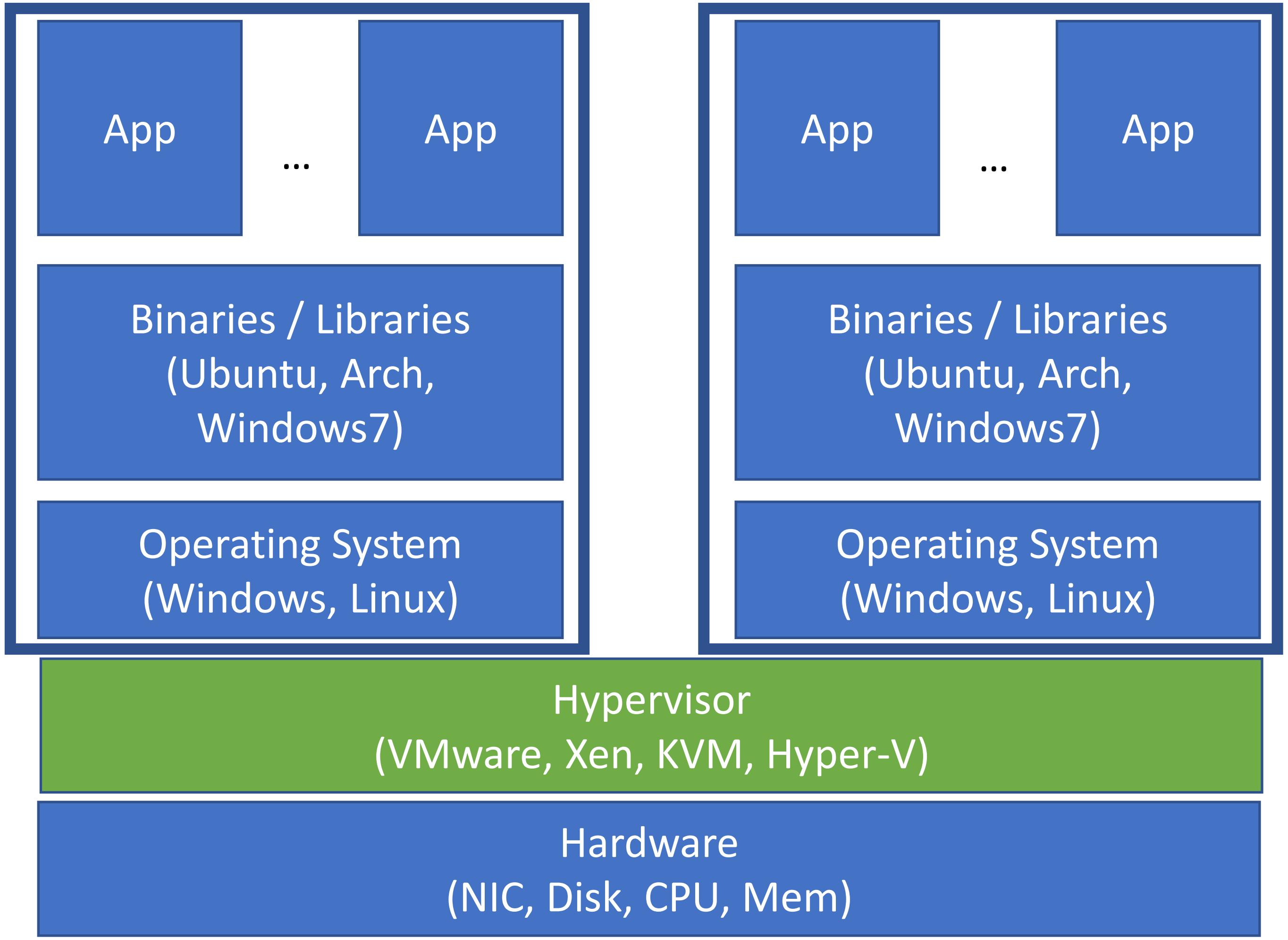
Bare Metal -> Virtual Machines

Run a software layer (hypervisor) that makes it possible to run multiple, independent OSes



Bare Metal -> Virtual Machines

Run a software layer (hypervisor) that makes it possible to run multiple, independent OSes



Challenges with Bare Metal:

- Resource wastage (using only part of a server), so want to run multiple applications on the same server

Virtualization SOLVED:

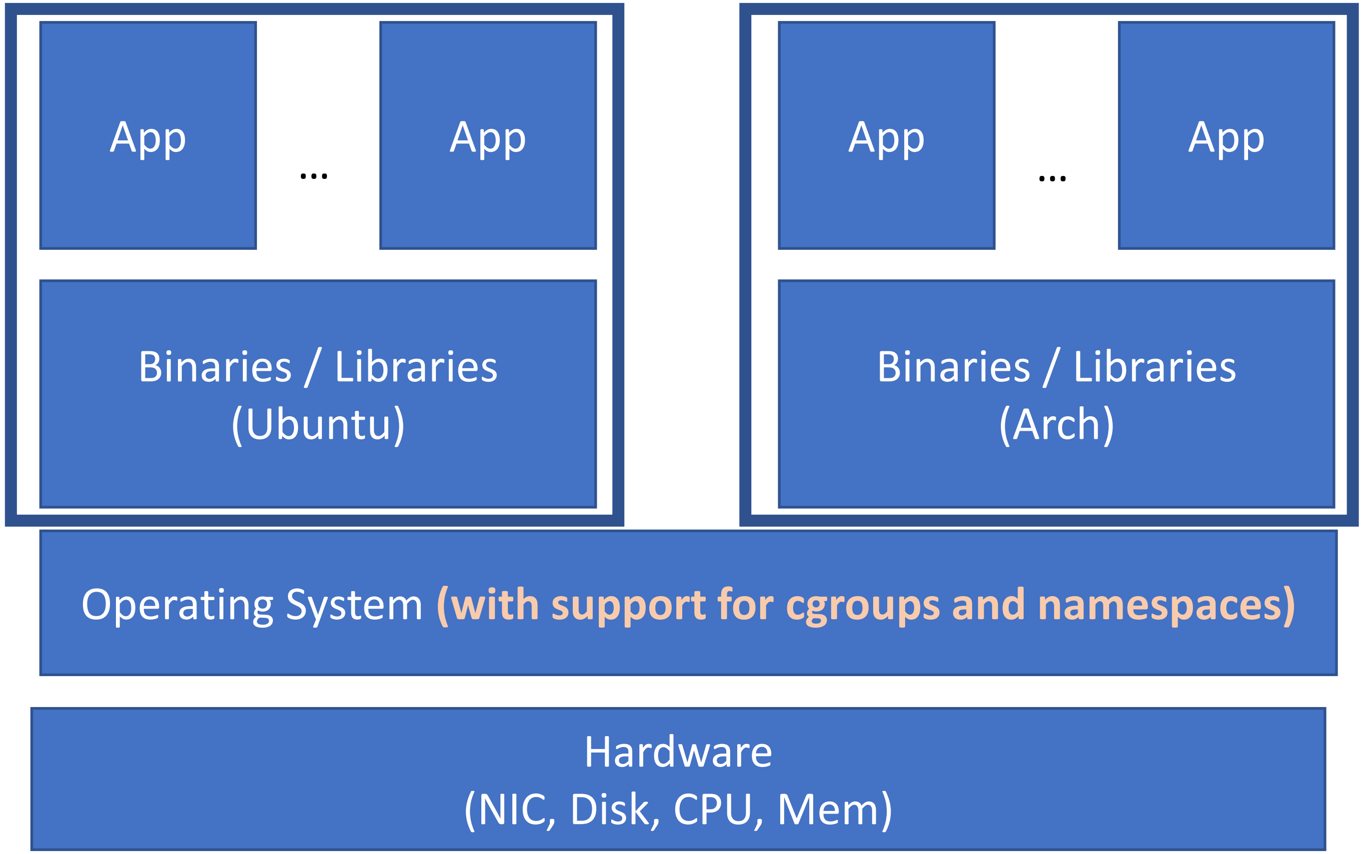
- Applications might need different Operating Systems
- Applications might have variable workloads
- Applications might have conflicting dependencies (e.g., App1 needs python3, App2 needs python2) - sometimes complex to resolve

BUT:

- Heavyweight - CPU, Memory, Disk for extra OS and full set of binaries
- Overhead - OS traditionally assume they're directly on hardware. (CPU arch has improved)

Bare Metal -> Virtual Machines -> Containers

What if we don't need different OSes? Soln: Introduce isolation mechanisms into OS



chroot - 1982

Free BSD Jails - 2000
Virtuozzo OpenVZ - 2000
Linux Vserver - 2001

LXC - 2008 (required no patches to kernel)

namespace - what resources and naming of those resources a process sees (file descriptors, IP addresses)

cgroup - (control group) groups processes and allocates resources (CPU, Memory) that the kernel enforces

What? I thought Docker invented Containers

So, what did they bring?

What? I thought Docker invented Containers

So, what did they bring?

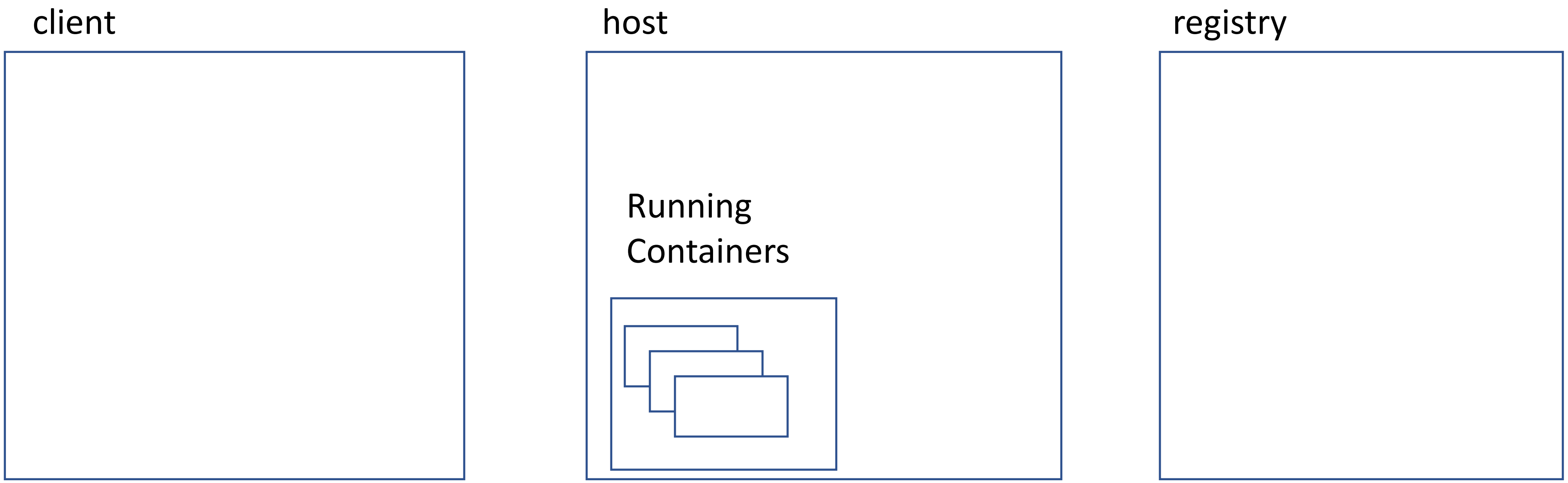
⇒Nice packaging around container technology

They made it **usable**

Overview of Docker Architecture

Note: these could all be the same machine, but we'll consider them separate for now

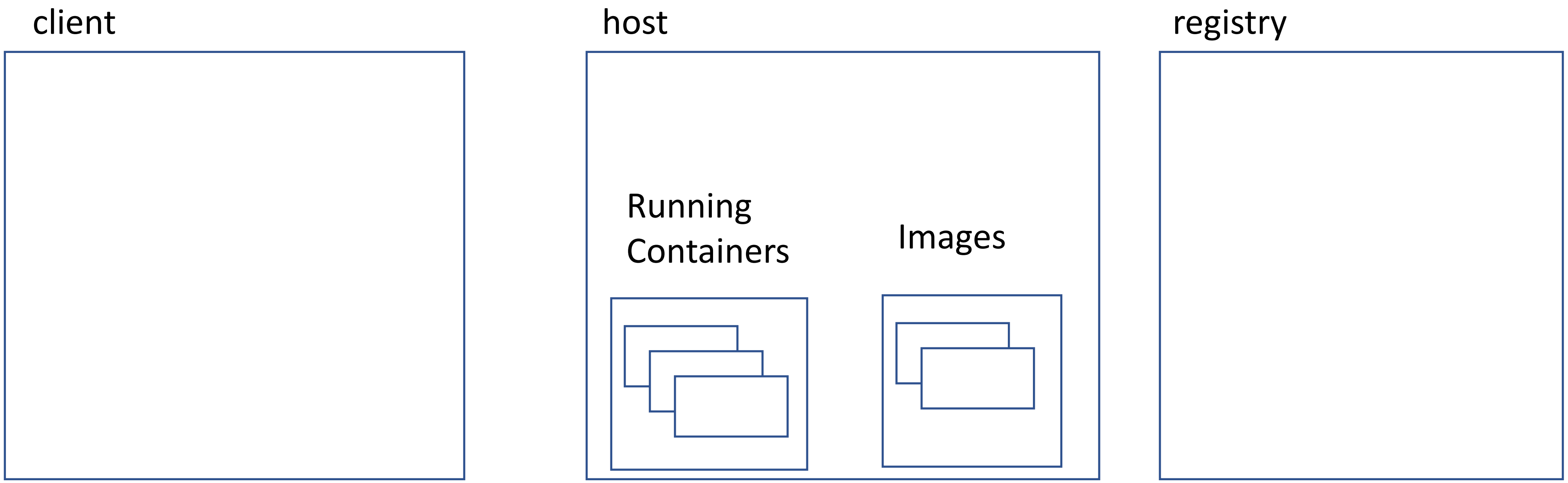
Docker Architecture - Containers



A container is a process that runs in it's own namespace (own file directory structure, own network resources, own file descriptors, etc.)
(can be in the running, stopped, or paused state)

Note: these could all be the same machine, but we'll consider them separate for now

Docker Architecture - Images

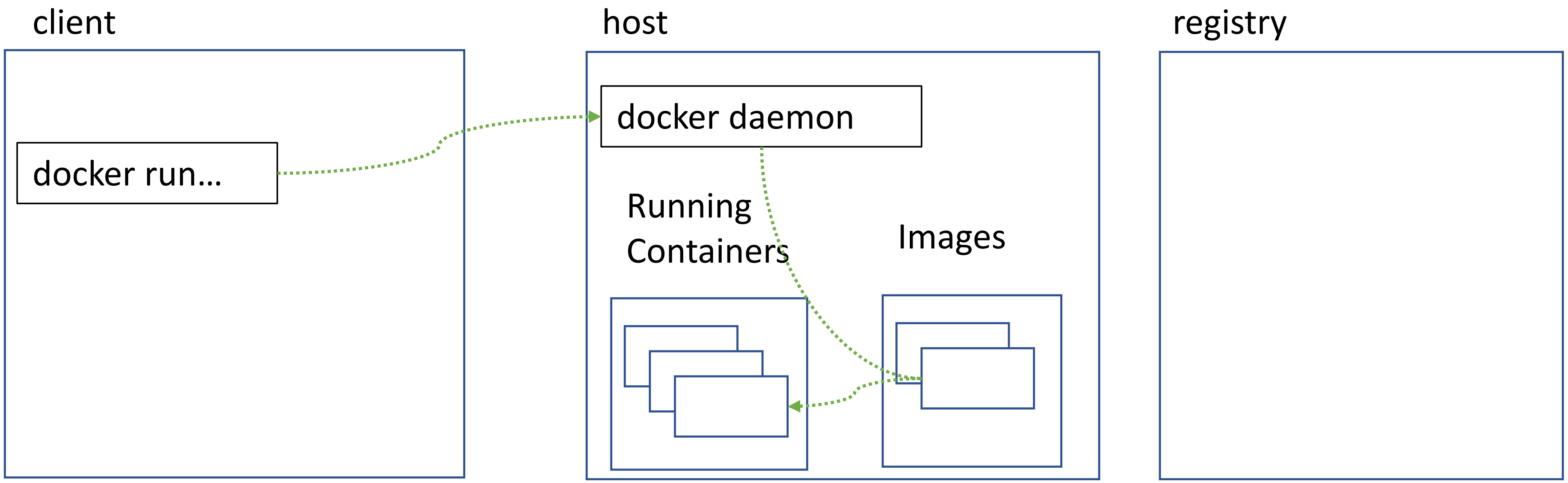


An image is an immutable (doesn't change) snapshot of a filesystem - i.e., a collection of files/directories such as /etc/ssh/sshd_config, /bin/ls

It is used as the initial collection of files within a container's namespace. Note: it'll be copy-on-write.

Note: these could all be the same machine, but we'll consider them separate for now

Docker Architecture - docker / docker daemon

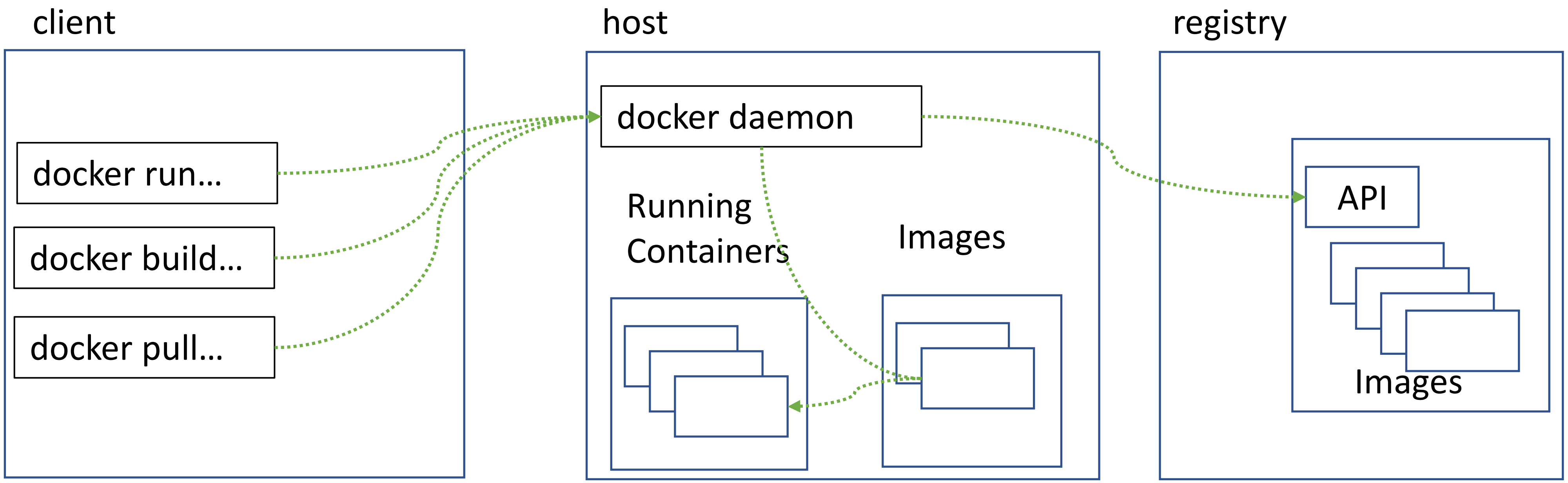


docker is an executable that interacts with the docker daemon to initiate docker commands.

The docker daemon is a process that runs on each machine you want to run containers on.
It manages the images and starting processes and interfacing to the cgroups and namespaces.
(see ‘sudo service status-all’ - you’ll see docker in there)

Note: these could all be the same machine, but we’ll consider them separate for now

Docker Architecture - registry



The docker daemon can be used to build images locally (snapshot running containers, or using a docker build system)

The registry is a storage for images, and enables an interface to pull/push images to/from a local host.

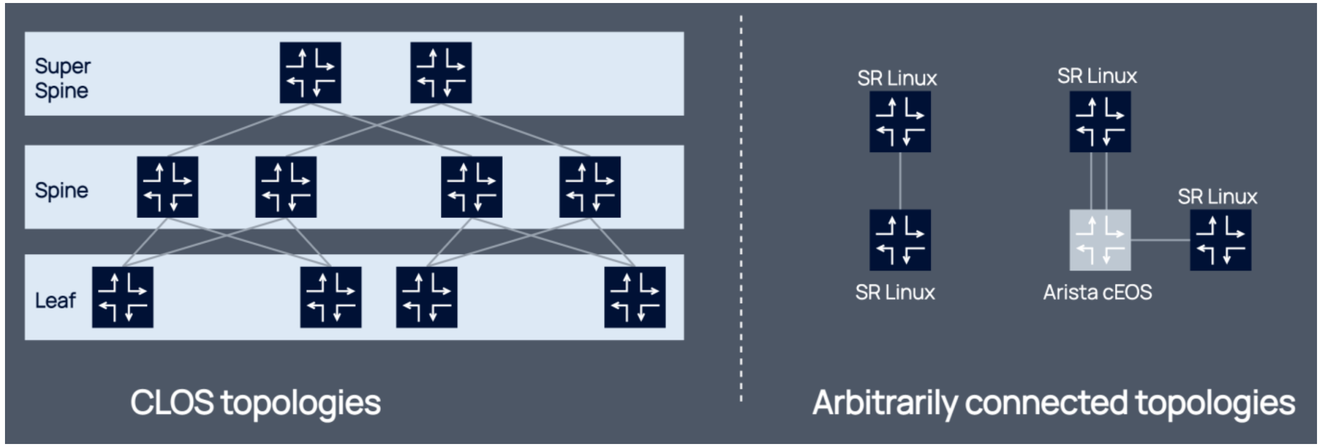
Note: these could all be the same machine, but we'll consider them separate for now

Container summary

- A temporary file system
  - layered over an image
  - fully writable (copy on write)
  - disappears when End of Life
- A Network Stack
- A Process Group - one main process, with possible subprocesses (which exits when main process exits)

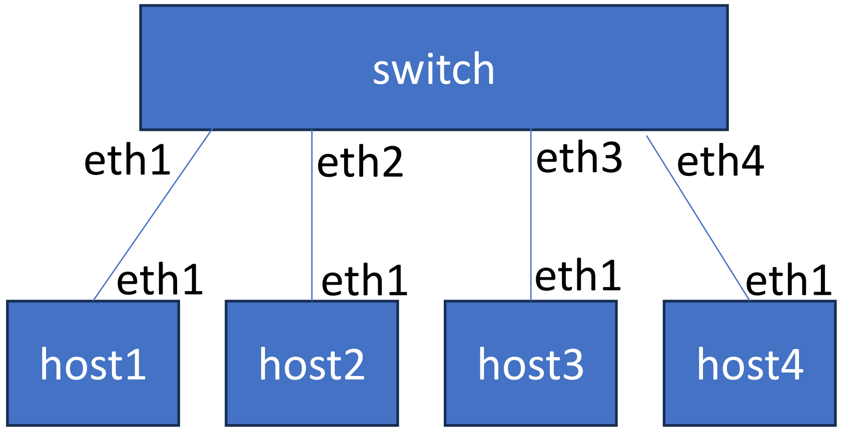
Containerlab

- https://containerlab.dev/
- Containerlab provides a CLI for orchestrating and managing container-based networking labs.
- It starts the containers, builds the virtual wiring between them to create lab topologies of user's choice and manages labs lifecycle.



Demo Overview

- Containerlab Configuration file
- “sudo containerlab deploy” command
- “docker exec” to run ifconfig in host1
  - To run a single command
  - To start a shell
- Aliases
- sudo containerlab destroy (to tear down)



University of Colorado **Boulder**

Screen captures from demo

vi 4node-part1.clab.yml

```yaml
name: lab1-part1

topology:
  nodes:
    host1:
      kind: linux
      image: ekellercu/network-testing:v0.1
      binds:
        - lab-host1:/lab-folder
    host2:
      kind: linux
      image: ekellercu/network-testing:v0.1
      binds:
        - lab-host2:/lab-folder
    host3:
      kind: linux
      image: ekellercu/network-testing:v0.1
      binds:
        - lab-host3:/lab-folder
    host4:
      kind: linux
      image: ekellercu/network-testing:v0.1
      binds:
        - lab-host4:/lab-folder
    switch:
      kind: linux
      image: ekellercu/network-testing:v0.1
      binds:
        - lab-switch:/lab-folder

  links:
    - endpoints: ["host1:eth1", "switch:eth1"]
    - endpoints: ["host2:eth1", "switch:eth2"]
    - endpoints: ["host3:eth1", "switch:eth3"]
    - endpoints: ["host4:eth1", "switch:eth4"]
```

sudo containerlab deploy

2. node1

```
vagrant@ubuntu-jammy:~/lab1/part1$ sudo containerlab deploy
INFO[0000] Containerlab v0.44.0 started
INFO[0000] Parsing & checking topology file: 4node-part1.clab.yml
INFO[0000] Creating docker network: Name="clab", IPv4Subnet="172.20.20.0/24", IPv6Subnet="2001:172:20:20::/64", MTU="1500"
INFO[0001] Creating lab directory: /home/vagrant/lab1/part1/clab-lab1-part1
WARN[0001] SSH_AUTH_SOCK not set, skipping pubkey fetching
INFO[0001] Creating container: "host1"
INFO[0003] Creating container: "host2"
INFO[0003] Creating container: "host3"
INFO[0004] Creating container: "host4"
INFO[0005] Creating container: "switch"
INFO[0006] Creating link: host1:eth1 <--> switch:eth1
INFO[0006] Creating link: host2:eth1 <--> switch:eth2
INFO[0006] Creating link: host3:eth1 <--> switch:eth3
INFO[0006] Creating link: host4:eth1 <--> switch:eth4
INFO[0006] Adding containerlab host entries to /etc/hosts file
INFO[0006] 🎉 New containerlab version 0.44.3 is available! Release notes: https://containerlab.dev/rn/0.44/#0443
Run 'containerlab version upgrade' to upgrade or go check other installation options at https://containerlab.dev/install/
+---+-----------------------+--------------+-------------------------------+-------+---------+----------------+----------------------+
| # |         Name          | Container ID |             Image             | Kind  |  State  |  IPv4 Address  |     IPv6 Address     |
+---+-----------------------+--------------+-------------------------------+-------+---------+----------------+----------------------+
| 1 | clab-lab1-part1-host1  | af988707f2d9 | ekellercu/network-testing:v0.1 | linux | running | 172.20.20.2/24 | 2001:172:20:20::2/64 |
| 2 | clab-lab1-part1-host2  | 71d4eed2aced | ekellercu/network-testing:v0.1 | linux | running | 172.20.20.3/24 | 2001:172:20:20::3/64 |
| 3 | clab-lab1-part1-host3  | a71868d56c56 | ekellercu/network-testing:v0.1 | linux | running | 172.20.20.4/24 | 2001:172:20:20::4/64 |
| 4 | clab-lab1-part1-host4  | c59a28ae55b9 | ekellercu/network-testing:v0.1 | linux | running | 172.20.20.5/24 | 2001:172:20:20::5/64 |
| 5 | clab-lab1-part1-switch | e49bd112df64 | ekellercu/network-testing:v0.1 | linux | running | 172.20.20.6/24 | 2001:172:20:20::6/64 |
+---+-----------------------+--------------+-------------------------------+-------+---------+----------------+----------------------+
```

docker exec -it clab-lab1-part1-host1 ifconfig

Run command from outside container to execute command inside container

2. node1

```
vagrant@ubuntu-jammy:~/lab1/part1$ docker exec -it clab-lab1-part1-host1 ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.20.20.2  netmask 255.255.255.0  broadcast 172.20.20.255
        inet6 2001:172:20:20::2  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::42:acff:fe14:1402  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:14:14:02  txqueuelen 0  (Ethernet)
        RX packets 55  bytes 5258 (5.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 10  bytes 876 (876.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::a8c1:abff:fe87:98ed  prefixlen 64  scopeid 0x20<link>
        ether aa:c1:ab:87:98:ed  txqueuelen 0  (Ethernet)
        RX packets 10  bytes 816 (816.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9  bytes 726 (726.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

docker exec -it clab-lab1-part1-host1 bash

Outside container

Inside container

Back outside container

2. node1

```
vagrant@ubuntu-jammy:~/lab1/part1$ docker exec -it clab-lab1-part1-host1 bash
root@host1:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.20.20.2  netmask 255.255.255.0  broadcast 172.20.20.255
        inet6 2001:172:20:20::2  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::42:acff:fe14:1402  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:14:14:02  txqueuelen 0  (Ethernet)
        RX packets 64  bytes 5888 (5.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 1016 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::a8c1:abff:fe87:98ed  prefixlen 64  scopeid 0x20<link>
        ether aa:c1:ab:87:98:ed  txqueuelen 0  (Ethernet)
        RX packets 13  bytes 1026 (1.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 936 (936.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@host1:/# exit
exit
vagrant@ubuntu-jammy:~/lab1/part1$
```

alias host1 = “docker exec -it clab-lab1-part1-host1”
host1 ifconfig

2. node1

```
vagrant@ubuntu-jammy:~/lab1/part1/provided$ alias host1="docker exec -it clab-lab1-part1-host1"
vagrant@ubuntu-jammy:~/lab1/part1/provided$ host1 ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.20.20.2  netmask 255.255.255.0  broadcast 172.20.20.255
        inet6 2001:172:20:20::2  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::42:acff:fe14:1402  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:14:14:02  txqueuelen 0  (Ethernet)
        RX packets 68  bytes 6168 (6.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 1086 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::a8c1:abff:fe87:98ed  prefixlen 64  scopeid 0x20<link>
        ether aa:c1:ab:87:98:ed  txqueuelen 0  (Ethernet)
        RX packets 14  bytes 1096 (1.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 1006 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

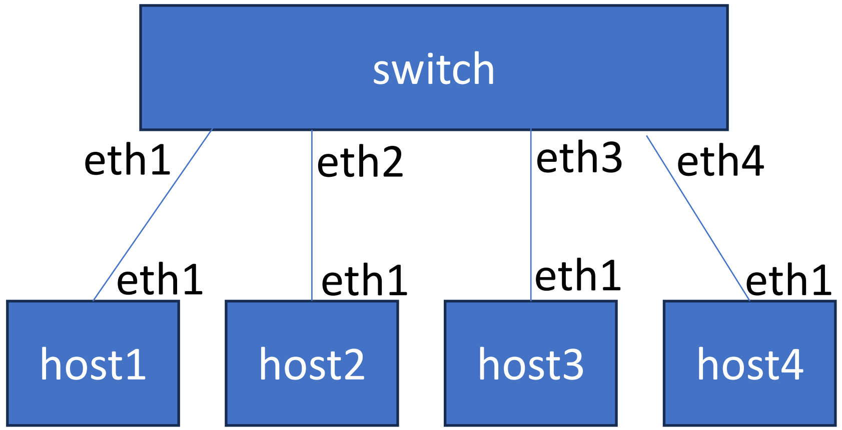
Troubleshooting Tools

Course: Networking Principles in Practice – Linux Networking
Module: Linux Networking Intro

University of Colorado Boulder

Demo Overview

- Containerlab Configuration file
- "sudo containerlab deploy" command
- "docker exec" to run ifconfig in host1
  - To run a single command
  - To start a shell
- Aliases
- sudo containerlab destroy (to tear down)



ping

- Quick check if a server is up
- Also includes round trip time, so can identify latency issues

2. node1

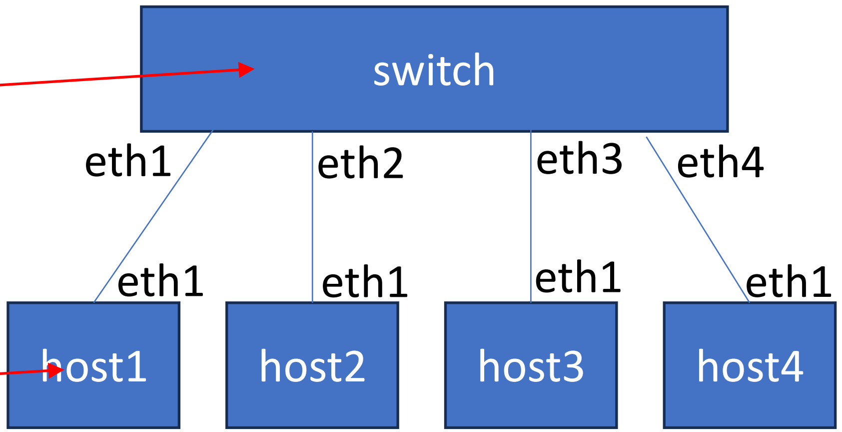
```
vagrant@ubuntu-jammy:~/lab1/part1/provided$ docker exec -it clab-lab1-part1-host1 ip addr add 192.168.1.2/24 dev eth1
vagrant@ubuntu-jammy:~/lab1/part1/provided$ docker exec -it clab-lab1-part1-switch ip addr add 192.168.1.1/24 dev eth1
vagrant@ubuntu-jammy:~/lab1/part1/provided$ docker exec -it clab-lab1-part1-host1 ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.127 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.077 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.066 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=0.041 ms
^C
--- 192.168.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3056ms
rtt min/avg/max/mdev = 0.041/0.077/0.127/0.031 ms
vagrant@ubuntu-jammy:~/lab1/part1/provided$
```

tshark

Network protocol analyzer.

Capture packets and then display them.

tshark -i eth1



ping 192.168.1.1

docker exec -it clab-lab1-part1-switch tshark -i eth1

docker exec -it clab-lab1-part1-host1 ping 192.168.1.1

Some common options

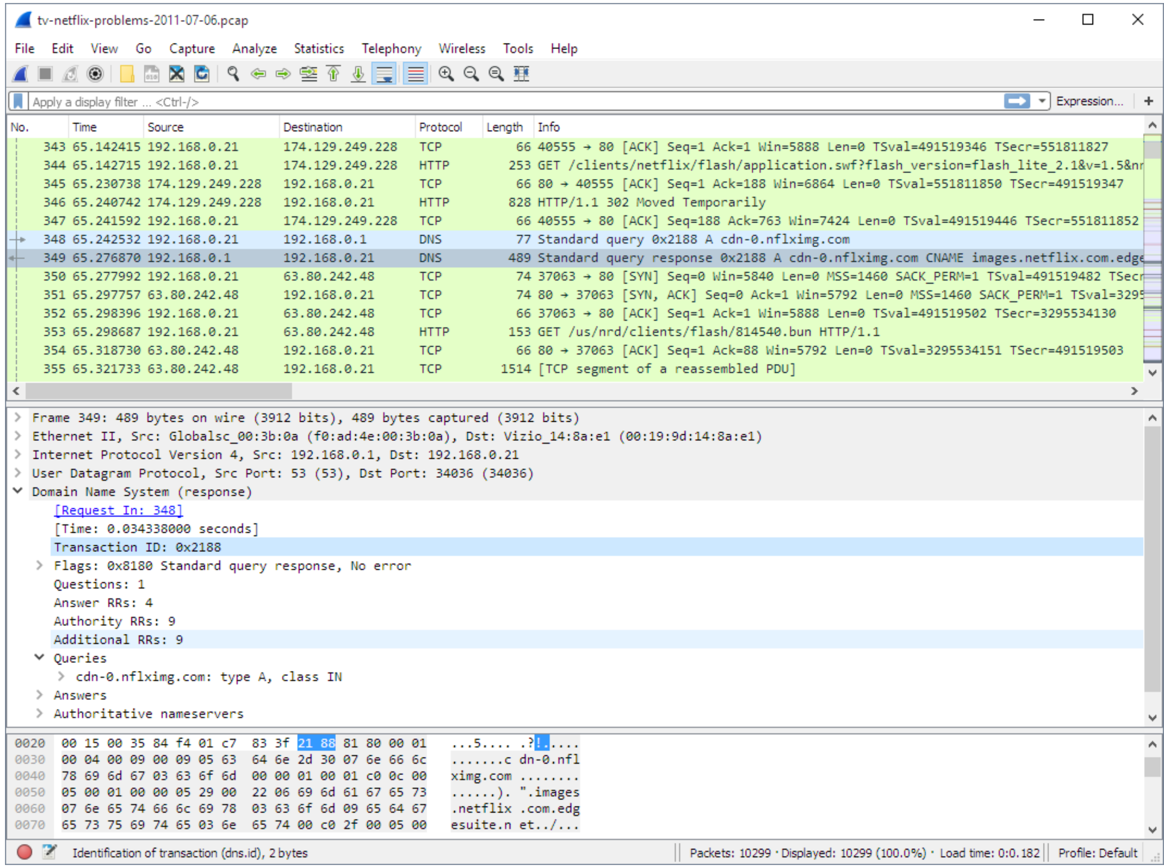
- -i <interface>. Which interface to capture on (e.g., tshark -i eth1). Defaults to eth0.
- -f <capture filter>. Capture only packets that match condition.
  - tshark -i eth1 -f “arp”
  - tshark -i eth1 -f “port 80”
  - tshark -i eth1 -f “host 1.2.3.4 and port 80”
- -Y <display filter>. Display only packets that match condition.
  - tshark -Y "tcp.port == 80“
  - tshark -Y -i eth1 "ip.src == 192.168.246.198“
  - tshark -Y -i eth1 "ip.ttl > 10“
  - tshark -Y -i eth1 "ip.src == 192.168.246.198 and ip.dest != 1.2.3.4“

Packet Capture (pcap) files

- With tshark can
  - read with –r <file> and
  - write –o <file>

Wireshark

- Graphical packet capture tool.
- May need extra setup to get to run inside container, or
- Run outside of container on veth device (we'll learn about later), or
- Run tshark to capture packet, then use wireshark to display (recall we used "bind" to mount a directory)



Scapy

Python-based interactive packet manipulation program and library

- Create/modify packets with a simple API (supports many protocols)
- Read/Write pcap files
- Capture/Send traffic on the wire

https://scapy.net/

Crafting a protocol header

- Approach 1: As arguments to class init

```
Ether(dst=“11:22:33:44:55:66”, src=“66:55:44:33:22:11”, type=0x0800)
```

- Approach 2: As fields

```
eth = Ether()
eth.dst=“11:22:33:44:55:66”
eth.src=“66:55:44:33:22:11”
eth.type=0x0800
```

Crafting a packet from protocol headers

Chain together with the “/” operator

pkt = Ether() / IP() / TCP()

Access individual headers:

eth = pkt[Ether]

Modifying individual fields:

pkt[IP].src = “1.2.3.4”

Helpful functions

ls()

ARP : ARP

DNS : DNS

Dot11 : 802.11

TCP : TCP

Ether : Ethernet

[...]

ls(IP)

version : BitField = (4)

ihl : BitField = (None)

tos : XByteField = (0)

…

src : Emph = (None)

dst : Emph = ('127.0.0.1’)

…

Read / Write pcap

```
from scapy.all import *
packets = rdpcap("capture.pcap")

# you can now print, iterate over the list, or access elements
packets.summary()
packets[0][IP].src = "1.2.3.4"
for p in packets:
   if (p.haslayer(IP)):
       p[IP].src == "1.2.3.4"

wrpcap("newpcap.pcap", packets)
```

Send / Receive from interface

Capturing packets:

capture = sniff()
capture = sniff(count=3)

Sending packets:

sendp() – layer 2

send() – layer 3

Sending and receiving packets:

sr()

sr1()

University of Colorado **Boulder**

Linux network device configuration (ip link)

Course: Networking Principles in Practice – Linux Networking
Module: Linux Networking Intro

University of Colorado Boulder

Goal: Network Device Configuration

Types of devices:

- Physical interfaces (ethernet)
- Attach to another device (vlan)
- Connect together multiple devices (bridge, bond)
- Tunnel traffic (vxlan, geneve)
- Virtual devices – (veth)

iproute2 vs net-tools

iproute2 - a collection of utilities for controlling TCP / IP networking and traffic control in Linux.

net-tools - collection of base networking utilities for Linux
(ifconfig, arp, route, etc.)

➔ We'll use iproute2

ip link

Network device configuration for Linux (as part of iproute2)

Key commands:

- Show
- Set
- Add
- Delete

Man page

https://manpages.ubuntu.com/manpages/xenial/man8/ip-link.8.html

```
ip [ OPTIONS ] link  { COMMAND | help }

OPTIONS := { -V[ersion] | -h[uman-readable] | -s[tatistics] | -r[esolve] | -f[amily] {
        inet | inet6 | ipx | dnet | link } | -o[neline] | -br[ief] }

ip link add [ link DEVICE ] [ name ] NAME
        [ txqueuelen PACKETS ]
        [ address LLADDR ] [ broadcast LLADDR ]
        [ mtu MTU ] [ index IDX ]
        [ numtxqueues QUEUE_COUNT ] [ numrxqueues QUEUE_COUNT ]
        type TYPE [ ARGS ]

TYPE := [ bridge | bond | can | dummy | hsr | ifb | ipoib | macvlan | macvtap | vcan |
        veth | vlan | vxlan | ip6tnl | ipip | sit | gre | gretap | ip6gre | ip6gretap |
        vti | nlmon | ipvlan | lowpan | geneve ]

ip link delete { DEVICE | group GROUP } type TYPE [ ARGS ]
```

ip link show

`ip link show [ DEVICE | group GROUP | up | master DEVICE | type TYPE ]`

(look at man page for ip link show details)

Examples:

ip link show

Limit the output:

ip link show eth0

ip link show up

Extra options:

-d - display more details (ip link show only)

-j - output in json

-p - pretty the output

ip –d –j –p link show eth0

ip link set

(look at man page for ip link set)

Examples:

Set the MAC address:

ip link set dev eth0 address 22:33:22:44:55:44

Set the interface state to up:

ip link set dev eth0 up

ip link add

(look at man page for ip link add)

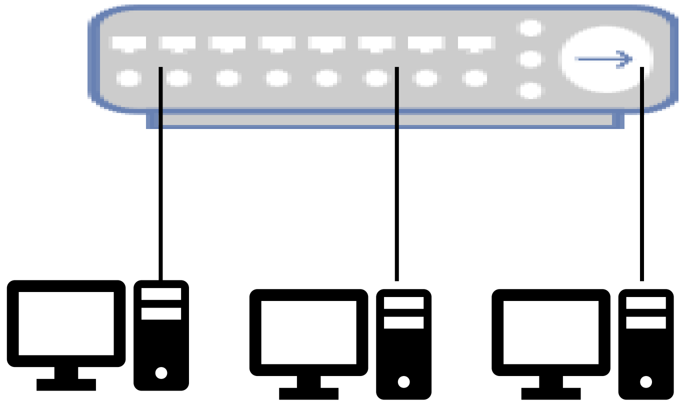
Recall - Types of devices:

- Physical interfaces (ethernet)
- Attach to another device (vlan)
- Connect together multiple devices (bridge, bond)
- Tunnel traffic (vxlan, geneve)
- Virtual devices – (veth)

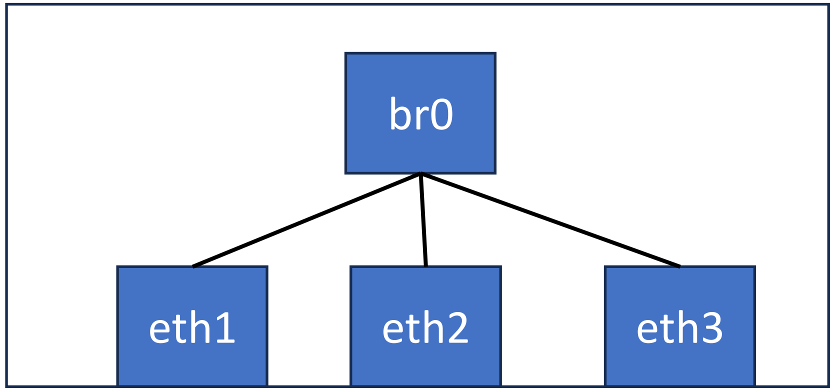
Each uses ip link add a bit differently.

Bridge

- Device that effectively implements a learning switch
- You create the bridge device
- Then make devices slaves to the bridge

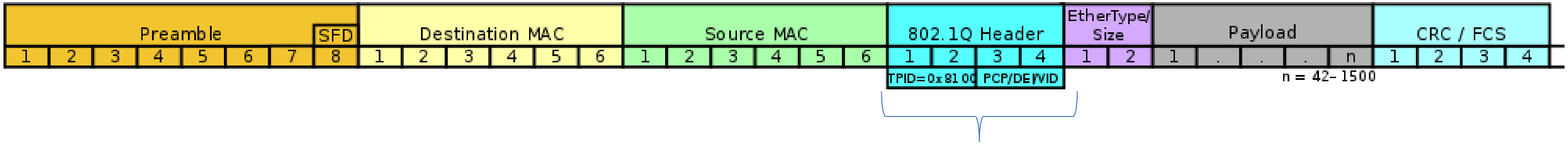


```
ip link add name mybridge type bridge
ip link set mybridge up
ip link set eth1 master mybridge
ip link set eth2 master mybridge
ip link set eth3 master mybridge
```



VLAN

- Device that adds VLAN tagging
- VLAN enables isolation in a shared L2 network (e.g., host1/2 from 3/4)



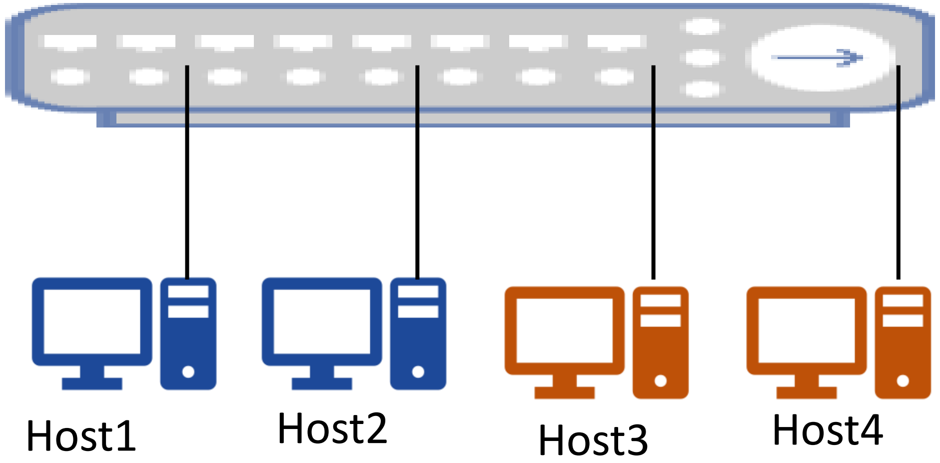
ip link add <u>link eth0</u> <u>name eth0.2</u> <u>type vlan</u> <u>id 2</u>

Specify eth device to associate it with

Name it (convention .y)

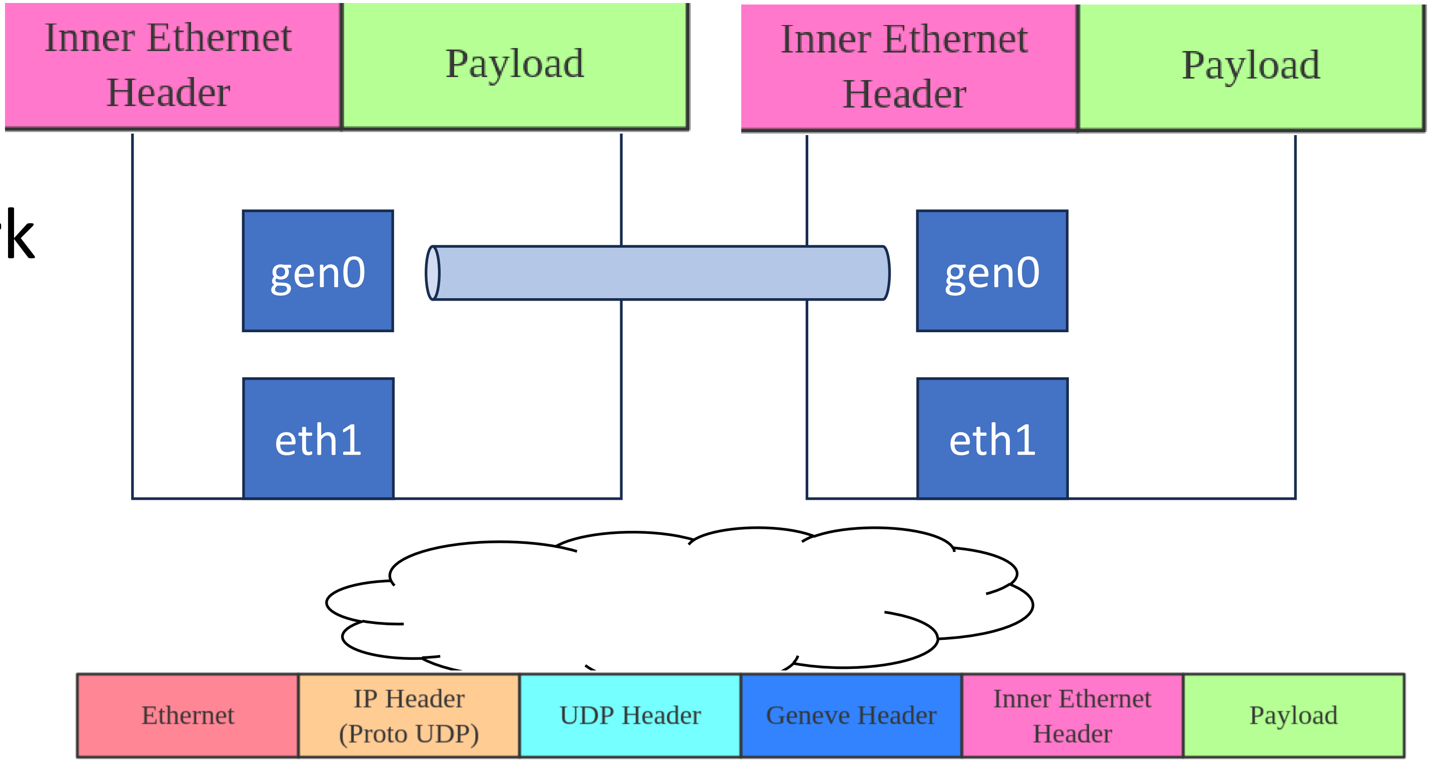
Device type adding is vlan

Main arg specifies a unique id



Tunnel (geneve, vxlan)

- Tunnels encapsulate traffic for transmission over some network



ip link add <u>name gen0</u> <u>type geneve</u> <u>id 55</u> <u>remote 1.2.3.4</u>

ip link delete

Example:

ip link delete dev eth0.2

Demo - setup

```
# show nothing running in docker
docker ps
# show container lab configuration
vi 2node-mod1.clab.yml
# create lab
sudo containerlab deploy

# some aliases to make it easier to run docker exec commands
vi make_aliases.sh
source make_aliases.sh

# go over the scapy code to craft one packet
vi onepkt.py
```

Demo – create bridge

```
# on h2 run tshark, and h1 run onepkt
h2 tshark -T fields -e eth -e vlan -e vxlan -i eth1
h1 python3 /lab-folder/onepkt.py 22:11:11:11:11:11 22:22:22:22:22:22 eth1 123
# then show pkt arriving on switch
sw tshark -T fields -e eth -e vlan -e vxlan -i eth1
h1 python3 /lab-folder/onepkt.py 22:11:11:11:11:11 22:22:22:22:22:22 eth1 123

# Create bridge on switch
sw   ip link add name mybridge type bridge
sw   ip link set mybridge up
sw   ip link set eth1 master mybridge
sw   ip link set eth2 master mybridge

# on h2 run tshark, on h1 run onepkt.py
h2 tshark -T fields -e eth -e vlan -e vxlan -i eth1
h1 python3 /lab-folder/onepkt.py 22:11:11:11:11:11 22:22:22:22:22:22 eth1 123
```

Demo – create VLAN

\# now, let's tag some traffic with VLAN id 2

h1 ip link add link eth1 name eth1.2 type vlan id 2

h1 ip link set eth1.2 up

\# on h2 run tshark, on h1 run onepkt.py

\# (note eth1.2 instead of eth1, and note VLAN 2 on tshark output)

h2 tshark -T fields -e eth -e vlan -e vxlan -i eth1

h1 python3 /lab-folder/onepkt.py 22:11:11:11:11:11 22:22:22:22:22:22 eth1.2 123

University of Colorado **Boulder**