Extended BPF and Data Plane Extensibility: An overview of networking and Linux

Fernando Sanchez
Principal SE, PLUMgrid Inc.
@fernandosanchez

Disclaimer

- I am a networking systems engineer/architect
  - I just happen to be lucky enough to work close to the guys working on/upstreaming these pieces of code into the kernel
- **This is mainly about *networking* and why/how to do it with the latest updates to the linux kernel**
- Expect no corporate/product pitch…. Almost ☺
- We can stay high-level or dive deep into APIs and code (as much as I can handle!), so feedback is appreciated

"Please do not shoot the pianist. He is doing his best."

Oscar Wilde (1854-1900)

Agenda

- Lessons from Physical Networks: Traditional Data Center Design and the effects of virtualization
- Hypervisor Networking Layer: Virtual Switches, Distributed Virtual Switches and Network Overlays
- (E)BPF and its applicability to an Extensible Networking Dataplane – From Virtual Switches to Virtual Networks
- A new Data Center Design

Lessons from Physical Networks: Traditional Data Center Design and the effects of virtualization

Server Virtualization

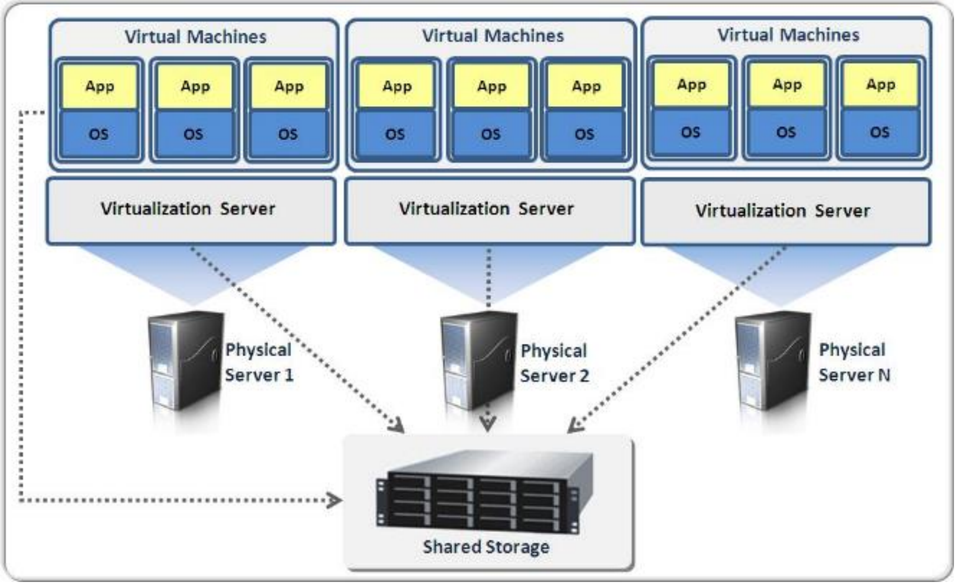
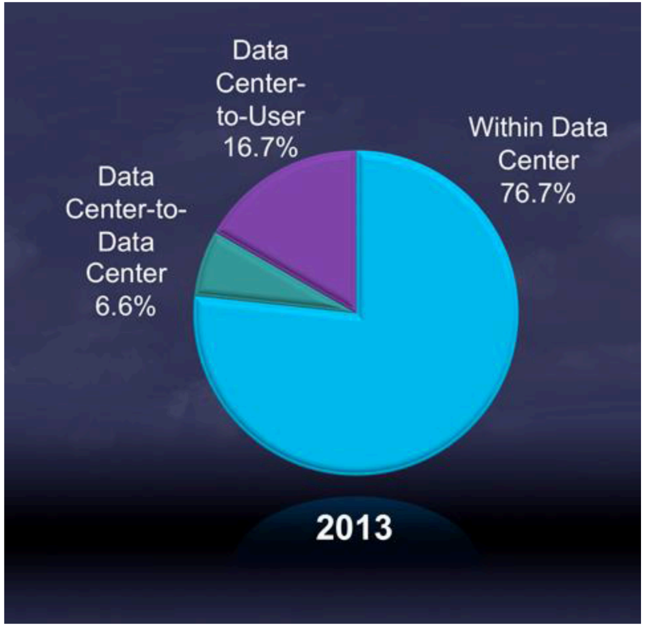


Figure 2. Global Data Center Traffic by Destination

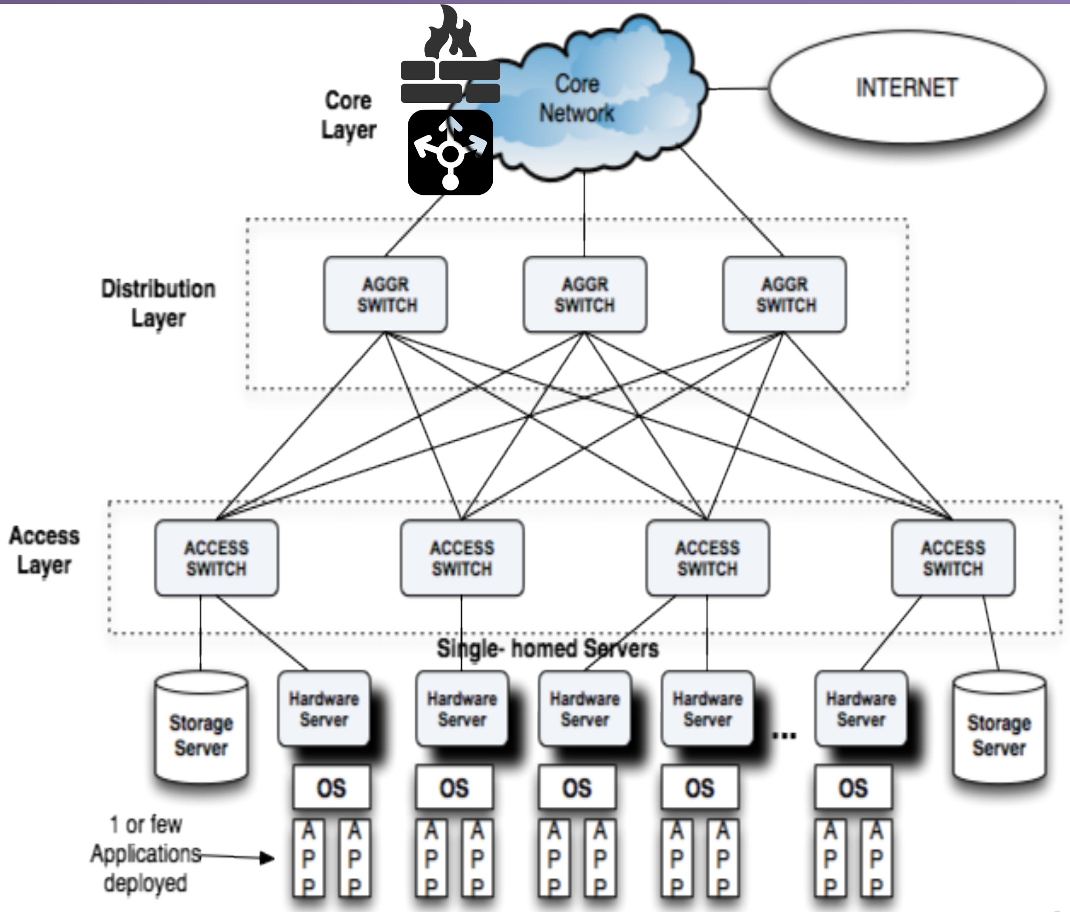


Source: Cisco Global Cloud Index, 2013–2018

How does this affect the network?

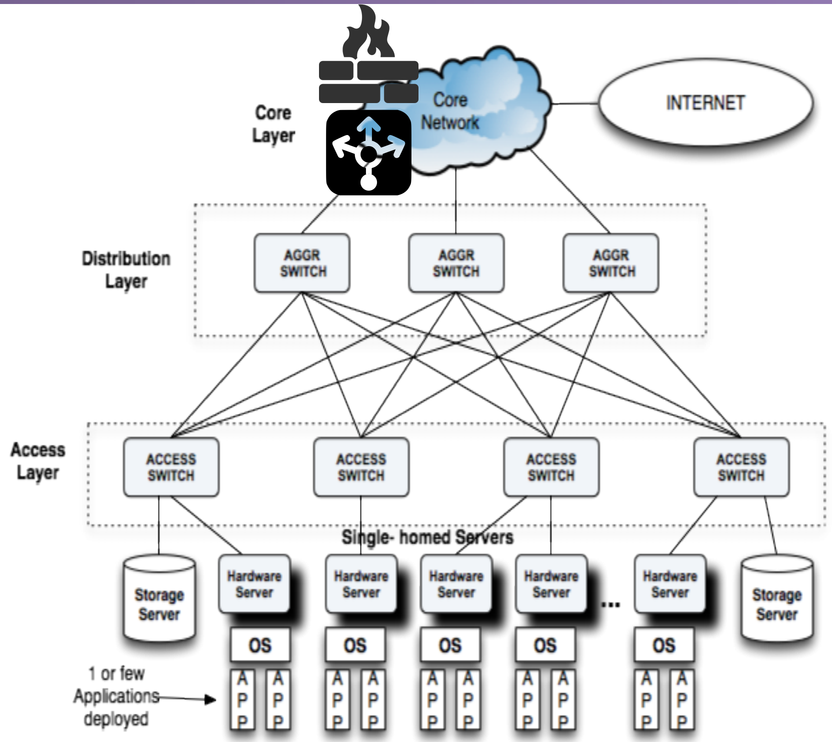
Traditional Data Center: Characteristics

- One host OS per server
- Three tier (Access, Distribution, Core) Networking Design
- Traditional L2 and L3 protocols
  - *spanning-tree issues, anyone?*
- HA based in physical server/link deployments



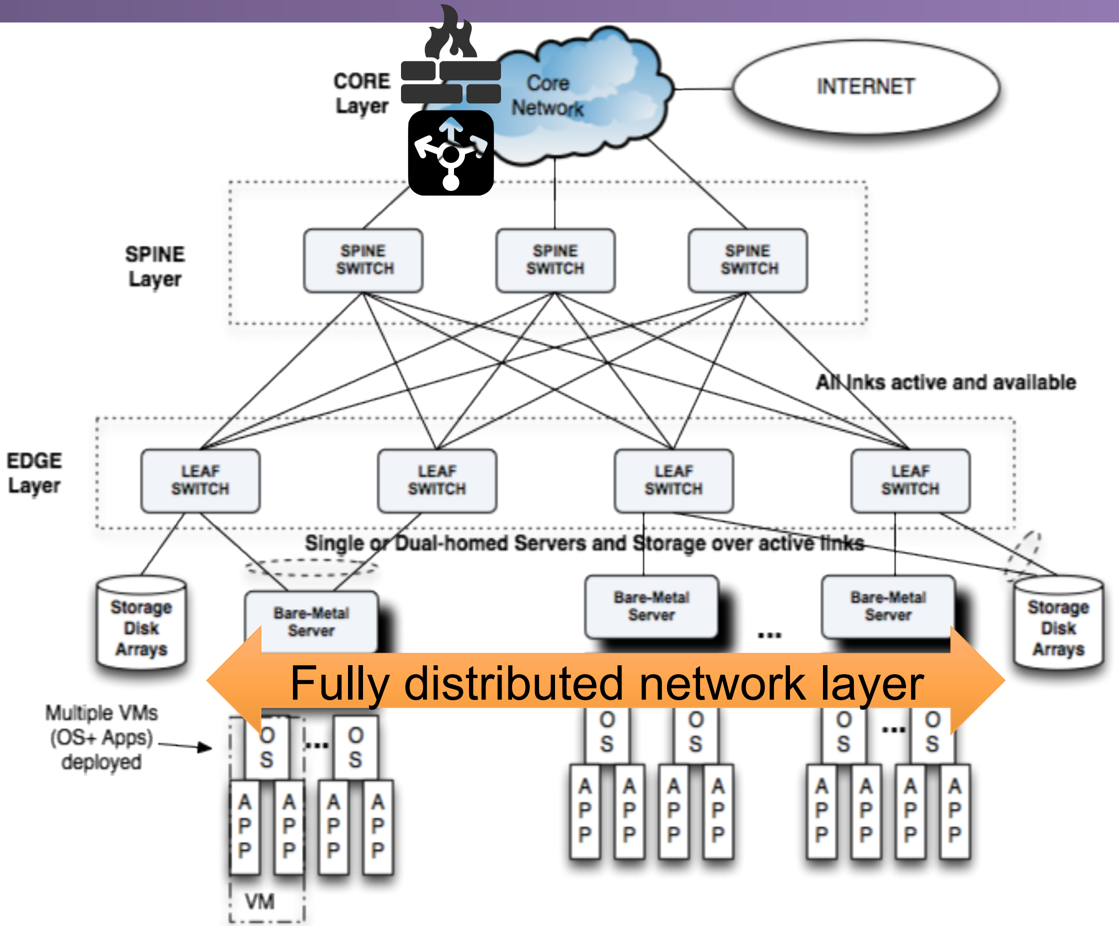
Traditional Data Center: General Issues

- Costly, Complex, and Constrained
  - Switch cross connects waste revenue generating ports
  - Scalability based on hardware and space
- Network sub-utilization
- Slow L2/3 failure recovery
- **Layer 4/7 is centralized at core layer**
- Quickly reaching HW limits (#MACs, #VLANs, etc.)



A Modern Data Center: Characteristics

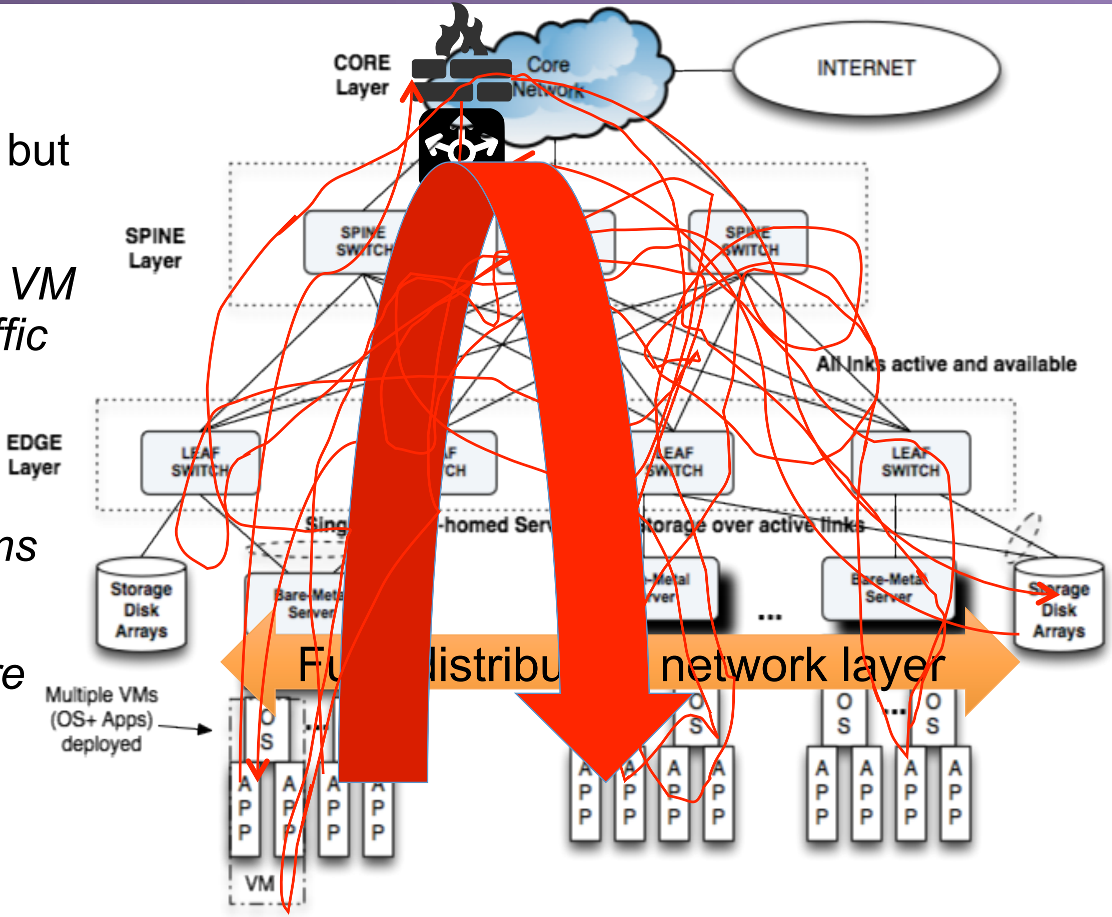
- Server Virtualization:
  - Multiple OS and VMs
- Efficient Network Virtualization:
  - Multiple link utilization
  - Fast convergence
  - Increased uptime
- Storage Virtualization:
  - Fast & efficient
- New design requirements needed!



Effects of Server Virtualization

Virtualization helped optimize compute but added to the network issues:

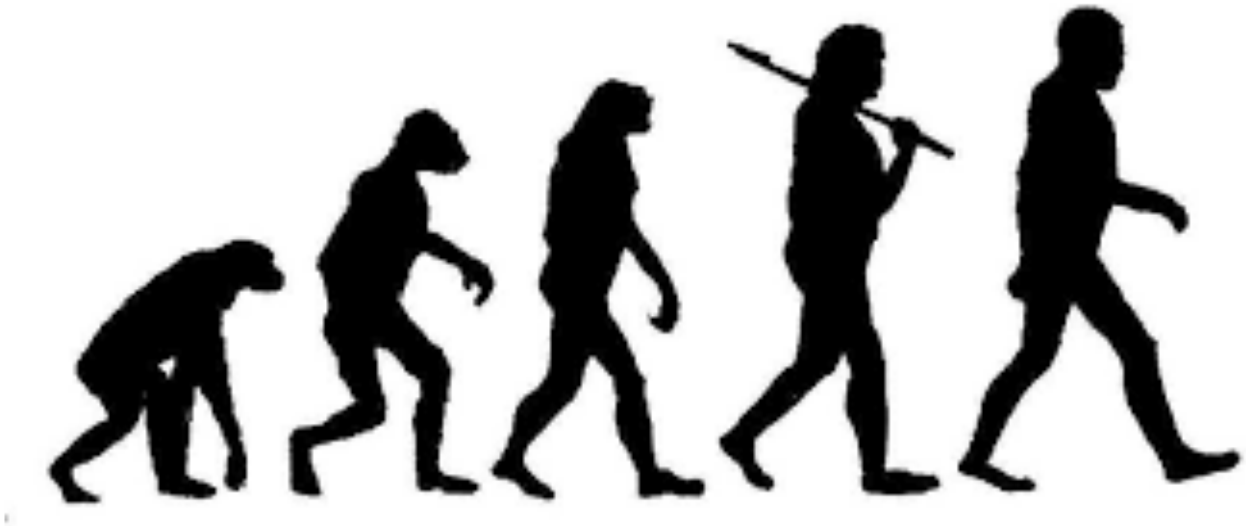
- **Traffic Flows**: *East West and VM to VM flows could cause hair-pinning of traffic*
- **VM Segmentation**: *More VLAN and MAC address issues*
- **VM Management**: *Traditional systems could not see past the hypervisor*
- **Intra Server Security**: *How to secure traffic within a server?*



Hypervisor Networking Layer: Virtual Switches, Distributed Virtual Switches and Network Overlays

A New Networking Layer

Your data plane matters … A LOT



vSwitches

Distributed vSwitches

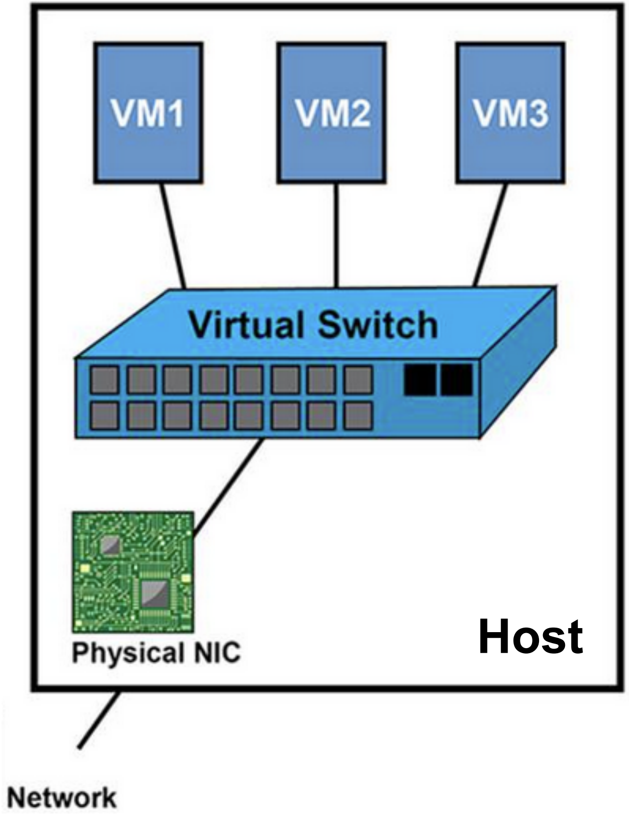
vRouters

Distributed topologies

Extensible data plane

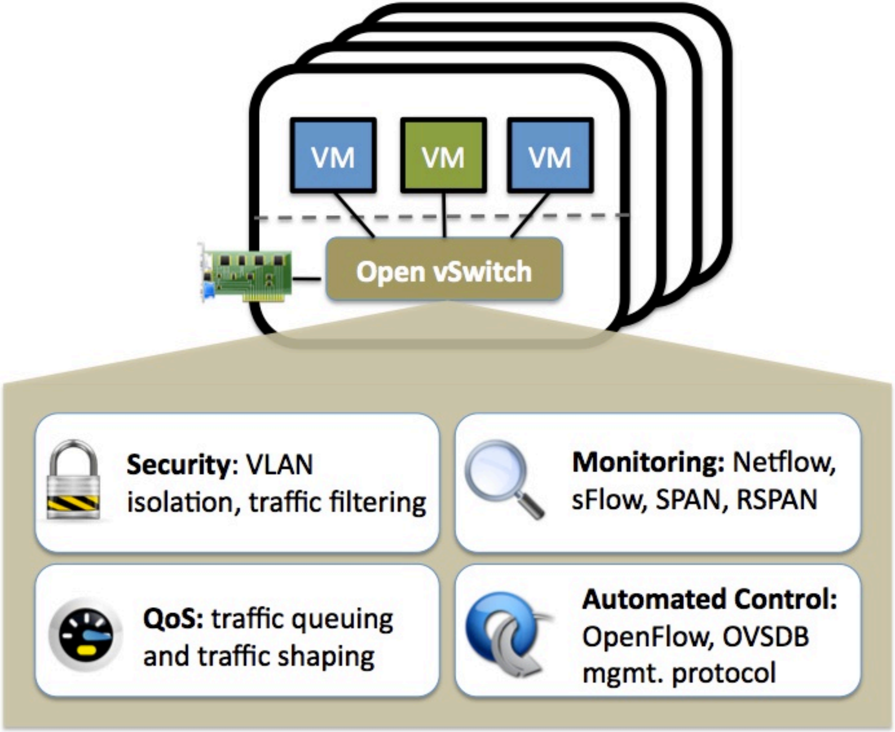
Virtual Switches

- A **Virtual Switch (vSwitch)** is a software component within a server that allows one inter-virtual machine (VM) communication as well as communication with external world
- A vSwitch has a few key advantages:
  - Provides network functionalities right inside the hypervisor layer
  - Operations are similar to that of the hypervisor yet with control over network functionality
  - Compared to a physical switch, it's easy to roll out new functionality, which can be hardware or firmware related

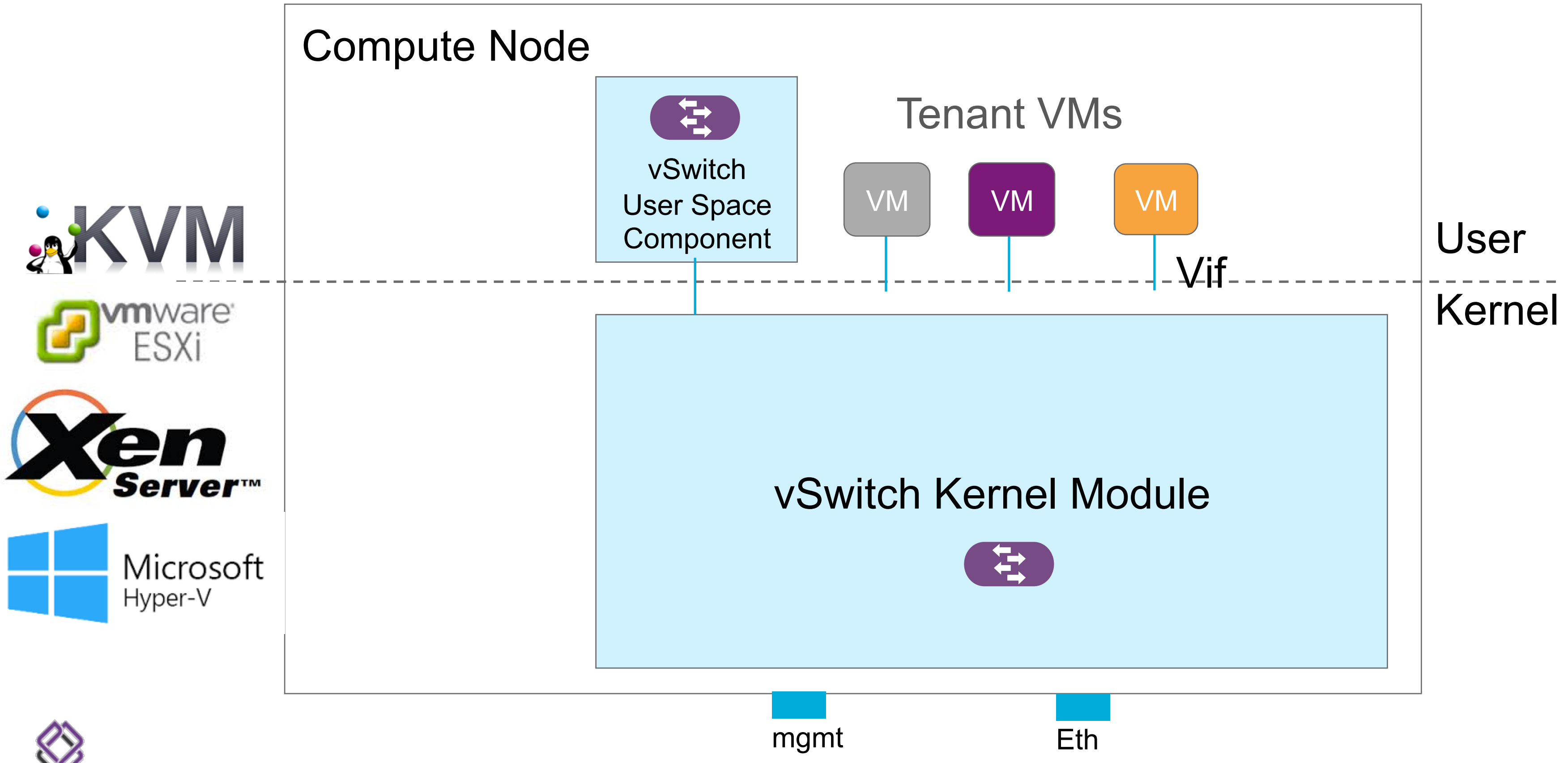


Open vSwitch

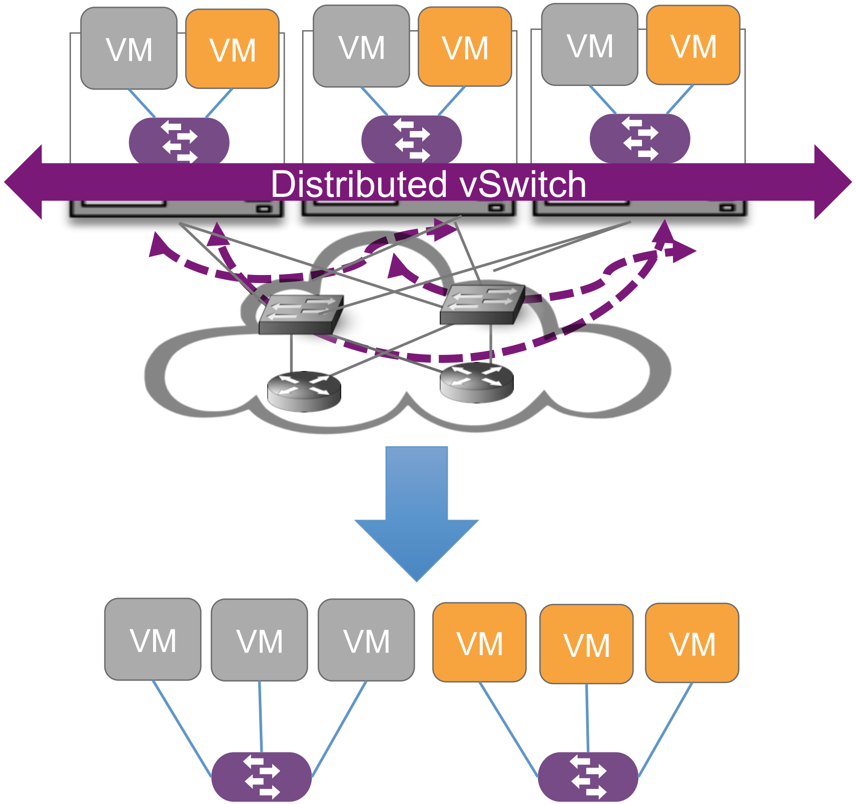
- Open vSwitch is a production quality, multilayer virtual switch licensed under the Apache 2.0 license
- Enables massive network automation
- Supports distribution across multiple physical servers



Inside a Compute Node

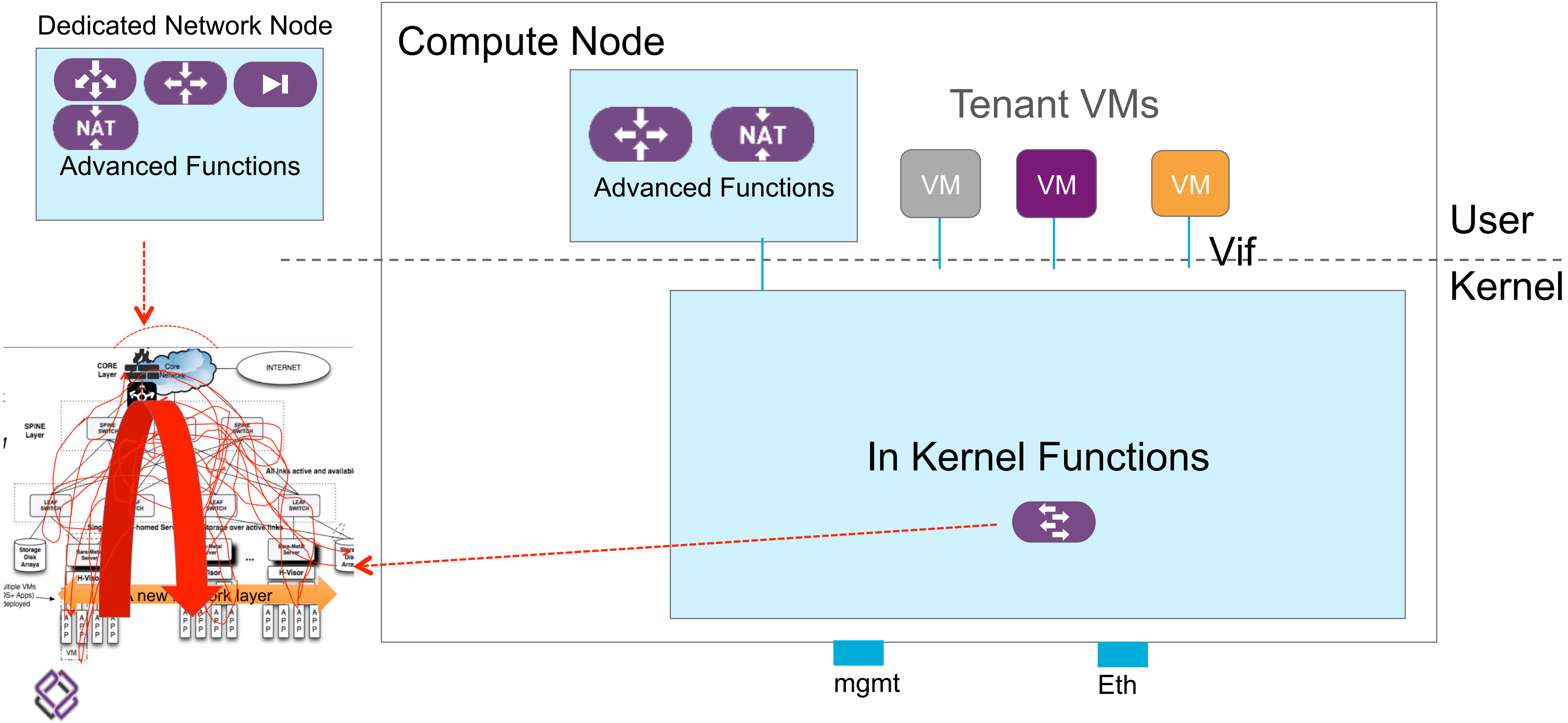


From vSwitch to Distributed vSwitch

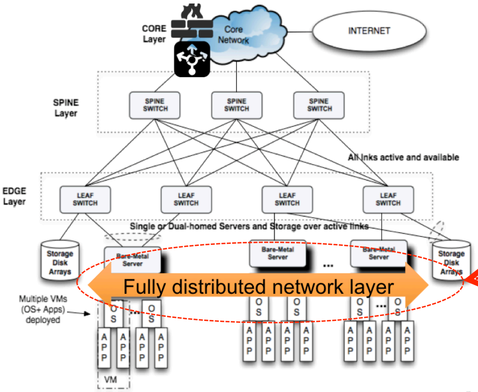


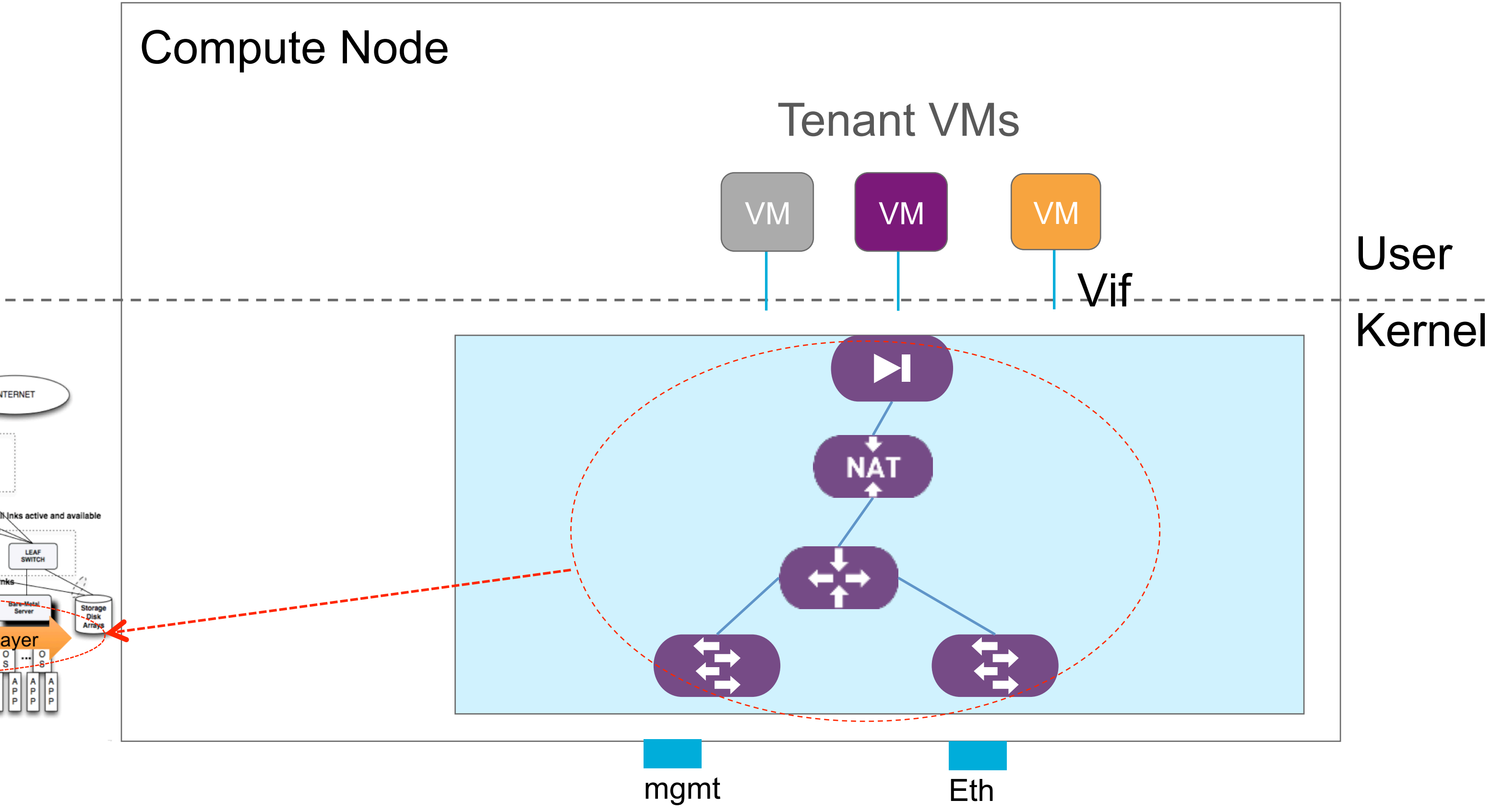
- Logically stretches across multiple physical servers
- Provides L2 connectivity for VMs that belong to the same tenant within each server and across them
- Generally uses **IP tunnel Overlays** (VxLAN, GRE) to create isolated L2 broadcast domains across L3 boundaries

How about L2+ Functions? “in-kernel switch” approach



Extensible In-Kernel Functions





Extensible Data Plane Architecture

- OVS is a great reference architecture however evolving needs of large-scale clouds dictate for a data plane that needs to be
    - Able to load and chain **Virtual Network Functions** dynamically
    - Extensible
    - In-kernel
- E-BPF Technology https://lwn.net/Articles/603983

(E)BPF and its applicability to an Extensible Networking Dataplane – From Virtual Switches to Virtual Networks

Classic BPF

- BPF - Berkeley Packet Filter
- Introduced in Linux in 1997 in kernel version 2.1.75
- Initially used as **socket filter** by packet capture tool tcpdump (via libpcap)

Use Cases:

- socket filters (drop or trim packet and pass to user space)
  - used by tcpdump/libpcap, wireshark, nmap, dhcp, arpd, ...
- In-kernel networking subsystems
  - cls_bpf (TC classifier) –QoS subsystem- , xt_bpf, ppp, team, ...
- seccomp (chrome sandboxing)
  - introduced in 2012 to filter syscall arguments with bpf program

Extended BPF

- **New** set of patches introduced in the Linux kernel since 3.15 (June 8th, 2014) and into 3.19 (Feb 8th, 2015), 4.0 (April 12th, 2015) and into 4.1
- More registers (64 bit), safety, … (next slide)
- In-kernel **JIT** compiler (safe) → x86, ARM64, s390, powerpc*, MIPS* ….
- "Universal in-kernel virtual machine"*
- LLVM backend: any platform that LLVM compiles into will work. (**GCC** backend in the works) → **PORTABILITY**!

Use Cases:

1. **networking**
2. tracing (analytics, monitoring, debugging)
3. in-kernel optimizations
4. hw modeling
5. crazy stuff...

*http://lwn.net/Articles/599755/

Extended BPF program = BPF instructions + BPF maps

- BPF instructions improvements:

BPF `insns` program (pre-3.15)	Extended BPF `insns` program
2 registers + stack 32-bit registers 4-byte load/store to stack 1-8 byte load from packet Conditional jump forward +, -, *, … instructions	10 registers + stack 64-bit registers 1-8 byte load/store to stack 1-8 byte load/store to packet Conditional jump fwd and backward Same + signed_shift + bswap

- BPF **map**: key/value storage of different types (hash, lpm, …)
  - `value = bpf_table_lookup(table_id, key)` – lookup key in a table
  - Userspace can read/modify the tables
  - More on this on later slide

Extended BPF Networking Program Example

Fully Programmable Dataplane Access

Restrictive C program to:

- obtain the protocol type (UDP, TCP, ICMP, …) from each packet
- keep a count for each protocol in a “map”:

```
int bpf_prog1(struct __sk_buff *skb)
{
    int index = load_byte(skb, ETH_HLEN +
offsetof(struct iphdr, protocol));
    long *value;

    value = bpf_map_lookup_elem(&my_map, &index);
    if (value)
            __sync_fetch_and_add(value, 1);

    return 0;
}
```

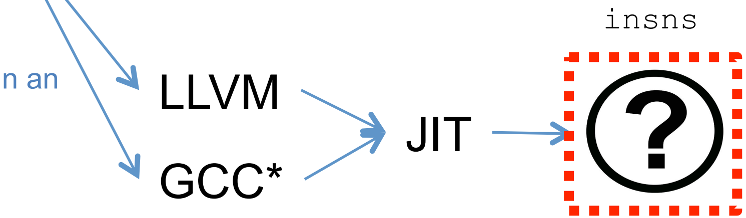
Load an incoming frame and get the IP protocol as “index” from it

Lookup that IP protocol “index” in an existing **map*** and get current “value”

If found, add 1 to the “value”

Equivalent eBPF program

```
struct bpf_insn prog[] = {
           BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
           BPF_LD_ABS(BPF_B, 14 + 9 /* R0 = ip->proto */),
           BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0,
-4), /* *(u32 *)(fp - 4) = r0 */
           BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
           BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4), /*
r2 = fp - 4 */
           BPF_LD_MAP_FD(BPF_REG_1, map_fd),
           BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0,
BPF_FUNC_map_lookup_elem),
           BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
           BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
           BPF_RAW_INSN(BPF_STX | BPF_XADD | BPF_DW,
BPF_REG_0, BPF_REG_1, 0, 0),
           BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
           BPF_EXIT_INSN(),
   };
```

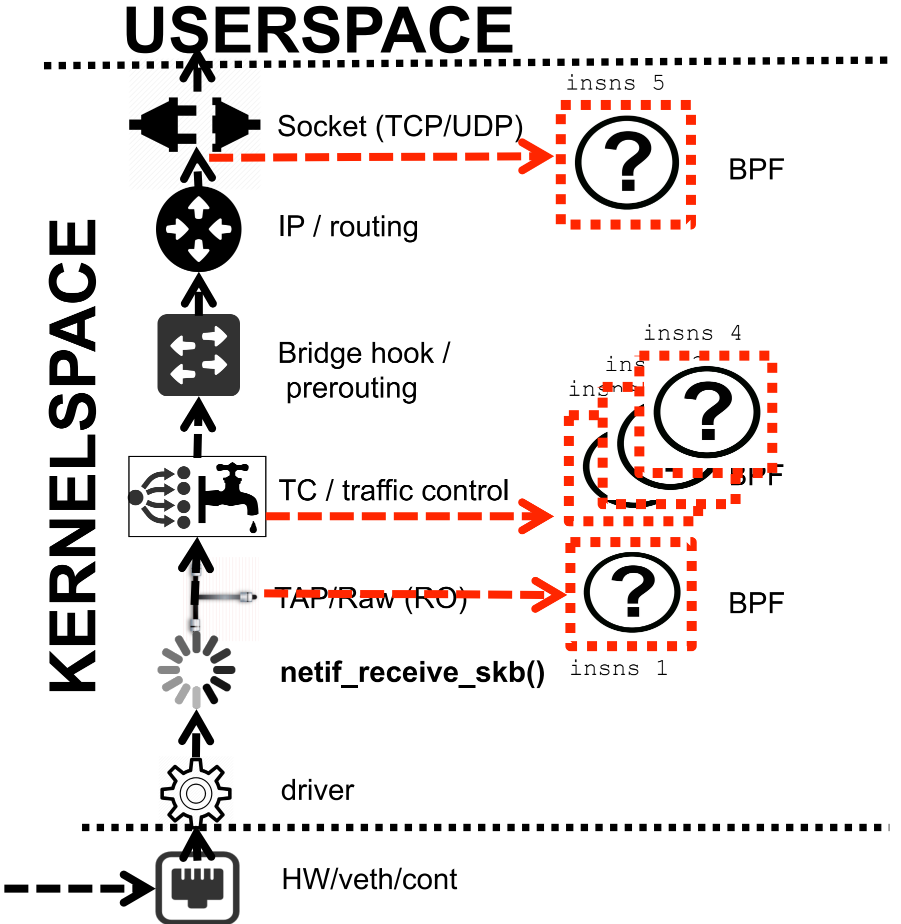


Blazing FAST in-kernel machine code!

https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/samples/bpf/sockex2_kern.c

Hooking into the Linux networking stack (RX)

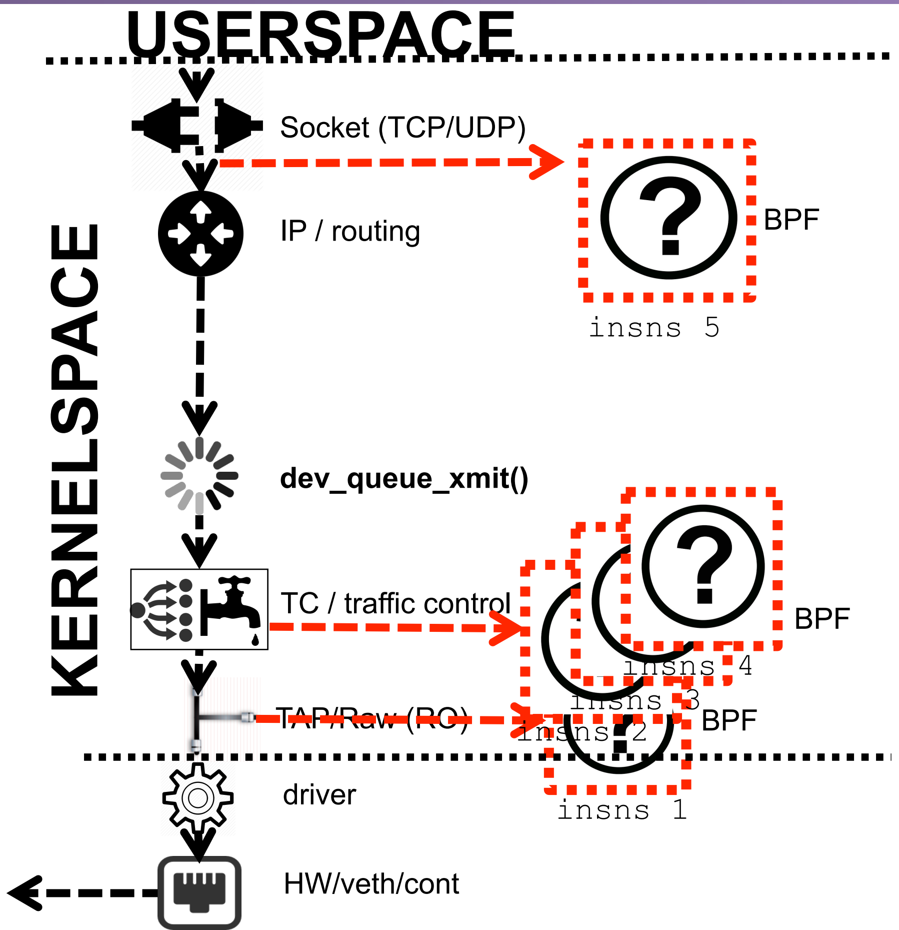
- BPF programs can attach to **sockets**, the **traffic control (TC)** subsystem, kprobe, syscalls, tracepoints…
- Sockets can be STREAM (L4/UDP), DATAGRAM (L4/TCP) or RAW (TC)
- This allows to hook at different levels of the Linux networking stack, providing the ability to act on traffic that has or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



Hooking into the Linux networking stack (TX)

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

For simplicity, the following slides simplify this view into a single "kernel networking stack"



Extended BPF system usage
Userspace "Call" and "Helpers"

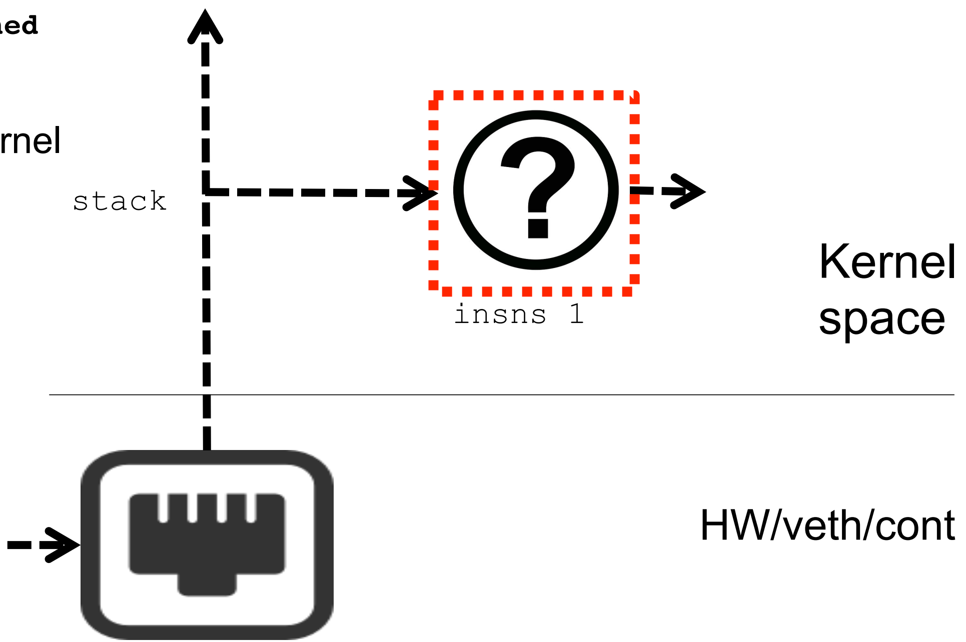
Enables "in-kernel VNFs"

- BPF Linux 'call' and set of in-kernel helper functions define what BPF programs can do

```
int bpf(BPF_PROG_LOAD, union bpf_attr *attr, unsigned
int size);
```

- BPF code itself acts as 'glue' between calls to in-kernel helper functions
- BPF helpers allow for additional functionality
  - `ktime_get`
  - `packet_write`
  - `fetch`
  - `map_lookup/update/delete`

  (more on maps later)



Extended BPF Program definition → struct bpf_attr

```
int bpf(BPF_PROG_LOAD, union bpf_attr
*attr, unsigned int size);
```

- struct bpf_attr defines the BPF program when it's passed onto the BPF call

```
struct { /* used by BPF_PROG_LOAD command */
__u32 prog_type; /* program type */
__u32 insn_cnt;
__aligned_u64 insns; /* 'struct insns *' */
__aligned_u64 license; /* 'const char *' */
__u32 log_level; /* verbosity level */
__u32 log_size;
__aligned_u64 log_buf;
};
```

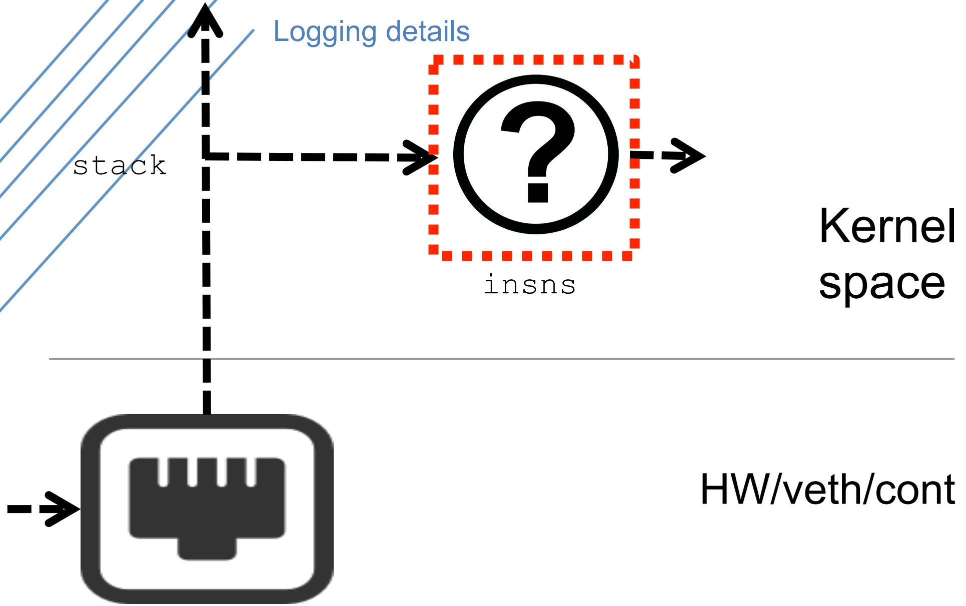
"TRACING" , "SOCKET",…

Size of the program (in BPF instructions)

The BPF **insns** program itself (see previous slides)

License (allows to force to GPL or not load)

Logging details



Extended BPF "maps"

- Maps are generic storage of different types for sharing data (key/value pairs) between kernel and userspace

- The maps are accessed from user space via BPF syscall, with commands:

  - create a map with given type and attributes and receive as file descriptor:

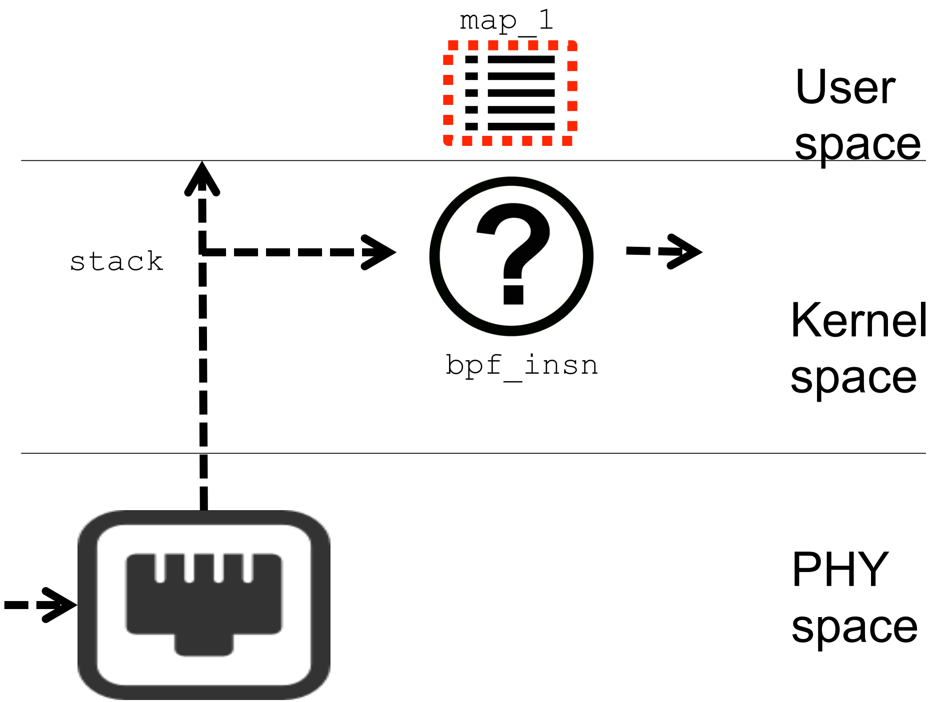
```
map_fd = bpf(BPF_MAP_CREATE, union bpf_attr *attr,
u32 size)
```

  - Additional calls to perform operations on the map:

`lookup key/value, update, delete, iterate, delete` a map

- userspace programs use this syscall to create/ access maps that BPF programs are concurrently updating

Tables for "in-kernel VNFs"



Putting it all together -- Networking with BPF
Attach a program to a socket

- User creates an eBPF program and obtains a

`union bpf_attr` (previous slides)
that includes the **`insns`** BPF instruction set for the program.

- A userspace program loads the eBPF program:

```
int bpf(BPF_PROG_LOAD, union bpf_attr *attr, unsigned int size);
```

- It also creates a map, controlled with a file descriptor

```
map_fd = bpf(BPF_MAP_CREATE, union bpf_attr *attr, u32 size)
```

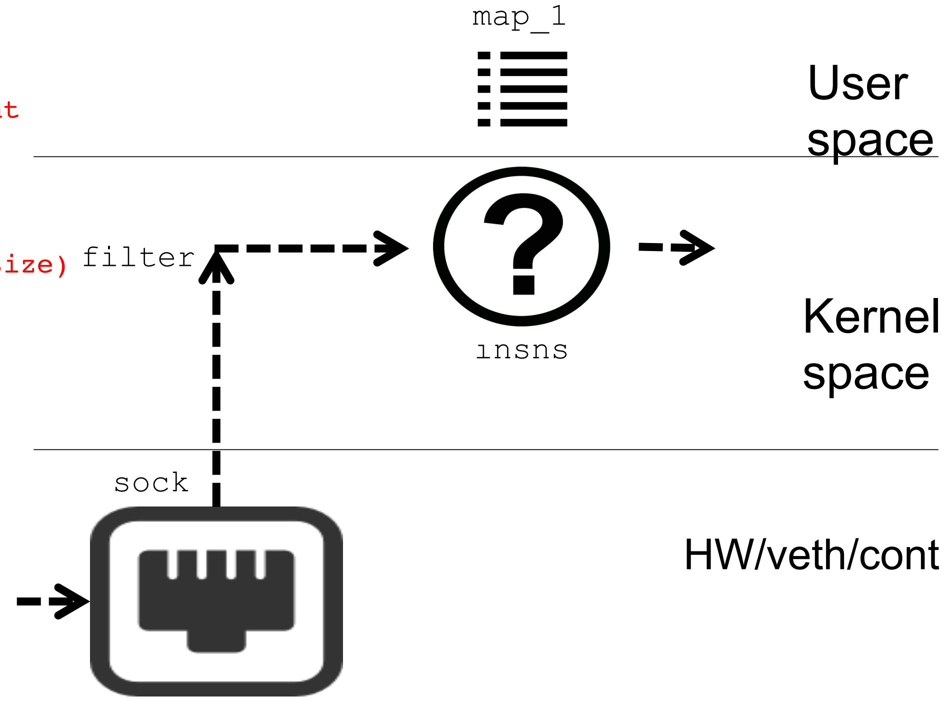
- Create a socket (varies depending on socket type):

```
socket = socket(PF_INET, SOCK_STREAM,IPPROTO_TCP)
```

- Attach the BPF program to a socket

```
setsockopt(socket, SOL_SOCKET, SO_ATTACH_BPF, &fd, sizeof(fd));
```

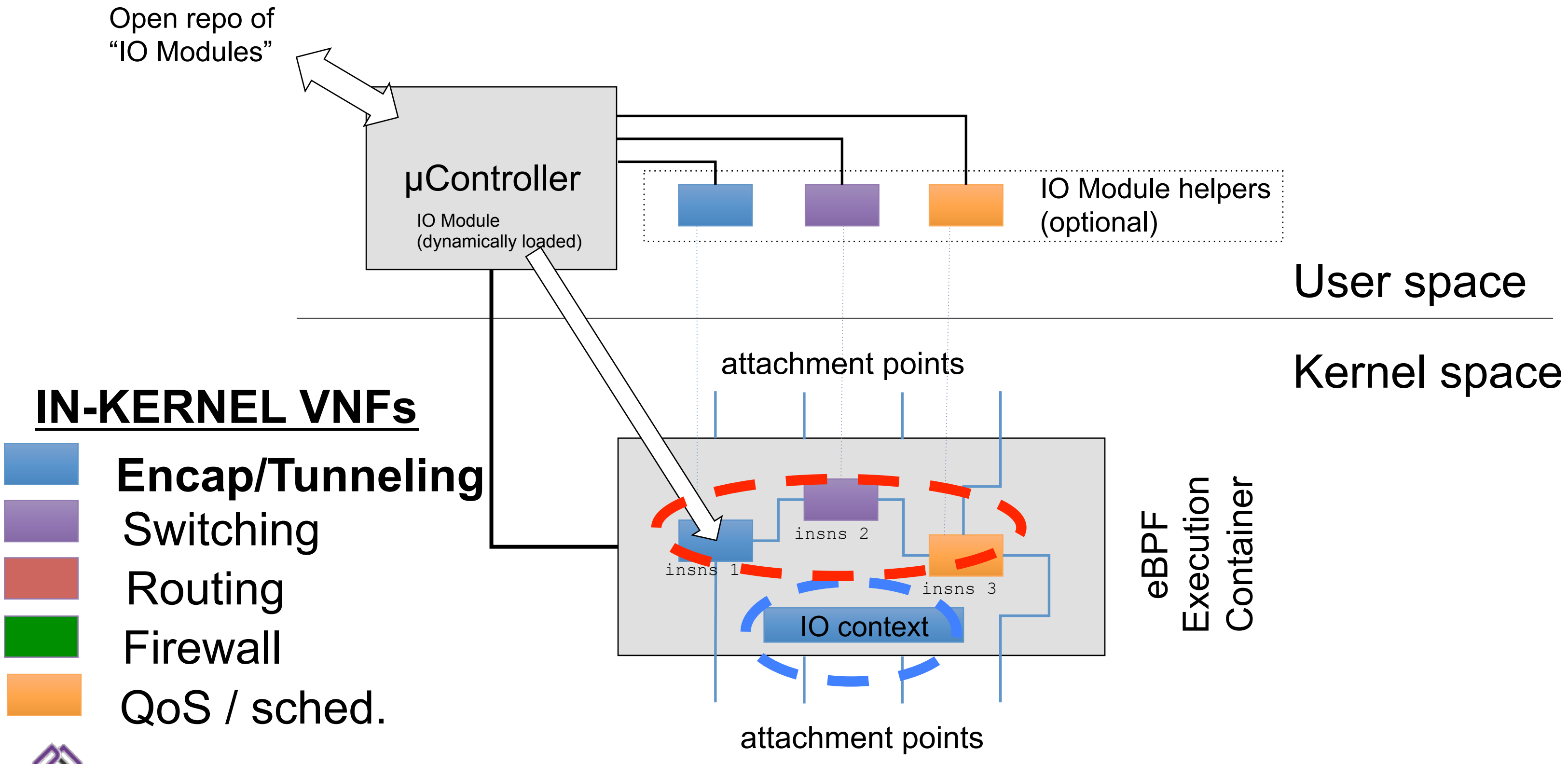
- **Enjoy in-kernel networking nirvana ☺**



Additional BPF Networking Usage Examples

- https://www.kernel.org/doc/Documentation/networking/filter.txt
- https://lkml.org/lkml/2014/11/27/10
- http://git.kernel.org/cgit/linux/kernel/git/shemminger/iproute2.git/tree/examples/bpf/bpf_prog.c?h=net-next

eBPF framework for networking
*Building **Virtual Network Infrastructure***

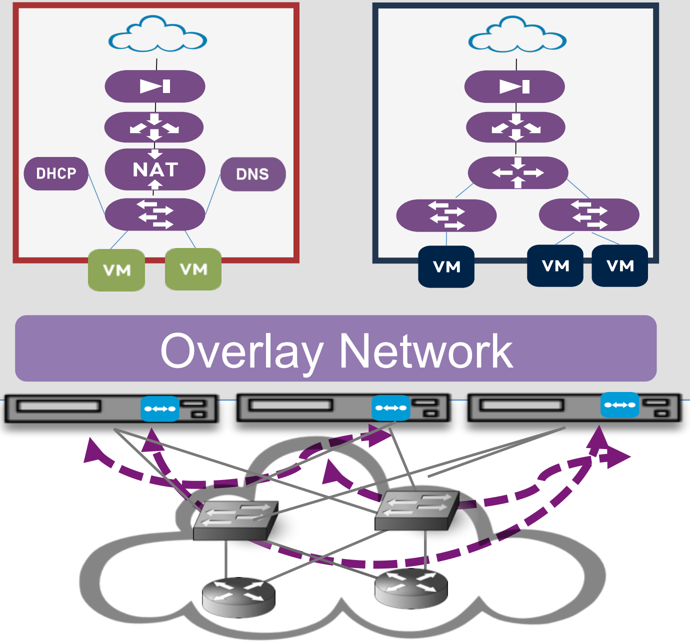


A new Data Center Design
Physical and Virtual Network Infrastructure

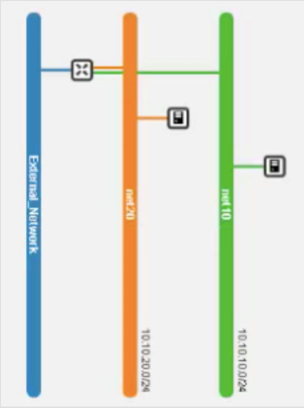
The new Data Center
Physical and Virtual Network Infrastructure

VIRTUAL INFRASTRUCTURE VIEW

TENANT NETWORKS



- On-Demand
- Multi Tenant
- Automated
- Self Service
- Secure
- Distributed



PHYSICAL INFRASTRUCTURE VIEW

- QoS, Bandwidth
- Latency
- Multicast
- Capacity
- Connectivity

The new Data Center
Physical and Virtual Network Infrastructure

Physical Network Infrastructure

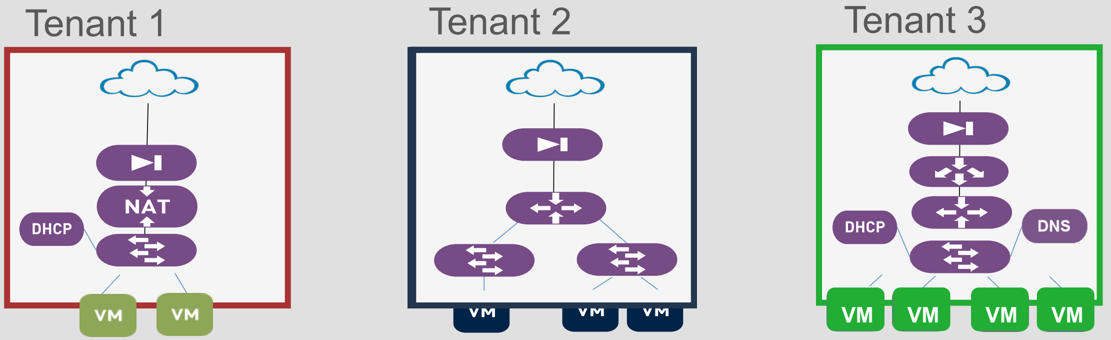
- Towards a non-blocking **transport** “fabric”
- Life-spine architecture for optimal connectivity
- “Install and maintain”
- Well understood routing protocols
- New pods can easily be rolled out with a flat networking design
- Multi-vendor

Virtual Network Infrastructure

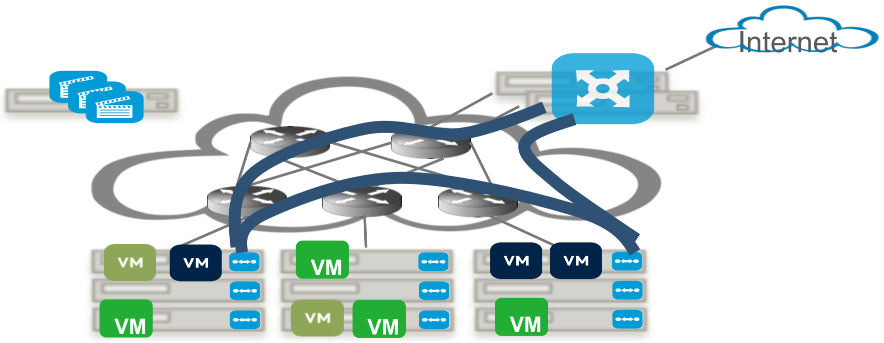
- Service provisioning layer
- Rich Networking topology to satisfy the most stringent application requirement
- Automatic service-chaining
- On-demand provisioning (devops model)
- Easy-to-manage operational model, upgrade cycles & fault containment

Virtual Network Infrastructure Application: *Multitenant Virtual Networks*

VIRTUAL INFRASTRUCTURE VIEW

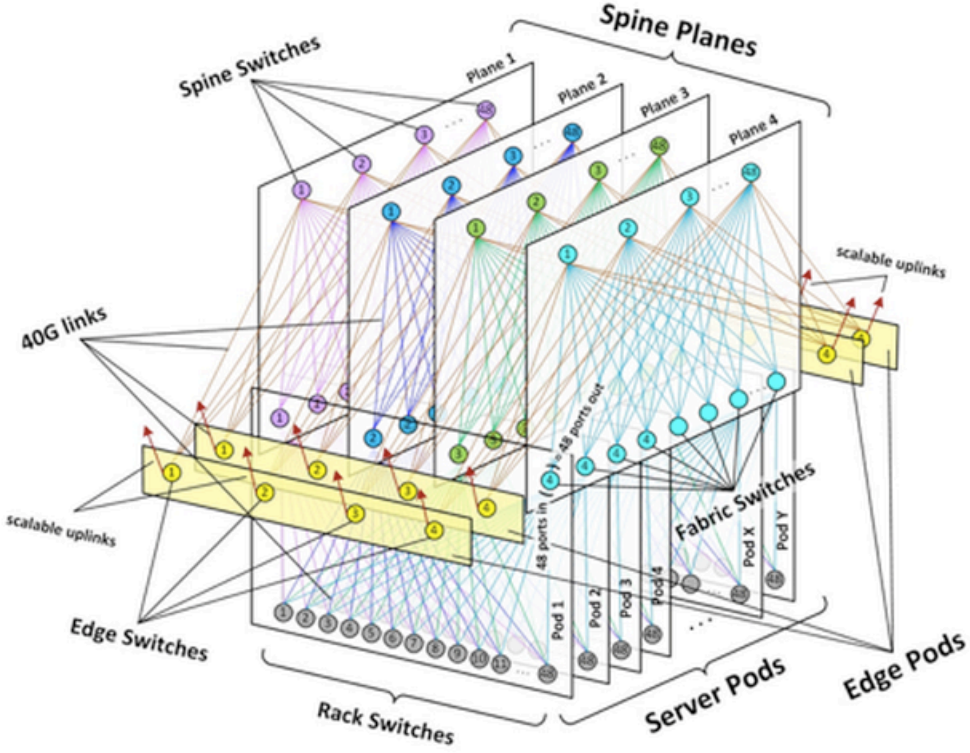


PHYSICAL INFRASTRUCTURE VIEW



Physical Network Infrastructure
Insights from Web-scale deployments

- Small efficient building blocks
- Highly-modular
- Scalable with a non-blocking architecture
- Automation, automation & automation



https://code.**facebook**.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/

Thank You