Lightweight Virtualization

Lecture 11

Srinivas Narayana

http://www.cs.rutgers.edu/~sn624/553-S25

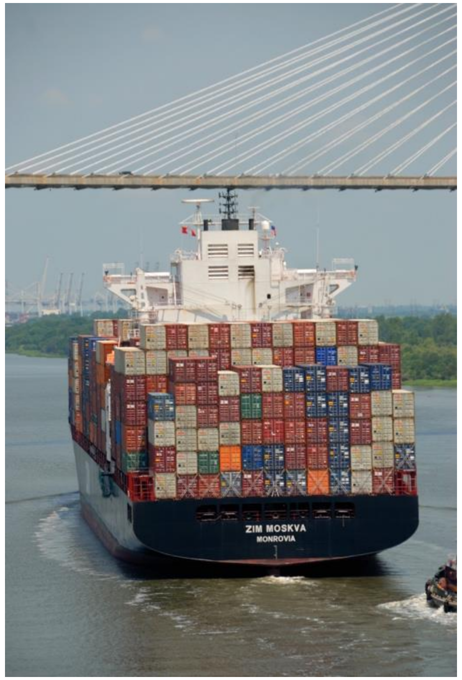
RUTGERS
UNIVERSITY | NEW BRUNSWICK

Hardware for System Virtualization

- Last lecture: system virtualization “without” hardware support
  - e.g., x86-32: Use techniques such as DBT and paravirtualization
- CPU and memory hardware support much improved since the 00s!
  - Instruction set and architectural extensions: Intel VT-x, AMD-v
  - Extended page tables: Hardware support for multiple address translations
  - IO support through SR-IOV and IOMMUs
- Paravirtualization is still useful for (further) efficiency but unnecessary for reasonably efficient basic VM functionality
- VMMs built for architectures with native hardware support:
  - KVM/qemu, integrated with Linux
- Xen was the basis of Amazon’s public cloud; KVM since ~2019
  - Do we always need the heavyweight hammer of full virtualization?

Lightweight, OS-level virtualization

- First-party workloads: e.g., within a single company
  - Some degree of implicit trust
  - Consolidation and efficient resource use is more important than full isolation
- Lightweight, operating-system-level virtualization: Containers
- Programs use the system call interface (ideally nothing else)
- No emulation, no need for special hardware support, or OS changes for e.g., paravirtualization
- Containers do need OS changes for finer-grained resource abstraction and control



Benefits of OS-level virtualization

- Application-centric view
- Run any app that is portable with the same system call interface
- No more management of machines & OSes; think of applications
- Decouple the management of OS & hardware from applications
  - Roll out new hardware and OS without worrying about breaking apps
- Match development, testing, and deployment environments
- Convenient access points to communicate with the application
  - E.g., expose health information, communicate resource allocations
- Relate machine telemetry to applications
  - No need to tease out per-app metrics from machine-level metrics
  - “The container is the application.”

Benefits of OS-level virtualization

- Resource sharing across virtualized units (containers)
- Shared OS kernel & utilities limit redundancy & improve consolidation
- Familiar kernel resource abstractions: process scheduling, memory allocation, etc.

- Container refers to two things at once:
  - the run-time abstraction (process, access+resource isolation, FS)
  - the stored software image (all software you need to run)

How are containers built?

What goes into a container?

- More like process virtualization than system virtualization
  - No ISA virtualization; no native hardware support
  - Memory and IO work the same way as processes
- What we call a container is a loose conglomeration of kernel-level mechanisms
- Namespaces: Access isolation for global resources
- Cgroups: Resource/Performance isolation of global resources
- UnionFS: Improving efficiency through shared filesystem data
- Access control mechanisms: capabilities, filtering (eBPF, seccomp, appArmor)

Namespaces

- Access isolation
- Show an instance of a global resource as available to all processes inside a namespace (multiplexing)
- Changes visible to other processes within namespace, but invisible outside the namespace
- Show different “copies” of resources associated with the kind of namespace
  - Network, IPC, mount, PID, …
- Every process starts in init namespace, change with `setns`
- Network: (software/hardware) network device; routing rules; port numbers. `veth` pair connects two network namespaces

Control groups

- Resource/Performance isolation
- Subsystem: a specific kind of resource
  - CPU time, memory, network bandwidth, block device access, priority, CPU and memory (numa) node assignment
  - Many configurable parameters per subsystem
- Control group or cgroup: a set of processes
  - fork()-ed process inherits a bunch of parent attributes including cgroup
- Hierarchy: a tree where each node is a cgroup
  - Many hierarchies can exist, unlike the process hierarchy
- Each subsystem “mounted” onto one hierarchy
  - Possible to use a single hierarchy for multiple subsystems (resources)
- Every process has exactly one reservation per resource

UnionFS: ~~“software images too big”~~

- Context: Data on storage typically “mounted” at some point in the virtual filesystem (/, /home/users/name, etc.)
- Containers want mostly the same files, with a small number of unique modifications per container (e.g., specific library versions)
  - Think: common third-party packages, utilities, shared library images
- Union filesystem: maintain a stack of filesystems at each mount point. Only the highest one is writable; lower layers are read-only
- Inspired by similar use cases in the past: data on a read-only medium that needed a small number of updates before refreshing into a new medium (e.g., working on a CD filesystem in memory)
- Write fresh to the top; copy-on-write; copy up; deletion with “whiteout”. Cache heavily
- Virtual Filesystem (VFS) layer accomplishes this with minimal changes to underlying filesystem

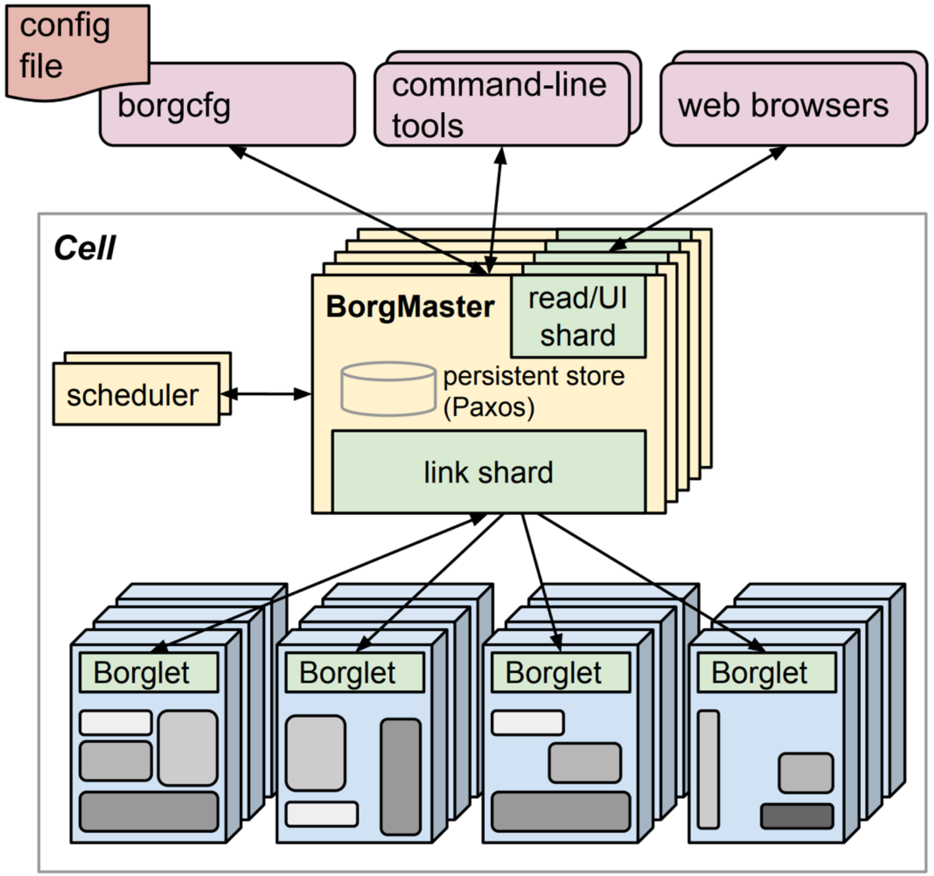
Orchestrating Containers

Why Orchestration?

- When containers are so easy to use, users will create many
- Example: Instances of a microservice
- Example: Co-locating latency-sensitive jobs with batch jobs
- Kubernetes: an orchestrator created and evolved at Google
  - Today, a well-established project with a significant open-source ecosystem
- Pod: a group of related containers (app and allied processes)
  - E.g., web service, along with logging, metrics
- Node: the machine (virtual or physical) on which pod scheduled

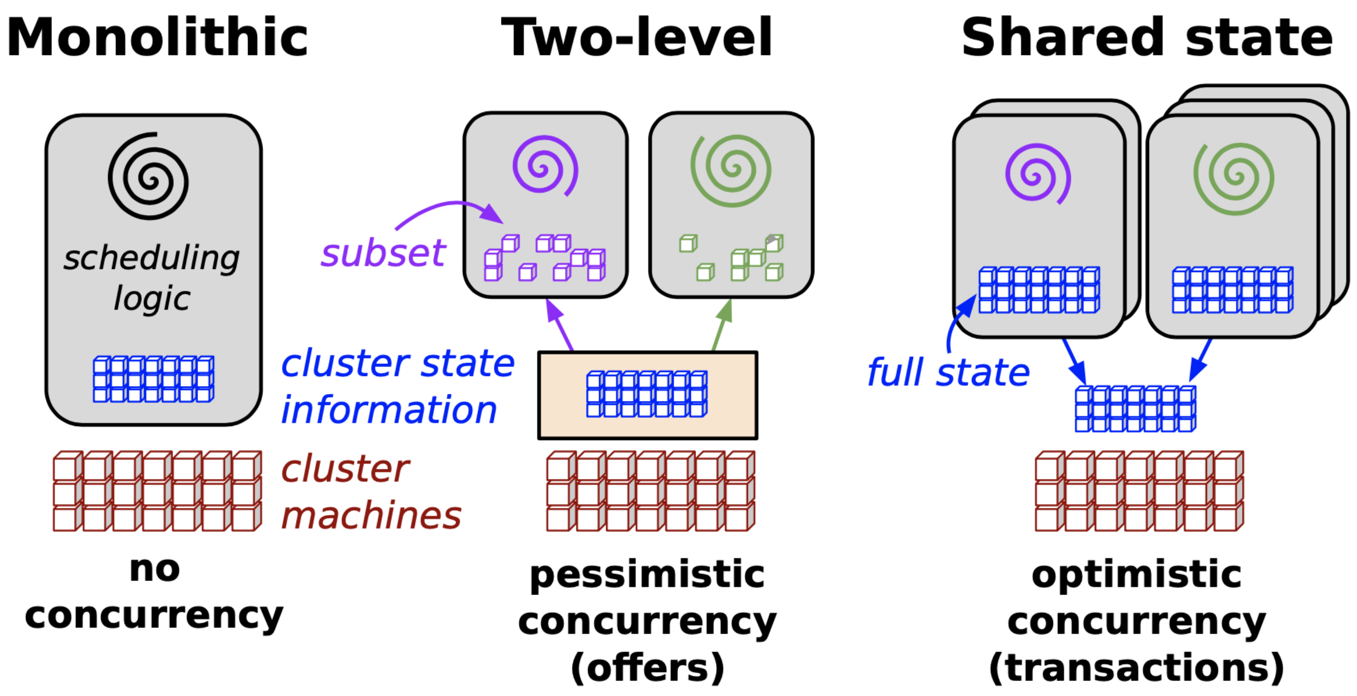
Evolution at Google: Borg

- Borg: a cluster manager
- Cells: units of machines managed by one controller
- Borgmaster: controller
- Borglet: program running locally on each machine to manage its resources
- Cluster state: the controller's view of the mapping from tasks (containers) to nodes, health, resource allocation, etc.
- State is persisted in a highly-available distributed data store (e.g. Paxos)



Evolution at Google: Omega

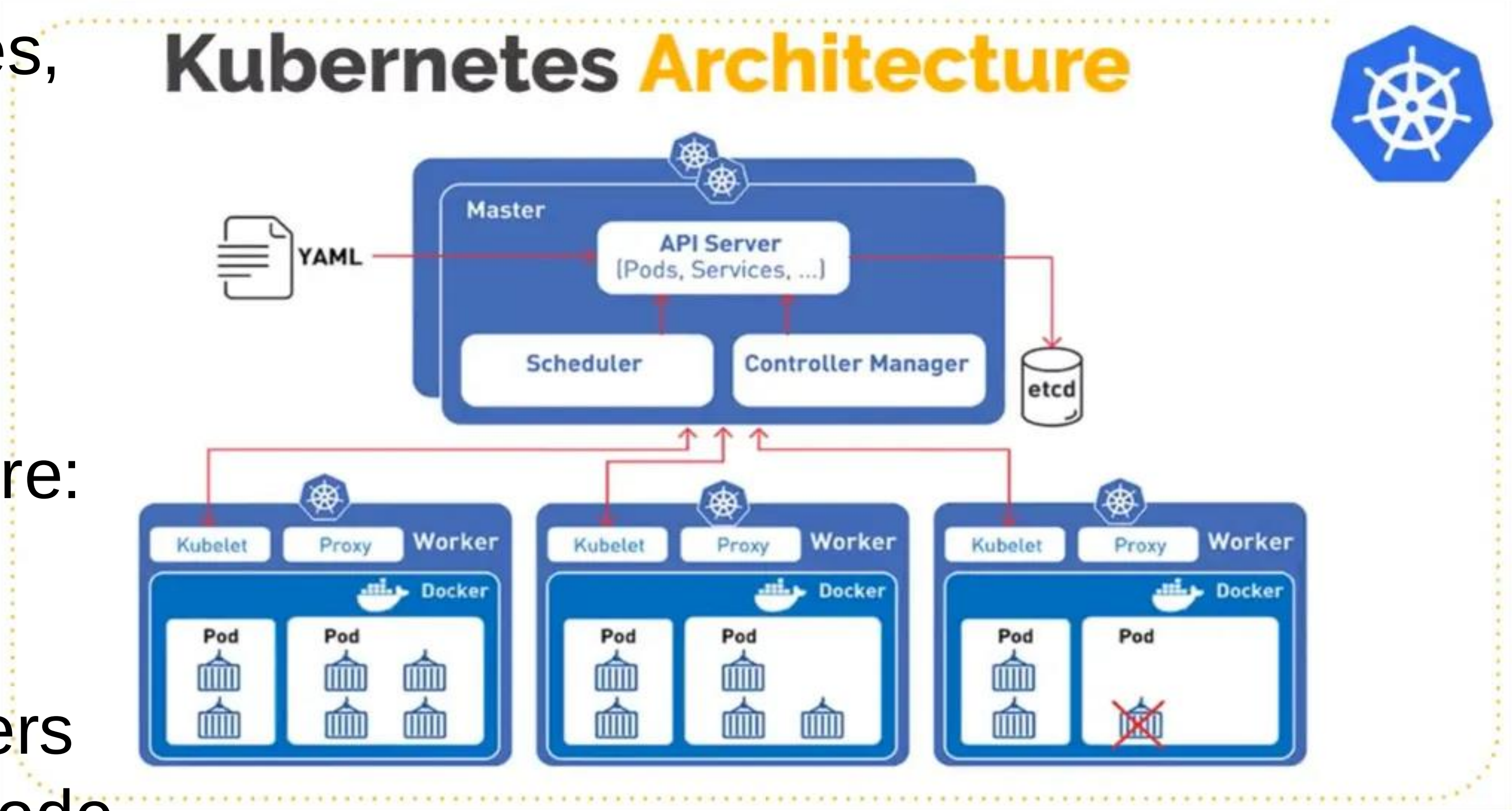
- The controller needs to manage many different aspects of the cluster
  - Mapping from container to node
  - Resource allocation per container
  - Number of instances per container
  - Automatic scaling based or demand and usage
- Decouple cluster state from the (one) controller
- Use many controllers



Clients of the cluster state can read/write the state directly

Evolution at Google: Kubernetes

- Manage access to the cluster state through an API server
- Validation of policies, versioning, default objects, etc.
- Highly-available strongly consistent distributed data store: etcd
- Still use many decoupled controllers
- Kubelet: manage node



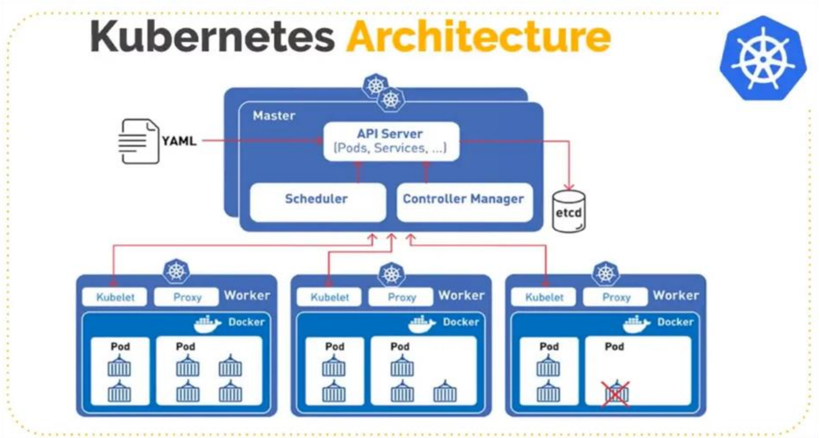
https://blog.devops.dev/the-kubernetes-architecture-e1ee01ccb76d

Kubernetes principles

- Consistent object representations
  - Metadata (name, ID, version, labels)
  - Specification (desired state)
  - Status (observed state, read-only)
- Reconciliation controller loop: Make the observed state (status) match the desired state (specification)
  - Example: number of replicas of a pod
- Many modular and interacting controllers
  - Example: Auto-scaling and Replica controllers
  - A failed-then-restarted controller has direct access to the observed state; no need to maintain complex internal state machines


```yaml
apiversion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  revisionHistoryLimit: 5
  minReadySeconds: 10
  selector:
      matchLabels:
        app: nginx
        deployer: distelli
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
    replicas: 3
    template:
      metadata:
        labels:
            app: nginx
            deployer: distelli
      spec:
        containers:
          - name: nginx
            image: nginx: 1.7.9
```

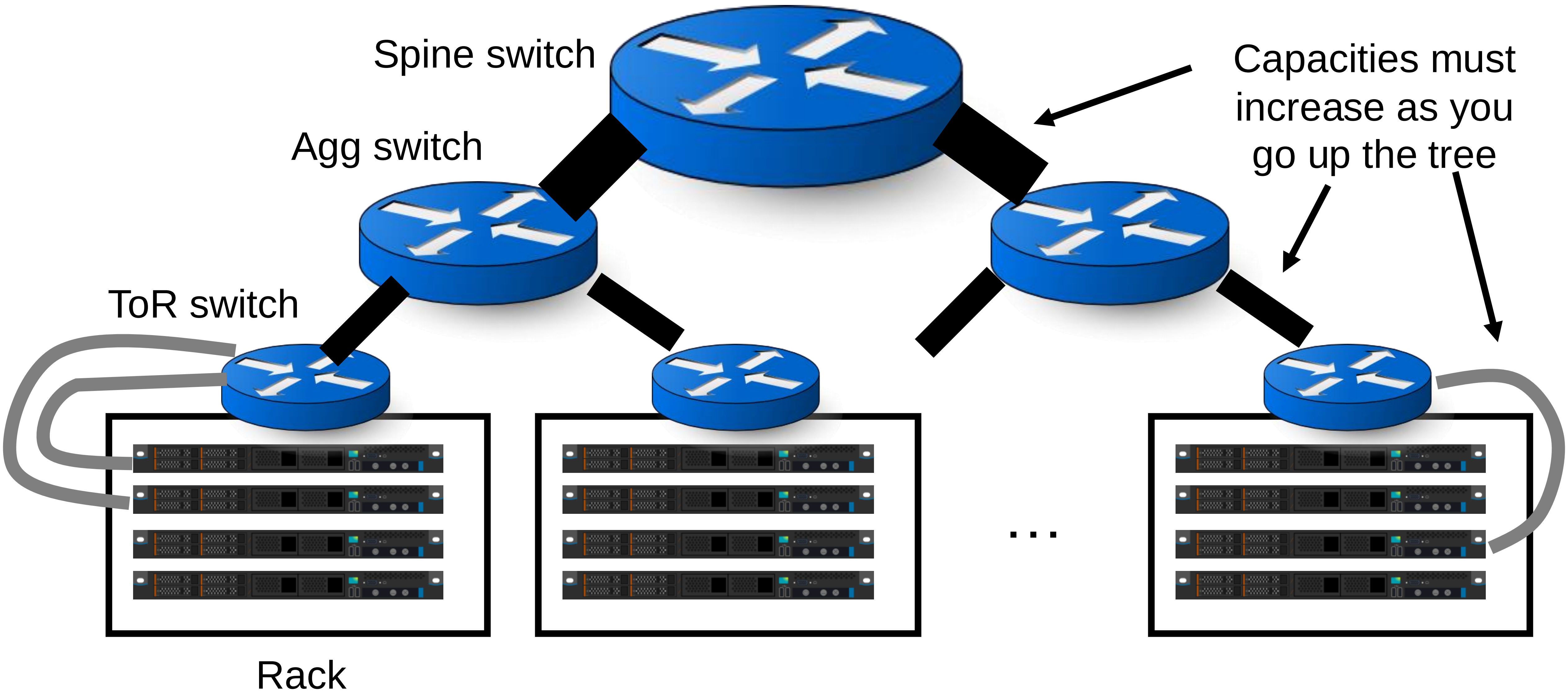
More principles



- Each pod gets its own IP address
  - Visible to other pods and apps in the same Kubernetes cluster
  - Full access to all ports
  - This IP address need not agree with physical IP address of the node
  - Container Network Interface (CNI) to manage addressing & routing
- Labels to group containers
  - Don't just number the containers
  - Key-value pairs that allow operator to define any attribute,
  - e.g., role=frontend
- Label selectors are sufficiently flexible to manage containers at time-varying granularity that is specified at operation time
  - e.g., a controller that only manages role=frontend pods

Virtualizing Networking in a Shared Cluster

Typical network structure: Fat Trees



Goals

- Terminology:
  - tenant/customer and provider
  - Virtual NIC (vNIC): network interface exposed with SR-IOV or network namespaces
- (1) Place tenant workloads on any physical machine
- (2) Scale or migrate tenant workload across physical machines at any time
- (3) Simplify configuration for everyone involved
  - Views of tenant addresses and interfaces
  - Tenant apps using load balancing, DNS-based IP discovery, etc.
  - Provider’s ability to plumb tenant workloads together
  - Migration from on-premise compute cluster to shared cloud

Design Choice: CA’s or PA’s?

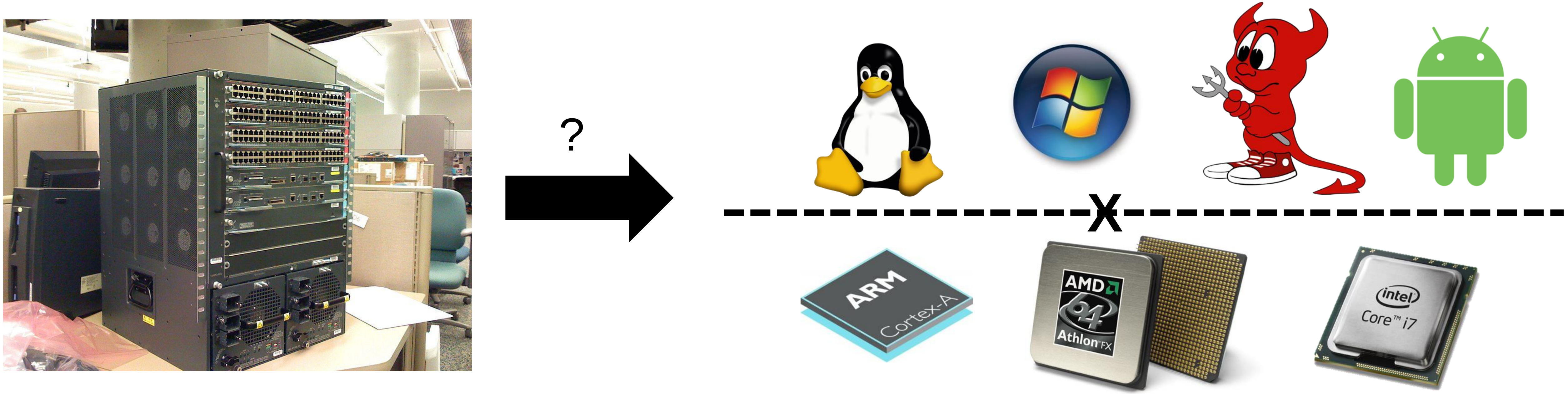
- Do VMs/pods use their own “customer addresses” (CA’s) or use the infrastructure’s “provider addresses” (PA’s)?
- PA’s: supporting routing is “business as usual”
  - But one tenant’s ports affected by other tenants on same machine
  - Need static allocation of ports to tenants, or dynamic port discovery
  - Reduced isolation, more complex configuration, app changes
- CA’s: dedicated IP per VM/pod, visible to applications
  - Clean and backwards compatible. e.g. DNS
  - If VM/pod A sees its own address to be X, any VM/pod B talking to A also thinks that A has address X. A is reachable with CA address X.
  - However, need to design networking to route between CA’s,
  - Example: migrate VMs/pods across PA’s with unchanging CA

Networking in a multi-tenant data center

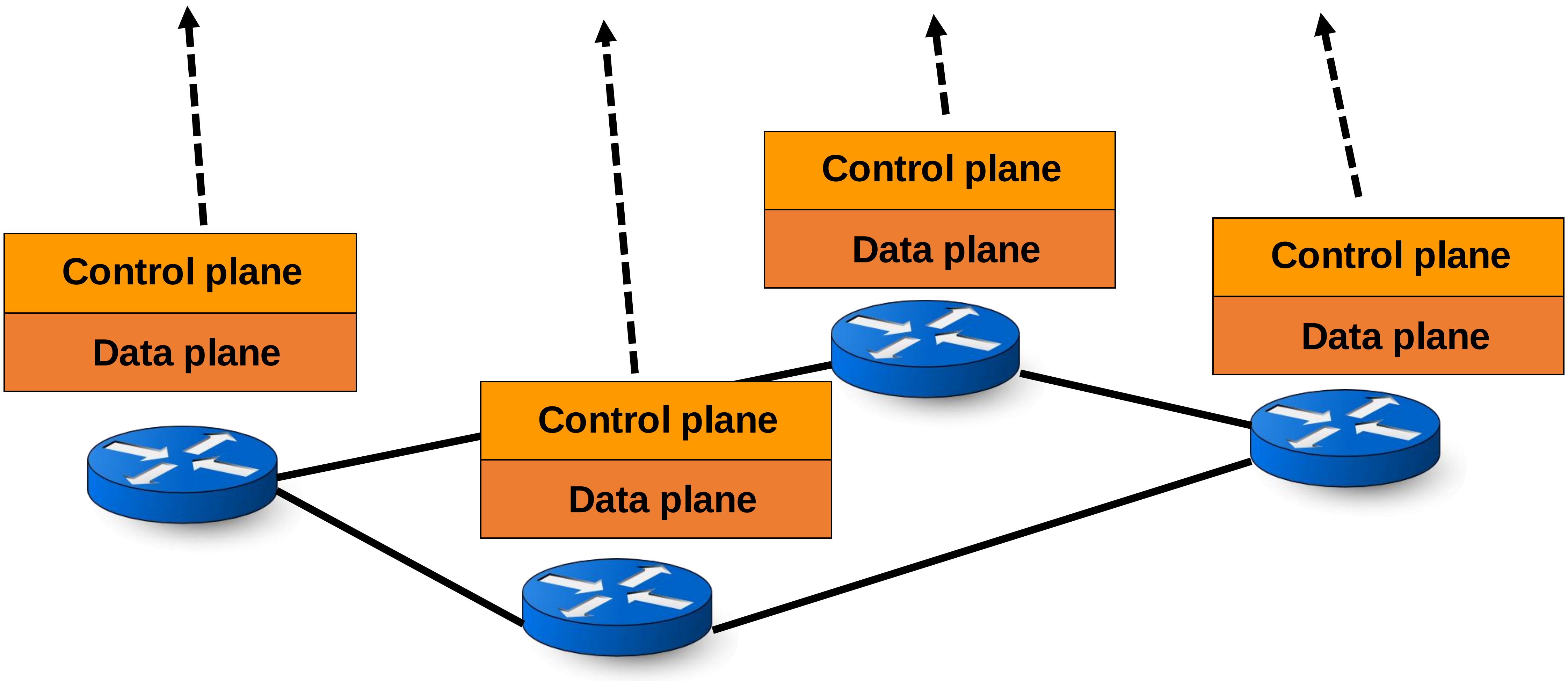
- Address virtualization: VMs/pods use own addresses (CA's)
  - Physical network does not know how to route CA's
  - Additional software to translate CA's between PA's: Tunneling
  - Tunneling endpoint (TEP): software tun/tap interface, NIC hardware, or software switch within a hypervisor. Overlay.
  - TEP encapsulates and decapsulates packet headers (VXLAN, GRE)
- Topology virtualization: Tenants should be able to bring own custom network topologies or assume "one big switch"
  - Facilitate migration into public cloud, consistent view for tenant's monitoring and maintenance tools, etc.
- Supporting service models for the network
  - e.g., rate limits and isolation across tenants sharing a physical machine

Network control is typically distributed

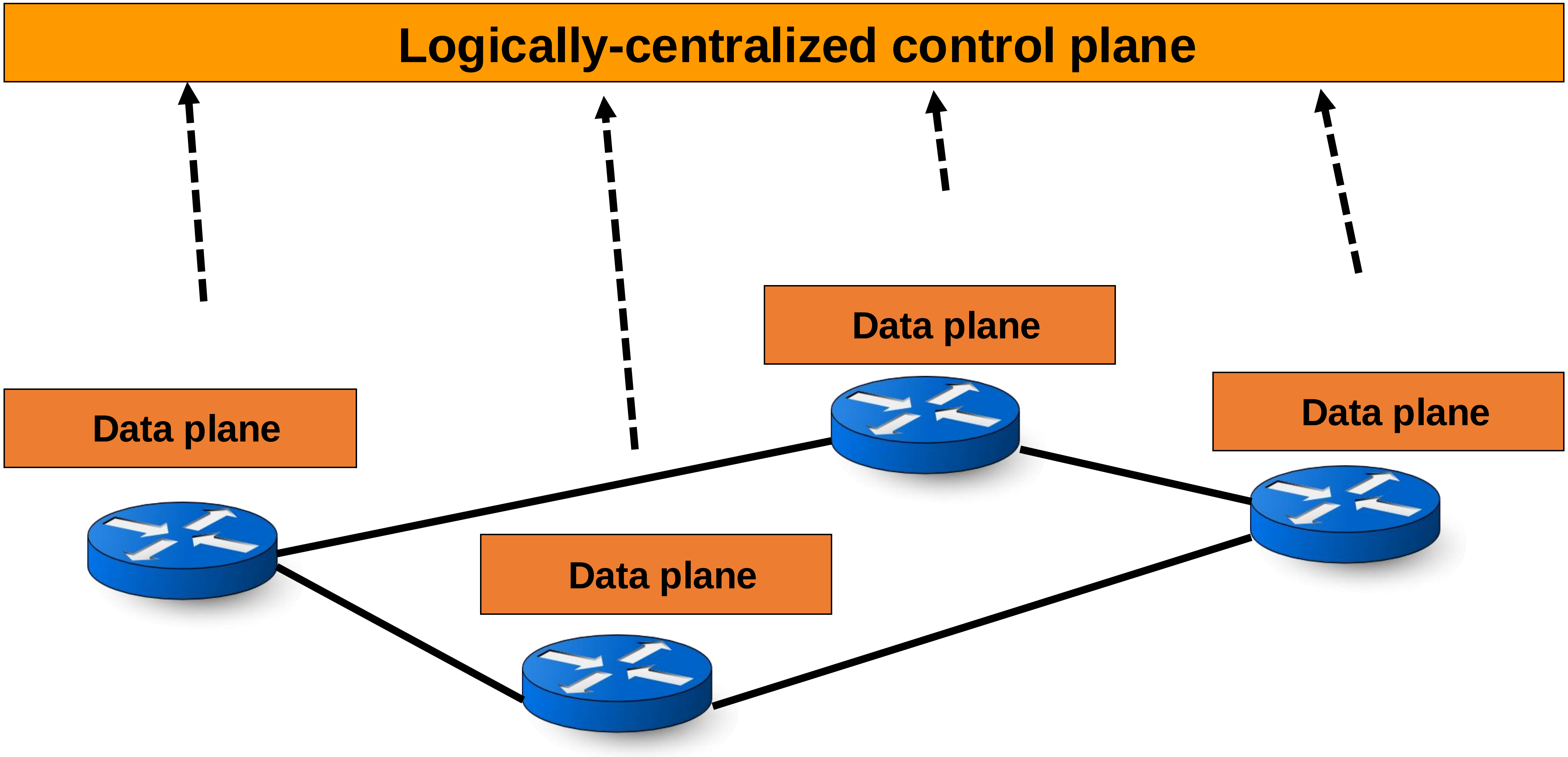
- Traditional IP network: Management tied to distributed protocols
  - Ex: Set OSPF link weights to force traffic through a desired path
  - Ex: Non-deterministic network state after a link failure
- Data and control plane controlled by vendors: proprietary interfaces



Traditional IP network



Software-defined network

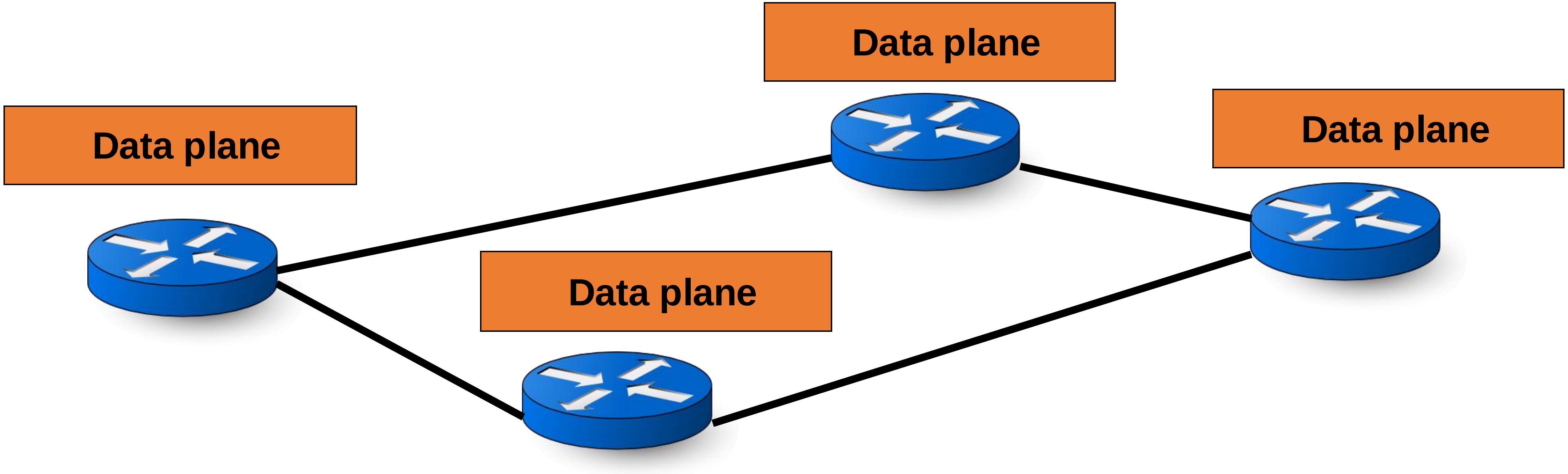


Software-Defined Networking

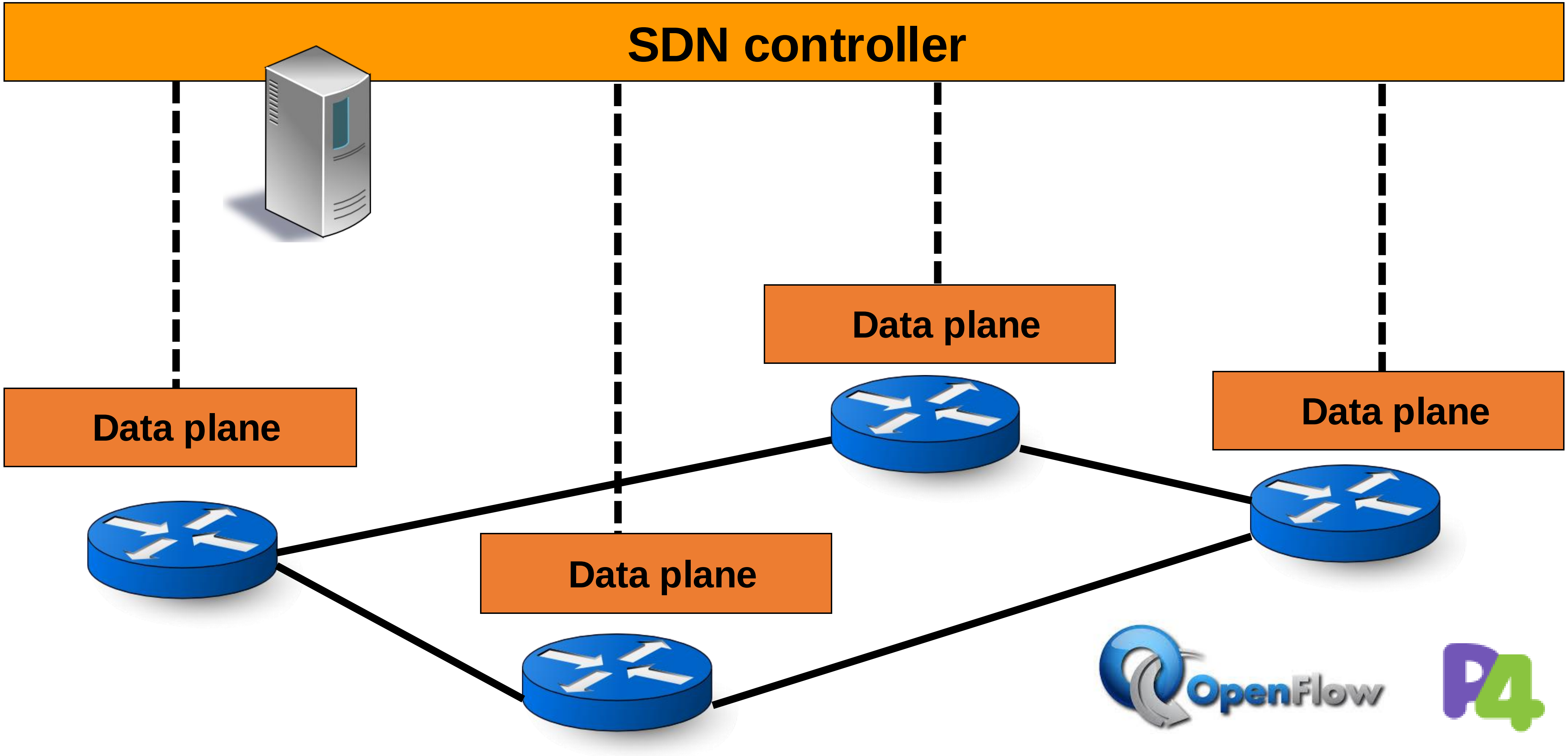
SDN (1/2): Centralized control plane

SDN controller

Control planes lifted from switches
… into a logically centralized *controller*
… running in a compute cluster

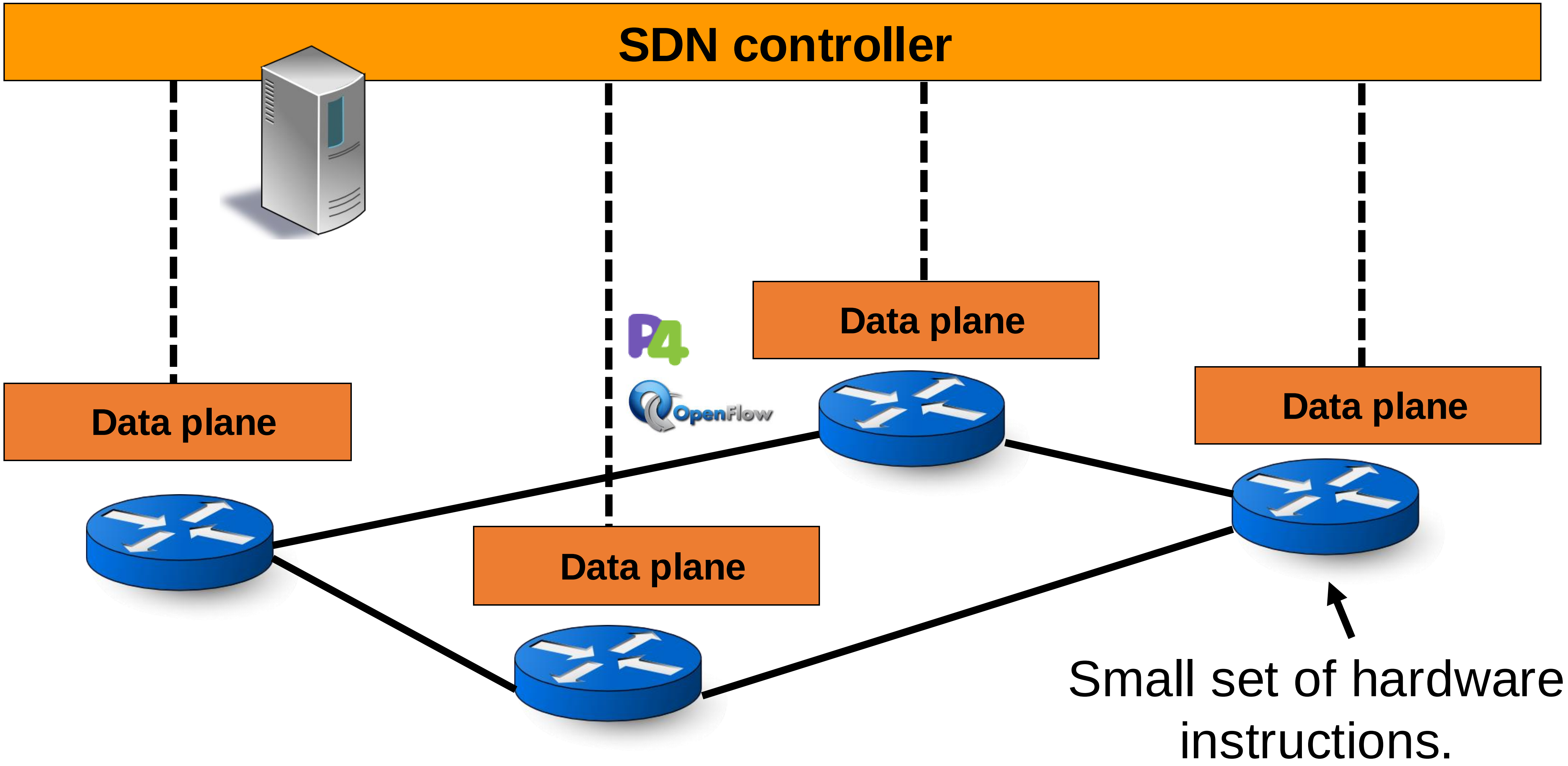


SDN (2/2): Open interface to data plane



Some immediate consequences

(1) Simpler switches



Data plane primitive: Match-action rules

- Match arbitrary bits in the packet header

Match: 1000x01xx01001x

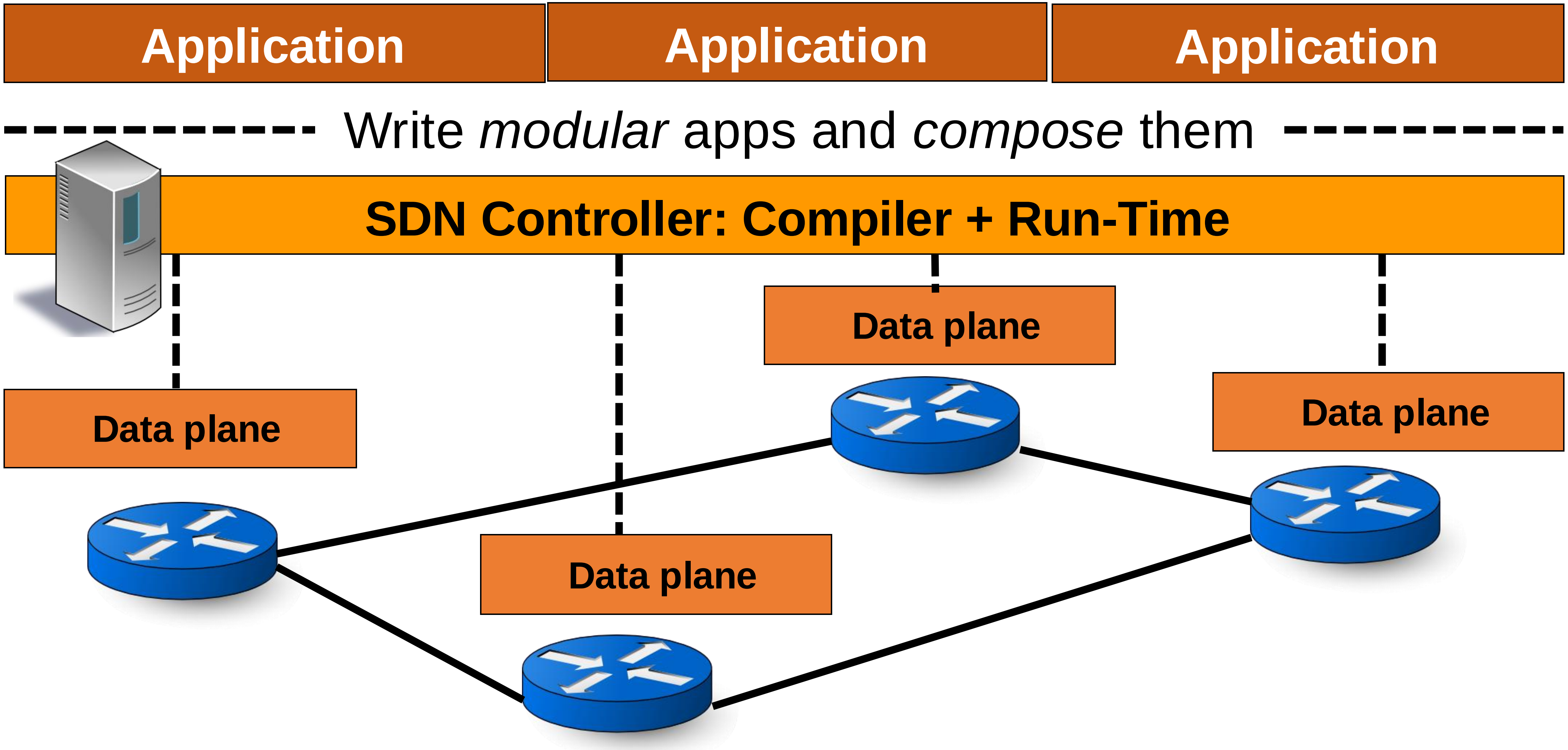
  - Match on any header, or new header
  - Match exact, a subset (ternary), or over a range
  - Allows any flow granularity
- Actions
  - Forward to port(s), drop, send to controller, count,
  - Overwrite header with mask, push or pop, …
  - Forward at specific bit-rate

Action: fwd(port 2)

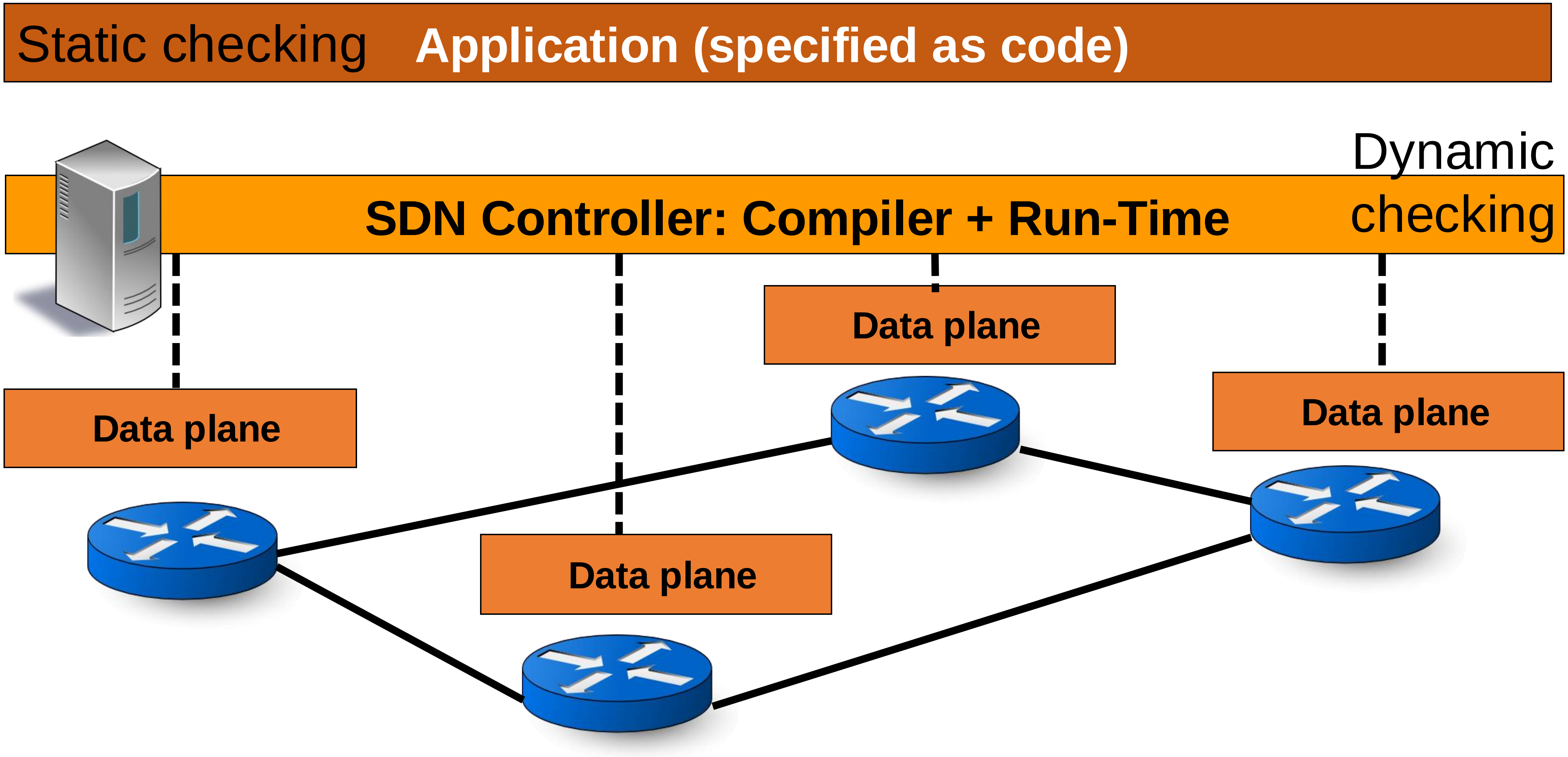
- Prioritized list of rules

Priority: 65500

(2) Network programming abstractions



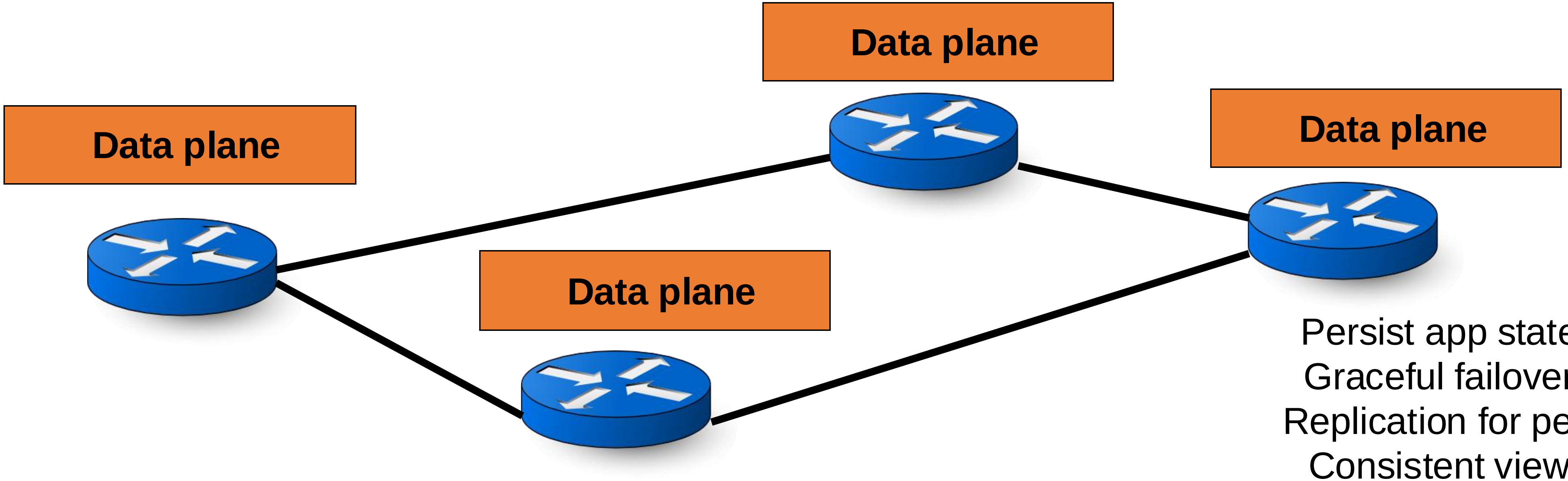
(3) Formal verification of Network Policy



(4) Unified network operating system

Application	Application	Application
Network Operating System		

Separate distributed system concerns from expressing intent



Persist app state
Graceful failover
Replication for perf
Consistent view

New technical challenges of SDN

- Availability: surviving failures of the controller
- Controller scalability: many routers, many events
  - Response time: Delays between controller and routers
- Consistency: Ensuring multiple controllers behave consistently
- Designing flexible router mechanisms
- Compilation: translating intent to mechanisms
- Verification: ensuring controller policy is faithfully implemented
- Security: entire network owned if the controller is exploited
- Interoperability: legacy routers; neighboring domains; …

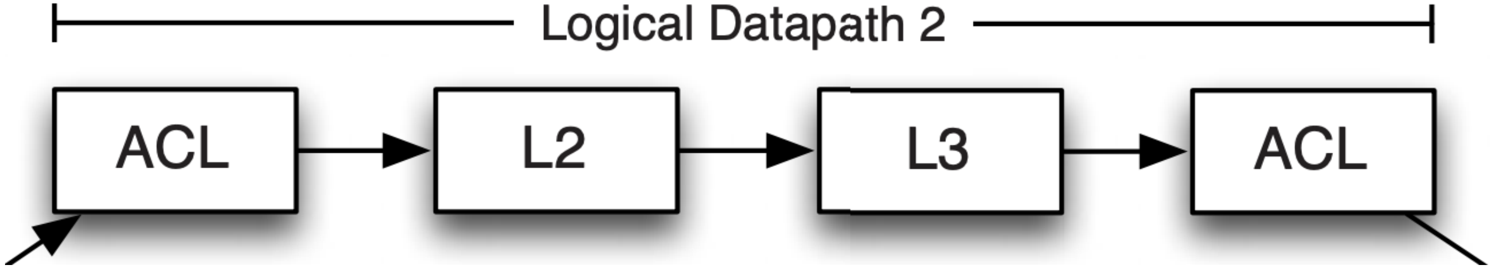
Legacy?

- Openflow is just a protocol. The details can change or become irrelevant, but the philosophy is longer-lasting
- Programming software switches: Match-action abstraction common everywhere
- Basic OVS modules available on the linux kernel, etc.
- Programmable hardware switches/routers common today
- P4: protocol independence and stateful behavior in switches
    - In-network computing

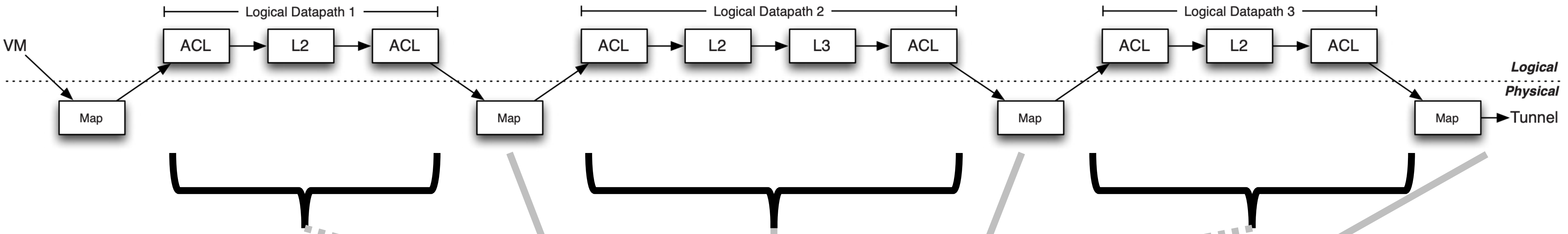
Examples of Network Virtualization

Example 1: Nicira Virtualization Platform

- NVP: Motivated by migration of on-premise cloud workloads as seamlessly as possible to cloud
- Address virtualization: VM’s see and use CA’s
- Topology virtualization (bring your own topology)
  - packets processed through logical switch/router tenant topology
  - Tables populated by classic routing protocols (e.g. OSPF, BGP)
- Edge: logical datapaths and TEPs (vNIC → hypervisor OVS)
- Network core is a simple pipe that routes between TEPs



Topology and Address Virtualization



Topology
virtualization

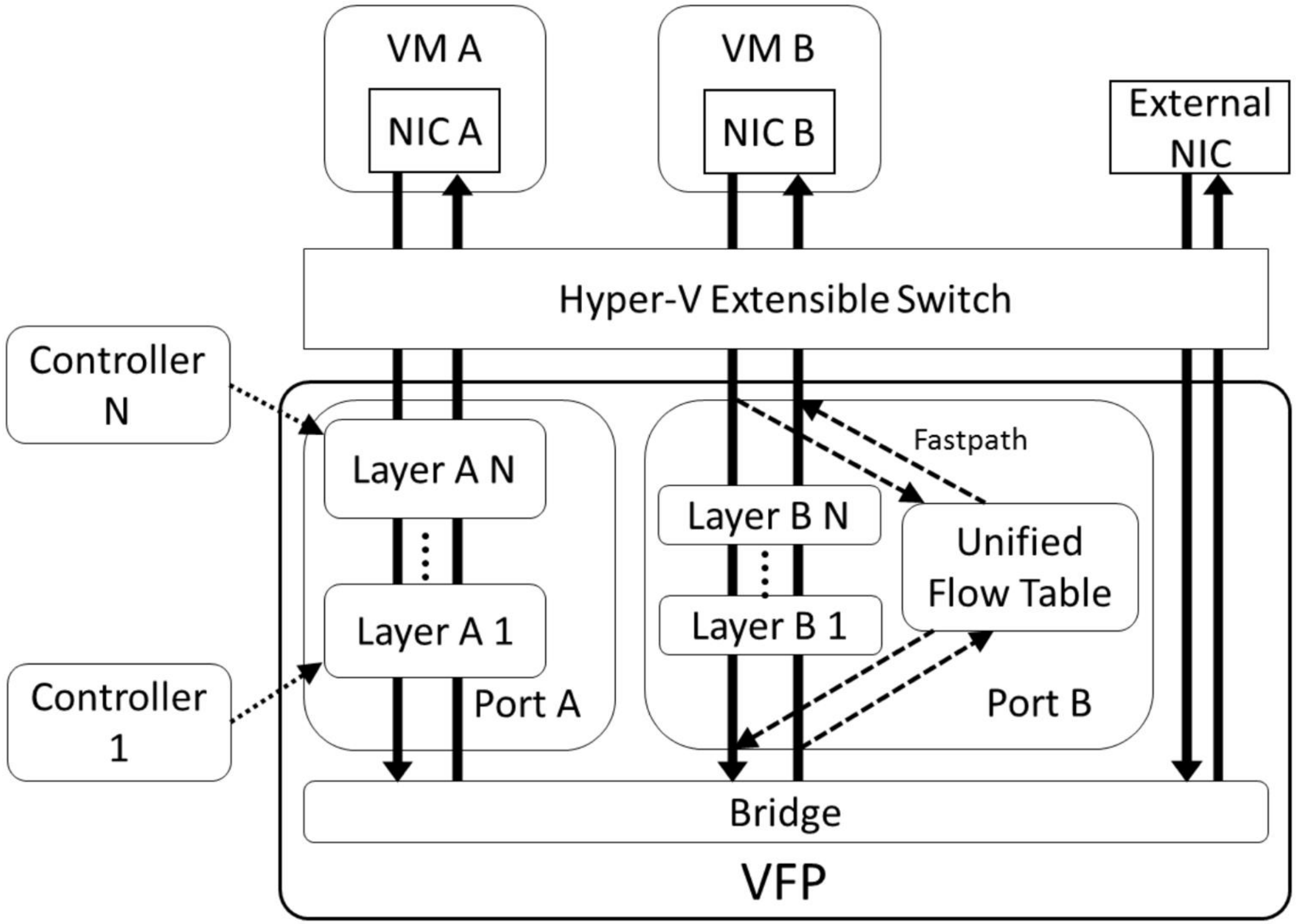
Address virtualization
(separation of tenant and provider addresses through tunneling)

Effect: Move packets from source vNIC to dest vNIC

Performance: Caching

Example 2: Azure VFP

- Tenants use CA-space addresses
- One big switch
- Multiple controllers, each programming distinct layer(s)
- Layer implements a part of the policy: NAT, etc.
- The TEP itself is a MAT
- Stateful actions (e.g. NAT) are first-class citizens
- Unified flow tables (caching)

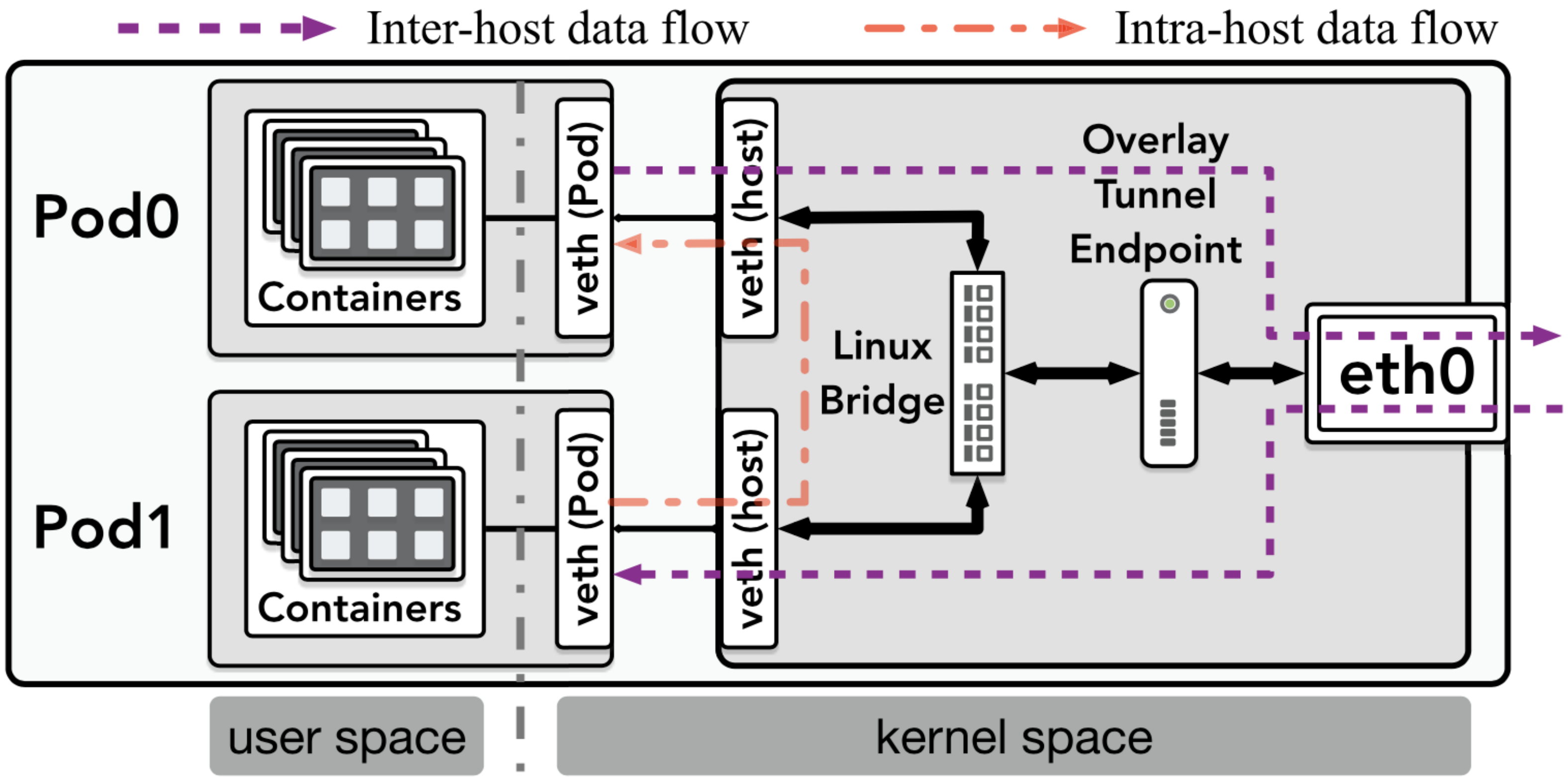


Example 3: Kubernetes/CNIs

- Container Network Interfaces: configuring networking for inter-pod networking
  - Within a pod, use loopback interface (e.g. service mesh)
- Pods use CA-space addresses (overlay); but PA also possible (underlay)
- Topology virtualization: If CA, TEP configured through
  - In-kernel forwarding (L3 forwarding tables, netfilter, iptables)
  - Bridging
  - Tun/tap software interface
  - eBPF
- Can use either L2 or L3 networking to interconnect CAs

Example 3: Kubernetes/CNIs

- Example with L2+L3 overlay



Making old software use new machines usually means making new machines behave like old ones.

(also applies when “machines” substituted by “networks”)