# Lecture 2:
# Cloud Computing

Data Center Network Architecture

# Cloud Computing, Common Needs

- Elasticity
  - Resources only when needed, on-demand utility/pay-per-use
- Multi-tenancy
  - Independent users result in need for security and isolation, while shring and amortizing costs for efficiency.
- Resilience and  Manageability
  - Isolate failures, replication and migration.

# Service Models

- Infrastructure as a Service (IaaS)
  - Supply virtualized resources, e.g. S3 or EC2
- Platform as a service (PaaS)
  - Let users focus on what they are delivering, not the resources needed to deliver it, e.g. Google's AppEngine, Salesforce, Microsoft Azure, Amazon Web Services
- Software as a Service (SaaS)
  - Software is delivered via the cloud, e.g. fee for use. Consider GoogleApps, Microsoft 360, etc.

# DC Networking vs Long Haul Network

- Long haul: Speed of light is significant source of latency, unlikely to be resolved soon
  - With a DC, distances are short, the speed of light is less of a factor
  - Serialization/Deserialization delay (bits/second) is a factor
- Long haul: Shortest path is not uncommon and likely best
  - Within a DC, tree-like topologies can concentrate traffic, generating bottle necks
- DCs have unique concerns: Multiple tenants and elastic demand, resulting in need for security and efficient provisioning and security.

# From Many Comes One

- Scaling, Robustness, Availability improved when service is provide by many aggregating interchangeable parts, rather than queuing for one, or a small number, of whole providers
  - Easier to add more parts
  - Easier to service parts
  - Easier to dial up and dial down

# Virtualization is Key Enabler

- Use independent of physical resources
- Multiple users per machine
- Isolation
- Migration
- Virtual Heterogeneity or Homogeneity
- Virtual CPUs, storage volumes, networks, etc.

# Tiered Architecture

- The old standby
- Redundant “core” routers connect DC to Internet
- “Aggregation” layer cross-connects to core routers and “Access” switches
  - Hence Levels 2 and 3
- “Access layer”, such as top of rack, connects to servers and cross-connects to “Aggregation” layer
  - Layer 2

# Tiered Architecture Attributes

- Redundancy at the top
- Massive scale at the bottom
- Manageable complexity
- Fail soft for most important, higher level resources

# Tiered Architecture Limitations

- Higher up gets over-subscribed since everything passes through
- Over-subscription increases with scale
- Most important paths are most over-subscribed

# Level-2 vs Level-3

- Level 2 is faster, cheaper, and easier to manage
- Level 3 scales better, because it is hierarchical
- Level 3 can manage potential congestion through routing
  - Multipath routing via equal cost multipath

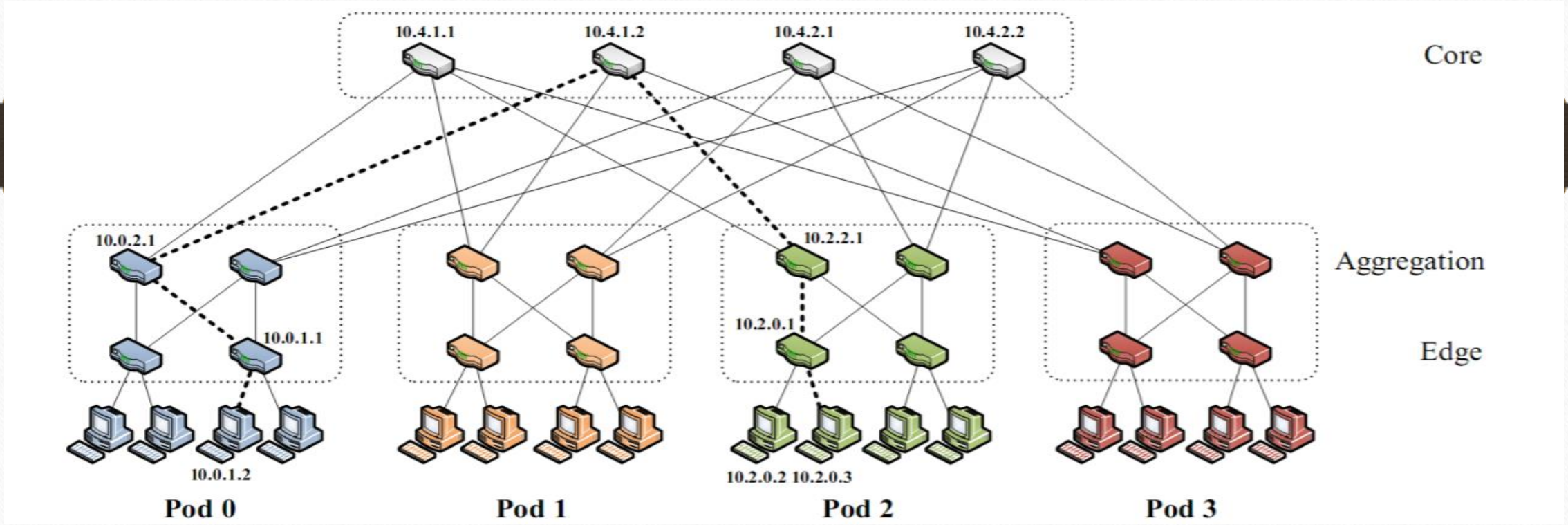
# Fat Tree

- Each layer has the same aggregate bandwidth
- Communication within a pod stays within a pod
- Need to be careful about purely equal-cost paths or there will be reordering
- Pre-bakes the paths to ensure diversity, while maintaining ordering

# Fat Tree Details

- K-ary fat free: three layers (core, aggregation, edge)
- Each pod consists of $(K/2)^2$ servers and 2 layers of $K/2$ K-port switches.
- Each edge switch connects $(K/2)$ servers to $(K/2)$ aggregator switches
- Each aggregator switch connects $(K/2)$ edge and $(K/2)$ core switches
- $(K/2)^2$ core switches, each ultimately connecting to K pods
  - Providing K different roots, not 1. Trick is to pick different ones
- K-port switches support $K^3/4$ servers/host:
    - (K/2 hosts/switch * K/2 switches per pod * K pods)

# Fat Tree


10.4.1.1
10.4.1.2
10.4.2.1
10.4.2.2
Core
10.0.2.1
10.0.1.1
10.0.1.2
10.2.2.1
10.2.0.1
10.2.0.2 10.2.0.3
Aggregation
Edge
Pod 0
Pod 1
Pod 2
Pod 3

# Path Diversity

- Path diversity is good, in that it distributed hot spots, enabling, for example, equal cost.
  - But, if equal cost is only goal, there could be a lot or reordering. Ouch!
- Fat Tree uses a special IP addressing and forwarding scheme to address this (pardon the pun), and additionally, allow intra-pod traffic to stay intra-pod

# “Special IP” Addressing

- “10.0.0.0/8” private addresses
- Pod-level uses “10.*pod.switch*.1“
  - pod,switch < *K*
- Core-level uses "10.*K.j.i*“
  - *K* is the same *K* as elsewhere, the number of ports/switch
  - View cores as logical square. *i, j* denote position in square.
- Hosts use “10.*pod.switch.ID*" addresses
  - 2 <= ID <= (K/2)
  - K=1 is pod-level switch; ID > 2 is too many hosts
  - 8-bits implies *K* < 256

# Static Routing Via Two-Level Lookups

- First level is prefix lookup
  - Used to route down the topology to end host
- Second level is a suffix lookup
  - Used to route up towards core
- Diffuses and spreads out traffic
- Maintains packet ordering by using the same ports for the same endhost

# Lookup Tables

| Prefix | Output port |
|---|---|
| 10.2.0.0/24 | 0 |
| 10.2.1.0/24 | 1 |
| 0.0.0.0/0 | |

| Suffix | Output port |
|---|---|
| 0.0.0.2/8 | 2 |
| 0.0.0.3/8 | 3 |

# Flow Classification

- Type of "diffusion optimization"
- Mitigate local congestion
- Assign traffic to ports based upon flow, not host.
  - One host can have many flows, thus many assigned routings
- Fairly distribute flows
- $(K/2)^2$ shortest paths available – but doesn't help if all pick same one, e.g. same root of multi-rooted tree
- Periodically reassign output port to free up corresponding input port

# Flow Scheduling

- Also a “Diffusion Optimization”
- Detect and deconflict large, long-lived flows
  - Threshholds for throughput and longevity

# Fault tolerance

- Can reject flows by setting cost to infinity.
- Plenty of paths, if failure
- Maintain sessions across switches to monitor