Data Centers

Tom Anderson

Transport Clarification

- RPC messages can be arbitrary size
  - Ex: ok to send a tree or a hash table
  - Can require more than one packet sent/received
- We assume messages can be dropped, duplicated, reordered
  - It is packets that are dropped, duplicated, reordered
  - Most RPCs are single packets
- TCP addresses some of these issues
  - But we'll assume the general case

RPC Semantics

- At least once (NFS, DNS, lab 1b)
  - true: executed at least once
  - false: maybe executed, maybe multiple times
- At most once (lab 1c)
  - true: executed once
  - false: maybe executed, but never more than once
- Exactly once
  - true: executed once
  - false: never returns false

When does at-least-once work?

- If no side effects
  - read-only operations (or idempotent ops)
- Example: MapReduce
- Example: NFS
  - readFileBlock (vs. Posix file read)
  - writeFileBlock
  - (but not: delete file)

At Most Once

Client includes unique ID (UID) with each request

- use same UID for re-send

Server RPC code detects duplicate requests

- return previous reply instead of re-running handler

```
if seen[uid] {
    r = old[uid]
} else {
    r = handler()
    old[uid] = r
    seen[uid] = true
}
```

Some At-Most-Once Issues

How do we ensure UID is unique?

- Big random number?
- Combine unique client ID (IP address?) with seq #?
- What if client crashes and restarts? Can it reuse the same UID?

- In labs, nodes never restart
  - Equivalent to: every node gets new ID on start
  - Client address + seq # is unique
  - Seq # alone is NOT unique

When Can Server Discard Old RPCs?

Option 1: Never?

Option 2: unique client IDs

- per-client RPC sequence numbers
- client includes "seen all replies <= X" with every RPC

Option 3: one client RPC at a time

- arrival of seq+1 allows server to discard all <= seq

Labs use Option 3

What if Server Crashes?

If at-most-once list of recent RPC results is stored in memory, server will forget and accept duplicate requests when it reboots

- Does server need to write the recent RPC results to disk?
- If replicated, does replica also need to store recent RPC results?

In Labs, server gets new address on restart

- Client messages aren't delivered to restarted server
- (discard if sent to wrong version of server)

A Data Center

Inside a Data Center

Data center

10k - 100k servers: 250k − 10M cores

1-100PB of DRAM

100PB - 10EB storage

1- 10 Pbps bandwidth (>> Internet)

10-100MW power

- 1-2% of global energy consumption

100s of millions of dollars

Servers

Limits driven by the power consumption

1-4 multicore sockets

20-24 cores/socket (150W each)

100s GB – 1 TB of DRAM (100-500W)

40Gbps link to network switch

Servers in racks

19” wide

1.75” tall (1u)

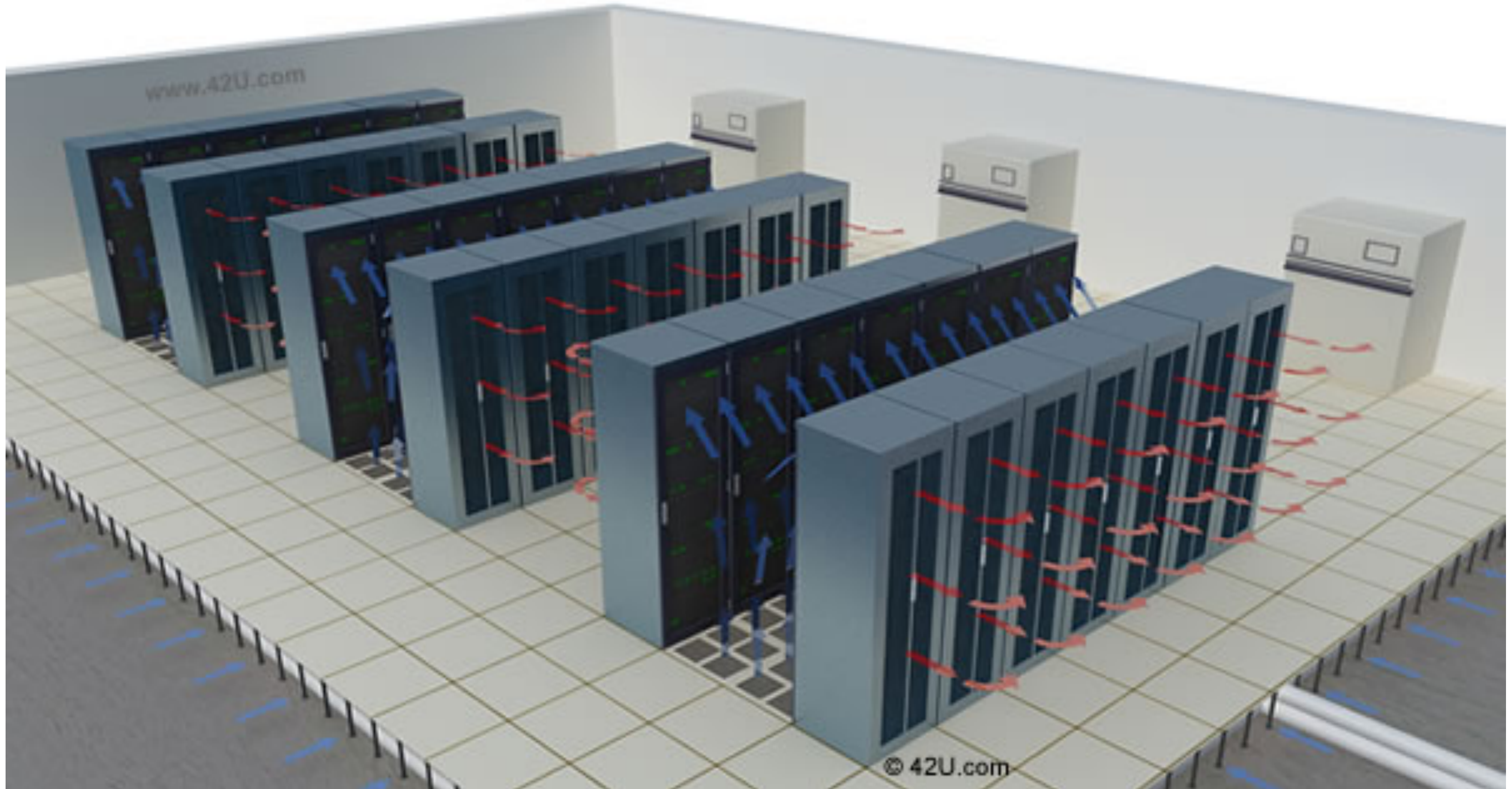
(defined in 1922!)

40-120 servers/rack

network switch at top

Racks in rows

Rows in hot/cold pairs



Hot/cold pairs in data centers

Where is the cloud?

Amazon, in the US:

- Northern Virginia
- Ohio
- Oregon
- Northern California

Why those locations?

MTTF/MTTR

Mean Time to Failure/Mean Time to Repair

Disk failures (not reboots) per year ~ 2-4%

- At data center scale, that's about 2/hour.
- It takes 10 hours to restore a 10TB disk

Server crashes

- 1/month * 30 seconds to reboot => 5 mins/year
- 100K+ servers

Typical Year in a Data Center (2008)

- ~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
- ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
- ~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
- ~1 network rewiring (rolling ~5% of machines down over 2-day span)
- ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

Typical Year in a Data Center (2008)

- ~5 racks go wonky (40-80 machines see 50% packetloss)
- ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
- ~12 router reloads (takes out DNS and external vips for a couple minutes)
- ~3 router failures (have to immediately pull traffic for an hour)
- ~dozens of minor 30-second blips for dns
- ~1000 individual machine failures
- ~thousands of hard drive failures
- slow disks, bad memory, misconfigured machines, flaky machines, etc

This Week: Primary Backup

- How do we build systems that survive a single node failure?
- State replication: run two copies of server
  - primary, backup
  - different racks, different PDUs, different DCs!
  - If primary fails, backup takes over
- How do we know primary failed vs. is slow?

Data Center Networks

Every server wired to a ToR (top of rack) switch

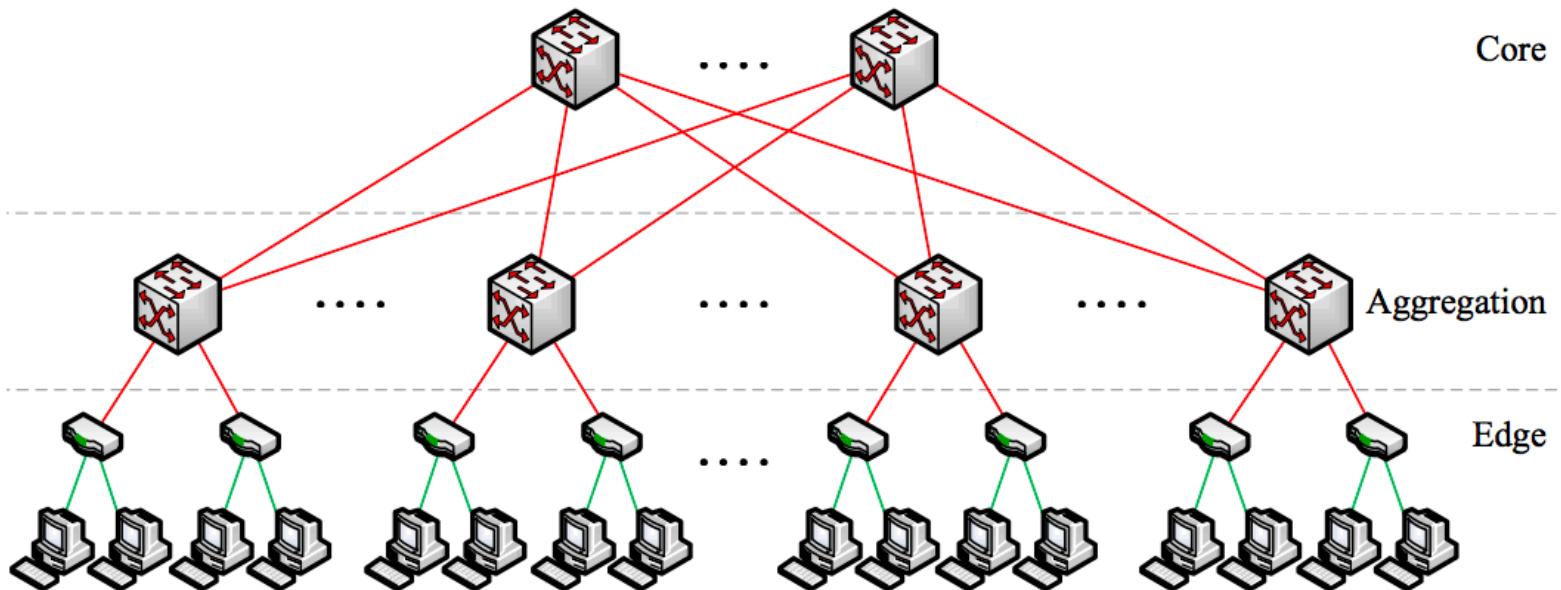
ToR's in neighboring aisles wired to an aggregation switch

Agg. switches wired to core switches

Early data center networks

3 layers of switches

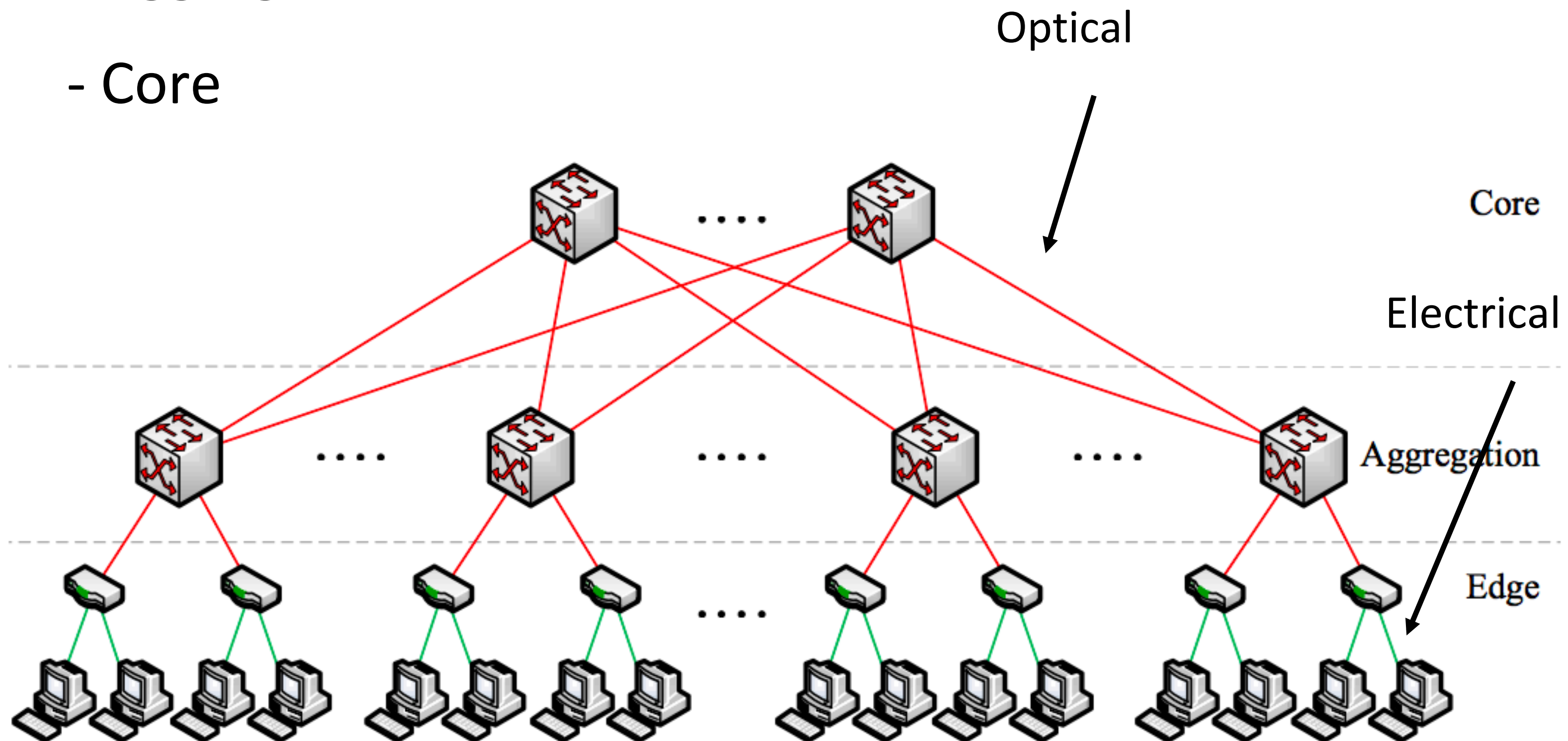
- Edge (ToR)
- Aggregation
- Core



Early data center networks

3 layers of switches

- Edge (ToR)
- Aggregation
- Core



Early data center limitations

Cost

- Core, aggregation routers = high capacity, low volume
- Expensive!

Fault-tolerance

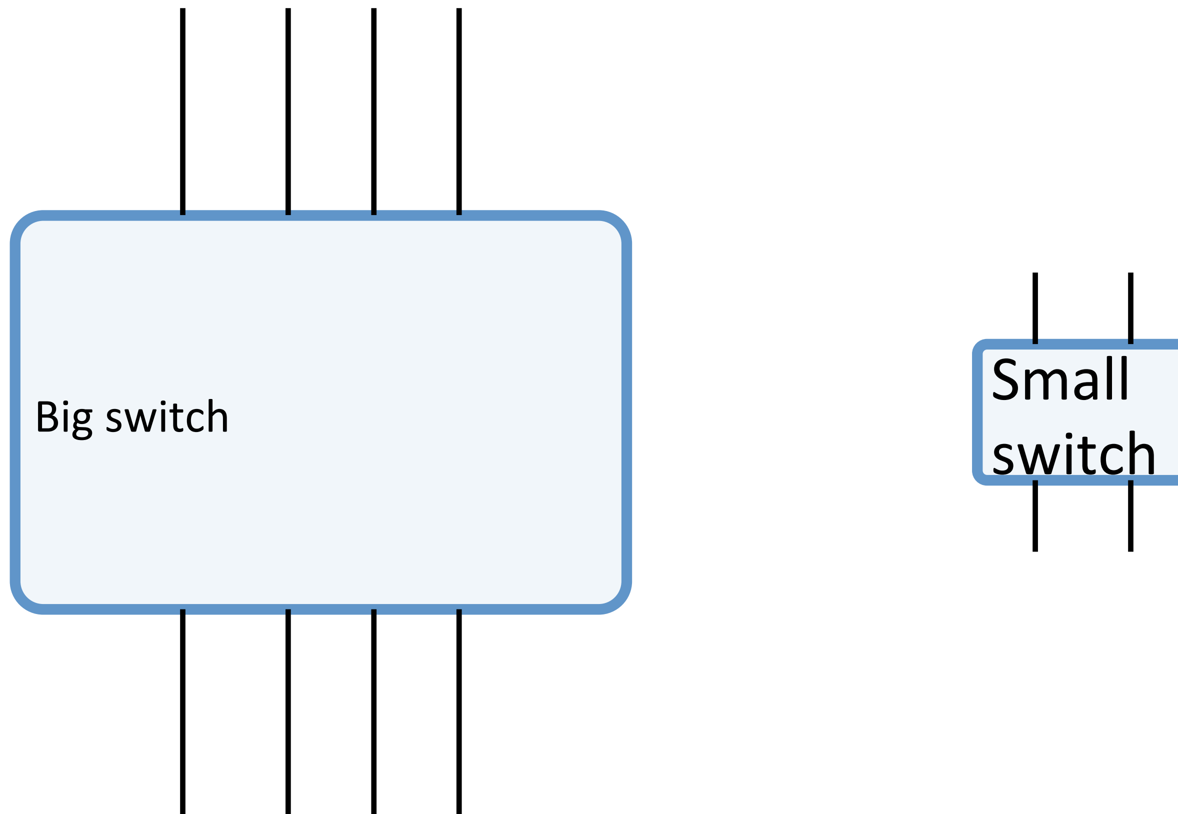
- Failure of a single core or aggregation router = large bandwidth loss

Bisection bandwidth limited by capacity of largest available router

- Google's DC traffic doubles every year!

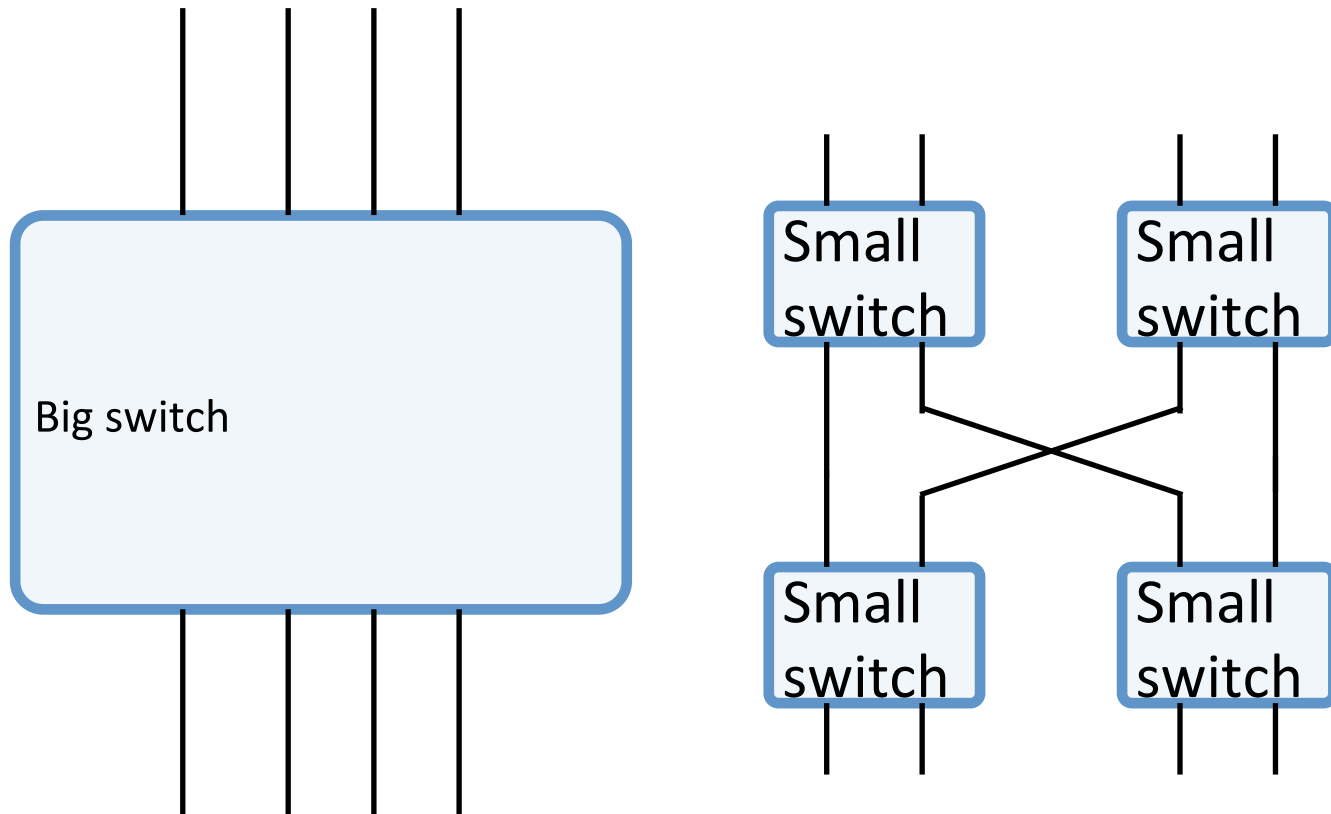
Clos networks

How can I replace a big switch by many small switches?



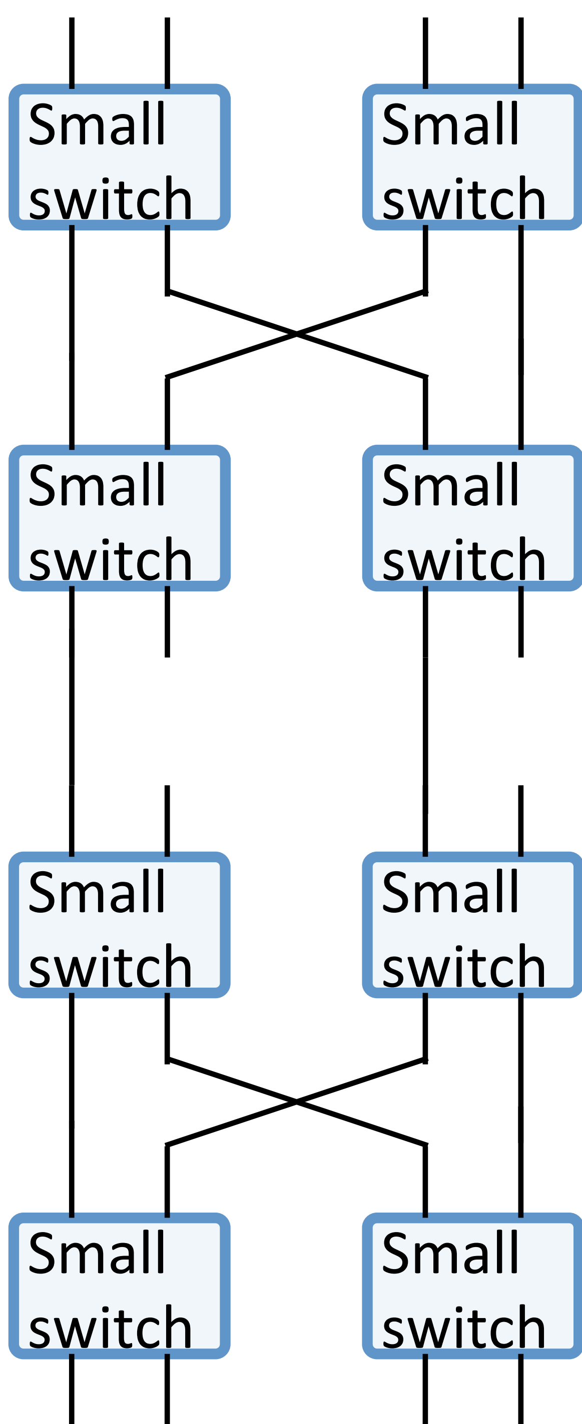
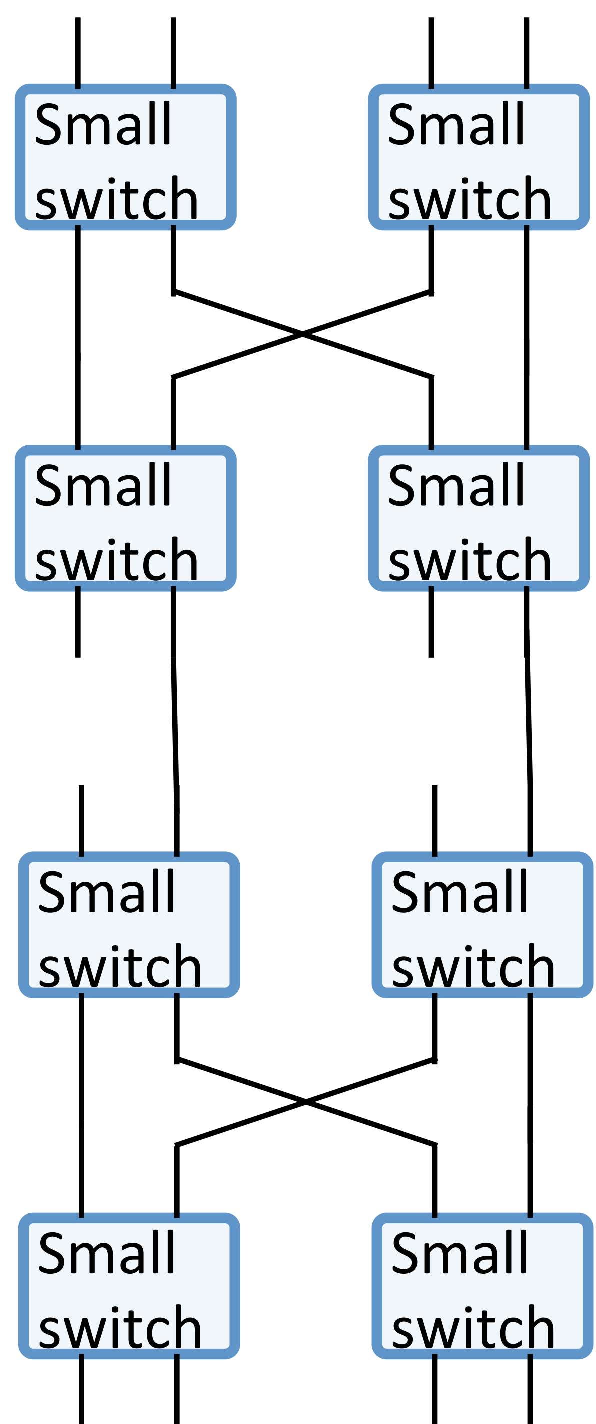
Clos networks

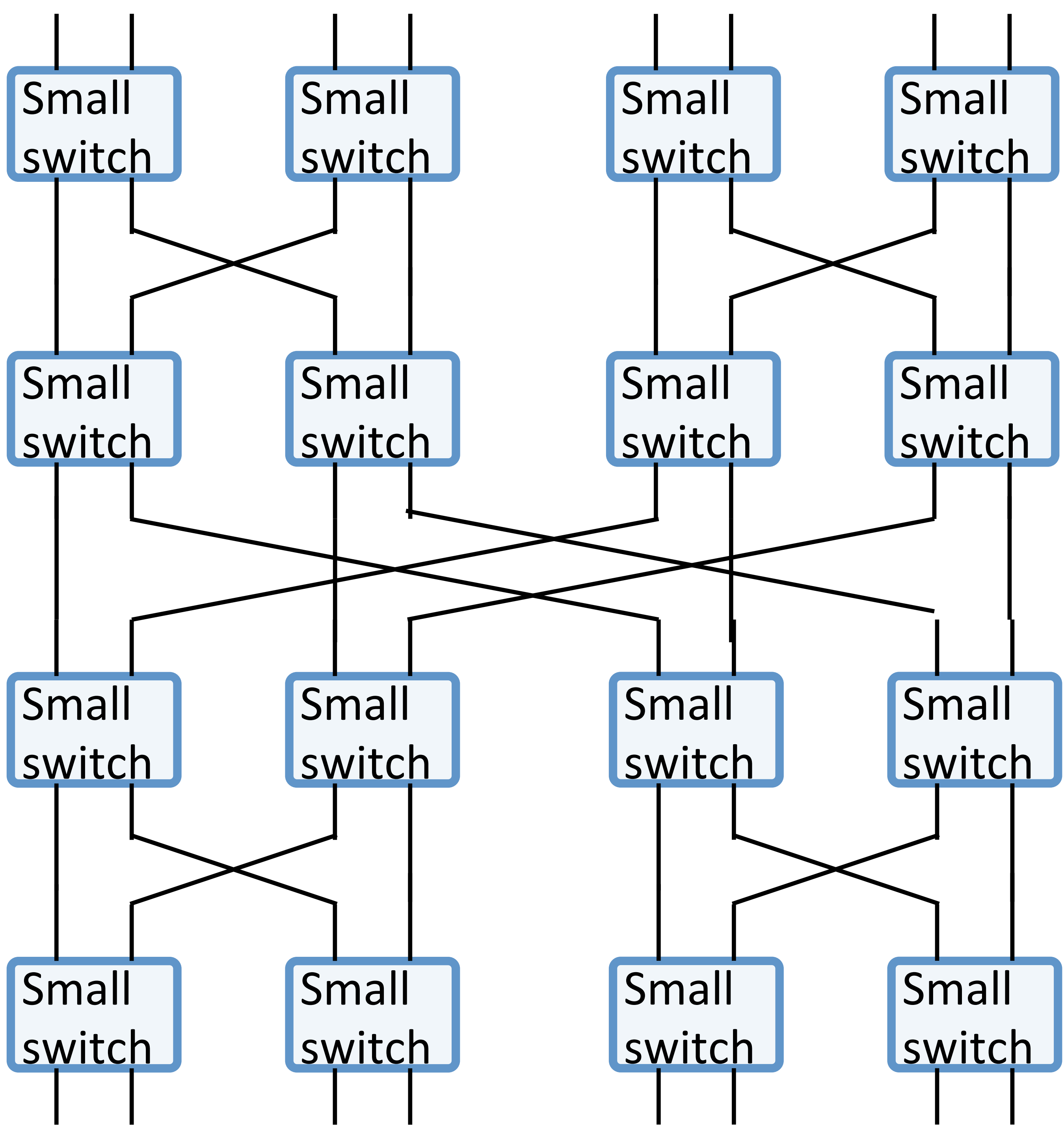
How can I replace a big switch by many small switches?



Clos Networks

What about bigger switches?


Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch

Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch


Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch
Small switch

Multi-rooted tree

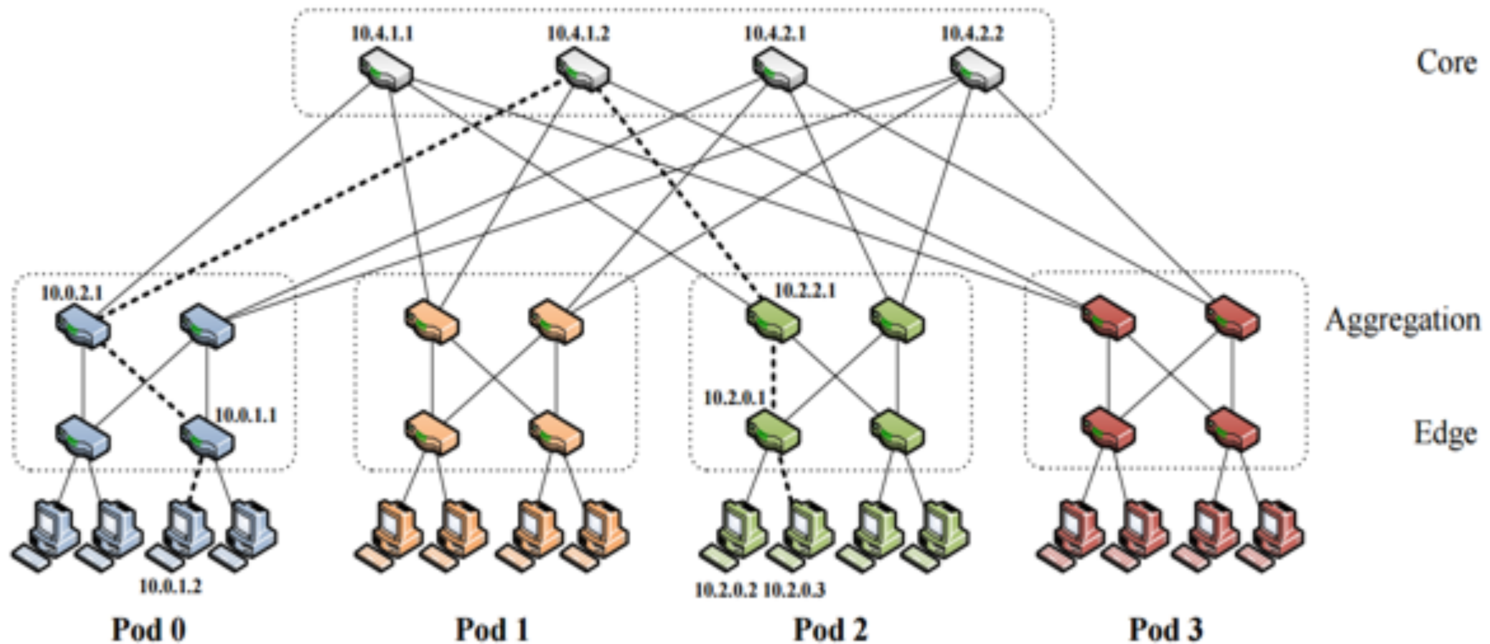


Figure 3: Simple fat-tree topology. Using the two-level routing tables described in Section 3.3, packets from source 10.0.1.2 to destination 10.2.0.3 would take the dashed path.

Every pair of nodes has many paths

Fault tolerant! But how do we pick a path?

Multipath routing

Lots of bandwidth, split across many paths

ECMP: hash on packet header to determine route

- (5 tuple): Source IP, port, destination IP, port, prot.
- Packets from client − server usually take same route

On switch or link failure, ECMP sends subsequent packets along a different route

=> Out of order packets!

Data Center Network Trends

RT latency across data center ~ 10 usec

40 Gbps links common, 100 Gbps on the way

- 1KB packet every 80ns on a 100Gbps link
- Direct delivery into the on-chip cache (DDIO)

Upper levels of tree are (expensive) optical links

- Thin tree to reduce costs

Within rack > within aisle > within DC > cross DC

- Latency and bandwidth: keep communication local

Local Storage

- Magnetic disks for long term storage
  - High latency (10ms), low bandwidth (250MB/s)
  - Compressed and replicated for cost, resilience
- Solid state storage for persistence, cache layer
  - 50us block access, multi-GB/s bandwidth
- Emerging NVM
  - Low energy DRAM replacement
  - Sub-microsecond persistence

Day in the Life of a Web Request

User types “google.com”, what happens

DNS = Domain Name System

- Global database of name->IP address

DNS query to root name server

- Root name server is replicated (600+)
- Hard coded IP multicast addresses
- Updates happen async in background (rare)

Root returns: IP address of google’s name server

- Cached on client (for a configurable period)
- Can be out of date

Name Server to DC

- DNS returns google's name server
- Google name server returns local name server
  - Name server in DC close to user's IP address
- Local name server returns IP of web front end
  - Also in local data center
  - With short timeout, reroute if front end failure
- Browser sends HTTP request to front end

Resilient Scalable Web Front End

- IP address of front end is an array of machines
- Packets first go through load balancer
  - Actually, an array of load balancers (all the same)
- Load balancer hashes 3 or 5 tuple -> front end
  - Source IP, port #, Destination IP, port #, protocol
  - This way, all traffic from user goes to same server
- When front end fails, load balancer redirects incoming packets elsewhere

Challenge

How do we build a hash function that's stable?

That reroutes only the packets for the failed node, leaving the packets to other nodes alone.

Web Front End -> Cache, Storage Layer

Key-value store

Sharded across many cache and storage nodes

Hash(key) -> cache, storage node

If cache node fails, rehash to new cache node

If storage node fails, ???