Load Management

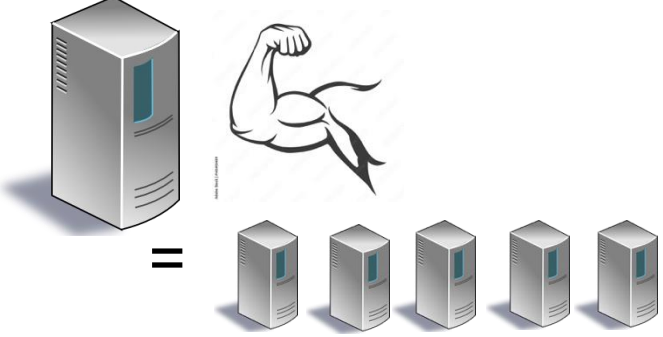
Lecture 14

Srinivas Narayana

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

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Distribution & its consequences



- Internet services use replication to tackle machine failures and unavailability
  - So far: Data replication: consistency
- But compute also requires replication
- Compute replication is critical for performance
  - Geo-replication (e.g., failover and balance across metro regions)
  - Replication within a cluster (e.g., 100s of processes serving the same app within a cluster)

Load management

- Global: How to direct user load across clusters?
  - Key performance considerations
  - Query traffic: Low latency
  - Data uploads: High throughput
- Local: Within a cluster, how to manage the load?
  - Machines within a cluster are presumably similar to each other
  - Key: Avoiding hotspots and reducing overprovisioning

Global load balancing

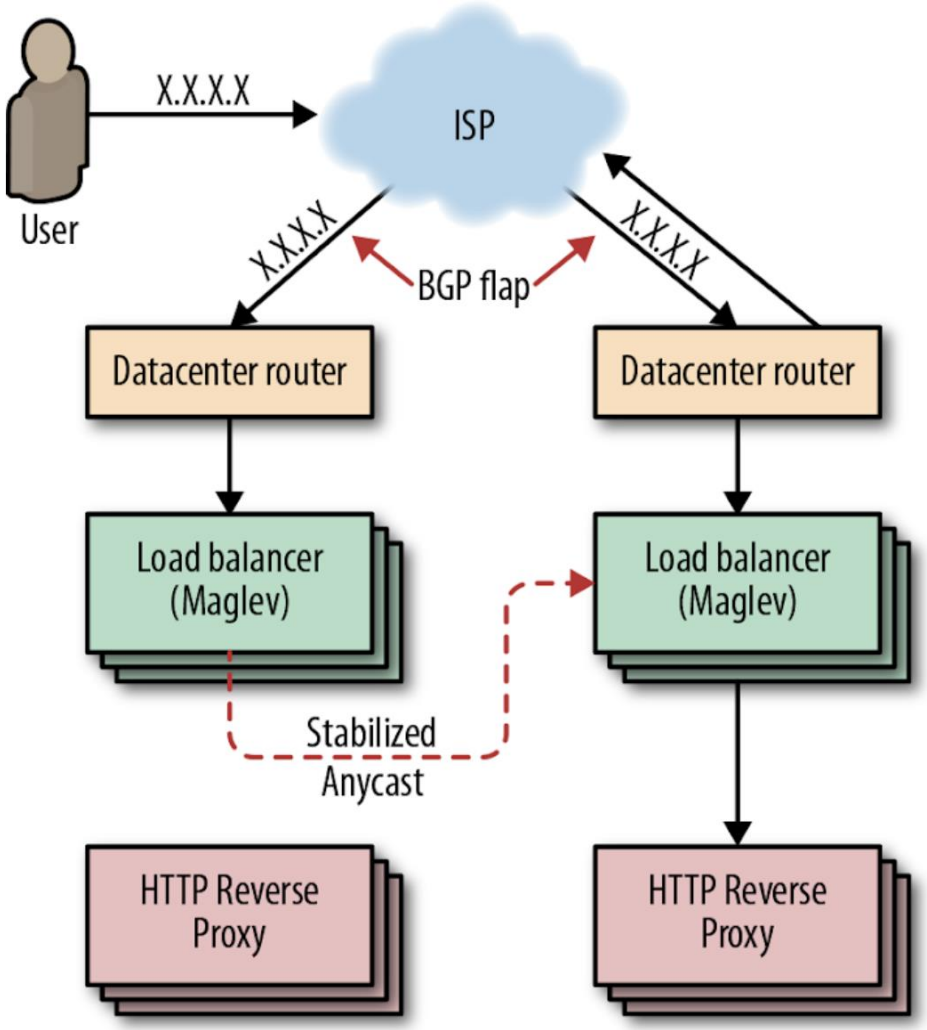
- Primary mechanism: DNS
- Use IP anycast to talk to “nearest” (acc to BGP) authoritative DNS servers. Auth servers redirect user to location/server closest to them through a single DNS response.
- Problem: clients rarely talk directly to auth DNS server (go through recursive resolvers). Resolvers hide client count and geo-diversity. They also cache responses.
- Mitigations: estimated users and geo-diversity behind resolvers. Issue low TTL responses (adds latency)
- DNS response sizes are bounded. Client will choose randomly from among responses; don’t know who is closest.

Virtual IP addresses per datacenter

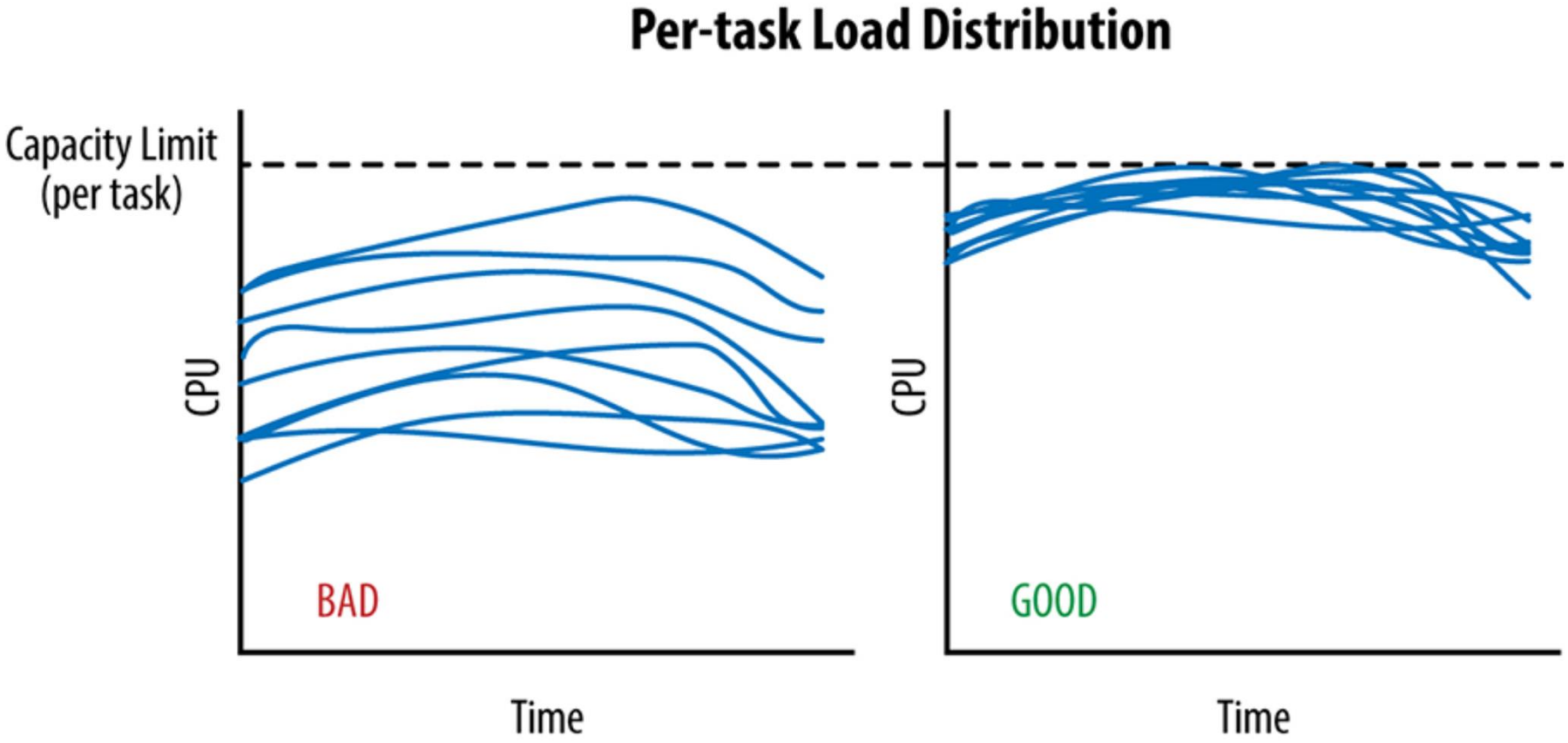
- Use a virtual IP address (VIP) to cover many real IP addresses
  - Hide growth, failures, maintenance in server pool from users
  - Use DNS with large TTL. Save latency.
  - Effectively decouple cluster-external from internal
- Alternative: can we use IP anycast directly to get to edge?
  - But anycast need not be stable! BGP route flaps
  - Send to a different edge at any time, even in the middle of a connection

Frontend load balancing

- Layer-4 load balancers spray connections across HTTP reverse proxies
- Reverse proxy terminates TCP/TLS and re-encrypts to backends. Maintain persistent connections to backends
- Terminate TCP/TLS as close to the user as possible
- ECMP: easily add more L4 LBs to pool
- Stabilize anycast through consistent hashing. Cannot rely on connection state shared across L4 LBs



Maintaining even load



Uneven load == stranded resources

Solution == spray requests across processes?

Problem: Statefulness

- A user's connection may have to reach the same machine (e.g., reverse proxy, or backend server)
- Choosing a different reverse proxy will break TCP connections.
- Having to estasblish new connections will degrade the performance advantages of reverse proxying in the first place
- Backend process may have additional state: e.g., HTTP cookies, or other local execution state per connection

Connection tracking and consistent hashing at the L4 LB

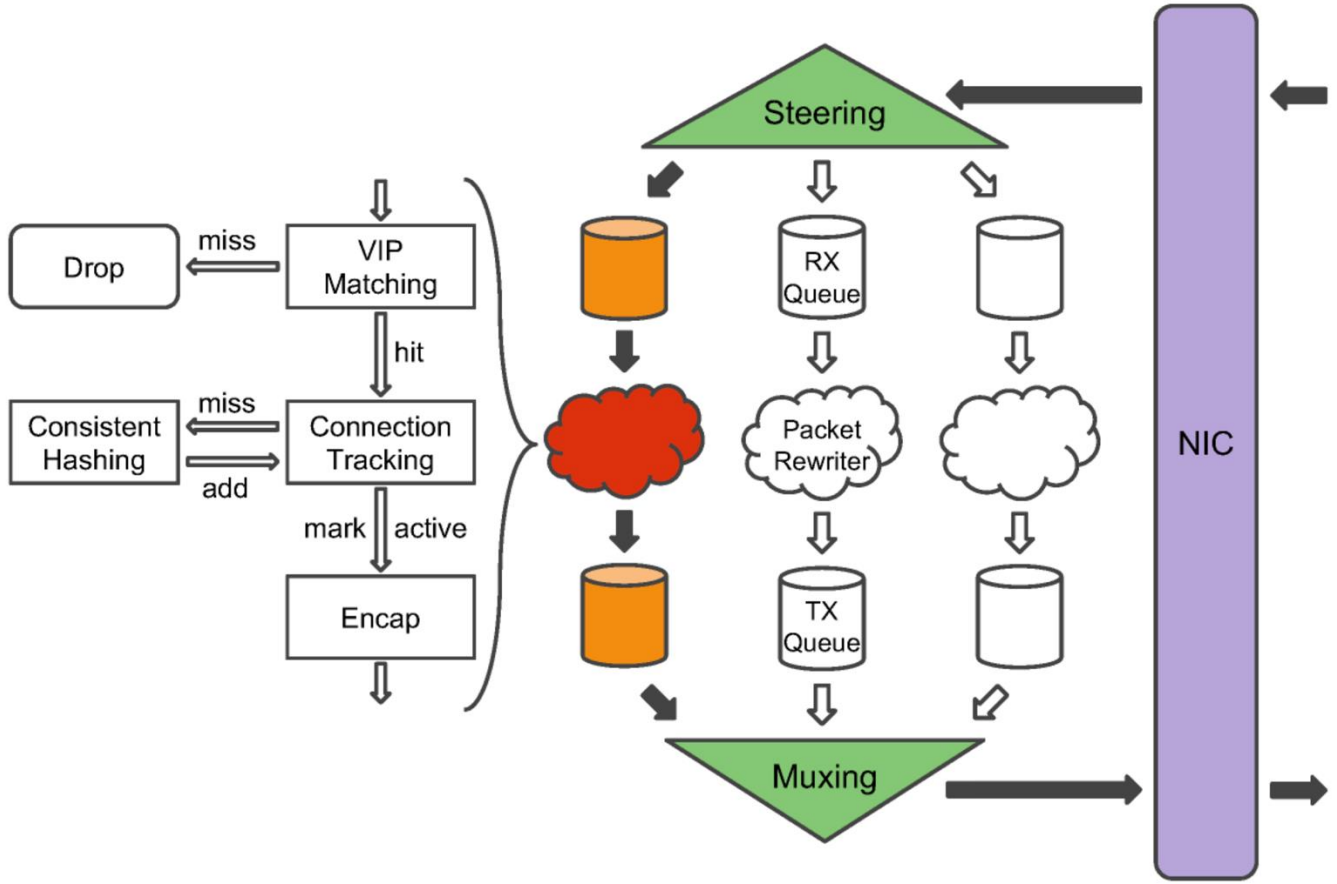
- Remembering connections by putting them in a connection tracking table: 5-tuple → backend
  - Not always possible
  - Even the load balancer forwarding a packet may change mid-connection
  - SYN floods and crowds may overwhelm connection tracking table
- If a packet's connection cannot be found in the connection, use a hash function h(packet) to determine the backend
  - Naïve choices: break connection when proxy pool changes
  - Need consistent hashing: even if the backends change, the backends for existing connections should be minimally disrupted

Maglev (L4 LB) forwarder

Multi-threaded (parallelism)

Don’t share state across threads. Each 5-tuple steered to a core.

Connection tracking table is local to the core



Hash table population

$$offset \leftarrow h_1(name[i]) \mod M$$

$$skip \leftarrow h_2(name[i]) \mod (M-1)+1$$

$$permutation[i][j] \leftarrow (offset + j \times skip) \mod M$$

Each backend now uses a random (unique) permutation

Backends choose slots based on the permutation.

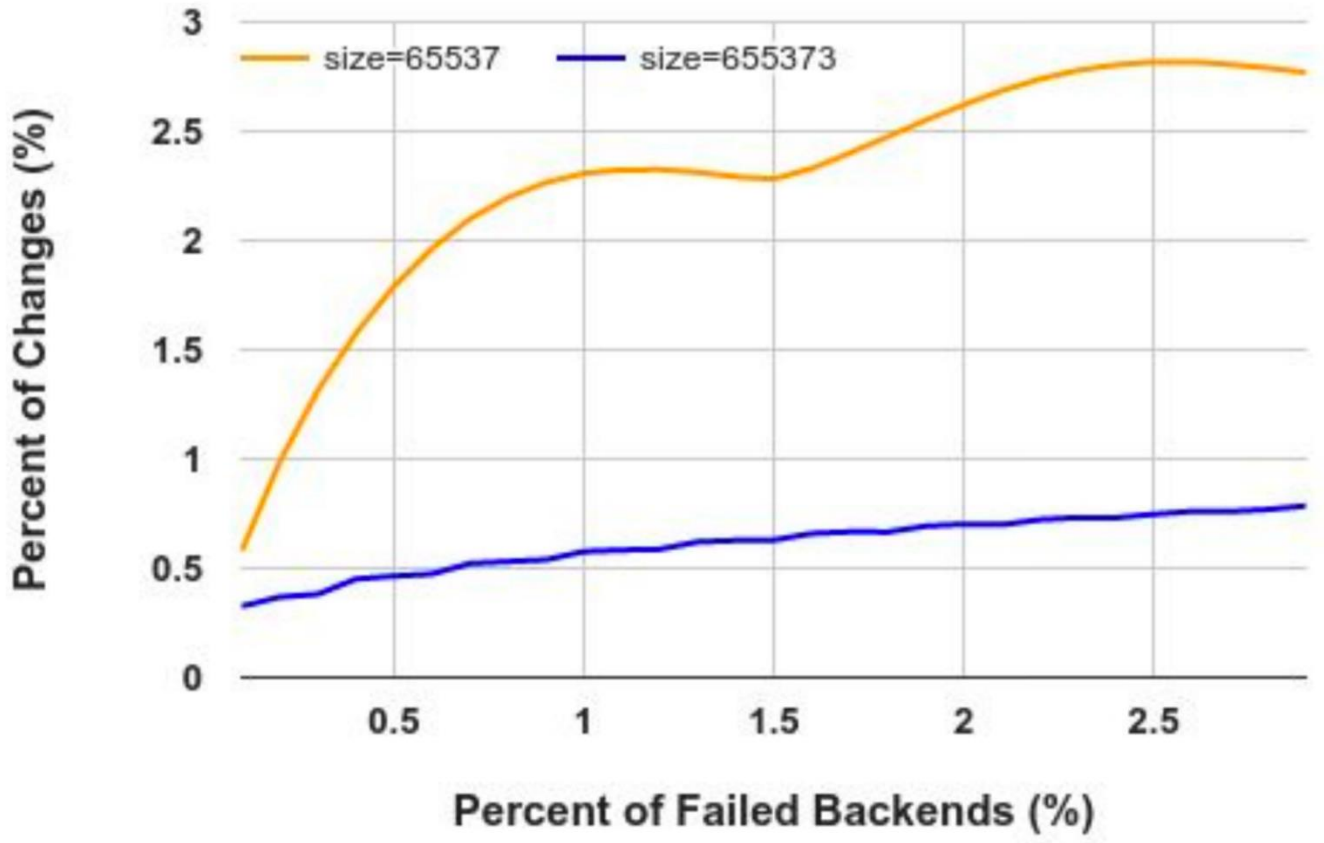
Pseudocode 1 Populate Maglev hashing lookup table.

```
1:  function POPULATE
2:      for each i < N do next[i] ← 0 end for
3:      for each j < M do entry[j] ← −1 end for
4:      n ← 0
5:      while true do
6:          for each i < N do
7:              c ← permutation[i][next[i]]
8:              while entry[c] ≥ 0 do
9:                  next[i] ← next[i] + 1
10:                 c ← permutation[i][next[i]]
11:             end while
12:             entry[c] ← i
13:             next[i] ← next[i] + 1
14:             n ← n + 1
15:             if n = M then return end if
16:         end for
17:     end while
18: end function
```

Actual packet forwarding

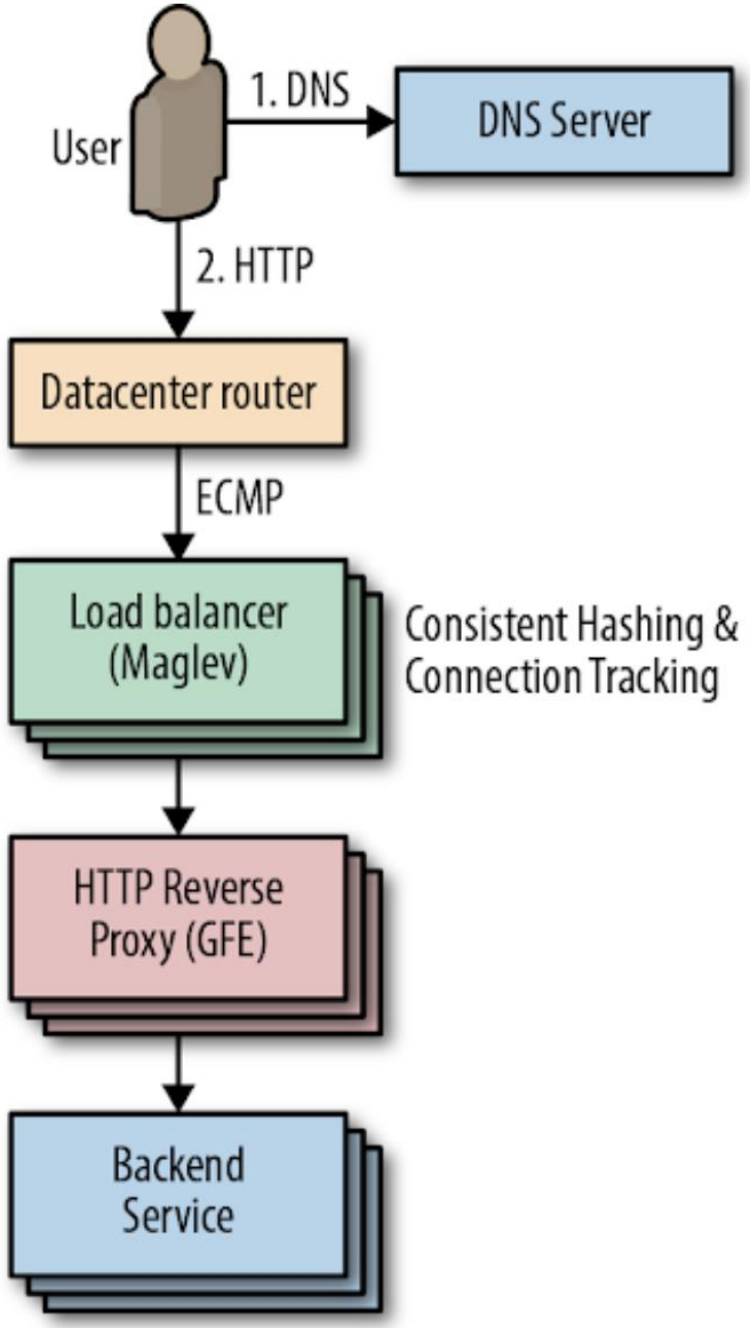
- (1): NAT tables: map incoming connections to outgoing
  - Stateful; large tables
- (2) Modify destination MAC address
  - Direct Server Return
  - But cannot have all machines in one L2 network
- (3) Encapsulation (e.g. GRE). If a route exists, it works.
  - Server will decapsulate the packet and use DSR
  - Inflate packet size and possibly cause fragmentation

Disruptions on lookup table change



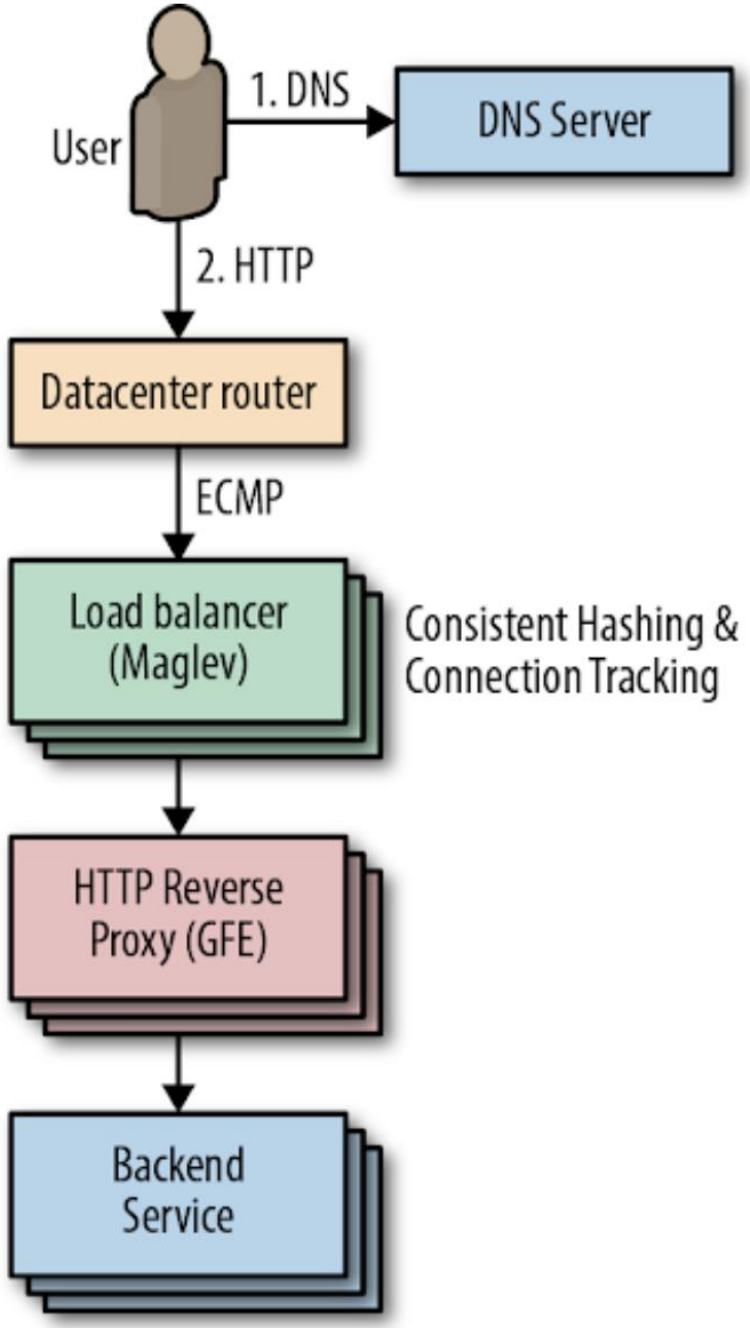
Beyond the reverse proxy

- Problem 1: avoid unhealthy backends first
- One strategy: Pick backend server with least outstanding requests. If too many outstanding requests, avoid those backends altogether
- Problems?
  - Only avoids extreme overload; backend can overload even before this limit is reached
  - Processing time of a request may be large, doesn't mean the server cannot take on more requests
- "Lame duck" state: a backend can proactively signal that it is unhealthy to avoid new connections, while finishing processing requests in flight



Beyond the reverse proxy

- Problem 2: choose among available healthy backends
  - Don’t maintain a connection to every backend: memory and CPU per connection
- Establish a new (backend) connection per (user) connection? Long lived?
- Connect to a subset of backends
  - How large?
  - Load variation from each reverse proxy
  - # backends >> #RPs
- Which backends of that size?
  - Random subsets unevenly load-balanced
  - Use deterministic subsets in time-based rounds



Strategies to choose backends

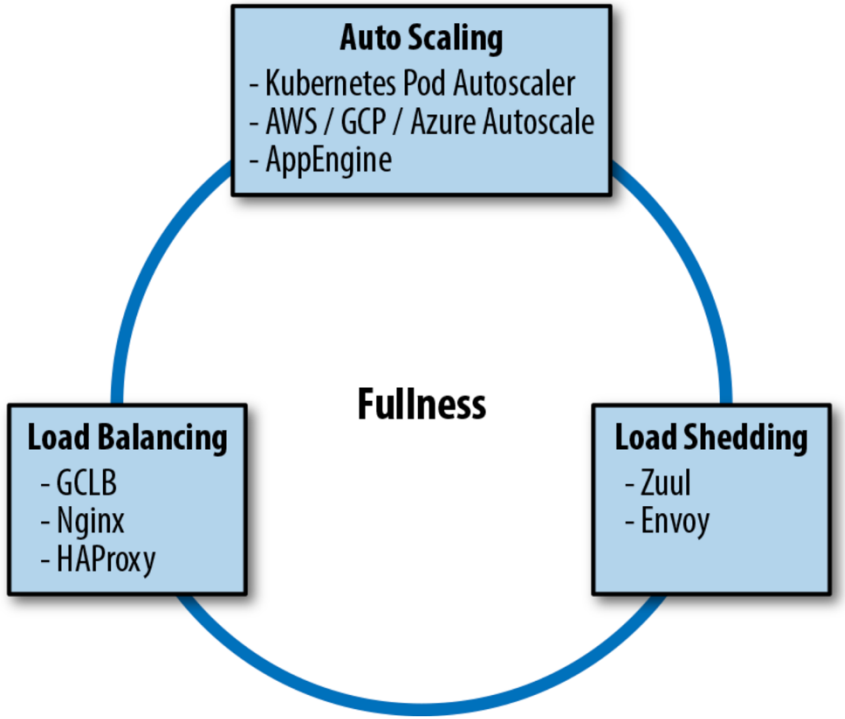
- Backend load and capacity agnostic: round robin. Insufficient
  - Small subsets: some clients heavier than others
  - Diversity in machine capacities (CPU architectures, speeds, cores)
  - Variation in work for each request (1000x). Hard to predict
  - Unpredictable performance changes (noisy neighbors, task restarts)
- Assign to least loaded backend? (currently active load)
  - Good: move load away from loaded backends
  - Bad: Typically considers load without regard to available capacity
  - Bad: Long-lived requests
  - Bad: per-RP view of load
- One approach: weighted (RR) splitting with load and error feedback from backends

Autoscaling

- Sometimes, you just don’t have enough capacity
- Vertical autoscaling
- Horizontal autoscaling
- Don’t just rely on server utilization metrics. For example, error codes returned very quickly have low CPU utilization
- Creating new instances is never instant
- Doesn’t always work:
  - Failure to do useful work but consuming resources
  - Overloading downstream dependencies by autoscaling upstream tier
  - Shared quotas across tiers: reason with dependencies carefully

Load shedding

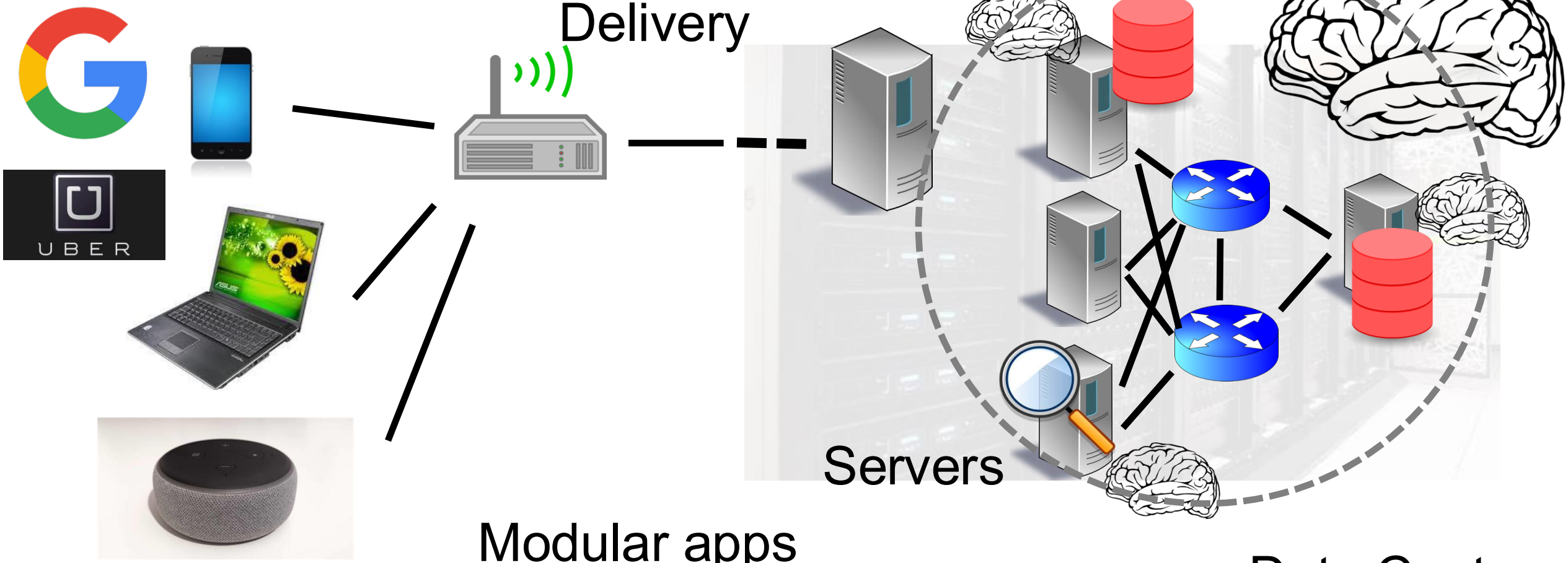
- Return errors upon high load; process what you can
- Combination of all techniques useful. But consider their interactions carefully



Internet Services

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Internet Services



Operations

- Monitoring, security
- Release engineering, canarying
- Crafting and maintaining SLOs
- People and processes
- Incident response, postmortems
- Designing and managing cluster configurations
- And many others

What next?

Thanks, and all the best!

- Live with a deeper appreciation of Internet services
  - How is content delivered? How are apps designed and what architectures may I use for my own? What does my platform look like? How do I manage critical data and achieve high performance?
- Put your knowledge to good use
  - This course: Programming homeworks; Project
  - Significant tech work builds on concepts from this course
  - Design your own Internet and data management services
  - Principles to organize your application
- Go deeper
  - Research projects, independent study, theses, …