Operations; Load Management

Lecture 12

Srinivas Narayana

http://www.cs.rutgers.edu/~sn624/553-S23

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Operations

- How to run and manage an Internet service?
- Monitoring, security
- Load management
- Release engineering, canarying
- Crafting and maintaining SLOs
- People and processes
- Incident response, postmortems
- Designing and managing configurations
- …

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

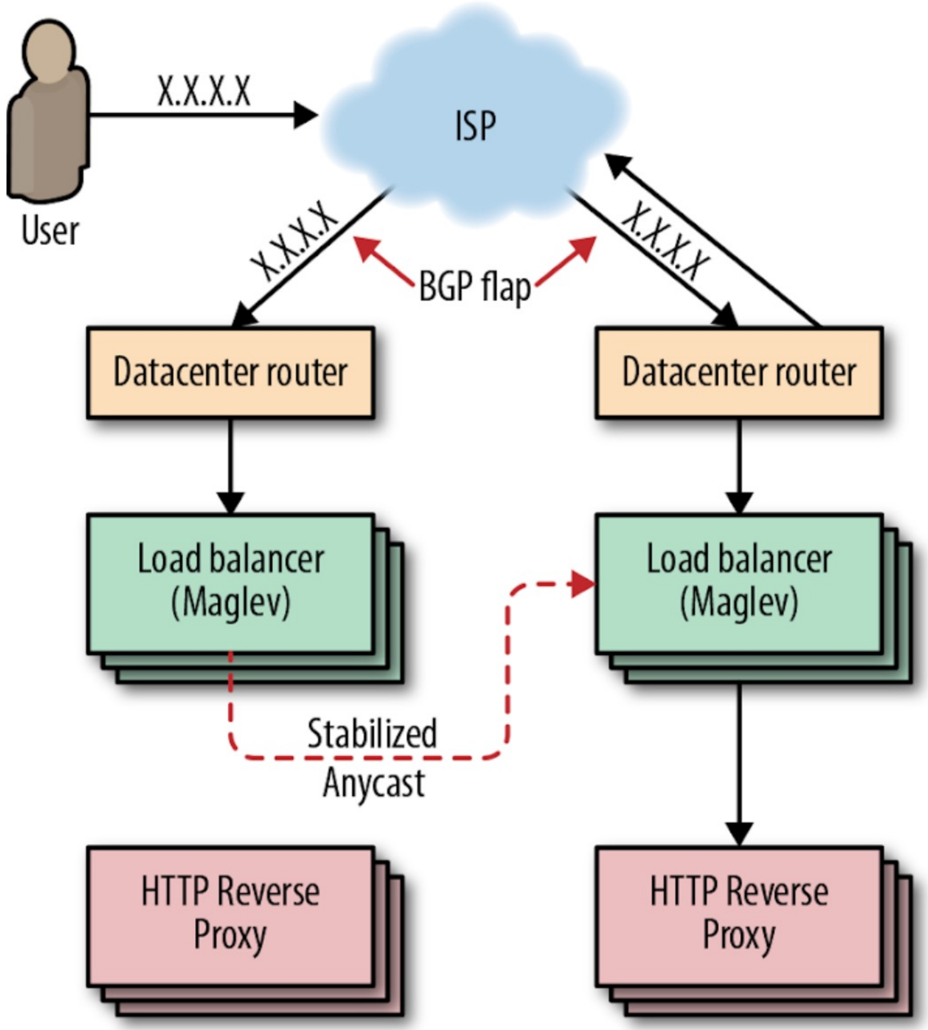
- Primary mechanism: DNS
- DNS response sizes are bounded. Client will just choose randomly from among responses; don’t know who is closest.
- Use IP anycast to talk to “nearest” (acc to BGP) authoritative DNS servers. Auth servers redirect user to closest through a single DNS response.
- Problem: clients 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)

Alternative: Virtual IP address

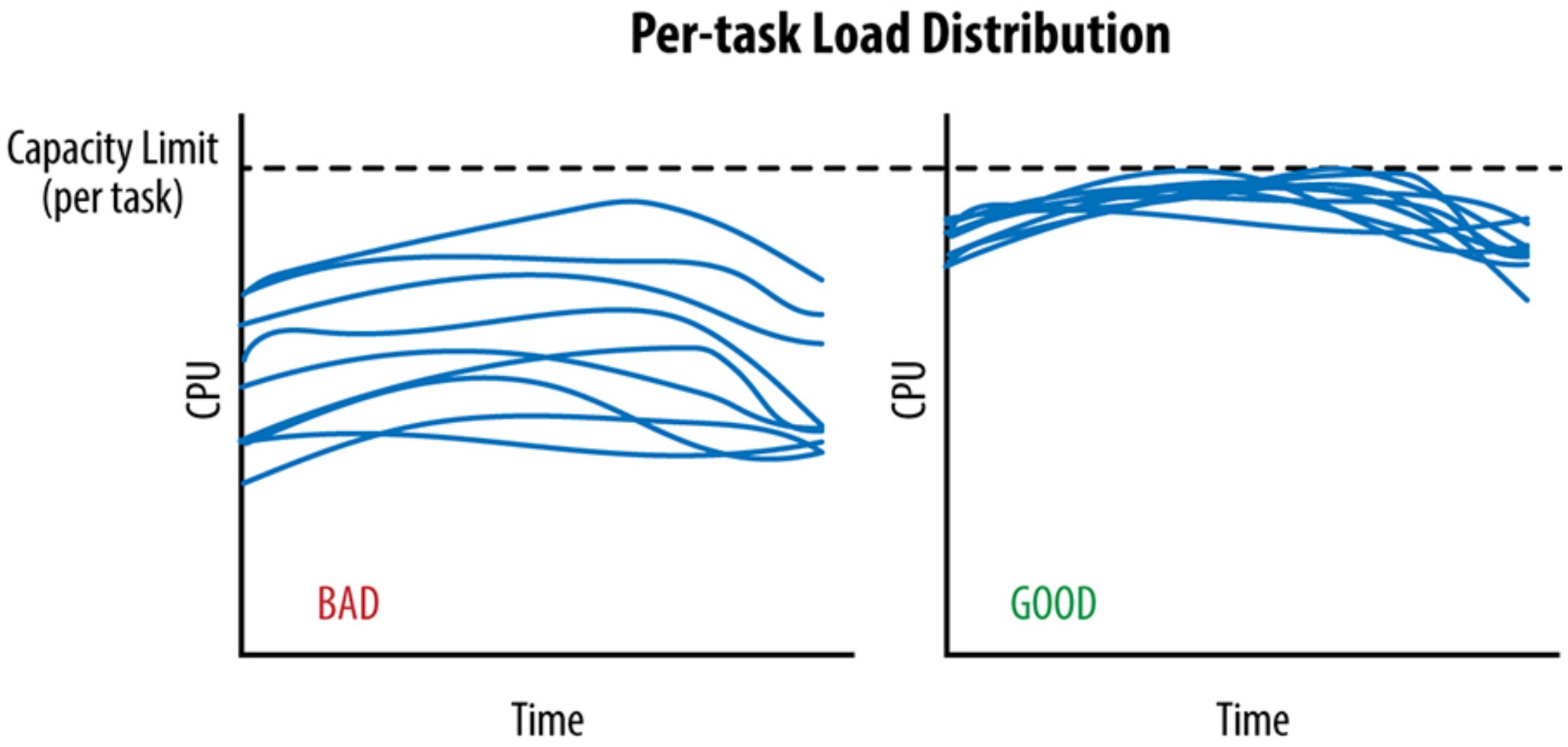
- Use a virtual IP address (VIP) to cover many real IP addresses
  - Hide growth, failures, maintenance in server pool from users
  - Use DNS with large TTL. Save latency.
  - Effectively decouple cluster-external from internal
- Can also use IP anycast directly to get to the edge
  - But anycast need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

Frontend load balancing

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



Even load is critical



Uneven load == stranded resources

Problem: Statefulness

- A user's TCP connection must always be sent to the same reverse proxy
- Important for performance advantages of reverse proxying
- If not, connection breaks!

Connection tracking and consistent hashing

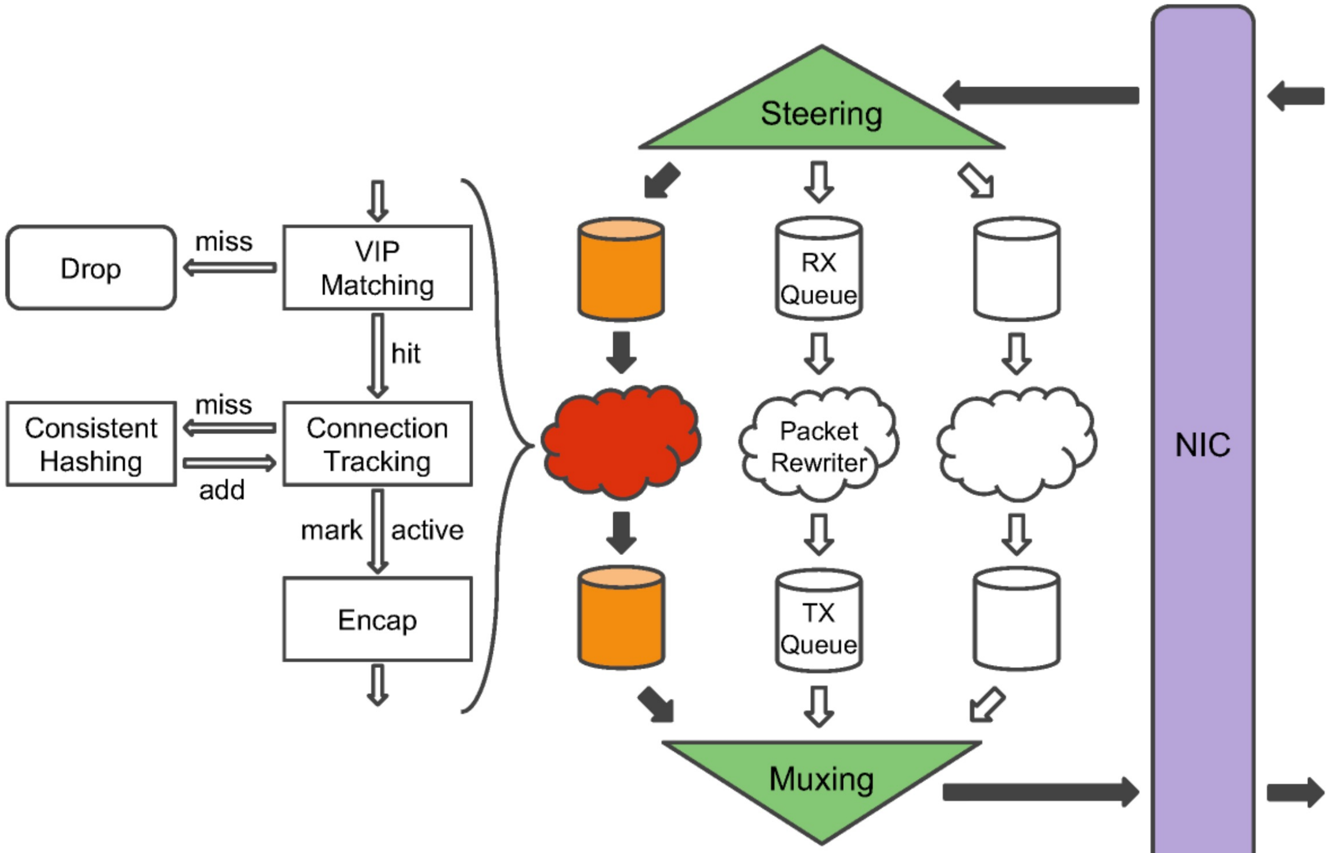
- Remembering connections by putting them in a connection tracking table: 5-tuple → backend
  - Not always possible
  - Even the load balancer forwarding a packet may change mid-connection
  - SYN floods and crowds may overwhelm connection tracking table
- If a packet's connection cannot be found in 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 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

$$offset \leftarrow h_1(name[i]) \mod M$$
$$skip \leftarrow h_2(name[i]) \mod (M-1)+1$$
$$permutation[i][j] \leftarrow (offset + j \times skip) \mod M$$

Backends choose slots based on permutation.

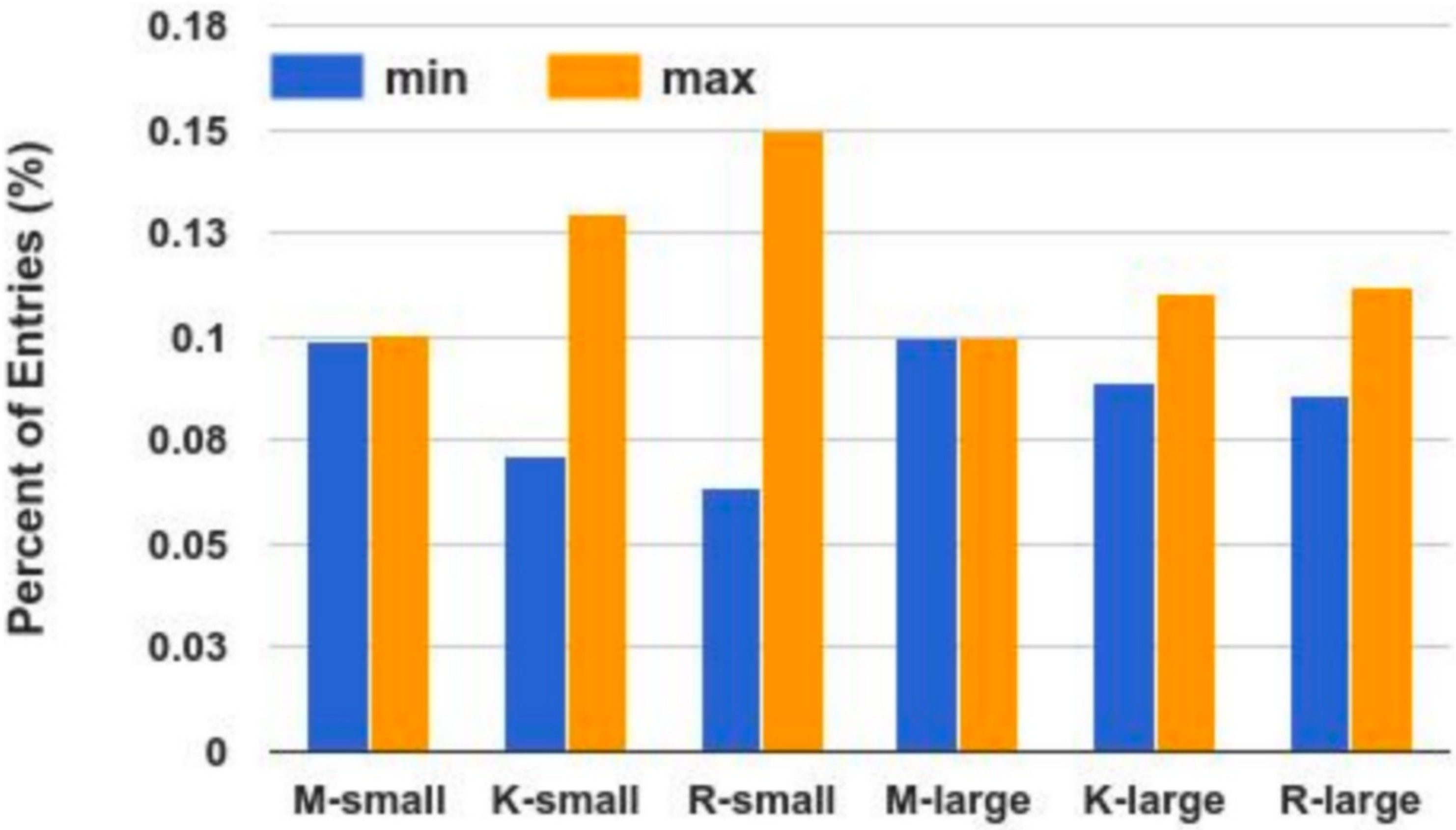
Pseudocode 1 Populate Maglev hashing lookup table.

```
function POPULATE
    for each i < N do next[i] ← 0 end for
    for each j < M do entry[j] ← −1 end for
    n ← 0
    while true do
        for each i < N do
            c ← permutation[i][next[i]]
            while entry[c] ≥ 0 do
                next[i] ← next[i] + 1
                c ← permutation[i][next[i]]
            end while
            entry[c] ← i
            next[i] ← next[i] + 1
            n ← n + 1
            if n = M then return end if
        end for
    end while
end function
```

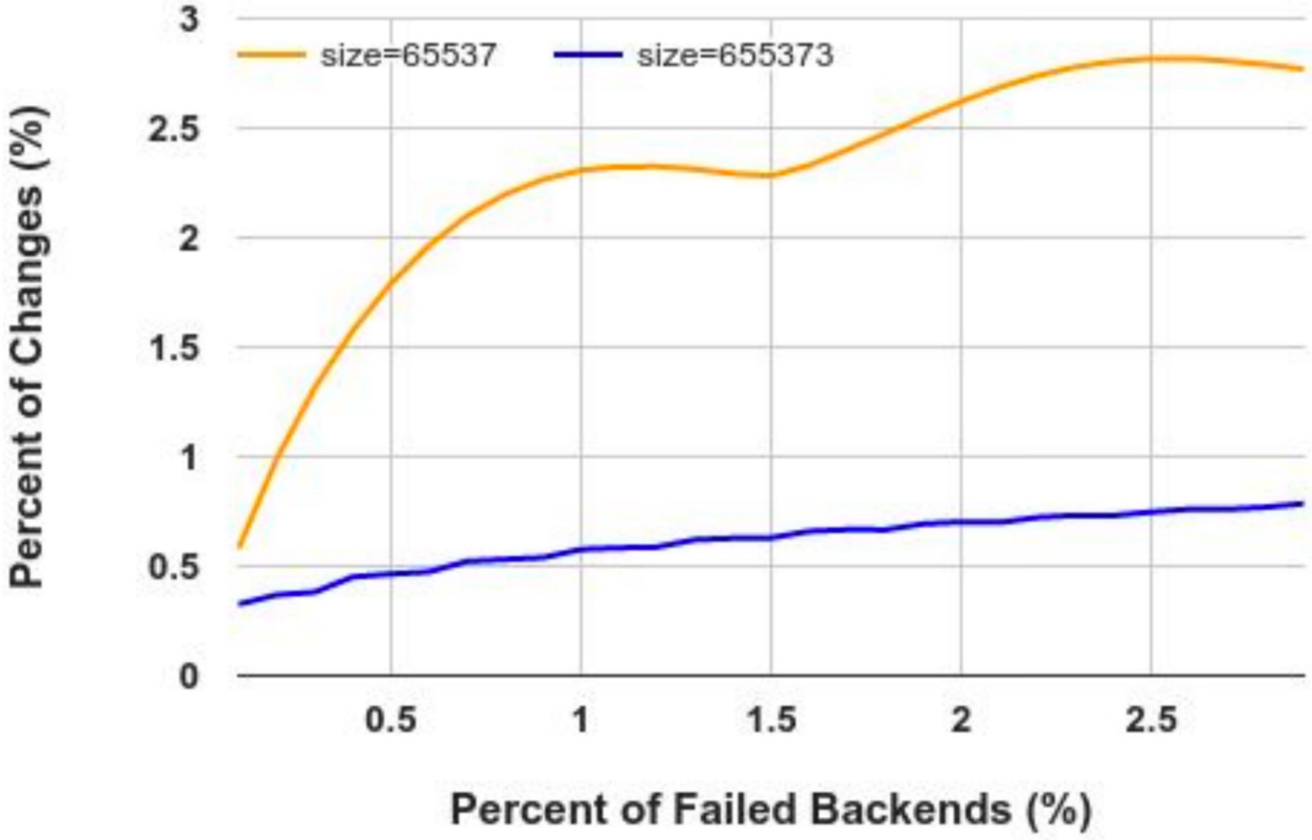
Actual packet forwarding

- (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

Balancing quality

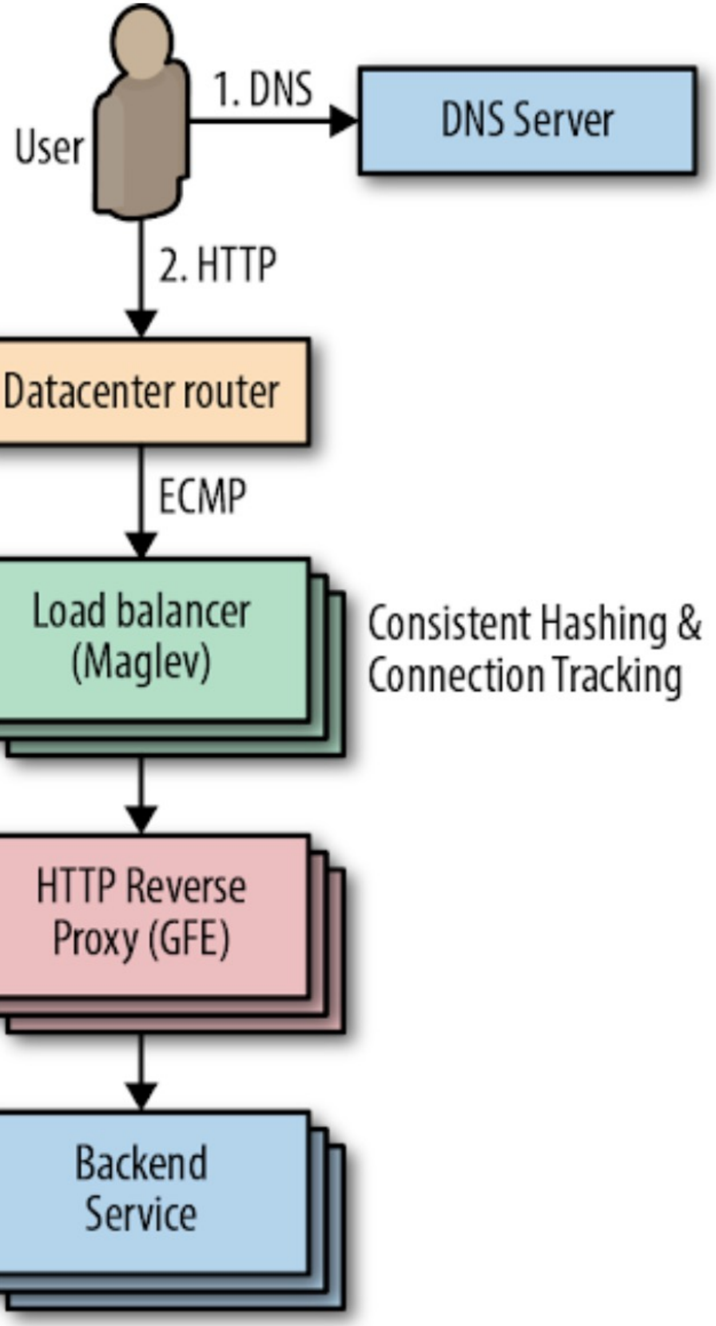


Disruptions on lookup table change



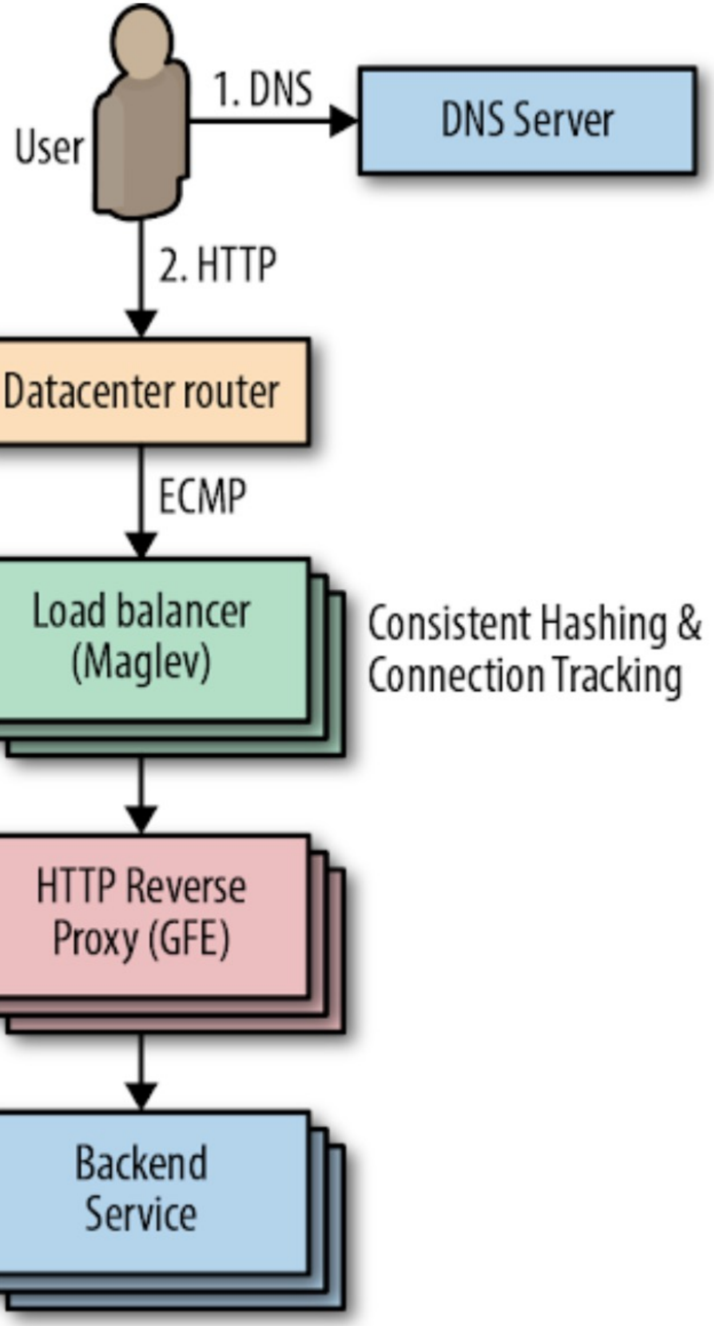
Beyond the reverse proxy

- Problem 1: avoid unhealthy backends first
  - "Least outstanding requests": If too many outstanding requests, avoid those backends
  - Only avoids extreme overload
  - Also, may waste capacity under diverse backend machines
- "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
- Connect to a subset of backends
  - How large?
  - Client load variation
  - # backends >> # clients
- Which backends of that size?
  - Random subsets can be bad



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-client view of load
- Good 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