Load Balancing at the Frontend

Written by Piotr Lewandowski
Edited by Sarah Chavis

We serve many millions of requests every second and, as you may have already guessed, we use more than a single computer to handle this demand. But even if we *did* have a supercomputer that was somehow able to handle all these requests (imagine the network connectivity such a configuration would require!), we still wouldn't employ a strategy that relied upon a single point of failure; when you're dealing with large-scale systems, putting all your eggs in one basket is a recipe for disaster.

This chapter focuses on high-level load balancing—how we balance user traffic *between* datacenters. The following chapter zooms in to explore how we implement load balancing *inside* a datacenter.

Power Isn't the Answer

For the sake of argument, let's assume we have an unbelievably powerful machine and a network that never fails. Would *that* configuration be sufficient to meet Google's needs? No. Even this configuration would still be limited by the physical constraints associated with our networking infrastructure. For example, the speed of light is a limiting factor on the communication speeds for fiber optic cable, which creates an upper bound on how quickly we can serve data based upon the distance it has to travel. Even in an ideal world, relying on an infrastructure with a single point of failure is a bad idea.

In reality, Google has thousands of machines and even more users, many of whom issue multiple requests at a time. *Traffic load balancing* is how we decide which of the many, many machines in our datacenters will serve a particular request. Ideally, traffic is distributed across multiple network links, datacenters, and machines in an "optimal" fashion. But what does "optimal" mean in this context? There’s actually no single answer, because the optimal solution depends heavily on a variety of factors:

- The hierarchical level at which we evaluate the problem (global versus local)
- The technical level at which we evaluate the problem (hardware versus software)
- The nature of the traffic we’re dealing with

Let’s start by reviewing two common traffic scenarios: a basic search request and a video upload request. Users want to get their query results quickly, so the most important variable for the search request is latency. On the other hand, users expect video uploads to take a non-negligible amount of time, but also want such requests to succeed the first time, so the most important variable for the video upload is throughput. The differing needs of the two requests play a role in how we determine the optimal distribution for each request at the *global* level:

- The search request is sent to the nearest available datacenter—as measured in round-trip time (RTT)—because we want to minimize the latency on the request.
- The video upload stream is routed via a different path—perhaps to a link that is currently underutilized—to maximize the throughput at the expense of latency.

But on the *local* level, inside a given datacenter, we often assume that all machines within the building are equally distant to the user and connected to the same network. Therefore, optimal distribution of load focuses on optimal resource utilization and protecting a single server from overloading.

Of course, this example presents a vastly simplified picture. In reality, many more considerations factor into optimal load distribution: some requests may be directed to a datacenter that is slightly farther away in order to keep caches warm, or non-interactive traffic may be routed to a completely different region to avoid network congestion. Load balancing, especially for large systems, is anything but straightforward and static. At Google, we’ve approached the problem by load balancing at multiple levels, two of which are described in the following sections. For the sake of presenting a concrete discussion, we’ll consider HTTP requests sent over TCP. Load balancing of stateless services (like DNS over UDP) differs slightly, but most of the mechanisms described here should be applicable to stateless services as well.

Load Balancing Using DNS

Before a client can even send an HTTP request, it often has to look up an IP address using DNS. This provides the perfect opportunity to introduce our first layer of load balancing: *DNS load balancing*. The simplest solution is to return multiple `A` or `AAAA` records in the DNS reply and let the client pick an IP address arbitrarily. While conceptually simple and trivial to implement, this solution poses multiple challenges.

The first problem is that it provides very little control over the client behavior: records are selected randomly, and each will attract a roughly equal amount of traffic. Can we mitigate this problem? In theory, we could use SRV records to specify record weights and priorities, but SRV records have not yet been adopted for HTTP.

Another potential problem stems from the fact that usually the client cannot determine the closest address. We *can* mitigate this scenario by using an anycast address for authoritative nameservers and leverage the fact that DNS queries will flow to the closest address. In its reply, the server can return addresses routed to the closest datacenter. A further improvement builds a map of all networks and their approximate physical locations, and serves DNS replies based on that mapping. However, this solution comes at the cost of having a much more complex DNS server implementation and maintaining a pipeline that will keep the location mapping up to date.

Of course, none of these solutions are trivial, due to a fundamental characteristic of DNS: end users rarely talk to authoritative nameservers directly. Instead, a recursive DNS server usually lies somewhere between end users and nameservers. This server proxies queries between a user and a server and often provides a caching layer. The DNS middleman has three very important implications on traffic management:

- Recursive resolution of IP addresses
- Nondeterministic reply paths
- Additional caching complications

Recursive resolution of IP addresses is problematic, as the IP address seen by the authoritative nameserver does not belong to a user; instead, it's the recursive resolver's. This is a serious limitation, because it only allows reply optimization for the shortest distance between resolver and the nameserver. A possible solution is to use the EDNS0 extension proposed in [Con15], which includes information about the client's subnet in the DNS query sent by a recursive resolver. This way, an authoritative nameserver returns a response that is optimal from the user's perspective, rather than the resolver's perspective. While this is not yet the official standard, its obvious advantages have led the biggest DNS resolvers (such as OpenDNS and Google[103]) to support it already.

Not only is it difficult to find the optimal IP address to return to the nameserver for a given user's request, but that nameserver may be responsible for serving thousands or millions of users, across regions varying from a single office to an entire continent. For instance, a large national ISP might run nameservers for its entire network from one datacenter, yet have network interconnects in each metropolitan area. The ISP's nameservers would then return a response with the IP address best suited for their datacenter, despite there being better network paths for all users!

Finally, recursive resolvers typically cache responses and forward those responses within limits indicated by the time-to-live (TTL) field in the DNS record. The end result is that estimating the impact of a given reply is difficult: a single authoritative reply may reach a single user or multiple thousands of users. We solve this problem in two ways:

- We analyze traffic changes and continuously update our list of known DNS resolvers with the approximate size of the user base behind a given resolver, which allows us to track the potential impact of any given resolver.
- We estimate the geographical distribution of the users behind each tracked resolver to increase the chance that we direct those users to the best location.

Estimating geographic distribution is particularly tricky if the user base is distributed across large regions. In such cases, we make trade-offs to select the best location and optimize the experience for the majority of users.

But what does "best location" really mean in the context of DNS load balancing? The most obvious answer is the location closest to the user. However (as if determining users' locations isn't difficult in and of itself), there are additional criteria. The DNS load balancer needs to make sure that the datacenter it selects has enough capacity to serve requests from users that are likely to receive its reply. It also needs to know that the selected datacenter and its network connectivity are in good shape, because directing user requests to a datacenter that's experiencing power or networking problems isn't ideal. Fortunately, we can integrate the authoritative DNS server with our global control systems that track traffic, capacity, and the state of our infrastructure.

The third implication of the DNS middleman is related to caching. Given that authoritative nameservers cannot flush resolvers' caches, DNS records need a relatively low TTL. This effectively sets a lower bound on how quickly DNS changes can be propagated to users.[104] Unfortunately, there is little we can do other than to keep this in mind as we make load balancing decisions.

Despite all of these problems, DNS is still the simplest and most effective way to balance load before the user's connection even starts. On the other hand, it should be clear that load balancing with DNS on its own is not sufficient. Keep in mind that all DNS replies served should fit within the 512-byte limit[105] set by RFC 1035 [Moc87]. This limit sets an upper bound on the number of addresses we can squeeze into a single DNS reply, and that number is almost certainly less than our number of servers.

To *really* solve the problem of frontend load balancing, this initial level of DNS load balancing should be followed by a level that takes advantage of virtual IP addresses.

Load Balancing at the Virtual IP Address

Virtual IP addresses (VIPs) are not assigned to any particular network interface. Instead, they are usually shared across many devices. However, from the user's perspective, the VIP remains a single, regular IP address. In theory, this practice allows us to hide implementation details (such as the number of machines behind a particular VIP) and facilitates maintenance, because we can schedule upgrades or add more machines to the pool without the user knowing.

In practice, the most important part of VIP implementation is a device called the *network load balancer*. The balancer receives packets and forwards them to one of the machines behind the VIP. These backends can then further process the request.

There are several possible approaches the balancer can take in deciding which backend should receive the request. The first (and perhaps most intuitive) approach is to always prefer the least loaded backend. In theory, this approach should result in the best end-user experience because requests are always routed to the least busy machine. Unfortunately, this logic breaks down quickly in the case of stateful protocols, which must use the same backend for the duration of a request. This requirement means that the balancer must keep track of all connections sent through it in order to make sure that all subsequent packets are sent to the correct backend. The alternative is to use some parts of a packet to create a connection ID (possibly using a hash function and some information from the packet), and to use the connection ID to select a backend. For example, the connection ID could be expressed as:

```
id(packet) mod N
```

where `id` is a function that takes `packet` as an input and produces a connection ID, and `N` is the number of configured backends.

This avoids storing state, and all packets belonging to a single connection are always forwarded to the same backend. Success? Not quite yet. What happens if one backend fails and needs to be removed from the backend list? Suddenly `N` becomes `N-1` and then, `id(packet) mod N` becomes `id(packet) mod N-1`. Almost every packet suddenly maps to a different backend! If backends don't share any state between themselves, this remapping forces a reset of almost all of the existing connections. This scenario is definitely *not* the best user experience, even if such events are infrequent.

Fortunately, there *is* an alternate solution that doesn't require keeping the state of every connection in memory, but won't force all connections to reset when a single machine goes down: *consistent hashing*. Proposed in 1997, consistent hashing [Kar97] describes a way to provide a mapping algorithm that remains relatively stable even when new backends are added to or removed from the list. This approach minimizes the disruption to existing connections when the pool of backends changes. As a result, we can usually use simple connection tracking, but fall back to consistent hashing when the system is under pressure (e.g., during an ongoing denial of service attack).

Returning to the larger question: how exactly should a network load balancer forward packets to a selected VIP backend? One solution is to perform a Network Address Translation. However, this requires keeping an entry of every single connection in the tracking table, which precludes having a completely stateless fallback mechanism.

Another solution is to modify information on the data link layer (layer 2 of the OSI networking model). By changing the destination MAC address of a forwarded packet, the balancer can leave all the information in upper layers intact, so the backend receives the original source and destination IP addresses. The backend can then send a reply directly to the original sender—a technique known as *Direct Server Response* (DSR). If user requests are small and replies are large (e.g., most HTTP requests), DSR provides tremendous savings, because only a small fraction of traffic need traverse the load balancer. Even better, DSR does not require us to keep state on the load balancer device. Unfortunately, using layer 2 for internal load balancing *does* incur serious disadvantages when deployed at scale: all machines (i.e., all load balancers and all their backends) must be able to reach each other at the data link layer. This isn't an issue if this connectivity can be supported by the network and the number of machines doesn't grow excessively, because all the machines need to reside in a single broadcast domain. As you may imagine, Google outgrew this solution quite some time ago, and had to find an alternate approach.

Our current VIP load balancing solution [Eis16] uses packet encapsulation. A network load balancer puts the forwarded packet into another IP packet with Generic Routing Encapsulation (GRE) [Han94], and uses a backend's address as the destination. A backend receiving the packet strips off the outer IP+GRE layer and processes the inner IP packet as if it were delivered directly to its network interface. The network load balancer and the backend no longer need to exist in the same broadcast domain; they can even be on separate continents as long as a route between the two exists.

Packet encapsulation is a powerful mechanism that provides great flexibility in the way our networks are designed and evolve. Unfortunately, encapsulation also comes with a price: inflated packet size. Encapsulation introduces overhead (24 bytes in the case of IPv4+GRE, to be precise), which can cause the packet to exceed the available Maximum Transmission Unit (MTU) size and require fragmentation.

Once the packet reaches the datacenter, fragmentation can be avoided by using a larger MTU within the datacenter; however, this approach requires a network that supports large Protocol Data Units. As with many things at scale, load balancing sounds simple on the surface—load balance early and load balance often—but the difficulty is in the details, both for frontend load balancing and for handling packets once they reach the datacenter.

[103] See *https://groups.google.com/forum/#!topic/public-dns-announce/67oxFjSLeUM*.

[104] Sadly, not all DNS resolvers respect the TTL value set by authoritative nameservers.

[105] Otherwise, users must establish a TCP connection just to get a list of IP addresses.

← PREVIOUS
Chapter 18 - Software Engineering in SRE

NEXT
Chapter 20 - Load Balancing in the Datacenter