blog.envoyproxy.io

Introduction to modern network load balancing and proxying

Matt Klein

40-50 minutes

It was brought to my attention recently that there is a dearth of introductory educational material available about modern network load balancing and proxying. I thought to myself: How can this be? Load balancing is one of the core concepts required for building reliable distributed systems. Surely there must be quality information available? I searched and found the pickings are indeed slim. The Wikipedia articles on load balancing and proxy servers contain overviews of some concepts but not a fluid treatment of the subject, especially as it pertains to modern microservice architectures. A Google search for load balancing primarily turns up vendor pages that are heavy on buzzwords and light on details.

In this post I attempt to rectify the lack of information by providing a gentle introduction to modern network load balancing and proxying. This is, frankly, a massive topic that could be the subject of an entire book. In the interest of keeping this article (somewhat) blog length, I try to distill a set of complex topics into a simple overview; depending on interest and feedback I will consider more detailed follow up posts on individual topics at a later point.

With a bit of background on why I wrote this out of the way — here we go!

What is network load balancing and proxying?

Wikipedia defines load balancing as:

> In computing, load balancing improves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy. Load balancing usually involves dedicated software or hardware, such as a multilayer switch or a Domain Name System server process.

The above definition applies to all aspects of computing, not just networks. Operating systems use load balancing to schedule tasks across physical processors, container orchestrators such as Kubernetes use load balancing to schedule tasks across a compute cluster, and network load balancers use load balancing to schedule network tasks across available backends. *The remainder of this post will cover network load balancing only*.

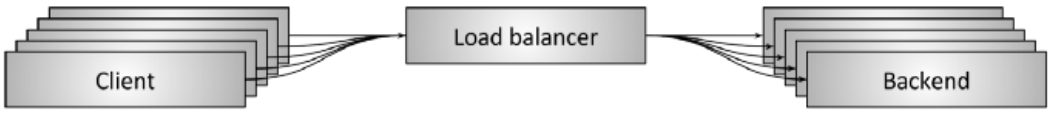


Figure 1: Network load balancing overview

Figure 1 shows a high level overview of network load balancing. Some number of clients are requesting resources from some number of backends. A load balancer sits between the clients and the backends and at a high level performs several critical tasks:

- **Service discovery**: What backends are available in the system? What are their addresses (i.e., how should the load balancer talk to them)?
- **Health checking**: What backends are currently healthy and available to accept requests?

- **Load balancing**: What algorithm should be used to balance individual requests across the healthy backends?

  Proper use of load balancing in a distributed system provides several benefits:

- **Naming abstraction**: Instead of every client needing to know about every backend (service discovery), the client can address the load balancer via a predefined mechanism and then the act of name resolution can be delegated to the load balancer. The predefined mechanisms include built-in libraries and well known DNS/IP/port locations and will be discussed in more detail below.
- **Fault tolerance**: Via health checking and various algorithmic techniques, a load balancer can effectively route around a bad or overloaded backend. This means that an operator can typically fix a bad backend at their leisure vs. as an emergency.
- **Cost and performance benefits**: Distributed system networks are rarely homogenous. The system is likely to span multiple network zones and regions. Within a zone, networks are often built in a relatively undersubscribed way. Between zones, oversubscription becomes the norm. (In this context over/undersubscription refers to the amount of bandwidth consumable via NICs as a percentage of bandwidth available between routers). Intelligent load balancing can keep request traffic within zones as much as possible, which increases performance (less latency) and reduces overall system cost (less bandwidth and fiber required between zones).

Load balancer vs. proxy

When talking about network load balancers, the terms *load balancer* and *proxy* are used roughly interchangeably within the industry. This post will also treat the terms as generally equivalent. (Pedantically, not all proxies are load balancers, but the vast majority of proxies perform load balancing as a primary function).

Some might additionally argue that when load balancing is done as part of an embedded client library, the load balancer is not really a proxy. However, I would argue that distinction adds needless complexity to an already confusing topic. The types of load balancer topologies are discussed in detail below, but this post treats the embedded load balancer topology as just a special case of proxying; the application is proxying through an embedded library that offers all the same abstractions as a load balancer that is outside of the application process.

L4 (connection/session) load balancing

When discussing load balancing across the industry today, solutions are often bucketed into two categories: *L4* and *L7*. These categories refer to *layer 4* and *layer 7* of the OSI model. For reasons that will become obvious when I discuss L7 load balancing, I think it's unfortunate that these are the terms that we use. The OSI model is a very poor approximation of the complexity of load balancing solutions that include traditional layer 4 protocols such as TCP and UDP but often end up including bits and pieces of protocols at a variety of different OSI layers. i.e., if a L4 TCP load balancer also supports TLS termination, is it now an L7 load balancer?

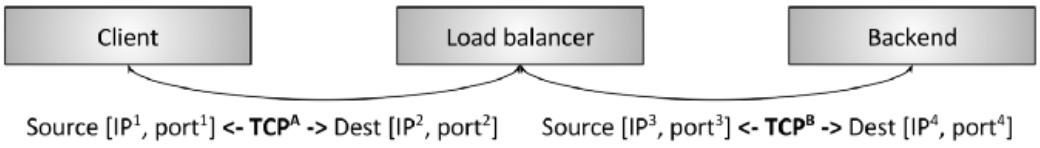


Figure 2: TCP L4 termination load balancing

Figure 2 shows a traditional L4 TCP load balancer. In this case, a client makes a TCP connection to the load balancer. The load balancer *terminates* the connection (i.e. responds directly to the SYN), selects a backend, and makes a new TCP connection to the backend (i.e. sends a new SYN). The details of the diagram aren't

important and will be discussed in detail in the section below dedicated to L4 load balancing.

The key takeaway of this section is that an L4 load balancer typically operates only at the level of the L4 TCP/UDP connection/session. Thus the load balancer roughly shuffles bytes back and forth, and makes sure that bytes from the same session wind up at the same backend. The L4 load balancer is unaware of any application details of the bytes that it is shuffling. The bytes could be HTTP, Redis, MongoDB, or any other application protocol.

L7 (application) load balancing

L4 load balancing is simple and still sees wide use. What are the shortcomings of L4 load balancing that warrant investment in L7 (application) load balancing? Take the following L4 specific case as an example:

- Two gRPC/HTTP2 clients want to talk to a backend so they connect through an L4 load balancer.
- The L4 load balancer makes a single outgoing TCP connection for each incoming TCP connection, resulting in two incoming and two outgoing connections.
- However, client A sends 1 request per minute (RPM) over its connection, while client B sends 50 requests per second (RPS) over its connection.

  In the previous scenario, *the backend selected to handle client A will be handling approximately 3000x less load then the backend selected to handle client B*! This is a large problem and generally defeats the purpose of load balancing in the first place. Note also that this problem happens for any *multiplexing*, *kept-alive* protocol. (Multiplexing means sending concurrent application requests over a single L4 connection, and kept-alive means not closing the connection when there are no active requests). All modern

protocols are evolving to be both multiplexing and kept-alive for efficiency reasons (it is generally expensive to create connections, especially when the connections are encrypted using TLS), so the L4 load balancer impedance mismatch is becoming more pronounced over time. This problem is fixed by the L7 load balancer.

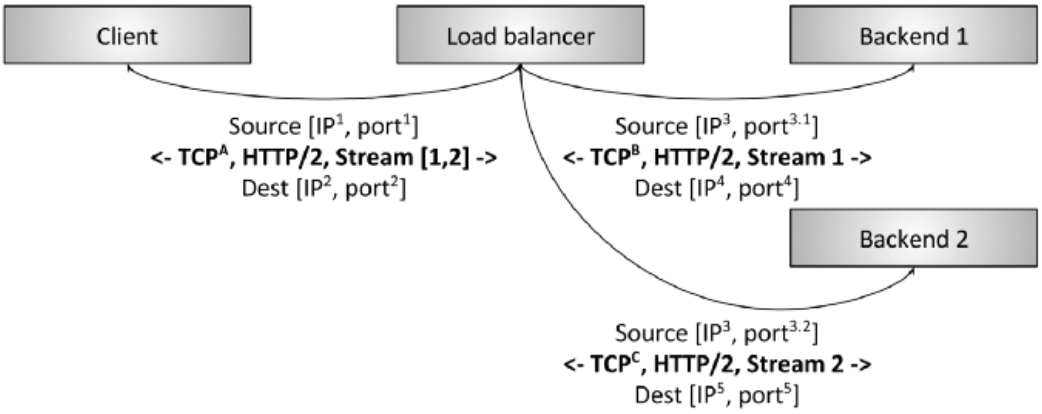


Figure 3: HTTP/2 L7 termination load balancing

Figure 3 shows an L7 HTTP/2 load balancer. In this case, the client makes a single HTTP/2 TCP connection to the load balancer. The load balancer then proceeds to make *two* backend connections. When the client sends two HTTP/2 streams to the load balancer, stream 1 is sent to backend 1 while stream 2 is sent to backend 2. Thus, even multiplexing clients that have vastly different request loads will be balanced efficiently across the backends. This is why L7 load balancing is so important for modern protocols. (L7 load balancing yields a tremendous amount of additional benefits due to its ability to inspect application traffic, but that will be covered in more detail below).

L7 load balancing and the OSI model

As I said above in the section on L4 load balancing, using the OSI model for describing load balancing features is problematic. The reason is that L7, at least as described by the OSI model, itself encompasses multiple discrete layers of load balancing abstraction.

e.g., for HTTP traffic consider the following sublayers:

- Optional Transport Layer Security (TLS). Note that networking people argue about which OSI layer TLS falls into. For the sake of this discussion we will consider TLS L7.
- Physical HTTP protocol (HTTP/1 or HTTP/2).
- Logical HTTP protocol (headers, body data, and trailers).
- Messaging protocol (gRPC, REST, etc.).

A sophisticated L7 load balancer may offer features related to each of the above sublayers. Another L7 load balancer might only have a small subset of features that place it in the L7 category. In short, the L7 load balancer landscape is vastly more complicated from a feature comparison perspective than the L4 category. (And of course this section has just touched on HTTP; Redis, Kafka, MongoDB, etc. are all examples of L7 application protocols that benefit from L7 load balancing).

Load balancer features

In this section I will briefly summarize the high level features that load balancers provide. Not all load balancers provide all features.

Service discovery

Service discovery is the process by which a load balancer determines the set of available backends. The methods are quite varied and some examples include:

- Static configuration file.
- DNS.
- Zookeeper, Etcd, Consul, etc.
- Envoy's universal data plane API.

Health checking

Health checking is the process by which the load balancer determines if the backend is available to serve traffic. Health checking generally falls into two categories:

- **Active**: The load balancer sends a ping on a regular interval (e.g., an HTTP request to a `/healthcheck` endpoint) to the backend and uses this to gauge health.
- **Passive**: The load balancer detects health status from the primary data flow. e.g., an L4 load balancer might decide a backend is unhealthy if there have been three connection errors in a row. An L7 load balancer might decide a backend is unhealthy if there have been three HTTP 503 response codes in a row.

Load balancing

Yes, load balancers have to actually balance load! Given a set of healthy backends, how is the backend selected that will serve a connection or request? Load balancing algorithms are an active area of research and range from simplistic ones such as random selection and round robin, to more complicated algorithms that take into account variable latency and backend load. One of the most popular load balancing algorithms given its performance and simplicity is known as power of 2 least request load balancing.

Sticky sessions

In certain applications, it is important that requests for the same *session* reach the same backend. This might have to do with caching, temporary complex constructed state, etc. The definition of a session varies and might include HTTP cookies, properties of the client connection, or some other attribute. Many L7 load balancers have some support for sticky sessions. As an aside, I will note that

session stickiness is inherently fragile (the backend hosting the session can die), so caution is advised when designing a system that relies on them.

TLS termination

The topic of TLS and its role in both edge serving and securing service-to-service communication is worthy of its own post. With that said, many L7 load balancers do a large amount of TLS processing that includes termination, certificate verification and pinning, certificate serving using SNI, etc.

Observability

As I like to say in my talks: “Observability, observability, observability.” Networks are inherently unreliable and the load balancer is often responsible for exporting stats, traces, and logs that help operators figure out what is wrong so they can remediate the problem. Load balancers vary widely in their observability output. The most advanced load balancers offer copious output that includes numeric stats, distributed tracing, and customizable logging. I will point out that enhanced observability is not free; the load balancer has to do extra work to produce it. However, the benefits of the data greatly outweigh the relatively minor performance implications.

Security and DoS mitigation

Especially in the edge deployment topology (see below), load balancers often implement various security features including rate limiting, authentication, and DoS mitigation (e.g., IP address tagging and identification, tarpitting, etc.).

Configuration and control plane

Load balancers need to be configured. In large deployments, this can become a substantial undertaking. In general, the system that configures the load balancers is known as the “control plane” and varies widely in its implementation. For more information on this topic please see my post on service mesh data plane vs. control plane.

And a whole lot more

This section has just scratched the surface of the types of functionality that load balancers provide. Additional discussion can be found in the section on L7 load balancers below.

Types of load balancer topologies

Now that I have covered a high level overview of what a load balancer is, the difference between L4 and L7 load balancers, and a summary of load balancer features, I will move on to the various distributed system topologies in which load balancers are deployed. (Each of the following topologies are applicable to both L4 and L7 load balancers).

Middle proxy

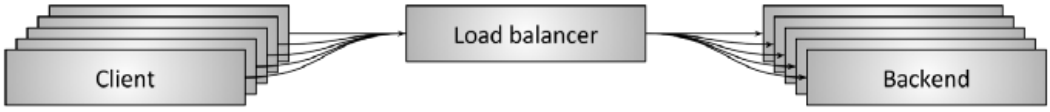


Figure 4: Middle proxy load balancing topology

The middle proxy topology shown in **figure 4** is likely the most familiar way to obtain load balancing for most readers. This category encompasses hardware devices from Cisco, Juniper, F5, etc.; cloud software solutions such as Amazon’s ALB and NLB and Google’s Cloud Load Balancer; and pure software self-hosted solutions such as HAProxy, NGINX, and Envoy. The pro of a middle

proxy solution is user simplicity. In general, users connect to the load balancer via DNS and don't need to worry about anything else. The con of a middle proxy solution is the fact that the proxy (even if clustered) is a single point of failure as well as a scaling bottleneck. A middle proxy is also often a black box that makes operations difficult. Is an observed problem in the client? In the physical network? In the middle proxy? In the backend? It can be very hard to tell.

Edge proxy

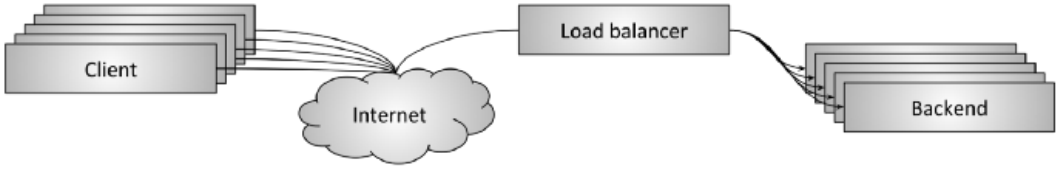


Figure 5: Edge proxy load balancing topology

The edge proxy topology shown in **figure 5** is really just a variant of the middle proxy topology in which the load balancer is accessible via the Internet. In this scenario the load balancer typically must provide additional "API gateway" features such as TLS termination, rate limiting, authentication, and sophisticated traffic routing. The pros and cons of the edge proxy are the same as the middle proxy. A caveat is that it is typically unavoidable to deploy a dedicated edge proxy in a large Internet-facing distributed system. Clients typically need to access the system over DNS using arbitrary network libraries that the service owner does not control (making the embedded client library or sidecar proxy topologies described in the following sections impractical to run directly on the client). Additionally, for security reasons it is desirable to have a single gateway by which all Internet-facing traffic ingresses into the system.

Embedded client library

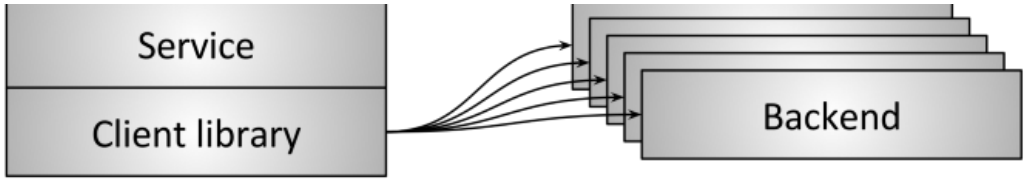


Figure 6: Load balancing via embedded client library

To avoid the single point of failure and scaling issues inherent in middle proxy topologies, more sophisticated infrastructures have moved towards embedding the load balancer directly into services via a library, as shown in **figure 6**. Libraries vary greatly in supported features, but some of the most well known and feature-rich in this category are Finagle, Eureka/Ribbon/Hystrix, and gRPC (loosely based on an internal Google system called Stubby). The main pro of a library based solution is that it fully distributes all of the functionality of the load balancer to each client, thus removing the single point of failure and scaling issues previously described. The primary con of a library-based solution is the fact that the library must be implemented in every language that an organization uses. Distributed architectures are becoming increasingly “polyglot” (multilingual). In this environment, the cost of reimplementing an extremely sophisticated networking library in many different languages can become prohibitive. Finally, deploying library upgrades across a large service architecture can be extremely painful, making it highly likely that many different versions of the library will be running concurrently in production, increasing operational cognitive load.

With all of that said, the libraries mentioned above have been successful for companies that have been able to limit programming language proliferation and overcome library upgrade pains.

Sidecar proxy

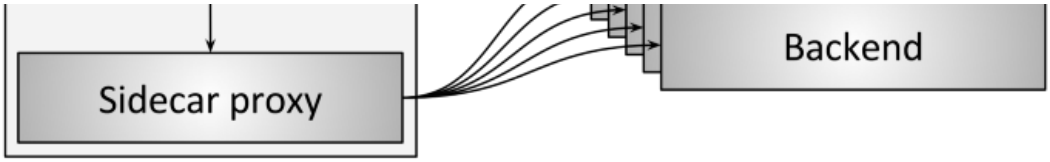


Figure 7: Load balancing via sidecar proxy

A variant of the embedded client library load balancer topology is the sidecar proxy topology shown in **figure 7**. In recent years, this topology has been popularized as the "service mesh." The idea behind the sidecar proxy is that at the cost of a slight latency penalty via hopping to a different process, all of the benefits of the embedded library approach can be obtained without any programming language lock-in. The most popular sidecar proxy load balancers as of this writing are Envoy, NGINX, HAProxy, and Linkerd. For a more detailed treatment of the sidecar proxy approach please see my blog post introducing Envoy as well as my post on the service mesh data plane vs. control plane.

Summary and pros/cons of the different load balancer topologies

- The middle proxy topology is typically the easiest load balancing topology to consume. It falls short due to being a single point of failure, scaling limitations, and black box operation.
- The edge proxy topology is similar to middle proxy but typically cannot be avoided.
- The embedded client library topology offers the best performance and scalability, but suffers from the need to implement the library in every language as well as the need to upgrade the library across all services.
- The sidecar proxy topology does not perform as well as the embedded client library topology, but does not suffer from any of the limitations.

Overall, I think the sidecar proxy topology (service mesh) is gradually going to replace all other topologies for service-to-service communication. The edge proxy topology will always be required prior to traffic entering the service mesh.

Current state of the art in L4 load balancing

Are L4 load balancers still relevant?

This post has already discussed how great L7 load balancers are for modern protocols and will move on to L7 load balancer features in further detail below. Does this mean that L4 load balancers are no longer relevant? No! Although in my opinion L7 load balancers will ultimately completely replace L4 load balancers *for service-to-service communication*, L4 load balancers are still extremely relevant *at the edge* because almost all modern large distributed architectures use a two-tiered L4/L7 load balancing architecture for Internet traffic. The benefits of placing dedicated L4 load balancers before L7 load balancers in an edge deployment are:

- Because L7 load balancers perform substantially more sophisticated analysis, transformation, and routing of application traffic, they can handle a relatively small fraction of the raw traffic load (measured in packets per second and bytes per second) than an optimized L4 load balancer can. This fact generally makes L4 load balancers a better location to handle certain types of DoS attacks (e.g., SYN floods, generic packet flood attacks, etc.).
- L7 load balancers tend to be more actively developed, are deployed more often, and have more bugs than L4 load balancers. Having an L4 load balancer in front that can do health checking and draining during L7 load balancer deploys is substantially easier than the deployment mechanisms used with modern L4 load balancers, which typically use BGP and ECMP (more on this below). And finally, because L7 load balancers are more likely to

have bugs purely due to the complexity of their functionality, having an L4 load balancer that can route around failures and anomalies leads to a more stable overall system.

In the following sections I will describe several different designs for middle/edge proxy L4 load balancers. The following designs are generally not applicable to the client library and sidecar proxy topologies.

TCP/UDP termination load balancers

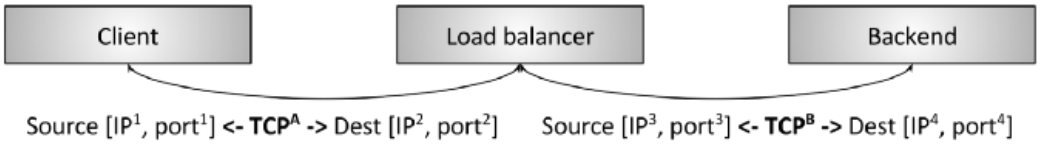


Figure 8: L4 termination load balancer

The first type of L4 load balancer still in use is the termination load balancer shown in **figure 8**. This is the same load balancer that we saw in the introduction to L4 load balancing above. In this type of load balancer, two discrete TCP connections are used: one between the client and the load balancer and one between the load balancer and the backend.

L4 termination load balancers are still used for two reasons:

1. They are relatively simple to implement.
2. Connection termination in close proximity (low latency) to the client has substantial performance implications. Specifically, if a terminating load balancer can be placed close to clients that are using a lossy network (e.g., cellular), retransmits are likely to happen faster prior to the data being moved to reliable fiber transit en-route to its ultimate location. Said another way, this type of load balancer might be used in a Point of Presence (POP) scenario for raw TCP connection termination.

TCP/UDP passthrough load balancers

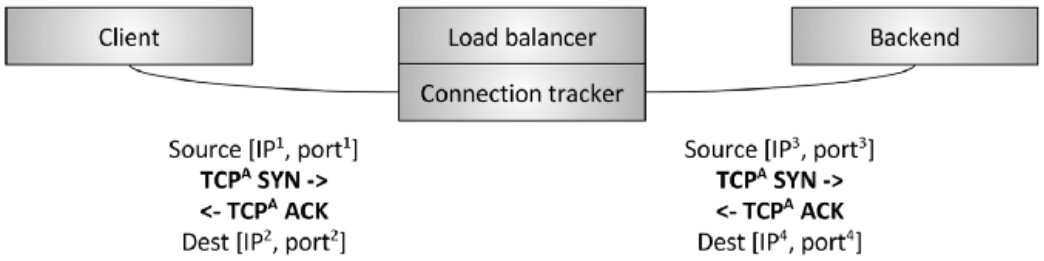


Figure 9: L4 passthrough load balancer

The second type of L4 load balancer is the passthrough load balancer shown in **figure 9**. In this type of load balancer, the TCP connection is *not* terminated by the load balancer. Instead, packets for each connection are forwarded on to a selected backend after connection tracking and Network Address Translation (NAT) take place. First, let's define connection tracking and NAT:

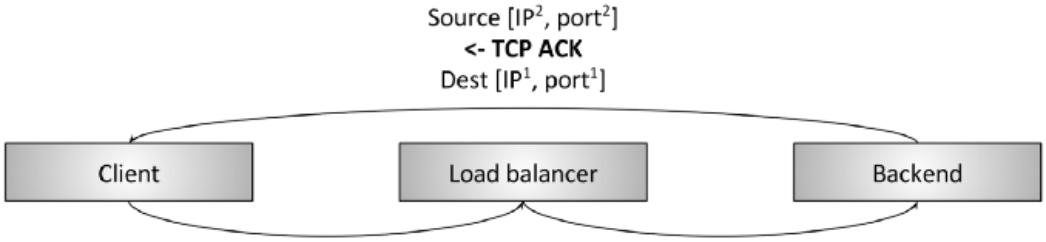
- **Connection tracking**: Is the process of keeping track of the state of all active TCP connections. This includes data such as whether the handshake has completed, whether a FIN has been received, how long the connection has been idle, which backend has been selected for the connection, etc.
- **NAT**: NAT is the process of using connection tracking data to alter IP/port information of packets as they traverse the load balancer.

  Using both connection tracking and NAT, the load balancer can passthrough *mostly* raw TCP traffic from the client to the backend. For example, let's say the client is talking to `1.2.3.4:80` and the selected backend is located at `10.0.0.2:9000`. The client TCP packets will arrive at the load balancer at `1.2.3.4:80`. The load balancer will then swap the destination IP and port of the packet with `10.0.0.2:9000`. It will also swap the source IP of the packet with the IP address of the load balancer. Thus, when the backend responds on the TCP connection, the packets will go back to the load balancer, where connection tracking takes place and NAT can happen again in the reverse direction.

Why would this type of load balancer be used in place of the termination load balancer described in the previous section given that it is more complicated? A few reasons:

- **Performance and resource usage**: Because passthrough load balancers are not terminating TCP connections, they do not need to buffer any TCP connection window. The amount of state stored per connection is quite small and is generally accessed via efficient hash table lookups. Because of this, passthrough load balancers can typically handle a substantially larger number of active connections and packets per second (PPS) than a terminating load balancer.
- **Allows backends to perform customized congestion control**: TCP congestion control is the mechanism by which endpoints on the Internet throttle sending data so as to not overwhelm available bandwidth and buffers. Since a passthrough load balancer is not terminating the TCP connection, it does not participate in congestion control. This fact allows backends to use different congestion control algorithms depending on their application use case. It also allows easier experimentation on congestion control changes (e.g., the recent BBR rollout).
- **Forms the baseline for Direct server return (DSR) and clustered L4 load balancing**: Passthrough load balancing is required for more advanced L4 load balancing techniques such as DSR and clustering with distributed consistent hashing (discussed in the following sections).

Direct server return (DSR)



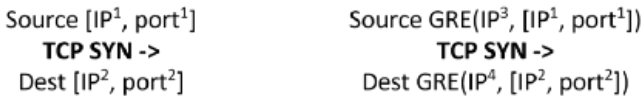


Figure 10: L4 Direct server return (DSR)

A Direct Server Return (DSR) load balancer is shown in **figure 10**. DSR builds on the passthrough load balancer described in the previous section. DSR is an optimization in which only *ingress/request* packets traverse the load balancer. *Egress/response* packets travel around the load balancer directly back to the client. The primary reason why it's interesting to perform DSR is that in many workloads, response traffic dwarfs request traffic (e.g., typical HTTP request/response patterns). Assuming 10% of traffic is request traffic and 90% of traffic is response traffic, if DSR is being used a load balancer with *1/10* of the capacity can meet the needs of the system. Since historically load balancers have been extremely expensive, this type of optimization can have substantial implications on system cost and reliability (less is always better). DSR load balancers extend the concepts of the passthrough load balancer with the following:

- The load balancer still typically performs *partial* connection tracking. Since response packets do not traverse the load balancer, the load balancer will not be aware of the complete TCP connection state. However, the load balancer can strongly infer the state by looking at the client packets and using various types of idle timeouts.
- Instead of NAT, the load balancer will typically use Generic Routing Encapsulation (GRE) to encapsulate the IP packets being sent from the load balancer to the backend. Thus, when the backend receives the encapsulated packet, it can decapsulate it and know the original IP address and TCP port of the client. This allows the backend to respond directly to the client without the response packets flowing through the load balancer.
- An important part of the DSR load balancer is that the *backend*

participates in the load balancing. The backend needs to have a properly configured GRE tunnel and depending on the low level details of the network setup may need its own connection tracking, NAT, etc.

Note that in both the passthrough load balancer and DSR load balancer designs there are a large variety of ways that connection tracking, NAT, GRE, etc. can be setup across the load balancer and the backend. Unfortunately that topic is beyond the scope of this article.

Fault tolerance via high availability pairs

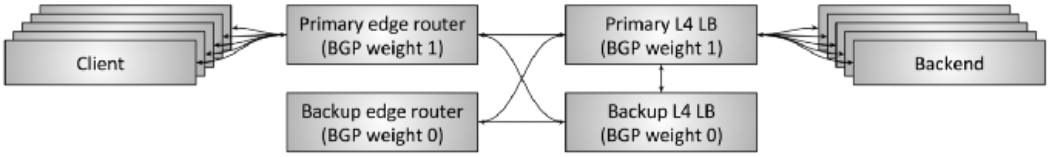


Figure 11: L4 fault tolerance via HA pairs and connection tracking

Up to now, we have been considering the design of L4 load balancers in isolation. Both the passthrough and DSR load balancer require some amount of connection tracking and state in the load balancer itself. What if the load balancer dies? If a single instance of the load balancer dies, all connections traversing the load balancer will be severed. Depending on the application, this may have substantial impact on application performance.

Historically, L4 load balancers have been hardware devices purchased from the typical vendors (Cisco, Juniper, F5, etc.). These devices are extremely expensive and handle a large amount of traffic. In order to avoid a single load balancer failure severing all connections and leading to a substantial application outage, load balancers have typically been deployed in *high availability pairs* as shown in **figure 11**. A typical HA load balancer setup has the following design:

- A pair of HA edge routers service some number of virtual IPs (VIP). These edge routers announce the VIPs using Border Gateway Protocol (BGP). The primary edge router has a higher BGP weight than the backup, so at steady state it is serving all traffic. (BGP is an extremely complicated protocol; for the purpose of this article, just consider BGP a mechanism by which network devices announce that they are available to take traffic from other network devices and that each link can have a weight that prioritizes link traffic).
- Similarly, the primary L4 load balancer announces itself to the edge routers with a higher BGP weight than the backup, so at steady state it is serving all traffic.
- The primary load balancer is *cross-connected* to the backup, and shares all of its connection tracking state. Thus, if the primary dies, the backup can take over handling all active connections.
- The two edge routers and the two load balancers are all *cross-connected*. This means that if one of the edge routers or one of the load balancers dies, or has its BGP announcement withdrawn for some other reason, the backup can take over serving all traffic.

  The above setup is how many high traffic Internet applications are still served today. However, there are substantial downsides to the above approach:

- VIPs must be correctly sharded across HA load balancer pairs taking into account capacity usage. If a single VIP grows beyond the capacity of a single HA pair, the VIP needs to be split into multiple VIPs.
- The resource usage of the system is poor. 50% of capacity sits idle at steady state. Given that historically hardware load balancers are extremely expensive, this leads to a substantial amount of idle capital.

- Modern distributed system design prefers greater fault tolerance than active/backup provides. e.g., optimally, a system should be able to suffer multiple simultaneous failures and keep running. An HA load balancer pair is susceptible to total failure if both the active and backup load balancer die at the same time.
- Proprietary large hardware devices from vendors are extremely expensive and lead to vendor lock-in. It is generally desirable to replace these hardware devices with horizontally scalable software solutions built using commodity compute servers.

Fault tolerance and scaling via clusters with distributed consistent hashing

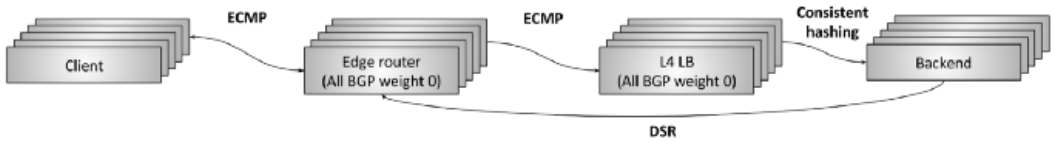


Figure 12: L4 fault tolerance and scaling via clustered load balancers and consistent hashing

The previous section introduced L4 load balancer fault tolerance via HA pairs as well as the problems inherent in that design. Starting in the early to mid 2000s, large Internet infrastructures started to design and deploy new massively parallel L4 load balancing systems as shown in **figure 12**. The goals of these systems are:

- Mitigate all of the downsides of the HA pair design described in the previous section.
- Move away from proprietary hardware load balancers from vendors to commodity software solutions built using standard compute servers and NICs.

This L4 load balancer design is best referred to as *fault tolerance and scaling via clustering and distributed consistent hashing*. It works as follows:

- N edge routers announce all Anycast VIPs at an identical BGP weight. Equal-cost multi-path routing (ECMP) is used to ensure that in general, all packets from a single *flow* arrive at the same edge router. A flow is typically the 4-tuple of source IP/port and destination IP/port. (In short, ECMP is a way of distributing packets over a set of identically weighted network links using consistent hashing). Although the edge routers themselves don't particularly care which packets arrive where, in general it is preferred that all packets from a flow traverse the same set of links so as to avoid out of order packets which degrade performance.
- N L4 load balancer machines announce all VIPs at an identical BGP weight to the edge routers. Again using ECMP, the edge routers will generally select the same load balancer machine for a flow.
- Each L4 load balancer machine will typically perform partial connection tracking, and then use consistent hashing to select a backend for the flow. GRE is used to encapsulate the packets sent from the load balancer to the backend.
- DSR is then used to send packets directly from the backend to the client via the edge routers.
- The actual consistent hashing algorithm used by the L4 load balancer is an area of active research. There are tradeoffs primarily around equalizing load, minimizing latency, minimizing disruption during backend changes,
  and minimizing memory overhead. A complete discussion of this topic is outside the scope of this article.

  Let's see how the above design mitigates all of the downsides of the HA pair approach:
- New edge routers and load balancer machines can be added as needed. Consistent hashing is used at every layer to decrease the

number of affected flows as much as possible when new machines are added.

- The resource usage of the system can be run as high as desired while maintaining sufficient burst margins and fault tolerance.
- Both the edge routers and load balancers can now be built using commodity hardware at a tiny fraction of the cost of traditional hardware load balancers (more on this below).

  One question that is typically asked about this design is “why don’t the edge routers talk directly to the backends via ECMP? Why do we need the load balancer at all?” The reasons for this are primarily around DoS mitigation and backend operational ease. Without the load balancer, each backend would have to participate in BGP and would have a substantially harder time performing rolling deploys.

  All modern L4 load balancing systems are moving towards this design (or some variant of it). The two most prominent publicly known examples are Maglev from Google and the Network Load Balancer (NLB) from Amazon. There is not currently any OSS load balancer that implements this design, however, there is a company that I know of planning to release one to OSS in 2018. I’m very excited for this release as a modern L4 load balancer is a crucial piece of missing OSS in the networking space.

Current state of the art in L7 load balancing

Yes, indeed. The last several years have seen a resurgence in L7 load balancer/proxy development. This tracks very well with the continued push towards microservice architectures in distributed systems. Fundamentally, the inherently faulty network becomes that much more difficult to operate efficiently when it is used more frequently. Furthermore, the rise of auto-scaling, container schedulers, etc. means that the days of provisioning static IPs in static files are long gone. Systems are not only utilizing the network

more, they are becoming substantially more dynamic, requiring new functionality in load balancers. In this section I will briefly summarize the areas that are seeing the most development in modern L7 load balancers.

Protocol support

Modern L7 load balancers are adding explicit support for many different protocols. The more knowledge that the load balancer has about the application traffic, the more sophisticated things it can do with regard to observability output, advanced load balancing and routing, etc. For example, as of this writing, Envoy explicitly supports L7 protocol parsing and routing for HTTP/1, HTTP2, gRPC, Redis, MongoDB, and DynamoDB. More protocols are likely to get added in the future including MySQL and Kafka.

Dynamic configuration

As described above, the increasingly dynamic nature of distributed systems is requiring a parallel investment in creating dynamic and reactive control systems. Istio is one example of such a system. Please see my post on service mesh data plane vs. control plane for more information on this topic.

Advanced load balancing

L7 load balancers now commonly have built-in support for advanced load balancing features such as timeouts, retries, rate limiting, circuit breaking, shadowing, buffering, content based routing, etc.

Observability

As described in the section above on general load balancer features, the increasingly dynamic systems that are being deployed

are becoming increasingly hard to debug. Robust *protocol specific* observability output is possibly the most important feature that modern L7 load balancers provide. Outputting numeric stats, distributed traces, and customizable logging is now virtually required for any L7 load balancing solution.

Extensibility

Users of modern L7 load balancers often want to easily extend them to add custom functionality. This can be done via writing pluggable filters that are loaded into the load balancer. Many load balancers also support scripting, typically via Lua.

Fault tolerance

I wrote quite a bit above about L4 load balancer fault tolerance. What about L7 load balancer fault tolerance? In general, we treat L7 load balancers as expendable and stateless. Using commodity software allows L7 load balancers to be easily horizontally scaled. Furthermore, the processing and state tracking that L7 load balancers perform is substantially more complicated than L4. Attempting to build an HA pairing of an L7 load balancer is technically possible but it would be a major undertaking.

Overall, in both the L4 and L7 load balancing domains, the industry is moving away from HA pairing towards horizontally scalable systems that converge via consistent hashing.

And more

L7 load balancers are evolving at a staggering pace. For an example of what Envoy provides please see Envoy's architecture overview.

Global load balancing and the centralized control

plane

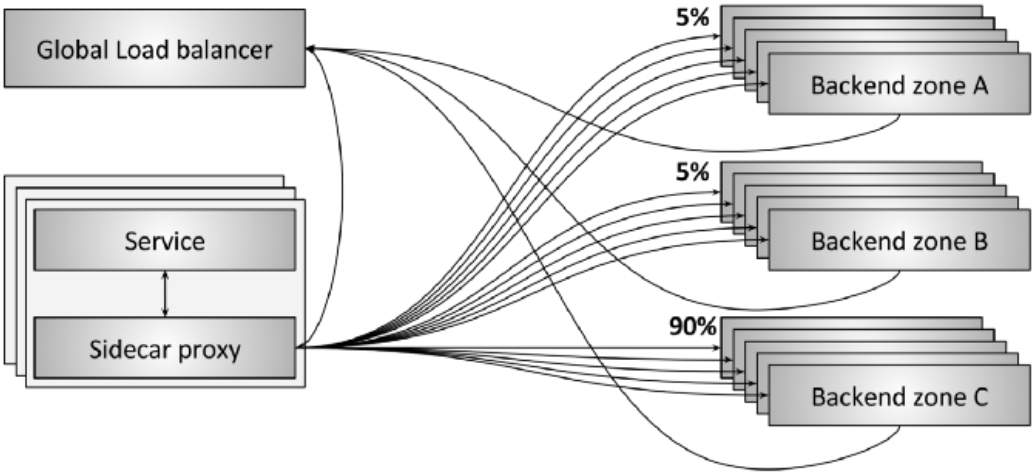


Figure 13: Global load balancing

The future of load balancing will increasingly treat the individual load balancers as commodity devices. In my opinion, the real innovation and commercial opportunities all lie within the control plane. **Figure 13** shows an example of a *global load balancing system*. In this example, a few different things are happening:

- Each sidecar proxy is communicating with backends in three different zones (A, B, and C).
- As illustrated, 90% of traffic is being sent to zone C while 5% of traffic is being sent to both zone A and B.
- The sidecar proxy and the backends are all reporting periodic state to the global load balancer. This allows the global load balancer to make decisions that take into account latency, cost, load, current failures, etc.
- The global load balancer periodically configures each sidecar proxy with current routing information.

  The global load balancer will increasingly be able to do sophisticated things that no individual load balancer can do on its own. For example:

- Automatically detect and route around zonal failure.
- Apply global security and routing policies.
- Detect and mitigate traffic anomalies including DDoS attacks using machine learning and neural networks.
- Provide centralized UI and visualizations that allow engineers to understand and operate the entire distributed system in aggregate.

  In order to make global load balancing possible, the load balancer used as the data plane must have sophisticated dynamic configuration capabilities. Please see my posts on Envoy's universal data plane API as well as the service mesh data plane vs. control plane for more information on this topic.

The evolution from hardware to software

So far this post has only briefly mentioned hardware vs. software, primarily in the context of the historical L4 load balancer HA pair. What are the industry trends in this area?

The previous tweet is a humorous exaggeration but still sums up pretty well the trends, which are:

- Historically, routers and load balancers have been provided as extremely expensive proprietary hardware.
- Increasingly, most proprietary L3/L4 networking devices are being replaced with commodity server hardware, commodity NICs, and specialized software solutions built on top of frameworks such as IPVS, DPDK, and fd.io. A modern data center machine that costs less than $5K can easily saturate an 80Gbps NIC with very small packets using Linux and a custom user-space application written using DPDK. Meanwhile, cheap and basic router/switch ASICs that can do ECMP routing at astounding aggregate bandwidths and packet rates are being packaged as commodity routers.

- Sophisticated L7 software load balancers such as NGINX, HAProxy, and Envoy are also rapidly iterating and encroaching on what was previously the domain of vendors like F5. Thus, L7 load balancers are also aggressively moving towards commodity software solutions.
- At the same time, the move towards IaaS, CaaS, and FaaS by the industry as a whole and facilitated by the major cloud providers means that increasingly only a tiny portion of engineers will need to understand how the physical network works (these are the “black magic” and “something we no longer need to know squat about” sections above).

Conclusion and the future of load balancing

To summarize, the key takeaways of this post are:

- Load balancers are a key component in modern distributed systems.
- There are two general classes of load balancers: L4 and L7.
- Both L4 and L7 load balancers are relevant in modern architectures.
- L4 load balancers are moving towards horizontally scalable distributed consistent hashing solutions.
- L7 load balancers are being heavily invested in recently due to the proliferation of dynamic microservice architectures.
- Global load balancing and a split between the control plane and the data plane is the future of load balancing and where the majority of future innovation and commercial opportunities will be found.
- The industry is aggressively moving towards commodity OSS hardware and software for networking solutions. I believe traditional load balancing vendors like F5 will be displaced first by OSS

software and cloud vendors. Traditional router/switch vendors such as Arista/Cumulus/etc. I think have a larger runway in on-premise deployments but ultimately will also be displaced by the public cloud vendors and their homegrown physical networks.

Overall, I think this is a fascinating time in computer networking! The move towards OSS and software for most systems is increasing the pace of iteration by orders of magnitude. Furthermore, as distributed systems continue their march to dynamism via “serverless” paradigms, the sophistication of the underlying network and load balancing systems will need to be commensurately increased.