Engineer

Load balancer

⚖ a server that spreads requests across several others

What it is

A load balancer is a server that receives requests from clients and spreads them across a pool of other servers that all do the same job, so that no single machine carries the whole load and the loss of one does not take the service with it. Configuration files and diagrams often shorten it to LB, and the work it does is called load balancing. Clients see one address, the balancer's. The servers behind it form a pool, which nginx calls an upstream group and HAProxy calls a backend.

Every balancer makes two decisions for each request: which servers are fit to receive it, and which of those gets it. The second is the balancing method. Round robin hands requests to the servers in turn and is what nginx does when nothing else is configured. Least connections sends the next request to the server with the fewest active connections, which evens out the load when some requests take much longer than others; the HAProxy manual recommends it for long sessions such as LDAP or SQL and calls it a poor fit for short ones such as HTTP. Hashing turns a key, such as the client's IP address or the requested URL, into a choice of server, so the same key keeps landing on the same machine. nginx also offers a random method that can pick two servers at random and send the request to the less busy one. Weights tilt the choice toward bigger machines: in the nginx guide's own example, one server with `weight=3` and two at the default weight of 1 receive three, one and one of every five requests.

Balancers also differ in how much of the traffic they read. HAProxy's `mode tcp`, its default, joins client and server in a full-duplex connection and performs no layer 7 examination at all, which suits TLS passed straight through, SSH or SMTP. In `mode http` it parses every request before choosing a server, rejects requests that break the protocol, and can route by path or header. A balancer working at the HTTP level ends the client's connection and opens its own to the server, so the application sees the balancer as its only client unless the balancer passes the original address along, usually in an `X-Forwarded-For` header. That is also why

a load balancer and a reverse proxy are so often the same process: nginx is both, and its own guide describes load balancing as part of its reverse proxy implementation.

Use the concept in a decision

Put “What is a load balancer?” to work in a real decision. We can help relate the concept to the system you are building or buying. Servers sized for what you actually use, moved into the cloud or back out of it, with the monthly bill worked out first.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Ran 20 Docker apps on Google Cloud and AWS — 20 containerised applications deployed and watched

Ran 20 Docker apps on Google Cloud and AWS Get in touch →

Why it matters

One server has a ceiling. When a service outgrows it, a balancer is what lets a team add a second and a third copy instead of buying a bigger machine, and every copy behind it adds capacity without the clients knowing the pool changed. The same arrangement makes maintenance ordinary: take one server out of rotation, update it, check it, put it back, and repeat for the next, while users keep being served by the rest.

It also turns the failure of a server from an outage into a smaller event. When a server stops answering, the balancer stops choosing it, and the remaining servers absorb its share. That only works if they have the headroom to do so. Three servers each running at 80% of their capacity cannot cover for one another; losing one puts 120% of a server’s work on each of the other two.

A balancer changes how the application has to be written. With round robin or least connections, nginx’s guide points out, consecutive requests from one client can reach different servers. Anything a server keeps in its own memory or on its own disk between requests, such as a login session, an uploaded file waiting for processing or a cache of the user’s basket, will be missing whenever the next request lands on another server. The application either keeps that state somewhere every server can reach, a database or a shared cache, or the balancer pins each client to one server, which nginx’s `ip_hash` does except when that server is unavailable.

Last, the balancer itself becomes the one machine every request passes through. A pool of ten application servers behind a single balancer has a single point of failure; it has just moved. A serious setup runs the balancer in a pair with an address that can move between them, or uses a managed balancer whose provider runs the redundancy.

What it looks like in practice

Here an nginx server spreads requests over three application servers, with a fourth held in reserve:

```
upstream app {
    least_conn;
    server 10.0.0.11:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.12:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.13:8080 max_fails=3 fail_timeout=30s weight=2;
    server 10.0.0.20:8080 backup;
}

server {
    listen 80;
    location / {
        proxy_pass http://app;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_next_upstream error timeout http_502 http_503;
        proxy_next_upstream_tries 2;
        proxy_connect_timeout 2s;
    }
}
```

`least_conn` sends each new request to the server with the fewest active connections, and breaks ties by weighted round robin. The third server is the larger machine, so `weight=2` counts it double. `max_fails=3 fail_timeout=30s` says that three failed attempts within 30 seconds take a server out of the pool for 30 seconds, after which nginx tries it again with live requests and puts it back if they succeed. Without those parameters the defaults apply: one failure within 10 seconds is enough. The `backup` server receives requests only while all three primaries are unavailable.

In the `location` , `proxy_next_upstream` decides when a request is handed on to another server: on a connection error, a timeout, or a 502 or 503 response. `proxy_next_upstream_tries 2` caps how often that can happen for one request; the

default, 0, means no limit. The two `proxy_set_header` lines keep the host name the client asked for, which nginx would otherwise replace with the upstream name `app`, and add the client's address to `X-Forwarded-For` so the application can log real visitors instead of the balancer. `proxy_connect_timeout 2s` gives up on a server that has not accepted the connection within two seconds, where the default would wait 60.

These checks are passive: nginx learns that a server is sick only when real requests to it fail. HAProxy can also probe each server on its own schedule. `option httpchk GET /healthz` in a backend, with `check` on each `server` line, sends that request every two seconds by default; three failures in a row mark the server down and two successes bring it back. Without `check`, HAProxy runs no health checks and treats the server as always available.

Before nginx is a balancer it is a web server, and an ordinary one: it is one of the two behind the forty websites administered on Apache and Nginx between 2003 and 2014, which ran on Ubuntu Linux hosting servers and were tuned and maintained to stay reliable under real traffic. The `backup` line has a cousin one level up, where the servers are whole providers. Email built with failover across three providers keeps SendGrid as the primary with SMTP2GO and Azure Communication Services behind it, switches between them automatically, and turned a provider's bad afternoon into a degraded path rather than an outage.

Where it goes wrong

A server that answers every request with a 500 stays in the pool. By default nginx counts only errors in talking to the server, timeouts and malformed response headers as failures. A server whose application has lost its database and returns 500 to everything answers quickly and politely, so the balancer keeps sending it a share of the traffic. The 500, 502, 503, 504 and 429 responses count against a server only when they are listed in `proxy_next_upstream`, and 403 and 404 never do. List the codes that mean "this server is broken", and give the servers an active check against an endpoint that exercises what the application needs to serve a page.

One slow request runs on every server in turn. `timeout` is in the default `proxy_next_upstream`, and `proxy_read_timeout` defaults to 60 seconds. A report that takes 70 seconds to build times out on the first server, is handed to the second, times out there, and moves on to the third, so the client waits three minutes while every server does the same work. A timeout always counts as a

failed attempt, so with the default `max_fails` of 1 each server it touched is also marked unavailable for 10 seconds. Cap the hand-offs with `proxy_next_upstream_tries` and `proxy_next_upstream_timeout`, give the slow endpoint its own `location` with a longer timeout, or move the work off the request path into a background job.

A payment goes through twice. nginx refuses by default to resend a POST, LOCK or PATCH request to another server once it has reached one, because the first server may already have acted on it. Adding `non_idempotent` to `proxy_next_upstream` to make timeouts disappear removes that guard: a payment that the first server completed but answered too slowly is completed again by the second. Leave the option off unless every endpoint behind it is idempotent, for example because it takes an idempotency key and refuses to repeat a request it has seen.

Every visitor appears to come from the balancer. Without the headers in the example, the application logs the balancer's address for every request, and any limit kept per IP address now applies to all users at once. Adding `X-Forwarded-For` brings its own trap. `$proxy_add_x_forwarded_for` appends the client's address to whatever `X-Forwarded-For` the client already sent, so the leftmost entries are whatever the client chose to write. The application should trust only the entry added by its own balancer, and accept the header only from the balancer's address.

One office ends up on one server. nginx's `ip_hash` hashes the first three octets of an IPv4 address, so every client sharing them lands on the same server, and an entire company behind one NAT gateway arrives as a single client. Changing the pool causes the opposite surprise. With a plain `hash`, adding or removing a server can move most keys to a different server, and every user pinned by it loses whatever their old server held. The nginx documentation advises marking a server `down` instead of deleting it from an `ip_hash` group, and offers `hash ... consistent`, which moves only a few keys when the pool changes. Better still, keep session state off the servers, and pinning stops mattering.

Further reading

- Using nginx as HTTP load balancer — the three basic methods, weights, session persistence and the passive health checks, each with a short configuration.

- Module ngx_http_upstream_module — every parameter of a `server` line, including `max_fails`, `backup` and the single-server exception, and the `hash`, `ip_hash`, `least_conn` and `random` methods.
- Module ngx_http_proxy_module — `proxy_next_upstream` and what counts as a failed attempt, the timeouts, and the headers nginx sets on the way to the server.
- HAProxy 3.4 Configuration Manual — `balance` and its algorithms, the TCP and HTTP modes, and active health checks with `check`, `inter`, `rise` and `fall`.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a load balancer?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/load-balancer/