BASICS OF SCALING: LOAD BALANCERS

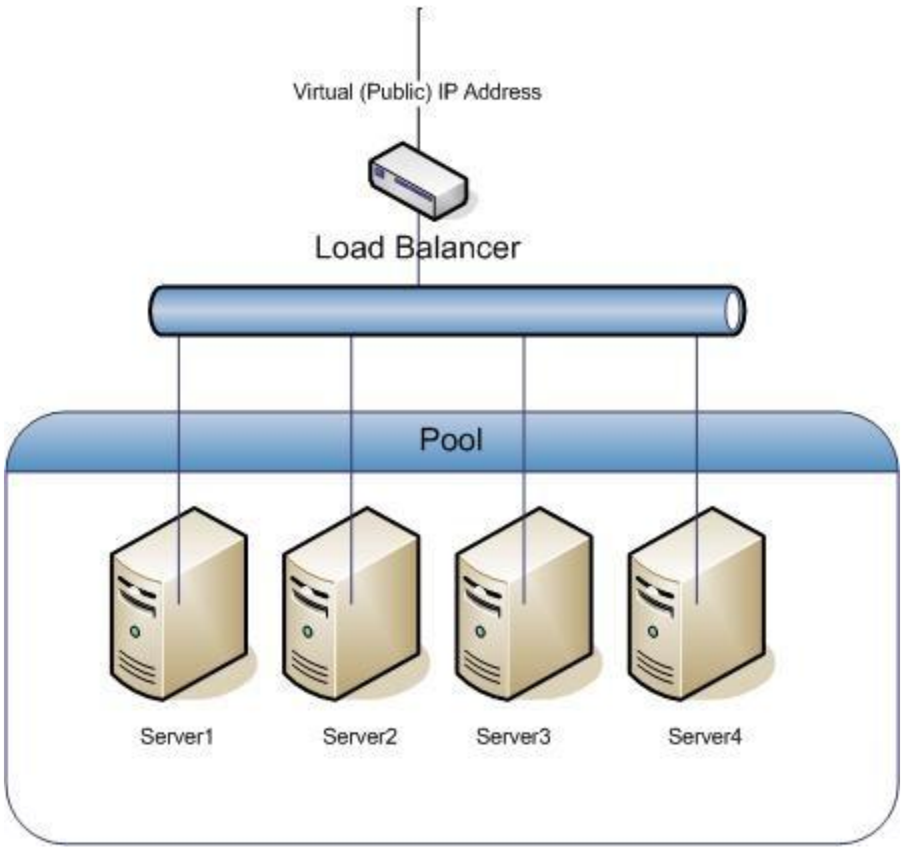
Lately, I’ve been doing a lot of work on systems that require a high degree of scalability to handle large traffic spikes. This has led to a lot of questions from friends and colleagues on scaling, so I thought I’d do a blog series on the basics of scaling .

Why Use A Load Balancer?

One of the most important pieces of a high scale architecture is the load balancer. Load balancers allow you to distribute load (e.g. HTTP requests) between several servers. This is vital as it allows for horizontal expansion – you can increase your capacity (ability to serve *x*number of users) by simply adding more servers. It really depends on your application, but I’ve found that web boxes are frequently the bottleneck initially, caused by your programming language of choice (e.g. Python/PHP/Ruby).

For example, a small PHP application running on a box with Nginx and PHP-FPM isn’t going to have a bottleneck at the database level, but it will be bottlenecked by PHP. Each request is handled by a separate PHP-FPM process, and each PHP-FPM process needs a certain amount of memory. Lets say your server can handle 256 concurrent users. The easiest way to scale at this point is to add more web boxes. Each web box will increase your capacity by 256 users. However, you need a way of distributing visitors between each web box, and that’s where the load balancer comes in.

Traditionally in a web server architecture, your load balancer(s) will sit in front of the rest of your stack, directly accepting requests from your users. All traffic will go through your load balancer(s), and get routed to a web box using a defined scheduling algorithm (e.g. sticky sessions, round robin, least load, etc).



Other benefits of a load balancer include health checks that can remove an unhealthy server from the pool automatically (e.g. if the server isn’t responding, the load balancer will temporarily stop directing traffic to it). You also eliminate your application/web servers as a single point of failure. Servers can fail, and just be removed from the pool without your visitors noticing.

Which Load Balancer?

There are many load balancers available, divided into two categories – hardware and software load balancers. I recommend starting off with a software load balancer, as hardware load balancers are very expensive and designed for larger infrastructure. Software load balancers include multipurpose applications such as Varnish, Nginx and Apache, as well as dedicated applications like HAProxy.

I personally recommend HAProxy as it's a superb piece of software, very fast and reliable. If you're building your infrastructure in AWS, you could also use an ELB, although I don't recommend it as those are closed systems and provide limited customization.

Your load balancer should go on a dedicated server, to maximize reliability and resources.

Which Scheduling Algorithm?

There are three common scheduling algorithms employed by load balancers, and you can traditionally configure which one you'd like to use.

Sticky sessions use cookies to ensure a user always hits the same backend server. If your application currently stores state information about a request/user on the server (not a database or other source), this can be useful to add load balancing to your application without making code changes, but I don't recommend it as it can put too much load on one server.

Round robin sends each successive request to the next server in the pool, e.g. A -> B -> C -> D -> A. This is the most common load balancing technique, and is normally fine for most websites.

Load scheduling directs requests to the server with the lowest load (e.g. CPU load). This is the best load balancing technique and is still quite easy to setup. Not all load balancers support this however.

Application Compatability

Unless you use sticky sessions, you'll probably have to make changes to your application to be compatible with a load balancing system.

The biggest thing that will normally cause issues is stateful information being stored on the server. A great example of this is user sessions. PHP uses file based sessions by default, so a user may login on server A, and there next request goes to server B which doesn't have their session info on it. Luckily, PHP and most other languages and/or frameworks support alternative session storage mechanisms like

Memcached or storing sessions in your database. I most frequently use Memcached based sessions.

Any other state information that gets saved to a single server will need to be adapted to be stored in a shared location or replicated amongst servers.

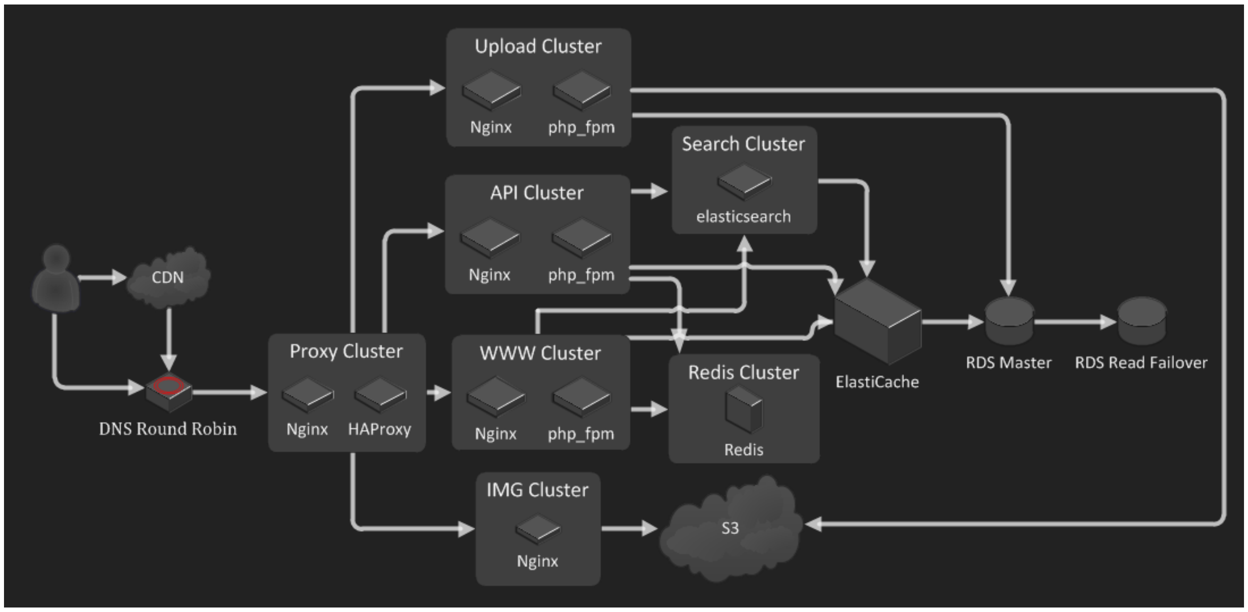
Automatic Scaling

A logical next step is adding auto scaling to your web/application server cluster. In this system, you would have a management server of some sort that monitors your web server cluster, and automatically adds or removes servers based on load. This works best with a virtualization infrastructure or cloud based hosting such as Amazon AWS. Your management server would then add or remove the instance from your load balancer.

Again, if you’re using Amazon AWS, they have this built in. It’s called Auto Scaling groups, and works well with Amazon ELBs (elastic load balancers).

Closing Notes

Most large and/or complex architectures will include load balancers at several points throughout the infrastructure, and will normally use a cluster of load balancers at each point to avoid having a single point of failure. Certain DNS providers will allow you to specify multiple IP addresses for a domain name, and it will either return the closest IP to the user, or use a round robin approach (this is what Google does). If you have 10 web servers, but 1 load balancer, you have a single point of failure that can take down everything. Load balancers are traditionally quite stable, but you should always avoid single points of failure if you want a robust architecture.



Imgur's server architecture, pictured above, uses round robin DNS, multiple server clusters, and a cluster of load balancers (shown here as the "proxy cluster").

Source: http://brandonwamboldt.ca/basics-of-scaling-load-balancers-1432/