CHAPTER 8

Load Balancing with Nginx

So far in this book, you have seen the power of Nginx as a web server. However, Nginx is much more than that. It is frequently used as a proxy server. A proxy server's job is to front end the request and pass it on to the proxied server, which is also known as an upstream server. The upstream server processes the request and sends the response back to the Nginx server, which further relays the response to the clients who made the request. You may think, why do you need to complicate things as much? Wouldn't it make the processing slower because of the number of hops? This chapter focuses on answers to similar questions, and you will learn about setting up servers based on different scenarios.

Defining High Availability

Let's begin with the fundamentals of high availability. A system can be considered highly available if it is continuously operational as long as desired. That is easier said than done. There are basically three main aspects that you need to consider while designing a highly available system.

1. Eliminate a single point of failure. In simple words, it means that you should design your system such that failure at a specific point doesn't bring down the entire system. In a web server context, it means that having just one server, serving your requests for `www.yoursite.com` is not recommended. Even though you can have multiple worker processes for Nginx, it doesn't take care of scenarios where a server has to be patched for security and rebooted. Also, it doesn't take care of hardware or network failures. Having a single server serving your web pages, hence, can become a single point of failure and should be avoided.

2. Reliable failover. In simplistic terms, a failover implies that if one of the servers goes down, another one takes its place without the end user noticing the disruption in service. Consider a car. If one of the tires gets punctured, even though you have an extra tire , it takes a while to manually change it. You can say that the failover is reliable, but it isn't quick. In today's world, for a busy e-commerce website, slow failover means revenue loss that could run in millions. It is to be noted that the revenue loss here is not only from the lost business, but also due to the lack of trust that ensues following a major failure.

3. Failure detection at run time. No system is 100 percent perfect. Although, if monitored well, it can appear to be 100 percent reliable. High-availability design suggests that you create the system in such a way that failure could be detected and fixed with time in hand. However, it is easier said than done.

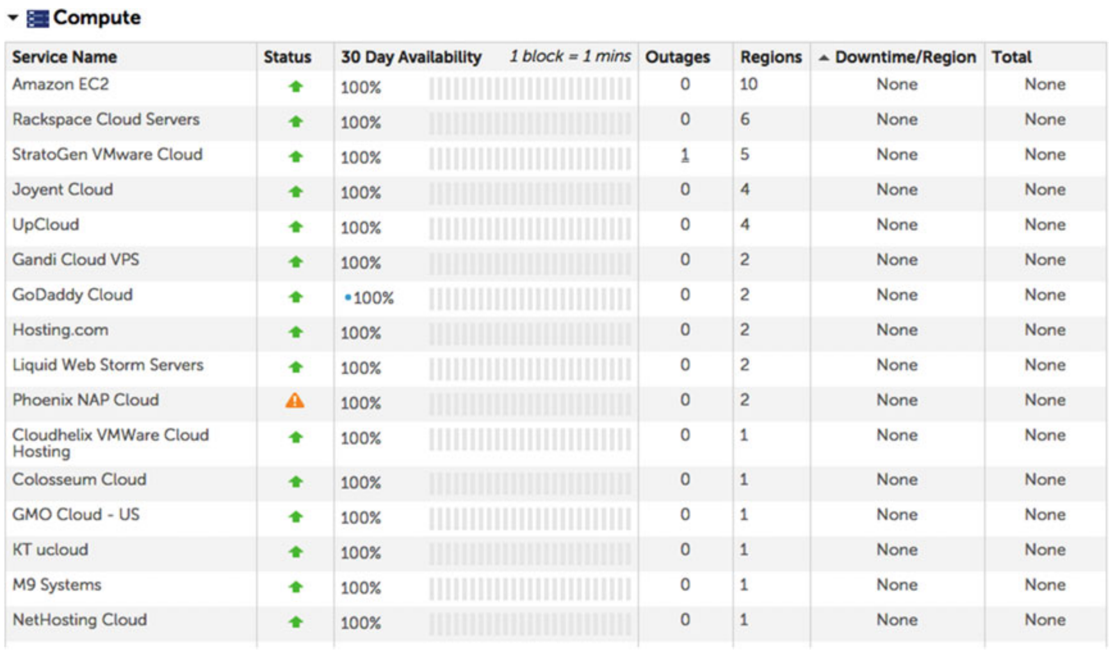
© Rahul Soni 2016
R. Soni, *Nginx*, DOI 10.1007/978-1-4842-1656-9_8

All the three points mentioned above sound pretty obvious, but designing such a system is hard. In fact, very hard! As an engineer or architect, you will be often asked to maintain a Service Level Agreement (SLA) in terms of percentage of uptime. Take a look at Table 8-1.

Table 8-1. *Service Level Agreement Chart*

Availability %	Downtime per year	Downtime per day
99% (two nines)	3.65 days	14.4 mins
99.9% (three nines)	8.76 hours	1.44 mins
99.99% (four nines)	52.56 mins	8.66 seconds
99.999% (five nines)	5.26 mins	864.3 milliseconds
99.9999% (six nines)	31.5 seconds	86.4 milliseconds

That's right: it is just 31.5 seconds per year for a six niner SLA. In today's world, 4 nines and 5 nines have almost become a norm for major cloud providers, and 100 percent is not farfetched either. See Figure 8-1.



Compute

Service Name	Status	30 Day Availability *1 block = 1 mins*	Outages	Regions	▲ Downtime/Region	Total
Amazon EC2	⬆	100%	0	10	None	None
Rackspace Cloud Servers	⬆	100%	0	6	None	None
StratoGen VMware Cloud	⬆	100%	1	5	None	None
Joyent Cloud	⬆	100%	0	4	None	None
UpCloud	⬆	100%	0	4	None	None
Gandi Cloud VPS	⬆	100%	0	2	None	None
GoDaddy Cloud	⬆	•100%	0	2	None	None
Hosting.com	⬆	100%	0	2	None	None
Liquid Web Storm Servers	⬆	100%	0	2	None	None
Phoenix NAP Cloud	⚠	100%	0	2	None	None
Cloudhelix VMWare Cloud Hosting	⬆	100%	0	1	None	None
Colosseum Cloud	⬆	100%	0	1	None	None
GMO Cloud - US	⬆	100%	0	1	None	None
KT ucloud	⬆	100%	0	1	None	None
M9 Systems	⬆	100%	0	1	None	None
NetHosting Cloud	⬆	100%	0	1	None	None

Figure 8-1. *Service status by different cloud providers (courtesy:* `https://cloudharmony.com/status-group-by-regions`*)*

Load Balancing for High Availability

When you manage web servers, one of your primary duties is to maintain a specific SLA. As discussed in the previous section, keeping a server running 24x7x365 becomes crucial. At the same time, patching the server regularly is equally important. Sometimes, patching requires you to reboot the server before the settings take effect. During the rebooting process, the website remains down. Ask yourself, what if the server doesn't come up after the reboot or a system failure occurs due to hardware? Scary, right? To avoid such scenarios, you set up a load balancer. Figure 8-2 shows what the network would look like.

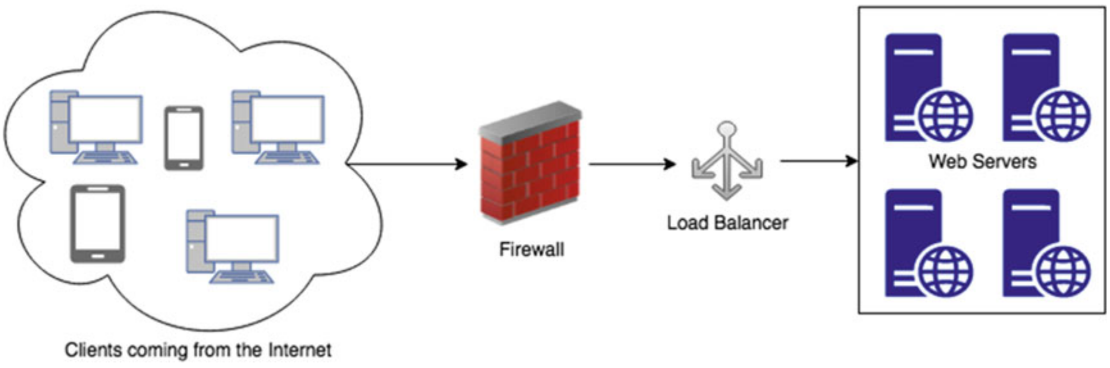


Figure 8-2. *A typical network load balancer in action*

As you can see, this setup doesn't have web server as a single point of failure any more. You now have four web servers handling the load from various clients. It is the responsibility of the load balancer to appropriately route the Internet traffic to one of the web servers, retrieve the response from the web server, and return it back to the client. It takes care of bullet #1 discussed in the high-availability section.

Regarding bullet #2, if your application is designed in such a way that the entire application is deployed in one web server, you are good to go and will not have to worry about failover. This is because, if one server goes down, the Network Load Balancer (NLB) will detect it, and won't route the traffic to the server that is down. Quite often, the "entire application in one server" is not practical and the servers have to be created based on the role that it serves. So, in effect, you will have a set of database servers, another set of application servers, and yet another set of front-end servers. The front end in this case doesn't have a single point of failure (the rest of the system has not been drawn in the figure for brevity). The idea is to avoid single point of failures at all levels.

Bullet #3, that is, failure detection, is what helps the load balancer determine which server is down. Ideally, you should have a monitoring solution that keeps monitoring the logs, service levels, and other server vitals to ensure everything runs smoothly in a production farm. The monitoring solution typically has an alerting mechanism that alerts the administrators when the server starts showing signs of stress.

■ **Note** The network diagram in Figure 8-2 might look robust, but it is not. That is because the load balancer and firewall has now become a single point of failure. You can design them to be fault tolerant by making them failover smoothly in case of issues. However, this won't be covered in this book since it is out of its scope.

Hardware Load Balancer

As the name suggests, a hardware load balancer is a device that is installed in a datacenter and does the job of splitting traffic. Since the decision is made at the electronics level, it happens to be extremely fast. Moreover, the failure rates are low, too. There are multiple vendors like F4, Cisco, Citrix, etc., who provide the hardware. Typically, the configuration is done through a console or a web interface.

Some benefits of using a hardware load balancer include the following:

- It helps generate excellent statistics out of the box. Since the devices are made primarily for one purpose, they do it really well. It is a mature solution and comes in various flavors for different needs.
- You can rely on a specific vendor and call for support when the need arises. In worst-case scenarios, you have to simply replace the device with a new one and reconfigure from backup.
- Lower maintenance costs since the appliance just works, and there is not much to manage once it is configured.

There are some disadvantages too. Here are a few important ones:

- Not all devices have lower maintenance costs, since every device has its nuances. You might need to hire a consultant or employee who understands these devices well. That eventually bumps up the cost of ownership.
- Hardware load balancers are mostly black boxes, and you can only do as much as the console or the API allows. Beyond that, you might have to evaluate another device that has bigger/better feature sets.
- The devices are generally quite costly, and needs you to have a datacenter that you control. In today's world, a lot is being migrated to the cloud and if you have a solution that is deployed mostly in the cloud, a hardware NLB is out of the race.

Software Load Balancer

With time, the load balancers have evolved. They are often referred to as *Application Delivery Controllers* (ADC) and they do much more than traffic routing. The load balancer is not just a black box as it once was, and it has to be a lot more scalable due to the goals of today's massively scalable applications. The future is software, and right from servers, to switches, firewalls, routers, and load balancers, the inclination has been toward a software solution due to various reasons. With the advent of cloud computing this inclination is turning even more toward a software solution. Primarily, it helps in the long run if you are not stuck with a proprietary hardware and its limitations.

Nginx can help you load balance your traffic and much more. There are some very good reasons why you should use Nginx as a software load balancer.

Flexibility

This is the most important reason why you may want to use a software load balancer. Please keep in mind that installing a hardware load balancer requires a lot more work than a software load balancer. A software load balancer can be used anywhere including containers, hypervisors, commodity hardware, and even in the cloud!

Cost

Nginx provides a software-based application delivery platform that load balances HTTP and TCP applications at a fraction of the cost compared to hardware solutions. The open source version is free, and the paid version Nginx PLUS offers 24x7 support at a much lower cost factor.

Sizing

You buy what you need in case of a software LB, whereas, you have to appropriately size for the requirements and often keep an additional buffer for growth. In effect, when it comes to hardware LB, you buy more than what you need to begin with. Sizing correctly is not an easy task and you often undersize or oversize, and this leads to complications in the running deployment. Even if you have purchased it just right, you will have to make the payment up front for the need that you might have had *after* three or five years, based on your initial estimate.

Application vs. Network

The guiding force behind the purchase of a hardware LB is often the network setup. In comparison, software LB is often preferred if you think more about your application and its scalability. This becomes even more important if your application is modern and geared toward cloud deployment.

Elasticity

A software LB is elastic in the sense that you can easily provision bigger servers or spawn additional ones during the spikes. With hardware LB, you will always have to consider the spikes in your sizing calculation and buy appropriately. In effect, it means more resources blocked than you would need on average, and you will end up paying for the resources that you never really utilized.

Consider Figure 8-3. Assume that the average number of hits you anticipate is around 5K per minute throughout the year. Due to a marketing campaign or any other reason you happened to see a spike that took the number of requests to 8K+ per minute. In this case, a hardware LB might start throttling or might even fail to respond appropriately. The solution is to buy the device appropriately considering the maximum throughput that you anticipate. This also implies that you will end up paying for a device that handles more concurrent requests (8K per min), whereas your average is much less (just 5K per minute).

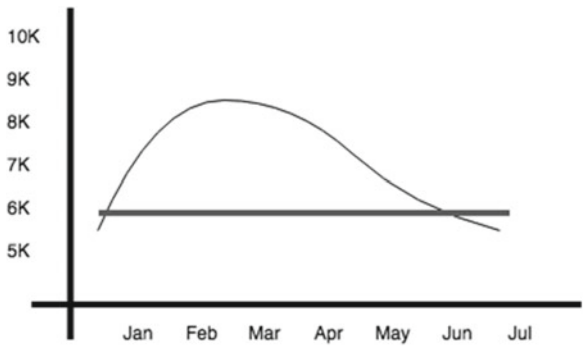


Figure 8-3. *Sales spike between Jan to May*

Easy Deployment

Setting up a software LB is much simpler and easier than hardware LB. Additionally, a software LB is same whether it runs on a bare metal server, virtual server, container, or cloud. The functionality and configuration method doesn't change. This is not the case with hardware load balancers. Every device is different and has different requirements and capacities. Maintenance requires specific knowledge of the given hardware and the variety of choices makes it even more difficult to evaluate different devices.

Multi-Tenancy

If you have multiple applications, buying a different hardware LB for each application might be too expensive. To counter that, sharing the LB between multiple applications also means that one noisy application can negatively impact others. Software LB can be easily multi-tenanted, and it turns out to be a lot more effective in the long run since it doesn't suffer from a *noisy-neighbor* issue.

Load Balancing in Nginx

Now that you have learned about the basics of load balancing and advantages of using a software load balancer, let's move forward and work on the Nginx servers you already created in the previous chapters.

Clean Up the Servers

Before setting up anything new, clean up the previous applications so that you can start afresh. This is to keep things simpler. You will be settings up applications in different ways in the upcoming sections of this chapter. The idea is to give you information about different scenarios from a practical perspective.

1. Log on to the WFE1 using `ssh -p 3026` user1@127.0.0.1
2. Remove everything from the Nginx home directory.

```
sudo rm -rf /usr/share/nginx/html/*
```

3. Reset your configuration (`sudo vi /etc/nginx/nginx.conf`) to the following:

```
user  nginx;
worker_processes  1;
error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}
```

```
http {
    include           /etc/nginx/mime.types;
    default_type      application/octet-stream;
    log_format        main  '$remote_addr - $remote_user - [$time_local] - $document_root -
    $document_uri - '
                      '$request - $status - $body_bytes_sent - $http_referer';

    access_log        /var/log/nginx/access.log  main;
    sendfile          on;
    keepalive_timeout  65;
    index               index.html index.htm;
    include /etc/nginx/conf.d/*.conf;
}
```

4. Now, remove the entries in conf.d by using the following command:

```
sudo rm -f /etc/nginx/conf.d/*.conf
```

5. Repeat the steps for WFE2.

Create Web Content

Let's create some content so that it is easy to identify which server served the request. In practical situations, the content on the WFE1 and WFE2 will be same for the same application. Run the following command on both WFE1 and WFE2:

```
uname -n | sudo tee /usr/share/nginx/html/index.html
```

This command is pretty straightforward. It uses the output of `uname -n` and dumps it in a file called index.html in the default root location of Nginx. View the content and ensure that the output is different on both the servers.

```
$cat /usr/share/nginx/html/index.html
wfe1.localdomain
```

■ **Tip** The tee command reads from the standard input and writes to standard output as well as files. It is handy, since it shows you the output along with creating the file at the same time.

Configure WFE1 and WFE2

The content is available on both servers now, but since you have already cleaned up the configuration you will need to re-create the configuration file by using the following command:

```
sudo cp /etc/nginx/conf.d/default.template /etc/nginx/conf.d/main.conf
```

The command will create a copy of the configuration for a default website. If you recall, the default. template contained the following text:

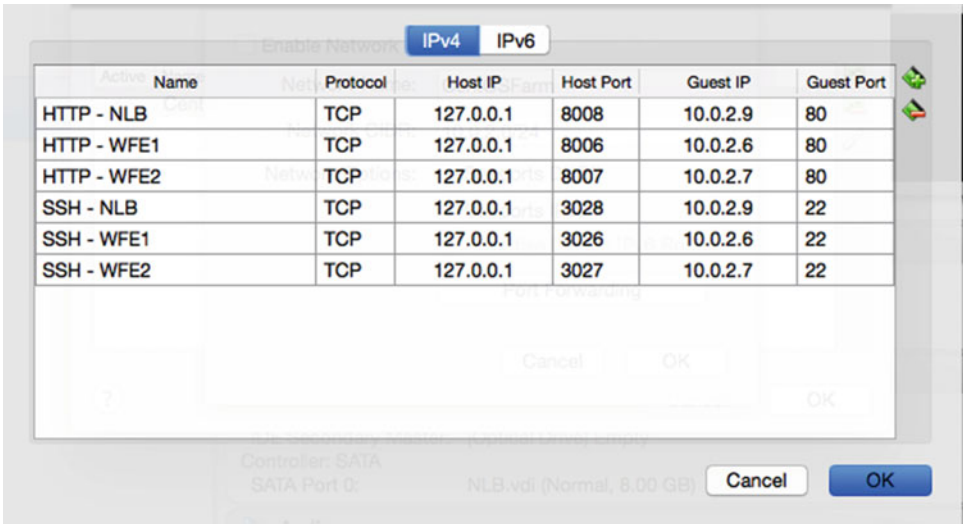
```
server {
    listen       80;
    server_name  localhost;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
}
```

- Restart the service: `sudo systemctl restart nginx.`
- Repeat the steps on WFE2.
- Once done, you should be able to execute `curl localhost` on both servers, and you should get output as `wfe1.localdomain` and `wfe2.localdomain` respectively. Notice that even though the request is same (`curl localhost`), the output is different. In practice, the output will be the same from both servers.

Set Up NLB Server

Setting up an NLB server is no different than setting up a regular web server. The installation steps are similar to what you have learned already. The configuration, however, is different and you will learn about it in the upcoming sections.

1. Create a new virtual machine called NLB.
2. Set up a NAT configuration as you have learned in previous chapters. It should look similar to Figure 8-4.



Name	Protocol	Host IP	Host Port	Guest IP	Guest Port
HTTP - NLB	TCP	127.0.0.1	8008	10.0.2.9	80
HTTP - WFE1	TCP	127.0.0.1	8006	10.0.2.6	80
HTTP - WFE2	TCP	127.0.0.1	8007	10.0.2.7	80
SSH - NLB	TCP	127.0.0.1	3028	10.0.2.9	22
SSH - WFE1	TCP	127.0.0.1	3026	10.0.2.6	22
SSH - WFE2	TCP	127.0.0.1	3027	10.0.2.7	22

Figure 8-4. *Network configuration for NLB with two WFEs*

3. Install Nginx (refer to chapter 2) on the NLB server.

4. Since it is a new server, when you execute `curl localhost`, you will see the default welcome page. You can ignore it for the time being.

5. Open the configuration file (`/etc/nginx/conf.d/default.conf`) and make the changes as follows:

```
upstream backend{
        server 10.0.2.6;
        server 10.0.2.7;
}

server {
    listen       80;
    location / {
        proxy_pass http://backend;
}
```

6. Restart the service.

7. Try the following command a few times and notice how it gives you output from WFE1 and WFE2 in an alternate fashion.

```
[root@nlb ~]# curl localhost
wfe1.localdomain
[root@nlb ~]# curl localhost
wfe2.localdomain
[root@nlb ~]# curl localhost
wfe1.localdomain
[root@nlb ~]# curl localhost
wfe2.localdomain
```

So, what just happened? Basically, you have set up a load balancer using Nginx and what you saw was the load balancer in action. It was extremely simple, right? There are a couple of directives at play here.

- upstream directive: The upstream directive defines a group of servers. Each server directive points to an upstream server. The server can listen on different ports if needed. You can also mix TCP and UNIX-domain sockets if required. You will learn more about it in the upcoming scenarios.
- proxy_pass directive: This directive sets the address of a proxied server. Notice that in this case, the address was defined as back end, and in turn contained multiple servers. By default, if a domain resolves to several addresses, all of them will be used in a round-robin fashion.

Load Balancing Algorithms

When a load balancer is configured, you need to think about various factors. It helps if you know the application and its underlying architecture. Once you have found the details, you will need to configure some parameters of Nginx so that you can route the traffic accordingly. There are various algorithms that you can use based on your need. You will learn about it next.

Round Robin

This is the default configuration. When the algorithm is not defined, the requests are served in round-robin fashion. At a glance, it might appear way too simple to be useful. But, it is actually quite powerful. It ensures that your servers are equally balanced and each one is working as hard.

Let's assume that you have two servers, and due to the nature of your application you would like three requests to go to the first server (WFE1) and one request to the second server (WFE2). This way, you can route the traffic in a specific ratio to multiple servers. To achieve this, you can define `weight` to your server definitions in the configuration file as follows.

```
upstream backend{
        server 10.0.2.6 weight=3;
        server 10.0.2.7 weight=1;
}

server {
    listen       80;
    location / {
        proxy_pass http://backend;
    }
}
```

Reload Nginx configuration and try executing `curl localhost` multiple times. Note that three requests went to the WFE1 server, whereas one request went to WFE2.

```
[root@nlb ~]# curl localhost
wfe1.localdomain
[root@nlb ~]# curl localhost
wfe1.localdomain
[root@nlb ~]# curl localhost
wfe1.localdomain
[root@nlb ~]# curl localhost
wfe2.localdomain
```

Least Connected, Optionally Weighted

In scenarios where you cannot easily determine the ratio or weight, you can simply use the least connected algorithm. It means that the request will be routed to the server with the least number of active connections. This often leads to a good load-balanced performance. To configure this, you can use the configuration file like so:

```
upstream backend{
        least_conn;
        server 10.0.2.6 weight=1;
        server 10.0.2.7 weight=1;
}
```

Without a load testing tool, it will be hard to determine the output using command line. But the idea is fairly simple. Apart from the least number of active connections, you can also apply weight to the servers, and it would work as expected.

IP Hash

There are quite a few applications that maintain state on the server: especially the dynamic ones like PHP, Node, ASP.NET, and so on. To give a practical example, let's say the application creates a temporary file for a specific client and updates him about the progress. If you use one of the round-robin algorithms, the subsequent request might land on another server and the new server might have no clue about the file processing that started on the previous server. To avoid such scenarios, you can make the session sticky, so that once the request from a specific client has reached a server, Nginx continues to route the traffic to the same server. To achieve this, you must use `ip_hash` directive like so:

```
upstream backend{
        ip_hash;
        server 10.0.2.6;
        server 10.0.2.7;
}
```

The configuration above ensures that the request reaches only one specific server for the client based on the client's IP hash key. The only exception is when the server is down, in which case the request can land on another server.

Generic Hash

A hash algorithm is conceptually similar to an IP hash. The difference here is that for each request the load balancer calculates a hash that is based on the combination of text and Nginx variables that you can specify. It sends all requests with that hash to a specific server. Take a look at the following configuration where hash algorithm is used with variables $scheme (for http or https) and $request_uri (URI of the request):

```
upstream backend{
        hash $scheme$request_uri;
        server 10.0.2.6;
        server 10.0.2.7;
}
```

Bear in mind that a hash algorithm will most likely not distribute the load evenly. The same is true for an IP hash. The reason why you still might end up using it is because of your application's requirement of a sticky session. Nginx PLUS offers more sophisticated configuration options when it comes to session persistence. The best use case for using hash is probably when you have a dynamic page that makes data intensive operations that are cachable. In this case, the request to that dynamic page can go to one server only, which caches the result and keeps serving the cache result, saving the effort required at the database side and on all the other servers.

Least Time (Nginx PLUS), Optionally Weighted

Nginx PLUS has an additional algorithm that can be used. It is called the least time method where the load balancer mathematically combines two metrics for each server—the current number of active connections and a weighted average response time for past requests —and sends the request to the server with the lowest value. This is a smarter and more effective way of doing load balancing with heuristics.

You can choose the parameter on the least_time directive, so that either the time to receive the response header or the time to receive the full response is considered by the directive. The configuration looks like so:

```
upstream backend{
        least_time (header | last_byte);
        server 10.0.2.6 weight=1;
        server 10.0.2.7 weight=1;
}
```

Most Suitable Algorithm

There is no silver bullet or straightforward method to tell you which method will suit you best. There are plenty of variables that need to be carefully determined before you choose the most suitable method. In general, least connections and least time are considered to be best choices for the majority of the workloads.

Round robin works best when the servers have about the same capacity, host the same content, and the requests are pretty similar in nature. If the traffic volume pushes every server to its limit, round robin might push all the servers over the edge at roughly the same time, causing outages.

You should use load testing tools and various tests to figure out which algorithm works best for you. One thing that often helps you make good decision is the knowledge of the application's underlying architecture. If you are well aware about the application and its components, you will be more comfortable in doing appropriate capacity planning.

You will learn about load testing tools, performance, and benchmarking in the upcoming chapters.

Load Balancing Scenarios

So far in this chapter you have seen an Nginx load balancer routing to the back-end Nginx servers. This is not a mandatory requirement. You can choose Nginx to route traffic to any other web server. As a matter of fact, that is what is done mostly in practical scenarios and as far as the request is HTTP based, it will just work. Nginx routes the request based on the mapped URI. You can use Nginx easily to front end the PHP, ASP. NET, Node.js, or any other application for that matter and enjoy the benefits of Nginx as you will see in the upcoming scenarios.

Nginx Routing Request to Express/Node.js

If you recall, in the previous chapter you configured Nginx for MEAN stack. Assuming WFE1 and WFE2 are hosting applications based on MEAN stack and the application is running on port 3000, your NLB server's configuration will look like the following:

```
upstream nodeapp {
        server 10.0.2.6:3000;
        server 10.0.2.7:3000;
}

server {
    listen       80;
    server_name  localhost;

    location / {
       proxy_pass http://nodeapp;
    }
}
```

A common mistake that usually happens is that the additional ports are not opened in the firewall. So, you need to ensure that ports are opened explicitly by using the following command on both WFE1 and WFE2:

```
[user1@wfe1 ~]$ sudo firewall-cmd --permanent --add-port=3000/tcp
success
[user1@wfe1 ~]$ sudo firewall-cmd --reload
success
```

Once you have opened the ports, Nginx will start routing the request successfully. Note that the opened ports are not exposed to the Internet. It is just for Nginx that is load balancing the requests.

Passing the HOST Header

Since everything has been working in these simple demos, it might mislead you into thinking that all you need to pass to the back-end server is the URI. For real world applications you might have additional information in request headers that—if missed—will break the functionality of the application. In other words, the request coming from Nginx to the back-end servers will look different than a request coming directly from the client. This is because Nginx makes some adjustments to headers that it receives from the client. It is important that you are aware of these nuances.

- Nginx gets rid of any empty headers for performance reasons.
- Any header that contains an underscore is considered invalid and is eventually dropped from the headers collection. You can override this behavior by explicitly setting `underscores_in_headers on`;
- The "HOST" header is set to the value of $proxy_host, which is a variable that contains the domain name of IP address grabbed from the proxy_pass definition. In the configuration that follows, it will be `backend`.
- Connection header is added and set to close.

You can tweak the header information before passing on by using the proxy_set_header directive. Consider the following configuration in the NLB:

```
upstream backend{
        server 10.0.2.6;
        server 10.0.2.7;
}

server {
    listen       80;

    location / {
        proxy_set_header HOST $host;
        proxy_pass http://backend;
    }
}
```

In the previous configuration, an explicit HOST header has been set using proxy_set_header directive. To view the effect, follow these steps:

- Ensure that your NLB configuration appears as the previous configuration block. Restart Nginx service.
- On WFE1, change the nginx.conf (`sudo vi /etc/nginx/nginx.conf`) such that the log_format has an additional field called $host as follows:

```
log_format         main  '$host - $remote_addr - $remote_user - [$time_local] - $document_
root - $document_uri - $request - $status - $body_bytes_sent - $http_referer';
```

- Save the file and exit. Restart Nginx service.
- Switch back to NLB and make a few requests using `curl localhost`
- View the logs on the WFE1 using `sudo tail /var/log/nginx/access.log -n 3`.

```
[user1@wfe1 ~]$ sudo tail /var/log/nginx/access.log -n 3
localhost - 10.0.2.9 - - - - /usr/share/nginx/html - /index.html - GET / HTTP/1.0 - 200 - 17 - -
localhost - 10.0.2.9 - - - - /usr/share/nginx/html - /index.html - GET / HTTP/1.0 - 200 - 17 - -
localhost - 10.0.2.9 - - - - /usr/share/nginx/html - /index.html - GET / HTTP/1.0 - 200 - 17 - -
```

- As you can see, the requests had localhost as the hostname and it is because you have used proxy_set_header HOST $host.
- To view what the result would have looked like without this header change, comment the line in NLB's configuration:

```
    location / {
        # proxy_set_header HOST $host;
       proxy_pass http://backend;
    }
```

- Restart Nginx on NLB and retry curl localhost a few times.
- If you view the logs on WFE1 using the tail command, you should see an output similar to this:

```
localhost - 10.0.2.9 - - - - /usr/share/nginx/html - /index.html - GET / HTTP/1.0 - 200 - 17 - -
backend - 10.0.2.9 - - - - /usr/share/nginx/html - /index.html - GET / HTTP/1.0 - 200 - 17 - -
backend - 10.0.2.9 - - - - /usr/share/nginx/html - /index.html - GET / HTTP/1.0 - 200 - 17 - -
```

- Notice the last couple of lines where the hostname appears as back end. This is the default behavior of Nginx if you don't set the HOST header explicitly. Based on your application, you might need to set explicitly or ignore this header in the NLB configuration.

Forwarding IP Information

Since the requests are forwarded to the back end, it has no information about where the requests have actually come from. To the back-end servers, it knows the NLB as the client. There are scenarios where you might want to log information about the actual visitors. To do that, you can use proxy-set-header just as you did in the previous example but with different variables like so:

```
location / {
    proxy_set_header HOST $proxy_host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_pass http://backend;
}
```

In this configuration apart from setting HOST header, you are also setting the following headers:

- X-Real-IP is set to $remote_addr variable that contains the actual client IP.
- X-Forwarded-For is another header set here, which contains $proxy_add_x_forwarded_for. This variable contains a list of $remote_addr - client IPs - separated by a comma.
- To log the actual client IP, you should now modify the log_format to include $http_x_real_ip variable that contains the real client IP information.
- By default, X-Real-IP is stored in $http_x_real_ip. You can change this behavior by using - `real_ip_header X-Forwarded-For;` - in your http, location or server directive in order to save the value of X-Forward-For header instead of X-Real-IP header.

Buffering

As you can guess, with an NLB in between the real back-end server, there are two hops for every request. This may adversely affect the client's experience. If the buffers are not used, data that is sent from the back-end server immediately gets transmitted to the client. If the clients are fast, they can consume this immediately and buffering can be turned off. For practical purposes, the clients will typically not be as fast as the server in consuming the data. In that case, turning buffering on will tell Nginx to hold the back-end data temporarily, and feed that data to the client. This feature allows the back ends to be freed up quickly since they have to simply work and ensure that the data is fed to Nginx NLB. By default, buffering is on in Nginx and controlled using the following directives:

- proxy_buffering: Default value is on, and it can be set in http, server, and location blocks.
- proxy_buffers *number size*: proxy_buffers directive allows you to set the number of buffers along with its size for a single connection. By default, the size is equal to one memory page, and is either 4K or 8K depending on the platform.
- proxy_buffer_size *size*: The headers of the response are buffered separately from the rest of the response. This directive sets that size, and defaults to proxy_buffers size.
- proxy_max_temp_file_size *size*: If the response is too large, it can be stored in a temporary file. This directive sets the maximum size of the temporary file.
- proxy_temp_file_write_size *size*: This directive governs the size of data written to the file at a time. If you use 0 as the value, it disables writing temporary files completely.
- proxy_temp_path *path*: This directive defines the directory where temporary files are written.

Nginx Caching

Buffering in Nginx helps the back-end servers by offloading data transmission to the clients. But the request actually reaches the backend server to begin with. Quite often, you will have static content, like 3rd party JavaScript libraries, CSS, Images, PDFs, etc. that doesn't change at all, or rarely changes. In these cases, it makes sense to make a copy of the data on the NLB itself, so that the subsequent requests could be served directly from the NLB instead of fetching the data every time from the backend servers. This process is called caching.

To achieve this, you can use the proxy_cache_path directive like so in the HTTP block:

```
proxy_cache_path path levels=1:2 keys_zone=my_cache:10m max_size=10g inactive=60m
```

Before you use this directive, create the path as follows and set appropriate permissions:

```
mkdir -p /data/nginx/cache
chown nginx /data/nginx/cache
chmod 700 /data/nginx/cache
```

- Levels define the number of subdirectories Nginx will create to maintain the cached files. Having a large number of files in one flat directory slows down access, so it is recommended to have at least a two-level directory hierarchy.
- keys_zone defines the area in memory which contains information about cached file keys. In this case a 10MB zone is created and it should be able to hold about 80,000 keys (roughly).

- max_size is used to allocate 10GB space for the cached files. If the size increases, cache manager process trims it down by removing files that were used least recently.
- inactive=60m implies the number of minutes the cache can remain valid in case it is not used. Effectively, if the file is not used for 60 minutes, it will be purged from the cache automatically.

By default, Nginx caches all responses to requests made with the HTTP GET and HEAD methods. You can cache dynamic content too where the data is fetched from a dynamic content management system, but changes less frequently, using `fastcgi_cache`. You will learn about caching details in chapter 12.

Server Directive Additional Parameters

The server directive has more parameters that come in handy in certain scenarios. The parameters are fairly straightforward to use and simply require you to use the following format:

```
server address [parameters]
```

You have already seen the server address in use with weight. Let's learn more about some additional parameters.

- max_fails=number: Sets the number of unsuccessful attempts before considering the server unavailable for a duration. If this value is set to 0, it disables the accounting of attempts.
- fail_timeout=time: Sets the duration in which max_fails should happen. For example, if max_fails parameter is set to 3, and fail_timeout is set to 10 seconds, it would imply that there should be 3 failures in 10 seconds so that the server could be considered unavailable.
- backup: Marks the server as a backup server. It will be passed requests when the primary servers are unavailable.
- down: Marks the server as permanently unavailable.
- max_conns=number: Limits the maximum number of simultaneous active connections. Default value of 0 implies no limit.

Configure Nginx (PLUS) for Heath Checks

The free version of Nginx doesn't have a very important directive, and it is called health_check. This feature is available in Nginx PLUS, and enabling it gives you a lot of options related to health of the upstream servers.

- `interval=time`: Sets the interval between two health checks. The default value is 5 seconds and it implies that the server checks the upstream servers every 5 seconds.
- `fails=number`: If the upstream server fails x number of times, it will be considered unhealthy. The default value is 1.
- `passes=number`: Once considered unhealthy, the upstream server needs to pass the test x number of times before it could be considered healthy. The default value is 1.
- `uri = path`: Defines the URI that is used to check requests. Default value is /.

- `match=name`: You can specify a block with its expected output in order the test to succeed. In the following configuration, the test is to ensure that the output has a status code of 200, and the body contains "Welcome to nginx!"

```
http {
    server {
        location / {
            proxy_pass http://backend;
            health_check match=welcome;
        }
    }

    match welcome {
        status 200;
        header Content-Type = text/html;
        body ~ "Welcome to nginx!";
    }
}
```

- If you specify multiple checks, any single failure will make the server be considered unhealthy.

Activity Monitoring in Nginx (PLUS)

Nginx PLUS includes a real-time activity monitoring interface that provides load and performance metrics. It uses a RESTful JSON interface, and hence it is very easy to customize. There are plenty of third-party monitoring tools that take advantage of JSON interface and provide you a comprehensive dashboard for performance monitoring.

You can also use the following configuration block to configure Nginx PLUS for status monitoring.

```
server {
    listen 8080;
    root /usr/share/nginx/html;

    # Redirect requests for / to /status.html
    location = / {
        return 301 /status.html;
    }

    location = /status.html { }

    location /status {
        allow x.x.x.x/16; # permit access from local network
        deny all; # deny access from everywhere else

        status;
    }
}
```

Status is a special handler in Nginx PLUS. The configuration here is using port 8080 to view the detailed status of Nginx requests. To give you a better idea of the console, the Nginx team has set up a live demo page that can be accessed at `http://demo.nginx.com/status.html`.

Summary

In this chapter, you have learned about the basic fundamentals of high availability and why it matters. You should also be comfortable with the basic concepts about hardware and software load balancing. Nginx is an awesome product for software load balancing and you have learned about how easily you can set it up in your web farm. The architecture of Nginx allows you to have a very small touch point for front-end servers, and the flexibility ensures that you can customize it precisely based on your requirements. You can scale out your farm easily with Nginx, and use Nginx PLUS to achieve even more robustness in your production farm when the need arises.