Load Balancing Methods

In this lesson, you will get an insight into hardware and software load balancing.

We'll cover the following

- Hardware load balancers
- Software load balancers
- Algorithms/Traffic Routing Approaches Leveraged by Load Balancers
  - Round Robin and Weighted Round Robin
  - Least connections
  - Random
  - Hash

There are primarily three modes of load balancing:

1. *DNS Load Balancing*
2. *Hardware-based Load Balancing*
3. *Software-based Load Balancing*

We discussed *DNS load balancing* in the previous lesson. So, in this one, we will discuss hardware and software load balancing.

Without further ado. Let’s get on with it.

Hardware-based and *software-based* are common ways of balancing traffic loads on large-scale services. Let’s begin with hardware-based load balancing.

Hardware load balancers#

Hardware load balancers are highly performant physical hardware. They sit in front of the application servers and distribute the load based on the number of existing open connections to a server, compute utilization, and several other parameters.

Since these load balancers are physical hardware they need maintenance and regular updates, just like any other server hardware would need. They are expensive to set up in comparison to *software load balancers*, and their upkeep may require a certain skill set.

If the business has an *IT team* and *network specialists* in house, they can take care of these load balancers. Otherwise, the developers are expected to wrap their heads around how to set up these hardware load balancers with some assistance from vendors. This is why developers prefer working with software load balancers.

When using *hardware load balancers*, we may also have to overprovision them to deal with the peak traffic, which is not the case with *software load balancers*.

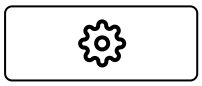
Hardware load balancers are primarily picked because of their top-notch performance.

Now, let's take a look at *software-based load balancing*.

Software load balancers#

Software load balancers can be installed on commodity hardware and *VMs*. They are more cost-effective and offer more flexibility to the developers. *Software load balancers* can be upgraded and provisioned

easily compared to *hardware load balancers*.



You will also find several *Load Balancers as a Service (LBaaS)* services online that enable you to directly plug in load balancers into your application without you having to do any sort of setup.

Software load balancers are pretty advanced compared to *DNS load balancing* as they consider many parameters such as *content hosted by the servers, cookies, HTTP headers, CPU and memory utilization, load on the network*, and so on to route traffic across the servers.

They also continually perform health checks on the servers to keep an updated list of *in-service* machines.

Development teams prefer to work with *software load balancers* as *hardware load balancers* require specialists to manage them.

HAProxy (https://www.haproxy.com/) is one example of a *software load balancer* that is used widely by the big guns in the industry to scale their systems, including *GitHub, Reddit, Instagram, AWS, Tumblr, StackOverflow*, and many more.

Besides the *Round Robin algorithm* which *DNS Load balancers* use, *software load balancers* leverage several other algorithms to efficiently route traffic across the machines. Let’s take a look.

Algorithms/Traffic Routing Approaches Leveraged by Load Balancers#

Round Robin and Weighted Round

Robin#

We know that the *Round Robin algorithm* sends *IP addresses* of machines sequentially to the clients. Parameters on the servers, such as *load*, their *CPU consumption*, and so on, are not taken into account when sending the *IP addresses* to the clients.

We have another approach known as the *Weighted Round Robin* where based on the *server's compute* and *traffic handling capacity* weights are assigned to them. And then, based on server weights, traffic is routed to them using the *Round Robin algorithm*.

With this approach, more traffic is converged to machines that can handle a higher traffic load, thus efficiently using the resources.

This approach is pretty useful when the service is deployed in different data centers having different compute capacities. More traffic can be directed to the larger data centers containing more machines.

Least connections#

When using this algorithm, the traffic is routed to the machine that has the least open connections of all the machines in the cluster. There are two approaches to implement this.

In the first, it is assumed that all the requests will consume an equal amount of server resources, and the traffic is routed to the machine with the least open connections based on this assumption.

Now, in this scenario, there is a possibility that the machine with the least open connections might already be processing requests demanding most of its *CPU* power. Routing more traffic to this machine would not be a good idea.

In the other approach, the *CPU utilization* and the *request processing time* of the chosen machine is also taken into account before routing the traffic to it. Machines with the shortest request processing time, smallest CPU

utilization, and the least open connections are the right candidates to process future client requests.

The least connections approach comes in handy when the server has long opened connections. For instance, consider persistent connections in a gaming application.

Random#

Following this approach, the traffic is randomly routed to the servers. The load balancer may also find similar servers in terms of existing load, request processing time, and so on. Then it randomly routes the traffic to these machines.

Hash#

In this approach, the *source IP* where the request is coming from and the request URL are hashed to route the traffic to the backend servers.

Hashing the *source IP* ensures that the request of a client with a certain *IP* will always be routed to the same server.

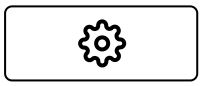
This facilitates a better user experience as the server has already processed the initial client requests and holds the client's data in its local memory. There is no need for it to fetch the client session data from the session memory of the cluster and process the request. This reduces latency.

Hashing the *client IP* also enables the client to re-establish the connection with the same server, that was processing its request in case the connection drops.

Hashing a *URL* ensures that the request with that *URL* always hits a certain cache that already has data on it. This is to ensure that there is no cache miss.

This also averts the need for duplicating data in every cache and is, thus, a

more efficient way to implement caching.



Well, this is pretty much it on the fundamentals of load balancing. In the next chapter, we will cover *monoliths* and *microservices*.

← Back

DNS Load Balancing

Next →

Load Balancing Quiz

Mark as Completed

Report an Issue