Load Balancing Methods

In this lesson, we will have 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 & 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've already discussed *DNS load balancing* in the former lesson. In this one, we will discuss hardware and software load balancing.

So, without further ado. Let's get on with it.

Hardware-based & *Software-based*, both are pretty common ways of balancing traffic load 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 & regular updates, just like any other server hardware would need. They are expensive to setup in comparison to *software load balancers* and their upkeep may require a certain skill set.

If the business has an *IT team* & *network specialists* in house, they can take care of these load balancers else the developers are expected to wrap their head around how to set up these hardware load balancers with some assistance from the vendor. This is the reason 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 that is not the case with *software load balancers*.

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

Now let's have an insight into *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 in comparison to *hardware load balancers*.

You will also find several *LBaaS Load Balancers as a Service* 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 when compared to *DNS load balancing* as they consider many parameters such as *content that the servers host, cookies, HTTP headers, CPU & memory utilization, load on the network* & 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 is one example of a *software load balancer* that is widely used by the big

HAProxy is one example of a *software load balancer* that is widely used by the big guns in the industry to scale their systems such as *GitHub, Reddit, Instagram, AWS,*

Tumblr, StackOverflow & many more.

Besides the *Round Robin algorithm* that the *DNS Load balancers* use, *software load balancers* leverage several other algorithms to efficiently route traffic across the machines. Let's have an insight.

Algorithms/Traffic Routing Approaches Leveraged By Load Balancers

Round Robin & Weighted Round Robin

We know that *Round Robin algorithm* sends *IP address* of machines sequentially to the clients. Parameters such as *load on the servers*, 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* & *traffic handling capacity* weights are assigned to them. And then based on the 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 making efficient use of 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 & the traffic is routed to the machine, having the least open connections, based on this assumption.

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

power. Routing more traffic to this machine wouldn't be such a good idea.

In the other approach, the *CPU utilization* & the *request processing time* of the chosen machine is also taken into account before routing the traffic to it. Machines with less request processing time, CPU utilization & simultaneously having the least open connections are the right candidates to process the future client requests.

The least connections approach comes in handy when the server has long opened connections, for instance, 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 and 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 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 & then 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, let's understand *monoliths* and *microservices*.