



White Paper

Load Balancing 101: Nuts and Bolts

Load balancing technology is the basis on which today's Application Delivery Controllers operate. But the pervasiveness of load balancing technology does not mean it is universally understood, nor is it typically considered from anything other than a basic, network-centric viewpoint. To maximize its benefits, organizations should understand both the basics and nuances of load balancing.

by KJ (Ken) Salchow, Jr.

Sr. Manager, Technical Marketing and Syndication



Contents

Introduction	3
<hr/>	
Basic Load Balancing Terminology	3
Node, Host, Member, and Server	3
Pool, Cluster, and Farm	4
Virtual Server	5
Putting It All Together	5
<hr/>	
Load Balancing Basics	6
The Load Balancing Decision	7
To Load Balance or Not to Load Balance?	9
<hr/>	
Conclusion	10



Introduction

Load balancing got its start in the form of network-based load balancing hardware. It is the essential foundation on which Application Delivery Controllers (ADCs) operate. The second iteration of purpose-built load balancing (following application-based proprietary systems) materialized in the form of network-based appliances. These are the true founding fathers of today's ADCs. Because these devices were application-neutral and resided outside of the application servers themselves, they could load balance using straightforward network techniques. In essence, these devices would present a "virtual server" address to the outside world, and when users attempted to connect, they would forward the connection to the most appropriate real server doing bi-directional network address translation (NAT).

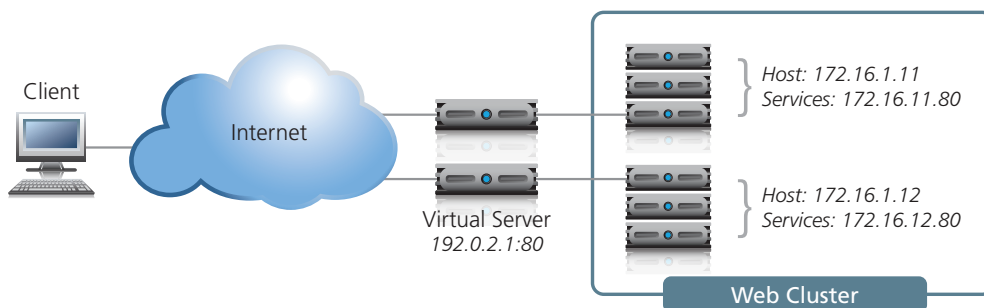


Figure 1: Network-based load balancing appliances.

Basic Load Balancing Terminology

It would certainly help if everyone used the same lexicon; unfortunately, every vendor of load balancing devices (and, in turn, ADCs) seems to use different terminology. With a little explanation, however, the confusion surrounding this issue can easily be alleviated.

Node, Host, Member, and Server

Most load balancers have the concept of a node, host, member, or server; some have all four, but they mean different things. There are two basic concepts that they all try to express. One concept—usually called a node or server—is the idea of the physical server itself that will receive traffic from the load balancer. This is



synonymous with the IP address of the physical server and, in the absence of a load balancer, would be the IP address that the server name (for example, `www.example.com`) would resolve to. For the remainder of this paper, we will refer to this concept as the *host*.

The second concept is a member (sometimes, unfortunately, also called a node by some manufacturers). A member is usually a little more defined than a server/node in that it includes the TCP port of the actual application that will be receiving traffic. For instance, a server named `www.example.com` may resolve to an address of `172.16.1.10`, which represents the server/node, and may have an application (a web server) running on TCP port 80, making the member address `172.16.1.10:80`. Simply put, the member includes the definition of the application port as well as the IP address of the physical server. For the remainder of this paper, we will refer to this as the *service*.

Why all the complication? Because the distinction between a physical server and the application services running on it allows the load balancer to individually interact with the applications rather than the underlying hardware. A host (`172.16.1.10`) may have more than one service available (HTTP, FTP, DNS, and so on). By defining each application uniquely (`172.16.1.10:80`, `172.16.1.10:21`, and `172.16.1.10:53`), the load balancer can apply unique load balancing and health monitoring (discussed later) based on the services instead of the host. However, there are still times when being able to interact with the host (like low-level health monitoring or when taking a server offline for maintenance) is extremely convenient.

Remember, most load balancing-based technology uses some concept to represent the host, or physical server, and another to represent the services available on it—in this case, simply *host* and *services*.

Pool, Cluster, and Farm

Load balancing allows organizations to distribute inbound traffic across multiple back-end destinations. It is therefore a necessity to have the concept of a collection of back-end destinations. *Clusters*, as we will refer to them herein, although also known as pools or farms, are collections of similar services available on any number of hosts. For instance, all services that offer the company web page would be collected into a cluster called “company web page” and all services that offer e-commerce services would be collected into a cluster called “e-commerce.”



The key element here is that all systems have a collective object that refers to “all similar services” and makes it easier to work with them as a single unit. This collective object—a cluster—is almost always made up of services, not hosts.

Virtual Server

Although not always the case, today there is little dissent about the term virtual server, or virtual. It is important to note that like the definition of services, *virtual server* usually includes the application port as well as the IP address. The term “virtual service” would be more in keeping with the IP:Port convention; but because most vendors use virtual server, this paper will continue using virtual server as well.

Putting It All Together

Putting all of these concepts together makes up the basic steps in load balancing. The load balancer presents virtual servers to the outside world. Each virtual server points to a cluster of services that reside on one or more physical hosts.

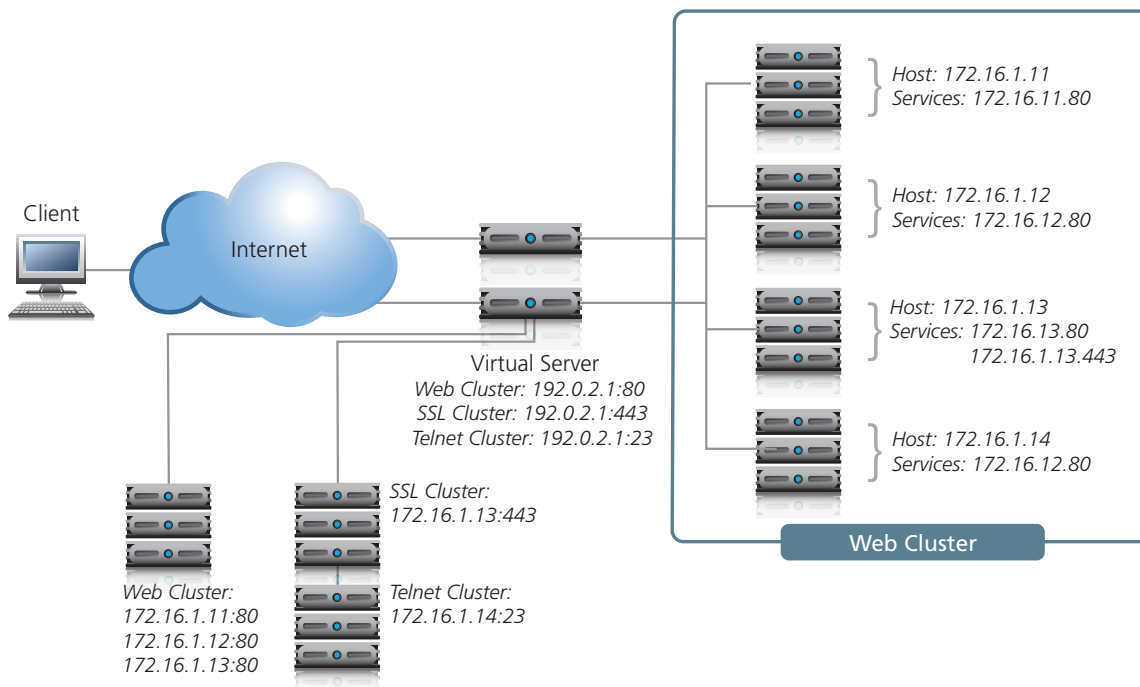


Figure 2: Load balancing comprises four basic concepts—virtual servers, clusters, services, and hosts.



While Figure 2 may not be representative of any real-world deployment, it does provide the elemental structure for continuing a discussion about load balancing basics.

Load Balancing Basics

With this common vocabulary established, let's examine the basic load balancing transaction. As depicted, the load balancer will typically sit in-line between the client and the hosts that provide the services the client wants to use. As with most things in load balancing, this is not a rule, but more of a best practice in a typical deployment. Let's also assume that the load balancer is already configured with a virtual server that points to a cluster consisting of two service points. In this deployment scenario, it is common for the hosts to have a return route that points back to the load balancer so that return traffic will be processed through it on its way back to the client.

The basic load balancing transaction is as follows:

1. The client attempts to connect with the service on the load balancer.
2. The load balancer accepts the connection, and after deciding which host should receive the connection, changes the destination IP (and possibly port) to match the service of the selected host (note that the source IP of the client is not touched).
3. The host accepts the connection and responds back to the original source, the client, via its default route, the load balancer.
4. The load balancer intercepts the return packet from the host and now changes the source IP (and possible port) to match the virtual server IP and port, and forwards the packet back to the client.
5. The client receives the return packet, believing that it came from the virtual server, and continues the process.

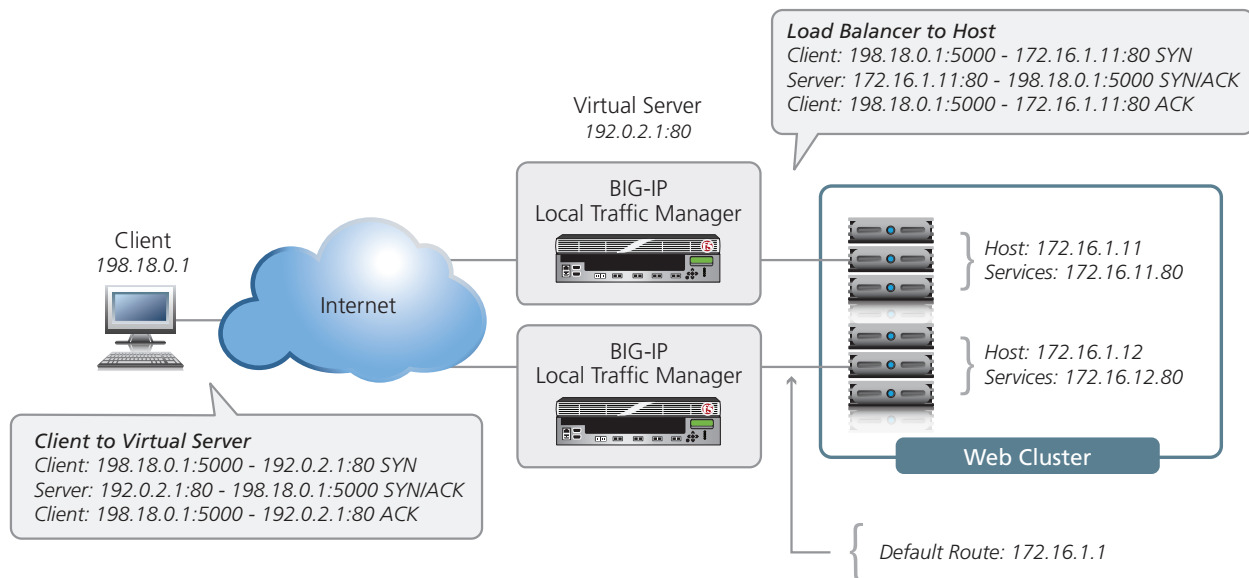


Figure 3: A basic load balancing transaction.

This very simple example is relatively straightforward, but there are a couple of key elements to take note of. First, as far as the client knows, it sends packets to the virtual server and the virtual server responds—simple. Second, the NAT takes place. This is where the load balancer replaces the destination IP sent by the client (of the virtual server) with the destination IP of the host to which it has chosen to load balance the request. Step three is the second half of this process (the part that makes the NAT “bi-directional”). The source IP of the return packet from the host will be the IP of the host; if this address were not changed and the packet was simply forwarded to the client, the client would be receiving a packet from someone it didn’t request one from, and would simply drop it. Instead, the load balancer, remembering the connection, rewrites the packet so that the source IP is that of the virtual server, thus solving this problem.

The Load Balancing Decision

Usually at this point, two questions arise: how does the load balancer decide which host to send the connection to? And what happens if the selected host isn’t working?

Let’s discuss the second question first. What happens if the selected host isn’t working? The simple answer is that it doesn’t respond to the client request and the connection attempt eventually times out and fails. This is obviously not a preferred circumstance, as it doesn’t ensure high availability. That’s why most load balancing technology includes some level of health monitoring that determines whether a host is actually available *before* attempting to send connections to it.



There are multiple levels of health monitoring, each with increasing granularity and focus. A basic monitor would simply PING the host itself. If the host does not respond to PING, it is a good assumption that any services defined on the host are probably down and should be removed from the cluster of available services. Unfortunately, even if the host responds to PING, it doesn't necessarily mean the service itself is working. Therefore most devices can do "service PINGs" of some kind, ranging from simple TCP connections all the way to interacting with the application via a scripted or intelligent interaction. These higher-level health monitors not only provide greater confidence in the availability of the actual services (as opposed to the host), but they also allow the load balancer to differentiate between multiple services on a single host. The load balancer understands that while one service might be unavailable, other services on the same host might be working just fine and should still be considered as valid destinations for user traffic.

This brings us back to the first question: How does the load balancer decide which host to send a connection request to? Each virtual server has a specific dedicated cluster of services (listing the hosts that offer that service) which makes up the list of possibilities. Additionally, the health monitoring modifies that list to make a list of "currently available" hosts that provide the indicated service. It is this modified list from which the load balancer chooses the host that will receive a new connection. Deciding the exact host depends on the load balancing algorithm associated with that particular cluster. The most common is simple round-robin where the load balancer simply goes down the list starting at the top and allocates each new connection to the next host; when it reaches the bottom of the list, it simply starts again at the top. While this is simple and very predictable, it assumes that all connections will have a similar load and duration on the back-end host, which is not always true. More advanced algorithms use things like current-connection counts, host utilization, and even real-world response times for existing traffic to the host in order to pick the most appropriate host from the available cluster services.

Sufficiently advanced load balancing systems will also be able to synthesize health monitoring information with load balancing algorithms to include an understanding of service dependency. This is the case when a single host has multiple services, all of which are necessary to complete the user's request. A common example would be in e-commerce situations where a single host will provide both standard HTTP services (port 80) as well as HTTPS (SSL/TLS at port 443). In many of these circumstances, you don't want a user going to a host that has one service operational, but not the other. In other words, if the HTTPS services should fail on a host, you also want that host's HTTP service to be taken out of the cluster list of available services. This functionality is increasingly important as HTTP-like services become more differentiated with XML and scripting.



To Load Balance or Not to Load Balance?

Load balancing in regards to picking an available service when a client initiates a transaction request is only half of the solution. Once the connection is established, the load balancer must keep track of whether the following traffic from that user should be load balanced. There are generally two specific issues with handling follow-on traffic once it has been load balanced: connection maintenance and persistence.

Connection maintenance

If the user is trying to utilize a long-lived TCP connection (telnet, FTP, and more) that doesn't immediately close, the load balancer must ensure that multiple data packets carried across that connection do not get load balanced to other available service hosts. This is connection maintenance and requires two key capabilities: 1) the ability to keep track of open connections and the host service they belong to; and 2) the ability to continue to monitor that connection so the connection table can be updated when the connection closes. This is rather standard fare for most load balancers.

Persistence

Increasingly more common, however, is when the client uses multiple short-lived TCP connections (for example, HTTP) to accomplish a single task. In some cases, like standard web browsing, it doesn't matter and each new request can go to any of the back-end service hosts; however, there are many more instances (XML, e-commerce "shopping cart," HTTPS, and so on) where it is extremely important that multiple connections from the same user go to the same back-end service host and *not* be load balanced. This concept is called persistence, or server affinity. There are multiple ways to address this depending on the protocol and the desired results. For example, in modern HTTP transactions, the server can specify a "keep-alive" connection, which turns those multiple short-lived connections into a single long-lived connection that can be handled just like the other long-lived connections. However, this provides little relief. Even worse, as the use of web services increases, keeping all of these connections open longer than necessary would strain the resources of the entire system. In these cases, most load balancers provide other mechanisms for creating artificial server affinity.

One of the most basic forms of persistence is source-address affinity. This involves simply recording the source IP address of incoming requests and the service host

they were load balanced to, and making all future transaction go to the same host. This is also an easy way to deal with application dependency as it can be applied across *all* virtual servers and *all* services. In practice however, the wide-spread use of proxy servers on the Internet and internally in enterprise networks renders this form of persistence almost useless; in theory it works, but proxy-servers inherently hide many users behind a single IP address resulting in none of those users being load balanced after the first user's request—essentially nullifying the load balancing capability. Today, the intelligence of load balancer-based devices allows organizations to actually open up the data packets and create persistence tables for virtually anything within it. This enables them to use much more unique and identifiable information, such as user name, to maintain persistence. However, organizations one must take care to ensure that this identifiable client information will be present in every request made, as any packets without it will not be persisted and will be load balanced again, most likely breaking the application.

Conclusion

It is important to understand that basic load balancing technology, while still in use, is now only considered a feature of Application Delivery Controllers. ADCs evolved from the first load balancers and completed the service virtualization process, they can not only improve availability, but also affect the security and performance of the application services being requested.

Today, most organizations realize that simply being able to reach an application doesn't make it usable; and unusable applications mean wasted time and money for the enterprise deploying them. ADCs enable organizations to consolidate network-based services like SSL/TLS offload, caching, compression, rate-shaping, intrusion detection, application firewalls, and even remote access into a single point that can be shared and reused across *all* application services and *all* hosts to create a virtualized Application Delivery Network. Basic load balancing is the foundation without which none of the enhanced functionality of ADCs would be possible.

