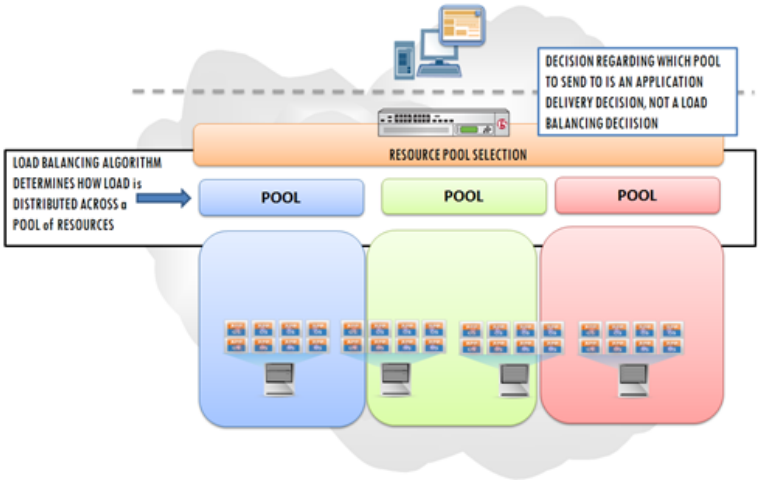
Choosing a Load Balancing Algorithm Requires DevOps Fu

Lori MacVittie, 2010-07-09

Knowing the algorithms is only half the battle, you've got to understand a whole lot more to design a scalable architecture.

Citrix's Craig Ellrod has a series of blog posts on the basic (industry standard) load balancing algorithms. These are great little posts for understanding the basics of load balancing algorithms like round robin, least connections, and least (fastest) response time. Craig's posts are accurate in their description of the theoretical (designed) behavior of the algorithms. The thing that's missing from these posts (and maybe Craig will get to this eventually) is context. Not the context I usually talk about, but the context of the application that is being load balanced and the way in which modern load balancing solutions behave, which is not as a simple Load balancer.



Different applications have different usage patterns. Some are connection heavy, some are compute intense, some return lots of little responses while others process larger incoming requests. Choosing a load balancing algorithm without understand the behavior of the application and its users will almost always result in an inefficient scaling strategy that leaves you constantly trying to figure out why load is unevenly distributed or SLAs are not being met or why some users are having a less than stellar user experience.

One of the most misunderstood aspects of load balancing is that a load balancing algorithm is designed to choose from a pool of resources and that an application is or can be made up of multiple pools of resources. These pools can be distributed (cloud balancing) or localized, and they may all be active or some may be designated as solely existing for failover purposes. Ultimately this means that the algorithm does not actually choose the pool from which a resource will be chosen – it only chooses a specific resource. The relationship between the **choice of a pool** and the **choice of a single resource in a pool** is subtle but important when making architectural decisions – especially those that impact scalability. A pool of resources is (or should be) a set of servers serving similar resources. For example, the separation of image servers from application logic servers will better enable **scalability domains** in which each resource can be scaled individually, without negatively impacting the entire application.

To make the decision more complex, there are a variety of factors that impact the way in which a load balancing algorithm actually behaves in contrast to how it is designed to act. The most basic of these factors is the network layer at which the load balancing decision is being made.

THE IMPACT of PROTOCOL

There are two layers at which applications are commonly load balanced: TCP (transport) and HTTP (application). The layer at which a load balancing is performed has a profound effect on the architecture and capabilities.

Layer 4 (Connection-oriented) Load Balancing

When load balancing at layer 4 you are really load balancing at the TCP or connection layer. This means that a connection (user) is bound to the server chosen on the initial request. Basically a request arrives at the load balancer and, based on the algorithm chosen, is directed to a server. Subsequent requests over the same connection will be directed to that same server.

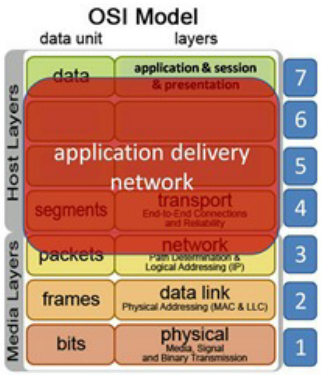
Unlike Layer 7-related load balancing, in a layer 4 configuration the algorithm is usually the routing mechanism for the application.

Layer 7 (Connection-oriented) Load Balancing

When load balancing at Layer 7 in a connection-oriented configuration, each connection is treated in a manner similar to Layer 4 Load Balancing with the exception of how the initial server is chosen. The decision may be based on an HTTP header value or cookie instead of simply relying on the load balancing algorithm. In this scenario the decision being made is which pool of resources to send the connection to. Subsequently the load balancing algorithm chosen will determine which resource within that pool will be assigned. Subsequent requests over that same connection will be directed to the server chosen.

Layer 7 connection-oriented load balancing is most useful in a virtual hosting scenario in which many hosts (or applications) resolve to the same IP address.

Layer 7 (Message-oriented) Load Balancing



Layer 7 load balancing in a message-oriented configuration is the most flexible in terms of the ability to distribute load across pools of resources based on a wide variety of variables. This can be as simple as the URI or as complex as the value of a specific XML element within the application message. Layer 7 message-oriented load balancing is more complex than its Layer 7 connection-oriented cousin because a message-oriented configuration also allows individual requests – over the same connection – to be load balanced to different (virtual | physical) servers. This flexibility allows the scaling of message-oriented protocols such as SIP that leverage a single, long-lived connection to perform tasks requiring different applications. This makes message-oriented load balancing a better fit for applications that also provide APIs as individual API requests can be directed to different pools based on the functionality they are performing, such as separating out requests that update a data source from those that simply read from a data source.

Layer 7 message-oriented load balancing is also known as “request switching” because it is capable of making routing decisions on every request even if the requests are sent over the same connection. Message-oriented load balancing requires a full proxy architecture as it must be the end-point to the client in order to intercept and interpret requests and then route them appropriately.

OTHER FACTORS to CONSIDER

If that were not confusing enough, there are several other factors to consider that will impact the way in which load is actually distributed (as opposed to the theoretical behavior based on the algorithm chosen).

Application Protocol

HTTP 1.0 acts differently than HTTP 1.1. Primarily the difference is that HTTP 1.0 implies a one-to-one relationship between a request and a connection. Unless the HTTP Keep-Alive header is included with a HTTP 1.0 request, each request will incur the processing costs to open and close the connection. This has an impact on the choice of algorithm because there will be many more connections open and in progress at any given time. You might think there’s never a good reason to force HTTP 1.0 if the default is more efficient, but consider the case in which a request is for an image – you want to GET it and then you’re finished. Using HTTP 1.0 and immediately closing the connection is actually better for the efficiency and thus capacity of the web server because it does not maintain an open connection waiting for a second or third request that will not be forthcoming. An open, idle connection that will eventually simply time-out wastes resources that could be used by some other user. Connection management is a large portion of resource consumption on a web or application server, thus anything that increases that number and rate decreases the overall capacity of each individual server, making it less efficient.

HTTP 1.1 is standard (though unfortunately not ubiquitous) and re-uses client-initiated connections, making it more resource efficient but introducing questions regarding architecture and algorithmic choices. Obviously if a connection is reused to request both an image resource and a data resource but you want to leverage fault tolerant and more efficient scale design using scalability domains this will impact your choice of load balancing algorithms and the way in which requests are routed to designated pools.

Configuration Settings

A little referenced configuration setting in all web and application servers (and load balancers) is the maximum number of requests per connection. This impacts the distribution of requests because once the configured maximum number of requests has been sent over the same connection it will be closed and a new connection opened. This is particularly impactful on AJAX and other long-lived connection applications. The choice of algorithm can have a profound affect on availability when coupled with this setting. For example, an application uses an AJAX-update on a regular interval. Furthermore, the application requests made by that updating request require access to session state, which implies some sort of persistence is required. The first request determines the server, and a cookie (as one method) subsequently ensures that all further requests for that update are sent to the same server. Upon reaching the maximum number of requests, the load balancer must re-establish that connection – and because of the reliance on session state it must send the request back to the original server. In the meantime, however, another user has requested a resource and been assigned to that same server, and that user has caused the server to reach its maximum number of connections (also configurable). The original user is unable to be reconnected to his session and regardless of whether the request is load balanced to a new resource or not, “availability” is effectively lost as the current state of the user’s “space” is now gone.

You’ll note that a combination of factors are at work here: (1) the load balancing algorithm chosen, (2) configuration options and (3) application usage patterns and behavior.

IT JUST ISN’T ENOUGH to KNOW HOW the ALGORITHMS BEHAVE

This is the area in which devops is meant to shine – in the bridging of that gap across applications and the network, in understanding how the two interact, integrate, and work together to achieve high-scalability and well-performing applications. Devops must understand the application but they must also understand how load balancing choices and configurations and options impact the delivery and scalability of that application, and vice-versa. The behavior of an application and its users can impact the way in which a load balancing algorithm performs **in reality rather than theory**, and the results are often very surprising. Round-robin load balancing, is designed to equally distribute **requests** across a pool of resources, not **workload**. Thus in the course of a period of time one application instance in a pool of resources may become overloaded and either become unavailable or exhibit errors that appear only under heavy load while other instances in the same pool may be operating normally and under nominal load conditions.

You certainly need to have a basic understanding of load balancing algorithms, especially moving forward toward elastic applications and cloud computing . But once you understand the basics you really need to start examining the bigger picture and determining the best set of options and configurations for each application. This is one of the reasons testing is so important when designing a scalable architecture – to uncover the strange behavior that can often only be discovered under heavy load, to examine the interaction of all the variables and how they impact the actual behavior of the load balancer at run time.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com

©2016 F5 Networks, Inc. All rights reserved. F5, F5 Networks, and the F5 logo are trademarks of F5 Networks, Inc. in the U.S. and in certain other countries. Other F5 trademarks are identified at f5.com. Any other products, services, or company names referenced herein may be trademarks of their respective owners with no endorsement or affiliation, express or implied, claimed by F5. CS04-00015 0113