MAKING APPLICATIONS SCALABLE WITH LOAD BALANCING

September 2006

Willy Tarreau

w@1wt.eu

Revision 1.0

http://1wt.eu/articles/2006_lb/

TABLE OF CONTENTS

Why this article

As the author of the HAProxy[1] Load Balancer, I'm often questioned about Load Balancing architectures or choices between load balancers. Since there is no obvious response, it's worth enumerating the pros and cons of several solutions, as well as a few traps to avoid. I hope to complete this article soon with a deeper HTTP analysis and with architecture examples. You can contact me for further information.

Introduction

Originally, the Web was mostly static contents, quickly delivered to a few users which spent most of their time reading between rare clicks. Now, we see real applications which hold users for tens of minutes or hours, with little content to read between clicks and a lot of work performed on the servers.

The users often visit the same sites, which they know perfectly and don't spend much time reading. They expect immediate delivery while they unconsciously inflict huge loads on the servers at every single click.

This new dynamics has developed a new need for high performance and permanent availability.

What is load balancing ?

Since the power of any server is finite, a web application must be able to run on multiple servers to accept an ever increasing number of users. This is called scaling. Scalability is not really a problem for intranet applications since the number of users has little chances to increase. However, on internet portals, the load continuously increases with the availability of broadband Internet accesses. The site's maintainer has to find ways to spread the load on several servers, either via internal mechanisms included in the application server, via external components, or via architectural redesign. Load balancing is the ability to make several servers participate in the same service and do the same work. As the number of servers grows, the risk of a failure anywhere increases and must be addressed. The ability to maintain unaffected service during any predefined number of simultaneous failures is called high availability. It is often mandatory with load balancing, which is the reason why people often confuse those two concepts. However, certain load balancing techniques do not provide high availability and are dangerous.

[1] http://haproxy.1wt.eu/

Load balancing techniques

DNS

The easiest way to perform load balancing is to dedicate servers to predefined groups of users. This is easy on Intranet servers, but not really on internet servers. On common approach relies on DNS round robin. If a DNS server has several entries for a given hostname, it will return all of them in a rotating order.

This way, various users will see different addresses for the same name and will be able to reach different servers. This is very commonly used for multi-site load balancing, but it requires that the application is not impacted to the lack of server context. For this reason, this is generally used to by search engines, POP servers, or to deliver static contents.

This method does not provide any means of availability. It requires additional measures to permanently check the servers status and switch a failed server's IP to another server.

For this reason, it is generally used as a complementary solution, not as a primary one.

```
$ host -t a google.com
google.com. has address 64.233.167.99
google.com. has address 64.233.187.99
google.com. has address 72.14.207.99

$ host -t a google.com
google.com. has address 72.14.207.99
google.com. has address 64.233.167.99
google.com. has address 64.233.187.99

$ host -t a mail.aol.com
mail.aol.com. has address 205.188.162.57
mail.aol.com. has address 205.188.212.249
mail.aol.com. has address 64.12.136.57
mail.aol.com. has address 64.12.136.89
mail.aol.com. has address 64.12.168.119
mail.aol.com. has address 64.12.193.249
mail.aol.com. has address 152.163.179.57
mail.aol.com. has address 152.163.211.185
mail.aol.com. has address 152.163.211.250
mail.aol.com. has address 205.188.149.57
mail.aol.com. has address 205.188.160.249
```

Fig.1 : DNS round robin : multiple addresses assigned to a same name, returned in a rotating order

REDUCING THE NUMBER OF USERS PER SERVER

A more common approach is to split the population of users across multiple servers. This involves a load balancer between the users and the servers. It can consist in a hardware equipment, or in a software installed either on a dedicated front server, or on the application servers themselves. Note that by deploying new components, the risk of failure increases, so a common practise is to have a second load balancer acting as a backup for the primary one.

Generally, a hardware load balancer will work at the network packets level and will act on routing, using one of the following methods :

- **Direct routing** : the load balancer routes the same service address through different local, physical servers, which must be on the same network segment, and must all share the same service address. It has the huge advantage of not modifying anything at the IP level, so that the servers can reply directly to the user without passing again through the load balancer. This is called "direct server return". As the processing power needed for this method is minimal, this is often the one used on front servers on very high traffic sites. On the other hand, it requires some solid knowledge of the TCP/IP model to correctly configure all the equipments, including the servers.

- **Tunnelling** : it works exactly like direct routing, except that by establishing tunnels between the load balancer and the servers, theses ones can be located on remote networks. Direct server return is still possible.

- **IP address translation (NAT)** : the user connects to a virtual destination address, which the load balancer translates to one of the servers' addresses. This is easier to deploy at first glance, because there is less trouble in the server configuration. But this requires stricter programming rules. One common error is application servers indicating their internal addresses in some responses. Also, this requires more work on the load balancer, which has to translate addresses back and forth, to maintain a session table, and it requires that the return traffic goes through the load balancer as well. Sometimes, too short session timeouts on the load balancer induce side effects known as ACK storms. In this case, the only solution is to increase the timeouts, at the risk of saturating the load balancer's session table.

On the opposite side, we find software load balancers. They most often act like reverse proxies, pretending to be the server and forwarding them the traffic. This implies that the servers themselves cannot be reached directly from the users, and that some protocols might never get load-balanced.

They need more processing power than those acting at the network level, but since they splice the communication between the users and the servers, they provide a first level of security by only forwarding what they understand. This is the reason why we often find URL filtering capabilities on those products.

TESTING THE SERVERS

To select a server, a load balancer must know which ones are available. For this, it will periodically send them pings, connection attempts, requests, or anything the administrator considers a valid measure to qualify their state. These tests are called "health checks".

A crashed server might respond to ping but not to TCP connections, and a hung server might respond to TCP connections but not to HTTP requests. When a multi-layer Web application server is involved, some HTTP requests will provide instant responses while others will fail.

So there is a real interest in choosing the most representative tests permitted by the application and the load balancer. Some tests might retrieve data from a database to ensure the whole chain is valid. The drawback is that those tests will consume a certain amount of server resources (CPU, threads, etc...).

They will have to be spaced enough in time to avoid loading the servers too much, but still be close enough to quickly detect a dead server. Health checking is one of the most complicated aspect of load balancing, and it's very common that after a few tests, the application developers finally implement a special request dedicated to the load balancer, which performs a number of internal representative tests.

For this matter, software load balancers are by far the most flexible because they often provide scripting capabilities, and if one check requires code modifications, the software's editor can generally provide them within a short timeframe.

SELECTING THE BEST SERVER

There are many ways for the load balancer to distribute the load. A common misconception is to expect it to send a request to "the first server to respond".

This practise is wrong because if a server has any reason to reply even slightly faster, it will unbalance the farm by getting most of the requests.

A second idea is often to send a request to "the least loaded server". Although this is useful in environments involving very long sessions, it is not well suited for web servers where the load can vary by two digits factors within seconds.

For homogeneous server farms, the "round robin" method is most often the best one.

It uses every server in turn. If servers are of unequal capacity, then a "weighted round robin" algorithm will assign the traffic to the servers according to their configured relative capacities.

These algorithms present a drawback : they are non-deterministic. This means that two consecutive requests from the same user will have a high chance of reaching two different servers. If user contexts are stored on the application servers, they will be lost between two consecutive requests. And when complex session setup happens (eg: SSL key negociation), it will have to be performed again and again at each connection.

To work around this problem, a very simple algorithm is often involved : address hashing. To summarize it, the user's IP address is divided by the number of servers and the result determines the right server for this particular user. This works well as long as the number of servers does not change and as the user's IP address is stable, which is not always the case.

In the event of any server failure, all users are rebalanced and lose their sessions. And the few percent of the users browsing through proxy farms will not be able to use the application either (usually 5-10%). This method is not always applicable though, because for a good distribution, a high number of source IP addresses is required.

This is true on the Internet, but not always within small companies or even ISP's infrastructures. However, this is very efficient to avoid recomputing SSL session keys too often.

Persistence

The solution to the limits above is then to use persistence. Persistence is a way to ensure that a given user will keep going to the same server for all his requests, where its context is known. A common cheap solution is for the application to send back a redirection to the local server address using an HTTP 302 response. A major drawback is that when the server fails, the user has no easy way to escape, and keeps trying to reach the dead server. Just like with DNS, this method is useful only on big sites when the address passed to the user is guaranteed to be reachable.

A second solution is for the load balancer to learn user-server associations, the cheapest way being to learn which user's IP went to which server last time. This generally solves the problem of the server farm size varying due to failures, but does not solve the problem of users with a variable IP address.

So what's left ? From the start, we're stating that we're trying to guarantee that a user will find his context on subsequent requests to the same server. How is the context identified by the server ? By a cookie. Cookies were invented exactly for that purpose : send an information to the user which he will pass back on future visits so that we know what to do with him. Of course, this requires that the user supports cookies, but this is generally the case on applications which need persistence.

If the load balancer could identify the server based on the cookies presented by the user, it would solve mosts of the problems. There are mainly two approaches regarding the cookies : the load balancer can learn the session cookies assigned by the servers, or they can insert cookie for server identification.

COOKIE LEARNING

Cookie learning is the least intrusive solution. The load balancer is configured to learn the application cookie (eg. "JSESSIONID"). When it receives the user's request, it checks if it contains this cookie and a known value. If this is not the case, it will direct the request to any server, according to the load balancing algorithm. It will then extract the cookie value from the response and add it along with the server's identifier to a local table. When the user comes back again, the load balancer sees the cookie, lookups the table and finds the server to which it forwards the request. This method, although easily deployed, has two minor drawbacks, implied by the learning aspect of this method :

- The load balancer has finite memory, so it might saturate, and the only solution to avoid this is to limit the cookie lifetime in the table. This implies that if a user comes back after the cookie expiration, he will be directed to a wrong server.
- If the load balancer dies and its backup takes over, it will not know any association and will again direct users to wrong servers. Of course, it will not be possible to use the load balancers in active-active combinations either. A solution to this is real-time session synchronization, which is difficult to guarantee.
- A workaround for these drawbacks is often to choose a deterministic load balancing algorithm such as the user's address hashing when possible. This way, should the cookie be lost on the load balancer, at least the users who have a fixed IP address will be kept on their server.

COOKIE INSERTION

If the users support cookies, why not add another one with fixed text ? This method, called "cookie insertion", consists in inserting the server's identifier in a cookie added to the response. This way, the load balancer has nothing to learn, it will simply reuse the value presented by the user to select the right server.

This solves both problems of cookie learning : memory limitations and load balancer takeover. However, it requires more efforts from the load balancer, which must open the stream an insert data. This is not easily done, especially on ASIC-based load balancers which have very limited knowledge of TCP and HTTP. It also requires that the user agent accepts multiple cookies.

This is not always the case on small mobile terminals for instance, which are sometimes limited to only one cookie. Then, several variations on this concept may be applied, such as cookie modification, consisting in prefixing an already existing cookie with the server's identifier.

Note: Special care should also be taken on the load balancer regarding the response's cacheability : front caches might store the load balancer cookie and pass it to all users requesting the index page for instance, which would lead to all users being directed to the same server.

PERSISTENCE LIMITATIONS - SSL

We've reviewed methods involving snooping on the user-server exchanges, and sometimes even modifications. But with the ever growing number of applications relying on SSL for their security, the load balancer will not always be able to access HTTP contents. For this reason, we see more and more software load balancers providing support for SSL.

The principle is rather simple : the load balancer acts as a reverse proxy and serves as the SSL server end. It holds the servers' certificates, deciphers the request, accesses the contents and directs the request to the servers (either clear text or slightly re-ciphered).

This gives a new performance boost to the application servers which do not have to manage SSL anymore.

However, the load balancer becomes the bottleneck : a single software-based load balancer will not be faster to process SSL than an eight-servers farm. Because of this architectural error, the load balancer will saturate before the application servers, and the only remedy will be to put another level of load balancers in front of it, and adding more load balancers to handle the SSL load. Obviously, this is wrong. Maintaining a farm of load balancers is very difficult, and the health-checks sent by all those load balancers will induce a noticeable load to the servers. **The right solution is then to have a dedicated SSL farm.**

DEDICATED SSL-CACHE FARM

The most scalable solution when applications require SSL is to dedicate a farm of reverse-proxies only for this matter. There are only advantages to this solution :

1. SSL reverse proxies are very cheap yet powerful. Anyone can build powerful Apache-based SSL reverse proxies for less than $1000. This has to be compared to the cost of a same performance SSL-enabled load balancer (more than $15000), and to the cost of multi-processor servers used for the application, which will not have to be wasted to do SSL anymore.
2. SSL reverse proxies almost always provide caching, and sometimes even compression. Most e-business applications present a cacheability rate between 80 and 90%, which means that these reverse proxy caches will offload the servers by at least 80% of the requests.
3. adding this to the fact that SSL proxies can be shared between multiple applications, the overall gain reduces the need for big and numerous application servers, which impacts maintenance and licensing costs too.
4. the SSL farm can grow as the traffic increases, without requiring load balancer upgrades nor replacements. Load balancers' capacities are often well over most site's requirements, provided that the architecture is properly designed.
5. the first level of proxy provides a very good level of security by filtering invalid requests and often providing the ability to add URL filters for known attacks.
6. it is very easy to replace the product involved in the SSL farm when very specific needs are identified (eg: strong authentication). On the opposite, replacing the SSL-enabled load balancer for this might have terrible impacts on the application's behaviour because of different health-checks methods, load balancing algorithms and means of persistence.
7. the load balancer choice will be based only on load balancing capabilities and not on its SSL performance. It will always be more adapted and far cheaper than an all-in-one solution.

An SSL-proxy-cache farm consists in a set of identical servers, almost always running any flavour of Apache + ModSSL (even in disguised commercial solutions), which are load-balanced either by an external network load balancer, or by an internal software load balancer such as LVS under Linux.

These servers require nearly no maintenance, as their only tasks is to turn the HTTPS traffic into HTTP, check the cache, and forward the non-cacheable requests to the application server farm, represented by the load balancer and the application servers. Interesting to note, the same load balancer may be shared between the reverse proxies and the application servers.

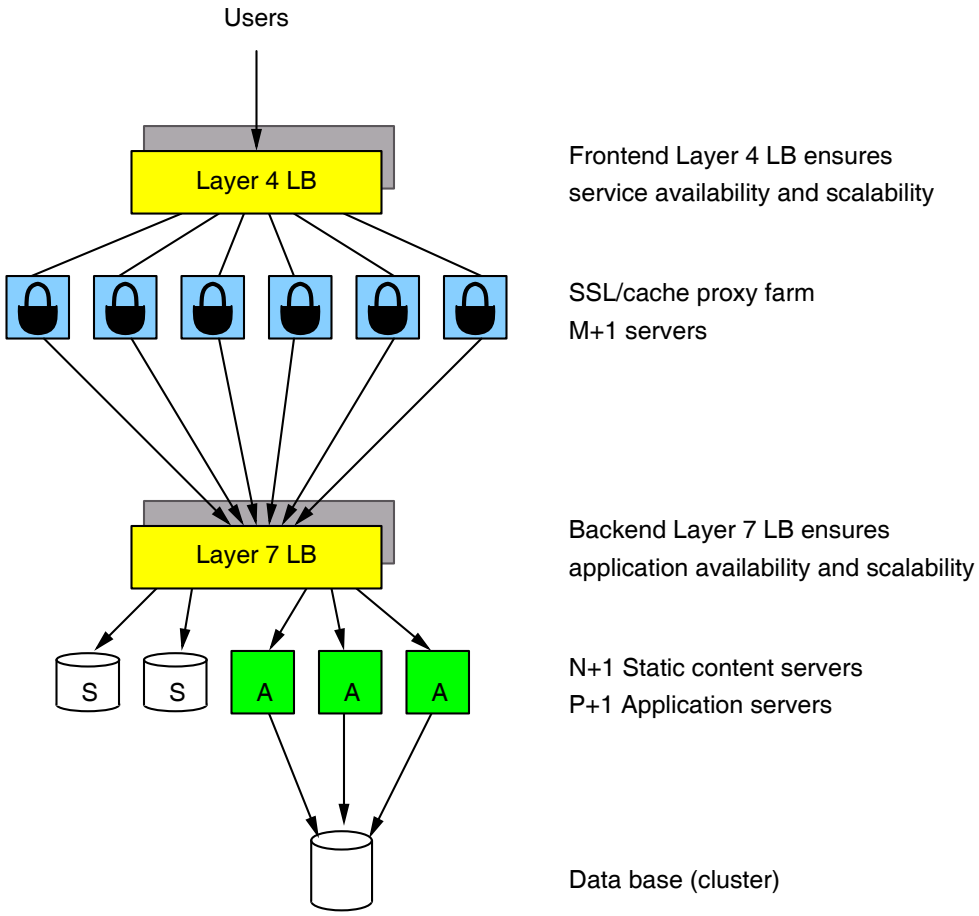

Fig.2 : SSL farm before the layer 7 LB

How to choose between hardware and software load balancer

Although all solutions tend to look like they can do everything, this is not the case. Basically, there are three aspects to consider to choose a load balancer :

- performance
- reliability
- features

Performance is critical, because when the load balancer becomes the bottleneck, nothing can be done. Reliability is very important because the load balancer takes all the traffic, so its reliability must be orders of magnitude above the servers', in terms of availability as well as in terms of processing quality. Features are determinant to choose a solution, but not critical. It depends on what tradeoffs are acceptable regarding changes to the application server.

SESSION COUNT

One of the most common questions when comparing hardware-based to proxy-based load balancers is why is there such a gap between their session count. While proxies generally announce thousands of concurrent sessions, hardware speaks in millions.

Obviously, few people have 20000 Apache servers to drain 4 millions of concurrent sessions ! In fact, it depends whether the load balancer has to manage TCP or not. TCP requires that once a session terminates, it stays in the table in TIME_WAIT state long enough to catch late retransmits, which can be seen several minutes after the session has been closed. After that delay, the session is automatically removed.

In practise, delays between 15 and 60 seconds are encountered depending on implementations. This causes a real problem on systems supporting high session rates, because all terminated sessions have to be kept in the table. A 60 seconds delay would consume 1.5 million entries at 25000 sessions per second.

Fortunately, sessions in this state do not carry any data and are very cheap. Since they are transparently handled by the OS, a proxy never sees them and only announces how many active sessions it supports. But when the load balancer has to manage TCP, it must support very large session tables to store those sessions, and announces the global limit.

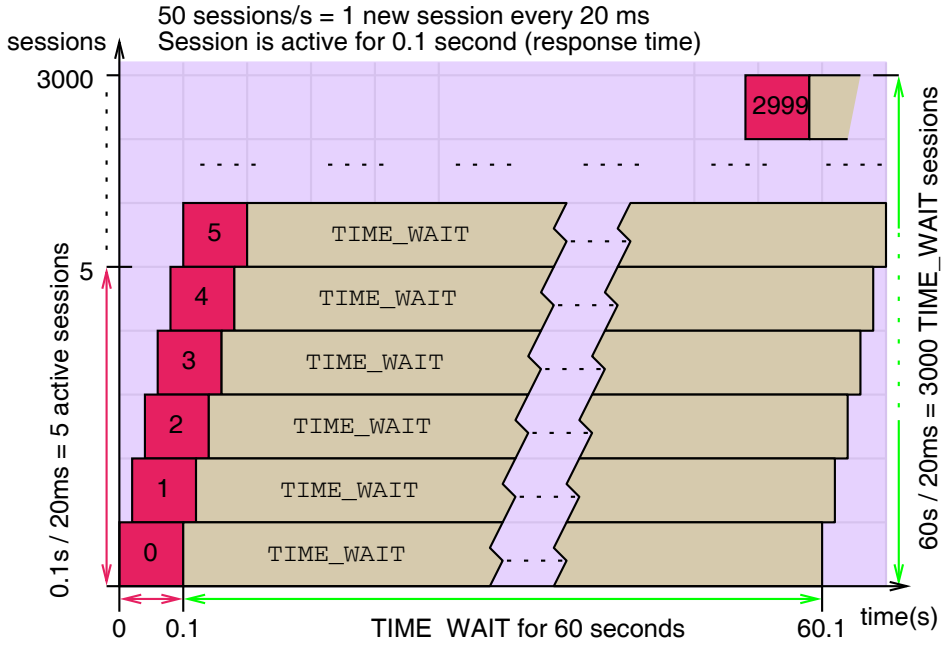

Fig.3 : How session tables fill up with traffic

LAYER 3/4 LOAD BALANCING

Obviously, the best performance can be reached by adding the lowest overhead, which means processing the packets at the network level. For this reason, network-based load balancers can reach very high bandwidths when it comes to layer-3 load balancing (eg: direct routing with a hashing algorithm). As layer-3 is almost always not enough, a network load balancer has to consider layer-4. Most of the cost lies in the session setup and tear down, which can still be performed at very high rates because they are a cheap operations.

More complex operations such as address translation require more processing power to recompute checksums and look up tables. Everywhere we see hardware acceleration providing performance boosts, and load balancing is no exception.

A dedicated ASIC can route network packets at wire speed, set up and tear down sessions at very high rates without the overhead imposed by a general purpose operating system.

Conversely, using a software proxy for layer-4 processing leads to very limited performance, because for simple tasks that could be performed at the packet level, the system has to decode packets, decapsulate layers to find the data, allocate buffer memory to queue them, pass them to the process, establish connections to remote servers, manage system ressources, etc... For this reason, a proxy-based layer-4 load balancer will be 5 to 10 times slower than its network-based counterpart on the same hardware.

Memory is also a limited resource : the network-based load balancer needs memory to store in-flight packets, and sessions (which basically are addresses, protocols, ports and a few parameters, which globally require a few hundreds of bytes per session). The proxy-based load balancer needs buffers to communicate with the system. Per-session system buffers are measured in kilobytes, which implies practical limits around a few tens of thousands of active sessions.

Network-based load balancers also permit a lot of useful fantasies such as virtual MAC addresses, mapping to entire hosts or networks, etc... which are not possible in standard proxies relying on general purpose operating systems.

One note on reliability though. Network-based load balancers can do nasty things such as forwarding invalid packets just as they received them, or confuse sessions (especially when doing NAT). We talked earlier about the ACK storms which sometimes happen when doing NAT with too low session timeouts. They are caused by too early re-use of recently terminated sessions, and cause network saturation between one given user and a server. This cannot happen on proxies because they rely on standards-compliant TCP stacks offered by the OS, and they completely cut the session between the client and the server.

Overall, a well tuned network-based load balancer generally provides the best layer-3/4 solution in terms of performance and features.

LAYER 7 LOAD BALANCING

Layer-7 load balancing involves cookie-based persistence, URL switching and such useful features. While a proxy relying on a general purpose TCP/IP stack obviously has no problem doing the job, the network-based load balancers have to resort to a lot of tricks which don't always play well with standards and often cause various trouble. The most difficult problems to solve are :

1. **multi-packet headers** : the load balancer looks for strings within packets. When the awaited data is not on the first packet, the first ones have to be memorized and consume memory. When the string starts at the end of one packet and continues on the next one, it is very difficult to extract. This is the standard mode of operation for TCP streams though. For reference, a 8 kB request as supported by Apache will commonly span over 6 packets.

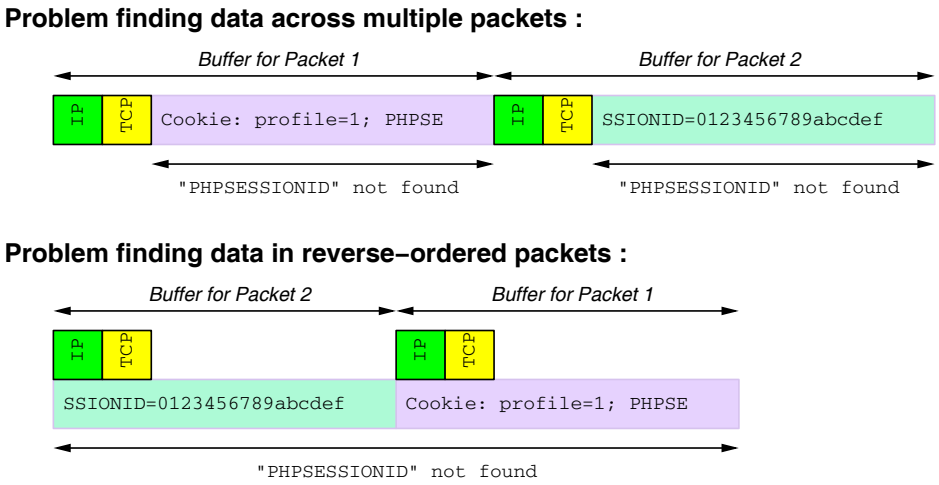

Fig.4 : An HTTP request can span over several packets

2. **reverse-ordered packets** : when a big and a small packets are sent over the Internet, it's very common for the small one to reach the target before the large one. The load balancer must handle this gracefully.

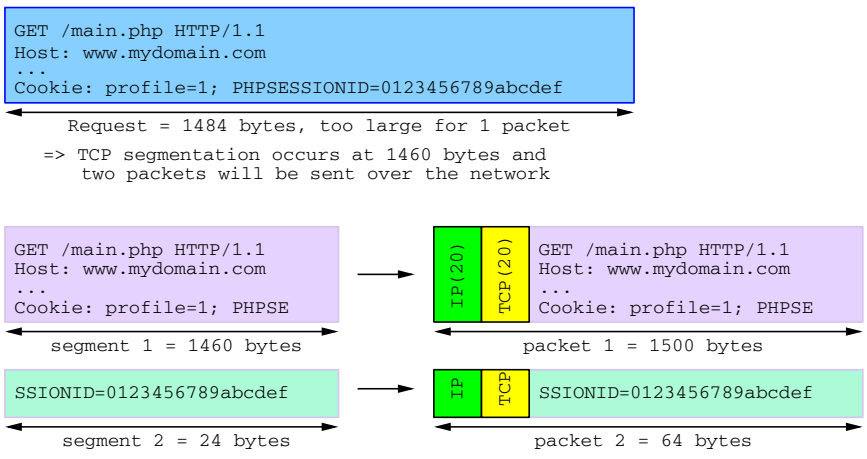

Fig.5 : How HTTP can be difficult to handle on some hardware LB

3. **fragments** : when a packet is too big to be routed on a medium, an intermediate router is allowed to cut it in small fragments. This is rare today on the Internet, but it is more and more common on internal networks because of VPNs which limit the payload size. Fragments are very hard to process because they arrive in varying order, need to be buffered to reconstruct the packet, and only the first one contains the session information. Network-based load balancers generally don't cope well with fragments, which they sometimes qualify as "attacks" as an excuse for not processing them.
4. **packets losses and retransmits** : over the Internet, there are a huge number of packets lost between a user and a server. They are transparently retransmitted after a short delay, but the processing performed on one packet has to be performed again, so the load balancer must not consider that what has been done will not have to be done again (typically when end of headers is reached).
5. **differentiate headers and data** : in HTTP, protocol information, known as headers, lies at the beginning of the exchanges, and data follows the first empty line. Loose packet matching on network-based load balancers sometimes leads to cookies being matched or modified in the data section, which corrupts information. A related problem often happens after a session ends : if the user reuses the same source port too early, the load balancer may confuses it with data and will not perform the L7 processing.
6. **data insertion** : when data (eg: a cookie) is inserted by the load balancer, TCP sequence numbers as well as checksums have to be recomputed for all packets which pass through, causing a noticeable performance impact and sometimes erroneous behaviours.

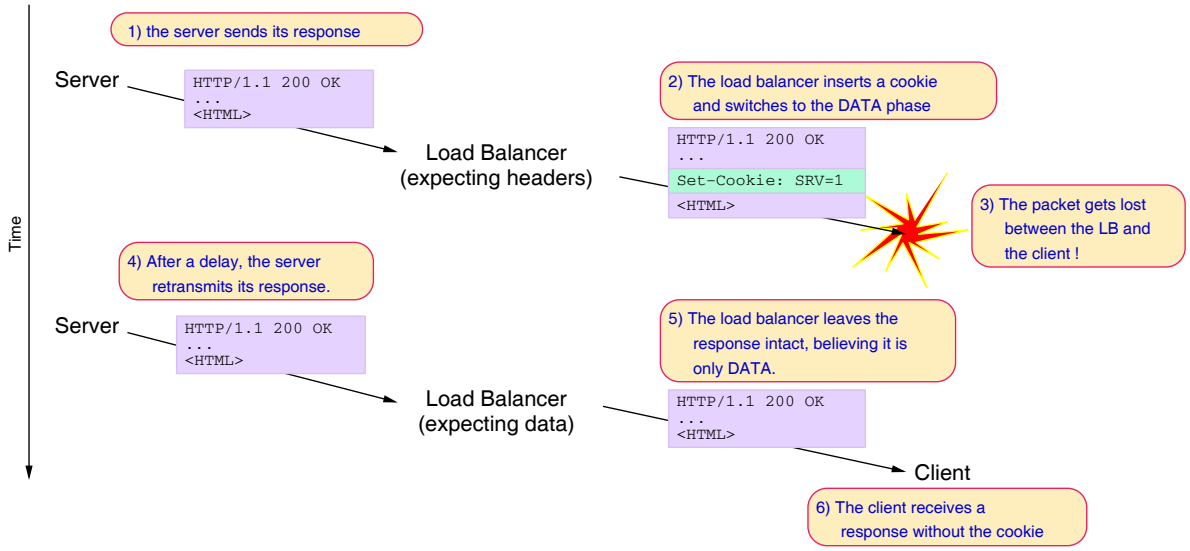

Fig.6 : Common problems in simplified TCP stacks

That said, it is very hard to process layer-7 at the packet level. Sometimes, contents will be mis-identified, and requests will be sent to a wrong server, and sometimes a response will not be processed before being returned to the user. In fact, the full-featured load balancers often involve local proxies to perform the most complicated actions. Some switch-based load balancers transfer part the layer-7 processing to the management processor which is often very small, and others have dedicated processors for this, but generally they are not as powerful as what can be found in latest server hardware. Also, as a side effect, they are hit by the memory consumption caused by buffering.

Generally, they don't announce how many simultaneous sessions they can process in layer-7, or they tend to claim that this has no impact, which is obviously wrong : for a load balancer to be compatible with the Apache web server, it would have to support 8 kB per request, which means keeping up to 8 kB of data per established session at one moment. Supporting 4 millions of simultaneous sessions at 8 kB each would required 32 GB of RAM which they generally don't have. Tests must then be performed by the customer to detect the load balancer's behaviour in corner cases, because easy denials of services are commonly found.

Conversely, the proxy-based load balancer sees nearly no overhead in layer-7 processing, because all problems listed above are naturally solved by the underlying operating system. The load balancer only has to focus on content and perform the right actions. Most often, they will also provide very detailed logs at no extra cost, which will not only offload the servers, but also provide vital data for capacity planning.

Last, layer-7 is something which evolves very fast, due to new persistence methods or content analysis required by constantly evolving application software. While software updates are very easy to provide for an editor, and to install for the customer, complete image updates for hardware load balancers require a higher level of validation and it is very difficult for their editors to offer the same level of reactivity.

THE BEST COMBINATION, FOR THOSE WHO CAN AFFORD IT

The best combination is to use a first level of network-based load balancers (possibly hardware-accelerated) to perform layer-4 balancing between both SSL reverse proxies and a second level of proxy-based layer-7 load balancers. First, the checks performed by the layer-4 load balancer on the layer-7 load balancers will always be more reliable than the self-analysis performed by proxies. Second, this provides the best overall scalability because when the proxies will saturate, it will always be possible to add more, till the layer-4 load balancer saturates in turn. At this stage, we are talking about filling multi-gigabit pipes. It will then be time to use DNS round robin techniques described above to present multiple layer-4 load balancers, preferably spread over multiple sites.

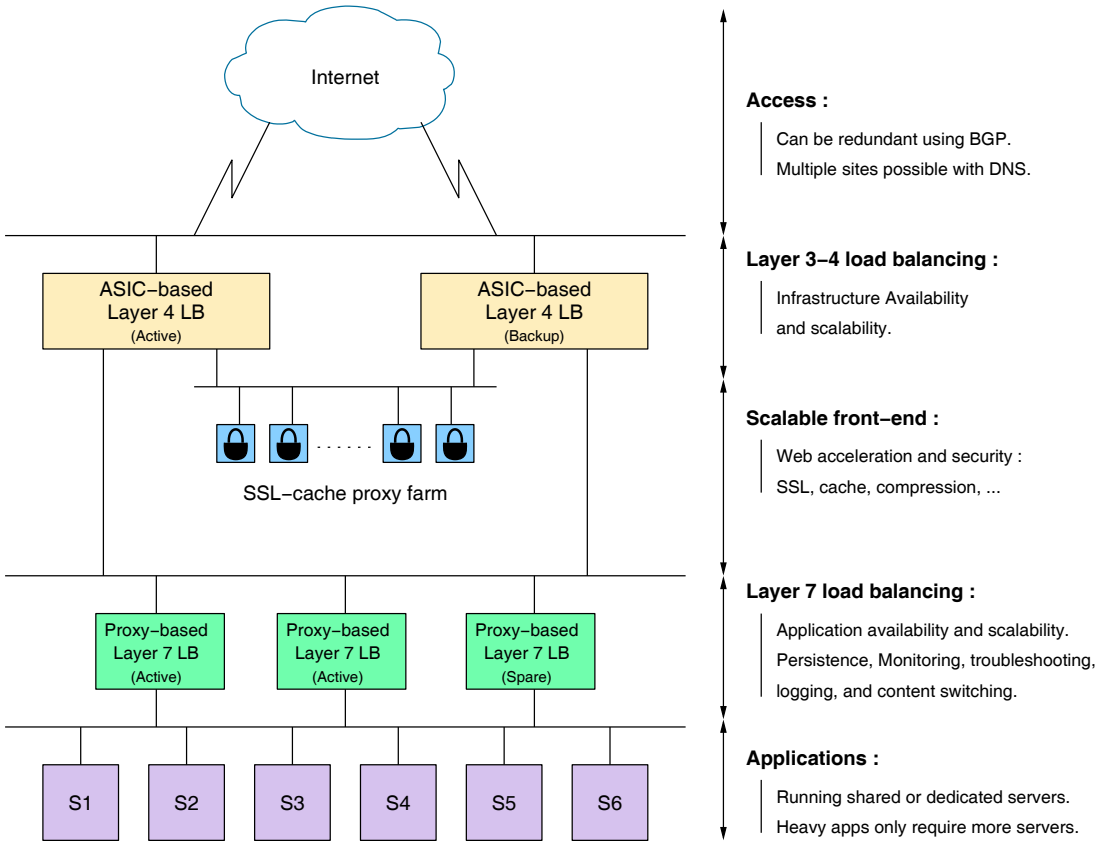

Fig.7 : Typical example of a scalable architecture

Application tuning regardless of the load balancer

Installing a load balancer is good for scaling, but it does not exempt anyone from tuning the application and the servers.

SEPARATE STATIC AND DYNAMIC CONTENTS

It might sound very common, but it is still rarely done. On many applications, about 25% of the requests are for dynamic objects, the remaining 75% being for static objects. Each Apache+PHP process on the application server will consume between 15 and 50 MB of memory depending on the application. It is absolutely abnormal to dedicate such a monster to transfer a small icon to a user over the Internet, with all the latencies and lost packets keeping it active longer. And it is even worse when this process is monopolized for several minutes because the user downloads a large file such as a PDF !

The easiest solution consists in relying on a reverse proxy cache in front of the server farm, which will directly return cacheable contents to the users without querying the application servers. The cleanest solution consists in dedicating a lightweight HTTP server to serve static contents. It can be installed on the same servers and run on another port for instance. Preferably, a single-threaded server such as Lighttpd or Thttpd should be used because their per-session overhead is very low. The application will then only have to classify static contents under an easily parsable directory such as "/static" so that the front load balancer can direct the traffic to the dedicated servers. It is also possible to use a completely different host name for the static servers, which will make it possible to install those servers at different locations, sometimes closer to the users.

WHAT CAN BE TUNED ON THE SERVER SIDE - EXAMPLE WITH APACHE

There are often a few tricks that can be performed on the servers and which will dramatically improve their users capacity. With the tricks explained below, it is fairly common to get a two- to three-fold increase in the number of simultaneous users on an Apache + PHP server without requiring any hardware upgrade.

First, disable keep-alive. This is the nastiest thing against performance. It was designed at a time sites were running NCSA httpd forked off inetd at every request. All those forks were killing the servers, and keep-alive was a neat solution against this. Right now, things have changed. The servers do not fork at each connection and the cost of each new connection is minimal. Application servers run a limited number of threads or processes, often because of either memory constraints, file descriptor limits or locking overhead. Having a user monopolize a thread for seconds or even minutes doing nothing is pure waste.

The server will not use all of its CPU power, will consume insane amounts of memory and users will wait for a connection to be free. If the keep-alive time is too short to maintain the session between two clicks, it is useless. If it is long enough, then it means that the servers will need roughly one process per simultaneous user, not counting the fact that most browsers commonly establish 4 simultaneous sessions !

Simply speaking, a site running keep-alive with an Apache-like server has no chance of ever serving more than a few hundreds users at a time.

Second, observe the average per-process memory usage. Play with the 'MaxClient' parameter to adjust the maximum number of simultaneous processes so that the server never swaps. If there are large differences between processes, it means that some requests produce large data sets, which are a real waste when kept in memory. To solve this, you will need to tell Apache to make its processes die sooner, by playing with the 'MaxRequestsPerChild' value. The higher the value, the higher the memory usage. The lower the value, the higher the CPU usage. Generally, values between 30 and 300 provide best results. Then set the 'MinSpareServers' and 'MaxSpareServers' to values close to 'MaxClient' so that the server does not take too much time forking off new processes when the load comes in.

With only those tricks, a recent server with 2 GB of RAM will have no problem serving several hundreds clients. The rest is the load balancer's job.

More to read about Load balancing

Two competing analysis on this subject, nonetheless interesting lectures :

http://www.zeus.com/news/pdf/white_papers/7_myths.pdf

http://www.foundrynet.com/pdf/wp-server-load-bal-web-enterprise.pdf