Scalable Internet Architectures

George Schlossnagle <george@omniti.com>
Theo Schlossnagle <theo@omniti.com>

Introduction

As worldwide usage of the web explodes, companies and the individuals who power them are constantly forced to ask 'How can I make this application scale.' While presenting a general answer to this question is impossible, we feel that there are enough general principals and particular experiences to fuel a healthy conversation about methodologies for approaching such problems. Thus, this talk is not a recipe of how to scale your particular app but rather an outline of how you might approach answering that problem for yourself (which is in fact a central point in itself: you know your data better than anyone else, and thus you are uniquely suited to answering your scalability problems in an efficient manner.)

The foremost issue is to define what we mean by 'scalability.' The Free On-line Dictionary of Computing defines it thus: Definition: How well a solution to some problem will work when the size of the problem increases.

This is still a very vague definition, so we will refine it to say that a scalable solution must retain the following attributes as demands increase:

- Performance
- Manageability
- Stability
- Efficiency

Performance

Defining performance is perhaps harder than defining scalability. Everyone has their own yardstick for performance - memory statistics, io statistics, cpu utilization statistics to name a few. While all these stats are good for diagnosing system and application problems, they are really poor metrics for benchmarking webserver performance. Webservers exist to provide content to their clients. The client (who either philosphically or financially justifies the existence of the webserver) does not care whether the server is paging hard, has a hot disk or a high load average. The client cares that their page comes back quickly and reliably. System level statistics can provide excellent tools for diagnosing problems, but they themselves are not what is critical to monitor.

Manageability

As systems evolve they naturally tend to increase in complexity, particularly when they combine various custom and 3rd party applications. It's very easy for a collection of simple components to turn into a management nightmare. Do your applications require daily human attention to run correctly? If you are struck by a bus tomorrow, how long would it take your replacements to stabilize all your information systems?

Stability

Stability and manageability are close kin. If you're running a high-capacity website, you need to be available 24x7. As demand grows and your server farms expand, you encounter a double-edged sword. On one hand, the growing client base means that downtime is tolerated less and less, while the increase in components means that the mean time to failure of a component is reduced. It is critical to plan for failure. The loss of any single component can not seriously impact overall site performance.

Efficiency

An oft overlooked element of building any high-capacity system is efficiency Datacenter space costs money. Machine resources cost money. Still, people routinely write inefficient application code, run untuned systems and poorly designed databases, and waste bandwidth. Often times this is a product of overly fast development cycles, or low expected utilization. Just remember it's always easier to build something correctly in the first place rather than rebuild it later when it breaks in production.

In this talk we will talk about design philosophies to improve the scalability of your architecture, and analyze two case studies. Our exact agenda is:

- Choosing a Hardware/Application/Availability Architecture
- Deciding Between Commercial/Open-Source Solutions
- Case Study I: Setting up Local Proxies
- Case Study II: Application-Integrated Caching
- Case Study III: Distributed Logging

Choosing a Architecture

For the purposes of this paper we'll break hardware down into two categories: Enterprise and Commodity. Enterprise hardware is characterized as being highly reliable, easily maintainable, multiprocessor boxen, designed purposely for operating in a high-availability/high-performance environment. Machines in this class tend to expensive and come from vendors like Sun, HP, SGI, IBM, and Compaq. Commodity Hardware is (generally) x86 architecture machines, using 'off-the-shelf' parts. They are cheap, relatively unreliable, and surprisingly fast. Choosing the hardware that's right for you is very much a matter of role. How important is the data? How available does it need to be?

A good rule of thumb for choosing appropriate hardware is how replicateable the data is. Data such as html or php/perl sources are highly replicateable - it's very easy to stick master copies in some location and synchronously copy them out to all your web servers. This is an ideal place for using commodity hardware. An array of Linux or BSD boxes can provide a great deal of horsepower at a fraction of the cost of enterprise equipment. If a single box dies it is easy to replace and the impact is nominal.

A database is a much harder object to replicate, especially in a highly transactional system. Systems that provide real zero-downtime seemless failure are expensive and incur performance penalties. With single-points-of-failure, all steps should be taken to insure platform stability and data integrity. This is a natural place for enterprise hardware and software. Basically, if you have to put all your eggs in one basket, you should make it a good basket.

We will skip over the topic of what overall application platform to choose - this is an almost religous question, and beyond the scope of this talk. That having been said, there are application architecure ideas which can be incorporated anywhere. One critical idea is to bring the data to your applications, not the application to the data. A prime example of this is ssl load-balancing. The 'traditional' method for maintaining ssl sessions (or say a Java App that requires a users session to remain on a single server) is to implement 'sticky' sessions on your load balancer. This is an awful way to approach this problem. Advanced switching features have to be used, any application server death is service-affecting, and it is impossible to do resource utilization based load balancing. A smarter approach is to leverage distributed systems technology (such as Ben Laurie's Splash!) to maintain a distributed session cache, so that sessions can be transparently handled by any machine in the cluster. Splash! currently distributes SSL session keys, but could trivially be modified to distribute application session data.

Deciding Between Commercial and Open Source Systems

There may not be an Open Source solution to your problem…
until you write it yourself.

Choosing between a Commercial and Open Source solution is a hard choice, and not one to be taken lightly. Political considerations aside (which are something for a different talk), how do you make a technical decision on whether to pay for software, use freely available software, or write your own custom software? The traditional reasons for buying commercial software are code 'maturity' and support. Code maturity is based on the idea that a major vendors product has been in many environments before yours, and likely all the bugs and corner-cases you might experience have been troubleshot and fixed already. While this is in many cases trues (especially in widely used products), many sales people will have you believe that code maturity is unique to commercial software. Apache itself is a clear counter-example showing that Open Source products can exibit the same levels of quality as commercial software.

Support is another oft-toted reason to buy commercial products. Paying for support is a good thing. 24x7 uptime demands that problems be resolved quickly and efficiently. The more complex and critical a product, the more reasonable it is to pay for support. There is definite value to being able to pick up the phone at 2am and at least have someone looking at your problem. Does this mean support is a panacea? There are enough poor support service stories for any product to make you think that support is worthless. But once the disatisfaction of having outstanding bugs open for long periods of time wears off, most professional managers will tell you that vendor support (at least from major vendors) comes through 95% of the time. Support doesn't only come from vendors, though. People will always be a technology companies greatest resource. Much as sensible people wouldn't run a large Oracle install without a experienced group of DBAs, if you rely on Apache to make your buisness run, it makes sense to have someone on staff or retainer who knows Apache intimately.

Another important selection criteria is how well a given product fits your needs. Especially in the fast-changing web industry, the market is bloated with products that do 'everything you need'. When does it make sense to eschew existing products and forge your own?

This is a very difficult question. Custom products tend to be buggy (at least compared with popular and widely used commercial and open source products – as they say, many eyes make bugs shallow). Custom products can be difficult to impossible to support (especially if the developers leave your organization). Custom products take time and money (at least in developer time, if nothing else) to develop. Custom products are often less flexible than widely available products which by their nature are designed to fit a number of situations.

The most important thing that custom solutions have going for them is that you understand your data better than anyone else, and thus it is possible to design a solution specifically to your needs. So how do you decide when it's time? If you're not having performance issues, and don't foresee performance issues within your growth expectations, custom solutions are probably not worth the hassle. When you having problems, look at the available offerings: Were they originally designed for your tasks, or was that functionality tacked on in a recent version? Is your data model sufficeintly different from their focus that there are significant performance gains to be made? Can you afford the developer time necessary to implement a robust custom solution?

If a custom software solution you develop in house is not in itself a key business deliverable, consider open sourcing it. Not only is this a positive thing for the open source community, but it is positive for your application as well. Exposing your products to a wide range of people accelerates bug detection, and can help mitigate risk by not requiring the product to be supported entirely in-house.

Now let's analyze in depth some actual applications of these ideas.

Case Study I: Local Proxies

One of the benefits of a N-tier architectureis reducing the number of ‘heavyweight’ application processes by adding a light weight layer in front of it which handles connections to clients. Why is this beneficial? Let’s do the math by looking at the Components of the request time for a web request:

- Network transfer time ~8 packets at network latency at say 80ms latency
- Page generation time say 1/2 second

With these rough figures, network time is over half the transaction time. If we can eliminate or reduce the network transfer time component, we can serve the same amount of traffic requiring fewer processes handling application code. Not only does this reduce the memory requirements on your applications, it also reduces the number of connections made to your database. As applications grow (say past 1000 active connections), maintaining database scalability becomes a challenge. Although creating a distributed database architecture is possible, it’s much more difficult than linearly scaling most applications, so avoiding that is best if possible.

We can extract this same benefit with a minimilistic setup and avoid the hardware redundancy and configuration complexity of a fully 3 tier architecture by setting up a tiny mod_proxy instance on the same host as our applications severs. The proxy instance will handle the high-latency connections to end-clients, while using low-latency connections to a dynamic-content-serving apache server on the same machine.

We start setting up two apache configurations

Proxy Configuration

```
<IfDefine PROXY
DcocumentRoot /var/apache/htdocs
Listen myexternal_ip:80
MaxSpareServers          32
MaxClients               256
MaxRequestsPerChild      0
KeepAlive                off
LoadModule proxy_module libexec/libproxy.so
LoadModule rewrite_module libexec/mod_rewrite.so
AddModule mod_proxy.c
AddModule mod_rewrite.c
ProxyRequests       on
NoCache
ProxyPassReverse    /  http://127.0.0.1
RewriteRule   ^proxy:             -      [F]
RewriteRule   ^(http:|ftp:)       -      [F]
RewriteCond    !^/static/
RewriteRule   ^/(.*\.html)$  http://127.0.0.1/$1 [P,L,T]
</IfDefine>
```

Dynamic Configuration

```
<IfDefine DYNAMIC>
DocumentRoot /var/apache/htdocs
Listen                  localhost:80
MaxClients              40
MaxRequestsPerChild     0
KeepAlive               off
LoadModule perl_module libexec/libperl.so
AddModule mod_perl.c
<Files *.html>
SetHandler perl-script
</Files>
</IfDefine>
```

Things to note about the proxy setup:

- The instance binds to the public ip
- The configuration is setup to allow for high concurrency (MaxClients is 256)
- Any request for a '.html' file outside of the /static directory is proxied to the loopback address
- Keepalives are off.

Things to note about the dynamic setup:

- The instance binds to the loopback address
- The configuration is setup to tightly manage our number of persistent db connections.

Case Study II: Application-Integrated Caching

In our second case-study we will look at a caching technologies. When trying to choose the optimal caching strategy for an application, we need to first answer a few questions about our data.

- Is the data static for long periods of time?
- Is the data static for a short period of time, or is it always completely different?
- Does the data contain components which are static for a short period of time?
- How well does the data match the design of any commercial products being considered?

A good way to illustrate these questions is by designing a caching solution for a hypothetical application.

Web Forums

Our sample application will be a web-forums application. We need to be able to support a multi-board forum site which supports message deletion and ordered display.

Initial Implementation

The initial implementation of the application is very straightforward. Forum entries are stored in a database, and assembled and ordered on the fly. As the size and popularity of the forums grow, the pages take an increasingly long time to run.

Second Implementation

Almost always the easiest caching solution to add to an architecture is a network level caching appliance – either a proxy cache like squid or mod_proxy, or a similar commercial black-box cache. From an application standpoint all we need to make are some slight modifications to our application to enable it to respond correctly to HEAD requests. This method works reasonably well, but still requires a good amount of database accesses. Also, it requires us to install an entire new physical layer to our application infrastructure. It would be nice to avoid this cost.

Third Implementation

To make further improvements to our caching strategy, we will attempt to exploit our specific knowledge about how our application works, and add the caching at a deep level. To execute on this, we will add output caching to our application code, and attempt to avoid entering the application at all when unnecessary by using mod_rewrite to handle our cache searching functionality.

Our plan will be as follows:

- On file request, mod_rewrite will look for the requested file.
- If the file is there, it will served directly.
- If the file is not present, a redirect to a generating url will be issued.
- This generating url will cache the page and return the content.
- Cache poisoning will be handled via an unlink() call if shared storage is used, or via a distributed cache poisoning service if shared storage is not used.

The benefits of this system are:

- No additional hardware, and no additional physical layer of any kind.
- Cache generation and poisoning happen in the application, exactly when they need to happen.
- Minimal database access.

To put this together we use the following rewrite block in our httpd.conf:

```
RewriteCond %{REQUEST_FILENAME} !-f

RewriteRule ^/forums/(.*)$ /admin/generator.php?forumid=$1
```

Generator.php is a script which uses output buffering to capture it's output and cache it.

```
<?php
    $forumid = $_GET['forumid'];
     if(!$uri) {
         return_error();
    }
    ob_start();
    if(generate_page($forumid)) {
        $content = ob_get_contents();
        $fp = fopen($SERVER['DOCUMENT_ROOT'].$uri, "w");
        fwrite($fp, $content);
        ob_flush();
    }
    ob_clean();
    return_error();
?>
```

We need to flush the pages when they are updated as well. If we are using shared storage (perhaps a centralized nfs mount), we can have the update script do something like:

```
<?php
    …
    update_page($uri);
    purge_cache($uri);
?>
```

Where purge_cache() does a filesystem unlink(). Alternatively, if we go with a fully independent cache setup (no shared storage), we can have purge_cache() trigger a cluster-wide cache poisoning via some sort of distributed message bus (for example an XML-RPC server implemented over Spread).

Case Study III: Distributed Logging

In our third case study, we look at the problem of manageing logging in a large cluster. Anyone who has managed a large site knows that consolidating access logs is a major headache. Further it would be nice to be able to perform real-time analysis of the logs.

In our starting architecture, we do local logging on each webserver and then periodically copy and consolidate them on a central server. This sort of scheme in general requires alot of maintenance (especially as the number of machines who need to copy off logs grows) and the merge operation itself is painful for large log sizes. Plus any sort of real-time analysis is impossible.

There are a number of commercial solutions for doing distributed logging, but they're expensive and lack the flexibility we need. There are also Apache modules for logging directly to a database, but for exteremly high-volume sites this can be a bottleneck (especially if you want to query the databse while you're pouring logs into it). In the end, none of these solutions provide quite what we want, so we use a true distributed logging setup, mod_log_spread.

Mod_log_spread works by using a reliable multicast transport to send logs to a multicast group. This allows us to use any number of clients that can join the log group and recieved a real-time stream of logs. Logs can be written to a central file without need for any post-processing. Further, since the logs are being sent to a multicast group, it costs nothing to add an extra client that does real-time processing of the logs or inserts them into a database. In addition to traditional logging, have a freely useable stream of all requests to the cluster offers many interesting possibilities for tracking user session state asynchronously but in real-time.