High Scalability for SSL and Apache

for ApacheCon 2000, London

Prepared by:

Mark J Cox
Geoff Thorpe

Revision 3
25th September 2000

www.awe.com/mark/apcon2000
www.geoffthorpe.net/geoff/apcon2000

ABSTRACT

This paper discusses the deployment of secure web servers, using them as proxies to back-end systems, load balancing SSL, and other issues of performance and reliability for large-scale systems. We investigate the impact of secure transactions and explore innovative approaches to load sharing in multi-server environments. This includes distributing session caches and CPU-intensive operations across machines (and to dedicated hardware), and the optimisation of systems that incorporate cryptographic hardware acceleration and key management.

AUTHOR BIOGRAPHIES

Mark Cox works for Red Hat, Inc in England. He has developed a number of free and open-source software products for more than 9 years; being a founding member of both the OpenSSL group and the Mozilla Crypto Group, a core Apache developer since 1995, and the editor of Apache Week.

mark@awe.com

Geoff Thorpe is a Senior Cryptographic Software Engineer and has extensive experience with cryptography and SSL, particularly OpenSSL and Cryptlib. He's mostly fascinated by crypto, networking, and saturating systems with a combination of both.

geoff@geoffthorpe.net

BACKGROUND

The aim of this paper is to look at the effects of doing secure transactions with your web server; specifically how it affects the performance and layout of your systems. We take a look at some of the standard (http) solutions in terms of machine layouts and configurations being used by high-volume sites and see if they can scale up to handling secure (https) transactions. Although some solutions to this exist in the hardware space, we look at how you may be able to do the same thing using Apache.

Recent publicity has focussed on the issues of keeping your private keys in software, so we take a look at whom this affects and if there are other ways of making your systems secure. We try to find out if hardware-based cryptography devices are a cost-effective solution and give you help on how to make sense of manufacturer claims.

Since the majority of browsers and servers support the SSL and TLS security protocols we will focus on them in particular. Since this is an Apache conference, we are not going to consider other servers or go into too much vendor-specific detail, although a good deal of this material generalises to other servers (and to other security-related services also).

SSL IN APACHE

So what is the difference between a secure and a non-secure connection? When you access a site using the prefix "https" you are attempting to establish a connection using the SSL or TLS protocol. Once that connection is established the requests and responses between browser and server are encrypted end-to-end[1]. The two stages of this connection are; the handshake (authentication and key agreement), and the tunnelling (passing back and forth of the encrypted requests and responses)[2].

SSL Cryptography

Once the browser and the server have completed the handshake, they can encrypt the rest of their communication using a standard encryption algorithm such as DES or RC4. These algorithms are known as ciphers and this technique is called symmetric cryptography. Symmetric cryptography uses a common key for both encrypting and decrypting data, and this cryptography is generally very fast. But as the browser and server may not have interacted before, they need some way of establishing that common key. This is achieved in the handshake by key-exchange techniques that are based on asymmetric encryption (public/private key cryptography). This is commonly based on the RSA algorithm[3] and typically uses a 1024-bit RSA key-pair. As the following table illustrates (showing speed tests of 1024-bit RSA signing on various platforms), public key cryptography can be very CPU-intensive.

Table 1 - RSA sign speeds

Machine	Operating System	Signs/second
Athlon 600MHz	Linux	100
Intel PIII 450MHz	Linux	73
Sparc Ultra 5	Solaris 7	27
IBM RS6000 43P/140 330MHz	AIX 4.3	27
SGI Indy	IRIX 6.4	13

The figures in Table 1 were obtained using all of the available CPU resources on an idle machine and, where available, using assembly-optimised versions of the RSA algorithms.

---

[1] https requests are always server-authenticated and are optionally client-authenticated also.
[2] The SSL/TLS protocols permit either end of the connection to force a renegotiation (new handshake) at any time. Most servers and browsers do not do this (and it would not alter significantly the points being discussed) so we shall not address this further.
[3] Alternatives to RSA exist such as DSA-based certificates and keys, but these are not so widely supported and anyway impose similar computational demands.

So in the worst-case scenario, where each connection to the server requires a new signing operation, this would effectively limit our Sparc machine to handling less than 27 connections a second! Even that assumes that the machine is doing very little other processing to answer the browser's https request besides performing the RSA sign operations.

In practice, the HTTP 1.1 protocol provides the ability to keep a connection open and make multiple requests with it (one after another, but not concurrently). Unfortunately this "keep alive" functionality is not always enabled at the server end[4]. Additionally, we will often find that browsers will open a number of simultaneous connections to the web server for downloading things such as inline images to avoid the latencies of performing each request one after the other.

So in trying to assess the work required by the server for it to provide SSL support, we must examine what cryptographic operations generate significant overhead, and how often they are required. Aside from these RSA sign operations, and once the initial SSL transaction (handshake) has been completed, is the encryption time of the request and response data significant? It turns out that this symmetric encryption has very low overhead for most machines, and the extra effort over sending data unencrypted is relatively negligible. Table 2 illustrates the volume of data various idle machines can encrypt per second using two common SSL ciphers.

Table 2 - Symmetric encryption speeds (in megabits per second)

Machine	RC4 (Mbps)	3DES (Mbps)
Athlon 600MHz (Linux)	541	40
Intel PIII 450MHz (Linux)	408	30
IBM RS6000 43P/140 330MHz	192	17
Sparc Ultra 5 (Solaris 7)	176	14
SGI Indy (IRIX 6)	63	5

Where are your keys?

In order to perform the SSL private key operations, the web server needs to have a copy of the server's private key in its memory whilst it is processing SSL transactions. In early 1999, nCipher found a very efficient way to scan large amounts of data looking for private keys.

Some operating systems in certain configurations allow applications running on the system as the same user to read each other's memory. As CGI programs normally run in this way, anyone who has access to place their own CGI programs on a web server could potentially scan the web server memory to find the private keys of any secure sites on that server. nCipher demonstrated a CGI program that would do exactly this, returning the private key of any secure virtual host running on that web server.

The solution that they gave to this problem was to secure your keys in an external hardware device, specifically in a hardware accelerator. Then, not only does the accelerator perform the key operations, but it also has the only copy of the key. The web server does not need to supply the keys to the accelerator and indeed the usual technique is to generate keys inside the device and never export them. The web server then has to hand off all key operations to the hardware accelerator, and so each web server will need to have its own hardware accelerator attached.

The attack nCipher demonstrated only works on a few operating systems that allow other processes to read each other's memory space, and even on those systems it is simple to defeat the attack. Apache just needs to be started as the root user and allowed to fold-back to a non-root user in order to serve pages (which is actually the default configuration). Even if Apache is started as a non-root user you can only be affected if you allow people to write their own CGI programs and run them on the same web server that is being used for secure transactions. Unfortunately this is not quite the picture painted by recent press releases covering the issue.

SSL Session Caching

In order to speed up the processing of requests, the SSL protocol specifically allows a client to ask the server to resume a previously negotiated session when opening a new connection. So once the client and server have performed the public key operations and negotiated a session key (i.e.

---

[4] Or the keep-alive functionality is configured for low time-outs to keep the number of open connections and processes under control.

completed the handshake), then in theory a new connection can be made at a future time without having to perform the intensive public key operations again.

Each session must have a timeout period associated with it (for security reasons), so the server administrator can force a new session to be established at least every day or every hour for example.

This means that the server must have a way of remembering the sessions if it is to make use of this feature; a session cache. With Apache 1.3 we have the complication that we have a number of independently forked children running on the server which have no common store. Each request that comes into Apache could potentially end up talking to a different child process. In SSL, session resuming is initiated when the client indicates to the server the session it wishes to resume. Even if the client has negotiated sessions with each child process, it will only be able to resume an SSL session if it correctly guesses which child process it has connected to! With heavily loaded sites having tens or hundreds of children this isn't useful, and a session cache shared between all Apache child processes is required.

Although this is a fairly simple concept it has taken a while for secure Apache solutions to get it right. Apache-SSL[5] started with session cache running as an external process. Each child would establish a connection to this process that had a memory cache of sessions.

But this solution turned out to be fairly unwieldy for server administrators, as the web server would have to have its external process restarted in-sync with the web server. Also, problems would occur if the process was killed (or crashed) because the web server would continue to run and make attempts to serve requests that would always fail.

mod_ssl[6] had its own set of solutions. It started out with the Apache-SSL approach then moved to a session cache that was implemented using a lightweight database (DBM) instead of an external process. In the last few months mod_ssl moved to supporting a similar method to Stronghold, using a shared memory cache.

Table 3– Caching techniques used on common SSL solutions

Server	Cache
Apache + mod_ssl (includes Raven, Red Hat Secure Server)	Shared memory (recommended) or DBM file
Stronghold	Shared memory (recommended) or file based
Apache-SSL	Cache server process (TCP/IP or Unix domain socket)

With the new session cache comes the responsibility to configure it correctly. The more sessions that need to be operating concurrently and the longer the session expiry times need to be, the larger the block of memory the session cache will need. Also important is to examine how speed-critical the session cache becomes as a result. All reading and writing in the session cache must be synchronised to avoid data corruption, so if the cache is big and only one Apache child process can communicate with it at a time, then it may itself become the limiting factor to performance rather than solving it. For example, a heavily loaded server running many intensive SSL key operations would correspondingly cause all session cache operations to slow down, which increases the likelihood that child processes will be queuing up for access to it.

As we will see with key operations as well, there is a disadvantage to running an SSL session cache on the same system as a web server. If the cache is too large, the synchronisation of access to it may become the bottleneck, or it may just consume too many shared resources. If the cache is not large enough, then it will soon fill up with entries before they are ready to expire. When the session cache becomes full, the choice is to either honour the session timeouts (in which case new SSL sessions will not be cached at all until entries in the cache begin to expire) or to prematurely expire older sessions. Unfortunately the majority of SSL Apache servers in operation do the former, but neither is particularly ideal.

---

[5] Apache-SSL from Ben Laurie, http://www.apache-ssl.org/
[6] mod_ssl from Ralf Engelschall, http://www.modssl.org/

If it was possible to run an SSL session cache as a dedicated service on another system, then it could be scaled and resourced independently of the web server.

SPEEDING UP THE CRYPTO

Once these software problems are solved and we have a fast working session cache and browsers that support session resuming, it no longer matters as much whether we are using keep-alive connections. The limiting factor now becomes how many new secure requests we can serve per second, giving us an idea of the delay in establishing a new secure connection (and the maximum load level the server can sustain without growing steadily slower[7]).

One advantage that we have over regular http is that secure addresses are rarely advertised, and so are less likely to get a rush of new secure connections over a very short timescale (a load spike)[8]. But in the worst case, if you have 200 new connections to your site and you can only handle 50 signs a second, you've got potential customers waiting at least 4 seconds for that new connection. A report from Zona Research suggests that customers are unwilling to wait more than eight seconds to place their order before giving up (and this has to include the time to do all the processing of the request, not just the SSL part). What solutions to this sort of problem exist?

Higher performance machines

You could buy a faster machine - one that can handle more connections per second. We've already shown that an inexpensive Athlon processor is three times faster at RSA sign operations than the more expensive Sparc Ultra 5 machine. Indeed a whole new range of processors are being developed with engineering in them specifically for accelerating cryptographic operations such as key signing. The Ultrasparc III processor will contain instructions that make public-key cryptography operations more efficient. Intel have also announced that their new Itanium 64-bit processor can perform SSL operations 10 times faster than their fastest current chip, the 32-bit Xeon.

The Crypto Accelerator approach

The alternative to dealing with the crypto operations on the processor is to use some sort of co-processor that can take all that annoying maths out of the hands of your web server. This is the approach that many hardware companies who make cryptographic accelerators would like you to take.

A number of cryptographic accelerators are currently available with major players including Rainbow, DEC and nCipher. You can buy an accelerator board that gives you a specified level of performance for handling the time-consuming RSA operations. When the web server gets a new connection it passes the maths off to the board, which performs the calculation, and sends it back – hopefully in a fraction of the time it would have taken the web server to do the same task.

Once the initial SSL session is established, the accelerator board plays no part in further connections with the same SSL session. Although some of the boards can do symmetric encryption, the overheads of actually sending the information to the hardware to be encrypted or decrypted is often higher (both in terms of latency and system resources) than just performing the operations directly in the CPU.

The leading cryptographic accelerators are usually in the form of add-on hardware, typically connected by PCI or SCSI, and normally allowing multiple units to be chained to one machine to scale performance (and therefore capacity).

---

[7] If a server can sustain 20 new connections a second, but is receiving 40 each second, then it will be receiving connections faster than it is able to deal with them. The system will then eventually grind to a halt (or to the point that the delays become unacceptable to the users and the site begins to receive less than 20 requests a second!).

[8] E.g. when a online shopping advert appears during the middle of a prime-time show, their site's homepage may receive a burst of requests, but the number of browsers actually following through to the secure purchasing section will be less dramatic.

What price/performance should you expect from a board and how can you measure it? These companies usually quote the number of 1024-bit RSA signs per second that the board can sustain so they can be easily compared to other accelerators or our software-only figures. Obviously there will be some latencies introduced because the web server child process has to talk to the board via some software and hardware layers and wait for a reply. However, this "blocking" places no demands on the CPU which will be free to attend to other things, so the increase in latency does not directly affect throughput (but may require more running child processes as a result of the time they spend blocking).

We have investigated the prices and offerings of the major hardware crypto manufacturers, as identified in a report done by ICSA Information Security Magazine. We have removed vendor and model names and just listed statistics from a representative list of products available from some major suppliers that can perform RSA key operations and key-management. These units can offer different security features, which may explain the price variations.

Table 4 - Price/performance of selected hardware crypto units

Machine	Approximate cost (US$)[9]	Signs per second	US$ per sign/sec
Crypto Unit A	12000	300	40
Crypto Unit B	5000	75	67
Crypto Unit C	5000	50	100
Crypto Unit D	2000	15	133
Crypto Unit E	12500	50	250

We can compare this to the costs and performance of doing the same crypto work on some commonly available systems.

Table 5 – Price/performance of selected systems

Machine	Approximate cost (US$)[10]	Signs per second	US$ per sign/sec
Athlon 600MHz (Linux)	1200	100	12
Intel PIII 450MHz (Linux)	1000	73	14
Sparc Ultra 5 (Solaris 7)	3000	27	111
IBM RS6000 43P/140 330MHz	7000	27	259

We can see that if we had three Athlon machines and some way in software of spreading the load between them, then we could produce the same signing throughput as the high-end hardware crypto unit examined for a third the cost. We would also have architecture more easily upgraded and configured, with greater redundancy, and would obviate the need for a controlling system to manage the hardware crypto unit. The software to do this does not currently exist, so at present each machine would have to handle SSL connections and a load balance installed in front of them.

Another problem with deploying hardware crypto accelerators is that running more than one web server requires an accelerator board for each one. If, for redundancy considerations, you have a large number of web server machines, then the additional cost of accelerator boards can soon mount up. This is because the current crypto accelerators are designed for a one-to-one or one-to-many situation (one web server connected to one or more accelerator boards).

Additionally, many sites use hardware crypto units primarily for key-management rather than acceleration. This is the term used when private keys are generated inside tamper-proof units that never export their keys. This makes the keys invisible even to administrators who have access to start and stop the web-server (and who would normally need to know any pass-phrases required for the web server to decrypt private keys). If you have 10 web-servers to handle the loading of database or web-application logic but your SSL requirements are small, you would still need to deploy 10 different hardware crypto units if you wish to use hardware key management. Most of those 10 hardware crypto units would be severely under-utilised.

---

[9] Prices were based on price lists and specifications available to us at January 2000.
[10] Prices from UK suppliers, January 2000, and includes low-spec monitors, software and accessories.

REMOVING THE BOTTLENECK

There are a number of solutions available for load balancing normal HTTP requests, both software and hardware based. The normal operation of a load balancer is to handle an incoming request, pick one from a number of back-end servers, and pass the request to it. Because HTTP is a stateless protocol this works well in practise as each connection (requesting each page, image, etc) can be sent to a different back-end server without any problem. However with SSL we don't want each request being sent to a different back-end server because each new server reached will require a new SSL session to be negotiated – something we would like to avoid. (This problem can also become a noticeable performance-hit in the user's browser too).

Some load balancers try to ensure that the same client always gets routed to the same back-end server, perhaps by looking at the incoming IP address. However this isn't always possible and defeats the principle of load balancing (think of the number of people that come from behind a corporate proxy server all presenting the same IP address).

If you require hardware accelerators, and have multiple back-end web servers handling the SSL connections, then you probably need a hardware accelerator attached to each box[11]. This will mean the hardware accelerator companies will like you, but it will be pretty expensive (and you could be buying a lot more "signs-per-second" than you are actually using).

Since load balancing of normal HTTP connections is well understood, we can look at common ways this is handled and see how they scale to balancing SSL for HTTPS connections.

Round-Robin DNS

The simplest form of load balancing is round-robin DNS. This is where a host name lookup will be converted to one of several possible IP addresses via a DNS server that reorders its list after each lookup. As more server capacity is required, administrators can add more IP addresses to the rotation. This method isn't ideal even for standard HTTP requests because it takes no account of the load on the back-end servers or their operating status. It can also be a problem when a large number of users come from a network that has a proxy and caching DNS server (think of all users within AOL attempting to use the same web server address). In essence, the problem is that the "load-balancing" is being managed outside the server environment rather than inside it. Once a browser has obtained an IP address, it will stick with that one and if many others have obtained the same address (e.g. if a large ISP has cached the DNS lookup) then that IP address will be buried in requests when the others are comparatively idle.

Using SSL with round-robin DNS will at least work, and does have the positive of it being likely that the next request from the same browser will hit the same server allowing more SSL session resuming.

Hardware Traffic Switching

A number of manufacturers produce load-balancing hardware such as the Cisco LocalDirector product that intelligently load-balances traffic across multiple servers. The web servers can then be changed or taken out of service at will. Also, the hardware can monitor the back-end servers and use this information to determine where each request should be sent.

Once again however, with SSL we would then have the problem where a different back-end secure server could be used for each connection (each image on a page for example), requiring a new session to be established with each one. So the overheads go up, the performance goes down, and more servers are required to sustain the load.

The Cisco product seems fairly unique as they have come up with the ability for SSL requests to become "sticky". The hardware looks at the SSL requests and keeps a track of which back-end servers have handled a particular SSL session. It will then try to route new connections to the same back-end server that originally established the session. However, the hardware still needs to

---

[11] This is certainly true if the security policy is to use hardware key-management, but for acceleration-only purposes, having accelerators connected to some machines and not others would require either very complicated front-end load balancing, or some machines being more heavily utilised than others.

take account of spreading load and dealing with any broken or out-of-service web servers and so new sessions may have to be established with other back-end servers over time.

Software Gateway (Apache Mirror Proxy)

A popular way of load balancing normal HTTP connections is to use the Apache proxypass facility to emulate a hardware load balancer. Here a single front-end machine handles all web connections, but it can seamlessly pass on any time-consuming requests (such as working with databases) to back-end servers. You may for example decide that all html and jpeg files are static and so can be served quickly by the local machine, whilst requests for CGI processing get handled by a back-end Unix machine running another copy of Apache. This flexibility can even map different server architectures into the same machine URL space, for example having ASP (Active Server Pages) files mirror-proxied to a back-end Microsoft NT box running IIS.

Of course since a single box is handling all the connections, the idea is just to make the back-end machines do the time consuming tasks, or tasks that the server hasn't the resources to perform (such as communicating with databases or running ASP scripts).
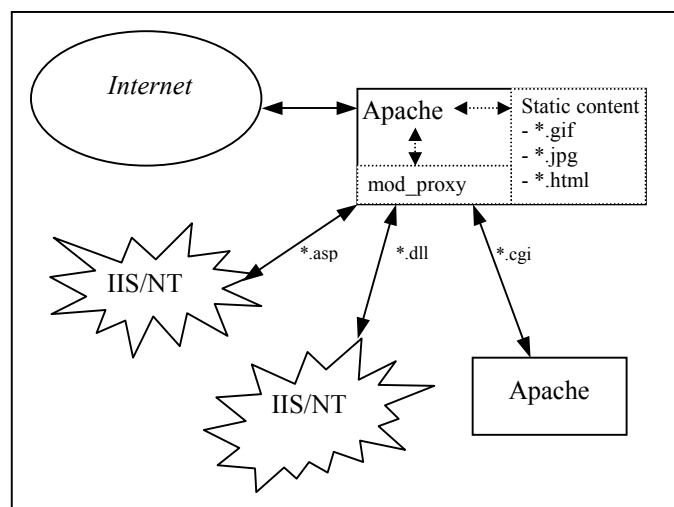
Over the years we have seen many administrators, who were stuck using Microsoft and Netscape servers with export-crippled security, put a copy of a full-strength SSL Apache-based server onto a spare machine to act as an HTTPS-to-HTTP gateway.

Now because simple static pages and images are very quick to serve, the SSL gateway can be configured to serve these pages itself and just pass on other requests. With all cryptography now handled by a single machine, one or more cryptographic accelerator boards could be attached to it to cope with the necessary load. Using this mirror proxy approach works well for balancing SSL requests; you end up with a single machine that can take in SSL requests, establish the sessions as required, and let back-end machines actually perform the other web operations. It is this theme of "componentisation" that we are converging towards, where the resources required to service the CGI or ASP requests can be assessed independently of the resources required to deal with the expected SSL overheads.

However this approach presents a risk-management issue and also doesn't scale well enough for very large sites:

1. There will be a limit to the number of cryptographic accelerators you can add to a single machine.

2. The internal processing and system-level considerations of that one machine will eventually become a bottleneck.

3. The architecture becomes critically dependent on this one gateway - there is no redundancy.

Hardware Gateway

At least one manufacturer has taken the idea of a software gateway one step further by producing a stand-alone hardware solution. The Intel 'iPivot' is a self-contained unit that can be configured in a similar way to the software gateway and includes some crypto acceleration. It normally doesn't have the ability to serve pages locally, and is less easy to upgrade than a standard Apache proxying system. Using a software gateway solution also has a scalability advantage in that you can just switch to a faster processor or add more accelerators as and when required whereas hardware gateways are a little less upgradable when resourcing requirements can change rapidly. Software gateways are also more customisable; allowing client certificate rules to be set up and maintained in-sync with the web-server configurations, and are able to pass on headers to the back-end servers with details such as the encryption algorithms used or details of client certificates.

SOLVING THE PROBLEM

To summarise, having examined the current solutions to SSL-enabling Apache, we have identified the following as major limitations:

a) Session caching needs to be more flexible and not localised to each web server.

b) Resourcing of key operations (such as RSA and DSA) needs to be more scalable and independent of the web server. It should be possible to provide operating resources for each separately. The systems best suited to performing key operations are often not the preferred platform for running web servers and web-applications.

c) Private keys and key operations need to be protected from web server applications. Also, it is important to distinguish between access rights to administrate the web server and access to the private keys.

d) The current architectures don't allow the resourcing of systems to follow the demographics of a site's traffic; certain platforms may be better suited to different tasks, e.g. you may wish to perform RSA operations on Intel platforms but run your web servers on Sparc platforms. Similarly, we may wish to increase our capacity for performing key operations even if our web serving and session caching are sufficiently resourced.

Our solution
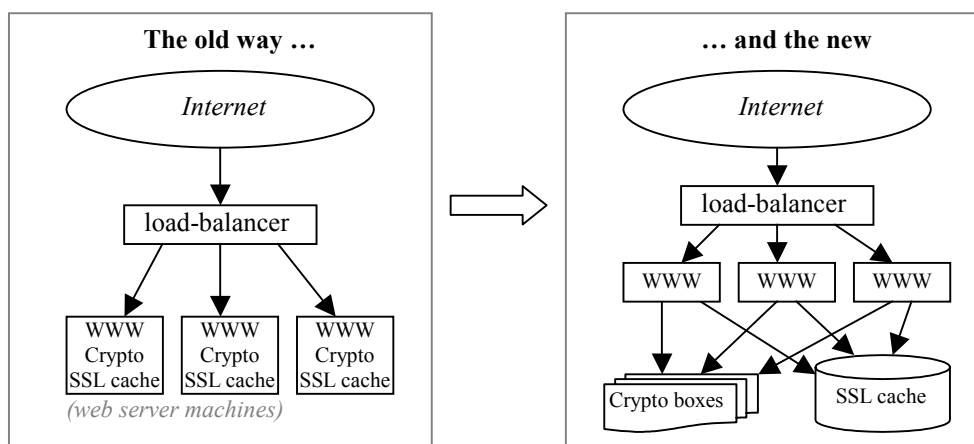
The solution we suggest is not unlike the solution used by many web applications that require back-end databases, namely to separate out the component services and have them communicate through an internal network. As with the database analogy, the latencies involved in processing requests (negotiating a new SSL session and processing an https request) will increase slightly due to network communication, but this facilitates a net gain in system capacity, throughput, and performance as a whole.

Previously there was no latency at all in performing key and SSL cache operations within the web server, but the web servers will have to be resourced to satisfy the most demanding requirements of all three services. The solution we propose does introduce minor, but for the most part constant, latencies in the communication channels between the various services, but system throughput can be maximised and each service can be resourced to meet their own specific demands. Crypto operations perform better on certain CPU types and require little in the way of memory, storage, disk and network IO, or sophisticated operating system functionality. Web servers have virtually the opposite requirements, so in this way the solution provides more choice and flexibility. It can also be seen that heavy loading on one service does not directly impact the performance of the other services. So for example, a sharp spike of new https requests could cause the crypto and SSL cache services to become heavily loaded and slow down, but this will not affect the web servers' ability to serve non-SSL requests <u>at all</u>.

We can summarise how the four problem points detailed earlier are addressed by this approach:

a) Session caching is a dedicated service that can be resourced and customised without touching the "web pool". Also, the same browser can hit any web server at all from one request to the next and continue to resume the same SSL session, without the load-balancer having to attempt any "intelligent" routing. This also helps in the event of a web server failure.

b) Key operations can be performed on any configuration of machines, crypto accelerators, or both and all web servers will share these resources. Not only can the resourcing be maintained independently, but bursts of SSL-only traffic will not cause the web servers themselves to slow down and affect non-SSL traffic.

c) Keys are stored well away from where the web server can get to them - the web server needs only the public keys (certificates) to operate and administrative access to the web servers does not automatically grant access to the critical private keys.

d) We can choose the right number of units of hardware and software for each of these components separately, allowing us to build redundancy into the architecture.

Selected examples

We have previously illustrated a number of ways in which normal site configurations find performance limitations in one form or another. Here we illustrate real-world scenarios where the above model can free us up from most of these limitations and improve scalability, performance, and cost efficiency.

(1) High web server loads, low SSL requirements, and hardware key-management required.

This example is a site that operates many web servers servicing very high loads with only a small SSL requirement (the traffic is predominantly plain HTTP requests). This site also has a policy of using hardware key-management for all private-key generation and storage. As the SSL loads are comparatively low, it is expensive to have to attach a distinct hardware crypto unit to every web server simply because we need hardware key-management. One or two dedicated hardware crypto units, attached to one or two computers, may be all the processing power that is required to service the SSL crypto and caching requirements. Indeed the SSL session cache could well reside on one of the computers controlling the hardware crypto units.

Using the approach we have outlined, these two small systems can provide the desired hardware key-management and session caching, and will be shared by all the web-servers. Any change in SSL requirements can be handled and resourced directly without altering the web server resources. Previously hardware key-management would have necessitated having a separate hardware crypto unit attached to every running web server and there would have been no sharing of SSL sessions between servers.

(2) High SSL loads requesting predominantly static web content.

This second example is a site with a high hit rate but which is serving relatively easy content (lots of static pages and images). A lot of the traffic is through SSL and comes from many different users. Traditionally this has been solved using intelligent load balancing in front of high-specification web server systems. Each web server has to maintain large session due to the number of different concurrent users. Each web server has to perform the crypto operations

quickly, which requires high-specification CPUs, as well as being a good web server that requires a good multitasking system with solid IO performance.

By distributing the crypto and caching operations, an array of cost-effective Intel-based machines could provide all the necessary crypto resourcing. These crypto boxes would need very little in the way of memory, storage, or IO throughput; their role is very much CPU-bound and so can be purchased accordingly. The web servers do then not need to be so numerous now that the crypto overhead has been offloaded elsewhere; they will simply need sufficient disk and network IO performance to process the predominantly static content (and will not need to possess anywhere near the collective CPU power of the machines performing the crypto). Purchasing web servers to perform all services concurrently can be very costly, as these systems must simultaneously satisfy the resourcing requirements of each service, and in sufficient numbers. For example, if the SSL load does not increase but the web serving requirements become more demanding, then more servers (of the same high specifications) will need to be deployed. By distributing these services as distinct components in the architecture, additional resourcing becomes more cost effective, and more finely managed.

How do we do this?

Although most of the approaches we have discussed in this paper are readily available today, a system to implement our final solution is not. However, all of the conceptual ideas discussed (and the solution) can be implemented by extensions to existing open source software.

The authors already have a functioning prototype that distributes key operations from a number of web-servers to a number of key-servers. This prototyped framework is a proof-of-concept, and it is expected that this functionality will be released as open source once completed.