Implementering av prestandatest för att undersöka skalbarheten utav två webbservrar

LUNDS UNIVERSITET
Lunds Tekniska Högskola

LTH Ingenjörshögskolan vid Campus Helsingborg
Datateknik

Examensarbete:
Niclas Tall

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2011

Sammanfattning

Detta examensarbete handlar om hur man konfigurerar, implementerar och utför ett automatiskt prestandatest för att undersöka skalbarheten av två webbservrar. Vad som ska testas och hur man implementerar prestandatestet kommer att beskrivas i denna rapport. Webbservern kommer att testas emot statiskt material, bilder och dynamiskt material med olika konfigurationer som SSL 2048-bitars och Gzip nivå 1,5,9.

Resultatet av undersökningen kan användas som fingervisning för vilken webbserver som är att föredra vid olika typer av belastning och vad som kan göras för att uppnå bättre prestanda. Prestandatesten visade att hastigheten för att leverera statiskt material är nästan 20x snabbare än dynamiskt material för Nginx och 10x för Apache.

Nyckelord: Prestandatest, Apache, Nginx, Skalbarhet, Prestanda, Webbserver

Abstract

This paper discusses how to configure, implement and perform an automatic benchmark to analyze the scalability of two common web servers. How it's implemented and why specific objects are tested will be revealed in this report. The web server is benchmarked with static material, pictures and dynamic material with different configurations such as SSL 2048-bits and Gzip level 1,5,9.

The result of this survey could be used as statement for which web server to prefer in different type of load and how to increase the performance. The benchmark results indicate that requesting static material is about 20x faster than dynamic material for Nginx and about 10x for Apache.

Keywords: Benchmarking, Apache, Nginx, Scalability, Performance, Webserver

Förord

Jag vill tacka min handledare Christian Nyberg, för hans råd och vägledning genom detta examensarbete. Jag även tacka Anders Larsson på Gertrud för hans stöd och att jag har fått arbeta ifrån Gertruds kontor här i Helsingborg. Vill med tacka CityCloud för deras sponsor utav virtuell hårdvara.

Innehållsförteckning

1. Inledning

1.1 Bakgrund

Antalet Internetanvändare har sedan 1995 växt från cirka 16 miljoner till två miljarder (2010). Tillväxten av användare på Internet beror på att fler länder har börjat få tillgång till tekniken och att utvecklingen utav smarta telefonen möjliggör för oss att vara uppkopplade var vi än befinner oss.

Den ökade trafiken över Internet leder till att hemsidor idag också måste kunna hantera mer trafik, samt fortsätta att möta de krav som ställs på en hemsida att vara responsiv. En webbplats består av ett antal grundkomponenter, för att klara den ökande trafiken krävs det att man ser över de olika delarna för att maximera sin prestanda. Ett alternativ är att investera i mer resurser, men det leder till en enbart kortsiktig lösning på ett mer komplext problem.

Det var Gertrud Produktion i Helsingborg som kom med iden till examensarbetet för att se vad det finns för alternativ för att klara av den ökande belastningen. Det är inget specifikt problem för just Gertrud Produktion utan resultatet kan vara av intresse för alla som arbetar med webbservrar. Gertrud Produktion är en webb-/reklambyrå i Helsingborg, bestående av fyra delägare, två anställda och startades för lite mer än ett år sedan.

1.2 Problembeskrivning

Apache HTTP Server släppte sin första version 1995 och har levererat till klienter i över 15 års tid, men det var inte byggt för att hantera den ökande trafiken som vi ser idag. Idag måste populära hemsidor klara av att hantera flera miljoner sidvisningar per dag. För att hantera det måste de ingående komponenterna vara optimerade för det.

1.3 Syfte

Studiernas fokus ligger i att undersöka hur man kan förbättra prestandan och att kunna hantera en ökande belastning. Att identifiera vad det är som kräver resurser och hur det beter sig under hög belastning.

1.4 Förutsättningar

Omfattningen av denna rapport är 22,5 Högskolepoäng vilket motsvarar studier på heltid i 10 veckor. Arbete med implementering och testning kommer att ske mot flera virtuella servrar sponsrade av CityCloud.

1.5 Frågeställningar

- Vilka webbservrar är intressanta att testa?
- Hur kan man testa prestandan och skalbarhet hos webbservrar?
- Vad blev resultatet av testerna?

1.6 Avgränsningar

Antal webbservermjukvaror kommer att begränsas. Detta görs för att istället öka antalet tester och optimera varje program. Detta görs för att försöka få en så rättvis jämförelse mellan dem som möjligt.

1.7 Målgrupp

Personer som sköter servrarna bakom webbplatser (inte personer som tillför material till sidan) och som ser en ökad belastning. Denna rapport kan också vara av intresse för den som vill veta hur ett prestandatest utförs och dessutom på ett automatiserat sätt. Rapporten är skriven för personer som har en grundläggande IT-kompetens.

1.8 Förväntat resultat

Det förväntade resultatet av arbetet är att med frågeställningarna som utgångspunkt ge förståelse av hur en webbserver beter sig under hög belastning. En automatisk testmiljö kommer även att vara utvecklad.

1.9 Arbetsprocess

Arbetsprocessen är utformad i en iterativ modell.
I den inledande fasen av arbetsprocessen planeras arbetet, vilken struktur rapporten ska ha och studier genomförs för att samla in kunskaper om vad som vad som kan behövas. Här implementeras också de olika systemen, olika konfigurationerna som ska testas. Små tester utförs under tiden för att säkerställa att det fungerar.

I den andra fasen planeras hur systemet ska testas, hur implementeringen av det automatiska prestandatestet ska ske och själva implementeringen av det ska utföras.

I den tredje fasen analyseras resultaten, rapporten färdigställs och presentationen förbereds.

2. Grundteori

I detta kapitel kommer de grundläggande tekniska begreppen att förklaras.

2.1 HTTP

HTTP står för Hypertext Transfer Protocol. HTTP är ett nätverksprotokoll i applikationslagret och är grunden för datakommunikation på Internet.

HTTP har till uppgift att vara request-response (förfrågning-svar) modell i klient-server kommunikation. Vanligast är att en webbläsare agerar klient och skickar HTTP-förfrågningar till webbservern. I förfrågnings-headern beskrivs vad som förfrågas och vad som accepteras.

2.1.1 Förfrågan

Förfrågan om en bild:

```
 GET /img/logo.png HTTP/1.1
 Host: www.example.com
 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10.6; rv:2.0.1) Gecko/20100101   Firefox/4.0.1
 Keep-Alive: 115
 Connection: Keep-Alive
```

2.1.2 Svar

Webbservern agera på förfrågan och svarar klienten med:

```
  HTTP/1.1 200 OK
  Accept-Ranges: bytes
  Content-Length: 4071
  Connection: close
  Content-Type: image/png
```

Klienten får svarskoden HTTP/1.1 200 OK vilket innebär att gick rätt till. Klienten får också reda på storleken av bilden, vad det är för MIME-typ och att anslutningen ska avslutas. [1]

2.2 Webbserver

En server som regerar på HTTP-förfrågningar benämns som en Webbserver. Den har till uppgift att leverera material som har åtkomst via HTTP-protokollet.

En webbserver ska även reagera på material från klienten, t.ex. från webbformulär och uppladdning av filer.

Webbservrar kan, förutom att servera material, vara integrerad i system bestående av skrivare, routrar, webbkameror och liknande. [2]

Figur 2.2.1 Ett exempel på två klienter som kommunicerar med en webbserver via HTTP-protokollet.

2.3 Apache HTTP Server

Apache HTTP server är en robust och kostnadsfri HTTP Server-mjukvara. Apache är ett open-source-projekt där källkoden finns att ladda ner. Apache HTTP Server version 1.0 släpptes år 1995 och har idag vuxit till att vara den mest använda mjukvaran för webbservrar. [3]

2.4 Nginx

Nginx (uttalas Engine-X) är en webbserver/omvänd proxy som effektivt använder en servers resurser. Nginx bygger på en asynkron event-driven modell för att hantera förfrågningar. Det ger en mer stabil prestanda under högre belastning. Nginx har stöd för samma grundläggande HTTP-funktioner som Apache. [4]

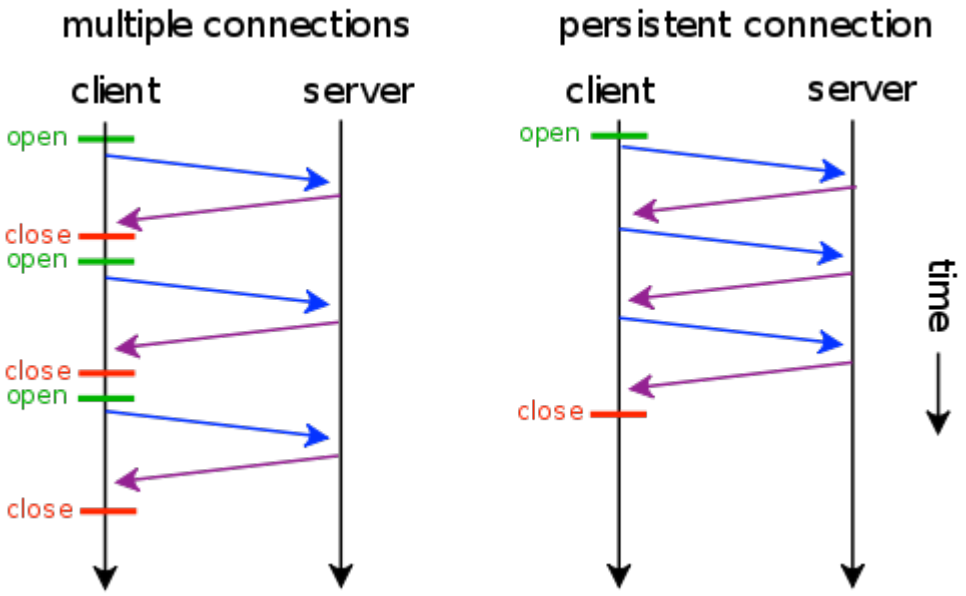
2.5 Förfrågan

Förfrågningar görs av klienter mot en server där ett svar förväntas. Förfrågningarna görs via HTTP-protokollet.

2.6 Keep-Alive

Keep-Alive är ett direktiv som säger att en ihållande TCP (Persistent Connection) anslutning ska används. I HTTP-protokollet version 1.1 är alla anslutningar ihållande som standard.

Fördelarna med ihållande anslutning är att mindre CPU och minneskapacitet används på grund av färre öppna anslutningar. Detta minskar redundansen genom att skapa nya TCP-anslutningar där handskakning måste göras varje gång.



Figur 2.6.1 Bilden visar en icke ihållande- och en ihållande anslutning.

Ett exempel på det kan vara att klienten gör en förfrågan om en html-fil som i sin tur kräver en bild samt en CSS fil för design. För en icke-ihållande anslutning krävs det då att en ny anslutning skapas för varje förfrågan. [5]

2.7 Gzip

Gzip är ett program för att komprimera och dekomprimera filer. Gzip bygger på DEFLATE-algoritmen som är en kombination utav Lempel-Ziv och Huffman-kodning. Ett vanligt användningsområde är mellan klient och server, där servern komprimerar det som ska skickas och klienten dekomprimerar paketet. Det medför att mindre data behöver skickas över Internet. [6]

2.8 SSL/TLS

SSL/TLS är ett krypteringsprotokoll för att säkerställa säker kommunikation över Internet. SSL är det kända namnet, men är idag förlegat och den korrekta förkortningen är TLS (Transport Layer Security).

TLS använder sig av symmetrisk kryptering för integritet och MAC (Message Authentication Code) för att säkerställa meddelandets tillförlitlighet.

TLS passar bra för att skydda trafiken över Internet, eftersom krypteringen kan ske enbart om en sida är verifierad genom att användaren granskat serverns certifikat. [7]

2.9 Prestandatest (Benchmark)

Är ett försök till att mäta prestanda av ett program/system, genom att utföra ett antal olika tester. Att prestanda-testa en webbserver görs genom att simulera förfrågningar mot webbservern och mäta genomströmningen över tid.

Resultaten kan sedan användas för att jämföra olika konfigurationer av program/system.

2.10 Skalbarhet

Med skalbarhet menar man hur väl ett system klarar av att hantera en ökad användning utan att förlora sin prestanda eller nå ett tak. Det är ett viktigt attribut för webbservrar eftersom populära hemsidor besöks från ett globalt perspektiv och ett stort antal av förfrågningar då måste hanteras.

2.11 ApacheBench

ApacheBench (ab) är ett verktyg för att prestandatesta HTTP-webbservrar. Det var från början enbart tänkt för att prestandatesta Apache HTTP Server, men den visade sig vara tillräckligt generisk för att testa andra HTTP-servrar.

I ApacheBench finns där bland annat möjligheter till att ställa in antalet samtidiga förfrågningar, max antal förfrågningar, om Keep-Alive ska användas och om anslutningen ska krypteras. [8]

2.12 JavaScript

JavaScript är ett prototyp-baserat, objektorienterad skriptspråk som är dynamiskt och svagt datatypdefinierat. JavaScript är mest förknippat med klient-programmering som en del av webbläsaren för att förstärka användargränssnittet.

JavaScript kan också användas på serversidan (Node.js), kommando-prompt-program och grafiska skrivbordsprogram. [9]

2.13 MongoDB

MongoDB är ett projekt med öppen källkod som började utvecklas 2007 av 10gen. MongoDB är en skalbar, schemafri och dokument-orienterad databas. MongoDB tillhör kategorin NoSQL databaser. Det som skiljer klassiska RDBM (ex MySQL) och MongoDB är möjligheten att kunna lagra komplexa strukturer i ett dokument och exekvera frågesatser utan att behöva förena (Join) resultat med varandra som i exempelvis MySQL. Det gör att informationssökningar och uppdateringar av dokument utförs snabbt. [10]

2.14 Node.js

Node.js är ett event-drivet I/O ramverk byggt på JavaScript-motorn V8. Det är skapat för att kunna skriva skalbara nätverksprogram som exempelvis webbservrar. Programmeringsspråket som används är JavaScript som exekveras på seversidan.
I Node.js utförs nästan inga funktioner som direkt blockerar I/O, detta leder till att Deadlock inte kan uppstå. Det hanteras genom callbacks, för att istället vänta på ett resultat från I/O kan Node.js exekvera annan kod under tiden. När I/O är färdig anropas callback-referensen och resultatet hanteras. [11]

2.15 PHP

PHP är ett skript-språk utvecklat för att webbutvecklare ska kunna skapa dynamiska webbsidor. PHP kod innesluts bland HTML-kod för att senare hanteras av en webbserver som vidarebefordrar det till en PHP-modul som generar webbsidan.

PHP är idag det mest använda språket för att generera dynamiska webbsidor. [12]

2.16 Wordpress

Wordpress är ett blogg- och innehållsverktyg skapat med hjälp av PHP och MySQL. Wordpress är ett projekt med öppen källkod och är kostnadsfritt att använda. [13]

Idag är Wordpress ett fullfjädrat CMS och det är idag cirka 20 miljoner webbplatser som är byggda i Wordpress. Enligt en rapport från waterstone.com, använder cirka 12 % av Alexa’s topp en miljon sidor Wordpress. [14]

2.17 Statiskt material

Statiskt material är material som sparats på hårddisken och levereras som det var lagrat. Det till skillnad från dynamiskt material som måste genereras innan det skickas. Statiskt material kan vara html-, javascript-, cssfiler etc. Statiskt material går väldigt snabbt att leverera till klienten.

2.18 Ubuntu

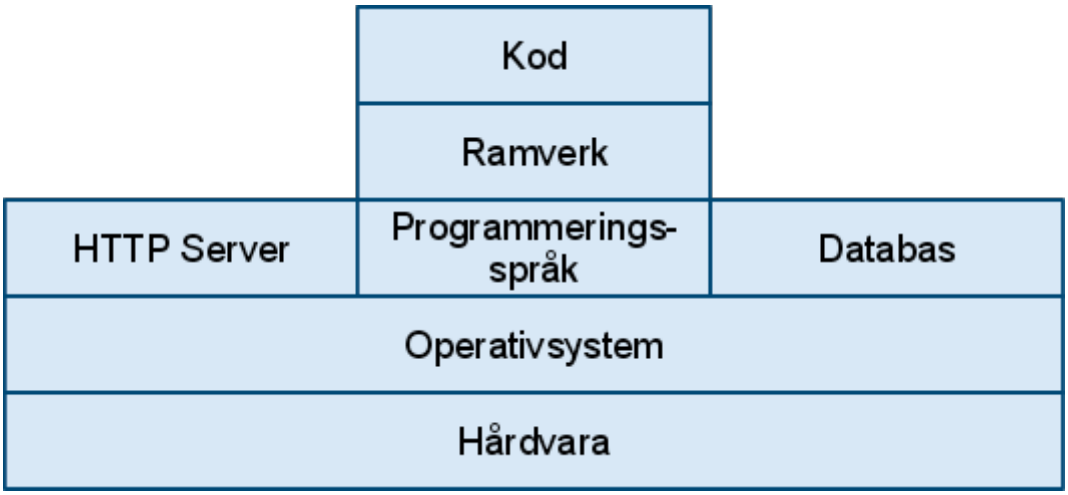
Ubuntu är ett operativsystem baserat på Debian Linux distribution och utgivet som fri och öppen mjukvara. Ubuntu finns som skrivbord-, netbook- och serverversion. Idag har Ubuntu en marknadsandel på 50 % av Linux skrivbordsanvändning.

Linux är en UNIX-liknande kärna som var skapad för Pc-datorer och som släpptes fritt på Internet. Varierande Linux-distributioner har länge varit mest populärt bland serveroperativsystem och representerar idag sex av världens topp tio servrar. [15]

3. Förstudier

3.1 Arkitektur

En webbserver består av olika lager utav system som ska kommunicera med varandra, för att hantera en ökad belastning gäller det att se över de ingående delarna för att uppnå ett så bra resultat som möjligt. De olika lagerna illustreras i bilden nedan.



Figur 3.1.1 Skiss över vilka lager en webbserver kan bestå av

3.2 Hårdvara

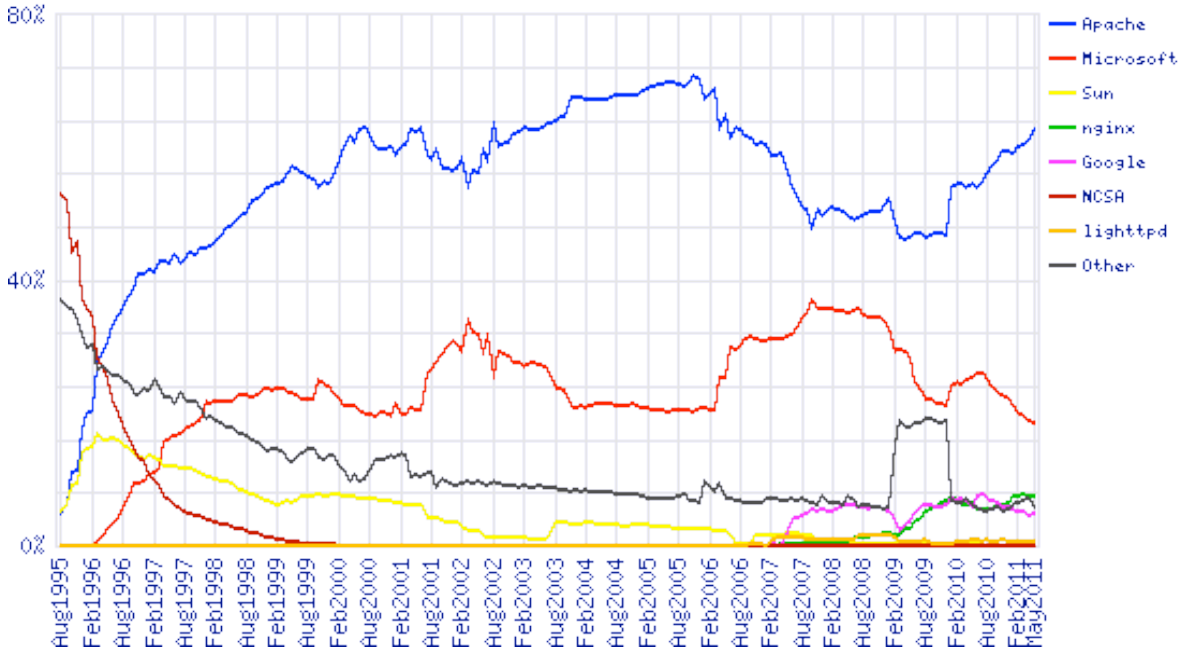
Hårdvaran som kommer att användas under hela projektet är virtuella servrar från CityCloud. Det är Dell bladservrar med nätverksnoder för hårddiskar. Fördelarna med att använda sig utav virtuella servrar är att man kan starta flera servrar på samma fysiska maskin. Det gör att man kan installera flera operativsystem och testa olika konfigurationer utan att de påverkar varandra.

3.3 Operativsystem

Operativsystemen som CityCloud erbjuder är CentOS, Debian, Fedora, FreeBSD, Gentoo, RedHat Enterprise, Ubuntu och Windows 2008 Server. Valet blev Ubuntu Server, version 10.04 LTS, på grund av tidigare erfarenheter, att det finns en bra community bakom det och att både Apache och Nginx fungerar på det.

3.4 HTTP Server

Valet av vilka webbservermjukvaror som ska testat har valts genom att titta på Netcraft's statistik, där över cirka 325 miljoner webbplatser har indexerats. Netcraft har fört statistik sedan 1995 vilket ger en pålitlig bild över hur marknadsfördelningen ser ut. [16]



Figur 3.4.1 Netcraft's statistik över de populäraste webbservermjukvarorna

Utvecklare	April 2011	Andel	Maj 2011	Andel	Skillnad
Apache	191 139 966	61.13%	203 609 890	62.71%	1.58
Microsoft	58 867 097	18.83%	59 646 778	18.37%	-0.46
Nginx	23 463 669	7.50%	23 850 265	7.35%	-0.16
Google	14 690 422	4.70%	16 219 824	5.00%	0.30
lighttpd	1 862 963	0.60%	1 884 876	0.58%	-0.02

Apache HTTP Server är en given kandidat till testet eftersom den har en marknadsandel på 62 % och är det absolut vanligaste alternativet när en webbplats ska skapas.

Microsofts IIS mjukvara kommer inte att ingå i testerna eftersom den har haft en stadigt minskande trend samt att licensen för att använda det är inte kostnadsfri.

Nginx kommer att ingå i testerna, det på grund av att den på 10 år har skaffat en marknadsandel på 7,35 %, vilket är imponerande med tanke på att Apache, Microsoft och Sun har varit med sedan 1995. Men det som imponerar mest är att Nginx används av flera stora populära hemsidor som Wordpress, Hulu, GitHub, SourceForge med flera. Detta gör Nginx till en väldigt bra kandidat att ha med i testen.

Google är förtegna att berätta vad Google Web Server är, men säger att det är skräddarsydd webbserver för linux. Därför utgår Google från testen.

Lighttpd är av ungefär samma typ som Nginx, de bygger båda på samma event-drivna modell och därför utesluts den med.

3.5 Databas

Valet av vilken databas man använder beror mycket vad för typ av data man ska spara, ifall skriv- eller läs operationer utförs ofta och ifall ACID är ett krav.

Wordpress är implementerad i en relationsdatabas, specifikt MySQL. Därför kommer arbetet att begränsas till att inte prestandatesta på andra relationsdatabaser eller NoSQL.

3.6 Programmeringsspråk

Det mest populära språket inom webbprogrammering är PHP: Hypertext Preprocessor.
Det finns alternativ till PHP som exempel ASP.NET, Python och Perl.

Wordpress har en marknadsandel på cirka 50 % bland CMS och är programmerat i PHP.

4. Konfigurering

I detta kapitel avhandlas viktiga konfigureringar för att uppnå bättre prestanda.

4.1 Ubuntu

Som standard i Ubuntu är flera variabler tilldelade ett värde som fungerar utan problem för det flesta användare.

Som standard tillåts endast 1024 filer att vara öppna samtidigt, vilket begränsar antalet samtidiga anslutningar till 1024. För att öka det till 65000 utförs exekvering av följande kommandon:

```
echo “root  soft  nofile 50000” >>
etc/security/limits.conf
echo “root  hard  nofile 65000” >>
etc/security/limits.conf
```

echo “session required pam_limits.so” >> /etc/pam.d/common-session
För att validera utförs:

```
ulimit -n
```

En annan variabel som ska ändras är kö längden för en lyssnande socket.

```
echo "net.core.somaxconn = 6000" >> /etc/sysctl.conf
/sbin/sysctl -p /etc/sysctl.conf
```

4.2 Apache

Apache använder modulen Prefork som standard i Unix. Prefork använder multipla processer, där varje process har en tråd. Processerna ansvarar för förfrågningar. Det ger en ökad säkerhet och stabilitet, om en process/tråd kraschar påverkar det endast den förfrågningen.

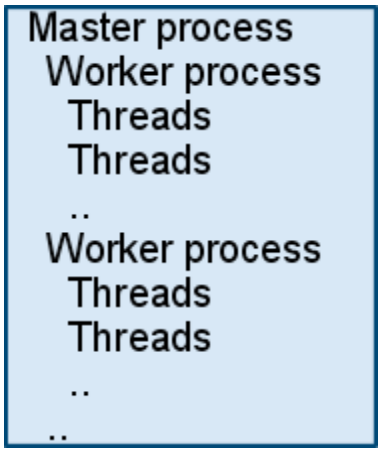
Detta innebär att för varje förfrågan som görs skapas det en ny process. Att skapa en ny process och att byta kontext är dyrt, det kostar CPU tid och minne. Detta leder till att detta inte är ett bra alternativ för hantera stora mänger förfrågningar. Som alternativ till Prefork finns det en Worker-modul.

Worker är en hybrid-modul som använder flera processer, där varje process har många trådar och där förfrågningar hanteras av varje enskild tråd. Detta leder till minskad resursanvändning men har fortfarande god säkerhet och stabilitet. Detta är ett bättre alternativ för att kunna hantera större belastning.

Konfigurationen för worker-modulen är inställd enligt:

```
StartServers           2
MinSpareThreads       25
MaxSpareThreads       75
ThreadLimit           64
ThreadsPerChild       25
MaxClients           150
MaxRequestsPerChild   0
```

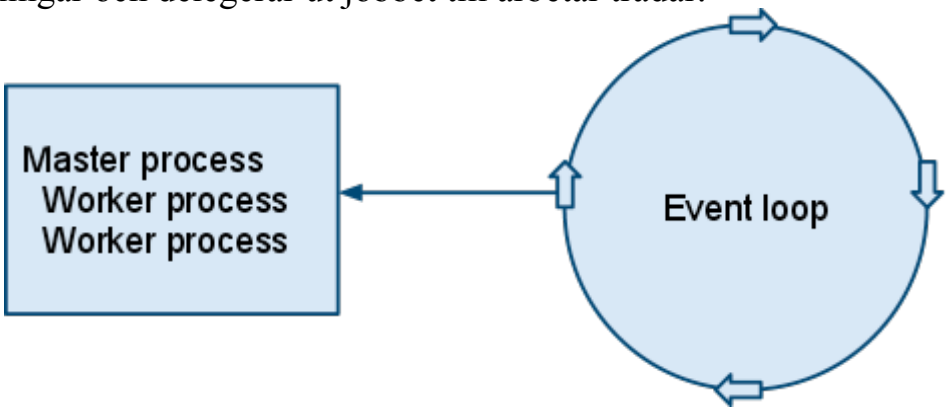
Versionen som används är Apache 2.2.14.



Figur 4.2.1 Visar hur varje worker process startar upp flera trådar för varje anslutning som sker.

4.3 Nginx

Nginx bygger på en main-event-loop tråd som hanterar inkommande förfrågningar och delegerar ut jobbet till arbetar trådar.



Figur 4.3.1 Bilden visar hur event loopen reagerar på anslutningar och meddelar master processen som delegerar arbetet till worker processerna.

CPU'n som används under prestandatesten har två kärnor, därför används inställningen `worker_processes 2;`

vilket tillåter Nginx att starta två arbetarprocesser, det tillåter Nginx att kunna använda fler processorkärnor och det kan leda till minskad fördröjning av hårddisk I/O.

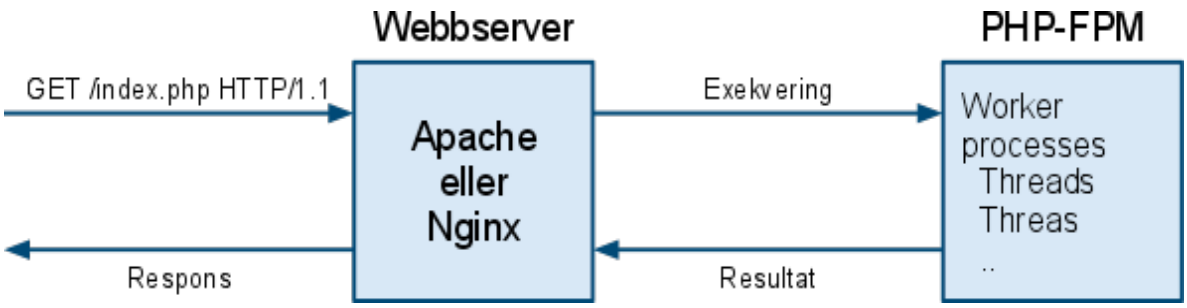
För varje `worker_processes` går det att ange hur många tillåtna anslutningar den tillåter genom `worker_connections 10000;` Antalet anslutningar Nginx kan hantera då är `maxClients = worker_processes * worker_connections.`

En viktig inställning som ökar prestandan är att använda `use epoll;`, det är ett modernt API för att notifiera när ett I/O-event sker, exempel på det är när en användare försöker att skapa en anslutning till servern. För att använda epoll krävs Linux kernel 2,6+.
Versionen som används är Nginx 1.0.0.

4.4 PHP

I en installation av Apache och PHP integreras de två programmen ihop av Apache-modulen `mod_php`. I Nginx används FastCGI för att kommunicera med PHP, FastCGI är ett interface som är skalbart och programmerat för att snabbt kommunicera med webbserverns skriptspråk. I Apache finns det idag stöd för FastCGI med hjälp av modulen `mod_fcgi` men den valdes att inte användas.

PHP-FPM blev valet att använda på grund av det bygger på FastCGI, det tillåter att exekveras i seperata processer, kan hantera pooler av arbetar-trådar, tillåter mjuk start/stop och fungerar tillsammans med både Apache och Nginx.



Figur 4.4.1 Bilden visar hur webbservern delegerar det dynamiska innehållet som ska genereras till PHP-FPM och väntar på svar.

5. Prestandatest

Alla prestandatest kommer att utföras lokalt på servern för att utesluta nätverks-fördröjningar.

5.1 Verktyg

För att prestandatesta en HTTP-server finns det några olika alternativ.

- ApacheBench
- JMeter
- Curl-Loader
- OpenSTA
- HTTP Test Tool
- Httperf

Valet blev ApacheBench för att den har de funktioner som behövs och den var lätt att komma igång med eftersom den ingår i Apache Utilities. Den har inställningar för antalet samtidiga anslutningar, antalet förfrågningar, ifall krypterad anslutning ska användas, möjlighet till egenanpassade headers och en utförlig rapport levereras efteråt.

5.2 Protokoll

Alla tester som utförs kommer att följa nedanstående protokoll:

- Samma hårdvara/UNIX för alla tester.
- Samla in information om systemets nuvarande form (Drifttid, ledigt minne).
- Testen ska utföras tre gånger, det bästa resultatet används.
- Efter varje test ska en systemomstart utföras.
- Resultatet ska sparas i en databas på en annan server.
- Alla förfrågningar ska skickas med “Connection: Keep-Alive” headern.

5.3 Testfall

Testen numreras enligt x.y.z
x = Vilken webbserver.
y = Vilket objekt.
z = Vilken konfiguration som används.

5.3.1 Webbserver

Apache

Referens nr 0.

Nginx

Referens nr 1.

5.3.2 Objekt

Olika objekt kommer att levereras för att se hur webbservern hanterar uppgiften. Det som levereras är en statisk sida, en bild och en dynamisk sida.

Tanken var att ren installation av Wordpress startsida skulle ingå i testerna. Det testet har uteslutits på grund av att det är en kombination utav att skicka statiskt material, bilder och dynamiskt material hämtat från en databas. Då detta inte ger något värde är det bättre att analysera de enskilda komponenterna var för sig.

Statisk sida

Referens nr 0.
Den statiska sidan är en HTML sida. Se Appendix A för dokumentets struktur.

Bild

Referens nr 1.
En bild på 565KB kommer att användas, det för att se hur webbservern hantera att skicka större filer som inte är text. Se Appendix B för bild.

Dynamisk sida

Referens nr 2.
En dynamisk sida är en fil som exekveras på servern för att hämta in dynamisk information. Det kan exempel vara beroende på vad som skickats in via POST/GET parametern. PHP scriptet som kommer att testas är ett enkelt skript. (Den accepterar en inparameter, kör en for-loop ett antal gånger samt skriver ut information om server miljön.) Se Appendix C.

5.3.3 Konfiguration

Tre olika typer av konfigurationer kommer att testas. Standard för att skicka som det är, SSL/TLS för krypterad förbindelse och Gzip för komprimering.

Standard

Referens nr 0.
Mätningen utförs utan SSL och Gzip.

Kryptering (SSL/TLS)

Referens nr 1.
Testen kommer att utföras med säker SSL/TLS förbindelse. Certifikatet som testas emot är ett egen signerat x509 certifikat med 2048-bitars publik nyckel.

Komprimering (Gzip) nivå 1

Referens nr 2.
Gzip är ett program för fil komprimering/dekomprimering, det finns 1 - 9 nivåer, där 1 är minst komprimerat (snabbast) och 9 hårdast komprimerats (Långsammast)

De nivåer som kommer att testas är 1, 5, 9.

Komprimering (Gzip) nivå 5

Referens nr 3.

Komprimering (Gzip) nivå 9

Referens nr 4.

Uppdatering av konfigurationstester

Test med kombination utav kryptering (SSL/TLS) och Gzip 1,5,9 utgår eftersom det i teorin endast bör addera tiderna av de enskilda uppgifterna.

Ett prestandatest på Nginx servern utfördes med en användare, 1000 förfrågningar utan Keep-Alive headern, varje test utfördes tre gånger och bästa resultat används. Resultaten från testet följer i tabellen nedan.

Gzip9 + SSL	Gzip 9	SSL	Differens
10.192s	0.331s	9.929s	-0.068s

Eftersom avvikelsen från teorin ($\tau_{Gzip9+SSL} - \tau_{Gzip9} - \tau_{SSL} = 0$) ligger runt 1/15 av en sekund, är det rimligt att utesluta de tre konfigurationerna från prestandatesten.

5.4 Belastning

Antalet samtidiga anslutningar kommer att ökas enligt:

$$2^x, x = 5,6,...,10$$

För att simulera verkligheten när en användare besöker en sida, granskades Alex's Sweden Top 25 Sites [17], där de högst rankade .se domänerna som inte är nyhetssidor valdes. Anledningen till att nyhetssidor inte var av intresse är på grund av att de listar det mesta av sitt aktuella innehåll på förstasidan samt även på undersidor.

De utvalda webbsidorna granskades manuellt genom att besöka varje och notera antalet förfrågningar som utförs för att ladda startsidan.

Domän	**Förfrågningar**
Blocket.se	45
Swedbank.se	44
Tradera.se	60
Blogg.se	57
Medelvärde	52

Antalet förfrågningar som ska utföras räknas ut enligt

$$requests = 2^x * 52$$

Exempel vid $2^7 = 128$ samtidiga anslutningar utförs 128*52 = 6656 förfrågningar mot webbservern.

5.5 Implementering

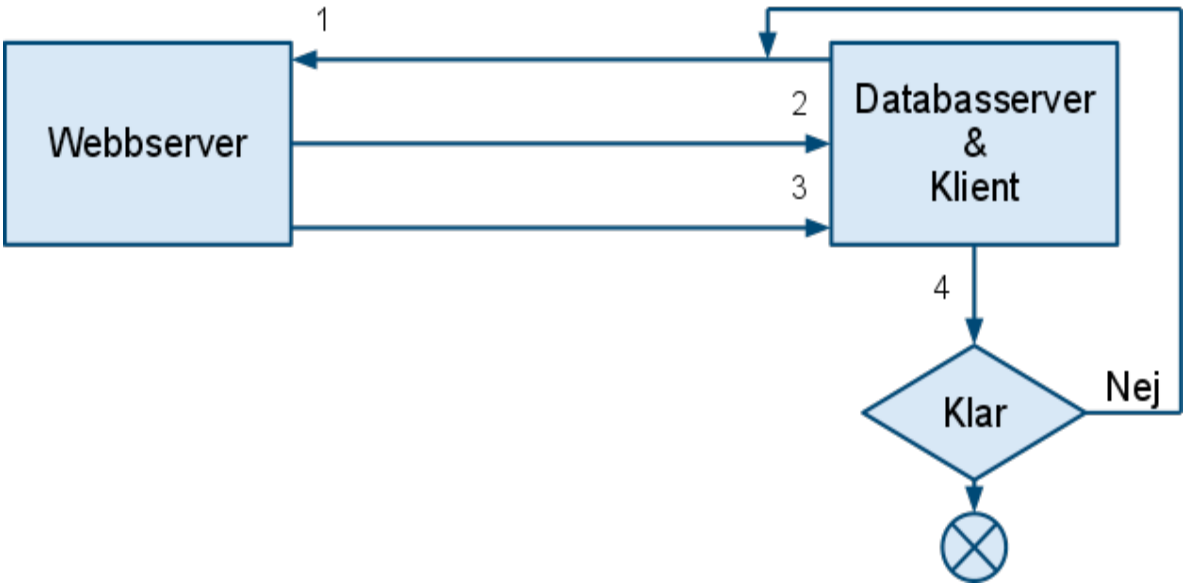
I detta kapitel kommer det att förklaras hur det var planerat att genomföra prestandatestet. Ett automatiserat prestandatest var tvunget att skapas för att genomföra de 450 st. olika testfallen. Att utföra testerna manuellt var aldrig aktuellt, istället togs tillfället i akt att försöka programmera ett automatiserat prestandatest. Valet av programspråk blev Node.js för att det är ett nytt programmeringsspråk, men som bygger på JavaScript. Det är också ett relativt nytt sätt att programmera på, där inga funktioner stoppar exekvering av koden, utan allt bygger på callbacks som utförs när uppgiften är klar.

5.5.1 Kommunikation

Det automatiserade testet körs ifrån en ny virtuell server, vars syfte är att agera testcentral och "klient" och som skickar kommandon till webbservern som skall prestandatestas. SSH används för att skapa en säker kanal mellan klienten och webbservern. För att inte behöva ange ett lösenord när man försöker ansluta till servern skapades nycklar för identifiering, den privata nyckeln sparades på webbservern och den publika nyckeln hos klienten.

För att utföra ett kommando på servern utfördes följande kommando på klienten:

```
ssh -i /path/to/pub.key user@ip "command"
```



Figur 5.5.1.1 Skiss över kommunikationen mellan klient och server

1. Klinten utför `ssh -i /path/to/pub.key user@ip "command"`.
2. Webbservern sparar resultatet på databasservern.
3. Webbservern har exekverat prestandatestet och meddelar klienten.
4. Ifall det finns fler tester att utföra, fortsätt.

5.5.2 Klientsidan

På klientsidan handlar det enbart att loopa igenom alla tester som ska utföras. I pseudo kod blir det

```
För varje server
  För varje objekt
    För varje konfiguration
      För varje konstant av antal förfrågningar
       För varje iteration
         ssh -i /path/to/pub.key user@ip "node
serverbench.js testId=0.0.0 n=1664 c=32 itr=0"
         Vänta 40 sekunder för att webbservern ska
startas om
```

För detaljer hur det är implementerat se Appendix E.

5.5.3 Serversidan

På serversidan finns programmet `serverBenchmark.js` som har testfall, antal förfrågningar, antal samtidiga anslutningar och vilken iteration som inparametrar. De skickas utav klienten.

Det som utförs först är att filtrera inparamterarna och översätta test id:et till vilken webbserver som ska startas och vilken konfiguration som ska användas.

Efter det samlas information in om hur mycket minne som finns totalt och hur mycket av det som är ledigt. Funktion för minnesövervakning konfigureras till att utföras periodiskt med periodtiden 100ms.

Efter det startas ApacheBench med de korrekta inparametrarna. När det har exekverat klart stoppas funktionen som övervakar minnet. Sedan filtreras datan från filer och sparas i en databas. När det är klart utförs en omstart av servern.

För detaljer hur det är implementerat se Appendix F.

5.5.4 Lagring av data

Lagring av datan sker i textfiler på webbservern som backup men det sparas även i databas på samma server som klienten ansluter ifrån. Anledningen till att spara det i en databas är att det blir mycket lättare att söka och filtrera. Dessutom kan till exempel ett PHP-skript skapas för att hämta och presentera datan på ett sätt som senare kan importeras för att skapa grafer.

Valet av databashanterare blev MongoDB, det på grund av att det är en NoSQL databas som det just nu pratas mycket om och att kunskapen om det var liten. I detta projekt skulle en relationsdatabas exempelvis MySQL fungerat med.

Se Appendix D för dokumentet struktur.

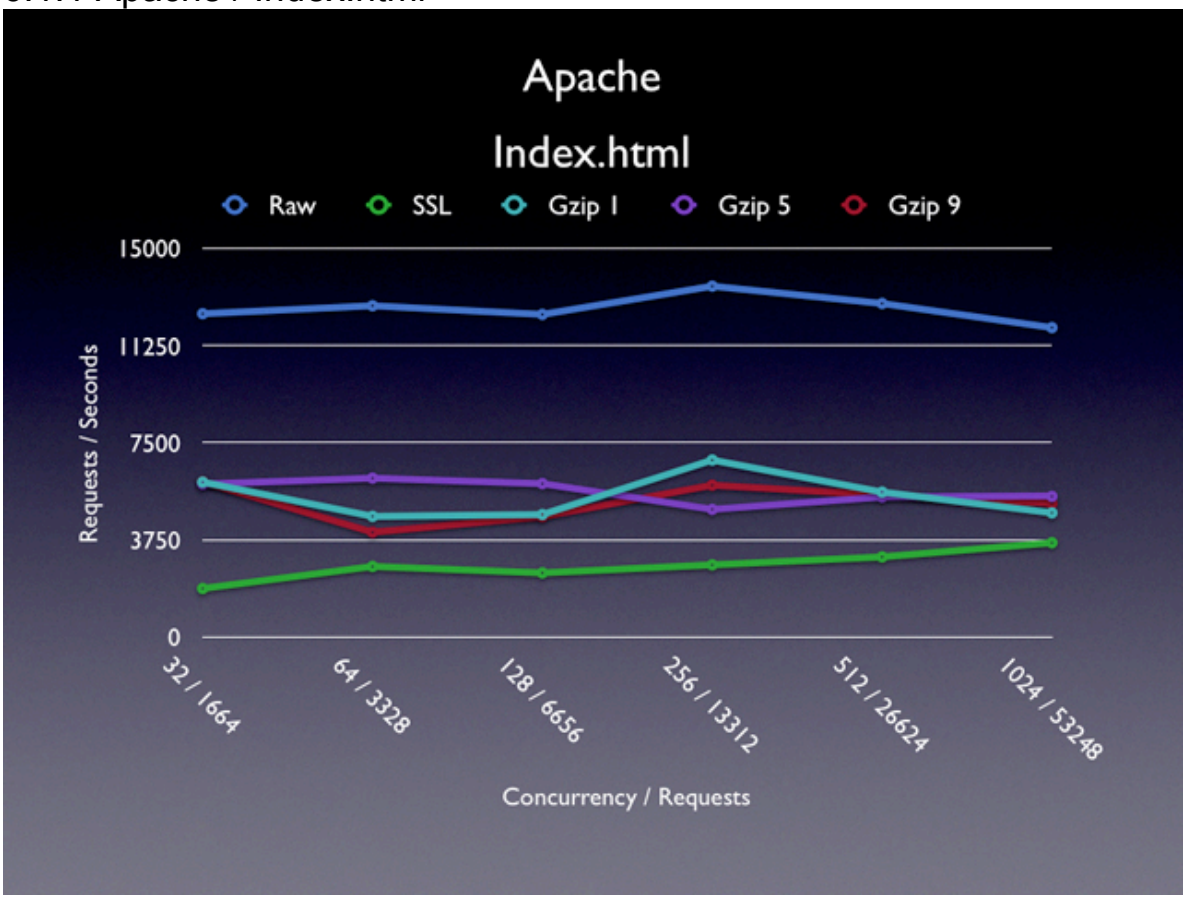
6. Resultat

I det här kapitlet analyseras och diskuteras resultaten av de utförda prestandatesterna.

6.1 Hastighet

Hastighet mäts i antal förfrågningar per sekund och presenteras här med en graf tillhörande varje webbserver.

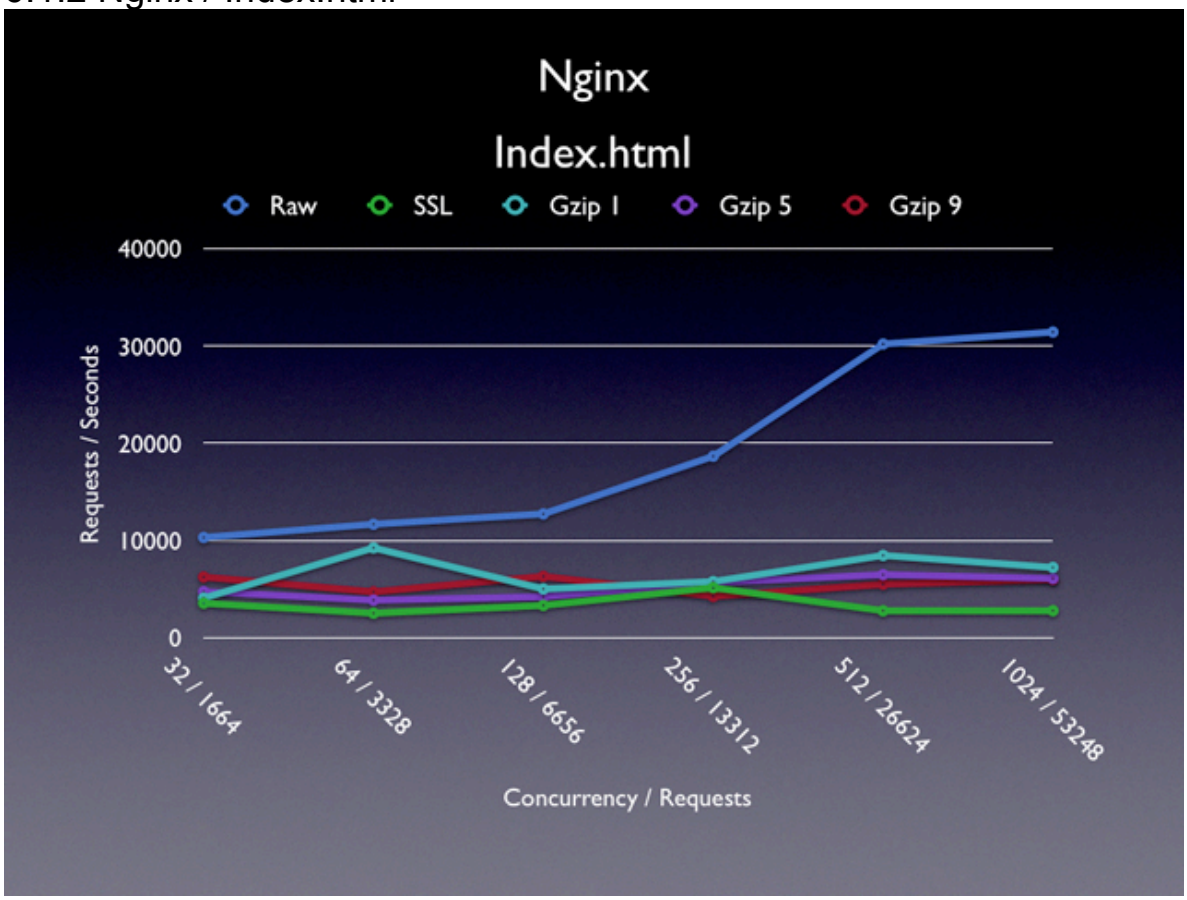
6.1.1 Apache / Index.html



Figur 6.1.1.1 Hastigheterna Apache klarar av att leverera statiskt material vid olika konfigurationer och ökande belastning.

Denna graf visar hur Apache hanterar att skicka statiskt material. Att skicka datan som den är sparad är ungefär 2x snabbare än Gzip och 6x snabbare än krypterat. Vi ser att hastigheten minskar efter 256 samtidiga anslutningar vilket indikerar på att Apache inte klarar av att hantera den ökande belastningen.

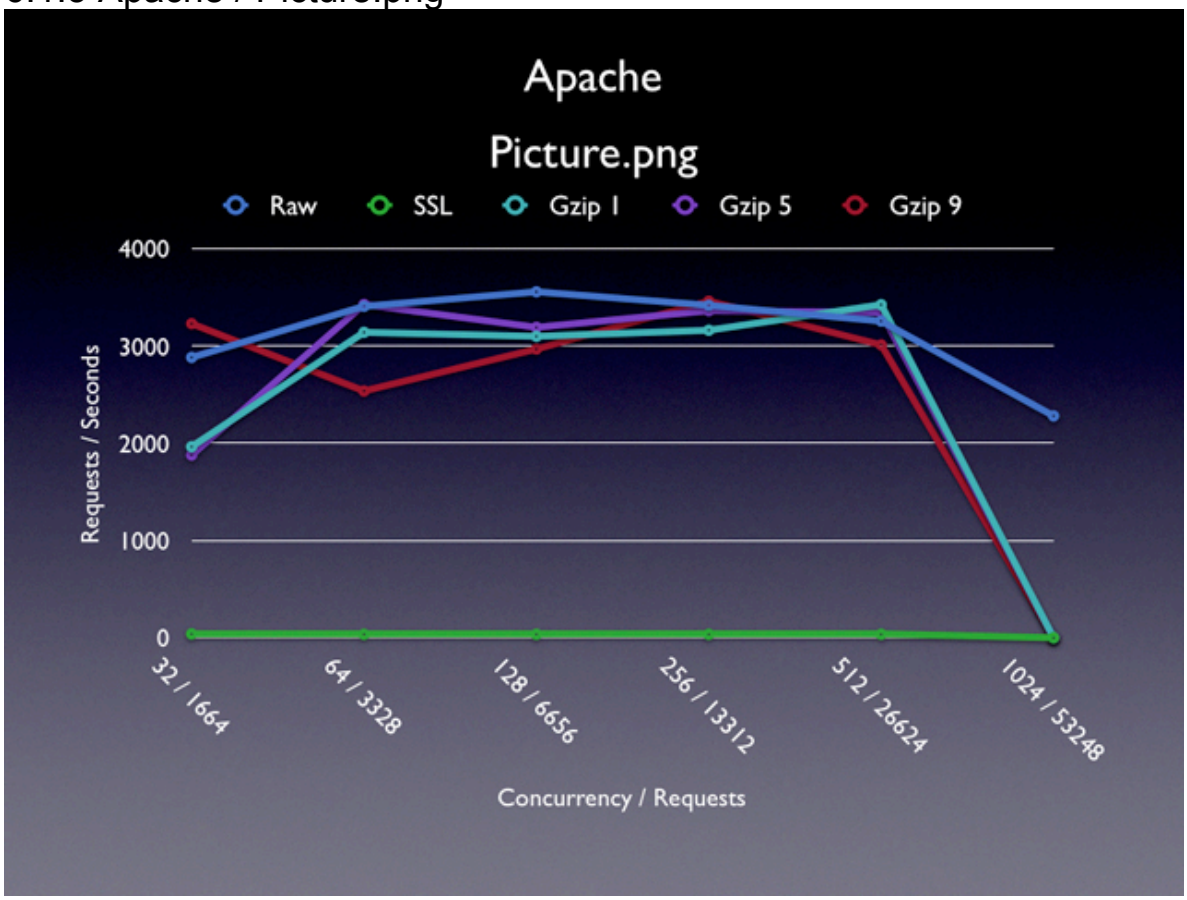
6.1.2 Nginx / Index.html



Figur 6.1.2.1 Hastigheterna Nginx klarar av leverera statiskt material vid olika konfigurationer och ökande belastning.

Här ser vi hur Nginx hanterat att skicka statiskt material. Att skicka som det är sparat är ungefär 3x snabbare än att behandla det innan. Notera här är att Nginx klarar att leverera 30000 förfrågningar/sekund när den är som hårdast belastad. Vi ser också att hastigheten inte tenderar att sjunka när belastningen ökar. Vid 512 samtidiga anslutningar börjar hastigheten att plana ut, det är möjligen att ett tak på hastigheten är nådd.

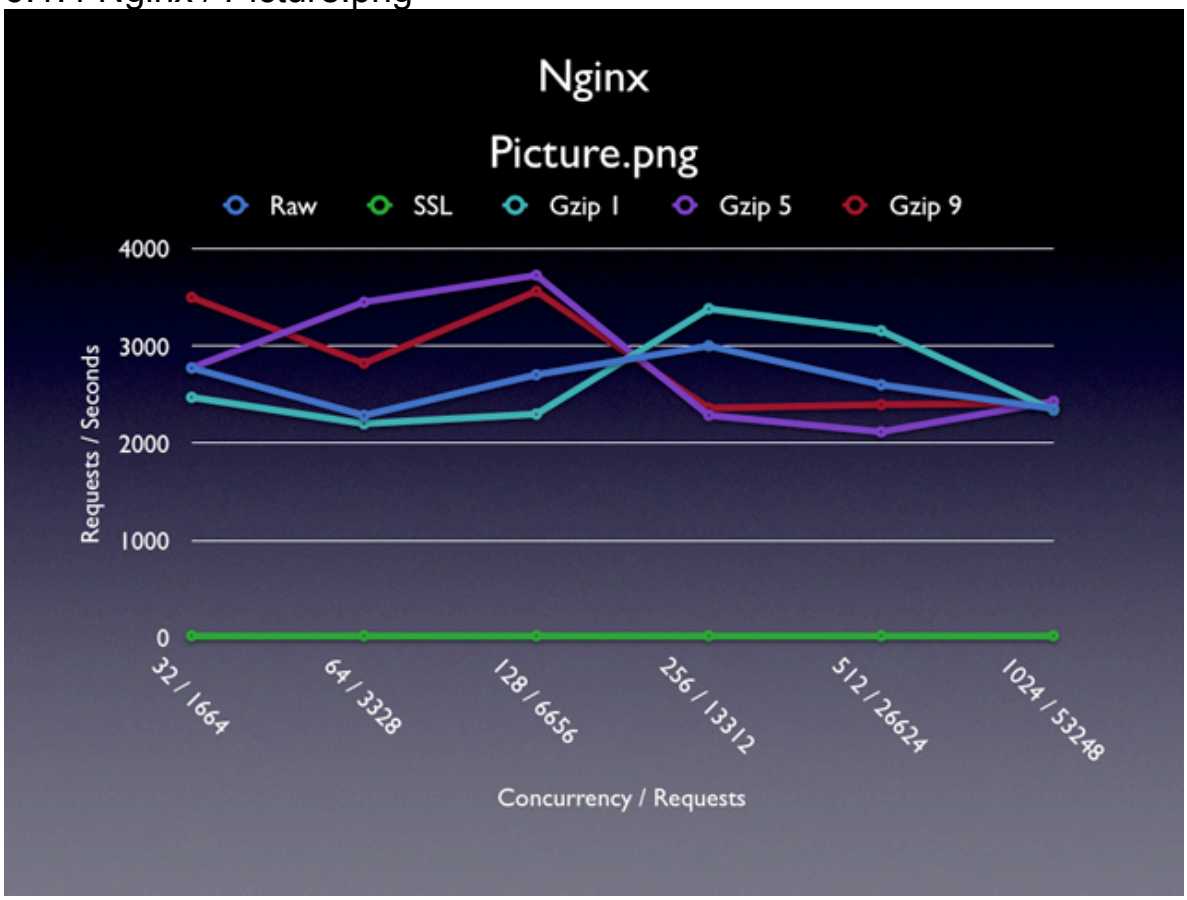
6.1.3 Apache / Picture.png



Figur 6.1.3.1 Hastigheterna Apache klarar av att leverera en bild vid olika konfigurationer och ökande belastning.

Denna graf visar hur apache klarar av att leverera bilder. Anledningen till att Raw, Gzip 1, Gzip 5 och Gzip 9 har ungefär samma hastighet beror på att bilder inte komprimeras, de är redan hårt komprimerade och det behövs därför inte. Att skicka det krypterat drar ner väldigt på hastigheten, den ligger på cirka 35 förfrågningar/sekund. Anledningen till att Gzip 1,5,9 ligger på 0 förfrågningar/sekund är på grund av att Apache har kraschat och testet slutförs med noll klara förfrågningar.

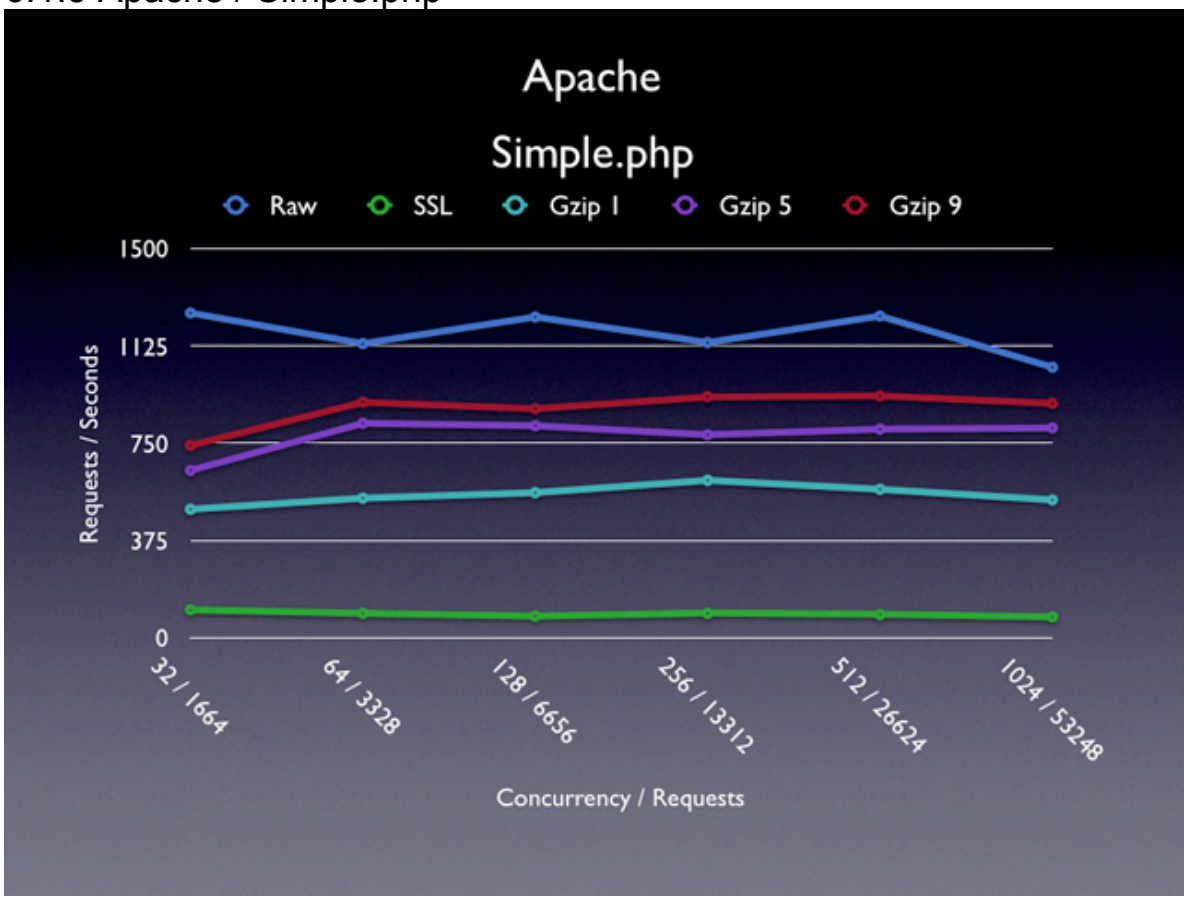
6.1.4 Nginx / Picture.png



Figur 6.1.4.1 Hastigheterna Nginx klarar av att leverera en bild vid olika konfigurationer och ökande belastning.

Vi kan se att hastigheterna ligger runt 3000 förfrågningar/sekund, liknande i Figur 6.1.3.1, anledningen till att resultaten skiljer sig med ±500 kan vara på grund av att det bästa resultatet från tre tester används istället för medelvärdet. Att skicka det krypterat minskar hastigheten avsevärt.

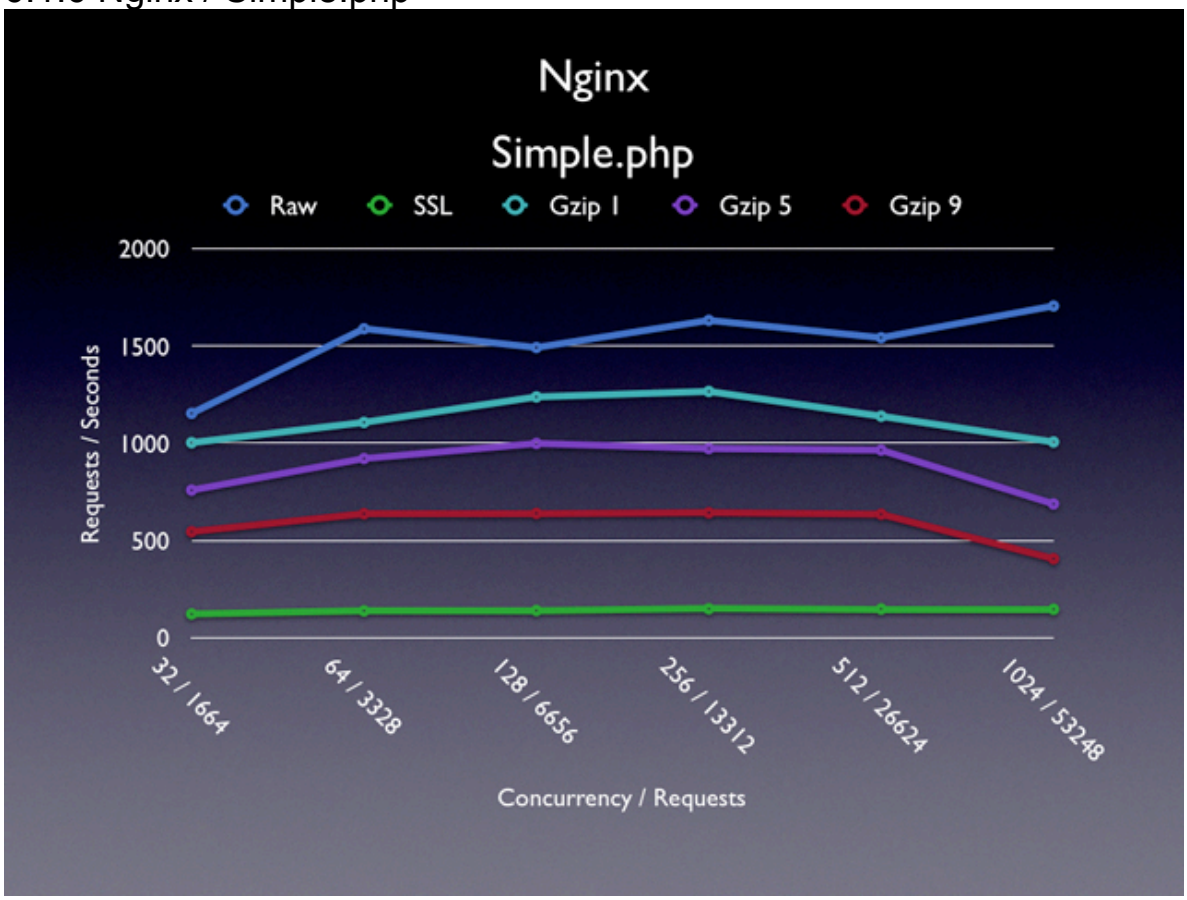
6.1.5 Apache / Simple.php



Figur 6.1.5.1 Hastigheterna Apache klarar av att leverera dynamiskt material vid olika konfigurationer och ökande belastning.

Här ser vi hur Apache och PHP-FPM klarar av att leverera dynamiskt material. Att skicka det som det är utan komprimering eller kryptering är snabbast. Gzip hastigheterna ligger relativt nära varandra och har konstant hastighet.

6.1.6 Nginx / Simple.php



Figur 6.1.6.1 Hastigheterna Nginx klarar av att leverera dynamiskt material vid olika konfigurationer och ökande belastning.

Denna graf visar hur Nginx och PHP-FPM klarar av att leverera dynamiskt material, här ser vi att Nginx ligger på ungefär samma värden som Apache gör, se Figur 6.1.5.1.

6.1.7 Slutsats

Vi kunde se att Nginx var mer än dubbelt så snabb på att leverera statiskt material jämfört med Apache, vid den högsta belastningen på 1024 samtidiga anslutningar som totalt utför 53248 förfrågningar med en hastighet på cirka 30000 förfrågningar per sekund. Vi såg även att Apache började dala vid 256 samtidiga anslutningar när Nginx fortsatte att leverera med en ökande belastning. Med kryptering och Gzip aktiverat ligger de båda på samma hastighet.

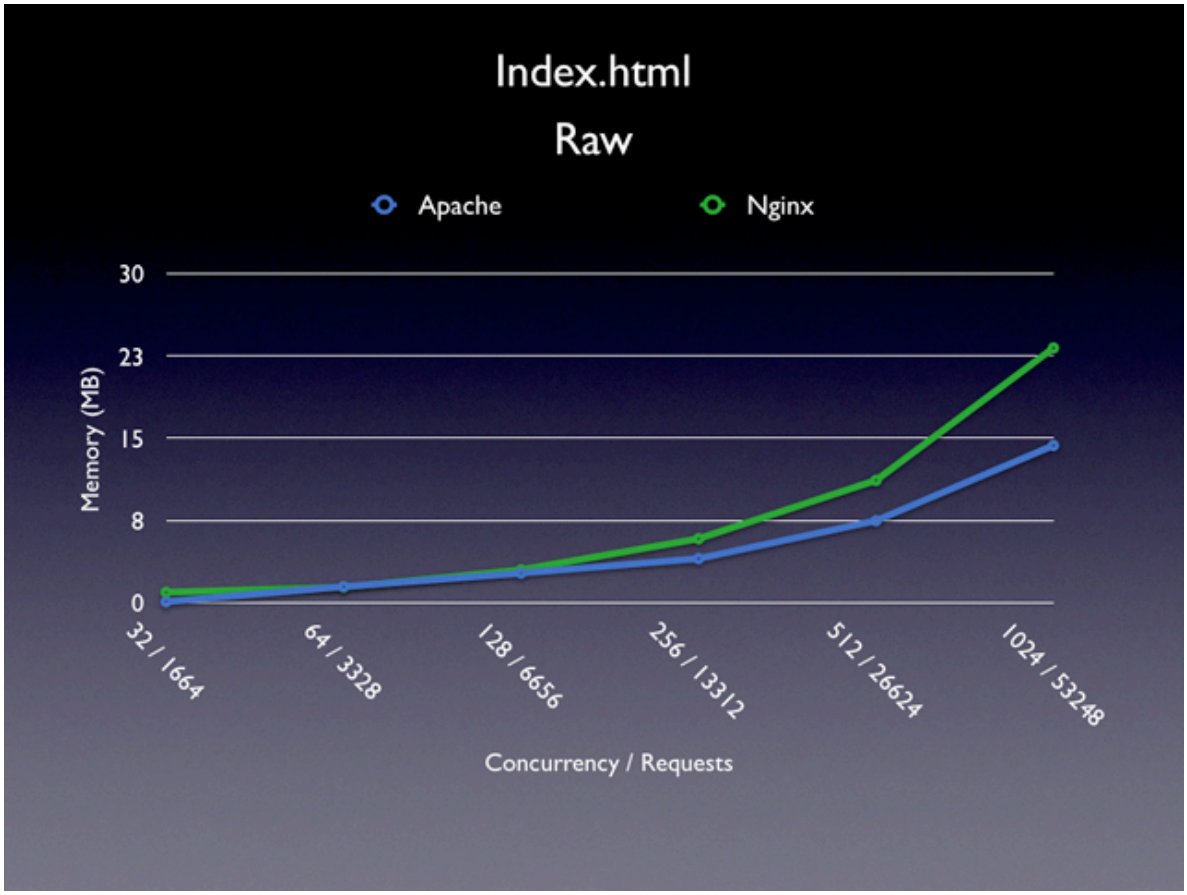
Vid förfrågan om en bild klarar både Apache och Nginx runt 3000 förfrågningar/sekund. Att ha aktiverat kryptering när bilder sänds drar ner hastigheten till cirka 35 förfrågningar/sekund.

När det gäller att leverera dynamiskt är Nginx en aning snabbare än Apache, det även när Gzip är aktiverat. Den troliga anledningen till varför det har lika värden beror antagligen på att de båda använder sig utav PHP-FPM.

6.2 Minneskonsumtion

Minneskonsumtionen mäts i hur många Megabytes (MB) webbservern upptar under prestandatestet. Graferna visar hur mycket minne som konsumeras mellan Apache och Nginx.

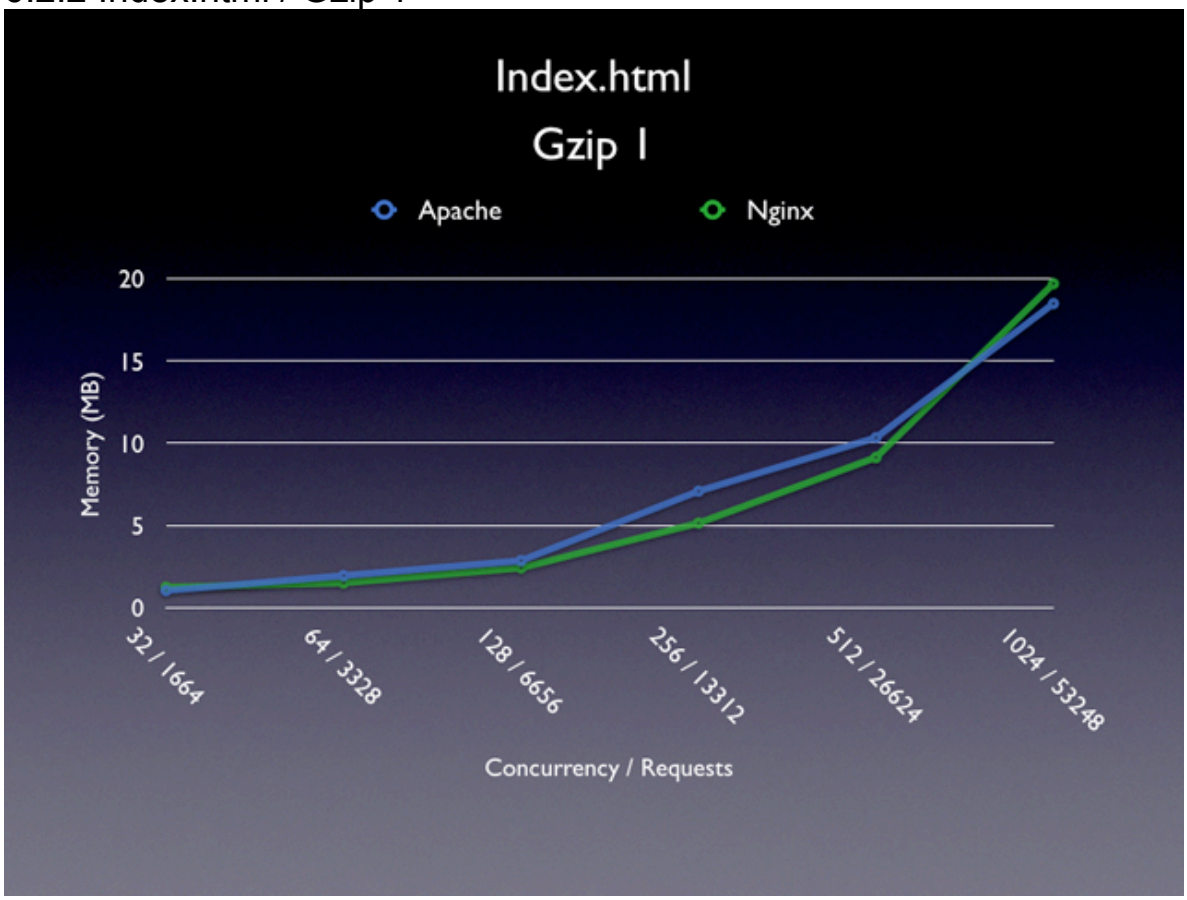
6.2.1 Index.html / Raw



Figur 6.2.1.1 Minneskonsumtionen vid leverans av statiskt material och ökande belastning.

Denna graf visar minnesförbrukningen mellan Nginx och Apache när ett statiskt material ska levereras. Vi ser här att Nginx använder ungefär 8MB mer i minne än Apache när belastningen är som störst. Minneskonsumtionen ökar exponentiell vilket antalet samtidiga anslutningar också gör.

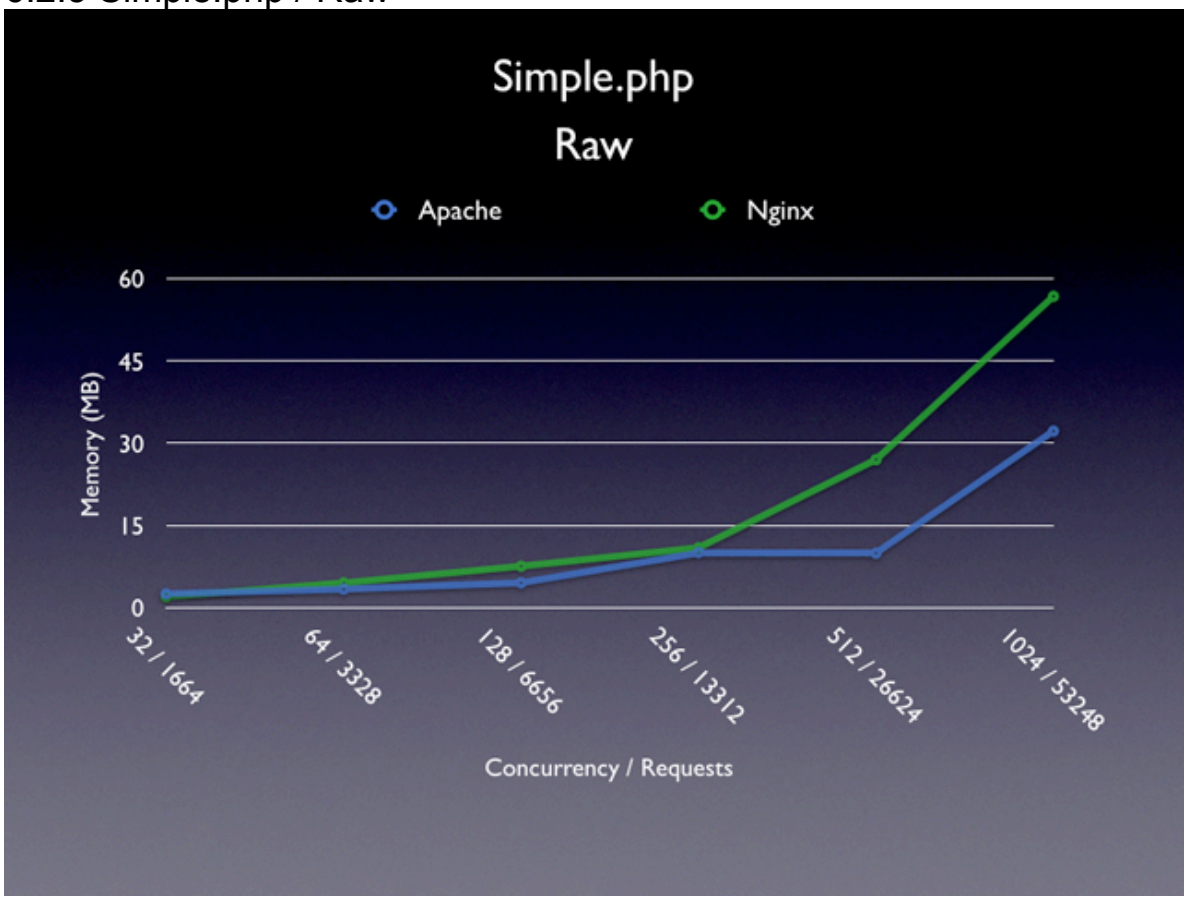
6.2.2 Index.html / Gzip 1



Figur 6.2.2.1 Minneskonsumtionen vid leverans av komprimerat statiskt material och ökande belastning.

Denna graf visar minneskonsumtionen mellan Nginx och Apache när ett statiskt material ska komprimeras med Gzip 1 innan det levereras.

6.2.3 Simple.php / Raw



Figur 6.2.3.1 Minneskonsumtionen vid leverans av dynamiskt material och ökande belastning.

Denna graf visar minneskonsumtionen mellan Nginx och Apache när dynamiskt innehåll ska levereras. Det är ungefär 30MB skillnad mellan Nginx och Apache när belastningen är som störst.

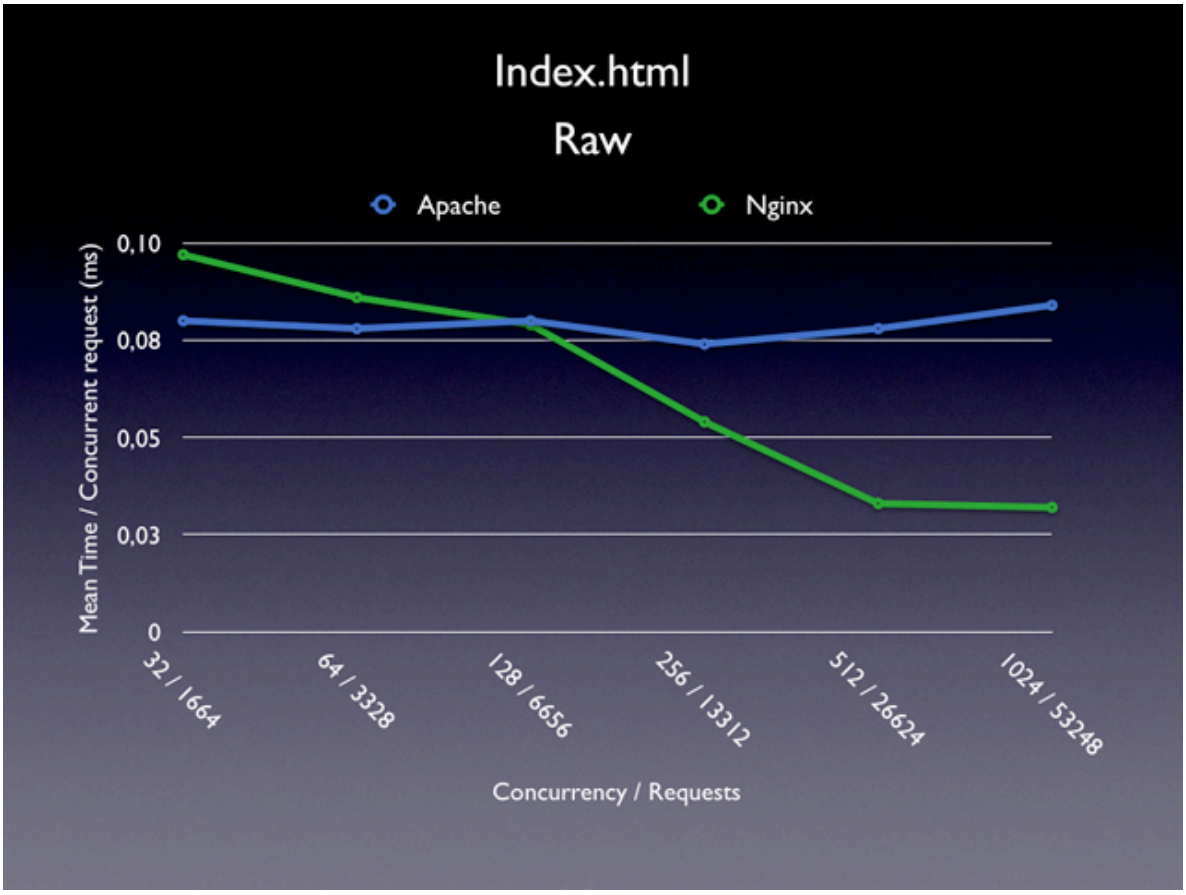
6.2.4 Slutsats

Det vi kan se bland minneskonsumtionen är att både Apache och Nginx använder i stort sätt lika mycket. Men vi ser att vid den högsta belastningen använder Nginx 60MB av minnet, detta får man se som väldigt lite med tanken på hur mycket som levereras och att idag är det inget ovanligt med servrar som har 32GB minne installerat.

6.3 Svarstid vid anslutning

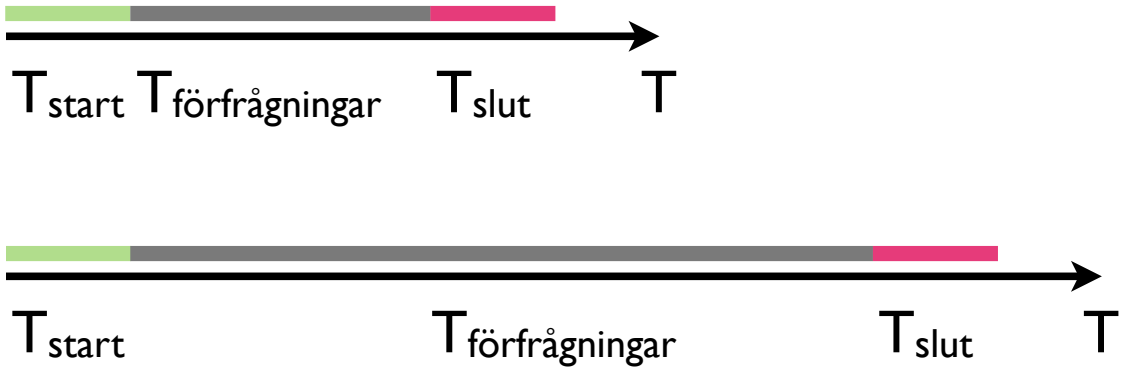
Tiden det i snitt tar för varje anslutning att få sitt svar.

6.3.1 Index.html / Raw



Figur 6.3.1.1 Genomsnittstiden för att få ett svar för varje samtidig anslutning vid leverans av statiskt material.

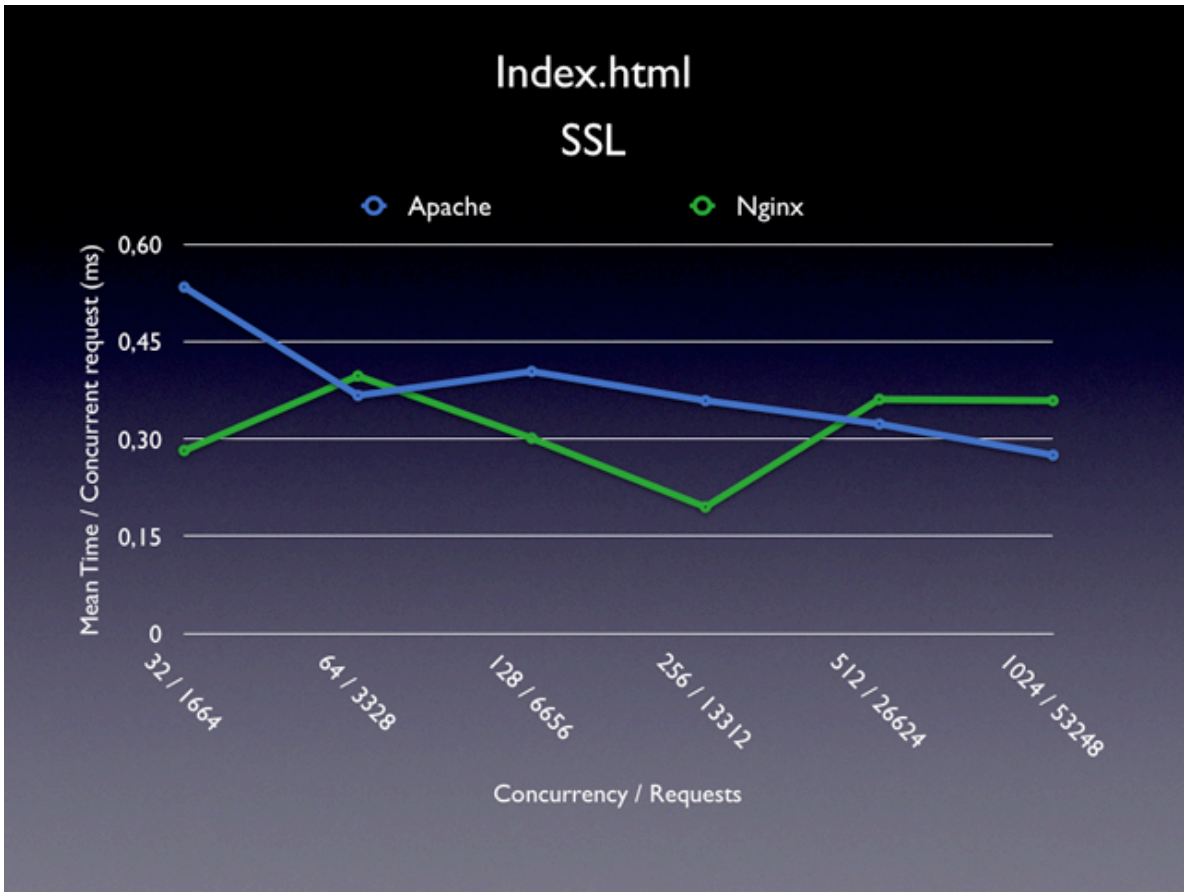
Det vi kan se här är att när belastningen ökar så klarar Nginx att leverera snabbare jämfört med Apache som ligger mer på en konstant nivå. Det som är intressant är att när belastningen är låg är Nginx långsammare än Apache men när belastningen ökar minskar svarstiderna från Nginx.



Figur 6.3.1.2 Skiss över den totala tiden Nginx kräver för att hantera ett antal förfrågningar.

En anledning till varför Nginx är långsammare vid lägre belastning kan vara tiden för att hantera förfrågningar, krävs det en uppstart (T_{Start}), tiden det tar för att leverera förfrågningarna ($T_{Förfrågningar}$) och tiden det tar för att avsluta (T_{Slut}). T_{Start} och T_{Slut} kan vara konstanta och när belastningen är låg har de en större inverkan men vid hög belastningen blir de obetydliga.

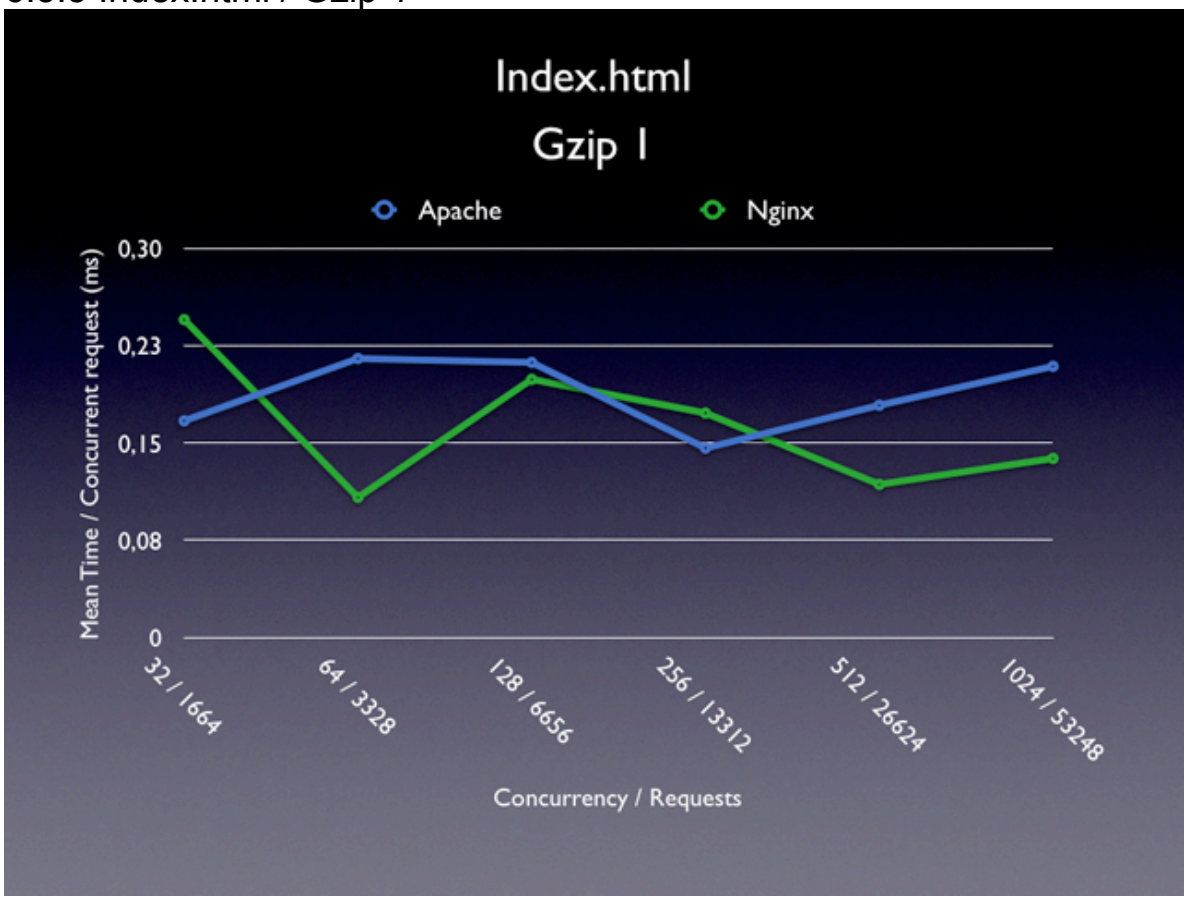
6.3.2 Index.html / SSL



Figur 6.3.2.1 Genomsnittstiden för att få ett svar för varje samtidig anslutning vid leverans av krypterat statiskt material.

Här ser vi att Apache och Nginx är rätt jämna i tiden det tar för att leverera krypterat material. De olika topparna/dalarna kan bero på slumpen och att bästa resultat används istället för ett medelvärde.

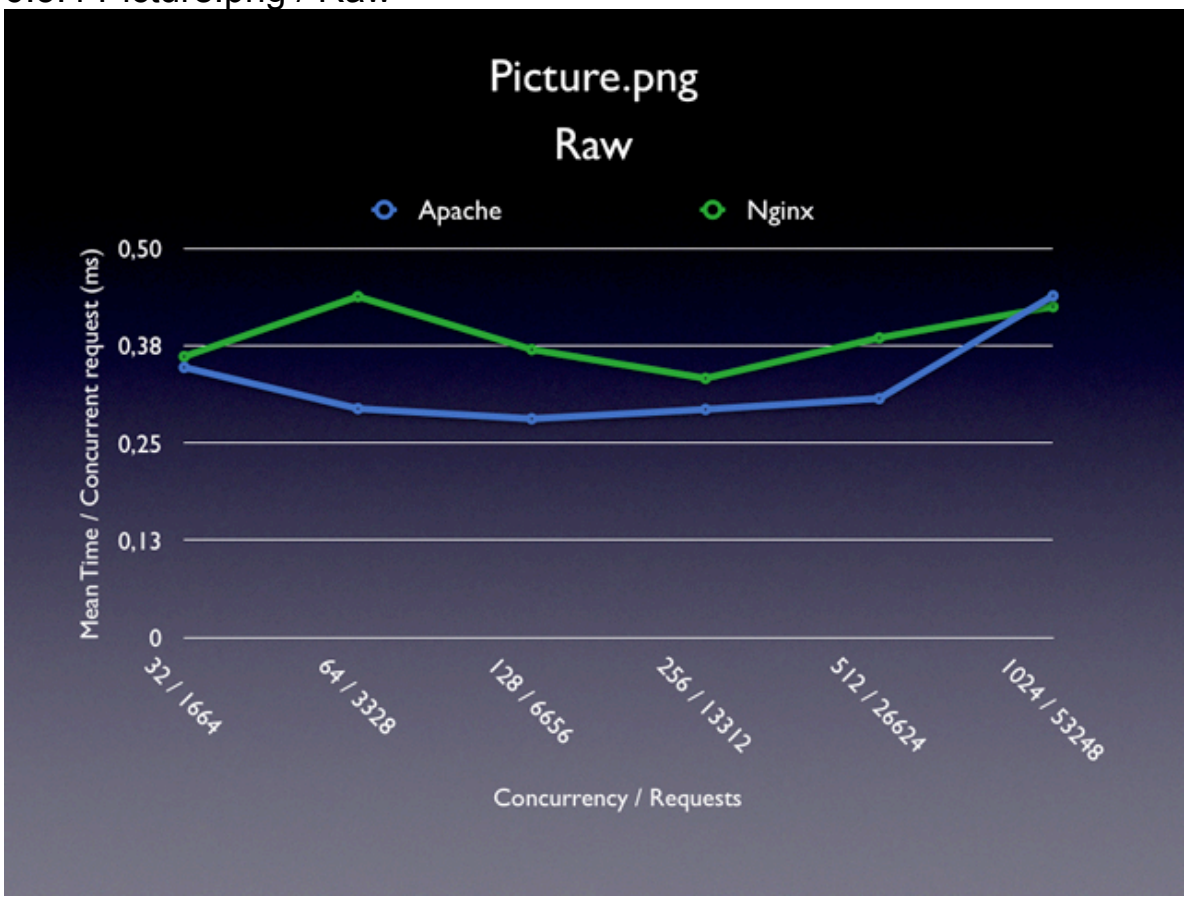
6.3.3 Index.html / Gzip 1



Figur 6.3.3.1 Genomsnittstiden för att få ett svar för varje samtidig anslutning vid leverans av komprimerat statiskt material.

Här ser vi tiden det tar för att leverera Index.html med Gzip nivå 1 komprimering. Dalen vid 64 samtidiga anslutningar för Nginx kan beror på slumpen och sen att bästa resultatet användes av de tre test som utfördes.

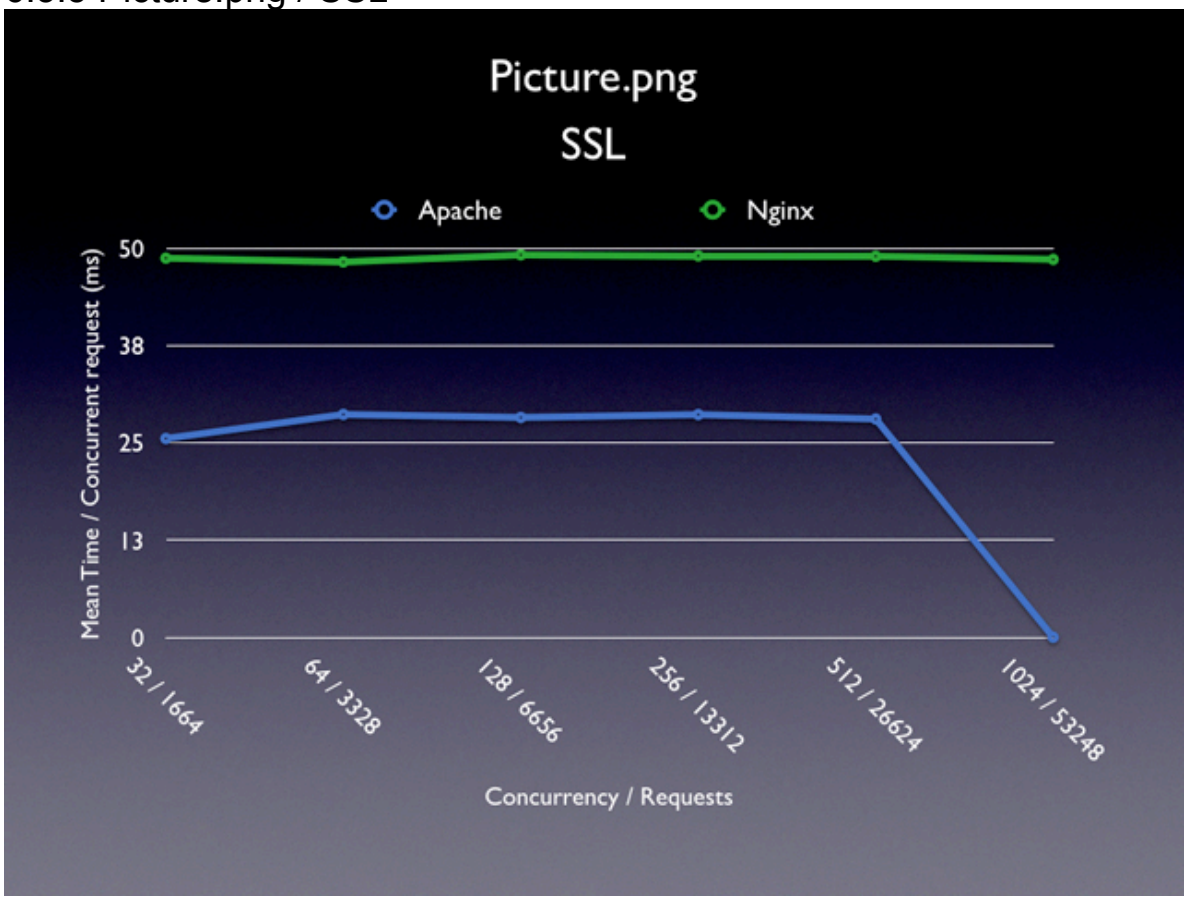
6.3.4 Picture.png / Raw



Figur 6.3.4.1 Genomsnittstiden för att få ett svar för varje samtidig anslutning vid leverans av en bild.

Här ser vi tiden det tar för att Apache och Nginx att leverera Picture.png.

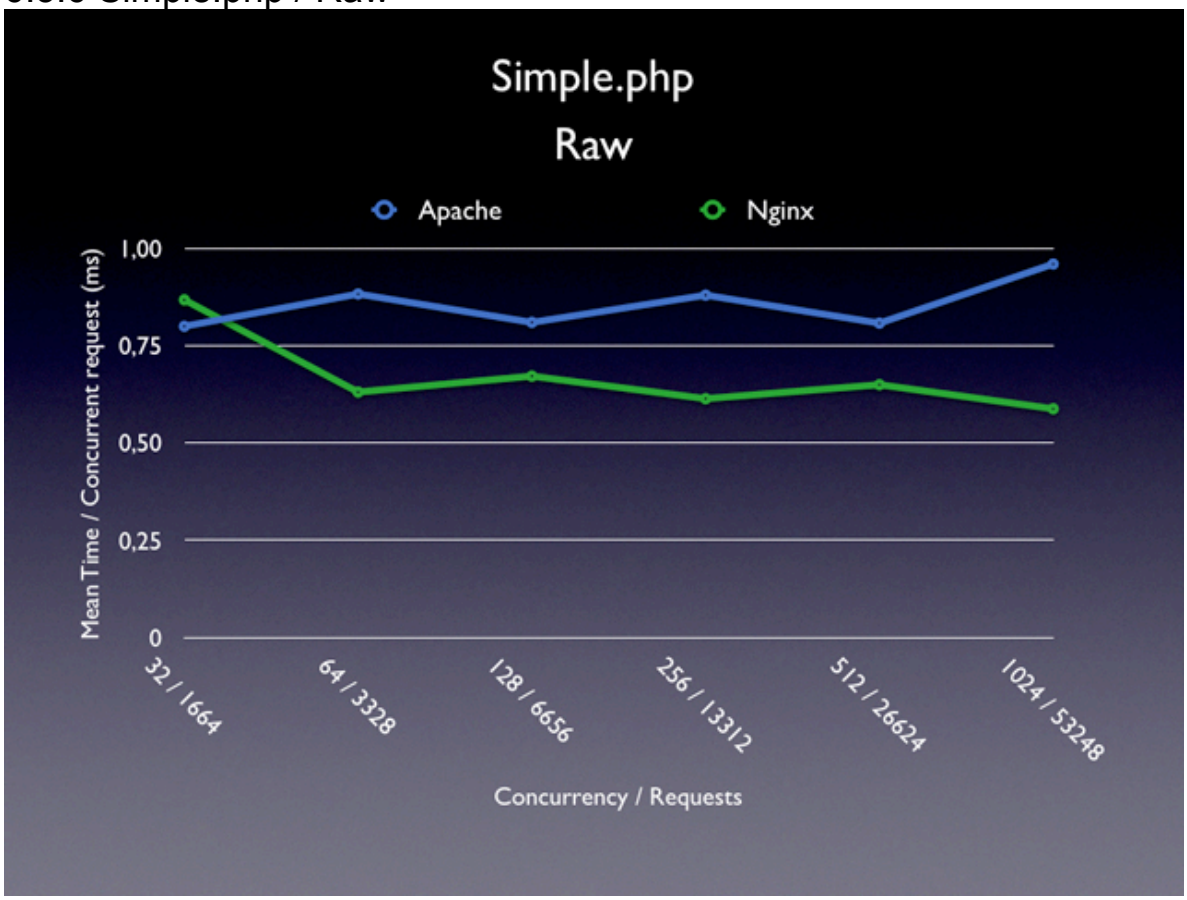
6.3.5 Picture.png / SSL



Figur 6.3.5.1 Genomsnittstiden för att få ett svar för varje samtidig anslutning vid leverans av en krypterad bild.

Här ser vi att Nginx är ungefär dubbelt så långsam än Apache, men att Apache kraschar vid 1024 samtidiga anslutningar medan Nginx klarar av att hantera det. Anledningen till att grafen visar att den ligger på 0 är på grund av att testet genomförs korrekt men att alla testen returnerat ett felmeddelande. Testet har då genomförts på korttid vilket då ger att varje anslutning fått svar väldigt snabbt, men ett fel svar.

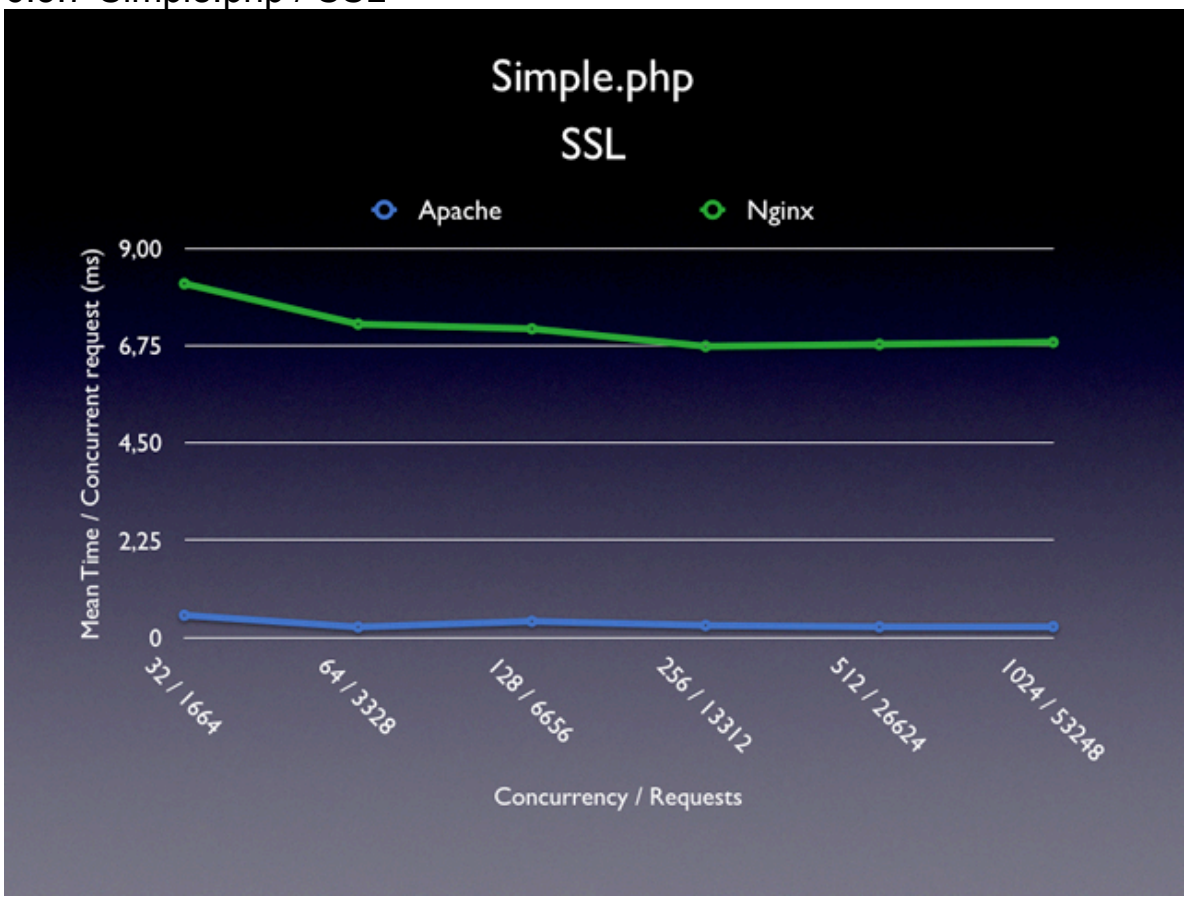
6.3.6 Simple.php / Raw



Figur 6.3.6.1 Genomsnittstiden för att få ett svar för varje samtidig anslutning vid leverans av dynamiskt material.

Här ser vi att det är jämnt mellan Apache och Nginx när det gäller att leverera dynamiskt innehåll. Det dels på att jobbet förs vidare till PHP-FPM.

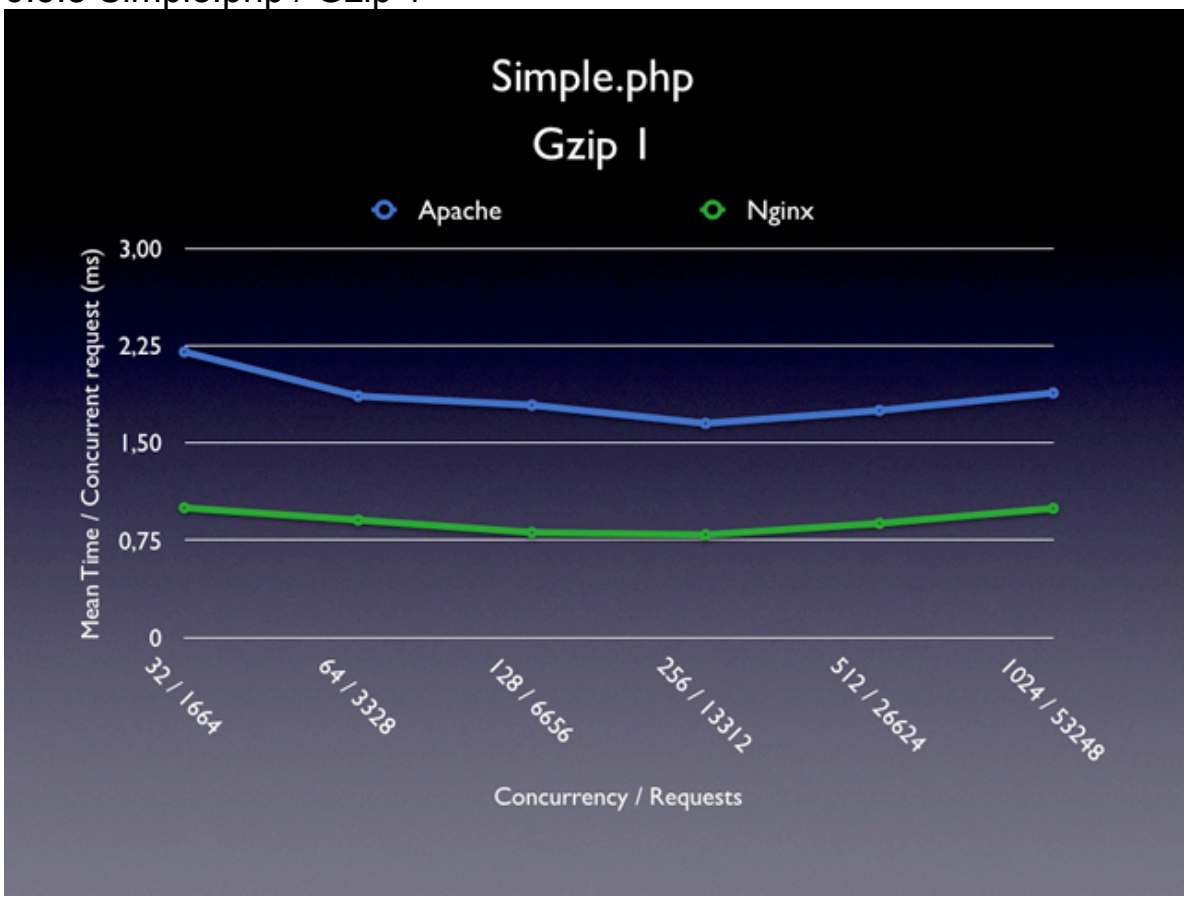
6.3.7 Simple.php / SSL



Figur 6.3.7.1 Genomsnittstiden för att få ett svar för varje samtidig anslutning vid leverans av krypterat dynamiskt material.

I denna graf ser vi att Apache är betydligt snabbare än Nginx, med en faktor nio. En anledning till det kan vara att Apache är snabbare är för Nginx har inställningen `ssl_session_cache` som standard ställd till `none` vilket gör att klienten inte kan återanvända sessionen.

6.3.8 Simple.php / Gzip 1



Figur 6.3.8 Genomsnittstiden för att få ett svar för varje samtidig anslutning vid leverans av komprimerat dynamiskt material.

Vi ser här att Nginx är snabbare än Apache, båda har liknande kurva och det kan bero på hur Gzip hanterar komprimeringen medan Nginx hanterare förfrågningarna snabbare än Apache.

6.3.9 Slutsats

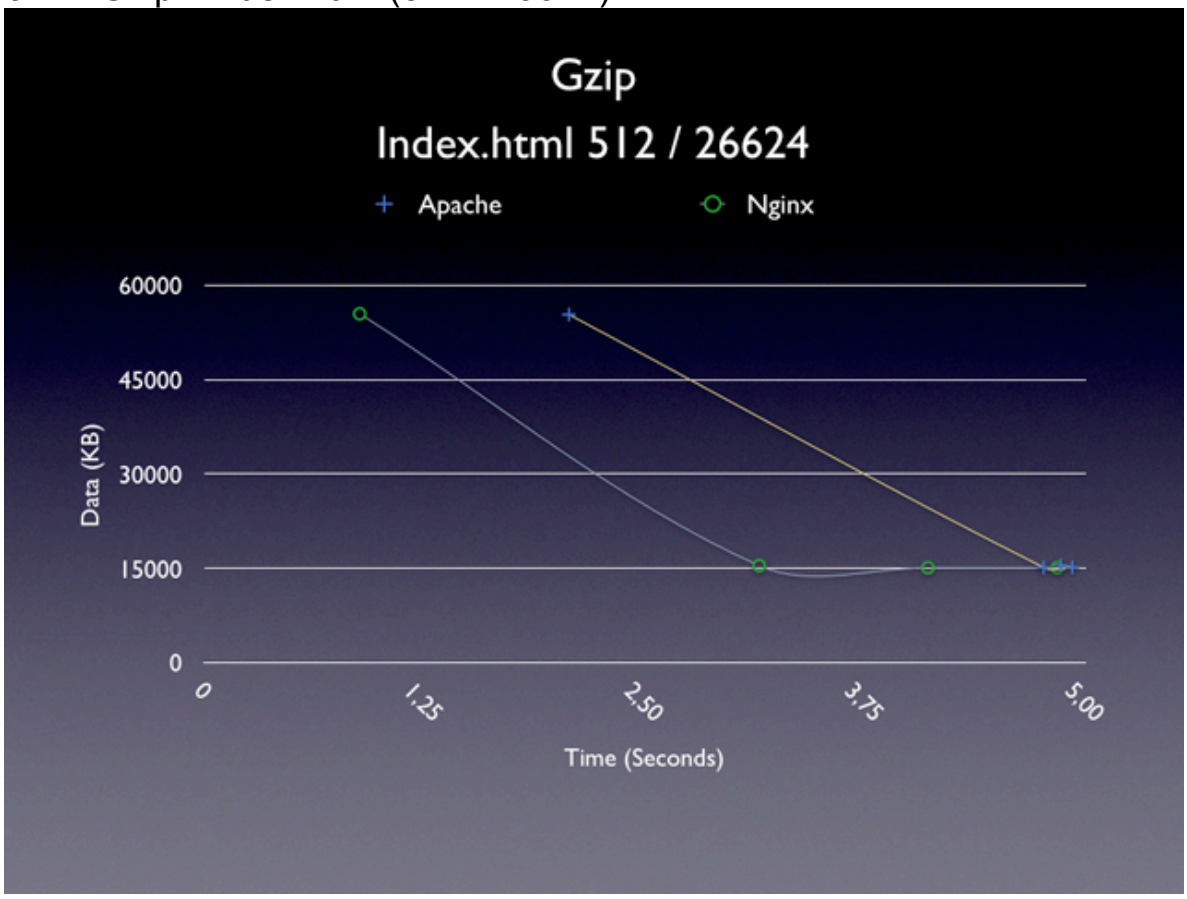
Apache och Nginx levererar båda väldigt snabbt vid statiskt material, Apache levererar konstant runt 0,08ms medan Nginx har som lägst 0.04ms när den är som mest belastad. När SSL aktiveras förhöjs leveranstiden med faktorn fyra och med Gzip 1 med faktorn 2.

Att leverera en bild på cirka 535KB gör både Apache och Nginx på samma tid, cirka 0.38ms i snitt. Däremot när kryptering aktiveras blir det stora skillnader mellan Apache och Nginx. Apache ligger på cirka 25ms per förfrågning men blir ett avbrott vid 1024 samtidiga anslutningar. Nginx ligger konstant på 50ms per förfrågning.

6.4 Gzip

Graf nedan visar förhållandet mellan storleken på datan och tiden det tar att leverera.

6.4.1 Gzip / Index.html (512 / 26624)



Figur 6.4.1.1 Visar fördelningen mellan storleken på datan och tiden det tar att komprimera.

De två översta punkterna representerar när det skickas utan komprimering.

6.4.2 Slutsats

Vi ser att datan minskade med 72 % vid enbart Gzip 1 efter det sker ingen mer komprimering oavsett ifall vi ökar komprimeringen. Det beror på att fil är relativt liten och att redan vid Gzip 1 är den så komprimerad den kan bli. Nginx är aningen snabbare än Apache när det gäller Gzip 1 och Gzip 5.

7. Slutsatser

De intressanta webbservrarna valdes genom att titta på statistik över vilka webbservrar som används mest idag och ifrån det göra ett urval av vilka som ska ingå i testet, se kapitel 3.4.

För att prestandatesta webbservrarna användes ApacheBench och ett protokoll för hur testen ska utföras skapades. Tydliga uppgifter och konfigurationer konstruerades för att uppnå varierande tester, se kapitel 5.

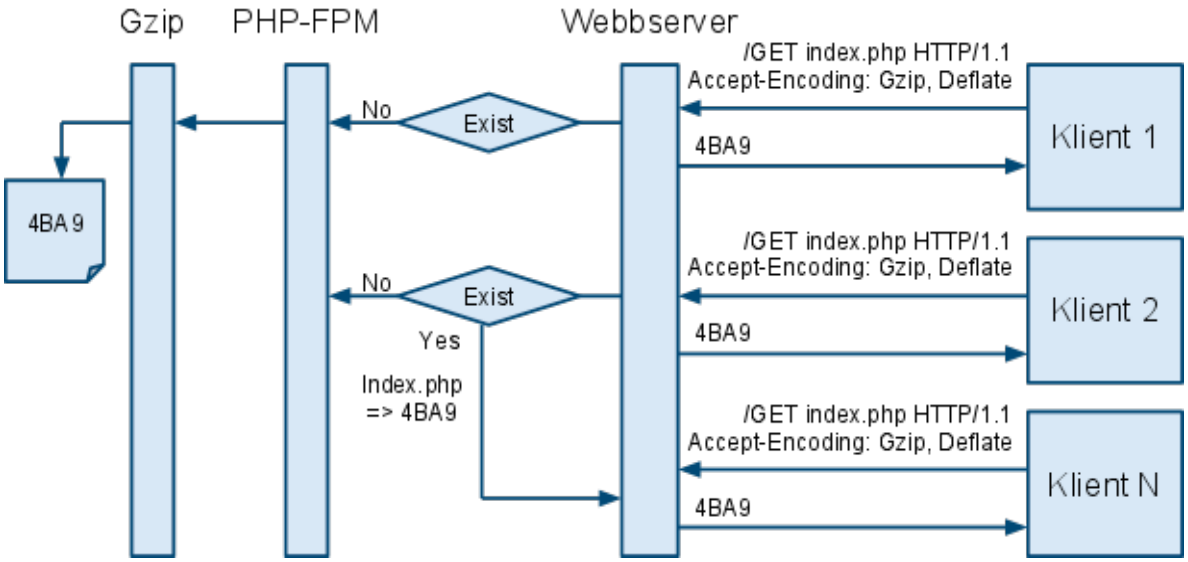
Efter att ha prestandatestat både Apache med Worker-modulen och Nginx kan vi se att båda två klarar av att hantera hög belastning. Det som gör att Nginx sticker ut är hur den hanterar att leverera statiskt material vid hög belastning, vid över 30000 förfrågningar per sekund klarar den av att leverera. Om vi räknar om att vi ligger på en sådan belastning konstant under 24 timmar blir det över 2,5 Miljarder sidvisningar, då får man en uppfattning över den verkliga mängden.

När det gäller förfrågningar utav bilder eller dynamiskt innehåll håller de båda webbservrarna ungefär samma hastighet. Varken Apache eller Nginx har något med hastigheten för det dynamiska innehållet att göra, det är endast att förfrågningar skickas vidare till PHP-FPM processen och sedan returnerar tillbaka till webbservern. Att generera dynamiskt innehåll är en tiondel långsammare för Apache och för Nginx är det en tjugondel så långsamt än att leverera statiskt material.

Att Gzip:a materialet drar ner på hastigheten men det är något som uppskattas av den med sämre internetanslutning, ex mobiltelefonen, då de behöver ladda ner mindre data och sidan upplevs som snabbare.

Att skicka datan krypterad var det som påverkade prestandan mest och det var inte förväntat att det skulle göra, en av anledningarna kan vara att 2048-bitars kryptering som används istället för vad som brukar användas, nämligen 1024-bitars kryptering. Anledningen till att 2048-bitars kryptering valdes var dels för att Ben Smeets (Ansvarig för EDA625 Säkerhet) sa att idag är 1024-bitar för svagt samt att exempelvis handelsbanken använder 2048-bitar för kryptering, medan Facebook använder 1024-bitar.

Det viktigaste med att klara av en högre belastning är att leverera statiskt material till klienten. Det betyder att allt som genereras dynamiskt för första gången sen måste sparas till en fil och att även Gzip:a innehållet och spara det. Har man det i tanken kan en dynamisk sida klara av högre belastning.



Figur 7.1 Skiss över hur komprimerat dynamiskt innehåll kan levereras snabbare till flera klienter.

Här ser vi en skiss på hur cache:ning av dynamiskt innehåll som komprimerats kan levereras till klient. Första gången klienten frågar efter `Index.php` så existerar ingen statisk fil, PHP-FPM generar denna och skickar tillbaka till webbservern som då Gzip:ar filen och sparar den på disken med namnet `4BA9` och sänder den till klienten. Nästa klient som frågar efter `Index.php` levereras då `4BA9` som går betydligt snabbare.

Eftersom Nginx är snabbare på att leverera statiskt material, vilket är nyckeln till att kunna hantera hög belastning tycker jag det är det rätta valet.

8. Vidareutveckling

Jag tror absolut att det finns fler sätt att optimera på operativsystemnivå, har man bara jobbat med Ubuntu Server i ungefär tre månader så finns säkert saker man har missat. Optimering på databaslagret är något som också bör göras, det kan vara att ha ett cache:nings lager framför databasen, för att inte repetera samma databasfrågor.

I prestandatestet skulle man kunna utöka med större statiska filer för att se en bättre skillnad på de olika nivåerna av Gzip och öka antalet samtidiga anslutningar Att göra tre tester och att använda det bästa, ger resultat där det enskilt bästa resultatet kan ha sket av en slump. Det hade varit bättre att ta medel av alla tester.

Det viktigaste är att identifiera var störst belastning ligger och sedan optimera det för att klara av en ökande belastning.

9. Terminologi

RDMS	Relational database management system
SQL	Structured Query Language
NoSQL	Databaser som inter bygger på traditionella RDMS
CMS	Content management system

10. Referenser

[1] HTTP
http://www.w3.org/Protocols/rfc2616/rfc2616.html. 2011-05-04

[2] Webbserver
http://www.webdevelopersnotes.com/basics/what_is_web_server.php. 2011-05-04

[3] Apache
http://httpd.apache.org/ABOUT_APACHE.html. 2011-04-12

[4] Nginx
http://nginx.org/en/. 2011-04-12

[5] Keep-Alive
http://en.wikipedia.org/wiki/HTTP_persistent_connection. 2011-05-01

[6] Gzip
http://www.freebsd.org/cgi/man.cgi?gzip. 2011-03-28

[7] SSL/TLS
http://tools.ietf.org/html/rfc5246. 2011-03-28

[8] ApacheBench
http://httpd.apache.org/docs/2.2/programs/ab.html. 2011-04-01

[9] JavaScript
http://en.wikipedia.org/wiki/JavaScript. 2011-04-01

[10] MongoDB
http://www.mongodb.org/. 2011-04-02

[11] Node.JS
http://nodejs.org/#about. 2011-04-02

[12] PHP
http://www.php.net/. 2011-04-05

[13] Wordpress
http://wordpress.org/about/. 2011-04-05

[14] Waterstone rapport
http://www.waterandstone.com/sites/default/files/2010%20OSCMS%20Report.pdf. 2011-05-07

[15] Ubuntu
http://www.ubuntu.com/. 2011-05-28

[16] Netcraft statistik
http://news.netcraft.com/archives/2011/05/02/may-2011-web-server-survey.html. 2011-05-15

[17] Alexa Topsites Sweden
http://www.alexa.com/topsites/countries/SE. 2011-04-17

Appendix A Index.html

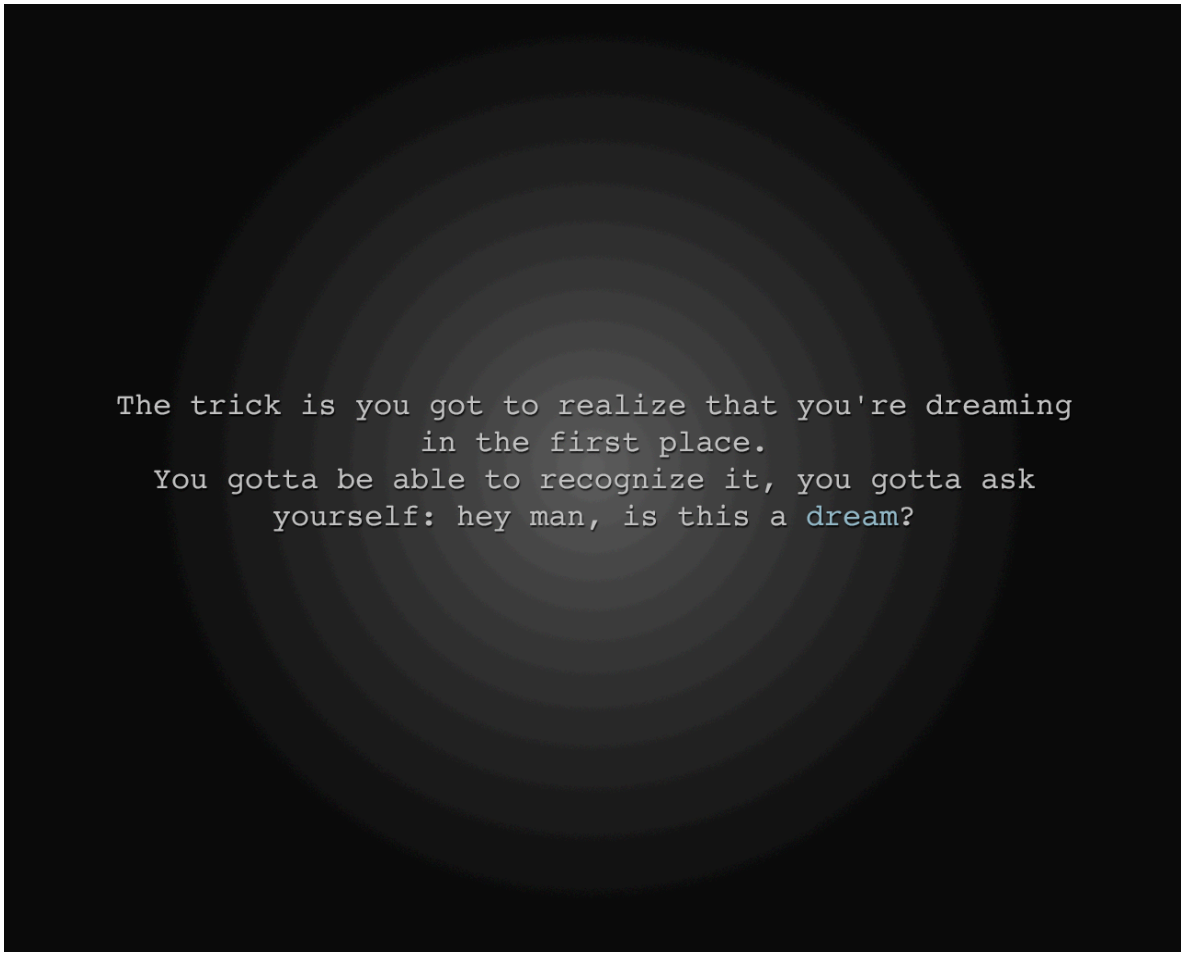
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Static page</title>
  <meta name="description" content="" />
  <meta name="keywords" content="" />
  <meta name="robots" content="" />
</head>
<body>
  <h1>Welcome to the static page</h1>
  <nav>
    <ul>
      <li><a href="#">Lorem</a></li>
      <li><a href="#">Ipsum</a></li>
      <li><a href="#">Lorem</a></li>
      <li><a href="#">Ipsum</a></li>
      <li><a href="#">Lorem</a></li>
      <li><a href="#">Ipsum</a></li>
      <li><a href="#">Lorem</a></li>
      <li><a href="#">Ipsum</a></li>
    </ul>
  </nav>
  <div id="wrapper">
    <h1>Lorem Ipsum</h1>
    <div id="sidebar">Nice with sidebar</div>
    <div id="container">
    <p>Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute
irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur
sint occaecat cupidatat non proident, sunt in culpa
qui officia deserunt mollit anim id est laborum.</p>

<p>Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim
```

```
veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute
irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur
sint occaecat cupidatat non proident, sunt in culpa
qui officia deserunt mollit anim id est laborum.</p>

<p>Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute
irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur
sint occaecat cupidatat non proident, sunt in culpa
qui officia deserunt mollit anim id est laborum.</p>
    </div>
    <footer>
      <p>This is FOOTER!!</p>
    </footer>
  </div>
</body>
</html>
```

Appendix B Picture.png



Appendix C Simple.php

```
<?php

for($i = 0; $i < 100; $i++) {
  echo $i+$i;
  echo '<br />';
}

echo "<h1>Testing PHP</h1>";

echo phpinfo();

?>
```

Appendix D MongoDB dokument

```
{
"config"  : {
  "testId"      : "2.3.6"
  "OS"          : "OS Information",
  “concurrency” : 512,
  “requests”    : 12234
},
"results"    : [{
  "iteration” : 1,
  "uptime"    : "Uptime of the system",
  "mem"       : {
    "total"   : 1024,
    “free”    : 800,
    "lowestPeak" : 330
  },
  “timetaken” : 11,
  “completeRequest” : 333,
  “faileRequests” : 1,
  “requestPSecond” : 2323,
  “htmlTransferred” : 932435,
  “totalTransferred” : 21454,
  “ab” : “Raw ApacheBench output”,
  “abCsv” “Apachebench CSV output”
}]
}
```

Appendix E clientBenchmark.js

```
//Needed libraries
var util = require('util'),
    exec = require('child_process').exec;

//Global varibals
var params = {},
    sshCmd = "";

//Filter out arguments
process.argv.splice(2).forEach(function (val, index,
array) {
  tmp = /(\w+)=(\S+)/.exec(val);
  params[tmp[1]] = tmp[2];
});

function web(w) {
  if(w < 2) {
    var o = 0;
    obj(w, o);
  }
}

function obj(w, o) {
  if(o < 4) {
    var c = 0;
    cfg(w, o, c);
  } else {
    web(w + 1);
  }
}

function cfg(w, o, c) {
  if(c < 5) {
    powx(w, o, c, 5);
  } else {
    obj(w, o + 1);
  }
}

function powx(w, o, c, x) {
  if(x <= 10) {
```

```
    itr(w, o, c, x, 1);
  } else {
    cfg(w, o, c + 1);
  }
}

function itr(w, o, cfg, x, i) {
  if(i <= 3) {
    var testId = w + '.' + o + '.' + cfg;
    var c = Math.pow(2, x),
    n = c * 52;
    sshCmd = 'ssh -i ~/.ssh/id_rsa_exjobb ' +
params.ssh + ' "node serverBench.js testId=' +
testId + ' itr=' + i + ' c=' + c + ' n=' + n + '"';

    exec(sshCmd, function(error, stdout, stderr) {
    //Wait for the reboot
       setTimeout(function() {
         itr(w, o, cfg, x, i + 1);
       }, 40000);
    });
  } else {
    powx(w, o, cfg, x + 1);
  }
}

web(0);
```

Appendix F serverBench.js

```
//Needed libraries
var util = require('util'),
      fs = require('fs'),
      os = require('os'),
      exec = require('child_process').exec,
      child,
      mongodb = require('/root/christkv-node-
mongodb-native-b308c59/lib/mongodb');

//Global varibals
var params = {},
    testInfo = {},
    document = {},
    results = {};

//"Constants"
var CONFIGS = {
  service : ['apache2', 'Nginx'],
  testObj : ['index.html', 'pic.png', 'simple.php',
'wordpress/'],
  conf    : ['no', 'ssl', 'gzip_1', 'gzip_5',
'gzip_9']
}

//Extract arguments
process.argv.splice(2).forEach(function (val, index,
array) {
  tmp = /(\w+)=(\S+)/.exec(val);
  params[tmp[1]] = tmp[2];
});

//Filter out the test ID
params.testId =
params.testId.match(/(\d+)\.(\d+)\.(\d+)/);

//Helper functions
var startWebservice = function(serv, callback) {
  serv = serv == 'Nginx' ? 'Nginx' : 'apache2';
  exec('/etc/init.d/' + serv + ' start',
function(error, stdout, stderr) {
  if(callback && typeof(callback) === "function") {
```

```
  callback();
  }
  });
};

//Watch memory
//Return a intervalId object
function memWatcher(cb) {
  return setInterval( function(exec) {
    var cmd = "cat /proc/meminfo | fgrep MemFree |
sed s/[^0-9]//g >> /var/www/results/" +
params.testId[0]  + "-" + params.itr + "-" +
params.c + "-" + params.n + ".mem";
    exec(cmd, function(error, stdout, stderr) {
    });
  }, 100, exec);
}

function getMemoryPeak(cb) {
  exec('sort -n < /var/www/results/' +
params.testId[0] + '-' + params.itr + '-' + params.c
+ '-' + params.n + '.mem | head -1', function(error,
stdout, stderr){
    testInfo['memPeak'] = stdout;
    if(cb && typeof(cb) === 'function') cb();
  });
}

//This functions will return the properly ab
settings, and activate necessary modules for
webserver
//No -> disableGzip
var no = function(cb) {
  cb('');
},

ssl = function(cb) {
  cb('-Z DHE-RSA-AES256-SHA -f TLS1 ');
},

gzip_1 = function(cb) {
  gzip(1, cb);
},
```

```
gzip_5 = function(cb) {
  gzip(5, cb);
},

gzip_9 = function(cb) {
  gzip(9, cb);
},

gzip = function(lvl, cb) {
  //Activate gzip and set correct compression
  var abGzip = '-H "Accept-Encoding:gzip,deflate" ';

  if( CONFIGS.service[params.testId[1]] === 'Nginx'
) {
    //Switch configs
    fs.renameSync('/etc/Nginx/Nginx.' + lvl +
'.conf', '/etc/Nginx/Nginx.conf');

    //Callback
    if(cb && typeof(cb) === "function") {
      cb(abGzip);
    }
  } else {
     //Switch configs
    fs.rename('/etc/apache2/mods-
available/deflate.conf.' + lvl, '/etc/apache2/mods-
available/deflate.conf', function() {
       exec('a2enmod deflate', function(error,
stdout, stderr) {
         //Callback
         if(cb && typeof(cb) === "function")
cb(abGzip);
       });
    });
  }
},

no_disable = function(cb) {
 if(cb && typeof(cb) === "function") cb();
},

ssl_disable = function(cb) {
  if(cb && typeof(cb) === "function") cb();
},
```

```
gzip_1_disable = function(cb) {
  disableGzip(1, cb);
},

gzip_5_disable = function(cb) {
  disableGzip(5, cb);
},

gzip_9_disable = function(cb) {
  disableGzip(9, cb);
},

disableGzip = function(lvl, cb) {
  if( CONFIGS.service[params.testId[1]] === 'Nginx'
) {
    fs.rename('/etc/Nginx/Nginx.conf',
'/etc/Nginx/Nginx.' + lvl + '.conf', function() {
      exec('cp /etc/Nginx/Nginx.conf.bak
/etc/Nginx/Nginx.conf', function(error, stdout,
stderr) {
         //Callback
         if(cb && typeof(cb) === "function") {
           cb();
         }
      });
    });
  } else {
    fs.rename('/etc/apache2/mods-
available/deflate.conf', '/etc/apache2/mods-
available/deflate.conf.' + lvl, function() {

      exec('cp /etc/apache2/mods-
available/deflate.conf.bak /etc/apache2/mods-
available/deflate.conf', function(error, stdout,
stderr) {
        exec('a2dismod deflate', function(error,
stdout, stderr) {
          //Callback
          if(cb && typeof(cb) === "function") {
            cb();
          }
        });
      });
```

```
    });
  }
},

apacheBench = function(abSubStr, cb) {
  var ab = 'ab ';

  //All request are done with keep-alive header,
HTTP 1.1 standard
  ab += '-k ';

  //Concurrent connection
  ab += '-c ' + params.c + ' ';

  //Number of requests
  ab += '-n ' + params.n + ' ';

  //extra header, proberly ssl
  ab += abSubStr

  //Save a csv file at /var/www/testId-iteration-
concurency-request.csv
  ab += '-e /var/www/results/' + params.testId[0] +
'-' + params.itr + '-' + params.c + '-' + params.n +
'.csv ';

  //protocol
  ab += CONFIGS.conf[params.testId[3]] == 'ssl' ?
'https://' : 'http://';

  //host
  ab += '127.0.0.1/';

  //Test obj
  ab += CONFIGS.testObj[params.testId[2]] + ' ';

  //save stdout to file at /var/www/testId-
iteration-concurency-request.txt
  ab += '> /var/www/results/' + params.testId[0] +
'-' + params.itr + '-' + params.c + '-' + params.n +
'.txt';
```

```
  exec(ab, function(error, stdout, stderr) {
    if(cb && typeof(cb) === 'function') cb();
  });
};

function buildDocument(cb) {
  document['config'] = {
    'testId'      : params.testId[0],
    'os'          : 'Linux/2.6.32-21',
    'webserver'   :
CONFIGS.service[params.testId[1]] == 'Nginx' ?
'Nginx/1.0.0' : 'Apache/2.2.14',
    'database'    : 'MySQL/5.1.41',
    'php'         : '5.3.2-1 (php-fpm)',
    'concurrency' : parseInt(params.c),
    'requests'    : parseInt(params.n),
  };
  document['results'] = new Array();

  results = {
    'iteration'   : params.itr,
    'uptime'      : os.uptime(),
    'mem'         : {
      'total'     : os.totalmem(),
      'free'      : os.freemem(),
    }
  };

  getMemoryPeak( function() {
    results['mem']['lowestPeak'] =
parseInt(testInfo['memPeak']);


  //I know its ugly, running out of time
  //Get time it took
  exec("cat /var/www/results/" + params.testId[0] +
"-" + params.itr + "-" + params.c + "-" + params.n +
".txt | grep 'Time taken' | sed s/[^0-9\.]//g",
function(err, stdout, stderr) {
results['timeTaken'] = stdout.replace('\n', '');

    //Get complete requests
    exec('cat /var/www/results/' + params.testId[0]
+ '-' + params.itr + '-' + params.c + '-' + params.n
```

```
+ '.txt | grep Complete | sed s/[^0-9]//g',
function(err, stdout, stderr) {
      results['completeRequests'] =
stdout.replace('\n', '');

      //Get failed requests
      exec('cat /var/www/results/' +
params.testId[0] + '-' + params.itr + '-' + params.c
+ '-' + params.n + '.txt | grep Failed | sed s/[^0-
9]//g', function(err, stdout, stderr) {
        results['failedRequests'] =
stdout.replace('\n', '');

        //Get requests / second
        exec('cat /var/www/results/' +
params.testId[0] + '-' + params.itr + '-' + params.c
+ '-' + params.n + '.txt | grep Requests | sed
s/[^0-9\.]//g', function(err, stdout, stderr) {
          results['requestsPSecond'] =
stdout.replace('\n', '');

          //Get html transferred
          exec('cat /var/www/results/' +
params.testId[0] + '-' + params.itr + '-' + params.c
+ '-' + params.n + '.txt | grep HTML | sed s/[^0-
9]//g', function(err, stdout, stderr) {
            results['htmlTransferred'] =
stdout.replace('\n', '');

            //Get total transferred
            exec('cat /var/www/results/' +
params.testId[0] + '-' + params.itr + '-' + params.c
+ '-' + params.n + '.txt | grep Requests | sed
s/[^0-9]//g', function(err, stdout, stderr) {
              results['totalTransferred'] =
stdout.replace('\n', '');

              //Store whole ab output
              exec('cat /var/www/results/' +
params.testId[0] + '-' + params.itr + '-' + params.c
+ '-' + params.n + '.txt', function(err, stdout,
stderr) {
                results['ab'] = stdout;
```

```
                    exec('cat /var/www/results/' +
params.testId[0] + '-' + params.itr + '-' + params.c
+ '-' + params.n + '.csv', function(err, stdout,
stderr) {
                      results['abCsv'] = stdout;

                      if(cb && typeof(cb) ===
'function') cb();

                    });
                  });
                });
              });
            });
          });
        });
      });
    });
});
};

function storeDocument(cb) {

  var db = new mongodb.Db('node', new
mongodb.Server('91.123.195.204', 27017, {}), {});

  db.open( function(err, db) {
    var collection = new
mongodb.Collection(db,'apachePHPSSL');

    //Insert new document
    if( params.itr == 1) {
      document['results'].push(results);

      collection.insert( document, {safe : true} ,
function( err, obj) {
        db.close();

        if(cb && typeof(cb) === 'function') cb();
      });
    } else { //Update existing test document

      //WHERE testId = xxx AND concurrency = x AND
requests = x
      var query = {
```

```
        "config.testId" : params.testId[0],
        "config.concurrency" : parseInt(params.c),
        "config.requests" : parseInt(params.n)
      };
      collection.update(query, {$push : { 'results'
: results}}, {safe : true}, function(err) {

        db.close();
        if(cb && typeof(cb) === 'function') cb();
      });
    }
  });
};

var startBench = function() {

  //Turn on gzip, switch configs or get ssl str
  eval(CONFIGS.conf[params.testId[3]])(
function(abSubStr) {

    //Start Webbserver
    startWebservice(
CONFIGS.service[params.testId[1]], function() {
      console.log("Ready to start bench now");

      //Start watch the memory consumption
      var memWatcherId = memWatcher();

      apacheBench(abSubStr, function() {
        console.log("ab complete");

        //Kill memory watcher
        clearInterval(memWatcherId);

        //save system info to ab file

        //restore configs
        eval(CONFIGS.conf[params.testId[3]] +
'_disable')( function() {

          //Build the document to be stored
          buildDocument( function() {

            //save to database
```

```
            storeDocument(function() {
              console.log("Reboot");
              //process.exit(0);
                    //reboot the server
                    exec('shutdown -r 0', function()
{
                     //Aint gonna happend :)
                    });
            });
          });
        });
      });
    });
  });
};

//Lets rock n roll
startBench();
```