Wirksames Tuning für viel besuchte Webauftritte

Häuptling Schnelles Wiesel

Wenn der Apache-Server häufiger nicht oder viel zu langsam antwortet, ist das ärgerlich und erzeugt für den Webmaster Handlungsdruck. Die denkbaren Ursachen sind genauso vielfältig wie die Optimierungsmaßnahmen, um den Engpässen zu begegnen. Christian Kruse

Wenn die User beginnen sich über zu hohe Ladezeiten zu ärgern, ist das für den Webmaster der betroffenen Site ein Grund tätig zu werden. Die Ursachen können vielfältig sein: Netzengpässe, ein überlasteter Server oder die angeforderte Datenmenge ist übermäßig groß. Dieser Beitrag dröselt die verschiedenen Wege auf, um die Antwortzeiten des Apache-Webservers und die Übertragungszeiten zu minimieren, und zeigt zudem, wie jeder Webmaster die Wirksamkeit der eingesetzten Tunigmaßnahmen mit einfachen Benchmarks belegt.

Netz- und Router-Probleme

Bei der Analyse doktert der Admin natürlich nicht an Symptomen herum, sondern geht am besten systematisch vor. Das Wahrscheinlichste ist ein Netzengpass, etwa wegen eines ausgefallenen Routers oder einer kaputten Leitung. Für die Diagnose eignen sich »tracepath« oder »traceroute«, die prüfen, bis wohin eine Verbindung wie schnell zustande kommt, indem sie die Time to Live herabsetzen und auf die »TIME_EXCEEDED«-Antwort jedes Routers warten.

Die Time to Live enthält keine Zeitangabe, sondern einen Wert, über wie viele Hops ein Paket wandern darf. Probleme mit der Netzwerkanbindung kann der Admin so oft nachvollziehen. Listing 1 zeigt ein Beispiel: Beim vierten Hop antwortet der Router »ar-essen2.g-win.dfn.de« verhältnismäßig langsam, aber 0,2 Sekunden sind nicht wirklich relevant. Also liegt der Engpass woanders.

Die nächste Maßnahme ist eine Lastanalyse auf dem Server. Das geht am besten im produktiven Betrieb, beispielsweise per SSH. Die Werkzeuge »top« und »ps« liefern erste Informationen, »uptime« gibt Auskunft über den akuellen Server-Load. Gute Dienste leistet hier auch das spätestens seit dem letzten Linux-Magazin bekannte Tool Apachetop [3].

Leider ist ein Engpass nicht zwingend von permanenter Natur, sondern kann auch nur sporadisch auftreten. Die Lösung: Programme wie MRTG [4] protokollieren unter anderem die Auslastung des Servers über eine längere Zeit. Unter [5] kann man eine Beispiel-Installation von MRTG betrachten.

Wahrscheinlich ergibt die Analyse der MRTG-Graphen, dass entweder der Server überlastet oder der Traffic einfach zu groß ist. Der erste Fall ist am Graphen für Load Average und der Prozessorauslastung erkennbar. Der Graph für die Anzahl der Prozesse liefert den ersten Hinweis auf den Übeltäter: Bleibt die Anzahl der Prozesse weithin konstant, fällt der Verdacht auf ein außer Kontrolle geratenes Programm.

Schwankt die Zahl der Prozesse, sollte der Admin prüfen, ob ein CGI-Skript viele Ressourcen beansprucht. Interessant sind an dieser Stelle das Access-Log des Apachen sowie die Prozessliste (»top«) zu den Lastzeiten. Sie geben Aufschluss darüber, welche URLs besonders häufig aufgerufen wurden. Diese Information versetzen den Admin in die Lage auszumachen, welches Skript oder welches Programm die übergroße Last verursacht. Ein Zuviel an Traffic dagegen stellt sich anders dar. Es ist gekennzeichnet durch konstant niedrige Werte beim Last-Graphen auf dem Server und auffällig hohe für den Netzwerk-Traffic.

Ist der Server überlastet?

Dass der Webserver überlastet ist, muss nichts Besonderes sein. Große Auftritte wie SELFHTML leiden stets an Überlast. Leider kann man dagegen nicht in jedem Fall wirksam etwas tun. Zuerst kommt das Übersetzen der Software mit geeigneten Compilerflags in Betracht, denn

viele Distributionen kompilieren das Apache-Paket aus Kompatibilitätsgründen so, dass es auf 386ern läuft.
Um Geschwindigkeit zu gewinnen, kompiliert der aufmerksame Systembetreiber kritische Software wie Apache selber und nutzt dabei die Gelegenheit, die Compilerflags anzupassen. Bei seinem Athlon-Server (siehe Kasten „Test-Umgebung“) beispielsweise benutzt der Autor dieses Beitrags:

```
CFLAGS="-march=athlon ➥
-fexpensive-optimizations -O3"
```

Diese Flags sagen dem Compiler, dass er Maschinencode erzeugen darf, der nur auf Athlon-CPUs läuft. Er darf auch Optimierungsmethoden einsetzen, die viel Zeit kosten. Die Optimierungstiefe ist Level drei. Klar sollte aber sein, dass die Compilerflags keinesfalls Hand-Optimierungen am Algorithmus ersetzen.
Viele Apache-Module legen während der Requests temporäre Dateien an; auch PHP erzeugt Wegwerf-Dateien zur Sessionsverwaltung. Darum verfrachtet der nächste Optimierungsschritt das »/tmp«-Verzeichnis in eine Memory-Disk, meist reicht eine Größe von 128 MByte. Obwohl Linux effiziente Cashing-Strategien fährt, verhilft die RAM-Disk hier meist zu besseren Reaktionszeiten des Servers.

Tuning vom Apache-HTTPD

Das größte Optimierungspotenzial bietet jedoch Apache selber. Zwar sind seine Grundeinstellungen relativ gut, aber Distributionen wie Suse oder Red Hat Linux laden viel zu viele Module. Im Grunde sind für den alltäglichen Betrieb nur die in Tabelle 1 genannten Module notwendig. Entfernt man alle nicht dringend erforderlichen Module, schrumpfen sowohl der HTTPD-Prozess als auch die Dauer für die Abarbeitung der Requests.
Hintergrund: Apache muss bei jeder Anfrage jedes Modul durchlaufen, das dabei prüft, ob es zuständig ist. Bei Apache 2 muss der Admin allerdings vorsichtig sein: Das Modul »config_log_module« heißt hier »log_config_module«.

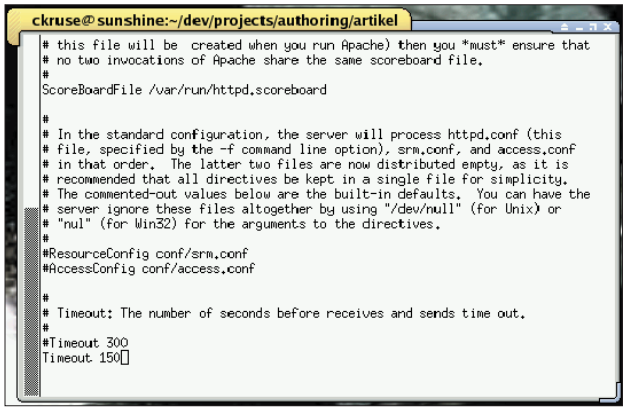

Abbildung 1: Die Praxis zeigt, dass es einiges bringt, den Verbindungs-Timeout in der Apache-Konfigurationsdatei mit der Direktive »Timeout« herunterzusetzen.

Die Praxis zeigt, dass es zudem einiges bringt, den Verbindungs-Timeout in der Direktive »Timeout« herunterzusetzen – 300 Sekunden, um auf neue Pakete und/oder einen Request zu warten, sind einfach zu viel. Derart langatmige Requests blockieren den Kindprozess für neue Verbindungen, die Apache vielleicht schon lange hätte abhandeln können. Ein Timeout von 120 bis 150 Sekunden sollte völlig ausreichend sein, auch bei einer langsamen Internetverbindung (siehe Abbildung 1).
Der Admin sollte auch sicherstellen, dass »KeepAlive« auf »On« steht. Die Direktive schaltet Keep-Alive-Requests ein oder aus. Keep-Alive-Requests eröffnen die Möglichkeit, die Verbindung für mehr als nur eine Anfrage zu nutzen. Ansonsten muss ein Client für jede Anfrage eine neue Verbindung aufbauen – mit dem kompletten TCP-Handshake, was drei Pakete verschwendet. Das macht bei 50 Requests pro Sekunde 9000 Pakete pro Minute, die allein für die Verbindung verschickt werden. Geht man davon aus, dass im Durchschnitt drei Clients diese 50 Requests verursachen, braucht man mit »KeepAlive« auf »On« nur drei Verbindungen pro Sekunde und somit nur noch 540 Paketen pro Minute für den Verbindungsaufbau.

Timeouts runtersetzen

Auch der »KeepAliveTimeout« sollte relativ niedrig gehalten werden. Ein Wert von 15 Sekunden reicht aus. Die »MaxSpareServers«-Direktive beschränkt die Zahl der »httpd«-Prozesse, die im Leerlauf sind. Das heißt, dass in Zeiten geringer Last die Zahl der »httpd«-Prozesse sinkt. Das bedeutet allerdings auch, dass der Apache in Zeiten großer Last erst neue »httpd«-Prozesse aufforken muss, was Rechenzeit kostet.
Bei Servern, die mit sehr starken Lastproblemen kämpfen, sollte der Admin diese Direktive auskommentieren und beobachten, ob sich das Verhalten verbessert. Die »MaxSpareServers« auskommentieren sollte er jedoch wirklich nur

Listing 1: Ein »tracepath«-Auszug

```
01 ckruse@sunshine:~ $ tracepath www.defunced.de
02 1?: [LOCALHOST]      pmtu 1500
03 1:  fogg.defunced.de (192.168.1.1)                                2.625ms
04 2:  10.3.11.1 (10.3.11.1)                                         5.109ms
05 3:  gwin-gw-gig00-112.HRZ.Uni-Dortmund.DE (129.217.129.190)      13.403ms
06 4:  ar-essen2.g-win.dfn.de (188.1.44.33)                        222.119ms
07 5:  cr-essen1-ge4-0.g-win.dfn.de (188.1.86.1)                    12.839ms
08 6:  cr-frankfurt1-po8-1.g-win.dfn.de (188.1.18.89)               24.964ms
09 7:  188.1.80.42 (188.1.80.42)                                    24.872ms
10 8:  gi-0-3-ffm2.noris.net (80.81.192.88)                         26.466ms
11 9:  ge0-2-151-nbg5.noris.net (62.128.0.209)                      40.772ms
12 10:  no.gi-5-1.RS8K1.RZ2.hetzner.de (213.133.96.25)              31.012ms
13 11:  et-1-16.RS3K1.RZ2.hetzner.de (213.133.96.230)               32.433ms
14 12:  srv001.occuris.de (213.133.103.124)                      38.802ms reached
15 Resume: pmtu 1500 hops 12 back 12
```

Tabelle 1: Essenzielle Apache-Module

Modulname	Modulname
env_module	alias_module
config_log_module	rewrite_module
	access_module
mime_module	auth_module
includes_module	setenvif_module
autoindex_module	headers_module
dir_module	expires_module
cgi_module	php4_module
action_module	gzip_module

bei ständig unter starker Last ächzenden Servern, weil so die Menge der »httpd«-Prozesse stetig zunimmt.

Anders sieht es bei der »MinSpareServers«-Direktive aus. Sie bestimmt, wie viele Prozesse mindestens nichts zu tun haben sollten. Diese Prozesse bilden ein Polster für Lastspitzen. Achtung: Bei Apache 2 hängt das Verhalten vom gewählten MPM-Modul ab [6], [7]. Wird das traditionelle Prefork-Modell benutzt, ändert sich nichts. Ist aber das Worker-MPM im Spiel, ändern sich die Direktiven-Namen in »MaxSpareThreads« und »MinSpareThreads«.

Hier bezieht sich die Angabe auf die Anzahl der Threads pro Apache-Prozess. Generell ist bei Apache 2 das Erzeugen neuer Threads simpler als das von Prozessen. Denn ein Thread hat keinen eigenen Speicherbereich, der kopiert werden muss. Er hat auch sonst keine Prozesseigenschaften, die teilt er sich mit den anderen Threads.

Listing 2: Ergbnisse vor Optimierung

```
01 Concurrency Level:      10
02 Time taken for tests:   2.842457 seconds
03 Complete requests:      100
04 Failed requests:        0
05 Write errors:           0
06 Total transferred:      11023620 bytes
07 HTML transferred:       10994986 bytes
08 Requests per second:    35.18 [#/sec] (mean)
09 Time per request:       284.246 [ms] (mean)
10 Time per request:       28.425 [ms] (mean, across all
   concurrent requests)
11 Transfer rate:          3787.22 [Kbytes/sec] received
12
13 Connection Times (ms)
14               min  mean[+/-sd] median   max
15 Connect:        0    0   1.0      0       5
16 Processing:    39  278 342.6    105    1012
17 Waiting:      -83  -22  19.1    -21       1
18 Total:         39  278 342.8    106    1012
```

Listing 3: Ergebnisse nach Optimierung

```
01 Concurrency Level:      10
02 Time taken for tests:   1.58400 seconds
03 Complete requests:      100
04 Failed requests:        0
05 Write errors:           0
06 Total transferred:      11290258 bytes
07 HTML transferred:       11261068 bytes
08 Requests per second:    94.48 [#/sec] (mean)
09 Time per request:       105.840 [ms] (mean)
10 Time per request:       10.584 [ms] (mean, across all
   concurrent requests)
11 Transfer rate:          10416.67 [Kbytes/sec] received
12
13 Connection Times (ms)
14               min  mean[+/-sd] median   max
15 Connect:        0    0   0.3      0       3
16 Processing:    34   98  21.1     95     155
17 Waiting:     -102  -46  17.9    -42       0
18 Total:         34   98  21.0     95     155
```

Ein Kompromiss ist gefragt

Die nächste Performance-interessante Direktive ist »MaxRequestsPerChild«. Sie sorgt dafür, dass nach einer bestimmten Anzahl von Requests der Kindprozess beendet und neu aufgefork wird. Leider gibt es Bibliotheken und Module, die Speicher-Leeks aufweisen, weshalb der Gebrauch dieser Direktive erforderlich ist. Ob dies auf dem eigenen System so ist, muss aber jeder selbst ausprobieren. Sollten nach einem halben Tag Betrieb bei viel Last die Apache-Prozesse eine ungewöhnliche Größe erreichen – feststellbar mit dem Tool »top« –, ist das Einschalten der Direktive wohl oder übel nötig. Dann sollte deren Wert recht hoch gewählt werden, wie hoch genau, hängt von der Systemkonfiguration ab und ist auszuprobieren. Der Wert ist ein Kompromiss zwischen Performance und der Ressource Speicher.

Zentral ist die »HostnameLookups«-Direktive. Aus Geschwindigkeitssicht darf sie nur einen Wert haben: »Off«. Ihr Einschalten würde für jeden Request einen Reverse-DNS-Lookup erzwingen, der wiederum oft mehrere DNS-Requests notwendig macht, unter Umständen zu netztopologisch sehr weit entfernen Nameservern. Das erzeugt nicht nur viel Traffic, es kostet auch unnötig Zeit.

Wie viel bringen Apache-Optimierungen?

Die Wirkung solcher Optimierungen ist schwer zu messen. Das Skript [1] eignet sich nicht, da es den Server nicht belastet. Der Apache-Benchmark »ab« (Version 2, [2]) leistet aber das Gewünschte (siehe Kasten „Test-Umgebung“). Eingangs muss ein Benchmark-Lauf vor allen Optimierungen erfolgen:

```
ckruse@sunshine:~ $ ab2 -n 100 -c 10 ➥
http://rain/artikel/
```

Das Programm initiiert 100 Request-Phasen, wobei immer zehn Requests gleichzeitig starten. Listing 2 zeigt ein auf die relevanten Werte gekürztes Resultat. Interessant sind »Requests per second« und »Time per request«: rund 35 Requests pro Sekunde und durchschnittlich 28 Millisekunden pro Request. Nun nimmt man die Optimierungen vor und führt den Benchmark erneut nach diesem Muster aus, Listing 3 zeigt dies. Offenbar war hier das Tuning von Erfolg gekrönt: 100 Prozent. Vorsichtig überschlagen verträgt die Site nun doppelt so viele Requests wie bisher.

Optimierung auf HTTP-Ebene

Das Optimieren auf HTTP-Level zielt primär auf das Sparen von Traffic und Re-

Performance-Killer Standard-CGIs

Auf dem Webserver ausgeführte CGI-Skripte bedürfen der erhöhten Aufmerksamkeit des Admins. Diese an sich nützlichen Programmchen stellen gern ein ebenso häufiges wie merkliches Performanceproblem dar, da sie das Starten aufwändiger Interpreter-Binaries nach sich ziehen.

Das folgende, anonymisierte Beispiel für diese These ist dem Autor persönlich bekannt: Auf einem im deutschsprachigen Raum sehr bekannten Webserver arbeitete ein großes und aufwändiges Perl-CGI-Skript. Da der Server von Lastproblemen heimgesucht wurde, entwickelte das verantwortliche Webteam für das Skript eine neue Routine, die den Load Average überprüft und bei Überlast den CGI-Prozess killt. Die Wirksamkeit dieser Maßnahme stellte sich aber als gering heraus: Die Last war zu Spitzenzeiten noch immer viel zu hoch – das Starten des Interpreters und das interpretative Abarbeiten des Skripts schluckten Ressourcen en masse.

Im nächsten Schritt schrieben die Programmierer in C ein 804 Byte kleines Binary, das Performance-abhängig verzweigt: Ist die Last zu hoch, gibt es nur eine Fehlermeldung aus, andernfalls startet es das ursprüngliche CGI-Skript. Diese Maßnahme erwies sich als wirkungsvoll, denn das System bewältigte fortan Lastspitzen viel leichter.

Würden die Programmierer der Site einen Schritt weiter gehen und ein Apache-Modul »mod_loadavg« schreiben, könnten sie die Last noch weiter senken. Solche Module ersparen den Aufruf des zugehörigen Interpreters. Es gibt für fast alle Sprachen passende Apache-Module: »mod_perl«, »mod_php«, »mod_ruby«, »mod_fastcgi« und so weiter.

1/1 A

210 x 297 mm
(213 x 303 mm inkl. Beschnitt)

NMG Ulm

quests. Das erreicht man mit Caching- und bedingten Headern. Caching-Header teilen dem User Agent (UA) mit, dass er Inhalte eine bestimmte Zeit lang nicht neu anzufordern braucht. Das bietet sich etwa für statische HTML-Dateien oder CSS-Files an. Dazu bezeichnet der »Expires«-Header den Zeitpunkt, ab dem ein Dokument als veraltet zu betrachten und neu anzufordern ist.

Möchte man dem UA – sei es ein Browser, sei es ein Proxy oder ein anderer Client – sagen, dass das Dokument bereits bei seiner Auslieferung veraltet ist und damit jedes Mal neu angefordert werden muss, setzt man den Wert dieses Feldes auf den Wert des »Date«-Headers. Nicht gültig sind Werte wie »now« oder »0«, sie werden aber im Sinne der Fehlertoleranz akzeptiert und bezeichnen eine bereits abgelaufene Gültigkeit. Es steht jedoch jedem Client frei, ob er sich daran hält oder nicht.

Listing 4: Setzen der »Expires«- und »Cache-Control«-Header

```
01 ExpiresActive On
02 ExpiresByType text/html "access plus 1 month"
03 ExpiresByType text/css "access plus 6 month"
04 ExpiresByType text/javascript "access plus 6 month"
05 ExpiresByType image/gif "access plus 6 month"
06 ExpiresByType image/jpeg "access plus 6 month"
07 ExpiresByType image/png "access plus 6 month"
08
09 <Files ~ "\.(js|css|gif|jpe?g|png)$">
10  Header append Cache-Control "public"
11 </Files>
```

Listing 5: Beispiel-HTTP-Anfrage

```
01 ckruse@sunshine:~ $ telnet rain 80
02 Trying 192.168.1.3...
03 Connected to rain.defunced.de.
04 Escape character is '^]'
05 GET /artikel/ HTTP/1.1
06 Host: rain
07 Connection: close
08
09 HTTP/1.1 200 OK
10 Date: Sun, 09 Nov 2003 19:52:04 GMT
11 Server: Apache/1.3.27 (Unix) PHP/4.2.3 AuthMySQL/2.20
12 Cache-Control: max-age=2592000
13 Expires: Tue, 09 Dec 2003 19:52:04 GMT
14 Last-Modified: Tue, 04 Nov 2003 23:50:42 GMT
15 ETag: "a3a068-1a9d9-3fa83b52"
16 Accept-Ranges: bytes
17 Content-Length: 109017
18 Connection: close
19 Content-Type: text/html
20
21 <I>HTML-Inhalte<I>
```

Der »Cache-Control«-Header ist da strikter. Er bezeichnet Cache-Anweisungen, die jeder RFC-konforme HTTP-Client befolgen muss. Die Syntax von »Cache-Control« unterscheidet sich ein wenig von der »Expires«-Syntax. Die Verfallszeit eines Dokuments wird über »max-age = *Wert*« angegeben. *Wert* ist hierbei die Gültigkeitsdauer in Sekunden ab dem Zeitpunkt, der im »Date«-Feld angegeben ist. Gibt man hier den Wert »0« an, bedeutet dies, dass ein Dokument gar nicht gecacht werden darf – was jedoch nicht zu empfehlen ist.

Ignoranz unterstützen

Gelegentlich ist auch das Attribut »public« im Cache-Control-Header für die Geschwindigkeit von Vorteil, denn es verhilft Proxy-Caches zu größerer Wahlfreiheit: Trifft ein so instruierter Proxy auf einen Request, der eigentlich nicht gecacht werden darf – beispielsweise ein über HTTP-Auth authentifizierter Request –, kann er sich über diese Anweisung hinwegsetzen und trotzdem zwischenspeichern.

Beide Header sind erst mit den Modulen »mod_expires« und »mod_headers« nutzbar. Per »mod_expires« kann der Admin festlegen, wann ein Dokument verfallen soll. Jedoch kennt das Modul das Attribut »public« nicht, was »mod_headers« nötig macht, um auch Bild-, Javascript- und CSS-Dateien in den Genuss des »public«-Attributs kommen zu lassen. Das Beispiel einer HTTP-Anfrage an einen so konfigurierten Server zeigt Listing 4. Die in Listing 5 gestellte Anfrage könnte der UA jetzt einen Monat lang cachen, ohne den Server zwischenzeitlich kontaktieren zu müssen.

Die Praxis jedoch zeigt, dass zwischenspeichernde UAs das Ergebnis eine Weile cachen, in der Größenordnung von zwei Tagen, und danach über einen bedingten Header anfragen, ob die interne Version noch aktuell ist oder neu anzufordern ist. Das geschieht mit der »If«-Headergruppe. Dort sind »If-Match«, »If-Modified-Since«, »If-None-Match«, »If-Range« und »If-Unmodified-Since« erlaubt.

In der freien Wildbahn kommen aber nur »If-None-Match« (Opera ab Version 7 und Mozilla) sowie »If-Modified-Since« (Mozilla, Opera ab Version 7 und Internet Explorer ab 5.5) vor. Die anderen Header erklärt [8].

Bedingte Header

Beide bedingten Header beziehen sich auf Felder, die der Server aufgrund einer vorherigen Anfrage geliefert hat. »If-Modified-Since« steht in Verbindung zum Feld »Last-Modified«, das das Datum der letzten Änderung des Dokuments bezeichnet. Und der Header »If-None-Match« bezieht sich auf »ETag«, eine Prüfsumme für das Dokument.

Der User Agent speichert diese Werte und stellt einen entsprechenden Request, wie in Listing 6 zu sehen. Wie hier erkennbar wird, beantwortet der Server die Anfrage nur noch mit »304 Not Modified« sowie einigen anderen Headern, jedoch ohne Inhalt.

Mit Blick auf CGI-Skripte gibt es hier einiges zu beachten: Das CGI-Skript soll einen »Last-Modified«- oder einen »ETag«-Header schicken, damit der Browser seine Conditional-Get-Algorithmen verwenden kann. Hinzu kommt, dass das Skript auch die Header auswerten muss. Und es hat die Aufgabe, das Datum in eine geeignete Form zu bringen (meist ist das die Anzahl der Sekunden seit 1970) und dann prüfen, ob die Version, die der »If-Modified-Since«-Header beschreibt, veraltet ist.

Das gilt auch für den »If-None-Match«-Header: Das Skript wertet die Prüfsumme aus und stellt auf diese Weise fest, ob die im Header beschriebene Version veraltet ist. Als Prüfsummen-Algorithmen eignen sich kryptographische Hashing-Verfahren wie MD5. Der Zugriff auf die Werte der beiden Header erfolgt über die beiden CGI-Umgebungsvariablen »HTTP_IF_MODIFIED_SINCE« und »HTTP_IF_NONE_MATCH«.

20fach beschleunigt

Für den Nachweis der Wirksamkeit der gerade beschriebenen Maßnahmen lässt sich das Skript ([1], siehe Kasten „Test-Umgebung“) einsetzen:

```
ckruse@sunshine:~ $ ./measure.pl --base-url
 http://rain/artikel/
Getting http://rain/artikel/...
[...]
Time elapsed: 4.642203 seconds
```

Beim Zuschalten der konditionale Header dauert der ganze Request plötzlich weniger als eine Sekunde:

```
ckruse@sunshine:~ $ ./measure.pl ➥
--send-if-modified-since ➥
--send-if-none-match --base-url ➥
http://rain/artikel/
Getting http://rain/artikel/...
[...]
Time elapsed: 0.181876 seconds
```

Kompressor-Technik

Die andere Möglichkeit, über HTTP etwas zu tunen, ist Content Encoding. Der UA schickt beim Anfordern eines Dokuments den Header »Accept-Encoding« mit, in dem Angaben über die Form der Verarbeitung des Inhalts stehen. Gültige Werte sind zum Beispiel »gzip« [9] oder »compress«. Beide Angaben bezeichnen Algorithmen, die den Inhalt des Dokuments vor dem Ausliefern Server-seitig komprimieren und Client-seitig entpacken. Die Methode spart sehr viel Traffic – bis zu 90 Prozent.

Die Sache hat aber gewichtige Nachteile: Erstens kommen nicht alle UAs damit klar – Netscape 4 zum Beispiel schickt »Accept-Encoding: gzip«, obwohl der Browser die Methode nicht richtig beherrscht. Zweitens kostet es Server-seitig Performance, denn der Apache-Server muss jedes Mal jedes Dokument vor dem Ausliefern komprimieren.

Drittens leidet die Technik an der mangelnden Akzeptanz diverser Zwischenstationen: Viele Contentfilter entfernen einfach den Accept-Encoding-Header. Auch reagiert die Mehrzahl der Proxies allergisch auf den wegen der Komprimierung nötigen »Vary«-Header. »Vary« führt alle Header auf, von dem diese Variante des Requests abhängt. Älteren Versionen des HTTP-Proxies Squid schalten alle Formen des Cashings ab

Test-Umgebung

Abbildung 2: Ausschnitt aus dem Benchmark-Perl-Skript des Autors, das auf der Linux-Magazin-Webseite zu downloaden ist.

Ob eine Optimierung sinnvoll oder überflüssig war, beweisen Benchmarktests. Es liegt dabei nahe, die Zeit zu messen, die der Server von der Anfrage bis zum Ende der Übermittlung braucht. Der Autor betreibt zu diesem Zweck zwei PCs, die über ein internes 100-MBit-Ethernet verbunden sind. Auf dem Server-PC (AMD Athlon 600 mit 64 MByte SD-RAM) laufen FreeBSD 4.6 und Apache 1.3.27.

Alle im Artikel beschriebenen Webserver-Eigenschaften gelten auch für Apache 2; andernfalls wird explizit darauf hingewiesen. Um ISDN-Geschwindigkeit zu simulieren – sie macht Zeitdifferenzen deutlicher – ist außerdem Mod_bandwidth installiert.

Auf dem Client-Rechner (AMD Duron 800 mit 312 MByte SD-RAM) laufen Gentoo Linux 1.4 mit Kernel 2.4.22 und Perl 5.8.0. Als wichtigstes Benchmarking-Tool dient ein vom Autor dieses Beitrags entworfenes Perl-Skript, das unter [1] zu haben ist. Abbildung 2 zeigt einen Ausschnitt. Es lädt die ihm als Parameter übergebene URL und alle darin enthaltenen Referenzen herunter, die zum Rendern nötig sind. Dabei misst es die dafür gebrauchte Zeit (Abbildung 3).

Das zweite hier eingesetzte Benchmark-Tool kommt bereits mit dem Apache: »ab« [2]. Es ist haupsächlich für Lastanalysen gedacht. Dafür simuliert es eine sehr hohe Anzahl zeitgleicher Anfragen und misst, wie viele Requests pro Sekunde möglich sind und wie lange der Apache für deren Bearbeitung braucht.

Abbildung 3: Das Benchmark-Skript des Autors in Aktion. Es lädt die übergebene URL herunter und misst die dafür gebrauchte Zeit.

und fordern das Dokument bei jeder Anfrage neu an – für den Website-Betreiber kaum erstrebenswert. Der Einsatz von »mod_gzip« will also genau überlegt und auf das Zielpublikum abgestimmt sein.

Tuning auf HTML-Level

In HTML-Dateien schlummert das größte Optimierungspotenzial. Viel HTML-Code – gerade im kommerziellen Bereich – liefern HTML-Designer, die mit Wysiwyg-Editoren wie Dreamweaver arbeiten. So sieht der Code auch aus. Nicht selten lässt er sich um mehr als 50 Prozent kürzen, bei gleichem Aussehen versteht sich. Generell ist es sinnvoll, so viel Formatierungsangaben wie möglich aus HTML-Dateien zu verbannen und in CSS-Dateien zu legen, was Wysiwyg-Editoren eigentlich – oft im Gegensatz zu ihren Benutzern – beherrschen.

CSS-Dateien werden, anders als eingebettete Formatierungsangaben, in der Regel nur einmal geladen, sodass der HTTP-Server die Formatierung nur einmal übermittelt muss. Unnötige Verschachtelungen von Tabellen, wie Dreamweaver sie macht, sind zu vermeiden. Der Webdesigner sollte solchen Code auf keinen Fall ungeprüft übernehmen, sondern zum Beispiel mit HTML-Tidy **[10]** überarbeiten.

Bilder und Grafiken

Bei Bildern kann der Webdesigner oft viel Traffic sparen. Von Bedeutung und von allen Browsern unterstützt sind die Bildformate Gif, Jpeg und PNG. Einfache Strichgrafiken und Cartoons sind eine Domäne von Gif, während Jpeg für Fotos und sehr farbenreiche Bilder gedacht ist, allerdings ist sein Kompressionsalgorithmus verlustbehaftet.

Das Gif-Format wäre für diesen Fall besonders ungeeignet, da seine Farbpalette auf 256 Farben beschränkt ist. Andere Farben muss es durch ein Rasterverfahren (Dithering) herstellen, was die Datei unschön groß werden lässt. Ist die Qualität primär, wird das verlustfrei kompimierende PNG zum Format der Wahl, wenngleich die Dateien etwas größer als Jpegs ausfallen.

Oft sieht man, dass Bilder zwar in sehr hohen Auflösungen veröffentlicht, aber dann durch die zugehörige HTML-Datei über »width« und »height« heruntersklaliert werden. Das hat gleich zwei Nachteile: Zum einen passieren unnötig große Bilddateien die Leitungen. Zum anderen leidet die Darstellung im Webbrowser, da übliche Rendering-Engines Bilder sichtbar schlechter herunterrechnen als es Grafikprogramme vermögen.

Heutige Webdesigner brauchen aber nicht gleich ins Gegenteil zu verfallen wie in Zeiten der 14400-Baud-Modems und des teuren Webspace. Damals war es üblich, Bilder kleiner abzuspeichern, um sie dann vom Browser hochskalieren zu lassen. Aber auch in dieser Richtung skalieren gängige Clients mäßig. Sinnvoll ist, die Bilder auf genau jene Größe zu skalieren, die die Anwendung zu 100 Prozent darstellen wird – das ist der optimale Kompromiss zwischen Datenmenge und bester Qualität.

Listing 6: Ein bedingtes GET

```
01 ckruse@sunshine:~ $ telnet rain 80
02 Trying 192.168.1.3...
03 Connected to rain.defunced.de.
04 Escape character is '^]'.
05 GET /artikel/ HTTP/1.1
06 Host: rain
07 Connection: close
08 If-Modified-Since: Tue, 04 Nov 2003 23:50:42 GMT
09 If-None-Match: "a3a068-1a9d9-3fa83b52"
10
11 HTTP/1.1 304 Not Modified
12 Date: Sun, 09 Nov 2003 20:28:43 GMT
13 Server: Apache/1.3.27 (Unix) PHP/4.2.3 AuthMySQL/2.20
14 Connection: close
15 ETag: "a3a068-1a9d9-3fa83b52"
16 Expires: Tue, 09 Dec 2003 20:28:43 GMT
17 Cache-Control: max-age=2592000
18
19 Connection closed by foreign host.
```

Leerzeichen und Zeilenumbrüche

Da HTML eine weitgehend formatfreie Sprache ist, darf man alle syntaktisch überflüssigen Leerzeichen und Zeilenumbrüche entfernen. Dem UA ist gleichgültig, ob der HTML-Code schön formatiert wurde oder Salat ankommt. Das eröffnet eine weitere Möglichkeit, um Traffic zu sparen. Das gleiche Ziel verfolgt, wer Windows- in Unix-Zeilenumbrüche konvertiert, die in einem statt in zwei Byte stecken. Diese Arbeit verrichten alle modernen Editoren.

Apache-2-Benutzer können auf das Modul »mod_blanks« zurückgreifen, das genau diese Aufgaben erledigt. Alle anderen erreichen einen vergleichbaren Effekt, wenn sie zwei Versionen pflegen, eine für die Entwicklung und eine für den Auftritt. Nach jeder Änderung an einem Entwicklerfile lässt man einfach ein kleines Skript, zum Beispiel **[11]**, rüberlaufen. Die dabei entstehende Datei ist aller entbehrlichen Leerzeichen und Zeilenumbrüche beraubt.

Wer alle diese Tuningmaßnahmen auf seine Site anwendet, erreicht zweifellos bei mittlerer und hoher Last eine bessere Performance. Wie stark der Gewinn ist, lässt sich in einem Vorher-nachher-Vergleich mit den beschriebenen Benchmarks beweisen. *(jk)* ■

Infos

[1] Das Benchmark-Skript des Autors: [http://www.linux-magazin.de/Service/Listings/2004/01/Apache-Tuning]
[2] Apache Benchmarkingtool: [http://httpd.apache.org/docs/programs/ab.html]
[3] Charly Kühnast, „Aus dem Alltag eines Sysadmin: Apachetop“: Linux-Magazin 1/2004, S. 53
[4] Multi Router Traffic Grapher: [http://people.ee.ethz.ch/~oetiker/webtools/mrtg/]
[5] Beispiel-Installation von MRTG: [http://www.defunced.de/mrtg/]
[6] Multi-Processing-Module: [http://httpd.apache.org/docs-2.0/mpm.html]
[7] Th. Grahammer, „Apache 2.0“: Linux-Magazin 10/2002, S. 44
[8] HTTP-RFC: [ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt]
[9] U. Keil, „ Dynamische Komprimierung von Webseiten mit Mod_gzip und Apache“: Linux-Magazin 10/2002, S. 39
[10] HTML-Tidy: [http://sourceforge.net/projects/tidy/]
[11] Skript zum Entfernen der Leerzeichen: [http://www.defunced.de/blanks.pl.gz]

Der Autor

Christian Kruse ist Informatikstudent an der FH Dortmund. Nebenberuflich administriert er eine Reihe von Linux- und FreeBSD-Servern, darunter das bei HTML-Entwicklern recht bekannte Angebot SELFHTML.