Das einzige deutschsprachige Linux-Magazin

LINUX MAGAZIN 8/98

LINUX MAGAZIN

ISSN 1432 - 640 X DM 9,80 ÖS 75,- SFR 9,80

8/98

Linux News

Neues aus dem Internet

Bericht

LinuxExpo 98

RoboCup

Mit Linux zur Fußball-WM

Know-how

Cron-Jobs

Linux-Kontor

Warenwirtschafts-programm für Linux

www.linux-magazin.de

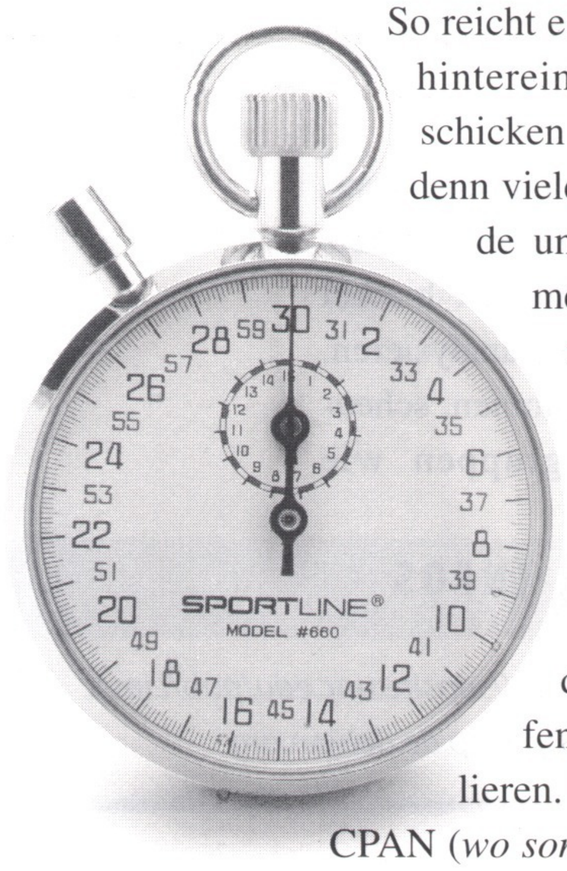
Perl Snapshot

Bis in den roten Bereich

von Michael Schilli

Wieviele Seiten schafft mein Web-Server pro Sekunde? Wieviele Clients kann ich mit CGI-Skripts pro Sekunde bedienen? Was bringen die Apache-Beschleuniger *fast-cgi* und *mod_perl* wirklich? Ein Testskript simuliert parallel anstürmende Web-Browser und zeigt, wo die Grenzen des eigenen Web-Servers liegen.

Spricht man von der Performance eines Web-Servers, sind zwei Parameter ausschlaggebend. Da ist einmal die Verzögerung (*Latency*), die Zeit, die verstreicht, bis der Server die Anfrage eines Clients bearbeitet hat und die verlangte Seite ausliefert. Wichtiger noch ist allerdings der maximale Durchsatz (*Throughput*), der angibt, wieviele Anfragen der Server pro Sekunde bearbeiten kann, bis parallel anstürmende Clients die CPU des Servers so stark belasten, daß dieser mit der Abarbeitung nicht mehr nachkommt und manche Clients irgendwann die Geduld verlieren und einen Timeout melden.



So reicht es denn nicht, von einem Test-Client aus hintereinander Anfragen an den Server zu schicken und die verstrichene Zeit zu messen, denn viele Server sind zu merklich mehr imstande und handeln Requests parallel ab. Vielmehr muß der Test-Client eine einstellbare Anzahl von Requests quasi gleichzeitig an den Server stellen und deren Ergebnisse asynchron bearbeiten. Für diesen Zweck ist das Modul *LWP::Parallel::UserAgent* von *Marc Langheinrich* hervorragend geeignet, denn mit ihm lassen sich parallel laufende UserAgents erzeugen und kontrollieren. Die neueste Version *2.32* liegt auf dem CPAN (*wo sonst!*) unter

```
modules/by-module/LWP/ParallelUserAgent-
2.32.tar.gz
```

vor. Es setzt die installierte *libwww*-Distribution voraus, die auf dem CPAN unter

```
modules/by-module/LWP/libwww-perl-5.33.tar.gz
```

zur Abholung bereitliegt.

Genaue Zeitmessungen

Um die zwischen zwei Meßpunkten verstrichene Zeit nicht nur sekundengenau zu ermitteln, sondern auch Bruchteile von Sekunden zu erfassen, muß das Modul *Time::HiRes* von *Douglas E. Wegscheid* 'ran. Die neueste Version *1.16* liegt unter

```
modules/by-module/Time/Time-HiRes-01.16.tar.gz
```

auf dem CPAN. Die Funktion *Time::HiRes::gettimeofday* ermittelt die aktuelle Uhrzeit so genau, daß ein anschließender Aufruf von *Time::HiRes::tv_interval* mit zwei Meßpunkten als Parametern die dazwischen verstrichene Zeit als Fließkommazahl in Millisekunden-Auflösung zurückliefert. Das Skript *hires.pl* (siehe Listing 1) ermittelt die Zeit, die ein Aufruf von *sleep(1)* verbrät und gibt sie aus:

```
Verstrichene Zeit: 0.99861 Sekunden
```

Hämmern mit dem parallelen UserAgent

Das Modul *LWP::Parallel::UserAgent* erzeugt UserAgents, die (beinahe) gleichzeitig auf einen Webserver losfeuern. Der Konstruktor

```
$ua = LWP::Parallel::UserAgent->new();
```

erzeugt das Mutter-Objekt, das die wilde Horde unter Kontrolle hält. Jeder einzelne Request muß mit

```
$ua->register($request);
```

beim *Parallel::UserAgent* registriert werden, wobei

$request eine Referenz auf ein Objekt vom Typ *HTTP::Request* ist, das vorher zum Beispiel mit

```
my $request = HTTP::Request->new('GET', $url);
```

erzeugt wurde. Die Methode

```
$ua->wait($timeout);
```

startet dann den Ansturm. Jeder Client wartet maximal *$timeout* Sekunden auf prompte Bedienung. Dabei achtet der multiple *UserAgent* darauf, nur eine über

```
$ua->max_req($max_clients);
```

voreingestellte Anzahl von Clients *gleichzeitig* zu starten und erst dann neue Clients nachzulegen, falls alte ihre Mission beendet haben.

Das ist genau, was das Skript *pounder.pl* zur Performance-Messung tut (siehe Listing 2): Die Zeilen 60 bis 71 erzeugen das Mutter-Objekt und pressen den gleichen Request so oft hinein, wie es die Variable *$nof_requests_total* aus der Konfigurationssektion ab Zeile 9 vorgibt. Die Zeilen dort sind vor dem Skriptstart an die lokalen Gegebenheiten anzupassen, *$nof_parallel_connections* legt die Anzahl ``gleichzeitig'' loslegender Clients fest, *$url* den URL der zu testenden Webseite und *$timeout* die maximale Anzahl von Sekunden, die ein Client auf Bedienung wartet.

Zeile 76 hält noch schnell die Startzeit fest, bevor die *wait*-Methode aus Zeile 77 den Massentest startet und erst zurückkehrt, wenn auch der letzte Client Erfolg oder Misserfolg gemeldet hat. Nach Abschluß des Rennens hält Zeile 78 die Stoppuhr an, indem es die Funktion *tv_interval* aus dem Paket *Time::HiRes* mit nur einem Parameter (der Startzeit) aufruft, was *tv_interval* dazu veranlaßt, die seit dem Startzeitpunkt vergangene Zeit in Sekunden plus Bruchteilen als Fließkommazahl zurückzugeben.

Die *wait*-Methode liefert eine Referenz auf einen Hash zurück, der als Values Referenzen auf Objekte vom Typ *LWP::Parallel::UserAgent::Entry* enthält: Das Ergebnis jeder während des Tests aufgebauten HTTP::Verbindung läßt sich so nochmal abfragen. Ein *Entry*-Objekt liefert, falls es mit der *response*-Methode dazu aufgefordert wird, eine Referenz auf ein *HTTP::Response*-Objekt zurück, das wiederum mit der Methode *is_success* Erfolg meldet oder im Fehlerfall mit den Methoden *code* und *message* Fehler-Code und -Meldung bereitstellt.

Im Erfolgsfall zählt *pounder.pl* die Variable *$succeeded* um Eins hoch, falls etwas schieflief, speichert der Hash *%errors* die Fehlermeldung als Key und die Häufigkeit des aufgetretenen Fehlers als Value (Zeile 93). Die Zeilen 100-102 machen daraus eine kommaseparierte Liste von Fehlermeldungen und deren Häufigkeiten und legen sie im String *$errors* ab.

Listing 1: hires.pl

```
 1 #!/usr/bin/perl -w
 2
 3 use Time::HiRes;
 4
 5             # Zeit erfassen ...
 6 $start = [Time::HiRes::gettimeofday];
 7
 8 sleep(1);  # ... schlafen ...
 9
10             # ... und abermals die Zeit erfassen
11 $stop  = [Time::HiRes::gettimeofday];
12
13             # Verstrichene Zeit ermitteln
14 $elapsed = Time::HiRes::tv_interval($start,
$stop);
15
16 print „Verstrichene Zeit: $elapsed Sekunden\n";
```

Die Ausgabe der Ergebnisse im Format

```
URL:
http://localhost/perl/dump.cgi
Total Requests:  100
Parallel Agents: 5
Succeeded:       100 (100.00%)
Errors:          NONE
Total Time:      22.75 secs
Throughput:      4.39 Requests/sec
Latency:         1.02 secs/Request
```

übernehmen die Zeilen 107 bis 143. Zunächst legt das Skript im Array *@P* die linken und rechten Seiten der Ausgabe als aufeinanderfolgende Elemente ab. Die korrekte Formatierung übernimmt anschließend die Formatanweisung aus den Zeilen 136-139, die die linke Spalte der ausgegebenen Tabelle jeweils 16 Zeichen breit linksbündig setzt und eine beliebig breite rechte Spalte daneben setzt.

Die Zeilen 141 bis 143 extrahieren jeweils zwei Elemente aus *@P* und die *write*-Anweisung gibt sie schön formatiert aus. Nun war ich bislang kein besonderer Fan von Perls Format-Befehlen, aber für diese Anwendung taugt's mir — *öfter mal was Neues!* Näheres hierzu zeigt übrigens die Manualseite, die mit *perldoc perlform* zum Vorschein kommt.

Die wichtigste Ausgabe ist zweifellos der Durchsatz. Die Zeile

```
Throughput:       4.39 Requests/sec
```

kommt dadurch zustande, daß das Skript die Zeit mißt, die zwischen dem Starten der *$ua->wait*-Methode und deren Beendigung vergeht und anschließend durch die Anzahl der erfolgreichen Requests teilt.

Abbildung 1: Schnell, schneller, am schnellsten

Die Meßwerte aus einer Testreihe mit 100 Requests und 5 parallelen User-Agents. Dargestellt sind die Ergebnisse für eine statische Seite (index.html), ein CGI-Skript (cgi-bin/dump.cgi), ein CGI-Skript unter dem Apache-Beschleuniger *mod_perl* (perl/dump.cgi) und einen Fehlerfall, der nach einem nicht existierenden URL verlangt.

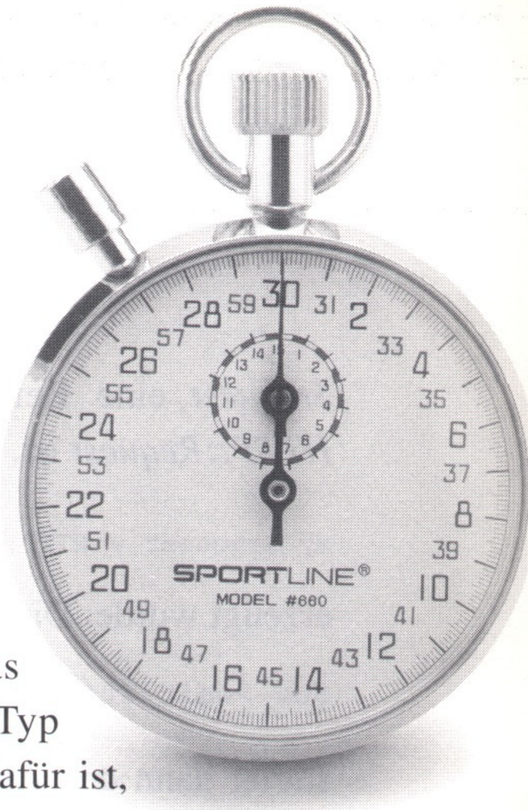
```
URL:                http://localhost/index.html
Total Requests:   100
Parallel Agents: 5
Succeeded:          100 (100.00%)
Errors:             NONE
Total Time:         15.75 secs
Throughput:         6.35 Requests/sec
Latency:            0.67 secs/Request

URL:                http://localhost/cgi-bin/dump.cgi
Total Requests:   100
Parallel Agents: 5
Succeeded:          100 (100.00%)
Errors:             NONE
Total Time:         106.73 secs
Throughput:         0.94 Requests/sec
Latency:            5.22 secs/Request

URL:                http://localhost/perl/dump.cgi
Total Requests:   100
Parallel Agents: 5
Succeeded:          100 (100.00%)
Errors:             NONE
Total Time:         38.07 secs
Throughput:         2.63 Requests/sec
Latency:            1.84 secs/Request

URL:                http://localhost/bogus
Total Requests:   100
Parallel Agents: 5
Succeeded:          0 (0.00%)
Errors:             File Not Found (100)
Total Time:         12.86 secs
Throughput:         7.77 Requests/sec
Latency:            0.57 secs/Request
```

Wir lassen's krachen - mit einer abgeleitete Klasse



Wahrscheinlich hat sich der eine oder andere bislang schon gewundert — ``Wieso ist das erzeugte Mutter-Objekt vom Typ *MyParallelAgent*?'' Der Grund dafür ist, daß es *LWP::Parallel::UserAgent* nicht gestattet, direkt Messungen der *Latency* vorzunehmen, also der Zeitspanne zwischen dem Absetzen eines (einzelnen!) Requests und dessen Abarbeitung. In weiser Voraussicht hat der Entwickler von *LWP::Parallel::UserAgent* jedoch eine Hintertür eingebaut: Objekte vom Typ *LWP::Parallel::UserAgent* rufen *vor* jedem Einzel-Request die Methode *on_connect* auf und springen nach getaner Arbeit je nach Erfolgsstatus *on_return* oder *on_failure* an. Diese Methoden läßt die Implementierung in *LWP::Parallel::UserAgent* bewußt leer — abgeleitete Klassen dürfen sie überschreiben und zusätzliche Funktionalität integrieren.
Das Mutterobjekt ruft *on_connect*, *on_return* und *on_failure* mit drei zusätzlichen Parametern auf: Neben der für die objektorientierte Programmierung mit Perl obligatorischen Objektreferenz kommen

- mit *$request* eine Referenz auf das *HTTP::Request*-Objekt,
- mit *$response* eine Referenz auf das *HTTP::Response*-Objekt und
- mit *$entry* eine Referenz auf ein von *LWP::Parallel:UserAgent* intern verwendetes Objekt vom Typ *LWP::Parallel:UserAgent::Entry*

als Parameter herein. In *pounder.pl* definieren die Zeilen 17 bis 51 eine neue, von *LWP::Parallel::UserAgent* abgeleitete Klasse *MyParallelAgent*, die folglich alles kann, wozu *LWP::Parallel::UserAgent* imstande ist. Der *@ISA*-Array aus Zeile 19 legt die Vererbungshierarchie fest. Die in der abgeleiteten Klasse überschriebene Methode *on_connect* bringt das Kunststück fertig, die Startzeit des jeweiligen Requests in einer Instanzvariablen für die spätere Latency-Berechnung abzulegen. Da das Mutter-Objekt die Startzeit *aller* Einzel-Requests speichern muß, speichert es die Einzel-Startzeiten in einem Hash *__start_times*, den sie mit der als letzten Parameter nach *on_connect* hereingereichten Referenz *$entry* indiziert, da diese Variable für jeden Einzel-Request eindeutig ist.
Der Name *__start_times* enthält deshalb zwei Unterstriche, damit es nicht zu unbeabsichtigten Überlap-

pungen mit eventuell in *LWP::Parallel::UserAgent* schon definierten gleichnamigen Instanzvariablen kommt.

$self->{__start_times}->{$entry} enthält also für jeden Request eine für das *Time::HiRes*-Modul taugliche Startzeit.

Die *on_return*-Methode addiert im Erfolgsfall am Ende eines Requests die inzwischen verstrichene Zeit zu einer Instanzvariablen des Mutter-Objekts: *__latency_total*, einer Fließkommazahl, die einen Sekundenwert für alle bislang ausgeführten Requests festhält.

on_failure verhält sich analog — da im Fehlerfall genau derselbe Ablauf erwünscht ist, ruft die überschriebene *on_failure*-Methode einfach *on_return* mit allen übergebenen Parametern auf.

Um die Definition der Klasse *MyParallelAgent* abzuschließen und mit dem eigentlichen Skript anzufangen, steht *package main* in Zeile 54.

Abbildung 1 zeigt Meßwerte für verschiedene URLs. Die erste Testserie holte ständig die statische Seite *http://localhost/index.html* und der Server lieferte mit mehr als 6 Requests pro Sekunde ein respektables Ergebnis. Für ein CGI-Skript, das noch dazu das *CGI.pm*-Modul einbindet, wird's schon sehr langsam: Die zweite Testreihe zeigt, daß der Server nur noch einen Requests pro Sekunde schafft und es etwa fünf Sekunden dauert, bevor der Anwender eine Antwort bekommt. Die dritte Testreihe spricht mit *http://localhost/perl/dump.cgi* dasselbe Skript an, nur daß dieses diesmal mit dem Apache-Beschleuniger *mod_perl* ausgeführt wird - das Ergebnis: Fast dreimal so schnell. Daß *pounder.pl* auch Fehlerfälle korrekt behandelt, zeigt die vierte Testserie: Für den nicht existierenden URL *http://localhost/bogus* meldet es korrekt 100 mal den Fehler *File Not Found*.

Weitere Einflüsse

Bei Performance-Messungen spielt natürlich auch die Netzwerkverbindung eine wichtige Rolle: Kommen die übertragenen Daten nicht schnell genug durch die Leitung, spiegelt das Meßergebnis letztlich die Bandbreite der Leitung wider und nicht die Serverleistung. Laufen, wie bei den durchgeführten Tests, Client und Server auf ein und demselben Rechner, zieht auch der Client nicht unerheblichen Saft aus der CPU. Zwar hält sich dies beim *LWP::Parallel::UserAgent*-Client in Grenzen, da dieser alle parallelen Verbindungen mit nur einem einzigen Prozeß und dem guten alten *select*-Trick kontrolliert, doch sollten für exakte Messungen Client und Server auf getrennten

LINUX USER GROUPS

Im folgenden die Liste der uns bekannten Linux User Groups im Deutschsprachigen Raum in Kurzfassung. Änderungen und Updates bitte der Redaktion (redaktion@linux-magazin.de) mitteilen (Name, Beschreibung, Treffpunkt, Adresse, Ansprechpartner, Homepage, Email, Tel, Fax, Mitgliederzahl,...).

Aachener Linux User Group (ALUG) http://aachen.heimat.de/alug/

Linux User Group Augsburg (LUGA) http://bsmux.rz.uni-augsburg.de/luga

Linux User Group Austria http://www.luga.or.at

Berliner Linux User Group (BeLUG) http://www.belug.org

Linux User Group Bochum (LugBo) http://www.ping.de/sites/lecter/lug.html

Linux Anwendergruppe der Universität Bochum
http://www.ruhr-uni-bochum.de/linux/

Linux User Group Bodensee (LLUGB) http://llugb.amsee.de/

Bonner Linux User Group (BoLUG) http://www.bonn.linux.de

Linux User Group Braunschweig http://www.tu-bs.de/initiativen/LUG

Chemnitz Linux User Group (CLUG) http://www.tu-chemnitz.de/linux/

Linux Users Group Darmstadt http://www.swb.de/personal/runner/dalug.html

Linux User Group Dortmund (LUGRUDO) lugrudo@outerspace.de

Linux Users Group Duisburg (LUG-DUI) http://mti26.mti.uni-duisburg.de

Düsseldorf Linux User Group (DLUG) http://www.hsp.de/~dlug/

Essener Linux-Stammtisch (ELiSta) http://www.uni-essen.de/initiative/elista/

Freiburger Linux User Group (FLUG) http://www.freiburg.linux.de

IN-Kompetent e.V. Fulda http://www.rhoen.de/

Linux User Grup Giessen (LUGG) http://lugg.tg.fh-giessen.de

Linux User Group Kassel (LUGK) jochen.hein@delphi.central.de

Unix-Gruppe der Hamburger MH e.V.
http://swt-www.informatik.uni-hamburg.de/~1willamo/hmh.html

Linux User Group Hannover (LUGH) http://www.han.de/lugh/

Projektgruppe Linux der UNIX-AG Kaiserslautern
http://www.unix-ag.uni-kl.de/~linux/

Karlsruher Linux User Group (KaLUG) http://www.karlsruhe.linux.de

Linux-Workshop Köln http://www.uni-koeln.de/themen/linux/

Leipziger Linux-Stammtisch http://rzaix340.rz.uni-leipzig.de/~linux/

Lüneburger Unix Stammtisch (LUSt) morningdew2@geocities.com

Münchner Linux Usergroup (MLUG)
http://www.muc.de/~cygnus/mlug/mlug.html

Linux Stammtisch Münster (MueSLi)
http://hottemax.uni-muenster.de/~grover/muesli.html

Linux User Group Nürnberg (LUGNü) http://www.linux.noris.de/

Oldenburger Linux Stammtisch (OLiS)
http://www.Infodrom.North.DE/Linux/Stammtisch/

Linux User Group Osnabrück (LUGO) http://www.lugo.de

Linux User Group Paderborn (LUG-PB) http://www.bowle.de/~lug-pb/

Linux User Group Saar (LUGSaar) http://www.lugs.linux.de/

UNIX-AG Siegen http://www.si.unix-ag.org

Linux User Group Stormarn
http://www.geocities.com/SiliconValley/Bay/7117/

Linux User Group Switzerland http://www.lugs.ch

Linux User Group Traunstein (LugTra) http://www.spotlight.de/lug/

Linux User Group Trier TriLUG http://www.trilug.fh-trier.de/

Linux User Group Tübingen (LUGTÜ) http://www.linux.de/lug/tuebingen/

Linux User Group TU Wien (LLL) lll@radawana.cg.tuwien.ac.at

Linux User Group Wolfsburg (LUGWo)
http://www.escape.de/user/laura/lug_wolfsburg.html

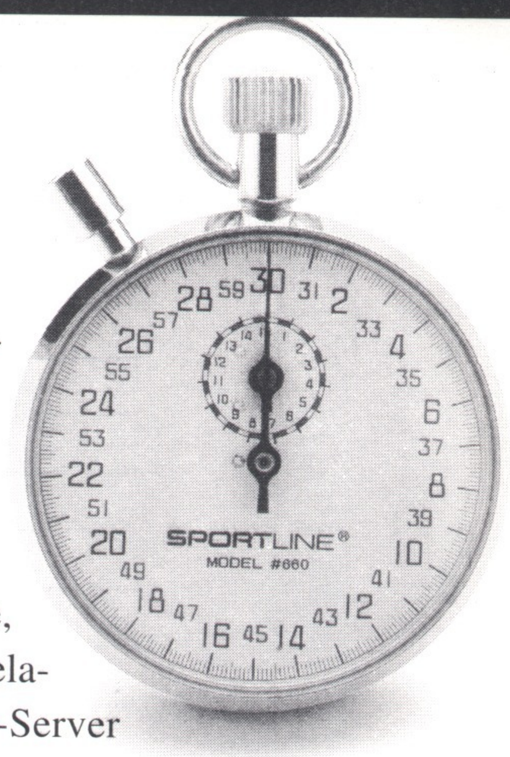
Brainstorm Computer-Club Würzburg (BraCCWü)
http://www.wuerzburg.de/brainstorm/ ???

Listing 2: pounder.pl

```
1 #!/usr/bin/perl -w
2
3 use LWP::Parallel::UserAgent;
4 use Time::HiRes qw(gettimeofday tv_interval);
5
6 ###
7 # Configuration
8 ###
9 $nof_parallel_connections = 7;
10 $nof_requests_total       = 100;
11 $url = „http://localhost/index.html“;
12 $timeout                  = 10;
13
14 ######################################################
15 # Derived Class for latency timing
16 ######################################################
17 package MyParallelAgent;
18
19 @ISA = qw(LWP::Parallel::UserAgent);
20
21 ###
22 # Is called when connection is openend
23 ###
24 sub on_connect {
25     my ($self, $request, $response, $entry) = @_;
26     $self->{__start_times}->{$entry} =
27          [Time::HiRes::gettimeofday];
28 }
29
30 ###
31 # Are called when connection is closed
32 ###
33 sub on_return {
34     my ($self, $request, $response, $entry) = @_;
35
36     my $start = $self->{__start_times}->{$entry};
37
38     $self->{__latency_total} +=
39     Time::HiRes::tv_interval($start);
40 }
41
42 sub on_failure {
43     on_return(@_);   # Same procedure
44 }
45
46 ###
47 # Access function for new instance var
48 ###
```

Listing 2: pounder.pl

```
49 sub get_latency_total {
50     return shift->{__latency_total};
51 }
52
53 ######################################################
54 package main;
55 ######################################################
56
57 ###
58 # Init parallel user agent
59 ###
60 $ua = MyParallelAgent->new();
61 $ua->agent(„pounder/1.0“);
62 $ua->max_req($nof_parallel_connections);
63 $ua->redirect(0);     # No redirects
64
65 ###
66 # Register all requests
67 ###
68 foreach (1..$nof_requests_total) {
69     my $request = HTTP::Request->new('GET', $url);
70     $ua->register($request);
71 }
72
73 ###
74 # Launch processes and check time
75 ###
76 $start_time = [gettimeofday];
77 $results = $ua->wait($timeout);
78 $total_time = tv_interval($start_time);
79
80 ###
81 # Requests all done, check results
82 ###
83 $succeeded      = 0;
84 foreach $entry (values %$results) {
85     my $response = $entry->response();
86
87     if($response->is_success()) {
88       $succeeded++; # Another satisfied customer
89     } else {
90           # Error, save the message
91       $response->message(„TIMEOUT“)
92           unless $response->code();
93       $errors{$response->message}++;
94     }
95 }
96
```



Listing 2: pounder.pl

```
97 ###
98 # Format errors if any from %errors
99 ###
100 $errors =
101    join(',', map „$_ ($errors{$_})", keys %errors);
102 $errors = „NONE" unless $errors;
103
104 ###
105 # Format results
106 ###
107 @P = (
108 „URL" => $url,
109
110 „Total Requests" => „$nof_requests_total",
111
112 „Parallel Agents" => $nof_parallel_connections,
113
114 „Succeeded" =>
115    sprintf(„$succeeded (%.2f%%)\n",
116        $succeeded * 100 / $nof_requests_total),
117
118 „Errors" => $errors,
119
120 „Total Time" =>
121     sprintf(„%.2f secs\n", $total_time),
122
123 „Throughput" =>
124     sprintf(„%.2f Requests/sec\n",
125         $nof_requests_total / $total_time),
126
127 „Latency" =>
128     sprintf(„%.2f secs/Request",
129             $ua->get_latency_total() /
130               $nof_requests_total),
131 );
132
133 ###
134 # Print out statistics
135 ###
136 format STDOUT =
137 @<<<<<<<<<<<<<<< @*
138 „$left:",          $right
139 .
140
141 while(($left, $right) = splice(@P, 0, 2)) {
142     write;
143 }
```

Maschinen mit guter Netzwerkverbindung laufen.

Noch eine Bitte

Und noch eine persönliche Bitte, meine lieben Leser: *pounder.pl* belastet einen angesprochenen Web-Server erheblich — tut mir den Gefallen und testet damit nur Eure eigenen Installationen. Ich denke, wer genügend Grips besitzt, das Skript zu starten und anzupassen, ist auch ein guter Netizen. Daran denken: Wir sind alle gute Freunde! Bis zum nächsten Mal!

DER AUTOR

Michael Schilli arbeitet als Web-Engineer für America Online Inc., San Mateo. Er ist Autor des im Juni bei Addison-Wesley erscheinenden Buches „GoTo Perl 5" und unter *mschilli1@aol.com* oder *http:// mission.base.com/mschilli* zu erreichen.

Autorenliste

Georg Basse
Marco Budde
Robert Gehring
Andreas Gutowski
Patricia Jung
Timo Kanera
Bernhard Kuhn
Martin von Löwis
Patrick Murmann
Oliver Obst
Joachim Schaaf
Michael Schilli
Tom Schwaller
Christian Zankel

Service für DeLUG-Mitglieder

Monatsdiskette

Krypto-Info:

DFN-PCA Schlüssel: 2048/35DBF565 1997/04/16 DFN-PCA, CERTIFICATION ONLY KEY (Low-Level: 1997-1998) <not-for-mail>
Key fingerprint = 09 7C 09 19 D3 C3 86 DC 7A 30 15 11 12 95 8D E3

IN-Root-CA Schlüssel: 2048/19980101 SIGN EXPIRE:1999-12-31
Key fingerprint = B3 06 9A 8D 38 04 3C 75 41 32 EE DC 8B 7D 61 0D
http://www.in-ca.individual.net/
Hashwert des Ewigen Logfiles vom 02.07.1998 15:13:05
sha1=f7d171f5dd06e042cd8b48b0434c2121f82a334e
https://www.iks-jena.de/mitarb/lutz/logfile/

Inserentenverzeichnis

IMPRESSUM

Informationen, Berichte und Neuigkeiten aus der Linux-Welt

Herausgeber: Rudolf Strobl
Redaktion:
Email: redaktion@linux-magazin.de
Post: **Linux - Magazin Verlag**
Stefan-George-Ring 23
81929 München
Tel.: 089/99315-310
Fax: 089/99315-329
WWW: http://www.linux-magazin.de
ISSN: 1432 - 640 X
Chefredakteur: Tom Schwaller, tom.schwaller@linux-magazin.de (ts)
Coverdesign: Tilo Wondollek, TW-Design
Layout : M &. S Art&Work
Mitarbeiter: siehe Autorenliste
Vertriebsmarketing: info@linux-magazin.de
Aboverwaltung: Monika Scheck
Tel: 089/99315-310
Email: info@linux-magazin.de
Anzeigen: Agentur Neumann
Kalvarienbergstraße 31
Postfach 1352
87503 Immenstadt
Tel: 08323 / 98100
Fax: 08323 / 963226
Email: ThomasNeumann@t-online.de

Anzeigenpreise: Es gilt die Anzeigenpreisliste vom 01.12.1997

Das Linux - Magazin erscheint monatlich.
Einzelheft: **DM 9,80**
Preis des Jahresabonnements:
Incl. DeLUG-Mitgliedschaft **DM 180,-**
Ohne Mitgliedschaft **DM 103,-**
(Ausland: 115,-)

Für Schüler und Studenten 20% Ermäßigung (Vorlage Schüler- oder Studentenausweis)