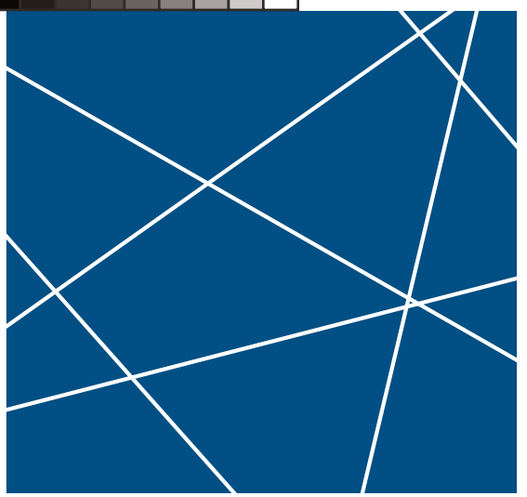

Was wäre wenn? Mathematische Lastsimulation mit Perl

Berechenbare Performance

Wer nicht bei einem Monitoring stehen bleiben will, das nur feststellt, ob ein Dienst verfügbar ist oder nicht, für den stellt sich schnell die Frage: Was tun mit den gesammelten Performancedaten? Man kann sie interpolieren und mit statistischen Mitteln Trends ableiten, klebt damit allerdings an den aktuellen Voraussetzungen, die man in die Zukunft fortschreiben muss. Oder man simuliert auf dieser Datengrundlage, wie sich beispielsweise eine Aufstockung der Hardware oder auch Laständerungen auswirken würden. Wie das geht, demonstriert dieser Beitrag. Neil J. Gunther

Drei aufeinander folgende Prozesse machen Performance-Management aus: Monitoring, Analyse und Modellierung (**Abbildung 1**). Das Monitoring sammelt die Daten, die Analyse erkennt in ihnen wiederkehrende Muster, und das Modeling sagt auf dieser Grundlage künftige Ereignisse wie etwa Ressourcenengpässe voraus. PDQ (Pretty Damn Quick) ist ein Queueing-Analysetool in Gestalt eines Perl-Moduls, das dazu dient, solche Vorhersagen zu ermöglichen.

Einführung

Eine passende Monitoring-Lösung ist ausgewählt, installiert und konfiguriert – jetzt läuft sie im Produktivbetrieb. Das aber ist nicht etwa das Ende, sondern ein neuer Anfang und Ausgangspunkt. Erfolgreiches Performance-Management umfasst nämlich mindestens drei Phasen: Das Performance-Monitoring, die Performance-Analyse und das Performance-Modeling (**Abbildung 1**).

Diese Phasen sind eng miteinander verknüpft. So besteht eine wesentliche Voraussetzung zunächst darin, die Leistung zu messen und Performancedaten zu sammeln. Ohne Daten lassen sich die Leistungseigenschaften der zu überwachenden Systeme und Applikationen nicht quantifizieren. Dies ist auch die Aufgabe der Monitoring-Phase.

Allerdings ist das Monitoring auf sich alleine gestellt so sinnlos wie das bloße Starren auf die tänzelnden Zeiger des Armaturenbretts eines Autos. Um die Lage richtig einschätzen zu können, muss man durch die Windschutzscheibe schauen, um andere Fahrzeuge in der Nähe zu erkennen. Mit anderen Worten: Wer sich ausschließlich auf das Monitoring verlässt, der erhält lediglich einen kurzfristigen Eindruck des Systemverhaltens (**Abbildung 2**).

Der Blick aus dem Fenster eröffnet dagegen die Fernsicht. Doch je weiter weg ein Beobachtungsobjekt ist, desto schwieriger lässt sich mit Bestimmtheit sagen, wie wichtig es einmal werden könnte.

Um aus Beobachtungswerten später Prognosen ableiten zu können, muss man die Leistungsdaten aus der Überwachungsphase mit Zeitstempeln versehen und sie in einer Datenbank speichern. Darauf baut die nächste Phase, die Performanceanalyse auf, die den Admin in die Lage versetzt, die gewonnenen Daten aus einer historischen Perspektive zu betrachten, um in ihnen Muster und Trends zu erkennen.

Auf das Performance-Modeling baut die Leistungsvorhersage auf, die Phase des Performance-Managements, die einen in die Lage versetzt, aus dem Fenster und in die Zukunft zu blicken. Genau wie beim Wetterbericht (um ein weiteres Gleichnis zu bemühen) benötigt man dafür zusätzliche Tools, mit deren Hilfe man die Daten so aufbereitet, dass sie in Leistungsmodelle eingehen können. Dazu ist etwas Mathematik nötig. Aber es ist ja auch so gut wie unmöglich, das Wetter ohne Instrumente vorherzusagen, nur indem man auf das Rascheln der Blätter im Wind hört.

Statistik versus Warteschlange

Es gibt zwei klassische Ansätze für das Performance-Modeling: die statistische Datenanalyse und das Warteschlangenmodell. Beide Methoden schließen sich gegenseitig nicht aus. Die Unterschiede lassen sich wie folgt zusammenfassen: Die statistische Datenanalyse, eine Aufgabe, die jede Buchhaltung kennt, basiert auf der Berechnung von Trends in den Rohdaten. Statistiker entwickelten viele schlaue Techniken und Tools im Laufe der Jahre, und ein Großteil dieser Intelligenz ist in Form von Open-Source-

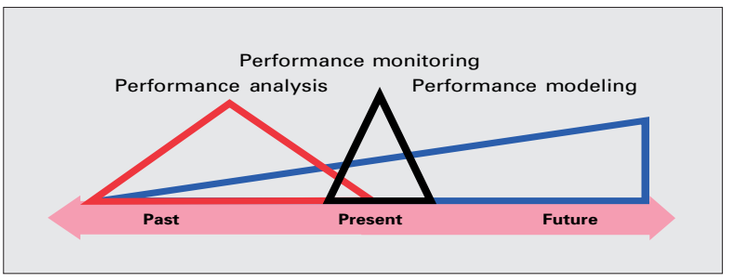

Abbildung 1: Die drei Phasen des Performance-Managements – Monitoring, Analyse und Modeling – sind eng miteinander verknüpft.

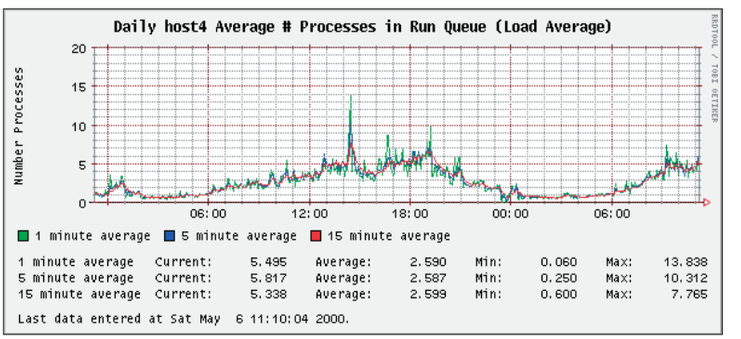

Abbildung 2: Ein 24-Stunden-Protokoll der durchschnittlichen Auslastung, das die Orca-Tools für Linux (1) auf eine Zeitachse projizieren.

Abbildung 3: Kunden stehen in einem Lebensmittelmarkt Schlange.

Paketen für das statistische Modeling erhältlich. Ein Beispiel für ein mächtiges freies Tool dieser Art ist R (2).

Dieser Ansatz ist jedoch dadurch begrenzt, dass er sich ausschließlich auf bestehende Daten stützt. Wenn die Zukunft jedoch (angenehme oder unangenehme) Überraschungen bereithält, die sich aus den aktuellen Daten nicht erkennen lassen, leidet die Zuverlässigkeit der Vorhersage. Und wer sich etwa mit der Börse beschäftigt, der weiß, dass dies ständig geschieht.

Warteschlangenmodelle sind von diesen Beschränkungen nicht betroffen. Das liegt daran, dass sie per Definition mit Hilfe des Queueing-Paradigmas vom realen System abstrahieren. Allerdings verursacht diese Vorgehensweise einen größeren Aufwand als die Trendanalyse der Rohdaten, und sie setzt voraus, dass die Queueing-Abstraktion das echte System genau nachbildet. Je stärker das Modell von der Wirklichkeit abweicht, desto höher ist die Ungenauigkeit der Vorhersagen. Wie dieser Artikel hoffentlich demonstriert, ist es aber bei weitem nicht so schwer, Warteschlangenmodelle zu gestalten, wie man aus den bisherigen Ausführungen vielleicht vermuten würde. In Wirklichkeit ist es oft erstaunlich einfach.

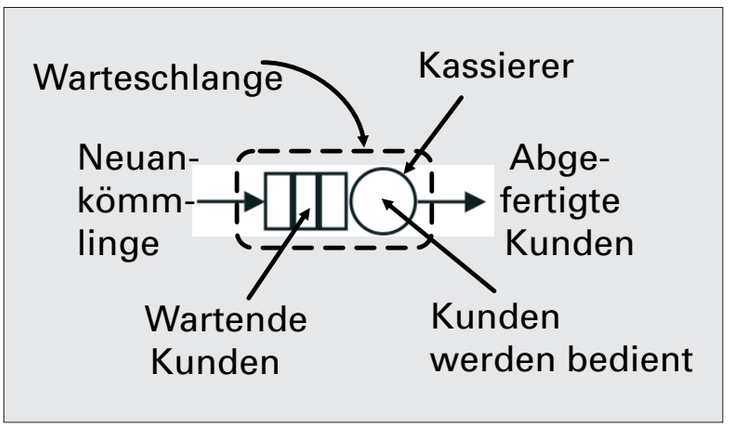

Abbildung 4: Komponenten einer symbolischen Warteschlange.

Marschroute

Zu Anfang dieses Beitrags soll das bekannte Beispiel der Warteschlange im Lebensmittelgeschäft ein paar grundlegende Queueing-Konzepte erklären. Hier gibt es an jeder Kasse eine einfache Warteschlange. Danach erweitert der Autor das fundamentale Konzept so, dass sich damit die Skalierbarkeit einer dreischichtigen E-Commerce-Anwendung vorhersagen lässt.

Gegen Ende des Artikels rücken realitätsnahe Erweiterungen der vorgestellten Leistungsmodelle und praktische Ratschläge für den Aufbau von Leistungsmodellen ins Blickfeld. Alle Beispiele verwenden Perl und das Open-Source-Queueing-Analysetool Pretty Damn Quick (PDQ), das der Autor zusammen mit Peter Harding pflegt. Es steht unter (6) zum Download bereitsteht. Die aktuelle Release von PDQ unterstützt den Aufbau von Modellen in C, Perl und Python; die nächste Version von PDQ soll außerdem noch Java und PHP supporten.

Warum Warteschlangen?

Buffer und Stacks sind in Computersystemen allgegenwärtig. Beim Buffer handelt es sich um eine Warteschlange, bei der die Reihenfolge des Eintreffens von Anforderungen die Reihenfolge ihrer Abarbeitung diktiert. Man spricht hier auch von FIFO (first-in, first-out) oder FCFS (first come, first served). Im Gegenzug dazu bedient ein Stack Anforderungen in LIFO-Reihenfolge (last-in, first-out); es handelt sich um eine LCFS-Warteschlange (last-come, first served).

Tabelle 1: Interessante Leistungsmetriken

Symbol	Metrik	PDQ
λ	Ankunftrate	Input
S	Bedienzeit	Input
N	User-Last	Input
Z	Denkzeit	Input
R	Verweilzeit	Output
$\mathcal{R}$	Antwortzeit	Output
X	Durchsatz	Output
ρ	Auslastung	Output
Q	Warteschlangenlänge	Output
N	Optimale Last	Output

Unter Linux ist der History-Buffer eine bekannte Warteschlange, welche die kürzlich aufgerufenen Shell-Befehle speichert. Genau wie der History-Buffer, ist jede physikalische Implementierung durch die finite Menge an Speicherplatz (ihre Kapazität) beschränkt. Theoretisch kann eine Warteschlange jedoch eine unendliche oder unbegrenzte Kapazität besitzen, so wie das auch bei PDQ der Fall ist.

Eine Warteschlange ist eine gute Abstraktion für gemeinsam genutzte Ressourcen. Ein sehr bekanntes Beispiel ist die Kasse im Supermarkt. Diese Ressource besteht aus Aufträgen (den Menschen, die Schlange stehen) sowie einer Bedienstation (der Kassiererin). Wenn man mit dem Einkaufen fertig ist, will man so schnell wie möglich den Laden verlassen, das ist das Performanceziel. Man kann dieses Ziel auch so formulieren, dass es darauf ankommt, möglichst wenig Zeit in der Schlange zu verbringen – man spricht hier von der Verweilzeit (R).

Nachdem sich ein Kunde für eine bestimmte Kasse entschieden hat und sich anstellt, besteht seine Verweilzeit aus zwei Komponenten: zum einen aus der Zeit, die er in der Warteschlange verbringt bevor er zur Kasse gelangt, und zum anderen aus der Zeit für die Bedienung durch den Kassierer, welche dieser benötigt, um die Einkäufe einzugeben, das Geld anzunehmen und herauszugeben.

Geht man nun davon aus, dass jede Person in der Warteschlange mehr oder weniger die gleiche Menge an Artikeln im Einkaufswagen hat, dann kann man erwarten, dass sich die Bedienzeiten pro Kunde im Schnitt angleichen. Darüber hinaus steht die Länge der Warteschlange offensichtlich im direkten Bezug zur Anzahl der Kunden. Wenn der Laden fast leer ist, wird die durchschnittliche Wartezeit um einiges kürzer sein als zu Stoßzeiten.

Die Abstraktion der Warteschlange (**Abbildung 4**) bietet ein leistungsfähiges Paradigma, mit dessen Hilfe sich (unter anderem) die Leistung von Computersystemen und Netzwerken ermitteln lässt. Ihr besonderer Vorzug ist, dass sie die ansonsten unterschiedlichen Leistungsdaten, die die Monitoring-Tools liefern, in einem Modell zusammenfasst.

Dieser Artikel nimmt immer wieder Bezug auf eine der Beziehungen zwischen den Metriken in **Tabelle 1** und zwar auf die Beziehung zwischen der Verweilzeit (R), der Servicezeit (S) und der Ankunftsrate (λ):

$$R = \frac{S}{1 - \lambda S} \quad (1)$$

Man kann Gleichung (1) als sehr einfaches Performancemodell betrachten. Die Eingaben für das Modell stehen auf der rechten Seite, die Ausgaben auf der linken. Nach genau demselben Schema funktionieren auch die Berechnungen mit PDQ. Durch dieses einfache Modell sieht man sofort, dass bei geringem Publikumsverkehr, die zum Passieren der Kasse benötigte Zeit (die Verweilzeit) ausschließlich aus der eigenen Servicezeit besteht. Wenn keine weiteren Personen eintreffen ($\lambda = 0$), dann fällt nur die Zeit an, die man selbst benötigt, um die Waren eingeben zu lassen und zu bezahlen.

Wenn das Geschäft jedoch stark frequentiert ist, sodass für das Produkt $\lambda S \rightarrow 1$ gilt, dann steigt auch die Verweilzeit sehr stark an. Das rührt daher, dass sich die Länge der Warteschlange aus der Gleichung

$$Q = \lambda R \quad (2)$$

berechnet. Mit anderen Worten: Die Verweilzeit steht im direkten Bezug zur Warteschlangenlänge mal Eintreffrate und umgekehrt.

Gleichung (2) stellt außerdem den Bezug zu den Überwachungsdaten her. Die Daten für die durchschnittliche Auslastung in **Abbildung 2** sind in Wirklichkeit Echtzeitwerte, gemessen über relative kurze Zeitabschnitte, beispielsweise eine Minute. Für die Berechnung der Warteschlangenlänge (Q) wird dagegen der Durchschnittswert über die gesamte Messdauer von 24 Stunden verwendet. Will man sich das bildlich vergegenwärtigen, so kann man sich Q als Höhe eines imaginären Rechtecks vorstellen, das die gleiche Fläche hat wie die Kurve der Überwachungsdaten über demselben 24-Stunden-Zeitabschnitt.

Eine dazu analoge Beziehung gilt, wenn man die Wartezeit aus der Verweilzeit (R) in (2) ausklammert:

$$\rho = \lambda S \quad (3)$$

Mit anderen Worten: Ersetzt man auf der rechten Seite der Gleichung die Verweilzeit R durch die Servicezeit S, dann entspricht auf der linken Seite die Ausgabemenge der Auslastung in **Tabelle 1**. ▶

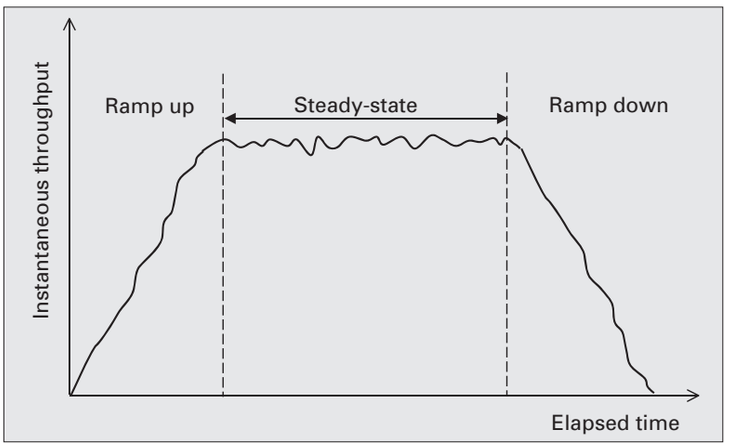

Abbildung 5: Durchsatzmessungen im stabilen Zustand.

Geschichte der Warteschlangentheorie

Die mathematische Theorie der Warteschlangen ist noch sehr jung; tatsächlich gibt es sie seit weniger als 100 Jahren. Agner Erlang hat im Jahre 1917 das erste formale Warteschlangenmodell entwickelt, um die Leistung des Telefonsystems (des Internets der damaligen Zeit) zu analysieren. Seine Aufgabe bestand darin, für Amtsgespräche aus Kopenhagen die Puffergröße für Vermittlungen festzulegen. Die heutige Terminologie bezeichnet dieses Modell als Einzel-Warteschlangenmodell.

Einer der nächsten wesentlichen Schritte in der Entwicklung der Warteschlangentheorie folgte 1957, als James Jackson die ersten formalen Lösungen für die Berechnung eines Netzwerks oder einer Kette aus Warteschlangen entwickelte. Dieses Ergebnis blieb 20 Jahre lang rein akademisch, bis man schließlich den Bezug zur Implementierung des Internets erkannte. Das Warteschlangenmodell erwies sich dafür als zutreffend mit einer Abweichung von weniger als fünf Prozent. 1967, etwa fünfzig Jahre nach dem ersten Modell von Erlang, setzte ein Doktorand namens Allan Scherr ein Warteschlangenmodell ein, um die Leistung der CTSS- und Multics-Time-Sharing-Computersysteme zu berechnen, die in vielen Beziehungen die Vorläufer der Unix- und letztendlich damit auch der Linux-Rechner waren.

In den späten 70er und frühen 80er Jahren haben einige neue theoretische Entwicklungen zu vereinfachten Algorithmen für die Berechnung der Leistungsmetriken von Warteschlangensystemen geführt. Diese Algorithmen sind auch in Tools wie PDQ implementiert. Bei den neuesten Entwicklungen in der Warteschlangentheorie geht es um Modelle für den so genannten selbstähnlichen oder fraktalisierten Paketverkehr im Internet. Diese Begriffe sprengen zwar den Rahmen dieses Artikels; wenn Sie sich aber für weiterführende Informationen interessieren, so sei Ihnen Kapitel 10 von **(4)** empfohlen.

Annahmen in PDQ-Modellen

Eine der grundlegenden Annahmen in PDQ ist, dass sowohl die durchschnittliche Zwischenankunftszeit wie die durchschnittliche Bedienedauer beide statistisch zufällig sind. Mathematisch gesehen bedeutet das, dass jede Ankunft und jedes Bedienereignis zu einem Poisson-Prozess gehört. Dann entspricht die Dauer dem durchschnittlichen oder Mittelwert einer exponentialen Wahrscheinlichkeitsstreuung. Erlang hat festgestellt, dass sich der Verkehr im Telefonnetz tatsächlich dieser Anforderung entsprechend verhält. Es gibt Methoden **(5)**, mit deren Hilfe sich feststellen lässt, wie gut gegebene Monitoring-Daten diese Anforderung erfüllen.

Wenn diese Monitoring-Daten wesentlich von den Bedingungen des Exponentials abweichen, ist es vielleicht sinnvoller, auf einen ereignisbasierten Simulator auszuweichen, wie beispielsweise SimPy **(3)**, mit dessen Hilfe sich eine größere Bandbreite an Wahrscheinlichkeitsstreuungen berücksichtigen lässt. Das Problem dabei ist, dass die Programmierung und das Debuggen mehr Zeit in Anspruch nehmen (bei jeder Simulation geht es gleichzeitig um die Programmierung); außerdem benötigen Sie länger, um sicherzustellen, dass Ihre Ergebnisse statistisch gültig sind.

Ein weiterer Aspekt, der bei jeder wie auch immer gearteten Vorhersage stört, sind die Fehler, die grundlegende Annahmen des Modells verursachen. Annahmen in Modellen verursachen

Tabelle 2: PDQ-Funktionen für Abbildung 4

Physikalisch	Warteschlange	PDQ-Funktion
Kunde	Arbeitslast	CreateOpen()
Kassierer	Bedienknoten	CreateNode()
Buchhaltung	Bedienzeit	SetDemand()

tatsächlich systematische Fehler im Vorhersageprozess, und daher sollte man sich alle PDQ-Ergebnisse in Wirklichkeit als Streuung plausibler Werte vorstellen. Was außerdem oft unberücksichtigt bleibt, ist die Tatsache, dass jede Quantifizierung fehlerbehaftet ist. Davon sind auch die Monitoring-Daten betroffen – entgegen der landläufigen Meinung sind sie nicht gottgegeben. Aber wer kennt schon den exakten Fehlerbereich seiner Monitoring-Daten?
Weil es sich bei allen PDQ-Performance-Ein- und -Ausgaben um Durchschnittswerte handelt, ist unbedingt sicherzustellen, dass es sich dabei um zuverlässige Mittelwerte handelt. Die lassen sich beispielsweise in der stabilen Phase (steady state) eines Lasttests messen (**Abbildung 5**).
Der durchschnittliche Durchsatz im stabilen Zustand (X) für eine bekannte Benutzerlast (N) ergibt sich, indem man Messungen über einen längeren Zeitraum T nimmt und alle Anfahr- oder Herunterfahrzeiten aus den Daten eliminiert. Eine nominelle Zeit T kann beispielsweise fünf bis zehn Minuten betragen, je nach Anwendung. Auch bei Industriestandard-Benchmarks wie SPEC und TPC gilt die Anforderung, dass alle gemeldeten Durchsatzergebnisse aus einem stabilen Zustand gemessen stammen.

Eine einfache Warteschlange in PDQ

Die Beziehung zwischen dem Szenario im Lebensmittelmarkt und den PDQ-Funktionen fasst **Tabelle 2** zusammen.

Nach diesem Schema kann man leicht ein einfaches Modell der Kasse in einem Lebensmittelmarkt nachbilden, wie das folgende Listing für die Perl-Variante von PDQ zeigt (**Listing 1**).
Im PDQ-Code (unten) befinden sich die Input-Werte für die Ankunftsrate (λ) und die Bedienzeit (S):

$$\lambda = 3 / 4 \quad (4)$$

$$S = 1.0 \quad (5)$$

Wendet man nun die metrische Beziehung (2) an, ergibt sich die Auslastung der Kasse wie folgt:

$$\rho = \frac{3 * 1}{4} = 0.75 \quad (6)$$

Analog dazu ergibt sich für die Verweildauer durch Anwenden der metrischen Beziehungen (1) und (2):

$$R = \frac{1.0}{1 - \frac{3}{4} * 1.0} = 4.0\,\text{seconds} \quad (7)$$

So erfährt man, dass die der Verweildauer an der Kasse gleich vier durchschnittliche Bedienzeiten ist, wenn der Kassierer zu 75 % ausgelastet ist. Die sich daraus ergebende durchschnittliche Warteschlangenlänge ist: ▶

Listing 1: Kassenmodell in PDQ

```
01 #! /usr/bin/perl
02 # groxq.pl
03 use pdq;
04 #------------------------ INPUTS
   --------------------
05 $ArrivalRate = 3/4; # Kunden je Sekunde
06 $ServiceRate = 1.0; # Kunden je Sekunde
07 $SeviceTime = 1/$ServiceRate;
08 $ServerName = "Cashier";
09 $Workload = "Customers";
10 #------------------------ PDQ Model
   -------------------
11 # Interne PDQ-Variable initialisieren
12 pdq::Init("Grocery Store Checkout");
13 # Die von PDQ::Report() genutzten Einheiten
   anpassen
14 pdq::SetWUnit("Cust");
15 pdq::SetTUnit("Sec");
16 # PDQ Bedienknoten erstellen (KassiererIn)
17 $pdq::nodes = pdq::CreateNode($ServerName, $pdq::
   CEN, $pdq::FCFS);
18 # Die PDQ-Aufgabe mit Ankunftsrate erstellen
19 $pdq::streams = pdq::CreateOpen($Workload,
   $ArrivalRate);
20 # Bedienrate je Kunden an der Kasse definieren
21 pdq::SetDemand($ServerName, $Workload,
   $SeviceTime);
22 #------------------------ OUTPUTS
   --------------------
23 # Das PDQ-Modell lösen
24 pdq::Solve($pdq::CANON);
25 pdq::Report(); # einen vollständigen PDQ-Report
   generieren
```

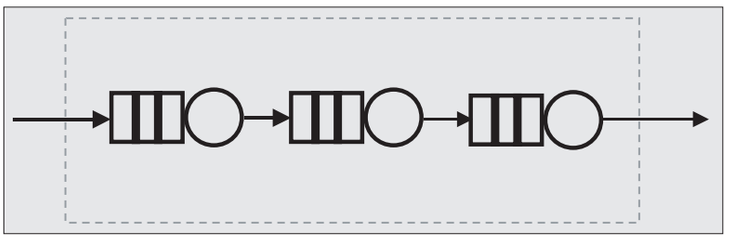

Abbildung 6: Ein offenes System mit drei Queueing-Phasen.

8

$$Q = \frac{3}{4 * 4.0} = 3.0 \text{ customers}$$

Aus Platzgründen kann hier lediglich die Ausgabeseite des generischen PDQ-Reports für dieses Modell gezeigt werden (Kasten PDQ-Resultate). Die berechneten PDQ-Werte stimmen genau mit den theoretischen Vorhersagen für den Durchsatz (X = λ), die Auslastung (ρ), die Warteschlangenlänge (Q) und die Verweildauer (R) überein.

Nachdem die fundamentalen Begriffe bekannt sind und PDQ als Tool für die Berechnungen eingeführt ist, lassen sich diese Werkzeuge auch auf Computerprobleme anwenden, etwa auf die Vorhersage der Leistung individueller Hardwareressourcen wie der Ausführungswarteschlange der CPU (siehe Kapitel 4 in (5)) oder eines Festplattengerätetreibers. Die meisten Lehrwerke zur Warteschlangentheorie bieten Beispiele auf diesem Niveau.

Wichtiger für die Vorhersage der Leistung von echten Computersystemen ist allerdings die Fähigkeit, den Workflow zwischen mehreren Warteschlangenressourcen abbilden zu können, also die Interaktion zwischen Anforderungen, die gleichzeitig an Prozessoren, Festplatten und das Netzwerk gestellt werden. Die nächsten Abschnitte zeigen, wie man diese Aufgabe mithilfe von PDQ erfüllt.

PDQ-Resultate

```
01 ***************************************
02 ***** Pretty Damn Quick REPORT ******
03 ***************************************
04 *** of : Sun Feb 4 17:25:39 2007  ***
05 *** for: Grocery Store Checkout   ***
06 *** Ver: PDQ Analyzer v3.0 111904 ***
07 ***************************************
08 ****** RESOURCE Performance   *******
09 Metric Resource Work Value Unit
10 --------- ------ ---- ----- ----
11 Throughput Cashier Customers 0.7500 Cust/Sec
12 Utilization Cashier Customers 75.0000 Percent
13 Queue Length Cashier Customers 3.0000 Cust
14 Residence Time Cashier Customers 4.0000 Sec
```

Warteschlangensysteme

Anforderungen, die aus einer Warteschlange in eine andere fließen, entsprechen einer Warteschlangenkette oder einem Warteschlangennetz .

Erfasst man anstelle der Anzahl der Anforderungen lediglich die Ankunftsrate (λ), spricht man von einem offenen System oder Kreis (**Abbildung 6**). Ein Beispiel für eine Situation dieser Art, die nicht aus der Welt der Computer stammt und sich durch das offene Modell in **Abbildung 6** abbilden ließe, wäre das Boarding am Flughafen. Die drei Phasen sind: Warten am Gate, Schlangestehen auf der Fluggastbrücke, um ins Flugzeug zu gelangen, und zum Schluss Schlangestehen im Gang des Flugzeugs beim Boarding, während Passagiere weiter vorn Platz nehmen.

Die durchschnittliche Antwortzeit (R) ergibt sich aus den in jeder Warteschlangenphase verbrachten Zeiten, also der Summe der drei Verweildauern. Im Computerkontext ließe sich das auf eine dreistufige Webapplikation anwenden, von der lediglich die Rate der HTTP-Requests bekannt ist.

Eine weitere Art von Warteschlangensystemen wird durch eine finite Menge (N) von Kunden oder Anforderungen charakterisiert. Genau diese Situation ergibt sich bei einem Lasttest. Eine finite Menge von Client-Lastgeneratoren sendet Anforderungen an das Testsystem, wobei keine weiteren Anforderungen von außerhalb

Tabelle 3: Die aus Tabelle 4 ermittelten Bedienzeiten

N	Sws	Sas	Sdb
1	0.0088	0.0021	0.0019
2	0.0085	0.0033	0.0012
4	0.0087	0.0045	0.0007
7	0.0095	0.0034	0.0005
10	0.0097	0.0022	0.0006
20	0.0103	0.0010	0.0006
Avg	0.0093	0.0028	0.0009

in das isolierte System eintreten können. (Open-Source Last- und Stresstesttools finden Sie unter (7).)
Darüber hinaus wirkt eine Art Rückkopplungsmechanismus, weil zu jeder Zeit nicht mehr als eine Anforderung unbearbeitet bleiben darf. Mit anderen Worten: Jeder Lastgenerator sendet erst dann eine weitere Anforderung, wenn die vorherige abgearbeitet worden ist. In der Sprache der Warteschlangentheorie geht es hier um einen geschlossenen Warteschlangenkreis.

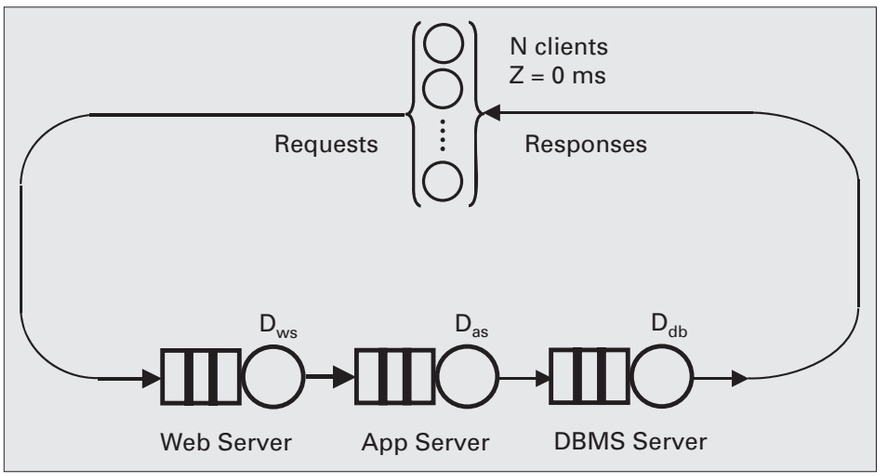

Abbildung 7: Ein geschlossener Warteschlangenkreis mit drei Warteschlangenphasen in Verbindung mit einer speziellen Wartephase (oben), die N Client-seitigen Lastgeneratoren mit einer durchschnittlichen Bedenkzeit von Z entspricht.

E-Commerce-Applikation in PDQ

Dieser Abschnitt zeigt, wie sich das PDQ-Modell des geschlossenen Systems aus Abbildung 7 einsetzen lässt, um die Durchsatzleistung der dreistufigen E-Commerce-Architektur aus Abbildung 8 vorherzusagen. Wegen der hohen Kosten solcher Installationen sind Lasttests mit einem kleineren Modell der späteren Produktivumgebung die Regel. In unserem Beispiel soll jeweils ein Server jede Stufe abbilden. Ein Testsystem dieser Art eignet sich sehr gut für Leistungsmessungen, wie sie für die Parametrisierung eines PDQ-Modells vonnöten sind.
Den Durchsatz misst man in HTTP-Gets/Sekunde (GPS) und die entsprechende Antwortzeitleistung in Sekunden (s). Aus Platzgründen konzentriert sich dieser Artikel ausschließlich auf die Nachbildung der Durchsatzleistung. Wer sich für Antwortzeitmodelle interessiert, findet dazu Einzelheiten in (5).
Die kritischen Lasttestergebnisse für dieses Beispiel fasst Tabelle 4 zusammen. Leider wurden die Daten nicht mit einer gleich bleibenden Inkrementierung der User-Last erzeugt, was für eine korrekte Leistungsanalyse nicht gerade optimal ist, aber dennoch kein unüberwindliches Problem darstellt. X und R sind Systemmetriken auf der Clientseite. Die Auslastung wurde eigenständig durch Performance-Monitore auf jedem der lokalen Server gemessen. Die gemessene Auslastung (ρ) und der Durchsatz (X) in Tabelle 4 lassen sich in eine umgestellten Version der Gleichung (3) einsetzen, um die entsprechenden Bedienzeiten für jede Stufe zu ermitteln:

$$S = \frac{\rho}{X} \qquad (9)$$

Der nächste Abschnitt zeigt, wie der Durchschnittswert der ermittelten Bedienzeiten (die letzte Zeile in Tabelle 3) einzusetzen ist, um das PDQ-Modell zu parametrisieren. ▶

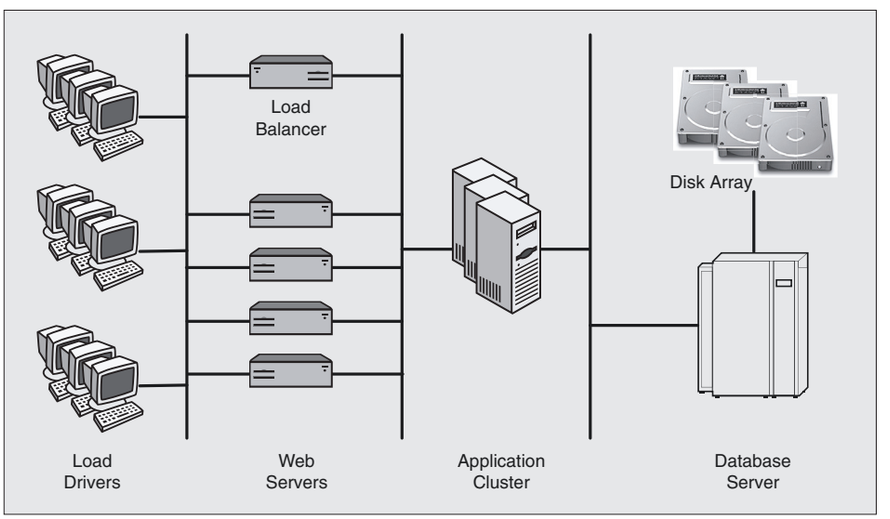

Abbildung 8: Multitier-E-Commerce-Anwendung.

Tabelle 4: Gemessene Leistungsdaten in jeder Stufe

N (Clients)	X (GPS)	R (s)	S_{ws} (%)	S_{as} (%)	S_{db} (%)
1	24	0.039	21	8	4
2	48	0.039	41	13	5
4	85	0.044	74	20	5
7	100	0.067	95	23	5
10	99	0.099	96	22	6
20	94	0.210	97	22	6

Die Bedienzeiten für jede Last zeigt Tabelle 3 zusammen mit den Durchschnittswerten in der letzten Zeile der Tabelle.

$$X_{\max} = \frac{1}{\max(S_{ws}, S_{as}, S_{db})} \quad (10)$$
$$= \frac{1}{0{,}0093}$$
$$= 107.53 \text{ GPS}$$

Der höchstmögliche Durchsatz wird, wie sich hier zeigt, in unmittelbarer Nähe des errechneten optimalen Belastungspunkts N^* erreicht (siehe dazu **Tabelle 1**):

Naives PDQ-Modell

Der erste Versuch, die Leistungscharakteristik von **Abbildung 7** nachzubilden, stellt jeden Anwendungsserver einfach als eigenständigen PDQ-Knoten unter Einsatz der durchschnittlichen Bedienzeiten aus **Tabelle 3** dar. In Perl::PDQ wird die Parametrisierung der Warteschlangenknoten so, wie in **Listing 2** zu sehen, codiert.

Ein Diagramm des Durchsatzes, den dieses erste, sehr einfache Modell vorhersagt, zeigt die **Abbildung 9**. Man sieht auf den ersten Blick, dass das naive PDQ-Modell einen Durchsatz prophezeit, der im Vergleich mit den real gemessenen Daten der Testumgebung zu schnell absättigt.

Allerdings teilt uns PDQ ebenfalls mit, dass der bestmögliche Durchsatz für dieses System – auf Basis der gemessenen Bedienzeiten aus **Tabelle 3** – etwa 100 GPS beträgt. Diese Leistung wird durch einen Ressourcenengpass begrenzt (die Warteschlange mit der längsten durchschnittlichen Bedienzeit), das ist im Beispiel der Frontend-Webserver. Im muss man lasten, dass der maximale Durchsatz nicht über einen Wert steigen kann, der sich aus der Beziehung (10) ergibt.

$$N^* = \frac{S_{ws} + S_{as} + S_{db} + Z}{\max(S_{ws}, S_{as}, S_{db})} \quad (11)$$
$$= \frac{0.0093 + 0.0028 + 0.0009 + 0.0}{0.0093}$$
$$= 1.40 \text{ clients}$$

Der liegt bei 1.40 und nicht 20 Clients. Ändern sich die Bedienzeiten in Zukunft, beispielsweise durch eine neue Release der Anwendung, kann sich der Ressourcenengpass verschieben; das PDQ-Modell kann dann die Auswirkung Durchsatz und Antwortzeiten vorhersagen.

Offensichtlich ist es wünschenswert, den gesamten Datenbestand besser nachzubilden als das mit dem naiven PDQ-Modell gelang. Eine einfache Methode, um der schnellen Sättigung des Durchsatzes zu begegnen, wäre, die Bedenkzeit auf einen Wert ungleich Null zu setzen:

```
Z > 0:
$think = 28.0 * 1e-3; # freier Parameter
...
pdq::Init($model);
$pdq::streams = pdq::CreateClosed(
```

Listing 2: Parametrisierung des PDQ-Modells

```
01 pdq::Init($model);
02 $pdq::streams = pdq::CreateClosed($work, $pdq::
   TERM, $users,
03 $think);
04 ...
05 # eine Warteschlange für jede der drei Stufen
   erstellen
06 $pdq::nodes = pdq::CreateNode($node1, $pdq::CEN,
   $pdq::FCFS);
07 5
08 $pdq::nodes = pdq::CreateNode($node2, $pdq::CEN,
   $pdq::FCFS);
09 $pdq::nodes = pdq::CreateNode($node3, $pdq::CEN,
   $pdq::FCFS);
10 ...
11 # Zeitbasis sind Sekunden, die in Millisekunden
   ausgedrückt werden
12 pdq::SetDemand($node1, $work, 9.3 * 1e-3);
13 pdq::SetDemand($node2, $work, 2.8 * 1e-3);
14 pdq::SetDemand($node3, $work, 0.9 * 1e-3);
```

```
$work, $pdq::TERM, $users,
$think);
```

Auf diese Art werden neue Anforderungen verlangsamt ins System injiziert. Man spielt hier mit der Bedenkzeit, als wäre sie ein freier Parameter. Der positive Wert von Z = 0.028 Sekunden stimmt nicht mit den Einstellungen überein, die beim Lasttest tatsächlich verwendet wurden, aber er kann einen Hinweis darauf geben, in welcher Richtung nach einem verbesserten PDQ-Modell zu suchen ist.
Wie **Abbildung 10** zeigt, verbessert die positive Bedenkzeit das Durchsatzprofil entscheidend.

$$N^* = \frac{0.0093 + 0.0028 + 0.0009 + 0.0028}{0.0093} \quad (12)$$

$$= 4.41 \text{ clients}$$

Der Trick mit der Bedenkzeit verrät uns, dass weitere Latenzen existieren, die in den Stresstestmessungen keine Berücksichtigung fanden. Die positive Bedenkzeit erzeugt eine Latenz, sodass sich die Round-Trip-Time der Anforderung verlängert. Als Nebenwirkung verringert sich der Durchsatz bei niedriger Last. Aber in der Praxis betrug die Bedenkzeit während der realen Lastmessungen Null! Wie löst man dieses Paradoxon?

Versteckte Latenzen berücksichtigen

Als nächsten Trick fügt man dem PDQ-Modell aus **Abbildung 11** Dummy-Knoten hinzu. Allerdings gibt es Bedingungen, die von den Bedienanforderungen der virtuellen Knoten zu erfüllen sind. Die Bedienanforderung eines jeden Dummy-Knotens ist so zu wählen, dass sie die Bedienanforderung des Engpassknotens nicht übersteigt.
Darüber hinaus ist die Anzahl der Dummy-Knoten so zu wählen, dass die Summe der Serviceanforderungen einen Wert von Rmin = R(1) nicht übersteigt, sofern keine Konkurrenz auftritt, das heißt für eine Einzelanforderung. Wie sich herausstellt, lassen sich diese Bedingungen erfüllen, wenn man zwölf einheitliche Dummy-Knoten einführt, von denen jeder eine Serviceanforderung von 2,2 ms aufweist. Die Änderungen des entsprechenden PDQ-Codes sehen folgendermaßen aus:

```
use constant MAXDUMMIES => 12;
```

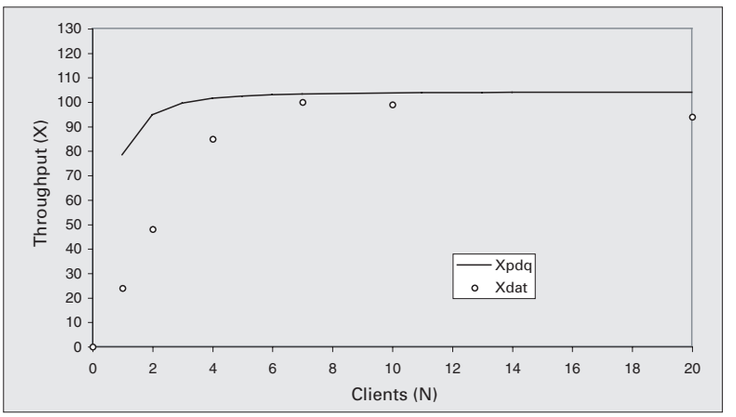

Abbildung 9: Naives PDQ-Durchsatzmodell

```
$think = 0.0 * 1e-3; #same as in test rig
$dtime = 2.2 * 1e-3; #dummy service time
# Dummy-Knoten mit Bedienzeiten erstellen
for ($i = 0; $i < MAXDUMMIES; $i++) {
$dnode = "Dummy" . ($i < 10 ? "0$i" :
"$i");
$pdq::nodes = pdq::CreateNode($dnode,
$pdq::CEN, $pdq::FCFS);
pdq::SetDemand($dnode, $work, $dtime);
}
```

Man beachte, dass die Bedenkzeit wieder auf Null zurückgesetzt ist. Die Ergebnisse dieser Änderungen am PDQ-Modell finden sich in **Abbildung 12**. Das Durchsatzprofil ist immer noch für geringe Lasten (N < N*) passend, wobei

$$N^* = \frac{0.0093 + 0.0028 + 0.0009 + 12(0.0022)}{0.0093} \quad (13)$$

$$= 4.24 \text{ clients}$$

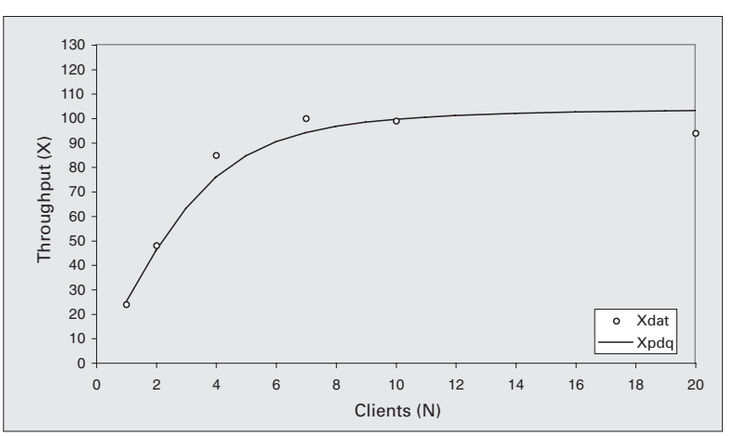

Abbildung 10: Durchsatzmodell mit positiver Bedenkzeit.

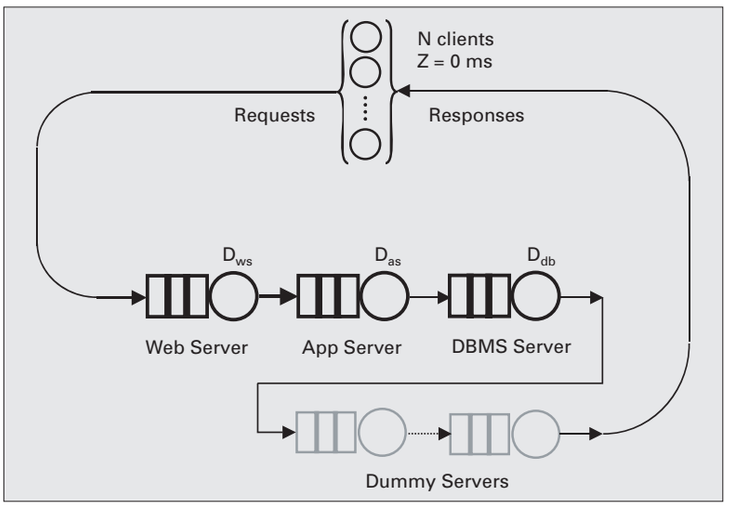

Abbildung 11: Dummy-Knoten bilden versteckte Latenzen ab.

aber der Nachbesserung oberhalb von N* bedarf.

Lastabhängige Server

Bestimmte Aspekte des physikalischen Systems wurden nicht gemessen, sodass die Validierung des PDQ-Modells schwer fällt. Bisher haben wir versucht, die Intensität der Arbeitslast durch Einführung einer positiven Bedenkzeit anzupassen. Die Einstellung von Z = 0.028 Sekunden beseitigte das Problem der schnellen Sättigung; gleichzeitig stimmt der Wert nicht mit dem Wert von Z = 0 Sekunden überein, der für die eigentlichen Messungen eingestellt wurde.

Durch die Einführung von Dummy-Warteschlangenknoten ins PDQ-Modell wurde das Modell für Szenarien mit geringer Last verbessert, aber dadurch wird dem in den Daten beobachteten Abfall des Durchsatzes nicht Rechnung getragen. Um diesen Effekt nachzubilden, können wir den Webserverknoten durch einen lastabhängigen Knoten ersetzen. Die allgemeine Theorie der lastabhängigen Server wird in (5) besprochen. In unserem Beispiel wenden wir einen etwas einfacheren Ansatz an. Aus der Bedienzeit (S_{ws}) in **Tabelle 4** erkennt man, dass sie nicht für alle Clientlasten konstant bleibt. Es wird also eine Methode benötigt, um diese Variabilität auszudrücken. Wenn man ein Diagramm der Messdaten für S_{ws} erstellt, lässt sich eine statistische Regressionsanpassung durch-

Listing 5: E-Commerce-Modell

```
01 #! /usr/bin/perl
02 # ebiz_final.pl
03 use pdq;
04 use constant MAXDUMMIES => 12;
05 # Hash AV pairs: (in vusers laden, durchsatz in
   gets/sec)
06 %tpdata = ( (1,24), (2,48), (4,85), (7,100),
   (10,99), (20,94) );
07 @vusers = keys(%tpdata);
08 $model = "e-Commerce Final Model";
09 $work = "ebiz-tx";
10 $node1 = "WebServer";
11 $node2 = "AppServer";
12 $node3 = "DBMServer";
13 $think = 0.0 * 1e-3; # wie beim testsystem
14 $dtime = 2.2 * 1e-3; # dummy-bedienzeit
15 # Header für benutzerspezifischen Report
16 printf("%2s\t%4s\t%4s\tD=%2d\n", "N", "Xdat",
   "Xpdq", MAXDUMMIES);
17 foreach $users (sort {$a <=> $b} @vusers) {
18 pdq::Init($model);
19 $pdq::streams = pdq::CreateClosed($work, $pdq::
   TERM, $users,
20 $think);
21 $pdq::nodes = pdq::CreateNode($node1, $pdq::CEN,
   $pdq::FCFS);
22 $pdq::nodes = pdq::CreateNode($node2, $pdq::CEN,
   $pdq::FCFS);
23 $pdq::nodes = pdq::CreateNode($node3, $pdq::CEN,
   $pdq::FCFS);
24 # Zeitbasis in Sekunden “ in Millisekunden
   ausgedrückt
25 pdq::SetDemand($node1, $work, 8.0 * 1e-3 *
   ($users ** 0.085));
26 pdq::SetDemand($node2, $work, 2.8 * 1e-3);
27 pdq::SetDemand($node3, $work, 0.9 * 1e-3);
28 # Dummy-Knoten mit entsprechenden Bedienzeiten
   erstellen ...
29 for ($i = 0; $i < MAXDUMMIES; $i++) {
30 $dnode = "Dummy" . ($i < 10 ? "0$i" : "$i");
31 $pdq::nodes = pdq::CreateNode($dnode, $pdq::CEN,
   $pdq::FCFS);
32 pdq::SetDemand($dnode, $work, $dtime);
33 }
34 pdq::Solve($pdq::EXACT);
35 printf("%2d\t%2d\t%4.2f\n", $users,
   $tpdata{$users},
36 pdq::GetThruput($pdq::TERM, $work));
37 }
```

führen, wie sie **Abbildung 13** zeigt. Die sich daraus ergebende Potenzgesetzgleichung lautet:

$$D_{ws}(N) = 8.0000 N^{0.0850} \quad (14)$$

Damit wird Node1 des PDQ-Modells wie folgt parametrisiert:

```
pdq::SetDemand($node1, $work, ➥
8.0 * 1e-3 * ($users ** 0.085));
```

Die angepasste Ausgabe des fertigen PDQ-Modells zeigt **Tabelle 3**. Sie zeugt von einer guten Übereinstimmung mit den gemessenen Daten für D = 12 Dummy-PDQ-Knoten.

Die Auswirkung auf das Durchsatzmodell lässt sich in **Abbildung 14** erkennen. Die mit Xpdq2 gekennzeichnete Kurve zeigt den vorhergesagten übersteuerten Durchsatz auf Basis des lastabhängigen Servers für das Webfrontend, und die Vorhersagen liegen locker innerhalb des Fehlerbereichs der gemessenen Daten.

In diesem Fall bringt es wenig, PDQ für die Vorhersage einer Last einzusetzen, die oberhalb der gemessenen Last von N = 20 Clients liegt, weil der Durchsatz nicht nur gesättigt ist, sondern auch rückgängig. Nachdem nun ein PDQ-Modell existiert, das mit den Testdaten validiert wurde, kann man jetzt alle erdenklichen Was-wäre-wenn-Szenarien durchspielen.

Weitere Forschungen mit PDQ

Das vorherige Beispiel ist bereits ziemlich anspruchsvoll, und ähnliche PDQ-Modelle sind für die meisten Anwendungsfälle vollkommen ausreichend. Allerdings gibt es auch Situationen, in denen detailliertere Modelle erforderlich sein können. Zwei Beispiele für diese Szenarien sind multiple Server und multiple Aufgaben.

Multiple Server

Ein Szenario außerhalb der Computerwelt, das man mithilfe der Warteschlange mit multiplen Servern aus **Abbildung 15** nachbilden könnte, wäre das Schlangestehen in einer Bank oder einem Postamt. Im Kontext der Computer-Performance könnte **Abbildung 15** als einfaches Modell eines symmetrischen Mehrprozessorsystems dienen.

Weitere Informationen zu diesem Thema finden Sie in Kapitel 7 von **(5)**. Die Antwortzeit in (1) wird durch folgendes ersetzt:

$$R \simeq \frac{S}{1-\rho^m} \quad (15)$$

wobei ρ = λS, und *m* ist die integrale Anzahl von Servern. Technisch betrachtet handelt es sich um eine Annäherung, aber keine schlechte! Die genaue Lösung ist komplexer und lässt sich mit dem folgenden Perl-Algorithmus entdecken (**Listing 3**). ▶

Tabelle 5: Modellresultate

N	Xdat	Xpdq D=12
1	24	26.25
2	48	47.41
4	85	77.42
7	100	98.09
10	99	101.71
20	94	96.90

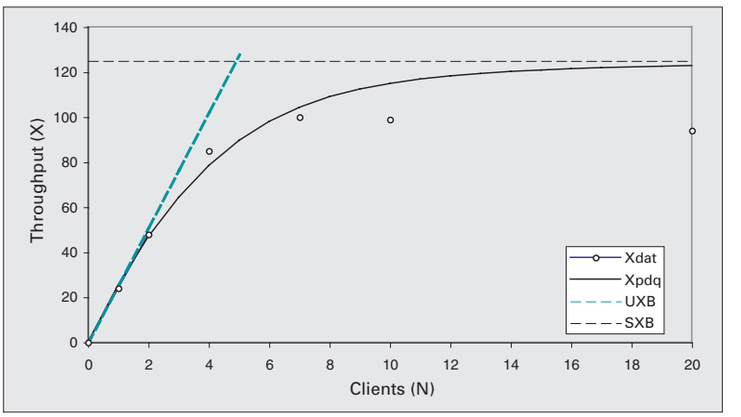

Abbildung 12: Durchsatz bei Z = 0 mit Dummy-Knoten

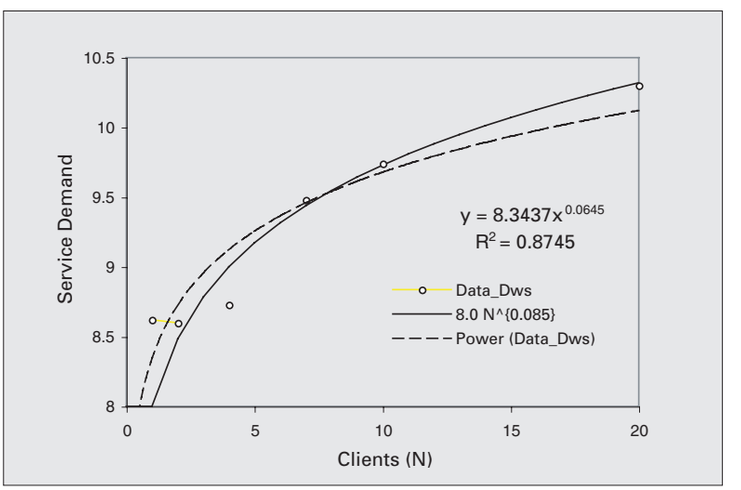

Abbildung 13: Regressionsanpassung der Webserver-Zeiten

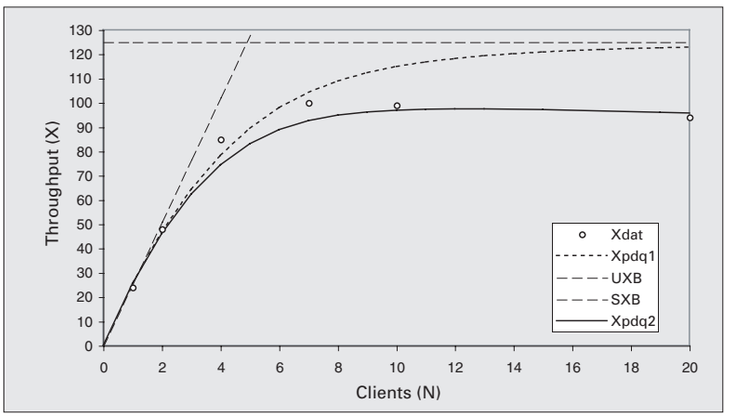

Abbildung 14: Modell des lastabhängigen Durchsatzes

Es handelt sich um genau das Warteschlangenmodell, das Erlang vor 100 Jahren entwickelt hat. Damals stellte jeder Server eine Hauptleitung im Telefonnetz dar.

Multiple Aufgaben

Was in Wirklichkeit sehr häufig vorkommt, ist, dass eine einzelne Ressource, etwa ein Datenbankserver, mit verschiedenen Transaktionstypen umgehen muss. Zum Beispiel kann der Kauf eines Flugtickets oder die Buchung eines Hotelzimmers im Internet ein halbes Dutzend unterschiedliche Transaktionen erfordern, bevor das Ticket ausgestellt oder das Zimmer endlich reserviert ist. Situationen dieser Art lassen sich wie folgt mit PDQ abbilden.

Man betrachte den einfacheren Fall von drei unterschiedlichen Transaktionstypen, die durch die Farben Rot, Grün und Blau gekennzeichnet werden. Jede dieser eingefärbten Aufgaben kann auf eine gemeinsam genutzte Ressource zugreifen, zum Beispiel einen Datenbankserver. Im Warteschlangenparadigma (**Abbildung 16**) wird jede der bunten Aufgaben durch die unterschiedlichen Bedienzeiten charakterisiert. Mit anderen Worten: Die rote Last erhält eine rote Bedienzeit, die grüne eine grüne Bedienzeit und so weiter. Für jede Farbe gilt auch eine eigene Ankunftsrate.

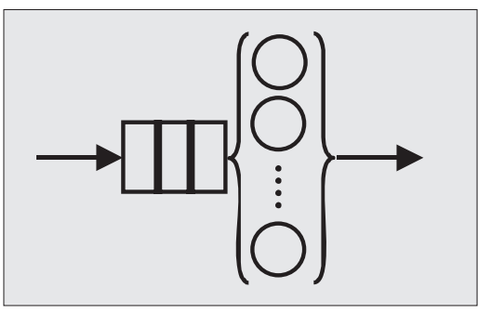

Abbildung 15: PDQ-Modell einer Multiserver-Warteschlange

Geht man davon aus, dass sich die Bedienzeiten unterscheiden, resultiert die tatsächliche Auswirkung auf die Warteschlange beim Eintreffen beispielsweise einer roten Anforderung nicht mehr allein aus der Anzahl von bereits wartenden Anforderungen (das trifft nur für eine „monochrome“ Aufgabe zu), sondern aus der Farbkombination der wartenden Anforderungen. In PDQ lässt sich **Abbildung 16** vielleicht so darstellen wie im **Kasten „Gemischte Workloads“**.

Natürlich ist der durch multiple Aufgaben generierte PDQ-Report komplexer aufgrund der vielen möglichen Interaktionen. Auf jeden Fall wirft er etwas Licht auf die vielfältigen Erweiterungsmöglichkeiten von PDQ, um realistische Computerarchitekturen abbilden zu können. Diese Thematik noch weiter auszuführen, würde den Rahmen sprengen, aber man findet weitere Details in **(5)**.

Richtlinien für den Einsatz von PDQ

Modelle jeder Art zu erstellen, ist teils Wissenschaft und teils Kunst, und daher ist es unmöglich, ein komplettes Regelwerk oder eine komplette Sammlung von Algorithmen bereitzustellen, die immer das richtige Modell liefern. Wie dieser Artikel illustriert, handelt es sich in Wirklichkeit um einen Prozess der ständigen Verbesserung. Erfahrung ist durch nichts zu ersetzen, und sie gewinnt man bekanntlich durch die ständige Wiederholung.

In diesem Sinne folgen nun einige Richtlinien, die unter Umständen helfen, wenn man PDQ-Modelle kreiert: .

- Keep it simple: Ein PDQ-Modell sollte so einfach wie irgend möglich sein, aber auch nicht einfacher. Es ist kaum zu vermeiden, dass man umso mehr Details in das PDQ-Modell stopfen möchte, je mehr man über die Systemarchitektur weiß. Das führt jedoch unausweichlich zu einer Überlastung des Modells.
- Eher die Streckenkarte als die U-Bahn im Hinterkopf behalten: Ein PDQ-Modell verhält sich zum Computersystem wie eine Karte der U-Bahn zum physikalischen U-Bahn-Netz. Eine U-Bahn-Karte ist eine Ab-

straktion, die kaum etwas mit der physischen Beschaffenheit des Netzes zu tun hat. Sie bietet gerade ausreichende Details, sodass man von Punkt A nach Punkt B kommt. Sie enthält jedoch keine überflüssigen Details wie die Höhe der Bahnhöfe über Normalnull, ja noch nicht einmal die Entfernung zwischen ihnen. Ein PDQ-Modell ist eine ähnliche Abstraktion.

- Das große Bild: Im Gegensatz zu vielen Aspekten der Computertechnologie, bei denen man große Mengen winziger Details aufnehmen muss, geht es beim PDQ-Modell darum zu entscheiden, wie viele Details man ignorieren kann!
- Suchen nach dem Operationsprinzip: Wer das Operationsprinzip nicht in weniger als 25 Worten beschreiben kann, versteht es wahrscheinlich nicht gut genug, um mit dem Aufbau eines PDQ-Modells zu beginnen. Das Operationsprinzip für ein Timesharing-Computersystem lässt sich beispielsweise wie folgt ausdrücken: Timesharing gibt jedem Benutzer die Illusion, dass er der einzige aktive Benutzer des Systems ist. Die Hunderte Zeilen Code in Linux, um Timeslicing zu implementieren, dienen lediglich dem Zweck, diese Illusion zu untermauern.
- Den schwarzen Peter weitergeben: Beim PDQ-Modeling geht es auch darum, die Verantwortung zu verteilen. Als Performanceanalyst muss man lediglich das Leistungsproblem aufdecken – dann tritt man beiseite und lässt die anderen es beheben.
- Wo anfangen? Oder „Spaß mit Bauklötzen“: Ein möglicher Ausgangspunkt für ein PDQ-Modell wäre ein Diagramm mit Funktionsblöcken. Das Ziel dabei ist, zu erkennen, wo in welcher Phase Zeit für die Verarbeitung der zu untersuchenden Aufgabe aufgewandt wird. Letztendlich wird jeder Funktionsblock in ein Warteschlangensubsystem umgewandelt. Dabei kann man zwischen der sequenziellen und parallelen Verarbeitung unterscheiden. Andere, ebenfalls auf Diagrammen basierende Techniken, beispielsweise UML-Diagramme, können außerdem nützlich sein.
- Input und Output: Beim Definieren eines PDQ-Modells ist es nützlich, eine Liste der Inputs (Messungen oder Schätzungen, die zum Parametrisieren des Modells eingesetzt werden) und Outputs (Zahlen, die sich durch die Berechnung des Modells ergeben) aufzustellen.
- Keine Bedienung, keine Warteschlange: Analog zur Regel fürs Restaurant: „Keine Schuhe, keine Bedienung!“ Nun ja, die Regel für PDQ-Modelle lautet: Keine Bedienung, keine Warteschlange. In PDQ-Modellen ist es sinnlos, Warteschlangenknoten zu erstellen, für die es keine gemessenen Bedienzeiten gibt. Wenn die Messungen des echten Systems keine Bedienzeit für einen zu modelllierenden PDQ-Knoten enthalten, dann lässt sich der Knoten nicht definieren.
- Bedienzeiten schätzen: Bedienzeiten lassen sich oft nur schwer direkt messen. Aber oft lässt sich die Bedienzeit aus anderen Leistungsmetriken ableiten, die sich leichter überwachen lassen. Siehe dazu **Tabelle 4**.
- Ändern der Daten: Wenn die Messungen nicht zum PDQ-Leistungsmodell passen, müssen die Messungen wahrscheinlich wiederholt werden. ▶

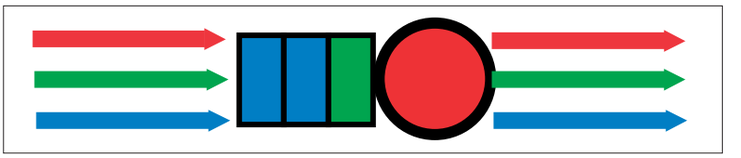

Abbildung 16: PDQ-Modell einer Warteschlange mit multiplen Aufgaben

Listing 3: »erlang.pl«

```
01 #! /usr/bin/perl
02 # erlang.pl
03 ## Input-Parameter
04 $servers = 8;
05 $erlangs = 4;
06 if($erlangs > $servers) {
07 print "Error: Erlangs exceeds servers\n";
08 exit;
09 }
10 $rho = $erlangs / $servers;
11 $erlangB = $erlangs / (1 + $erlangs);
12 for ($m = 2; $m <= $servers; $m++) {
13 $eb = $erlangB;
14 $erlangB = $eb * $erlangs / ($m + ($eb * $erlangs));
15 }
16 ## Ausgabe der Ergebnisse
17 $erlangC = $erlangB / (1 - $rho + ($rho * $erlangB));
18 $normdwtE = $erlangC / ($servers * (1 - $rho));
19 $normdrtE = 1 + $normdwtE; # Exact
20 $normdrtA = 1 / (1 - $rho**$servers); # ca.
```

Gemischte Workloads

```
01 $pdq::nodes = pdq::CreateNode("DBserver", $pdq::CEN, $pdq::
   FCFS);
02 $pdq::streams = pdq::CreateOpen("Red", $ArrivalsRed);
03 $pdq::streams = pdq::CreateOpen("Grn", $ArrivalsGrn);
04 $pdq::streams = pdq::CreateOpen("Blu", $ArrivalsBlu);
05 pdq::SetDemand("DBserver", "Red", $ServiceRed);
06 pdq::SetDemand("DBserver", "Grn", $ServiceGrn);
07 pdq::SetDemand("DBserver", "Blu", $ServiceBlu);
08 ...
```

- Offene oder geschlossene Warteschlange? Wenn man überlegt, welches Warteschlangenmodell anzuwenden ist, fragt man sich, ob die zu verarbeitende Menge an Anforderungen endlich ist. Lautet die Antwort „ja“ (und das sollte sie beispielsweise für eine Lasttestplattform immer tun), dann handelt es sich immer um ein geschlossenes Warteschlangenmodell. In allen anderen Fällen benutzt man am besten ein offenes Warteschlangenmodell.
- Messungen im stabilen Zustand: Die Dauer der Messung im stabilen Zustand sollte in einer Größenordnung liegen, die um den Faktor 100 größer ist als die längste auftretende Bedienzeit.
- Welche Zeiteinheit sollte man verwenden? Am besten benutzt man immer die Zeiteinheit des Messtools. Wenn das Tool also in Sekunden misst, dann verwendet man Sekunden; misst es Mikrosekunden, tut man es ihm gleich. Wer mehrere Datenquellen zugleich überwacht, der sollte stets zuerst alle Messwerte auf die gleichen Einheiten umrechnen, und erst danach mit der Modellierung beginnen.
- Aufgaben treten in der Regel in Dreiergruppen auf: In einem gemischten Aufgabenmodell (Multiclass-Streams in PDQ) vermeide man die Nutzung von mehr als drei gleichzeitigen Aufgabenstreams, wo immer das möglich ist. Davon abgesehen, dass der resultierende PDQ-Report ansonsten ganz bestimmt unhandlich ist, interessiert man sich in der Regel lediglich für die Interaktion zweier Aufgaben, das heißt für einen Vergleich von Aufgabenpaaren. Alles andere gehört zur dritten Aufgabe (auch unter dem Namen „Hintergrund“ bekannt). Wenn man nicht erkennen kann, wie dieses Problem praktisch zu lösen wäre, dann ist man wahrscheinlich noch nicht soweit, das PDQ-Modell erstellen zu können.

Fazit

Performance-Modeling ist eine anspruchsvolle Disziplin, die man am besten durch ständige Wiederholung trainiert. Ein Großteil der Bemühungen kreisen dabei immer wieder um die Erstellung und Validierung eines Modells der zu untersuchenden Umgebung und ihrer Anwendungen. Sobald das PDQ-Modell erst einmal validiert ist, muss es nicht immer wieder aufs Neue gebaut werden. Im Allgemeinen reicht dann etwas Tuning – und schon lassen sich Performance-Änderungen durch Hardwareupgrades oder durch neue Software berücksichtigen.
Die dreistufige E-Commerce-Applikation, die dieser Beitrag beispielhaft nachgebildet hat, liefert einen recht guten Ausgangspunkt, auf dem sich aufbauen lässt, um multiple Server und zusätzliche Aufgaben zu berücksichtigen.
Eines der erstaunlichsten Ergebnisse des PDQ-Modells ist die Tatsache, dass es den analysierenden Admin bestimmte Effekte – etwa versteckte Latenzen – erkennen lässt, die in den überwachten Daten für ihn sonst nicht ersichtlich waren.
Aber das vielleicht allerwichtigste Resultat des PDQ-Einsatzes sind gar nicht die Leistungsmodelle an sich, sondern es ist die Tatsache, dass der PDQ-Modellierungsprozess einen organisatorischen Rahmen für die Beurteilung aller Leistungsdaten liefert, in dem Erkenntnisse aus dem Monitoring bis hin zur Trendvorhersage zusammenfließen können. *(jcb)* ■■■

Infos

(1) Kenneth Hess, „Monitoring Linux performance with Orca“: (http://www.linux-magazine.com/issue/65/Linux_Performance_Monitoring_With_Orca.pdf)
(2) R, Open Source statistical analysis package: (http://www.r-project.org)
(3) SimPy, Open Source simulator written in Python: (http://sourceforge.net/projects/simpy/)
(4) N. J. Gunther, „Guerrilla Capacity Planning“, Springer-Verlag, 2007
(5) N. J. Gunther: „Analyzing Computer System Performance with Perl::PDQ“, Springer-Verlag, 2005
(6) PDQ Download: (http://www.perfdynamics.com/Tools/PDQcode.html)
(7) Linux-Stresstesttools: (http://www.opensourcetesting.org/performance.php)

Der Autor

Neil Gunther, M.Sc., Ph.D. ist ein international anerkannter Consultant und Gründer der Firma Performance Dynamics Company (http://www.perfdynamics.com).

Nach einer Ausbildung in theoretischer Physik nahm er verschiedene Forschungs- und Management-Aufgaben wahr, unter anderem an der San Jose State University und bei der NASA (Voyager- und Galileo- Missionen.)

Dr. Gunther ist Mitglied von AMS, APS, ACM, CMG, IEEE und INFORMS.