Moving Beyond Monitoring ... PDQ (Pretty Damn Quick)

Neil J. Gunther

Abstract

Performance management can be broken into three sequential processes: performance monitoring, performance analysis, and performance modeling. (Fig. 1) Monitoring is the theme of this issue, analysis refers to the capability of looking for patterns in monitored data that reside in a database, while modeling attempts to use monitored data to predict future events, such as resource bottlenecks. PDQ (*Pretty Damn Quick*) is a queueing analysis tool aimed at expediting the prediction process.

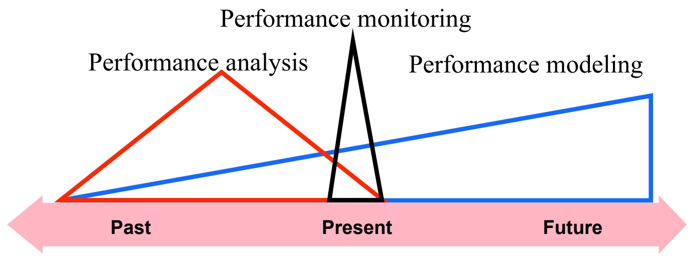


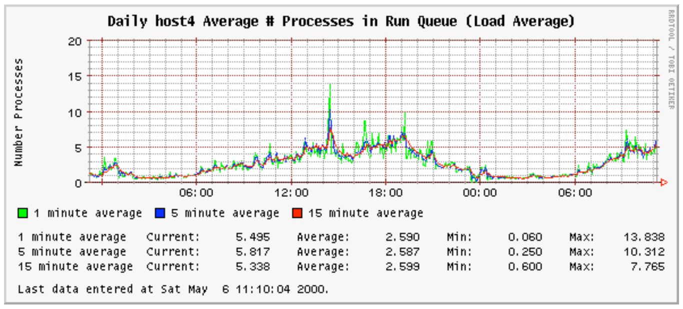
Figure 1: A schematic timeline showing the relationship between performance monitoring, analysis, and modeling

1 Introduction

Having selected and installed monitoring tools for your environment, you can go a lot further. There are at least three stages in successful performance management: performance monitoring, performance analysis, and performance modeling. (Fig. 1) Each of these stages is related like the layers of an onion.

The core requirement is to collect monitored performance data. Without that, the performance characteristics of systems and applications cannot begin to be quantified. That is the monitoring phase. But monitoring alone is akin to watching needles jitter on the dashboard of a car or the cockpit of an aircraft. To assess the future picture, it is important to look out of the window and see what is down the road or what other aircraft may be flying near you. The problem with just relying on monitoring alone is that it only provides a short-term view of system behavior (Fig. 2). Looking out the window provides the longer-range view, but the further away things are the more difficult it is to discern their importance.

In addition, the monitored performance data needs to be time-stamped and stored in a performance database. This is the performance analysis layer of the onion. Performance analysis enables you to review monitored data from an historical perspective to detect patterns and extract trend information.



```
1 minute average   Current:    5.495   Average:    2.590   Min:    0.060   Max:   13.838
5 minute average   Current:    5.817   Average:    2.587   Min:    0.250   Max:   10.312
15 minute average  Current:    5.338   Average:    2.599   Min:    0.600   Max:    7.765

Last data entered at Sat May  6 11:10:04 2000.
```

Figure 2: A 24 hour sample of monitored load-average data, e.g., from `procinfo`, displayed as a time series using Orca tools for Linux [1]

The last layer of the onion is performance modeling or performance prediction—the stage in performance management that allows you too look out the window and see ahead. Just like weather forecasting (to draw on another analogy), you need supplemental tools that can manipulate the data in the performance database to parameterize performance models. Afterall, it is next impossible to try and forecast the weather by simply listening to the leaves rustle in the wind.

There are two classic approaches to performance modeling: statistical data analysis and queueing models. Although these methodologies are not mutually exclusive, the distinction between them can be summarized as follows. Statistical data analysis, the kind of thing that is done in every accounting department, is based on modeling trends in the raw data. Many clever techniques and tools have been developed by statisticians over the years and much of this intelligence is available in open source statistical modeling packages like R [2]. The limitation of this approach, however, is that it is based solely on previous data. If the future holds some surprises (good or bad) that were not somehow foretold in the current data, the forecast will be unreliable. Those who play the stock market know that this happens all the time.

Queueing models, on the other hand, are not subject to these limitations. The reason is that queueing models, by definition, require that an abstraction of the real system be constructed using the queueing paradigm. The trade off is that this typically involves more work than just doing trend analysis on raw data, and it assumes that the queueing abstraction is a faithful representation of the real system. To the degree that it is not, its predictions will also be unreliable. As I attempt to show in this article, constructing queueing models is nowhere near as difficult as it may seem from this comparison. In fact, it is often stunningly simple.

We begin by reviewing some basic queueing concepts based on the familiar example of shopping in a grocery store. Each checkout stand is a simple queue. Next, we extend these fundamental concepts to predicting the scalability of a 3-tier e-commerce application. At the end of this article we consider some more realistic extensions to the performance models presented here and we offer some guidelines for building predictive performance models. All the examples are expressed in Perl using the open source queueing analyzer called *Pretty Damn Quick* (PDQ), which is maintained by the author and Peter Harding, and can be downloaded from `http://www.perfdynamics.com/Tools/PDQcode.html`. The current release of PDQ allows performance models to be built in *C*, Perl and python, while the next release will extend PDQ to Java and PHP.

2 Why Queues?

Buffers and stacks are ubiquitous forms of storage in computer systems and communication networks. A buffer is a type of *queue* where the order of servicing the request is determined by the order in which it arrives into the buffer. This is called FIFO (first-in, first-out) or FCFS (first come, first served) in queueing parlance. By contrast, a stack is serviced in LIFO (last-in, first-out) order, so it is a LCFS (last-come, first served) queue. In Linux the *history buffer* is a familiar queue for storing recently invoked shell commands. Like the history buffer, any physical implementation is generally constrained to a finite amount of storage or capacity. Theoretically, however, queues can have unlimited or unbounded capacity. This is the case in PDQ.

A queue is an abstraction used to represent a shared resource. Consider a very familiar resource; the checkout at a grocery store or a hypermarket. This resource comprises requests for service (the people waiting in line) and a service center (the cashier). Once they have completed their shopping, everyone in the store would like to get out of it as quickly as possible; that's a performance goal. That goal translates to spending the least time getting through the checkout; technically known as the *residence time* (R).

Figure 3: Customers queueing in a grocery store

Once you choose a particular checkout isle, your residence time consists of two components: the time you spend waiting in line to get to the cashier, plus the time it actually takes to get serviced by the cashier, i.e., have your groceries accounted and paid for.

If we assume for the moment that everyone in line (including you) has more or less the same amount of grocery items in their respective shopping carts, then their average service time will be the same. Moreover, the length of the waiting line will be directly related to the number of people waiting. When there are very few people in the store, your average waiting time will be shorter rather than when the store is very busy.

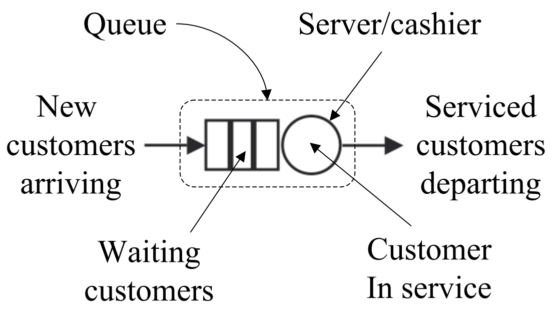


Figure 4: Components of a symbolic queue in Fig. 3

The queue abstraction (Fig. 4) offer a powerful paradigm for characterizing the performance of computer systems and networks (amongst other things) because they tie together the otherwise disparate performance metrics that are measured by monitoring tools.

Table 1: Performance Metrics of Interest

Symbol	Metric	PDQ
λ	Arrival rate	Input
S	Service time	Input
N	User load	Input
Z	Think time	Input
R	Residence time	Output
$\mathcal{R}$	Response time	Output
X	Throughput	Output
ρ	Utilization	Output
Q	Queue length	Output
N^*	Optimal load	Output

One of the relationships between the metrics in Table 1 that we shall draw on throughout this article, is that between the residence time (R), the service time (S) and the arrival rate (λ):

$$R = \frac{S}{1 - \lambda S} \tag{1}$$

We can think of (1) as a very simple performance model. Notice that the model *inputs* go on the right-hand side of (1), while the model output appears on the left-hand side. PDQ works exactly the same way.

This simple model tells us immediately that if traffic in the store is light, such that there are no other arrivals ($\lambda = 0$) at your checkout stand (Fig. 3), the time it takes you to get through checkout (your residence time) is simply the your own service time to have your groceries accounted and paid for.

On the other hand, if traffic in the store is heavy, such that the product $\lambda S \to 1$, the residence time climbs very rapidly. This follows from the fact that the queue length is given by:

$$Q = \lambda R \tag{2}$$

In other words, the residence time is directly related to the queue length by the arrival rate, and vice versa.

Equation (2) also provides contact with monitored data. The load-average data in Fig. 2 are actually *instantaneous* values sampled over relatively short time intervals, e.g., 1 minute. The queue length (Q) is time-averaged over the entire measurement period of 24 hours. More intuitively, Q corresponds to the height of an imaginary rectangle that has the same area as that under the monitored data curve over the same 24 hour period.

An analogous relationship also holds if the waiting is excluded from the residence time (R) in (2):

$$\rho = \lambda S\,. \tag{3}$$

In other words, if we replace R by the service time S on the right-hand side, the quantity on the left-hand side (the output) becomes equivalent to the utilization in Table 1.

The theory of queues is very young from a mathematical standpoint; less than 100 years, in fact. Agner Erlang developed the first formal queueing model in 1917 to analyze the performance of the Internet of his day—the telephone system. He was tasked with determining the buffer size for switches that routed trunk calls from Copenhagen, Denmark. Today, we call this a single queue model, and present more details about single queue models in Sections 4 and 10.

One of the next major advances in the theory of queues was in 1957, when James Jackson derived the first formal solutions for the performance of a network or circuit of queues, rather than just one queue. This result remained rather academic for another 20 years until its applicability to engineering the Internet was realized. His queueing model was accurate to within five percent of the measured performance.

In 1967, some fifty years after Erlang's first queueing model, a Ph.D. student, named Allan Scherr, used a queueing model to estimate the performance of the CTSS and Multics time-share computer system, which in many respects were the precursors to the UNIX and ultimately, the Linux operating systems.

By the late 1970's and early 1980's several new theoretical developments led to simplified algorithms for computing the performance metrics of queueing circuits and it is these algorithms that are incorporated in tools like PDQ.

The latest developments in queueing theory have to do with modeling so-called *self-similar* or fractalized packet traffic on the Internet. These concepts lie well beyond what we shall treat here, but the interested reader can explore them more in Chapter 10 of reference [4].

3 Assumptions in PDQ Models

One of the underlying assumptions embedded into PDQ is that the average duration between arrivals and the average duration of service periods, are both statistically randomized. Mathematically, this means that each arrival and service event belongs to a *Poisson* process. Then, the durations correspond to the average or mean of a negative *Exponential* probability distribution. Erlang found that telephonic traffic does conform to this requirement. There are methods [5] to check how well your monitored data conforms to this requirement.

If your monitored data differs very significantly from the Exponential requirement, it may be more prudent to resort to an event-based simulator e.g., SimPy [3], which is able to accommodate a broader class of probability distributions. The trade off is that this takes longer to set up and debug (all simulation is about programming), and it takes longer to validate that your simulation results are statistically valid.

Another aspect that often bothers people about *any* kind of prediction is the errors introduced by the modeling assumptions. While it is true that modeling assumptions do introduce systematic errors into predictions and all PDQ results should really be presented as a range of plausible values with their attendant errors, what is often not appreciated is that *everything* that is quantified introduces errors; including monitored data. Contrary to popular opinion, data is not divine. Do you know the error range on your data?

Since all PDQ performance inputs and outputs are averages, it is important to ensure that monitored data also represents reliable averages. One way to do this is with a controlled load-test platform, like that in Section 6. Such a platform is actually a workload simulator, and it is therefore important that all performance measurements be collected in *steady-state*. A steady-state average throughput

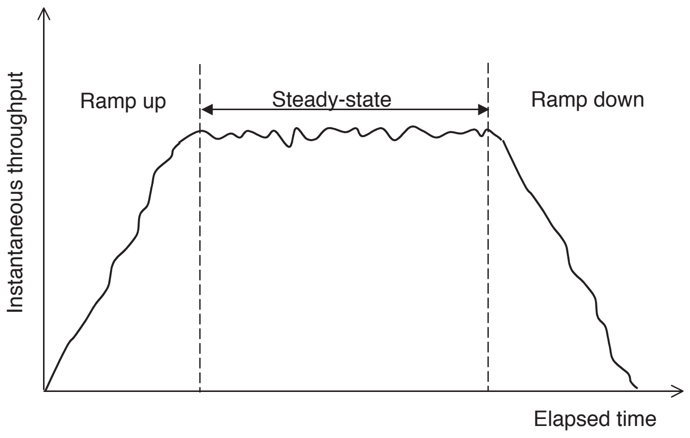


Figure 5: Steady-state throughput measurements

(i.e., $\lambda \equiv X$) for a given user load (N) is found by taking measurements over some lengthy measurement period T and eliminating any ramp-up or ramp-down periods from the data. A nominal T might be on the order of 5 to 10 minutes, depending on the application.

Industry standard benchmarks, such as those designed by SPEC (`www.spec.org`) and TPC (`www.tpc.org`), have a requirement that all reported throughput results be measured in steady state.

4 A Simple Queue in PDQ

The relationship between the grocery store scenario and PDQ functions, is summarized in Table 2. This allows us to construct a simple model of a grocery store checkout (Figs. 3 and 4) expressed in the Perl variant of PDQ:

Table 2: PDQ functions for Fig. 4

Physical	**Queue**	**PDQ Function**
Customers	Workload	`CreateOpen()`
Cashier	Service node	`CreateNode()`
Accounting	Service time	`SetDemand()`

```
#! /usr/bin/perl
# groxq.pl

use pdq;

#------------------------ INPUTS ---------------------
$ArrivalRate = 3/4; # customers per second
$ServiceRate = 1.0; # customers per second
$SeviceTime  = 1/$ServiceRate;
$ServerName  = "Cashier";
$Workload    = "Customers";

#----------------------- PDQ Model -------------------
# Initialize PDQ internal variables
pdq::Init("Grocery Store Checkout");

# Change the units used by PDQ::Report()
pdq::SetWUnit("Cust");
pdq::SetTUnit("Sec");

# Create the PDQ service node (Cashier)
$pdq::nodes = pdq::CreateNode($ServerName, $pdq::CEN, $pdq::FCFS);

# Create the PDQ workload with arrival rate
$pdq::streams = pdq::CreateOpen($Workload, $ArrivalRate);

# Define service rate per customer at the cashier
pdq::SetDemand($ServerName, $Workload, $SeviceTime);

#----------------------- OUTPUTS ---------------------
# Solve the PDQ model
pdq::Solve($pdq::CANON);
pdq::Report(); # Generate a full PDQ report
```

In the above PDQ code, we have as input values for the arrival rate (λ) and the service time (S), respectively:

$$\lambda = 3/4 \tag{4}$$

$$S = 1.0 \tag{5}$$

Using metric relation (2), the cashier utilization is given by:

$$\rho = \frac{3 \times 1}{4} = 0.75\,, \tag{6}$$

Similarly, using metric relations (1) and (2), the residence time output is given by:

$$R = \frac{1.0}{1 - \frac{3}{4} \times 1.0} = 4.0 \text{ seconds}\,, \tag{7}$$

which states that the time spent at the checkout stand is equivalent to four average service periods, when the cashier is 75% busy. The corresponding average queue length is:

$$Q = \frac{3}{4 \times 4.0} = 3.0 \text{ customers} \tag{8}$$

For brevity, we only show the output portion of the generic PDQ report for this model.

```
        ***************************************
        ****** Pretty Damn Quick REPORT *******
        ***************************************
        ***  of : Sun Feb  4 17:25:39 2007  ***
        ***  for: Grocery Store Checkout    ***
        ***  Ver: PDQ Analyzer v3.0 111904  ***
        ***************************************

        ******   RESOURCE Performance   *******

Metric          Resource     Work              Value   Unit
---------       ------       ----              -----   ----
Throughput      Cashier      Customers        0.7500   Cust/Sec
Utilization     Cashier      Customers       75.0000   Percent
Queue Length    Cashier      Customers        3.0000   Cust
Residence Time  Cashier      Customers        4.0000   Sec
```

The computed PDQ values are identical to the theoretical predictions for throughput ($X = \lambda$), utilization (ρ), queue length (Q), and residence time (R).

With these fundamental queueing concepts in place, and PDQ as the tool to remove the drudgery of calculation, we can easily apply them to predicting the performance of individual hardware resources such as the CPU run-queue (see Chap. 4 of [5]) or a disk device driver, and so on. Most queueing theory books provide examples at this level. More important for predicting the performance real computer systems, however, is the ability to represent the workflow between multiple queueing facilities (resources). In other words, the interaction between requests made concurrently at processors, disks, and network, for example. We now explain how to do this with PDQ.

5 Circuits of Queues

Requests that flow from one queue to another correspond to a *circuit* or network of queues. (Fig. 6) When only the arrival rate (λ) of the requests, rather than the number of requesters is monitored, Fig. 6 is called an *open* circuit. An example of a non-computer situation that might be modeled by the open circuit in Fig. 6, is boarding an aircraft. The three stages are: waiting at the gate, queueing the jetway to get onto the aircraft, and finally queueing in the cabin aisle while passengers ahead of you get seated.

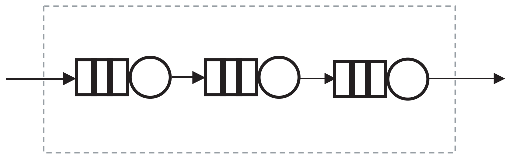


Figure 6: An open circuit involving three queueing stages

The average *response* time ($\mathcal{R}$) is given by the total time spent in each queueing stage or the sum of the three residence times. In the computer performance context, Fig. 6 could be applied to a 3-tier web application where only the rate at which HTTP requests is known.

Another form of queueing circuit involves a *finite* number (N) of customers or requests. This is precisely the situation created by a load-test platform. A finite number of client load generators issue requests into the test rig, and no additional requests can enter from outside the isolated system. Open source load and stress test tools can be found at `http://www.opensourcetesting.org/performance.php`. Moreover, there is a kind of feedback mechanism in operation in that no more than one request can be outstanding at a time. In other words, each script running on a load generator (e.g., a client PC) does not issue the next request until the previous one has completed. In queueing theory parlance, this is known as a *closed* queueing circuit.

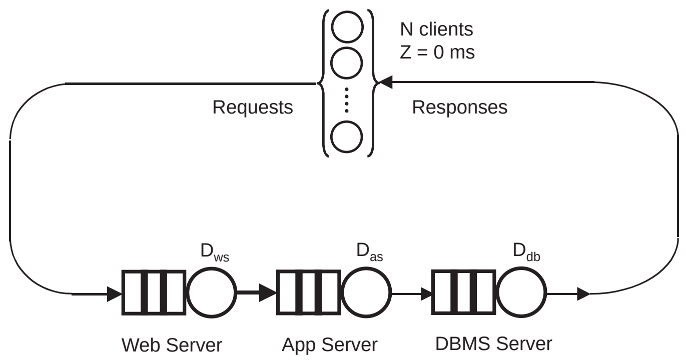


Figure 7: An closed circuit involving three queueing stages together with a special waiting stage (top) corresponding to N client-side load generators with average think time Z

6 e-Commerce Application in PDQ

In this section, we show how the closed circuit PDQ model in Fig. 7 can be applied to predicting the throughput performance of the 3-tier e-commerce architecture shown in Fig. 8. Due to financial and other constraints, the application is often load-tested on a miniature rendition of the production environment, prior to full deployment. In the case considered here, each tier was represented by a single server. Such a test rig can provide an excellent platform for generating steady-state performance measurements which are useful for parameterizing a PDQ model.

The throughput is measured as HTTP Gets per second (GPS), and the corresponding response time performance is measured in seconds (s). Due to space limitations, we focus exclusively on modeling throughput performance. The interested reader can find the details of modeling response times in Ref. [5].

For this example, the key load-test measurements are summarized in Table 3. Unfortunately, the data are not presented in equal user-load increments, which is less than ideal for proper performance analysis but not a show-stopper.

Both X and $\mathcal{R}$ are system metrics reported from the client-side. The utilization was obtained separately from performance monitors on each of the local servers.

The monitored utilizations (ρ) and throughputs (X) in Table 3 can be used together with a rearranged version of equation (3):

$$S = \frac{\rho}{X}\,, \tag{9}$$

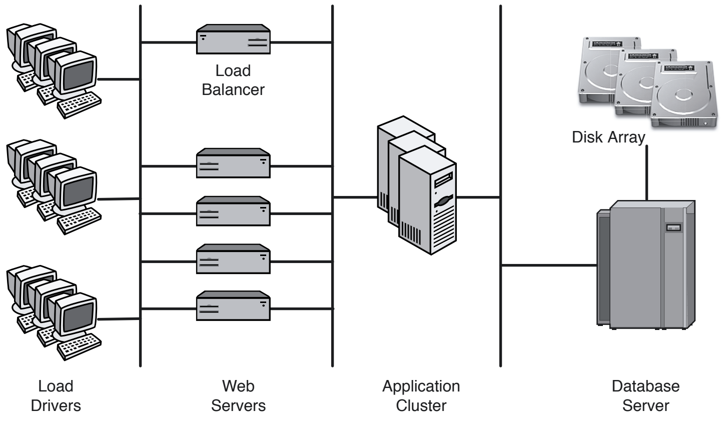


Figure 8: Multitier e-commerce application environment

Table 3: Measured performance data on each tier

N (Clients)	X (GPS)	$\mathcal{R}$ (s)	ρ_{ws} (%)	ρ_{as} (%)	ρ_{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

to calculate the corresponding service times for each tier. The service times for each load are summarized in Table 4 together with the summary average on the last line of the table.

Table 4: Service times derived from Table 3

N	S_{ws}	S_{as}	S_{db}
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 the next section, we show how the average of the derived service times (last row in Table 4) can be used to parameterize the PDQ model.

7 Naive PDQ Model

As a first attempt to model the performance characteristics of Fig. 7, we simply represent each application server as a separate PDQ node using the averaged service times from Table 4. In Perl PDQ, the parameterization of the queueing nodes is coded as follows:

```
pdq::Init($model);
$pdq::streams = pdq::CreateClosed($work, $pdq::TERM, $users,
                     $think);
...
# Create a queue for each of the three tiers
$pdq::nodes = pdq::CreateNode($node1, $pdq::CEN, $pdq::FCFS);
```

```
$pdq::nodes = pdq::CreateNode($node2, $pdq::CEN, $pdq::FCFS);
$pdq::nodes = pdq::CreateNode($node3, $pdq::CEN, $pdq::FCFS);
...
#  Timebase is seconds expressed as milliseconds
pdq::SetDemand($node1, $work, 9.3 * 1e-3);
pdq::SetDemand($node2, $work, 2.8 * 1e-3);
pdq::SetDemand($node3, $work, 0.9 * 1e-3);
```

A plot the predicted throughput in Fig. 9 shows that the naive PDQ model has a throughput which saturates too quickly when compared with the test rig data. However, PDQ also tells us that, given the measured service times in Table 4, the best possible throughput for this system is about 100 GPS. This is due to the bottleneck resource (the queue with the longest average service time), which is the front-end web server. It constrains the throughput in such a way that the maximum throughput:

$$\begin{aligned} X_{max} &= \frac{1}{max(S_{\text{ws}}, S_{\text{as}}, S_{\text{db}})} \\ &= \frac{1}{0.0093} \\ &= 107.53 \text{ GPS} \end{aligned} \tag{10}$$

is reached near the optimal load point (Table 1):

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

not 20 clients. If the service times change in the future, e.g.,

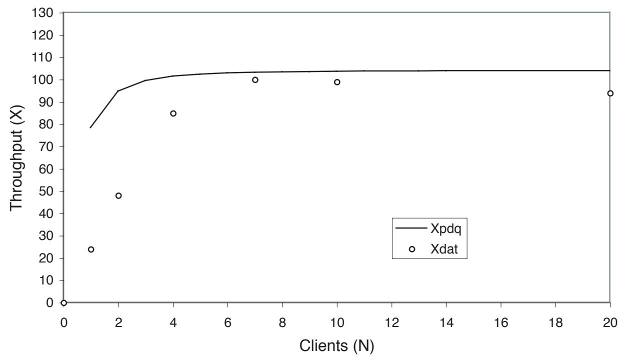


Figure 9: Naive PDQ model of throughput

due to a new code release for the application, the bottleneck resource may change and the PDQ model will be able to predict its effect on throughput and response times.

Clearly then, it is also desirable to model the overall data set better than the naive PDQ model has achieved, so far. One simple method to offset this rapid saturation in the throughput is to introduce a nonzero value to the think-time $Z > 0$:

```
$think = 28.0 * 1e-3; # free parameter
...
pdq::Init($model);
$pdq::streams = pdq::CreateClosed($work, $pdq::TERM, $users,
                                  $think);
```

This has the effect of slowing down the issuing of new requests into the system. In this sense, we are playing with the think-time as if is a *free* parameter. The non-zero $Z = 0.028$ seconds disagrees with the settings in the actual load test scripts, but it can give some perspective on how far away we are from finding an improved PDQ model. As Fig. 10

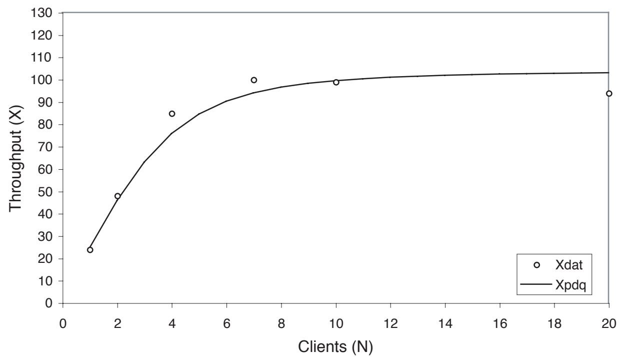


Figure 10: Throughput model with nonzero think-time

shows, this nonzero think-time improves the throughput profile quite dramatically.

$$\begin{aligned} N^* &= \frac{0.0093 + 0.0028 + 0.0009 + 0.028}{0.0093} \\ &= 4.41 \text{ clients} \end{aligned} \tag{12}$$

This trick with the think time tells us that there are additional latencies not accounted for in the load test measurements. The effect of the nonzero think-time is to add latency and to make the round trip time of a request longer than anticipated. This also has the effect of reducing the throughput at low loads. But the think-time was set to zero in the actual measurements. How can this paradox be resolved?

8 Including Hidden Latencies

The next trick is to add *dummy nodes* to the PDQ model in Fig. 11. There are, however, constraints that must be satisfied by the service demands of these virtual nodes. The service demand of each dummy node must be chosen in such a way that it does not exceed the service demand of the bottleneck node. In addition, the number of dummy

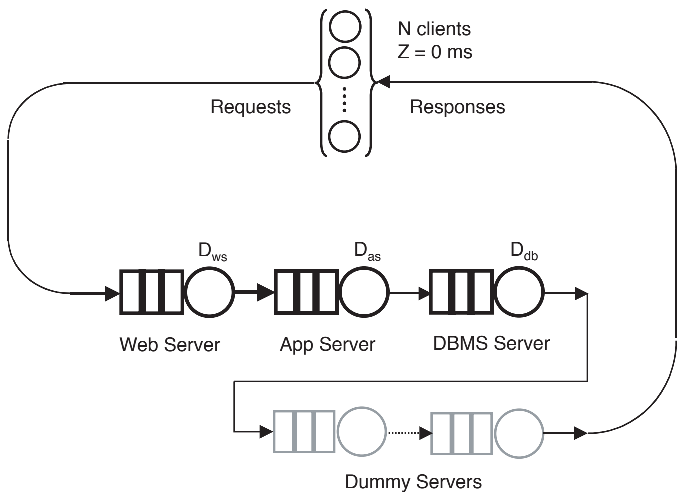


Figure 11: Hidden latencies modeled as dummy nodes

nodes must be chosen such that the sum of their respective service demands does not exceed $R_{\min} = R(1)$ when there is no contention, i.e., for a single request. It turns out that we can satisfy all these constraints if we introduce 12 uniform dummy nodes, each with a service demand of 2.2 ms. The change to the relevant PDQ code fragment is:

```
use constant MAXDUMMIES => 12;
$think    = 0.0 * 1e-3;  # same as in test rig
$dtime    = 2.2 * 1e-3;  # dummy service time
# Create dummy nodes with their service times ...
for ($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);
}
```

Notice that the think-time is now set back to zero. The results of this change to the PDQ model are shown in Figs. 12. The throughput profile still maintains a good fit at low loads ($N < N^*$) where:

$$\begin{aligned} N^* &= \frac{0.0093 + 0.0028 + 0.0009 + 12(0.0022)}{0.0093} \\ &= 4.24 \text{ clients} \end{aligned} \tag{13}$$

but needs improvement above N^*.

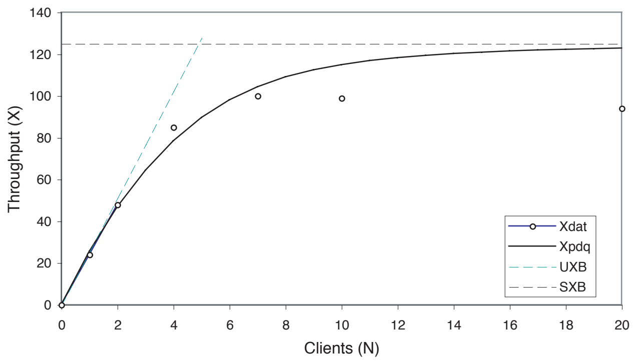


Figure 12: Throughput with $Z = 0$ and dummy nodes

9 Load-dependent Server

Certain aspects of the physical system were not measured, and this makes PDQ model validation difficult. So far, we have tried adjusting the workload intensity with a nonzero think-time. Setting $Z = 0.028$ seconds removed the rapid saturation but is inconsistent with $Z = 0$ seconds used for the actual test measurements.

Introducing the dummy queueing nodes into the PDQ model improved the low-load model, but it does not accommodate the throughput *roll-off* observed in the data. To model this effect, we replace the web-server node in with a load-dependent node. The general theory behind load-dependent servers is discussed in reference [5]. Here, we adopt a slightly simpler approach. The service time (S_{ws}) in Table 4 indicates that it is not constant across all client loads. We need a way to express this variability. If we plot the monitored data for S_{ws}, we can do a statistical regression fit like that shown in Fig. 13. The resulting power-law equation is given by:

$$D_{\text{ws}}(N) = 8.0000\, N^{0.0850}, \tag{14}$$

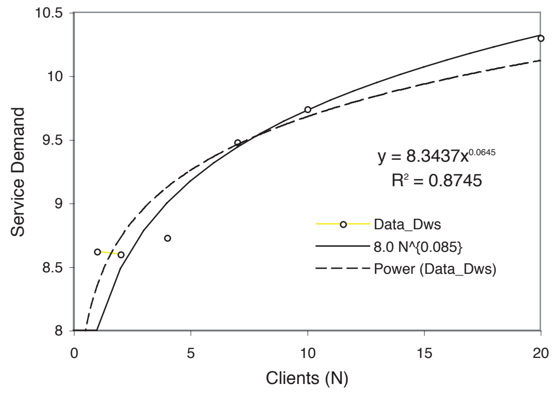


Figure 13: Regression fit of web server times

which parameterizes `node1` of the PDQ model as:

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

The customized output of the complete PDQ model (see listing in Section 13) is:

```
 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
```

which shows good agreement with the measured data for $D = 12$ dummy PDQ nodes.

The impact on the throughput model can be gauged from Fig. 14. The curve labeled $Xpdq2$ is the predicted over-

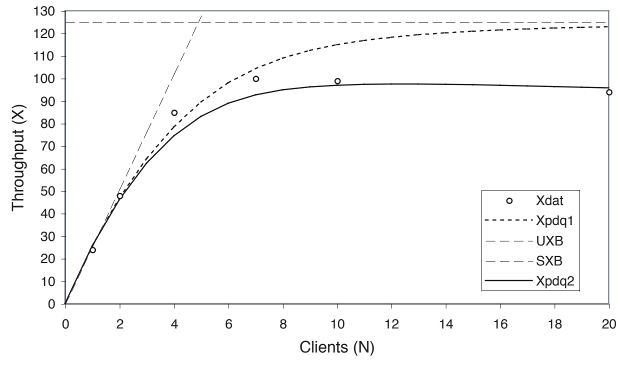


Figure 14: Model of load-dependent throughput

driven throughput based on the load-dependent server for web front-end and the predictions fit well within the error margins of the measured data.

In this case, there is little virtue in using PDQ to project beyond the measured load of $N = 20$ clients because the throughput is not only saturated, it is also retrograde. However, now that a PDQ model has been constructed and validated against test data, it can be used to address any number of *what if* scenarios.

10 Going Further with PDQ

The previous example is already very sophisticated and similar PDQ models usually cover most needs. There are situations, however, where more detailed models are needed. Two examples that can arise are: multiple servers and multiple workloads.

10.1 Multiple servers

A non-computer situation that might be modeled by the multiple server queue in Fig. 15, is waiting in line at a bank or post office. In the computer performance context, Fig. 15

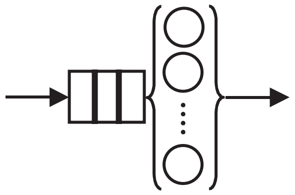


Figure 15: PDQ model of multiserver queue

could be used as a simple model of a symmetric multiprocessor. See Chapter 7 of reference [5] for more on this topic.

The response time in (1) is replaced by:

$$R \simeq \frac{S}{1-\rho^m} \tag{15}$$

where $\rho = \lambda S$ and m is the integral number of servers. Technically, this is an approximation but a good one. The exact solution is more complex and can be found using the following algorithm in Perl:

```
#! /usr/bin/perl
# erlang.pl

## Input parameters
$servers = 8;
$erlangs = 4;

if($erlangs > $servers) {
    print "Error: Erlangs exceeds servers\n";
    exit;
}

$rho     = $erlangs / $servers;
$erlangB = $erlangs / (1 + $erlangs);
for ($m  = 2; $m <= $servers; $m++) {
    $eb  = $erlangB;
    $erlangB = $eb * $erlangs / ($m + ($eb * $erlangs));
}

## Output results
$erlangC  = $erlangB / (1 - $rho + ($rho * $erlangB));
$normdwtE = $erlangC / ($servers * (1 - $rho));
$normdrtE = 1 + $normdwtE;              # Exact
$normdrtA = 1 / (1 - $rho**$servers);   # Approx
```

This is precisely the queueing model developed by Erlang 100 years ago, that we discussed in Section 2. In that case, each server represented a trunk telephone line.

10.2 Multiple Workloads

In reality it is common for a single resource, like a database server, to handle a multitude of transaction types. For example, purchasing an airline ticket or a hotel room on the web may involve up to half a dozen different transactions before the ticket or room is finally issued. This kind of situation can be modeled in PDQ as follows.

Consider the simpler case of three different transaction types labeled by the colors red, green and blue. Each of these colored workloads may a access a common resource e.g., a database server. In the queueing paradigm (Fig. 16), each

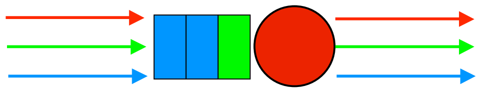


Figure 16: PDQ model of a multiple workloads queue

of these colored workloads is distinguished by their different service time requirements when they get serviced. In other words, the red work takes a red service period, the green work takes a green service period, and so on. Each color has its own arrival rate as well.

Assuming these service times are quite different from each other (otherwise why bother making any distinction?), the real impact occurs in the waiting line because when say a red request arrives into the queue, it's response time will be determined not by the number of requests ahead of it (which is only true for a "monochromatic" workload), but the particular combination of colors ahead of it. In PDQ, Fig. 16 might be represented as follows:

```
$pdq::nodes = pdq::CreateNode("DBserver", $pdq::CEN, $pdq::FCFS);

$pdq::streams = pdq::CreateOpen("Red", $ArrivalsRed);
$pdq::streams = pdq::CreateOpen("Grn", $ArrivalsGrn);
$pdq::streams = pdq::CreateOpen("Blu", $ArrivalsBlu);

pdq::SetDemand("DBserver", "Red", $ServiceRed);
pdq::SetDemand("DBserver", "Grn", $ServiceGrn);
pdq::SetDemand("DBserver", "Blu", $ServiceBlu);
...
```

Naturally, the PDQ report generated by multiple workloads will be more complicated because of all the possible interactions.

This just gives some idea of the ways in which PDQ can be extended to reflect more realistic computer architectures. Further discussion of these matters would take us too far afield, but the interested reader can find more details in my book [5].

11 Guidelines for Applying PDQ

Performance modeling of any type is part science and part art. It is therefore, impossible to provide a complete set of rules or algorithms that will furnish the correct model. In fact, as shown here, the process is one of iterative improvement. The best guide is experience, and experience comes from simply doing it repeatedly.

In that vein, here are some guidelines that might be of help when you are trying to construct PDQ models.

Keep it simple: A PDQ model should be as simple as possible, but no simpler! It is almost axiomatic that the more you know about a system architecture, the more detail you will try to throw into the PDQ model; including the kitchen sink.

More like the map than the metro: A PDQ model is to a computer system as a subway map is to the actual subway system. A subway map is an abstraction that has very little to do with the physical subway system. It encodes only sufficient detail to enable you to transit from point A to point B. It does not include a lot of irrelevant details such as altitude of the stations, or even their actual geographical proximity. A PDQ model is a similar kind of abstraction.

The big picture: Unlike most aspects of computer technology, which require you to absorb large amounts of minute detail, PDQ modeling is all about deciding how much detail can be *ignored!*

Seek the principle of operation: If you cannot describe the principle of operation in 25 words or less, you probably do not understand it well enough to start constructing a PDQ model. The principle of operation for a time-share computer system, for example, can be stated as: *Time-share gives every user the illusion that they are the ONLY user active on the system.* All the hundreds of lines of code in Linux to implement time-slicing, priority queues, etc., are there merely to support this illusion.

Guilt is golden: PDQ modeling is also about spreading the guilt around. You, as the performance analyst, only have to shine the light on the performance problem, then stand back while others flock to fix it.

Where to start? Have some fun with blocks; *functional blocks*! One place to start constructing a PDQ model is by drawing a *functional block diagram.* The objective is to identify where time is spent at each stage in processing the workload of interest. Ultimately, each functional block gets converted to a queueing subsystem. This includes the ability to distinguish sequential and parallel processing. Other diagrammatic techniques e.g., UML diagrams, may also be useful.

Inputs and outputs: When defining PDQ models, it helps to write down a list of *inputs*; measurements or estimates that are used to parameterize the model, and *outputs*; numbers that are generated by calculating the model. See Section 4.

No Service, no queues: You know the restaurant rule: "No shoes, no service!" Well, this is the PDQ modeling rule: no service, no queues. In your PDQ models, there is no point creating more queueing nodes than those for which you have measured service times. If the measurements of the real system do not include the service time for a PDQ node that you think ought to be in your model, then that PDQ node cannot be defined.

Estimating service times: Service times are notoriously difficult to measure directly. Often, however, the service time can be calculated from other performance metrics that are easier to monitor. See Table 4.

Change the data: If the measurements do not support your PDQ performance model, more than likely, the measurements need to be redone.

Closed or open queue? When trying to figure out which queueing model to apply, ask yourself whether or not you have a finite number of requests to service. If the answer is yes (as it should be for a load-test platform), then it is a *closed* queueing model. Otherwise use an *open* queueing model. See Section 5.

Steady-state measurements: The steady-state measurement period should on the order of 100 times larger than the largest service time. See Section 3.

Which timebase?: Use the timebase of your measurement tools. If the tool reports in seconds, use seconds, if it reports in microseconds, use microseconds. If there are multiple monitoring sources of data, then ALL numbers should be normalized to the same units before doing any calculations.

Workloads come in threes: In a mixed workload model (multiclass streams in PDQ), avoid using more than three concurrent work streams whenever possible. Apart from generating a PDQ report that is unwieldy to read, generally you are only interested in the interaction of two workloads, i.e., a pairwise comparison. Everything else goes in the third workload (AKA "the background"). If you cannot see how to do this, you are probably not ready to create the PDQ model.

12 Conclusion

Performance modeling is a demanding discipline which can best be conquered through persistent practice. Most of the effort goes into constructing and validating the model of your particular environment and applications. Once the PDQ model is validated, however, it is not necessary to keep reconstructing it. In general, it will only need to be tweaked to reflect performance changes occasioned by hardware upgrades and new software releases.

The e-commerce application modeled in Section 6 provides a good starting point which can be extended to include multiple servers and additional workloads. One of the remarkable outcomes of that PDQ model is that it allowed us to "see" effects (hidden latencies) that were not explicitly captured in the monitored data. But, perhaps a more significant result of using PDQ is not performance models, per se. Rather, the process of using PDQ provides an organizational framework within which all performance data, from monitoring to prediction, can be properly comprehended.

References

[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

13 Listing for e-Commerce Model

```
#! /usr/bin/perl
# ebiz_final.pl

use pdq;
use constant MAXDUMMIES => 12;

# Hash AV pairs: (load in vusers, thruput in gets/sec)
%tpdata   = ( (1,24), (2,48), (4,85), (7,100), (10,99), (20,94) );
@vusers   = keys(%tpdata);

$model    = "e-Commerce Final Model";
$work     = "ebiz-tx";
$node1    = "WebServer";
$node2    = "AppServer";
$node3    = "DBMServer";
$think    = 0.0 * 1e-3;  # same as in test rig
$dtime    = 2.2 * 1e-3;  # dummy service time

# Header for custom report
printf("%2s\t%4s\t%4s\tD=%2d\n", "N", "Xdat", "Xpdq", MAXDUMMIES);

foreach $users (sort {$a <=> $b} @vusers) {
    pdq::Init($model);

    $pdq::streams = pdq::CreateClosed($work, $pdq::TERM, $users,
                       $think);

    $pdq::nodes = pdq::CreateNode($node1, $pdq::CEN, $pdq::FCFS);
    $pdq::nodes = pdq::CreateNode($node2, $pdq::CEN, $pdq::FCFS);
    $pdq::nodes = pdq::CreateNode($node3, $pdq::CEN, $pdq::FCFS);

    # Timebase in seconds expressed as milliseconds
    pdq::SetDemand($node1, $work, 8.0 * 1e-3 * ($users ** 0.085));
    pdq::SetDemand($node2, $work, 2.8 * 1e-3);
    pdq::SetDemand($node3, $work, 0.9 * 1e-3);

    # Create dummy nodes with their service times ...
    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);
    }

    pdq::Solve($pdq::EXACT);
    printf("%2d\t%2d\t%4.2f\n", $users, $tpdata{$users},
        pdq::GetThruput($pdq::TERM, $work));
}
```

14 Author

Neil Gunther, M.Sc., Ph.D. is an internationally recognized consultant who founded Performance Dynamics Company (www.perfdynamics.com) in 1994. Prior to that, Dr. Gunther applied his training in theoretical physics to research and management positions at San Jose State University, JPL/NASA (Voyager and Galileo missions), Xerox PARC and Pyramid/Siemens Technology. Performance Dynamics has recently embarked on joint research into Quantum Information Technology. Dr. Gunther is a member of the AMS, APS, ACM, CMG, IEEE, and INFORMS.