

# Analyzing Computer System Performance with Perl::PDQ

Bearbeitet von  
Neil J Gunther

1. Auflage 2004. Buch. XXIII, 436 S. Hardcover

ISBN 978 3 540 20865 5

Format (B x L): 15,5 x 23,5 cm

Gewicht: 1820 g

[Weitere Fachgebiete > EDV, Informatik > Hardwaretechnische Grundlagen > Externe Speicher & Peripheriegeräte](#)

Zu [Inhaltsverzeichnis](#)

schnell und portofrei erhältlich bei

The logo for beck-shop.de features the text 'beck-shop.de' in a bold, red, sans-serif font. Above the 'i' in 'shop' are three red dots of varying sizes, arranged in a slight arc. Below the main text, the words 'DIE FACHBUCHHANDLUNG' are written in a smaller, red, all-caps, sans-serif font.

**beck-shop.de**  
DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung [beck-shop.de](http://beck-shop.de) ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.

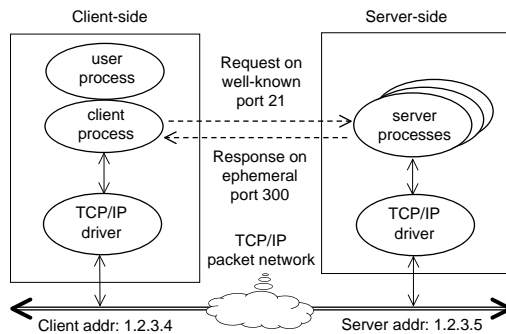
---

## Client/Server Analysis with PDQ

### 9.1 Introduction

In this chapter we present performance analysis for client/server architectures using PDQ. This material is the most complex use of PDQ so far in that it draws on the techniques presented in previous chapters and extends them to software and communication network analysis.

Many modern computing environments are moving to a more distributed paradigm, with client/server being one of the most common distributed architectures in use today (Fig. 9.1). But the fragmentation of computer resources across networks has also broken the notion of centralized performance management that worked for mainframe applications. Consequently, performance



**Fig. 9.1.** Classical client/server process architecture

management is a major problem once again, and those business operations that have been quick to adopt client/server, technology are now starting to recognize the hidden costs of providing services with consistent performance. Those implementing client/server applications are learning to demand assurance about performance in terms of *service level agreements* (SLAs).

In this chapter, you will learn how to apply PDQ to the performance analysis of a multitier B2C client/server environment. A key point to note is that PDQ can be used to predict the scalability of distributed *software* applications, not just hardware as in Chap. 7. This is achieved by using the workflow analysis of Sect. 9.3.3.

Moreover, the approach presented here shows you how to make benchmarking and load testing more cost-effective. Load test results only need to be obtained on a relatively sparse set of smaller test platform configurations. These data can be used to parameterize a PDQ model which, in turn, can be used to predict the performance of the client/server application once it is deployed into production. Thus, PDQ offers another way to keep the cost of load testing complex distributed applications under control.

## 9.2 Client/Server Architectures

From the standpoint of computer technology, client/server refers to a software interface specification shown schematically in Fig. 9.1. The client/side of the interface makes requests and the server side of the interface provides the service usually without the client process understanding how the service is actually implemented. From the standpoint of the services provided, it is irrelevant where the client and server processes are physically located and which resources they consume in order to do the processing. This creates a very flexible computing environment.

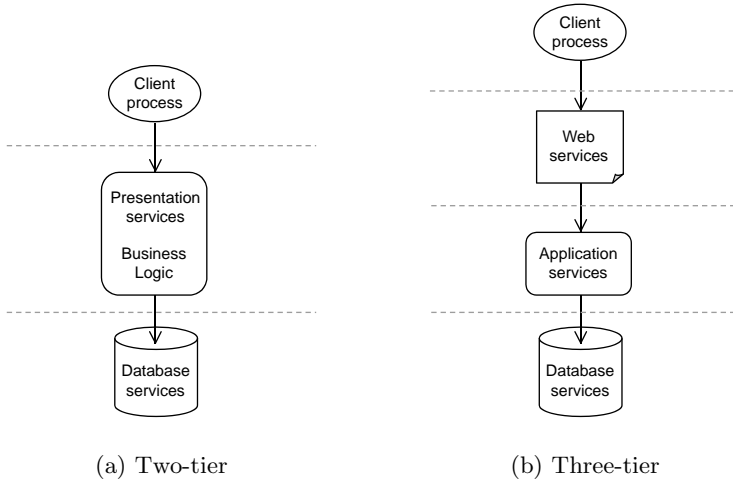
So how does the client/server software interface work? The rules that the client and server processes communicate are called a protocol. Client/server is a special case of distributed computing.

Client/server applications can be implemented using a variety of protocols, but Transmission Control Protocol over Internet Protocol (TCP/IP) is among the most commonly chosen. Any TCP/IP network connection between a client process and services provided by a remote host requires the following items:

- Client-side IP host address, e.g., 1.2.3.4
- Server-side host address, e.g., 1.2.3.5
- Server process on a well-known port number, e.g., 21
- Client process on an ephemeral port number, e.g., anything above 255
- Process communication protocol, e.g., TCP, UDP

All this information gets encapsulated in one or more IP packets, which are then transmitted over the network, along with either the request message or the data response. This is the basis of the transparency offered by client/server architectures.

From the performance analyst's standpoint, however, knowing which software processes are executing and on which remote hardware resources is vital to any kind of performance analysis. As we shall see shortly, this is the Achilles' heel of many client/server applications.



**Fig. 9.2.** Comparison of (a) two-tier and (b) three-tier client/server architectures

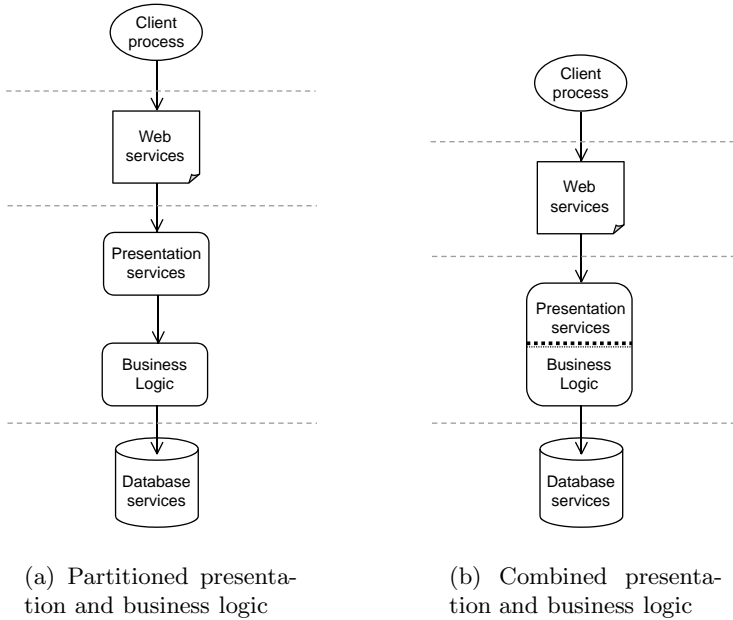
The kind of distributed performance measurements required are discussed in Gunther [2000a, Chap. 4], but this level of integrated performance monitoring is still in a rather immature state. A whimsical analogy with remotely monitored pizza delivery in Gunther [2000a, Chap. 8] highlights the requirements. The upshot of that analysis is that you need a client/server architecture (tools) to manage a client/server architecture (application). This is easy to say, but hard to do.

### 9.2.1 Multitier Environments

The classic client/server model is a master/slave relationship, where the client process resides in one physical location, e.g., on a desktop, hand-held personal digital assistant (PDA), or cell phone, and the server process resides in a different physical location. Both processes are connected via a network. Figure 9.2(a) exemplifies this is the basic *two-tier* architecture.

### 9.2.2 Three-Tier Options

In a three-tier approach, like that shown in Fig. 9.2(b), another layer of servers is inserted between the clients and the data servers. These second-tier servers provide dual functionality: they can act as clients that make requests to the appropriate data sources (application servers), and they function as servers for the desktop clients (function servers). A three-tiered architecture divides applications into parts that run on different types of platforms.



**Fig. 9.3.** Comparison of logical three-tier client/server architectures. In (a) the presentation logic and business logic separated across tiers, while in (b) they are combined on the same tier and interact through an API

The clients support the presentation layer of the application to display data retrieved from the server side, e.g., databases. The application servers process business applications such as financial transactions and marketing queries. The data sources range from SQL databases to legacy application programs. A client/server architecture also lets the system administrator or architect separate the business logic from the actual processing logic (Fig. 9.3). This modularity enables business changes to be rapidly incorporated into applications.

A three-tier architecture can also be expanded horizontally by adding different types of application servers and different types of databases. This additional capacity and functionality can be enhanced in a way that is quite transparent to the clients.

The flexibility and scalability of a three-tiered (or  $n$ -tiered) architecture comes at the price of greater complexity of applications and management. Modern client/server development environments offer facilities such as object libraries, 4GL development languages, and dynamic directories to assist in dealing with multitiered complexity. A primary goal of these tools is to facilitate changes in the environment without rebuilding the client applications.

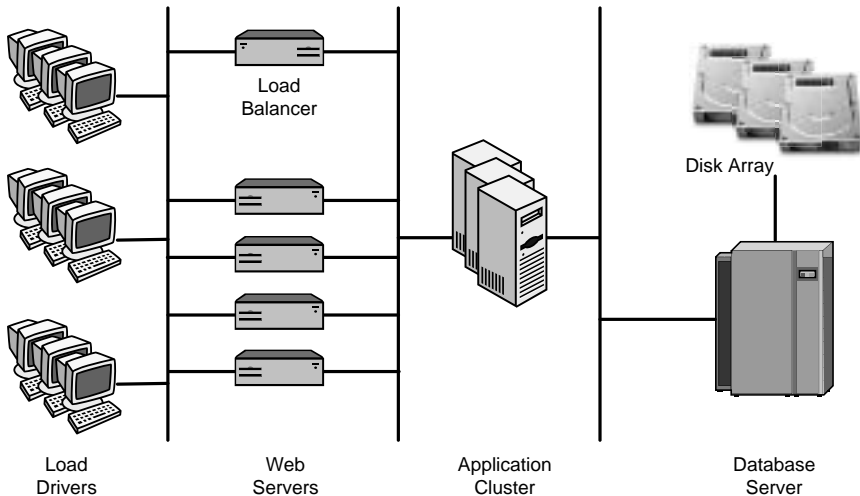
In practice, the performance of a multitiered application will also be determined by such factors as:

- How much of the client user interface runs on the application server.
- Whether or not presentation services and business logic reside on the same physical server. (Fig. 9.3)
- How much of the application logic runs on the database servers.
- How much data is placed on the application server.

For more on the impact of business logic placement on performance and scalability is [www.onjava.com/pub/a/onjava/2003/10/15/php\\_scalability.html](http://www.onjava.com/pub/a/onjava/2003/10/15/php_scalability.html).

### 9.3 Benchmark Environment

In this section we demonstrate how to construct and evaluate performance models of modern three-tier client/server architectures. The environment to be analyzed is a *business-to-consumer* (B2C) application intended to support 1,000 to 2,000 concurrent Web-based users. The analysis is to be made against the benchmark platform in Fig. 9.4, which is on the scale of a TPC-W-type benchmark. The application spans a distributed architecture involv-



**Fig. 9.4.** Multitier client/server benchmark environment

ing PC benchmark drivers, multiple Web servers, an application cluster, a database server, and attached storage array. All the services are connected via a 100Base-T switched Ethernet. The hardware resources together with some relevant performance measures are listed in Table 9.1.

### 9.3.1 Performance Scenarios

The objectives of the benchmark are to assess scalability of the application prior to deployment. The primary performance criterion is that the 95th percentile of user response times for Web-based transactions should not exceed 500 ms.

The performance scenarios that we shall consider are:

- 100 client generators on the baseline benchmark platform (Sect. 9.4.1)
- 1,000 client generators predicted by PDQ (Sect. 9.4.2)
- 1,500 client generators predicted by PDQ (Sects. 9.4.3, 9.4.4, and 9.4.5)
- determine the client generator load at saturation (Sect. 9.4.6)

In reality, different load points can be chosen. The key idea is to predict many points using PDQ, but only benchmark a few for validation. Using PDQ in this way is intended to be more cost effective than reconfiguring the real platform to measure each load point.

The various speeds and feeds of the benchmark platform are summarized in Table 9.1 where CPU2000 refers to the integer SPEC CPU benchmark ratings available online at [www.spec.org](http://www.spec.org). Arguably, the SPEC CPU2000 metric has replaced the notion of nominal MIPS (cf. Sect. 7.3.5). With these assumptions

**Table 9.1.** Baseline benchmark configuration ratings

| Node                | Number | CPU2000 | Ops/s | MB/s |
|---------------------|--------|---------|-------|------|
| Desktop driver      | 100    | 499     | –     | –    |
| Load balancer       | 1      | 499     | –     | –    |
| Web server          | 2      | –       | 400   | –    |
| Application cluster | 1      | 792     | –     | 4.00 |
| Database server     | 1      | 479     | –     | –    |
| Disk arrays         | 4      | –       | 250   | 5.72 |
| 100Base-T LAN       | 1      | –       | –     | 0.40 |

in mind, we construct a baseline model in PDQ and then use it to predict performance for a target level of 2,000 clients.

### 9.3.2 Workload Characterization

The same client application runs on all of the PC drivers, and it can initiate any of three transactions types with the mean request rates shown in Table 9.1. These rates also determine the multiclass workload mix as discussed in Chap. 3. A *business work* unit is a high-level measure appropriate for both business analysis and system-level performance analysis. For example, the three transactions listed in Table 9.2 might be used to process claims

**Table 9.2.** Client transactions

| Transaction name | Prefix symbol | Rate (per minute) |
|------------------|---------------|-------------------|
| Category display | CD            | 4                 |
| Remote quote     | RQ            | 8                 |
| Status update    | SU            | 1                 |

at an insurance company. The number of claims processed each day is a quantity that measures both business performance and computational performance since it is also a measure of aggregate throughput.

The underlying distributed processes that support the three key transactions in Table 9.2 are listed in Table 9.3. The prefixes in the process names (first column) identify which transaction each process supports. The process identification number (PID) can be obtained using performance tools such as the UNIX `ps -aux` command. The third column shows how many kilo-instructions are executed by each process. These numbers will be used to calculate the process service demands. For compiled applications written in C or C++, for example, the assembler instruction counts for each software component of the application can be obtained from compiler output as outlined in Example 9.1. For J2EE applications, other tools are available for determining component service demands (see Appendix D). The last two columns of Table 9.3 show I/O rates and caching effects that will also be used as parameters in the PDQ model.

*Example 9.1.* To demonstrate how instruction counts in Table 9.3 can be obtained, we took the source code for the C version of the *baseline* PDQ model (`baseline.c` is included in the PDQ download) in Sect. 9.4.1 and compiled it using `gcc` with the `-S` switch. The command is:

```
gcc -S baseline.c
```

This produced a corresponding file `baseline.s` containing assembler code only. The UNIX `wc` command was then applied to that file to count the number of assembler instructions (assuming 1 assembly code instruction per line).

```
wc -l baseline.c 477
wc -l baseline.s 1583
```

The latter number corresponds to the values appearing in the third column of Table 9.3. The only difference is that each client/server component corresponds not to 1.58  $k$ -instructions, but to hundreds of  $k$ -instructions.  $\square$

The service demand given in (2.9), defined in Chap. 2, can be derived from Tables 9.1 and 9.3 for each type of workload by using a variant of the *iron law of performance* in (7.9) in Chap. 7. The service demand for process  $c$  executing on resource  $k$  is given by:

$$D_k(c) = \frac{(\text{instruction count})_c}{\text{MIPS}_k}, \quad (9.1)$$

**Table 9.3.** Workload parameters for client/server processes

| Process name | Process PID | Assembler $k$ -instructions | Disk I/Os | Percent cached |
|--------------|-------------|-----------------------------|-----------|----------------|
| CD_Request   | 1           | 200                         | 0         | 0              |
| CD_Result    | 15          | 100                         | 0         | 0              |
| RQ_Request   | 2           | 150                         | 0         | 0              |
| RQ_Result    | 16          | 200                         | 0         | 0              |
| SU_Request   | 3           | 300                         | 0         | 0              |
| SU_Result    | 17          | 300                         | 0         | 0              |
| ReqCD_Proc   | 4           | 500                         | 1.0       | 50             |
| ReqRQ_Proc   | 5           | 700                         | 1.5       | 50             |
| ReqSU_Proc   | 6           | 100                         | 0.2       | 50             |
| CDMsg_Proc   | 12          | 350                         | 1.0       | 50             |
| RQMsg_Proc   | 13          | 350                         | 1.5       | 50             |
| SUMsg_Proc   | 14          | 350                         | 0.5       | 50             |
| DBCD_Proc    | 8           | 5000                        | 2.0       | 0              |
| DBRQ_Proc    | 9           | 1500                        | 4.0       | 0              |
| DBSU_Proc    | 10          | 2000                        | 1.0       | 0              |
| LB_Send      | 7           | 500                         | 0         | 0              |
| LB_Recv      | 11          | 500                         | 0         | 0              |
| LAN_Inst     | 18          | 4                           | 0         | 0              |

where  $(\text{instruction count})_c$  refers to the number of executable instructions belonging to workload  $c$ , and  $\text{MIPS}_k$  is the throughput rating of hardware resource  $k$  in *mega instructions per second*. Expressing (9.1) as thousands of instructions per process and replacing MIPS by  $10^6$  instructions/s produces:

$$D_k(c) = (10^3 \times \text{instructions})_c \times \left( \frac{\text{seconds}}{10^6 \times \text{instructions}} \right)_k, \quad (9.2)$$

which renders the service demand in units of milliseconds.

In the PDQ model `baseline.pl` presented in Sect. 9.4, the service demand is represented by a two-dimensional array, denoted `$demand[[ ]]`, where the first entry is a process and then second is a physical resource. The *Category Display* process, for example, has a service demand defined by:

```
$demand[$CD_Req] [$PC] = 200 * $K / $PC_MIPS;
```

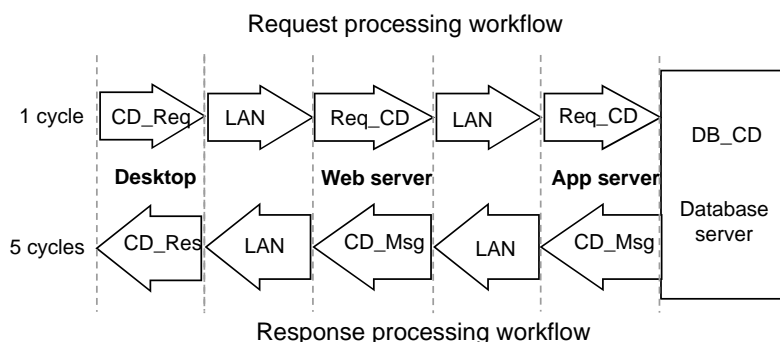
when executing in the PC benchmark driver. Here, `$K` is a Perl scalar representing the constant 1024, and the scalar `$PC_MIPS` represents the MIPS rating of the PC driver.

### 9.3.3 Distributed Workflow

To build a useful performance model in PDQ, the flow of work between the queuing nodes must be expressed in terms of the processes that support the B2C transactions. For example, processing the *CD* transaction incurs the following workflow:

1. A process on the PC driver (representing the user) issues a request `CD_Request` to the web sever. The web server `Req_CD` in turn activates a process `App_CD` on the application server that propagates a request to the database server `DB_CD`.
2. The database server is activated by this request, some data are read, and are sent back to the application server.
3. Upon receipt of the data, another Web server process updates the transaction status and routes the result back to the originating user. The user's display is then updated with the retrieved data.

The complete chain of processes and the hardware resources that support the CD transaction are shown in Fig. 9.5. The right-pointing arrows represent



**Fig. 9.5.** The processing workflow required by the category display transaction

the client request being processed, while the left-pointing arrows represent the response data being returned to the client process. The response arrows are wider to reflect the fact that the database server issues multiple requests to the application server in order return the necessary volume of data. The CD transaction invokes five calls to the application server which are then propagated back to the client process. This is similar to the situation in the NFS timing chain discussed in Chap. 1.

As each process executes it consumes both CPU cycles and generates physical disk I/O. Table 9.3 summarizes the resource demands for each process, including physical I/O. Knowledge of such detailed performance information—including instruction traces and cached I/O rates—can be critical for meaningful performance analysis of client/server applications.

## 9.4 Scalability Analysis with PDQ

We construct a baseline model with 100 desktops, which we then use to evaluate the impact of a 10-fold increase in users. The PDQ model (Fig. 9.6),

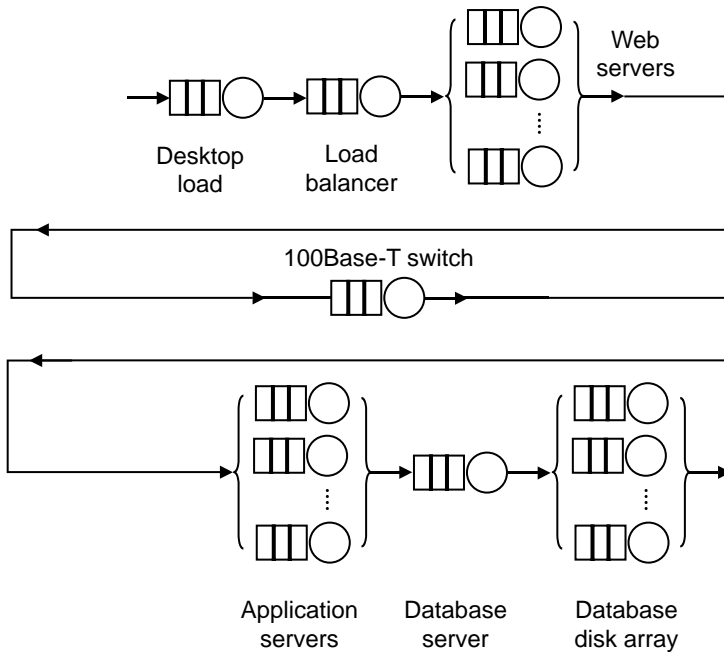


Fig. 9.6. PDQ performance model of client/server system in Fig 9.4

called `baseline.pl`, is constructed as an open-circuit model. Although 100 users suggests there is a finite number of requests in the system and that therefore a *closed* PDQ queuing model would be appropriate, we can use the result of Sect. 2.8.7 to justify the use of an open PDQ model. Moreover, in a Web or intranet environment requests can be submitted asynchronously without waiting for the response to a previous request.

We also note, in passing, that the LAN is a shared resource and could be modeled more accurately using a *load-dependent* PDQ queue (Chap. 10), but we found that it did not appreciably alter our results.

#### 9.4.1 Benchmark Baseline

The following Perlcode contains the PDQ model for the baseline system in Fig. 9.6. We present the complete PDQ baseline model without reviewing the corresponding PDQ output in this section. Instead, we review the certain PDQ outputs for the upgrade scenarios in Sect. 9.3.1 while omitting the respective PDQ scripts.

```
#!/usr/bin/perl
# cs_baseline.pl
use pdq;
```

```

#-----
# PDQ model parameters
#-----
$scenario          = "Client/Server Baseline";

# Useful constants
$K                 = 1024;
$MIPS              = 1E6;

$USERS            = 100;
$WEB_SERVS        = 2;
$DB_DISKS         = 4;
$PC_MIPS          = 499 * $MIPS;
$AS_MIPS          = 792 * $MIPS;
$LB_MIPS          = 499 * $MIPS;
$DB_MIPS          = 479 * $MIPS;
$LAN_RATE         = 100 * 1E6;
$LAN_INST         = 4;
$WEB_OPS          = 400;
$DB_IOS           = 250;

$MAXPROC          = 20;
$MAXDEV           = 50;

$PC               = 0;    # PC drivers
$FS               = 1;    # Application cluster
$GW               = 2;    # Load balancer
$MF               = 3;    # Database server
$TR               = 4;    # Network
$FDA              = 10;   # Web servers
$MDA              = 20;   # Database disks

## Process PIDs
$CD_Req           = 1;
$CD_Rpy           = 15;
$RQ_Req           = 2;
$RQ_Rpy           = 16;
$SU_Req           = 3;
$SU_Rpy           = 17;
$Req_CD           = 4;
$Req_RQ           = 5;
$Req_SU           = 6;
$CD_Msg           = 12;
$RQ_Msg           = 13;
$SU_Msg           = 14;
$GT_Snd           = 7;
$GT_Rcv           = 11;
$MF_CD           = 8;
$MF_RQ           = 9;

```

```

$MF_SU           = 10;
$LAN_Tx          = 18;

# Initialize array data structures
for ($i = 0; $i < $WEB_SERVS; $i++) {
    $FDarray[$i]->{id} = $FDA + $i;
    $FDarray[$i]->{label} = sprintf("WebSvr%d", $i);
}
for ($i = 0; $i < $DB_DISKS; $i++) {
    $MDarray[$i]->{id} = $MDA + $i;
    $MDarray[$i]->{label} = sprintf("SCSI%d", $i);
}

$demand[$CD_Req][$PC] = 200 * $K / $PC_MIPS;
$demand[$CD_Rpy][$PC] = 100 * $K / $PC_MIPS;
$demand[$RQ_Req][$PC] = 150 * $K / $PC_MIPS;
$demand[$RQ_Rpy][$PC] = 200 * $K / $PC_MIPS;
$demand[$SU_Req][$PC] = 300 * $K / $PC_MIPS;
$demand[$SU_Rpy][$PC] = 300 * $K / $PC_MIPS;

$demand[$Req_CD][$FS] = 500 * $K / $AS_MIPS;
$demand[$Req_RQ][$FS] = 700 * $K / $AS_MIPS;
$demand[$Req_SU][$FS] = 100 * $K / $AS_MIPS;
$demand[$CD_Msg][$FS] = 350 * $K / $AS_MIPS;
$demand[$RQ_Msg][$FS] = 350 * $K / $AS_MIPS;
$demand[$SU_Msg][$FS] = 350 * $K / $AS_MIPS;

$demand[$GT_Snd][$GW] = 500 * $K / $LB_MIPS;
$demand[$GT_Rcv][$GW] = 500 * $K / $LB_MIPS;

$demand[$MF_CD][$MF] = 5000 * $K / $DB_MIPS;
$demand[$MF_RQ][$MF] = 1500 * $K / $DB_MIPS;
$demand[$MF_SU][$MF] = 2000 * $K / $DB_MIPS;

# Packets generated at each of the following sources
$demand[$LAN_Tx][$PC] = 2 * $K * $LAN_INST / $LAN_RATE;
$demand[$LAN_Tx][$FS] = 2 * $K * $LAN_INST / $LAN_RATE;
$demand[$LAN_Tx][$GW] = 2 * $K * $LAN_INST / $LAN_RATE;

# Parallel web servers
for ($i = 0; $i < $WEB_SERVS; $i++) {
    $demand[$Req_CD][$FDarray[$i]->{id}] = (1.0 * 0.5 /
        $WEB_OPS) / $WEB_SERVS;
    $demand[$Req_RQ][$FDarray[$i]->{id}] = (1.5 * 0.5 /
        $WEB_OPS) / $WEB_SERVS;
    $demand[$Req_SU][$FDarray[$i]->{id}] = (0.2 * 0.5 /
        $WEB_OPS) / $WEB_SERVS;
    $demand[$CD_Msg][$FDarray[$i]->{id}] = (1.0 * 0.5 /
        $WEB_OPS) / $WEB_SERVS;
}

```

```

$demand[$RQ_Msg][$FDarray[$i]->{id}] = (1.5 * 0.5 /
    $WEB_OPS) / $WEB_SERVS;
$demand[$SU_Msg][$FDarray[$i]->{id}] = (0.5 * 0.5 /
    $WEB_OPS) / $WEB_SERVS;
}

# RDBMS disk arrays
for ($i = 0; $i < $DB_DISKS; $i++) {
    $demand[$MF_CD][$MDarray[$i]->{id}] = (2.0 / $DB_IOS) /
        $DB_DISKS;
    $demand[$MF_RQ][$MDarray[$i]->{id}] = (4.0 / $DB_IOS) /
        $DB_DISKS;
    $demand[$MF_SU][$MDarray[$i]->{id}] = (1.0 / $DB_IOS) /
        $DB_DISKS;
}

# Start building the PDQ model
pdq::Init($scenario);

# Define physical resources as queues
$nodes = pdq::CreateNode("PC", $pdq::CEN, $pdq::FCFS);
$nodes = pdq::CreateNode("LB", $pdq::CEN, $pdq::FCFS);
for ($i = 0; $i < $WEB_SERVS; $i++) {
    $nodes = pdq::CreateNode($FDarray[$i]->{label},
        $pdq::CEN, $pdq::FCFS);
}
$nodes = pdq::CreateNode("AS", $pdq::CEN, $pdq::FCFS);
$nodes = pdq::CreateNode("DB", $pdq::CEN, $pdq::FCFS);
for ($i = 0; $i < $DB_DISKS; $i++) {
    $nodes = pdq::CreateNode($MDarray[$i]->{label}, $pdq::CEN,
        $pdq::FCFS);
}
$nodes = pdq::CreateNode("LAN", $pdq::CEN, $pdq::FCFS);

# Assign transaction names
$txCD = "CatDsply";
$txRQ = "RemQuote";
$txSU = "StatusUp";
$dumCD = "CDBkgnd ";
$dumRQ = "RQBkgnd ";
$dumSU = "SUBkgnd ";

# Define focal PC load generator
$streams = pdq::CreateOpen($txCD, 1 * 4.0 / 60.0);
$streams = pdq::CreateOpen($txRQ, 1 * 8.0 / 60.0);
$streams = pdq::CreateOpen($txSU, 1 * 1.0 / 60.0);

# Define the aggregate background workload
$streams = pdq::CreateOpen($dumCD, ($USERS - 1) * 4.0 / 60.0);

```

```

$streams = pdq::CreateOpen($dumRQ, ($USERS - 1) * 8.0 / 60.0);
$streams = pdq::CreateOpen($dumSU, ($USERS - 1) * 1.0 / 60.0);
#-----
# CategoryDisplay request + reply chain from workflow diagram
#-----
pdq::SetDemand("PC", $txCD,
    $demand[$CD_Req] [$PC] + (5 * $demand[$CD_Rpy] [$PC]));
pdq::SetDemand("AS", $txCD,
    $demand[$Req_CD] [$FS] + (5 * $demand[$CD_Msg] [$FS]));
pdq::SetDemand("AS", $dumCD,
    $demand[$Req_CD] [$FS] + (5 * $demand[$CD_Msg] [$FS]));
for ($i = 0; $i < $WEB_SERVS; $i++) {
    pdq::SetDemand($FDarray[$i]->{label}, $txCD,
        $demand[$Req_CD] [$FDarray[$i]->{id}] +
        (5 * $demand[$CD_Msg] [$FDarray[$i]->{id}]));
    pdq::SetDemand($FDarray[$i]->{label}, $dumCD,
        $demand[$Req_CD] [$FDarray[$i]->{id}] +
        (5 * $demand[$CD_Msg] [$FDarray[$i]->{id}]));
}
pdq::SetDemand("LB", $txCD, $demand[$GT_Snd] [$GW] +
    (5 * $demand[$GT_Rcv] [$GW]));
pdq::SetDemand("LB", $dumCD, $demand[$GT_Snd] [$GW] +
    (5 * $demand[$GT_Rcv] [$GW]));
pdq::SetDemand("DB", $txCD, $demand[$MF_CD] [$MF]);
pdq::SetDemand("DB", $dumCD, $demand[$MF_CD] [$MF]);
for ($i = 0; $i < $DB_DISKS; $i++) {
    pdq::SetDemand($MDarray[$i]->{label}, $txCD,
        $demand[$MF_CD] [$MDarray[$i]->{id}]);
    pdq::SetDemand($MDarray[$i]->{label}, $dumCD,
        $demand[$MF_CD] [$MDarray[$i]->{id}]);
}
# NOTE: Synchronous process execution causes data for the CD
# transaction to cross the LAN 12 times as depicted in the
# following parameterization of pdq::SetDemand.
pdq::SetDemand("LAN", $txCD,
    (1 * $demand[$LAN_Tx] [$PC]) + (1 * $demand[$LAN_Tx] [$FS])
    + (5 * $demand[$LAN_Tx] [$GW]) + (5 * $demand[$LAN_Tx] [$FS]));
pdq::SetDemand("LAN", $dumCD,
    (1 * $demand[$LAN_Tx] [$PC]) + (1 * $demand[$LAN_Tx] [$FS])
    + (5 * $demand[$LAN_Tx] [$GW]) + (5 * $demand[$LAN_Tx] [$FS]));

#-----
# RemoteQuote request + reply chain ...
#-----
pdq::SetDemand("PC", $txRQ,
    $demand[$RQ_Req] [$PC] + (3 * $demand[$RQ_Rpy] [$PC]));
pdq::SetDemand("AS", $txRQ,
    $demand[$Req_RQ] [$FS] + (3 * $demand[$RQ_Msg] [$FS]));

```

```

pdq::SetDemand("AS", $dumRQ,
    $demand[$Req_RQ] [$FS] + (3 * $demand[$RQ_Msg] [$FS]));
for ($i = 0; $i < $WEB_SERVS; $i++) {
    pdq::SetDemand($FDarray[$i]->{label}, $txRQ,
        $demand[$Req_RQ] [$FDarray[$i]->{id}] +
        (3 * $demand[$RQ_Msg] [$FDarray[$i]->{id}]));
    pdq::SetDemand($FDarray[$i]->{label}, $dumRQ,
        $demand[$Req_RQ] [$FDarray[$i]->{id}] +
        (3 * $demand[$RQ_Msg] [$FDarray[$i]->{id}]));
}
pdq::SetDemand("LB", $txRQ, $demand[$GT_Snd] [$GW] +
    (3 * $demand[$GT_Rcv] [$GW]));
pdq::SetDemand("LB", $dumRQ, $demand[$GT_Snd] [$GW] +
    (3 * $demand[$GT_Rcv] [$GW]));
pdq::SetDemand("DB", $txRQ, $demand[$MF_RQ] [$MF]);
pdq::SetDemand("DB", $dumRQ, $demand[$MF_RQ] [$MF]);
for ($i = 0; $i < $DB_DISKS; $i++) {
    pdq::SetDemand($MDarray[$i]->{label}, $txRQ,
        $demand[$MF_RQ] [$MDarray[$i]->{id}]);
    pdq::SetDemand($MDarray[$i]->{label}, $dumRQ,
        $demand[$MF_RQ] [$MDarray[$i]->{id}]);
}
pdq::SetDemand("LAN", $txRQ,
    (1 * $demand[$LAN_Tx] [$PC]) + (1 * $demand[$LAN_Tx] [$FS])
    + (3 * $demand[$LAN_Tx] [$GW]) + (3 * $demand[$LAN_Tx] [$FS]));
pdq::SetDemand("LAN", $dumRQ,
    (1 * $demand[$LAN_Tx] [$PC]) + (1 * $demand[$LAN_Tx] [$FS])
    + (3 * $demand[$LAN_Tx] [$GW]) + (3 * $demand[$LAN_Tx] [$FS]));

#-----
# StatusUpdate request + reply chain ...
#-----
pdq::SetDemand("PC", $txSU, $demand[$SU_Req] [$PC] +
    $demand[$SU_Rpy] [$PC]);
pdq::SetDemand("AS", $txSU, $demand[$Req_SU] [$FS] +
    $demand[$SU_Msg] [$FS]);
pdq::SetDemand("AS", $dumSU, $demand[$Req_SU] [$FS] +
    $demand[$SU_Msg] [$FS]);
for ($i = 0; $i < $WEB_SERVS; $i++) {
    pdq::SetDemand($FDarray[$i]->{label}, $txSU,
        $demand[$Req_SU] [$FDarray[$i]->{id}] +
        $demand[$SU_Msg] [$FDarray[$i]->{id}]);
    pdq::SetDemand($FDarray[$i]->{label}, $dumSU,
        $demand[$Req_SU] [$FDarray[$i]->{id}] +
        $demand[$SU_Msg] [$FDarray[$i]->{id}]);
}
pdq::SetDemand("LB", $txSU, $demand[$GT_Snd] [$GW] +
    $demand[$GT_Rcv] [$GW]);
pdq::SetDemand("LB", $dumSU, $demand[$GT_Snd] [$GW] +

```

```

    $demand[$GT_Rcv] [$GW]);
pdq::SetDemand("DB", $txSU, $demand[$MF_SU] [$MF]);
pdq::SetDemand("DB", $dumSU, $demand[$MF_SU] [$MF]);
for ($i = 0; $i < $DB_DISKS; $i++) {
    pdq::SetDemand($MDarray[$i]->{label}, $txSU,
        $demand[$MF_SU] [$MDarray[$i]->{id}]);
    pdq::SetDemand($MDarray[$i]->{label}, $dumSU,
        $demand[$MF_SU] [$MDarray[$i]->{id}]);
}
pdq::SetDemand("LAN", $txSU,
    (1 * $demand[$LAN_Tx] [$PC]) + (1 * $demand[$LAN_Tx] [$FS])
    + (1 * $demand[$LAN_Tx] [$GW]) + (1 * $demand[$LAN_Tx] [$FS]));
pdq::SetDemand("LAN", $dumSU,
    (1 * $demand[$LAN_Tx] [$PC]) + (1 * $demand[$LAN_Tx] [$FS])
    + (1 * $demand[$LAN_Tx] [$GW]) + (1 * $demand[$LAN_Tx] [$FS]));
pdq::SetWUnit("Trans");

pdq::Solve($pdq::CANON);
pdq::Report();

```

A key point is the use of three PDQ streams to represent the three workloads of interest, as well as three *dummy* streams to represent the other workloads as background resource consumption.

The complete standard PDQ report produced by `baseline.pl` is extremely long (about 8 pages) and is not reproduced here. The reader can find it in the PDQ code distribution available from [www.perfdynamics.com](http://www.perfdynamics.com). A more compact approach to generating the full standard PDQ report is to make use of PDQ functions like `PDQ::GetResponse()` (Sect. 6.6.9) and `PDQ::GetUtilization()` (Sect. 6.6.12) to report individually the most important transaction response times and the node utilizations. An example follows.

```
*** Resource Breakout "Client/Server Baseline" (100 clients) ***
```

| Transaction | Rmean  | R80th  | R90th  | R95th  |
|-------------|--------|--------|--------|--------|
| -----       | -----  | -----  | -----  | -----  |
| CatDsply    | 0.0431 | 0.0718 | 0.1005 | 0.1292 |
| RemQuote    | 0.0393 | 0.0655 | 0.0917 | 0.1180 |
| StatusUp    | 0.0152 | 0.0253 | 0.0354 | 0.0455 |
| CDbkgn      | 0.0416 | 0.0694 | 0.0971 | 0.1248 |
| RQbkgn      | 0.0378 | 0.0630 | 0.0882 | 0.1133 |
| SUBkgn      | 0.0139 | 0.0232 | 0.0325 | 0.0418 |

| PDQ Node      | % Busy |
|---------------|--------|
| -----         | -----  |
| 100Base-T LAN | 1.5838 |
| PC Driver     | 0.0003 |
| Appln Server  | 5.0532 |

|              |         |
|--------------|---------|
| Web Server10 | 7.5729  |
| Web Server11 | 7.5729  |
| Balancer CPU | 12.2812 |
| Database CPU | 12.1141 |
| SCSI Array20 | 6.8333  |
| SCSI Array21 | 6.8333  |
| SCSI Array22 | 6.8333  |
| SCSI Array23 | 6.8333  |

This breakout of PDQ performance metrics has also made use of the rules of thumb for the 80th, 90th and 95th percentiles discussed in Sect. 1.5.2 of Chap. 1:

1. 80th percentile is  $R_{80th} = 5 * PDQ::GetResponse() / 3;$
2. 90th percentile is  $R_{90th} = 7 * PDQ::GetResponse() / 3;$
3. 95th percentile is  $R_{95th} = 9 * PDQ::GetResponse() / 3;$

As expected, the *mean* response times  $R_{mean}$  are identical to those previously reported in the **SYSTEM Performance** section of the standard PDQ report. It can also be verified that they are the respective sums of the residence times listed in the **RESOURCE Performance** section of the PDQ report. Figure 9.7 shows that each of the baseline transaction response times are well within the SLA requirements.

The consumption of hardware resources by the aggregate of the three transactions follows the response time statistics in the breakout report, but these numbers are *total* utilizations that are useful for bottleneck ranking. Alternatively, we can identify which transaction is consuming the most resources at any PDQ node by referring to the **RESOURCE Performance** section of the standard PDQ report. This information will be useful in the subsequent sections as we apply more client load to the PDQ model. The PDQ baseline model should be validated against measurements on the actual benchmark platform with end-to-end response time statistics compared. The next step is to scale up the client load to 1,000 users.

### 9.4.2 Client Scaleup

To assess the impact of scaling up the number of user to 1,000 clients in PDQ is simply a matter of changing the `$USERS` parameter (line 16) in the `cs_baseline.pl` model. However, it is better practice to copy the original `cs_baseline.pl` file to another file, e.g., `cs_scaleup.pl` and make the edits to that version.

If you persist in making a succession of edits to the same PDQ model file, there will inevitably come a point where you can no longer recall what the succession of changes mean or what motivated them in the first place. The best practice is to keep separate PDQ model files for each set of scenario parameters.

A result of scaling the client load to 1,000 in `cs_scaleup.pl` and running that scenario is the following PDQ error message:

```
ERROR in model:" 122.81% (>100%)" at canonical():
Total utilization of node LB is 122.81% (>100%)
```

which tells us that the PDQ node LB representing the *load balancer* in Fig. 9.6 is oversaturated ( $\rho > 1$ ). This would make the denominator in the response time formula (2.35) negative, as well as render other calculations meaningless. Therefore, PDQ does not attempt solve the model.

### 9.4.3 Load Balancer Bottleneck

The SPEC CPU2000 rating of the load balancer is 499 in Table 9.1. We consider an upgrade scenario (line 21 in the PDQ Perlcode) where the load balancer is replaced by a model that has a rating of 792 SPECint2000. The parameter change is made in the file called `cs_upgrade1.pl` but causes the following PDQ error report when run:

```
ERROR in model:" 121.14% (>100%)" at canonical():
Total utilization of node DB is 121.14% (>100%)
```

which tells is that the PDQ node (DB) representing the *database server* in Fig. 9.6 is over-saturated.

### 9.4.4 Database Server Bottleneck

The SPEC CPU2000 rating of the database server is 479 in Table 9.1. We consider an upgrade scenario (line 22 in the PDQ Perlcode) where the database server is replaced by a model which has a CPU rating of 792 SPECint2000. The parameter change is made in the file called `cs_upgrade2.pl`, which when run produces the following performance report:

```
*** Resource Breakout "Client/Server Upgrade2" (1000 clients) ***
```

| Transaction | Rmean  | R80th  | R90th  | R95th  |
|-------------|--------|--------|--------|--------|
| -----       | -----  | -----  | -----  | -----  |
| CatDsply    | 0.0986 | 0.1643 | 0.2300 | 0.2958 |
| RemQuote    | 0.1022 | 0.1704 | 0.2386 | 0.3067 |
| StatusUp    | 0.0319 | 0.0532 | 0.0745 | 0.0958 |
| CDbkgnd     | 0.0971 | 0.1619 | 0.2267 | 0.2914 |
| RQbkgnd     | 0.1007 | 0.1678 | 0.2350 | 0.3021 |
| SUBkgnd     | 0.0307 | 0.0512 | 0.0716 | 0.0921 |

| PDQ Node      | % Busy  |
|---------------|---------|
| 100Base-T LAN | 15.8379 |
| PC Driver     | 0.0000  |
| Appln Server  | 50.5320 |
| Web Server10  | 75.7292 |
| Web Server11  | 75.7292 |
| Balancer CPU  | 62.1776 |
| Database CPU  | 72.8978 |
| SCSI Array20  | 68.3333 |
| SCSI Array21  | 68.3333 |
| SCSI Array22  | 68.3333 |
| SCSI Array23  | 68.3333 |

We see that at 1,000 users, the mean and the 95th percentile response times still do not exceed the 0.5000 s SLA requirement. The *Web server*, however, is likely to become a bottleneck at production-level loads.

#### 9.4.5 Production Client Load

We increment the client load to 1,500 and make some additional parameter changes (for reasons that lie outside the scope of this discussion) in the PDQ file called `cs_upgrade3.pl`:

```

1 #!/usr/bin/perl
2 #
3 ##  cs_upgrade3.pl
4
5 use pdq;
6
7 #####
8 # PDQ model parameters
9 #####
10 $scenario          = "Client/Server Upgrade3";
11
12 # Useful multipliers ...
13 $K                 = 1024;
14 $MIPS              = 1E6;
15
16 $USERS             = 1500;
17 $WEB_SERVS         = (2 + 4);
18 $DB_DISKS          = (4 + 1);
19 $PC_MIPS            = (499 * MIPS);
20 $AS_MIPS            = (792 * MIPS);
21 $LB_MIPS            = (1056 * MIPS);
22 $DB_MIPS            = (1244 * MIPS);
23 $LAN_RATE           = (100 * Mbps);

```

```

24 $LAN_INST           = 4; # scale factor
25 $WS_OPS             = 400; # Web server SPEC Web99 ops/sec
26 $MS_DKIOS          = 250; # RDBMS SCSI IOs

```

The following performance report is produced:

```
*** Resource Breakout "Client/Server Upgrade3" (1500 clients) ***
```

| Transaction | Rmean  | R80th  | R90th  | R95th  |
|-------------|--------|--------|--------|--------|
| -----       | -----  | -----  | -----  | -----  |
| CatDsply    | 0.0948 | 0.1579 | 0.2211 | 0.2843 |
| RemQuote    | 0.1233 | 0.2056 | 0.2878 | 0.3700 |
| StatusUp    | 0.0364 | 0.0607 | 0.0850 | 0.1093 |
| CDbkgnd     | 0.0933 | 0.1555 | 0.2178 | 0.2800 |
| RQbkgnd     | 0.1218 | 0.2030 | 0.2842 | 0.3654 |
| SUBkgnd     | 0.0352 | 0.0587 | 0.0822 | 0.1056 |

| PDQ Node      | % Busy  |
|---------------|---------|
| -----         | -----   |
| 100Base-T LAN | 23.7568 |
| PC Driver     | 0.0000  |
| Appln Server  | 75.7980 |
| Web Server10  | 37.8646 |
| Web Server11  | 37.8646 |
| Web Server12  | 37.8646 |
| Web Server13  | 37.8646 |
| Web Server14  | 37.8646 |
| Web Server15  | 37.8646 |
| Balancer CPU  | 70.3030 |
| Database CPU  | 69.9678 |
| SCSI Array20  | 82.0000 |
| SCSI Array21  | 82.0000 |
| SCSI Array22  | 82.0000 |
| SCSI Array23  | 82.0000 |
| SCSI Array24  | 82.0000 |

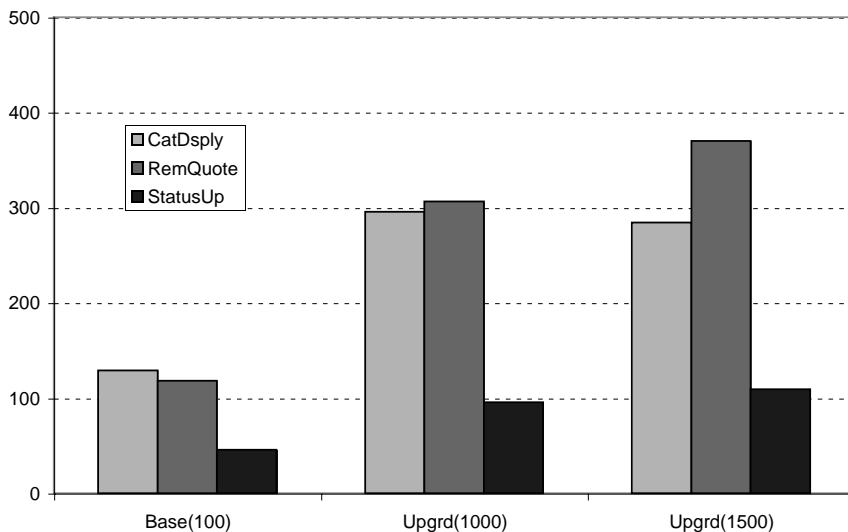
The impact of these upgrades on each of the response time metrics compared to the baseline benchmark system is summarized in Fig. 9.7.

We see the SCSI disk array becoming the next bottleneck. With that in mind, we consider the last of the scenario objectives in Sect. 9.3.1.

#### 9.4.6 Saturation Client Load

Maintaining the same system parameters as those in Sect. 9.4.5, we adjust the \$USERS parameter to find where the PDQ model reaches saturation.

```
*** Resource Breakout "Client/Server Upgrade4" (1800 clients) ***
```



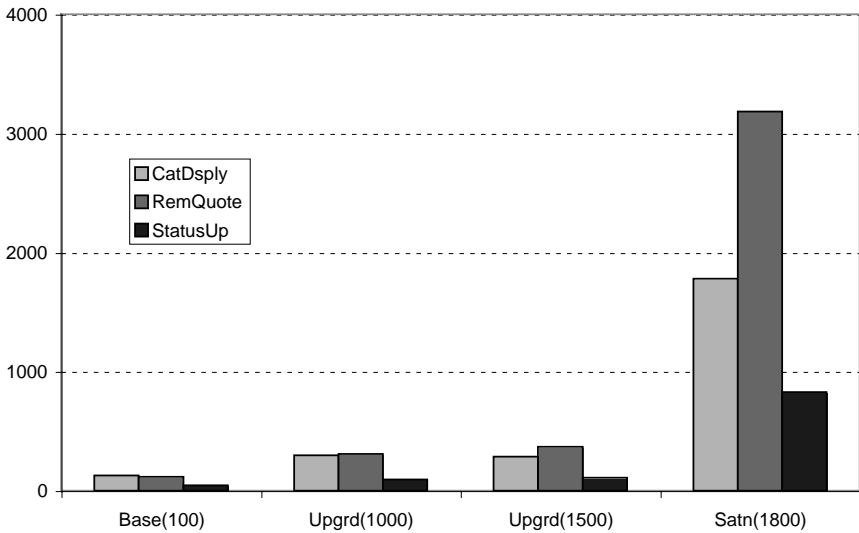
**Fig. 9.7.** Summary of the response time (ms) statistics for baseline client/server performance together with two of the possible upgrade scenarios presented in Sect. 9.4

| Transaction | Rmean  | R80th  | R90th  | R95th  |
|-------------|--------|--------|--------|--------|
| CatDsply    | 0.5930 | 0.9883 | 1.3837 | 1.7790 |
| RemQuote    | 1.0613 | 1.7689 | 2.4764 | 3.1840 |
| StatusUp    | 0.2762 | 0.4603 | 0.6445 | 0.8286 |
| CDbkgnd     | 0.5916 | 0.9859 | 1.3803 | 1.7747 |
| RQbkgnd     | 1.0598 | 1.7663 | 2.4728 | 3.1794 |
| SUBkgnd     | 0.2750 | 0.4583 | 0.6416 | 0.8249 |

| PDQ Node      | % Busy  |
|---------------|---------|
| 100Base-T LAN | 28.5082 |
| PC Driver     | 0.0000  |
| Appln Server  | 90.9576 |
| Web Server10  | 45.4375 |
| Web Server11  | 45.4375 |
| Web Server12  | 45.4375 |
| Web Server13  | 45.4375 |
| Web Server14  | 45.4375 |
| Web Server15  | 45.4375 |
| Balancer CPU  | 84.3636 |
| Database CPU  | 83.9614 |
| SCSI Array20  | 98.4000 |

|              |         |
|--------------|---------|
| SCSI Array21 | 98.4000 |
| SCSI Array22 | 98.4000 |
| SCSI Array23 | 98.4000 |
| SCSI Array24 | 98.4000 |

We determine that around 1,800 users both the application servers and the database disk arrays are nearing saturation, even with all of the previous upgrades in place. And naturally, nearly every response time statistic grossly exceeds the SLA objective. A comparison of all the response times (in ms) is summarized in Fig. 9.8.



**Fig. 9.8.** Response times (ms) including those predicted when the system reaches saturation with 1,800 users

#### 9.4.7 Per-Process Analysis

An analysis of the transaction times can also be carried out at the per-process level by further inspection of the PDQ report. For example, the time taken by the `$CD_Msg` process can be assessed as follows.

Using any of the PDQ model files, a global search for the string `CD_Msg` reveals that it runs on both the *application servers* (AS) and the *Web servers* (WS). That is also consistent with the process flows shown in Fig. 9.5. Without loss of generality, we focus on the `$CD_Msg` process executing on the application server in the *baseline* configuration. Specifically, line 20 of `baseline.rpt` states:

```
20 CEN FCFS AS          CatDsply  TRANS    0.0029
```

which corresponds to a service demand of 2.9 ms for the `$CD_Msg` process running the application server. In the presence of contention from other work, however, the residence time at the application server has become 3.1 ms for the `$CD_Msg` process, as indicated at line 196 of the PDQ report:

```
196 Residence Time AS  CatDsply    0.0031    Sec
```

By the time we get to the production loads of Sect. 9.4.5 with 1500 users, this time has grown to 12 ms:

```
205 Residence Time AS  CatDsply    0.0120    Sec
```

In other words, the effective `$CD_Msg` process *stretch factor* is 4 times the baseline service demand due to increased queueing contention (waiting time). The complete PDQ report for this scenario is not shown here but is available for download from [www.perfdynamics.com](http://www.perfdynamics.com).

## 9.5 Review

In this chapter, we have seen how to apply PDQ to the performance analysis of a multitier B2C client/server environment. A key point to note is that PDQ can be used to predict the scalability of distributed *software* applications, not just hardware as in Chap. 7. This is achieved by using the workflow analysis of Sect. 9.3.3.

Another merit of the techniques presented in this chapter pertains to more cost-effective benchmarking and load testing. The performance of many large-scale benchmark configurations can be predicted using PDQ, and those results only need be verified against a relatively sparse set of selected platform configurations. PDQ offers another way to keep the cost of load testing and benchmarking down.

## Exercises

**9.1.** How do the predicted performance metrics change in the PDQ model `cs_baseline.pl` if there is just a single workload, rather than the two-class workload discussed in this chapter?

**9.2.** How does the predicted performance outcome change in the PDQ model `cs_upgrade4.pl` if the ordering of the hardware components is reversed in the Perlcode?

