PERFORMANCE CONCEPTS

This chapter attempts to lay a foundation for software performance engineers by introducing the fundamental concepts that are encountered in computer performance evaluation. It defines key terms like response time, bandwidth and throughput that are used throughout this book. Many of the fundamental concepts discussed here are mathematical in nature. A few simple formulas are discussed, including the Utilization Law that shows the relationship between request rates, service times, and resource utilization. Some basic concepts from analytic queuing theory are also discussed, including Little's Law that relates request rates, response times and queue depth.

Insights from queuing models are especially useful in understanding and predicting the behavior of computer applications under the impact of load. Experienced computer performance professionals realize that computer systems seldom degrade gracefully. Sometimes the root cause of this sudden degradation is simply an overloaded resource, a *bottleneck* that constricts or disrupts the regular flow of requests and responses. So long as the load on this bottlenecked resource remains below its critical threshold, performance of the application is adequate. But once this a threshold is crossed, the application can be subject to severe degradation. What is even more diabolical is that once this critical threshold is reached, a very small increase in the amount of work will lead to a disproportionately large increase in the response time of the application.

Sometimes the root cause is a flaw in the code being used to solve the problem. For example, any algorithm characterized as NP-complete has the fatal flaw that as the problem domain expands, the amount of computation expands exponentially, potentially rendering the algorithm unusable as the problem space grows larger and larger. Sometimes the problem results from a mistake or an oversight, such as the use of an inefficient algorithm wasn't noticed unit testing until the result set being operated upon grew sufficiently large.

One familiar and very well documented example of this phenomenon refers back to the early days of the Internet in the mid-1980s when it would suffer a "collapse." As the technology associated with the Internet began to roll out to colleges and universities connected to the early network, the TCP/IP networking traffic between interconnected computer systems started to increase, the response times of the file transfer programs in use at the time suffered major degradation. This "collapse" was associated with a performance issue that rendered the original version of the Internet virtually unusable – ironically, just as the idea of using the Internet to communicate between computers of many different kinds was beginning to prove very popular.

Networking engineers began to understand that the root cause of this serious performance problem was not the overall capacity of the network to handle the traffic between the network connections. It was related more to the *variability* in the capacity of the different computers that were connected with each other. More specifically, these problems were associated with dropped packets that led to an increase in the number of retransmitted packets whenever a very powerful Sender machine was trying to send a large amount of data to a much less powerful Receiver endpoint. Ultimately, this understanding led to the development of the *congestion control* algorithms that were then added to the TCP protocol.

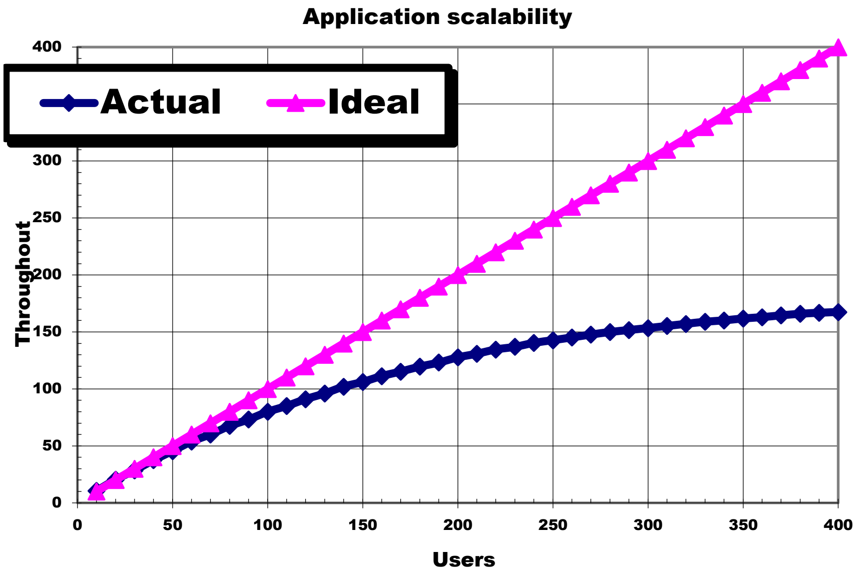
We begin by discussing *scalability* – the ability of an application to sustain levels of performance in the face of more customers, larger data volumes, or more usage in general. Whenever computer professionals are discussing application or hardware scalability, they are concerned with core computer performance and capacity planning issues.

SCALABILITY

Computer professionals worry about *scalability* based on their experience that many computer systems encounter performance problems as the number of users of those systems grows. Computer equipment today is extremely powerful, yet performance problems have not disappeared. The computing resources you have in place are finite. They have definite limits on their processing capability. *Scalability* concerns what those finite limitations are.

Figure 2.1 shows two scalability curves. The left hand y-axis represents some measure of work being performed by the computer — it could be database transactions per second, disk I/Os per second, files transferred, web visitors per hour, or e-mail messages processed per minute. The horizontal x-axis shows the growth in the number of users of this application.

Figure 2.1. Ideal vs. actual application scalability.



The first curve is a straight line that shows performance increasing *linearly* as a function of the number of users. This is the ideal that application architects strive for. Ideally, as the number of concurrent users of an application grows, the user experience does not degrade due to elongated or erratic response times. The second curve models the performance obstacles actual system encounter as the workload grows. Initially, the actual throughput curve diverges very little from the ideal case. But as the number of users grows, actual performance levels tend to be *non-linear* with respect to the number of users. As more users are added and the system reaches its capacity limits, the throughput curve eventually plateaus, as illustrated.

Since scalability considerations emerge with almost every widely used application as it grows, you would think experienced developers would anticipate these problems and adjust accordingly. Unfortunately, that is too frequently not what happens. There is often intense pressure on the development team to deliver a functionally complete application on time and under the budget, which inevitably leads to compromises and short cuts in the quality of the software. It is also true that performance concerns may not arise until the latter stages of the application development life cycle. Experienced developers may warn about the risk of "premature optimization," making critical design decisions with too little information about the state of the final application. Relegating most performance testing and stress testing to the final stages of development is even riskier, however. Superior results are almost always achieved when rigorous and robust performance testing is performance throughout the application development life cycle.

Deferring large scale performance testing to the latter stages of development also has the undesirable effect of conjoining late stage customer Acceptance Testing with performance testing. Moreover, when computers applications are initially deployed, the number of users in the beginning stages is frequently quite small. Initially, because the current system is not close to pushing up against any of its capacity limits, response times, or, more generally, *service levels* remain at acceptable levels. But, as users are added and usage of the application increases, performance problems are inevitably encountered.

Scalability is a core concern whenever you are planning for an application deployment that must accommodate a large number of users. Since computer hardware and software have finite processing limits, the non-linear behavior illustrated in Figure 2.1 — which is evidence of some form of performance degradation — can be expected at some point. The focus of computer capacity planning is to determine with some precision at what point, as the number of users grows, there is enough performance degradation that it begins to interfere with the smooth operation of this application.

The branch of software engineering most associated with performance investigations – figuring out where the capacity limits of a current application lie and designing a new application or feature so that it meets its performance goals – is known as *software performance engineering*. In many organizations, performance engineers are also the team responsible for late stage performance stress testing for a new application or feature as a part of certifying the quality of new feature prior to releasing it.

Inevitably, computer systems reach their capacity limits. At that point when more users are added, they no longer scale linearly. The focus of computer capacity planning for real world workloads, of course, is to anticipate at what point serious performance degradation can be expected. Once you understand the characteristics of your workload and the limitations of the computer environment in which it runs, you ought to be able to forecast the capacity limits of an application server.

In many instances, you can use stress testing tools to simulate a growing workload until you encounter the capacity limits of your hardware. In simulated *benchmark* runs using a stress testing tool, from which the throughput curves in Figure 2.1 are drawn, the number of users is increased steadily until tell-tale signs of non-linear scalability appear. Stress testing your important applications to determine at what point serious performance degradation occurs is one effective approach to capacity planning. There are also analytic and modeling approaches to capacity planning that are effective. The mathematical relationships between key performance measurements that are discussed in the next section of this chapter are the basis for the analytic approaches.

For example, suppose you are able to measure the following:

- The current utilization of a potentially saturated resource like a processor, disk, or network interface
- The average contribution from an individual user to that's resource's utilization, and
- The rate at which the number of application users is increasing

Using a simple formula called the Utilization Law, which is defined below, you will be able to estimate the number of users it will take to drive the designated resource to its capacity limits when it is bound to become a performance *bottleneck*. Both stress testing your application and the analytic techniques lead to a prediction about when you are about to run out of capacity. Once you understand the circumstances that could cause you to run out of capacity, you ought to be able to formulate a strategy to cope with the problem in advance.

> **Caution**
>
> Many published articles that discuss the application scalability display graphs of performance levels that are reported as a function of an ever increasing number of connected users, similar to Figure 2.1. These articles often compare two or more similar applications and show which has the better performance. They are apparently intended to provide capacity planning guidance. But unless the workload used in the tests matches your own, the results of these benchmarks may have little applicability to your specific problems.

Experienced computer performance analysts understand that non-linear scalability is to be expected when you reach the processing capacity at some bottlenecked resource. You can expect that computer performance will cease scaling linearly suddenly at some point as the number of users increases. As the system approaches its capacity limits, various performance statistics that measure *throughput*, the amount of work being performed, tend to level off. Moreover, computer systems do not degrade gracefully. When a performance bottleneck develops, measures of application *response time* tend to increase very sharply.

What is particularly unforgiving about working with machines with finite capacity limits is that, when those limits are reached, *a slight increase in the amount of work that needs to be processed causing a very sharp increase in the response time*. This non-linear relationship between utilization and response time is also explored in this chapter.

Being able to observe a capacity constraint that limits the performance of some real-world application as the load increases, as illustrated in Figure 2.1, is merely the starting point of computer performance analysis. Once you understand that a bottleneck is constraining the performance of the application, your analysis should proceed to identify the component of the application (or the hardware environment that the application runs under) that is the root cause of the constraint. This book tries to provide examples and guidance on how to perform a bottleneck analysis, but you should be prepared that this is a step that may require considerable effort and skill.

If you can successfully reproduce the scalability issues an application encounters in a performance test lab environment, there is an opportunity to use *profiling* tools to investigate the source of the problem. Different types of profiling tools are useful in identifying different types of performance problems – investigating memory problems requires a very different set of performance measurements than does investigating a program that uses too much CPU time to execute. In fact, the most frequent mistake made by users of profiling tools is they select the wrong profiling tool for their particular problem. We will some profiling tools in action in the next chapter.

Once you have identified a bottleneck, you may then proceed to consider various steps that can be taken to relieve this capacity constraint on your system. This is also a step that may require considerable effort and skill. The alternatives you evaluate are likely to be very specific to the problem at hand. For example, if you determine that network capacity is a constraint on performance for one of your important applications, you will need to consider any of the practical approaches to reducing the application's network load, for example, by compressing data before it is transmitted over the wire, or, alternately, adding network bandwidth. You may also need to weigh both the potential cost and benefits of the alternatives proposed before deciding on an effective course of action to remedy the problem. Some of the factors to consider include:

- how long it will take to implement the change and bring some desperately needed relief to the situation,
- how long the change will be effective, considering the current growth rate in the application's usage, and
- how to pay for the change, assuming there are additional costs involved in making the change (i.e., additional hardware or software that must be procured, etc.)

Bottleneck analysis is a proven technique that can be applied to diagnose and resolve a wide variety of performance problems. Your success in using this technique depends on gathering the relevant performance statistics you will need to understand where the bottleneck is. Effective performance monitoring procedures are a necessary first step. Understanding how to interpret the performance information you gathered is also quite important.

In benchmark runs, simulated users continue to be added to the system beyond its saturation point. Since these articles only report on the behavior of simulated "users," they can safely ignore the impact on real customers and how they react to a computer system that has reached its capacity limits. In real life, you must deal with dissatisfied customers that are reacting harshly to erratic performance conditions. There may also be serious economic considerations associated with performance degradations. Workers that rely on computer systems to get their daily jobs done on time will lose productivity. Customers that rely on your computer-based applications may become so frustrated that they start to turn to your competitors for better service. In an e-commerce application, conversion rates may suffer. When important business applications reach the limits of their scalability on current hardware and software, one of those crisis-mode interventions discussed in the Introduction is likely to ensue.

KEY PERFORMANCE MEASUREMENTS

This section introduces the standard computer performance terminology that will be used often in the remaining chapters of this book. Before you are able to apply the practices and procedures that are recommended here, it is a good idea to acquire some familiarity with these basic computer measurement concepts.

Computers are electronic machines designed to perform calculations and other types of arithmetic and logical operations. The components of a computer system – its Central Processing Unit (CPU) or processor, disks, Network interface card, etc. – that actually perform the work are known generically as its *resources*. Each resource has a finite *capacity* to perform designated types of work. *Customers* generate work requests for a server machine (or machines) to perform. In this book, we are concerned primarily with server machines designed to process requests from multiple customers concurrently. Nevertheless, the same concepts apply to single user workstations and mobile devices, as well. In analyzing the performance of a particular computer system with a given workload, we need to measure

- the *capacity* of those machines to perform this work,
- the *rate* at which they are currently performing it, and
- the *time* it takes to complete specific tasks.

The next section defines the terms that are commonly used to describe computer performance and capacity and describes how they are related to each other.

Definitions

It is almost a cliché to declare that you cannot manage what you cannot measure, but this bit of wisdom certainly applies to computer performance management. To be able to evaluate the performance of a computer system, you need a thorough understanding of the important metrics used to *measure* computer performance. Computers are machines designed to perform work, and we measure their capacity to perform the work, the rate at which they are currently performing it, and the time it takes to perform specific tasks.

Collectively, the measurements of computer performance we gather at specific points in time represent a *time series*, a sequence gathered at pre-determined intervals or when specific events of interest occur. We will see in a later section that the clocks and timer facilities available in Windows to use for our measurements have special considerations of their own. For example, each socket in a multi-socket server relies on its own on-board clock. The separate clocks on the different sockets are not synchronized and are subject to some amount of drift over time. Consequently, within a multi-socket computer there is no single clock value that can be referred to that is absolutely authoritative.

You can try to synchronize the clock on your computer periodically to an authoritative radio time signal using the Network Time Protocol, or NTP. For instructions on how to set up NTP on Windows, see the Knowledge Base article “How to configure an authoritative time server in Windows Server.” NTP, experts say, should allow the computer’s clock to be accurate to within about 30-50 milliseconds.[1] For two computers located in the same data center, even a disparity as small as 30 milliseconds can be problematic for some fine-grained measurements made on computers that are capable of executing millions of instructions in that time interval. For two different computers located in two different regions of the world, even if both attempt to synchronize to the same authoritative remote time source, you need to accept the fact that time itself is relative in the physical universe.[2]

Most performance problems can be analyzed in terms of the computer resources utilized, queues, service requests, and response time. This section defines these basic performance measurement concepts. It describes what they mean and how they are related.

Two of the key measures of computer capacity are *bandwidth* and *throughput*. Bandwidth is a measure of capacity, the rate at which work can be completed, while throughput measures the rate at which work requests are completed. *Scalability*, as discussed in the previous section, is often defined as the throughput of the machine or device as a function of the total number of Users requesting service. How busy the various *resources* of a computer system get is known as their *utilization*. How much work resources can process at their maximum level of utilization is defined as their *capacity*.

The key measures of the time it takes to perform specific tasks are *queue time*, *service time*, and *response time.* The term, *latency* is often used in an engineering context to refer to either service time or response time. Response time will be used consistently here to refer to the sum of service time and queue time. In networks, another key measure is *round trip time*, the amount of time it takes to send a message and receive a confirmation message (an *Acknowledgement* or *ACK*, for short) in reply.

1.

[1] The National Institute of Standards and Technology publishes very detailed specifications on the authoritative radio time signal that is maintained for use in the United States. See “How Accurate is a Radio Controlled Clock?” for more details.

[2] Reportedly, it was while examining patent applications for inventions that attempted to address the problem of clock synchronization for geographically distributed railroad stations across Europe for the Swiss Patent Office where he worked shortly after graduating with a degree in Physics that stimulated Albert Einstein to formulate his Theory of Relativity. For an interesting historical perspective on the problem of time synchronization, see Peter Galison, *Einstein's Clocks and Poincare's Maps: Empires of Time*.

When a work request arrives at a busy resource and cannot be serviced immediately, the request is then queued. Queued requests are subject to a *queue time* delay before they are serviced. The number of requests delayed waiting for service is known as the *queue length*.

Note

The way terms like response time, service time, and queue time are defined here is consistent with the way these same terms are defined and used in Queuing Theory, a formal, mathematical approach that is used widely in computer performance analysis. See the References section at the conclusion of this chapter for more information about Queuing Theory.

THE ELEMENTS OF A QUEUING SYSTEM

Figure 2.2 illustrates the elements of a simple queuing model. It depicts customer requests arriving at a server for processing. This example illustrates customer requests for service arriving intermittedly. The customer requests are for different amounts of service. Because individual requests are independent of each other, both the service request *arrival rate* and service time distributions are *non-uniform*. The server that is depicted could be a processor, disk, or Network Interface card (NIC). If the device is free when the request arrives, it goes into service immediately. If the device is already busy servicing some previous request, the request is queued. Service time refers to time spent at the device while the request is being processed. Queue time represents time spent waiting in the queue until the server becomes available. Response time is the sum of both service time and queue time. How busy the server gets is its *utilization*.

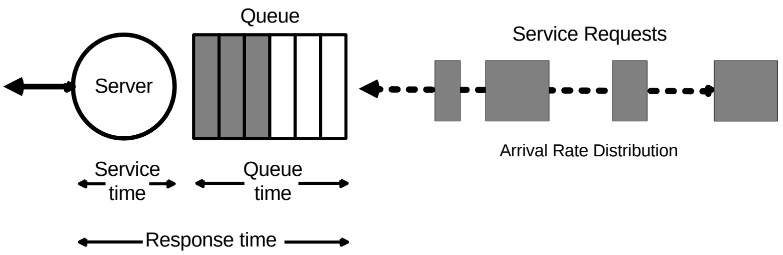


Figure 2.2. The elements of a queuing system.

The computer resource and its queue of service requests depicted in Figure 2.2 leads to a set of mathematical formulas that can characterize the performance of this queuing system. Some of these basic formulas in queuing theory are described below. Of course, this model is too simple. Real computer systems are much more complicated. They have many resources, not one, and these resources are interconnected. At a minimum, you might want to depict some of these additional resources, including the processor, one or more disks and Network Interface cards. Conceptually, these additional components can be linked together in a network of queues, as illustrated in Figure 2.3. Computer scientists can successfully model the performance of complex computer systems using *queuing networks* such as the one depicted here in Figure 2.3. When specified in sufficient detail, queuing networks, similar to the one illustrated, can model the performance of complex computer systems with great accuracy.

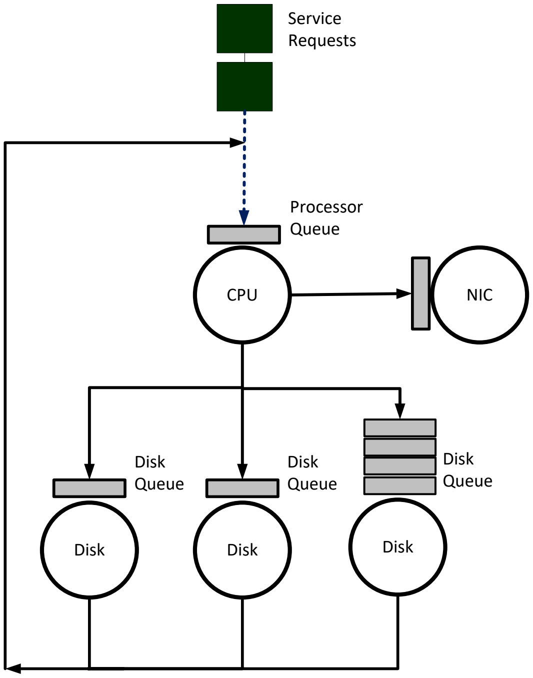


Figure 2.3 A simple computer system modeled as a network of queues.

BANDWIDTH

Bandwidth measures the capacity of a link, bus, channel, interface, or the device itself to transfer data. It is usually measured in either bits/second or bytes/second (where there are eight bits in a data byte). For example, the bandwidth of a 1000BaseT Ethernet connection is 1000 Mbits/sec (Mega*bits*), the bandwidth of a Serial ATA 3.0 disk is 600 MBytes/sec (Mega*Bytes*), the bandwidth of a USB 3.0 Port is 625 MB/sec, and the bandwidth of the PCI-X 3.0 32-bit interface is 64 GBytes/sec. (Wikipedia contains a table of device bit rates that is a useful reference.)

Bandwidth usually refers to the maximum theoretical data transfer rate of a device under ideal operating conditions. Therefore, it is an *upper-bound* on actual performance. You are seldom able to measure the device actually performing at its full rated bandwidth. Devices do not obtain their advertised performance level because there is often *overhead* associated with servicing work requests. For example, you can expect operating system overhead, protocol message processing time, and delay in disk positioning to absorb some of the available bandwidth for each request to read or write a disk. Packet-oriented data transfer protocols, for example, automatically add some amount of error detection bits to the bit stream as they transfer it to the disk. These overhead factors mean that the application can seldom use the full rated bandwidth of a disk for data transfer.

For another example, various protocol headers and the overhead of adding error detection bits associated with the network communication protocols reduce the theoretical capacity of a 1000 Mbit/sec Gigabit Ethernet link to significantly less than 100 megabytes/sec. Consequently, it is usually more realistic to discuss *effective bandwidth* or *effective capacity* — the amount of work that can be accomplished using the device under real world conditions.

THROUGHPUT

Throughput measures the rate work requests are completed, from the point of view of some observer. Examples of throughput measurements include the number of reads per second from the disk or file system, the number of instructions per second executed by the processor, HTTP requests processed by a Web server, and transactions per second that can be processed by a database engine.

Throughput and bandwidth are very similar. Bandwidth is often construed as the *capacity* of the system to perform work, while throughput is the current *observed rate* at which that work is being performed.

UTILIZATION

Utilization measures the fraction of time that a device is *busy* servicing requests, usually reported as a percent busy. Utilization of a device varies between 0 and 1, where 0 is idle and 1 (or 100%) represents utilization of the full bandwidth of the device. It is customary to report that the processor or CPU is 75% busy or the disk is 40% busy. It is not possible for a single device to ever appear greater than 100% busy.

Measures of resource utilization are commonly reported as Windows performance counters. Later in this book, many of the specific resource utilization measurements that you are able to gather on your Windows machines will be described. You can easily find out how busy the processors, disks, and network interfaces are on your machines. I will also describe how these utilization measurements are derived by the operating system, often using indirect measurement techniques that save on overhead. Knowing how certain resource utilization measurements are derived will help you understand how to interpret them.

The utilization of a device is related directly to the observed throughput (or request rate) and service time as follows:

- ♦ *Utilization = Throughput × Service Time*

This simple formula relating device *utilization*, *throughput*, and *service time* is known as the Utilization Law. The Utilization Law makes it possible to measure the throughput and service time of a disk, for example, and derive the disk utilization.

For a simple example of the Utilization at work, consider a disk that processes 20 input/output (I/O) operations per second with an average service time of 10 milliseconds is busy processing requests 20 × 0.010 every second, and is said to be 20% busy.

Alternatively, we might measure the utilization of a device using a sampling technique while also keeping track of throughput. Using the Utilization Law, we can then derive the service time of requests. Suppose we sample a communications bus 1000 times per second and find that it is busy during 200 of those measurements, or 20%. If we measure the number of bus requests at 2000 per second over the same sampling interval, we can derive the average service time of bus requests equal to 0.2 ÷ 2000 = .0001 seconds or 100 microseconds (μsecs). Notice under these circumstances that sampling the utilization is a more efficient way to calculate the service time of the requests than measuring service times directly.

Monitoring the utilization of various hardware components is an important element of any capacity planning exercise. If an application server is currently processing 60 requests per second with a CPU utilization measured at 20%, the server apparently has considerable reserve capacity to process requests at an even higher rate. On the other hand, a server processing 60 transactions per second running at a CPU utilization of 98% is operating at or near its maximum capacity.

In forecasting your future capacity requirements based on current performance levels, understanding the resource profile of workload requests is very important. If you are monitoring an IIS web server, for example, and you measure processor utilization at 20% busy and the transaction rate at 50 HTTP Get Requests/sec, it is easy to see how you might create the following capacity forecast:

Table 2.1

HTTP Get Requests/sec	% Processor Time
50	20%
100	40%
150	60%
200	80%
250	100%

The measurements you have taken and the analysis you have performed allow you to anticipate that having to process 250 HTTP Get Requests/sec at this web site would exhaust the current processor capacity. This should then lead you to start tracking the growth of your workload, with the idea of recommending adding processor capacity as the Get Request rate approaches, say, 200 per second, for example.

You have just executed a simple capacity plan designed to cope with the scalability limitations of the current computer hardware environment for this workload. Unfortunately, computer capacity planning is rarely so simple. For example, web requests use other resources besides the CPU, and one of those other resources might reach its effective capacity limits long before the CPU becomes saturated.

But there are other complicating factors. One operating assumption in this simple forecast is that processing an HTTP Get Request in this environment requires 0.4% processor utilization, on average. This is based on your empirical observation of the current system. Other implicit assumptions in this approach include:

a. Processor utilization is a simple, linear function of the number of HTTP Get Requests being processed, and

b. The service time distribution for execution time on a processor per HTTP Get Request — the amount of Processor utilization per Request — remains constant.

Unfortunately, these implicit assumptions may not hold true as the workload changes and grows. Due to *caching* effects, for example, it is quite plausible that the amount of processor time per request may vary as the workload grows. If the caching is very effective, the average amount of processor time per request may actually decrease. If the caching loses effectiveness as the workload grows, the average amount of processor time consumed per request may increase. The amount of CPU time consumed on average per request may also change as new features are added to the application or optimizations or other improvements are introduced. It may also change depending on the hardware the application runs on, or under the impact of virtualization.

You will need to continue to monitor this system as it grows to see which of these cases holds.

The component functioning as the constraining factor on throughput — in this case, the processor — is designated as the *bottlenecked device*. If you improve performance at the bottlenecked device — by upgrading to a faster component, for example — you are usually able to extend the effective capacity of the computer system to perform more work.

Tip

Measuring utilization is often very useful in detecting system bottlenecks. Bottlenecks are usually associated with processing constraints at some overloaded device. It is usually safe to assume that devices observed operating at or near their 100% utilization limits are bottlenecks, although things are not always that simple, as discussed below.

It is not always so easy to identify the bottlenecked device in a complex, inter-related computer system or network of computer systems For one thing, 100% utilization is not necessarily the target threshold for all devices.

For example, one complicating factor in assessing the capacity limits of processor hardware is the feature Intel calls Hyper-Threading or HT, which is more generically known as *simultaneous multithreading*. Simultaneous multithreading is the capability of a single processor core to process multiple instructions concurrently. Intel's HT technology can be very effective at low utilization and when the threads executing in parallel are accessing disjoint areas of memory. It is as if a single processor core can do the work of two (logical) processors. On the other hand, if the threads being executed in parallel represent similar processing tasks that share access to the same memory locations, internal processor resource conflicts can arise. Under these circumstances, the performance of a single processor core with two logical processors performs worse than the single processor core with HT disabled. (We will review an example of this later in this book.) The target utilization for HT logical processors is closer to 75% than 100%.

Other kinds of computer equipment perform more efficiently under heavier loads, especially when caching techniques are involved. These and other anomalies make the simple, straight line projections shown in Table 2.1 quite prone to error.

SERVICE TIME

Service time measures how long it takes to process a specific customer work request. Engineers alternatively often speak of the length of time processing a request as the device's *latency*, another word for delay. For example, memory latency measures the amount of time it takes for the processor to fetch data or instructions from RAM or one of its internal memory caches. Other related measures of service time are the *turnaround time* for requests, usually ascribed to longer running tasks, such as disk-to-tape backup runs. The *round trip time* is an important measure of network latency because when a request is sent to a destination across a communications link using the TCP/IP protocol, the sender must wait for a reply.

The service time of a file system request, for example, will vary based on whether the request is cached in memory or requires a physical disk operation. The service time will also vary if it is a sequential read of the disk, a random read of the disk, or a write operation. The expected service time of the physical disk request also varies depending on the block size of the request. These workload dependencies demand that you measure disk service time directly, instead of relying on projections based on some idealized model of disk performance.

The service time for a work request is often assumed to be constant, a simple function of the device's speed or its capacity. While this is largely true, under many circumstances this assumption is false. Device service times can vary as a function of utilization. Using intelligent scheduling algorithms, it is often possible for processors and disks to work more efficiently at higher utilization rates. You are able to observe noticeably better service times for these devices when they are more heavily utilized. Many of these intelligent scheduling algorithms are described in greater detail later in this book.

CACHE EFFECTS

Caching algorithms also effect service time distributions, and caching techniques are in widespread use. Caches are used inside the processor and disk hardware, for instance. Virtual memory functions like a cache across process virtual address spaces. In fact, one of the physical memory pools that the Windows Manager creates is informally known as the "Cache." A database management system like SQL Server implements extensive data caching. The use of caching is ubiquitous in web applications. Caching is performed inside the web browser client, in a Content Delivery Network (or CDN), at the IIS web level, and, potentially, inside the ASP.NET application itself.

When the data requested by the application is found in the cache, it is known as a *cache hit*. On a cache hit, the service time for the request is significantly reduced compared to a *cache miss* because it is not necessary to expend time re-creating the requested data. Due to caching effects, service time distributions are often bi-modal due to the large differential in the amount of processing needed for cache hits versus cache misses.

When an application that relies on caching initializes, the cache itself is empty of the content the application requires. So, initially, the cache is very ineffective and the cache hit ratio is near zero. This initial condition is known as a *cold start*. Then, as the application runs, the cache begins to fill and gains effectiveness, eventually reaching a steady state where cache hit rates are stable. When you run a performance stress test, it is a good practice to have a warm-up period where you allow the application under stress to execute for an initial period of time until this steady state is reached and service time measurements stabilize.

Inside the processor hardware, a thread *context switch* occurs when the processor stops executing one program thread and begins to execute another. On a context switch, the instruction cache and virtual address translation buffers of the processors are effectively flushed. For some period of time after the context switch occurs, execution of instructions from the new thread are subject to a cold start in the hardware caches. The Windows Thread Scheduler implements *processor affinity* on multiprocessors by dispatching a new thread on the same processor that it ran on last, assuming that processor is available. Processor affinity in thread scheduling is designed to ameliorate the big performance impact that context switches have due to cold starts in the processor-level caches.

Decomposition

The service time spent processing a .NET Web application request can be broken down into respective processing components: for example, time spent in the application code, time spent during processing in the business logic layer performed by .NET components, time spent in the operating system, and time spent in the database processing tier. For each one of these subcomponents, the application service time can be further decomposed into time spent at various hardware components, e.g., the CPU, the disk, the network, etc.

Decomposition is an important technique used in computer performance analysis to relate a workload to its various hardware and software processing components. To decompose application service times into their component parts, you must understand how busy various hardware components are and, specifically, how workloads contribute to that utilization. This can be very challenging in many Windows web applications due to their complexity. You may need to gather detailed trace data to map all the resources used by applications definitively back to the component parts of individual requests.

Response Time

Response time is the sum of service time and queue time:

- *Response time = service time + queue time*

Consequently, it represents both the device latency and any queuing delays that accrue while the request is queued waiting for the device. At heavily utilized devices, queue time can represent the bulk of the observed response time. Queue time is discussed in greater detail in the next section.

Application response time measurements are important for two main reasons. The first is that application response time measurements correlate with customer satisfaction. In survey after survey of customers, performance concerns usually rank just below reliability (i.e., bugs and other defects) as the factor most influential in forming either a positive or negative attitude towards an application. They are a critical aspect of software quality that you can measure and quantify.

In performance analysis, application response time measurements are also essential to apply any of the important analytic techniques that people have developed over the years for improving application response time. These techniques include using queuing models and related mathematical techniques used by capacity planners to predict response time in the face of growing workloads and changing hardware. Any form of optimization or tuning you want to apply to your application also needs to be grounded – how can you know if this or that optimization leads to an improvement if you are not measuring response times, both before and after. Even knowing which aspect of the application's response time to target requires measurements that allow you to break the response times you observe into their component parts – CPU, IO, network, etc., the analysis technique mentioned above that is known as response time *decomposition*.

Because they best encapsulate the customer's experience interacting with an application hosted on a Windows machine, measures of application response time are among the most important in computer performance and capacity planning. Quite simply, when application response time measurements are not readily available, performance management becomes very arbitrary. If you are limited to a mere anecdotal account that some application is "slow," you don't know how to interpret that report. How slow is it? Is it slow compared to yesterday, or last week, or the last time a major change was applied to the system? How many customers are experiencing this slowness? Is the problem limited to a few, select scenarios, or is this "slowness" a systemic condition?

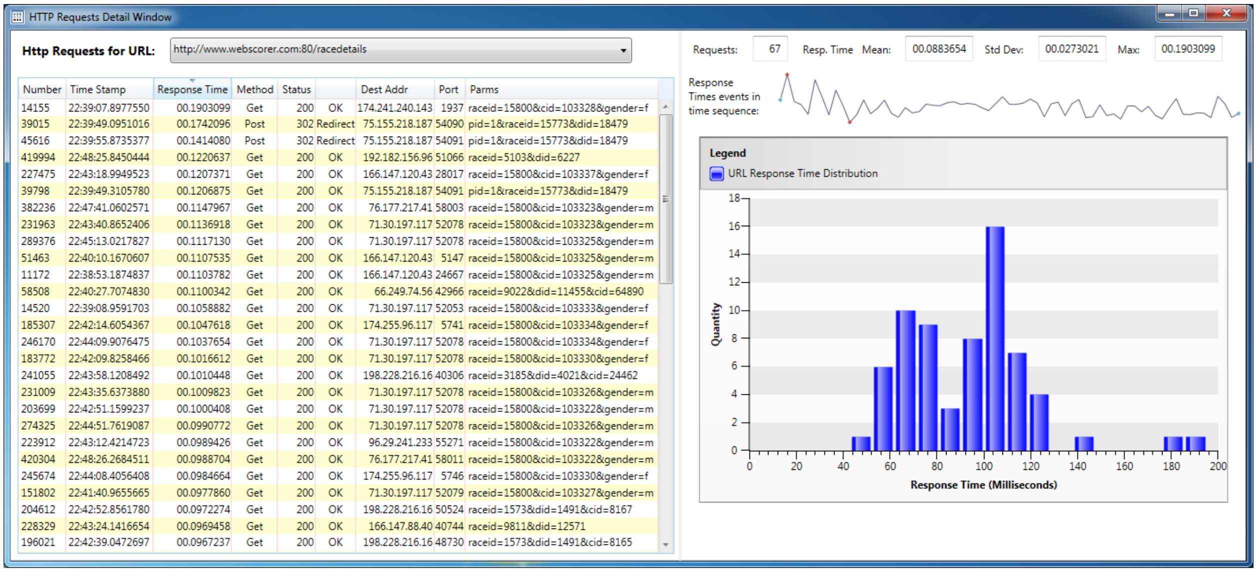
Service Level Reporting

Because measurements of application response time are so important, there are important guidelines and best practices to use when reporting response time measurements. Reports detailing application response times are preferred to ones that merely show the utilization of computer resources like the CPU or disk or the service times for these devices, and are greedily consumed by managers and IT executives. These are called *service level* reports. IT organizations often have *service level agreements* (SLAs) in place with customers that specify response time targets and, often, penalties when those response time targets are not met.[3] Service level reporting is used under those circumstances to track compliance with the SLA.

The first rule for service level reporting is that it is important to emphasize the response time *distribution*, not just report averages. It is too easy for average response time values to obscure the fact that certain requests are taking much long to execute. Excessive response time leads to dissatisfaction, lower fulfillment rates, and outright customer abandonment in favor of a faster application. See Figure 2.4 for a representative example of a service level report that shows a histogram of the response time distribution for a specific ASP.NET web page request, a *sparkline* to

1.

[3] Service level agreements that specify application response time targets with penalties when those targets are not met also need to stipulate the rate or volume of customer requests that require processing within the limits specified. This step is necessary because computer systems do have finite processing capacity.



HTTP Requests Detail Window

Http Requests for URL: http://www.webscorer.com:80/racedetails

Number	Time Stamp	Response Time	Method	Status		Dest Addr	Port	Parms
14155	22:39:07.8977550	00.1903099	Get	200	OK	174.241.240.143	1937	raceid=15800&cid=103328&gender=f
39015	22:39:49.0951016	00.1742096	Post	302	Redirect	75.155.218.187	54090	pid=1&raceid=15773&did=18479
45616	22:39:55.8735377	00.1414080	Post	302	Redirect	75.155.218.187	54091	pid=1&raceid=15773&did=18479
419994	22:48:25.8450444	00.1220637	Get	200	OK	192.182.156.96	51066	raceid=5103&did=6227
227475	22:43:18.9949523	00.1207371	Get	200	OK	166.147.120.43	28017	raceid=15800&cid=103337&gender=f
39798	22:39:49.3105780	00.1206875	Get	200	OK	75.155.218.187	54091	pid=1&raceid=15773&did=18479
382236	22:47:41.0602571	00.1147967	Get	200	OK	76.177.217.41	58003	raceid=15800&cid=103323&gender=m
231963	22:43:40.8652406	00.1136918	Get	200	OK	71.30.197.117	52078	raceid=15800&cid=103323&gender=m
289376	22:45:13.0217827	00.1117130	Get	200	OK	71.30.197.117	52078	raceid=15800&cid=103325&gender=m
51463	22:40:10.1670607	00.1107535	Get	200	OK	166.147.120.43	5147	raceid=15800&cid=103325&gender=m
11172	22:38:53.1874837	00.1103782	Get	200	OK	166.147.120.43	24667	raceid=15800&cid=103325&gender=m
58508	22:40:27.7074830	00.1100342	Get	200	OK	66.249.74.56	42966	raceid=9022&did=11455&cid=64890
14520	22:39:08.9591703	00.1058882	Get	200	OK	71.30.197.117	52053	raceid=15800&cid=103333&gender=f
185307	22:42:14.6054367	00.1047618	Get	200	OK	174.255.96.117	5741	raceid=15800&cid=103334&gender=f
246170	22:44:09.9076475	00.1037654	Get	200	OK	71.30.197.117	52078	raceid=15800&cid=103334&gender=f
183772	22:42:09.8258466	00.1016612	Get	200	OK	71.30.197.117	52078	raceid=15800&cid=103330&gender=f
241055	22:43:58.1208492	00.1010448	Get	200	OK	198.228.216.16	40306	raceid=3185&did=4021&cid=24462
231009	22:43:35.6373880	00.1009823	Get	200	OK	71.30.197.117	52078	raceid=15800&cid=103326&gender=m
203699	22:42:51.1599237	00.1000408	Get	200	OK	71.30.197.117	52078	raceid=15800&cid=103322&gender=m
274325	22:44:51.7619087	00.0990772	Get	200	OK	71.30.197.117	52078	raceid=15800&cid=103326&gender=m
223912	22:43:12.4214723	00.0989426	Get	200	OK	96.29.241.233	55271	raceid=15800&cid=103322&gender=m
420304	22:48:26.2684511	00.0988704	Get	200	OK	76.177.217.41	58011	raceid=15800&cid=103322&gender=m
245674	22:44:08.4056408	00.0984664	Get	200	OK	174.255.96.117	5746	raceid=15800&cid=103330&gender=f
151802	22:41:40.9655665	00.0977860	Get	200	OK	71.30.197.117	52079	raceid=15800&cid=103327&gender=m
204612	22:42:52.8561780	00.0972274	Get	200	OK	198.228.216.16	50524	raceid=1573&did=1491&cid=8167
228329	22:43:24.1416654	00.0969458	Get	200	OK	166.147.88.40	40744	raceid=9811&did=12571
196021	22:42:39.0472697	00.0967237	Get	200	OK	198.228.216.16	48730	raceid=1573&did=1491&cid=8165

Requests: 67 Resp. Time Mean: 00.0883654 Std Dev: 00.0273021 Max: 00.1903099

Response Times events in time sequence:
Legend
URL Response Time Distribution
18
16
14
12
10
8
6
4
2
0
Quantity
0
20
40
60
80
100
120
140
160
180
200
Response Time (Milliseconds)

represent the data visually in time sequence, and provides access to the individual response time measurements themselves.

Figure 2.4. A histogram reporting the response time distribution for a web application

So, instead of merely computing average values, it is usually far better to report on the overall response time distribution when it is available. One variation is to target the 90th or 95th percentile of the response time distribution. Reporting against a 90th percentile target explicitly lets you know that 10% of the requests experienced response times in excess of the threshold.

To create a histogram or report percentile values of the underlying distribution requires access to the individual response time measurements, which then must be sort to calculate the percentiles for the range. Whenever the volume of events is large, this approach is problematic – storing all that measurement data is expensive and post-processing can be time-consuming. In contrast, statistics like the mean, variance and standard deviation are very easy to calculate from continuously maintained counters. This is the reason why so many real-time monitoring tools report those statistics instead.

An alternative approach commonly used in real-time monitoring is to maintain a set of buckets for response time events, where each bucket is a count of the response time measurements that exceeded the target value for each bucket. Mail delivery response time counters are reported using buckets in Microsoft Exchange, for instance. Buckets can be problematic if the target values for each bucket are not a good match to the underlying distribution, in which case too many of the response time events fall within just one or two of the available buckets. The solution is to configure the targets for the buckets based on the underlying distribution, but not too many reporting tools provide that option. The alternative is to provide lots of buckets, or hope you simply get lucky with the predetermined settings you have chosen.

Given how important response time measurement data is, it is unfortunate that the data we need is not more readily available in many of the Windows applications that you run. For instance, for ASP.NET web applications, there are Windows performance counters called the Request Execution Time and Request Queue Time. Since the former is a measure of service time and the latter is queue time, we ought to be able to add them together to calculate web application response time. These two counters actually report the execution time and the queue time of the *last* ASP.NET request that was observed. Since the measurements reflect a single web application request, this measurement is, in effect, a randomly sampled value from the overall response time distribution for a busy web site. This is useful performance data, but nowhere near as valuable as the full response time distribution. Also, you may notice there is no further information about what specific ASP.NET page was requested for the measurement reported.

Fortunately, while good Windows performance counters measuring ASP.NET server-side response times are missing, it is possible to acquire web application response time measurements from other data sources, including IIS logs and network and HTTP trace data. When we look in depth at ASP.NET web application performance in Section 4, we will investigate alternative sources for web application response data.

Note also that the ASP.NET performance counters or tools that process the IIS log can only report the time spent processing an HTTP request on the web server, ignoring network transmission times and interaction down at the web client. Many authorities prefer *end-to-end* measurements of web application response times that includes the time spent in the network and in web browser composition. We will return to this topic in earnest in Section 4.

Something most people don't know is that response times for a large number of Windows scenarios are instrumented so that Microsoft developers can understand how the software they deliver performs. These measurements are gathered under the umbrella of the Microsoft Customer Experience Improvement Program, and the data is sent back to Microsoft where it is scrutinized by executives and analyzed by the engineering teams. Windows began instrumenting scenarios and gathering response time data using predefined buckets following the release of Vista, which was poorly received partly due to customer perception about its responsiveness. Here is a blog entry discussing some aspects of this program:

> Perftrack is a very flexible, low overhead, dynamically configurable telemetry system. For key scenarios throughout Windows 7, there exist "Start" and "Stop" events that bracket the

scenario. Scenarios can be pretty much anything; including common things like opening a file, browsing to a web page, opening the control panel, searching for a document, or booting the computer. Again, there are over 500 instrumented scenarios in Windows 7 for Beta.

Obviously, the time between the Stop and Start events is meant to represent the responsiveness of the scenario and clearly we're using our telemetry infrastructure to send these metrics back to us for analysis. Perftrack's uniqueness comes not just from what it measure but from the ability to go beyond just observing the occurrence of problematic response times. Perftrack allows us to "dial up" requests for more information, in the form of traces.

Response time and User Satisfaction

Service level reporting shows the percentage of customer requests that exceed a threshold value chosen because it is directly associated with customer satisfaction are also very effective. Lacking a direct measurement of customer satisfaction, many e-commerce web sites establish reporting thresholds based on *conversion rates*. In e-commerce, the conversion rate is calculated as follows:

♦ *Conversion rate = # of confirmed orders / # of landing page hits*

It is the rate that *visitors* to a particular landing page of a web application are converted into *customers*, which is a good enough proxy for customer satisfaction for most business enterprises. Many studies show that improvements in web application response times lead to significantly improved conversion rates.[4]

Lacking empirical data on conversion rates, some authorities recommend using a standard endorsed by several performance companies that is known as the Apdex index. The Apdex index was developed by Peter Sevcik, a respected networking performance and capacity consultant. The customer satisfaction index he developed is based on three zones of application responsiveness – response times that lead to fully satisfied customers, response times that are merely tolerated, and response times that lead to outright dissatisfaction or frustration. On the face of it, these certainly appear to be appropriate categories to use to map user satisfaction to application response times.

However, the actual relationship between the response times your application provides and customer satisfaction is actually somewhat complicated. Human factors research in computer science has produced evidence that customers tend to grow accustomed over time to current service levels. This acculturation forms a psychological basis for the Toleration category suggested by

1.

[4] See, for example, http://blog.mozilla.org/metrics/2010/04/05/firefox-page-load-speed-%E2%80%93-part-ii/, http://kylerush.net/blog/meet-the-obama-campaigns-250-million-fundraising-platform/, and the Walmart experience presented at the SF & SV Web Performance Group – 2012-02/16.

the Apdex methodology. While customers certainly prefer better response times, keep in mind that past experiences set baseline expectations – in many respects, human subjects are not that different from Pavlov's dogs in having their expectation shaped by their experiences.

A better model to use is that humans adapt their behavior to their environment, understanding that past experience biases expectations concerning future interactions. Human factors researcher Steve Seow in his book *Designing and Engineering Time: the Psychology of Time Perception in Software*, reports that application response time improvements that are *less* than 15-20% better than current levels hardly register with customers. By the same token, performance regressions that degrade response times by less than 15-20% are generally not perceived either. Presumably, toleration levels are also related to the variability of the response times that customers experience, but Seow's book is silent on that conjecture.

Since past experience establishes current expectations, in the absence of conversion data or other indirect measurements of customer satisfaction, thresholds based on current service levels are a reasonable way to get started with service level reporting.

In a few specific areas, Seow's book makes practical recommendations for the level of application response time that results in customer satisfaction. For instance, there are some human-computer interactions where software is used to simulate physical gestures and other similar behavior. Examples include echoing a keyboard press or a mouse move gesture on the screen in real-time. Human factors research suggests that response times for these kinds of interactions in excess of 300 milliseconds generate some degree of frustration. In effect, a computer simulation of some physical behavior needs to meet the performance expectations of that actual behavior in the real world.

Outside the narrow area where our software attempts to mimic some physical behavior, in Dr. Seow's view, the relationship between user tolerance and frustration with response times tends to be much more context-sensitive that the strict, functional categories that Apdex defines. For instance, his book recommends practical software engineering techniques, such as status bars and escape key sequences, which can be used to manipulate user perception in your application and boost customer tolerance. The salient point is that a customer's perception of good or bad response time is informed by personal history and the expectations that arise based on that history.

This makes actual measurements of your customers that reflect the relationship between application response times and customer satisfaction, like the conversion rate data discussed above, that much more valuable. In web applications, the availability of Real User Measurements (RUM) using boomerang.js from Yahoo and similar Site Speed measurements in the Google Analytics measurement framework has enabled many organizations to assess the actual relationship between web application response times and conversion rates for their sites and for their customers. Again and again, organizations have seen that performance matters.

In the absence of actual measurements, the developers of the Apdex index recommend using a 1-second threshold value to represent satisfied users and a 4-second frustration threshold. Apdex has failed to publish any empirical studies to justify using these threshold values, however. Instead of plugging in arbitrary customer satisfaction values from Apdex or any other source, it makes better sense to adopt thresholds for service level reporting based on current measurements.

Kano Models

Another way to think about the relationship between application response times and customer satisfaction is to use the approach known as the Kano Model, named after the pioneering work of Noriaki Kano. Kano argued that not all product features are equally valued in the eyes of customers and created a framework for assessing that value. In a Kano Model approach, customers are surveyed to determine how much they value certain current and prospective new product features to determine if they

- reflect mandatory minimum requirements,
- confer a competitive advantage where more of the feature is better, or
- serve as a positive differentiator that will produce an unexpected and pleasant surprise that will delight a potential customer should you succeed in delivering that feature.

For instance, in an automobile, a vehicle that starts reliably is a minimum requirement, while better gas mileage is a competitive advantage. Meanwhile, today a vehicle that can execute a parallel parking maneuver automatically is a key product differentiator that is likely to delight a new customer.[5] These three categories that Kano Models use to identify how customers assess the value of a product feature are illustrated in Figure 2.5.

This is a good way to think about computer performance, too. For computer software, in Kano Model terms, surveys of customers tend to show performance as an expected quality feature. Customers expect and are accustomed to demanding a certain level of performance. It is a minimum requirement. Unlike a Satisfier feature in the Kano Model, better performance does not necessarily lead to greater satisfaction. But poor performance is a guaranteed dissatisfier.

1.

[5] In the Kano model, the features that are critical to customer satisfaction and quality perception do change over time. In the case of automobiles, once virtually every vehicle comes equipped with a parallel parking assist feature, adding that feature will no long delight new customers. Over time, a feature that once delighted customers will drift into the category of mandatory, minimum requirements as more and more products incorporate that innovation. For more on the Kano Model approach, see http://www.six-sigma-material.com/Kano.html.

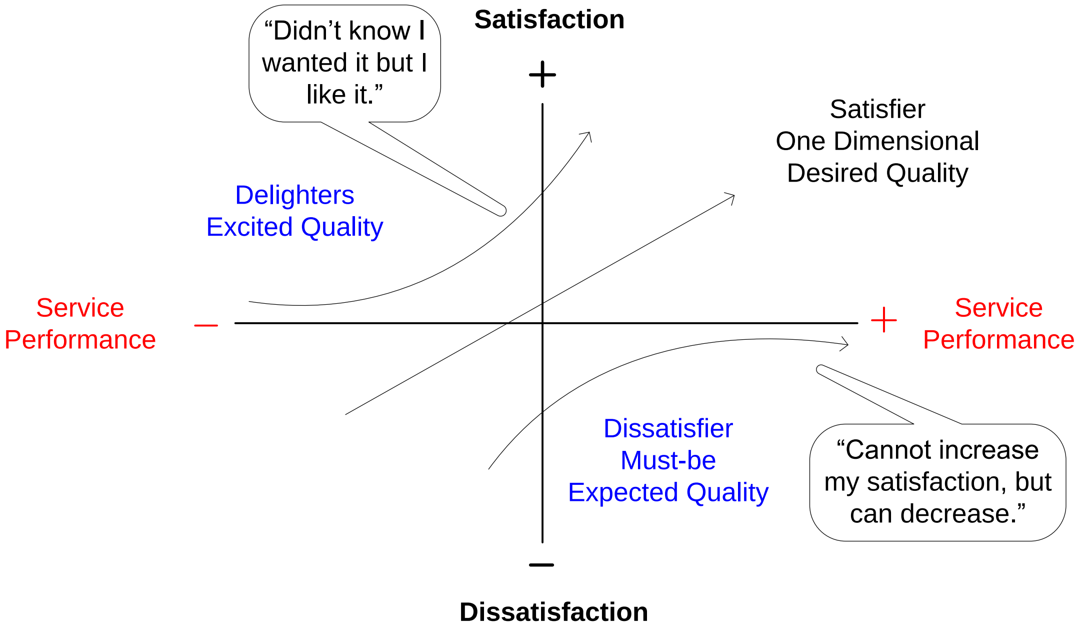


Figure 2.5. Kano Models organize the way customers assess the value of a product feature into three categories. Customers usually regard software performance as a minimum requirement.

QUEUE TIME

Since response time is the sum or service time and queue time, it is time we discussed queue time. When a work request arrives at a busy resource and cannot be serviced immediately, the request is then queued. Requests are subject to a *queue time* delay once they begin to wait in a queue before being serviced.

Queue time arises in a multi-user computer system because important computer resources are *shared*. These shared resources include the processor, the disks, and network interfaces. This sharing of devices is performed by the operating system in a way that is largely transparent to the individual programs that are running. Process A that is accessing data located on the hard drive does not know the process B is also trying to access the same drive. It is the operating system that makes this sharing possible, ensuring that resources that must be acquired and used *serially*, i.e., one at a time, are accessed in the proper manner. If process B directs a request to the C: drive while it is already busy with a request from process A, it is a function of the operating system to serialize those requests. The request from process is parked in a queue. When the disk signals it has completed the previous request, the OS removes a queued request and sends it to the device. This behavior is transparent to both process A and B.

The operating system guarantees the integrity of shared resources like the processor, disks, and network interfaces by ensuring that contending applications access them serially. When a work request to access a shared resource that is already busy servicing another request occurs, the operating system queues the request and *queue time* begins to accumulate. Queuing delays occur because there are shared resources with multiple applications that are attempting to access them in parallel. When there is significant contention for a shared resource because two or more programs are attempting to access it at the same time, performance may suffer.

Of course, one of the major advantages of a multi-user operating system like Windows, compared to early versions of Windows that were only designed for single user workstations, is that they can be shared among multiple users. However, the one aspect of sharing resources that may not totally transparent to executing programs is the potential performance impact due to resource sharing. When there is a performance problem on a Windows workstation, only one user suffers. When there is a performance problem on a Windows application server, a multitude of computer users can be affected.

On a very heavily utilized system, queue time can become a very significant source of delay. It is not uncommon for queue time delays to be longer than the amount of time actually spent receiving service at the device. No doubt, you can relate to many real world experiences where queue time is significantly greater than service time. Consider waiting in line in your car at a toll booth, back in the days when we all did that. The amount of time it takes you to pay the toll is often insignificant compared to how long you spent waiting in line. The amount of time spent waiting in line to have your order taken and filled at a fast food restaurant during the busy lunchtime period is often significantly longer than the time it takes to process your order. Similarly, queuing delays at an especially busy shared computer resource can be prolonged. It is important to monitor the queues at shared resources closely to identify periods where excessive queue time delays are occurring.

> **Important**
>
> Measurements of either the queue time at a shared resource or the queue length are some of the most important indicators of performance you will encounter. Queues that are discussed in this book include the operating system’s thread scheduling Ready Queue, the logical and physical disk queues, the file server request queue, and the queue of ASP.NET web server requests.

Queue time can be difficult to measure directly without adding excessive measurement overhead. Fortunately, direct measurements of the queue length based on sampling can suffice in many situations. A simple equation known as Little’s Law, discussed in more detail below, shows how the rate of requests, the response time, and the queue length are related. Little's Law allows us to calculate average response times for requests if queue length and arrival rate measurements are available.

If you know the queue length at a device and the average service time, the queue time delay can be estimated reliably, as follows:

- ♦ *Queue time = queue length * Ave. service time*

This simple formula reflects the fact that any queued request must wait for the request currently being serviced to complete.

Actually, this formula over-estimates queue time to a small degree. When a device is busy and another request arrives and finds the queue empty, it must wait until the previous request completes. Sometimes the previous request is almost finished and the queue time is much less than the average service time. Sometimes the previous request just started and the queue time is quite close to the average service time. On *average*, the queue time of the first request in the queue is approximately ½ the service time. Subsequent requests that arrive and find the device busy and at least one other request already in the queue are then forced to wait:

- ♦ *Queue time = ((queue length-1) * Ave. service time) + (Ave. service time/2)*

Not all computer resources are shared under Windows, which means that these devices effectively have no queuing time delays. Input devices like the mouse and keyboard, for example, are managed by the operating system so that they are only accessible by one application at a time. Because these devices are *buffered* to match the speed of the people operating them, there is never any queue time delay that can be measured.

OPTIMIZING FOR PERFORMANCE

It may seem that the goal of any performance and tuning exercise is to maximize throughput *and* minimize response time. In practice, these goals are contradictory. If you try to optimize for throughput, you are apt to drive up response time to unacceptable levels as a result. Alternatively, minimizing response time by stockpiling only the fastest and most expensive equipment and then keeping utilization low in order to minimize queuing delays is not very cost-effective. In practice, skilled performance engineers are satisfied if they can achieve a balance between these often conflicting objectives:

- There is relatively high throughput leading to the cost-effective use of the hardware.

- Queuing delays are minimal, so users are satisfied with performance levels.

- System performance is relatively stable, consistent, and predictable.

Insights from queuing theory, a branch of operations research frequently applied to the analysis of computer systems performance, help to explain why these objectives are not so easy to reconcile. I have referenced queuing models repeatedly in this chapter. It is time we drill into that topic in greater depth.

Figure 2.3 showed a conventional representation of a computer system modeled as a network of queues, with customer requests arriving at a server and a simple queuing mechanism. These are the basic elements of a queuing system: customers, servers, and a queue. Figure 2-3 shows a single server – and is often labeled the Single Server model – but extensions to multiple servers are certainly possible.

If the server is idle, a customer request arriving at the server is processed immediately. On the other hand, if the server is already busy with another request, then the incoming request is queued. If multiple requests are waiting in the server queue, the *queuing discipline* determines the order in which queued requests are serviced. The simplest and fairest way to order the queue is to service requests in the order in which they arrive. This is also called *FIFO* (First In, First Out) or sometimes First Come, First Served (*FCFS*). There are many other ways that a queue of requests can be ordered, including by priority or by the speed with which they can be processed.

Figure 2.6 shows a simple queue annotated with symbols that represent a few of the more common parameters that define its behavior. There are a large number of parameters that can be used to describe the behavior of this simple queue, and we only look at a couple of the more basic and commonly used ones. Response time (***W***) represents the total amount of time a customer request spends in the queuing system, including both service time (***W_s***) at the device and wait time (*W_q*) in the queue. By convention, the Greek letter **λ** (lambda) is used to specify the arrival rate or frequency of requests to the server. Think of the arrival rate as an activity rate such as the rate at which HTTP GET requests arrive at a web server, I/O requests are sent to a disk, or packets are sent to a NIC card. Larger values of λ indicate that a more intense workload is applied to the system, whereas smaller values indicate a light load. Another Greek letter, **μ**, is used to represent the rate at which the server can process requests. The output from the server is either λ or μ, whichever is less, because it is certainly possible for requests to arrive at a server faster than they can be processed.

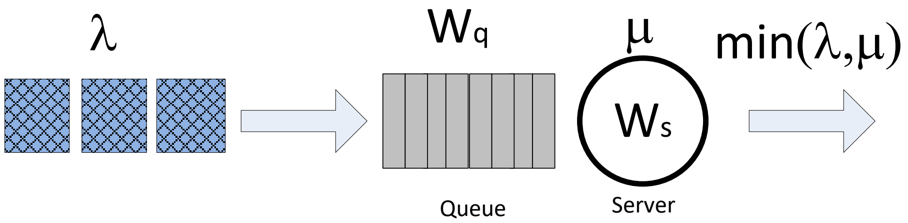


Figure 2.6. A simple queuing model with its parameters.

Ideally, we hope that the arrival rate is lower than the service rate of the server, so that the server can keep up with the rate of customer requests. In that case, the arrival rate is equal to the completion rate, an important equilibrium assumption that allows us to consider models that are mathematically tractable (i.e., solvable). If the arrival rate, ***λ,*** is greater than the server capacity ***μ***, then requests begin backing up in the queue.

Mathematically, we assume arrivals are drawn from an *infinite* population. This assumption is known as an *open* queuing model. It is an assumption that makes the mathematics easier, but it can lead to unrealistic results since the population of potential customer isn't actually infinite. But, in an open model, when $\lambda > \mu$, the queue length can grow to an infinite length since the arrival rate never slows down and the server cannot keep up with the rate of requests. Under these circumstances, the wait time (***Wq***) at the server grows infinitely large. Those of us who have tried to retrieve a service pack from a Microsoft web server immediately after its release know this situation and its impact on performance. It is actually not possible to solve an open queuing model with $\boldsymbol{\lambda > \mu}$, but mathematicians have developed *heavy traffic approximations* to deal with this familiar case.

A very pointed example of an arrival rate, ***λ,*** that is greater than the server capacity ***μ*** occurs in computer networking whenever a fast router attempts to send bulk data to a slow router. This happens when you are trying to stream high definition video from a datacenter or cloud media server to a distant workstation. The media server has the capability to place video data onto the network faster than the workstation can receive and process it. The *flow control* algorithm used in the TCP protocol is designed to deal with this situation by throttling back the arrival rate in order to match the service rate of the receiver. In a well-designed video streaming app, the sender attempts to negotiate a send rate that matches the capacity of the receiver in order to achieve the highest possible bit rate without excessive queuing.

In real-world situations, we can expect there are actual limits on the population of customers. The population of customers could be limited by the number of network interfaces configured in a LAN environment, the number of file server connections, the number of TCP connections a web server can support, or the number of concurrent open files on a disk. These may be very large limits, but they are limits that eventually function to throttle back the arrival rate of requests whenever a large enough portion of the customer population have in-flight requests that are queued for service. By the way, these are known as *closed* queuing models.

Even in the worst bottlenecked situation, in a closed model the queue depth is limited to the size of the population. When enough customer requests are bogged down in queuing delays, in real systems, the arrival rate for new customer requests is constrained. Customer requests that are backed up in the system choke off the rate at which new requests are generated.

For a public-facing web portal, the population of potential visitors – effectively, the number of humans on the planet with web access – is so large that queuing behavior with virtually no limits is eminently possible. The potential number of visitors is so large that popular web applications can reasonably be approximated as open queuing models. There is plenty of evidence that shows that when web customers encounter large delays in accessing a web site, they are more likely to abandon it in favor of some other site with similar content.[6]

The purpose of this rather technical discussion is twofold. First, it is necessary to make an unrealistic assumption about arrivals being drawn from an infinite population to generate readily solvable models. Queuing models break down as $\boldsymbol{\lambda} \rightarrow \boldsymbol{\mu}$, which is another way of saying they break down near 100% utilization of some server component. Second, you should not confuse a queuing model result that shows a nearly infinite queue length with reality. In real systems there are often practical limits on queue depth. In the remainder of our discussion here, we will assume that $\lambda < \mu$. Whenever the arrival rate of customer requests is equal to the completion rate, this is viewed as an *equilibrium assumption*. Please keep in mind that the solution of simple queuing models are invalid if the equilibrium assumption does not hold; in other words, when the arrival rate of requests is greater than the completion rate ($\boldsymbol{\lambda} > \boldsymbol{\mu}$ over some measurement interval) simple queuing models do not work. The equilibrium assumption becomes very important when we investigate the relationship of response times to utilization and the derivation of Little's Law.

BOTTLENECKS

One of most effective methods in performance tuning is systematically identifying bottlenecked resources and then work to remove or relieve them. When the throughput of a particular system reaches its effective capacity limits, we say that the system is *bottlenecked*. The resource bottleneck is that component which is functioning at its capacity limit. The bottlenecked resource can also be recognized operationally as the resource with the fastest growing queue as the number of customers increases.

1.

[6] One of the performance statistics that web site analytics provides is the *bounce rate*. The bounce rate is calculated as the percentages of visitors to a landing page that never visit any other pages of your web site, as a percentage of all visitors. This is a statistic that organizations that pay for Google AdWords on per click basis track because it is a good measure for how effective your advertising is. When increased Page load times results in higher bounce rates, the implication is that potential customers are abandoning the site in favor of a competitor's web site that may be more responsive.

Important

The corollary of bottleneck analysis is that a well-tuned system is *balanced*, with no resource queue growing faster than any other.

Understanding that a computer system is operating at the capacity limit of one of its components is an important thing to know. It means, for example, that no amount of tweaking tuning parameters is going to overcome the capacity constraint and allow the system to perform more work. You need more capacity and any other resolution short of providing some capacity relief is going to fall short!

Once you have identified a bottlenecked resource, there are some systematic ways that you can go about relieving that limit on performance and permit more work to get done. You might consider, for example:

- tune the application so that it runs more efficiently (i.e., utilizes less bandwidth) against the specific resource,
- upgrade the component of the system that is functioning at or near its effective bandwidth limits so it runs faster, or
- spread the application across multiple resources by adding more processors, disks, or network segments, etc., and assuming it has the capability to process in parallel.[7]

It is possible that none of these alternatives to relieve a capacity constraint will succeed in fixing the problem quick enough to satisfy your Users. This is when it might be worthwhile to resort to tweaking this or that system or application tuning parameter to provide some short term relief. The most important settings for influencing system and application performance in Windows are discussed in Section 2. In addition, there are many run-time settings associated with applications such as MS Exchange or the MS Internet Information System (IIS) that can impact performance. Many application-oriented optimizations are documented in other technical publications or in white papers available at www.microsoft.com.

There are a number of tuning adjustments that you may be able to take advantage of that are built into Windows. Many of the tuning mechanisms are designed to kick in automatically, but there may be additional tweaks that are worthwhile for you to consider making manually. Some of these built-in performance optimizations employ intelligent scheduling algorithms. Keep in mind that

1.

[7] Just because an application is not currently running in parallel does not mean that it cannot be restructured to do so. There is little alternative for applications that become too large for the available hardware except figuring out how to parallelize them. Necessity being the mother of invention, there is no shortage of innovative solutions to create parallel versions of programs that previously only ran serially.

scheduling algorithms have a good opportunity to improve performance *only* when there are enough queued requests outstanding that it makes a difference which request the operating system schedules next. Consequently, these performance optimizations only have a significant impact on performance when the underlying resource is quite busy. The next section explains the basic theory under which these scheduling algorithms operate.

Managing queues for optimal performance

If there are multiple requests waiting in a queue, the *queuing discipline* is what determines which request is serviced next. Most queues that humans occupy when they are waiting for service are governed by the principle of *fairness*. A fair method of ordering the queue is First Come, First Serve (FCFS). This is known as a FIFO queue, which stands for First In, First Out. That is how you find yourself waiting in line in a bank in order to cash a check, for example. FIFO is considered fair because no request that arrives *after* another can be serviced before requests that arrived earlier are themselves satisfied. Round Robin is another fair scheduling algorithm where customers take turns receiving service in the order in which they arrive. Life is not always fair, but fairness is certainly something that human beings recognize when they are stuck waiting in a queue.

Unfair scheduling algorithms

For performance reasons, the Windows operating system does not always use fair scheduling policies for the resource queues it is responsible for managing. Where appropriate, Windows makes uses of *unfair scheduling* policies that can produce better results at a heavily loaded device. The unfair scheduling algorithms that are implemented make it possible for devices such as processors and disks to work more efficiently under heavier loads. The two most common unfair scheduling policies are priority queuing (and often together with *preemptive* scheduling) and sorting the queue based on the expected service time of the request.

Priority queuing with preemptive scheduling

Certain work requests are regarded as higher priority than others. If both high priority and low priority requests are waiting in the queue, it makes sense for the operating system to schedule the higher priority work first. In Windows, queued requests waiting for the processor are ordered by priority, with higher priority work taking precedence over low priority work. The priority queuing scheme that the operating system uses to manage the processor queue is reviewed in more detail later in Section 2.

The priority queuing scheme used to manage the processor queue in Windows has at least one additional wrinkle worth discussing further here. Processor hardware also services high priority *interrupts*. Devices such as disks interrupt the processor to signal that an I/O request that was initiated earlier is now complete. When a device interrupt occurs, the processor queues the then currently executing program thread and services the device that generated the higher priority interrupt immediately. When higher priority work is scheduled to run immediately and will even interrupt a lower priority thread that is already running, it is called *preemptive scheduling*. Windows uses both priority queuing and preemptive scheduling to manage the system's processor queue.

Priority queuing has a well-known side effect that becomes apparent when a resource is very heavily utilized. If there are enough higher priority work requests to saturate the processor, lower priority may get very little service. This is known as *starvation*. When a resource is saturated, priority queuing ensures that higher priority work receives preferred treatment, but lower priority work may suffer from starvation. Lower priority work may remain delayed in the queue, receiving little or no service for extended periods. The resource utilization measurements that are available in Windows for the processor allow you to assess whether the processor is saturated, what work is being performed at different priority levels, and whether low priority tasks are suffering from starvation.

Servicing shorter requests first

When queued requests can be sorted according to the expected amount of service time that will be needed to complete the request, higher throughput is normally achieved if the shorter work requests are serviced first. This is the same sort of optimization that supermarkets use when they have shoppers sort themselves into two sets of queues, based on the number items in their shopping carts. Sometimes this sorting can be performed on the fly, based on the expected service time. Windows implements a form of dynamic sorting to boost the priority of short processor service requests that is equivalent to the Mean Time to Wait (MTTW) algorithm. Another area of the system where queued requests are ordered by expected service time is when disks are enabled for SCSI command tag queuing.

In scheduling which thread to run next on the processor, giving consideration to processor caching effects probably qualifies as a shortest service time first optimization, too. Scheduling a thread to run on the same processor in a multiprocessor where it was running last provides an opportunity to take advantage of a warm start in the processor hardware caches. Scheduling a thread to run on a processor *different* from the one it was running on last provides guarantees a cache cold start, reducing the instruction execution rate until the caches start to reach an equilibrium state. On multi-socket machines with non-uniform memory access times (or NUMA), the Windows OS thread Scheduler also has to consider what socket the thread executed on last. Performance considerations for NUMA architectures are discussed in more detail in Section 5.

It is important to remember that reordering the device's queue can only have a significant performance impact when there is a long queue of work requests that can be rearranged. Over and above the magic that intelligent scheduling algorithms can work, it is important to recognize that computer systems where a resource remains saturated for an extended period of time, allowing lengthy lines of queued requests to build up, is out of capacity. You need to configure machines large enough that they have enough capacity to service normal peak loads. While intelligent scheduling at the device can provide some relief, it should never divert your attention from the underlying problem, which is a shortage of capacity where you need it most.

Bottleneck Analysis

To make the best planning decisions, a traditional approach is to try and understand hardware *speeds and feeds* —how fast different pieces of equipment are capable of running. This is much more difficult than it sounds. For example, it certainly sounds like a hard drive attached to a USB 2.0 interface will run much slower than one attached to a USB 3.0 adaptor. USB 3.0 should beat ol' USB 2 every time. But the fact is there may be little or no practical difference in the performance of the two configurations. (The reason there may be no difference is because the disk may only transfer data at the lower USB 2.0 rate anyway, so the extra capacity of the USB 3.0 interface is not being utilized.)

This example illustrates the principle *that a complex system will only run as fast as its slowest component*. Because there is both theory and extensive empirical data to back up this claim, this statement can be propagated as a good expert's Rule of Thumb. It fact, it provides the theoretical underpinning for a very useful analysis technique called *bottleneck analysis*. The slowest device in a configuration is often the weakest link. Find it and replace it with a faster component and you have a good chance that performance will improve. Sounds good, but you probably noticed that the rule does not tell you *how* to go about finding this component. Given the complexity of many modern computer networks, this seemingly simple task is as easy as finding a needle in the proverbial haystack. After all, replace some component *other* than the bottleneck device with a faster component and performance will remain the same.

In both theory and practice, performance tuning is the process of locating the bottleneck in a configuration and removing it somehow. The system's performance will improve until the next bottleneck is manifest, which you can then identify and remove. Removing it usually entails replacing it with a newer, faster version of the same component. For example, if network bandwidth is a constraint on performance, upgrade the configuration from 1 Gigabit Ethernet to 10 Gigabit Ethernet. If network bandwidth actually is the bottleneck, performance should improve.

A system in which all the bottlenecks have been removed can be said to be a *balanced* system. All the components in a balanced system are at least capable of handling the flow of work from component to component without delays building up at any one particular component. For a moment, let's return to the network of computing components where work flows from one component (the CPU) to another (the disk), back again, then to another (the network) and back again to the CPU depicted in Figure 2-3. When different workload processing components are evenly distributed across the hardware devoted to doing the processing, that system is balanced.

A balanced system (and not one which is simply over-configured) can be visualized as one where workload components are evenly distributed across the processing resources. If there are delays, the work that is waiting to be processed is also evenly distributed in the system. Work that is evenly distributed around the system waiting to be processed is illustrated in Figure 2-7. Suppose you could crank up the rate at which requests arrive to be serviced (think of requests to a SQL Server database, for example, or logon requests to an Active Directory authentication server). If the system is balanced, you will observe that work waiting to be processed remains evenly distributed across system components, as in Figure 2-7.

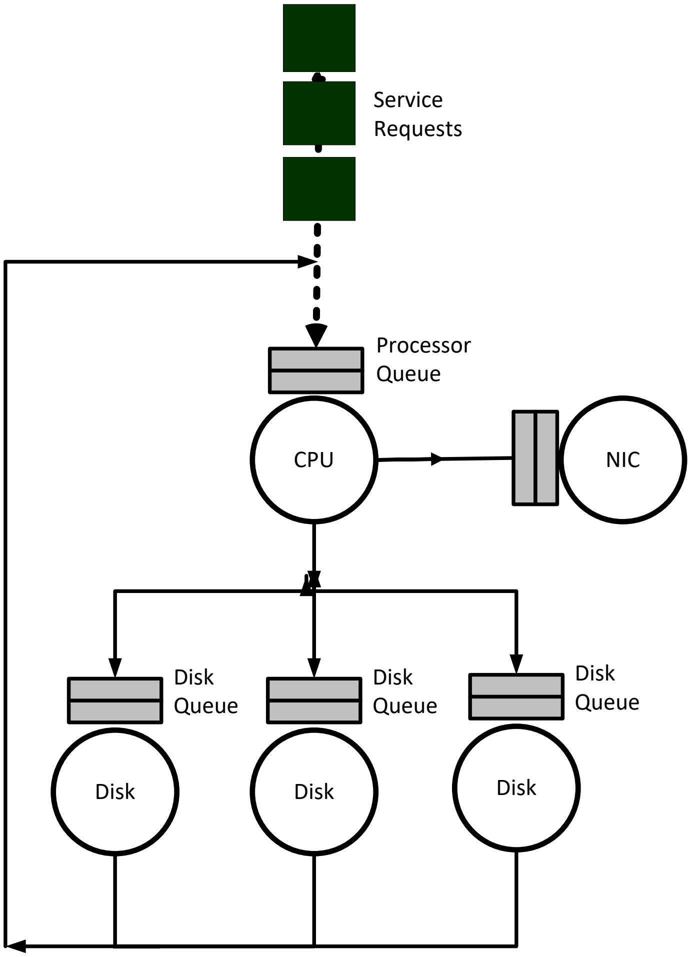
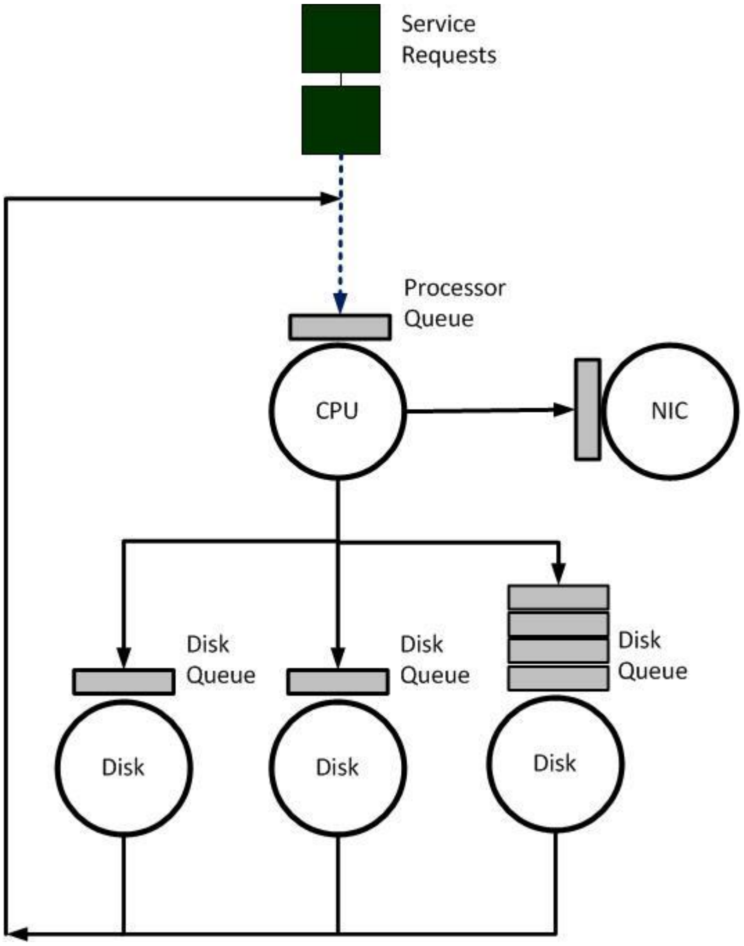


Figure 2.7. A balanced system is one in which all resource queues grow at the same rate.

If instead you observe something like what is depicted in Figure 2-8 where the queue in front of one of the resources is much longer than any other resource, it is possible to identify with some authority the component that is the bottleneck in the configuration. When work backs up behind a bottlenecked device, delays there can cascade, causing delays to build up elsewhere in the configuration. Because the flow of work through the system is not like a simple chain of events, but more like an interconnected network, how the work that gets backed up overflows and impacts processing at other components may not always be obvious. Empirically, it is sufficient to observe that work accumulates behind the bottlenecked device *at the fastest rate* as the workload rate increases. Replacing this component with a faster processing component should improve the rate that work that can flow through the entire system.

Figure 2.8. A bottlenecked system is one where work requests back up in the queue for the bottlenecked resource faster than any other resource.



RESPONSE TIME AND UTILIZATION

The final topic for discussion in this brief survey of queuing theory is the relationship between utilization and response time. The Utilization law specifies that utilization is the product of the arrival rate times the service time. If a request arrives at a resource which is idle, it is serviced immediately. If the resource is already busy servicing another request when the request arrives, it is queued for service, forced to wait until the resource becomes free. It ought to be apparent that the busier the resource gets, the more likely it is for a new request to encounter a busy device and be forced to wait in a queue.

Because response time is the sum of service time and queue time,

♦ $$W = W_s + W_q$$

we will solve some simple queuing models to calculate the queue time based on utilization.

It is important to remember that these simple models probably do not model reality too closely – these simple models were chosen mainly because they are easy to calculate. Yet experience shows that queuing theory can be very useful in explaining how many of the computer performance measurements you will encounter behave – up to a point. Some of the important ways these simple models fail to reflect the reality of the more complex computer systems also need to be understood to allow you to use these mathematical insights wisely.

Simple queuing models, as depicted in Figure 2.6, are characterized by three elements: the arrival rate of requests, the service time of those requests, and the number of servers to service those requests. If those three components can be measured, simple formulas can be used to calculate other interesting metrics, including queue time. In addition, both the queue length and the amount of queue time requests are delayed waiting for service can be calculated using a simple formula known as Little's Law. Queue time and service time, of course, can then be added together to form response time, which is often the term that you are interested in deriving.

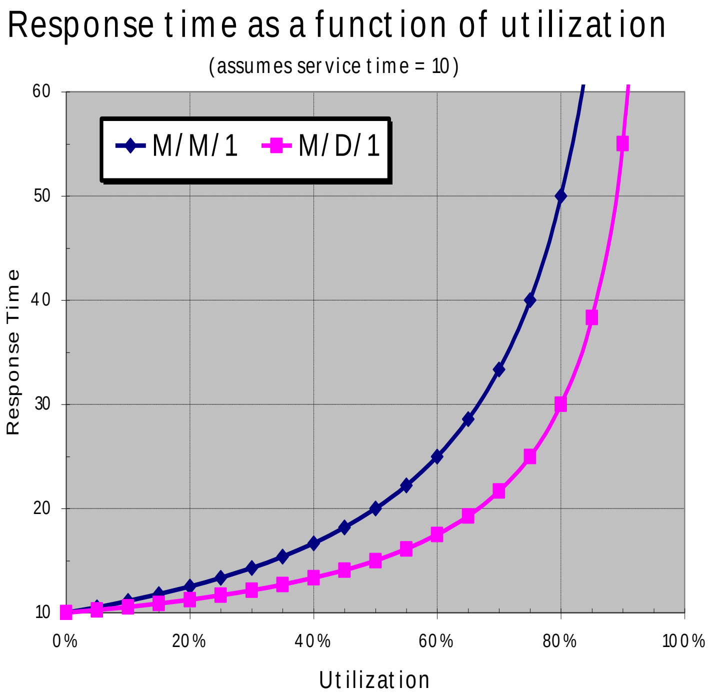


Figure 2.9. Response time as a function of utilization in a M/M/1 and M/D/1 queuing model.

Arrival rate distribution

To put even this simple mathematics to work, however, it is necessary to know both the average rate that requests arrive and the *distribution* of arrivals around the average value. The arrival rate distribution describes whether requests are spaced out evenly (or *uniformly*) over the measurement interval or whether they tend to be bunched together (or *bursty)*. Lacking precise measurement data on the arrival rate distribution, it is usually necessary to assume that the distribution is bursty (or random). A random arrival rate distribution is often a reasonable assumption, especially if there are many, independent customers generating the requests. A large population of customers for an Internet e-business web site, for example, is apt to generate a randomly distributed arrival rate.

Similarly, an MS Exchange Server servicing the e-mail requests of employees from a large organization are also likely to approximate a randomly distributed arrival rate over short intervals of an hour or less.

Over longer intervals, arrival rates for most web and mail requests are likely to show some evidence of periodicity. For mail requests are very heavy at a business at the beginning of the business day, and tend to spike again in the early afternoon as people return to their desks after lunch. The periodicity of web requests at an e-commerce is associated with seasons of the year. For example, customer requests tend to peak on Black Monday in the U.S. –the first Monday after the Thanksgiving holiday. These characteristic peaks and values are predictable, certainly, but they are anything but *uniform*.

The assumption that customer arrivals are independent can also be a very poor one, especially when the number of customers is very small. Consider, for example, a disk device with only one customer like a back-up process or a virus scan. Instead of random arrivals, a disk back-up process *schedules* its I/Os to disk in a very efficient manner. A program execution thread from a back-up program issuing disk I/I requests will generally not release another I/O request until the previous one completes. When requests are scheduled in this fashion, it is possible for a single program to drive the disk to virtually 100% utilization levels without incurring any queuing delay.

Service time distribution

It is also necessary to understand both the average service time for requests and the distribution of those service times around the average value. Again, lacking precise measurement data, it is simple to assume that the service time distribution is also random. It will also be helpful to compare and contrast the case where the service time is relatively constant, or *uniform*. We will see that when the service time distribution relatively uniform – most requests for service take about the same amount to service – queuing delays are minimized. Whenever there is an opportunity to schedule work in a way that creates a more uniform service time distribution, it is usually a good idea to try to do this. This is one the rationales behind sorting customers at a supermarket checkout line based on the number of items in their shopping carts. This is also the rationale behind taking large network data transfer requests in and executing them in standard-sized packets.

The scheduling rationale comes from queuing theory. The two simple cases that are illustrated in Figure 2.9 are denoted as an M/M/1 and an M/D/1 model. The standard notation identifies the *arrival rate distribution* / *service time distribution / number of servers*, where *M* is an *exponential* or random (or Markovian) distribution, *D* is a uniform distribution and *1* is the number of servers (resources).

Both curves in Figure 2.9 show that the response time of a request increases sharply as the server utilization increases. Since these simple models assume that service time remains constant under load (as discussed above, not always a valid assumption), the increase in response time is due solely to increases in the request queue time. While device utilization is relatively low, the response time curve remains reasonably flat. But by the time the device reaches 50% utilization, in the case of M/M/1, the average queue length is approximately 1.

At 50% utilization in an M/M/1 model, the amount of queue time that requests encounter is equal to the service time. To put it another way, at approximately 50% busy, you can expect that queue time results in response time being *double* the amount of time spent actually servicing the request. Above 50% utilization, queue time increases even faster and more and more queue time delays accumulate. This is an example of an *exponential* curve where queue time (and response time) is a *non-linear* function of utilization. As resources saturate, queue time comes to dominate the application response time that customers experience.

The case of M/D/1 shows queue time for a uniform service time distribution that is precisely ½ of the M/M/1 case with the same average service time. Evidently, reducing the variability of the service time distribution works to reduce queue time delays significantly. Many tuning strategies exploit this fact. If work requests are *scheduled* to create a more uniform service time distribution, queue time – and response time – are significantly reduced. As mentioned above, that is why supermarkets, for example, separate customers into two or three sets of lines based on the number of items in their shopping carts. This smoothes out the service time distribution at each server and reduces the overall average queue time for shoppers waiting in line to check out.

By the way, the formula used to derive the Queue time from utilization and service time for an M/M/1 model is shown below:

- $W_q = (W_s * u) / (1 - u)$

In this formula, *u* is the device utilization, W_s is the service time, and W_q is the queue time. Note the divisor term, which is ***1 – u***. As u → ∞, W_q also approaches infinity. W_q, of course, is undefined for ***u*** = 1, or utilization at 100% busy. As discussed above, the formula applies to an open queuing model.

Figure 2-9 charts this formula for W_s = 10 ms, illustrating that the relationship between the queue time W_q and utilization is nonlinear. Up to a point, queue time increases gradually as utilization increases, and then each small incremental increase in utilization causes a sharp increase in response time. Notice that there is an inflection point or “knee” in the curve where this nonlinear relationship becomes marked.[8]

1.

[8] Since the line chart in Figure 2.9 is the graph of a continuous function, some authorities doubt whether there is an actual “knee” in the response time curve that the M/M/1 model depicts. A more empirical description of

Notice also that as the utilization $\boldsymbol{u \rightarrow 1}$, the denominator term $\boldsymbol{1 - u \rightarrow 0}$ where $\boldsymbol{W_q}$ is undefined. This accounts for the steep rise in the right-hand portion of the curve as u → 1.

A chart like the one shown in 2.9 illustrates some of the reasons why it is so difficult to optimize for both maximum throughput and minimal queue time delays:

- As we push a computer system toward its maximum possible throughput, we also force response time to increase rapidly.
- Reaching for maximum throughput also jeopardizes stability. Near the maximum throughput levels, slight fluctuations in the workload cause disproportionately large changes in the amount of queue time delay that transactions experience.
- Maintaining the stability of application response time requires low utilization levels, which may not be cost-effective.

Figure 2-9 strongly suggests that the three objectives we set earlier for performance optimization are not easily reconcilable, which certainly makes for an interesting job assignment.

For the sake of a concrete example, let us return to our example of the wait queue for a disk. As in Figure 2-9, let's assume that the average service time $\boldsymbol{W_s}$ at the disk is 10 ms. When requests arrive and the disk is free, requests can be processed at a maximum rate of $\boldsymbol{1/W_s\ (1 \div 0.010)}$ or 100 requests per second. That is the capacity of the disk. To maximize throughput, we should keep the disk constantly busy so that it can reach its maximum capacity and achieve throughput of 100 requests/sec. But to minimize response time, the queue should always be kept empty so that the total amount of time it takes to service a request is as close to 10 milliseconds as possible.

If we could just control the arrival of customer requests (perhaps through scheduling), timing each request so that one arrived precisely every $\boldsymbol{W_s}$ seconds (once every 10 ms), we would improve performance. If the service time distribution is also uniform, when requests are paced to arrive regularly every 10 milliseconds, the disk queue is always empty and each successive request can be processed immediately. This succeeds in making the disk 100% busy, attaining its maximum potential throughput of $\boldsymbol{1/W_s}$ requests per second. With perfect scheduling and perfectly uniform service times for requests, it *is* possible to achieve 100% utilization of the disk with no queue time delays.

1.

this curve is that there is an *inflection point* on the curve where the slope = 1. Above that inflexion point, an increase of $\boldsymbol{x}$ in the utilization produces an increase in response time of $\boldsymbol{mx}$, where $\boldsymbol{m > 1}$. Below that inflexion point, an increase of $\boldsymbol{x}$ in the utilization produces an increase in response time of $\boldsymbol{mx}$, where $\boldsymbol{m < 1}$. Whether or not you perceive a "knee" in the curve or not, clearly, the underlying system behavior is characteristically different above the inflexion point than below it.

In reality, we have little or no control over when I/O requests arrive at the disk in a typical computer system. In a typical system, there are periods when the server is idle and others when requests arrive in bunches. (The arrival rate is bursty.) Moreover, some requests are for large amounts of data located in a distant spot on the disk, while others are for small amounts of data located near the current disk actuator. In the language of queuing theory, neither the arrival rate distribution of customer requests or the service time distribution is uniform. This lack of uniformity causes some requests to queue, with the further result that as the throughput starts to approach the maximum possible level, significant queuing delays occur. As the utilization of the server increases, the response time for customer requests increases significantly during periods of congestion, as illustrated by the behavior of the simple M/M/1 queuing model in Figure 2.9.

In most real world cases, there is an upper limit on $\boldsymbol{W_q}$ based on the size of the population. In the case of a web portal, the upper limit on the size of the population can be quite large. In the case of a queue where disk requests are parked when the device is busy, the size of the population from which arrivals are drawn is likely to be quite small, limited to the number of files that are currently active on the disk in the case of a file server, or the number of active threads waiting on IO, in the case of an application like SQL Server. Moreover, in the case of disk IO, much of the activity has the character of being *scheduled*, not independent arrivals. If you consider the disk activity associated with accessing a file on a file server, successive requests are issued in a serial fashion. Logically, the application does not request the next record in the file until the previous request is satisfied. The simple assumptions of the M/M/1 model are not always applicable for disk IO.[9]

LITTLE'S LAW

A mathematical formula known as Little's Law relates response time and utilization. In its simplest form, Little's Law expresses an equivalence relation between response time $\boldsymbol{W}$, the arrival rate λ, and the number of customer requests in the system, $\boldsymbol{Q}$ (also known as the Queue Length):

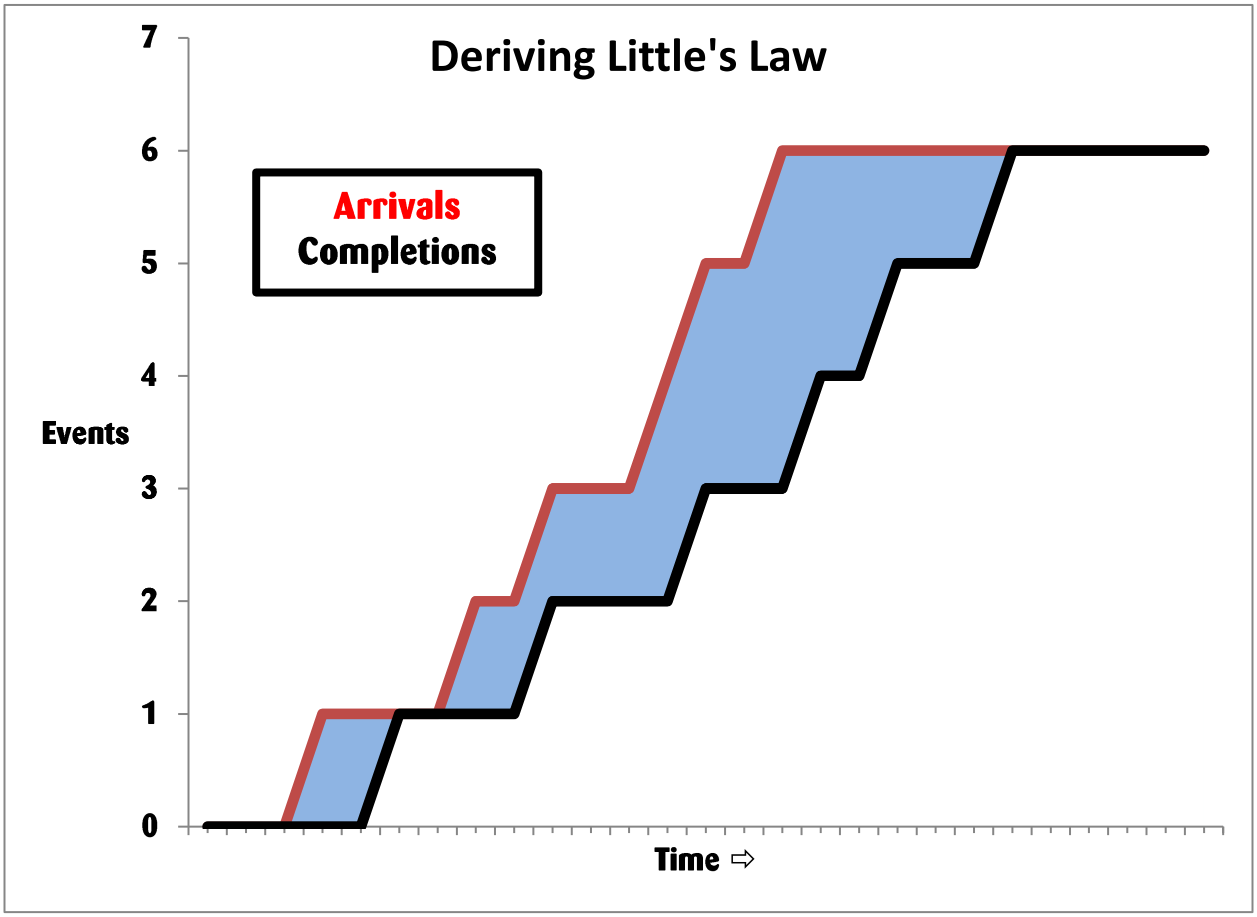
- $Q = \lambda * W$

1.

[9] The simple M/M/1 model we discuss here was chosen because it is simple, not because it is necessarily representative of real behavior on live computer systems. There are many other kinds of queuing models that reflect arrival rate and service time distributions different from the simple M/M/1 assumptions. However, although M/M/1 may not be realistic, it is easy to calculate. This contrasts with G/G/1, which uses a general arrival rate and service time distribution, i.e., any kind of statistical distribution: bimodal, symptomatic, etc. No general solution to a G/G/1 model is feasible, so mathematicians that employ computer models are often willing to compromise reality for solvability.

Note that in this context the queue length Q refers both to customer requests in service (***Q**ₛ*) and waiting in a queue (***Q**q*) for processing.

Little’s Law is an important enough result that it is probably worth showing how to derive it. Little’s Law is a very general result in queuing theory that applies to a large class of queuing models —only the equilibrium assumption is required to prove it. Figure 2-11 shows some measurements that were made at disk at discrete intervals shown on the X axis. The Y axis indicates the number of disk requests during the interval. Our observation of the disk is delimited by the two arrows between the start time and the stop time. When we started monitoring the disk, it was idle. Then, whenever an arrival occurred at a certain time, we incremented the arrivals counter. That variable is plotted using the thick line. Whenever a request completed service at the disk, we incremented the completions variable. The completions are plotted using the thin line.



The difference between the arrival and completion lines is the number of requests at the disk at that moment in time. For example, at the eleventh tick-mark on the X axis to the right of the start arrow, there are two requests at the disk, since the arrival line is at 4 and the completion line is at 2. When the two lines coincide, it means that the system is idle at the time, since the number of arrivals is equal to the number of departures.

To calculate the average number of requests in the system during our measurement interval, we need to sum the area of the rectangles formed by the arrival and completion lines and divide by the measurement interval. Let's define the more intuitive term "accumulated time in the system" for the sum of the area of rectangles between the arrival and completion lines, and denote it by the variable P. Then we can express the average number of requests at the disk, ***N***, using the expression:

$$N = \frac{P}{T}$$

We can also calculate the average response time, R, that a request spends in the system (***Ws + Wq***) by the following expression:

$$R = \frac{P}{C}$$

where ***C*** is the completion rate (and ***C = λ***, from the equilibrium assumption). This merely says that the overall time that requests spent at the disk (including queue time) was ***P***, and during that time there were ***C*** completions. So, on average, each request spent ***P/C*** units of time in the system.

Now, all we need to do is combine these two equations to derive Little's Law:

$$N = \frac{P}{T} = \frac{C}{T} \times \frac{P}{C} = \lambda \times R$$

Little's Law says that the average number of requests in the system is equal to the product of the rate at which the system is processing the requests (with ***C = λ***, from the equilibrium assumption) times the average amount of time that a request spends in the system.

To make sure that this new and important law makes sense, let's apply it to our disk example. Suppose we collected some measurements on our system and found that it can process 200 read requests per second. The Windows Performance Monitor also told us that the average response time per request was 20 ms (milliseconds) or 0.02 seconds. If we apply Little's Law, it tells us that the average number of requests at the disk, either in the queue or actually in processing, is 200 × 0.02, which equals 4. This is how Windows derives the Physical Disk Avg. Disk Queue Length and Logical Disk Avg. Disk Queue Length counters. More on that topic in Section 2.

Applying Little's Law

In Windows there are several places where the developers try to take advantage of Little's Law to estimate response time of requests where only the arrival rate and queue length are known. For the record, Little's Law is a very general result. It allows you to estimate response time in a situation where measurements for both the arrival rate and the queue length are available. Note that Little's Law itself provides no insight into how response time is broken down into service time and queue time delays.

Direct measures of application response time are not as plentiful in Windows, and this is unfortunate. However, it is very difficult to define suitable boundaries for requests in Windows desktop applications, considering the sometimes continuous flow of mouse-move messages, for example. This perhaps explains why there are not more direct measurements of response time available. Using Little's Law, there is at least one instance that will be discussed later in this section of the book where it is possible to derive estimates of response time from the available performance counters, but these are inferior to direct measurements.

The response time to service a request at a resource is usually a nonlinear function of its utilization. This nonlinear relation between response time and utilization that usually holds is known as Little's Law. Little's Law explains why linear scalability of applications is so difficult to achieve. It is a simple and powerful construct, with many applications to computer performance analysis. However, don't expect simple formulas like Little's Law to explain everything in computer performance. The next Section, for example, will highlight several common situations where intelligent scheduling algorithms are used that actually reduce service time at some computer resources the busier they get. You cannot apply simple concepts like Little's Law unreflexively to many of the more complicated situations you can expect to encounter.

Queuing theory in practice

This brief mathematical foray is intended to illustrate the idea that attempting to optimize throughput while at the same time minimizing response is inherently difficult, if not downright impossible. It also suggests that any tuning effort that makes the arrival rate or service time of customer requests more uniform is liable to be productive. It is no coincidence that this happens to be the goal of many popular optimization techniques. It further suggests that a better approach to computer performance tuning and optimization should stress analysis, rather than simply dispensing advice about what tuning adjustments to make when problems occur.

The first thing you should do whenever you encounter a performance problem is to measure the system experiencing the problem to try to understand what is going on. Computer performance evaluation begins with systematic, empirical observations of the systems at hand.

In seeking to understand the root cause of many performance problems, I rely on a practical approach to the analysis of computer systems performance that is informed by theory, drawing heavily on insights garnered from queuing theory. Ever notice how construction blocking one lane of a three-lane highway causes severe traffic congestion during rush hour? You may notice a similar phenomenon on the computer systems you are responsible for when a single component becomes bottlenecked and delays at this component cascade into a problem of major proportions. Network performance analysts use the descriptive term *storm* to characterize the chain reaction that sometimes results when a tiny aberration becomes a big problem.

Long before chaos theory, it was known that queuing models of computer performance can accurately depict this sort of behavior mathematically. Little's Law predicts that as utilization of a resource increases linearly, response time increases exponentially. The nonlinear relationship that was illustrated in Figure 2-9 means that quite small changes in the workload can result in extreme changes in performance indicators, especially as important resources start to become bottlenecks.

This is exactly the sort of behavior you can expect to see in the computer systems you are responsible for. Most computer systems do not degrade gradually. Queuing theory is a useful tool in computer performance evaluation because it is a branch of mathematics that models systems performance realistically. The painful reality we face is that performance of the systems we are responsible for is acceptable day after day, until quite suddenly it goes to hell in a handbasket. This abrupt change in operating conditions often begets an atmosphere of crisis. The typical knee-jerk reaction is to scramble to identify the thing that changed and precipitated the emergency. Sometimes it is possible to identify what caused the performance degradation, but more often it is not so simple.

With insights from queuing theory, we learn that it may not be a major change in circumstances that causes a major disruption in current performance levels. In fact, the change can be quite small and gradual and still produce a dramatic result. This is an important perspective to have the next time you are called upon to diagnose a Windows performance problem. Perhaps the only thing that changed is that a component on the system has finally reached a high enough level of utilization where slight variations in the workload provoke drastic changes in behavior.

A full exposition of queuing theory and its applications to computer performance evaluation is well beyond the scope of this book. Readers interested in pursuing the subject might want to look at Daniel Mensace's *Performance by Design* or Raj Jain's encyclopedic *The Art of Computer Systems Performance Analysis*.

EXPLORATORY DATA ANALYSIS

Investigating the cause of some form of performance problem requires computer measurement. Understanding what is measured, how measurements are gathered, and how to interpret them is the critical to computer performance evaluation. You cannot manage what you cannot measure, but fortunately, measurements are pervasive. Many, many hardware and software components support measurement interfaces.

In fact, due to the volume of computer measurement data that is available, it is frequently necessary to use statistical techniques to summarize and report computer usage data. This certainly holds true on Windows, which is capable of providing copious amounts of measurement data. Hopefully, the Reader has a solid background in basic statistical concepts, including descriptive statistics, multivariate analysis, and time series data.

Sifting through all the measurements that are potentially available demands an understanding of a statistical approach known as *exploratory data analysis*,[10] rather than the more familiar brand of statistics that uses probability theory to test the likelihood that a hypothesis is true or false. For a good reference introducing exploratory data analysis, see the introductory chapter in the NIST/Sematech *Engineering Statistics Handbook* that is published online at http://www.itl.nist.gov/div898/handbook/index.htm. This books make frequent use of visual displays of information, including histograms, scatter plots and box plots.

Many of the case studies described in this book apply the techniques of exploratory data analysis to understanding Windows computer performance. Rather than explore the copious performance data available on a typical Windows platform at random, however, our search is informed by the workings of computer hardware and operating systems in general, and of the Windows OS, Intel processor hardware, computer disks, and network interfaces in particular.

Because graphical displays of information are so important to this process, techniques of scientific visualization are also relevant. (It is always gratifying when there is an important-sounding name for what you are doing!) Edward Tufte's absorbing books on scientific visualization, including *Visual Explanations* and *The Visual Display of Quantitative Information*, are an inspiration. In many place in this book, we will illustrate the discussion using exploratory data analysis techniques. In particular, we will look for key relationships between important measurement variables. Those illustrations rely heavily on visual explanations of charts and graphs to explore key correlations and associations. It is a technique I recommend highly.

1.

[10] John Tukey's classic textbook entitled *Exploratory Data Analysis* (recently back in print) or *Understanding Robust and Exploratory Data Analysis* by Hoaglin, Mosteller,and Tukey provide a very insightful introduction to using statistics in this fashion.

In computer performance analysis, measurements, models, and statistics remain the tools of the trade. Knowing what measurements exist, how they are taken, and how they should be interpreted is a critical element of the analysis of computer systems performance. That topic is addressed in detail in Section 2 of this book, which is designed to help you acquire a firm grasp of the most important categories of measurement data available in Windows, how it is obtained, and how it should be interpreted.