CHAPTER 1

Scalable and High-Performance Web Applications

The Emergence of Web Applications

In the span of just a few years, the Internet has transformed the way information is both provided and consumed throughout the world. Its hardware and software technologies have made it possible for anyone not only to be an information consumer, but also for nearly anyone to be an information provider. Although the Internet—specifically the World Wide Web (the Web)—has been treated seriously as a platform for information sharing among the mass public for only a short time, many organizations have managed to create useful Web applications that provide significant value to consumers.

These Web applications allow consumers to buy books and compact discs online. They enable businesses to use the Internet for secure data transactions. Workers use Web applications to find jobs; employers use them to find employees; stocks are bought and sold using online applications provided by brokerages; and travelers book flight and hotel reservations using Web applications. The list goes on and on. Obviously, many useful Web applications are available on the public Internet as well as within countless corporate intranets today.

This book describes general techniques for building *high-performance and scalable enterprise Web applications*. Generally speaking, this means building applications that are reasonably and consistently fast and have a strong, gradual tolerance for rising user and request demands. Although we will spend a lot of time considering this topic in general, the core of our discussion will be phrased in terms of a solution built around the Java 2 Enterprise Edition (J2EE) specification. Now, before we dive into the details of building these kinds of applications, it is important to identify and understand the overall problem. More specifically, it is important to define *Web applications* and *scalability*.

Basic Definitions

In this book, *Web application* has a very general definition—client/server software that is connected by Internet technologies to route the data it processes. By "Internet technologies," I mean the collection of hardware and software that comprises the network infrastructure between consumers and providers of information. Web applications can be made accessible *by specific client software or by one or more related Web pages that are logically grouped for a specific productive purpose*. That purpose can be one of any number of things, for example, to buy books, to process stock orders, or to simply exist as content to be read by the consumer.

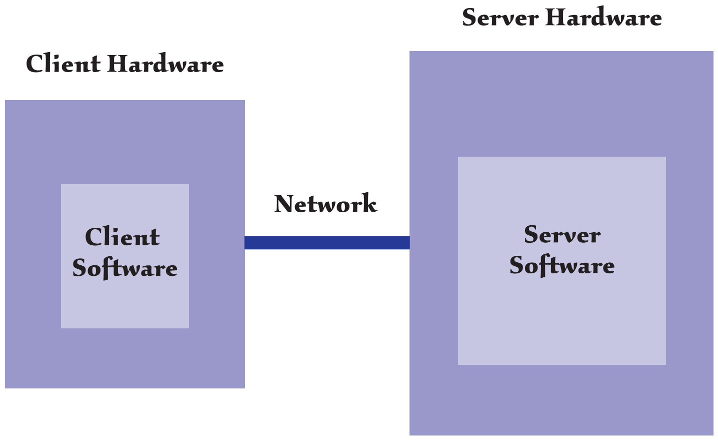
Notice that our discussion is about Web applications, not just "Web sites." In truth, the difference between the two is essential to understanding one of the key themes of this book. Most nonengineers do not make a distinction between a Web site and a Web application. Regardless of the term, it's the thing that allows them to buy their books online, to make plane reservations, to purchase tickets, and so forth.

If you're an engineer, however, there is a difference. For you, it's likely that when someone talks about, say, the performance of a Web site, you start thinking of back-end details. And so do I. Your mind begins to consider if it's running an Apache or IIS and whether it works using Java servlets, PHP, or CGI-bin Perl scripts. This difference in thinking between engineers and nonengineers could be confusing. Engineers, by habit, tend to associate "Web site" with the server side. However, as we all know, there is more to a Web application than just the server side; there's the network and the client. So, based on just that, a Web site (server) is not the same thing as a Web application (the client, network, and server).

While this book emphasizes server-side solutions, it is also concerned with client-side and networking topics because they have a fundamental impact on how end users perceive Web applications. That is, we will be concerned with the *end-to-end* interaction with a Web site, which simply means from *client to server and back to client*. This is a reasonable focus. After all, most people who use the Web are concerned with its end-to-end behavior. If it takes them a while to buy concert tickets online, it doesn't matter if the problem is caused by a slow modem, an overtaxed server, or network congestion. Whatever the reason(s), the effect is the same—a slow application that's eating up time. As engineers, we are concerned not only that such applications might be slow for one user, but also that the system becomes slower as more users access it.

Now that we have a better fix on the scope of a Web application, let us review its core components. These are the major pieces of any online application and each represents an opportunity—a problem or a challenge, depending on how you look at it. Although you're probably familiar with the components, it doesn't hurt to make sure everyone is on the same page, especially since these terms appear throughout the book. Let's start with the client side.

Figure 1–1
Client, network, and server



We will say that Web applications are used by consumers via client software (i.e., Web browsers or applications that use the Web to retrieve or process data) running on client hardware (i.e., PCs, PDAs). Application data is provided and processing is handled by producers via server software (i.e., Web server, server-side component software, database) running on server hardware (i.e., high-end multiprocessor systems, clusters, etc.). Connecting the client to the server (from the modem or network port on the client device to the networking equipment on the server side) is the *networking infrastructure*. Figure 1–1 shows the client/server relationship graphically. Notice that the server side is bigger; in general, we assume that the server side has more resources at its disposal.

At this point, it is important to distinguish one piece of server software, the Web server, because it nearly always plays a central role in brokering communication (HTTP traffic) between client and server. In this book, when I refer to the "server side," I am nearly always including the Web server. When it is necessary to distinguish it from the rest of the software on the server side, I will do so explicitly.

The Nature of the Web and Its Challenges

Although Web applications have rapidly made the Internet a productive medium, the nature of the Internet poses many engineering puzzles. Even the most basic of challenges—engineering how a provider can quickly and reliably deliver information to all who want it—is neither simple nor well understood. Like other challenges, this problem's complexity has to do with the nature of the medium. The Internet is different from the information-sharing paradigms of radio, television, and newspapers for several reasons. Perhaps two of the most important reasons are its incredibly wide audience (unpredictable number of customers) and the potential at any time for that wide audience to request information from any given provider (unpredictable work demands).

Unlike in other media, Internet information providers simply do not have the ability to know their audience in advance. Newspapers, for example, know their circulation before they print each edition. They also have the advantage of being able to control their growth, making sure they have enough employees to deliver the paper daily, and have the resources and time to go from deadline on the previous night to delivery on the next morning. Furthermore, newspapers do not have to deal with sudden jumps in circulation. Compared to the Internet, the growth of big newspapers in metropolitan areas seems far more gradual. For example, when the *Washington Post* was founded in 1877, it had a circulation of 10,000. By 1998, that circulation had reached nearly 800,000 for its daily edition and more than that for its Sunday edition.* That's an average growth rate of just over 6,500 subscribers per year, or 17 per day.

Deployers of Web applications have a love/hate relationship with their growth rates. In one sense, they would love the gradual growth of 17 new users per day. How nice life would be if you had to worry about scaling at that rate! You could finally go home at 5 P.M., not 9:30 P.M. At the same time, such growth rates are the reason that people are so excited about Web applications—because you can potentially reach the whole world in a matter of seconds. Your growth rate out of the gate could be hundreds of thousands of users. Although this bodes well for the business side of the things, it creates a tremendous challenge in terms of dealing with such demands.

On the Internet, circulation is akin to *page hits*, that is, the number of requests for a given document. Page hits can jump wildly overnight. A favorite example in the Web-caching community is the popularity of the online distribution of the Starr report. As most Americans know, this report was put together by the Office of the Independent Counsel during the Clinton administration. Let us just say that, while it was not flattering by any means, it was eagerly awaited by both the American public and the international press corps.

When the Starr report was released online in the summer of 1998 at government Web sites, tens of thousands of people tried to download it. A representative for Sprint, Inc., one of the Internet's backbone providers, reported a surge in bandwidth demand that ranged between 10 and 20 percent above normal; a representative of AOL reported an "immediate 30 percent spike"; and NetRatings, a Nielsen-like Internet content popularity company, estimated that at one point, more than

*Source: *http://www.thewashingtonpost.com*.

one in five Web users was requesting the report or news about it. CNET.COM ran a number of stories about the event and its ramifications for Internet scalability in the Fall of 1998.*

The conclusion among network administrators and engineers was universal. There were simply too many requests to be handled at once, and the distribution mechanisms were unable to scale to demand. It was a real test of the scalability of the Internet itself. Not only were the Web servers that provided this information overloaded, but the networking infrastructure connecting consumers to providers became heavily congested and severely inefficient. The effect was much like that of a traffic jam on a freeway.

This phenomenon was unique because it demonstrated the effects of sudden popularity as well as the short-lived nature of that popularity. For example, it is unlikely that you or anyone else remembers the URL(s) where the report was first available. And it is unlikely that you have it bookmarked. Thus, even had those sites been able to accommodate the demands of the time by buying bigger and faster machines, it would likely have been money wasted because the need for those resources dropped dramatically after the public lost interest in the report.

Other media, such as radio and television, are broadcast and do not need to worry about the size of their audience affecting their ability to deliver information. Consider television or radio programs, such as the national and local news. Their programmers know in advance when they are scheduled to broadcast. They have the luxury of being able to prepare ahead of time. Even when live radio or television beckons, the fact that both media are broadcast means that there is only one audience to address. Cable companies and good old TV antennae are already in place to facilitate the transport of that information. If we all watch or listen to the same channel, we all see or hear the same program. This is not the case with Internet audiences, where it is usually impossible to prepare for every request, where every consumer of information requires a unique response, and where there is a continual need for new virtual links (HTTP connections) between consumer and provider to be both created and then destroyed.

Performance and Scalability

Have you ever gone to a Web site, clicked on a link, and really *waited* for a response? Of course you have; we all have. It's annoying and frustrating. Worst are those content-laden sites that are meant to be read like newspapers. You want to jump from link to link, but every time you click, you have to wait *seconds* (not milliseconds) for

*Source: *http://news.cnet.com/news/0-1005-204-332427.html*.

the page and the ads and the embedded applets to download. You almost begin to hate clicking on a link because you know you will have to wait. You've learned to classify this kind of site as *slow*.

Then there are sites that are suspiciously slow. In these cases, you have reason to believe that bazillions of people are trying to connect, and this mass, not the technology, is responsible for the slowness. Say you're ordering a book at a site that has just announced a 50%-off sale. Or suppose tickets for a really hot concert have just gone on sale. When you're able to connect, the site seems unresponsive. When it does respond, it crawls. You guess that the site is buckling under the demand caused by the event. You've learned to classify this kind of site as *not scalable*.

As users, we have learned what poor performance and scalability are because we have experienced them. As engineers, we would like to understand these faults better so that our own users don't experience them. Because that is the focus of this book, let's start our discussion of performance and scalability by defining our terms.

Performance

Performance can be described simply as the raw speed of your application in terms of a single user. How long does a single application-level operation take? How long does it take to search for a book? How long does it take to confirm an online registration once we click Confirm? How long does it take to check out and pay at an online store? Notice that some of these examples describe atomic operations and some don't. When describing performance, we have to be clear if we are talking about one application operation or an entire session.

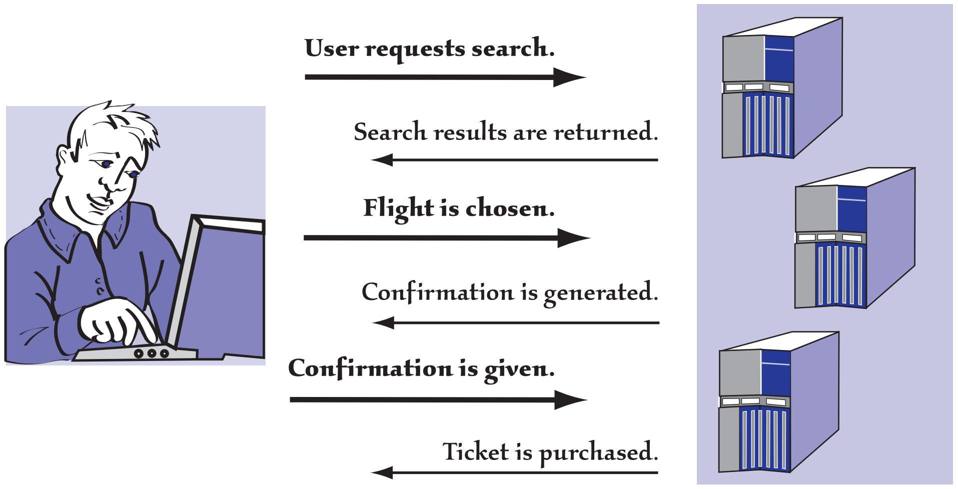
Consider the user interaction required to buy an airline ticket in Figure 1–2: In this session, there are three application operations, each consisting of a roundtrip between client and server. The operations are listed in Table 1–1 with their code names.

When we are talking about the performance of an operation, such as selection, we are interested in the end-to-end time required to complete that operation. In other words, the clock starts ticking when the user clicks the button and stops ticking when the user sees the information delivered. Why all this focus on end-to-end performance?

Table 1–1: Application Operations

Code Name	*User Action*	*Server Action*
`Search`	Criteria specified	Search based on criteria
`Selection`	Flight chosen	Confirmation for that flight generated
`Confirmation`	Flight confirmed	Confirmation processed

Figure 1–2 Application operations associated with buying an airline ticket



We could, of course, judge performance by measuring the speed of the Web server's response, of the network, of our database retrievals, and so on. But we know that all of these performance marks are irrelevant when compared to the overall time for a logical operation. Although unit performance numbers make us happy or proud (especially if we designed that piece of the application!), end-to-end performance is the one that really counts—this is the metric that either scares users off or wins their loyalty. And thus, this is the one that can spell life or death for your application.

Addressing end-to-end performance means making operations faster for the user. To do that, we can improve the unit performance of some of the components involved in the operation(s). For example, we can improve the performance of the Web server, the database, or the application servers. The exact solution (e.g., better algorithms, more efficient queries, etc.) depends on the unit being tuned. The point is that measuring performance should be a top-down process: Start with the user, move to the components, and then to parts in the components. Look for trends and ask if a single instance of poor performance can be traced to a larger, general problem.

Scalability

Informally, engineers describe the challenge of dealing with large audiences and high demand as a problem of **scalability**. More specifically, we say that a Web application can scale if it continues to be available and functional at consistent speeds as the number of users and requests continues to grow, even to very high numbers. A provider's inability to deliver a document, such as the Starr report, because of server overload was thus a problem of scalability. Note that this definition has nothing to do with performance. As long as a slow application continues to provide consistent performance in the wake of rising demand, it is classified as scalable!

Now, although *scalability* is commonly defined strictly as a measurement of resiliency under ever-increasing user load, nobody expects a single instance of an application server on a single machine to accommodate millions of users. Often people consider how well an application can "scale up" by describing how effective it is to add resources, such as more CPUs, more memory, or more disks. An application is considered to scale up well if it requires additional resources at a low rate. For example, if we need to add 300MB RAM per 10 concurrent users on our system, we are in trouble. As I discuss later, this scale-up attribute is often represented as a cost, for example, a cost per concurrent transaction.

Generally, three techniques can be employed to improve scalability:

- Increase the resources (bigger machines, more disk, more memory).
- Improve the software.
- Increase the resources *and* improve the software.

Although the long-term answer is the third technique, our bias is toward the second. Good design at the beginning of a project is the most cost-effective way to improve scalability. No doubt you will need greater resources to deal with higher demands, but this is never the whole story. Although it can take the purchaser part of the distance, throwing money at the problem cannot ensure scalability. I don't deny the need to spend money at certain points in the process. Rather, I suggest strategic places to spend and strategic opportunities during the design that can give application designers the biggest bang for their buck, thereby reducing their need to purchase more resources than necessary.

The Internet Medium

Six attributes of the Internet as a medium compound the challenge of delivering performance and scalability. The better we understand and appreciate these attributes, the more strategic we can be in meeting the challenge to build Web applications that perform and scale well.

First, as mentioned earlier, there is potentially a *wide audience* for Web application providers to manage—wider than in any other medium. Second, the Web is an *interactive* medium: Consumers not only receive information, they also submit it. Third, the Internet is *dynamic* in the sense that a given user request does not always result in the same server-side response. Fourth, the Internet as a utility is *always on* and providers have no guarantees about when and how often their information will be accessed. Fifth, providing information over the Internet is an *integrated* process that often depends on the coordination of multiple provider subsystems to deliver information. And sixth, providers *lack complete control* in terms of the delivery of information to consumers: There are many networking elements that exist between provider and consumer, most of which are not controlled by the provider.

Some of these attributes may seem obvious; some may not. In either case, thinking about the details and their implications will prepare you for the solutions part of this book.

Wide Audience

I'm not going to beat you over the head with the fact that millions of people use the Internet every day. That is obvious and the increasing numbers are the primary reason that application architects worry about things like scalability in the first place. However, I will inform you of a few things that you may not know—or just may not appreciate, yet.

One is that there is another Internet "audience" to consider, one that is not often addressed. This quieter, hidden, but rapidly growing group of Web clients are better known as "bots." If you are familiar with search engine technology, you already know that search engines use automated softbots to "spider" (recursively traverse) the Web and update search engine indices. This process has been going on since search engines were first deployed; bots are a simple example of one type of *information agent*.

Today's bots are just the tip of the iceberg. More sophisticated information agents are just around the corner that will allow users to monitor multiple sites continuously and automatically. For example, instead of using the Web interactively to watch and participate in online auctions (like those at eBay and Yahoo), users will configure information agents to watch continuously and bid automatically. This is an inevitable and obvious future direction for the Web: People want to do more than sit around watching their monitors all day, manually hunting for information.

Bots and information agents are particularly fond of things like data feeds, which are information sources that continually change and require monitoring. When the Internet was first being commercialized, it was popular to connect real-time data feeds (such as the newswire services) and build access methods to them via Web applications. This trend shows no sign of slowing; in fact, it threatens to become much greater as Web applications gradually become data feeds in themselves.

I've avoided boring, albeit frightening, statistics about the growing number of human Internet users. Instead, I've reminded you that there are and will be new types of application clients, not just those with two eyes. An increasing number of information agents will automate Web querying and a growing trend will be to treat Web applications like data feeds. In short, the Web's audience is definitely growing, not to mention changing, and so are its demands. What's more, this newer audience is persistent and regular, and does not mind testing the 24x7 feature of the Web and its applications!

Interactive

On the Internet, consumers query providers for information. Unlike in other media, information is not distributed at the whim of the provider. Instead, consumers request information via queries, which consist of a series of interactions between the client and server.

In addition to querying, consumer requests can contain submitted information that must be processed. This submission mechanism can be explicit or implicit. Explicit submission is the user's deliberate transmission of information to the provider, such as a completed HTML form. In contrast, implicit submission is the provision of data through the user's Web session. Cookies are a good example of this, in that they consist of data that is chosen by either the provider (e.g., for page tracking) or the consumer (e.g., for personalization).

Regardless of how the information is submitted, the application's processing must often be based on this information. Thus, the Internet is not simply a library where clients request items that exist on shelves; rather, requests involve calculations or processing, sometimes leading to a unique result. Furthermore, the interactive nature of the Web means that a request cannot be fulfilled in advance—instead, the application must respond at the time the request is made, even though substantial processing may be associated with that request.

Dynamic

Web applications present information that depends on data associated with the user or session. As far as the user goes, countless demographic and historical session attributes can make a difference in how an application responds. The response may also depend on things unrelated to the user, such as a temporal variable (e.g., the season or the day or the week) or some other external real-time data (e.g., the current number of houses for sale). In any case, the data being generated by a Web application is often dynamic and a function based on user and/or session information.

The main problem that a dynamic Web application creates for the designer is the inability to use the results of prior work. For example, if you use a Web application to search for a house online, searching with the same criteria one week from the date of the first search may very well return different results. Of course, this is not always the case. If you conduct the same house search 10 minutes after the first one, you will very likely get the same results both times. Obviously, the designer must know when it is safe to reuse results and when it is not.

There is a subtle relationship between interactive and dynamic application behavior. To avoid confusion, keep the following in mind: Interactivity has to do with the Web application executing in response to a user, whereas dynamism has to do with the response being a product of the user, her response, or some temporal or external variable. Thus, dynamic behavior is the more general notion: An application response is the product of a set of variables, some user-specified, some not. Interactivity is simply one means to achieve a dynamic response. Put another way, interactivity describes a cause; dynamism describes an effect.

Always On

This Internet is never supposed to sleep. Banks advertise Web banking 24 hours a day, 7 days a week. This 24x7 mentality is part of what makes the Internet so enticing for users. People naturally assume that, at any time, it exists as an available resource. However nice this feature is for users, it is equally daunting for Web application designers. A good example of what can happen when an application is not available 24x7 is the trouble users had with eBay, the online auctioneer, in late 1999 and 2000.

During various system or software upgrades over that time, eBay suffered intermittent problems that made it unavailable to users. In June of 1999, it was unavailable for 22 hours. Since the purpose of eBay's service is to manage millions of time-limited auctions, its core business was directly affected. Instead of selling to the highest bidder, some sellers were forced to sell to the "only bidder." Users complained, demanding a reduction in fees. The problems made the news, and the company was forced to issue apologies in addition to refunding some fees. This is not to say that eBay is not a scalable service or that the system is always unstable; indeed, eBay is one of the most trafficked sites on the Internet, and except in rare instances, has done a tremendous amount of successful 24x7 processing.

However, this simple example does underscore the importance of 24x7 when it comes to Web applications. Nobody will write news stories about how well you perform 24x7 service, but they will definitely take you to task for glitches when you don't. These problems can affect your whole company, especially if part of its revenue comes via the Web.

Observant readers might argue that failure to provide 24x7 service is not a question of scalability but of *reliability*. True, the inability to provide service because of a system failure is a question of reliability and robustness. From the practical standpoint of the user, however, it does not matter. Whether the application is unavailable because of a power problem with the site's Internet service provider (as was the case in one of eBay's outages) or because the system can't handle a million simultaneous users, the result is the same: The application is unavailable.

Integrated

When consumers request information, providers often refer to multiple local and remote sources to integrate several pieces of information in their responses. For example, if you use the Internet to make an airline reservation, it is common for multiple systems (some of which are not directly connected to the Internet) to be indirectly involved in the processing of your reservation. The "confirmation code" you receive when the reservation is made comes only after all steps of the transaction have been completed successfully.

Integration on the server side is common for most Web applications. To some extent, this is a medium-term problem. The Web is a young technology and most of its important processing still involves some legacy or intermediate proprietary systems. These systems have proved reliable and have seemed scalable. Certainly, they are still part of the loop because organizations believe in their ability to handle workloads, but the question is whether these systems are ready for Internet-level scale.

Consider an airline that migrates its ticketing to the Web. To do so, server-side processing is required to connect to a remote, proprietary invoice database for each request. In the past, hundreds of phone-based human agents had no trouble using such a system to do processing. But it may be the case that, for example, there are some hard limits to the number of concurrent connections to this database. When there were never more than a few hundred agents, these limits were never exposed. However, putting such a system in the server-side mix may turn out to be the bottleneck in a Web application.

Lack of Complete Control

To a provider of information, one of the most frustrating aspects about the Web is the fact that, no matter how much money is thrown at improving application scalability, it does not mean that the application will become scalable. The culprit here is the Internet itself. While its topology of interconnected networks enables information to be delivered from anywhere to anywhere, it delivers very few quality of service (QoS) guarantees. No matter how much time you spend tuning the client and server sides of a Web application, no authority is going to ensure that data will travel from your server to your clients at quality or priority any better than that of a student downloading MP3 files all night. And despite your best efforts, an important client that relies on a sketchy ISP with intermittent outages may deem your application slow or unreliable, though no fault of your own.

In short, the problem is decentralization. For critical Web applications, designers want complete control of the problem, but the reality is that they can almost never have it unless they circumvent the Web. This is another reminder that the solution to scalable Web applications consists of more than writing speedy server-side code. Sure, that can help, but it is by no means the whole picture.

When we talk about the lack of control over the network, we are more precisely referring to the inability to reserve bandwidth and the lack of knowledge or control over the networking elements that make up the path from client to server. Without being able to reserve bandwidth between a server and all its clients, we cannot schedule a big event that will bring in many HTTP requests and be guaranteed that they can get through. Although we can do much to widen the path in certain areas (from the server side to the ISP), we cannot widen it everywhere.

In terms of lack of knowledge about networking elements, we have to consider how clients reach servers. On the Internet, the mechanism for reaching a server from a client involves querying a series of routing tables. Without access or control over those tables, there is no way that designers can ensure high quality of service.

Techniques like Web caching and content distribution allow us to influence QoS somewhat, but they don't provide guarantees. As it turns out, the lack of control over the underlying network represents the biggest question mark in terms of consistent application performance. We simply cannot understand or address the inefficiencies of every path by which a client connects to our application. The best we can do is design and deploy for efficiency and limit our use of the network, and thus limit performance variability, when possible.

Measuring Performance and Scalability

Thus far, I have defined the problem of performance and scalability in the context of Web applications, but I have not said much about their measurement. The measurement of performance and scalability is a weighty subject, and is different from the focus of this book. However, as you apply the various techniques that we cover here to your systems, you will want some simple measurements of the success of your efforts. In this section, we'll cover a few metrics that will tell you if your application is fast and scalable.

Measuring Performance

It's fairly easy to measure performance. We can use the application being tested or we can design an automatic benchmark and observe the original speed of the application against it. Then we can make changes to the software or hardware and determine if the execution time has improved. This is a very simple approach, but by far the most common metric we will use in our study.

It is important that, when measuring performance in this way, we identify the complete path of particular application operation. That is, we have to decompose it into its parts and assign values to each. Let us return to an earlier example, that of buying airline tickets online, and imagine that we're analyzing the performance of the "confirmation" process, which takes 2.8 seconds. Table 1–2 shows one possible set of results.

The way to read this table is to consider that completing the operation in the first (far left) column occurs at some point in time offset by the user's click (shown in the second column) and thus some percentage of time (shown in the third column) of the end-to-end execution. Some of this requires interpretation. For example, "Web server gets request" does not mean that the single act of getting of the request is responsible for over 6 percent of the execution time. It means that 6 percent of the execution time is spent between the initial user's click and the Web server's getting the request; thus, 6 percent was essentially required for one-way network communication. Building these kinds of tables is useful because it allows you to focus your efforts on the bottlenecks that count. For example, in Table 1–2, we can clearly see that the database query is the bottleneck.

To build accurate tables requires two important features. One is that your system be instrumented as much as possible; that is, all components should have logging

Table 1–2: Confirmation Process

Unit Action	*Elapsed Time of Action (ms)*	*End-to-End Time (%)*
User clicks	0	N/A
Web server gets request	170	6.07
Servlet gets request	178	0.29
EJB server gets request	1.68	
Database query starts	440	7.68
Database query ends	2250	64.64
EJB server replies	2280	1.07
Servlet replies	2360	2.86
User gets information	2800	15.71

features that allow them to be debugged or benchmarked. Web servers, become familiar with how these systems allow logging to be turned on and off. Make sure that you turn on logging for benchmark testing but turn it off when resuming deployment; if it's on, logging will slow down your application. Also, your code is actually the *least* likely place to be instrumented. Thus, it can be good to place some well-chosen logging statements in your code. For example, if an application server makes three queries (as part of a single transaction) before replying, it would be useful to put logging statements before each query.

The second important requirement is clock synchronization. The components being measured may be on different machines and without synchronizing your clocks, you can mistakenly assess too little or too much blame to an action that is actually much faster than you thought. Exact synchronization of clocks is a bit unrealistic, but as long as you know the clocks' relative drifts, you should be able to compensate in your calculations. Don't overdo synchronization or calibration—for example, being off by less than a hundred milliseconds for an entire operation is not a big deal because it won't be perceptible.

Beyond Benchmarking

In addition to benchmarking, there are other types of performance measurements that are well-suited to certain classes of problems. For example, suppose your Web applications are very CPU bound. To improve performance, you can add multiple processors to your system or process the problem over a cluster of workstations. Both approaches assume that it is possible to either automatically parallelize your computations or leverage explicit parallelization (i.e., thread use) and allocate parallel blocks of instructions to different CPUs/workstations. Whichever solution you choose, you will need to measure its net effect. If you don't, then you're shooting in the dark.

When trying to assess improvement in pure computational performance, we can measure the **speedup** associated with that computation. Speedup is generally defined as:

$$\text{Speedup} = T_{\text{old}}/T_{\text{new}}$$

where T_{old} is the execution time under the previous computational scenario and T_{new} is the execution time under the new scenario.

The term *scenario* is general because there are two general ways to investigate speedup: at the software level and at the hardware level. At the software level, this means changing the program code: If a program takes 10 seconds to run with the old code and 5 seconds to run with the new code, the speedup is obviously 2. At the hardware level, this means adding processors or cluster nodes. Correspondingly, for multiprocessor or cluster-based systems, the speedup metric is commonly redefined as:

$$\text{Speedup} = T_1/T_p$$

where T_1 is the execution time with one processor and T_p is the execution time when the program is run on p processors.

Ideally, speedup increases linearly, as processors are added to a system. In reality, however, this is never the case. All sorts of issues—processor-to-processor communication cost, program data hazards, and the like—contribute to an overall overhead of computing something on p processors instead of one.

Measuring Scalability

Scalability is almost as easy to measure as performance is. We know that scalability refers to an application's ability to accommodate rising resource demand gracefully, without a noticeable loss in QoS. To measure scalability, it would seem that we need to calculate how well increasing demand is handled. But how exactly do we do this?

Let's consider a simple example. Suppose that we deploy an online banking application. One type of request that clients can make is to view recent bank transactions. Suppose that when a single client connects to the system, it takes a speedy 10 ms of server-side time to process this request. Note that network latency and other client or network issues affecting the delivery of the response will increase the end-to-end response time; for example, maybe end-to-end response time will be 1,000 ms for a single client. But, to keep our example simple, let's consider just server-side time.

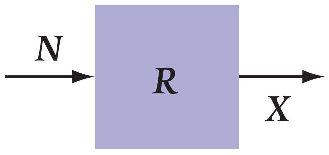
Next, suppose that 50 users simultaneously want to view their recent transactions, and that it takes an average of 500 ms of server-side time to process each of these 50 concurrent requests. Obviously, our server-side response time has slowed because of the concurrency of demands. That is to be expected.

Our next question might be: How well does our application scale? To answer this, we need some scalability metrics, such as the following:

- *Throughput*—the rate at which transactions are processed by the system
- *Resource usage*—the usage levels for the various resources involved (CPU, memory, disk, bandwidth)
- *Cost*—the price per transaction

A more detailed discussion of these and other metrics can be found in *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning* (Menasce and Almeida, 2000). Measuring resource use is fairly easy; measuring throughput and cost requires a bit more explanation.

What is the throughput in both of the cases described, with one user and with 50 users? To calculate this, we can take advantage of something called Little's law, a simple but very useful measure that can be applied very broadly. Consider the simple



Throughput = $X = N/R$.

Figure 1–3
Little's law

black box shown in Figure 1–3. Little's law says that if this box contains an average of N users, and the average user spends R seconds in that box, then the throughput X of that box is roughly

$$X = N/R.$$

Little's law can be applied to almost any device: a server, a disk, a system, or a Web application. Indeed, any system that employs a notion of input and output and that can be considered a black box is a candidate for this kind of analysis.

Armed with this knowledge, we can now apply it to our example. Specifically, we can calculate application throughput for different numbers of concurrent users. Our N will be transactions, and since R is in seconds, we will measure throughput in terms of transactions per second (tps). At the same time, let's add some data to our banking example. Table 1–3 summarizes what we might observe, along with throughputs calculated using Little's law. Again, keep in mind that this is just an example; I pulled these response times from thin air. Even so, they are not unreasonable.

Based on these numbers, how well does our application scale? It's still hard to say. We can quote numbers, but do they mean anything? Not really. The problem here is that we need a comparison—something to hold up against our mythical application so we can judge how well or how poorly our example scales.

Table 1–3: Sample Application Response and Throughput Times

Concurrent Users	*Average Response Time (ms)*	*Throughput (tps)*
1	10	100
50	500	100
100	1200	83.333
150	2200	68.182
200	4000	50

One good comparison is against a "linearly scalable" version of our application, by which I mean an application that continues to do exactly the same amount of work per second no matter how many clients use it. This is not to say the average response time will remain constant—no way. In fact, it will increase, but in a perfectly predictable manner. However, our throughput will remain constant. Linearly scalable applications are perfectly scalable in that their performance degrades at a constant rate directly proportional to their demands.

If our application is indeed linearly scalable, we'll see the numbers shown in Table 1–4. Notice that our performance degrades in a constant manner: The average response time is ten times the number of concurrent users. However, our throughput is constant at 100 tps.

To understand this data better, and how we can use it in a comparison with our original mythical application results, let's view their trends in graph form. Figure 1–4 illustrates average response time as a function of the number of concurrent users; Figure 1–5 shows throughput as a function of the number of users. These graphs also compare our results with results for an idealized system whose response time increases linearly with the number of concurrent users.

Figure 1–4 shows that our application starts to deviate from linear scalability after about 50 concurrent users. With a higher number of concurrent sessions, the line migrates toward an exponential trend. Notice that I'm drawing attention to the nature of the line, not the numbers to which the line corresponds. As we discussed earlier, scalability analysis is not the same as performance analysis; (that is, a slow application is not necessarily unable to scale). While we are interested in the average time per request from a performance standpoint, we are more interested in *performance trends* with higher concurrent demand, or how well an application deals with increased load, when it comes to scalability.

Figure 1–5 shows that a theoretical application should maintain a constant number of transactions per second. This makes sense: Even though our average response

Table 1–4: Linearly Scalable Application Response and Throughput Times

Concurrent Users	*Average Response Time (ms)*	*Throughput (tps)*
1	10	100
50	500	100
100	1000	100
150	1500	100
200	2000	100

Figure 1–4
Scalability from the client's point of view

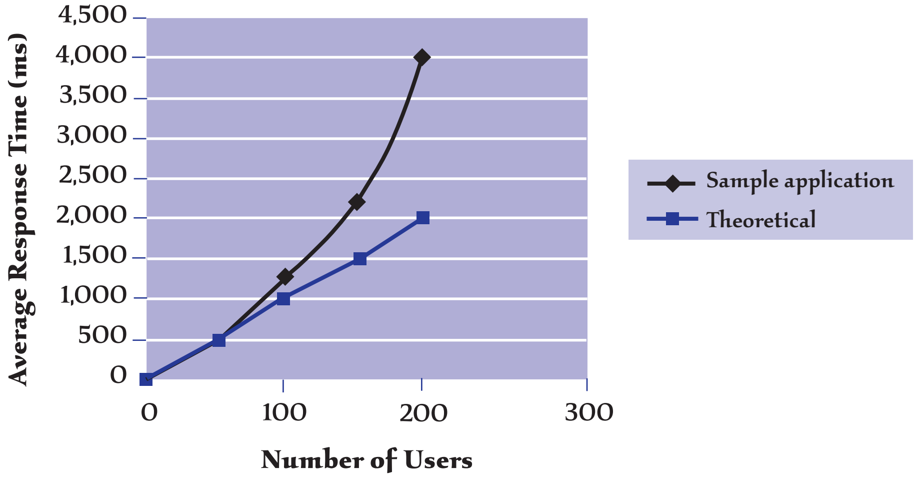
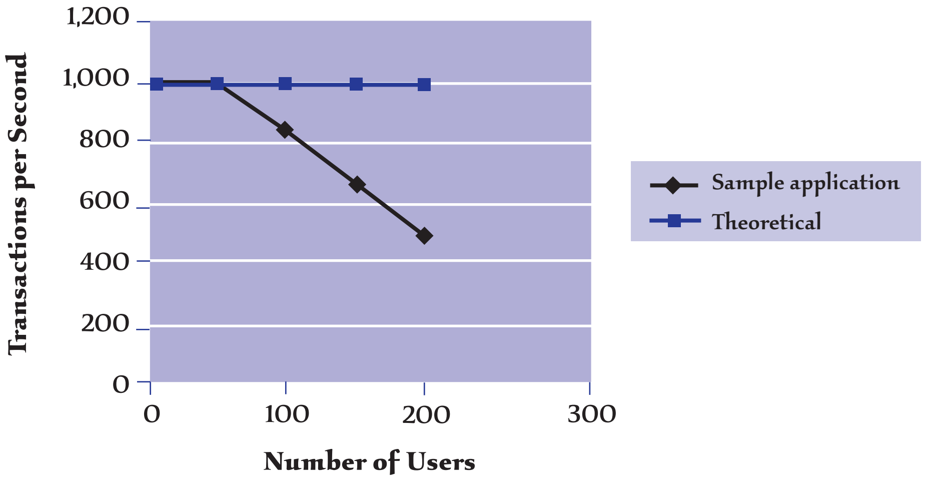


Figure 1–5
Scalability from the server's point of view



time may increase, the amount of work done per unit time remains the same. (Think of a kitchen faucet: It is reasonable that even though it takes longer to wash 100 dishes than to wash one, the number of dishes per second should remain constant.) Notice that our mythical application becomes less productive after 50 concurrent users. In this sense, it would be better to replicate our application and limit the number of concurrent users to 50 if we want to achieve maximum throughput.

Analyzing response time and throughput trends, as we have done here, is important for gauging the scalability of your system. Figures 1–4 and 1–5 show how to compare an application and its theoretical potential. Figure 1–4 illustrates the efficiency from the client's point of view, where the focus is on latency; Figure 1–5 shows application efficiency from the server's point of view, where the focus is on productivity (work done per time unit).

Throughput and Price/Performance

In measuring throughput, we have ignored the cost of the systems we are analyzing. If a system costing $100 can handle 1,000 transactions per second and a system costing $500,000 can handle 1,200 transactions per second, the latter obviously has better throughput—but it's gained at a much higher cost. The idea of measuring throughput and its relationship to price is something that has been popularized by the Transaction Processing Council (TPC), which has created database benchmarks, better known as the TPC-style benchmarks.

There are three TPC benchmarks: TPC-A, TPC-B, and TPC-C. The most recently developed (as of this writing) is the TPC-C. It measures database transaction processing in terms of how efficiently it supports a mythical ordering system. Specifically, it measures how many "new order" transactions can be handled while the system is busy handling four other types of order-related transactions (payment, status, etc.). While the TPC specification is meant to measure database throughput, you can use the same principle with your systems. After all, Web application transactions are at their core a set of database transactions.

Although it is unlikely that you will benchmark your system against another, you can measure how well your system is improving or lagging over its own evolution. For example, if release 1 of your application requires $100,000 worth of hardware and software and nets 10,000 transaction per second, you can calculate a price/performance index by dividing the price by the performance:

$$100{,}000/10{,}000 = \$10 \text{ per transaction.}$$

This doesn't mean that it costs $10 to execute a transaction on your system. It is simply a measure of throughput as it relates to the overall cost of the system. Suppose that a year later, release 2 of your application requires $150,000 worth of hardware and handles 40,000 transactions per second. The release 2 price/performance index would be:

$$150{,}000/40{,}000 = \$3.75 \text{ per transaction.}$$

Obviously, release 2 is more efficient than release 1 by evidence of its lower price/performance figure.

My interest in price/performance in this section is a reminder of the more general bias throughout this book: *Favor architectural strategies over resources when developing your application*. Once the application is deployed, you can always buy more resources to meet demand. On the other hand, rewriting code, changing designs, or re-architecting your application after deployment comes at a much higher cost. The best solution is obviously good design at the outset for scalability and performance. Not only does this eliminate the need for massive design changes after deployment, but it also typically leads to more cost-efficient resource acquisitions. CPUs, memory, disk, and other resources are purchased less frequently for applications that are inherently fast and scalable. In short, well-designed systems adapt and evolve much better than poorly designed ones do.

Scalability and Performance Hints

Nearly all of the chapters in this book include a section on hints for scalability and performance. The idea is to provide some conclusions or suggestions that have to do with the material presented in the chapter. Since we've just started our journey, there is nothing terribly complicated to conclude. However, we can remind ourselves of a few useful things covered earlier.

Think End-to-End

If nothing else, this chapter should have made clear that scalability and performance are end-to-end challenges. Don't just focus on the server; consider client and network issues as well. Spending all your time optimizing your server and database is not going to help if one part of your solution doesn't scale. You will always be hampered by your weakest link, so spend more time thinking about all of the parts involved in an application session, not just the ones you suspect or the ones you read articles about. Keep an open mind: While many applications face similar dilemmas, not all have the same clients, the same growth rate, or the same 24x7 demands.

Scalability Doesn't Equal Performance

Another thing you should have gotten out of this chapter is that scalability is not the same as performance. The two have different metrics and measure distinct things.

Performance has to do with the raw speed of the application, perhaps in a vacuum where only one user is using it. When we talk about performance, we mean response time—it's as simple as that. Optimizing performance has to do with improving the performance for that one user. If we measure average response time of 100 concurrent users, our performance challenge is to improve the average response time of the same 100 concurrent users.

Scalability, on the other hand, has to do with the ability to accommodate increasing demand. A primary metric for scalability is throughput, which measures transactions or users per second. There is no such thing as infinite scalability—the ability to handle arbitrary demand. Every application has its limits. In fact, for many deployments it is satisfying to achieve just linear scalability, although the optimizer in all of us wants to achieve much better than that. Not unexpectedly, the most successful examples of scalability are those that simply minimize the rate at which new resources are required.

Measure Scalability by Comparison

Scalability is difficult to ensure because its metrics don't allow you to compare it easily to an average (nonlinearly scalable) baseline and make some conclusions. One thing you can do, however, is measure how the scalability of your application evolves. First, define what kind of throughput is reasonable: Create (or buy) an automated stress-testing system that identifies whether your current system achieves that goal for a reasonable number of users. Then, as the application evolves, periodically retest and determine if it's improving relative to past scalability—this is without a doubt something that even your end users will notice.

Another strategy is to measure throughput as the number of users increases and identify important trends. For example, measure the throughput of your applications with 100 concurrent transactions, then with 1,000, and then with 10,000 transactions. Look at how your throughput changes and see how it compares with linear scalability. This comparison will likely give you a better sense for whether your application architecture is inherently scalable.

Summary

In this first chapter, we focused on defining Web applications and the nature of their deployment on the Internet. We also defined and discussed performance and scalability—two important concepts that will remain our focus throughout this book—and described their related metrics.

One very important subtheme of this chapter was the focus on the entire application, not just its parts. Although it may be academically interesting to optimize our bandwidth or CPU use, the end user does not care about such things. Instead, he or she thinks only in terms of time, that is, whether the application is fast. And he or she wants that same response time regardless of how many other users are on the system at the same time. Now that we are focused on the goal of end-to-end performance and scalability, let's move on to talk in more detail about application architectures and the specific challenges that lie ahead.