Pro Java EE 5 Performance Management and Optimization

Steven Haines

Apress®

Pro Java EE 5 Performance Management and Optimization

Copyright © 2006 by Steven Haines

ISBN-13: 1-59059-610-2

ISBN-10: 978-1-59059-610-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewers: Mark Gowdy, Dilip Thomas
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editors: Heather Lang, Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert
Proofreader: Liz Welch
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski
Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 9

Performance and Scalability Testing

"I sent my architect on a tuning mission following the wait-based tuning approach that you showed us. We decided to scale both vertically and horizontally inside a cluster, carefully choosing our replication strategy. So now how do I set my expectations for the environment?"

John took a step that I was hoping that he would. He recognized that, while tuning efforts and deployment planning exercises improve the performance of your applications, they do nothing to instill you with confidence until after you have run performance and scalability tests and can support the effectiveness of your efforts with hard numbers.

"That is a very good question. Remember when we talked about the behavior of an application when it is exposed to an increasing user load?"

"Do you mean the response time, utilization, and throughput graph?" John asked.

"Exactly," I replied. "As user load increases, the system is required to do more, so resource utilizations increase; the application performs more work, so throughput increases; and thread contexts switch, so response time is marginally affected. But there is a point where resources saturate, causing the system overhead to hinder application performance, reduce throughput, and finally increase the response time exponentially." Whenever I describe this graph (see Figure 9-2) without any props, my arms move in upward curves and crosses, which are meaningless to anyone who has not seen the graph before. Luckily John knew exactly what I was talking about and tolerated my flapping arms.

"Yes, that graph scares me, because when things start going bad, they go bad very quickly!"

"Yes, so the key to attaining confidence in your tuning efforts and deployment planning exercises is to run performance and scalability tests. And the final goal in running these tests is overlaying your performance metrics on top of such a graph. In the end, you can state with confidence the number of users your application can support while meeting SLAs and how it behaves afterward. Let me show you the strategies I employ to perform these tests."

Performance and scalability testing are crucial to the successful deployment of your applications. You can design solid applications, test each component individually, test the integrated solution, and even test your applications in a production staging environment, but until you discover the limits of your environment, you will never attain sufficient confidence that your application can meet the changing needs and expectations of your users.

Here's a coarse comparison: back in 1997, I bought a 1995 Mazda RX7 (the Batmobile-looking car). It is a high-performance sports car boasting acceleration from 0 to 60 mph in less than five seconds and a top speed of over 160 mph. It corners fast and handles incredibly, but it has a limitation referred to as "unpredictable oversteer"—it can take corners very fast, but once you pass a certain threshold, the car's behavior is unpredictable. Shortly after buying the car, I took my friend Chris for a ride, and we discovered that threshold. I approached a tight corner, and instead of letting off the gas, I accelerated. The car performed two 360-degree spins before coming to a (thankfully) safe stop. Needless to say, Chris and I needed to take another short trip around the neighborhood to restart our hearts, but I never pushed the car past the limit that I discovered.

Let's bring this back to performance and scalability testing: you may be comfortable with the performance of your application in general and even with its performance at your current or projected usage, but until you discover its breaking point, you will always experience uncertainty every time marketing runs another promotion or a new, high-profile customer endorses your application. The key to attaining this confidence is to perform the following tasks:

- Assess the performance of your application at expected load.
- Determine the load that causes your application to exceed its SLAs.
- Determine the load that causes your application to reach its saturation point and enter the buckle zone.
- Construct a performance degradation model from expected usage to exceeded SLAs to saturation point.

With this information in hand, you will be well equipped to project the effects of changes in usage patterns on your environment and intelligently recommend environment changes to your CIO.

Performance vs. Scalability

The terms "performance" and "scalability" are commonly used interchangeably, but the two are distinct: *performance* measures the speed with which a single request can be executed, while *scalability* measures the ability of a request to maintain its performance under increasing load. For example, the performance of a request may be reported as generating a valid response within three seconds, but the scalability of the request measures the request's ability to maintain that three-second response time as the user load increases.

Scalability asks the following questions about the request:

- At the expected usage, does the request still respond within three seconds?
- For what percentage of requests does it respond in less than three seconds?
- What is the response time distribution for requests that do not respond within three seconds?

If you recall from Chapter 5, an SLA is defined by the following three key criteria:

- It is specific.
- It is flexible.
- It is reasonable.

The "specific" value measures the performance of a single request: the request must respond within three seconds. The "flexibility" value, however, measures the scalability of the request: the request must respond within three seconds for 95 percent of requests and may fluctuate, at most, one standard deviation from the mean.

The strategy is to first ensure the performance of a request or of a component and then test the request or component for scalability. Ensuring the performance of a request or of a component depends on where your application is in the development life cycle. Optimally, you want to implement proactive performance testing in the development phase of your application, which includes developing unit tests using a unit testing framework, like JUnit, and implementing code profiling, memory profiling, and coverage profiling against those unit tests.

From code profiling, you want to watch for the following three key things:

- Poorly performing methods
- Methods invoked a significant number of times
- Classes and methods that allocate an excessive number of objects

The purpose of code profiling is to identify any egregiously slow algorithms or methods that are creating a surplus of objects; for example, trying to sort 1 million items using a bubble sort algorithm can result in up to 10^{12} object comparisons, which could take minutes or hours to execute.

When implementing memory profiling, you look for the following two things:

- Loitering objects
- Object cycling

Loitering objects, also referred to as *lingering object references*, are unwanted objects that stay in the heap after the end of a use case. They reduce the amount of available heap memory and typically are tied to one or more requests, so they are leading the heap down a path to its ultimate demise. Another side of memory mismanagement is *object cycling*, or the rapid creation and destruction of objects in the heap. While these objects do not linger in the heap and consume permanent resources, they force the garbage collector to run more frequently and hence hinder performance.

Finally, *coverage profiling* establishes the level of confidence you have in your unit tests. The coverage profiler tells you each condition that was and was not executed for every line of code. If you test the functionality and performance of your code in unit tests, and your coverage is high, then you can feel comfortable that your code is solid. On the other hand, if your coverage is low, then you can have very little confidence in your test results.

Chapter 5 details how to use each of these performance profiling tools and how to interpret the results. If you jumped right to this chapter, I suggest you review the material in Chapter 5 before investigating performance problems.

Thus far we have been looking at the bottom-up approach to performance tuning from an application's inception. This approach is great in theory, and you can definitely apply it in the *next* project, but what do you do today? Luckily, we can apply similar principles with a top-down approach. I hope that you have a production staging environment that you can test against, but if not, you can capture performance data from a running production environment.

■ **Note** Configuring a diagnostic tool to run in a production environment can be a tricky task, but depending on the tool itself, its impact on the environment can be mitigated. The core factors that can help you mitigate the impact of such a tool in production are configuring filters, increasing aggregate sampling intervals, and bundling components. Filters allow you to capture only the requests you are interested in (in some cases, you may need to record several iterations of data with different filters to capture all interesting data). Call traces are aggregated at a specific interval, and increasing this interval reduces the workload on the data correlator (for example, aggregate data every 1 or 2 minutes, rather than every 10 or 30 seconds). To isolate poorly performing components, sometimes bundling related code packages together into individual components and returning a single value, rather than reporting method-by-method data, can provide valuable information at lesser overhead. You will need to spend significant time with your performance diagnostic tools and tool vendors to determine the best approach to recording detailed information in a live production environment.

The process is to record detailed, method-level performance information against your environment while it is subjected to load. From this data, you want to extract the following information:

- What are the slowest running requests?
- What are the most frequently executed requests?
- From these requests, what are the hot paths through the requests?
- What are the hot points in those hot paths?
- Are the performance problems code issues, configuration issues, or external dependency issues?

Examine each slow request and determine why it is slow: is it a slow method, a slow database call, or an environment threading or memory issue? If you can identify the problem as being code related, then examine the offending methods. If the performance issues still evade you, then examine the offending methods inside a code profiler by replaying the problematic requests against your profiled environment and examining the offending methods line by line. Continue this process until all major code bottlenecks have been resolved.

To fine-tune your application, identify requests and/or methods that are executed frequently. Try to find methods that you have a chance of tuning; for example, your application may call `StringBuffer.append()` millions of times, but you cannot improve its performance. (Actually, in this case, you want to find out why the method is being called so many times and substantially reduce that if possible.) Shaving a few milliseconds off of a frequently executed method can yield dramatic improvements in application performance.

All performance issues not related to your code need to be investigated on their own; for example, database issues need to be evaluated in a database diagnostic tool. Your Java EE information should provide you with problematic contexts such as SQL statements and the call path that generated the undesirable behavior.

The goal is to resolve as many performance bottlenecks as possible prior to performing a capacity assessment. While a capacity assessment can help you tune your environment, tuning is only the secondary goal—the primary goal is to determine the maximum load you can support while meeting your SLAs and establish a degradation model for your application once SLAs are violated.

Capacity Assessment

The purpose of a capacity assessment is to identify the following key data points:

- The response time of your requests at expected usage
- The usage when the first request exceeds its SLA
- The usage when the application reaches its saturation point
- The degradation model from the first missed SLA to the application's saturation point

A capacity assessment should be performed against your production staging environment, but only after you have verified that your application can sustain expected usage through a performance load test. The capacity assessment is the last test performed at the end of an iteration or prior to moving the application to production. Your production staging environment should mirror your production environment, if possible, or be a scaled-down version of it; otherwise, the results of the capacity assessment are of little value.

Graduated Load Tester

The key tool that empowers you to perform a capacity assessment is a *graduated load tester.* A graduated load tester is configured to climb to your expected usage in a regular and predefined pattern and then increase load in graduated steps. The purpose behind this configuration is to allow you to capture and analyze performance metrics at discrete units of load. The behavior of graduated load generation is illustrated in Figure 9-1.

Graduated step sizes need to be thoughtfully chosen: the finer their granularity, the better your assessment will be, but the longer the test will take to run. You can usually find a good compromise between the test run time, the analysis time, and the granularity of the steps; it varies from application to application, but as a general rule, I configure these steps to be about 5 percent of the expected load. For example, if you need to support 500 simultaneous users, then your graduated step might start at 25 users. Graduated steps are required because of the nature of enterprise applications, as Figure 9-2 illustrates.

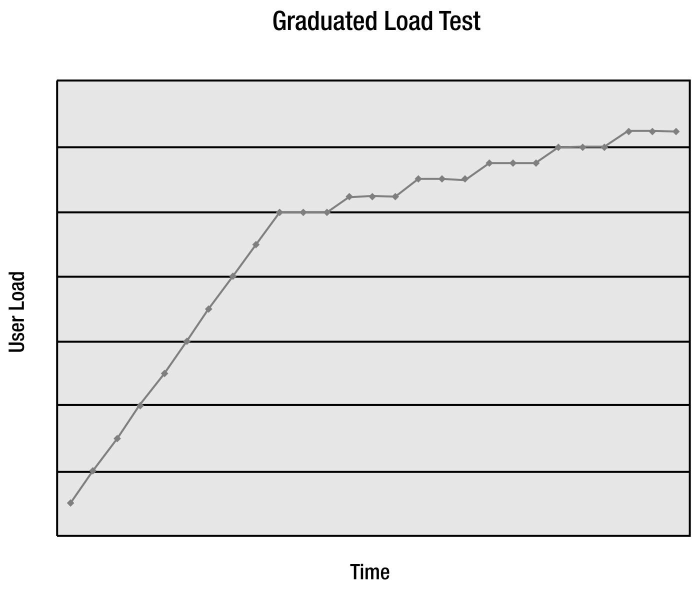

Figure 9-1. *A graduated load test*

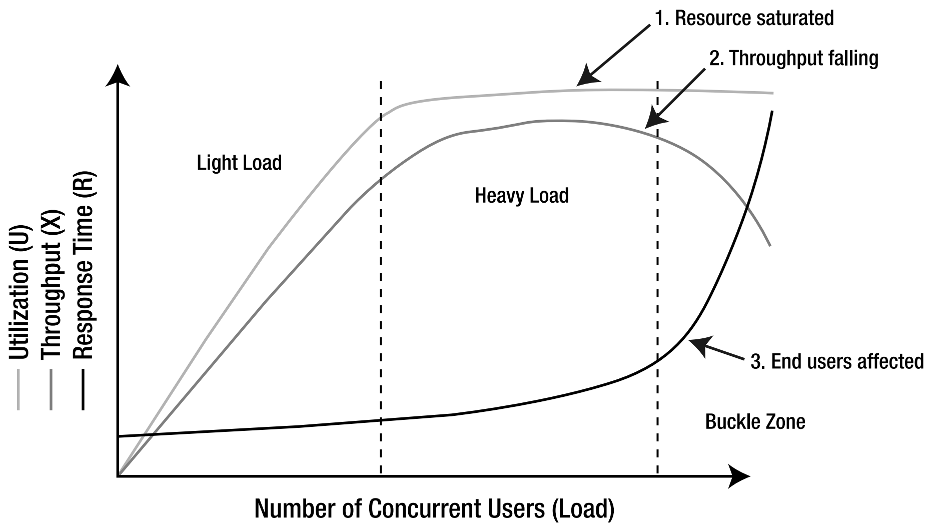

Figure 9-2. *A loaded enterprise application follows this typical pattern.*

As the number of users (load) increases, the system utilization naturally increases, because you need more resources to do more work. Likewise the throughput increases, because as you are sending more work to the application, it is performing more work. When the load becomes too heavy for the environment to support, then resources become saturated, which manifests in excessive CPU context switching, garbage collections, disk I/O, network activity, and so on.

These manifestations result in a decline in request throughput, which means that requests are left pending and response time increases. If the load continues to increase when the system is in this state, then the response time performance degrades exponentially. The point at which performance time degrades is referred to as the *saturation point*, or, more colorfully, the *buckle zone*.

Capacity Assessment Usage Mappings

The capacity assessment attempts to map the behavior of your environment to the graph shown in Figure 9-2. Therefore, at the end of the capacity assessment, you should be able to construct a graph similar to Figure 9-3.

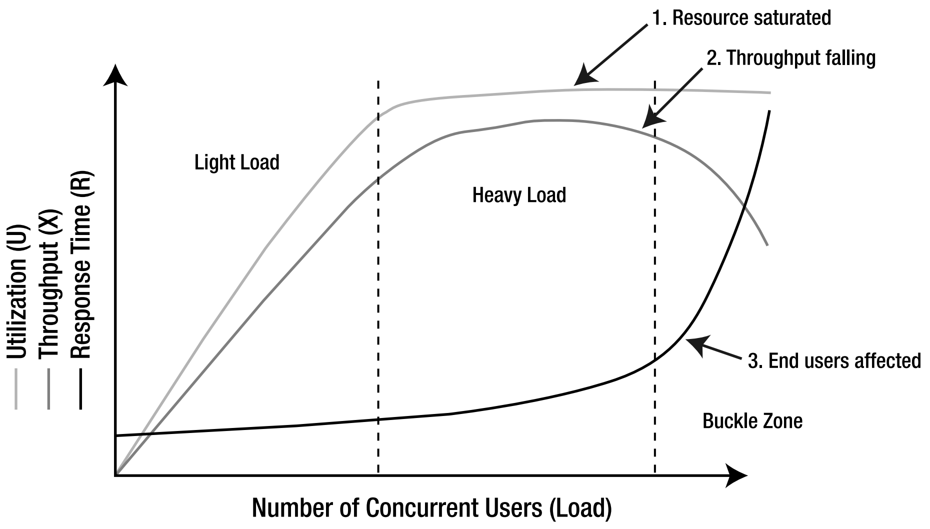

Figure 9-3. *In this test, we successfully satisfied our expected users, and we have a buffer before we start exceeding SLAs.*

Figure 9-3 illustrates that the application is successfully satisfying its 500 expected users, but at 550 users the application starts exceeding its SLAs, showing that we have a 10 percent user buffer that can be supported. The buffer percentage represents your comfort zone for your application—if your load never moves outside of your buffer, even during peaks, then you can sleep well at night. You should strive to maintain approximately a 25 percent buffer to allow for marketing promotions or any other significant increase in traffic. A buffer greater than 40 percent is typically too large, because it means that you are maintaining additional hardware and software licenses that really are not needed.

■ **Note** Although a 25–40 percent buffer is generally ideal, you need to analyze your usage patterns to determine what is best for your environment. If your usage patterns are very volatile, then you might want a larger buffer, but if you are running an intranet application with a fixed maximum load, a smaller buffer may be sufficient.

The environment's resources start to become saturated at 600 users, and by 650 users the request throughput degrades substantially. User requests begin backing up in thread pools and finally at the socket level, and by 700 users, the application has entered the buckle zone. Once the application is in the buckle zone, the only option is to stop accepting requests and allow the application to process all pending requests. But in reality, at this point, the only feasible answer is to begin restarting application servers to manually flush out user requests.

The user response time, resource utilizations, and the request throughput between the expected usage point and the buckle zone can be used to construct a degradation model. The degradation model explicitly identifies the support buffer and response time patterns. The purpose of constructing a degradation model is to allow your CIO to determine when to acquire additional hardware and software resources. For example, if usage patterns are increasing month-over-month in an identified trend, and a degradation model identifies that within 12 weeks end-user SLAs will be violated, then a strong case is presented for acquiring additional resources.

Measurements

Before configuring your graduated load tester and firing load at your environment, you need to put the tools in place to gather the appropriate measurements to assess the various states of your environment. In the previous section, we identified three categories of metrics:

- Resource utilization
- Throughput
- Response time

The executive summary of your capacity assessment may state simply that at 600 users resources became saturated, but the detailed resource analysis is going to report far more than a simple saturation point metric. We look at a variety of resources in a capacity assessment and identifying the limiting resources is important. For example, if the application is CPU-bound, meaning that the core resource that becomes saturated and brings down the application is the CPU, then adding additional RAM may not be very helpful. You need to configure each relevant resource with monitoring tools and record its behavior during the capacity assessment.

Throughput, the second category of metrics, is defined simply as work performed over a period of time. In a transactional application, requests committed per second is a common measure of throughput, and in some environments, I have used the load tester's recording of successful requests processed per second as a measure of throughput. Your choice of measurement is not as important as the fact that throughput is recorded consistently.

Finally, response time can be measured in terms of single requests, business transactions that are a composite of requests, or a combination of the two. The most effective measure of response time that I have observed has been a high-level recording of business transactions with the ability to drill deeper into the individual requests that comprise the business transaction. But in the end, you are measuring your response times against your use case SLAs, so be sure that your measurements are consistent.

Resource Utilization

The most common resource measured is CPU utilization, because it always increases in direct proportion to the user load: more requests made against the system require more CPU resources to process them. But you need to capture other important metrics, namely the following:

- Physical memory utilization
- Operating system disk I/O rates
- Operating system thread/process utilization
- Application server thread pool utilization
- Application server connection pool utilization
- Application server heap utilization and garbage collection rates (frequency and duration)
- Application server cache and pool utilizations
- Messaging system utilizations
- External dependency profiles (databases, legacy systems, Web services, and so forth)
- Network traffic sent between application nodes

The analysis of CPU utilization is pretty straightforward: under 80 percent utilization means that the CPU is not under serious duress, but as it increases from 80–95 percent, the entire system begins to suffer. If the CPU becomes pinned at 95–100 percent utilization, then it is going to have difficulties processing anything in a timely manner.

The key to physical memory analysis is identifying anything that is being swapped out to disk. Virtual memory and swapping are manageable on your desktop client, but you know that as soon as your operating system's memory requirements exceed the amount of physical memory that you have, your computer becomes less and less responsive. The same thing happens on a server, only the manifestations of the conditions are much more severe for your end users. Throughout the capacity assessment, you need to record physical and virtual memory usage as well as paging rates: high physical and virtual memory usages combined with increased paging rates signify physical memory saturation.

Disk I/O rates represent how much data is being read from and written to the hard disk. Disk reads and writes are expensive relative to reading and writing to physical memory, and typically, high disk I/O rates mean that caching has not been configured optimally. For example, when you make a request from a database, its data resides on the file system, but it maintains frequently accessed data in memory caches. If the database is properly tuned, it should be serving the majority of its requests from a cache, rather than having to read from the file system to satisfy every request.

Different operating systems define their threading strategies differently, but regardless of the implementation, the operating system controls its applications' access to the CPU. The operating system's ability to maximize the use of its CPUs needs to be analyzed: if the threading configuration prohibits the maximum use of its CPUs, then the environment can be thread or process bound. For example, in older Linux threading models, each thread was represented by a new process, so the Linux maximum process configuration limited the maximum number of threads an application server could use. On the other hand, Solaris allows a single process to maintain its own internal threads, but only through the configuration of multiple processes can the operating system's CPUs be fully utilized. Tracking application server thread utilization is not sufficient for a capacity assessment; you need to track the operating system's threads as well.

When a request is received by an application server, it is placed into an execution queue that is processed by a thread from the designated application server's thread pool. If the application server is out of threads, then the request will sit waiting in the execution queue for a thread to process it. If the application server thread utilization approaches 100 percent and pending requests are waiting in the execution queue, then the environment is bound by the number of application server threads. Of course, this value must be evaluated in the context of the entire system. For example, if the application server thread pool utilization is at 100 percent with pending requests, and the operating system CPU is at 40 percent, then the thread pool size should be increased. But if the application server thread pool is at 100 percent, and the CPU became pinned at 100 percent 30 seconds earlier, then the CPU bottleneck caused the request backup. All of these components are tightly integrated, and your job is to uncover the root cause of bottlenecks.

Most applications access a database or some other external dependency, such as a legacy system. Rather than establishing a new connection to the external dependency each time that the application needs to interact with it, the preferred implementation is to create a pool of connections to it (either a JDBC or JCA connection pool). If your application server is processing a request that depends on a connection from a connection pool, and no connection is available, then the processing thread must wait for a connection to become available. If you recall Chapter 6, where we explored the concept of wait-based tuning, application server wait points represent application server metrics that you need to monitor and analyze during a capacity assessment—the same principles established in Chapter 6 apply here as well.

The application server heap can significantly degrade performance as load increases if it is not configured properly. While performing the capacity assessment, you want to observe the heap utilization and the garbage collection rate. As load increases, the rate of temporary object creation and destruction increases, which adds additional burdens on the garbage collector. If these objects are created and destroyed faster than the garbage collector can reclaim them, then the garbage collector will have to freeze all threads in the JVM while it performs a full mark and sweep (and optionally compact) garbage collection. During this time, nothing is processing, and all of your SLAs may be compromised.

Session storage requirements also can affect heap performance under load. If each session requires 1MB of memory, then each additional user will consume an additional 1MB of memory—700 users require 700MB of memory. These storage requirements are a strong reason to keep sessions light, but depending on your business requirements, doing so may not be an option. Therefore, you need to monitor the heap growth throughout the capacity assessment and potentially increase the heap size to meet the application requirements.

Application server caches and pools can become wait points in your application call stacks, because as requests obtain objects from caches and pools, they may have to wait for the objects to be returned. The wait time can be mitigated by proper sizing and management algorithms (such as using a least-recently-used caching algorithm), and your capacity assessment will identify if these caches or pools are causing your waits. Look for cache thrashing (high passivation rates) and threads waiting on pools to detect this condition.

Many applications make significant use of messaging, either for asynchronous processing or as a communication mechanism between Java EE and legacy systems (IBM shops tend to do this a lot). You need to ensure that messages are passing through the messaging server quickly and efficiently without being rejected. The configuration parameters for tuning a message server will differ from vendor to vendor, but some common things to look at are message and/or byte utilizations and message throughput. If the messaging server resides on its own physical machine, then you need to watch all operating system statistics and internal messaging thread pools.

The same operating system– and domain-specific metrics need to be observed for all external systems that your application interacts with, such as databases, legacy systems, and internal Web service providers. Consider that each technology your application interacts with is built on top of a technology stack that has the potential to cripple the capacity of your application. Your goal is to identify the potential wait points in those various technology stacks, capture relevant metrics during a capacity assessment, and analyze their behavior as it relates to your application.

Finally, one of the most important metrics that can yield significant insight into the capacity of your environment is network traffic, including communication and latency. Every remote call that you make has the potential to disrupt the performance of your application, and the impact increases in proportion to your load. And depending on the number of network hops that a request makes, the impact may not compare one to one with the user load. For example, if the majority of requests pass from a load balancer, to a Web server, to an application server Web tier, to an application server business tier, and finally to a database, then a single request to the Web server resolves to three additional network hops. Therefore 500 simultaneous requests resolve to 1,500 network calls, which can dramatically affect the performance of your environment.

In order to perform a valid capacity assessment, you need to capture metrics on each machine in your environment related to the application servers, operating system, external dependencies, messaging, and network communications, as Figure 9-4 illustrates.

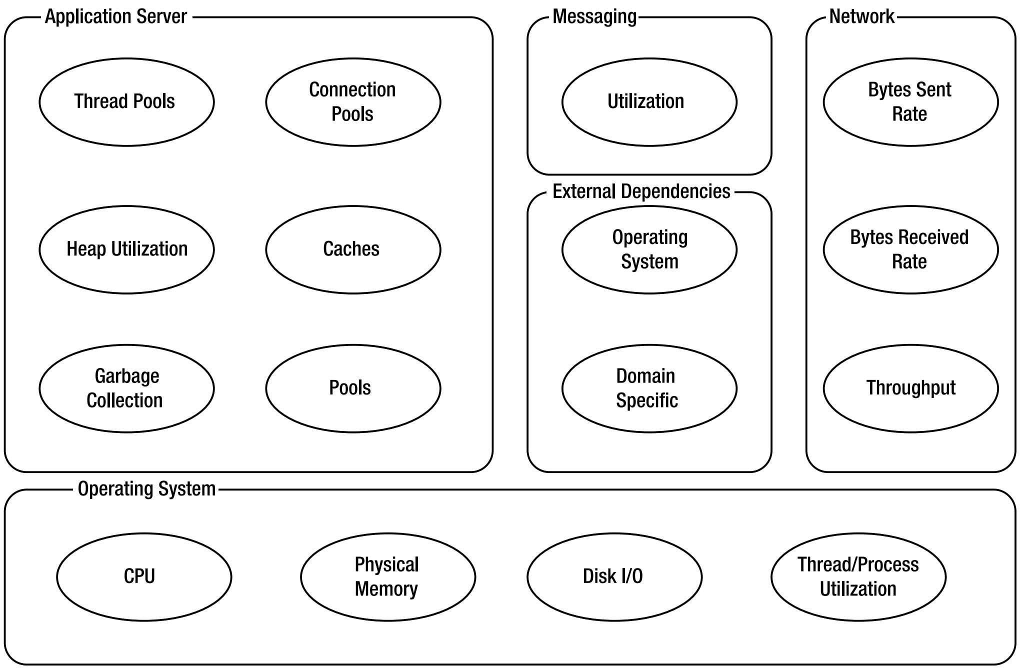

Figure 9-4. *Some of the key metrics that you need to capture on each machine during the capacity assessment*

Building a Capacity Assessment Report

The capacity assessment report serves the following two purposes:

- The communication mechanism to express the capacity of an environment
- A historical marker to assess the impact of application changes

First and foremost, the capacity assessment identifies the capacity of a given environment. It must clearly state the performance of the environment under expected or projected usage and the maximum capacity of the environment while maintaining SLAs. For example, it might summarize performance information by stating that at the expected 500 users, the average response time is 20 percent below defined SLAs, and the environment can support a maximum of 550 users before violating SLAs. Furthermore, the capacity analysis provides models from which hardware and software forecasts can be constructed.

The secondary purpose of the capacity assessment is to establish historical performance markers against which future application releases can be compared. Quantifying the impact of software enhancements against a known baseline is important, so that as your applications evolve, you know what features helped and hindered performance. At the end of each capacity assessment, you should compare the results to previous capacity assessments and add them to your Capacity Assessment Report.

The Capacity Assessment Report is composed of the following components:

- Executive summary
- Test profile
- Capacity analysis
- Degradation model
- Impact analysis
- Final analysis and recommendations

This section reviews each component of a Capacity Assessment Report and defines the articles that should be included in each.

Executive Summary

The executive summary is the overview of the capacity assessment, suitable to be presented to an IT manager or CIO. The executive summary should convey the pertinent results with minor substantiation, but not with the same level of detail the capacity analysis will provide. In general, your CIO will be very interested in the results themselves, but much less interested in many of the details that informed your conclusions. And those details are examined elsewhere in the report for review by any interested party.

Specifically, the executive summary needs to include the following information:

- *Performance of the environment at the expected usage*: This should be presented by stating the expected usage along with the average difference, minimum difference, and maximum difference (as percentages) between the use case SLAs and observed response times at the expected load.
- *Load at the time that the first use case SLA was violated*: Note that use cases are defined in terms of a specific value, a measure of flexibility, and a hard upper limit, which is usually defined in terms of standard deviations from the mean. This definition means that your results do not report that the SLA has been violated on a single incidence but rather when the violation exceeds the level of flexibility.
- *The resource saturation point, including the resources being saturated*: Rather than reporting that resources are saturated at 600 users, reporting that at 600 users the environment became CPU-bound as the Web tier CPU became saturated is more valuable.
- *The end user experience fail point*: This point is where the request response time begins increasing at an exponential rate, and it highlights the point of no return that can only be resolved by an application server restart.
- *A graph*: The graph, similar to Figure 9-3, visually summarizes the preceding four bullet points.
- *An overview of the impact analysis results*: The overview primarily focuses on capacity differences, either as improvements or degradations of supported load.

As with the executive summary of any report, the emphasis should be on the results of the analysis with minimal supporting evidence; the body of the document provides all necessary supporting evidence. The goal is for you to quickly convey your results to the reader.

Test Profile

The test profile describes the load test scripts and load tester configurations used to perform the capacity assessment. This description is valuable, because it establishes the context for which the capacity assessment is valid. If product management provides you with a collection of usage profiles and asks you to first tune the environment and then perform a capacity assessment against the environment, your ability to successfully perform these tasks is only as valid as the usage profiles. By adding a detailed test profile, you gain the benefit of a peer review process—the production management team, as well as your peers, can analyze the nature of your capacity assessment and provide feedback as to why the test may or may not necessarily yield valid results.

Specifically, the test profile should answer the following questions:

- What use cases were executed?
- What was the balance between use cases?

- What was the frequency of each use case?
- What was the specific configuration for each use case, such as think-time settings, image downloads, and so on?
- What was the load testing profile, such as the ramp-up period and pattern, graduated step sizes, load duration, and so forth?
- How were the physical and logical servers configured in the test environment?
- What was the monitoring profile? For example, what was monitored and at what level?

The test profile serves both as a historical marker for the test context and as a point for later validation against actual production usage. After the application is running in production, validating user patterns against the test profile and the observed behavior against expected behavior makes a very good exercise. This exercise can help you refine your future estimates.

Capacity Analysis

The capacity analysis presents your findings, with all of the details to support your conclusions. This section supplies all supporting evidence in graphs and detailed textual analysis of the graphs. It should begin by presenting the same graph displayed in the executive summary and providing a more detailed overview of your conclusions. The sections following the initial graph detail the behavior of the environment at each critical point in the graph, including sections for the environment behavior at the following times:

- Expected load
- The SLA violation point
- The saturation point
- The buckle zone

Each section should include a table presenting a summary of the behavior of each use case at the expected load. A sample is presented in Table 9-1. Your assessment might introduce the table by explaining, "At the expected load of 500 users, the following behavior was observed."

Table 9-1. *Sample Use Case Summary*

Use Case	SLA Ave	SLA Dist	SLA Max	Actual Ave	Actual Dist	Actual Max	Actual SD	Actual 2xSD	Delta Resp Time Buffer
Login	4 sec.	95%	6 sec.	3.2 sec.	97%	4.7 sec.	1.2 sec.	2.0 sec.	20%
Search	3 sec.	95%	5 sec.	2.6 sec.	96%	4.0 sec.	0.6 sec.	1.0 sec.	13.3%
Input Claim	7 sec.	95%	10 sec.	5.5 sec.	98%	9 sec.	2.0 sec.	3.0 sec.	21.4%
Summary									18.2%

The Use Case Summary columns are defined in Table 9-2.

Table 9-2. *Use Case Summary Column Definitions*

Column	Description
Use Case	The use case name or number being presented.
SLA Ave	The SLA's "specific" value, or the average maximum value for the defined distribution.
SLA Dist	The SLA's "flexibility" value, or the percentage of requests that must fall below the average in order for the use case to uphold its SLA.
SLA Max	The maximum value permissible for any request, the hard limit (or relative limit if working in standard deviations) that if exceeded immediately causes an SLA violation.
Actual Ave	The average observed response time for the use case.
Actual Dist	The observed percentage of requests below the average SLA value.
Actual Max	The maximum observed response time for the use case.
Actual SD	The standard deviation of observed response times.
Actual 2xSD	Two standard deviations of the observed response times.
Delta Resp Time Buffer	The response time buffer percentage. This is a measure of the buffer that the use case has between the observed average response time and the SLA average response time. It roughly identifies the amount that the use case can grow before it is in danger of violating its SLA.

In addition to providing information about the use cases, these sections should also present summary information about pertinent resources. From a Java EE perspective, this information is going to include CPU utilization, heap utilization, garbage collection rates, thread pool utilization, pending requests, connection pools, caches, and request throughput.

These four sections present a snapshot of the state of use case response times, resource utilization, and throughput. The conclusion of each section should include an analysis of the raw data, including articles required to substantiate your conclusions. For example, you can include charts and graphs, numerical analysis of the presented data, historical capacity assessment data, and so on.

Degradation Model

While the capacity analysis sections provide detailed information with snapshots captured at specific points in the assessment, the degradation model reports the entire assessment in a timeline. It identifies trends in response time and resource utilization data throughout the assessment, but its primary focus is on the segment between the expected load and the buckle zone.

The degradation model contains a considerable number of graphs, illustrating the following information:

- Use case response times
- Utilization of each relevant resource
- Application throughput

Each of these graphs should be overlaid with the following identified performance zones:

- Expected usage to SLA violation point
- SLA violation point to resource saturation point
- Resource saturation point to buckle zone

The purpose of this section is to identify not only the behavior of use cases at various user loads, but also why performance issues arise. For example, if the Login use case degrades at 550 users and exceeds its SLA, is it because of an external dependency, the application server CPU utilization, a database connection pool, a database call, or an application server thread pool? Domain knowledge of your environment and your applications empowers you to be able to correlate metrics and derive accurate conclusions in this section of the Capacity Assessment Report. When I am on-site with customers, I spend a considerable amount of time interviewing them to learn the following:

- What technologies are they using (for example, servlets, JSP, stateless session beans, entity beans, JMS)?
- What design patterns have they employed and where?
- What does a whiteboard sketch of the path of a typical request through the application look like?
- What objects are cached, and what are those objects used for?
- What objects are pooled?
- How is the environment configured (for example, thread pools, the heap, and connection pools)?
- What is their network topology?
- What external systems are their applications interacting with and through what communication mechanisms?

Through this interview process I "cheat": I anticipate where performance problems might occur, so that when I analyze the customer's environment and see them occur in the capacity assessment, I have a strong idea about what metrics to check for relevant correlations. Without this information, constructing an accurate degradation model is difficult at best.

Impact Analysis

Once a capacity assessment has been performed against a Java EE environment, it should be saved for future comparisons; these performance comparisons are explored in the impact analysis. The impact analysis identifies the differences between two or more capacity assessments with the primary intent of quantifying the impact of code changes against system capacity.

If your organization is mature enough to routinely perform capacity assessments throughout the development of an application, then the impact analysis can be mostly automated, because it tracks performance differences between response times and resource utilizations for the same use cases and very similar, if not identical, test scripts. But if you are like most companies for whom performing a formal capacity assessment on each significant iteration is not feasible and who reserve capacity assessments for released code, then the task is a little more daunting and requires deep, domain-specific analysis. In this case, the summary of the impact analysis should be performed using response time buffer percentages. Recall that the response time buffers measure the percentage difference between the observed performance at a specific user load and the SLA. With this measurement, you can assess the performance of application functionality at specific user loads and determine whether a particular functional element degraded or improved in a subsequent release; you measure the degradation or improvement against the SLA defined for that functionality. If the SLA is renegotiated as a result of new or changed functionality, then an altered response time will not skew the impact analysis.

The purpose of the impact analysis is to identify the following:

- General capacity impact of code changes, including the performance at the expected load, the SLA violation point, the resource saturation point, and the buckle zones
- Specific degradations and improvements of use cases
- Specific degradations and improvements in resource utilizations

The sample Capacity Assessment Report later in this chapter provides additional details about the impact analysis.

Analysis and Recommendations

The analysis and recommendations section provides a conclusion to the Capacity Assessment Report. As such, it summarizes the findings again, but includes information about the impact of the findings on the business process and provides recommendations. It attempts to answer the following questions:

- What is the performance at the current or expected load?
- What load can the environment support and still satisfy SLAs?
- At what point does the environment need to be upgraded?
- What is the nature of that upgrade? Should it add more application server instances, or modify application server configurations (heap size, thread pools, connection pools, and so on)?
- At what point does the environment require additional hardware?

If you have any insight into seasonal patterns, marketing promotions, or any other trending information that will affect user load, this information should be summarized or referenced here to justify your recommendations with forecasted behavior.

Sample Capacity Assessment Report

Excerpts from the various sections of a Capacity Assessment Report follow. An actual Capacity Assessment Report may be 20 to 50 pages or more in length, so this sample attempts to reproduce each major section and include at least one major item in each section. You can fill in the remaining components with performance observations relevant to your environment.

Executive Summary

In this capacity assessment, the Acme Buy High, Sell Low stock application was evaluated in a mirrored production environment for performance. The expected user load for this application is 500 users, and the observations extracted from the test are illustrated in Figure 9-5.

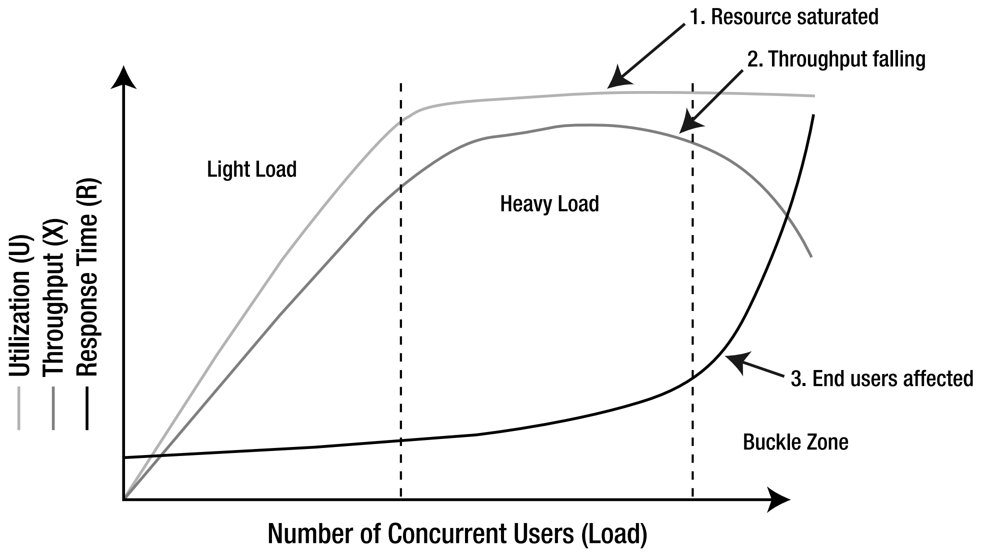

Figure 9-5. *The Buy High, Sell Low environment's behavior as load increases*

Figure 9-5 can be summarized by the following observations:

- At the expected user load of 500, all use cases satisfy their SLAs.
- The first SLA violation is observed at 550 users.
- The environment's saturation point occurs at 600 users.
- The environment enters the buckle zone at 700 users.

Use cases currently maintain an average response time buffer of 19.24 percent and based upon current trend analysis, this will dissipate rapidly over the next three months. My estimates suggest that the current environment will be in violation of its SLAs within five months. The test results indicate that the environment is CPU-bound and requires additional application server hardware to mitigate the five-month degradation point.

The performance of the Buy High, Sell Low application has degraded with the release of version 2.0. The average response time degradation is 12 percent, and the average resource utilization at expected load has increased by 7 percent. Throughput at expected usage has likewise degraded by 10 percent. The maximum capacity has decreased from 650 users to 550 users, a degradation of 15.3 percent.

I recommend additional hardware resources for addition to the environment while the source code is examined to identify the root of the performance degradation.

Test Profile

The capacity assessment was implemented using in-house load testing technology exercising the use cases shown in Table 9-3.

Table 9-3. *Test Profile*

Use Case	Distribution Weight
Login	0.1
Add Stock	0.1
Historical Query	0.2
Historical Graphing	0.2
Stock Discovery	0.2
Profile Management	0.2

The load test was configured to ramp up linearly over 30 minutes to the expected user load of 500 users. The test then implemented a graduated step sized at 25 users to ramp up over 5 minutes and hold for 5 minutes before initiating the next step.

Test Script Configurations

The following section summarizes the test script configurations. It includes detailed information about the primary use case scenario and summarizes the scenario distributions.

Login

The primary scenario for this use case is the successful login of a user with a valid username and a valid password. The steps for this scenario are summarized as follows:

Request	Think Time	SLA Ave	SLA Flexibility	SLA Maximum
`GET /stock/index.html`	10 sec.	3 sec.	95%	5 sec.
`POST /stock/login.do`	End	5 sec.	95%	8 sec.

with the following scenario distribution:

Scenario	Distribution
Successful Login	94.5%
Valid username, invalid password	5%
Invalid username	0.5%

Test Platform Topology

The test platform consisted of six physical machines:

- Two Web servers
- Two application servers
- Two database servers

Two application server instances run on each physical application server, totaling four application server instances. Figure 9-6 illustrates this topology.

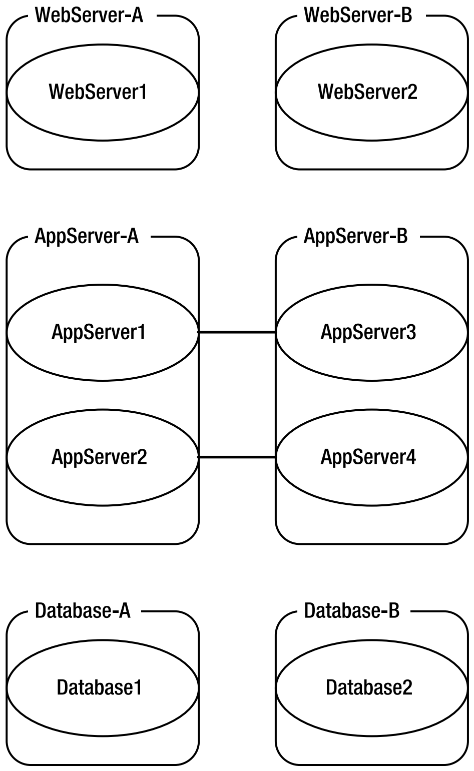

Figure 9-6. *The test environment topology*

The test environment includes clustering with AppServer1 using AppServer3 as its secondary server, AppServer2 using AppServer4 as its secondary server, and vice versa. In this way, the environment is resilient not only to application server instance failure, but also to hardware failure. This configuration adds additional performance overhead, but it meets the predefined availability and failover requirements.

Monitoring Configuration

The monitoring employed during this capacity assessment was configured to poll operating system, database, Web server, and application server statistics after every minute. The load tester was responsible for recording the overall request response times, while light bytecode instrumentation was employed to report tier-level response times. The bytecode instrumentation was not configured to record method-level statistics.

Capacity Analysis

This section presents the observations and conclusions derived in this capacity assessment.

Expected Usage

The expected usage for the Buy High, Sell Low application is 500 users. Table 9-4 reports a summary of the use case behavior at the expected load.

Table 9-4. *Use Case Summary at Expected Usage*

Use Case	SLA Ave	SLA Dist	SLA Max	Actual Ave	Actual Dist	Actual Max	Actual SD	Actual 2xSD	Delta Resp Time Buffer
Login	8 sec.	95%	13 sec.	6.2 sec.	97%	9.7 sec.	1.2 sec.	2.0 sec.	22.5%
Add Stock	10 sec.	95%	14 sec.	8.5 sec.	97%	12 sec.	2.0 sec.	3.2 sec.	15%
Hist Query	10 sec.	95%	14 sec.	8.2 sec.	96%	12 sec.	2.2 sec.	3.5 sec.	18%
Hist Graph	12 sec.	95%	16 sec.	10.1 sec.	98%	14.2 sec.	1.8 sec.	2.2 sec.	15.8%
Stock Disc	8 sec.	95%	12 sec.	6 sec.	99%	8.2 sec.	1.5 sec.	2.0 sec.	25%
Profile Mgmt	12 sec.	95%	16 sec.	9.7 sec.	95.5%	15 sec.	2.5 sec.	5 sec.	19.16%
Summary									19.24%

The SLA Violation Point

The SLA violation point for the Buy High, Sell Low application occurred at 550 users. Table 9-5 reports a summary of the use case behavior at the SLA violation point.

Table 9-5. *Use Case Summary at the SLA Violation Point*

Use Case	SLA Ave	SLA Dist	SLA Max	Actual Ave	Actual Dist	Actual Max	Actual SD	Actual 2xSD	Delta Resp Time Buffer
Login	8 sec.	95%	13 sec.	7.8 sec.	92%	12 sec.	2.2 sec.	4.0 sec.	Violation
Add Stock	10 sec.	95%	14 sec.	9.5 sec.	95%	13.7 sec.	2.0 sec.	3.9 sec.	5%
Hist Query	10 sec.	95%	14 sec.	8.9 sec.	96%	13 sec.	2.9 sec.	3.9 sec.	11%
Hist Graph	12 sec.	95%	16 sec.	11.1 sec.	91%	15.8 sec.	2.8 sec.	3.2 sec.	Violation
Stock Disc	8 sec.	95%	12 sec.	7.4 sec.	95%	10 sec.	2.5 sec.	3.0 sec.	7.5%
Profile Mgmt	12 sec.	95%	16 sec.	10.2 sec.	94%	15.7 sec.	2.5 sec.	5 sec.	Violation
Summary									50% Violation

The Saturation Point

The saturation point for the Buy High, Sell Low application occurred at 600 users. Table 9-6 reports a summary of the use case behavior at the saturation point.

Table 9-6. *Use Case Summary at the Saturation Point*

Use Case	SLA Ave	SLA Dist	SLA Max	Actual Ave	Actual Dist	Actual Max	Actual SD	Actual 2xSD	Delta Resp Time Buffer
Login	8 sec.	95%	13 sec.	12.8 sec.	22%	37 sec.	6.2 sec.	12.0 sec.	Violation
Add Stock	10 sec.	95%	14 sec.	19.5 sec.	14%	32.7 sec.	8.0 sec.	12.9 sec.	Violation
Hist Query	10 sec.	95%	14 sec.	18.9 sec.	34%	23 sec.	3.9 sec.	4.9 sec.	Violation
Hist Graph	12 sec.	95%	16 sec.	17.1 sec.	33%	25.8 sec.	4.8 sec.	3.2 sec.	Violation
Stock Disc	8 sec.	95%	12 sec.	15.4 sec.	27%	18 sec.	6.5 sec.	3.0 sec.	Violation
Profile Mgmt	12 sec.	95%	16 sec.	20.2 sec.	42%	27.7 sec.	5.5 sec.	5 sec.	Violation
Summary									100% Violation

The Buckle Zone

The buckle zone for the Buy High, Sell Low application occurred at 700 users. At this point, all use cases exceeded their SLA average values for greater than 80 percent of requests.

Degradation Model

The aggregate use case response time degradation model is shown in Figure 9-7. The aggregate response time degradation model plots the average response time buffer percentage against the user load.

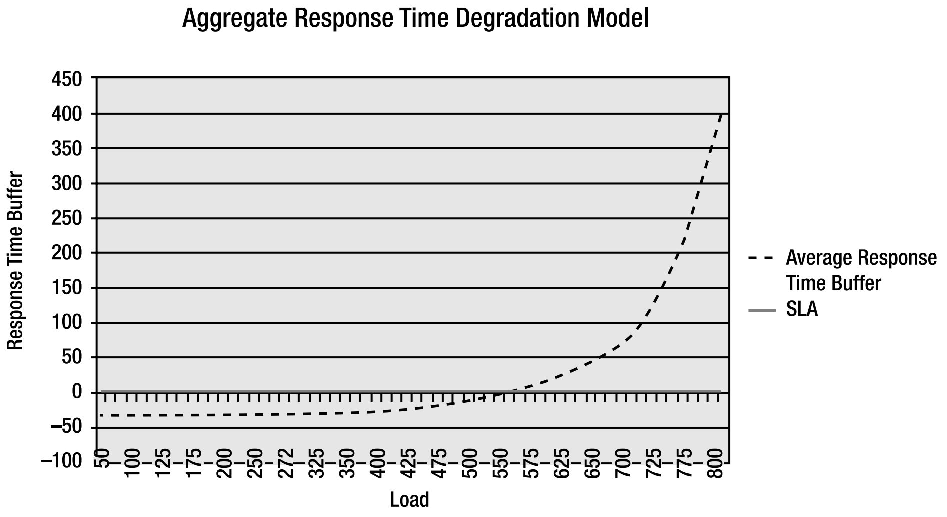

Figure 9-7. *The aggregate response time degradation model plots the average response time buffer against user load. In this case, the average response time buffer percentage hits zero at a user load of 550 users.*

The response time buffer follows nearly an exponential pattern and crosses SLA boundaries at 550 users, so once the SLA is violated, the system can only sustain 100 to 125 users until the application is deemed completely unusable by the users. “Unusable” is defined as response times that exceed their buffer by more than 50 percent.

The environment is primarily bound by CPU utilization in the application server tier. Figure 9-8 displays an aggregate of all CPUs present in the application server tier, and in this figure, you can plainly see that CPU utilization is trending upward.The application server tier CPU aggregate, shown in Figure 9-8, illustrates that by 650 users, the CPU spikes at over 90 percent and then continues to increase, staying over 95 percent utilization at 725 users. The alarming component of Figure 9-8 is the near linear increase of CPU utilization to user load. A linear increase indicates that if the application cannot be refactored to reduce CPU utilization, then tuning efforts are always going to be battling CPU limitations.

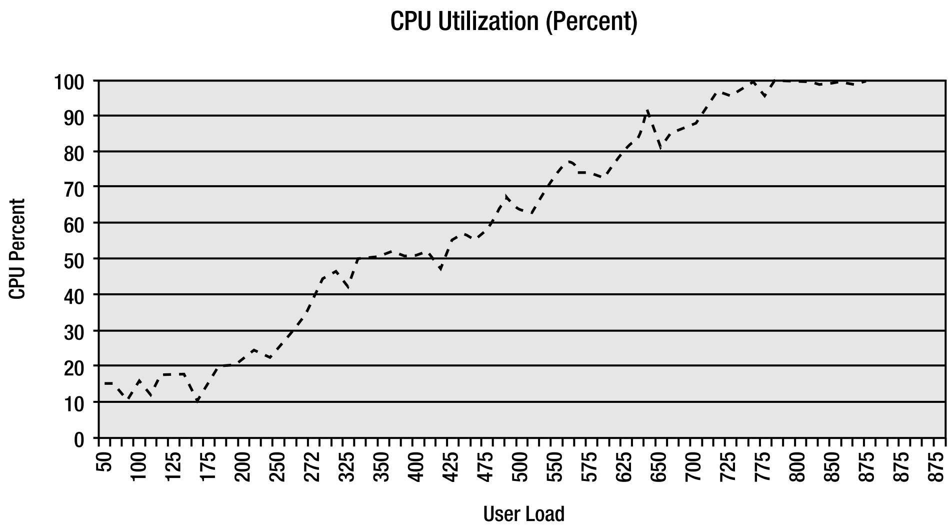

Figure 9-8. *The application server tier CPU aggregate*

Use Case Degradation Model

This section presents the performance of each use case performed during the capacity assessment.

Use Case: Login

Figure 9-9 illustrates the performance of the Login use case throughout the capacity assessment.

The response time for the Login request stayed relatively steady below the four-second mark until reaching about 500 users; it experienced a couple spikes above four seconds up until this point, but nothing sustained or in violation of the flexibility of the use case's SLA. Consistent with the observed degradation patterns, after the user load exceeded 550 users, the response time started to increase significantly.

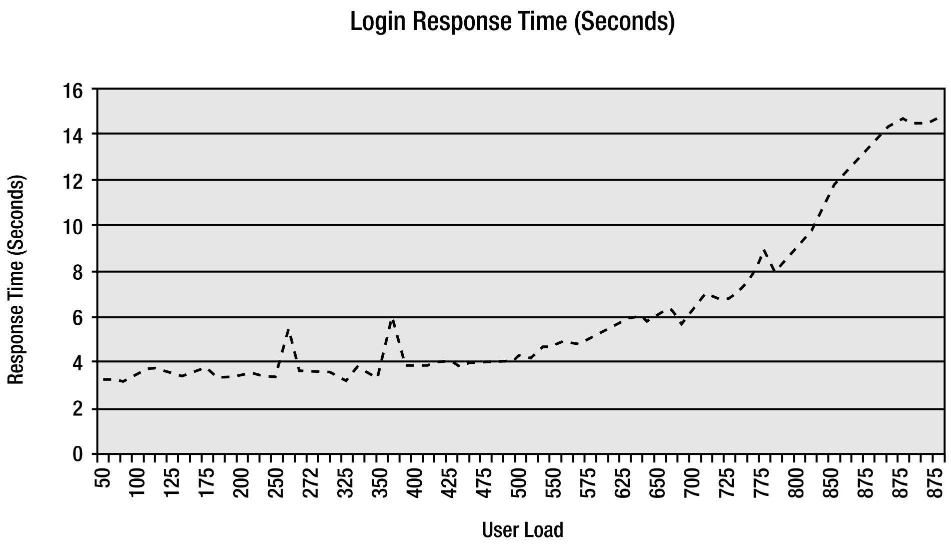

Figure 9-9. *The response time for the Login use case*

Resource Degradation Model

The Buy High, Sell Low application is CPU-bound primarily on the application server tier. Figure 9-8 shows the aggregate CPU utilization for the two machines running in the application server tier.

This section details the performance of each component in each tier as observed during the capacity assessment.

■ **Note** This section, the largest in the Capacity Assessment Report, contains performance graphs for all physical machines, internal servers, and network interactions. Pick the resources that are relevant for your environment and combine multiple related metrics on the same graph. The point of this section is to illustrate the degradation of resources over user load, and to record the behavior of resources for historical impact analysis. Therefore this section reports resources that degrade as well as resources that do not.

Web Tier

Insert graphs of the Web servers, including the following:

- CPU utilization
- Physical memory utilization
- Process memory utilization
- Request throughput
- Thread pool utilization

Application Tier

Insert graphs of the application server machines and instances, including the following information:

- CPU utilization
- Physical memory utilization
- Process memory utilization
- Application server instance heap utilization and garbage collection rates
- Application server instance thread pool utilizations
- Application server instance pool utilizations
- Application server instance cache counts of activation/passivation, hits, and misses

Database Tier

Insert graphs of the database machine and performance information, including the following information:

- CPU utilization
- Physical memory utilization
- Process memory utilization
- Disk I/O rates
- Cache hit/miss counts

Network

Insert graphs that display the response time and load between all servers against user load.

Impact Analysis

The performance of the Buy High, Sell Low application has degraded with the release of version 2.0. The average response time degradation is 12 percent, and the average resource utilization at expected load has increased by 7 percent. Throughput at expected usage has likewise degraded by 10 percent. The maximum capacity has decreased from 650 users to 550 users, a degradation of 15.4 percent.

Figure 9-10 illustrates the impact of the version 2.0 code against the version 1.1 code.

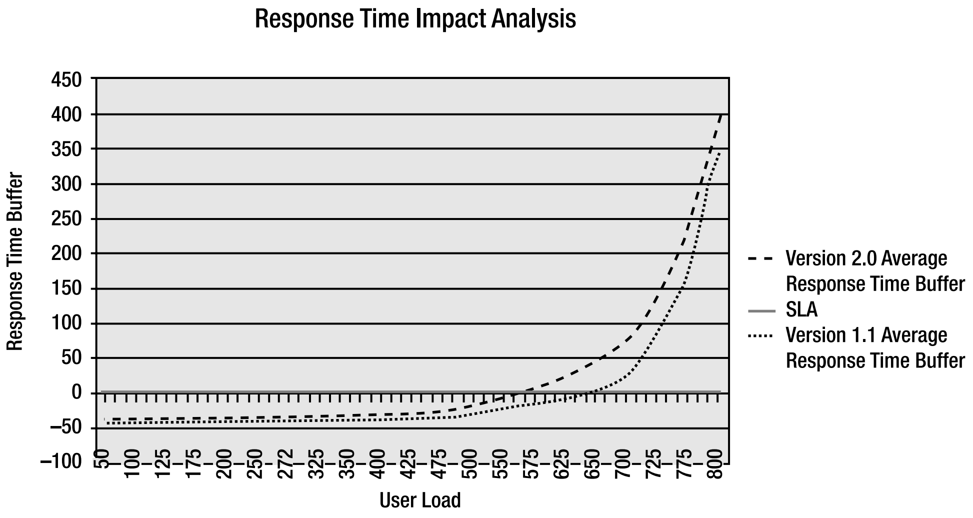

Figure 9-10. *The response time impact analysis*

The growth patterns between the two response time buffers are similar, but version 2.0 code crosses the SLA violation point at 550 users while the version 1.1 code crosses the SLA violation point at 650 users. This pattern shift represents a degradation in overall application capacity of 15.4 percent.

The resource utilization of both code versions was CPU-bound, but version 2.0 code degraded sooner than the previous version, as illustrated in Figure 9-11.

Continue to add resource degradation graphs for relevant resources in your capacity analysis, and where relevant, include graphs illustrating the performance differences between unchanged use cases. For example, if the login functionality did not change between versions, and the performance degraded, then calling out resource utilization differences in the impact analysis is important. However, if the functionality dramatically changed, such as swapping a text file for an LDAP server for user validation upon login, then a direct comparison of response times is not particularly useful to include (unless, of course, this comparison was requested). When functionality changes significantly, the best option is to compare the response time buffers between the application versions: is the new login functionality (with its new SLA) satisfying its SLA as well or better than the previous version?

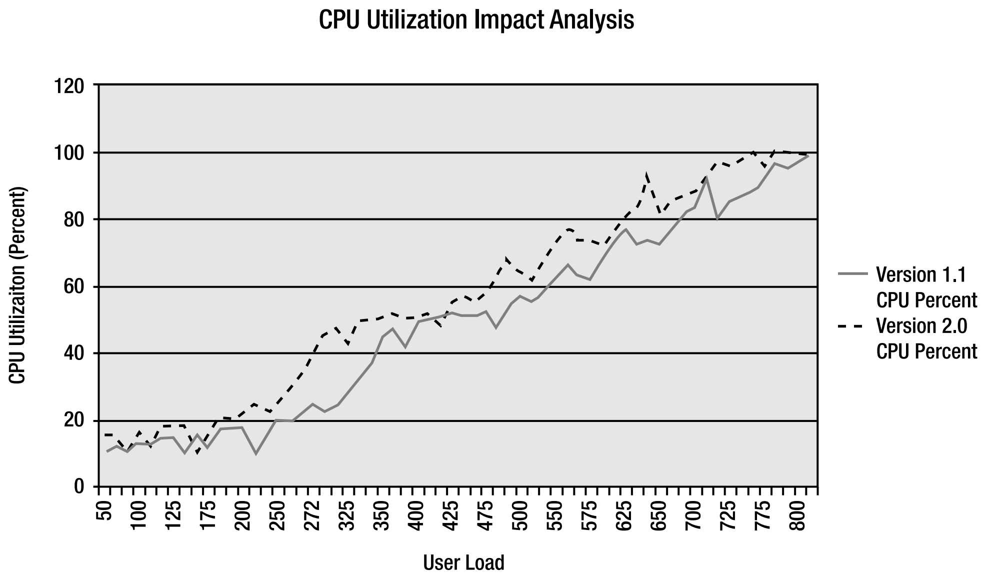

Figure 9-11. *The aggregate CPU utilizations at the application server tier between versions 1.1 and 2.0 of the Buy High, Sell Low application demonstrate that version 2.0 makes heavier use of its CPUs at lower user load than version 1.1.*

■ **Note** Many times the code functionality between major versions of an application is significantly different, so a deterministic impact analysis is impossible to perform. In these cases, performing the impact analysis is still beneficial, but focus on comparing the response times and resource utilizations of the expected usage of the new version against the response time and resource utilizations of the expected usage of the previous version. Figure 9-10 plots the average response time buffers rather than request response times to focus on that comparison—the average response time buffer value is relative to each use case's SLA. You may need to add this disclaimer to your impact analysis: "The impact analysis reports the differences between the performance of an enterprise environment executing two sets of application code functionality against their individual performance criteria; it does not necessarily reflect a direct response time impact."

Final Analysis and Recommendations

At the current user load of 500 users, all use cases are satisfied, and the environment can support an additional 50 users. This represents a 10 percent user buffer, which by industry standards is too low: the optimal user buffer for a volatile application like the Buy High, Sell Low application is anywhere between 25 and 40 percent.

The primary factor impeding the performance of the application is the CPU utilization of servers running in the application server tier. Therefore, either the addition of CPUs to existing servers in this tier or of a new physical machine is recommended.

Furthermore, the latest version of the application uses and saturates the CPU faster than the previous version; therefore, analyzing in a code-profiling tool the use cases identified as being problematic in the use case degradation model is recommended to determine the root of the performance changes.

Summary

This chapter discussed the difference between the concepts of performance and scalability: performance measures the speed with which a single request can be executed, while scalability measures the ability of a request to maintain its performance under increasing load. It also outlined the strategy of ensuring performance before testing for scalability. This strategy led to a detailed exploration into the ultimate scalability test—the capacity assessment. A capacity assessment identifies the following key points about your environment:

- Its performance at expected load
- The load that causes it to violate its SLAs
- The load that causes the environment to become saturated
- The load that forces the environment into the buckle zone

Equipped with the correct load testing strategy and monitoring tools, we explored how to ascertain this information and assemble it into a formal Capacity Assessment Report.

In the next chapter, we explore performance assessments that occur more frequently and are used to diagnose performance issues and validate the accuracy of capacity assessments.