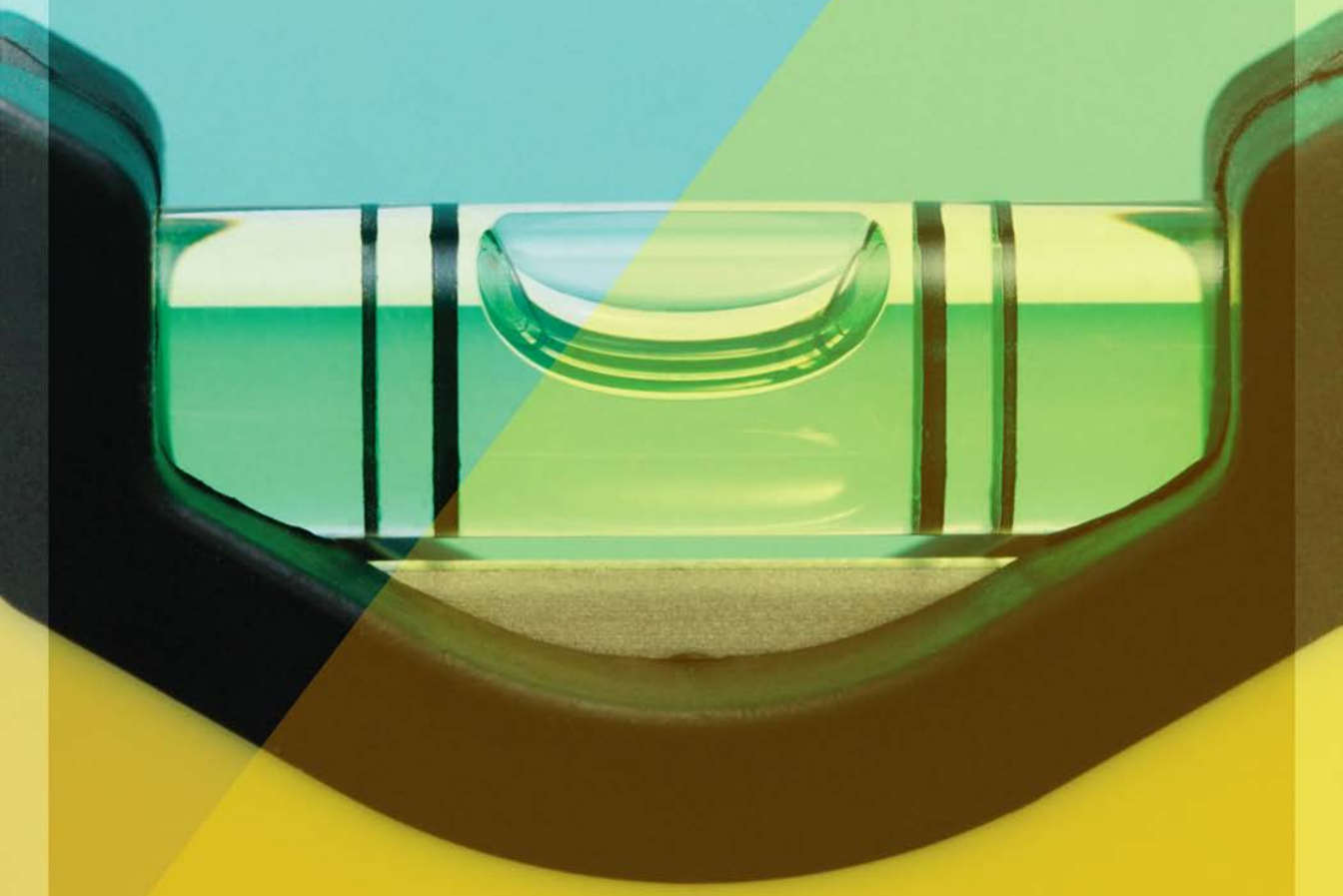
/Theory/In/Practice

The Art of Application Performance Testing



O'REILLY®

Ian Molyneaux

Software Engineering

The Art of Application Performance Testing

"Ian has maintained a vendor-agnostic methodology beautifully in this material. The metrics and graphs, along with background information provided in his case studies, eloquently convey to the reader, 'Methodology above all, tools at your discretion...' Ian's expertise shines through throughout the entire reading experience."

–Matt St. Onge, Enterprise Solution Architect,
HCL Technologies America / Teradyne

Businesses today live and die by network applications and web services. Because of the increasing complexity of these programs and the pressure to deploy them quickly, many professionals don't take the time to ensure that they'll perform well or scale effectively. *The Art of Application Performance Testing* explains the complete life cycle of the testing process and demonstrates best practices to help you plan, gain approval for, coordinate, and conduct performance tests on your applications.

You'll learn how to:

- Set realistic performance testing goals
- Implement an effective application performance testing strategy
- Interpret performance test results
- Use automated performance testing tools
- Test traditional local applications, web-based applications, and web services (SOAs)
- Recognize and resolves issues that are often overlooked in performance tests

Written by a consultant with 30 years of experience in the IT industry and over 12 years experience with performance testing, this easy-to-read book is illustrated with real-world examples and packed with practical advice. *The Art of Application Performance Testing* thoroughly explains the pitfalls of an inadequate testing strategy and offers you a robust, structured approach for ensuring that your applications perform well and scale effectively when the need arises.

Ian Molyneaux, EMEA SME (Subject Matter Expert) for Application Performance Assurance at Compuware, has held many roles in IT over the past 30 years. A techie at heart, he's shied away from anything management related.

US $34.99 CAN $34.99
ISBN: 978-0-596-52066-3
9 780596 520663 53499

O'REILLY® www.oreilly.com

The Art of Application Performance Testing

The Art of Application Performance Testing

Ian Molyneaux

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

The Art of Application Performance Testing
by Ian Molyneaux

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram
Production Editor: Adam Witwer
Production Services: Newgen Publishing and Data Services

Cover Designer: Mark Paglietti
Interior Designer: Marcia Friedman
Illustrator: Robert Romano

Printing History:
January 2009: First Edition.

ISBN: 978-0-596-52066-3

[V]

1231608032

CONTENTS

PREFACE

This book is written by an experienced application performance specialist for the benefit of those who would like to become specialists or have started working at application performance testing.

Businesses in today's world live and die by the performance of mission-critical software applications. Sadly, many applications are deployed without being adequately tested for scalability and performance. Effective performance testing identifies performance bottlenecks in a timely fashion and tells you where they are located in the application landscape.

The Art of Application Performance Testing addresses an urgent need in the marketplace for reference material on this subject. However, this is *not* a book on how to tune technology X or optimize technology Y. I've intentionally stayed well away from specific technologies except where they actually affect how you go about performance testing. My intention is to provide a commonsense guide that focuses on planning, execution, and interpretation of results and is based on a decade of experience in performance testing projects.

In the same vein, I won't touch on any particular industry performance testing methodology because—truth be told—they don't exist. Application performance testing is a unique discipline and is crying out for its own set of industry standards. I'm hopeful that this book may in some small way act as a catalyst for the appearance of formal processes.

Although I work for a company that's passionate about performance, this book is tool- and vendor-neutral. The processes and strategies described here can be used with any professional automated testing solution.

Hope you like it!

—Ian Molyneaux, December 2008

Audience

Although this book is for anyone interested in learning about application performance testing, it especially targets seasoned software testers and project managers looking for guidance in implementing an effective application performance testing strategy.

The book assumes that readers have some familiarity with software testing techniques, though not necessarily performance-related ones.

As a further prerequisite, effective performance testing is really possible only with the use of automation. Therefore, to get the most from the book you should have some experience of automated testing tools.

About This Book

Based on a number of my jottings (that never made it to the white paper stage) and ten years of hard experience, this book is designed to explain why it is so important to performance test any application before deploying it. The book leads you through the steps required to implement an effective application performance testing strategy.

Here are brief summaries of the book's chapters and appendixes.

Chapter 1, *Why Performance Test?*, discusses the rationale behind application performance testing and looks at performance testing in the IT industry from a historical perspective.

Chapter 2, *The Fundamentals of Effective Application Performance Testing*, introduces the building blocks of effective performance testing and explains their importance.

Chapter 3, *The Process of Performance Testing*, suggests a best-practice approach. It builds on Chapter 2, applying its requirements to a model for application performance testing.

Chapter 4, *Interpreting Results: Effective Root-Cause Analysis*, teaches effective root-cause analysis. It discusses the typical output of a performance test and how to perform effective analysis.

Chapter 5, *Application Technology and Its Impact on Performance Testing*, discusses the effects of particular software environments on testing. The approach is generic, so many details regarding your applications will depend on the characteristics of the technologies you use.

Appendix A, *Transaction Examples*, provides examples of how to prepare application transactions for inclusion in a performance test.

Appendix B, *POC and Performance Test Quick Reference*, reiterates the practical steps presented in the book.

Appendix C, *Automated Tool Vendors*, lists sources for the technologies required by performance testing. It is not an endorsement and is not intended to be complete.

Appendix D, *Sample KPI Monitoring Templates*, provides some examples of the sort of Key Performance Indicators you would use to monitor server and network performance as part of a typical performance test configuration.

Appendix E, *Sample Project Plan*, provides an example of a typical performance test plan based on Microsoft Project.

Conventions Used in This Book

The following typographical conventions will be used.

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings and also within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
> This icon signifies a tip, suggestion, or general note.

> **CAUTION**
> This icon indicates a warning or caution.

Glossary

The following terms are used in this book.

Application landscape
: A generic term describing the server and network infrastructure required to deploy a software application.

ICA
: Citrix proprietary protocol, Independent Computing Architecture.

ITIL
: Information Technology Infrastructure Library.

ITPM
: Information Technology Portfolio Management.

ITSM
: Information Technology Service Management.

JMS
: Java Message Service (formerly Java Message Queue).

Load injector
: A PC or server used as part of an automated performance testing solution to simulate real end-user activity.

IBM/WebSphere MQ
: IBM's Message Oriented Middleware.

POC
: Proof of Concept, a term used to describe a pilot project often included as part of the sales cycle. The intention is to compare the proposed software solution to a customer's, current application and so employ a familiar frame of reference. *Proof of value* is another term for a POC or Proof of Concept.

SOA
: Service-Oriented Architecture

Transaction
: A set of end-user actions that represent typical application activity. A typical transaction might be: log in, navigate to a search dialog, enter a search string, click the search button, and log out. Transactions form the basis of automated performance testing.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses

several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*The Art of Application Performance Testing* by Ian Molyneaux. Copyright 2009 Ian Molyneaux, 978-0-596-52066-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596520663

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

Many thanks to everyone at O'Reilly who helped to make this book possible and put up with the fumbling efforts of a novice author. These include editor Andy Oram, assistant editor Isabel Kunkle, managing editor Marlowe Shaeffer, Robert Romano for the figures and artwork, Jacquelynn McIlvaine and Karen Crosby for setting up my blog and providing me with the materials to start writing, and Karen Crosby and Keith Fahlgren for setting up the DocBook repository and answering all my questions.

In addition I would like to thank my employer, Compuware Corporation, for their kind permission to use screenshots from a number of their performance solutions to help illustrate points in this book.

I would also like to thank the following specialists for their comments and assistance on a previous draft: Peter Cole, President and CTO of Greenhat, for his help with understanding and expanding on the SOA performance testing model; Adam Brown of Quotium; David Collier-Brown of Sun Microsystems; Matt St. Onge; Paul Gerrard, principal of Gerrard consulting; Francois MacDonald of Compuware's Professional Services division; and Alexandre Mechain from Compuware France.

Finally, I would like to thank the many software testers and consultants whom I have worked with over the last decade. Without your help, this book would not have been written!

CHAPTER ONE

Why Performance Test?

Faster than a speeding bullet . . .

—Superman, Action Comics

This chapter poses some fundamental questions concerning the subject of this book. What is performance? Why carry out performance testing in the first place? Here I also define when an application is considered performant versus nonperformant and then discuss some common causes of a suboptimal end-user experience.

Nonperformant (i.e., badly performing) applications generally don't deliver their intended benefit to the organization. That is, they create a net cost of time, money, and loss of kudos from the application users and therefore can't be considered reliable assets. If an application is not delivering benefits, its continued existence is definitely on shaky ground—not to mention that of the architects, designers, coders, and testers (hopefully there were some!).

Performance testing is a neglected cousin of unit, functional, and system testing, which are well understood in most businesses and where the maturity level is high in many organizations. It is strange but true to say that executives do not appreciate the importance of performance testing. This has changed little over the past ten years despite despite the best efforts of consultants like myself and the many highly publicized failures of key software applications.

What Is Performance? The End-User Perspective

When is an application considered to be performing well?

My years of working with customers and performance teams suggest that the answer is ultimately one of perception. A well-performing application is one that lets the end user carry out a given task without undue *perceived* delay or irritation. Performance really is in the eye of the beholder.

With a performant application, users are never greeted with a blank screen during login and can achieve what they set out to accomplish without letting their attention wander. Casual visitors browsing a web site can find what they are looking for and purchase it without experiencing too much frustration, and the call-center manager is not being harassed by complaints of poor performance from the operators.

It sounds simple enough, and you may have your own thoughts on what constitutes good performance. But no matter how you define it, many applications struggle to deliver an acceptable level of performance.

Of course, when I talk about an application I'm actually referring to the sum of the whole, since an application is made up of many component parts. At a high level we can define these as the application software plus the application landscape. The latter includes the servers required to run the software as well as the network infrastructure that allows all the application components to communicate.

If any of these areas has problems, application performance is likely to suffer.

You might think that *all* we need do to ensure good application performance is observe the behavior of each of these areas under load and stress and correct any problems that occur. The reality is very different because this approach is often "too little, too late" and so you end up dealing with the symptoms of performance problems rather than the cause.

Performance Measurement

So how do we go about measuring performance? We've discussed end-user perception, but in order to accurately measure performance there are a number of key indicators that must be taken into account. These indicators are part of the performance requirements discussed further in Chapter 2 but for now we can divide them into two types: *service-oriented* and *efficiency-oriented*.

Service-oriented indicators are *availability* and *response time*; they measure how well (or not) an application is providing a service to the end users. Efficiency-oriented indicators are *throughput* and *utilization*; they measure how well (or not) an application makes use of the application landscape. We can define these terms briefly as follows:

Availability
: The amount of time an application is available to the end user. Lack of availability is significant because many applications will have a substantial business cost for even a small outage. In performance testing terms, this would mean the complete inability of an end user to make effective use of the application.

Response time
: The amount of time it takes for the application to respond to a user request. For performance testing, one normally measures *system response time*, which is the time between the user's requesting a response from the application and a complete reply arriving at the user's workstation.

Throughput
: The rate at which application-oriented events occur. A good example would be the number of hits on a web page within a given period of time.

Utilization
: The percentage of the theoretical capacity of a resource that is being used. Examples include how much network bandwidth is being consumed by application traffic and the amount of memory used on a server when a thousand visitors are active.

Taken together, these indicators can provide us with an accurate idea of how an application is performing and its impact, in capacity terms, on the application landscape.

Performance Standards

By the way, if you were hoping I could point you to a generic industry standard for good and bad performance, you're out of luck because no such guide exists. There have been various informal attempts to define a standard, particularly for browser-based applications. For instance, you may have heard the term "minimum page refresh time." I can remember a figure of 20 seconds being bandied about, which rapidly became 8 seconds. Of course, the application user (and the business) wants "instant response" (in the words of the Eagles band, "Everything all the time"), but this sort of performance is likely to remain elusive.

Many commercial Service Level Agreements (SLAs) cover infrastructure performance rather than the application itself, and they often address only specific areas such as network latency or server availability.

The following list summarizes research conducted in the late 1980s (Martin 1988) that attempted to map user productivity to response time. The original research was based largely on green-screen text applications, but its conclusions are probably still relevant.

Greater than 15 seconds
: This rules out conversational interaction. For certain types of applications, certain types of users may be content to sit at a terminal for more than 15 seconds waiting for the answer

to a single simple inquiry. However, to the busy call-center operator or futures trader, delays of more than 15 seconds may seem intolerable. If such delays can occur, the system should be designed so that the user can turn to other activities and request the response at some later time.

Greater than 4 seconds

These delays are generally too long for a conversation requiring the end-user to retain information in short-term memory (end-user's memory, not the computer's!). Such delays would inhibit problem-solving activity and frustrate data entry. However, after the completion of a transaction, delays of 4 to 15 seconds can be tolerated.

2 to 4 seconds

A delay longer than 2 seconds can be inhibiting to operations that demand a high level of concentration. A wait of 2 to 4 seconds at a terminal can seem surprisingly long when the user is absorbed and emotionally committed to completing the task at hand. Again, a delay in this range may be acceptable after a minor closure. It may be acceptable to make a purchaser wait 2 to 4 seconds after typing in her address and credit card number, but not at an earlier stage when she is comparing various product features.

Less than 2 seconds

When the application user has to remember information throughout several responses, the response time must be short. The more detailed the information to be remembered, the greater the need for responses of less than 2 seconds. Thus, for complex activities such as browsing camera products that vary along multiple dimensions, 2 seconds represents an important response-time limit.

Subsecond response time

Certain types of thought-intensive work (such as writing a book), especially with applications rich in graphics, require very short response times to maintain the users' interest and attention for long periods of time. An artist dragging an image to another location must be able to act instantly on his next creative thought.

Deci-second response time

A response to pressing a key of seeing the character displayed on the screen or to clicking a screen object with a mouse must be almost instantaneous: less than 0.1 second after the action. Many computer games require extremely fast interaction.

As you can see, the critical response time barrier seems to be 2 seconds. Response times greater than this have a definite impact on productivity for the average user, so our nominal page refresh time of 8 seconds for Internet applications is certainly less than ideal.

The Internet Effect

The explosive growth of the Internet has contributed in no small way to the need for applications to perform at warp speed. Many (or is that most?) businesses now rely on cyberspace for a good deal of their revenue in what is probably the most competitive

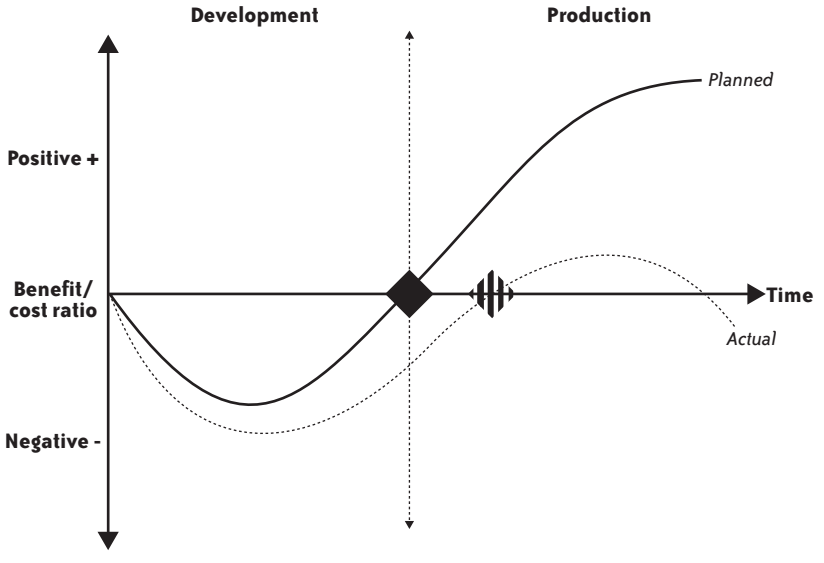


FIGURE 1-1. The IT business value curve

environment imaginable. If an end user perceives bad performance from your web site, their next click will likely be on *your-competition.com*.

Customer interface applications are also the most vulnerable to sudden spikes in demand, as more than a few high-profile retail companies have discovered at peak shopping times of the year.

Bad Performance: Why It's So Common

OK, I've tried to provide a basic definition of good and bad performance. It seems obvious, so why do many applications fail to achieve this noble aspiration? Let's look at some common reasons.

The IT Business Value Curve

Performance problems have a nasty habit of turning up late in the application life cycle, and the later you discover them, the greater the cost to resolve. Figure 1-1 illustrates this point.

The solid line (planned) indicates the expected outcome when the carefully factored process of developing an application comes to fruition at the planned moment (black diamond). The application is deployed successfully on schedule and immediately starts to provide benefit to the business with little or no problems after deployment.

Resolving Performance Defects (2006)

Approach	*% Resolved in Production*
Firefighting	100%
Performance Validation	30%
Performance Driven	5%

Source: Forrester Research

FIGURE 1-2. Forrester Research on resolution of performance defects

The broken line (actual) demonstrates the all-too-frequent reality when development and deployment targets slip (striped diamond) and significant time and cost is involved in trying to fix performance issues in production. This is bad news for the business because the application fails to deliver the expected benefit.

This sort of failure is becoming increasingly visible at the board level as companies seek to implement Information Technology Service Management (ITSM) and Information Technology Portfolio Management (ITPM) strategies on the way to the holy grail of Information Technology Infrastructure Library (ITIL) compliance. The current frame of reference considers IT as just another (important) business unit that must operate and deliver within budgetary constraints. No longer is IT a law unto itself that can consume as much money and resources as it likes without challenge.

Performance Testing Maturity: What the Analysts Think

But don't just take my word for it. Figure 1-2 is based on data collected by Forrester Research in 2006 looking at the number of performance defects that have to be fixed in production for a typical application deployment.

As you can see, three levels of performance testing maturity were identified. The first one, "Firefighting," occurs when little or no performance testing was carried out prior to application deployment, so effectively all performance defects must be resolved in the live environment. This is the least desirable approach but, surprisingly, is still relatively common. Companies in this mode are exposing themselves to serious risk.

The second level, "Performance Validation (or Verification)," covers companies that set aside time for performance testing but not until late in the application life cycle. Hence a significant number of performance defects are still found in production (30%). This is where most organizations currently operate.

The final level, "Performance Driven," is where performance considerations have been taken into account at every stage of the application life cycle. As a result, only a small number of

performance defects are discovered after deployment (5%). This is what companies should aim to adopt as their performance testing model.

Lack of Performance Considerations in Application Design

Returning to our discussion of common reasons for failure: if you don't take performance considerations into account during application design, you are asking for trouble. Good design lends itself to good performance, or at least the agility to change or reconfigure an application to cope with unexpected performance challenges. Design-related performance problems that remain undetected until late in the life cycle are difficult to overcome completely, and doing so is sometimes impossible without significant application reworking.

Most applications are built from software components that can be tested individually and may perform well in isolation, but it is equally important to consider the application as a whole. These components must interact in an efficient manner in order to achieve good performance.

Performance Testing Is Left to the Last Minute

As mentioned, most companies are in "Performance Validation (or Verification)" mode. Here performance testing is done just before deployment, with little consideration given to the amount of time required or to the ramifications of failure. Although better than "firefighting," this mode still carries a significant degree of risk that you won't identify serious performance defects that appear in production or won't allow enough time to correct problems identified before deployment.

One typical result is a delay in the application rollout while the problems are resolved. An application that is deployed with significant performance issues will require costly, time-consuming remedial work after deployment. Even worse, the application might have to be withdrawn from circulation entirely as it's battered into shape.

All of these outcomes have an extremely negative effect on the business and on the confidence of those expected to use the application. You need to test for performance issues as early as you can, rather than leave it to the last minute.

How Many Users Are There?

Often, not enough thought is given to capacity or sizing. Developers and testers may overlook the size and geography of the end-user community. Many applications are developed and subsequently tested without more than a passing thought for the following considerations.

- How many end users will actually use the application?
- How many of these users will use it concurrently?
- How will the end users connect to the application?
- How many additional end users will require access to the application over time?

- What will the final application landscape look like in terms of the number and location of the servers?
- What effect will the application have on network capacity?

Neglect of these issues manifests itself in unrealistic expectations for the number of concurrent users that the application is expected to support. Furthermore, developers tend to ignore the large numbers of users who are still at the end of low-bandwidth, high-latency WAN links. I will cover connectivity issues in more detail in Chapter 2.

Underestimating Your Popularity

This might sound a little strange, but many companies underestimate the popularity of their new web applications. This is partly because they are deployed without taking into account the "novelty factor." When something's shiny and new, people generally find it interesting and so they turn up in droves. Therefore, the 10,000 hits you had carefully estimated for the first day of deployment suddenly becomes 1,000,000 hits and your application infrastructure goes into meltdown!

Putting it another way, you need to plan for the peaks rather than the troughs.

SPECTACULAR FAILURE: A REAL-WORLD EXAMPLE

Some years ago, the U.K. government decided to make available the results of the 1901 census on the Internet. This involved a great deal of effort converting old documents into a modern digital format and creating an application to provide public access.

I was personally looking forward to the launch, since I was tracing my family history and this promised to be a great source of information. The site was launched and I duly logged in. Although I found things a little slow, I was able to carry out my initial searches without too much issue. However, when I returned to the site 24 hours later, I was greeted with an apologetic message saying that the site was unavailable. It remained unavailable for many weeks until finally being relaunched.

This is a classic example of underestimating your popularity. The amount of interest in the site was far greater than anticipated, so it couldn't deal with the volume of hits. This doesn't mean that no performance testing was carried out prior to launch. But it does suggest that the performance expectations for the site were too conservative.

You have to allow for those peaks in demand.

Performance Testing Is Still an Informal Discipline

As mentioned previously, performance testing is still very much an informal discipline. The reason for this is hard to fathom, because functional/regression testing has been well

established for many years. There is a great deal of literature and expert opinion available in that field, and many established companies specialize in test execution and consulting.

For performance testing, the converse is true, at least in terms of reference material. One of the reasons that I was prompted to put (virtual) pen to paper was the abject lack of anything in the way of written material that focused on application performance testing. There are myriad publications that explain how to tune and optimize an application, but nothing about how to set up effective performance testing in the first place. If you're a budding performance tester, you have been very much on your own (until now!).

Try it for yourself. Searching for performance testing on the Internet is more likely to yield information about sports cars or even washing machines.

Not Using Automated Testing Tools

You can't really carry out effective performance testing without using automated test tools. Getting a hundred (disgruntled) staff in on a weekend (even if you buy them all lunch) and strategically deploying people with stopwatches just won't work. Why? You'll never be able to repeat the same test twice. Furthermore, making employees work for 24 hours if you find problems is probably a breach of human rights.

Also, how do you possibly correlate response times from 100 separate individuals, not to mention what's happening on the network and the servers? It simply doesn't work unless your application has fewer than five users, in which case you probably don't need this book.

A number of vendors make great automated performance testing tools. Costs will vary greatly depending on the scale of the testing you need to execute, but it's a competitive market and biggest is not always best. So you need to do your homework and prepare a report for your finance department. Appendix C contains a list of the leading vendors.

Application Technology Impact

Certain technologies that are commonly used in creating applications didn't work well with the first and even second generation of automated test tools. This has become a considerably weaker excuse for not doing any performance testing, since the vast majority of applications are now web-enabled to some degree. Web technology is generally well supported by the current crop of automated test solutions.

Development and deployment have crystallized by now onto a (relatively) few core technologies. Accordingly, most automated tool vendors have followed suit with the support that their products provide. I will look at some common application technologies and their impacts on performance testing in Chapter 5.

Summary

This chapter has served as a brief discussion about application performance, both good and bad. I've touched on some of the common reasons why failure to do effective performance testing leads to applications that do not perform well. You could summarize the majority of these reasons with a single statement:

> Testing for performance is not given the consideration that its importance deserves as part of the application's entire life cycle.

In the next chapter we move on to a discussion of the building blocks that are required to implement an effective application performance testing strategy.

CHAPTER TWO

The Fundamentals of Effective Application Performance Testing

For the want of a nail . . .

—Anonymous

In this chapter we cover a basic task that seems to be avoided even by companies that take on the task of performance testing: capturing performance requirements. Unlike the "functional" requirements upon which regression and unit testing are based, performance related requirements can be considered "nonfunctional." Semantics aside, they are vital activities that must be addressed in order to carry out effective performance testing.

The idea of a formal approach to application performance testing is still considered novel by many. Just why is something of a mystery, because (as with any project) failing to plan properly will inevitably lead to misunderstandings and problems. Performance testing is no exception.

With this thought in mind, let's reiterate my performance testing mantra:

Performance awareness should be built into the application life cycle as early as possible.

In other words: if you don't start your planning with performance in mind, then you expose yourself to significant risk that your application will never perform to expectations.
With any new project you should ask the following questions.

- How many end users will the application need to support at release? After 6 months, 12 months, 2 years?

- Where will these users be located, and how will they connect to the application?
- How many of these users will be concurrent at release? After 6 months, 12 months, 2 years?

The answers then lead to other questions, such as:

- How many and what specification of servers will I need for each application tier?
- What sort of network infrastructure do I need to provide?

You may not be able to answer definitively or immediately, but the point is that you've started the ball rolling by thinking early on about two vital things, capacity and performance.

You will need to address many factors before you can implement an effective performance testing strategy. Of course, there's a lot more to performance testing than simply generating a load and watching what happens. In my experience, the most important requirements include:

- Choosing an appropriate performance testing tool
- Designing an appropriate performance test environment
- Setting realistic and appropriate performance targets
- Making sure your application is stable enough for performance testing
- Obtaining a code freeze
- Identifying and scripting the business-critical transactions
- Providing sufficient test data of high quality
- Ensuring accurate performance test design
- Identifying the server and network monitoring key Performance Indicators (KPIs)
- Allocating enough time to performance test effectively

> **NOTE**
>
> **There are a number of possible mechanisms for gathering requirements, both functional and nonfunctional. For many companies this requires nothing more sophisticated than Microsoft Word. But serious requirements management, like serious performance testing, requires automation. A number of vendors provide tools that allow you to manage requirements in an automated fashion; these scale from simple capture and organization to solutions with full-blown Universal Modeling Language (UML) compliance. You will find a list of tools from leading vendors in Appendix C.**

Many of these requirements are obvious, but some are not. It's the requirements you overlook that will have the greatest impact on the success or failure of a performance testing project. This chapter examines each of them in detail.

Choosing an Appropriate Performance Testing Tool

Automated tools have been around in some form for the best part of 15 years. During that period, application technology has gone through many changes, moving from a norm of "fat client" to web-enablement. Accordingly, the sort of capabilities that automated tools must now provide is very much biased toward web development, and there is much less requirement to support legacy technologies that rely on a two-tier application model. This "web focus" is good news for the performance tester, because there are now many more automated tool vendors in the marketplace to choose from with offerings to suit even modest budgets. There are also a number of free tools available as shareware or open source (*http//www.opensource.org*); see Appendix C.

Another sea change may well be about to occur with the emergence of Service Oriented Architecture (SOA), where the performance focus is not just on the end-user transaction but also includes the business process. This concept is generally too abstract for the current crop of performance tools, although it is possible to test business process components that make use of technologies like web services. I will discuss SOA and other technology challenges in Chapter 5.

All of this is well and good, but here's a note of caution. When your performance testing needs do move outside of the Web, the choice of tool vendors diminishes rapidly and technology challenges that have plagued automated tools for many years are still very much in evidence. These problems center not on execution and analysis but rather on being able to successfully record application activity and then modify the resulting scripts for use in a performance test. Phrases such as "encryption" and "compression" are not good news for performance test tools; unless these technologies can be disabled, it is unlikely that you will be able to create scripts.

Even technologies that are web-based can present problems for performance test tools. For example, if you need to deal with streaming media or client certificates then not all vendors will be able to offer a solution. You should carefully match your needs to performance tool capabilities before making your final choice, and I recommend you insist on a Proof of Concept (POC) before making any commitment to buy.

Despite these challenges, automated tools are needed to carry out serious performance testing. In Chapter 1 I mentioned this as a reason why many applications are not properly performance tested prior to deployment. There is simply no practical way to provide reliable, repeatable performance tests without using some form of automation.

Accordingly, the aim of any automated performance testing tool is to simplify the testing process. This is normally achieved by providing the ability to record end-user activity and to render this data as transactions or scripts. The scripts are then used to create load testing sessions or scenarios that represent a mix of typical end-user activity. These are the actual performance tests and, once created, they can easily be rerun on demand, which is a big advantage over any sort of manual testing.

Another huge advantage over manual testing is the ability to quickly correlate performance data from various sources (such as the servers, network, and application response time) and then present this in a single view. This information is normally stored for each test run, allowing easy comparison of the results of multiple test executions.

Testing Tool Architecture

Automated performance test tools typically have the following components.

Scripting module
: Enables recording of end-user activity and may support many different middleware protocols. Allows modification of the recorded scripts to associate internal/external data and configure granularity of response-time measurement.

Test management module
: Allows the creation and execution of load test sessions or scenarios that represent different mixes of end-user activity. These sessions make use of nominated scripts and one or more load injectors.

Load injector(s)
: Generates the load—normally from multiple workstations or servers, depending on the amount of load required.

Analysis module
: Provides the ability to analyze the data collected from each test execution. This data is typically a mixture of autogenerated reports and graphical or tabular reporting. There may also be an "expert" capability that provides automated analysis of results and highlights areas of concern.

Complementing these components may be additional modules that monitor server and network performance while a load test is running. Figure 2-1 demonstrates a typical automated performance test tool deployment. The application users have been replaced by a group of servers or workstations that will be used to inject application load by creating "virtual users."

What to Look for in an Automated Performance Testing Tool

Whether you are delivering a performance testing service to a customer or executing in-house you need to make sure that the proposed testing tool is actually compatible with your application technology. This is a pretty obvious statement, but many performance testing projects run into problems during the transaction scripting stage that are due to insufficient technical evaluation of the tool being used. Many testing service providers keep a "toolbox" of solutions from a number of different vendors, which allows them to choose the most appropriate performance testing tool for a particular performance testing engagement.

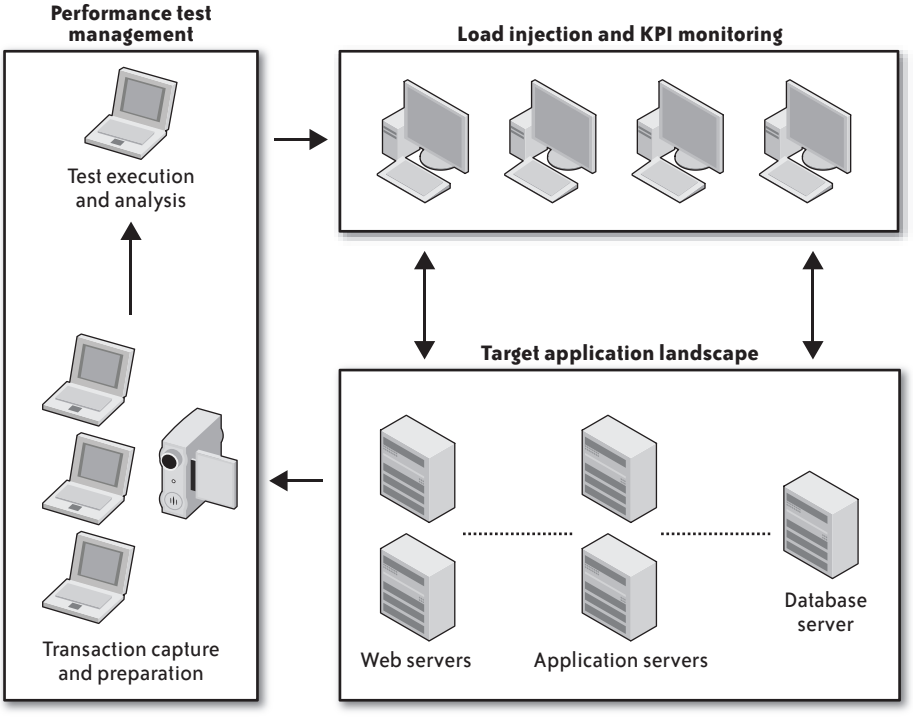


FIGURE 2-1. Typical performance tool deployment

I will discuss the impact of different technologies on performance testing in more detail in Chapter 5, but for now it is sufficient to point out that evaluation of a performance testing tool should include the following steps.

Tool vendor support

Ensure that the performance testing tool vendor "officially" supports your application technology—specifically, how the application client talks to the next application tier. For example, a typical browser-based client will in most cases be using HTTP or HTTPS as its primary communication protocol.

Licensing model

Most performance tools offer a licensing model based on two components.

- *The largest load test you can execute in terms of virtual users*

  This may be organized in breaks or bands, perhaps offering an entry level of up to 100 virtual users with additional cost for higher numbers. You may also find tool vendors who offer unlimited number of virtual users for a (higher) fixed cost.

- *The application technologies that the tool can support*

You may purchase the tool with support for the HTTP protocol, but a later request to support, for example, Citrix would incur an additional cost. The application technology may or may not be locked to the virtual users. Some vendors also offer a term-license model whereby you can temporarily extend your license for one-off or occasional requirements to test much greater numbers of virtual users.

Make sure that you are clear in your own mind about the licensing model before making the decision to short-list or buy.

Proof of Concept (POC)

You need to try the tool out against your application, so insist on a POC. Identify a minimum of two transactions to record: one that can be considered "read-only" (e.g., a simple search activity that returns a result set of one or more entries); and one that inserts or makes updates to the application database. This will allow you to check that the recorded transaction actually replays correctly. If your application is "read only," make sure to check the replay logs for each transaction to ensure that there are no errors indicating replay failure.

Scripting effort

Many performance tool vendors claim that there is little or no need to make manual changes to the scripts that their tools generate. This may be true for simple browser transactions that navigate around a couple of pages, but the reality is that you *will* have to delve into the code at some point during transaction scripting.

As a result, you may find that many manual changes are required to your recorded transaction before it will replay successfully. If each set of changes requires several hours per script to implement with one tool but seems to be automatically handled by another, then you need to consider the potential cost savings and the impact of the extra work on your performance testing project and the testing team members.

Consider also the skill level of your testing team. If they come from a largely development background then getting their hands dirty coding should not present too much of a problem. If, however, they are relatively unfamiliar with coding you may be better off considering a tool that provides a lot of Wizard-driven functionality to assist script creation and preparation.

Solution versus load testing tool

Some vendors offer only a load testing tool whereas others offer a performance testing solution. Solution offerings will inevitably cost more but generally provide a much greater degree of granularity in the analysis they provide. In addition to load testing, they may include any or all of the following: automated requirements management, automated data creation and management, pre-performance test application tuning and optimization, response-time prediction and capacity modeling, application server analysis to component level, and integration with end-user experience (EUE) monitoring after deployment.

In-house or outsource?

If you have limited resources internally or are working within strict time constraints, consider outsourcing the performance testing project to an external vendor. Some tool vendors offer a complete range of services from tool purchase with implementation assistance through to a complete performance testing service, and there are many companies that specialize in testing and can offer the same kind of service using whatever performance toolset they consider appropriate. This has the advantage of moving the commercials to time and materials and removes the burden of selecting the right testing tool for your application. The main disadvantage of this approach is that—if you need to carry out frequent performance tests—the cost per test will rapidly exceed the cost of buying an automated performance testing solution outright.

The Alternatives

If your application is web-based and customer-facing, you might want to consider one of several vendors who provide a completely external performance testing service. Essentially they manage the whole process for you, scripting key transactions and delivering the load via multiple remote "points of presence." This approach has the advantages of removing the burden of providing enough injection hardware and of being a realistic way of simulating real user load, since these vendors typically connect directly to the Internet backbone via large-capacity pipes. This is an attractive alternative for one-off or occasional performance tests, particularly if high volumes of virtual users are required. These vendors often provide an End User Experience monitoring service that may also be of interest.

The downside is that you will have to pay for each test execution, and any server or network KPI monitoring will be your responsibility to implement and synchronize with the periods of performance test execution. This could become an expensive exercise if you find problems with your application that require many test reruns to isolate and correct.

Still this alternative is well worth investigating. Appendix C includes a list of vendors who provide this solution.

Designing an Appropriate Performance Test Environment

Next we need to consider our performance test environment. In an ideal world, it would be an exact copy of the deployment environment, but this is rarely the case for a variety of reasons.

The number and specification of servers (probably the most common reason)

It is often impractical, for reasons of cost and complexity, to provide an exact replica of the server content and architecture in the deployment environment. Nonetheless, even if you cannot replicate the numbers of physical servers at each application tier then do try to match the specification of the live servers. This will allow you to determine the capacity of an individual server and provide a reliable baseline for horizontal scaling.

Bandwidth and connectivity of network infrastructure

In a similar vein, it is rare for the target servers to be deployed in the same location as their live counterparts, although it is often possible to share the same network infrastructure.

Number of application tiers

In many cases, the number of application tiers has greater impact on the validity of performance testing than simply the number of servers. Therefore, it is crucial to retain the live tier deployment model unless there is no alternative. For instance, if a proxy server lies between your application server and the user, try to insert a proxy server in the test environment as well.

Sizing of application databases

The size of the test environment database should closely approximate the live one; otherwise, the difference will have a considerable impact on the validity of performance test results. Executing tests against a 1 GB database when the live deployment will be 50 GB is completely unrealistic.

Therefore, the typical performance test environment is a subset of the deployment environment. Satisfactory performance here suggests that things can only get better as we move to full-blown deployment. (Unfortunately, that's not always true!)

I have come across projects where all performance testing was carried out on the live environment. However, this is fairly unusual and adds its own set of additional considerations such as the effect of other application traffic and the impact on real application users when a test is executed. Such testing is usually scheduled out of normal working hours in order to minimize external effects on the test results and the impact of the testing on the live environment.

In short, you should strive to make the performance test environment as close to a replica of the live environment as possible within existing constraints. This requirement differs from unit testing, where the emphasis is on ensuring that the application works correctly. The misconception often persists that a minimalist deployment will be suitable for both functional and performance testing.

For example, one important U.K. bank has a test lab setup to replicate their largest single branch. This environment comprises over 150 workstations, each configured to represent a single teller, with all the software that would be part of a standard desktop build. On top of this is deployed test automation software providing an accurate simulation environment for functional and performance testing projects.

Setting up a performance test environment is rarely a trivial task, and it may take many weeks or even months to achieve. Therefore, you need to plan for a realistic amount of time to complete this activity.

Having a thorough understanding of the entire test environment at the outset enables more efficient test design and planning and helps you identify testing challenges early in the project. In some situations, this process must be revisited periodically throughout the project's life cycle.

To summarize, there are three levels of preference when it comes to designing a test environment:

An exact or very close copy of the live environment
: This is the ideal but as already stated is often difficult to achieve for practical and commercial reasons.

A subset of the live environment with fewer servers but specification and tier-deployment matches to that of the live environment
: This is frequently achievable—the important consideration being that from a bare-metal perspective, the specification of the servers at each tier must match that of the live environment. This allows an accurate assessment to be made of the capacity limits of individual servers providing a reliable model for horizontal scaling.

A subset of the live environment with fewer servers of lower specification
: Probably the most common situation; the test environment is sufficient to deploy the application but the number, tier deployment, and specification of servers differ significantly from the live environment.

Virtualization

A relatively new factor influencing test environment design is the emergence of virtualization technology that allows multiple "virtual" server instances to exist on a single physical machine.VMWare has been the market leader for some years, although this position is now being challenged by offerings from other vendors such as Microsoft with their virtual server technology. The important thing to note is that virtualization makes a closer simulation possible with regard to the number and specification of servers present in the live environment.

If the live environment also makes use of virtualization then so much the better, since then a very close approximation will be possible between test and live. If this is not the case then there are some things to bear in mind when comparing logical (virtual) to physical (real) servers:

Bus versus LAN-WAN
: Communication between virtual servers sharing the same physical bus will exhibit different characteristics than servers communicating over LAN or WAN. Although such virtual communication may use virtual Network Interface Cards (NICs), it will not suffer (unless artificially introduced) typical network problems like bandwidth restriction and latency effects. In a data center environment there should be few network problems with physical servers, so this should not be an issue. However, if your live physical servers connect across distance via LAN or WAN, then substituting common-bus virtual servers in test will not be a true representation of server-to-server communication. In a similar

vein, it is best (unless matching the live architecture) to avoid deploying virtual servers from different tiers on the same physical machine. That is, don't mix and match; keep all web application and database virtual servers together but on different physical servers.

Physical versus virtual NIC

Virtual servers tend to use virtual NICs. This means that, unless there is one physical NIC for each virtual server, multiple servers will have to share the same physical NIC. Current NIC technology is pretty robust, so it takes a lot to overload a card. However, channeling several servers worth of network traffic down a single NIC increases the possibility of overload. I always advise minimizing the ratio of physical NICs to virtual servers.

Injection Capacity

As part of setting up the performance test environment, you need to make sure there are sufficient hardware resources to generate the required load. Automated performance test tools use one or more machines as load injectors to simulate real user activity. Depending on the application technology, there will be a limit on the number of virtual users that you can generate from a given machine.

Some automated performance tools allow you to monitor the load being placed on an injector machine. It's important to ensure that none of the injectors are overloaded in terms of CPU or memory utilization, since this can introduce inaccuracy into your performance test results. Although tool vendors generally provide guidelines on injection requirements, there are many factors that can influence injection limits per machine. You should always carry out a "dress rehearsal" to determine how many virtual users can be generated from a single platform. This will give you an accurate idea of how many additional injector machines are required to create a given virtual user load.

Automated performance testing will always be a compromise simply because you are using a single machine to represent many users. Always use as many injector machines as possible to spread the load. If you have ten machines available but could generate the load with four machines, it's still a better strategy to use all ten.

There is also the question of how the application reacts to the Internet Protocol (IP) address of each incoming virtual user. This has relevance in the following situations.

Load balancing

Some load balancing strategies use the IP address of the incoming user to determine the server to which the user should be connected for the current session. If you don't take this into consideration then all users from a single injector will have the same IP address and will be allocated to the same server, which will not be an accurate test of load balancing.

In these circumstances you need to implement "IP spoofing," where multiple IP addresses are allocated to the NIC on each injector machine and the automated performance tool

allocates a different IP address to each virtual user. Not all automated performance test tools provide this capability, so bear this in mind when making your selection.

User session limits
: Application design may enforce a single user session from one physical location. In these situations, performance testing will be difficult unless this limitation can be overcome. If you are thus limited to one virtual user per injector machine, then you will need a large number of injectors!

Certain other situations will also affect how many injectors you will need to create application load.

The application technology may not be recordable at the middleware level
: Chapter 5 covers the impact of technology on performance testing. In terms of load injection, if you cannot create middleware-level scripts for your application then you have a serious problem. Your options are limited to (a) making use of functional testing tools to provide load from the presentation layer or (b) making use of some form of thin-client deployment that *can* be captured with your performance testing tool—for example, Citrix ICA or MS Terminal Services RDP protocol. If you can't take the thin-client route then your injection capability will probably be limited to one virtual user per machine.

You need to measure performance from a presentation layer perspective
: Performance testing tools tend to work at the middleware layer and so have no real concept of activity local to the application client apart from periods of dead time on the wire. If you want to time, for example, how long it takes a user to click on a combo-box and then choose the third item, this will require the use of presentation layer scripts and functional testing tools. Some tool vendors allow you to freely combine load and functional scripts in the same performance test, but this is not a universal capability. If you need this functionality, check to see that the vendors on your short list can provide it.

Addressing Different Deployment Models

From where will end users access the application? If everybody is on the local area network (LAN), then your load injection can be entirely LAN-based. However, if you have users across a wide area network (WAN), then you need to take into account the prevailing network conditions they will experience. These primarily include the following.

Available bandwidth
: A typical LAN currently offers a minimum of 100 Mbits, and many LANs now boast 1,000 or even 10,000 Mbits of available bandwidth. WAN users, however, are not as fortunate and may have to make do with as little as 256 Kbits. Low bandwidth and high data presentation do not generally make for good performance, so this must be factored into your performance testing model.

Network latency

Think of this as delay. Most LANs have little or no latency, but the WAN user is often faced with high latency conditions that can significantly affect application performance.

Application design can have a major impact on how "WAN friendly" the application turns out to be. I have been involved in many projects where an application flew on the LAN but crawled on the WAN. Differing deployment models will have a bearing on how you design your performance testing environment.

There are tools available that allow you to model application performance in differing network environments. If significant numbers of your application users will be WAN-based, then I encourage you to consider using such. Typically they allow you to vary the bandwidth, latency, and congestion against a live recording of application traffic. This allows you to accurately re-create the sort of experience an end user would have at the end of a modest ADSL connection.

You may also wish to include WAN-based users as part of your performance test design. There are a number of ways to achieve this.

Modify transaction replay

Some performance testing tools allow you to simulate WAN playback even though the testing is carried out on a LAN environment. This is achieved by altering the replay characteristics of nominated transactions to represent a reduction in available bandwidth—in other words, slowing down the rate of execution. In my experience there is considerable variation in how automated tool vendors implement this feature, so you should satisfy yourself that what is provided will be accurate enough for your requirements.

Load injection from a WAN location

This is certainly the most realistic approach although it is not always achievable in a test environment. You need to position load injector machines at the end of real WAN links and simulate the number of users expected to use this form of connectivity as part of your performance test execution.

Network simulation

There are products available that allow you to simulate WAN conditions from a network perspective. Essentially a device is inserted into your test network that can introduce a range of network effects, including bandwidth reduction.

Environment Checklist

The following checklist will help you determine how close your test environment will be to the live deployment.

From the deployment model for the application, collect the following information where relevant, for each server tier. This includes black-box devices such as load balancers and content servers if they are present.

Number of servers
: The number of physical or virtual servers for this tier.

Load balancing strategy
: The type of load balancing mechanism in use (if relevant).

Hardware inventory
: Number and type of CPUs, amount of RAM, number and type of NICs.

Software inventory
: Standard build software inventory *excluding* components of application to be performance tested.

Application component inventory
: Description of application components to be deployed on this server tier.

External Links
: Any links to other internal or third-party systems. These can be challenging to replicate in a test environment and are often completely ignored or replaced by some sort of mock-up. Failing to take them into account is to ignore a potential source of performance bottlenecks. At the very minimum you should provide functionality that represents expected behavior. For example, if the external link is a web service request to a credit reference service that needs to provide subsecond response, then build this into your test environment. You will then have confidence that your application will perform well as long as external services are doing their part. (Make sure that any external link functionality you provide is robust enough to cope with the load that you create!)

> NOTE
> **If some of this required information is hard to come by, then probably not enough consideration has been given to the design of the deployment environment.**

Network connectivity is usually less challenging to replicate during testing, at least with regard to connections between servers. Remember that any load you apply should be present at the correct location in the infrastructure. For incoming Internet or intranet traffic, this is typically in front of any firewall or security devices that may be present.

Software Installation Constraints

An important consideration that is often overlooked is to identify any constraints that may apply to the use of third-party software within the test environment. By "constraints" I mean internal security policies that restrict the installation of software or remote access to servers and network infrastructure. This may limit the amount and granularity of server and network monitoring that will be possible and in the worst case may prevent the use of any monitoring capability associated with the automated performance test tool you wish to use.

Although not normally a concern when performance testing in-house, such constraints are a real possibility when providing performance testing services to other organizations. This situation is more common than you might think and is not something you want to discover at the last minute! If this situation does arise unexpectedly then you will be limited to whatever monitoring software is already installed in the test environment.

Setting Realistic and Appropriate Performance Targets

Now, what are your performance targets? These are often referred to as performance goals or a Service Level Agreement (SLA). Unless you have some clearly defined performance targets in place against which you can compare the results of your performance testing, you could be wasting your time.

Consensus

It's crucial to get consensus on these performance targets from all stakeholders. Every group on whom the application will have an impact—including the application users and senior management—must agree on the same performance targets. Otherwise, they won't accept the results of the testing. This is equally true if you are testing in-house or providing a testing service to a third party.

Application performance testing should be an integral part of an internal strategy for application life cycle management. Performance testing has traditionally been an overlooked or last-minute activity, and this has worked against promoting consensus on what it delivers to the business.

Strategies for gaining consensus on performance testing projects within an organization should center on promoting a culture of consultation and involvement. You should get interested parties involved in the project at an early stage so that everyone has a clear idea of the process and the deliverables. This includes the following groups or individuals:

The business

- Chief information officer (CIO)
- Chief technology officer (CTO)
- Chief financial officer (CFO)
- Departmental heads

Remember that you may have to build a business case to justify purchase of an automated performance testing solution and construction of an (expensive) test environment. So it's a good idea to involve the people who make business-level decisions and manage mission-critical business services (and sign the checks!).

The rest

- The developers
- The testers (internal or outsourced)
- The infrastructure team(s)
- The end users

All these groups are intimately involved in the application, so they need to be clear on expectations and responsibility. The developers, whether in-house or external, put the application together; hence there needs to be a clear path back to them should application related problems occur during performance testing execution.

The testers are the people at the sharp end, so among other considerations they need to know the correct transactions to script, the type of test design required, and the performance targets they are aiming for.

Just as important are the people who look after the IT infrastructure. They need to be aware well in advance of what's coming to ensure that there is enough server and network capacity and that the application is correctly configured and deployed.

Last but not least are the end users. It is vital that those who will use the application have input into setting the performance targets. End users will (or should have been) involved in the application design process and may have been involved in User Acceptance Testing (UAT). If they have been using a legacy application that the application to be tested will replace, then they will have a clear and informed view of what will be considered acceptable performance.

Key Performance Targets

Moving onto specifics, I would argue that three performance targets apply to any performance test. These are based on the service-oriented performance indicators that we have already discussed briefly:

- Availability or uptime
- Concurrency, scalability, and throughput
- Response time

To these can be added the following, which are as much a measure of capacity as of performance:

- Network utilization
- Server utilization

Availability or Uptime

This requirement is simple enough: the application must be available to the end user at all times, except during planned maintenance. It must not fail within the target level of concurrency or throughput.

Actually testing for availability is a matter of degree. A successful ping of the Web server's physical machine doesn't necessarily mean that the application is available. Likewise, just because you can connect to the web server from a client browser doesn't mean that the application's home page is available. Finally, the application may be available at modest loads but may start to time out or return errors as the load increases, which (assuming no application problems) would suggest a lack of capacity for the load being generated.

Concurrency, Scalability, and Throughput

Concurrency is probably the least understood performance target. Customers will often quote some number of concurrent users that the application must support without giving enough thought to what this actually entails.

In his excellent white paper "Get performance requirements right: Think like a user," Scott Barber suggests that calculating concurrency based on a hourly figure is a more realistic and straightforward way of doing things. Concurrency from the perspective of a performance testing tool is the number of active users generated by the software, which is not necessarily the same as the number of users concurrently accessing the application. Being able to provide an accurate level of concurrency depends very much on the design of your transaction scripts, discussed later in this chapter.

Scott also makes the point that from concurrency and *scalability* we derive capacity goals. I read this as meaning that achieving our scalability targets demonstrates sufficient capacity in the application landscape for the application to deliver to the business. Yet equally important is finding out just how much extra capacity is available.

In performance testing terms, "concurrency" refers to the following two distinct areas.

Concurrent virtual users
: The number of active virtual users *from the point of view of your performance testing tool.* This number is often very different from the number of virtual users actually accessing the application under test.

Concurrent application users
: The number of virtual users concurrently accessing the application. By "accessing" I mean actually logged in to the application under test. This number is the key measure of how many virtual users are active at a given moment in time. It depends to a large degree on how you design your perfomance testing transactions (discussed later in the chapter). For now you need to decide if the login and logout process—whatever that involves—will be part of the application activity that is to be tested.

If you do include login-logout then there will be a recurring point for every virtual user during test execution where they are logged out of the application and therefore not truly concurrent with other users. Achieving a certain number of concurrent virtual users is a balancing act between the time it takes for a transaction iteration to complete and the amount of pacing (or delay) you apply between transaction iterations (see "Pacing" later in the chapter). This is also called *transaction throughput.*

An example of this situation occurred in a recent project I was involved with. The customer involved wanted to test to 1,000 concurrent users. But because each user was logging in, completing an application activity, and then logging out, there were never more than a couple of hundred users actively using the application at any point during the performance test. This was despite the testing tool indicating 1,000 active users.

The solution was (1) to increase the execution time or persistence of each transaction iteration so that the user involved remained active for longer, and (2) to apply sufficient pacing between transaction executions to slow down the rate of execution. At the same time, it was important to ensure that the rate of transaction throughput achieved in the performance test was an accurate reflection of actual application behavior.

If you do not intend to include login-logout as part of transaction iteration then concurrency becomes a whole lot easier to deal with, because every user that successfully logs in remains active (errors aside) for the duration of the performance test.

The concurrency target quoted will either be an extrapolation of data from an existing application or, in the case of a new application, will probably be based on a percentage of the expected total user community.

Like many things in business, the 80/20 rule often applies: out of a total user community of 100, an average of 20 users will be using the application at any time during the working day. That said, every application will be different, and it's better to go for the high end of the scale rather than the low end. It is just as important that you look beyond deployment to determine what the concurrency will be 6 months, 12 months, or even 18 months later.

You must also include allowances for usage peaks outside of normal limits. Take, for example, an application that provides services to the students and staff of a large university. Day-to-day usage is relatively flat, but at certain times of the year—such as during student enrollment or when examination results are published online—concurrent usage increases significantly. Failure to allow for such usage peaks is a common oversight that degrades performance or, even worse, results in system failure.

In certain circumstances *throughput* rather than concurrency is the performance testing target. This often occurs with applications that are considered stateless; that is to say, there is no concept of the traditional logged-in application user. Casual browsing of any web site's home page would fall into this category. In these situations, the specific number of users who are simultaneously accessing the site is less important than the number of accesses or "hits" in a given time frame. This kind of performance is typically measured per minute or per second.

Another variation is to define a set of throughput requirements based on different end-user skill levels. How many will be at expert or novice level, and what percentage of each should be included in the transaction allocation? Ten expert users will in most cases create more transactions in a given time frame than ten novice users, assuming the same activity.

> **TIP**
>
> **A good rule of thumb is to add 10 percent to your anticipated go live concurrency or throughput target; this way, you will be testing beyond predicted requirements.**

Response Time

Upon reaching the target concurrency (or throughput), the application must allow the end users to carry out their tasks in a timely fashion.

As we discussed in Chapter 1, from an end-user perspective good response time is very much a matter of perception; for example, an average response time of 60 seconds for a call-center application may be perfectly acceptable, whereas the same response for a critical stock trading application would render it completely unusable. It can also be that the time actually taken to complete an activity such as a stock trade is more important than a visual indication of success appearing on the end user's PC.

This is often the challenge when migrating from a legacy green-screen application to a web-enabled replacement that is feature-rich but cannot provide the same kind of instant response. Thoughtful application design, making use of asynchronous functionality where appropriate, can go a long way toward convincing application users that the new version is a winner.

> **NOTE**
>
> **In programming, asynchronous events are those occurring independently of the main program flow. Asynchronous actions are those executed in a nonblocking scheme, allowing the main program flow to continue processing. The effect on the application users is that they don't have to wait for something to complete before they can get on with a new activity.**

If there is an existing system that the application under test will replace, then there may be some performance statistics on current functionality to provide guidance on what is an acceptable response time for key application tasks. Absent this, your next resort is to use "baselining." This is where a single user—running a single transaction for a set period of time with no other activity occurring in the test environment—can be used to provide a "best possible" response-time figure, assuming no significant application design problems.

Assuming this baseline value is acceptable, it will be the amount of variance from this figure at target concurrency (or throughput) that determines success or failure. For example, you

may baseline the time it takes to raise a purchase order as 75 seconds. If this response time increases to an average of 500 seconds when 1,000 users are active, then this may be considered unacceptable. However, if the increase is only to 250 seconds then this may be a perfectly acceptable result to maintain end-user productivity.

The bottom line is that there will always be some increase in response time when users are added, but the increase should not be in lock-step with the additional load.

Network Utilization

Every application presents data to the network. Just how much of an impact this has on performance will depend on the available bandwidth between servers and the end user. Within a modern data center, there is little chance of exhausting available bandwidth (thus, in-house testing is not likely to turn up a performance problem here). However, as you move progressively closer to the user, performance can change dramatically—especially when communication involves the Internet.

High data presentation rates with large numbers of network conversations will typically have the strongest impact on the transmission path with the lowest bandwidth. So the data presented by our application may have to stay below a predetermined threshold, particularly for the "last mile" connection to the Internet user.

In summary, the typical network metrics that you should measure when performance testing include the following.

Data volume
: The amount of data presented to the network. As discussed, this is particularly important when the application will have end users connecting over low-bandwidth WAN links. High data volume, when combined with bandwidth restrictions and network latency effects, does not usually yield good performance.

Data throughput
: The rate that data is presented to the network. You may have a certain number of bytes per second as a performance target. Monitoring data throughput will let you know if this rate is being achieved or if any "throttling back" of throughput is occurring. Often a sudden reduction in data throughput is the first symptom of capacity problems, where the servers cannot keep up with the number of requests being made and virtual users start to suffer from server time-outs.

Data error rate
: Large numbers of network errors that require retransmission of data will slow down throughput and degrade application performance.

Server Utilization

In a similar fashion, there may be fixed limits on the amount of server resource that an application is allowed to use. This information can be determined only by monitoring KPIs while the servers are under load. Obtaining this information relies on appropriately set server KPIs, which are discussed later in this chapter. There are many server performance metrics that can be monitored, but in terms of performance targets the most common are:

- CPU utilization
- Memory utilization
- Disk input/output
- Disk space

In an ideal world we would own the entire application landscape, but in real life the servers or network are often hosted by external vendors. This forces us to reassess the scope of performance targets that can be realistically set.

You can probably test the data center comprehensively, but you often can't do so for the environment outside of the data center, where application traffic may pass through the hands of multiple Internet service providers (ISPs) before reaching the end user. You have little real control over how the performance of your application will be affected by events in cyberspace, but you can ensure that there are no performance bottlenecks within the data center.

> NOTE
> **It is possible to enter into performance agreements with ISPs, but in my experience these are very difficult to enforce and will only cover a single ISP's area of responsibility.**

In these situations, often the best you can aim for is to ensure your application is optimized to be as "WAN friendly" as possible by minimizing the negative effects of bandwidth restriction, latency, and network congestion.

Making Sure Your Application Is Stable Enough for Performance Testing

Having provided a test environment and set performance targets, you need to confirm that your application is stable enough for performance testing. This may seem like stating the obvious, but all too often performance testing morphs into a frustrating bug-fixing exercise, with the time allocated to the project dwindling rapidly.

Stability is confidence that an application does what it says on the box. If you want to create a purchase order, this promise should be successful every time, not 8 times out of 10. If there are significant problems with application functionality, then there is little point in proceeding

with performance testing because these problems will likely mask any that are the result of load and stress.

It goes almost without saying that code quality is paramount to good performance. You need to have an effective unit and a functional test strategy in place.

I can recall being part of a project to test the performance of an insurance application for a customer in Dublin, Ireland. The customer was adamant that the application had passed unit/regression testing with flying colors and was ready to performance test. A quick check of the database revealed a stored procedure with an execution time approaching 60 minutes for a single iteration!

I couldn't believe that the parts of the application that made use of this stored procedure were adequately tested from a functional perspective. There is simply no way that 60 minutes could be considered an acceptable execution time! This is an extreme example, but it serves to illustrate my point.

There are tools available (see Appendix C) that allow you to assess the suitability of your application to proceed with performance testing. Highlighted are the following areas that may hide problems.

High data presentation
: Your application may be functionally stable but have a high network data presentation due to coding or design inefficiencies. If your application's intended users have limited bandwidth then such behavior will have a negative impact on performance. Excessive data may be due to large image files within a web page or large numbers of redundant conversations between client and server.

Poorly performing SQL
: If your application makes use of an SQL database then there may be SQL calls or database stored procedures that are badly coded or configured. These need to be identified and corrected before proceeding with performance testing; otherwise, their effect on performance will only be magnified by increasing load (see Figure 2-2).

Large numbers of application network round trips
: Another manifestation of poor application design are large numbers of conversations or "turns" between application tiers. High numbers of turns make an application vulnerable to the effects of latency, bandwidth restriction, and network congestion. The result is performance problems in this sort of network condition.

Undetected application errors
: Although the application may be working successfully from a functional perspective, there may be errors occurring that are not apparent to the users (or developers). These errors may be creating inefficiencies that affect scalability and performance. An example is an HTTP 404 error in response to a nonexistent or missing web page element. Several of these in a single transaction may not be a problem, but when multiplied by several thousand transactions per minute the impact on performance could be severe.

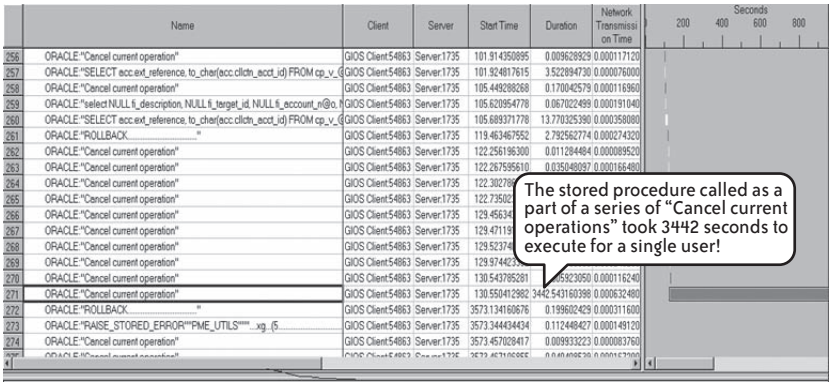


FIGURE 2-2. Example of (dramatically) bad SQL performance

Obtaining a Code Freeze

There's little point in performance testing a moving target. It is absolutely vital to carry out performance testing against a consistent release of code. If you find problems during performance testing that require a code change, that's fine, but make sure the developers aren't moving the goalposts between test cycles without good reason. If they do, make sure somebody tells the testing team!

As mentioned previously, automated performance testing relies on scripted transactions that are a recording of real user activity. These scripts are version dependent; that is, they represent a series of requests and expected responses based on the state of the application *at the time* they were created. An unanticipated new release of code may partially or completely invalidate these scripts, requiring in the worst case that they be completely recreated. More subtle effects may be an apparently successful execution but an invalid set of performance test results because code changes have rendered the scripts no longer an accurate representation of end-user activity.

I have seen many cases where this problem has an immense impact on performance testing projects. It's often the result of a communication breakdown between test teams and management.

Identifying and Scripting the Business-Critical Transactions

The core of any performance test is the transactions that drive the load. A simple example of a transaction is logging in to the application, navigating to the search page, executing a search, and then logging out again. You need to determine the high-volume, mission-critical activities

that an average user will carry out during a normal working day. These will form the basis of all your performance tests, so you must be sure that you have identified them correctly.

Don't confuse the key activities in performance testing with functional testing use cases. Remember that your aim is not to test the application functionality (you should have already done that) but to create a realistic load to stress the application and then assess its response behavior from a performance perspective. This should reveal any problems that are a result of concurrency or a lack of adequate capacity.

Transaction Checklist

For each transaction you should complete the following actions.

Define and document each execution step so that there is no ambiguity in the navigation path
: Each step must be clearly defined with appropriate inputs and outputs. This is important not only for generating accurate scripts but also for defining the parts of each transaction that will be timed during performance test execution. (See Figure 2-3 for an example of a well-documented transaction.)

Identify all input data requirements and expected responses
: This is the information that needs to entered by the user at each navigation step and is a critical part of defining the transaction. Examples include logging in, entering a password, and—significantly—what should happen in response to these activities.

Determine the type of user that the transaction involves
: Examples include supervisor, customer, call-center operator, and team leader. Deciding on the type of user will usually determine whether this is a high- or low-volume transaction. For example, there are more call-center operatives than supervisors, so supervisor users will likely limit themselves to background administration tasks whereas the operators will be dealing with large numbers of customer enquiries.

What is the connectivity path for this transaction?
: This could be via the LAN, the WAN, the Internet, or perhaps a combination of these methods. As we have discussed, a transaction that performs well in a LAN environment may not do so when deployed in a low-bandwidth WAN environment.

Will this be an active or passive transaction?
: Active transactions represent the user-visible activities that you want to perform well, whereas passive transactions are background activities included just to provide the test with a realistic load. For example, there may be a requirement to include a proportion of users who are logged in but not actively using the application. The transaction definition for this sort of user may simply be logging in, doing nothing for a period of time, and then logging out—a passive transaction.

TIP

Don't worry about this step, even if your application is considered complex and rich in features. In my experience, the number of transactions that are finally selected rarely exceeds twenty, pretty much regardless of what the application does or how it is architectured. If you should find yourself with a larger-than-expected number of transactions, it will be necessary to carry out a risk assessment to determine those activities that must be tested in the time available for performance testing.

Transaction Replay Validation

Once you have identified the transactions, you must convert them to some form of *script* using your performance testing tool. This is typically achieved by putting the testing tool into "record" mode and then carrying out the activity to be scripted. At the end of the recording process, the tool should automatically generate a script file representing your transaction activity.

Once the script has been created you will need to prep it by addressing any data requirements (discussed in the next section) and making any other changes required for the script to replay correctly. Upon completing this task, your final step is to ensure that the script actually will replay correctly. The following considerations should be taken into account as part of this validation process.

Verify single user replay
: Always check that each script replays correctly using whatever validation process is provided by your performance testing tool. Obvious enough, you might think, but it is easy to wrongly assume that—because no obvious errors are present—your script has worked OK. Your performance testing tool should make it easy to verify successful replay by providing a trace of every client request and server response. On many occasions, a performance testing project has been completed only to find that one or more transactions were not replaying correctly, severely compromising the results.

Verify multiuser replay
: Once you are satisfied that the scripts are replaying correctly, you should always try a small multiuser replay to ensure that there are no problems that relate to concurrent execution. You don't want to discover in your first performance test that something you missed during the scripting process prevents your test from running.

What to Measure?

As just discussed, after identifying the key transactions you need to record and script them using your performance tool of choice. As part of this process you must decide which parts of the transaction to measure in terms of response time.

You can indicate areas of interest within a transaction simply by inserting comments, but most performance testing tools will allow you to "checkpoint" parts of a transaction by inserting *begin* and *end* markers around individual or groups of requests.

FIGURE 2-3. Example of checkpoints within a transaction script, in this case APP_SVR_TESTING

When the scripts are eventually used in a performance test, these checkpoints provide better response-time granularity than does the response time for the entire transaction. For example, your may choose to checkpoint the login process or perhaps one or more search activities within the script. Simply replaying the whole transaction without this sort of breakdown will make it much harder to identify problem areas.

Checkpoints are really your first port of call when analyzing the results of a performance test, because they provide initial insight into any problems that may be present. For example, your total average response time for raising a purchase order may be 30 seconds but analysis of your checkpoints shows that *Log into App Server* was taking 25 of the 30 seconds.

To Log In or Not to Log In

Recall from our previous discussion of concurrency that the transactions you script should reflect how the application will be used on a day-to-day basis. An important consideration here is the end-user *usage profile*. By this I mean whether users log in, complete an activity, and log out or more typically log in once and remain logged in to the application during the working day.

Logging in to an application is often a load-intensive activity, so if users log in just a few times during a working day then it is unrealistic to include this step in every script iteration. Most performance testing tools allow you to specify which parts of a transaction are repeated during test execution.

Remember that including or excluding the login-logout process in your transaction scripts will have a significant impact on achieving your target virtual user concurrency.

Peaceful Coexistence

Something else that is important to consider is whether your application will exist in isolation (very unlikely) or have to share resource with other applications.

An application may perform magnificently on its own but fall down in a heap when it must coexist with others. Common examples include web traffic and email when used together or with other core company applications.

It may be that your application has its own dedicated servers but must still share network bandwidth. Simulating network effects may be as simple as factoring in enough additional traffic to approximate current bandwidth availability during testing. Creating an additional load where applications share mid-tier and database servers will be more complex, ideally requiring the generation of load for other key applications to be included in performance testing.

This means that you may need to identify and script transactions from other applications in order to provide a level of "background noise" when executing performance tests.

Providing Sufficient Test Data of High Quality

OK, you have your performance targets and you have your transactions; the next thing to consider is *data*. The importance of providing enough quality test data cannot be overstated. It would be true to say that performance testing lives and dies on the quality and quantity of the test data provided. It is a rare performance test that does not require any data to be provided as input to the scripted transactions.

Creating even a moderate amount of test data is a nontrivial task. Most automated test tools take a file in comma-delimited values (CSV) format as input to their scripts, so you can potentially use any program that can create a file in this format to make data creation less painful. Common examples are MS Excel and Compuware Corporation's FileAid Client/Server.

Three types of test data are critical: input data, target data, and runtime data.

Input Data

This is data that will be provided as input to your transactions. You need to look at exactly what is required, how much of it you require, and—significantly—how much work is required to create it. If you have allocated two weeks for performance testing and it will take a month to produce enough test data, you need to think again!

Some typical examples of input data are as follows.

User credentials
: For applications that are designed around user sessions, this data would typically consist of a login ID and password. Many performance tests are executed with a limited number of test user credentials. This introduces an element of risk regarding multiple users with the same login credentials being active simultaneously, which can lead to misleading results and execution errors. Whenever possible, you should provide unique login credentials for every virtual user to be included in a performance test.

Search criteria

There will almost always be transactions in any performance test that are designed to carry out various kinds of searches. In order to make these searches realistic, you must provide a variety of data that will form the search criteria. Typical examples include customer name and address details, invoice numbers, and product codes. You may also want to carry out wildcard searches where only a certain number of leading characters are provided as a search key. Wildcard searches typically take longer to execute and return more data to the client. If the application allows the end user to search for all customer surnames that begin with "A," then you need to include this information in your test data.

Associated documents

For certain types of performance tests it may be a necessary to associate documents with transactions. This is a common requirement with document management systems, where the time to upload and download document content is a critical indicator of performance. These documents may be in a variety of formats (e.g., PDF, MS Word), so you need to ensure that sufficient numbers of the correct size and type are available.

Target Data

What about the target database? (It's rare not to have one.) This needs to be populated with realistic volumes of valid data so that your inquiries are asking the database engine to perform realistic searches. If your test database is 50 MB and the live database is 50 GB, this is a surefire recipe for misleading results.

So what are the challenges when trying to create and manage a test database?

Sizing

It is important to ensure that you have a realistically sized test database. Significantly smaller amounts of data than will be present at deployment provide the potential for misleading database response times, so this option should be considered only as a last resort. Often it's possible to use a segment of an existing live database, which has the added benefit of being real rather than test data. However, for new applications this won't normally be possible and the database will have to be populated to realistic levels in advance of any testing.

Data rollback

If any of the performance tests that you run change the content of the data within the test database then ideally—prior to each performance test execution—the database should be restored to the same state it was before the start of the first performance test. It's all about minimizing the differences between different test runs so that comparisons between sets of results can be carried out with confidence.

You need to have a mechanism in place to accomplish this data rollback in a realistic amount of time. If it takes two hours to restore the database, then this must be factored into the total time allowed for the performance testing project.

Runtime Data

During performance test execution, it is often necessary to intercept and make use of data returned from the application. This data is distinct from that entered by the user. A typical example is information related to the current user session that must be returned as part of every request made by the client. If this information is not provided or is incorrect, the server would return an error or disconnect the session. Most performance testing tools give you some ability to handle this kind of data.

In these situations, if you fail to deal correctly with runtime data then it will usually be pretty obvious, because your scripts will fail to replay. However, there will be cases where a script appears to work but the replay will not be accurate; in this case, your performance testing results will be suspect. *Always verify that your scripts are working correctly before using them in a performance test.*

Data Security

Its all very well getting your hands on a suitable test database, but you must also consider the confidentiality of information contained within. It may be necessary to depersonalize details such as names, addresses, and bank account numbers in order to prevent personal security from being compromised by someone casually browsing through test data. Given the current climate of rampant identity fraud, this is a common condition of use that must be addressed. Data management tools are available that can greatly simplify the extraction and safe rendering of sensitive data (see Appendix C).

Ensuring Accurate Performance Test Design

Once you've identified the key transactions and their data requirements, the next step is to use your list to create a number of different types of performance tests. The final choice will largely be determined by the nature of the application and how much time is available for performance testing. The following testing terms are generally well known in the industry, although there is often confusion over what they actually mean.

Baseline test
: This establishes a point of comparison for further test runs, typically for measuring transaction response time. The test is normally executed for a single transaction as a single virtual user for a set period of time or for a set number of transaction iterations. This execution should be carried out without any other activity on the system to provide a "best case" measurement. The value obtained can then be used to determine the amount of performance degradation that occurs in response to increasing numbers of users or throughput.

Load test

This is the classic performance test, where the application is loaded up to the target concurrency but usually no further. The aim is to meet performance targets for availability, concurrency or throughput, and response time. Load testing is the closest approximation of real application use, and it normally includes a simulation of the effects of user interaction with the application client. These include the delays and pauses experienced during data entry as well as (human) responses to information returned from the application servers.

Stress test

This has quite a different aim from a load test. A stress test causes the application or some part of the supporting infrastructure to fail. The purpose is to determine the upper limits or sizing of the infrastructure. Thus, a stress test continues until something breaks: no more users can log in, response time exceeds the value you defined as acceptable, or the application becomes unavailable. The rationale for stress testing is that if our target concurrency is 1,000 users but the infrastructure fails at only 1,005 users, then this is worth knowing because it clearly demonstrates that there is very little extra capacity available.

The results of stress testing measure capacity as much as performance. It's important to know your upper limits, particularly if future growth of application traffic is hard to predict. For example, the scenario just described would be disastrous for something like an airport air traffic control system, where downtime is not an option.

Soak or stability test

The soak test is intended to identify problems that may appear only after an extended period of time. A classic example would be a slowly developing memory leak or some unforeseen limitation in the number of times that a transaction can be executed. This sort of test cannot be carried out effectively unless appropriate server monitoring is in place. Problems of this sort will typically manifest themselves either as a gradual slowdown in response time or as a sudden loss of availability of the application. Correlation of data from the users and servers at the point of failure or perceived slowdown is vital to ensure accurate diagnosis.

Smoke test

The definition of smoke testing is to focus only on what has changed. Therefore, a performance smoke test may involve only those transactions that have been affected by a code change.

NOTE

The term smoke testing originated in the hardware industry. The term derived from this practice: after a piece of hardware or a hardware component was changed or repaired, the equipment was simply powered up. If there was no smoke (or flames!), the component passed the test.

Isolation test

This variety of test is used to home in on an identified problem. It usually consists of repeated executions of specific transactions that have been identified as resulting in a performance issue.

I believe that you should always execute a baseline, load, and stress test. The soak test and smoke test are more dependent on the application and the amount of time available to testing—as is the requirement for isolation testing, which will largely be determined by what problems are discovered.

Whatever type of test you decide on, you need to consider the topics discussed in the next six subsections.

Pacing

Think Time And Think time represents the delays and pauses that any user will introduce while interacting with an application. Examples include pausing to speak to a customer during data entry and deciding how much to bid for an item on eBay. The important thing about accounting for think time is that it introduces an element of realism into the test execution. With think time removed, as is often the case in stress testing, execution speed and throughput can increase tenfold, rapidly bringing an application infrastructure that can comfortably deal with a thousand real users to its knees. Therefore, always include think time in a load test.

Pacing is another way of affecting the execution of a performance test. Whereas think time influences the rate of transaction execution, pacing affects transaction throughput. This is an important difference that I'll take a moment to explain.

Consider this example. Even though an online banking application has 6,000 customers, we know that the number of people who choose to view their statements is never more than 100 per hour even at the busiest time of day. Therefore, if our load test results in 1,000 views per hour then it is not a realistic test of this activity.

We might want to increase the rate of transaction execution as part of a stress test, but not for load testing. As usual, our goal is to make the performance test as true to life as we can. By inserting a pacing value we can reduce the number of iterations and, by definition, the transaction throughput.

For example, if a typical "mini-statement" transaction takes 60 seconds to execute, then applying a pacing value of 2 minutes will limit the maximum number of transactions of this type per hour to 30 per virtual user. This is accomplished by making each virtual user wait 2 minutes before beginning the next iteration. In other words, the amount of time between each execution of "mini-statement" cannot be less than 2 minutes.

You might also consider varying the amount of pacing that you apply in a load test to increase realism. For example, you could randomly apply plus or minus 10 percent to the 2-minute

value in our example for each iteration. Some automated performance tools provide this capability.

Injection Profile

Next decide on how you will inject the application load. There are five common types of injection profile that are typically used in combination within a performance test.

Big Bang
: This is where all virtual users start at the same time. Typically these users will run concurrently but not synchronously; that is, they will not be doing exactly the same thing at the same moment during test execution. This provides a better simulation of real user behavior.

Ramp-up
: This mode begins with a set number of virtual users (typically 0 or 1) and then adds more users at specified time intervals until a target number is reached. This is the usual way of testing to see if an application can support a certain number of concurrent users.

Ramp-up (with step)
: In this variation of straight ramp-up, there is a final target concurrency or throughput, but the intention is to pause injection at set points during test execution. For example, the target concurrency may be 1,000 users, but there is a need to observe the steady-state response time at 250, 500, and 750 concurrent users; so, upon reaching each of these values, injection is paused for a period of time. Not all automated performance test tools provide this capability within a single performance test. But it is a simple matter to configure separate tests for each of the step values and so mimic the same behavior.

Ramp up (with step), Ramp down (with step)
: Building on the previous profile you may want to ramp up to a certain number of concurrent users (with or without steps) and then gradually reduce the number of virtual users(with or without steps) back to zero, which may signal the end of the test. Again, not all automated performance test tools provide this capability within a single performance test.

Delayed start
: This may be combined with any of the preceding injection profiles; it simply involves delaying the start of injection for a period of time for a particular script deployment.

If you choose the ramp-up approach, aim to build up to your target concurrency for each script deployment in a realistic period of time. You should prorate the injection rate for each transaction to reach full load simultaneously. Remember that the longer you take ramping up, the less time you will have to performance test.

For Big Bang load injection, caution is advised because large numbers of virtual users starting together can create a fearsome load, particularly on web servers. This may lead to system failure before the performance test has even had a chance to start properly.

For example, many client server applications have a user login period at the start of the working day that may encompass an hour or more. Replicating 1,000 users logging in at the same moment when this would actually be spread over an hour is unrealistic and increases the chance of system failure by creating an artificial period of extreme loading. This sort of injection profile choice is not recommended unless it is intended to form part of a stress test.

Setting the Number of Virtual Users per Transaction

Decide on the number of virtual users to allocate to each transaction deployment per test. Most automated performance test tools allow you to deploy the same transaction multiple times with different numbers of virtual users and different injection profiles. In fact, this would be an extremely good idea for an application that has significant numbers of users connecting over differing WAN environments. Each deployment could simulate a different WAN bandwidth, allowing you to observe the effects of bandwidth reduction on application performance under load.

You should apportion the number of users per transaction to represent the type of transaction variety and behavior that would occur during a normal working day. Where possible, make use of any existing usage data to ensure that your user allocation is as accurate as possible. For new applications, the allocation can be based only on expectation and thus may need refinement during test execution.

Table 2-1 provides an example of ten transactions distributed among a total application concurrency of 1,000 virtual users.

TABLE 2-1. Virtual user transaction allocation

Transaction	Allocated virtual users
MiniStatement	416
CustomStatement	69
ViewStandingOrders	69
TextAlertCreateCancel	13
TransferBetweenOwnAccounts	145
BillPaymentCreate	225
BusMiniStatement	33
BusCustomStatement	4
BusTransferBetweenOwnAccounts	10

Transaction	Allocated virtual users
BusBillPaymentCreate	16

Deciding on Performance Test Types

I tend to structure my performance tests in the following format, although the ideal order may differ based on your own requirements.

Baseline test each transaction

I like to establish a baseline set of performance data for each transaction before starting any kind of load testing. This provides an indication of the best performance you're likely to get from whatever activity the transaction represents. You should run this sort of test for a set period of time or a number of iterations. There should be no other application activity going on at the same time; otherwise, your results will be skewed.

Load test each transaction

Decide on the maximum number of concurrent users or throughput you want for each transaction; then run a separate test for each transaction up to the limit you set. This will provide a good indication of whether there are any concurrency-related issues on a per-transaction basis. Depending on the number of concurrent users, you may want to use a straight ramp-up or the ramp-up with step injection profile. I tend to use the ramp-up with step approach, since this allows fine-grain analysis of problems that may occur at a particular level of concurrency or throughput.

Isolation test individual transactions

If problems do occur then isolation testing is called for until the problems are resolved.

Load test transaction groups

Once you've tested each transaction in isolation, the next step is normally to combine all transactions in a single load test. This then tests for problems with infrastructure capacity or possible conflicts between transactions such as database lock contention. Once again, you may decide to use the ramp-up or ramp-up with step injection profile.

Isolation test transaction groups

As with individual transaction performance tests, any problems you find may require running isolation tests to confirm diagnoses and point to solutions.

Soak test transaction groups

Repeat the individual transaction and/or transaction group performance tests for an extended duration. This should reveal any problems that manifest themselves only after a certain amount of application activity has occurred.

Stress test transaction groups

Repeat the individual transaction and/or transaction group tests but reduce the amount of sleep time and pacing to create a higher throughput than was achieved during the load

tests. This allows you to ascertain the upper capacity limits of the application landscape and to determine how the application responds to sudden peaks of activity.

Non-performance tests
: Finally, carry out any tests that are not specifically related to load or stress. These may include testing different methods of load balancing or perhaps testing failover behavior by disabling one or more servers in the application landscape.

Load Injection Point of Presence

Having identified the key transactions and designed the performance test content, you next have to decide from where you will inject the load. This is an important consideration, because you need to ensure that you are not introducing artificially high data presentation into areas where this simply wouldn't happen in the live environment. For example, trying to inject the equivalent of 100 users worth of traffic down a 512 kb ADSL link that would normally support a single user is completely unrealistic—you will simply run out of bandwidth. Even if you were successful, the resulting performance data would bear no relationship to the real end-user experience (unless it is anticipated that 100 users may someday share this environment).

If you consider how users connect to most applications, there is a clear analogy with a system of roads. The last connection to the end-user is the equivalent of a country lane that slowly builds into main roads, highways, and finally freeways with progressively more and more traffic. The "freeway" connection is the Internet or corporate LAN pipe to your data center, so this is where the high-volume load needs to be injected.

In other words, you must match the load you create at any given point in your test infrastructure with the amount of load that would be present in a real environment. If you don't do this, then you run the risk of bandwidth constraints limiting the potential throughput and number of virtual users. The result is a misleading set of perfomance data.

Putting It All Together

After taking into consideration all the points discussed so far, you should have a clear idea of how your performance tests will look. Figure 2-4 provides an example of a comprehensive set of data taken from an actual performance test document. I'm not saying that you will always need to provide this level of detail, but it's an excellent example to which we can aspire.

Observe that there is a section in the figure for baseline and a separate section for simulated load. The application was already supporting 200 virtual users without issue, and data was available to determine the transaction rate per day and per hour for the existing load. This is the information presented in the baseline section and used as the first target for the performance test.

The simulated load section specified the final target load for the performance test, which was 500 virtual users. An important part of achieving this increased load was ensuring that the

Ref No	Transaction	Duration	Transaction Rate as per baseline		Calculated Users as per baseline			Simulated Load (Load factors applied)			
			tpd	tph	Min VU	Dist (%)	VU	tph	VU	Ramp-up (hh:mm:ss)	Pacing (mm:ss.ms)
1.7	Movement Continuation	25.00	2000	363.64	2.53	18.43	37	1,181.82	93	0:00:17	04:41.8
11.1.10	Customer Account Enquiry - Personal	21.00	2000	363.64	2.12	15.48	31	1,181.82	78	0:00:20	03:56.1
7.2	Approval Process	15.00	1900	345.45	1.44	10.50	21	1,122.73	53	0:00:32	02:48.3
2.3	Enquire & Amend on a Partner Record -	20.00	1000	181.82	1.01	7.37	15	590.91	38	0:00:43	03:48.5
11.1.10.1	Customer Account Enquiry - Group	21.00	1000	181.82	1.06	7.74	15	590.91	38	0:00:43	03:48.5
4.5	Add members to a group scheme	34.00	600	109.09	1.03	7.52	15	354.55	38	0:00:39	06:20.8
11.3.1	Process Receipts (2 payments)	54.00	600	109.09	1.64	11.94	24	354.55	60	0:00:20	10:09.2
5.1	Create & Issue a quote	77.00	400	72.73	1.56	11.35	23	236.36	58	0:00:16	14:35.8
1.8.1	Add interested Party Role	47.00	350	63.64	0.83	6.06	12	206.82	30	0:00:44	08:42.2
1.8.2	Remove Interested Party Role	28.00	350	63.64	0.49	3.61	7	206.82	18	0:01:31	05:04.6
Total			10200	1854.55	13.7	100.0	200	6,027.27	500		

Rampup Interval (hh:mm:ss)	00:30:00	Duration of business day (hrs)	5.5
Start up VU's / transaction	1	Transaction rate load factor (%)	30
Baseline VU's	200	VU load factor (%)	150

Legend

tpd = Transactions per Day
tph = Transactions per Hour
Min VU = Minimum Virtual Users
Dis (%) = Distribution Percentage
VU = Virtual Users

FIGURE 2-4. Example performance test configuration

transaction execution and throughput rates were a realistic extrapolation of current levels. Accordingly, the virtual user (distribution), ramp-up, and pacing values were calculated to achieve this aim.

Here is a line-by-line description of the figure.

- Ref No: Sections in the master document that involve the *Transaction*
- Transaction: Name of the application *Transaction*
- Duration: Amount of time in seconds that the *Transaction* takes to complete
- *Transaction* Rate (per baseline)

  tpd: *Transaction* rate per day

  tph: *Transaction* rate per hour
- Calculated Users (per baseline)

  Min VU: Minimum number of virtual users for this *Transaction*

  Dist %: Distribution of this *Transaction* as a percentage of the total number of virtual users in this test

  VU: Number of virtual users allocated to this *Transaction* based on the Dist %
- Simulated Load (load factors applied):

  tph: Target *Transaction* rate per hour

  VU: Total number of virtual users allocated for this *Transaction*

  Ramp-up (hh:mm:ss): Rate of virtual user injection for this *Transaction*

  Pacing (hh:mm:ss): Rate of pacing for this *Transaction*

Identifying the Server and Network Key Performance Indicators (KPIs)

You need to identify the key server and network performance metrics that should be monitored for your application. This information is vital to achieve accurate root-cause analysis of any problems that may occur during performance test execution. Ideally, the monitoring is integrated with your automated performance test solution. However, a lack of integration is no excuse for failing to address this important requirement.

You are aiming to create a set of monitoring models or templates that can be applied to the servers in each tier. Just what these models comprise will depend on the technology that was used to create and deploy the application.

Server KPIs

Server performance is measured by monitoring software configured to observe the behavior of specific performance metrics or counters. This software may be included or integrated with your automated performance testing tool or it may be an independent product.

Perhaps you are familiar with the Perfmon (Performance Monitor) tool that has been part of Windows for many years. If so, then you are aware of the literally hundreds of performance counters that could be monitored on any given server. From this vast selection there is a core of a dozen or so metrics that can reveal a lot about how a (Windows) server is performing.

In the Unix/Linux world there are long-standing utilities like monitor, top, vmstat, iostat, and SAR that provide the same sort of information. In a similar vein, mainframes have their own monitoring tools that can be employed as part of your performance test design.

It is important to approach server KPI monitoring in a logical fashion—ideally, using a number of layers. The top layer is what I call generic monitoring, which focuses on a small set of counters that will quickly tell you if any server (Windows or Unix) is under stress. The next layer of monitoring should focus on specific technologies that are part of the application landscape: the web, application, and finally database servers.

> TIP
>
> **The ideal approach is to build separate templates of performance metrics for each layer of monitoring. Once created, these templates can form part of a reusable resource for future performance tests.**

The final layer involves the application server (if relevant) and shifts the focus away from counters to component and method level performance. Essentially this is looking inside the application server technology to reveal its contribution to performance problems revealed by generic monitoring. It is clearly impractical to provide lists of suggested metrics for each application technology. Hence you should refer to the documentation provided by the

appropriate technology vendors for guidance on what to monitor. To aid you in this process many performance testing tools provide suggested templates of counters for popular application technologies.

So to summarize, depending on the application architecture, any or all of the following models or templates may be required.

Generic templates

This is a common set of metrics that will apply to every server in the same tier that has the same operating system. Its purpose is to provide first-level monitoring of the effects of load and stress. Typical metrics would include monitoring how busy the CPUs are and how much available memory is present. Appendix D contains examples of generic and other templates. If the application landscape is complex then you will likely have several versions.

In my experience, a generic template for monitoring a Windows server OS should include at a minimum the following counters (which cover the key areas of CPU, memory, disk I/O, and some visibility of the network terms of errors):

- Processor utilization percentage
- Top 10 processes
- Available memory in bytes
- Memory pages/second
- Processor queue length
- Context switches per second
- Physical disk: average disk queue length
- Physical disk: % Disk Time
- Network interface: Packets Received Errors
- Network interface: Packets Outbound Errors

Web and application server tier

These templates focus on a particular application server technology, which may involve performance counters that differ from those provided by Microsoft's Perfmon tool. Instead, this model may refer to the use of monitoring technology to examine the performance of a particular application server such as Oracle, WebLogic, or IBM's WebSphere. This technology gives insight into poorly performing or configured components down to method level. Examples include:

- IIS (Microsoft Internet Information Server)
- Oracle application server (OC4J)
- Oracle WebLogic (JRockit)
- IBM WebSphere
- JBOSS

Database server tier

Enterprise database technologies are provided by a number of vendors. Most are similar in architecture and modus operandi, but differences abound from a monitoring perspective. As a result, each type of database will require its own unique template. Examples familiar to most include:

- Microsoft SQL Server
- Oracle
- IBM DB2
- MySQL
- Sybase
- Informix

Mainframe tier

If there is a mainframe tier in your application deployment, you need to include it in performance monitoring to provide true end-to-end coverage. Mainframe monitoring tends to focus on a small set of metrics based around memory and CPU utilization per job and Logical Partition (LPAR). Some vendors allow integration of mainframe performance data into their performance testing solution. Performance monitoring tools for mainframes tend to be fairly specialized. The m but the actual scripting effort is typically half a day per business case ost common are:

- Strobe from Compuware Corporation
- Candle from IBM

Network KPIs

In performance testing, network monitoring focuses mainly on packet round-trip time, data presentation, and the detection of any errors that may occur as a result of high data volumes. As with server KPI monitoring this capability can be built into an automated performance test tool or it may be provided separately.

If you have followed the guidelines on where to inject the load and have optimized the data presentation of your application, then network issues should prove the least likely cause of problems during performance test execution.

For the Windows and Unix/Linux operating systems there are performance counters that monitor the amount of data being handled by each NIC card as well as the number of errors (both incoming and outgoing) detected during a performance test execution. These counters can be part of the suggested generic template described previously.

To help better differentiate between server and network problems, some automated performance test tools separate server and network time for each element within a web page. See Figure 2-5.

Name	Response Code	Total Requests	Average response time	Average server	Average network	Server / Network
https://OnlineBanking.uk/online/img/arrow.gif	200	2	2.0816	99.9881 %	0.0119 %	
https://OnlineBanking.uk/online/css/advIE.css	200	2	2.9855	31.8739 %	68.1261 %	
http://OnlineBanking.uk/js/loan.js	200	6	0.4469	99.9912 %	0.0088 %	
http://OnlineBanking.uk/img/warning.jpg	200	6	0.0219	99.863 %	0.137 %	
http://OnlineBanking.uk/img/logo.gif	200	6	0.0519	99.9554 %	0.0446 %	
http://OnlineBanking.uk/img/award.gif	200	6	0.1396	99.9756 %	0.0244 %	
http://OnlineBanking.uk/img/small.jpg	200	6	0.4033	99.9864 %	0.0136 %	

FIGURE 2-5. Example server/network response time breakdown

Allocating Enough Time to Performance Test Effectively

Building on the previous points, it is extremely important to allow enough time to performance test an application effectively. This cannot be a "finger in the air" decision and must take into account the following elements.

Lead time to prepare test environment

If you already have a dedicated test environment, this requirement may be minimal. Alternatively, you may have to build the environment from scratch, with the associated time and costs. These include sourcing and configuring the relevant hardware as well as installing and configuring the application into this environment.

Lead time to prepare the injection environment

In addition to the test environment itself, consider also the time needed to prepare the resource required to inject the load. This typically involves a workstation or server to manage the performance testing and multiple workstations/servers to provide the load injection capability.

Time to identify and script business transactions

It is vitally important to identify and script the business cases that will form the basis of your performance testing. Identifying the business cases may take from days to weeks, but the actual scripting effort is typically half a day per transaction.

Time to identify and create enough test data

Because test data is key to a successful performance testing project, you must allow enough time to prepare it. This is often a nontrivial task and may take many days or even weeks. You should also consider how long it may take to reset the target data repository or recreate test data between test executions, if necessary.

Time to instrument the test environment

This covers the time to put in place any monitoring of the application landscape to observe the behavior of servers and the network under load. The actual time required will depend

on the number of servers and network segments to be monitored and the geographics of where they are deployed. (This is more important than you might think; I've been involved in projects where the load injection was based in the United Kingdom and the target servers were located in Miami!)

Time to prepare and execute performance test runs

Usually the shortest part of any performance testing engagement, this is the time to create and execute the actual tests.

Time to deal with any problems identified

This is a significant challenge, since you are dealing with unknowns. However, if sufficient attention has been given to performance during development of the application, the risk of significant performance problems during the testing phase is substantially reduced. That said, you still need to allocate time for resolving any issues that may crop up. This may involve the application developers and code changes.

Summary

In this chapter we have taken a look at essential requirements. In the next chapter we turn our attention to making use of these requirements in order to build an effective application performance testing checklist.

CHAPTER THREE

The Process of Performance Testing

I see plans within plans.

—from Dune *by Frank Herbert*

As discussed in Chapter 1, many performance testing projects come together as a last-minute exercise. I would happily wager that you have been involved in a project that falls into this category. In such cases, you are constrained by limited time frames and pressure to deploy by a certain date, even though the application may have serious undetected performance problems. This chapter describes a performance testing approach to follow so that any new projects you participate in don't suffer from the same pitfalls.

In Chapter 2 my intention was to cover performance testing requirements in a logical but informal way. This chapter is about using these requirements to build a plan: a performance testing checklist divided into logical stages. We'll also look at how this plan can be applied to a couple of case studies based on real projects. Each case study will demonstrate different aspects of the performance test process and will provide some of the examples used in Chapter 4 to demonstrate the interpretation of performance test results.

Each case study features a review of how closely (or not) the performance project aligned itself with the requirements discussed in Chapter 2 and the suggested approach provided in this chapter.

The Proof of Concept (POC)

Before looking at the performance testing process, I want to spend a few moments reiterating why a Proof of Concept (POC) is an important prerequisite to undertaking any sort of performance testing project. Frequently part of a sales cycle, the POC provides the following information.

Provides an opportunity for a technical evaluation of the performance testing tool against the target application

In Chapter 2 we looked at the process of choosing a performance testing tool. Obviously, technical compliance is a key goal, for otherwise you will struggle (or fail!) during the transaction scripting phase. You must try out the tool against the real application so that any problems are exposed and dealt with before committing (or not) to the project.

Identifies scripting data requirements

The typical POC is built around a small number of the transactions that will form the basis of the performance testing project. It is effectively a dress rehearsal for the scripting phase that allows you to identify the input and the runtime data requirements for successful test execution. Because the POC tests only two or three transactions, you'll have additional data requirements for the remaining ones; these need to be identified later in the scripting phase of the project. However, the POC will frequently identify such common test input data requirements as login credentials and stateful information that keeps a user session valid.

Allows assessment of scripting effort

The POC allows you to estimate the amount of time it will take to script a typical application transaction and take into account the time to make alterations to scripts based on results of the POC.

Demonstrates the capabilities of the performance testing solution versus the target application

If you're trying to sell the benefits of automated performance testing tools, there is no better place to do it than during a POC.

Proof of Concept Checklist

The following checklist provides a guide to carrying out an effective POC. In terms of time scales, every POC will have its unique aspects but you can anticipate no more than a couple of days for completion—assuming that the environment and application are available from day one.

Prerequisites

Allocate these early in the project to make sure they're in place by the time you need to set up the POC environment.

- A set of success or exit criteria that you and the customer (or internal department) have agreed on as determining the success or failure of the POC. Have this signed off and in writing before starting the POC.
- Access to a standard build workstation or client platform that meets the minimum hardware and software specification for your performance testing tool or solution. This machine must have the application client *and* any supporting software installed.
- Permission to install any monitoring software that may be required (such as server and network monitors) into the application landscape.
- Ideally, sole access to the application for the duration of the POC.
- Access to an person who is familiar with the application (i.e, a "power user") and can answer your usability questions as they arise.
- Access to an expert who is familiar with the application (i.e., a developer) for times when you come up against technical difficulties or require an explanation of how the application architecture works at a middleware level.
- A user account that will allow correct installation of the performance testing software onto the standard build workstation and access to the application client.
- At least two sets of login credentials (if relevant) for the target application. The reason for having two is that many operations behave differently when done by one user and by more than one. Therefore, you need to be able to run the POC with multiple users.
- Two sample transactions to use as a basis for the POC. One transaction should be a simple "read-only" operation, and the other should be a complex transaction that updates the target data repository. These let you check that your transaction replay works correctly.

Process

This list helps you make sure the POC provides a firm basis for the later test.

- Record two instances of each sample transaction and compare the differences between them using whatever method is most expedient. (Windiff is one possibility, although there are better alternatives.) Your performance testing tool may provide this capability. Identifying what has changed between recordings of the same activity should highlight any runtime data requirements that need to be addressed.
- After identifying input and runtime data requirements as well as any necessary modifications to scripts, ensure that each transaction will replay correctly in single user *and* multiuser mode. Make sure that any database updates occur as expected and that there are no errors in the replay logs for your transactions. In the POC (and the subsequent project), make certain that any modifications you have made to the scripts are free from memory leaks and other undesirable behavior.

NOTE

Windiff.exe is a Windows utility program that allows you to identify the differences between two files; it's free and has been part of Windows for some time. If you need greater functionality, try these tools:

- ExamDiff Pro from prestoSoft (*http://www.prestosoft.com/*)
- WinMerge (*http://winmerge.org/*)

Deliverables

These are the results of the POC and the basis for approaching any subsequent project with confidence:

- The output of a POC should be a go/no-go assessment of the technical suitability of your performance testing tool for successfully scripting and replaying application transactions.
- You should have identified the input and runtime data requirements for the sample transactions and gained insight into the likely data requirements for the performance testing project.
- You should identify any script modifications required to ensure accurate replay and assess the typical time required to script an application transaction.
- If this is part of a sales cycle, you should have impressed the customer, met all the agreed success criteria, and be well on the way to closing the sale!

From Requirements to Performance Test

The requirements checklist described next is applicable to most performance testing projects whether you are executing in-house or providing a service to a customer. By design this does not enforce the use of any specific tools, so you can keep using your preferred project management and requirements capture solution even if it amounts to nothing more than MS Word.

In keeping with the spirit of this book, I've adopted a "shopping list" approach to emphasize the key points at each phase.

NOTE

The methodology steps are repeated in Appendix B to provide a handy quick reference.

Activity Duration Guidelines

You should find that the majority of project time is spent capturing, validating, and implementing requirements so that your performance testing foundation is built on rock, not sand. Any time outside this is typically devoted to scripting the transactions and to performance test execution and analysis.

Activities like building and commissioning a suitable test environment are too varied and complex to allow meaningful estimates. However, for certain key tasks I can offer the following guidance on likely time scales.

- *Scripting performance test transactions:* Allow half a day per transaction

  The reality is that some transactions will take more and others less time to script. But from the experience of many performance testing engagements, this is a realistic estimate.

- *Creating and validating performance test sessions or scenarios:* Typically one to two days' work

  This should simply be a job of work, since you will have already defined the structure and content of each performance test as part of your requirements capture. Most of the time will likely be spent conducting dress rehearsals to ensure that the tests execute correctly.

- *Performance test execution:* Allow a minimum of five days

  The unknown in this process is how many reruns will be required to deal with any problems. What can also significantly affect test execution time is needing to restore the database between performance test executions. Yet if you've correctly addressed requirements and made sure that your application is free from obvious problems, you can get a lot of performance testing done in five days.

- *Data collection (and software uninstall):* Allow one day.

  If testing in-house then software uninstall is unlikely to be required. If you are providing a service then you may have to remove the performance testing software. Give yourself a day to do this and to collect the results of the tests and KPI monitoring.

Step 1: Pre-Engagement Requirements Capture

Your first task in any performance testing project should be to set the following tasks in motion.

Gather or elicit performance requirements (specifications) from all relevant stakeholders; this was discussed in Chapter 2. You will need this information to create a successful project plan or statement of work (SOW).

At the very minimum you should have the following requirements agreed to and signed off on before undertaking anything else.

> NOTE
> This is often termed a *scoping* exercise.

- Deadlines available to complete performance testing, including the scheduled deployment date.
- Whether to use internal or external resources to perform the tests. This will largely depend on time scales and in-house expertise (or lack thereof).
- Test environment design agreed upon. Remember that the test environment should be as close an approximation of the live environment as you can achieve and will require longer to create than you estimate.
- Ensuring that a code freeze applies to the test environment within each testing cycle.
- Ensuring that the test environment will not be affected by other user activity. Nobody else should be using the test environment while performance test execution is taking place; otherwise, there is a danger that the test execution and results may be compromised.
- All performance targets identified and agreed to by appropriate business stakeholders. This means consensus from all involved and interested parties on the performance targets for the application. See Chapter 2 for a discussion on how to achieve it with the stakeholders that need to be involved.
- The key application transactions identified, documented, and ready to script. Remember how vital it is to have correctly identified the key transactions to script. Otherwise, your performance testing is in danger of becoming a wasted exercise.
- Which parts of transactions (such as login or time spent on a search) should be monitored separately. This will be used in Step 3 for "checkpointing."
- Identify the input, target, and runtime data requirements for the transactions that you select. This critical consideration ensures that the transactions you script run correctly and that the target database is realistically populated in terms of size and content. As discussed in Chapter 2, data is critical to performance testing. Make sure that you can create enough test data of the correct type within the time frames of your testing project. You may need to look at some form of automated data management, and don't forget to consider data security and confidentiality.
- Performance tests identified in terms of number, type, transaction content, and virtual user deployment. You should also have decided on the think time, pacing, and injection profile for each test transaction deployment.
- Identify and document server, application server, and network KPIs. Remember that you must monitor the application landscape as comprehensively as possible to ensure that you have the necessary information available to identify and resolve any problems that occur.
- Identify the deliverables from the performance test in terms of a report on the test's outcome versus the agreed performance targets. It's a good practice to produce a document template that can be used for this purpose.
- A procedure is defined for submission of performance defects discovered during testing cycles to the development or application vendor. This is an important consideration that is often overlooked. What happens if, despite your best efforts, you find major application,

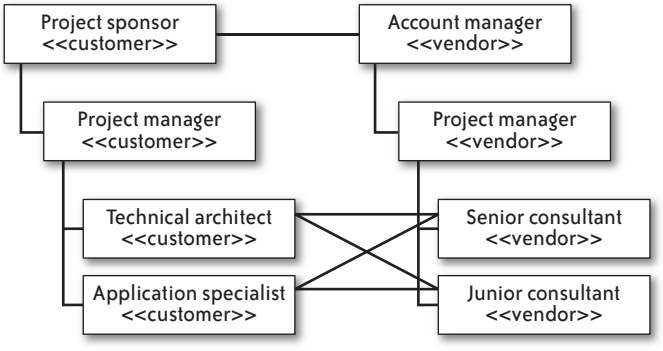


FIGURE 3-1. Example performance testing team structure

related problems? You need to build contingency into your test plan to accommodate this possibility. There may also be the added complexity of involving offshore resources in the defect submission process.

If your plan is to carry out the performance testing in-house then you will also need to address the following points, relating to the testing team:

- Test team members and reporting structure. Do you have a dedicated performance testing team? Often such teams are hurriedly put together by grabbing functional testers and anyone else unlucky enough to be available. To be fair, if you have only an intermittent requirement to performance test then it's hard to justify keeping a dedicated team on standby. Larger organizations, however, should consider moving toward establishing an internal center of testing excellence.

  Ensure at the very minimum that you have a project manager and enough testing personnel assigned to handle the scale of the project. Even large-scale performance tests rarely require more than a project manager and a couple of operatives. Figure 3-1 demonstrates a sample team structure and its relationship to the customer.
- The tools and resources available to the performance test. Make sure the team has what it needs to test effectively.
- Ensure that all team members have had adequate training in the tools to be used. A lack of testing tool experience is a major reason why many companies' introduction to automated performance testing is an outsourced project.

With this information available, you can proceed with the following:

- Develop a high-level plan that includes resources, time lines, and milestones based on these requirements.
- Develop a detailed performance test plan that includes all dependencies and associated time lines, detailed scenarios and test cases, workloads, and environment information.

- Make sure that you include a risk assessment of not meeting schedule or performance targets just in case things don't go according to plan.

> IMPORTANT
> **Be sure to include contingency for additional testing cycles and defect resolution if problems are found with the application during performance test execution. This precaution is frequently overlooked.**

With these actions underway, you can proceed through the remaining steps. Not everything mentioned may be relevant to your particular testing requirements, but the order of events is important.

> NOTE
> **See Appendix E for an example MS Project-based performance testing project plan.**

Step 2: Test Environment Build

You should have already identified the hardware, software, and network requirements for your test environment. Recall that you should strive to make your test environment a close approximation of the live environment. If this is not possible then at a minimum your setup should reflect the server tier deployment of the live environment and your target database should be realistically populated in terms of both content and sizing. This activity should be started as soon as possible because it frequently takes (much) longer than expected.

> WARNING
> **You may have to build a business case to justify creating your test environment!**

The test environment build involves the following steps:

- Allow enough time to source equipment and to configure and build the environment.
- Take into account all deployment models. You may need to test several different configurations over LAN and WAN environments for your application, so you need to ensure that each of these can be created in your test environment.
- Take into account links to external systems. You should never ignore external links, since they are a prime location for performance bottlenecks. As discussed in Chapter 2, you need either to use real links or to create some kind of simulation that represents external communication working as designed.
- Provide enough load injection capacity for the scale of testing envisaged. Think about the locations from which you will need to inject load. If these are not all local to the application

infrastructure, then you must ensure that the load injector machines can be managed remotely or station local personnel at each remote site.

- Ensure that the application is correctly deployed into the test environment. I have been involved in more than one performance test engagement that was unfavorably affected by mistakes made when the application—supposedly ready for testing—was actually deployed.
- Provision of sufficient software licenses for the application and supporting software (e.g., Citrix or SAP licenses). You'd be surprised how often an oversight here causes problems.
- Deploy and configure performance testing tools. Make sure that you *correctly* install and configure your performance testing solution.
- Deploy and configure KPI monitoring tool. This may be part of your performance testing solution or an entirely separate tool set. In either case, make sure that it is correctly configured to return the information you need.

Step 3: Transaction Scripting

For each transaction that you have selected to script, the following tasks must be performed:

- Identify the transaction *runtime* data requirements. Some of this information may be available from a Proof of Concept (POC) exercise, although this will have focused on only a few transactions. In many cases you will not be able to confirm runtime data requirements for the remaining transactions until you begin the scripting process.
- Confirm and apply transaction *input* data requirements. These should have been identified as part of the pre-engagement requirements capture. See Appendix A for an example of the sort of detail you should provide, particularly with regard to data for each transaction that will form part of your performance testing project.
- Decide on how you will "checkpoint" the transaction in terms of what parts you need to monitor separately for response time. This is an important consideration because it provides the first level of analysis of potential problem areas within the transaction. You should have addressed this issue after identifying the key transactions during pre-engagement requirements capture.
- Identify and apply any scripting changes required for the transaction to replay correctly. If you have already carried out a POC then you should have a sense of the nature of these changes and the time required to implement them.
- Ensure that the transaction replays correctly—from both a single user and a multiuser perspective—before signing it off as ready to include in a performance test. Make sure that you can verify what happens on replay either by checking that database updates have occurred correctly and/or by examining replay log files.

Step 4: Performance Test Build

For each performance test that you create, consider the following points:

- What kind of test will this represent—baseline, load, soak, or stress? A typical scenario is to have baseline tests for each transaction first in isolation as a single user and then up to the target maximum currency or throughput for the transaction. If problems are encountered at this stage, you may need to run isolation tests to identify and deal with what's wrong. This would be followed by a load test combining all transactions up to target concurrency followed by further isolation tests if problems are discovered. You may then want to run soak and stress tests for the final testing cycles, followed perhaps by non-performance related tests that focus on optimizing the load balancing configuration or perhaps different disaster recovery scenarios.
- Decide on how you will represent think time and pacing for each transaction included in the test. You should normally include think time and pacing in all test types (except the stress test); otherwise, you run the risk of creating an unrealistic transaction throughput.
- For each transaction, decide how many load injector deployments you require and how many virtual users to assign to each point of presence. As already mentioned, if your load injectors are widely distributed then be sure there's a local expert available to deal with any problems that may surface.
- Decide on the injection profile for each load injector deployment: Big Bang, ramp-up, ramp-up/ramp-down with step, or delayed start. Depending on what you are trying to achieve, your load test choice will likely involve a combination of Big Bang deployments to represent static load and one or more of the ramp variations to test for scalability. Figure 3-2 demonstrates a performance test plan using this approach. The darker rectangles on the bottom show the static load created by the Big Bang deployment, on top of which a variety of extra load tests are deployed in ramp-up steps.
- Will the test execute for a set period of time or rather be stopped by running out of data, reaching a certain number of transaction iterations, or user intervention? If the test will be data-driven, make sure that you have created enough test data!
- Do you need to spoof IP addresses to correctly exercise application load-balancing requirements? If so, then the customer will need to provide a list of valid IP addresses.
- Do you need to simulate different baud rates? If so, then confirm the different baud rates required. Any response-time prediction or capacity modeling carried out prior to performance testing should have already given you valuable insight into how the application reacts to bandwidth restrictions.
- What runtime monitoring needs to be configured using the server and network KPIs that have already been set up? if appropriate, the actual monitoring software should have been deployed as part of the test environment build phase, and you should already have a clear idea of exactly what to monitor in the application landscape.

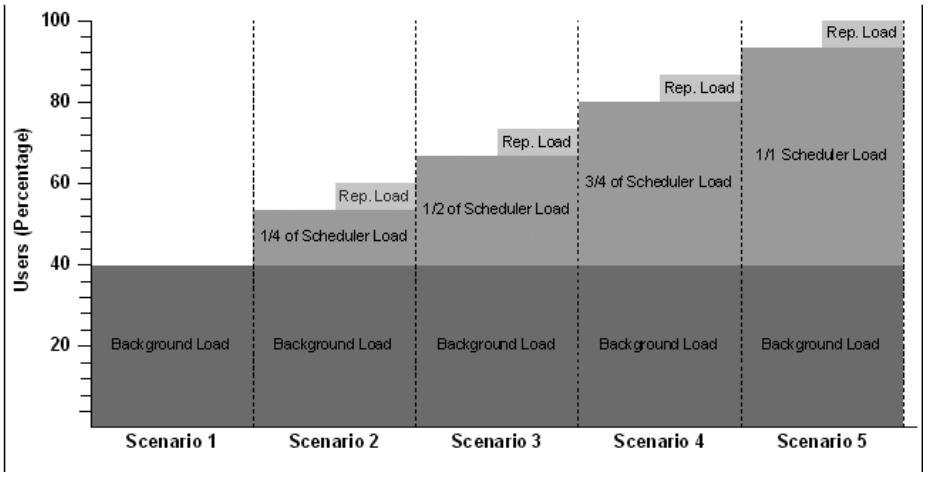


FIGURE 3-2. Performance test plan using background (static) load and ramp-up injection profiles

- If this is a web-based performance test, what level of browser caching simulation do you need to provide—new user, active user, returning user? This will very much depend on the capabilities of your performance testing solution. See Chapter 5 for a discussion on caching simulation.
- Consider any effects that the application technology will have on your performance test design. For example SAP performance tests that make use of the SAPGUI client will have a higher resource requirement than, say, a simple terminal emulator and thus will require more load injector machines to generate a given number of virtual users. Chapter 5 discusses additional considerations for SAP and other application technologies.

Step 5: Performance Test Execution

Run and monitor your tests. Make sure that you carry out a dress rehearsal of each performance test as a final check that there are no problems accessing the application and that there are no problems with the test configuration.

This phase should be the most straightforward part of any performance testing project. You've done the hard work preparing the test environment, creating the transaction scripts, addressing the data requirements, and building the performance tests. In an ideal world, performance test execution would be solely a matter of validating your application performance targets. It should not become a bug-fixing exercise.

The only unknown is how many test cycles will be required before your performance testing goals are achieved. I wish I could answer this question for you, but like many things in life this

is in the hands of the gods. But if you've followed the suggested performance testing checklist religiously to this point you're in pretty good shape to be granted a miracle!

- Execute dress-rehearsal tests to verify you have sufficient load injection capacity for the target concurrency. Unless your concurrency targets are very modest, you should always check the upper limits of your load injectors. It is all too easy to be caught without insufficient injector capacity when you can least afford it. (Trust me, it happens!) A dress rehearsal will also confirm that you can connect to the application and that you haven't omitted anything fundamental in your performance test configuration—for example, forgetting to include an external data file needed by a script.
- Execute baseline tests to establish "ideal" response-time performance. This is typically a single user per transaction for a set period of time or a certain number of transaction iterations.
- Execute load tests, ideally resetting target database content between executions. Once you have established performance baselines, the load test is normally the next step: all transactions should be apportioned among the target number of virtual users.
- Execute isolation tests to explore any problems revealed by load testing and then supply results to the developers or application vendor. This is why it's important to allow contingency time in your test plan, since even minor problems can have a significant impact on project time scales.
- Execute soak tests to reveal any memory leaks or problems related to high-volume transaction executions. Although it is not always feasible, I strongly recommend you include soak testing in your performance testing plan if possible.
- Execute stress tests, which are crucial from a capacity perspective. You should have a clear understanding of how much spare capacity remains within the application landscape to provide the data to plan for future growth of transaction volume and application users. You may also use stress testing to establish horizontal scalability limits for servers at a particular application tier.
- Execute any tests that are not performance related. For example, experiment with different load balancing configurations.

STEP 6 (Post-Test Phase): Analyze Results, Report, Retest

- Data collection (and possibly software uninstall if providing a service to a customer). Make sure that you capture and back up *all* data created as part of the performance testing project. It's easy to overlook important metrics and then discover they're missing as you prepare the project report.
- Determine success or failure by comparing test results to performance targets set as part of project requirements. Consensus on performance targets and deliverables *prior* to the

project's commencement makes the task of presenting results and proving application compliance a great deal less painful. (See Chapter 2 for a discussion on gaining consensus.)

- Document the results of the project using your preferred reporting template. The format of the report will be based on your own preferences and company or customer requirements, but it should include sections that address each of the performance targets. In other words, the deliverables of a performance testing project should be clearly defined as part of pre-test requirements capture. Then the resulting output document aligns the testing process and findings to agreed performance targets. This makes it much easier to present and *justify* the findings to the customer.
- Use the final results as baseline data for end-user experience (EUE) monitoring. If you have an EUE monitoring solution in place, then the output of a performance testing project provides a valuable point of reference to (re-)establish EUE SLAs for a new or already deployed application.

Case Study 1: Online Banking

Now let's move on to the first case study, which I will refer to as "Online Banking."

Online Banking is a critical customer-facing web application for a large multinational bank. It's been around for some time and provides a good service to the bank's customers. The motivation for the performance testing project that we'll discuss was an imminent expansion in terms of customer numbers and a need to explore the capacity limits of the current application infrastructure. The aim was to establish a baseline model for horizontal scaling of servers at the web, application server, and database tiers.

Application Landscape

Figure 3-3 shows the application landscape for Online Banking, which includes the following elements.

Clients
: This application provides a service to customers of the bank who connect using the Internet. This means that customers use whatever Internet browser software they prefer. Microsoft's Internet Explorer is still the most common choice, but there are many others including Mozilla, Firefox, Opera, and Netscape. There is no requirement for the end user to have any other software installed apart from their preferred browser.

Mid-tier servers
: Because the application is web-based, the first tier of servers are web servers. There are two quad blade servers, each running Windows 2003 Server as the operating system and Microsoft's IIS 6 as the web server software. The servers are load-balanced using Microsoft's Network Load Balancing (NLB) technology. Behind the two web servers is a mid-tier layer of a single dual-CPU application server.

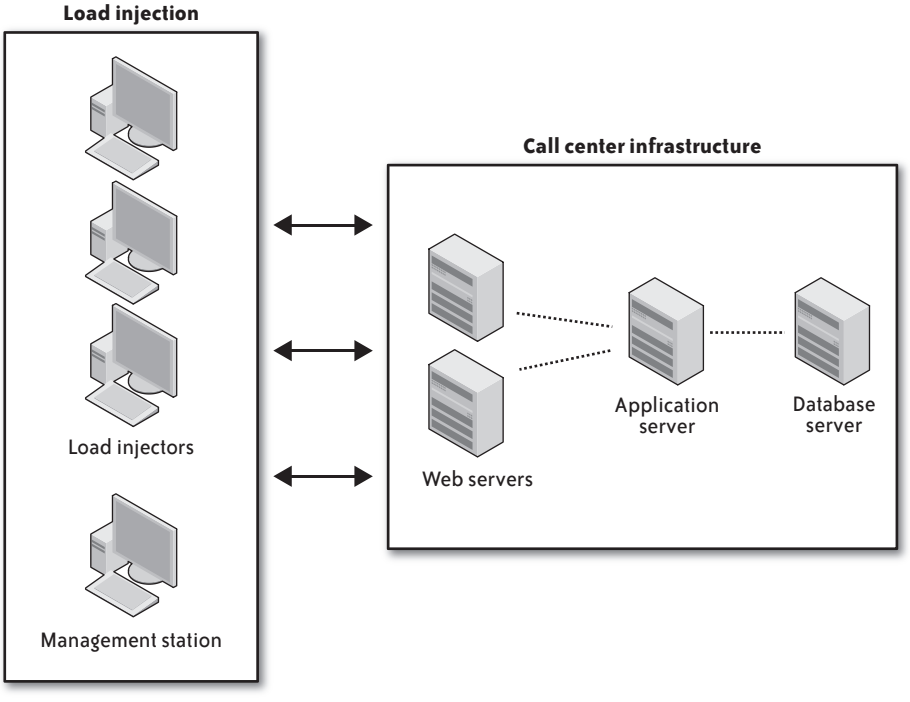


FIGURE 3-3. Online Banking application landscape

Database servers
: This is a single high-specification machine running MS SQL 2005 database software on Windows 2003 Server.

Network infrastructure
: All servers reside in a single data center with gigabit Ethernet connectivity. Internet connectivity to the data center consists of a single 8 Mbit WAN link to the local Internet backbone. There is also a 100 Mbit LAN connection to the company network.

Application Users

The application currently supports a maximum of approximately 380 concurrent users at the busiest point during a 24-hour period. These are mainly Internet customers, although there are likely to be a number of internal users carrying out administration tasks.

Every user has a unique set of login credentials and a challenge/response security question.

The services provided by the application are typical for online banking:

- Viewing statements
- Making payments
- Setting up direct deposits and standing orders
- Applying for a personal loan

Step 1: Pre-Engagement Requirements Capture

Project time scales amounted to a week of lead time and three days to carry out the testing engagement. The client decided to outsource the entire testing project because of the short schedule and lack of in-house testing expertise.

The test environment required little preparation, since testing was to be carried out using the live infrastructure. (More on this approach in Step 2.)

Performance targets for online banking were limited to availability and concurrency. The application had to be available and performant at a concurrent load of 1,000 virtual users. As mentioned, the application was coping with approximately 380 concurrent users at peak periods on a daily basis without problems, but this number was expected to increase significantly during the next year.

Ten transactions were identified as core to the performance testing project. Input data requirements amounted to a list of card numbers representing real user accounts and an accompanying PIN number that provided an additional level of account security. Target data requirements were unusually straightforward in that the live application database was to be used with the caveat of no "write" or "update" transactions unless the transaction backed up any changes made to the database as its final action.

The only performance test identified was a progressive ramp-up (without step) to the target concurrency of 1,000 virtual users. Had more time been available, a ramp-up with step variation would have provided a more granular view of steady-state performance.

Server and network KPIs focused on generic Windows performance metrics. There was no requirement to specifically monitor the application server or database layer. It subsequently transpired that application server monitoring would have been useful to identify poorly performing Java components whose presence had to be extrapolated from slow rendering of client-side content.

Step 2: Test Environment Build

What was unusual about this project was that there was no dedicated test environment. All performance testing was carried out against the live infrastructure! This is certainly not the norm, but neither is it a unique situation.

This case provided some unique challenges (although an accurate test environment was not one of them). First of all was dealing with an application already deployed and being actively used by customers and internal users. Any testing could be affected by other user activity, so this had to be taken into account when examining test results. In addition, performance test execution would definitely have an impact on the experience of real end users, particularly if high volumes of load are being generated.

So in this rather unusual situation, the performance test environment was immediately available and consisted of:

- Two load-balanced web servers
- One application server
- One database server

The one element missing was the resource needed to inject load. This was identified as requiring four PCs as injectors to provide a load of 1,000 virtual users (250 users per machine), with one PC also acting as the test management station.

Another consideration was that the volume load had to be injected onto the 8 Mbit WAN link that provided the connection from the Internet to the corporate data center. This involved distributing the load across a number of proxy servers to minimize the chance of overloading any single point of entry.

Because performance testing was carried out on the live environment, I couldn't use the integrated server and network monitoring capabilities of the chosen automated performance test tool. Corporate security constraints prevented the installation of any software not part of the standard build configuration onto production servers. (The infrastructure was, after all, that for a live banking application, so the restrictions were hardly surprising or draconian.)

This was an example of "Software Installation Constraints." Internal security policies can prohibit the installation of monitoring software onto the target servers or even connecting to them remotely. In these circumstances, the only options are to dispense with infrastructure monitoring altogether (not recommended) or to rely on whatever independent monitoring software has access to the application landscape.

For Online Banking it was possible to make use of Microsoft's Performance monitor (Perfmon) application to view server performance under load. This software is normally part of a default install on Windows server operating systems. A generic set of Windows server KPIs were instrumented using Perfmon and set to run in parallel with each performance test execution.

Step 3: Transaction Scripting

The ten transactions that were identified as key activities are listed in Table 3-1. Because this was a standard browser-based application, there were few challenges recording and preparing the transaction scripts. The only complication concerned runtime data requirements that

involved adding logic to the scripts to deal with the random selection of three characters from the account PIN number at application login.

Once logged in, users were restricted to visibility of their own account information. This limited search activity to date/account ranges and meant that the resulting scripts were navigation-driven rather than data-driven.

TABLE 3-1. Online Banking transactions

Transaction name	Description
MiniStatement	Login, view a mini statement, logout
CustomStatement	Login, view a custom statement, logout
ViewStandingOrders	Login, view standing orders, logout
TextAlertCreateCancel	Login, create or cancel a text alert, logout
TransferBetweenOwnAccount	Login, transfer an amount between personal accounts, logout
BillPaymentCreate	Login, create a bill payment to a nominated beneficiary, logout
BusMiniStatement	Login, generate a mini statement for nominated card numbers, logout
BusCustomStatement	Login, generate a custom statement for nominated card numbers, logout
BusTransferBetweenOwnAccounts	Login, transfer $5.00 between personal accounts for nominated card number, logout
BusBillPaymentCreate	Login, create a $5.00 bill payment to nominated beneficiary for nominated card numbers, logout

Step 4: Performance Test Build

For Online Banking the performance test design was based on scaling from 1 to 1,000 concurrent virtual users. The test was designed so that all users were active after a 30-minute period. Because different numbers of users were assigned to each transaction, each one had its own injection rate.

As shown in Table 3-2, the number of virtual users assigned to each transaction was based on an assessment of their likely frequency during a 24-hour period (this information is displayed in the table's fourth column). For example, the transaction "MiniStatement" was considered the most common activity, since most customers who use an online banking service will want to check their latest statement. (I know it's usually my first port of call!) As a result, 416 virtual users out of the 1,000 were allocated to this activity.

Differing numbers of users per transaction required different injection rates to ensure that all virtual users were active after a 30-minute period; this information is demonstrated in column three of Table 3-2, which shows how the virtual users were injected into the performance test. An important consideration to performance test design involves the accurate simulation of transaction throughput. You may recall our discussion in Chapter 2 about concurrency and how the number of concurrent virtual users does not necessarily reflect the number of users actually logged in to the application. For Online Banking it was necessary to apply specific

"think time" and "pacing" values to the performance test and transactions; this ensured that transaction throughput did not reach unrealistic levels during test execution.

TABLE 3-2. Virtual user injection profile

Transaction	Starting virtual users	Injection rate per one virtual user	Target virtual users
MiniStatement	1	4 seconds	416
CustomStatement	1	26 seconds	69
ViewStandingOrders	1	26 seconds	69
TextAlertCreateCancel	1	2 minutes 18 seconds	13
TransferBetweenOwnAccounts	1	12 seconds	145
BillPaymentCreate	1	8 seconds	225
BusMiniStatement	1	55 seconds	33
BusCustomStatement	1	7 minutes 30 seconds	4
BusTransferBetweenOwnAccounts	1	3 minutes	10
BusBillPaymentCreate	1	1 minute 52 seconds	16

Step 5: Performance Test Execution

Performance test execution principally involved multiple executions of ramp-up (without step) from 1 to 1,000 virtual users. As already mentioned, there was no integration of KPI monitoring of server or network data. Although the appropriate KPI's were identified as part of requirements capture, this information was monitored separately by the customer. Consequently, test execution had to be synchronized with existing monitoring tools—in this case, Microsoft's performance monitor (Perfmon).

The customer also monitored the number of active users independently over the period of the performance test. This raises an interesting point concerning the definition of concurrency. For an application, concurrency measures the number of users actually logged in. But automated performance tools measure concurrency as the number of active virtual users, which exclude virtual users who are logged in but kept in a waiting state. If your performance testing transactions include logging in and logging out as part of the execution flow (as was the case with Online Banking), then the number of active virtual users will be fewer than the number of users as seen by the application. Therefore, you will need to inflate the number of virtual users or (as I did with Online Banking) use a slower transaction pacing and execution rate to extend the time that virtual users remained logged in to the application and so create the proper concurrent load on the application.

Online Banking Case Study Review

Let's now assess the Online Banking case study in light of the suggested performance testing requirements checklist.

The test team

The test "team" amounted to a single individual—certainly not the ideal situation, but not uncommon. Companies that provide performance testing as a service will generally supply a minimum of a project manager and at least one testing operative. When performance testing is carried out in-house the situation is more fluid: many organizations do not have a dedicated performance testing team and tend to allocate resources on an ad hoc basis. For medium to large enterprises that carry out regular performance testing, a dedicated performance testing resource is a necessity in my view.

The test environment

In terms of providing an accurate performance testing environment, this case *was* ideal in terms of similarity to the deployment environment (i.e, they were one and the same). This unusual situation led to the challenges of other activity affecting the performance testing results and the potentially negative effect of scalability testing on real application users. If these challenges can be managed successfully, then this approach provides an ideal albeit unusual performance test environment.

KPI monitoring

As mentioned previously, internal constraints prevented the use of any integrated monitoring with the performance testing tool. It was therefore necessary for customers to carry out their own monitoring, after which the testing team had to manually integrate this data with that provided by the performance testing tool.

These situations are far from ideal because they complicate the analysis process and make it difficult to accurately align response time and scalability data with network and server performance metrics. However, sometimes there is no choice.

In this particular case, the performance data was exported to MS Excel in CSV format to allow graphical analysis. The ability to import third-party data into the analysis module of your performance testing tool is an extremely useful feature if you are unable to use integrated KPI monitoring.

Performance targets

The sole performance target was scalability based on 1,000 concurrent users. There was no formal requirement for a minimum response time, which could have been estimated fairly easily since the application was deployed and in regular use. The 1,000-user scalability target was arrived at in a rather arbitrary manner, and there was little evidence of consensus between the application stakeholders. This can be a significant risk to any performance testing project because it opens the door to challenges to interpretation of the results.

In this case, independent monitoring of application end-user activity during performance test execution allayed any fears that the stated load was not achieved. As discussed in "Step 5: Performance Test Execution," even though the performance test tool achieved 1,000 concurrent virtual users, this on its own would not have achieved the target of 1,000 concurrent application users. The difference was addressed by several hundred real users being active during performance test execution who effectively "topped up" the number of concurrent application users generated by the performance testing tool.

Transaction scripting

Ten transactions were selected and carefully documented in order to preclude any ambiguity over what was recorded. This followed the general trend whereby it is unusual rare to require more than 20 unique transactions in a performance testing engagement. The only required scripting changes involved login security.

Data requirements

Input data requirements for Online Banking were limited to a list of valid card numbers. These numbers provided initial access to the application for each virtual user.

The target database was the live database, which presented no problems in terms of realistic sizing. However, the performance test transactions were limited in terms of what data could be committed to the database. Although there was significant testing of user activity via database reads, there was little opportunity to test the effects of significant database write activity and this introduced an element of risk.

Runtime data requirements involved the identification and reuse of a set of standard challenge/response questions together with random characters selected from a range of PIN numbers associated with the card number data used as input to the test transactions.

Performance test design

Performance test design focused on a single test type that I would characterize as a load test. The application was already comfortably supporting 380 users at peak periods, and the stated intention was to try and achieve 1,000 users—some five times the existing capacity. As already mentioned, the test design incorporated specific "think time" and "pacing" changes to ensure that an accurate transaction throughput was maintained.

Case Study 2: Call Center

The second case study involves an application that is very different from online banking: a typical call-center application that provides driver and vehicle testing services to a region of the United Kingdom. The application undergoes frequent code changes as a result of government legislation changes, and there is a requirement to performance test the application prior to each new release. Extensive user acceptance testing (UAT) is carried out to eliminate any significant functional issues before the updated application is released for performance testing.

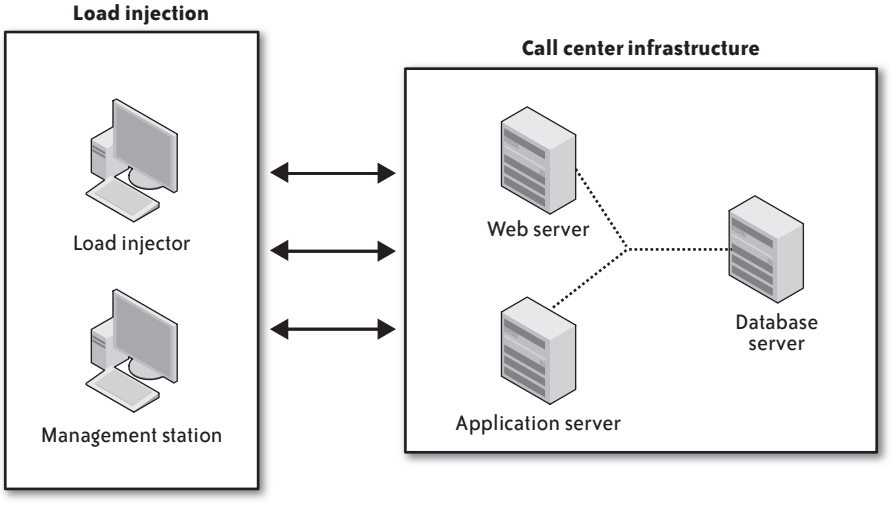


FIGURE 3-4. Call Center application landscape

Application Landscape

The application landscape for the call-center application comprises the following components (see Figure 3-4).

Clients
: The application client is slightly unusual in that it is a traditional fat client written in Visual Basic but makes use of web services technology to connect to the application server layer. There are two versions of the "fat" client deployed: one for the call-center operators and another for operators at each of the regional vehicle testing centers. As usual, public domain users can use whatever Internet browser software they prefer.

Mid-tier servers
: The mid-tier application architecture consists of two load-balanced application servers running Windows 2003 Server OS directly connected to call-center clients and a web server layer providing connectivity for public domain users.

Database servers
: The database server is a single high-specification machine running the MS SQL database software on Windows 2003 Server.

Network infrastructure
: All servers reside in a single data center with gigabit Ethernet connectivity. All connectivity to the application servers is via 100 Mbit LAN for call-center users or via ISP connection for public domain users.

Application Users

There is a typical maximum of 100 concurrent users during a normal working day in addition to an increasing number of self-service users through the 24/7 public-facing web portal. The call-center and regional test-center users each have their own login credentials, whereas web portal users provide information about their vehicle or driver's license to initiate a session. Typical transaction volumes are in the order of 1,500 transactions per 24-hour period.

Step 1: Pre-Engagement Requirements Capture

Project time scales were typical, amounting to a week of lead time and five days to carry out the testing engagement. The client had outsourced the entire testing project for a number of years because they lacked in-house performance testing expertise.

A dedicated test environment was provided, although (as is often the case) the number and specification of servers differed from the live environment.

Performance targets for the Call Center case study were availability, concurrency, and response time. The application had to be available and performant at a concurrent load of 100 virtual users. In terms of response time targets, performance at the web service level had to match or exceed that of the previous release.

Five transactions were identified as core to the performance testing project. Input data requirements for the call-center application were relatively complex and involved the following types of data.

Call-center use IDs
: A range of call-center login credentials was needed to provide variation in user sessions. For the target concurrency of 100 users, 20 sets of credential were provided.

Test centers
: Drivers can book appointments at a number of regional vehicle testing centers, so a list of all possible test centers was required for a realistic simulation.

Vehicle registration numbers
: Because appointments could be booked by vehicle registration number, a large number of valid registration numbers needed to be made available.

Driver license numbers
: Appointments could also be booked on behalf of the driver, so the vehicle registration numbers had to be accompanied by a list of valid driving license numbers.

The target database was a recent copy of the live database, so there were no problems with inadequate data sizing affecting performance test results. Although normally desirable, in this case the test database was rarely restored between performance test runs (it would have required a fresh "cut" of live data).

Performance tests used the ramp-up with step approach, reaching the target of 100 concurrent users. The tests would start focusing on baseline performance for each transaction, which ran with one virtual user in steady state for 10 minutes. Ramp-up then came in increments of 25 users with a steady-state period of 15 minutes at each step. The 15-minute period of stability allowed incremental observation of steady-state performance.

Server and network KPIs focused once again on generic Windows performance metrics, although this time it was possible to use the integrated monitoring capability of the performance testing tool.

The testing team for this case study was more conventional in that a project manager and testing consultant were assigned to the engagement. As mentioned before, I see this as the typical staffing requirement for the majority of performance testing projects.

Step 2: Test Environment Build

In Chapter 2 we discussed the need to set up the performance test environment correctly. What is important is to retain the deployment architecture in terms of the different types of server tiers. The test environment for our case study manages to retain the tier deployment, but it has only a single application server whereas the live environment has two that are load-balanced. This implies less capacity, which we must take into account when building our performance tests.

As shown in earlier in Figure 3-4, the test environment for the call center comprised:

- one web server
- one application server
- one database server

Load injection was provided by two workstations, one to inject the load and another to act as the management station.

Network analysis was carried out using packet level "sniffer" technology to determine the amount of data presentation per transaction and to model application performance in a WAN environment. This is a valuable exercise to carry out prior to the load test because it enables additional tuning and optimization of an application, potentially removing a layer of problems that may affect the success of performance testing.

Step 3: Transaction Scripting

For the Call Center case study, five transactions were identified that could all be considered active. Since the call-center and regional test-center clients were similar, the test transactions were selected only from the call center. Runtime data requirements were complex, requiring the extraction and reuse of information relating to the appointment booking selection. It was necessary to add code to the scripts to extract the required information. Unforeseen scripting

challenges can have a significant impact on performance testing projects; it is therefore extremely important to identify and resolve them in advance of the transaction scripting phase.

To ensure that the target concurrency was achieved, login and logout were not included as iterative activities during performance test execution. Each virtual user logged in at the start of the performance test and remained logged in until test conclusion, when the final action was to log out. Table 3-3 lists the transactions tested.

TABLE 3-3. Call Center transactions

Transaction	Description
DriverBookingSearch	Login, search for first available driver booking slots, logout.
FindSlotFirstAvailable	Login, search for first available vehicle booking slots, logout.
FindVRN	Login, find vehicle by registration number, logout.
SearchForAppointment	Login, find appointment by appointment ID number, logout
VehicleBooking	Login, make a vehicle booking, logout.

Step 4: Performance Test Build

The call-center application architecture is based on web services that invoke stored procedure calls on the MS SQL database. Historical problems with changes to web services and stored procedure calls between releases highlighted the importance of timing each web service call and monitoring the impact on the database server. Thus, we inserted checkpoints between all web service calls to allow granular timing of each request.

Step 5: Performance Test Execution

Performance test execution involved the following two steps.

- Baseline test execution for each transaction executing as a single user for a period of 10 minutes.
- Then ramp-up with step execution for each transaction in increments of 25 virtual users, with a steady-state period of 15 minutes at each increment.

Since the transactions in the Call Center case study did not include login and logout within their execution loop, there were no challenges maintaining true application user concurrency (as had occurred in the Online Banking case study).

Call Center Case Study Review

Let's now assess the Call Center case study in terms of our checklist.

The test team
: The test team consisted of a project manager and a single test operative, meeting what I consider to be typical requirements for a performance testing project. Because the

application was not developed in-house, any problems that arose had to be fed back to the application vendor. This meant that test execution had to be suspended while the problems were investigated, which led on more than one occasion to a postponement of the target deployment date.

The test environment

The environment was typical: dedicated hardware isolated from the live infrastructure in its own network environment. Application tier deployment largely matched that of the live application, the one major difference was the single application server versus the load-balanced pair of the live environment.

This difference in number of application servers could lead to a bottleneck in the mid-tier. In such cases you should approach any stress-testing requirements with caution. An excessively aggressive injection profile might result in premature application server overload and misleading results.

However, this test configuration does provide an opportunity to determine the upper limits of a single application server, in this case providing data on horizontal scalability for the application server tier.

KPI monitoring

Because a dedicated testing environment was available there were no obstacles to using the integrated server and network monitoring capability of the chosen performance testing tool. Server information was obtained by integrating the testing tool with Microsoft's Perfmon application. This is the preferred situation, and it allowed automated correlation of response time and monitoring data at the conclusion of each performance test execution —an important benefit of using automated performance testing tools.

Performance targets

Performance targets were supporting a concurrency of 100 users and ensuring that response time did not deteriorate from that observed during the previous round of testing. We used checkpointing to focus on web service call performance and to provide an easy mechanism for comparing releases.

Transaction scripting

Five transactions were selected from the call-center client to be used as the basis for performance testing the application.

Scripting challenges focused on the unusual nature of the application architecture, which involved large numbers of web service requests, each with an extensive set of parameters. These parameters would frequently change between releases, and the accompanying documentation was often unclear. This made it difficult to identify exactly what changes had occurred. As a result, scripting time varied considerably between testing cycles and sometimes required creation of a completely new set of scripts (rather than the preferred method of copying the scripts from the previous testing cycle and simply modifying the code). This shows the vulnerability of scripted transactions to code being invalidated by changes introduced in a new release.

Data requirements

Input data requirements for the Call Center were more complex than those for Online Banking. Large numbers of valid booking and vehicle registration numbers were needed before testing could proceed. This task was handled by extracting the relevant information from the target database via SQL scripts and then outputting the information in CSV format files.

The target database was a recent copy of the live database, which once again presented no problems in terms of realistic sizing. Since this was a true test environment, there were no restrictions on the type of end-user activity that could be represented by the scripted transactions. The one negative factor was the decision not to restore the database between test runs. I always advise that data be restored between test executions in order to avoid any effects on performance test results that are caused by changes in database content from previous test runs.

Runtime data requirements were rather complex and involved the interception and reuse of several different data elements, which represent server-generated information about the vehicle booking process. This made creating the original transaction scripts a nontrival exercise requiring the involvement of the application developers.

Performance test design

This test design used the "ramp-up with step" approach, which allowed us to observe steady-state response time of the application at increments of 25 virtual users for intervals of approximately 15 minutes.

Summary

It is hoped that the case studies in this chapter have struck some chords and that the checklists provided spell out a practical way to approach a Proof of Concept and application performance testing. The next step is to look at test execution and try to make sense of the information coming back from your performance testing solution.

CHAPTER FOUR

Interpreting Results: Effective Root-Cause Analysis

Statistics are like lampposts: they are good to lean on, but they don't shed much light.

—Anonymous

OK, so I'm running a performance test—what's it telling me? The correct interpretation of results is obviously vitally important. Since we're assuming you've (hopefully) set proper performance targets as part of your testing requirements, you should be able to spot problems quickly during the test or as part of the analysis process at test completion.

If your application concurrency target was 250 users, crashing and burning at 50 represents an obvious failure. What's important is having all the necessary information at hand to diagnose when things go wrong and what happened when they do. Performance test execution is often a series of false starts, especially if the application you're testing has significant design or configuration problems.

I'll begin this chapter by talking a little about the types of information you should expect to see from an automated performance test. Then we'll look at some real-world examples, some of which are based on the case studies in Chapter 3.

The Analysis Process

Analysis can be performed either as the test executes (in real time) or at its conclusion. Let's take a look at each approach in turn.

Real-Time Analysis

Real-time analysis is very much a matter of what I call "watchful waiting." You're essentially waiting for something to happen or for the test to complete without *apparent* incident. If a problem does occur, your KPI monitoring tools are responsible for reporting the location of the problem in the application landscape. If your performance testing tool can react to configured events, you should make use of this facility to alert you when any KPI metric is starting to come off the rails.

> NOTE
> **As it happens, "watchful waiting" is also a term used by the medical profession to describe watching the progress of some fairly nasty diseases. Here, however, the worst you're likely to suffer from is boredom or perhaps catching cold after sitting for hours in an overly air-conditioned data center. (These places are definitely not designed with humans in mind!)**

So while you are sitting there bored and shivering, what you should you expect to see as a performance test is executing? The answer very much depends on the capabilities of your performance testing tool. As a general rule, the more you pay, the more sophisticated the analysis capabilities are. But as an absolute minimum, I would expect to see the following:

- Response-time data for each transaction in the performance test in tabular and graphical form. The data should cover the complete transaction as well as any parts of the transaction that have been separately marked for analysis or checkpointed. This might include such activities as the time to complete login or the time to complete a search.
- You must be able to monitor the injection profile for the number of users assigned to each script and a total value for the overall test. From this information you will be able to see how the application reacts in direct response to increasing user load and transaction throughput.
- You should be able to monitor the state of all load injectors so you can check that they are not being overloaded.
- You need to monitor data that relates to any server, application server, or network KPIs that have been set up as part of the performance test. This may involve integration with other monitoring software if you're using a prepackaged performance testing solution rather than just a load testing tool.
- A display of any performance thresholds configured as part of the test and an indication of any breaches that occur.

- A display of all errors that occur during test execution that includes the time of occurrence, the virtual users affected, an explanation of the error, and possibly advice on how to correct the error condition.

Post-Test Analysis

All performance related information that was gathered during the test should be available at the test's conclusion and may be stored in a database repository or as part of a simple file structure. The storage structure is not particularly important so long as you're not at risk of losing data and it's readily accessible to the performance test team. At a minimum, the data you acquired during real-time monitoring should be available afterwards. Ideally, the tools should provide you with additional information, such as error analysis for any virtual users that encountered problems.

This is one of the enormous advantages of using automated performance testing tools: the output of each test run is stored for future reference. This means that you can easily compare two or more sets of test results and see what's changed. In fact, many tools provide a templating capability so that you can define in advance the comparison views you want to see.

As with real-time analysis, the capabilities of your performance testing tool will largely determine how easy it is to decipher what has been recorded. The less expensive and free tools tend to be weaker in the areas of post-test analysis and diagnostics (in fact, these are often practically non-existent).

Make sure that you make a record of what files represent the output of a particular performance test execution. It's quite easy to lose track of important data when you're running multiple test iterations.

Types of Output from a Performance Test

You've probably heard the phrase "Lies, damned lies, and statistics." Cynicism aside, statistical analysis lies at the heart of all automated performance test tools. If statistics are close to your heart then well and good, but for the rest of us I thought it a good idea to provide a little refresher on some of the jargon to be used in this chapter. For more detailed information, take a look at Wikipedia or any college text on statistics.

Statistics Primer

Mean and median

Loosely described, the *mean* is the average of a set of values. It is commonly used in performance testing to derive average response times. It should be used in conjunction with the *N*th percentile (described later) for best effect. There are actually several different

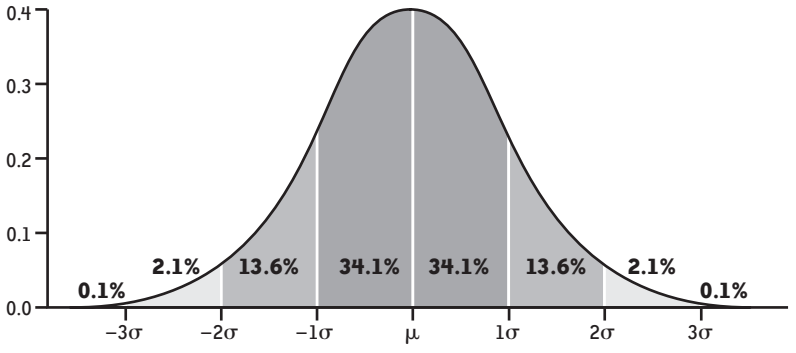


FIGURE 4-1. Simple bell curve example

types of mean value, but for the purpose of performance testing we tend to focus on what is called the "arithmetic mean."

For example: To determine the arithmetic mean of 1, 2, 3, 4, 5, 6, simply add them together and then divide by the number of values (6). The result is an arithmetic mean of 3.5.

Another related metric is the *median,* which is simply the middle value in a set of numbers. This is useful in situations where the calculated arithmetic mean is skewed by a small number of outliers, resulting in a value that is not a true reflection of the average.

For example: The arithmetic mean for the number series 1, 2, 2, 2, 3, 9 is 3.17, but the majority of values are 2 or less. In this case, the median value of 2 is a more accurate representation of the true average.

Standard deviation and normal distribution

Another common and useful indicator is standard deviation, which refers to the average variance from the calculated mean value. It's based on the assumption that most data in random, real-life events exhibit a normal distribution, more familiar to most of us from high school as a "bell curve." The higher the standard deviation, the farther the items of data tend to lie from the mean. Figure 4-1 provides an example courtesy of Wikipedia.

In performance testing terms, a high standard deviation can indicate an erratic end-user experience. For example, a transaction may have a calculated mean response time of 40 seconds but a standard deviation of 30 seconds. This would mean that an end user has a high chance of experiencing a response time as low as 25 and as high as 55 seconds for the same activity. You should seek to achieve a small standard deviation.

Nth percentile

Percentiles are used in statistics to determine where a certain percent of results fall. For instance, the 40th percentile is the value at or below which 40 percent of a set of results can be found. Calculating a given percentile for a group of numbers is not straightforward, but your performance testing tool should handle this automatically. All you normally need

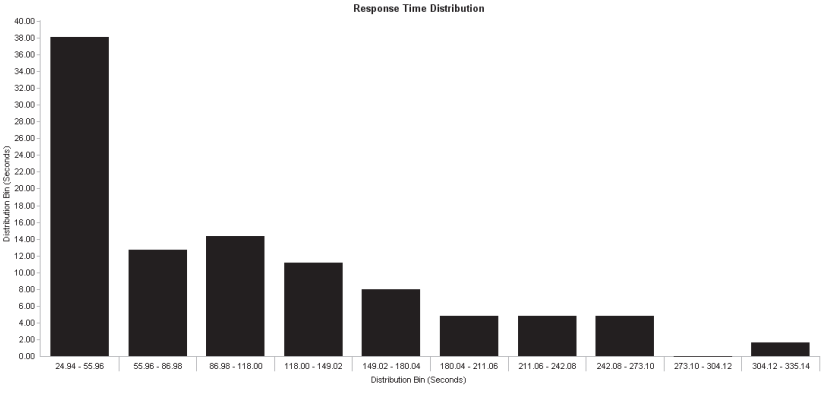


FIGURE 4-2. Response-time distribution

to do is select the percentile (anywhere from 1 to 100) to eliminate the values you want to ignore.

For example, let's take the set of numbers from our earlier skewed example (1, 2, 2, 2, 3, 9) and ask for the 90th percentile. This would lie between 3 and 9, so we eliminate the high value 9 from the results. We could then apply our arithmetic mean to the remaining 5 values, giving us the much more representative value of 2 (1 + 2 + 2 + 2 + 3 divided by 5).

Response-time distribution

Based on the normal distribution model, this is a way of aggregating all the response times collected during a performance test into a series of groups or "buckets." This distribution is usually rendered as a bar graph, where each bar represents a range of response times and what percentage of transaction iterations fell into that range. You can normally define how many bars you want in the graph and the time range that each bar represents. The Y-axis is simply an indication of measured response time. See Figure 4-2.

Response-Time Measurement

The first set of data you will normally look at is a measurement of application—or, more correctly, *server*—response time per transaction. Automated performance test tools typically measure the time it takes for an end user to submit a request to the application and receive a response. If the application fails to respond in the required time, the performance tool will record some form of time-out error. If this situation occurs then it is quite likely that an overload condition has occurred somewhere in the application landscape. We then need to check the server and network KPIs to help us determine where the overload occurred.

TIP

An overload doesn't always represent a problem with the application. It may simply mean that you need to increase one or more time-out values in the transaction script or the performance test configuration.

Any application time spent exclusively on the client is rendered as periods of think time, which represent the normal delays and hesitations that are part of end-user interaction with a software application. Performance testing tools generally work at the middleware level—that is, under the presentation layer—so they have no concept of events such as clicking on a combo-box and selecting an item unless this action generates traffic on the wire. User activity like this will normally appear in your transaction script as a period of inactivity or "sleep time" and may represent a simple delay to simulate the user digesting what has been displayed on the screen as well as individual or multiple actions of the type just described. If you need to time such activities separately then you may need to combine functional and performance testing tools as part of the same performance test (see Chapter 5).

These think-time delays are not normally included in response time measurement, since your focus is on how long it took for the server to send back a complete response after a request is submitted. Some tools may break this down further by identifying at what point the server started to respond and how long it took to complete sending the response.

Moving on to some examples, the next three figures demonstrate typical response-time data that would be available as part of the output of a performance test. This information is commonly available both in real time (as the test is executing) and as part of the completed test results.

Figure 4-3 depicts simple transaction response time (Y-axis) versus the duration of the performance test (X-axis). On its own this metric tells us little more than the response time behavior for each transaction over the duration of the performance test. If there are any fundamental problems with the application then response-time performance is likely to be bad regardless of the number of virtual users that are active.

Figure 4-4 shows response time for the same test but this time adding the number of concurrent virtual users at each point. Now you can see the effect of increasing numbers of virtual users on application response time. You would normally expect an increase in response time as more virtual users become active, but this should not vary in lockstep with increasing load.

Figure 4-5 builds on the previous two by adding response-time data for the checkpoints that were defined as part of the transaction. As mentioned in Chapter 2, adding checkpoints improves the granularity of the response-time analysis and allows correlation of poor response-time performance with the specific activities of a transaction. The figure shows that the spike in transaction response-time at approximately 1,500 seconds corresponded to an even more dramatic spike in checkpoints but did *not* correspond to the number of active virtual users.

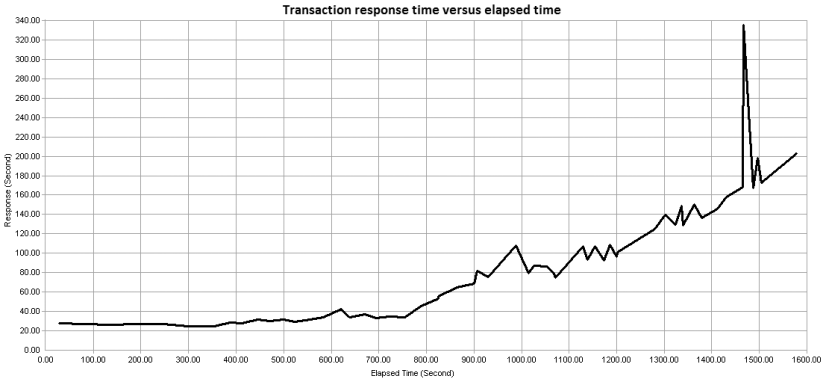


FIGURE 4-3. Transaction response time for the duration of a performance test

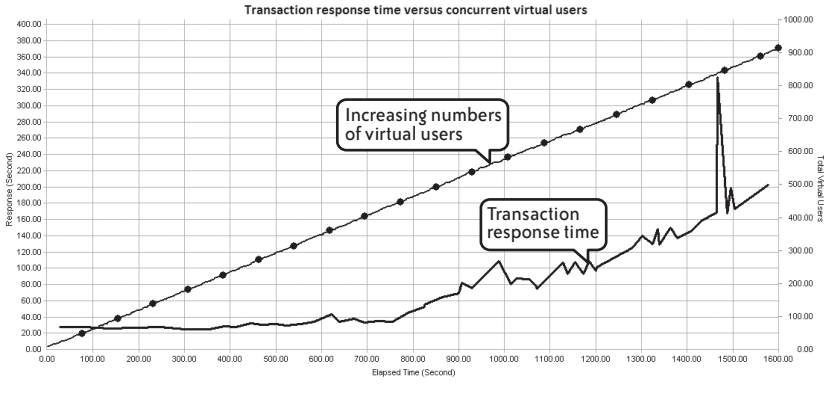


FIGURE 4-4. Transaction response time correlated with concurrent users for the duration of a test

In fact, the response-time spike at about 1,500 seconds was caused by an invalid set of login credentials supplied as part of the transaction input data. This clearly demonstrates the effect that inappropriate data can have on the results of a performance test.

Figure 4-6 provides a tabular view of response time data graphed in Figure 4-5. Here we see references to mean and standard deviation values for the complete transaction and for each checkpoint.

Performance testing tools should provide us with a clear starting point for analysis. For example, Figure 4-7 lists the ten worst-performing checkpoints for all the transactions within a performance test. This sort of graph is useful for highlighting problem areas when there are many checkpoints and transactions.

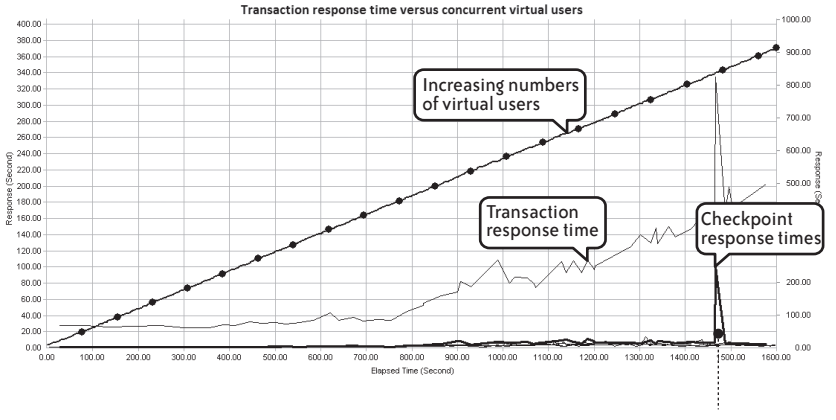


FIGURE 4-5. Transaction and checkpoint response time correlated with concurrent users

Name	Min	Mean	Max	Last	Std.Dev.
Transaction Response Time	24.9380	86.0258	335.1410	202.7350	61.5771
Total Running Virtual Users	10.0000	458.7383	913.0000	913.0000	261.8142
Login Response Time	0.7660	7.3413	24.2350	7.1870	5.3617
Security Question Response Time	0.5930	6.7524	20.6250	6.9690	5.0116
Click Custom Statements Response Time	0.8120	6.0654	35.0470	4.7970	5.2828
Show Statement Response Time	0.7970	6.0582	15.9690	8.4840	4.0250
Logoff Response Time	1.2030	15.1568	246.0940	10.7820	32.2105

FIGURE 4-6. Table of response-time data

Throughput and Capacity

Next to response time, performance testers are usually most interested in how much data or how many transactions can be handled simultaneously. You can think of this measurement as *throughput* to emphasize how fast a particular number of transactions are handled or as *capacity* to emphasize how many transactions can be handled in a particular time period.

Figure 4-8 illustrates transaction throughput per second for the duration of a performance test. This view shows when peak throughput was achieved and whether any significant variation in transaction throughput occurred at any point.

A sudden reduction in transaction throughput invariably indicates problems and may coincide with errors encountered by a virtual user. I have seen this frequently occur when the web server tier reaches its saturation point for incoming requests. Virtual users start to stall while waiting for the web servers to respond, resulting in an attendant drop in transaction throughput. Eventually users will start to time out and fail, however you may find that throughput stabilizes again (albeit at a lower level) once the number of active users is reduced

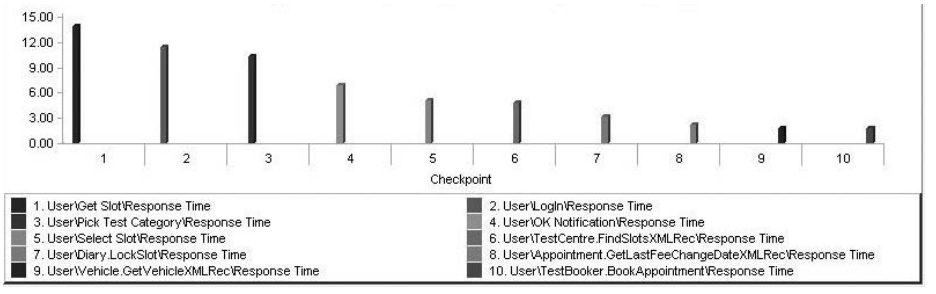


FIGURE 4-7. Top ten worst-performing checkpoints

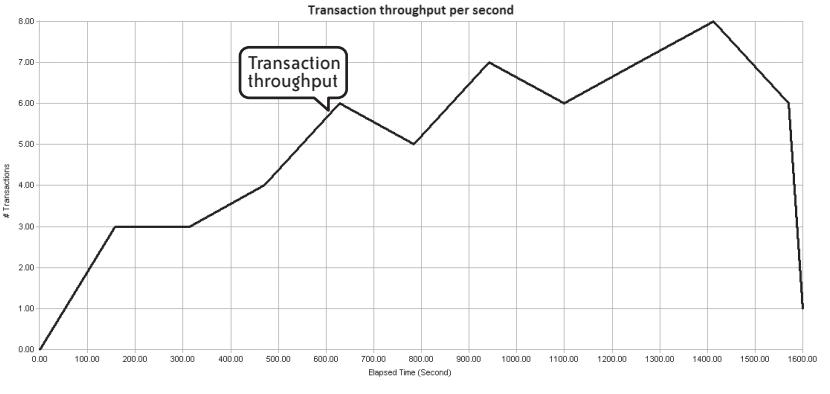


FIGURE 4-8. Transaction throughput

to a level that can be handled by the web servers. If you're really unlucky, the web or application servers may not be able to recover and all your virtual users will fail.

In short, reduced throughput is a useful indicator of the capacity limitations in the web or application server tier.

Figure 4-9 looks at the number of GET, CONNECT, and POST requests for active concurrent users during a web-based performance test. These values should gradually increase over the duration of the test, as they do in Figure 4-9. Any sudden drop-off, especially when combined with the appearance of virtual user errors, could indicate problems at the web server layer.

Of course, the web servers are not always the cause of the problem. I have seen many cases where virtual users timed out waiting for a web server response, only to find that the actual problem was a long-running database query that had not yet returned a result to the application or web server tier. This demonstrates the importance of setting up KPI monitoring for *all* server tiers in the application landscape.

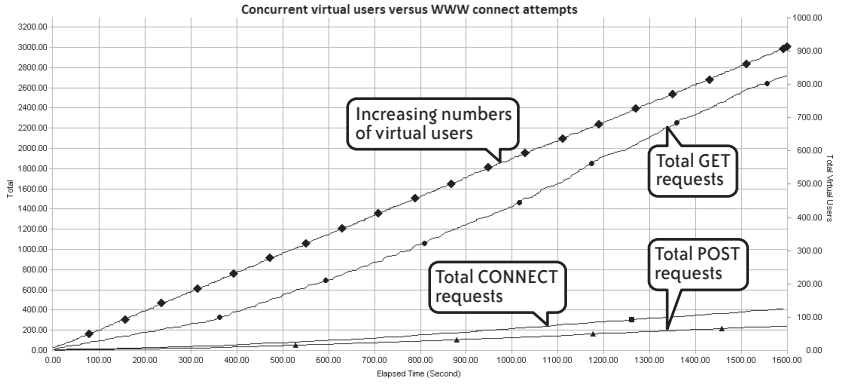


FIGURE 4-9. Concurrent virtual users correlated with web request attempts

Monitoring Key Performance Indicators (KPIs)

As discussed in Chapter 2, you can determine server and network performance by configuring your monitoring software to observe the behavior of key generic and application-specific performance counters. This monitoring software may be included in or integrated with your automated performance testing tool, or it may be an independent product. Any server and network KPIs configured as part of performance testing requirements fall into this category.

You can use a number of mechanisms to monitor server and network performance, depending on your application technology and the capabilities of your performance testing solution. The following sections divide the tools into categories, describing the most common technologies in each category.

Remote monitoring

These technologies provide server performance data (along with other metrics) to a remote system. That is, the server being tested passes data over the network to the part of your performance testing tool that runs your monitoring software.

The big advantage of using remote monitoring is that you don't usually need to install any software onto the servers you want to monitor. This circumvents problems with internal security policies that prohibit installation of any software that is not part of the "standard build." A remote setup also makes it possible to monitor many servers from a single location.

That said, each of these monitoring solutions needs to be activated and correctly configured. You'll need to be provided with an account that has sufficient privilege to access the monitoring software. You should also be aware that some forms of remote monitoring, particularly SNMP or anything using Remote Procedure Calls (RPC), may be prohibited by site policy because they can compromise security.

Common remote monitoring technologies include the following.

Windows Registry

This provides essentially the same information as Microsoft's Performance Monitor (Perfmon) application. Most performance testing tools provide this capability. This is the standard source of KPI performance information for Windows operating systems and has been in common use since Windows 2000 was released.

Web-Based Enterprise Management (WBEM)

Web-Based Enterprise Management is a set of systems management technologies developed to unify the management of distributed computing environments. WBEM is based on Internet standards and Distributed Management Task Force (DMTF) open standards: the Common Information Model (CIM) infrastructure and schema, CIM-XML, CIM operations over HTTP, and WS-Management. Although its name suggests that WBEM is web-based, it is not necessarily tied to any particular user interface.

Microsoft has implemented WBEM through their Windows Management Instrumentation (WMI) model. Their lead has been followed by most of the major Unix vendors, such as SUN and HP. This is relevant to performance testing because Windows Registry information is so useful on Windows systems and is universally used as the source for monitoring, WBEM itself is relevant mainly for non-Windows operating systems. Many performance testing tools support Microsoft's WMI, although you may have to manually create the WMI counters for your particular application and there may be some limitations in each tool's WMI support.

Simple Network Monitoring Protocol (SNMP)

A misnomer if ever there was one; I don't think *anything* is simple about using SNMP. However, this standard has been around in one form or another for many years and can provide just about any kind of information for any network or server device. SNMP relies on the deployment of Management Information Base (MIB) files that contain lists of Object Identifiers (OIDs) to determine what information is available to remote interrogation. For the purposes of performance testing, think of an OID as a counter of the type available from Perfmon. The OID, however, can be a lot more abstract, providing information such as the fan speed in a network switch. There is also a security layer based on the concept of "communities" to control access to information. Therefore, you need to ensure that you can connect to the appropriate community identifier; otherwise you won't see much. SNMP monitoring is provided by a number of performance tool vendors.

Java Monitoring Interface (JMX)

Java Management Extensions is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (such as printers), and service-oriented networks. Those resources are represented by objects called MBeans (for Managed Beans). JMX is useful mainly when monitoring Java application servers such as IBM WebSphere, ORACLE WebLogic, and JBOSS. JMX support is version-specific, so you need to check which versions are supported by your performance testing solution.

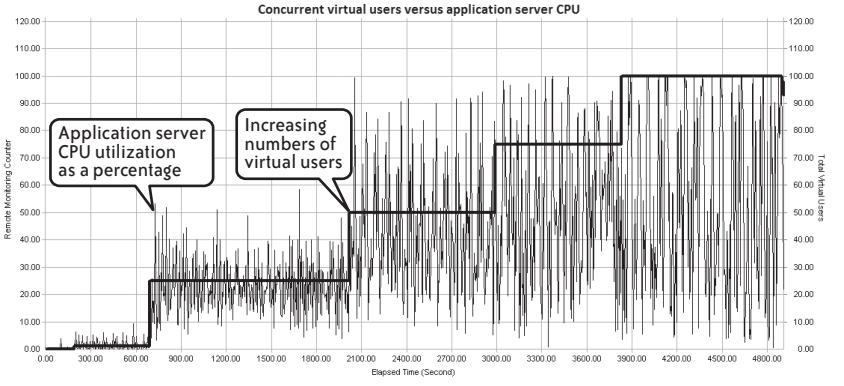


FIGURE 4-10. Concurrent virtual users correlated with database CPU utilization

Rstatd

This is a legacy RPC-based utility that has been around in the Unix world for some time. It provides basic kernel-level performance information. This information is commonly provided as a remote monitoring option, although it is subject to the same security scrutiny as SNMP because it uses RPC.

Installed agent

When it isn't possible to use remote monitoring—perhaps because of network firewall constraints or security policies—your performance testing solution may provide an agent component that can be installed directly onto the servers you wish to monitor. You may still fall foul of internal security and change requests, causing delays or preventing installation of the agent software, but it's a useful alternative if your performance testing solution offers this capability and the remote monitoring option is not available.

Server KPI Performance

Server KPIs are many and varied. However, two that stand out from the crowd are: how busy the server CPUs are and how much virtual memory is available. These two metrics on their own can tell you a lot about how a particular server is coping with increasing load. Some automated tools provide an expert analysis capability that attempts to identify any anomalies in server performance that relate to an increase in the number of virtual users or transaction response time (e.g., a gradual reduction in available memory in response to an increasing number of virtual users).

Figure 4-10 demonstrates a common correlation by mapping the number of concurrent virtual users against how busy the server CPU is. These relatively simple views can quickly reveal if a server is under stress. The figure depicts a "ramp-up with step" virtual user injection profile.

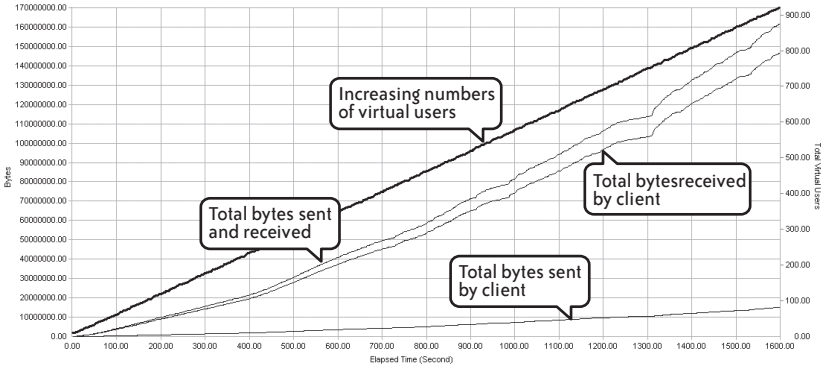


FIGURE 4-11. Network byte transfers correlated with concurrent virtual users

A notable feature of Figure 4-10 is the spike in CPU usage right after each step up in virtual users. For the first couple of steps the CPU soon settles down and handles that number of users better, but as load increases the CPU utilization becomes increasingly intense. Remember that the injection profile you select for your performance test scripts can create periods of artificially high load, especially right after becoming active, so you need to bear this in mind when analyzing test results.

Network KPI Performance

As with server KPIs, any network KPIs instrumented as part of the test configuration should be available afterwards for post-mortem analysis. The following example demonstrates typical network KPI data that would be available as part of the output of a performance test.

Figure 4-11 correlates concurrent virtual users with various categories of data presented to the network. This sort of view provides insight into the data "footprint" of an application, which can be seen either from the perspective of a single transaction or single user (as may be the case when baselining) or during a multitransaction performance test. This information is useful for estimating the application's potential impact on network capacity when deployed.

In this example it's pretty obvious that a lot more data is being received than sent by the client, suggesting that whatever caching mechanism is in place may not be optimally configured.

Load Injector Performance

Every automated performance test uses one or more workstations or servers as load injectors. It is very important to monitor the stress on these machines as they create increasing numbers of virtual users. As mentioned in Chapter 2, if the load injectors themselves become overloaded then your performance test will no longer represent real-life behavior and so will produce invalid results that lead you astray. Overstressed load injectors don't necessarily cause the test

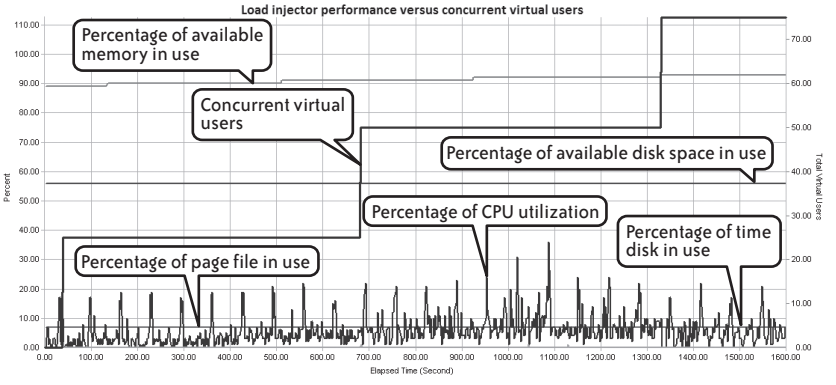


FIGURE 4-12. Load injector performance monitoring

to fail, but they could easily distort the transaction and data throughput as well as the number of virtual user errors that occur during test execution. Carrying out a dress rehearsal in advance of full-blown testing will help ensure that you have enough injection capacity.

Typical metrics you need to monitor include:

- Percent of CPU utilization
- Amount of free memory
- Page file utilization
- Disk time
- Amount of free disk space

Figure 4-12 offers a typical runtime view of load injector performance monitoring. In this example, disk space utilization is reassuringly stable, and CPU utilization seems to stay within safe bounds even though it fluctuates greatly.

Root-Cause Analysis

So what are we looking for to determine how our application is performing? Every performance test offers a number of KPIs that can provide the answers.

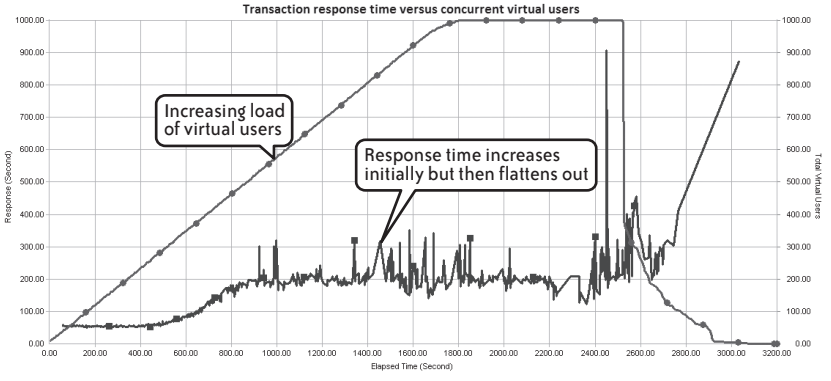


FIGURE 4-13. Good scalability/response time model

TIP

Before proceeding with analysis, you might want to adjust the time range of your test data to eliminate the start-up and shutdown periods that can denormalize your statistical information. This also applies if you have been using a "ramp-up with step" injection profile. If each ramp-up step is carried out en masse (e.g., you add 25 users every 15 minutes), then there will be a period of artificial stress immediately after the point of injection that may influence your performance stats (Figure 4-10). After very long test runs, you might also consider *thinning* the data, reducing the number of data points to make analysis easier. Most automated performance tools offer the option of data thinning.

Scalability and Response Time

A good model of scalability and response time demonstrates a moderate but acceptable increase in mean response time as virtual user load and transaction throughput increase. A poor model exhibits quite different behavior: as virtual user load increases, response time increases in lockstep and either does not flatten out or starts to become erratic, exhibiting high standard deviations from the mean.

Figure 4-13 shows good scalability. The line representing mean transaction response time increases to a certain point in the test but then gradually flattens out as maximum concurrency is reached. Assuming that the increased response time remains within your performance targets, this is a good result. (The spikes toward the end of the test were caused by termination of the performance and don't indicate a sudden application related problem.)

Figures 4-14 and 4-15 demonstrate undesirable response-time behavior. In Figure 4-14, the line representing mean transaction response time closely follows the line representing number of active virtual users until it hits approximately 750. At this point, response time starts to become erratic, indicating a high standard deviation and a potentially bad end-user experience. Figure 4-15 demonstrates this same effect in more dramatic fashion. Response time and concurrent virtual users in this example increase almost in lockstep.

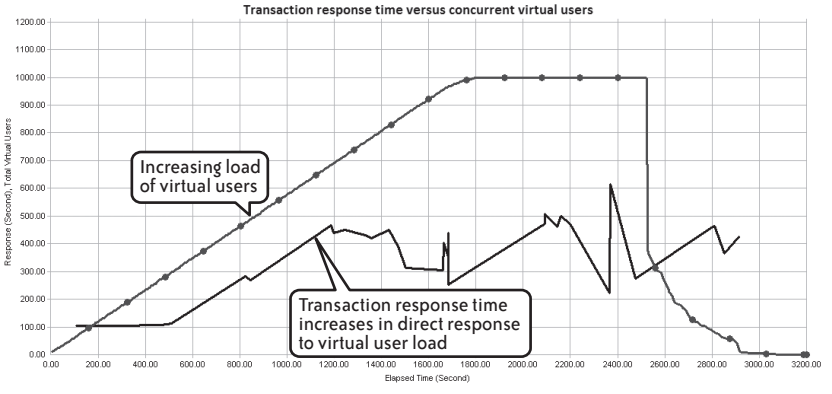


FIGURE 4-14. Poor scalability/response time model

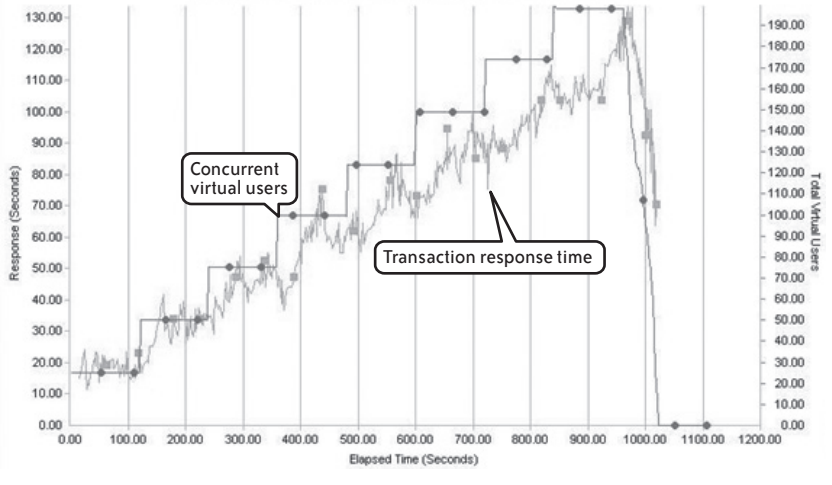


FIGURE 4-15. Consistently worsening scalability/response time model

Digging Deeper

Of course, scalability and response time behavior is only half the story. Once you see a problem, as in Figure 4-14, you need to find the cause. This is where the server and network KPIs come really into play.

Examine the KPI data to see whether any metric correlates with the observed scalability/response time behavior. Some performance testing tools have an autocorrelation feature that provides this information at the end of the test. Other tools require a greater or lesser degree of manual effort to achieve the same result.

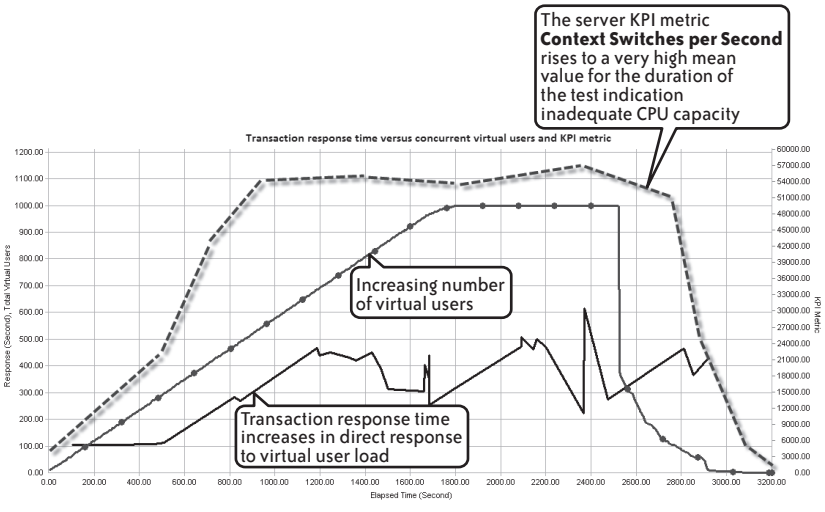


FIGURE 4-16. Mapping context switches per second to poor scalability/response time model

The following example demonstrates how it is possible to map server KPI data with scalability and response time information to perform this type of analysis.

Figure 4-16 builds on Figure 4-14, adding the Windows Server KPI metric called *context switches per second*. This metric is a good indicator on Windows servers of how well the CPU is handling requests from active threads. This example shows the CPU quickly reaching a high average value, indicating a lack of CPU capacity for the load being applied. This, in turn, is having a negative impact on the ability of the application to scale efficiently and thus on response time.

Inside the Application Server

In Chapter 2 we discussed setting appropriate server and network KPIs. I suggested defining these as layers, starting from a high-level, generic perspective and then adding others that focus on specific application technologies. One of the more specific KPIs concerned the performance of any application server component present in the application landscape. This type of monitoring lets you look "inside" the application server down to the level of Java or .NET components and methods.

Simple generic monitoring of application servers won't tell you much if there are problems in the application. In the case of a Java-based application server in a Windows environment, all you'll see is one or more *java.exe* processes consuming a lot of memory or CPU. You need to discover which specific component calls are causing the problem

SQL String	Using Prepared Statements	Total Tx Time %	TX % Not In Child Calls	Total CPU Time %	CPU % Not In Child Calls	Last 90 Sec Tx / sec	Last 90 Sec Avg Resp Time (sec)	Overall Avg Resp Time (sec)	Longest Resp Time (sec)	Total Invocation Count
select supplier.supplierid, supplier.name,unitprice from itemsup plier, supplier where supplier.supplierid=itemsupplier.supplierid and itemno=?	N	27.630	27.630	3.695	3.695	0.000	0.053	0.053	6.673	169
select password from customer where customerid=?	N	1.459	1.459	1.232	1.232	0.000	0.029	0.029	0.220	16
select ordertotal from orders where customerid=?	N	0.832	0.832	0.000	0.000	0.000	0.016	0.016	0.142	17
select * from orders where customerid=? order by orderno	N	1.853	1.853	4.926	4.926	0.000	0.020	0.020	0.297	30
select * from item where categoryname=? and subcategoryname=? order by itemno	N	1.065	1.065	2.545	2.545	0.000	0.023	0.023	0.125	15
select * from customer where customerid=?	N	1.176	1.176	5.008	5.008	0.000	0.027	0.027	0.204	14

FIGURE 4-17. Java application server's worst-performing SQL calls

Class Name	Method Name	Type	Last 90 Sec Avg Resp Time (sec)	Overall Avg Resp Time (sec)	Longest Resp Time (sec)	Total Invocation Count
.*BNTNET	itemlist_aspx(*)	ASP	0.085	0.085	6.874	133
BNTNETWebApp.itemlist	Page_Load(object,EventArgs)	.NET	0.084	0.084	6.843	127
System.Data.OleDb.OleDbC ommand	ExecuteReader(CommandBehavior)	SQL Query	0.046	0.046	6.796	217
.*BNTNET	account_aspx(*)	ASP	0.108	0.108	6.069	132
BNTNETWebApp.account	Page_Load(object,EventArgs)	.NET	0.105	0.105	5.610	129
.*BNTNET	default_aspx(*)	ASP	0.039	0.039	1.516	81
System.Web.UI.Page	ProcessRequest(HttpContext)	.NET	0.024	0.024	0.468	90
ASP.default_aspx	ProcessRequest(HttpContext)	.NET	0.031	0.031	0.468	40
.*BNTNET	myorders_aspx(*)	ASP	0.019	0.019	0.453	50
BNTNETWebApp.myorders	Page_Load(object,EventArgs)	.NET	0.019	0.019	0.421	47
.*BNTNET	WebResource_axd(*)	ASP	0.025	0.025	0.343	24
.*BNTNET	itemlink_aspx(*)	ASP	0.023	0.023	0.343	30
.*BNTNET	login_aspx(*)	ASP	0.018	0.018	0.314	33
ASP.itemlink_aspx	ProcessRequest(HttpContext)	.NET	0.021	0.021	0.311	25
.*BNTNET	checklogin_aspx(*)	ASP	0.024	0.024	0.299	30
BNTNETWebApp.checklogin	Page_Load(object,EventArgs)	.NET	0.023	0.023	0.283	28
ASP.itemlist_aspx	BeginProcessRequest(Http Context,AsyncCallback,object)	.NET	0.060	0.060	0.267	5
System.Data.Common.DbDat aAdapter	Fill(DataSet,string)	SQL Query	0.022	0.022	0.266	44
ASP.login_aspx	ProcessRequest(HttpContext)	.NET	0.017	0.017	0.204	25
BNTNETWebApp.Global	session_start()	.NET	0.024	0.024	0.187	22

FIGURE 4-18. Java application server's worst-performing methods

You may also run into the phenomenon of the stalled thread, where an application server component is waiting for a response from another internal component or from another server such as the database host. When this occurs, there is usually no indication of excessive CPU or memory utilization, just slow response time. The cascading nature of these problems makes them difficult to diagnose without detailed application server analysis.

Figures 4-17 and 4-18 present the type of performance data that this analysis provides.

Looking for the "Knee"

You may find that, at a certain throughput or number of concurrent virtual users during a performance test, the performance testing graph demonstrates a sharp upward trend—known colloquially as a *knee*—in response time for some or all transactions. This indicates that some capacity limit has been reached within the application landscape and has started to affect application response time.

Figure 4-19 demonstrates this effect during a large-scale multitransaction performance test. At approximately 260 concurrent users, there is a distinct "knee" in the measured response time of all transactions. After you observe this behavior, your next step should be to be look at server

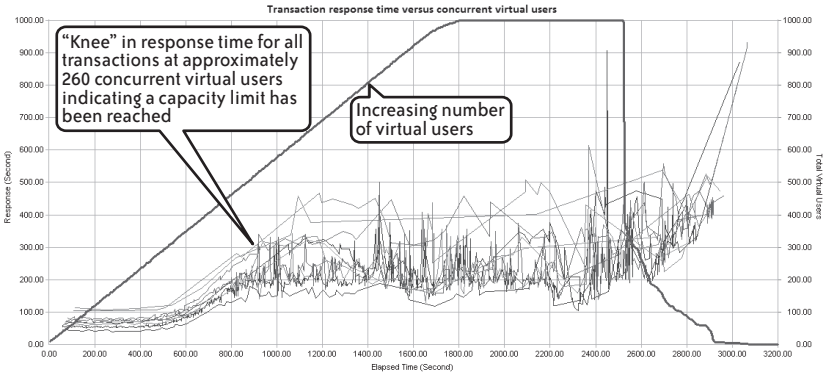


FIGURE 4-19. "Knee" performance profile indicating that capacity limits have been reached

and network KPIs at the same point in the performance test. This may, for example, reveal high CPU utilization or insufficient memory on one or more of the application server tiers.

In this particular example, the application server was found to have high CPU utilization and very high context switching, indicating a lack of CPU capacity for the required load. To fix the problem, the application server was upgraded to a more powerful machine. It is a common strategy to simply throw hardware at a performance problem. This may provide a short-term fix, but it carries a cost and a significant amount of risk that the problem will simply re-appear at a later date. In contrast, a clear understanding of the problem's cause provides confidence that the resolution path chosen is the correct one.

Dealing with Errors

It is most important to examine any errors that occur during a performance test, since these can also indicate hitting some capacity limit within the application landscape. By "errors" I mean virtual user failures, both critical and noncritical. Your job is to find patterns of when these errors start to occur and the rate of further errors occurring after that point. A sudden appearance of a large number of errors may coincide with the knee effect described previously, providing further confirmation that some limit has been reached. Figure 4-20 adds a small line to Figure 4-14 showing the sudden appearance of errors. The errors actually start before the test shows any problem in response time, and they spike about when response time suddenly peaks.

Baseline Data

The final output from a successful performance testing project should be baseline performance data that can be used when monitoring application performance after deployment. You now have the metrics available that allow you to set realistic performance SLAs for client, network,

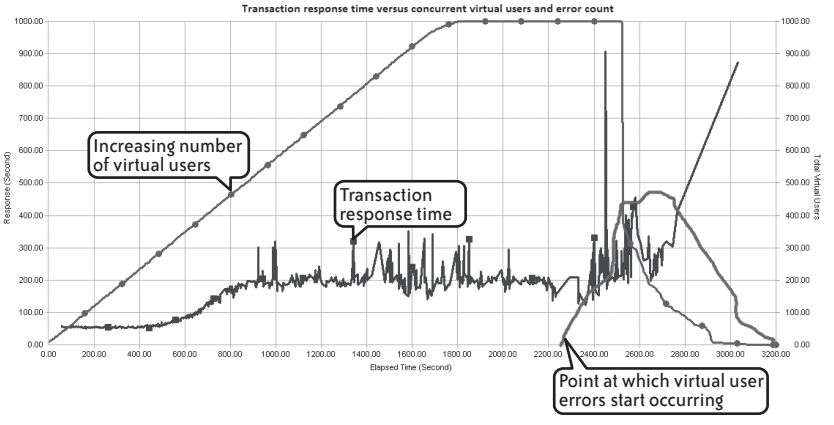


FIGURE 4-20. Example of errors occurring during a performance test

and server (CNS) monitoring of the application in the live environment. These metrics can form a key input to your Information Technology Service Management (ITSM) implementation, particularly with regard to End User Experience (EUE) monitoring (see "The IT Business Value Curve" in Chapter 1).

Analysis Checklist

To help you adopt a consistent approach to analyzing the results of a performance test both in real time and after execution, here is another checklist. There's some overlap with the checklist from Chapter 3, but the focus here is on analysis rather than execution. For convenience, I have repeated this information as part of the quick-reference guides in Appendix B.

Pre-Test Tasks

- Make sure that you have configured the appropriate server, application server, and network KPIs. If you are planning to use installed agents instead of remote monitoring, make sure there will be no obstacles to installing and configuring the agent software on the servers.
- Make sure that you have decided on the final mix of performance tests to execute. As discussed in Chapter 3, this commonly includes baseline tests, load tests, and isolation tests of any errors found, followed by soak and stress tests.
- Make sure that you can access the application from your injectors! You'd be surprised how often a performance test has failed to start because of poor application connectivity. This can also be a problem during test execution: testers may be surprised to see a test that was

running smoothly suddenly fail completely, only to find after much scratching of heads that the network team has decided to do some unscheduled "housekeeping."

- If your performance testing tool provides the capability, set any automatic thresholds for performance targets as part of your performance test configuration. This capability may simply count the number of times a threshold is breached during the test, and it may also be able to control the behavior of the performance test as a function of the number of threshold breaches that occur—for example, more than ten breaches of a given threshold could terminate the test.
- If your performance testing tool provides the capability, configure autocorrelation between transaction response time, concurrent virtual users, and server or network KPI metrics. This extremely useful feature is often included in recent generations of performance testing software. Essentially, it will automatically look for (undesirable) changes in KPIs or transaction response time in relation to increasing virtual user load or transaction throughput. It's up to you to decide what constitutes "undesirable" by setting thresholds for the KPIs you are monitoring, although the tool should provide some guidance.
- If you are using third-party tools to provide some or all of your KPI monitoring, make sure that they are correctly configured *before* running any tests. Ideally, include them in your dress rehearsal and examine their output to make sure it's what you expect. I have been caught running long duration tests only to find that the KPI data collection was wrongly configured or even corrupted!
- You frequently need to integrate third-party data with the output of your performance testing tool. Some tools allow you to automatically import and correlate data from external sources. If you're not fortunate enough to have this option, then you'll need to come up with a mechanism to do it efficiently yourself. I tend to use MS Excel or even MS Visio, as both are good at manipulating data. But be aware that this can be a very time-consuming task.

Tasks During Test Execution

- At this stage, your tool is doing the work. You need only periodically examine the performance of your load injectors to ensure that they are not becoming stressed. The dress rehearsal carried out before starting the test should have provided you with confidence that enough injection capacity is available, but it's best not to make assumptions.
- Make sure you document every test that you execute. At a minimum, record the following information:
  - The name of the performance test execution file and the date and time of execution.
  - A brief description of what the test comprised. (This also goes into the test itself, if the performance testing tool I'm using allows it.)

— If relevant, the name of the results file associated with the current test execution.

— Any input data files associated with the performance test and which transactions they relate to.

— A brief description of any problems that occurred during the test.

It's easy to lose track of how many tests you've executed and what each test represented. If your performance testing tool allows you to annotate the test configuration with comments, make use of this facility to include whatever information will help you to easily identify the test run. Also make sure that you document which test result files relate to each execution of a particular performance test. Some performance testing tools store the results separately from other test assets, and the last thing you want when preparing your report is to wade through dozens of sets of data looking for the one you need. (I speak from experience on this one!)

Finally, if your performance testing tool allows you to store test assets on a project basis, this can greatly simplify the process of organizing and accessing test results.

- Things to look out for during execution include the following:

The sudden appearance of errors

This frequently indicates that some limit has been reached within the application landscape. If your test is data-driven, it can also mean you've run out of data. It's worth determining whether the errors relate to a certain number of active virtual users. Some performance tools allow manual intervention to selectively reduce the number of active users for a troublesome transaction. You may find that errors appear when, say, 51 users are active, but by dropping back to 50 users the errors go away.

NOTE

Sudden errors can also indicate a problem with the operating system's default settings. I recall a project where the application mid-tier was deployed on a number of blade servers running Sun Solaris Unix. The performance tests persistently failed at a certain number of active users, although there was nothing to indicate a lack of capacity from the server KPI monitoring we configured. A search through system log files revealed that the problem was an operating system limit on the number of open file handles for a single user session. When we increased the limit from 250 to 500, the problem went away.

A sudden drop in transaction throughput

This is a classic sign of trouble, particularly with web applications where the virtual users wait for a response from the web server. If the problem is critical enough, the queue of waiting users will eventually exceed the time-out threshold for server responses and the test will exit. Don't immediately assume that the web server layer is the problem; it could just as easily be the application server or database tier. You may also find that the problem resolves itself when a certain number of users have

dropped out of the test, identifying another capacity limitation in the application landscape. If your application is using links to external systems, check to ensure that none of these links is the cause of the problem.

An ongoing reduction in available server memory
: You would expect available memory to decrease as more and more virtual users become active, but if the decrease continues after all users are active then you may have a memory leak. Application problems that hog memory should reveal themselves pretty quickly, but only a soak test can reveal more subtle problems with releasing memory. This is a particular problem with application servers, and it confirms the value of providing analysis down to the component and method level.

Panic phone calls from infrastructure staff
: Seriously! I've been at testing engagements where the live system was accidentally caught up in the performance testing process. This is most common with web applications, where it's all too easy to target the wrong URL.

Post-Test Tasks

- Once a performance test has completed—whatever the outcome—make sure that you collect all relevant data for each test execution. It is easy to overlook important data, only to find it missing when you begin your analysis. Most performance tools collect this information automatically at the end of each test execution, but if you're relying on other third-party tools to provide monitoring data then make sure you preserve the files you need.
- It's good practice to back up all testing resources (e.g., scripts, input data files, test results) onto a separate archive, because you never know when you may need to refer back to a particular test run.
- When producing your report, make sure that you map results to the performance targets that were set as part of the pre-test requirements capture phase. Meaningful analysis is possible only if you have a common frame of reference.

Summary

This chapter has served to demonstrate the sort of information provided by automated performance testing tools and how to go about effective root-cause analysis. The next chapter looks at how different application technologies affect your approach to performance testing.

CHAPTER FIVE

Application Technology and Its Impact on Performance Testing

IDIC, infinite diversity in infinite combinations.

—Mr. Spock in Star Trek, *1968*

As mentioned in the Preface, this book is designed to provide a practical guide to application performance testing. However, certain application technologies require us to refine our testing strategy to ensure that we achieve maximum benefit.

It therefore seems appropriate to offer some guidance on how different technologies may alter your approach to performance testing. I'll try to cover all the important technologies that I've encountered over many years of performance testing projects together with some of the newer offerings that have emerged alongside NET.

Asynchronous Java and XML (AJAX)

This is not the Greek hero of antiquity but rather a technology designed to make things better for the poor end users. The key word here is "asynchronous," which refers to breaking the mold of traditional synchronous behavior where "I do something and then I have to wait for you to respond before I can continue." Making use of AJAX in your application design removes this potential logjam and can lead to a great end-user experience. Unfortunately, most automated performance tools find this sort of technology difficult to handle.

The crux of the problem is that automated test tools are by default designed to work in synchronous fashion. Traditional applications make a request and then wait for the server to respond. In the AJAX world a client can make a request, the server responds that it has received the request (so the client can get on with something else), and then at some random time later the *actual* response to the original request is returned. You see the problem? It becomes a challenge to match requests with the correct responses.

From a programming standpoint, the performance tool must spawn a thread that waits for the correct response while the rest of the script continues to execute. A deft scripter may be able to code around this, but it may require significant changes to a script with the accompanying headache of ongoing maintenance. The reality is that not all testers are bored developers, so it's better for the performance tool to handle this automatically. The good news is that this capability is just starting to emerge from tool vendors.

If AJAX is part of your application design, check that any performance testing tool you are evaluating can handle this kind of challenge.

Push Versus Pull

While we're on the subject of requests and responses, let's discuss the case where an application makes requests to the client from the server (push) rather than the client initiating a request (pull). I have seen an example of this recently in an application designed to route incoming calls to operatives in a call center. The user logs in to announce that they are ready to receive calls and then sits and waits for the calls to arrive. This scenario is also known as "publish" and "subscribe."

How would you go about scripting this situation using traditional automated performance testing tools? You can record the user logging in, ready to receive calls, but then there's a wait until the calls arrive. This will require some manual intervention in the script to create some sort of loop construct that waits for incoming calls.

Sometimes you will have a mix of push and pull requests within the same script. If your performance testing tool doesn't handle this situation particularly well, then you must determine if the "push" content is important from a performance testing perspective. Can the server-initiated requests be safely ignored, or are they a critical part of the transaction flow? An example where you could probably ignore push content would be a ticker-tape stock-market feed to a client. It's always available, but it may not relate to any transaction that's been identified and scripted.

Citrix

Citrix is a leader in the field of thin-client technology. In performance testing terms, it represents a technology layer that sits between the application client and the application user. In effect, Citrix becomes the presentation layer for any applications "published" to the desktop

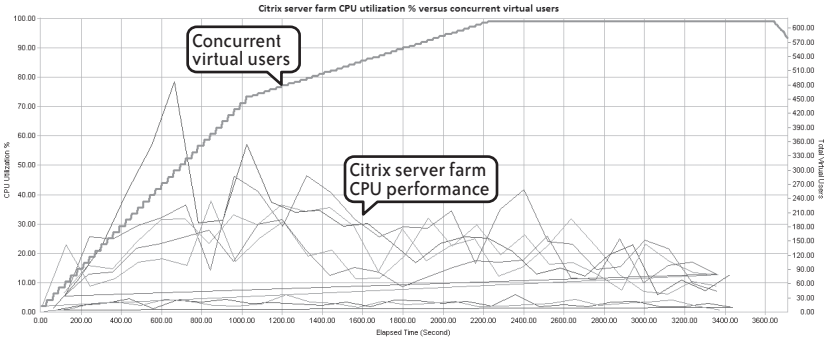


FIGURE 5-1. Citrix load balancing

and has the effect of moving the client part of an application to the Citrix server farm. Testing considerations will revolve around ensuring that you have enough capacity in the server farm to meet the needs of the number of virtual users that will be generated by the performance test. An important point that is often overlooked is to make sure that you have enough Citrix licenses for the load that you want to create. I have been involved in more than one Citrix performance testing engagement where this consideration (requirement!) had been overlooked, resulting in delay or cancellation of the project.

> **TIP**
>
> **Citrix provides guidelines on the number of concurrent sessions per server that can be supported based on CPU type and memory specifications, among other factors.**

Performance tests involving Citrix will often have validation of the server farm load balancing as a performance target. You must ensure that—when capturing your scripts and creating your load test scenarios—you are actually accessing the farm via the load balancer and not via individual servers. If there are problems then this should be pretty obvious: typically, there will be a few servers with high CPU and memory utilization while the other servers show little or no activity. What you want is a roughly even distribution of load across all the servers that make up the farm.

Figure 5-1 illustrates an example of Citrix load balancing under test from an actual engagement.

Citrix Checklist

- Make sure that you have enough Citrix licenses for the number of virtual users required for performance testing.

- If you are performance testing a load-balanced Citrix server farm, make sure that you capture your transactions using an Independent Computing Architecture (ICA) file; otherwise, on replay you may not correctly test the load balancing.
- Citrix connections rely on the appropriate version of the client ICA software being present. Make sure that the correct version is installed on the machine that will be recording your transactions *and* on the machines that function as load injectors. The Citrix client software is a free download from *http://www.citrix.com*.
- It is good practice to increase the default amount of time that your performance tool will wait for a response from the Citrix server farm. I typically increase the default value by a factor of 3 or 4 in order to counter any delays introduced by slow Citrix responses during medium- to large-scale performance tests.
- Always set the virtual memory page file size on each load injector machine to be 3 or 4 times total virtual memory. This is because the Citrix client was never intended to be used in the manner that it is by automated performance test tools, with dozens of simultaneous sessions active from a single workstation or server. This means that Citrix is resource hungry, especially regarding memory; I have seen many cases where servers with 3 or 4 gigabytes of memory have struggled to reliably run 100 virtual users. You should anticipate requiring a greater number of higher-specification load injectors than would be the case for a simple web client application.

HTTP Protocol

A number of unique considerations apply to applications that make use of the HTTP protocol. Rather than bundle them together under the term "Web," the advent of .NET and SOAP/XML has broadened the scope of HTTP applications beyond that of the traditional browser-based client connecting to a web server.

Web Services

Microsoft's Web Services RFC has introduced a new challenge for automated performance testing tools. Normally (although not exclusively) making use of the HTTP protocol, web services do not necessarily require deployment as part of the web server tier, which creates a problem for those tools that make use of browser proxy server settings to record transactions. It's possible to make any application that uses web services "proxy enabled," but you shouldn't assume that the customer or application developers will be prepared to make this change on demand.

The web service architecture is built around the web services definition language (WSDL) model, which exposes the web service to external consumers. This defines the web methods and the parameters you need to supply so that the web service can provide the information of interest. In order to test web services effectively, the ideal is to simply generate transaction

scripts directly from the WSDL file; a number of tool vendors provide this capability. There are also several (free!) test harnesses you can download that provide a mechanism to record web service activity, which allows you to get around the proxy server challenge.

See Appendix C for more details.

.NET Remoting

As part of. NET, Microsoft introduced the next generation of DCOM/COM+, which they termed "remoting." Essentially this provides a mechanism to build distributed applications using components that communicate over TCP or HTTP and thereby address some of the serious shortcomings of the older technology.

Like web services, remoting does not rely on web server deployment. It therefore presents the same challenges for automated performance testing tools with regard to proxy enablement.

In my experience it is normally possible to capture and script only those remoting applications that communicate over HTTP. I have had success with the TCP implementation in only a few cases (where the application transactions were extremely simple, little more than logging in to an application and then logging out). Microsoft initially recommended the TCP implementation as having superior performance over HTTP. Yet because this difference has largely disappeared in later versions of .NET, TCP implementations are uncommon.

Although it may be possible to record remoting activity at the Windows Socket (WINSOCK) level, the resulting scripts (because you are dealing with serialized objects) can be difficult to modify so that they replay correctly. This is doubly so if the application makes use of binary format data rather than SOAP/XML, which is in clear text. Such cases may require an unrealistic amount of work to create usable scripts; a better solution may be to look for alternatives (see "Oddball Application Technologies: Help, My Load Testing Tool Won't Record It!" later in this chapter).

Browser Caching

It is extremely important that the default caching behavior of Internet browser applications be accurately reflected by the performance tool that you use. If caching emulation is inaccurate, then it is likely that the number of requests generated between browser client and web server will be considerably greater than would actually occur in a live environment. This will have the effect of generating a much higher throughput for a given number of virtual users. I can recall a recent example where a few hundred virtual users generated as much throughput as 6,000 "real" users. This may be satisfactory for a stress test when you are simply trying to find the upper limits of capacity, but it is hardly a realistic load test.

Typical default caching behavior involves page elements such as images, style sheets, and the like being stored locally on the client after the initial request, removing the need for the server to resend this information. Various mechanisms exist to update the cached elements, but the

net result is to minimize the data presentation and network round trips between client and web server, reducing vulnerability to congestion and latency effects and (we hope) improving the end-user experience.

Secure Sockets Layer (SSL)

Many applications make use of the SSL extension to the HTTP protocol that adds an encryption layer to the HTTP stack. If your automated performance test tool can't handle SSL then you won't be able to record any transactions unless you can record them from a non-SSL deployment of the application. There are two other aspects of SSL that you should be aware of.

Certificates
: One way of increasing security is to make use of client certificates, which can be installed into the application client before access is permitted to an application. This implements the full private/public key model. The certificates are provided in a number of formats (typically pfx or p12) and must be made available to your performance testing tool for successful capture and replay. If your application makes use of client certificates, check that the performance testing tool can deal with client certification; otherwise, you won't be able to record your application. Typically the certificates are imported into the testing tool prior to capture and then automatically used at appropriate points within the resulting script.

Increased overhead
: Making use of SSL increases the number of network round-trips and the load on the client and web server. This is because there is an additional authentication request sent as part of every client request as well as the need to encrypt and decrypt each network conversation. Always try to retain the SSL content within your performance test scripts, otherwise you may end up with misleading results due to reduced network data presentation and a lighter load on the web server(s).

Java

Over recent years it would be fair to say that IT has embraced Java as the technology of choice for distributed application server design. Products from IBM such as WebSphere and Oracle (Bea) WebLogic dominate the market in terms of deployed solutions, and many other market leaders in packaged applications such as SAP provide the option of a Java-based mid-tier. In addition there are many other Java application server technologies for the software architect to choose from, including JBoss and JRun.

Java application clients can come in many forms, ranging from a "pure Java" fat client—which is probably the most challenging for a performance testing tool that records at the middleware level—to the typical browser client that uses the HTTP protocol to talk to the web server layer.

Because of the component-based nature of Java applications, it is vital to have access to the internal workings of the application server under load and stress. Several tools are available (see Appendix C) to provide this level of visibility, exposing components or methods that consume excessive amounts of memory and CPU as well as design inefficiencies that have a negative impact on response time and scalability.

By no means unique to Java, the following are some common problems that cannot easily be detected without detailed monitoring at the component and method level.

Memory leaks
: This is the classic situation where a component uses memory but fails to release it, gradually reducing the amount of memory available to the system. This is one reason why it's a good idea to run soak tests, because memory leaks may occur for a long time before affecting the application.

Excessive component instantiations
: There are often cases where excessive instances of a particular component are created by the application logic. Although this doesn't necessarily cause performance issues it does lead to inefficient use of memory and excessive CPU utilization. The cause may be coding problems or improper configuration of the application server. Either way, you won't be aware of these issues unless you can inspect the application server at component level.

Stalled threads
: This was mentioned briefly in Chapter 4. A stalled thread occurs when a component within the application server waits for a response from another internal component or (more commonly) an external call to the database or perhaps a third-party application. Often the only symptom is poor response time without any indication of excessive memory or CPU utilization. Without visibility of the the application server's internal workings, this sort of problem is difficult to isolate.

Slow SQL
: Application server components that make SQL calls can also be a source of performance issues. Calls to stored procedures will be at the mercy of code efficiency within the database. Note also that any SQL parsed on the fly by application server components may have performance issues not immediately obvious from casual database monitoring.

Oracle

There are two common types of Oracle application technology you will encounter. Each has its own unique challenges.

Oracle Two-Tier

Oracle two-tier applications were once the norm for a fat client that made use of the Microsoft Open Database Connectivity (ODBC) standard or, alternatively, the native Oracle Client

Interface (OCI) or User Programming Interface (UPI) software libraries to connect directly to an Oracle database. The OCI interface is pretty much public domain and a number of tool vendors can support this protocol without issue. However, the UPI protocol is an altogether different animal, and applications that make use of this technology can be extremely difficult to record. It is not uncommon to come across application clients that use a mix of both protocol stacks!

Oracle Forms Server (OFS)

Current Oracle technology promotes the three-tier model, commonly making use of Oracle Forms Server (OFS). This involves a browser-based connection using the HTTP (with or without the SSL extension) to connect to a Java-based mid-tier that in turn connects to a backend (Oracle) database. This HTTP traffic starts life as normal HTML requests and then morphs into a series of HTTP POSTS, with the application traffic represented as binary data in a proprietary format.

ORACLE provides a browser plug-in called JInitiator that manages the connection to the Forms Server, which is a free download from the Oracle web site.

Oracle Checklist

- If you are performance testing Oracle Forms, then make sure you have the correct version of JInitiator installed on the workstation where you will be recording your transactions *and* on the machines that will be acting as load injectors.
- Remember that your performance testing tool must specifically support the version of Oracle Forms that you will be using. It is not sufficient to use simple HTTP recording to create scripts that will replay correctly.
- If your application is the older Oracle two-tier model that uses the Oracle client communication, then the appropriate OCI/UPI library files must be present on the machine that will be used to capture your transactions *and* on the machines that will act as load injectors.

SAP

SAP has been a historical leader in customer relationship management (CRM) and other packaged solutions for the enterprise. SAP applications can have a traditional fat client or SAPGUI and can also be deployed using a variety of web clients.

One limitation that SAP places on performance testing tools is the relatively large resource footprint associated with the SAPGUI client. If you consider for a moment that the typical SAP user generally has only a single SAPGUI session running on their workstation, then this is

entirely understandable. With an automated performance test tool we're creating literally dozens of SAPGUI sessions on a single machine, a situation that never occurs in the real world.

In order to maximize the number of virtual SAPGUI users, you should maximize the amount of RAM in each injector machine and then increase the virtual memory page file size to three to four times the amount of virtual memory. This will help, but even with four gigabytes of RAM you may find it difficult to get more than 100 virtual SAPGUI users running on a single machine.

Like Citrix you should anticipate requiring a greater number of machines to create a given SAPGUI virtual user load than for a typical web application. Happily this is not so for the SAP web client, where you should find that injection requirements differ little from any other browser-based client application.

SAP Checklist

- Increase load injector page file size to three to four times virtual memory in order to maximize the number of virtual users per injector platform.
- Anticipate that a larger number of injector machines will be required to generate the load you are looking to simulate than would be the case for a typical browser-based application.
- If your performance testing tool makes use of the SAP Scripting API, then you will need server-side scripting to be enabled on the server and on the client where your transactions will be captured.
- To monitor SAP servers remotely you can use either SNMP or WMI; otherwise you will need the latest version of the SAP JCo package installed on the machine that will carry out the remote monitoring. You need to be a SAP customer or partner to have access to this software.
- Make sure that the Computing Center Management System (CCMS) is enabled on the SAP servers (and instances) that you wish to monitor, for otherwise you will fail to collect any KPI data.

Service-Orientated Architecture (SOA)

Service-Orientated Architecture (SOA) is one of the current crop of technology buzzwords—along with Web 2.0, Information Technology Inventory Library (ITIL), Information Technology Service Management (ITSM), and others. But trying to establish clearly just what SOA means has been like trying to nail jelly to a wall. Nonetheless, the first formal standards are starting to appear, and SOA adoptance is becoming more commonplace.

In essence, SOA redefines IT as delivering a series of services to a consumer (organization) based around Business Process Management (BPM). A service may consist of one or more business processes that can be measured in various ways to determine the value that the service

is providing to the business. You could think of a business process as something no more complicated than an end user bringing up a purchase order, but it is just as likely to be much more complex and involve data exchange and interaction between many different systems within an organization. The business processes themselves can also be measured, creating a multitiered view of how IT is delivering to the business.

Traditional applications make the encapsulation of end-user activity a pretty straightforward process, where each activity can be recorded as a series of requests and responses via a common middleware such as HTTP. With SOA, a business process can involve multiple transactions with interdependencies and a multitude of protocols and technologies that don't necessarily relate directly to end-user activity. How do you render this as a script that can be replayed?

This brings us to the concept of the "service." As you are probably aware, Microsoft's web service RFC is a mechanism for requesting information either internally or from an external source via HTTP, the protocol of the Internet. It is usually simple to implement and understand, which is one of the advantages of using web services. A web service can be integrated as part of an application client and/or deployed from the application mid-tier servers. If it's part of the client then this has little effect on our performance testing strategy; but if it's part of the mid-tier then we'll likely be testing for concurrent service execution rather than for concurrent users. The transactions that we script will not represent users but rather an execution of one or more web service requests.

There is, of course, much more to the SOA performance testing model than web services. Other "queue and message"-based technologies such as JMS and MQ have been around for some time and, along with web services, are application technologies that just happen to be part of many SOA implementations. Each of these technologies can be tested in isolation, but the challenge is to performance test at the Service and Business Process level, which is generally too abstract for the current generation of automated performance testing tools.

The good news is that the first generation of SOA-enabled tools with some performance testing ability are just starting to appear. The next edition of this book will have much more to say on SOA-based performance testing!

Web 2.0

Like SOA you have probably heard the term Web 2.0 being bandied about, but what does it actually mean? Well it pretty much describes a collection of software development technologies (some new, some not so new) that Microsoft and Adobe, in particular, would like for us to use for development of the next generation of web applications:

- Asynchronous Java and XML (AJAX)
- Flex from Adobe
- Web services

Apart from Flex, I've already discussed how these technologies can affect your approach to performance testing. The challenge that Web 2.0 introduces is the use of a combination of different middleware technologies in a single application client. This is not a new concept as a number of tool vendors have provided a "multiprotocol" capture facility for some time. Rather, it is the combination of .NET and Java technologies working at the component level that must be captured and correctly interpreted taking into account any encryption or encoding that may be present.

Working at this level requires an intimate understanding of how the application works and helpful input from the developers. The output of your capture process is much more likely to be a component map or template, which provides the information to manually build a working script.

> NOTE
> If you're not familiar with Flex, it essentially provides an easier programming model for application developers to make use of the powerful animation and graphic capabilities that are part of Adobe Flash. Have a look at Adobe's web site if you're interested in learning more.

Windows Communication Foundation (WCF) and Windows Presentation Foundation (WPF)

You may know Windows Communication Foundation (WCF) by its Microsoft codename "Indigo." WCF is an emerging standard that unifies the following (Microsoft) technologies into a common framework to allow applications to communicate internally or across the wire. You might call this "Web 2.0," according to Microsoft:

- Web services
- NET remoting
- Distributed transactions
- Message queues

I've already discussed .NET remoting and web services, and these technologies have their own challenges when it comes to performance testing. Distributed transactions refer to the component-level interaction that is an increasingly common part of application design; whereas message queues are rarely troublesome as they tend to reside on the application mid-tier rather than the client.

Windows Presentation Foundation (WPF), on the other hand, is very much focused on the user interface (UI) making use of the new features of the Windows Vista operating system and the latest version of the .NET framework (3.5 at time of writing). In performance testing terms, WPF will probably have little impact as it will be the underlying middleware changes introduced by Web 2.0. WCF will likely have the most potential for challenges to scripting technology.

Oddball Application Technologies: Help, My Load Testing Tool Won't Record It!

There will always be applications that cannot be captured by automated performance testing tools. Frequently this is because the application vendor declines to release information about the inner workings of the application or does not provide an Application Programming Interface (API) that will allow an external tool to "hook" the application so that it can be recorded. There may also be challenges involving encryption and compression of data.

However, if you are quite certain that your performance testing tool should work with the target application technology, then read on.

Before Giving Up in Despair . . .

Before conceding defeat, try the following suggestions to eliminate possible performance tool or environment configuration issues.

On Windows operating systems, check you are hooking the correct executable (EXE) file
: Sometimes the executable file you need to "hook" with your performance testing tool is not the one initiated by starting the target application. In these situations the application will work fine but your performance testing tool will fail to record anything. Use the Windows Task Manager to check what executables are actually started by the application; you may find that a second executable is lurking in the background, handling all the client communication. If so then hooking into this executable may achieve successful capture.

Proxy problems
: If your performance testing tool uses proxy substitution to record HTTP traffic, then check to see that you are using the correct proxy settings. These are typically defined within your browser client and specify the Internet Protocol (IP) address or host name of the proxy server together with the ports to be used for communication and any traffic exceptions that should bypass the proxy server. Most performance testing tools that use this approach will attempt to automatically choose the correct settings for you, but sometimes they get it wrong. You may have to manually change your browser proxy settings and then configure your performance tool appropriately. (Don't forget to record what the original settings were before you overwrite them!)

Can the .NET application be proxy-enabled?
: Applications written for .NET do not need a web server to communicate, so they may not actually use the client browser settings. In most cases it is fairly simple to "proxy-enable" these types of applications, so if the customer or development section is willing then this is a possible way forward.

Windows Firewall, IPsec, and other nasties

If you're convinced that everything is configured correctly but are still capturing nothing, check to see if Windows Firewall is turned on. It will (by design) usually prevent performance testing tools from accessing the target application. If you can turn the firewall off then your problems may well disappear.

Something else that you may run into is Internet Protocol Security (IPsec). From a technical perspective this is another form of traffic encryption (like SSL) but it works at layer 3 of the protocol stack rather than layer 4. In short, this means that an application need not be designed to make use of IPsec, which can simply be turned on between cooperating nodes (machines). However, by design IPsec is intended to *prevent* the replay of secure session traffic; hence the problem. Like Windows Firewall, IPsec can be enabled or disabled in the configuration section of Windows and in the configuration setup of your web or application server.

Antivirus software may also interfere with the capture process, and this may be more difficult to disable. It may be hardwired into the standard build of the application client machine.

Finally, check to see what other software is installed on the machine you're using to capture data. I have found, among other conflicts, that performance testing tools from different vendors installed on the same box can cause problems with the capture process. (Surely this is not by design!)

Alternatives to Capture at the Middleware Level

If all else fails you still have a number of choices.

One, you can apologize to the customer or company that it is not possible to performance test the application and quietly walk away.

Two, try recording transactions using a *functional* automated test tool that works at the presentation layer rather than the underlying middleware technology. These tools, as their name suggests, are designed for unit and functional testing of an application rather than for generating load, so the downside of this approach is that you can generally configure only a few virtual users per injection PC. Hence you will need a large number of machines to create even a modest load. You will also find that combining functional and performance testing scripts in the same run is not enabled by the majority of performance tool vendors.

Another approach is to convince your customer to deploy the application over thin-client technology like Citrix. Several tool vendors provide support for the ICA protocol, and Citrix generally doesn't care what technology an application is written in. This strategy may involve the purchase of Citrix and an appropriate number of user licenses, so be prepared to provide a convincing business case for return on investment!

Manual Scripting

If no capture solution is feasible then there is still the option of manually creating the transaction scripts required to carry out performance testing. Some performance testing tools provide a link to popular Integrated Development Environments (IDEs), such as Eclipse for Java and Microsoft's Visual Studio for .NET. This link allows the user to drop code directly from an application into the performance testing tool for the purpose of building transaction scripts. Often templates are provided to make this task easier and to keep the resulting script code within the syntax required by the performance testing tool.

In most cases manual scripting will greatly increase the time (and cost) of building transaction scripts and will complicate the process of ongoing maintenance. However, it may be the only alternative to no performance testing at all.

APPENDIX A

Transaction Examples

This appendix contains some examples of transactions taken from a real project, showing the parameters you generally need to script the transaction using an automated performance testing tool.

Table A-1 demonstrates the sort of detail you should provide for each transaction to be included in your performance test. The columns in the table refer to the following information.

- *Step*. Order of events in the transaction.
- *Action*. What the user interaction is with the application client.
- *Test Data*. The data entered by the user and whether this will come from an external file DATAPOOL refers to an external file of data typically is CSV format.
- *System Time*. The expected time taken for the system to respond during transaction capture.
- *User Think Time*. The amount of time taken by a user to react to what has been displayed by the application in response to the *Action* for the *Step.*
- *Checkpoint Name*. The name given to the *Action* that will be analyzed separately during performance test execution and post-test.

TABLE A-1. Sample transaction script

Step	Action	Test data	System time	User think time	Checkpoint name
1	Select *Search Locate→Partner Search Locate* from the main menu.			1 second	
2	Display *Partner Search / Locate* screen (Open)		1 second		Checkpoint 01
3	Select *Person* radio button and enter Surname and Forename on search screen and press *Search* button	*Forename*: From DATAPOOL *Surname*: From DATAPOOL		30 seconds	
4	Display search results		2 seconds		Checkpoint 02
5	Select partner record from *Search Results*	Select first record in returned results		10 seconds	
6	Load *Role/Relationships* information (Search)	Select Partner from the drop-down list	2 seconds		Checkpoint 03
7	Navigate to *Partner* and press *GO* button.				
8	Display *Partner Personal Details* screen (Open)		1 second		Checkpoint 04
9	Assess Partner Personal Details			20 seconds	
10	Click on the *Address & Comms* tab			0.25 seconds	
11	Load *Address / Communication Details* (Open)		0.25 seconds		Checkpoint 05
12	Click on *New Address* button			0.25 seconds	
13	Display *Search For An Address* details canvas		0.25 seconds		Checkpoint 06
14	Enter new address details (details should not exist on database) and press *SEARCH*	*Post Code*: From DATAPOOL *Address Line One*: From DATAPOOL *Address Line Two*: From DATAPOOL *Address Line Three*: From DATAPOOL		30 seconds	
15	Display *Address Details*		1 second		Checkpoint 07
16	Assess Address Details			5 seconds	
17	Click on *SAVE* button			0.25 seconds	
18	Display updated address details		4 seconds		Checkpoint 08

Step	Action	Test data	System time	User think time	Checkpoint name
19	Click on *EXIT* icon			0.25 seconds	
20	Click on *Search / Locate* screen displayed		1 second		Checkpoint 09
21	Press *EXIT* icon			0.25 seconds	
22	*Main Menu* screen displayed		1 second		Checkpoint 10

Table A-2 shows a schema for the input data required to accompany the example transaction. The columns in the table refer to the following information.

- *Field Number*. The order of the field within each data file record.
- *Description*. What the data in the *Field Number* represents. (The data type is invariably *string*.)

TABLE A-2. Sample transaction input data schema

Field Number	Description
1	*Forename*
2	*Surname*
3	*Post Code*
4	*Address Line One*
5	*Address Line Two*
6	*Address Line Three*

APPENDIX B

POC and Performance Test Quick Reference

This appendix contains a convenient quick reference, drawn from earlier chapters, of the steps required at each stage of planning and carrying out a Proof of Concept (POC), performance test execution, and performance test analysis.

The Proof of Concept

A Proof of Concept (POC) is an important prerequisite because it provides the following information (see Chapter 3 for details).

- Provides an opportunity for a technical evaluation of the performance testing tool against the target application.
- Identifies scripting data requirements.
- Allows assessment of scripting effort.
- Demonstrates the capabilities of the performance testing solution against the target application.

Proof of Concept Checklist

You should anticipate no more than a couple of days for completion, assuming that the environment and application are available from day one.

Prerequisites

The following should be in place before you set up the POC environment.

- A written set of success or exit criteria that you and the customer have agreed on as determining the success or failure of the POC.
- Access to a standard build workstation or client platform that meets the minimum hardware and software specification for your performance testing tool or solution; this machine must have the application client *and* any supporting software installed.
- Permission to install any monitoring software that may be required into the application landscape.
- Ideally, sole access to the application for the duration of the POC.
- Access to someone who is familiar with the application (i.e, a "power user") and can answer your usability questions as they arise.
- Access to an expert who is familiar with the application (i.e., a developer) in case you need an explanation of how the application architecture works at a middleware level.
- A user account that will allow correct installation of the performance testing software onto the standard build workstation and access to the application client.
- At least two sets of login credentials (if relevant) for the target application.
- Two sample transactions to use as a basis for the POC, a simple "read-only" operation as well as a complex transaction that updates the target data repository. These let you check that your transaction replay works correctly.

Process

- Record two instances of each sample transaction and compare the differences between them using whatever method is most expedient. Identifying what has changed between recordings of the same activity will highlight any runtime data requirements that need to be addressed.
- After identifying input and runtime data requirements and any modifications needed to the scripts, ensure that each transaction will replay correctly in single user *and* multiuser mode. Make sure that any database updates occur as expected and that there are no errors in the replay logs for your transactions. Make certain that any modifications you have made to the scripts are free from memory leaks and other undesirable behavior.

Deliverables

- The output of a POC should be a go/no-go assessment of the technical suitability of your performance testing tool for successfully scripting and replaying application transactions.
- You should have identified the input and runtime data requirements for the sample transactions and gained insight into the likely data requirements for the performance testing project.
- You should identify any script modifications required to ensure accurate replay and assess the typical time required to script an application transaction.

Performance Test Execution Checklist

See Chapter 3 for an expanded version of this material.

Activity Duration Guidelines

Most project time is spent capturing, validating, and implementing requirements. Remaining time is typically devoted to scripting the transactions and performance test execution and analysis.

- *Scripting performance test transactions:* Allow half a day per transaction.
- *Creating and validating performance test sessions or scenarios:* Typically one to two days' work.
- *Performance test execution:* Allow a minimum of five days.
- *Data collection (and software uninstall):* Allow one day.

Step 1: Pre-Engagement Requirements Capture

Gather performance requirements from all stakeholders. You will need this information to successfully create a project plan or statement of work (SOW):

- Deadlines available to complete performance testing, including the scheduled deployment date.
- Whether to use internal or external resources to perform the tests.
- Test environment design agreed. (An appropriate test environment will require longer to create than you estimate.)
- Ensure that a code freeze applies to the test environment within each testing cycle.
- Ensure that the test environment will not be affected by other user activity.
- All performance targets identified and agreed to by appropriate business stakeholders.
- The key application transactions identified, documented, and ready to script.
- Which parts of transactions should be monitored separately (i.e.,checkpointed).
- Identify the *input*, *target*, and *runtime* data requirements for the transactions that you select. Make sure that you can create enough test data of the correct type within the time frames of your testing project. Don't forget about data security and confidentiality.
- Performance tests identified in terms of number, type, transaction content, and virtual user deployment. You should also have decided on the think time, pacing, and injection profile for each transaction deployment.
- Identify and document server, application server, and network KPIs.
- Identify the deliverables from the performance test in terms of a report on the test's outcome versus the agreed performance targets.

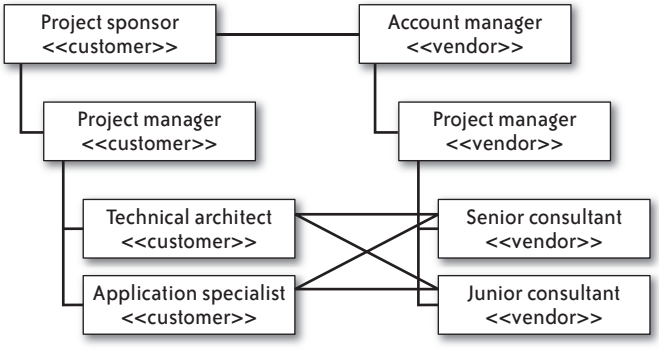


FIGURE B-1. Example performance testing team structure

- A procedure is defined for submission of performance defects discovered during testing cycles to development or the application vendor.

If your plan is to carry out the performance testing in-house then you will also need to address the following points related to the testing team:

- Do you have a dedicated performance testing team? At a minimum you will need a project manager and enough testing personnel (rarely are more than two needed) to handle the scale of the project. See Figure B-1.
- Make sure the team has the tools and resources it needs to performance test effectively.
- Ensure all team members are adequately trained in the testing tools to be used.

Given this information, proceed as follows:

- Develop a high-level plan that includes resources, time lines, and milestones based on these requirements.
- Develop a detailed performance test plan that includes all dependencies and associated time lines, detailed scenarios and test cases, workloads, and environment information.
- Include contingency for additional testing cycles and defect resolution if problems are found with the application during performance test execution.
- Include a risk assessment of not meeting schedule or performance targets.

With these actions underway, you can continue with each of the following steps. Not everything mentioned may be relevant to your particular testing requirements, but the order of events is important. (See Appendix E for an example MS Project-based performance testing project plan.)

Step 2: Test Environment Build

Strive to make your test environment a close approximation of the live environment. At a minimum it should reflect the server tier deployment of the live environment, and your target database should be populated realistically in terms of content *and* sizing. (This activity frequently takes much longer than expected.)

- Allow enough time to source equipment and to configure and build the environment.
- Take into account all deployment models (including LAN and WAN environments).
- Take external links into account, since they are a prime location for performance bottlenecks. Either use real links or create a realistic simulation of external communication.
- Provide enough load injection capacity for the scale of testing envisaged. Think about the locations where you will need to inject load from. If non-local load injector machines cannot be managed remotely then local personnel must be on hand at each remote location.
- Ensure that the application is correctly deployed into the test environment.
- Provide sufficient software licenses for the application and supporting software (e.g., Citrix or SAP).
- Ensure correct deployment and configuration of performance testing tools.
- Ensure correct deployment and configuration of KPI monitoring tool.

Step 3: Transaction Scripting

Proceed as follows for each transaction to be scripted.

- Identify the transaction *runtime* data requirements.

  NOTE

  POC data is limited in scope and thus insufficient.

- Confirm and apply transaction *input* data requirements (see Appendix A).
- Determine the transaction checkpoints that you will monitor separately for response time.
- Identify and apply any scripting changes required for the transaction to replay correctly.
- Ensure that the transaction replays correctly from both a single user and a multiuser perspective *before* including in a performance test. Make sure you can verify what happens on replay.

Step 4: Performance Test Build

For each performance test, consider the following points.

- Is it a baseline, load, stress, or soak test? A typical scenario is to have baseline tests for each transaction first in isolation as a single user and then up to the target maximum currency or throughput. Run isolation tests to identify and deal with any problems that occur, followed by a load test combining all transactions up to target concurrency. (You could then run stress and soak tests for the final testing cycles, followed perhaps by non-performance related tests.)
- Decide on how you will represent "think time" and "pacing" for each transaction included in the test (unless it is a stress test).
- For each transaction decide on how many load injector deployments you will make and how many "virtual users" should be assigned to each injection point.
- Decide on the injection profile for each load injector deployment: Big Bang, ramp-up, ramp-up/ramp-down with step, or delayed start. Your load test choice will likely involve a combination of Big Bang deployments to represent static load and one or more of the ramp variations to test for scalability. See Figure B-2.
- Will the test execute for a certain length of time or be halted by running out of data, reaching a certain number of transaction iterations, or user intervention?
- Do you need to spoof IP addresses to correctly exercise application load-balancing requirements? (If so, then you will need a list of valid IP addresses.)
- Do you need to simulate different baud rates? If so then confirm the different baud rates required. Any response time prediction or capacity modeling carried out prior to performance testing should have already given you valuable insight into how the application reacts to bandwidth restrictions.
- What runtime monitoring needs to be configured using the server and network KPIs that have already been set up? The actual monitoring software should have been deployed as part of the test environment build phase so you should already have a clear idea of exactly what you are going to monitor in the application landscape.
- If this is a web-based performance test what level of browser caching simulation do you need to provide? New user, active user, returning user? This will very much depend on the capabilities of your performance testing solution. See Chapter 5 for a discussion on caching simulation.
- Consider any effects that the application technology will have on your performance test design. For example, SAP performance tests that make use of the SAPGUI client will have a higher resource requirement than say a simple terminal emulator and will require more load injector machines to generate a given number of virtual users. Chapter 5 discusses additional considerations for SAP and other application technologies.

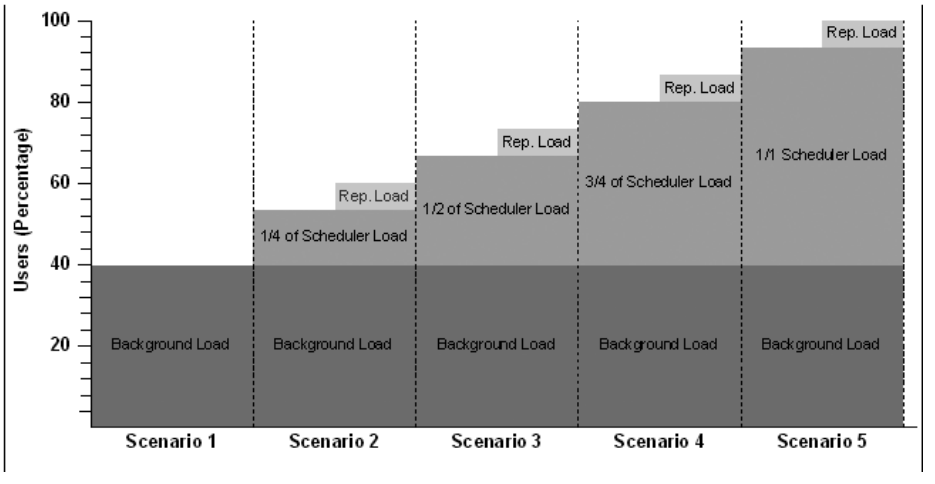


FIGURE B-2. Performance test plan using background (static) load and ramp-up injection profiles

Step 5: Performance Test Execution

Run and monitor your tests. Make sure that you carry out a "dress rehearsal" of each performance test as a final check that there are no problems accessing the application and that there are no problems with the test configuration.

This phase should be the most straightforward part of any performance testing project. You've done the hard work preparing the test environment, creating the transaction scripts, addressing the data requirements, and building the performance tests. In an ideal world performance test execution should be solely to validate your application performance targets. It should not become a bug-fixing exercise.

The only unknown is how many test cycles will be required before you achieve your performance testing goals.I wish I could answer this question for you but like many things in life this is in the lap of the gods , although if you've followed the suggested performance testing checklist religiously to this point your in pretty good shape to be granted a miracle!

- Execute "dress-rehearsal" tests to verify you have sufficient load injection capacity for the target concurrency. Unless your concurrency targets are very modest you should always check the upper limits of your load injectors.
- Execute baseline tests to establish "ideal" response-time performance. (This is typically a single user per transaction for a set period of time or a certain number of transaction iterations.)
- Execute load tests, ideally resetting target database content between executions. This test normally includes all transactions prorated among the target number of virtual users.

- Execute isolation tests to explore any problems revealed by load testing and then supply results to the developers or application vendor.
- Execute soak tests (if time allows) to reveal any memory leaks and problems related to high-volume transaction executions.
- Execute stress tests to generate data concerning future growth of transaction volume and application users.
- Execute any tests that are not performance related (e.g., different load balancing configurations).

Step 6 (Post-Test Phase): Analyze Results, Report, Retest

- Make sure that you capture and back up *all* data created as part of the performance testing project.
- Compare test results to performance targets set as part of project requirements; this will determine the project's success or failure.
- Document the results of the project. Your report should include sections that address each of the performance targets.
- Use the final results as baseline data for End User Experience (EUE) monitoring.

Analysis Checklist

The focus here is on analysis rather than execution. See Chapter 4 for more details.

Pre-Test Tasks

- Make sure that you have configured the appropriate server, application server, and network KPIs. Make sure there are no obstacles to installing and configuring agent software on the servers (unless you use remote monitoring).
- Decide on the final mix of performance tests to execute. This commonly includes baseline tests, load tests, isolation tests of any errors found, and soak and stress tests.
- Make sure that you can access the application from your injectors!
- Set any available automatic thresholds for performance targets as part of your performance test configuration.
- If your performance testing tool provides the capability, configure autocorrelation between transaction response time, concurrent virtual users, and server or network KPI metrics.
- If you are using third-party tools to provide some or all of your KPI monitoring, make sure that they are correctly configured *before* running any tests—ideally, include them in your dress rehearsal.

- Be sure you can integrate third-party data with the output of your performance testing tool. Unless your tools do this automatically, this can be a time-consuming task.

Tasks During Test Execution

- Periodically examine the performance of your load injectors to ensure that they are not becoming stressed.
- Document every test that you execute. At a minimum, record the following information:
  - — The name of the performance test execution file and the date and time of execution.
  - — A brief description of what the test comprised.
  - — The name of the results file (if any) associated with the current test execution.
  - — Any input data files associated with the performance test and which transactions they relate to.
  - — A brief description of any problems that occurred during the test.

  If your performance testing tool allows you to annotate the performance test configuration with comments, make use of this facility to include whatever information will help you to easily identify the test run.
- Things to be on the look-out for during execution include:
  - — Watch for the sudden appearance of errors. This frequently indicates that some limit has been reached within the application landscape; it can also mean that you've run out of data or that the operating system's default settings are interfering.
  - — Watch for a sudden drop in transaction throughput. This is a classic sign of trouble, particularly with web applications where the virtual users wait for a response from the web server. If your application is using links to external systems, check to ensure that none of these links is the cause of the problem.
  - — Watch for an ongoing reduction in available server memory. Available memory should decrease as more and more virtual users become active, but if the decrease continues after all users are active then you may have a memory leak.

Post-Test Tasks

- Collect all relevant data for each test execution. If you're relying on third-party tools to provide monitoring data, make sure that you preserve the files you need.
- Back up, onto a separate archive, all testing resources (scripts, input data files, test results).
- Your report should map results to the performance targets that were set as part of the pre-test requirements capture phase.

APPENDIX C

Automated Tool Vendors

This appendix provides a list of the leading automated tool vendors. I have also included entries for popular free tools where appropriate. The tool vendor market is fairly dynamic, but the list contains vendors who have the lion's share at the time of this writing. The list is in no particular order. For each section I have listed the tool vendor and web site followed by the product name and (where appropriate) a link for more information.

Application Performance Optimization

Compuware Ltd (www.compuware.com)
Application Vantage
http://www.compuware.com/solutions/appvantage.htm

Compuware Ltd (www.compuware.com)
Vantage Analyzer
http://www.compuware.com/products/vantage/vantageanalyzer.asp

Dynatrace (www.dynatrace.com)
DynaTrace
http://www.dynatrace.com/en/

Opnet (www.opnet.com)
Panorama
http://www.opnet.com/solutions/application_performance/panorama.html

Load and Performance Testing

Borland (www.borland.com)

Silk Performer

http://www.borland.com/us/products/silk/silkperformer/index.html

Compuware Ltd (www.compuware.com)

QALoad

http://www.compuware.com/products/qacenter/qaload.htm

Hewlett Packard (www.hp.com)

LoadRunner

http://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5E8_4000_100&jumpid=reg_R1002_USEN

Microsoft (www.microsoft.com)

Visual Studio Team System

http://www.microsoft.com/downloads/details.aspx?FamilyId=EEF7BB41-C686-4C9F-990B-F78ACE01C191&displaylang=en

Quotium (www.quotium.com)

Qtest

http://www.quotium.com/products/qtest-loadtesting.php

Free Tools

(The) Grinder

http://grinder.sourceforge.net/

JMeter

http://jakarta.apache.org/jmeter/

OpenSTA

http://www.opensta.org/

Web Remote Performance Testing and Monitoring

Keynote

http://www.keynote.com

Gomez

http://www.gomez.com

Functional Testing

Compuware Ltd (www.compuware.com)
TestPartner
http://www.compuware.com/375_eng_html.htm

Hewlett Packard (www.hp.com)
Quick Test Pro (QTP)

Requirements Management

Borland (www.borland.com)
Calibre-RM
http://www.borland.com/us/products/caliber/index.html

Compuware Ltd (www.compuware.com)
Optimal Trace
http://www.compuware.com/products/optimaltrace/default.htm

IBM (www.ibm.com)
Requisite-PRO
http://www-306.ibm.com/software/awdtools/reqpro/

Microsoft (http://www.microsoft.com)
Visual Studio Team System
http://www.microsoft.com/downloads/details.aspx?FamilyId=EEF7BB41-C686-4C9F-990B-F78ACE01C191&displaylang=en

Telelogic (www.telelogic.com)
DOORS
http://www.telelogic.com/Products/doors/index.cfm

Free Tools

TIGER PRO
http://www.seecforum.unisa.edu.au/SEECTools.html

TRUREQ
http://www.truereq.com/

Tracer
http://www.softreq.com/tracer/

Service-Oriented Architecture (SOA) Testing

Greenhat Consulting: http://www.greenhatconsulting.com/index.html

GH Tester

http://www.greenhatconsulting.com/ghtester/

APPENDIX D

Sample KPI Monitoring Templates

The following examples, reproduced here by kind permission of Compuware Corporation, demonstrate groups of common server KPIs that I favor in performance testing projects.

Windows Generic KPI: Counters

The first KPI template, shown in Figure D-1, provides a good indication of when a Windows server is under stress. This is the first level of monitoring that I use when performance testing a Windows OS server tier. You will probably recognize the metrics, since they taken from the Windows Performance Monitor (Perfmon) application. This is a nice mix of counters that monitor disk, memory, and CPU performance data together with some high-level network information measuring the number of errors encountered and data throughput in bytes per second.

I recommend that, in addition to using this template, you monitor the top ten processes in terms of CPU utilization and memory consumption. Identifying a process that is CPU- or memory-hungry is often your best link to the next level of KPI monitoring, where you need to drill down into specific software applications and components that are part of the application deployment.

This information is public domain so you could find it yourself, but it's nice to have it already available.

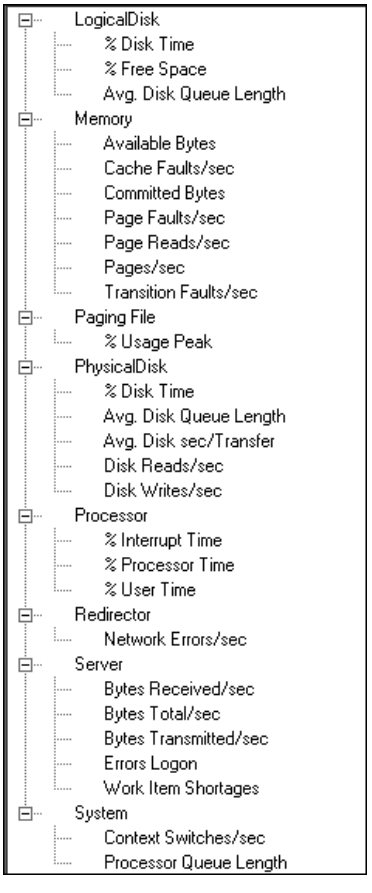


FIGURE D-1. Example generic server KPI monitoring template

Application Specific KPI Templates

Figures D-2 and D-3 focus on "level two" server KPI monitoring. Use these templates to monitor specific server software applications that are part of your application landscape, which might include Microsoft's Internet Information Server (IIS) or perhaps a "farm" of Citrix servers. Each application type has its own recommended set of counters to monitor, so refer to the vendor documentation to ensure that your KPI template contains the correct entries.

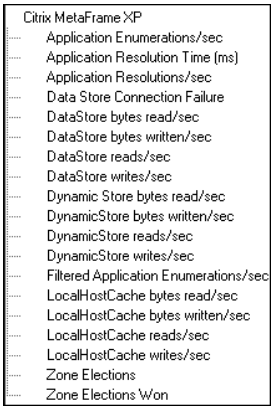


FIGURE D-2. Example server KPIs for Microsoft Internet Information Server (IIS)

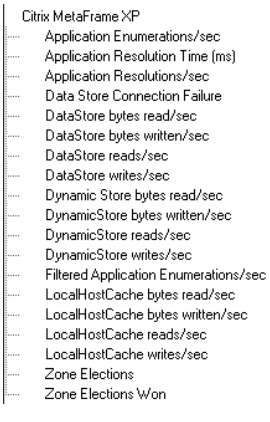


FIGURE D-3. Example Citrix server KPIs for Citrix server farm

APPENDIX E

Sample Project Plan

The following example is based on a typical performance project template in MS Project. The "profiling" section involves an additional transaction optimization step aimed at detecting and addressing—*prior* to performance test execution—application-related problems that may hurt scalability and response time.

> NOTE
> The MS Project file is available from the book's companion web site (*http://oreilly.com/catalog/9780596520663/*).

	Task Name	Duration	Start	Finish	Predecessors	Resource Names
1	**Customer**	**37.17 days**	**Thu 20/03/08**	**Mon 12/05/08**		
2	**Business Proposal**	**5 days**	**Thu 20/03/08**	**Wed 26/03/08**		
3	**Qualification**	**2 days**	**Thu 20/03/08**	**Fri 21/03/08**		
4	Receive customer questionaire	1 day	Thu 20/03/08	Thu 20/03/08		Tester
5	Choose right products	0.5 days	Fri 21/03/08	Fri 21/03/08	4	Tester
6	Check product technical fit	0.5 days	Fri 21/03/08	Fri 21/03/08	5	Tester
7	Execute proof of concept	0.5 days	Mon 24/03/08	Mon 24/03/08	3	Tester
8	Make a business proposal	2.5 days	Mon 24/03/08	Wed 26/03/08	7	Project manager | Customer :Lead Engineer | Customer: Application Specialist | : Account Manager | Tester
9	Execute scoping exercise with customer	4 days	Thu 27/03/08	Tue 01/04/08	2	Tester | Project Manager
10	**Start-up, profiling and scripting**	**12.17 days**	**Wed 02/04/08**	**Fri 18/04/08**	**9**	
11	Make full statement of work / projectplan	4 days	Wed 02/04/08	Mon 07/04/08		Project Manager | Tester
12	Sign-off statement of work	2 days	Tue 08/04/08	Wed 09/04/08	11	Customer Project Sponsor | Customer Project Manager | Project Manager | Account Manager
13	Project Kick-Off Meeting	1 day	Thu 10/04/08	Thu 10/04/08	12	Project Manager | :Account manager
14	Pre-requistes in place / based on statement of work	0 days	Thu 10/04/08	Thu 10/04/08	13	Tester | Customer : Lead Engineer
15	Install Software	2.67 days	Fri 11/04/08	Tue 15/04/08	14	
16	**Profilling based on trace analysis / iterative**	**1.25 days**	**Tue 15/04/08**	**Wed 16/04/08**	**15**	**Tester | Customer: Application Specialist**
17	**Capture Test Transactions**	**0.5 days**	**Tue 15/04/08**	**Wed 16/04/08**		
18	Transaction 01	0.5 days	Tue 15/04/08	Wed 16/04/08		
19	Analyse Application Traces	0.25 days	Wed 16/04/08	Wed 16/04/08	17	
20	Application DB Analysis	0.25 days	Wed 16/04/08	Wed 16/04/08	19	
21	Report of Application Traces	0.25 days	Wed 16/04/08	Wed 16/04/08	20	
22	Final Report Application Profiling	0 days	Wed 16/04/08	Wed 16/04/08	21	
23	**Scripting**	**1.25 days**	**Wed 16/04/08**	**Fri 18/04/08**	**16**	**Tester | Customer: Application Specialist**
24	**Produce Script**	**0.5 days**	**Wed 16/04/08**	**Thu 17/04/08**		
25	Transaction 01	0.5 days	Wed 16/04/08	Thu 17/04/08		
26	**Analyse Data Requirements**	**0.5 days**	**Thu 17/04/08**	**Thu 17/04/08**	**24**	
27	Transaction 01 - White Data Requirement Sp	0.5 days	Thu 17/04/08	Thu 17/04/08		
28	Produce Test Data	0.25 days	Thu 17/04/08	Fri 18/04/08	26	
29	Produce Scripts	0 days	Fri 18/04/08	Fri 18/04/08	28	
30	**Build test, collect and produce report**	**13 days**	**Fri 18/04/08**	**Wed 07/05/08**	**10**	
31	**Build Load Tests**	**1 day**	**Fri 18/04/08**	**Mon 21/04/08**		
32	**"Dress-rehearsal"**	**1 day**	**Fri 18/04/08**	**Mon 21/04/08**		**Tester**
33	Test Execution	0.5 days	Fri 18/04/08	Fri 18/04/08		
34	Test Analysis	0.25 days	Fri 18/04/08	Fri 18/04/08	33	
35	Test Reporting	0.25 days	Fri 18/04/08	Mon 21/04/08	34	
36	Dress-rehearsal Complete	0 days	Mon 21/04/08	Mon 21/04/08	35	
37	**Run tests based on test report / iteratief**	**5 days**	**Mon 21/04/08**	**Mon 28/04/08**	**31**	**Tester | Customer: Application Specialist**
38	**Execute Test Cycle 1 (Application)**	**1 day**	**Mon 21/04/08**	**Tue 22/04/08**		
39	Test Run 1 - Scenario 1	0.25 days	Mon 21/04/08	Mon 21/04/08		
40	Test Run 2 - Scenario 2	0.25 days	Mon 21/04/08	Mon 21/04/08	39	
41	Test Run 3 - Scenario 3	0.25 days	Mon 21/04/08	Mon 21/04/08	40	
42	Test Run 4 - Scenario 4	0.25 days	Mon 21/04/08	Tue 22/04/08	41	
43	Test Cycle 1 - Execution Complete	0 days	Tue 22/04/08	Tue 22/04/08	42	
44	**Analyse Test Cycle 1**	**0.5 days**	**Tue 22/04/08**	**Tue 22/04/08**	**38**	
45	Informal Headline Report Performance	0.5 days	Tue 22/04/08	Tue 22/04/08		
46	Application DB Analysis	0.25 days	Tue 22/04/08	Tue 22/04/08		
47	Test Cycle 1 - Analysis Complete	0 days	Tue 22/04/08	Tue 22/04/08		
48	**Produce Test Report**	**3.5 days**	**Tue 22/04/08**	**Mon 28/04/08**	**44**	
49	Test Tool - Analysis & Report	2 days	Tue 22/04/08	Thu 24/04/08		
50	Prepare & Publish Report	1.5 days	Thu 24/04/08	Mon 28/04/08	49	
51	Data Collection / Uninstall	5 days	Mon 28/04/08	Mon 05/05/08	37	Tester
52	**Produce Report**	**2 days**	**Mon 05/05/08**	**Wed 07/05/08**	**51**	
53	Produced Final Performance Test Report	1 day	Mon 05/05/08	Tue 06/05/08		Tester
54	Consolidate Final Report	1 day	Tue 06/05/08	Wed 07/05/08	53	Tester
55	Present Report	3 days	Wed 07/05/08	Mon 12/05/08	30	Tester | Project Manager

FIGURE E-1. Example MS Project performance test plan

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

About the Author

Originally hailing from Auckland, New Zealand, Ian Molyneaux ended up in IT purely by chance after applying for an interesting looking job advertised as "junior computer operator" in the mid-70s. The rest is history. Thirty years later, Ian has held many roles in IT. A techie at heart, he's shied away from anything management related. His current role is EMEA SME (Subject Matter Expert) for Application Performance Assurance at Compuware. Ian resides in Buckinghamshire, U.K., with his wife Sarah, daughter Sasha, and two cats, and is trying to get used to the idea of turning 49!

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The cover image is an original photograph. The cover fonts are Akzidenz Grotesk and Orator. The text font is Adobe's Meridien; the heading font is ITC Bailey.