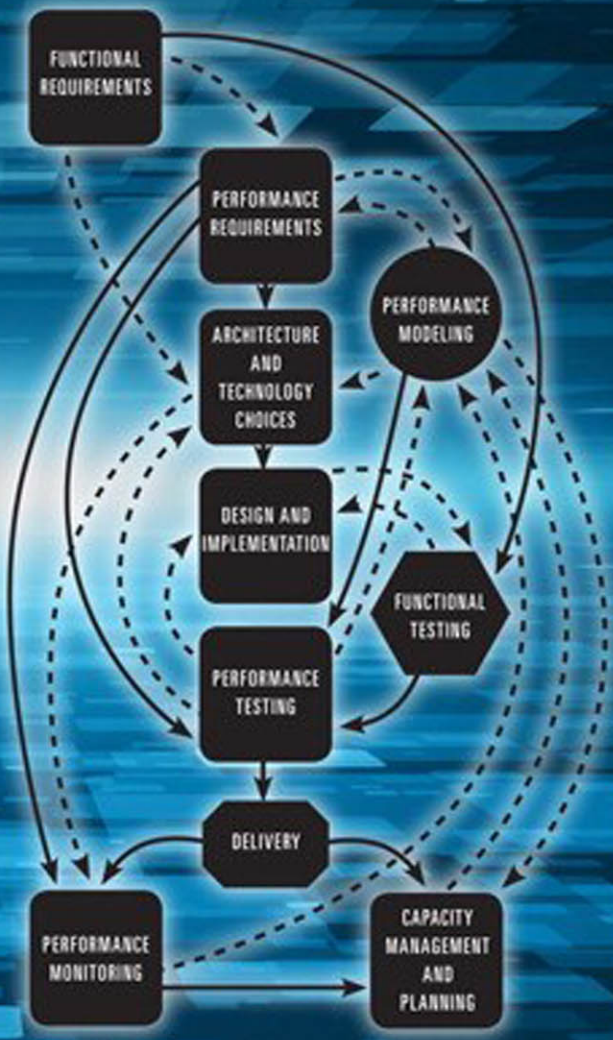
Foundations of Software and System Performance Engineering

Process, Performance Modeling, Requirements, Testing, Scalability, and Practice



André B. Bondi

Praise for *Foundations of Software and System Performance Engineering*

"If this book had only been available to the contractors building healthcare.gov, and they read and followed the lifecycle performance processes, there would not have been the enormous problems apparent in that health care application. In my 40-plus years of experience in building leading-edge software and hardware products, poor performance is the single most frequent cause of the failure or cancellation of a new, software-intensive project. Performance requirements are often neglected or poorly formulated during the planning and requirements phases of a project. Consequently, the software architecture and the resulting delivered system are unable to meet performance needs. This book provides the reader with the techniques and skills necessary to implement performance engineering at the beginning of a project and manage those requirements throughout the lifecycle of the product. I cannot recommend this book highly enough."

— Don Shafer, CSDP, Technical Fellow, Athens Group, LLC

"Well written and packed with useful examples, *Foundations of Software and System Performance Engineering* provides a thorough presentation of this crucial topic. Drawing upon decades of professional experience, Dr. Bondi shows how the principles of performance engineering can be applied in many different fields and disciplines."

— Matthew Scarpino, author of Programming the Cell Processor *and* OpenCL in Action

This page intentionally left blank

Foundations of Software and System Performance Engineering

This page intentionally left blank

Foundations of Software and System Performance Engineering

Process, Performance Modeling, Requirements, Testing, Scalability, and Practice

André B. Bondi

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Bondi, André B., author.
Foundations of software and system performance engineering : process, performance modeling, requirements, testing, scalability, and practice / André B. Bondi.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-83382-2 (pbk. : alk. paper)
1. Computer systems—Evaluation. 2. Computer systems—Reliability. 3. Computer software—Validation. 4. Computer architecture—Evaluation. 5. System engineering. I. Title.
QA76.9.E94B66 2015
005.1'4—dc23

2014020070

ISBN-13: 978-0-321-83382-2
ISBN-10: 0-321-83382-1

2 16

Executive Editor
Bernard Goodwin

Senior Development Editor
Chris Zahn

Managing Editor
John Fuller

Senior Project Editor
Kesel Wilson

Copy Editor
Barbara Wood

Indexer
Jack Lewis

Proofreader
Andrea Fox

Editorial Assistant
Michelle Housley

Cover Designer
Alan Clements

Compositor
LaurelTech

In memory of my father, Henry S. Bondi,
who liked eclectic solutions to problems,
and of my violin teacher, Fritz Rikko,
who taught me how to analyze and debug.

À tous qui ont attendu.

This page intentionally left blank

Contents

This page intentionally left blank

Preface

The performance engineering of computer systems and the systems they control concerns the methods, practices, and disciplines that may be used to ensure that the systems provide the performance that is expected of them. Performance engineering is a process that touches every aspect of the software lifecycle, from conception and requirements planning to testing and delivery. Failure to address performance concerns at the beginning of the software lifecycle significantly increases the risk that a software project will fail. Indeed, performance is the single largest risk to the success of any software project. Readers in the United States will recall that poor performance was the first sign that healthcare.gov, the federal web site for obtaining health insurance policies that went online in late 2013, was having a very poor start. News reports indicate that the processes and steps recommended in this book were not followed during its development and rollout. Performance requirements were inadequately specified, and there was almost no performance testing prior to the rollout because time was not available for it. This should be a warning that adequate planning and timely scheduling are preconditions for the successful incorporation of performance engineering into the software development lifecycle. "Building and then tuning" is an almost certain recipe for performance failure.

Scope and Purpose

The performance of a system is often characterized by the amount of time it takes to accomplish a variety of prescribed tasks and the number of times it can accomplish those tasks in a set time period. For example:

- A government system for selling health insurance policies to the general public, such as healthcare.gov, would be expected to determine an applicant's eligibility for coverage, display available options, and confirm the choice of policy and the

premium due within designated amounts of time regardless of how many applications were to be processed within the peak hour.

- An online stock trading system might be expected to obtain a quote of the current value of a security within a second or so and execute a trade within an even shorter amount of time.
- A monitoring system, such as an alarm system, is expected to be able to process messages from a set of sensors and display corresponding status indications on a console within a short time of their arrival.
- A web-based news service would be expected to retrieve a story and display related photographs quickly.

This is a book about the practice of the performance engineering of software systems and software-controlled systems. It will help the reader address the following performance-related questions concerning the architecture, development, testing, and sizing of a computer system or a computer-controlled system:

- What capacity should the system have? How do you specify that capacity in both business-related and engineering terms?
- What business, social, and engineering needs will be satisfied by given levels of throughput and system response time?
- How many data records, abstract objects, or representations of concrete, tangible objects must the system be able to manage, monitor, and store?
- What metrics do you use to describe the performance your system needs and the performance it has?
- How do you specify the performance requirements of a system? Why do you need to specify them in the first place?
- How can the resource usage of a system be measured? How can you verify the accuracy of the measurements?
- How can you use mathematical models to predict a system's performance? Can the models be used to predict the performance if an application is added to the system or if the transaction rate increases?
- How can mathematical models of performance be used to plan performance tests and interpret the results?

- How can you test performance in a manner that tells you if the system is functioning properly at all load levels and if it will scale to the extent and in the dimensions necessary?
- What can poor performance tell you about how the system is functioning?
- How do you architect a system to be scalable? How do you specify the dimensions and extent of the scalability that will be required now or in the future? What architecture and design features undermine the scalability of a system?
- Are there common performance mistakes and misconceptions? How do you avoid them?
- How do you incorporate performance engineering into an agile development process?
- How do you tell the performance story to management?

Questions like these must be addressed at every phase of the software lifecycle. A system is unlikely to provide adequate performance with a cost-effective configuration unless its architecture is influenced by well-formulated, testable performance requirements. The requirements must be written in measurable, unambiguous, testable terms. Performance models may be used to predict the effects of design choices such as the use of scheduling rules and the deployment of functions on one or more hosts. Performance testing must be done to ensure that all system components are able to meet their respective performance needs, and to ensure that the end-to-end performance of the system meets user expectations, the owner's expectations, and, where applicable, industry and government regulations. Performance requirements must be written to help the architects identify the architectural and technological choices needed to ensure that performance needs are met. Performance requirements should also be used to determine how the performance of a system will be tested.

The need for performance engineering and general remarks about how it is practiced are presented in Chapter 1. Metrics are needed to describe performance quantitatively. A discussion of performance metrics is given in Chapter 2. Once performance metrics have been identified, basic analysis methods may be used to make predictions about system performance, as discussed in Chapter 3. The anticipated workload can be quantitatively described as in Chapter 4, and performance requirements can be specified. Necessary attributes of performance

requirements and best practices for writing and managing them are discussed in Chapters 5 through 7. To understand the performance that has been attained and to verify that performance requirements have been met, the system must be measured. Techniques for doing so are given in Chapter 8. Performance tests should be structured to enable the evaluation of the scalability of a system, to determine its capacity and responsiveness, and to determine whether it is meeting throughput and response time requirements. It is essential to test the performance of all components of the system before they are integrated into a whole, and then to test system performance from end to end before the system is released. Methods for planning and executing performance tests are discussed in Chapter 9. In Chapter 10 we discuss procedures for evaluating the performance of a system and the practice of performance modeling with some examples. In Chapter 11 we discuss ways of describing system scalability and examine ways in which scalability is enhanced or undermined. Performance engineering pitfalls are examined in Chapter 12, and performance engineering in an agile context is discussed in Chapter 13. In Chapter 14 we consider ways of communicating the performance story. Chapter 15 contains a discussion of where to learn more about various aspects of performance engineering.

This book does not contain a presentation of the elements of probability and statistics and how they are applied to performance engineering. Nor does it go into detail about the mathematics underlying some of the main tools of performance engineering, such as queueing theory and queueing network models. There are several texts that do this very well already. Some examples of these are mentioned in Chapter 15, along with references on some detailed aspects of performance engineering, such as database design. Instead, this book focuses on various steps of the performance engineering process and the link between these steps and those of a typical software lifecycle. For example, the chapters on performance requirements engineering draw parallels with the engineering of functional requirements, and the chapter on scalability explains how performance models can be used to evaluate it and how architectural characteristics might affect it.

Audience

This book will be of interest to software and system architects, requirements engineers, designers and developers, performance testers, and product managers, as well as their managers. While all stakeholders should benefit from reading this book from cover to cover, the following stakeholders may wish to focus on different subsets of the book to begin with:

- Product owners and product managers who are reluctant to make commitments to numerical descriptions of workloads and requirements will benefit from the chapters on performance metrics, workload characterization, and performance requirements engineering.
- Functional testers who are new to performance testing may wish to read the chapters on performance metrics, performance measurement, performance testing, basic modeling, and performance requirements when planning the implementation of performance tests and testing tools.
- Architects and developers who are new to performance engineering could begin by reading the chapters on metrics, basic performance modeling, performance requirements engineering, and scalability.

This book may be used as a text in a senior- or graduate-level course on software performance engineering. It will give the students the opportunity to learn that computer performance evaluation involves integrating quantitative disciplines with many aspects of software engineering and the software lifecycle. These include understanding and being able to explain why performance is important to the system being built, the commercial and engineering implications of system performance, the architectural and software aspects of performance, the impact of performance requirements on the success of the system, and how the performance of the system will be tested.

This page intentionally left blank

Acknowledgments

This book is based in part on a training course entitled Foundations of Performance Engineering. I developed this course to train performance engineering and performance testing teams at various Siemens operating companies. The course may be taught on its own or, as my colleague Alberto Avritzer and I have done, as part of a consulting engagement. When teaching the course as part of a consulting engagement, one may have the opportunity to integrate the client's performance issues and even test data into the class material. This helps the clients resolve the particular issues they face and is effective at showing how the material on performance engineering presented here can be integrated into their software development processes.

One of my goals in writing this book was to relate this practical experience to basic performance modeling methods and to link performance engineering methods to the various stages of the software lifecycle. I was encouraged to write it by Dr. Dan Paulish, my first manager at Siemens Corporate Research (now Siemens Corporation, Corporate Technology, or SC CT); by Prof. Len Bass, who at the time was with the Software Engineering Institute in Pittsburgh; and by Prof. C. Murray Woodside of Carleton University in Ottawa. We felt that there was a teachable story to tell about the practical performance issues I have encountered during a career in performance engineering that began during the heyday of mainframe computers.

My thinking on performance requirements has been strongly influenced by Brian Berenbach, who has been a driving force in the practice of requirements engineering at SC CT. I would like to thank my former AT&T Labs colleagues, Dr. David Hoeflin and Dr. Richard Oppenheim, for reading and commenting on selected chapters. We worked together for many years as part of a large group of performance specialists. My experience in that group was inspiring and rewarding. I would also like to thank Dr. Alberto Avritzer of SC CT for many lively discussions on performance engineering.

I would like to thank the following past and present managers and staff at SC CT for their encouragement in the writing of this book.

Between them, Raj Varadarajan and Dr. Michael Golm read all of the chapters of the book and made useful comments before submission to the publisher.

Various Siemens operating units with whom I have worked on performance issues kindly allowed me to use material I had prepared for them in published work. Ruth Weitzenfeld, SC CT's librarian, cheerfully obtained copies of many references. Patti Schmidt, SC CT's in-house counsel, arranged for permission to quote from published work I had prepared while working at Siemens. Dr. Yoni Levi of AT&T Labs kindly arranged for me to obtain AT&T's permission to quote from a paper I had written on scalability while working there. This paper forms the basis for much of the content of Chapter 11.

I would like to thank my editors at Addison-Wesley, Bernard Goodwin and Chris Zahn, and their assistant, Michelle Housley, for their support in the preparation of this book. It has been a pleasure to work with them. The copy editor, Barbara Wood, highlighted several points that needed clarification. Finally, the perceptive comments of the publisher's reviewers, Nick Rozanski, Don Shafer, and Matthew Scarpino, have done much to make this a better book.

About the Author

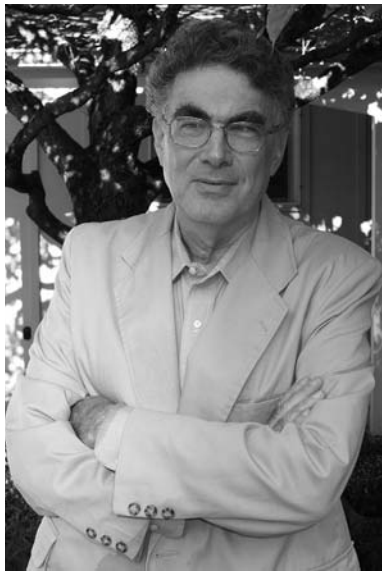


Photo by Rixt Bosma, www.rixtbosma.nl

André B. Bondi is a Senior Staff Engineer working in performance engineering at Siemens Corp., Corporate Technology, in Princeton, New Jersey. He has worked on performance issues in several domains, including telecommunications, conveyor systems, finance systems, building surveillance, railways, and network management systems. Prior to joining Siemens, he held senior performance positions at two start-up companies. Before that, he spent more than ten years working on a variety of performance and operational issues at AT&T Labs and its predecessor, Bell Labs. He has taught courses in performance, simulation, operating systems principles, and computer architecture at the University of California, Santa Barbara. Dr. Bondi holds a PhD in computer science from Purdue University, an MSc in statistics from University College London, and a BSc in mathematics from the University of Exeter.

This page intentionally left blank

Chapter 1

Why Performance Engineering? Why Performance Engineers?

This chapter describes the importance of performance engineering in a software project and explains the role of a performance engineer in ensuring that the system has good performance upon delivery. Overviews of different aspects of performance engineering are given.

1.1 Overview

The performance of a computer-based system is often characterized by its ability to perform defined sets of activities at fast rates and with quick response time. Quick response times, speed, and scalability are highly desired attributes of any computer-based system. They are also competitive differentiators. That is, they are attributes that distinguish a system from other systems with like functionality and make it more attractive to a prospective buyer or user.

If a system component has poor performance, the system as a whole may not be able to function as intended. If a system has poor performance, it will be unattractive to prospective users and buyers. If the project fails as a result, the investment in building the system will have been wasted. The foregoing is true whether the system is a command and control system, a transaction-based system, an information retrieval system, a video game, an entertainment system, a system for displaying news, or a system for streaming media.

The importance of performance may be seen in the following examples:

- A government-run platform for providing services on a grand scale must be able to handle a large volume of transactions from the date it is brought online. If it is not able to do so, it will be regarded as ineffective. In the United States, the federal web site for applying for health insurance mandated by the Affordable Care Act, healthcare.gov, was extremely slow for some time after it was made available to the public. According to press reports and testimony before the United States Congress, functional, capacity, and performance requirements were unclear. Moreover, the system was not subjected to rigorous performance tests before being brought online [Eilperin2013].
- An online securities trading system must be able to handle large numbers of transactions per second, especially in a volatile market with high trading volume. A brokerage house whose system cannot do this will lose business very quickly, because slow execution could lead to missing a valuable trading opportunity.
- An online banking system must display balances and statements rapidly. It must acknowledge transfers and the transmission of payments quickly for users to be confident that these transactions have taken place as desired.
- Regulations such as fire codes require that audible and visible alarms be triggered within 5 or 10 seconds of smoke being detected. In many jurisdictions, a building may not be used if the fire alarm system cannot meet this requirement.
- A telephone network must be able to handle large numbers of call setups and teardowns per second and provide such services as call forwarding and fraud detection within a short time of each call being initiated.

- A rail network control system must be able to monitor train movements and set signals and switches accordingly within very short amounts of time so that trains are routed to their correct destinations without colliding with one another.
- In combat, a system that has poor performance may endanger the lives or property of its users instead of endangering those of the enemy.
- A medical records system must be able to pull up patient records and images quickly so that retrieval will not take too much of a doctor's time away from diagnosis and treatment.

The foregoing examples show that performance is crucial to the correct functioning of a software system and of the application it controls. As such, performance is an attribute of system quality that presents significant business and engineering risks. In some applications, such as train control and fire alarm systems, it is also an essential ingredient of safety. Performance engineering mitigates these risks by ensuring that adequate attention is paid to them at every stage of the software lifecycle, while improving the capacity of systems, improving their response times, ensuring their scalability, and increasing user productivity. All of these are key competitive differentiators for any software product.

Despite the importance of system performance and the severe risk associated with inattentiveness to it, it is often ignored until very late in the software development cycle. Too often, the view is that performance objectives can be achieved by tuning the system once it is built. This mindset of "Build it, then tune it" is a recurring cause of the failure of a system to meet performance needs [SmithWilliams2001]. Most performance problems have their root causes in poor architectural choices. For example:

- An architectural choice could result in the creation of foci of overload.
- A decision is made that a set of operations that could be done in parallel on a multiprocessor or multicore host will be handled by a single thread. This would result in the onset of a software bottleneck for sufficiently large loads.

One of the possible consequences of detecting a performance issue with an architectural cause late in the software lifecycle is that a considerable amount of implementation work must be undone and redone.

This is needlessly expensive when one considers that the problem could have been avoided by performing an architectural review. This also holds for other quality attributes such as reliability, availability, and security.

1.2 The Role of Performance Requirements in Performance Engineering

To ensure that performance needs are met, it is important that they be clearly specified in requirements early in the software development cycle. Early and concise specifications of performance requirements are necessary because:

- Performance requirements are potential drivers of the system architecture and the choice of technologies to be used in the system's implementation. Moreover, many performance failures have their roots in poor architectural choices. Modification of the architecture before a system is implemented is cheaper than rebuilding a slow system from scratch.
- Performance requirements are closely related to the contractual expectations of system performance negotiated between buyer and seller, as well as to any relevant regulatory requirements such as those for fire alarm systems.
- The performance requirements will be reflected in the performance test plan.
- Drafting and reviewing performance requirements force the consideration of trade-offs between execution speed and system cost, as well as between execution speed and simplicity of both the architecture and the implementation. For instance, it is more difficult to design and correctly code a system that uses multithreading to achieve parallelism in execution than to build a single-threaded implementation.
- Development and/or hardware costs can be reduced if performance requirements that are found to be too stringent are relaxed early in the software lifecycle. For example, while a 1-second average response time requirement may be desirable, a 2-second requirement may be sufficient for business or engineering needs. Poorly specified performance

requirements can lead to confusion among stakeholders and the delivery of a poor-quality product with slow response times and inadequate capacity.

- If a performance issue that cannot be mapped to explicit performance requirements emerges during testing or production, stakeholders might not feel obliged to correct it.

We shall explore the principles of performance requirements in Chapter 5.

1.3 Examples of Issues Addressed by Performance Engineering Methods

Apart from mitigating business risk, performance engineering methods assist in answering a variety of questions about a software system. The performance engineer must frequently address questions related to capacity. For example:

- Can the system carry the peak load? The answer to this question depends on whether the system is adequately sized, and on whether its components can interact gracefully under load.
- Will the system cope with a surge in load and continue to function properly when the surge abates? This question is related to the reliability of the system. We do not want it to crash when it is most needed.
- What will be the performance impacts of adding new functionality to a system? To answer this question, we need to understand the extra work associated with each invocation of the functionality, and how often that functionality is invoked. We also need to consider whether the new functionality will adversely affect the performance of the system in its present form.
- Will the system be able to carry an increase in load? To answer this question, we must first ask whether there are enough resources to allow the system to perform at its current level.
- What is the performance impact of increasing the size of the user base? Answering this question entails understanding the memory and secondary storage footprints per user as well as in

total, and then being able to quantify the increased demand for memory, processing power, I/O, and network bandwidth.

- Can the system meet customer expectations or engineering needs if the average response time requirement is 2 seconds rather than 1 second? If so, it might be possible to build the system at a lower cost or with a simpler architecture. On the other hand, the choice of a simpler architecture could adversely affect the ability to scale up the offered load later, while still maintaining the response time requirement.
- Can the system provide the required performance with a cost-effective configuration? If it cannot, it will not fare well in the marketplace.

Performance can have an effect on the system's functionality, or its perceived functionality. If the system does not respond to an action before there is a timeout, it may be declared unresponsive or down if timeouts occur in a sufficiently large number of consecutive attempts at the action.

The performance measures of healthy systems tend to behave in a predictable manner. Deviations from predictable performance are signs of potential problems. Trends or wild oscillations in the performance measurements may indicate that the system is unstable or that a crash will shortly occur. For example, steadily increasing memory occupancy indicates a leak that could bring the system down, while oscillating CPU utilization and average response times may indicate that the system is repeatedly entering deadlock and timing out.

1.4 Business and Process Aspects of Performance Engineering

Ensuring the performance of a system entails initial and ongoing investment. The investment is amply rewarded by reductions in business risk, increased system stability, and system scalability. Because performance is often the single biggest risk to the success of a project [Bass2007], reducing this risk will make a major contribution to reducing the total risk to the project overall.

The initial performance engineering investments in a software project include

- Ensuring that there is performance engineering expertise on the project, perhaps including an individual designated as the lead performance engineer
- Drafting performance requirements
- Planning lab time for performance measurement and performance testing
- Acquiring and preparing performance measurement tools, load generation tools, and analysis and reporting tools to simplify the presentation and tracking of the results of the performance tests

Incorporating sound performance engineering practices into every aspect of the software development cycle can considerably reduce the performance risk inherent in the development of a large, complicated system. The performance process should be harmonized with the requirements, architectural, development, and testing phases of the development lifecycle. In addition to the steps just described, the performance engineering effort should include

1. A review of the system architecture from the standpoints of performance, reliability, and scalability
2. An evaluation of performance characteristics of the technologies proposed in the architecture specification, including quick performance testing of any proposed platforms [MBH2005]
3. Incremental performance testing following incremental functional testing of the system, followed by suggestions for architectural and design revisions as needed
4. Retesting to overcome the issues revealed and remedied as a result of the previous step

Performance engineering methods can also be used to manage cost-effective system growth and added functionality. For an existing system, growth is managed by building a baseline model based on measurements of resource usage and query or other work unit rates taken at runtime. The baseline model is combined with projected traffic rates to determine resource requirements using mathematical models and other methods drawn from the field of operations research [LZGS1984, Kleinrock1975, Kleinrock1976, MenasceAlmeida2000].

We now turn to a discussion of the various disciplines and techniques a performance engineer can use to perform his or her craft.

1.5 Disciplines and Techniques Used in Performance Engineering

The practice of performance engineering draws on many disciplines and skills, ranging from the technological to the mathematical and even the political. Negotiating, listening, and writing skills are essential for successful performance engineering, as is the case for successful architects and product owners. The set of original undergraduate major subjects taken by performance engineers the author has met includes such quantitative disciplines as mathematics, physics, chemical engineering, chemistry, biology, electrical engineering, statistics, economics, and operations research, as well as computer science. Those who have not majored in computer science will need to learn about such subjects as operating systems design, networking, and hardware architecture, while the computer scientists may need to acquire additional experience with working in a quantitative discipline.

To understand resource usage and information flow, the performance engineer must have at least a rudimentary knowledge of computer systems architecture, operating systems principles, concurrent programming principles, and software platforms such as web servers and database management systems. In addition, the performance engineer must have a sound grasp of the technologies and techniques used to measure resource usage and traffic demands, as well as those used to drive transactions through a system under test.

To understand performance requirements and the way the system will be used, it is necessary to know something about its domain of application. The performance and reliability needs of financial transaction systems, fire alarm systems, network management systems, conveyor belts, telecommunications systems, train control systems, online news services, search engines, and multimedia streaming services differ dramatically. For instance, the performance of fire alarm systems is governed by building and fire codes in the jurisdictions where the systems will be installed, while that of a telephone system may be governed by international standards. The performance needs of all the systems mentioned previously may be driven by commercial considerations such as competitive differentiation.

Because performance is heavily influenced by congestion, it is essential that a performance engineer be comfortable with quantitative analysis methods and have a solid grasp of basic statistics, queueing theory, and simulation methods. The wide variety of computer

technologies and the evolving set of problem domains mean that the performance engineer should have an eclectic set of skills and analysis methods at his or her disposal. In addition, it is useful for the performance engineer to know how to analyze large amounts of data with tools such as spreadsheets and scripting languages, because measurement data from a wide variety of sources may be encountered. Knowledge of statistical methods is useful for planning experiments and for understanding the limits of inferences that can be drawn from measurement data. Knowledge of queueing theory is useful for examining the limitations of design choices and the potential improvements that might be gained by changing them.

While elementary queueing theory may be used to identify limits on system capacity and to predict transaction loads at which response times will suddenly increase [DenningBuzen1978], more complex queueing theory may be required to examine the effects of service time variability, interarrival time variability, and various scheduling rules such as time slicing, preemptive priority, nonpreemptive priority, and cyclic service [Kleinrock1975, Kleinrock1976].

Complicated scheduling rules, load balancing heuristics, protocols, and other aspects of system design that are not susceptible to queueing analysis may be examined using approximate queueing models and/or discrete event simulations, whose outputs should be subjected to statistical analysis [LawKelton1982].

Queueing models can also be used in sizing tools to predict system performance and capacity under a variety of load scenarios, thus facilitating what-if analysis. This has been done with considerable commercial success. Also, queueing theory can be used to determine the maximum load to which a system should be subjected during performance tests once data from initial load test runs is available.

The performance engineer should have some grasp of computer science, software engineering, software development techniques, and programming so that he or she can quickly recognize the root causes of performance issues and negotiate design trade-offs between architects and developers when proposing remedies. A knowledge of hardware architectures, including processors, memory architectures, network technologies, and secondary storage technologies, and the ability to learn about new technologies as they emerge are very helpful to the performance engineer as well.

Finally, the performance engineer will be working with a wide variety of stakeholders. Interactions will be much more fruitful if the performance engineer is acquainted with the requirements drafting

and review processes, change management processes, architecture and design processes, and testing processes. The performance engineer should be prepared to work with product managers and business managers. He or she will need to explain choices and recommendations in terms that are related to the domain of application and to the trade-offs between costs and benefits.

1.6 Performance Modeling, Measurement, and Testing

Performance modeling can be used to predict the performance of a system at various times during its lifecycle. It can be used to characterize capacity; to help understand the impact of proposed changes, such as changes to scheduling rules, deployment scenarios, technologies, and traffic characteristics; or to predict the effect of adding or removing workloads. Deviations from the qualitative behavior predicted by queueing models, such as slowly increasing response times or memory occupancy when the system load is constant or expected to be constant, can be regarded as indications of anomalous system behavior. Performance engineers have used their understanding of performance models to identify software flaws; software bottlenecks, especially those occurring in new technologies that may not yet be well understood [ReeserHariharan2000]; system malfunctions (including the occurrence of deadlocks); traffic surges; and security violations. This has been done by examining performance measurement data, the results of simulations, and/or queueing models [AvBonWey2005, AvTanJaCoWey2010]. Interestingly, the principles that were used to gain insights into performance in these cases were independent of the technologies used in the system under study.

Performance models and statistical techniques for designing experiments can also be used to help us plan and interpret the results of performance tests.

An understanding of rudimentary queueing models will help us determine whether the measurement instrumentation is yielding valid values of performance metrics.

Pilot performance tests can be used to identify the ranges of transaction rates for which the system is likely to be lightly, moderately, or heavily loaded. Performance trends with respect to load are useful for predicting capacity and scalability. Performance tests at loads near or

above that at which any system resource is likely to be saturated will be of no value for predicting scalability or performance, though they can tell us whether the system is likely to crash when saturated, or whether the system will recover gracefully once the load is withdrawn. An understanding of rudimentary performance models will help us to design performance tests accordingly.

Methodical planning of experiments entails the identification of factors to be varied from one test run to the next. Fractional replication methods help the performance engineer to choose telling subsets of all possible combinations of parameter settings to minimize the number of experiments that must be done to predict performance.

Finally, the measurements obtained from performance tests can be used as the input parameters of sizing tools (based on performance models) that will assist in sizing and choosing the configurations needed to carry the anticipated load to meet performance requirements in a cost-effective manner.

1.7 Roles and Activities of a Performance Engineer

Like a systems architect, a performance engineer should be engaged in all stages of a software project. The performance engineer is frequently a liaison between various groups of stakeholders, including architects, designers, developers, testers, product management, product owners, quality engineers, domain experts, and users. The reasons for this are:

- The performance of a system affects its interaction with the domain.
- Performance is influenced by every aspect of information flow, including
    - The interactions between system components
    - The interactions between hardware elements and domain elements
    - The interactions between the user interface and all other parts of the system
    - The interactions between component interfaces

When performance and functional requirements are formulated, the performance engineer must ensure that performance and scalability requirements are written in verifiable, measurable terms, and that they are linked to business and engineering needs. At the architectural

stage, the performance engineer advises on the impacts of technology and design choices on performance and identifies impediments to smooth information flow. During design and development, the performance engineer should be available to advise on the performance characteristics and consequences of design choices and scheduling rules, indexing structures, query patterns, interactions between threads or between devices, and so on. During functional testing, including unit testing, the performance engineer should be alerted if the testers feel that the system is too slow. This can indicate a future performance problem, but it can also indicate that the system was not configured properly. For example, a misconfigured IP address could result in an indication by the protocol implementation that the targeted host is unresponsive or nonexistent, or in a failure of one part of the system to connect with another. It is not unusual for the performance engineer to be involved in diagnosing the causes of these problems, as well as problems that might appear in production.

The performance engineer should be closely involved in the planning and execution of performance tests and the interpretation of the results. He or she should also ensure that the performance instrumentation is collecting valid measurement data and generating valid loads. Moreover, it is the performance engineer who supervises the preparation of reports of performance tests and measurements in production, explains them to stakeholders, and mediates negotiations between stakeholders about necessary and possible modifications to improve performance.

If the performance of a system is found to be inadequate, whether in testing or in production, he or she will be able to play a major role in diagnosing the technical cause of the problem. Using the measurement and testing methods described in this book, the performance engineer works with testers and architects to identify the nature of the cause of the problem and with developers to determine the most cost-effective way to fix it. Historically, the performance engineer's first contact with a system has often been in "repairman mode" when system performance has reached a crisis point. It is preferable that performance issues be anticipated and avoided during the early stages of the software lifecycle.

The foregoing illustrates that the performance engineer is a performance advocate and conscience for the project, ensuring that performance needs are anticipated and accounted for at every stage of the development cycle, the earlier the better [Browne1981]. Performance advocacy includes the preparation of clear summaries of

performance status, making recommendations for changes, reporting on performance tests, and reporting on performance issues in production. Thus, the performance engineer should not be shy about blowing the whistle if a major performance problem is uncovered or anticipated. The performance reports should be concise, cogent, and pungent, because stakeholders such as managers, developers, architects, and product owners have little time to understand the message being communicated. Moreover, the performance engineer must ensure that graphs and tables tell a vivid and accurate story.

In the author's experience, many stakeholders have little training or experience in quantitative methods unless they have worked in disciplines such as statistics, physics, chemistry, or econometrics before joining the computing profession. Moreover, computer science and technology curricula seldom require the completion of courses related to performance evaluation for graduation. This means that the performance engineer must frequently play the role of performance teacher while explaining performance considerations in terms that can be understood by those trained in other disciplines.

1.8 Interactions and Dependencies between Performance Engineering and Other Activities

Performance engineering is an iterative process involving interactions between multiple sets of stakeholders at many stages of the software lifecycle (see Figure 1.1). The functional requirements inform the specification of the performance requirements. Both influence the architecture and the choice of technology. Performance requirements may be formulated with the help of performance models. The models are used to plan performance tests to verify scalability and that performance requirements have been met. Performance models may also be used in the design of modifications. Data gathered through performance monitoring and capacity planning may be used to determine whether new functionality or load may be added to the system.

The performance engineer must frequently take responsibility for ensuring that these interactions take place. None of the activities and skills we have mentioned is sufficient for the practice of performance engineering in and of itself.

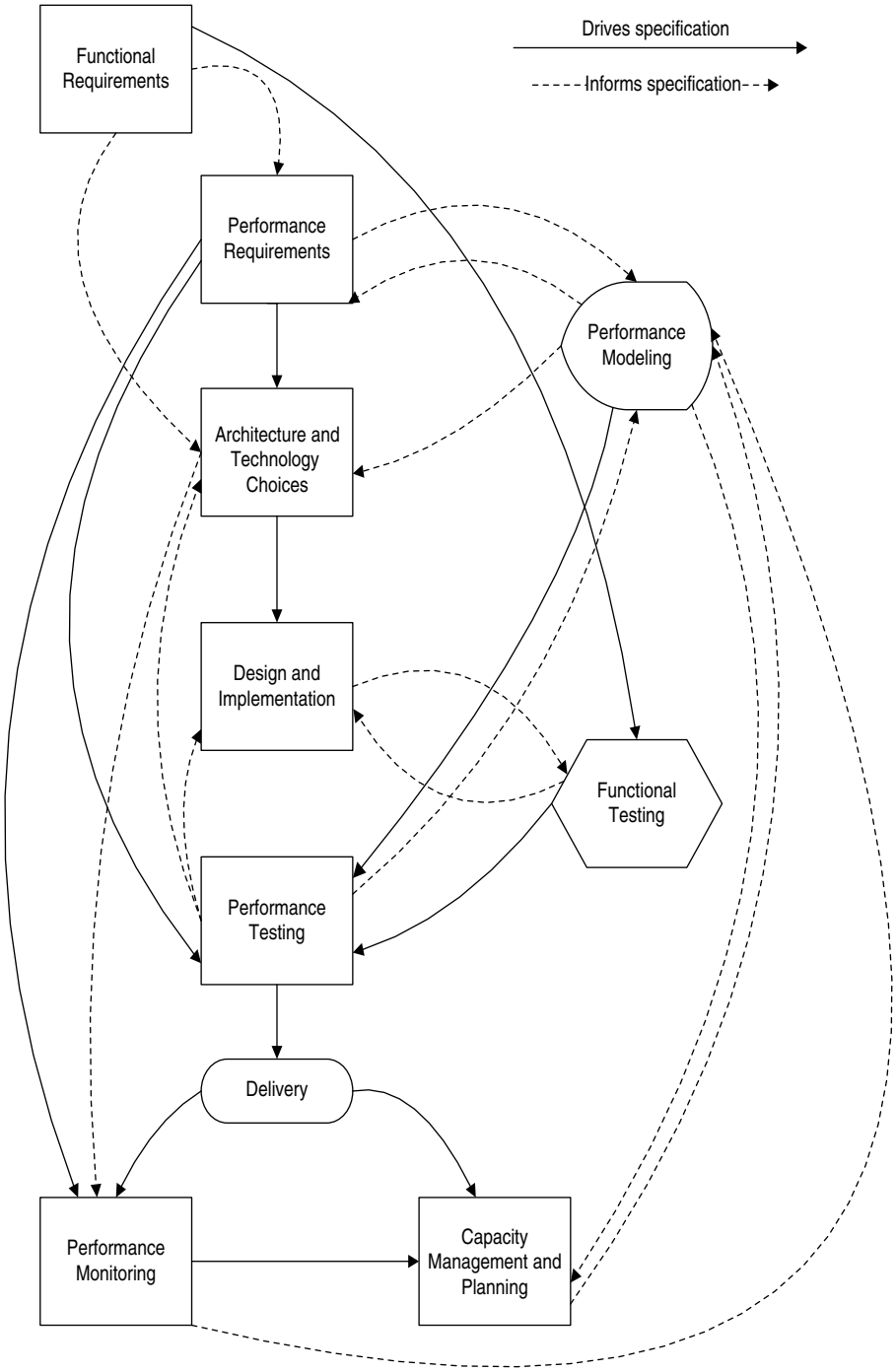


Figure 1.1 *Interactions between performance engineering activities and other software lifecycle activities*

1.9 A Road Map through the Book

Performance metrics are described in Chapter 2. One needs performance metrics to be able to define the desired performance characteristics of a system, and to describe the characteristics of the performance of an existing system. In the absence of metrics, the performance requirements of a system can be discussed only in vague terms, and the requirements cannot be specified, tested, or enforced.

Basic performance modeling and analysis are discussed in Chapter 3. We show how to establish upper bounds on system throughput and lower bounds on system response time given the amount of time it takes to do processing and I/O. We also show how rudimentary queueing models can be used to make predictions about system response time when a workload has the system to itself and when it is sharing the system with other workloads.

In Chapter 4 we explore methods of characterizing the workload of a system. We explain that workload characterization involves understanding what the system does, how often it is required to do it, why it is required to do it, and the performance implications of the nature of the domain of application and of variation in the workload over time.

Once the workload of the system has been identified and understood, we are in a position to identify performance requirements. The correct formulation of performance requirements is crucial to the choice of a sound, cost-effective architecture for the desired system. In Chapter 5 we describe the necessary attributes of performance requirements, including linkage to business and engineering needs, traceability, clarity, and the need to express requirements unambiguously in terms that are measurable, testable, and verifiable. These are preconditions for enforcement. Since performance requirements may be spelled out in contracts between a buyer and a supplier, enforceability is essential. If the quantities specified in a performance requirement cannot be measured, the requirement is deficient and unenforceable and should either be flagged as such or omitted. In Chapter 6 we discuss specific types of the ability of a system to sustain a given load, the metrics used to describe performance requirements, and performance requirements related to networking and to specific domains of application. In Chapter 7 we go into detail about how to express performance requirements clearly and how they can be managed.

One must be able to measure a system to see how it is functioning, to identify hardware and software bottlenecks, and to determine whether it is meeting performance requirements. In Chapter 8 we

describe performance measurement tools and instrumentation that can help one do this. Instrumentation that is native to the operating system measures resource usage (e.g., processor utilization and memory usage) and packet traffic through network ports. Tools are available to measure activity and resource usage of particular system components such as databases and web application servers. Application-level measurements and load drivers can be used to measure system response times. We also discuss measurement pitfalls, the identification of incorrect measurements, and procedures for conducting experiments in a manner that helps us learn about system performance in the most effective way.

Performance testing is discussed in Chapter 9. We show how performance test planning is linked to both performance requirements and performance modeling. We show how elementary performance modeling methods can be used to interpret performance test results and to identify system problems if the tests are suitably structured. Among the problems that can be identified are concurrent programming bugs, memory leaks, and software bottlenecks. We discuss suitable practices for the documentation of performance test plans and results, and for the organization of performance test data.

In Chapter 10 we use examples to illustrate the progression from system understanding to model formulation and validation. We look at cases in which the assumptions underlying a conventional performance model might deviate from the properties of the system of interest. We also look at the phases of a performance modeling study, from model formulation to validation and performance prediction.

Scalability is a desirable attribute of systems that is frequently mentioned in requirements without being defined. In the absence of definitions, the term is nothing but a buzzword that will engender confusion at best. In Chapter 11 we look in detail at ways of characterizing the scalability of a system in different dimensions, for instance, in terms of its ability to handle increased loads, called *load scalability*, or in terms of the ease or otherwise of expanding its structure, called *structural scalability*. In this chapter we also provide examples of cases in which scalability breaks down and discuss how it can be supported.

Intuition does not always lead to correct performance engineering decisions, because it may be based on misconceptions about what scheduling algorithms or the addition of multiple processors might contribute to system performance. This is the reason Chapter 12, which contains a discussion of performance engineering pitfalls, appears in

this book. In this chapter we will learn that priority scheduling does not increase the processing capacity of a system. It can only reduce the response times of jobs that are given higher priority than others and hence reduce the times that these jobs hold resources. Doubling the number of processors need not double processing capacity, because of increased contention for the shared memory bus, the lock for the run queue, and other system resources. In Chapter 12 we also explore pitfalls in system measurement, performance requirements engineering, and other performance-related topics.

The use of agile development processes in performance engineering is discussed in Chapter 13. We will explore how agile methods might be used to develop a performance testing environment even if agile methods have not been used in the development of the system as a whole. We will also learn that performance engineering as part of an agile process requires careful advance planning and the implementation of testing tools. This is because the time constraints imposed by short sprints necessitate the ready availability of load drivers, measurement tools, and data reduction tools.

In Chapter 14 we explore ways of learning, influencing, and telling the performance story to different sets of stakeholders, including architects, product managers, business executives, and developers.

Finally, in Chapter 15 we point the reader to sources where more can be learned about performance engineering and its evolution in response to changing technologies.

1.10 Summary

Good performance is crucial to the success of a software system or a system controlled by software. Poor performance can doom a system to failure in the marketplace and, in the case of safety-related systems, endanger life, the environment, or property. Performance engineering practice contributes substantially to ensuring the performance of a product and hence to the mitigation of the business risks associated with software performance, especially when undertaken from the earliest stages of the software lifecycle.

This page intentionally left blank

Chapter 2

Performance Metrics

In this chapter we explore the following topics:

- The essential role and desirable properties of performance metrics
- The distinction between metrics based on sample statistics and metrics based on time averaging
- The need for performance metrics to inform us about the problem domain, and how they occur in different types of systems
- Desirable properties of performance metrics

2.1 General

It is essential to describe the performance of a system in terms that are commonly understood and unambiguously defined. Performance should be defined in terms that aid the understanding of the system from both engineering and business perspectives. A great deal of time can be wasted because of ambiguities in quantitative descriptions, or because system performance is described in terms of quantities that cannot be measured in the system of interest. If a quantitative description in a contract is ambiguous or if a quantity cannot be measured, the contract cannot be enforced, and ill will between customer and supplier may arise, leading to lost business or even to litigation.

Performance is described in terms of quantities known as *metrics*. A metric is defined as a standard of measurement [Webster1988].

The following are examples of metrics that are commonly used in evaluating the performance of a system:

- The *response time* is a standard measure of how long it takes a system to perform a particular activity. This metric is defined as the difference between the time at which an activity is completed and the time at which it was initiated. The average response time is the arithmetic mean of the response times that have been observed.
- The *average device utilization* is a standard measure of the proportion of time that a device, such as a CPU, disk, or other I/O device, is busy. For this metric to be valid, the length of time during which the average was taken should be stated, because averages taken over long intervals tend to hide fluctuations, while those taken over short intervals tend to reveal them.
- The *average throughput* of a system is the rate at which the system completes units of a particular activity.

The start and end times of the measurement interval over which an average is taken should be stated, to enable the identification of a relationship between the values of performance metrics, to enable verification of the measurements, and to provide a context for the measurements, such as the load during the time at which they were taken.

The utilitarian philosopher Jeremy Bentham advocated economic policies that would result in the greatest happiness for the greatest number [Thomson1950, p. 30]. It is easy to define a standard metric for the greatest number, but not for happiness. The user's happiness with a system cannot be easily characterized by a metric or set of metrics. *Webster's Ninth New Collegiate Dictionary* cites the statement that "no metric exists that can be applied directly to happiness." Debate about metrics for happiness, the use of such metrics, and what constitutes happiness continues to this day [Mossburg2009, Bok2010].

The values of some metrics are obtainable by direct measurement, or else by arithmetic manipulation of measurements or of the values of other metrics. For example:

- The average response time during a particular time interval $[0,T]$ is the arithmetic mean of the response times that were observed during that interval. It is a sample statistic whose value is given by

$$\bar{R} = \frac{1}{N(T)} \sum_{i=1}^{N(T)} R_i \tag{2.1}$$

where $N(T) \geq 1$ is the number of individual observations of response times seen in $[0,T]$. Individual response times can be obtained by direct measurement of transactions. For example, if a transaction is generated by a load generation client, its response time may be obtained by associating an initiation time stamp with the transaction when it is created and comparing that time stamp with the current time when the transaction has been completed. The average response time is obtained by adding the individual response times to a variable that has been set to zero before measurements were accumulated, and then dividing by the number of recorded response times at the end of the run.

- The average utilization of a device may be obtained indirectly from the average service time, if known, and the number of jobs served during the observation period. Alternatively, it can be obtained by direct measurement of the system and is usually available directly from the host operating system. It is an example of a time-averaged statistic, which is not the same thing as a sample statistic. Let us define the function $U(t)$ for a single resource as follows:

$$U(t) = \begin{cases} 1 \text{ if the resource is busy at time } t \\ 0 \text{ if the resource is idle at time } t \end{cases} \tag{2.2}$$

Then, the average utilization of the resource during $[0,T]$ is given by

$$\bar{U} = \frac{1}{T}\int_0^T U(t)\,dt \tag{2.3}$$

In many computer systems, the processor utilization is obtained directly by accumulating the amount of time the CPU is not busy, dividing that by the length of the observation interval, and then subtracting that result from unity. The time the CPU is not busy is the time it spends executing an idle loop. The idle loop is a simple program of the form "jump to me" that is executed when there is no other work for the processor(s) to do. Thus, it executes at the lowest level of priority if the operating system supports priority scheduling.

For multiprocessor systems, one must distinguish between the utilizations of individual processors and the utilization of the entire set of processors. Both are available from standard operating system utilities. In Linux- and UNIX-based systems, these are available from *mpstat* and

sar, or from system calls. In Windows XP, Windows 7, and related systems, they may be obtained from *perfmon* or from system calls. The utilizations of the individual processors would be obtained from counters linked to individual idle loops. The utilization $u_{CPU,i}$ is the measured utilization of the ith processor for $1 \le i \le p$. The average utilization of the entire set of processors in a host is given by

$$U_{CPU} = \frac{1}{p}\sum_{i=1}^{p} u_{CPU,i} \tag{2.4}$$

Both the utilizations of the individual processors (or, in the case of multicore processors, the individual cores) and the total overall utilizations of the processors can be obtained in the Linux, Windows, and UNIX environments.

It is also possible to specify a metric whose value might not be obtainable, even though the formula for computing it is well known. For example, the unbiased estimator of the variance of the response time corresponding to the data used in equation (2.1) is given by

$$S^2_{N(T)-1} = \frac{\sum_{k=1}^{N(T)} R_k^2 - N(T)\bar{R}^2}{N(T)-1} \tag{2.5}$$

This metric is obtainable only if $N(T) \ge 2$ and the sum of squares of the individual response times has been accumulated during the experiment, or if the individual response times have been saved. The former incurs the cost of $N(T)$ multiplications and additions, while the latter incurs a processing cost that is unacceptable if secondary storage (e.g., disk) is used, or a memory cost and processing cost that may be unacceptable if main memory or flash memory is used. This example illustrates the point that collecting data incurs a cost that should not be so high as to muddy the measurements by slowing down the system under test, a phenomenon known as *confounding*. As we shall see in the chapters on measurement and performance testing, load generators should never be deployed on the system under test for this reason. Similarly, the resource costs associated with measurements should themselves be measured to ensure that their interference with the work under test is minimal. Otherwise, there is a serious risk that resource usage estimates will be spuriously increased by the costs of the measurements themselves. Measurement tools must never be so intrusive as to cause this type of problem.

2.2 Examples of Performance Metrics

Having seen some examples of system characteristics that can be measured and some that cannot, let us examine the useful properties of performance metrics in detail.

A performance metric should inform us about the behavior of the system in the context of its domain of application and/or in the context of its resource usage. What is informative depends on the point of view of a particular stakeholder as well as on the domain itself. For example, an accountant may be interested in the monthly transaction volume of a system. By contrast, an individual user of the system may be interested only in the response time of the system during a period of peak usage. This means that the system must be engineered to handle a set number of transactions per second in the peak hour. The latter quantity is of interest to the performance engineer. It is of interest to the accountant only to the extent that it is directly related to the total monthly volume. If the two metrics are to be used interchangeably, there must be a well-known understanding and agreement about the relationship between them. Such an example exists in telephony. In the latter part of the twentieth century, it was understood that about 10% of the calls on a weekday occur during the busy hour. In the United States, this was true of both local call traffic and long-distance traffic observed concurrently in multiple time zones [Rey1983]. It is also understood that the number of relevant business days in a month is about 22. Thus, the monthly traffic volume would be approximately 22 times the busy hour volume, divided by 10%. For example, if 50,000 calls occur in a network in the busy hour, the number of calls per month could be approximately estimated as $22 \times 50{,}000/10\% = 22 \times 500{,}000 = 11{,}000{,}000$ calls. As we shall see in Chapter 5, this relationship must be stated in the performance requirements if any requirement relies on it.

It is sometimes useful to distinguish between user experience metrics and resource usage metrics. Examples of user experience metrics include

- The response time of a system, as measured between two well-defined events such as a mouse click to launch a transaction and the time at which the response appears.
- The waiting time of a server or device. This is the server's response time minus its service time.

Examples of resource usage metrics include

- Processor utilization, the percent of time that the processor is executing. This can include the utilization in user mode, and the utilization in system, privileged, or kernel mode, when operating system functions are being executed.
- Bandwidth utilization. This is the number of bits per second at which data is being transmitted, divided by the maximum available bandwidth.
- Device utilizations of any kind.
- Memory occupancy.

Other metrics may be related to revenue, such as the number of calls per hour handled at a telephone switch or the number of parcels per hour handled by a conveyor system in a parcel distribution center or post office.

2.3 Useful Properties of Performance Metrics

Lilja has identified the following useful properties of performance metrics [Lilja2000]:

- *Linearity.* If a metric changes by a given ratio, so should the performance it quantifies. For example, if the average response time is decreased by 50%, the user should experience a drop in response time of 50% on average. Similarly, if the measured CPU utilization drops by 50%, this means that the processor load has actually dropped by 50%. The same is true for throughput and offered load. By contrast, loudness is measured in decibels (dB), which are on a base-10 logarithmic scale rather than a linear one. The Richter scale for earthquakes is also on a base-10 logarithmic scale. This means that a magnitude 5 earthquake is ten times as forceful as a magnitude 4 earthquake. Thus, the metrics for noise volume and earthquake force are nonlinear. It is therefore more difficult to intuitively grasp the impacts of changes in their value. A magnitude 5 earthquake could cause disproportionately more destruction than one of magnitude 4. On the other hand, the linearity of processor utilization means that reducing it by half enables the offered load on the processors to be doubled, provided (and only

provided) that no other system resource will be saturated as a result.

- *Reliability.* A metric is considered reliable if System A outperforms System B whenever the metric indicates that it does. Execution time is a reliable indicator of performance at a specific load, but clock speed and instruction rate are not, because a processor with a higher clock speed might be connected to a memory bank with a slower cycle time than a processor with a lower clock speed, while instruction sets can vary from one processor to the next.
- *Repeatability.* If the same experiment is run two or more times on the same system under identical configurations, the metrics that are obtained from the experiments should have the same value each time.
- *Ease of measurement.* A metric is particularly useful if it can be obtained directly from system measurements or simply calculated from them. If the collection of a metric is difficult, the procedure for doing so may be prone to error. The result will be bad data, which is worse than no data because incorrect inferences about performance could be drawn from it.
- *Consistency.* A metric is said to be consistent if its definition and the units in which it is expressed are the same across all systems and configurations. This need not be true of the instruction rate (often expressed in MIPS, or million instructions per second) because a processor with a reduced instruction set (RISC) may require more instructions to execute the same action as a processor with a complex instruction set (CISC). For example, a Jump Subroutine (JSR) instruction, which pushes a procedure's activation record onto the runtime stack, may be present in a stack-oriented CISC machine but not in a RISC machine. Hence, a procedure call takes more instructions in a RISC machine than in a CISC machine with a JSR instruction. Using MIPS to compare RISC and CISC machines may be misleading.
- *Independence.* A metric should not reflect the biases of any stakeholder. Otherwise, it will not be trusted. For this reason, MIPS should not be used to compare processors with different instruction sets.

More than one metric may be needed to meaningfully describe the performance of a system. For online transaction processing systems, such as a brokerage system or an airline reservation system, the metrics of interest are the response times and transaction rates for each type of transaction, usually measured in the hour when the traffic is heaviest, sometimes called the *peak hour* or the *busy hour*. The transaction loss rate, that is, the fraction of submitted transactions that were not completed for whatever reason, is another measure of performance. It should be very low. For packet switching systems, one may require a packet loss rate of no more than 10^{-6} or 10^{-8} to avoid retransmissions. For high-volume systems whose transaction loss rates may be linked to loss of revenue or even loss of life, the required transaction loss rate might be lower still.

We see immediately that one performance metric on its own is not sufficient to tell us about the performance of a system or about the quality of service it provides. The tendency to fixate on a single number or to focus too much on a single metric, sometimes termed *mononumerosis*, can result in a poor design or purchasing decision, because the chosen metric may not reflect critical system characteristics that are described by other metrics. For a home office user, choosing a PC based on processor clock speed alone may result in disappointing performance if the main memory size is inadequate or if the memory bus is too slow. If the memory is too small, there is a substantial risk of too much paging when many applications are open or when larger images are being downloaded, no matter how fast the CPU. If the memory bus is too slow, memory accesses and I/O will be too slow. The lessons we draw from this are:

1. One cannot rely on a single number to tell the whole story about the performance of a system.
2. A description of the performance requirements of a system requires context, such as the offered load (job arrival rate), a description of what is to be done, and the desired response time.
3. A performance requirement based on an ill-chosen single number is insufficiently specific to tell us how well the system should perform.

2.4 Performance Metrics in Different Domains

We now illustrate how performance metrics arrive in different problem domains. When the corresponding systems are computer controlled, many of the domain-specific metrics correspond to performance

metrics commonly used in computer systems such as online transaction processing systems.

2.4.1 Conveyor in a Warehouse

Figure 2.1 shows a conveyor system in a warehouse. Parcels arrive at the point marked *Induct* and are routed to one of the four branches. The conveyor system is controlled by program logic controllers (PLCs) that communicate with each other and with the outside world via a tree-structured network. Each parcel has a bar code that is read by a scanner. Once a parcel's bar code is read, the routing database (MFC DB) is queried to determine the path the parcel should follow through the system. The PLCs at the junctions, known as *diverts*, route the parcels accordingly as they pass sensors placed strategically alongside the conveyor. The PLCs also monitor the status of various pieces of the conveyor and send notifications of malfunctions or that all is well to a status monitoring system. The entire system must be able to stop abruptly if anyone pulls a red emergency cord.

Domain-related performance metrics include

- The number of parcels per second at each point on the conveyor, including at the bar code scanner
- The speed of the conveyor, for example, 2 meters per second
- The distance between parcels, and the average length of each parcel
- The query rate of the parcel routing database

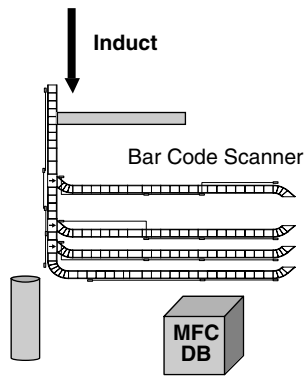


Figure 2.1 *A conveyor system*

- The response time of the parcel routing database
- Message rates and latencies between PLCs
- Volume of the status monitoring traffic, along with the corresponding message delivery times
- Status notification time, that is, the time from a change of status to its notification on a display
- The time from the pulling of an emergency cord to the shutdown of the entire conveyor line

Some of these metrics are related to one another. The database query rate is a function of the number of parcels passing the bar code scanner each second. This in turn is determined by the length of the parcels in the direction of travel, the distance between them, and the speed of the conveyor. The message rates between the PLCs are determined by the parcel throughput and by the volume of status monitoring traffic. The latencies between the PLCs depend on the characteristics of the PLCs, the message rates through them, the available network bandwidth, the bandwidth utilization, and the characteristics and topology of the network connecting them. For the system to function correctly, the PLCs, the routing database, and the local area network connecting them must be fast enough to set the diverts to route the parcels correctly before each one arrives there. The faster the conveyor moves the parcels and/or the closer the parcels are together, the shorter the combined time that is allowed for this.

2.4.2 Fire Alarm Control Panel

One or more fire alarm control panels are connected to a network of smoke sensors, alarm bells and flashers, door-closing devices, ventilation dampers, and the like. The control panels may be connected to each other. Control panels are installed at locations known to the local fire department and contain display panels whose function is also known to the fire department. Each control panel has a display and buttons for triggering resets, the playing of evacuation announcements, and so on. The National Fire Alarm Code [NFPA2007, Section 6.8.1.1] specifies that "actuation of alarm notification appliances or emergency voice communications, fire safety function, and annunciation at the protected premises shall occur within 10 seconds after the activation of an initiating device" [NFPA2007]. The performance metrics of interest include, but may not be limited to,

- The time from the activation of an initiating device (such as a smoke detector or a pull handle) to the time to sound the first notification appliance (such as strobe lights, gongs, horns, and sirens)
- The time to process the first A alarms coming from the first D devices to the time to activate N notification appliances
- The rates at which devices send messages to an alarm panel
- The rates at which alarm control panels in different parts of the protected premises send messages to each other
- The time to reset a fire alarm control panel once the reset button has been pushed

Notice that the set of metrics of interest here is determined by local codes and regulations as well as by the nature of the supporting configuration. Notice also that the set of metrics does not include average response times of any kind. In this application, it is not the average that matters, but the time to complete a specific set of tasks. We shall discuss this more in Section 2.7.

2.4.3 Train Signaling and Departure Boards

In a metropolitan train network, trains move from one segment of track to the next and from one station to the next. The position of each train is displayed on a rail network map in the signal control room or signal box. The times to the arrivals of the next few trains and their destinations are displayed on monitors on each platform in each station, as well as on monitors elsewhere in each station, such as at the tops and bottoms of escalators. The information displayed differs from station to station. Metrics describing performance objectives might include

- The time from a train moving past a sensor to the completion of the corresponding information update on the monitors in the stations
- The time from a train moving past a sensor to the completion of the corresponding information update on the rail network map

Metrics describing load and performance requirement drivers might include

- The number of train movements per minute.
- The speed of each train in each segment of track.

- The average time from arrival to departure of a train at each station, including the time for passengers to board and alight. The value of this metric depends on the station and the time of day. It will take longer for passengers to alight and board during the rush hour at Times Square in New York, Oxford Circus in London, or Marienplatz in Munich than at the ends of the corresponding lines at midnight.
- The number of trains en route or at intermediate stations.
- The number of stations to be notified about each train.
- The travel time between stations.

Some of these metrics are related to one another. One must take this into account when computing their values. For instance, the time between stations is a decreasing function of the train speed, provided there are no other variables to take into account.

2.4.4 Telephony

There is a long tradition of performance evaluation in the field of telecommunications. Teletraffic engineers use the terms *offered load* and *carried load* to distinguish between the rate at which calls or transactions are offered to the network and the rate at which they are actually accepted, or carried [Rey1983]. One would like these figures to be identical, but that is not always the case, particularly if the network is overloaded. In that case, the carried load may be less than the offered load. This is illustrated in Figure 2.2. The line $y = x$ corresponds to the entire offered load being carried. In this illustration, the offered load could be carried completely only at rates of 300 transactions per second or less. The uncarried transactions are said to be *lost*. The loss rate is 1 – (carried load/offered load). The carried load is sometimes referred to as the *throughput* of the system. The rate at which successful completions occur is sometimes called the *goodput*, especially in systems in which retries occur such as packet networks on which TCP/IP is transported.

2.4.5 An Information Processing Example: Order Entry and Customer Relationship Management

A clothing or electronics vendor may sell goods to customers who visit a store in person, who place orders over the phone, or who order online. Order tracking and the handling of complaints entail

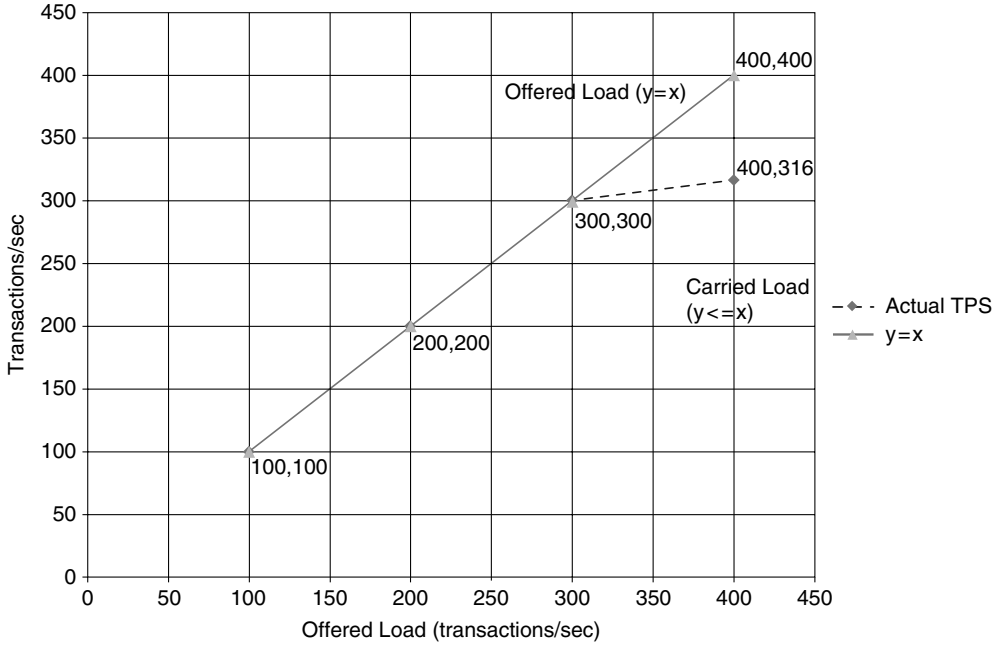


Figure 2.2 *Offered and carried loads (transactions per second, or TPS)*

the retrieval of customer sales records, whether the customer complains in person, on the phone, or online. During quiet periods, reports are generated to track the number of items of each type that were ordered, the number of shipments completed, the number of complaints associated with each catalog item, the number of orders entered, and the like.

For online transactions triggered by a customer or a company representative, metrics describing performance objectives might include

- The response times of online searches of each type (shop for an item, order tracking, lists of prior orders)
- The response times of order placement transactions
- The response times of transactions to enter a complaint
- The times to display dialog pages of any kind
- The number of agents and customers who could be simultaneously engaging in transactions of any kind
- The average number of times in the peak hour that each type of transaction succeeds or fails

Measures of demand would include

- The average number of times in the peak hour that each type of transaction occurs
- The average number of items ordered per customer session
- The duration of a customer session

For overnight, weekly, and monthly reports, a performance objective might be the completion of a report by the time the head office opens in the morning. To this end, performance measures might include

- The time to complete the generation of a report of each type

Measures of demand might include

- The number of entries per report
- The number of database records of each type to be traversed when generating the report

2.5 Examples of Explicit and Implicit Metrics

A metric may contain an explicit description of the load, or an implicit one. An explicit metric contains complete information about what it describes. The average number of transactions per second of a given transaction type is an explicit metric, as is the average response time of each transaction of each type. By contrast, the number of users logged in describes the offered load only implicitly. The number of users logged in is an implicit metric because the average number of transactions per second of each type generated by the average user is unspecified. In the absence of that specification, any requirement or other description involving the number of users logged in is ambiguous and incomplete, even if it is possible to count the number of users logged in at any instant. For example, a news article about the poor performance of the US government health insurance brokerage web site, healthcare.gov, under test did not say whether the users were trying to enroll in the system, check on enrollment status, or inquire about available coverage [Eilperin2013].

2.6 Time Scale Granularity of Metrics

To avoid ambiguity, the time scale within which an average metric is computed should be stated explicitly. An average taken over 24 hours

or over a year will be very different from one calculated from measurements taken during the peak hour. This is true whether one is measuring average response times or average utilizations.

It is much easier to meet a requirement for the average response time taken over a whole year than it is to meet the requirement during the busy hour, when it really matters. When one averages response times over a whole year, the short response times during quiet periods compensate for unacceptably high response times during the peak hour. Performance requirements should be specified for the peak hour, because it is that hour that matters the most in almost all applications. Moreover, if delay requirements can be met under heavy load, it is quite likely that they can also be met under light load unless there is a severe quirk in the system.

CPU utilizations tend to fluctuate over short periods of time depending on the amount of activity in the system. Even if the load is light, the CPU will appear to be 100% busy at times if observations are taken over intervals that are short enough. That means that the average utilization must be computed over time intervals that are long enough to be meaningful. As a rule of thumb, it is useful to collect measurements over 5- or 10-second intervals when testing performance in the lab, but over intervals of 1 or 5 minutes in production environments. Measurements should be taken over shorter intervals when it is necessary to relate spikes in resource usage to the inputs or outputs of the system under test or to the activities of particular processes or threads.

2.7 Performance Metrics for Systems with Transient, Bounded Loads

In some types of systems, the average value of a performance measure is less important than the amount of time required to complete a given amount of work. Let us consider a human analogy. During the intermission at a theater, attendees may wish to obtain refreshments at the bar. It is understood that attendees seated closest to the bar will get there first, and that very few attendees, if any, will leave their seats before the applause has ended. The average time to obtain a drink is of less importance than the ability to serve as many customers as possible early enough to allow them to consume what they have bought before the intermission ends, especially since refreshments may not be brought into the auditorium. It follows that the performance metrics of interest here are (1) the time from the beginning of the intermission until the time at which the last customer queueing for refreshments leaves the

bar after paying and (2) the number of customers actually served during the intermission. If service expansion is desired, the bar manager may also be interested in the number of customers who balked at queueing for a long time at the bar, perhaps because they thought that they would not have time to finish their drinks before the end of the intermission. If each customer queues only once, the number of customers requesting drinks is limited by the number of tickets that can be sold.

In some network management systems, large batches of status polling requests are issued at constant intervals, for example, every 5 minutes [Bondi1997b]. Status polls are used to determine whether a remote node is operational. They are sometimes implemented with a *ping* command. If the first polling attempt is not acknowledged, up to $N - 1$ more attempts are made for each node, with the timeout interval between unacknowledged attempts doubling from an initial value of 10 seconds. If the Nth message is unacknowledged, the target node is declared to be down, and an alarm is generated in the form of a red icon on the operator's screen, a console log message, and perhaps an audible tone. If the nth message is acknowledged by the target ($n \leq N$), the corresponding icon remains green. For $N = 4$ (as is standard for ICMP *ping*s), the time to determine that a node is down is at least 150 seconds.

In an early implementation of a network management system, no more than three outstanding status polling messages could be unacknowledged at any one time. This limitation is not severe if all polled nodes are responsive, but it causes the system to freeze for prolonged periods if there is a fault in the unique path to three or more nodes, such as a failed router in a tree-structured network. The rate at which status polling messages can be transmitted is the maximum number of outstanding polls divided by the average time to declare whether a node is either responsive or unresponsive. If many nodes are unresponsive, the maximum rate at which polling messages can be transmitted will be extremely limited. To ensure timely monitoring of node status, such freezes are to be avoided. A method patented by the author [Bondi1998] allows an arbitrary number of status polling messages to be unacknowledged at any instant. This allows the polling rate to be increased markedly, while avoiding polling freezes.

What metrics should be used to compare the performance of the original polling mechanism with that of the proposed one? For each polling request, the performance measure of interest is the time taken from the scheduling of the batch to the time at which the first attempt on this round is transmitted. If B nodes are being polled in a batch, one

performance measure of interest is the total time to transmit all B first attempts, as this is the time to transmit the last of the batch from the initiation of a polling interval. It is shown in [Bondi1997b] that the proposed method performs much better than the original method whether the probability of a node being unresponsive is high or low.

2.8 Summary

In this chapter we have seen that performance should be quantified unambiguously in terms of well-defined, obtainable metrics. We have also described useful properties that these metrics should have, such as repeatability and reliability. We have also briefly examined situations where average values of measures are useful, and those in which it is more meaningful to look at the total amount of time taken to complete a specified amount of work. We shall explore these points further in the chapters on performance requirements and performance testing.

2.9 Exercises

2.1. Why is the number of users logged into a system an unreliable indicator of demand? Explain.

2.2. Identify the performance metrics of a refreshment service that is open only during class breaks or breaks between sessions at a conference.

2.3. A large corporation has introduced a web-based time sheet entry system. The head of the accounts department has specified that the system should support 1 million entries per month. There are 50,000 potential users of the system. Company policy stipulates that time entries shall be made at the end of each working day. Is the number of entries per month a metric of interest to system developers? Why or why not? Explain.

This page intentionally left blank

Chapter 3

Basic Performance Analysis

This chapter contains an overview of basic performance laws and queueing models of system performance. Among the topics discussed are

- Basic laws of system performance, including the Utilization Law and Little's Law
- Open and closed queueing network representations of systems, and performance models to analyze them
- Product form queueing networks and Mean Value Analysis
- Loss systems
- The relationship between performance modeling principles and the formulation of performance requirements
- The use of performance models to inform the interpretation of performance measurements and performance test results

3.1 How Performance Models Inform Us about Systems

Performance models of computer systems are used to predict system capacity and delays. They can also be used to predict the performance impact of changes such as increasing or decreasing demand, increasing

the numbers of processors, moving data from one disk to another, or changing scheduling rules. Even if a performance model does not predict response times or system capacity accurately, it can be used to inform us about qualitative trends in performance measures as a function of the offered load and of the device characteristics. For example, basic laws tell us that device utilizations, including processor utilizations, are linear functions of the offered load. They also tell us that average performance measures should be constant as long as the average load is constant. The failure of a system to conform to these laws should be investigated, because it is usually a sign of a flaw in the implementation or a consequence of how the system is operated or used. Performance models can also tell us whether performance requirements are achievable while helping us to identify the maximum load level for which a performance test will be meaningful. Thus, we build performance models not only for predictive purposes, but also to help us understand whether and why a system might be deviating from expected behaviors.

In this chapter we shall describe how to quantify the performance characteristics of queues and show how to carry out basic performance modeling. After describing basic laws relating to the performance measures of queues, we shall examine a single queue in isolation. We then go on to describe the modeling of networks of queues. Our description of the behavior and modeling of queues will be qualitative and intuitive. The reader will be referred to the literature for detailed mathematical derivations of many of the models.

3.2 Queues in Computer Systems and in Daily Life

Many systems we encounter in daily life can be thought of as collections of queues. Whether lining up for lunch at a cafeteria, to enter a museum, to clear security and board a plane, to pay a road toll, or to use a cash dispenser, we all encounter queues. Even those who provide services may have to queue. For example, at some air terminals and railway stations, taxis have to queue for passengers, and waiters and waitresses fetching beer at Munich's Oktoberfest must queue at beer dispensing points to obtain drinks for their customers.

Computer systems usually contain abstract representations of queues for resources as well. Some of these may be for hardware objects such as processors, I/O devices, and network transmission devices.

Others may be for software objects such as locks, database records, thread pools, and memory pools. Schedulers may contain separate queues for those threads or processes wishing to read an object (the readers) and those wishing to modify it (the writers). Separate queues are required for readers and writers because readers may share the object, but each writer must have exclusive access to it, without sharing the object with readers or with other writers. Some abstract objects, including processors, may have schedulers that give some classes of processes or threads priority over others.

3.3 Causes of Queueing

Queueing for service occurs when a job or customer arriving at a server finds the server busy or otherwise unavailable, such as for maintenance or repair. If the server is available on arrival, service can begin immediately. If the calculated utilization of a server is less than one—that is, if the time between arrivals is greater than the service time, and both times are constant for all customers—queueing will not occur.

If both the service time and the time between arrivals (also called the *interarrival time*) are constant, queueing will not occur provided that the interarrival time is greater than the service time. To see this, notice that if this condition holds, service will have been completed before the next arrival occurs, because it begins immediately. This is illustrated in Figure 3.1.

If the interarrival times vary while the service time remains constant, queueing can occur because the next arrival could occur before the customer in service leaves. This is illustrated in Figure 3.2.

Similarly, if the interarrival times are constant while the service time varies, queueing can occur, because the customer currently in service could be there at the time the next arrival occurs. This is illustrated in Figure 3.3.

If both the interarrival time and the service time vary, even more queueing will occur. This is illustrated in Figure 3.4.

If at least one (and possibly both) of the mean service time or arrival rate (i.e., the reciprocal of the average interarrival time) increases to the point where the calculated server utilization exceeds 100%, queueing will occur whether or not the service time and interarrival time are constant, and an unceasing backlog will build up unless customers are dropped or go away because there is no place for them to wait.

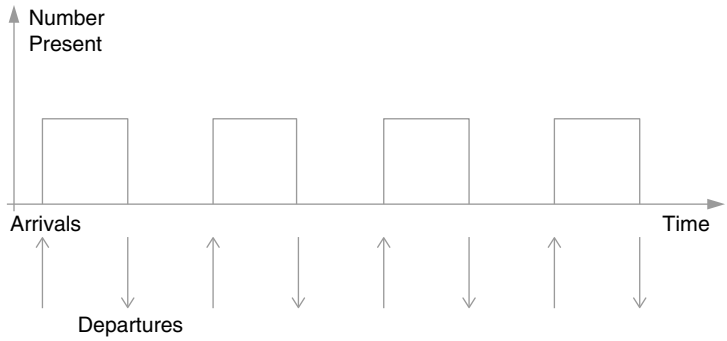


Figure 3.1 *Evolution of number present with constant service and interarrival time*

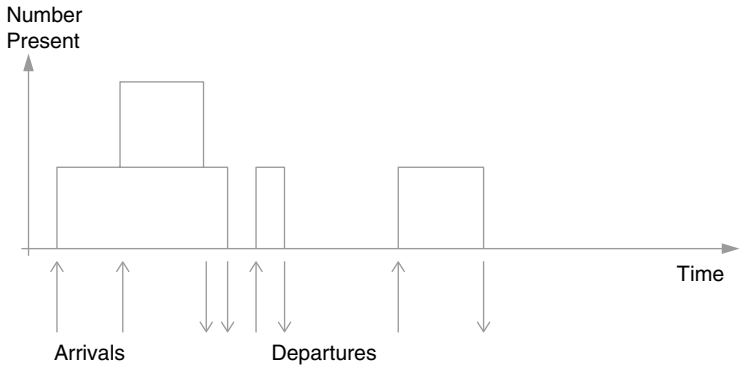


Figure 3.2 *Evolution of number present with variable interarrival and constant service times*

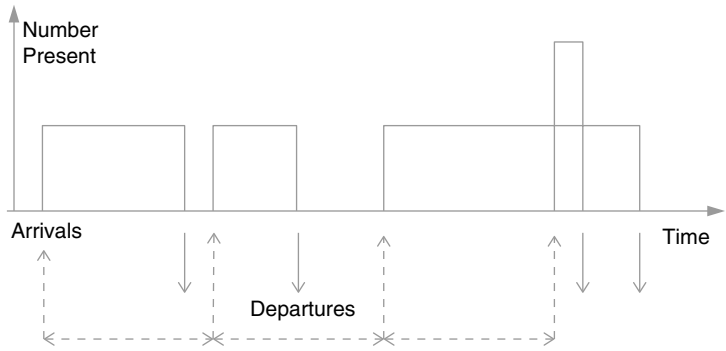


Figure 3.3 *Evolution of a queue's length with constant interarrival and variable service times*

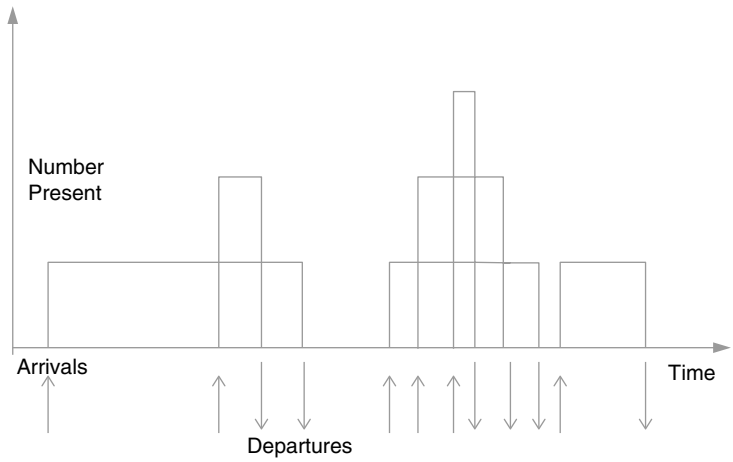


Figure 3.4 *Evolution of a queue's length with variable interarrival and service times*

The variability of the service and interarrival times may be characterized by the variances and higher-order moments of their respective distributions. For most performance engineering purposes, it is usually sufficient to characterize the distributions by their means (average values), variances, and their coefficients of variation. The coefficient of variation is the standard deviation or the square root of the variance divided by the mean.

Variability in either or both the service time and interarrival time is sufficient for queueing to occur. In principle, queueing can be reduced by reducing any one of these factors on its own, provided that the server is not overloaded because the arrival rate exceeds the service rate. Service time variability is usually inherent in the nature of the work to be done. Interarrival time variability has many possible causes, including but not limited to arrivals in bulk and interdeparture time variability from one or more servers upstream.

We close this section with some formal definitions:

- The *interarrival time* is the time between the successive arrivals of two customers or jobs at a queue. It may be constant, or it may vary randomly.
- The *arrival rate* is the reciprocal of the *average* interarrival time.
- The *service time* is the time a customer spends being served upon reaching the head of the queue.
- The *service rate* is the reciprocal of the *average* service time.

3.4 Characterizing the Performance of a Queue

A queue consists of a waiting line with a server at its head and a stream of customers or jobs arriving for service. In everyday life, the customers may be people wishing to complete a transaction. In a computer, the customers may be processes, threads, data packets, or other abstract objects awaiting the use of a resource. A single-server queue is shown in Figure 3.5.

The following are examples of the occurrence of single-server queues:

- Processes contending for a single CPU are placed in a run queue or ready list.
- Processes issuing read or write requests at a particular device, such as a disk, are placed in an I/O queue.
- Arriving packets are placed in an input buffer, while those awaiting transmission are placed in an output buffer.

Sometimes multiple parallel servers are present, as would be the case with parallel processors, a uniprocessor with multiple cores, or multiple windows in a bank or post office fed by a single queue. A single queue with multiple servers in parallel is illustrated in Figure 3.6.

Finally, one or more servers may be fed by multiple queues of customers. The queues may contain different classes of customers. The customers may have different classifications, such as priority or status. In many computer systems, jobs completing I/O are given priority for processing over other jobs so that the I/O buffer, a scarce resource, can be freed. Many airlines have separate boarding queues for their frequent fliers and for business-class customers, and the immigration halls at airports in the European Union have separate queues for EU passport holders and non–EU citizens.

A queue's performance may be characterized by the following variables:

- The arrival rate, sometimes denoted by A or λ.
- The throughput, often denoted by X. Since jobs may be lost, we have $X \leq \lambda$. Ideally, jobs are not lost, and the throughput equals the arrival rate.
- The mean service time, sometimes denoted by S or $1/\mu$, where μ is the mean service *rate*.
- The response time, which is the length of time from a job's arrival to its service completion. The response time is the

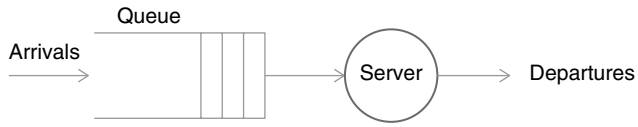


Figure 3.5 *A single-server queue*

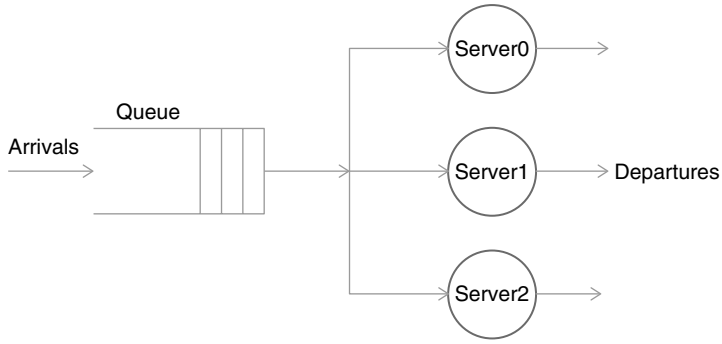


Figure 3.6 *A single queue with multiple servers at its head*

duration of the job's sojourn at the server. Hence, it is sometimes called the *sojourn time*.

- The waiting time, which is defined as the length of time between a job's arrival and the time at which it begins service.
- The queue length, that is, the number of jobs present in the system, including the number in service.
- The traffic intensity $\rho = \lambda / \mu = \lambda S$. The traffic intensity is a measure of the load offered to the system. For example, if a transaction has a mean service time of 3 seconds ($S = 3$), the traffic intensity is 0.9 when the arrival rate is 0.3 transactions per second ($\lambda = 0.3$), and 0.3 when the arrival rate is 0.1 transactions per second ($\lambda = 0.1$). Usually, higher traffic intensity means a longer delay for service, because the queue is longer. For a single-server queue, if the traffic intensity is greater than or equal to one ($\rho \geq 1$), the server cannot keep up with demand, and the average waiting time is undefined because each arriving job waits longer than the previous one.
- The utilization U. This is the proportion of time that the server is busy. In systems in which jobs are not lost, the utilization and traffic intensity are the same; thus $U = \rho$. If jobs can be lost, the utilization may be less than the traffic intensity, and $U \leq \rho$.

In practice, one usually refers to the averages of these values during a particular time interval. As we saw in Chapter 2, the average response time and average waiting time are sample statistics, while the average arrival rate, average queue length, and average utilization are all time-averaged statistics.

The load on a server is described by the job arrival rate, the mean service time per job, and the distributions of the service and interarrival times. Sometimes there is a single server at the head of the queue.

The server utilization ranges from 0% to 100%. If the server's predicted utilization is greater than or equal to 100%, the rate at which jobs arrive will exceed the maximum rate at which they can depart, and the server is said to be *saturated*. Then, the average queue length, average response time, and average waiting times are undefined, as the queue length and response time will steadily increase over time until the arrival process stops, until the memory allocated for the arriving jobs is exhausted, or until the system crashes for whatever reason, whichever comes first.

The order in which customers are served is called the *queueing discipline*. There are many queueing disciplines. Among the most common ones in computer systems are the following:

- First Come First Served (FCFS). In this case, jobs are served in the order in which they arrive, once the job in service has been completed. FCFS is sometimes referred to as First In First Out (FIFO).
- Last Come First Served (LCFS), in which the most recently arriving job is served next, once the job in service has been completed. LCFS is sometimes referred to as Last In First Out (LIFO).
- Last Come First Served Preemptive Resume (LCFSPR), in which the most recently arriving job preempts the job in service and is served to completion, at which point the interrupted job resumes being served.
- Time slicing or Round Robin. In many operating systems, jobs are given a maximum amount of processing time before a timer interrupt forces them to cede control of the processor to the next job in the CPU queue. The preempted job goes to the back of the queue and only executes once it reaches the front. The effect is to smooth the impact of highly variable processing times among jobs. This smoothing does not come without a cost, as the context of the interrupted job must be saved and that of the resuming or starting job loaded into registers with every interruption [Habermann1976].

- Processor Sharing (PS). In the limit as the duration of the time slice tends to zero, all n jobs present at the server receive service at $1/n$ times the service rate. Thus, if the service rate is μ, the service rate per job when n jobs are present is μ/n. Since n jobs are receiving service concurrently, the overall service rate (the reciprocal of the average service time) is $n\mu/n = \mu$.

Notice that the order in which jobs or transactions are processed does not affect the traffic intensity or server utilization. If the server can keep up with arriving transactions and transactions are not lost, work is conserved.

3.5 Basic Performance Laws: Utilization Law, Little's Law

There are basic laws that relate to the performance measures of a queue. Bearing these laws in mind can help tell us whether performance requirements are achievable and whether performance measurements are realistic. If the numerical parameters of the performance requirements do not conform to the laws, the requirements are unrealistic and unachievable. If the performance measurements do not conform to the basic performance laws, they should be regarded as suspect and the instrumentation investigated. For example, some versions of *perfmon*, the performance monitoring tool supplied with the Windows XP and NT operating systems, can produce measured disk utilizations that exceed 100%. This data should be treated with suspicion and an alternative measurement used where it is available [Willis2009].

3.5.1 Utilization Law

The first law we consider is the *Utilization Law*. It relates completion rate, average service time, and utilization. The Utilization Law states that the average utilization of a single server is its throughput multiplied by the average service time. Thus,

$$U = XS \tag{3.1}$$

Changing the queueing discipline does not change the average server utilization, because arriving jobs will always be served unless the server is saturated. This is similar to the principle of conservation of work in physics.

Example. We are told that three transactions per second arrive at a processor, and that the average service time is 100 msec per transaction. What is the utilization?

Solution. The average utilization of this server is $3 \times .01 = 0.3$.

Example. The measured utilization of a device with a transaction rate of 100 jobs per second is 90%. What is the average service time?

Solution. Applying the Utilization Law, we have

$$S = \frac{U}{X} = \frac{0.9}{100} = 9 \times 10^{-3} \text{ sec} = 9 \text{ msec per job}$$

Example. The average service time per transaction at a device is 10 msec. What is the maximum transaction rate that the device should be expected to handle?

Solution. We require that the utilization always be less than 100%. Thus, we require that

$$U = XS < 1 \tag{3.2}$$

Hence, we must have

$$X < 1/s = 1/(10 \times 10^{-3}) = 100 \text{ transactions per second}$$

We usually require that the average utilization of any device should not exceed 80% to avoid saturation. In that case, we require that

$$X < 0.8/S = 0.8/(10 \times 10^{-3}) = 80 \text{ transactions per second}$$

As we shall see in Section 3.6, the reason for this requirement is that the average response time is highly sensitive to changes in the offered load when the utilization exceeds this level. A system should not be engineered to operate at higher utilizations because users desire that response times be stable.

When interpreting the utilizations reported by measurement tools, it is important to consider the time scale over which their average values have been computed, and how the measurements were obtained. The utilizations of central processing units are often obtained by measuring the fraction of time that an idle loop is executing and subtracting that value from 100%. An idle loop is a process or thread of lowest priority that executes only when there is no other work for the processor to do. For sufficiently short measurement intervals, the resulting

CPU utilization could appear to be 100%. Taking averages over intervals that are too short will reveal higher variability in the value of the measurement, which may be regarded as a noisy signal. Taking an average over a longer interval will usually yield a lower measured average value, with a less noisy signal.

3.5.2 Little's Law

Little's Law relates the average response time, average throughput, and average queue length. Note that these are all time-averaged quantities.

Little's Law describes the average number of jobs or customers that are present and waiting at a server. It states that

$$\overline{n} = XR \tag{3.3}$$

where $\overline{n}$ is the average queue length, X is the average throughput, and R is the average response time [Little1961]. The variables on the right-hand side are jobs passing through in unit time and time. Thus, the dimension on the left-hand side is the number of jobs in the queue. Since jobs either arrive or depart one at a time, the number present increases or decreases one at a time. This is sometimes referred to as *one-step behavior*.

Figure 3.7 illustrates this evolution. Without loss of generality, suppose that the queueing discipline is FCFS, and that the queue is initially

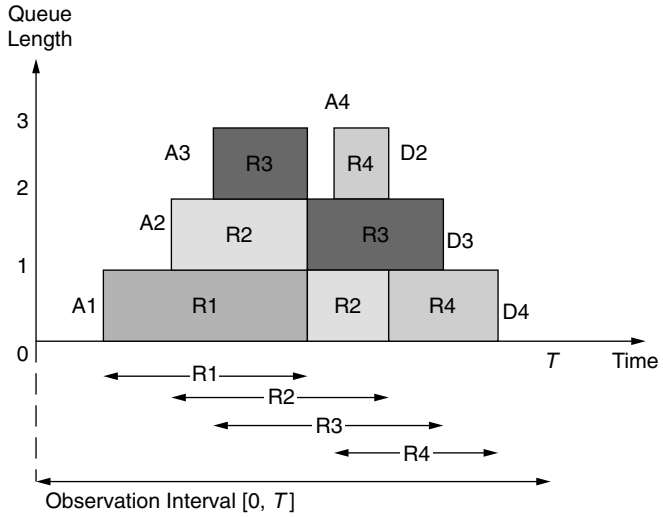


Figure 3.7 *Evolution of the queue length over time*

empty and the server initially idle. At arrival instant A1, the number of jobs present increases from 0 to 1. Since the first job finds the system empty and idle, it leaves after time R1. The second job to arrive arrives at A2 and has response time R2. Its arrival causes the number of jobs present to increase from 1 to 2, and job 3's arrival causes the queue length to increase from 2 to 3. Job 1's departure causes the queue length to drop by one. Now, the average queue length in the time interval [0,T] is equal to the area under the graph divided by T. A step up on the graph corresponds to the arrival of a job, while a step down corresponds to a departure. The time between a step up and a corresponding step down is the response time of a job. Hence, the area under the graph is the sum of the response times. The throughput is the number of jobs that both arrived and completed during the time interval [0,T], which we denote by C.

Now, the average response time is given by

$$\bar{R} = \frac{1}{C}\sum_{i=1}^{C} R_i \tag{3.4}$$

The average throughput in the observation period [0,T] is given by

$$X = \frac{C}{T} \tag{3.5}$$

The average is the area under the queue length graph divided by the length of the observation period. This is the sum of the response times divided by the length of the observation period T. Hence,

$$\bar{n} = = \frac{\sum_{i=1}^{C} R_i}{T} = \frac{C}{C}\frac{\sum_{i=1}^{C} R_i}{T} = \frac{C}{T}\frac{\sum_{i=1}^{C} R_i}{C} = XR \tag{3.6}$$

as we desired to show.

If the server is replaced by a faster one, we expect both the average response time and the mean queue length to decrease. The reverse is true if the server is replaced by a slower one. Intuitively, a faster server should have both shorter response times and average queue lengths given the same arrival rate. One can infer this linkage by an inspection of Figure 3.7. The server utilization is the fraction of time that the server was busy. In this example, the first arrival occurred at time A1, and the last departure occurred at time D4. The server was idle from time 0 to

time A1 and from time D4 to time T. Therefore, the utilization of the server in this example is

$$U = \frac{D4 - A1}{T} \tag{3.7}$$

A longer average service time corresponds to a higher utilization, and also to a longer time between the first arrival and the last departure, as illustrated in the first exercise at the end of the chapter.

3.6 A Single-Server Queue

Suppose that a single server is fed by an arrival stream. If the service times are fixed, and the times between arrivals are also fixed and far enough apart that the utilization of the server is less than 100%, the arrival stream and the server will be in lockstep, and queueing will not occur. This is shown in Figure 3.1 earlier in the chapter.

If service times are constant and jobs arrive randomly, it is possible that an arrival will occur while service is in progress and will have to queue for service, as will any job that subsequently arrives while service is in progress. This is depicted in Figure 3.2.

Similarly, if job interarrival times are constant and service times fluctuate according to some probability distribution or vice versa, one or more jobs may have to be queued while service is in progress. This is shown in Figure 3.3. Clearly, this also holds if both the interarrival times and service times are random, as shown in Figure 3.4. That is why buffers are needed to hold queues of packets, why run queues are needed for CPU scheduling, and why queues are also needed to schedule I/O at disks and other devices. It is variability that causes queueing [Whitt1984].

Suppose that jobs arrive at a server with FCFS queueing according to a Poisson process with rate λ and that the service time distribution at this server is exponential with rate μ. The mean service time at this system is $1/\mu$, and the server utilization is given by

$$\rho = \frac{\lambda}{\mu}$$

We also call ρ the *traffic intensity*. We require $\rho < 1$, since the server utilization cannot exceed 100%. If $\rho \geq 1$, the system is saturated, and a backlog of jobs will build up over time ad infinitum because the rate of customers entering the system exceeds the rate going out. We say that

the queue is *in equilibrium* if it is not saturated and if it displays long-term average behavior (see, e.g., [Cooper1981] or [Kleinrock1975] for details). This type of queue is known as an M/M/1 queue. Here, M stands for Markov, and the number 1 refers to the single server. It can be shown that the average number of jobs present, or mean queue length including the customer in service, is given by

$$\bar{n} = \frac{\rho}{1-\rho}, 0 \le \rho < 1 \tag{3.8}$$

This expression is undefined if $\rho = 1$ and does not make physical sense if $\rho > 1$, since a negative queue length cannot occur. The average response time may be obtained from this equation and Little's Law. We have

$$\begin{aligned} \bar{R} &= \frac{\bar{n}}{\lambda} \\ &= \frac{1}{\mu - \lambda} \end{aligned} \tag{3.9}$$

This quantity is defined only if $\lambda < \mu$, because it would be infinite or negative otherwise. Intuitively, λ is the rate at which jobs flow into the system, while μ is the rate at which jobs flow out. If $\lambda \ge \mu$, the jobs will flow in faster than they can flow out, and the backlog will grow over time without ever being cleared. In a computer system, it will grow until the memory allocated for the queue is fully occupied, at which point the system will either discard jobs or crash, depending on what it is programmed to do in this circumstance. Moreover, the *average* queue length will be undefined, because it is inherently meaningless when the system is not in equilibrium.

To see what these expressions tell us about the behavior of the queue at light, moderate, and heavy loads, let us examine the plot of the mean response time with $\lambda = 1$ as shown in Figure 3.8. With $\lambda = 1$, the mean queue length and response time are identical because of Little's Law. The mean queue length increases very slowly with respect to the arrival rate until it approaches 0.7. Beyond that point, the slope becomes very much steeper, with a very sharp increase as the load increases from 0.9 to 0.99.

Put another way, the system is much more sensitive to small increases in load when it is more heavily loaded. Now, running a system at very low levels of utilization might not be cost-effective, but running a system at a high level of utilization can result in very high

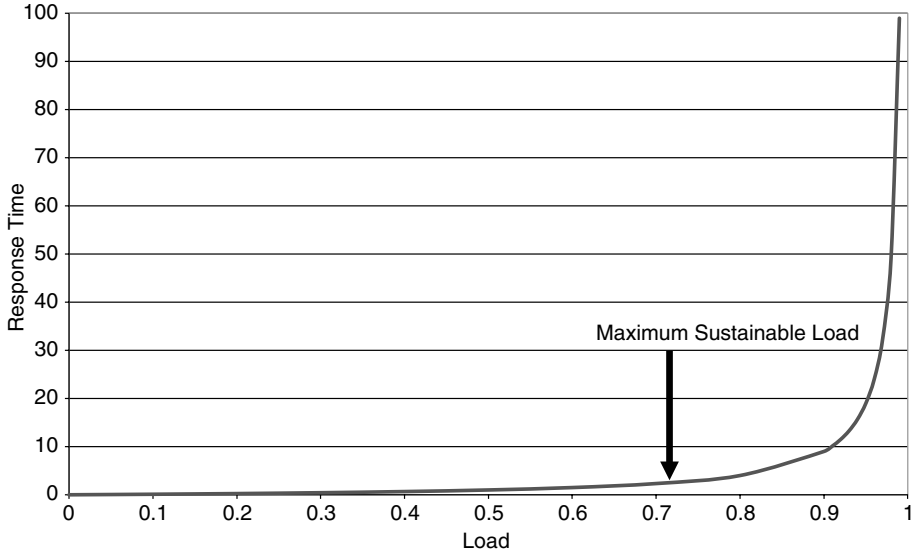


Figure 3.8 *Mean response time of an M/M/1 queue*

response times. Therefore, the system should be engineered so that the load on a server during peak periods is somewhere between 50% and 70%. Engineering the server to have an average utilization of at most 70% during the peak period keeps response times at acceptable levels, especially when arrival rates and/or service times are subject to major fluctuations. Greater loads will be seen by the user or system owner as unsustainable, because the response time is unacceptable.

The average queue length and average response time are affected not only by the average arrival rate and service time, but also by their respective distributions. We briefly discuss the impact of service time variability on a single FCFS queue in isolation here. For details, the reader is referred to [Kleinrock1975]. Briefly, the more variable the service time, the longer the mean queue length is expected to be. This is partly because an arriving customer sees not only those who arrived previously, but also the remaining service time of the customer being served at the instant the customer arrived. This is the residual service time, which is an increasing function of the variance of the service time distribution. For a single-server queue with Poisson arrivals, traffic intensity ρ, and service time coefficient of variation C^2, known as an M/G/1 queue, the average queue length is given by the Pollaczek-Khinchin formula,

$$\overline{n} = \rho + \frac{(1+C^2)\rho^2}{2(1-\rho)} \tag{3.10}$$

This equation shows that for a given traffic intensity ρ, the mean queue length is least when the service time is constant, that is, when $c^2 = 0$. When the service time is constant, we have

$$\bar{n} = \rho + \frac{\rho^2}{2(1-\rho)} \tag{3.11}$$

When the service time is exponentially distributed, we have $c^2 = 1$. Substituting, we have

$$\bar{n} = \rho + \frac{2\rho^2}{2(1-\rho)} = \frac{2\rho(1-\rho)+2\rho^2}{2(1-\rho)} = \frac{\rho}{1-\rho} \tag{3.12}$$

as expected. From equation (3.10), we see immediately that constant service time yields a lower mean queue length than exponentially distributed service time when the two servers have the same utilization or traffic intensity. Indeed, for suitably chosen parameters, a server with constant service can have a higher mean queue length than one with exponential service if the former's utilization is large enough. Thus, both utilization and service time variability are factors affecting mean queue length and response time. This comparison reinforces our intuition about the causes of queueing discussed earlier.

When two or more servers in parallel are fed by a single queue, as is the case in multiprocessor systems, including those on some inexpensive laptops today, the effects of service time variability are smoothed somewhat [BondiBuzen1984]. This is because not every customer who starts service ahead of a tagged customer will complete service before that tagged customer, since parallel service of multiple jobs allows the possibility of overtaking.

3.7 Networks of Queues: Introduction and Elementary Performance Properties

Many physical systems may be represented by networks of queues. Customers in a cafeteria move through waiting lines to obtain a tray and cutlery, drinks, different kinds of foods (soups, salads, main courses, desserts, etc.), and then queue to pay a cashier. In a computer, processes, threads, or tasks queue for processing at the CPU and for I/O when necessary. Each of these systems may be represented by a

network of queues. In this section we will see how a queueing network model can be used to represent a computer system, and then look at the way elementary properties of a network of queues can be used to derive rudimentary bounds on the average system response time and the maximum attainable system throughput. We will also explore the relationship between the utilizations of the devices in the queueing network and the system throughput.

3.7.1 System Features Described by Simple Queueing Networks

Figure 3.9 shows a queueing network representation of a computer system known as a *central server model* [Buzen1973]. In the central server model, jobs enter the system, queue for service at the CPU, and then move from the CPU to the I/O devices and other peripherals such as LAN cards and back again. They may also move from the CPU to itself if the CPU scheduler uses time slicing. The CPU queue is sometimes known as the run queue or the ready list. Each I/O device also has its own queue of requests. Jobs leave the system from the CPU upon completion. LAN cards may also have their own dispatch queues for outgoing packets, as well as receive queues for inbound packets. Concurrent execution of I/O and processing allows several jobs to be served simultaneously by the system as a whole.

Access to the processor and I/O devices is arranged by schedulers that determine the order in which queued processes or threads will be served. Data packets arrive at a host via receive buffers and are queued for transmission in send buffers. As they move across a network,

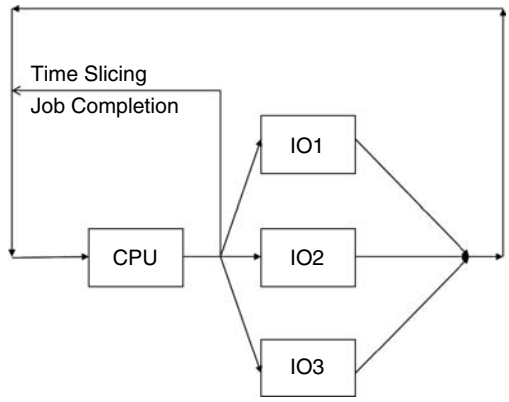


Figure 3.9 *A central server model with a single CPU and three I/O devices*

packets will go through a succession of receive and send buffers in switches and routers before arriving at their destinations.

The use of queues to mediate access to devices enables computer systems to perform I/O on one or more processes while processing is taking place. This allows multiple processes to receive service of various kinds concurrently. In early computers, this task was performed by human operators. Eventually, this function was taken over by a computer program, which, because it did the work of an operator, became known as the operating system [Habermann1976]. Concurrency enables computer systems to maintain the illusion that they are doing many things at once, even though a single processor can serve only one process at a time.

3.7.2 Quantifying Device Loadings and Flow through a Computer System

In this section we consider how to quantify the relative amounts of processing activity, input/output activity, and networking activity. For a particular action, there may be given numbers of reads or writes to one or more disks, flash memories, or network cards, with processing occurring in between. Each input/output action incurs some transfer time that is related to the amount of data transferred and the properties of the device involved. The processing time depends on the nature of the action, the algorithm used to implement it, as well as the properties of the processor and memory involved. The relative numbers of visits to the devices, including the processor, are called *visit ratios*, denoted by V_i for the ith device. The average time spent receiving service at each device on each visit is called the mean service time per visit to the device and is usually denoted by S_i. In the special case of a central server model like that shown in Figure 3.9, there is a visit to the CPU for every visit to an I/O device, as well as a visit when a job is started. Hence, for this central server model, we have

$$V_{CPU} = 1 + V_{IO1} + V_{IO2} + V_{IO3} \tag{3.13}$$

since

$$p_{IOj,CPU} = 1 \quad j = 1, 2, 3 \tag{3.14}$$

Also,

$$V_{IOj} = V_{CPU} p_{CPU,IOj} \tag{3.15}$$

It can be shown that equations (3.13) and (3.15) have a unique solution.

The throughputs of the individual devices and the system throughput are related to one another. Let X_0 denote the system throughput, and let X_i denote the throughput of the ith device. Since every visit to the system as a whole results in V_i visits to the ith device, we have

$$X_i = V_i X_0, \quad i = 1, 2, ..., K \tag{3.16}$$

Equation (3.16) is known as the *Forced Flow Law* [DenningBuzen1978]. The Forced Flow Law implies that the throughputs of the individual devices in the system increase together in proportion to the global system throughput X_0. Multiplying equation (3.16) on both sides by the mean service time S_i, we obtain

$$X_i S_i = X_0 V_i S_i, \quad i = 1, 2, ..., K \tag{3.17}$$

The left-hand side of this equation is the utilization of the ith device. We define the demand on server i as

$$D_i = V_i S_i \tag{3.18}$$

Then, we easily obtain

$$U_i = X_0 D_i, \quad i = 1, 2, ..., K \tag{3.19}$$

This equation shows that the utilizations of all the devices in the system rise in proportion to the global system throughput. If one plots the utilizations of the servers as functions of the global system throughput, they will appear as straight lines with slopes proportional to the corresponding demands D_i. If one plots the utilizations as functions of the global system throughput on a logarithmic scale, the resulting curves will be constant distances apart. To see this, observe that

$$\log U_i = \log D_i + \log X_0 \quad i = 1, 2, ..., K \tag{3.20}$$

and that for nonzero utilizations,

$$\log(U_i / U_k) = \log(U_i) - \log(U_k) = \log(D_i / D_k) \tag{3.21}$$

independent of X_0. An example of a set of utilization curves is shown in Figure 3.10 for demands $(D_1, D_2, D_3) = (0.09, 0.06, 0.03)$ in an open

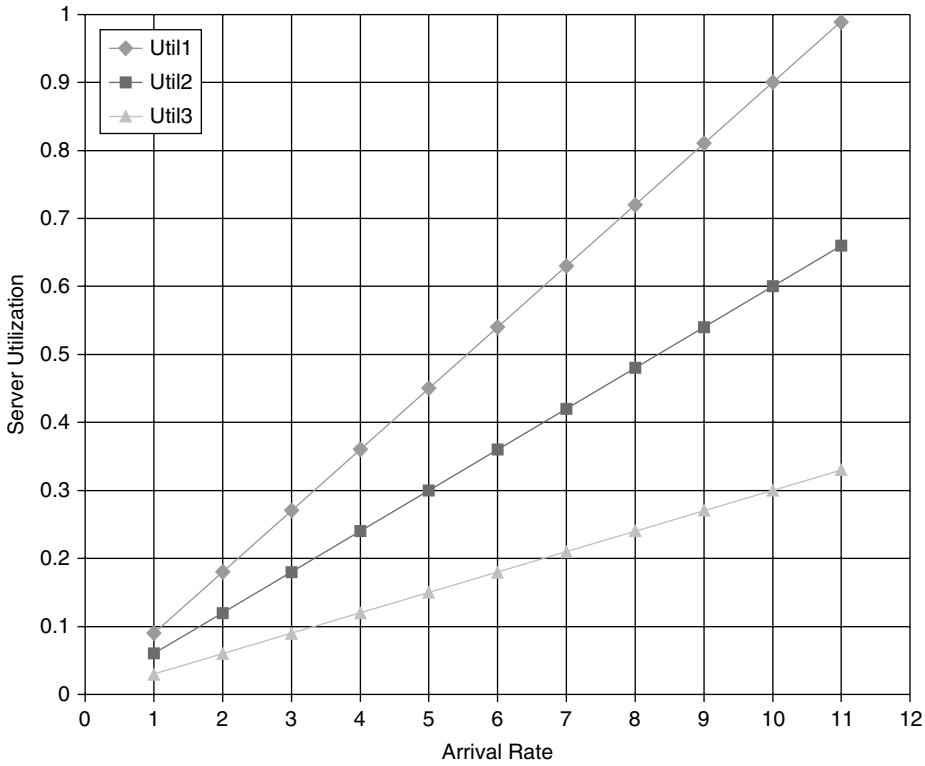


Figure 3.10 *Example utilizations versus arrival rate, vertical scale linear*

queueing system with external arrival rates ranging from 1 to 10. An example of a set of utilization curves on a logarithmic scale is shown in Figure 3.11. The curve for the utilization of device 2 is a constant linear vertical distance from its neighbors immediately above and below.

3.7.3 Upper Bounds on System Throughput

Equation (3.19) implies that the global system throughput is limited by the maximum throughput of the bottleneck device, that is, the one with the highest demand D_i. To see this, recall that the utilization of any device must be less than one if saturation is to be avoided. Hence, from equation (3.2), we have

$$D_i X_0 < 1, \quad i = 1, 2, ..., K \tag{3.22}$$

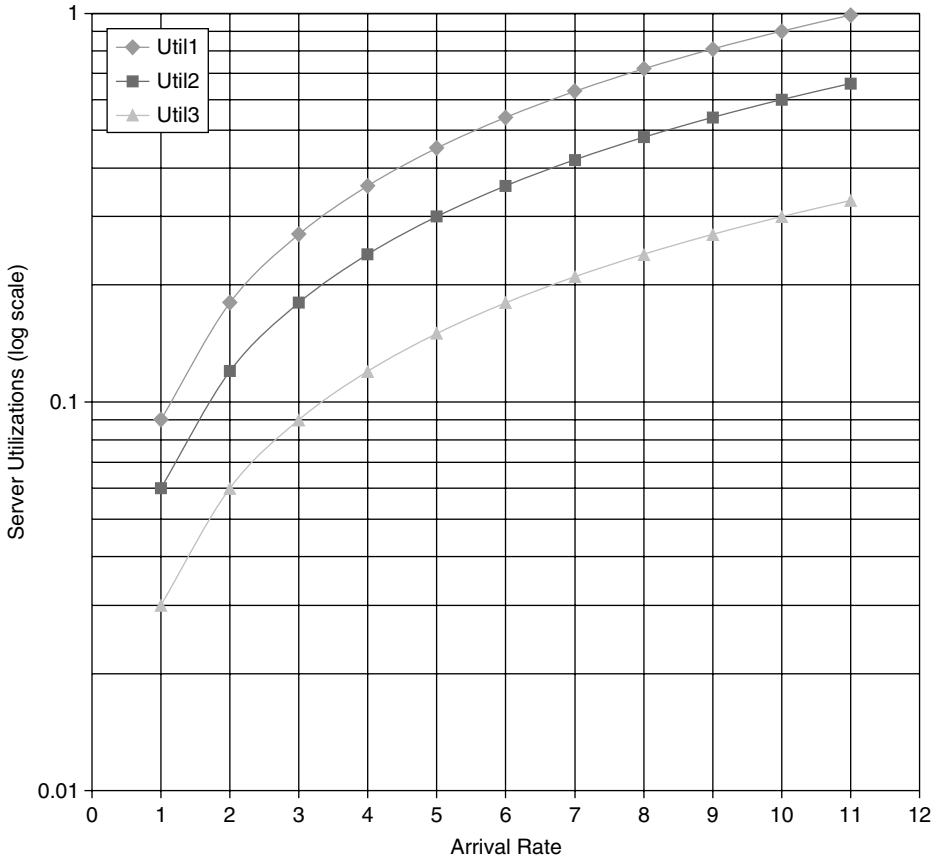


Figure 3.11 *Example utilizations versus arrival rate, vertical scale logarithmic*

Let b denote the index of the largest of the D_is. Since equation (3.22) holds for all demands D_i, it must hold for the largest of them. Therefore,

$$X_0 < 1/D_b \tag{3.23}$$

This shows that the maximum throughput of the system as a whole is bounded above by that of the most heavily loaded device, as one might expect. Moreover, by analogy with our analysis of the single-server queue in Section 3.6, we must limit the global system throughput so that the system response time is insensitive to small changes in the offered load. There are also ramifications for performance requirements, because equation (3.23) implies that a throughput requirement

is not achievable unless the desired system throughput is less than the reciprocal of the service demand made by a given task at the bottleneck device.

3.7.4 Lower Bounds on System Response Times

Suppose jobs are served at only one device at a time, that is, that all activity is synchronous. Just as throughputs are bounded above by the capacity of the system bottleneck, response times are bounded below by the total processing demands at all the devices visited while a job is being executed. If the response time at the ith device is R_i, and that device is visited V_i times by each job, the total execution time is the sum over all devices of the number of visits to each device multiplied by the response time at that device. Thus, the overall system response time is given by

$$R_0 = \sum_{i=1}^{K} V_i R_i \tag{3.24}$$

If the job has the system to itself, there is no queueing delay for service, and the time spent at the device on each visit is simply the service time. This means that a job will take at least as long as the sum of the total amounts of time being served at all the devices. Therefore, in general, provided no job is receiving service from more than one server at a time,

$$R_0 \geq \sum_{i=1}^{K} V_i S_i = \sum_{i=1}^{K} D_i \tag{3.25}$$

This inequality implies that a performance requirement that the overall average response time be less than the total demands at each of the devices is neither realistic nor achievable if the devices are visited one at a time.

3.8 Open and Closed Queueing Network Models

Queueing systems occur in a variety of forms. In the ones we experience in our daily lives, customers arrive, queue, receive service, perhaps progress to another service point in the system, and eventually leave. In others, the number of jobs or customers in the system is fixed. The jobs move from server to server, but the sum of the numbers of jobs at

individual servers is fixed. The first type of system is called an *open* queueing system, because customers enter and leave. The second type of system is called a *closed* system, because the jobs, or more precisely, entities representing the jobs, never leave the system but always circulate within it. One of the earliest closed queueing network models describes the circulation of carts from one service point to another in a coal mine [Koenigsberg1958]. Buzen was among the earliest to use a closed queueing network to represent a batch computer system with a constant number of concurrently executing jobs and an apparently infinite backlog, so called because a departing job is immediately replaced by a queued one [Buzen1973]. In the 1960s, 1970s, and even the early 1980s, jobs were more commonly initiated by loading a deck of punched cards into a reader than by pressing the Enter or Return key on an interactive terminal. Students of computer science during those years may well have experienced the backlog of batch jobs as being infinite.

We shall now examine simple performance models of open and closed queuing networks, and then discuss the qualitative differences between open and closed network representations of queueing systems.

3.8.1 Simple Single-Class Open Queueing Network Models

A queueing system is said to be open if jobs enter and leave it. If the system is functioning properly, jobs will enter and leave the system at the same rate. This is a necessary condition for the system to be in equilibrium. If a system is not in equilibrium, its long-term average performance measures are meaningless. To see this, consider that if at least one server in the system is saturated, the departure rate will be less than the arrival rate, because jobs will be able to leave only as fast as the saturated server allows. At least one queue will build up inside the system until queueing space, such as memory, is exhausted. Long-term average performance measures also have no meaning if a system has periodic behavior. Hence, a periodic system cannot be in equilibrium, even if the bounds of the performance measures are finite and stable.

In the special case in which the service time distribution at all FCFS servers in the network is exponential and the outside arrival process is Poisson, and routing from one server to another is probabilistic, the queue length distribution at all servers will be geometric, and the joint queue length distribution will be the product of the individual distributions. The joint queue length distribution is said to have a *product form*. For single-class open queueing networks, this result is known as *Jackson's Theorem* [Jackson1963].

Suppose that the network consists of K service centers, each with a single FCFS server. Let the arrival rate at the system be λ, and suppose that the visit ratio and mean service time of server k are V_k and S_k respectively. The utilization of the kth device is given by

$$U_k = \lambda V_k S_k, \quad k = 1, 2, ..., K \tag{3.26}$$

When $U_k < 1 \ \forall k$, none of the servers is saturated, and the joint queue length distribution is given by

$$\Pr(n_1, n_2, ..., n_K) = \prod_{k=1}^{K} (1 - U_k) U_k^{n_k} \tag{3.27}$$

The mean queue length at each server is given by

$$\bar{n}_k = \frac{U_k}{1 - U_k}, \quad k = 1, 2, ..., K \tag{3.28}$$

provided that $U_k < 1$ for $k = 1, 2, ..., K$, that is, that none of the servers is saturated. Notice that this expression has the same form as in equation (3.12), as if each server were in isolation. The joint probability distribution of the queue lengths factorizes into the probability distributions of the lengths of the individual queues. This shows that the probability distribution at one queue is independent of those of all the others in the network. Intuitively, this may be linked to the exponential distribution of the service times, the Poisson nature of the arrival process from outside the network, and state-independent probabilistic routing of jobs from one service center to the next.

3.8.2 Simple Single-Class Closed Queueing Network Model

In batch processing systems, the number of jobs circulating through the system may be regarded as fixed, because a newly completed job is immediately replaced by another, similar one upon completion. Examples of applications that are processed in batch include the large-scale processing of payrolls, monthly billing, monthly bank statements, and the preparation of data-intensive reports.

In interactive transaction systems with a fixed number of terminals logged in, a user launches a task by hitting Return or clicking with a mouse. The task is processed in the computer system and returns to the user, where it is delayed while the user thinks about what to do next.

The task returns to the computer system once the user has finished thinking and clicks once again. The average time the user spends thinking before relaunching the task is called the *think time*, usually denoted by Z, while the time spent circulating among the devices in the computer system is the *global response time* R_0. This is illustrated in Figure 3.12. In a pure batch processing system, there is no think time, and $Z = 0$.

The computer system is modeled as a network of queues. The set of processors and other devices that constitute the computer system is often called the *central subsystem*. The think time is modeled as a queue with infinite service (IS). Infinite service is so called because a server is available for every job that returns to the terminal. It is sometimes called a *pure delay server* because the instant availability of a server means there is no queueing for service.

The think time, average response time, and system throughput are related by the *Response Time Law*, which is a direct consequence of Little's Law. The Response Time Law is easily derived by observing that the average total time between task launches is the sum of the average response time and the think time. The number of circulating tasks is equal to the number of terminals logged in, denoted by M. The system throughput, X_0, and the average response time depend on the number of terminals logged in, since having a larger number logged in

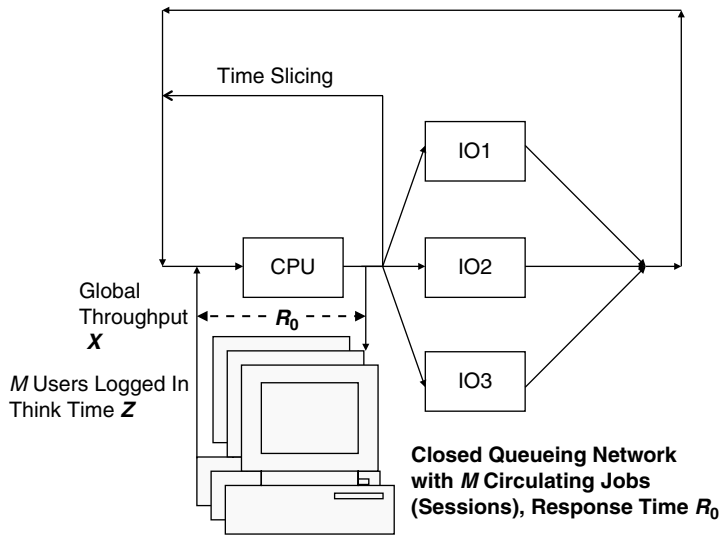


Figure 3.12 *Computer system with terminals, central processor, and three I/O devices*

tends to increase contention in the central subsystem. From Little's Law, we obtain

$$[R_0(M)+Z]X_0(M)=M \tag{3.29}$$

From this, we obtain the Response Time Law,

$$R(M)=\frac{M}{X_0(M)}-Z \tag{3.30}$$

Rearranging equation (3.29), we can see that a larger think time reduces the system throughput, since

$$X_0(M)=\frac{M}{[R_0(M)+Z]} \tag{3.31}$$

If users spend more time thinking, they will not be sending transactions to the central subsystem as frequently. Similarly, a higher throughput is associated with a lower response time, while a lower throughput is associated with a higher response time. As the think time is reduced to zero, the response time of the central subsystem will increase, and vice versa.

3.8.3 Performance Measures and Queueing Network Representation: A Qualitative View

A system in which a departing job is always instantaneously replaced may be represented as a closed network of queues, because the number of circulating jobs within it is constant for all intents and purposes. Such a system is said to have infinite backlog. Because the number of waiting jobs at a server in a closed system is bounded above by the total number of jobs in the system, the average queue length is bounded above by the number of (circulating) jobs, and the average response time at any server is bounded above by the number of jobs in the system multiplied by the average service time, even if the server is 100% busy. If the system were open, the number of jobs queued would be unconstrained, and the average response time could be larger than in a closed system with servers having the same utilizations and throughputs. It can be shown that this is indeed the case [Zahorjan1983]. Using simulations and experimental data, Schroeder et al. [SWH2006] show that the impacts of scheduling rules or service time variability on

response time behavior can differ markedly between open networks and closed ones. The effects of scheduling rules in open networks are always more marked, among other reasons because the number of queued jobs can grow without bound. In closed networks, while increased service time variability can degrade throughput, the effect need not be large [BondiWhitt1986].

3.9 Bottleneck Analysis for Single-Class Closed Queueing Networks

When designing a computer system, it is often necessary to determine the maximum throughput and the minimum response time that can be achieved. For systems that can be represented by a closed queueing network, these quantities depend not only on the characteristics of the devices and the workloads, but also on the number of concurrently circulating jobs or processes. In this section we will derive formulas for asymptotic bounds on throughput and response time in single-class networks and briefly examine the performance consequences of asynchronous activities, such as asynchronous I/O, which can be used to reduce response times. Performance bounds for multiple-class queueing networks are described in [Kerola1986].

3.9.1 Asymptotic Bounds on Throughput and Response Time

As in an open queueing network, the maximum possible throughput of a closed queueing network is constrained by the demand made on the bottleneck device. From equation (3.23), we know that the system throughput is bounded above by the reciprocal of the demand on the bottleneck device. Let us consider how the throughput changes as the number N of circulating jobs increases from one. At this stage of our analysis, we shall set the think time Z to zero, since we are considering only the throughput and response time of the central subsystem.

When only one job is circulating in the central subsystem, it does not have to queue for any server. Thus, with $N = 1$, the response time of the central subsystem is simply the sum of the demands made of all the servers. Hence, from equation (3.25), we have

$$R_o(1) = \sum_{i=1}^{K} V_i S_i \tag{3.32}$$

provided that a job can receive service at only one server at a time, that is, synchronously. Because this is the response time with only one job present and no contention, the response time of the central subsystem cannot be any better than this. When we study performance requirements in Chapters 5 through 7, we shall visit this notion once again.

Using Little's Law or the Response Time Law with $M = 1$ and $Z = 0$, the system throughput with the sole job having the central subsystem to itself is given by

$$X_0(1) = 1 / \sum_{i=1}^{K} V_i S_i \tag{3.33}$$

As the number of jobs in the central subsystem increases, the throughput attainable by the central subsystem will always be less than or equal to $NX_0(1) = N / R_0(1)$. But the throughput will also be less than or equal to the maximum achievable throughput at the bottleneck device. Hence, it must be less than or equal to the smaller of them, giving us

$$X_0(N) \le \min(N / R_0(1),\ \ 1 / D_b) \tag{3.34}$$

This shows that one must improve the bottleneck device if one wishes to increase system capacity at heavy loads. For systems with terminals and nonzero think times, inequality (3.34) becomes

$$X_0(M) \le \min(M / [R_0(1) + Z],\ \ 1 / D_b) \tag{3.35}$$

The throughput bounds for the central subsystem (i.e., not considering think time) resulting from these inequalities are shown in Figure 3.13.

Queueing at the bottleneck device ensues to the extent that it degrades throughput when the number of circulating jobs in the central subsystem is sufficiently large. It occurs when the two throughput bounds cross. Thus, queueing ensues for the smallest population such that

$$\frac{N}{R_0(1)} = \frac{1}{V_b S_b} \tag{3.36}$$

Thus, queueing ensues for some N^* such that

$$N^* \ge \frac{R_0(1)}{V_b S_b} = \frac{\sum_{i=1}^{K} V_b S_b}{V_b S_b} \tag{3.37}$$

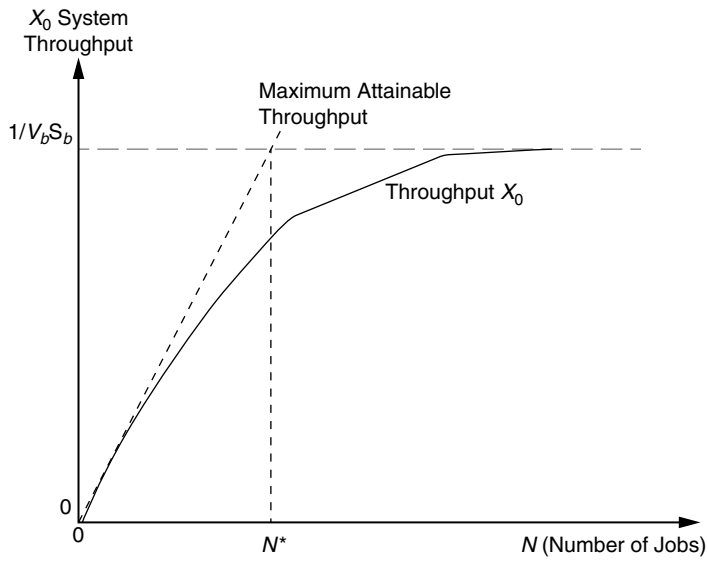


Figure 3.13 *Bounds on the throughput of the central subsystem*

N^* is known as the *saturation point*. It is the minimum network population for which queueing will occur. For systems with thinking terminals, queueing ensues when the number of terminals logged in exceeds

$$M^* \geq \frac{R_0(1)+Z}{V_bS_b} = \frac{\sum_{i=1}^{K} V_i S_i + Z}{V_bS_b} \tag{3.38}$$

Inequality (3.35) shows that thinking terminals and other sources of pure delay inhibit system throughput. Put another way, the more time users spend thinking before launching each new activity into the central subsystem, the less work they can throw at the central subsystem. Inequality (3.38) shows that a longer think time allows more terminals to be logged in before queueing ensues in the central subsystem. We shall take note of this when we consider how to configure load testing clients and how many to use.

Combining inequality (3.35) with the Response Time Law gives us a way to place lower bounds on the system response time. Since $X_0 \leq 1/D_b$ we must have

$$R_0(M) \geq MV_bS_b - Z \tag{3.39}$$

when the think time is nonzero. But the response time also has to be at least as large as the time to circulate through it unimpeded. Therefore, we have

$$R_0(M) \geq \max[MV_bS_b - Z, R_o(1)] \tag{3.40}$$

Response time bounds are illustrated in Figure 3.14.

3.9.2 The Impact of Asynchronous Activity on Performance Bounds

Asynchronous I/O is sometimes used to reduce application response times. Its purpose is to allow I/O to proceed without requiring the CPU to await its completion. The response time is reduced because I/O activity and processing activity can be overlapped. While the response

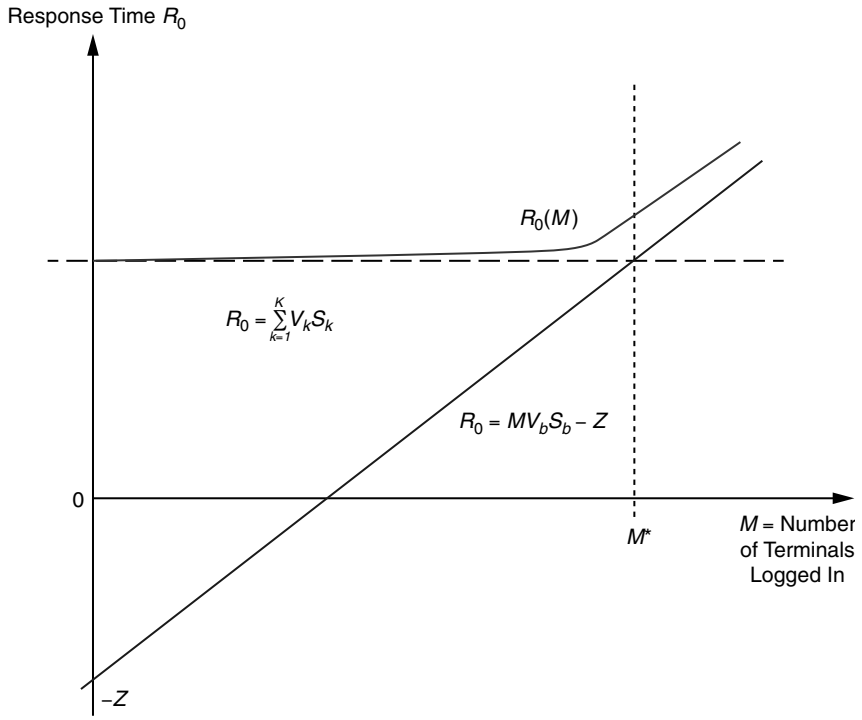


Figure 3.14 *Response time bounds with logged-in thinking terminals*

time is reduced, it cannot be less than the time spent at the bottleneck device, even if no other jobs are present. If only one job is present in a closed system, we must have

$$D_b \le R_0(1) \le \sum_{i=1}^{K} D_i \tag{3.41}$$

depending on how much overlap there is between activities at the various devices. Notice that this violates the principle that a job cannot receive service at more than one device at a time. That was one of the conditions for equations (3.24) and (3.32) and the equations that are derived from them.

For larger numbers of circulating jobs, suppose that the amount of overlap with N circulating jobs is $\alpha(N)$. In that case, the average response time is given by

$$R_0(N) = \sum_{i=1}^{i=K} V_i R_i(N) - \alpha(N) \tag{3.42}$$

and the system throughput with think time Z and M terminals logged in is given by

$$X_0(M) = \frac{M}{R_0(M) + Z - \alpha(M)} \tag{3.43}$$

This shows that the system throughput of a closed queueing network can be increased by using asynchronous I/O. The potential increase is limited, though, because the bottleneck device has the same amount of work to do as before, and therefore the same maximum throughput.

Measurements indicate that asynchronicity is present when the response time is less than the sum of the demands, or when it is less than the sum of the total times spent at the individual devices, that is,

$$R_0 \le \sum_{i=1}^{K} V_i R_i \tag{3.44}$$

This is also true for parallel executions of tasks within a job [Gunther1998].

3.10 Regularity Conditions for Computationally Tractable Queueing Network Models

For many queueing network models to have computationally tractable exact solutions, they must conform to a set of conditions, some of which are statistical and some of which relate to the operating environment. By computationally tractable, we mean that it is possible to find a reasonably fast algorithm to obtain the performance measures predicted by the model. Experience shows that these models are quite accurate even when some of these conditions are not satisfied.

Many queueing networks have joint queue length distributions that have a product form. We have already seen that this is the case for open queueing networks with probabilistic routing, Poisson arrivals, and exponential service. A single-class closed queueing network with FCFS servers has a joint queue length distribution of the form

$$P(n_1, n_2, ..., n_K) = \frac{1}{G(N)} \prod_{i=1}^{K} D_i^{n_i}, \quad \sum_{i=1}^{K} n_i = N \tag{3.45}$$

where $G(N)$ is a normalizing constant chosen so that the probabilities sum to one, that is,

$$G(N) = \sum_{n_1+n_2+...+n_K=N} \prod_{i=1}^{K} D_i^{n_i} \tag{3.46}$$

Notice that unlike the case with open queueing networks, the joint queue length distribution does not factorize into the marginal distributions at the individual nodes. This is because the presence of n jobs at one node implies that $N-n$ nodes are present at all the other nodes combined.

According to the *BCMP Theorem* [BCMP1975], a system with open, closed, or mixed networks and Poisson arrivals for the open networks has product form if each server in it satisfies the following conditions:

1. The service time distribution has a rational Laplace transform.
2. The service time discipline is First Come First Served (FCFS), Last Come First Served Preemptive Resume (LCFSPR), Processor Sharing (PS), or Infinite Service (IS).
3. If the server is FCFS, the service time has an exponential distribution with the same mean for all classes of jobs.
4. Routing is probabilistic and independent of the state of the network.

A queueing network satisfies the conditions for the BCMP Theorem if it satisfies the conditions for Jackson's Theorem. The queueing disciplines mentioned in condition 2 or approximations of them often occur in computer systems:

- First Come First Served queueing usually occurs unless some other queueing discipline has been implemented. For instance, packets arriving in a buffer are usually queued and forwarded in FCFS order.
- In a server with Processor Sharing (PS), each of n jobs present at the server receives service at a rate that is $1/n$ times the average service rate. Thus, if the service rate is μ, each job is served at rate μ/n. This is very similar to time slicing, in which each job receives a fixed slice of processing time τ before being sent to the end of the CPU queue. Processor sharing may be thought of as the limiting discipline as the length τ of each slice tends to zero.
- In some operating systems, the completion of a READ or WRITE at a disk triggers an interrupt that causes the job executing at the CPU to be preempted, and the job whose I/O has just been completed to take control of the CPU to finish the associated processing. This corresponds to Last Come First Served Preemptive Resume.
- Think time is usually modeled as an Infinite Server (IS) because the progress of the job through the network is only delayed by the think time, and the job does not queue. If a queueing network model has product form, it can be solved using a fast computational algorithm such as the convolution algorithm [Buzen1973] or Mean Value Analysis (MVA) [ReisLav1980]. Computational algorithms for closed, open, and mixed queueing networks are described in [LZGS1984] and [BruellBalbo1980]. In the next section we introduce Mean Value Analysis because its formulation can be easily related to Little's Law, and because one can easily implement a small-scale version of it in a spreadsheet.

3.11 Mean Value Analysis of Single-Class Closed Queueing Network Models

Suppose a closed queueing network has a single class of jobs circulating within it, and that it satisfies the conditions of the BCMP Theorem. Suppose that it has K FCFS servers and a single infinite server with

think time Z. Suppose further that the mean service time at the ith server is S_i and that the visit ratio is V_i. Mean Value Analysis uses a relationship between a server's response time with N jobs present in the system and the mean queue length at the server with one less circulating job in the system. The mean queue length observed by an arriving customer is the mean queue length at the server with the arriving customer removed. This is known as the *Arrival Theorem* or the *Sevcik-Mitrani Theorem* [SevMit1981]. Using the Arrival Theorem, we can write

$$R_i(N) = S_i[1+\overline{n}_i(N-1)] \tag{3.47}$$

Intuitively, the response time of an arriving customer is equal to the service time of that customer, together with the service times associated with those customers who are already waiting. The average number who are already waiting is the mean queue length in a closed network with the arriving customer removed, by the Sevcik-Mitrani Theorem. If the queueing discipline is PS or LCFSPR, the arriving customer will not see any remaining service time for the customer in service at the arrival instant. If the service discipline is FCFS, the expected remaining service time (also known as the residual service time) is equal to the mean service time only if the service time distribution is exponential. If the server is IS, the arriving customer begins service immediately. This is the case with a job returning to a terminal. At a terminal, the job delay is equal to the think time Z.

To build a recurrence relation on N, we need a starting condition and a means of computing $\overline{n}_i(N)$ given $\overline{n}_i(N-1)$.

- When there is only one job circulating in the system, it will never arrive at a server to find another one queueing or being served. Therefore, we have $N=1$ and $\overline{n}_i(0)=0$.
- To compute the queue lengths at servers when N jobs are circulating, we first need to compute the system throughput. This can be done using Little's Law.

Recall that the time to cycle through the system is the think time, together with the response time of the central subsystem. The system throughput is then obtained using Little's Law. Hence, we have

$$X_0(N) = \frac{N}{Z+\sum_{i=1}^{K} V_i R_i(N)} \tag{3.48}$$

```
for (i=1;i<=K;i++) {n̄_i(0) = 0 }
for (n=1;n<=N;n++) {
    R_i(n) = 0.0;
    for (i=1;i<=K;i++) {
        R_i(n) = S_i(1 + n̄_i(n − 1));
        R_0(n)+ = V_i R_i(n)
        }
    X_0(n) = n / (R_0(n) + Z);
    for (i=1;i<=K;i++) {
        n̄_i(n) = V_i X_0(n) R_i(n)
        U_i(n) = V_i X_0(n) S_i
        }
    }
```

Figure 3.15 *Algorithm for single-class Mean Value Analysis*

We can then obtain the mean queue length with N jobs present at each of the K servers using Little's Law once the throughput at each server is known. The latter is easily obtained from the Forced Flow Law, since

$$X_i(N) = V_i X_0(N) \tag{3.49}$$

Applying Little's Law at the ith server, we obtain

$$\overline{n}_i(N) = X_i(N) R_i(N), \quad i = 1, 2, ..., K \tag{3.50}$$

This gives us the complete set of equations we need to solve the model for the desired performance metrics using Mean Value Analysis. The algorithm is depicted in Figure 3.15.

The analysis inputs and outputs would include the following:

- *Input:* visit ratios, think times, mean service times, number of terminals logged in or multiprogramming level
- *Output:* utilizations, response times, throughputs, mean queue lengths

3.12 Multiple-Class Queueing Networks

Some types of systems may have different types of workloads, characterized by the use of different disk drives, different arrival frequencies, different processing requirements, different amounts of bandwidth,

and so on. The different job types are represented in queueing network models by different job classes. Here are some examples:

- A web-based online banking system has variable numbers of users logging in concurrently, generating different types of transactions, such as checking balances and initiating one-time and repeating payments. These activities take place interactively and may occur at the same time as background activities such as cleaning up databases or backing up databases and transaction histories. The online transactions enter the system and then leave once completed. They arrive randomly. Therefore, they may be modeled as a set of jobs traversing an open queueing network. The background activities are executed by a fixed set of processes that neither enter nor leave the system. Therefore, they may be modeled as a set of jobs that circulate in a closed queueing network.
- Consider a network management system that monitors the status of a collection of managed nodes consisting of routers, hubs, switches, file servers, gateway servers, and all manner of hosts performing various functions. The network management system (NMS) is deployed on a single workstation. The NMS is scheduled to send a status poll to each of the managed nodes according to a defined timetable. Many of the managed nodes are equipped with a program known as an *agent*. The agent can respond to the poll and also supply information about the node's status. The agent can also generate messages of its own accord if an alarm condition or other designated event occurs. Such messages are known as *traps*. From the standpoint of the NMS, the traps are random events, because they are not scheduled. In the NMS, the polling activity may be regarded as a set of jobs that circulate within a closed queueing network. The polling jobs alternate between sleeping for set periods and awakening to initiate polls. They sleep until the polls are acknowledged, or until a timeout occurs because the acknowledgment did not arrive. The trap stream is modeled as an open class of tasks, because the system performs a particular set of actions (such as interrogating a database and raising an alarm) for each trap that arrives and does not execute those actions until the next trap arrives. Polling activity may be modeled as a closed system, because the polling tasks never enter or leave the system; they simply alternate between sleeping and executing.

From these examples, we see that it is possible for a computer system to host workloads with fixed numbers of jobs and workloads consisting of jobs that enter the system, execute, and then leave. Regardless of what the jobs in the closed workloads are doing, the jobs in the open classes must be able to leave the system at the same rate at which they enter it, or they will be backed up inside. This is called *flow balance*. Flow balance causes open classes of jobs to effectively take capacity away from closed classes of jobs, even if they do not have priority over them. Jobs in both kinds of classes will delay each other as well as jobs in their own classes by queueing together, assuming no priority scheduling rule is in place. The presence of open job classes delays the progress of jobs in closed classes, thus reducing the throughput of the closed workloads.

To explain this further, we first need to enhance our notation by adding a subscript c to each model parameter and performance measure to denote the job class. We let C denote the set of closed job classes and W denote the set of open job classes.

- λ_c denotes the arrival rate of open job class c, that is, for $c \in W$.
- V_{ic} denotes the visit ratio of class c jobs to the ith device.
- S_{ic} denotes the mean service time of class c jobs at the ith device.
- N_c denotes the number of circulating jobs in closed class c. Without loss of generality, the vector $\mathbf{N} = (N_1, ..., N_B)$ denotes the set of B closed job classes. Hence, $B = \| C \|$, that is, B is the number of closed classes, the cardinality of the set C.

We have the following relationships:

- By the BCMP Theorem, if the ith device has FCFS queueing, we must have $S_{ic} = S_i$ for all classes c.
- By the Forced Flow Law, the arrival rate of class c jobs at the ith device is $\lambda_{ic} = \lambda_{ic} V_{ic}$ if c belongs to the set of open job classes W.

It follows immediately from the Forced Flow Law that the utilizations are given by

$$U_{ic} = \begin{cases} \lambda_{ic} V_{ic} S_{ic} & \text{if } c \in W \\ X_{ic} V_{ic} S_{ic} & \text{if } c \in C \end{cases} \tag{3.51}$$

for open and closed job classes respectively.

We now return to the impact of open job classes on the progress through the network of jobs in closed classes.

For servers with all queueing disciplines *except* IS, define the following scaling for the closed classes of jobs:

$$S_{ic}^{*} = \frac{S_{ic}}{1 - \sum_{r \in W} U_{ir}}, \quad c \in C \tag{3.52}$$

Notice that when there are no open job classes, $W = \varnothing$ and the sum in the denominator of equation (3.52) is empty. In that case, we have

$$S_{ic}^{*} = S_{ic}, \quad W = \varnothing \tag{3.53}$$

as expected. S_{ic}^{*} is used to compute the mean response times of the closed job classes at the individual servers. By analogy with equation (3.47), the device response time of a closed class is given by

$$R_{ic}(\mathbf{N}) = S_{ic}^{*}[1 + \overline{n}_{ic}(\mathbf{N} - \mathbf{1}_{c})] \quad c \in C \tag{3.54}$$

where $\mathbf{1}_{c}$ denotes a vector with a 1 in position c and zeros everywhere else.

For open job classes, we have

$$R_{ic} = \frac{S_{ic}[1 + \overline{n}_{i}(\mathbf{N})]}{1 - \sum_{c \in W} U_{ic}} \tag{3.55}$$

Thus, the delay of an arriving open job is caused by the delay due to the average number of queued closed jobs plus the service of the arriving open job, all inflated by the complement of the resource usage attributable to the open jobs. The form of the response time of the open jobs is similar to that for closed jobs, with the exception that an arriving closed job sees the average queue length with itself removed, while the arriving open job sees the average queue length with no closed jobs removed. The closed formula is a consequence of Jackson's Theorem and the Arrival Theorem, while the open formula is a consequence of the Arrival Theorem and the theorem that Poisson arrivals see time averages (PASTA) [Wolff1982].

3.13 Finite Pool Sizes, Lost Calls, and Other Lost Work

In many types of systems, an arriving customer, task, or job will be discarded if a server is not available or if there is no place for the customer to wait. For example:

- In circuit-switched telephone systems, calls may be dropped or rerouted if all of the circuits in a direct trunk group are busy. Calls are not queued until a trunk becomes available. Calls that cannot be routed along a particular trunk group are declared to be lost. Teletraffic engineers often wish to size trunk groups so as to keep the probability of a lost call below a small threshold. Sizing depends on the anticipated call volume and call duration during the busiest hour of the day.
- A barbershop has a fixed number of barber's chairs and a fixed number of chairs in the waiting area. If all the barbers are busy and all the chairs in the waiting area are occupied, an arriving customer will balk and go elsewhere. That customer is lost. A sufficiently high customer loss rate may justify the addition of more chairs in the waiting area and/or the addition of one or more barbers. The decision to do either must be made carefully: adding barbers may increase costs more than revenue, while adding chairs in the waiting area may not increase costs but could increase waiting times to the point that waiting customers will leave before being served.
- A multitiered computer system may consist of one or more web servers in parallel, one or more application servers in parallel, and a back-end database server. Communications between an application server and the database server are mediated via a pool of connections sometimes known as Java Database Connections (JDBCs). This pool is known as the JDBC pool. It has a configurable maximum size, M, say. If all of the M connectors in a JDBC pool are occupied when a thread on the application server needs to communicate with the database, the thread will be queued until a JDBC becomes free. If the queueing buffer itself overflows, the user's transaction may be lost. Since both delay and loss are undesirable, both the JDBC pool and the memory allocated to waiting threads should be sized to keep the probability of each occurrence below small thresholds.

Simple queueing models exist to aid the performance engineer in sizing the number of servers or abstract objects and the waiting area (if any) to prevent transaction loss and reduce the risk of delay. Here, we present the Erlang formula for the lost calls. The reader is referred to [Kleinrock1975] or [Cooper1981] for other cases.

Suppose that a telephone trunk group consists of s trunks, and that the average duration or holding time of a call is H seconds. Suppose further that calls arrive at the rate of λ per second. The traffic intensity is the product of the trunk holding time and the call arrival rate. It is sometimes called the *offered load*. In effect, it is the average number of trunks that calls would occupy if they were available and if calls were not lost. It is given by the following formula:

$$\rho = \lambda H \tag{3.56}$$

This is the expected number of occupied trunks. If the traffic intensity exceeds the number of trunks, that is, if $\rho > s$, a very large fraction of calls will be lost, but the trunk group will continue to function.

The probability that a call will be lost is given by

$$B(s,\rho) = \frac{\rho^s / s!}{\sum_{k=0}^{s} \rho^k / k!} \tag{3.57}$$

$B(s,\rho)$ is known as the *Erlang loss formula*. For a given offered load ρ, we can engineer the number of trunks or servers s to ensure that the probability of a lost call is below a certain level. The purpose of this is to make the carried load—that is, the expected number of completed calls—as large as possible. The choices of trunk group size and loss probability are often determined by engineering and cost considerations. The carried load is given by the offered load multiplied by 1 minus the probability of each call being lost. Thus, the carried load ρ' is given by

$$\rho' = \rho[1 - B(s,\rho)] \tag{3.58}$$

A teletraffic engineer desiring to engineer a trunk group size s for a particular loss objective, for example, one lost call in a million, would choose the smallest trunk group size to make $B(s,\rho)$ less than 10^{-6}. In the absence of cheap computing power, this used to be done with the

aid of Erlang loss curves. Examples of these curves can be found in [Cooper1981] and elsewhere.

A recurrence relation on the number of trunks can be used to compute the Erlang loss formula efficiently. When there are no trunks, all calls are lost, so

$$B(0,\rho) = 1 \tag{3.59}$$

For $s \geq 1$, it can be shown that

$$B(s,\rho) = \frac{\rho B(s-1,\rho)}{s + \rho B(s-1,\rho)}, \quad s = 1, 2, \ldots \tag{3.60}$$

The smallest value of the trunk group size s satisfying the loss requirement is found by iterating through equation (3.60) on s until $B(s,\rho)$ is less than the desired value. Applying this recurrence relation is computationally cheaper than adding up partial series as in equation (3.57) and may be numerically more stable as well, because the quantities in the numerator and denominator of equation (3.60) are of the same order of magnitude.

3.14 Using Models for Performance Prediction

Computer performance engineers, industrial engineers, and operations research practitioners have been successfully using models like the ones described in this chapter to make system performance predictions for computer systems, telecommunications systems, manufacturing systems, and other industrial systems for a very long time. Erlang's work on loss probabilities for telecommunications was published in 1917 [Erlang1917].

The steps of performance prediction include

1. The identification of the system structure, including paths followed by jobs through the system from server to server
2. The formulation of a queueing network model that represents the service centers and their queueing disciplines
3. The estimation of model parameters, whether by measurement or the application of experience (sometimes called *expert intent*)

4. Validation of the model, by comparing its predictions with measurements of the predicted values in the actual system under the same assumptions
5. Running the model with new parameters corresponding to system changes to predict the performance impact of those changes

Step 5 is sometimes called *what-if analysis*, because it is used to answer questions of the form "What if this change is made?" In operations research, running the model with different sets of parameters is sometimes called *sensitivity analysis*.

3.15 Limitations and Applicability of Simple Queueing Network Models

Queueing network models are very useful for predicting the direction of performance changes when various parameters are altered. The predictions cannot be better than the accuracy of the modeling assumptions, including assumptions about the values of the parameters. They also may not be able to take the effects of scheduling rules into account. In particular, the queueing models described in this chapter do not address priority queueing of any kind. They also do not address the modeling of specialized features such as graphics processor units, channel controllers, queueing for memory, and contention for shared threads, memory partitions, locks, and other discrete objects. However, the simple models can be combined to build very good approximate models of complex system characteristics. Both the simple models and the more complex models have been used for performance engineering with considerable success. Examples of models of specialized system features are given in [LZGS1984], [Gunther1998], and various conference papers and articles. It should be noted that even though [LZGS1984] predates the creation of the World Wide Web, it contains many examples that go beyond the simple queues presented here. Many types of problems occur in multiple technological guises. For example, similar queueing models can be used to predict the performance of queues for memory slots, discrete sets of objects in pools (such as threads or JDBCs), or the movement of packets subject to sliding window flow control.

3.16 Linkage between Performance Models, Performance Requirements, and Performance Test Results

Comparing the predictions of performance models with the results of performance tests can offer us insights into the functioning of the system under study. Conformance to performance predictions can be an indicator that the system is operating smoothly, while deviations from the predictions can be an indicator that something is amiss and afford us insights into possible improvements.

When choosing load generators for performance testing, care should be taken when deciding whether the workload to be tested is open or closed. The results in [SWH2006] show that open workloads tend to have larger response times than closed ones for the same throughput and utilizations, and that the effect of scheduling rules and service time variability is much more marked with open loads than with closed ones. This is predicted by performance models. The trends indicated by the performance models should be borne in mind when analyzing test results and when making design choices about scheduling rules.

The average values of performance measures such as utilizations, response times, and queue lengths depend on the loads also running close to their average values. If the average values of the factors driving the performance measures, the arrival rates and the service times, vary over time, the values of the utilizations, queue lengths, and response times will also vary over time.

Measured values of performance must conform to the laws described previously and to physical realities. As we shall see in the chapter on measurement, if the observed average utilization of a device with a single server, such as a disk, exceeds 100%, the validity of the measurement should be called into question. Similarly, if measured average response times, queue lengths, and arrival rates fail to satisfy Little's Law, the cause should be investigated. The cause could be clock drift (a failure of the system clock to maintain the correct time), difficulty keeping the clocks of different computers accurately synchronized, corruption of the arrival time stamp, or the collection of the response time in the wrong place in the job's execution path.

Performance requirements must be consistent with the Utilization Law, Little's Law, and the Response Time Law if they are to be achieved.

A system with a throughput requirement of 110 jobs per second at a server with mean service time of 1 second will have a predicted server utilization of 110%, which is clearly not achievable. Either the throughput requirement must be reduced, or the server must be sped up to the point where the predicted utilization is much less than 100% in order to have a stable average response time. We shall learn more about performance requirements in Chapter 5.

3.17 Applications of Basic Performance Laws to Capacity Planning and Performance Testing

Monitoring resource utilizations is usually easier than monitoring system response times, because counters for tracking the utilizations are usually provided by commonly used operating systems, while response times must often be measured using purpose-built load generation scripts. Throughputs can sometimes be obtained from application logs or platforms such as those used to implement business logic and databases. Tracking the resource utilizations and plotting them against transaction rates gives us a glimpse into the amount of spare capacity of the system and might provide the data needed to warn that a system might soon be saturated. As we shall see in Chapter 9, the Utilization Law can be used to plan the loads that could be realistically applied to performance tests, and the Response Time Law can be used to determine the number of load drivers that might be required to do so.

3.18 Summary

In this chapter we have presented the basic rules relating the performance measures of queues occurring in computer systems to one another, and then presented basic models of performance and their properties. We have explored the differences between open and closed representations of workloads and given an overview of differences between the values of performance measures they predict. These differences are inherent in the system structure: a closed system has a fixed number of jobs in it, while the number of jobs in an open system is unconstrained and potentially unbounded. Therefore, an open

system is much more sensitive to changes in service time distributions and scheduling disciplines. Performance measurements of systems must be consistent with the basic performance laws or regarded as suspect, and the values of performance measures stated in performance requirements must be consistent with the laws for the performance requirements to be achievable.

3.19 Exercises

3.1. We are given the following observations for a single-server queue. During the time interval [0, 8], 4 jobs were started and completed. The observed service times were 2.56, 0.4, 1.5, and 1.5. The observed response times were 4, 2, 4, and 4. The first arrival occurred at time 1, the last departure at time 7. There were no periods of idleness between the departures of customers. Compute the following for this observation period:

- **(a)** The average completion rate
- **(b)** The average service time
- **(c)** The average response time
- **(d)** The average throughput
- **(e)** The average utilization in two ways (*Hint:* No idle period between service times *in this example*.)
- **(f)** The mean queue length

3.2. An airline security checkpoint may be modeled as a system of two queueing networks. The passengers arrive at a rack of trays to hold items to be X-rayed, load the trays, and then queue to walk through a metal detector while the trays go through an X-ray machine. The network seen by the customers consists of one or more guard stations at which identities and boarding passes are checked, followed by a queue for trays, and another queue to go through the metal detector. The trays are stacked (queued) in racks, awaiting use by passengers and filled one at a time. The trays are then queued up to go through the X-ray machines. Once emptied by the passengers who loaded them, the trays are stacked on rolling pallets. The pallets are rolled to their positions at the benches before the X-ray machines to be reused.

(a) Identify the type of queueing network traversed by the trays and the nodes through which they pass.

(b) Explain the effect of delaying rolling the tray pallets back to their positions before the X-ray machines. Discuss the possible impact of having large stacks of trays on a small number of pallets versus small stacks of trays on a larger number of pallets. What happens if there are not enough pallets to station them both before and after the X-ray machines? How many pallets should there be to ensure smooth operation? (*Hint:* Consider the delay in moving a full pallet from the checkpoint exit to positions before the X-ray machines.)

(c) Explain the effect of having too few trays. How would you size the pool of trays? (*Hint:* Think about how long each one is in use, and about policies to return them to their original positions after use.) What basic performance laws can you use to obtain a rough estimate for the number of trays needed per X-ray machine?

(d) Identify the type of queueing network traversed by the passengers. Explain the effect on passenger delays of having
- **(i)** Multiple X-ray machines and tray racks
- **(ii)** Too few trays
- **(iii)** A single agent for checking boarding passes and identity documents

(e) Propose configurations of X-ray machines, tray pallets, ID inspection lines, and the like when the airport has a policy of giving priority to frequent fliers at the entrance to the security area. Explain what happens if the proportion of frequent fliers at a given hour is high or low.

3.3. Using a spreadsheet or otherwise, use equations (3.59) and (3.60) to generate a plot of the loss probability for trunk group sizes s varying one by one from 1 to 20 with an offered load of $\rho = 15$. Explain what happens to the loss probability as the number of trunks increases from 14 to 15 and then from 15 to 20. If there is a fixed amount of revenue r for every call, when does the expected increase in revenue fall below $r/10$ as the trunk group size increases?

3.4. Using a spreadsheet or otherwise, implement single-class Mean Value Analysis for a closed network with up to five service centers and thinking terminals.

(a) Your output should show the global system throughput X_0, the global response time R_0, and the throughputs, utilizations, mean queue lengths, and mean response times of the individual servers. If you are using a spreadsheet tool with plotting capabilities, plot the global system throughput, utilizations of the service centers, and average response times on separate sets of axes.

(b) Consider a closed queueing network with the parameters depicted in the following table. Identify the bottleneck device. Plot bounds on the system throughput when the think time is 0 and when the average think time is 2 seconds and 4 seconds. Plot bounds on the response time of the central subsystem.

Device Name	Visit Ratio	Service Time (sec)	Service Discipline
CPU	6.0	0.0090	PS
Disk 0	1.0	0.0400	FCFS
Disk 1	4.0	0.0250	FCFS
Thinking terminals	1.0	4.0	IS

(c) Use your MVA tool to predict the performance of the queueing network model with the same parameters with 1, 2, 3, . . ., 10 terminals logged in for think times of 0 and 4. Plot the predicted throughputs and response times on the same axes on which you plotted the performance bounds.

(d) Using a spreadsheet or otherwise, build a tool to predict the performance of an open network consisting of the CPU and two or three disks only, without thinking terminals, based on Jackson's Theorem.

- **(i)** Using the global system throughputs predicted by the closed queuing network model as inputs to the open model, predict the utilizations, response times, and mean queue lengths of the individual servers and the response time of the system as a whole. Also, compute the sum of the mean queue lengths of the individual servers.
- **(ii)** Compare your results with those predicted by the closed queueing network model when the think time is zero. Are the predicted utilizations the same

or different in the open and closed models? Are the predicted mean queue lengths and response times the same or different in the open and closed models? What is the sum of the mean queue lengths? Is it less than or equal to the number of logged-in terminals with zero think time?

(iii) Explain the differences between the predictions of the open and closed models.

(iv) Someone states that the CPU is the bottleneck in this system and should be replaced with a faster one. Is this view supported by the data in the preceding table or by your model outputs? Explain.

(v) Which disk should be improved? Using your models, predict the effect of spreading the load on Disk 1 evenly across two disks.

(vi) Based on your answers to (iv) and (v) above, what recommendation would you make to the system administrator for modifying the hardware configuration to improve performance with the current workloads?

3.5. Use Little's Law and Jackson's Theorem to derive an expression for the overall average response time of an open queueing network. (*Hint:* Start with equation (3.28).)

Chapter 4

Workload Identification and Characterization

We describe the need to specify the functionalities of a system and to identify the nature of the performance characteristics each functionality must have to be effective. The notion of a reference workload will be introduced as a vehicle for specifying a straw workload for the purposes of when several workloads are possible. We shall discuss the impact of time-varying behavior on system performance—for example, whether the load offered to it is rhythmic and regular, whether it varies seasonally or by time of day, whether it is growing over time, and whether it is inherently subject to potentially disruptive bursts of activity. These workload characteristics must be understood for performance requirements to be properly formulated and for the system to be architected in a cost-effective manner to meet performance and functional needs. Numerical examples of workloads from different application domains will be given.

4.1 Workload Identification

In Chapter 1 we saw that speed, capacity, and scalability are essential characteristics of a computer system, and that the nature of the system's functions and competitive and regulatory considerations

determine the desirable range of performance the system should have. Identifying the functions of the system and the recurring or continuous background activities needed to support those functions is a precondition for correctly formulating performance requirements and devising a cost-effective architecture for the system. It is also a precondition for developing a performance model of the system and/or of its components. Our focus is on linking quantitative characterizations of the workload to defined sets of functionalities rather than on statistical characterization. One of the reasons for this is that many types of computer-controlled systems have two or more types of workloads. Often, one of these involves ongoing background activity occurring at regular intervals, while another might involve handling a burst of message traffic within a very short amount of time so as to trigger certain types of actions, such as sounding alarms. With this kind of bursty traffic, analysis of average performance measures is meaningless because averages relate to steady-state behavior, while the application requires that the first transaction in a burst be completed within a short amount of time, and that a burst of transactions be processed within a somewhat longer period of time.

When qualitatively determining the performance needs of a system, one may ask the following questions:

- What is the main purpose of the system?
- What are the functions of the system?
- Where does the system traffic come from?
    - Is it externally driven?
    - Does it come from a limited set of sources?
    - Are there any monitoring and/or cleanup processes that make system demands at fixed intervals, at irregular intervals, or that run continuously in the background? What resources do they use and how fast must they be completed? Will the background activities be competing for the same resources as those activities with requirements for short response times?
- Is the system mission critical? Do life, limb, and/or safety of property depend upon it? Does national security depend upon it? Are real-time reactions to stimuli required?
- What are the risks and impacts if the system does not meet its performance requirements?

- Is there a mix of activities in the system, for example, record retrieval and image processing? Should some of the activities be offloaded to another system? Is this the intent?
- Are short response times needed for some kinds of activities? Will longer response times suffice for other kinds? Is there a mix of response time requirements among the applications?
- Are there any constraints on the response times?
- Is the system receiving data that must not be lost or allowed to accumulate for long periods of time without being processed?
  - How much data loss can be tolerated?
  - What are the consequences of data loss?
- What constituencies are served by each human interface?
- What constituencies, business needs, and engineering needs must be served by each data interface?

The answers to the questions about background activity influence performance requirements and operational practice. For instance, a background process that purges deleted database records may compete with a user query application for processing time, disk, memory, and table access. If ongoing cleanup is not critical to query performance but could adversely affect it, it may be wise to establish an operational procedure in which cleanup takes place only during periods of relative quiet, such as the middle of the night.

4.2 Reference Workloads for a System in Different Environments

Systems and suites of products are often targeted at multiple market segments. This can make it difficult to determine whether a system will meet the needs of a prospective customer. This is true of a wide variety of systems. Product managers and sales engineers are faced with the problem of convincing a prospective customer that a system can be configured in a way that meets their anticipated needs for functionality and capacity while allowing room for growth or even a decline in business volume. The problem is frequently compounded by the customer's inability to quantify what the demand on the system will be, whether because of uncertainty about it or because of an aversion to quantitative reasoning.

One way to address this difficulty is to identify a set of clearly described straw workloads that may be similar to those that will occur in the customer's environment. The customer and supplier can refer to these when making decisions about how to size a system. We call these workloads *reference workloads*. Reference workloads may be devised for a variety of situations and applications:

- Small banking institutions such as regional savings banks and credit unions need to provide the same services and comply with the same government reporting and security regulations as large banks, but they may lack the resources to develop their own software to do so. They will obtain a platform from an independent supplier instead. The supplier must be able to size the system for small and large banks. To illustrate the viability of its wares, the supplier should identify "typical" workloads with set transaction volumes for banks with perhaps tens of thousands of accounts and banks with hundreds of thousands or even millions of accounts. These workloads will include numbers of transactions per customer per month, the number of online transactions of various types occurring in the busiest hour of the month, the number of logged-in customers during the busiest hour of the month, as well as workloads for backing up the systems and generating seasonal reports.
- A supplier of a road traffic control system needs to be able to demonstrate that the system will function in jurisdictions with populations varying from tens of thousands to millions, covering both small and large areas. The number of lights the system will control and the number of road sensors depends on the number of intersections, the number of kilometers of road, the widths of the various roads, and other factors such as the density of road traffic. In this case, reference workloads would be devised for areas with populations of 10,000, 100,000, and 1 million, with road densities comparable to those of a sprawling city in a midwestern state or an intense amount of congestion comparable to that in a very densely populated city.
- A supplier of fire alarm and building surveillance systems can identify functionalities (such as detecting smoke and sounding alarms) that are common to small and large buildings. Reference workloads and their intended configurations may be identified for a small school, a large university campus, a small-scale chemical factory, or even a skyscraper.

4.3 Time-Varying Behavior

Many systems are subject to time-varying demands. The variation may be seasonal, or it may be sudden because of external events. For example:

1. Securities trading systems often cause order execution only when markets are open. If an account holder places a trade order when the markets are closed, it must be queued until the markets reopen. The trading systems may be subject to bursts of activity because of external events, such as the announcement that a drug has been approved by a regulatory body, the outbreak of a war, an airplane crash, or extraordinarily successful sales of an electronic gizmo or tickets to a newly released film. In addition, they may be subject to a rush of trading as the end of the trading day approaches.
2. Conveyor systems at airports are subject to higher demands during peak travel periods than during quiet periods. The demand on one or more portions of the conveyor system abruptly increases when several planes arrive at a terminal in succession. Similarly, conveyor systems in warehouses and parcel-sorting facilities experience considerable variation in activity by time of day.
3. Telephone call centers and e-commerce systems see considerable increases in traffic during holiday periods. In turn, the call center activity triggers computer activity through order entry. This triggers activity in the warehouses that process the orders. Call centers and the computer systems used to support the transactions performed by the agents may experience surges of activity during the commercial breaks in popular TV shows, such as sporting matches, if commercials exhort the viewers to buy something.
4. The processing of computerized images used in medicine or surveillance may go through different phases, each of which makes particular demands on the system as a whole. These demands can be for processing time, network bandwidth, memory, I/O, retrieval and storage, or some combination of these, at any time.
5. Fire alarm systems experience bursts of activity when smoke is detected. The level of activity may increase as the fire spreads.

6. In the United States and other countries, suppliers of income tax preparation software and tax filing services experience huge spikes in demand as the annual tax filing deadline approaches but experience little demand the rest of the year.

If time-varying behavior is not understood, there is a risk that the system will be engineered to cope only with average loads rather than peak ones. The consequences of an inability cope to with load variations vary from system to system. For the preceding examples:

1. If a securities trading system lacks the ability to process spikes in transaction volume in a timely manner, the prices of at least some of the securities may reach undesirable levels before trades are executed, perhaps resulting in losses or in missed opportunities to execute stop loss orders, that is, orders to sell when the price achieves a chosen level. Transactions will not be executed at all if the packets describing them are lost because of network congestion. Long transaction response times by online traders at home may raise concerns about whether a trade has been executed at all, especially if the market is volatile at the time.
2. If a conveyor's control system is overloaded, a suitcase approaching a diversion to the right or to the left might miss its flight because the necessary control messages arrive after the suitcase has passed the diversion point. In that case, the suitcase might have to be sent to an entry point in the sorting system and pass along the conveyor again. Moreover, the wrong suitcase might be diverted and be put on the wrong flight.
3. It is not reassuring to be told by a call center operator that "the computer is slow today." Nor is it reassuring if one cannot get quick response times when trying to order tickets for a popular show or for steeply discounted flights. Users who experience unacceptable delays at web sites may attempt to shop at competing web sites instead.
4. The demands of particular phases of image processing, storage, and retrieval may differ widely. It is important to understand how they differ to ensure that no one phase saturates the system and keeps it from functioning entirely for any length of time.

5. For safety-related systems such fire detection and alarm systems, the consequences of being unable to process bursts of traffic in a timely manner are potentially catastrophic, to the extent that the occupancy of a building may be prohibited by local inspectors until the problem is fixed.
6. A taxpayer could be unfairly penalized for filing a return late if system delays mean that the filing deadline is not met.

In each of these cases, there is the potential for economic loss and, in the case of safety systems, even for loss of life if the computer-based control system cannot cope with the loads to which it is subjected. It follows that computer systems must be engineered to handle spikes in load as well as average loads. To that end, an understanding of the peak workloads is essential.

4.4 Mapping Application Domains to Computer System Workloads

In this section we look at the process of mapping domain-specific actions to specific actions within a computer system, and then illustrate how they may be quantified.

4.4.1 Example: An Online Securities Trading System for Account Holders

A securities trading system is accessed via a browser interface. Account holders access the system to initiate balance queries, track history, and order trades in stocks, bonds, mutual funds, and other types of investments. The load on the system varies by time of day and perhaps by season. Spikes in load may occur for a variety of reasons, including political events or announcements about corporate earnings or losses or changes in interest rates by the central bank. Surges in mutual fund orders may occur during the last hour of trading because their value is based on the values of their holdings when the exchange closes. Balance inquiries and requests for statements might occur at any time but might be more frequent in the evening because some account holders would prefer to view them in the privacy of their homes rather than at work. Statement generation occurs in batch mode at the end of the month, and tax reporting documents, whether in electronic form or paper

form, might be generated in batch mode in January each year. Transaction logging takes place continuously to ensure the existence of an audit trail. Fraud monitoring may occur continuously or after the fact. Reports to regulators may have to be generated at a fixed time, such as 30 minutes after the occurrence of an important event.

It is important to understand when these activities are likely to take place, because each one makes a different kind of demand on system resources. The rates at which the online activities occur and the volumes of seasonal batch activities and the demands they are likely to make on system resources characterize the workload on the system. The performance requirements for distinct application functionalities differ from one another. A user may be concerned if an online trade is not executed promptly, because the strike price might differ from the desired price by the time the trade is completed. A user may be uncomfortable if it takes a few seconds longer than usual to retrieve transaction histories or statements, but that difference in retrieval time will not affect the value of the securities in the account, unlike a delay in completing a trade. These distinctions in desired response times or execution times must be reflected in performance requirements in the design and implementation of the system. The distinctions must be captured when the workloads are characterized so that the correct design and implementation choices can be made.

4.4.2 Example: An Airport Conveyor System

In an airport conveyor system, the principal units of work are the movement of a suitcase from a check-in point or from an arriving plane to its connecting flight, perhaps via a security inspection station, and from planeside to a baggage claim area, perhaps via a customs checkpoint in which it is X-rayed or sniffed by dogs to check for contraband. Each suitcase is associated with a sequence of actions, the one of principal concern being routing from its arrival point in the system to its destination. Upon check-in, each suitcase is equipped with an identifying bar code that is associated with its intended destination via a routing database. The bar code is scanned at various points in the system. A scan may trigger a database query whose result is a list of one or more routing instructions needed to send the suitcase to its correct destination, or simply the writing of a record indicating that the suitcase arrived at a specific point at a particular time. Without loss of generality, a destination may be a plane, a storage location, or a designated baggage carousel in the baggage claim area. To engineer the performance of the conveyor system and the components of the control system, including

the routing database, we need to understand how often database queries occur and develop an understanding of the message transmission pattern as the suitcase progresses from one diversion point to the next.

We may note the following when characterizing the workload attributable to luggage movement:

- Since a database access is triggered by the initial check-in of the suitcase and subsequent scans of the associated bar code, we will need to compute the average number of scans per suitcase passing through an airport, the number of suitcases passing through the airport per hour, and the sum of the rates at which the suitcases pass all the scanners.
- The rate at which suitcases pass a scanner is determined by the speed of the adjacent conveyor and the average distance between the suitcase handles to which the bar code tags have been attached. The conveyor speed and the distance from a scanner to the next diversion point determine how soon the system must know which way to send the suitcase. The faster the belt is moving and/or the shorter the distance between the scanner and the diversion point, the shorter the combination of database query time and message delivery time must be. For example, the response time requirement will be much more stringent if the conveyor is moving at 2 meters per second than if it is moving at 1 meter per second. For a given conveyor speed, the scanning rate will be twice as high if the suitcase handles are 1 meter apart rather than 2 meters apart.
- It may be useful to be able to estimate the distribution (or at least the variance) of the distance between the suitcases, since this will affect the variability in the times between database queries. As we saw in Chapter 3, that affects queue lengths within the system.
- The queue lengths in the system are also affected by the average journey time of a suitcase from check-in to loading into a cargo bin, and the average journey time from an arriving cargo bin to a conveyor in the baggage claim area.

There is also workload associated with monitoring functions. To reduce the risk of outages and to facilitate the targeting of preventive maintenance, control systems monitor the status of various components of the hardware, including the temperatures of motors and power supplies, revolutions per minute, and connectivity to the control system itself.

Depending on the design of the system, the monitoring may take the form of periodic polls to check status and/or the issuance of alarms when certain operating parameters, such as temperature, are exceeded. Excessive temperature could be a sign of bearing wear, and lack of connectivity to a programmable control unit would mean that that piece of the system could not be controlled at all. Finally, a monitoring system is needed to detect jams and other alarm conditions, such as the pulling of a red cord to stop the conveyor altogether. All of this monitoring incurs network, processing, and logging costs. Performance engineers must determine the permissible delays of these functionalities, especially the time to stop the conveyor after the cord is pulled, and ensure that safety monitoring is not impeded by payload functionality, such as moving luggage.

Combining this data with knowledge about the flow of information through the control system, the topology of the conveyor system, and the topology of the network that controls it, we can establish a baseline workload characterization of the system that will eventually be used as input to the preparation of performance requirements and may even impact the system architecture. For this system, the following types of workloads may be identified, each with its own demand characteristics and performance requirements: luggage movement and delivery, system monitoring, and quick response to alarm conditions such as the pulling of a red cord to stop the system altogether. In some systems, logging of luggage movement may also be required as a deterrent to tampering and theft. This logging may also be useful to improve efficiency, since luggage that is misrouted may be returned to the system entry point. This is wasteful.

4.4.3 Example: A Fire Alarm System

A fire alarm system interprets information about environmental conditions and must carry out multiple actions in response. Among these are the closing or opening of vents and doors, notification of the local fire brigade, and the generation of audible and visual alarms to prompt people to evacuate. Fire codes [NFPA2007] specify how soon these things must happen once smoke is detected. The workload for this system in emergency mode is defined by the number of stimuli such as smoke alarms coming in and the number of actions, such as turning on sirens and closing vents, that must occur as a result. The fire code is a source of requirements about how long it should take to carry out the necessary actions to get people out of the building safely.

As with conveyor systems, fire alarm systems may be equipped with their own internal monitoring functions. Apart from monitoring smoke alarms for the presence of smoke, they must repeatedly monitor the functional status of the smoke detectors, horns, pull handles, temperature indicators, and other safety-related devices and issue trouble indicators if any sign of a malfunction arises. This is essential to ensure that the absence of a smoke alarm is also evidence of the absence of smoke, and to ensure that maintenance teams are aware of malfunctions that must be corrected to ensure safety. As with the conveyor system, monitoring actions may be initiated according to a schedule. Some devices may be able to issue their own notifications of malfunctions (known in the United States as troubles), but they can do so only if they have connectivity to a monitoring system.

During a fire, the cost of processing a burst of alarm indicators may impede the processing of indications that sensors are not functioning. This information may be useful to the emergency responders, because detectors and/or connectivity with them may be disabled as a consequence of the fire itself. As part of the workload characterization of the alarm system, it is important to understand the mix of alarm indicators and trouble indicators. When performance requirements are drafted, one should determine whether it is necessary to ensure that trouble indicators are processed even in the presence of heavy alarm traffic.

From the foregoing, it is clear that there are at least three types of identifiable workloads in the fire alarm system, each of which makes its own resource demands and each of which is associated with performance requirements: alarm notification, trouble notification, and maintenance and monitoring activities. When the system is architected and implemented, care must be taken to ensure that the performance needs of each of them are accounted for individually and together.

4.5 Numerical Specification of the Workloads

The numerical characteristics of the workloads should be specified in terms of performance metrics that have all of the desirable properties we described in Chapter 2. In particular, they must be informative about the domain of application and be measurable. One must also be sure that factors that might affect memory occupancy and object pool usage, such as the anticipated durations of login sessions, are taken

into account when characterizing the workload, since a lack of such resources can result in considerable performance degradation, or even a system malfunction.

The identification and numerical specification of the workloads are preludes to writing the performance requirements that the system must meet. Let us illustrate these points using some of the examples mentioned earlier.

4.5.1 Example: An Online Securities Trading System for Account Holders

We can readily identify three types of workloads for this system: the transactions initiated by the users, the report generation workload consisting of statement processing and the generation of reports for the tax authorities, and a fraud or anomalous transaction detection workload that we cannot quantify because the brokerage house will not tell us how it works.

For the transaction aspect of the system, the number of users who might be logged into this system simultaneously cannot be larger than the number of account holders. Unless the trading system is meant exclusively for day traders, or unless there is some unusual event in the market, such as an initial public offering or an impending market crash, the number of users logged in simultaneously will be much lower than that. Since the system must be engineered to perform well during the peak hour of the day, the workload should be specified in terms of the load anticipated then. The following quantities are used in specifying the user transaction workload:

- The number of login sessions initiated in the peak hour
- The average duration of a login session
- If the system is already in production, the average and peak numbers of login sessions observed during the peak hour
- The average number of transactions of each type in a session—for example, balance inquiry, price inquiry, statement request, purchase of a security, sale of a security, placement of a limit order, and so on
- The frequency with which transaction logging and audit functions are activated in the background, and the amount of work they have to do as a function of the rate at which the transactions and login sessions occur

These figures tell us something about the anticipated external load on the system. Using Little's Law as described in Chapter 3, we can compute the average number of concurrent login sessions in the peak hour: it is the average duration of a login session multiplied by the number of login sessions per hour in the peak hour. This quantity will figure in the performance requirements for the number of concurrent sessions to be supported.

The following quantities may be of interest for the report generation workload:

- The number of account holders, the number of transactions recorded on each statement, and the number of different securities held by each account holder
- The number of statements to be generated each month
- The number of tax reports to be generated shortly after the end of each tax year

4.5.2 Example: An Airport Conveyor System

The characterization of the workload for the network of computers and programmable logic units controlling a conveyor system clearly depends on the size of the airport, the volume of arriving and departing flights in the peak hour, the proportion of luggage that must be routed from one plane to another, how far the luggage must travel, and what kind of security facilities it must go through (explosives detection chambers, X-ray machines, etc.).

When characterizing the workload, the performance engineer and other stakeholders must work together to identify and quantify the load drivers at various points in the system. Two workloads may be readily identified: the suitcase movement workload and the monitoring workload.

The suitcase movement workload might be described by the following quantities:

- The number of suitcases checked in during the busy hour.
- The number of suitcases in transit recorded during the busy hour. These will be recorded at the originating airport and transmitted to a transit airport before their arrival on an inbound plane.
- The spacing of suitcases on the belts.

- The numbers of bar code queries per hour from each scanner.
- The number of diversion points.
- The lengths of the conveyor segments.

The monitoring workload is used to ensure the continuous function and prompt repair of the various components of the conveyor system. The workload might be described by the following quantities:

- The number of motor devices being monitored.
- The number of polls of motor status per motor device per hour.
- The size of each polling status message for each motor. The message might include motor temperature, whether it is running, whether it is getting a clean power supply at the right voltage and amperage, and so on.
- The number of other hardware devices being monitored, the number of polling messages per hour associated with each one, and the sizes of those messages.
- The number of program logic controllers and other networked elements in the conveyor system, and the frequency and size of each type of associated status message.
- The actions to be taken upon receipt of the response to a polling message, or the absence of a response to a polling message for whatever reason.

4.5.3 Example: A Fire Alarm System

A fire alarm system might be characterized by the number of alarm monitoring stations in a facility, the extent to which they are connected with one another, the kind of information they exchange and how often they exchange it, the maximum number of alarm and equipment trouble messages that might arrive in a burst, and the number of notifications and commands that an alarm monitoring station must issue in case of an emergency. Examples of notifications might include an automatically generated call to the fire department, exchanges of alarm information with other monitoring stations, and the activation of alarms and lights. Examples of commands might include closing dampers and doors and generating programmed voice announcements instructing people to evacuate the building. In addition to receiving notifications from each other, alarm monitoring stations receive notifications from smoke detectors, heat detectors, and the like, which the

local program logic must turn into commands to be acted upon by automated devices and safety personnel. In addition to the emergency workload, there is a background workload that repeatedly checks the status of monitoring devices. These devices include, but are not limited to, smoke detectors, communication devices such as alarm boxes, and an automated telephone dialer to alert the fire department.

The three example systems have very different traffic patterns:

- The online securities trading system experiences a steady stream of work when the markets are open, with bursts of activity provoked by events such as profit announcements or natural disasters.
- The airport conveyor system's operations are somewhat rhythmical. Suitcases tend to be equally spaced along the most heavily traveled segments of the conveyor, which is moving at constant speed. Traffic will be more intense during peak travel periods. Monitoring in the background is very rhythmical. It is essential to the smooth functioning of the conveyor and to the expeditious handling of repairs.
- The fire alarm system has two distinct traffic patterns. Monitoring of detection devices and output devices occurs at regular intervals to ensure functionality in an emergency. When the emergency occurs, bursts of traffic must be cleared within a very short time interval to ensure the timely activation of alarms and other activities, such as closing ventilators and doors.

4.6 Numerical Illustrations

We now illustrate our examples with some numerical data. All of the data described here is fictional. None of it has been obtained from measurements. It should be understood that the data from your system could be very different.

The scenarios captured in the following tables are entirely fictitious. They are presented for the purpose of illustration only. A domain expert might challenge some of the values and ask for refinements based on observations or on knowledge of how the system is used. That is a normal part of the performance engineering process. It is also desirable, because it is easier to do quality performance engineering work if the domain experts are part of the engineering process.

4.6.1 Numerical Data for an Online Securities Trading System

Table 4.1 shows an example workload characterization for an online securities trading system. The principal online work driver is a login session. The expected numbers of transactions of each type in a login session are estimates based on user experience or on measurements of existing systems. Notice that the combined numbers of buy and sell orders in the peak hour are equal to the number of login sessions multiplied by the sum of the probabilities of each type of transaction occurring within a session. This consistency is an essential feature of a workload specification.

Table 4.1 *Example Trading System Workload Specification*

Quantity	Value
The number of login sessions initiated in the peak hour	100
The number of login sessions ended in the peak hour	100
The average duration of a login session	10 minutes
Calculated average number of sessions (100/hour × 10 minutes) = (100/hour × 0.167 hours)	16.7 sessions logged in
Maximum allowed number of login sessions	100
If the system is already in production, the average and peak numbers of login sessions observed during the peak hour	Compare this with the calculated value
The average number of buy transactions in a session	0.8
The average number of sell transactions in a session	0.8
The average number of limit order transactions in a session	0.05
The average number of balance inquiries in a session	0.95
The average number of statement requests in a session	0.05
Numbers of buys and sells during the peak hour	160
Statements generated at the end of each month	1,000,000
Logging of user interactions	1 record per mouse click
Transaction history requests generated online per session	0.3
The frequency with which transaction logging and audit functions are activated in the background, and the amount of work they have to do as a function of the rates at which the transactions and login sessions occur	1/minute
Fraud monitoring activities	Unknown

The table does not tell us when or how often fraud monitoring activities occur. Nevertheless, fraud monitoring is a critical activity with performance requirements and demands of its own. Fraud detection logic might be invoked during or after each transaction. It could also be an ongoing activity that occurs in the background. The cost of background processing might increase with the transaction volume, or it could be constant. Even if the security team does not wish anything to be shared about fraud detection logic, including processing costs and performance requirements, enough processing and storage resources must be provided to ensure that fraud detection is timely and that it does not interfere with the applications of interest. In the absence of detailed information, performance engineering to support fraud detection might be done by making assumptions about resource costs and performance requirements, and flagging these assumptions in requirements documents and the descriptions and parameterizations of performance models and their predictions. The parameters may be varied between their assumed best and worst cases so that a range of impacts on overall performance can be determined.

4.6.2 Numerical Data for an Airport Conveyor System

Table 4.2 shows data for an entirely fictional conveyor system with components from one or more fictional vendors. The conveyor system

Table 4.2 *Illustrative Parameters Describing the Workload of a Conveyor System*

Quantity	Value
Space between suitcase handles	1.5 m
Conveyor speed	1 m/sec
Bags checked in per hour	3,000
Departures per hour	15
Bags transferred between flights per hour	1,000
Arriving flights per hour	15
Bags claimed at this airport per hour	3,000
Induction points (entry points for luggage)	30
Bar code scanners	50
Diversion points	50
Programmable logic controllers	25
Number of status monitors	80
Number of status messages per monitor per hour	60

serves an airport with 15 departures and 15 arrivals during the peak hour. By way of comparison, Chicago's O'Hare International Airport (ORD) handles about 200 traffic movements in the peak hour [Hilkevitch2013]. These figures contain implicit assumptions about the number of passengers on each plane, the dimensions of suitcases, and the number of suitcases checked by each passenger. The number of diversions (junctions) in the conveyor system depends on the conveyor's topology. The number of program logic controllers and the number of status messages per device monitor depend on the system implementation and the technology.

4.6.3 Numerical Data for the Fire Alarm System

The numerical data in Table 4.3 illustrates the configuration for a hypothetical fire alarm system in a fictional office building, supplied by an entirely fictional vendor. The building has three levels, with 30 rooms per level (including restrooms, janitors' closets, and open spaces), two staircases, two elevators, a main entrance, and four emergency exits at ground level. There are smoke detectors in every room, on every stair landing, and above every exit door. There are five pull stations on every level, one on each stair landing, and five alarm devices (a pull station is a device with a handle that can be pulled by a person detecting a fire).

Table 4.3 *Numerical Data for a Fire Alarm System*

Quantity	Value
Number of smoke detectors	100
Time between status messages sent by each smoke detector or other device during the quiet period	5 minutes
Time between alarm messages for each alarmed smoke detector	10 seconds
Time between alarm messages for each activated pull station	10 seconds
Time between trouble messages sent by a malfunctioning device	15 seconds
Number of pull stations	20
Maximum number of alarm messages sent by a smoke detector once in the alarmed state	30
Maximum number of alarm messages sent by an activated pull station	30
Number of alarm panels	1
Number of enunciators	20

There are five enunciators (alarm bell, strobe light, or loudspeaker for playing stored messages instructing evacuation) on every floor. There is one alarm control panel for the entire building, near the main entrance. When there is no emergency, each smoke detector sends a status message to the alarm control panel every 5 minutes. Each smoke detector that has "smelled" smoke sends a message to the alarm control panel every 10 seconds. The alarm panel logs all messages from the pull stations and smoke detectors and displays the ten most recent ones to come in on a liquid crystal panel. This data is summarized in Table 4.3.

4.7 Summary

Workload identification proceeds naturally from an examination of the sets of functionalities of a system and the time patterns of their invocation. The specification of the invocation patterns is complemented by a description of the numbers of entities of various types involved in the functionality and perhaps their expected sojourn time in the system. Numerical specifications of the workloads must be mutually consistent, so as to avoid confusion about the scale of the system and how many user components the system must support. As we shall see in Chapter 5, this is a prerequisite for the correct identification of system performance requirements.

4.8 Exercises

4.1. A web-based news service allows viewing of the front page of a newspaper, the display of stories shown on the home page, and, for premium subscribers, access to all the news stories posted on the site in the last ten years. Premium users must be registered in the system and then may log in and out to access the story database as much as they desire. Payment for access may be by a period subscription or by the article viewed.

(a) Identify the activities a subscriber or an unsubscribed reader may perform.

(b) Describe the set of activities a journalist may perform on the site.

(c) Describe the set of activities an editor may perform on the site.

(d) Describe the set of activities a layout editor might perform on the site.

(e) Identify the workloads on this system. Give estimates for the frequencies with which each type of activity is performed.

(f) Explain how the workloads will be characterized if the site owner's mission is to provide editorial content and news articles according to the rhythm of a printed newspaper. Contrast this with a news service whose stated mission is the immediate display of breaking stories.

4.2. The design of a fabric woven on a continuous loom is implemented by a system that controls how different-colored threads are woven in. The fabric is advanced along the loom each time a set of shuttles has jumped across it. Complicated designs with many colors require more shuttle excursions than simple designs or a stretch that is just one color. The movement of the fabric and the placement of the finished weave on a roller are fully automated. The process is managed by a loom controller that also monitors the state of various pieces of machinery, including the amount of remaining thread on each shuttle. The design is stored on a disk drive accessible to the controller. Your task is to identify the workloads performed by the loom controller in conversations with the mill manager and a technician who tends the loom.

(a) Identify the events that are controlled by the loom controller.

(b) Identify the events that are monitored by the loom controller.

(c) Explain how you would determine the amount of shuttle activity occurring in the loom controller. Is this affected by the complexity of the design?

(d) Explain the monitoring activities of the loom controller and describe how often they would occur. Does this depend on the complexity of the design?

(e) Identify the workloads on this system. Give estimates for the frequencies with which each type of activity is performed.

Chapter 5

From Workloads to Business Aspects of Performance Requirements

We build a bridge from workload identification to performance requirements, explore how performance requirements relate to the software lifecycle, and explore how performance requirements fit into a business context, particularly as they relate to the mitigation of business risk and commercial considerations. We also describe criteria for ensuring that performance requirements are sound and meaningful, such as unambiguousness, measurability, and testability.

5.1 Overview

Poor computer system performance has been called the single most frequent cause of the failure of software projects [SmithWilliams2001] and is perceived as the single biggest risk to them [Bass2007]. The principal causes of poor performance are architectural choices that are

inappropriate for the intended scale of the system and inadequately specified performance requirements. In our experience, performance requirements may be vaguely written or might not even have been written at all by the time the software project is close to completion. At that point, serious concerns will arise about whether the customers' and users' expectations have been met. In the absence of well-written performance requirements or any performance requirements at all, it will be difficult to resolve any conflicts over what those expectations were and how they have been met. Thus, performance requirements are essential to the management of customer expectations, to the verification that customer expectations have been met, to the assurance of proper delivery after development, and to the avoidance of conflicts about these expectations.

The absence of sound performance requirements increases the risk that performance will receive inadequate attention during the architectural, development, and functional testing phases of a software project. Performance requirements are key drivers of computer and software architecture. Since performance problems often have their roots in poor architectural decisions, the early establishment of performance requirements for a new system is crucial to the project's success. As we shall see, the performance requirements must be formulated in terms of an agreed set of metrics that meet the criteria for metrics set out in Chapter 2, including a link between the choice of metrics, the values they take, and business and engineering needs.

In the remainder of this chapter we shall briefly discuss the transition from workload identification to performance requirements, the relationship between performance requirements and product management, the role of performance requirements in development and performance testing, and the role of performance requirements in formulating contracts and mitigating business and engineering risks. Finally, we examine criteria that make a performance requirement sound.

5.2 Performance Requirements and Product Management

A software system that does not meet the performance needs of the user or customer as the user perceives them is unlikely to achieve much success in the marketplace. Nor is a system that is scaled to perform at a much higher level than is perceived to be required if the cost of doing so

is a lot more than the customer is willing to spend. To complicate matters further, one cannot be sure that the customers themselves are able to quantify their needs well, perhaps because of market uncertainties or because they do not have adequate data to forecast their needs properly. In many organizations, the product manager is the interface between the customer and the architecture and development teams. The product manager's guidance can be useful in identifying and interpreting customer expectations to the architects and developers. Interpreting expectations about performance may be especially difficult when both the customer and product manager are uncertain about them. In this section we discuss some techniques for overcoming organizational uncertainty about performance needs. We also discuss the linkage of performance requirements to business, regulatory, and engineering needs. This linkage is essential to establishing a baseline set of performance requirements, while ensuring that physical constraints are taken into account.

5.2.1 Sizing for Different Market Segments: Linking Workloads to Performance Requirements

When product managers are responsible for identifying the desired features and performance characteristics of systems that are intended for small-scale and large-scale installations with a wide variety of needs, they are faced with the daunting task of identifying the performance requirements and performance capabilities in terms that are credible and appealing to a broad base of customers, perhaps while keeping cost parameters in mind. The performance requirements for one customer's system may not be suitable for another's, because the sizes of their systems and the mixes of tasks they do may be very different. Market segments, application domains, and regulations may evolve during the years that might be needed for a complete software development and testing cycle, while architecture and design decisions must be made early on. These in turn are heavily influenced by performance needs, which may not be completely understood.

One way to deal with this conundrum is to derive performance requirements for different market segments from reference workloads and reference scenarios like those described in Chapter 4. Basing the performance requirements on reference workloads has the following benefits:

- It enables the early identification of a system architecture and a system design that are intended to meet a wide range of performance needs.

- It allows performance testers to begin work on performance test suites to validate performance requirements in time for use once functional testing is completed.
- It enables the performance testers to design the performance test environment to accommodate deviations from the performance requirements.
- When negotiating with customers, the product managers will be able to point to reference sets of performance requirements and performance test results that show they have been met.

5.2.2 Performance Requirements to Meet Market, Engineering, and Regulatory Needs

All performance requirements must be linked to business and engineering needs. Linking to a business need reduces the risk of engineering the system to meet a requirement that is unnecessarily stringent, and linking to an engineering need helps us to understand why the requirement was specified in the first place. An example of a business need is the desire to provide a competitive differentiator from a slower product. An example of an engineering need is that a TCP packet must be acknowledged within a certain time interval to prevent timeouts. Another example of an engineering need is the standards requirement that an alarm be delivered to a console and/or sounded within a maximum amount of time from when the corresponding problem was detected [NFPA2007]. This is an engineering need because timely delivery of notifications is essential to the timely execution of other functions, such as the activation of a noise-making device or the activation of a mechanism for closing a door or a vent.

For mission-critical systems, a product manager must be able to convince a customer that a system will deliver good performance under various types of operating conditions, and also that the system will meet the performance requirements dictated by regulations and standards, whether these are set by government bodies, insurance companies, or industry associations. Even for systems that are not mission critical, the product manager must be able to convince a customer that the system will meet performance needs dictated by market conditions, or even that the performance requirements met by the system will be a competitive differentiator for the customer.

Examples of market-related performance needs include

- The ability to complete a particular action faster than on present systems
- The capability of supporting a higher level of throughput than on present systems
- The capability of supporting a broader set of functionalities than the current ones, while maintaining better or equal performance for the existing functionalities
- The capability of supporting larger numbers of customers or abstract objects than on present systems
- Higher network bandwidth than is presently available
- The ability to support administrative functions without diminishing the performance of payload functionality

Examples of performance-related engineering needs may be specific to the domain. In TCP/IP networks:

- The network round-trip times must be short enough to avoid timeouts and to avoid throttling traffic by causing the sliding window to close because the product of bandwidth and round-trip time exceeds the window size. Of course, the ability to meet this need may be constrained by physical realities, such as the lower bound on propagation delay for a given distance imposed by the speed of light.
- Bandwidth utilization must be low enough to avoid buffer overflows, since these lead to packet loss, the closure of the sliding window, and a potentially large retransmission rate.

On some railways, the status of the next signal on the line is delivered to a display on the engine driver's console. If the signal status message is not delivered by the time the train has reached the last point at which the train can be smoothly and safely brought to a stop, either the driver or an automatic system must activate the brakes. The train must then remain stationary until the signal status arrives. If the corresponding signal is red, nothing has been lost. If the corresponding signal is green, time will be lost, and there will be an energy cost associated with getting the train moving again, as well as wear and tear on the brakes and tracks. Therefore, a performance requirement is needed for the delivery time of signal status messages in terms of the train's speed and the associated stopping distance. In systems that control chemical

reactions, the system must respond to changes in temperature early enough to activate controls that reduce the temperature or cause valves to be opened. In some applications, there are regulations that influence performance requirements. For instance, fire alarms must be sounded within 5 or 10 seconds of smoke being detected. This is handled by an alarm control system [NFPA2007].

5.2.3 Performance Requirements to Support Revenue Streams

Many systems are tied to revenue models that depend on sound performance. Performance requirements constitute a useful mechanism for specifying what fraction of revenue loss can be tolerated if performance is insufficient. They are also a useful mechanism for specifying the quality of service (QoS) that must be provided.

In Chapter 3 we saw how the Erlang loss model can be used to estimate the number of telephone circuits between two points needed to carry designated volumes of calls of a given average duration with a given call loss probability. A performance requirement would stipulate that the fraction of calls lost would be less than some small quantity, such as 10^{-6}. If the call volume in the busy hour and the average revenue per call are known, the average revenue loss associated with lost calls can be easily computed. Similarly, stringent performance requirements must be written and followed to ensure that the call billing records are not lost. Failure to prevent the loss of billing records can result in lawsuits being brought against the equipment vendor by the telephone carrier.

Securities trading systems have very stringent requirements about transaction loss. Because of the need for auditing and because of the sums that can be lost if trades are not executed in a timely manner, trading systems must be engineered to have extremely low loss probabilities. A correspondingly stringent performance requirement is needed to ensure that measures are taken to avoid transaction losses at every phase of the software lifecycle. It is worth noting that a failure to execute a trade would result in a lost commission at the brokerage house, quite apart from the damage that would be suffered by the account holder.

A quick response time is a competitive differentiator for a web site. To ensure that the average response time is short and does not vary much under heavy loads, performance requirements should be written that specify the desired values of the average response times for various transaction types, together with the value of the (small) fraction of transactions whose response times may exceed a larger specified value.

5.3 Performance Requirements and the Software Lifecycle

Like functional requirements, performance requirements should be formulated as early as possible in the software lifecycle, preferably before the system architecture is decided [BPKR2009]. There are a number of reasons for this:

- Satisfactory system performance depends heavily on architectural choices being made to support it. This means that expectations about system performance should be well specified and commonly understood.
- Customer expectations about system performance can be managed and met only if both customer and supplier understand the metrics to be used to describe performance and the level of performance to be provided.
- The early specification and review of performance requirements allow timely decisions to be made about whether the cost of meeting the requirements is appropriate or excessive.
- Performance is widely seen as the biggest risk to the success of a system [Bass2007] and has been mentioned as the single largest cause of project cancellations. Careful attention to performance requirements can mitigate this risk.

The author is aware of organizations that now demand that every functional requirement have associated performance requirements. This is an interesting strategy for creating awareness of the performance ramifications of functional requirements and for creating a shared understanding of how fast the corresponding functionality must be executed, as well as how many objects might be involved with the functionality. This practice could be especially useful for capacity planning in highly modular systems or in service-oriented architectures, since the introduction of new applications creates new demands for the services. The main risk associated with this practice is that the proliferation of performance requirements and their origination among different sources may lead to inconsistencies that must be resolved before implementation and performance testing. To mitigate this risk, the performance requirements associated with functional requirements must be reviewed for mutual consistency and achievability before they are approved.

Performance requirements are also essential for the development of sound performance tests. The absence of performance requirements places a burden on the performance tester to identify the ranges of workloads to which the system should be subjected before delivery. In such cases, performance testers must make conjectures about the anticipated system load and use case mix, and then devise load tests accordingly. The tester must also make and state assumptions about what values of such performance metrics as throughput and response times will be acceptable. Forcing the tester to do either or both of these incurs a huge business risk:

- If the system is tested at a lighter load than will occur after installation and is declared to have passed, performance may well be unacceptable to the customer once the system is in production.
- On the other hand, if the system performs badly under much heavier loads than would occur in production, and management is not aware of the discrepancy, the project might needlessly be canceled. This could result in ill will, needless costs, and even job loss and litigation.

Therefore, it is absolutely essential that performance requirements and the load tests derived from them be carefully tailored to the load that is anticipated in production. Since it is very often the case that performance testing will be done on a system that is smaller than the one that will be used in production for cost reasons, performance requirements regarding the load must be specially written for the small-scale system. As we shall see in Chapter 9 on performance testing, the performance test results must then be interpreted in the context of an architecture review. One of the goals of the architecture review should be to verify, by modeling or otherwise, that the system capacity can be increased by scaling the system up, installing faster processors or multiple processors and perhaps more I/O devices, and memory or scaling the system out by adding parallel replicates of the system under test.

5.4 Performance Requirements and the Mitigation of Business Risk

One of the purposes of writing performance requirements is the formal specification of the performance expectations of the system in quantifiable, measurable terms that are related to the needs of the business and

that account for the engineering constraints of the problem domain. This is a prerequisite for ensuring that the system's performance will meet market expectations and/or contractual obligations in a cost-effective manner, while laying the groundwork for performance verification through performance testing.

We have found that poorly written performance requirements incur an insidious cost. They cause confusion among the developers charged with meeting them, as well as among the performance testers who must verify that they are met. The confusion must be resolved in meetings in order to understand what was meant. In the author's experience, clarification of the performance requirements often means rewriting them in keeping with the spirit in which they were meant, and then communicating the revisions to the various stakeholders for approval. The confusion can be reduced if performance requirements are written in terms of obtainable metrics whose meanings are clearly defined and that will actually be measurable when the time comes to test the performance of the system as a whole and/or the performance of its components.

By contrast, clearly specified performance requirements represent goals that architects, developers, functional testers, and performance testers can bear in mind when carrying out their respective tasks. This should aid the smooth execution of an implementation process in which the ability to meet performance requirements is clearly tracked throughout the software lifecycle. Once performance requirements have been drafted, the performance risks inherent in a chosen architecture, such as the choice of platform or operating system, can be mitigated by conducting performance benchmarking tests of the platform components on which the system can be implemented before full-blown development commences. This prevents money from being wasted on the development of large amounts of software that is based on the application program interfaces (APIs) of unsuitable environments. For example:

- In [AvWey1999], the unfortunate choice of an architecture was avoided when early performance tests based on early performance requirements showed that the synchronization operations in the operating system kernel were too slow to support the intended application.
- In [MBH2005], performance tests based on an early draft of performance requirements showed that one of the service environments under consideration was so inefficient that throughput objectives would not be achieved if it were used.

In both cases, performance problems that would have been difficult to overcome without extensive (and therefore expensive) reworking of the code were averted. The resulting delays in the time to market were avoided by the timely specification of clear performance requirements and early performance testing of the platforms or environments under consideration.

5.5 Commercial Considerations and Performance Requirements

Disclaimer: This section does not contain legal advice. You should seek the advice of legal counsel when drafting any agreements or documents incorporated into agreements by reference. Legal obligations and practice may differ from one jurisdiction to another. The author is not a lawyer.

5.5.1 Performance Requirements, Customer Expectations, and Contracts

Whether or not they are well formulated, performance requirements are a key ingredient of customer expectations of what the system will do. Therefore, they may constitute part of an agreement about what the supplier is supposed to deliver. Poorly drafted requirements increase the prospect of incurring customer ill will, which can have undesirable consequences, including loss of business and even litigation. It follows that the performance measures described in the agreement be expressed in terms of quantities that can actually be measured, and that the performance requirements mentioned or referred to in the agreement be testable. Otherwise, the performance requirements and the contractual agreements to which they relate may not be enforceable. This could adversely affect the customer and undermine the reputation of the supplier.

5.5.2 System Performance and the Relationship between Buyer and Supplier

Situations may arise in which the supplier has greater expertise in system performance than the buyer, or vice versa. In the author's experience, both are possible whether the buyer is a start-up and the supplier

is established, both are start-ups, both are established, or the supplier is a start-up and the buyer is established. In any of these cases, transparency and adherence to commonly accepted guidelines for writing requirements, such as those prescribed by [IEEE830] for software requirements documents, will go a long way toward preventing misunderstandings and disputes regarding performance requirements and the interpretation of performance test results. The guidelines in [IEEE830] describe the sections a requirements document must include, the form each section should take, and criteria for the soundness of functional and performance requirements. Some of these are discussed in Section 5.6.

5.5.3 Confidentiality

A great deal can be inferred about the competitiveness of a product or the commercial position of the intended customer by examining performance requirements. For example:

- The ability of an online order entry system or call center to handle transactions at a given rate in the busy hour may be an indicator of the owner's anticipated growth, with consequent impacts for revenue and market share. This intelligence could be valuable to a competitor or an investment analyst trying to forecast the future earnings of both the buyer and the supplier.
- The ability of a network management system to handle traps at a given peak rate, combined with knowledge of the number of nodes to be managed and the peak polling rate, can tell us about the intended market segment of the product while nourishing speculation about the product's feature set, or even about the nature of the site the system is intended to support. This can affect price negotiations between supplier and buyer, and perhaps the supplier's share price.

These examples illustrate why performance requirements and any contractual negotiations related to them should be treated as confidential and perhaps even covered by nondisclosure agreements (NDAs). The release of performance requirements and performance data outside a circle of individuals with a need to know should be handled with great care. Engineering, marketing, legal, and intellectual property departments should all be involved in setting up a formal process to release performance data to third parties under nondisclosure agreements or to the general public.

5.5.4 Performance Requirements and the Outsourcing of Software Development

The specification of both performance and functional requirements is crucial to successful delivery when software development is to be done by an outside supplier. The outsourcing party should have a clear understanding of the performance impact of the outsourced component on the rest of the system so that performance and functional requirements can be clearly conveyed to the outside supplier. Moreover, a formal process should be defined for the specification of the performance requirements and performance testing of the deliverable throughout the development process of the deliverable. This will reduce the risk of taking delivery of an inadequate software component whose performance undermines that of the system as a whole.

5.5.5 Performance Requirements and the Outsourcing of Computing Services

The performance requirements for outsourced computing services, such as those performed in the cloud, should be clearly communicated to the provider, preferably as part of a formal process that includes performance testing under conditions similar to those in production. The process might also include monitoring demand for the services and monitoring designated performance metrics of those services in production, to ensure that performance requirements continue to be met.

5.6 Guidelines for Specifying Performance Requirements

Because of their significance to the commercial success, engineering success, and even the safety of a product, it is important that performance requirements be clearly understood and measurable. One of the reasons they must be clearly understood is that the system supplier and the customer must read and interpret them the same way if functionality of the system is to be ensured and if disputes are to be avoided. There must also be clear agreement about the origin and correctness of the measurements that are used in the requirements. In this respect, performance requirements must meet criteria that are very similar to those for functional requirements. The IEEE guidelines for functional

requirements [IEEE830] are also applicable to performance requirements and mention performance requirements explicitly. In this section we examine the relationship between functional and performance requirements, and then examine criteria for performance requirements that are needed to make them sound.

5.6.1 Performance Requirements and Functional Requirements

The criteria for specifying performance requirements are a superset of those in [IEEE830] for functional requirements. In particular, like functional requirements, performance requirements must be unambiguous, traceable, verifiable, complete, and correct. Additional criteria relate to the quantitative nature of performance requirements. To be useful, they must be written in measurable terms, expressed in correct statistical terms, and written in terms of one or more metrics that are informative about the problem domain. They must also be written in terms of metrics suitable for the time scale within which the system must respond to stimuli. In addition, the requirements must be mathematically consistent. We now elaborate on each of these criteria in turn.

5.6.2 Unambiguousness

First and foremost, a performance requirement must be unambiguous. Ambiguity arises primarily from a poor choice of wording, but it can also arise from a poor choice of metrics.

Example 1: "The response times shall be less than 5 seconds 95% of the time."

This requirement is ambiguous. It opens the question of whether this must be true during 95% of the busy hour, during 95% of the busiest 5 minutes of the busy hour (both of which may be hard to satisfy), or during 95% of the year (which might be easy to satisfy if quiet periods are included in the average). In any case, the response time is a sampled discrete observation, not a quantity averaged over time.

Consider an alternative formulation:

Example 2: "The average response time shall be 2 seconds or less in each 5-minute period beginning on the hour. Ninety-five percent of all response times shall be less than 5 seconds."

This requirement is very specific as to the periods in which averages will be collected, as well as to the probability of a sampled response time exceeding a specific value.

Example 3: "The system shall support all submitted transactions."

Requiring that a system shall support all submitted transactions is ambiguous, because:

- There is no statement of the rate at which transactions occur.
- There no statement of what the transactions do.
- There is no explicit definition of the term *support.*

Instead, one might state that the submitted rate of transactions is 5 per second, or (equivalently) 300 per minute. If this requirement is coupled with an unambiguous response time requirement like that given in Example 2, and by a further requirement that no errors occur while the transactions are being handled, we may be able to say that the transaction rate is being *supported* if the response time and transaction loss rate requirements are also met. We may also be able to say that a desired transaction rate is *sustainable* if all resources in the system are at utilization levels below a stated average utilization that is less than saturation (e.g., 70%) to allow room for spikes in activity when this transaction rate occurs.

5.6.3 Measurability

A well-specified performance requirement must be expressed in terms of quantities that are measurable. If the source of the measurement is not known or is not trustworthy, the requirement will be unenforceable. Therefore, it must be possible to obtain the values of the metric(s) in which the requirement is expressed. To ensure this, the source of the data involved in the requirement should be specified alongside the requirement itself. The source of the data could be a measurement tool embedded in the operating system, a load generator, or a counter generated by the application or one of its supporting platforms, such as an application server or database management system. A performance requirement should not be adopted if it cannot be verified and enforced by measurement.

Example 4: The average, minimum, and maximum response times during an observation interval may be obtained from a commercial load generator, together with a count of the number of attempted, successful, and failed transactions of each type, but only if the load generator is set up to collect them. A performance requirement expressed in terms of these quantities should be written only if they are obtainable from the system under test or from the performance measurement tools available in the load drivers.

Example 5: The sample variance of the response times can be obtained only if the load generator also collects the sum of the squared response times during each observation interval, or if all response times have been logged, provided always that at least two response times have been collected. A performance requirement that refers to the variance of a quantity should be specified only if the variance is actually being collected.

5.6.4 Verifiability

According to [IEEE830], a requirement is verifiable ". . . if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable." For performance requirements, this means that each requirement must be testable, consistent, unambiguous, measurable, and consistent with all other performance and functional requirements pertaining to the system of interest.

Where a performance requirement is inherently untestable, such as freedom from deadlock, a procedure should be specified for determining that the design fails to meet at least one of the three necessary conditions for deadlock. These are circular waiting for a resource, mutual exclusion from a resource, and nonpreemption of a resource [CoffDenn1973]. On the other hand, if deadlock happens to occur during performance testing, we know that the requirement for freedom from it cannot be met. We also know that there is a possibility of a throughput requirement not being met, since throughput is zero when a system is in deadlock.

5.6.5 Completeness

A performance requirement is complete if its parameters are fully specified, if it is unambiguous, and if its context is fully specified. A requirement that specifies that a system shall be able to process 50,000 transactions per month is incomplete because the type of transaction has not been specified, the parameters of the transaction have not been specified, and the context has not been specified. In particular, to be able to test the requirement, we have to know how many transactions are requested in the peak hour, and then have some context for inferring that the peak hourly transaction rate is functionally related to the number of transactions per month. We also have to define a performance requirement for the acceptable time to complete the transaction.

5.6.6 Correctness

In addition to being correct within the context of the application to which it refers, a performance requirement is correct only if it is specified in measurable terms, is unambiguous, and is mathematically consistent with other requirements. In addition, it must be specified with respect to the time scale for which engineering steps must be taken.

5.6.7 Mathematical Consistency

There are multiple aspects to the mathematical consistency of performance requirements:

- Performance requirements must be mathematically consistent with one another. To verify consistency, one must ensure that no inference can be drawn from any requirement that would conflict with any other requirement and inferences drawn from it. Inferences could be drawn through the use of models. They could also be drawn by deriving an implied requirement from a stated one. If the implied requirement is inconsistent with other requirements, so is the source requirement.
- Each performance requirement must be consistent with stated performance assumptions, such as traffic conditions and engineering constraints. For example, a message round-trip time should be less than the timeout interval, while the product of the processing time and the system throughput must be less than 100% so that the CPU is not saturated.
- The performance requirements must not specify combinations of loads and anticipated service times that make it unachievable. This will happen if the product of the offered traffic rate and the anticipated average service time of any device is greater than the number of devices acting in parallel. If there is only one device on its own for a particular function, the product must be less than one.

5.6.8 Testability

We desire that all performance requirements be testable. By testable, we mean that a cost-effective, repeatable method exists for running an experiment that enables us to obtain a designated set of performance measurements under defined conditions in a controlled, observable

environment. Testability is closely related to measurability. If a metric mentioned in a performance requirement cannot be measured, the performance requirement cannot be tested.

Not all performance requirements are directly linked to the ability to attain specific values for metrics. Moreover, such requirements may be very difficult to test. For example, as discussed previously, freedom from deadlock must be verified from the system structure. Since the potential for deadlock can be masked under light loads, and since testing for freedom from deadlock involves the enumeration of all execution paths, freedom from deadlock is not verifiable by performance testing alone.

5.6.9 Traceability

Like functional requirements, performance requirements must be traceable. Traceability addresses the following points:

- Why has this performance requirement been specified?
- To what business need does the performance requirement respond?
- To what engineering needs does the performance requirement respond?
- Does the performance requirement enable conformance to a government or industrial regulation?
- Is the requirement consistent with industrial norms? Is it derived from industrial norms?
- Who proposed the requirement?
- How were the quantities in this requirement derived? If this requirement is based on a mathematical derivation or model, the parameters should be listed and a reference to or a description of the model provided.
- If this requirement is based on the outputs of a load model, a reference and pointer to the load model and its inputs should be provided, together with the corresponding version number and date of issue.

Traceability is inherently beneficial to cost containment. If all of the preceding points can be satisfactorily addressed, the risk will be reduced that a performance requirement that is expensive to implement goes beyond the stated needs of the product. Traceability also reduces the

risk of inconsistency propagating through performance requirements throughout the lifecycle, because it provides a framework for methodically responding to queries about a requirement in case there is doubt about its validity. Inconsistencies in performance requirements can lead to incorrect implementations, quite apart from causing time to be spent trying to resolve them.

5.6.10 Granularity and Time Scale

Performance requirements should be specified at a level of granularity that is commensurate with the time scale in which the corresponding functionality will be invoked and, in the case of functionalities that involve human interaction, experienced. Choosing the right time scale for a performance requirement like average response time or throughput is necessary because averaging over a long time period obscures the longer delays that occur during peak periods. These delays are the ones that inherently affect the largest numbers of users and could occur when the system is most needed. For example, a senior accountant at a large company might specify that a timekeeping system should be able to handle 100,000 entries per month, while the employees are concerned about being able to enter their timekeeping logs as quickly as possible at the end of the working day. Clearly, the performance requirements of the system should be expressed in terms related to the user experience to make the best use of employees' time. Thus, the throughput might be expressed in terms of the number of timekeeping sessions and entries occurring between 17:00 and 18:00 each working day, which translates to about 4,500 sessions in the busy hour, since 100,000 records/22 working days per month works out to 4,545 timekeeping sessions in the busy hour each day. The performance requirement can be broken down further to describe the number of screens each employee sees and how long it should take to load each one, how long it should take to log into the timekeeping system, how long it should take to save the records once entered, and so on.

5.7 Summary

Performance requirements define the performance expectations of the system. Expectations about the volume and nature of the work to be handled arise from identification of the workloads and sometimes from

performance requirements about individual functionalities. The performance requirements must be linked to business and engineering needs, such as transaction volume and response and delivery times of various kinds. To be sound, performance requirements must satisfy criteria that are related to the criteria for sound functional requirements, such as unambiguousness, measurability, and traceability. We have also seen that performance requirements can be closely linked to commercial considerations, such as the volume of business conducted by a company or to the competitiveness of the software product itself, and that they must therefore be handled with the appropriate sensitivity. The use of performance requirements and timely testing also mitigates the performance risk inherent in the development of any software system.

5.8 Exercises

5.1. A university requires students to use a web-based tool to upload essays on deadline throughout the term. Each essay-based course has its own deadline for submission. In addition to enabling teachers and graders to mark essays and give them back to the students, the system will archive the essays so that other essays may be compared with them to detect plagiarism. To be usable, the system should not take long to upload essays, even as the deadline approaches. The system should be able to carry out plagiarism checks early enough for the teachers to have enough time to grade the essays within a week of receipt or, in the case of essays that are part of take-home exams, well before the end of finals week.

(a) Explain how you might formulate the performance requirements of this system (i) if it is meant to be used at a small private college with a maximum enrollment of 1,500 students, (ii) if it is meant to support a large university with an enrollment of 30,000 students, (iii) if it is meant to support the university system of an entire state, such as New York or California.

(b) Identify performance requirements for different parts of the work, such as uploading by the students, downloading by the teachers, uploading by the teachers after marking, and checking for plagiarism.

(c) How would the storage and performance requirements differ if the plagiarism check were based on the archives of the small university, the large university, and entire university systems as in question (a)?

(d) Can response time requirements for upload, download, and plagiarism checks be formulated independently of the scale of the university being served? Explain.

5.2. The essay processing system will also provide the ability for teachers to enter grades and comments. The students will be able to access their own grades. Each teacher will be able access the grades of all students at the university. It may not be possible to support response time requirements for the grade checking system during the times just before the deadlines for essays in large classes or at the end of term. How would you write performance requirements to account for this possibility? How would you write response time requirements that insist that the quality of service for those checking grades be maintained even during periods of heavy upload traffic, such as at the end of the term?

Chapter 6

Qualitative and Quantitative Types of Performance Requirements

System performance requirements often state that a given load shall be sustainable by the system, and that the system shall be scalable, without specifying the meaning of sustainability and without specifying the dimensions in which a system is to be scaled, and what successful scalability means in terms of system performance and/or in terms of the number of objects encompassed by the system. In this chapter we shall describe the expression of performance requirements in quantitative, measurable terms. We shall show how they can be used to reformulate qualitative requirements in terms that meet the criteria for sound performance requirements, such as being measurable, testable, and unambiguous.

6.1 Qualitative Attributes Related to System Performance

Performance requirements may contain a statement of the form "The system shall be scalable." All too often, there is no mention of the dimension with respect to which the system should be scaled, or the extent to which the system might be scaled in the future. Absent these criteria for scalability, testers will not know how to verify that the system is indeed scalable, and product managers and sales engineers will not be able to manage customer expectations about the ability of the system to be expanded. Characteristics for scalability, such as load scalability, space-time scalability, space scalability, and structural scalability, are described in [Bondi2000] and in Chapter 11 of this book. Examples of the corresponding dimensions include transaction rates, the ability to exploit parallelism, storage available to users and the operating environment, and constraints imposed by the size of the address space.

Stability is a quality attribute that is related to scalability. If the system runs smoothly when N objects are present but crashes when $N + 1$ objects are present, the scalability of the system is limited by the number of objects the system can support. Clearly, the number of objects the system can support is a dimension of scalability that is limited in this case.

Stability or a tendency to instability is also indicated by characteristics of the performance metrics. For example, during a prolonged period when the average offered transaction rate is constant, one expects (1) that the completion rate will be equal to the transaction rate, (2) that average resource utilizations will be approximately constant, (3) that average response times will be approximately constant, and (4) that memory occupancy will be approximately constant. These expectations are consequences of the basic modeling equations described in Chapter 3. Performance requirements for these characteristics of performance metrics should be specified. Failure to meet them in performance tests or in production should be cause for an investigation. Upward trends in any or all of these measures are an indication of saturation or of an oncoming crash. In particular, increasing memory occupancy is a sign of a memory leak. If the system leaks enough to exhaust a memory pool or object pool before being rebooted, it will simply stop running.

It should be noted that fluctuations in memory occupancy need not be indicative of a problem, although they could be. The memory

occupancy in programming languages like Java that support garbage collection will drop whenever a garbage collection episode occurs. The occupancy may then rise, perhaps irregularly, until garbage collection is triggered once again. Since there are processing costs and sometimes delays associated with garbage collection, the system should be tuned to ensure that the performance impact of garbage collection episodes is not so severe that performance requirements cannot be met while they are in progress.

6.2 The Concept of Sustainable Load

Performance requirements sometimes state that a particular load shall be sustainable, or that the system shall support a designated number of users. As we saw in Chapter 5, a requirement formulated in these terms is potentially ambiguous unless sustainability is defined, and unless the activities performed by the users and how often each one requests particular actions of the system are specified. To define sustainability, we need to describe the desired performance of the system in terms of metrics that fall within a set of acceptable bounds. Thus, we might say that a load is sustained by the system if the distribution of the transaction response times has certain characteristics, such as an average being below a certain level, if all the resource utilizations under a stipulated load are less than a certain percentage, and if the system continues to accept transactions at a certain rate without crashing and without having large amounts of variability in the performance measures under a constant transaction rate. Another criterion for sustainability would be that the average response time is insensitive to small increases in load when the system is running at the desired engineered peak load.

The foregoing discussion leads us to the following working definition of sustainability. We say that a specified transaction mix can be sustained at a given rate if all of the following hold:

- The requirements for the average response time and the probability that a response time exceeds a certain level are both met.
- A small increase in the transaction rate will cause only a small increase in the average response time.
- The average utilizations of all resources such as CPU, I/O, and bandwidth are below designated thresholds.
- There are no memory leaks.

In addition, a transaction-based system can support *N* users if all of the following hold:

- The combined load they generate is sustainable by all resources (CPU, I/O, network, etc.).
- Their concurrent footprints fit into memory without causing so much paging that the load would not be sustainable.
- There is enough room for their combined disk footprint.

Notice that these criteria include not only temporal ones relating to response time, throughput, and utilization but also spatial ones relating to memory space occupancy and disk space occupancy. The spatial criteria could have a temporal ingredient, since some of the space might be occupied only temporarily, and since the total average occupancy could include a component that lasts at most only as long as a transaction does. The average occupancy due to that component is the product of the holding time of that space by a transaction multiplied by the throughput of transactions needing their own space, by Little's Law as described in Chapter 3.

As we saw in Chapter 3, the average response time at a single-server device increases abruptly when its utilization exceeds 70%. This is an indication that the arrival rate causing 70% utilization is the maximum rate that can be sustained by the system. This working definition is broadly applicable to systems that operate under fairly constant loads most of the time. The reader should be aware that other definitions of sustainability may be required for systems on which demands appear in bursts, such as alarm systems.

To guide the reader in the correct formulation of requirements, we provide illustrations of ill-formulated requirements and show how they may be corrected to conform to the guidelines for writing sound criteria we discussed in Chapter 5. These examples are taken directly from performance requirements documents reviewed by the author.

6.3 Formulation of Response Time Requirements

Before describing a way to word a response time requirement that conforms to the guidelines for requirements listed in Chapter 5, let us first look at some examples of performance requirements that fail to conform to them in one or more respects:

1. Ideally, the response time shall be at least 1 second.
2. The response time shall be at most 2 seconds.
3. Both requirements must hold simultaneously.

The first requirement is faulty because it places a lower bound on the acceptable response time rather than an upper bound. In addition, it contains a qualifier, "ideally," which induces ambiguity because there is no accompanying statement of why the desired value range is ideal. Moreover, a subjective qualifier such as "ideally" has no place in the statement of an objective, measurable requirement. The second requirement is not achievable because the single occurrence of a response time in excess of 2 seconds would violate it. In practice, response times are probabilistically distributed in a way that may not be known a priori. Instead, the requirement statement should say something like this:

1. The average response time during the busy hour shall be at most 1 second.
2. 99% of the response times shall be less than 2 seconds in the busy hour.
3. Both requirements must hold simultaneously.

The first part of the requirement gives an upper bound on the desired average value. The second part of the requirement gives a measurable criterion for ensuring that few response times will exceed 2 seconds. This set of response time requirements is consistent. It contains no qualifiers. Notice that the second requirement does not have the same meaning as a statement that the response time shall be less than 2 seconds long 99% of the time. The use of the colloquial qualifier "of the time" induces ambiguity for the following reasons:

- We do not know the observation period implied by "of the time." An observation period of 24 hours or a year may include many intervals in which an "of the time" response time requirement is easily satisfied because the load is light. The average response time requirement might be harder to meet during an observation period of 5 minutes during the busy hour.
- The average response time is a sample statistic, not a time-averaged statistic. Therefore, one must count the number of response times that satisfy the requirement, not the fraction of time during which the requirement is met.

The statement of the requirement may be accompanied by supporting commentary that explains why this requirement was chosen, why the upper bound of the average has the value 1 second, the points in the transaction at which the response time measurement begins and ends, and how the response time is measured. Including these points in the supporting commentary supports verifiability, traceability, and measurability, some of the criteria for sound requirements mentioned in Chapter 5 and in [IEEE830].

6.4 Formulation of Throughput Requirements

A transaction throughput requirement must unambiguously state that the peak average transaction rate of the system shall be *X* units of work in a given unit of time. Any set of performance requirements relating to throughput must be internally consistent.

Consider the following pair of performance requirements relating to throughput:

1. The system shall sustain a throughput of 1 transaction per second.
2. The system shall support 100 users, each of whom generates 0.1 transactions per second.

In the absence of criteria for the ability to sustain a given throughput, the first requirement is ambiguous. The second requirement is inconsistent with the first, because it implies a system throughput of $0.1 \times 100 = 10$ transactions per second. It is also ambiguous in the absence of criteria describing what constitutes support of a given transaction rate, such as a desired average value for the response time and the system not crashing at this transaction rate. This pair of requirements must be redrafted to make them internally consistent. Their context must be specified so as to make them unambiguous. Context may be provided by a statement in the performance requirements document that defines the terms *sustainable* and *support*, as described in Section 5.6.

The throughput requirements of networks are usually expressed in terms of packets per second. Of course, the minimum, maximum, and average packet sizes should be specified, and names of the transport and application-layer protocols involved should be specified, so that the reader can determine whether the requirement is consistent

with the available bit rate of the transmission medium. This is essential for determining whether the performance requirement is achievable.

6.5 Derived and Implicit Performance Requirements

Consider a situation in which a system freezes or goes into deadlock on overload. If there is no explicitly stated requirement that the system be free from deadlock, stakeholders who do not wish to incur the cost of fixing the problem may resist doing so on the ground that there was no requirement that deadlock not occur, especially if the cost is potentially considerable. They will argue that fixing a problem that allegedly occurs only rarely entails not only the cost of the repair but the cost of testing to make sure that the repair has had the desired effect. A purist might argue that this is like saying that there is no requirement that the system shall function continuously, which is like saying that the system need not always work. The purist might argue that there is always an implicit requirement that the system be free from deadlock. In practice, acceptance of that implicitness depends on the organizational mindset. Therefore, it may be necessary to leave as little to chance as possible by enumerating requirements that appear to be basic, lest there be organizational resistance to fixing systems that fail to meet them.

A requirement may also be derived from other requirements as the result of an algebraic relationship or as the result of a design choice. This can arise for both functional and performance requirements. For example, a resource pool may be implemented with methods to allocate an object in the pool to a particular process or thread, and to deallocate the object once it is no longer required. Since the pool is shared, mutual exclusion is needed to ensure correct operation. At the same time, a requirement for the pool size must be derived to ensure that the probability of the pool being exhausted is very low. The sizing requirement is derived from whatever information is available about the number of times pool members are demanded and freed per second, and the number of seconds for which each pool member is held on average.

The foregoing are examples of implied and derived requirements. Inattention to these types of requirements when the requirements are drafted and when implementation takes place can eventually result in a system malfunction and user dissatisfaction or worse. In the remainder of this section we explore derived and implicit performance

requirements in greater detail with a view toward preventing the shortcoming we have just mentioned.

6.5.1 Derived Performance Requirements

While performance requirements about transaction rates, throughputs, and response times are often explicitly stated, consequent requirements on subsystems, including object pool size and memory usage, are not. If they are not explicitly stated, they must be derived from the quantities that are given to ensure system stability and to ensure that sufficient numbers of concurrent activities can be supported. Even if explicit values are stated, care must be taken to verify that the stated values for the object pool and memory sizes are consistent with, or at least not less than, the space/time occupancy implied by the throughput and response time requirements. The implied occupancy is a consequence of Little's Law and the Utilization Law described in Chapter 3. The object pool and the memory size must be sized to keep the probability of overflow below a certain low threshold. One way of ensuring this is to apply the Erlang loss formula [Kleinrock1975] to the sizing of the pool, as is also discussed in Chapter 3. In the author's experience, an astute developer and/or tester may ask the performance engineer to specify the maximum size of the object pool so that testing can be done to verify that the pool does indeed have the desired size.

As an example, suppose that a transaction will be dropped if an object pool is exhausted. The required transaction response time may be thought of as an average value for the holding time, while the transaction rate multiplied by the number of times an object will be acquired and released by the transaction may be thought of as an arrival rate or request rate. If we require that the probability of object pool exhaustion be 10^{-6} or less, we can approximately size the object pool to achieve this requirement using the Erlang loss formula. The calculated object pool size is the derived requirement needed to achieve the desired probability of pool exhaustion. While the loss probability requirement is inherently hard to test because losses should not occur, the ability to store the desired number of objects is easily tested in principle, provided that the test harness is capable of doing so.

6.5.2 Implicit Requirements

We think of a requirement as being implicit if it seems apparent from the context or if it is taken as given by all stakeholders. The problem

with implicit requirements is that they are part of what might be described as a domain-specific, technical culture. Because of globalization, outsourcing, and high labor turnover, one should not assume that what is implicit to one set of stakeholders is implicit to another. It is particularly easy to assume that all stakeholders take implicit requirements for granted when one is part of a large organization whose membership is stable. In the author's experience, it is better to err on the side of safety by spelling out the context of each requirement. In effect, no requirement should be regarded as implicit, even if it trivially seems so.

One might assume that freedom from deadlock is always an implicit requirement. It is implied by any performance requirement that specifies or implies a nonzero throughput, because a system in deadlock has zero throughput. Freedom from deadlock is also prerequisite for system stability. There may be resistance to the specification of a requirement that the system be free from deadlock if a legacy system is prone to it, or if a deadlock, such as that occurring in a database, can be resolved within a stipulated amount of time. Notice that we have not said that deadlock should be resolved within an acceptable amount of time, because acceptability is in the eyes of the beholder, depends on the nature of the problem, or both. For an example of ambiguity leading to a possibly incorrect implementation with catastrophic results, the reader is referred to [Swift1912] for a narrative on the consequences of an agreement that an egg should be cut open at the convenient end, rather than at the big end or the little end.

Implicit requirements often arise in transaction-oriented systems and in telephony. A requirement that a telephone switch or trunk be able to handle n calls per hour may be based on an implicit assumption that a "typical" call lasts 3 minutes and does not make use of any special services such as charging with a prepaid card, a telephone credit card, conference calling, a voicemail system, or an automated announcement system. The services involved in the call and the average duration should be specified. The mix of services changes as technology changes, while the average call duration may depend on whether the called party is an individual, an agent in a call center, or a conference bridge. The duration of a call may also be driven by cost, provided one can assume that the user base is sensitive to cost. One is more likely to converse for an hour if one's phone has unlimited long-distance service than if one is billed at US$3 per minute or more.

A requirement that a web site be able to support n concurrently logged-in users assumes many hidden, implicit characteristics, such as what the users are assumed to be doing, how long they will remain

logged in, the average footprint per user, and how often they will be triggering activities of various kinds. Similarly, a performance requirement that a web site or an application be able to handle *X* user sessions per month may be adequate for forecasting revenue, but it is uninformative about how the web site has to be engineered for acceptable performance. To remedy this, there should be an explicit statement about the amount of user activity during the peak hour, or about how the activity in the peak hour is related to the total monthly activity.

6.6 Performance Requirements Related to Transaction Failure Rates, Lost Calls, and Lost Packets

Costs are incurred when a telephone system drops or fails to complete calls, when customers abandon a transaction or call because the response time is too long, or when packets are lost in a computer network. Even if the pricing model is such that no revenue is lost when a transaction fails or a call is lost, there is an inherent cost associated with the loss due to the delays caused to other work by wasted processing as well as the cost of processing or transmission associated with repeating the lost work. Moreover, these costs can propagate to other layers of the system. A prerequisite for containing the associated cost is the specification of limits on the probabilities that these events will occur.

- If a telephone call is lost, the processing power and network bandwidth associated with setting up the call will have been wasted. If the call is retried, the cost of setting it up will essentially be twice what it would have been had it been completed properly on the first attempt.
- If a caller on hold at a call center abandons the call and hangs up, the time spent on hold may be charged to the call center if the call is toll free, or to the caller if the call is not toll free or was placed from a mobile phone. In addition, the caller may decide to call a competitor's call center instead, if there is one.
- If a packet being sent over a TCP connection is lost, network throughput will be degraded and bandwidth consumed by the retransmission of all packets sent between the most recently acknowledged one and the one declared to be lost. This is a waste of bandwidth.

In each of these cases, a performance requirement must be specified that limits the loss rate, expressed in terms relating to the domain:

- In a telephone network that carries upward of 100 million revenue-generating calls per day, a loss rate of 1% will still result in the loss of a million calls per day, which is unacceptable. Therefore, it may be necessary to specify a performance requirement that the probability of not completing a call be less than 10^{-8} or even 10^{-9}.
- In a revenue-generating call center, and/or one in which there are high telecommunications costs when customers are kept on hold, it is necessary to keep the call abandonment rate below a specified level, such as 10^{-6} or whatever level is justified by a trade-off between costs and revenue. This requirement may lead to requirements on the call handling time, the time to answer, and the number of agents available to answer calls, not to mention the number of telephone circuits that are needed to support the traffic [Bondi1997a].
- In a packet-switched network, packet loss may cause both performance and safety issues at the application level. To prevent this, a very low packet loss rate may be required for each network element, for example, 10^{-9}.

6.7 Performance Requirements Concerning Peak and Transient Loads

Our focus up to now has been on systems in which transactions occur with some regularity. This is not always the case with control or monitoring systems, such as sensor networks that are used to monitor the functioning of a manufacturing plant or a train, or those that are used to monitor a building for intrusions or the occurrence of a fire. As we have seen in earlier chapters, a fire alarm system must be able to trigger enunciation devices such as bells, sirens, and blinking lights within a few seconds of smoke being detected, as well as within a few seconds of someone pulling a red handle. Similarly, a conveyor system must be able to shut down abruptly if someone pulls an alarm cord or if an overheated motor is detected. Of course, the term *abruptly* should not be used in a formal requirement specification because it is not specific. Instead, the time from event notification to shutdown should be

specified. In the case of a fire alarm system, notifications of the presence of smoke or about sensor malfunctions may arrive in very short bursts, and the responses to them must happen within time intervals that are specified in local fire codes, such as [NFPA2007]. For these types of systems, a statement of the corresponding performance requirements might take the following form:

1. An audible alarm shall be sounded in the vicinity of detected smoke, and a telephone call shall automatically be made to the fire brigade, within 5 seconds of the arrival of the first notification of the occurrence of smoke by a detector.
2. Door closers shall be activated within 30 seconds of the arrival of the first notification of the occurrence of smoke by a detector, whether or not notifications have arrived from up to N notification devices.
3. All N notifications shall be cleared within 3 minutes.

Here, N depends on the size and nature of the installation. A smaller building may have a smaller value of N than a larger building, depending on the nature of the building, the number of sensors in the building, how the building is used (e.g., whether it houses offices, a library, a laboratory, a chemical plant, a hospital, a museum, or a concert hall), and what is stored there.

Transient bursts of activity can also happen in systems in which there are bulk arrivals. For example:

- The arrival of a train or plane at an international terminus may result in the discharge of a large number of passengers who must be cleared through border inspection in order to make connections within a stipulated time of arrival.
- When a session ends at a conference, all attendees must be served refreshments with adequate time to mingle before the next session starts.

6.8 Summary

In this chapter we have seen examples of different forms of requirements and shown how some misleading or ambiguous statements of requirements can be reworded to render them sound and unambiguous. We have also seen that performance requirements that seem

implicit from their context may very well be ambiguous unless their context is specified, and that performance requirements must be formulated in terms of the time scale within which they are to be engineered if they are to be useful. We have also seen that requirements regarding the exhaustion probabilities of object pools must be specified, and how the desired sizes of these object pools can be derived from the values of other metrics and various mathematical assumptions. Finally, we have illustrated the formulation of performance requirements relating to bursts of activity, especially those that occur in mission-critical systems such as alarm systems.

6.9 Exercises

6.1. A conference has five concurrent tutorials with 30 attendees each. During a 20-minute break, one or more waiters will pour tea into cups from pots at a buffet. Tea must be consumed before the attendees return to class. Write performance requirements for the tea service using the following steps:

- **(a)** Identify the constraining factors.
- **(b)** Identify the demand variables.
- **(c)** Identify criteria for satisfactory performance.
- **(d)** Identify metrics
  - **(i)** Describing the work done
  - **(ii)** Describing the performance of the system
- **(e)** Explain which objectives are served by
  - **(i)** Having all tutorials on break at the same time
  - **(ii)** Having tutorials on break at different times

Explain how your performance requirements are shaped by your choice of which of these objectives to fulfill.

6.2. At a German beer festival, waitresses in dirndls circulate between a beer dispensing point and tables of revelers. Each waitress serves patrons at a predetermined set of tables. Beer may be consumed only by people sitting at tables. Waitresses take batches of orders from one table at a time. Each table seats at most ten revelers. The waitresses may carry up to ten mugs of beer at a time, each of which has a capacity of 1 liter and weighs 1 kilogram when empty. The distance from a group of tables to the beer dispensing point is about 50 meters.

(a) Identify a set of performance requirements for this system from the point of view of a patron at a table.
(b) Identify a set of performance requirements for this system from the point of view of the waitresses' union.
(c) Explain how the waitresses' performance requirements must be reflected in a set of performance requirements written by the beer dispensers' union.
(d) Identify a set of performance requirements from the point of view of the brewery that owns the tent in which the festival is held.
(e) Determine whether the performance requirements of the different sets of stakeholders are consistent or in conflict, and suggest a resolution if they are in conflict. Explain the trade-offs.

Chapter 7

Eliciting, Writing, and Managing Performance Requirements

We explore the processes of eliciting, gathering, and documenting performance requirements. We also examine some pitfalls that may arise in documentation, such as the expression of requirements in forms that are antipatterns because they can lead to ambiguity and/or cause difficulty in measurement. We show how pitfalls can arise when prescribing the performance requirements of a system or component that is replacing a legacy system, because the functionality in the new system may be different but hidden, and describe how circular dependence should be avoided. This chapter also contains guidance on the organization of a performance requirements document and the contents of individual requirements.

7.1 Elicitation and Gathering of Performance Requirements

Gathering performance requirements involves the identification of functionalities, the identification of sound performance metrics, and the specification of the values these metrics should take to support the functionalities. As with functional requirements, the gathering of performance requirements entails interviewing stakeholders from many different teams from many potential stakeholder groups.

In the author's experience, the set of stakeholders can include product managers and sales engineers, because they identify the market segments for a system, such as customers for large-scale and small-scale systems. Any pertinent regulations that could affect performance requirements must also be identified, such as fire codes in the case of alarm systems. The set of stakeholders also includes architects, designers, developers, and testers. It is important to interview the architects and developers. This provides an early opportunity to avoid designing bottlenecks into the system and to verify that technologies they propose to use are capable of meeting the envisaged system demand, while allowing for the early introduction of innovations.

Stakeholders may be reluctant to commit to a particular set of estimates of demand for system usage because Customer A may argue that his or her organization's load is not like Customer B's. For example, the work mix of a small rural clinic may be very different from that of a large hospital using similar sorts of computer-controlled diagnostic equipment for different purposes. Even the workloads of hospitals with similar numbers of beds may differ, because one hospital might specialize in orthopedics while the other does only cancer care. Their fire alarm systems may be quite different, too, because of the nature of materials held in storage areas or in use among patients, such as oxygen tanks and isotopes. Despite the possible disparities among potential locations and user communities, performance requirements specification and testing should not be avoided. Instead, the project team should resort to the use of a set of reference scenarios describing a workload clearly and reflecting standardized needs for particular response time levels applicable to all systems. These would constitute baseline performance requirements. Reference mixes of activities driving throughput needs can be tailored to the needs of a given site for sizing purposes if desired. That way, development and testing can proceed even if a particular customer for the system has not been

identified before the product is announced. The reference scenarios might be agreed to by product managers and/or sales engineers, and then mapped to the corresponding activities in the computer system, with corresponding descriptions of workloads.

Typically, the performance engineer and/or the system architect must take the lead to elicit performance requirements related to business needs from the product managers and the sales engineers, if only because the latter two may not be comfortable with numerical reasoning and almost certainly will not be comfortable with organizing the performance requirements into a form in which they can be stored by a requirements management system. Moreover, the performance engineer will often have the responsibility for ensuring that the performance requirements meet the guidelines set out in [IEEE830] as well as the performance extensions to them described in Chapters 5 and 6.

One of the difficulties often encountered when eliciting performance requirements is that a product manager or other customer stakeholder may request a performance requirement that appears to be realistic on the surface but turns out to be quite absurd and even overly expensive when taken to its logical conclusion. For example, a business building a call center to serve a finite subscriber base for an ongoing service might insist on sizing the center to handle very large volumes of traffic. Analysis might show that the business's traffic assumptions underlying the sizing requirement imply that a very large share of the subscriber base would be phoning in with complaints every two or three days. Engineering customer care for a service of such poor quality would lead to unnecessary capital and labor costs if the service turned out to be better than that. If the product were really that bad, subscribers would be inclined to cancel the service, which would reduce the call center load even more.

Performance requirements are sometimes specified after the functional requirements have been identified. Some organizations demand that every functional requirement be accompanied by a corresponding performance requirement. This may prompt the requirements writer to think about the feasibility of a requirement, its potential impact on resource consumption, and the resulting impact on the performance associated with other functional requirements. The pitfall of such an approach is that the overall performance picture may be submerged in a detailed enumeration of performance requirements that makes business-related and engineering aspects of the requirements less visible. For example, a demand that an alarm condition be indicated within a certain amount of time may obscure the notion that a set of alarms should

be processed within some longer time period. Moreover, potential interactions between individual performance requirements may be missed unless the time is taken to review the performance requirements in their entirety for consistency. This is not to say that associating one or more performance requirements with every functional requirement is bad practice. Rather, it means that the set of associated performance requirements should not be regarded as telling the whole story about the performance and capacity needs of the system. The individual performance requirements for each functionality may need to be augmented by global performance requirements as well as a set of underlying traffic assumptions that describe the desired performance of the system as a whole.

The following is a list of steps to be taken when eliciting and gathering performance requirements. These steps should be taken whether or not performance requirements are associated with each individual functional requirement.

1. Identify the activities to be performed by the system.
2. Identify any regulations and standards that might be drivers of performance requirements.
3. Identify any business and engineering needs that might be drivers of performance requirements.
4. Determine which of these activities are performed serially and which in parallel.
5. Determine which of these activities occur with repetitive regularity under normal conditions and which occur in bursts.
6. Identify which activities occur in the background, such as database cleanup, and which result in bulk arrivals of work, such as alarms in emergency situations.
7. Identify the salient domain-specific performance measures and map them to common performance measures, such as the throughputs offered to specific system components, as needed for performance modeling and requirements verification purposes.
8. Identify the means and tools by which each performance metric and each quantity specified in a performance requirement can be measured. This is essential to demonstrating the measurability of all performance quantities and the testability of all performance requirements.
9. Determine the desired response time associated with each activity, and how often each activity is likely to occur.

Identify activities that occur periodically and those that occur in bursts, along with the corresponding periods and burst sizes.

10. Working with product managers and other subject matter experts, identify traffic assumptions and reference scenarios related to the sets and volumes of actions to be performed concurrently at different times of the day, seasonally, and under different types of conditions, including normal conditions and emergency conditions.
11. Determine whether there are timing requirements for system reboot and system shutdown, and identify the factors affecting these times.
12. Document the performance requirements while gathering them, and review them for consistency, feasibility, and testability. Ensure that the requirements meet the guidelines set forth in [IEEE830].
13. Identify reference traffic scenarios for performance assessment.
14. Document the performance requirements continuously, and record any derivations of quantities and sources of information there as well.

The progression through these steps might be led by a performance engineer, a requirements engineer, or an architect who would be designated as the owner of the performance requirements and be responsible for managing any necessary iterations through the steps. As with functional requirements, the preparation of the performance requirements should involve functional testers, performance testers, developers, product management, and an architect to reduce the risk of overlooking potentially valuable insights.

A detailed discussion about the elicitation and gathering of requirements is beyond the scope of this book. For this, the reader is referred to [BPKR2009]. Here, our focus is performance requirements specifically.

7.2 Ensuring That Performance Requirements Are Enforceable

Recall that performance requirements are an enumeration of the expectations about such performance metrics as system throughput, response time, and the ability to have a stated number of entities under the

system's management. For performance requirements to be enforceable, they must satisfy the criteria for clarity described in Chapter 5, such as unambiguousness, correctness, and completeness, while also being consistent with one another and being verifiable. Moreover, the quantities in the performance requirements must be obtainable from measurements, and it must be possible to conduct performance tests whose results will show whether the performance requirements are met. Under no circumstances should the set of performance requirements be reduced to a single number. As we saw in Chapter 2, the chapter on performance metrics, doing so will mask potential complexities and obscure any possibilities for trade-offs. This phenomenon was described as mononumerosis. Since the value cited in a case of mononumerosis might not be measurable, there is a risk that a performance requirement cast in terms of a single variable may not be enforceable.

7.3 Common Patterns and Antipatterns for Performance Requirements

While performance requirements for specialized embedded systems may take unusual, domain-specific forms, those for transaction-oriented systems tend to fall into patterns for average response time, throughput, and number of supported users. We have already seen some examples of these in the foregoing. Smith and Williams have used the term *performance antipattern* to describe an aspect of system structure or algorithmic behavior that leads to poor performance [SmithWilliams2001]. We shall use the term *performance requirements antipattern* to denote a form of performance requirement that is ambiguous at best and misleading at worst. Antipatterns are to be avoided, even if they express sentiments that are laudable. We illustrate patterns and antipatterns with examples encountered by the author. Some these have already been encountered in the discussion of performance metrics in Chapter 2.

7.3.1 Response Time Pattern and Antipattern

In Chapter 6 we saw ill-formulated performance requirements like these:

1. Ideally, the response time shall be at most 1 second.
2. The response time shall be at most 2 seconds.

This pair of requirements is problematic. The term *ideally* expresses a sentiment about response times that is inconsistent with the second requirement and may not be attainable. The occurrence of a single response time in excess of 2 seconds would mean that the second requirement had not been met. Nothing is stated about when or how often the response time requirement must be met. If a sentiment like that in the first part of the requirement must be documented, it is best to place it in a section on supporting commentary rather than in the body of the requirement itself.

The reformulation of the requirement we proposed in Chapter 6 is an example of the pattern that the response time requirement for a transaction-oriented system should assume. It has the advantage of being expressed in measurable terms and of being testable.

1. The *average* response time during the busy hour shall be at most 1 second.
2. Ninety-nine percent of all response times shall be less than 2 seconds during the busy hour.
3. Both requirements shall be met simultaneously.

The wording in parts 1 and 2 of this requirement reflects the fact that the average response time is a sample statistic rather than a time-averaged statistic. The two parts of the requirement are consistent with one another and can be attained simultaneously, since the average is much lower than the desired maximum.

7.3.2 "… All the Time/… of the Time" Antipattern

Were a requirement to say that the response time should be less than 2 seconds 99% *of the time*, we would have to clarify the requirement by asking whether the requirement for the average response time would be met for $0.99 \times 3{,}600 = 3{,}564$ seconds in every hour, or during some fraction of the year, or some other time interval. The problem may be illustrated by a quote from former US President George W. Bush: "I talk to General Petraeus all the time. I say 'all the time'—weekly; that's all the time—…" [Bush2007]. The quantification is ambiguous because the time scale and frequency of interaction are unspecified, and because one cannot tell from colloquial use whether the "… all the time" or "… of the time" refers to a sample statistic such as average response time, to a time-averaged statistic such as utilization or queue length, or to a frequency of occurrence, such as the number of events per second or

even the number of communications between a president and a general per month or per week.

7.3.3 Resource Utilization Antipattern

A requirement that states that the CPU utilization shall be 60% is erroneous because the resource utilization depends on the hardware and on the volume of activity. The desired response time and throughput requirements might well be met at higher utilizations. Furthermore, the requirement would fail to be met under light loads, which is absurd.

When confronted with a requirement like this, the performance engineer could ask whether the stakeholder who originated the requirement is concerned about overload, and then offer to reformulate the requirement as an upper bound on processor utilization. Doing so helps to ensure that the system will be able to gracefully deal with transients that could cause the utilization to briefly exceed the stated level under normal conditions.

It is entirely appropriate to state a resource utilization requirement of the form "The average utilization of resource *X* shall be less than *Y*% in the peak hour." For single-server resources, *Y* might be set to 70%. For a pair of parallel servers in which one acts as a backup for the other, it is appropriate to state that the utilizations of individual processors must not exceed 40%, so that the maximum load on one of them after a failover would be no more than 80%. Anything higher than that could result in system saturation.

7.3.4 Number of Users to Be Supported Pattern/Antipattern

We have already mentioned performance requirements of the following form:

- *The system shall support* N *users*.

There are several problems with this requirement that make it ambiguous and incomplete:

- There is no statement about what the users do, or how often they do it.
- There is no statement about how many users are logged in at the same time.
- There is no distinction between types of users.

These difficulties can be mitigated by placing a description of what the users do and their behavior in a section about traffic and other assumptions, thus establishing a context for the requirement.

7.3.5 Pool Size Requirement Pattern

In a service-oriented architecture, the implementer of a service must ensure that it can cater for the possibility that it must provide a large number of objects. Failure to provide enough objects in the pool could cause the loss and/or delay of transactions and other system malfunctions. These objects could be memory partitions, semaphores, locks, threads, JDBC connectors, or instances of other kinds of type abstractions. A pool size requirement is often overlooked during the preparation of performance requirements and functional requirements.

A performance model can be used to determine the required size of an object pool. As we saw in Chapter 3, where basic performance models were discussed, requirements for object pool sizes are driven by average and peak transaction rates and by the holding times of resources. These in turn may be driven by the numbers of users logged in or by other factors, depending on the application and the implementation. The required size of the object pool may also be driven by the maximum probability of pool exhaustion that we can tolerate, that is, the maximum probability that an object is not available when it is needed. This probability, ε, will be very small, for example, in the range $10^{-12} \leq \varepsilon \leq 10^{-6}$. The value of the probability should be determined while considering the effect of not having an object available when it is needed. The choice of ε must be listed with the requirement, together with the mathematics used to do the sizing. Both of these will be listed under the "Derivation of quantities" section of the performance requirement, as discussed in the section on the structure of a performance requirement later in this chapter.

7.3.6 Scalability Antipattern

Consider a performance requirement of the form

- *The system shall be scalable.*

This statement indicates an intent to have the system support larger or smaller numbers of objects while maintaining similar response time objectives, but it does not tell us anything about the dimensions in which the system should be scaled, such as the number of logged-in

users, the number of account holders, the number of devices under management, or the throughput of units of work that the system might eventually be required to host or support. Nor does it tell us about the extent of scalability or the direction (up or down). Such a requirement is clearly ambiguous and incomplete. A less ambiguous scalability requirement might be written as follows:

- *A small system shall host 100 users, a large one 1,000 users.*

This requirement informs us about the dimension in which the system must be scaled, namely, the number of users, and what numbers of users are to be hosted in large-scale and small-scale systems. It is complete and unambiguous only to the extent that the performance requirements and demands of a user are specified. In other words, the clarity of the reformulated requirement depends on the context within which it is given.

7.4 The Need for Mathematically Consistent Requirements: Ensuring That Requirements Conform to Basic Performance Laws

There are multiple aspects to the mathematical consistency of performance requirements.

First, performance requirements must be mathematically consistent with one another. To verify consistency, one must ensure that no inference can be drawn from any requirement that would conflict with any other requirements or inferences that could be drawn from them. Inferences could be drawn through the use of models. They could also be drawn by deriving an implied requirement from a stated one. If the implied requirement is inconsistent with other requirements, so is the source requirement.

Second, each performance requirement must be consistent with stated performance assumptions, for example, assumptions about the traffic conditions and engineering constraints. For example, a message round-trip time in a protocol with acknowledgments should be less than the timeout interval.

Finally, the performance requirement must not specify combinations of loads and anticipated service times that make it unachievable. This will happen if the product of the offered traffic rate and the anticipated average service time of any device is greater than the number of devices acting in parallel. For instance, the product of the average processing time and the system throughput at a uniprocessor must be less than 100% so that the CPU is not saturated.

To reduce the risk of there being inconsistent performance requirements, the performance requirements document should be reviewed by at least one individual with both quantitative expertise and knowledge about the domain of application of the system before the document is approved for release.

7.5 Expressing Performance Requirements in Terms of Parameters with Unknown Values

Even if numerical values associated with performance requirements are unknown, it may be possible to identify algebraic relationships between them. These relationships should be spelled out in any case to ensure consistency among the requirements. Where a performance measure is an independent variable whose value is unknown, any performance requirements involving it and any quantities depending upon it should be identified. The corresponding performance requirements should list the variables in question as parameters to be filled in once their values have been identified. Notice that there might be more than one valid set of parameters, and that different sets of performance requirements would have to be generated for each one. For example, the desired workload throughputs and numbers of customers to be supported in small, medium, and large configurations might not be fully known when requirements elicitation begins. The preparation of performance requirements should proceed even when not all the values of parameters are known, so that dependencies between performance requirements or links to functional requirements may be identified. The parameterized requirements should initially be listed as placeholders to be completed as the document evolves or to be flagged as unknowns during requirements review. There should be no unknown values in the performance requirements document once product development begins, because these could affect the scale of the hardware platform and lead to the overengineering or underengineering of the system.

7.6 Avoidance of Circular Dependencies

A circular dependence can arise between two or more performance requirements if the values of the parameters in each depend on the values of parameters in the other, either directly or indirectly. Such circular dependencies are to be avoided because they make

performance testing and modification of the requirements difficult. They might also be a sign of self-expansion, a characteristic of system performance that undermines a system's scalability and potentially its stability under heavy loads [Bondi2000]. Self-expansion of memory requirements and response time can occur when the release of a resource bound to a process or thread is delayed by the presence of that process or thread in a queue ahead of the thread or process that must deliver the release signal. The increased holding time of the resource by the first process delays the acquisition of other like resources in the same pool by other resources, which in turn delays release. This problem will be discussed further in Chapter 11 on scalability.

7.7 External Performance Requirements and Their Implications for the Performance Requirements of Subsystems

The demands on a complex system may be regarded as being triggered by external stimuli. These stimuli may consist of arriving customers, the arrival of transactions, or other events. The rates at which these stimuli occur constitute the system throughput requirements. As the architecture of the system takes shape, the information flow through the system and the frequencies with which various software and hardware components are visited will become apparent. These frequencies are the throughputs offered to the corresponding components. The use of these components may involve the acquisition, retention, and freeing of objects within them. The number of such objects needed to assure performance should be derived from the throughput requirements, estimated service times, and predicted resource holding times, using methods such as those discussed in Section 7.3.5 on object pool sizes.

7.8 Structuring Performance Requirements Documents

We recommend a structure for a performance requirements document that is quite similar to that recommended in [IEEE830] for functional requirements. A section on traffic assumptions specific to the domain should be included in the document to reduce the risk of ambiguity or

misunderstanding. This is especially important when there is a risk of high staff turnover.

Reference work items and reference workloads are needed to establish the context for domain-specific metrics. A reference work item may be a particular kind of transaction or set of transactions and activities. A reference workload specifies the mix and volumes of the transactions and activities. A reference scenario might be a set of workloads, or a set of actions to be carried out upon the occurrence of a specific type of event. For instance, a reference scenario for a fire alarm system might be the occurrence of a fire that triggers summoning the fire brigade, the sounding of alarms, and the automated closure of a defined set of ventilators and doors. The performance metrics used in the requirements, especially those that are specific to the domain, should be defined and mapped to related system actions. The instrumentation used to gather the metrics should also be specified to the extent known, so that one can establish that a mechanism for verifying and enforcing the requirements via testing exists.

A performance requirements document must be structured to facilitate the reader's understanding of quantitative specifications of the performance requirements, as well as the requirements' context. The scope and purpose of the document should be clearly stated. Among the questions answered by the scope and purpose section would be the following:

1. How is the document going to be used? Will it be used as the basis for performance tests? Will it be used for marketing purposes?
2. What feature set is covered by the performance requirements document?
3. What aspects or features of the system are *not* covered by the document?
4. With what systems will the system or component to which the requirements pertain communicate or interact?

The last point arises because the interactions with other systems or components may be stimuli from them or responses to them. These trigger or are the results of work that must meet performance requirements. Moreover, the performance requirements may be influenced by the interoperability requirements of the interacting systems.

Next, the intended audience should be identified. This includes the set of stakeholders who must understand the document and

implement and test the requirements. It also includes stakeholders who need to understand the economic aspects of the system, such as the costs and revenues associated with achieving performance objectives. The intended audience might include architects, product managers, designers, developers, testers, and the product owner. Mentioning stakeholders implies that the requirements and/or their supporting commentary are understandable by at least one of them, and preferably by as many of them as possible.

A set of references should include the functional requirements documents to which this performance requirements document relates, any technical standards and draft standards that should be cited, any relevant government regulations and statutes, and any product management documents and contractual documents that will influence the content of the requirements. The reference list should also include any books, published articles, patents, and published patent applications. For convenience, one may wish to list proprietary and/or classified references in one subsection and published materials in another subsection.

Next, there should be a statement of the basic assumptions that underlie everything else that is stated in the document. These assumptions may be explicit or implicit. Since implicitness is in the eyes of the beholder, one should state both implicit and explicit assumptions. For example, there are implied assumptions among telecommunications engineers of a certain era that the average phone call lasts 3 minutes and that 10% of the traffic in a 24-hour day occurs during the peak hour. These assumptions depend on social behavior and on how telephone service is used. These two assumptions may have been valid when the price of telephone calls was high and before conference calls became ubiquitous. They may not be valid today. If these assumptions are used, they should be stated explicitly, so that they are apparent to all readers of the document present and future, and so that they can be questioned if measurements warrant it. The statement of assumptions should also describe any industry norms and standards relating to the functionality of the system. These should also be listed in the reference list. Assumptions about the intended traffic load will be stated here, together with criteria for load sustainability, definitions of the performance metrics to be used in the performance requirements, and assumptions about the instrumentation to be used to collect the measurements needed to verify and validate the performance of the system. The enumeration of the performance requirements follows the statement of assumptions.

Table 7.1 *Structure of a Performance Requirements Document*

1. Scope and purpose
2. Intended audience
3. References (including related functional requirements specification document)
4. Statement of assumptions:
a. Assumptions about the system
b. Assumptions and conventions about the problem domain
c. Traffic assumptions
d. Criteria for load sustainability
e. Definitions of metrics used for the requirements
f. Instrumentation to gather the metrics for verification
5. Performance requirements

A summary of the proposed document structure is shown in Table 7.1. The layout of an individual performance requirement is discussed in the next section.

7.9 Layout of a Performance Requirement

In addition to being recorded in a document in a format like the one in Table 7.1, performance requirements should be stored in a machine-readable database to enable classification, search, and retrieval. For ease of maintenance and to ensure consistency, the performance requirements document should be generated from the database in which the requirements are stored, as can be done for functional requirements. The requirements may be stored in a spreadsheet, or in special-purpose tools such as Rational Rose or Doors. We shall think of a performance requirements document as a set of records, each of which has a set of fields, including an index number for the requirement and a descriptive title.

The suggested fields of a performance requirements record reflect many of the concerns we have discussed in the earlier chapters on requirements. Some fields, like the list of precedents, sources, and standards, are intended to provide traceability. Separating supporting commentary from the statement of the requirement reduces the risk of ambiguity while providing an opportunity to document some of the

reasoning behind the requirement and the requirement's purpose. Listing dependent and precedent performance requirements helps one to see how requirements are intertwined.

Storing the statement of the requirement and the supporting commentary in separate fields facilitates the distinction between a requirement and the reasons for specifying it. The statement of the requirement will list the values or range of values, or a statistical description of the values, that a set of performance metrics should take. The supporting commentary will describe why this requirement was specified, citing standards, marketing documents, mathematical derivations, the related clauses in contracts, and/or engineering conventions as necessary. These documents should all be listed in the reference section of the performance requirements document.

The requirements record should contain a field listing the precedents, sources, standards documents, and other material supporting the requirement. This is essential for ensuring that the requirement is traceable.

Any quantities listed in the performance requirement should be justified either by reference to standards or by reference to calculations. The calculations may be listed in a suitably indexed appendix, or they may appear in public or proprietary documents. The documents should be cited and the citations included in the list of references.

Each requirement should be accompanied by a list of the requirements that depend upon it (the dependent requirements).

Each requirement should be accompanied by a list of the assumptions on which it depends and by a list of the other requirements on which it depends. The assumptions should be listed in the "Statement of assumptions" section of the performance requirements document.

If a performance requirement contains a numerical value that must be measured, the source and means of measurement should be specified.

There should be flags to show the following:

- An indicator if the requirement is independently modifiable
- An indicator that the requirement is traceable
- An indicator that the requirement is unambiguous, or if not, why not
- An indicator that the requirement is correct, or if not, why not
- An indicator that the requirement is complete, or if not, why not
- An indicator that the requirement has passed or failed review, or if not, why not

Table 7.2 *Suggested Fields of a Performance Requirements Record*

1.	Requirement number
2.	Title
3.	Statement of requirement
4.	Supporting commentary
5.	List of precedents, sources, standards
6.	Derivation of quantities
7.	List of dependent requirements
8.	List of assumptions and precedent performance requirements
9.	Sources of measurement data
10.	Name of a subject matter expert on this requirement
11.	Indicator if the requirement is independently modifiable, or if not, why not
12.	Indicator that the requirement is traceable
13.	Indicator that the requirement is unambiguous, or if not, why not
14.	Indicator that the requirement is correct, or if not, why not
15.	Indicator that the requirement is complete, or if not, why not
16.	Indicator that the requirement has passed or failed review, and why

These indicators will be used by the performance requirements engineer and by reviewers to track whether a performance requirement meets or is believed to satisfy each criterion. A summary of the suggested fields of a performance requirements record is shown in Table 7.2.

7.10 Managing Performance Requirements: Responsibilities of the Performance Requirements Owner

Performance requirements play a role in every stage of the software lifecycle, whether the lifecycle is managed using a waterfall process, an agile process, or otherwise. To facilitate access by the stakeholders, performance requirements should be centrally stored, perhaps in the same system that is used to store functional requirements. To ensure that performance considerations do not fall through the cracks, it is essential that an individual be designated as their owner, and that this individual

be visibly mandated and empowered to communicate with all project stakeholders about performance issues. Examples of possible owners include the chief architect of the project, the chief requirements engineer or a designate, or the chief performance engineer or a designate.

As functional requirements evolve or as new ones emerge, the performance requirements must be updated to ensure consistency with the new functionality. Similarly, as product managers and other stakeholders identify new performance requirements or changes to those already written, care must be taken to ensure that the new and modified requirements are consistent with the ones that have not been explicitly changed. Otherwise, ambiguities will arise that will complicate performance testing while potentially creating confusion about performance expectations and inconsistencies within the document.

In addition to addressing performance concerns, the performance requirements owner will be responsible for managing change control and requirements traceability, and for ensuring that every change or addition is linked to business and engineering needs. When performance requirements are negotiated and written or modified, the performance requirements owner will also play a pivotal role in mediating between different groups of stakeholders, including architects, designers, the owners of various system components, and perhaps even lawyers. Involvement with the latter is necessary to ensure that contracted levels of performance are described in measurable terms. If performance requirements are changed, the performance requirements owner must ensure that the changes are understood by all of these stakeholders, as well as by sales engineers, so that the necessary changes to architecture, implementation, and appropriate commitments to customers can be made.

7.11 Performance Requirements Pitfall: Transition from a Legacy System to a New System

When transitioning from a legacy system to a new one, it is easy to overlook subtle changes in functionality that might affect the way performance requirements should be formulated. The author encountered this pitfall when transitioning from his father's 1940s vintage 35mm coupled rangefinder camera to a modern point-and-shoot digital camera. With the old camera, pressing the shutter button causes the subject

to be captured pretty much in the state seen by the user. In this case, the subject was a walking cow with a bell hanging from its collar. The digital camera took so much time to capture the image that the resulting photo included the cow's udder, but not the bell. The difficulty was that the shutter reaction time on the digital camera included autofocus and exposure settings. With the vintage camera, these would have been done manually in advance of the shutter being released. The problem occurred because the author erroneously assumed that the digital camera would have the same shutter reaction time as the vintage camera. It does not, and the unexpected image was the result.

One might ask whether the comparison of the shutter reaction times is fair, given that the digital camera does so much more when the button is pressed. The answer is that a comparison should reflect expectations of the functionality that will be implemented, and that the user should plan the shot accordingly. With the vintage camera, planning the shot would have included several preparatory steps:

1. Opening the light meter, aligning the settings pointer on the light meter with the needle, noting the required combination of aperture setting and shutter speed, and setting these on the camera.
2. Composing the picture in the camera's viewfinder, and setting the camera's focus using the focus ring on the lens.
3. Pressing the shutter release button. The image is captured in the time it takes to open and close the shutter. This is known as the shutter speed.

The combined time to perform all of these actions could be long enough for the cow to walk out of view altogether. By contrast, pressing the shutter release button on the digital camera causes all three steps to be performed. The instant at which the image is captured may be later than the instant at which the shutter button is pressed, but the subject may still be close to the desired position by the time the image is captured.

Only one functionality is triggered by pressing the shutter button on the legacy camera: opening and closing the shutter to capture the image. On many digital cameras, multiple actions occur when the shutter button is pressed. The lesson we draw from this comparison is that one must evaluate the set of functionalities to be performed by both the legacy and the replacement system components when determining the performance requirements of the replacements. We must

also take into account any changes to the interface that are required when integrating the replacement into the system, including timing characteristics.

7.12 Formulating Performance Requirements to Facilitate Performance Testing

If performance requirements are ambiguous or not expressed in measurable terms, they cannot be tested. Adherence to the IEEE 830 guidelines for requirements should enable the implementation of performance tests to ensure that the system is running smoothly. Even if the performance requirements are not clear, performance testing should still be done because it facilitates the unmasking of functional problems that are the result of concurrent programming errors, memory leaks, and poor programming choices that slow a system down to the point of its being perceived as nonfunctional or dysfunctional. If carefully structured, performance tests can also reveal the performance limitations of the implementation.

Performance testing can reveal whether the performance metrics of a system behave as predicted by performance models. Failure to do so can be a sign of a system malfunction. Performance requirements should be written to ensure that performance tests are structured to reveal deviation from the properties predicted by performance models as well as to verify adherence to performance requirements.

In a well-behaved system subject to external arrivals of work at constant rates, the utilizations of such resources as the processors, disks, and network bandwidth should be linear functions of the offered load. The largest of those utilizations will be first to top out at 100% as the transaction loading increases. This is a consequence of the Utilization Law and the Forced Flow Law we saw in Chapter 3. Moreover, if the average service times and average arrival rates at the system are constant and the mix of work is constant with respect to time, the average utilizations and average response times should also be constant. Deviation from this behavior is a sign of a problem. Therefore, there should be performance requirements that state that:

1. The average resource utilizations measured at regular intervals should be constant or vary within a very small range (to be specified) during periods when the arrival rate is constant

yet low enough to avoid saturating any resource in the system. During such periods, the resource utilizations shall be free from trends of any kind.

2. The average resource utilizations for systems subject to constant loads over sufficiently long periods of time (to be specified) shall be linear functions of the offered load of a specified transaction mix.
3. The average memory occupancy for systems subject to constant loads over sufficiently long periods of time (to be specified) shall be constant and free of trends under the same conditions as the CPU and other resource utilizations would be.
4. The average response times measured during regular intervals should be constant or vary within a very small range (to be specified) during periods when the arrival rate is constant. During such periods, the average response time shall be free from trends of any kind.
5. If the arrival rate oscillates, oscillations of the resource utilizations and average response times should occur with peaks and troughs being the same distance apart as those of the arrival process, and at the same time. If the average arrival rate describes a square wave over time, the resource utilizations and average response times should also describe square waves.

Performance requirements 1 through 5 are applicable to systems whose workloads do not inherently saturate them and when the systems are in equilibrium. In the theory of Markov chains, a system is said to be in equilibrium if all states are recurrent, if each state is reachable from every other state (i.e., the chain is irreducible), if the long-term probability of being in a given state tends to a constant as time tends to infinity, and if the chain is aperiodic [BhatMiller2002]. We use the term *equilibrium* somewhat more loosely here. Requirements 1 through 5 are in keeping with the Markov chain definition. Failure to meet these requirements should trigger an investigation. In particular, since increasing memory occupancy is a sign of a memory leak, a requirement that there should be no leaks implies the third requirement listed. On the other hand, if the system has at least one component that is implemented in a language supporting garbage collection, such as Java, oscillation of memory occupancy within a fixed range may indicate that garbage collection is taking place as desired.

As we shall see in Chapter 9 on performance testing, the lengths of the sampling intervals and durations of the performance tests needed to verify that these requirements are being met depend on the anticipated lengths of the response times and the desired system throughputs. If offered load is one transaction per hour, the average performance measures will be very different from what would be observed if the transaction rate were one per second (3,600 per hour) or even one every 5 seconds (720 per hour). To verify that performance requirements 1 through 5 hold, we must subject our test system to increasing traffic loads and maintain each traffic load long enough to ensure that equilibrium is reached.

Performance requirements 1 through 5 are not applicable to systems that are subject to bursty traffic, such as alarm systems operating in an emergency situation, because they are not operating in a steady-state or equilibrium environment. Instead, performance requirements for such situations should be expressed in terms of the amount of time within which a fixed number of actions must be executed and, in the case of fire alarm systems, the maximum amount of time that may elapse between the first arrival of a signal from a smoke detector or the activation of a hand pull and notifications such as the sounding of the first alarm and the automatic telephoning of the fire brigade.

7.13 Storage and Reporting of Performance Requirements

Performance requirements can be quite complex and may change over time. At the same time, it is important that they always be accessible to the right stakeholders so that inquiries about them can be promptly dealt with. This is especially important because the perceived value of a system to its purchaser or owner hinges in large part on the system's capacity, responsiveness, and ability to recover gracefully from stressful loads. It is therefore important that they be well organized, easily retrievable, and easily tracked. At the very least, performance requirements should be tabulated in a spreadsheet or a text document. These two methods can become cumbersome as the number of performance requirements expands and as the links between performance requirements and functional requirements become increasingly numerous.

In many large projects, functional requirements are stored in an off-the-shelf requirements management system from which a requirements

document structured along the lines of Table 7.1 can be automatically generated. Storing the performance requirements in the same system may facilitate providing software links between functional requirements and performance requirements, so that the impacts of each upon the other can be readily assessed. For review purposes, it is useful to be able to configure the requirements management system to automatically generate requirements reports with structures like those in Table 7.1. There, the individual requirements can be laid out as in Table 7.2. Using the requirements management system allows updates to the requirements to be propagated to the document without major manual effort and its associated risk of errors.

7.14 Summary

In this chapter we have given an overview of how performance requirements documents might be prepared and structured, and about some of the pitfalls that can arise when writing them. In this and the two preceding chapters we have repeatedly underscored the need to avoid ambiguity in the formulation of performance requirements and the need to provide a clear context for them. We also insist that quantitative requirements not be expressed using colloquial phrases such as "all the time" or "of the time," to reduce the risk of confusion about what is meant and to provide a clear inference about how the quantities of interest are to be measured. This is a prerequisite for the formulation of meaningful and informative performance tests to verify that the performance requirements can indeed be met.

This page intentionally left blank

Chapter 8

System Measurement Techniques and Instrumentation

We describe the motivation for system measurement and explore tools and techniques for doing so. Measurement pitfalls and instrumentation flaws will be used to illustrate the need for the validation of measurement tools. We will also examine the applicability and limitations of profiling tools and measurements embedded in the applications.

8.1 General

In earlier chapters we underscored the need to ensure that performance requirements be expressed in measurable terms so that they can be verified. In this chapter we look at reasons for gathering measurements, including verifying performance requirements. We will examine some of the tools with which measurements can be gathered. Discussion of the procedures for planning measurement exercises is deferred to Chapter 9, where performance testing will be discussed.

Just as government statistics offices collect data to track social and economic trends, performance engineers and system managers need to measure system resource usage and system performance to track the

evolution of the load. This is done to ensure that the system is not overloaded, to verify the effects of system changes, and to ensure that the performance of the system is meeting requirements, engineering needs, and customer needs. Performance measurements can also be used to anticipate the onset of system malfunctions.

Measurement is necessary to identify relationships between resource usage measures, offered traffic, processed and lost traffic, and response times. If the response time of a system is too long, one's first instinct should be to measure its resource usage to identify the cause and fix the problem. This is true even of self-contained systems such as laptops. However, there are many other reasons for gathering performance measurements:

- A production system should be measured continuously so that baseline patterns for system resource usage can be established for different times of day and for different times of the year. Continuous measurement and presentation of the measurements by time of day can also reveal anomalies and trends in resource usage. Moreover, continuous measurement of a system in production is necessary to identify the time of day during which the offered load is greatest.
- A production system should be measured before and after any configuration change, so that the impact of the change on resource usage can be determined.
- System measurement is necessary for fault detection, the triggering of alerts that undesirable events are about to take place or are in progress, and the application of the correct measures to deal with them. A system or network in production should be monitored continuously so that a quick decision can be made to intervene with a remedy if unexpected changes in performance and/or resource usage occur. For example, sudden or otherwise unanticipated increases in measured response time could be used to trigger software rejuvenation and avert a system crash or detect the presence of intruders [AvBonWey2005, AvColeWey2007].
- The performance of a system under development should be measured whenever the development of new features has been completed so that their impact on system performance can be evaluated.
- Similarly, the performance characteristics of a subsystem or platform, whether bought off the shelf or purpose built, should be measured to ensure that it is suitable for the intended application.

- System measurement is essential for capacity planning. Resource exhaustion can be avoided and the timely introduction of new resources can be accomplished and managed in a cost-effective manner only if sufficient data is present to make informed decisions about resource additions and configuration changes.
- The performance of a new or changed system undergoing a performance test is measured to verify that performance requirements are being met, and to verify that the system is behaving in a stable manner.

It is essential to collect measurements over a period that is long enough for the average values of system performance measures to be meaningful. At the same time, a clear understanding of the workload is needed so that the relationship between the values of the performance measures and the offered load can be interpreted, and so that random fluctuations in the observations can be filtered out [CockcroftPettit1998].

Measurements may be needed of all layers and components of a system. System resource measures are provided by hooks in the operating system accessible through commands, utilities, or system calls. The performance of an application may be measured using a load generation tool that monitors the rate at which work requests are generated and collects the response times of individual transactions or even of elements of transactions. Commercial databases can sometimes be programmed to track the frequency with which tables are queried, the response time of a query, and the frequencies with which locks are set and the amounts of time the locks are held. Middleware can be instrumented analogously.

In many cases, the emergence and market acceptance of a new software platform such as a web application server platform or a database platform are followed by the introduction of software to monitor its performance, and software to gather and integrate the performance measures from all the other performance monitors. Sometimes the measurement instrumentation is provided by the platform vendor.

There is a special problem associated with the measurement of computer systems, compared with the measurement of other types of systems such as chemical processing plants or biological systems or people: the system being measured is used to gather the measurements. Processing power is required to run the instrumentation. The measurements may be stored on disk, in a dedicated section of memory, or sent for storage across a network. This means that the measurement of the

measurement of the system effectively changes its state. Moreover, a program that is violating address boundaries may function differently when debug statements are turned on or off.

To prevent system measurements from excessively muddying the results, one must measure the resource usage by the measurement tools themselves to verify that they are not interfering with the system payload to any great extent. We note in passing that the problem of instruments interfering with the results is not confined to measuring computer systems. It is analogous to the Heisenberg Uncertainty Principle [AIPWEBSITE], which states, "The more precisely the position [of a particle] is determined, the less precisely the momentum is known in this instant, and vice versa."

The correctness of the measurements is a necessary condition for correct inferences to be made from them. Therefore, performance instrumentation must be validated by experiment. Before the instrumentation is used, it is a good idea to find out if anyone else has had difficulties with its giving suspect results. The existence of online forums makes this much easier than it would have been prior to 1995, when the World Wide Web was not ubiquitous. The opinions expressed in the online forums should be corroborated with experiments conducted locally if there are any doubts.

In the remainder of this chapter, we will learn how the user can obtain measurements at the system level and at various software levels. We shall also examine how measurements might be gathered for multiple hosts simultaneously, as is usually the case for multitiered systems hosting web applications. We shall look at the individual measurements that are available from the system utilities and show how to relate them to the performance metrics we discussed in Chapter 2 and to the principles underlying the performance models we discussed in Chapter 3. Since performance requirements must always be expressed in measurable terms, we shall explore how measurements can be taken to determine whether performance requirements are being met.

Disclaimer: This chapter describes general principles of computer system measurement. There are many commercial and open-source tools and programs for measuring system resource usage. Some tools are supplied with the operating system. Many use operating system primitives, software probes embedded in middleware or the application, or some combination thereof. Mention of a tool by the author does not imply endorsement or criticism. Similarly, omission of a tool does not imply criticism. The discussion of a problem with a measurement tool is based on the author's experience with the

version of the tool used at the time and is stated for the purpose of illustration only, not as a warning about a particular tool. This chapter should not be regarded as a complete listing of the measurement tools that are available, or as a complete catalog of the characteristics and capabilities of these tools. It is incumbent on every tool user to validate the accuracy and effectiveness of the measurements produced, and of every counter used, as the results may vary from release to release of the host operating system and from one system configuration to another.

8.2 Distinguishing between Measurement and Testing

It is important to draw a distinction between performance measurement and performance testing. Performance measurement is concerned solely with how performance data should be measured and collected. Performance measurement can be done while a system is under test or while it is in production. Indeed, performance measurement should always be done for systems that are in production when possible to ensure that they are running properly and to enable the owner or user to ascertain whether the system is running out of capacity, is running at reasonable levels of usage, or has too much spare capacity. Performance testing is an exercise in which a system is subjected to loads in a controlled manner. Measurements of resource usage and system performance are taken during the performance test for subsequent analysis.

The following documents should be created as part of the performance engineering process, in part to underscore the distinction between testing and measurement, but also to enable the system architect, owners, and administrators to understand the resources that will be needed for measurement and testing:

- A performance measurement plan describes the instrumentation that will be used to collect measurements.
- A capacity management plan describes how a system will be measured in production, and how these measurements will be related to measurements of the work that is being offered to the system at different times of the day or under different operating scenarios. A performance measurement plan could be part of the capacity management plan.

- A performance test plan describes what measurements should be collected in the course of a performance test and why, the work plan for the test, the pre- and post-test conditions of the test environment, the set of workloads to be tested, and the workload drivers to be used. A performance measurement plan could be part of the test plan.

This chapter concerns performance measurements only. Performance test plans will be discussed in Chapter 9.

8.3 Validate, Validate, Validate; Scrutinize, Scrutinize, Scrutinize

Measurement tools consist of tools and programs created by people. Therefore, there is always the possibility that they might not function correctly. Since crucial calculations and decisions will be made based on the measurements, it is important that they be validated at every step. There are many known bugs in instrumentation and tools for the collection of measurement. The World Wide Web facilitates researching them. If any measurements look peculiar, they probably are. If the instrumentation was homegrown, check it carefully. If it was acquired externally or came with the operating system, check it thoroughly and research anything strange online. Your measurements must conform to physical laws and be algebraically consistent with one another. That means that they must conform to Little's Law, the Forced Flow Law, and the Utilization Law. Timing measurements must be sensible. This holds for IT systems and for the physical systems that they might be used to control. Einstein told us that nothing can move faster than the speed of light. An observation that brought this into question turned out to be due to a wiring error [ARSTECHNICA2012]. This story illustrates that the performance engineer's mantra must always be *Validate, validate, validate; scrutinize, scrutinize, scrutinize*.

8.4 Resource Usage Measurements

Resource utilization is a fundamental indicator of the demands being made on a system. Because it is usually too costly to measure individual service times, the measured utilization of a resource is usually the

primary source of the measured service time. It is obtained from the Utilization Law ($U = XS$) we studied in Chapter 3.

8.4.1 Measuring Processor Usage

Blaming the lack of sufficient processing power for poor system performance is very common. Measuring processor utilization and the utilizations of other devices is the best way to determine whether that is indeed the problem.

Processor utilizations are usually obtained from counters that measure idle time. The utilization generated by the system is the complement of the idle time. The idle time counter is increased whenever a processor spends time executing the idle loop, also known as the mill soaker. This is a process that operates at the lowest level of priority. It might consist of a single instruction to branch to itself in an infinite loop. It continues executing until it is preempted by a higher-priority activity such as an I/O completion interrupt or a user process. If P is the time-averaged fraction of time that the processor spends executing the idle loop, the utilization is computed as $1 - P$.

There are further refinements for collecting the amount of time the processor spends executing interrupt handlers, executing system calls in kernel mode or privileged mode, and executing user code.

Precisely how this counting is done depends on the operating system and on the architecture of the hardware on which the system is running.

One way to do so is to accumulate the number of idle cycles in one counter (perhaps a register) and the total execution time of the system in another counter (also a register). The two counters are set to zero at the same time. The ratio of the idle counter's value to that of the cumulative clock counter is the fraction of time that the system was idle. In the 1970s and 1980s, before microprocessor-based systems became ubiquitous, resource usage could be measured by attaching probes to each of the bits of the instruction address register (also known as the program counter). The probes would sense whether the bits were zeros or ones. The probes were connected to a hardware monitor consisting essentially of a plug-configurable logic board and a tape drive. ANDing the combination of bits corresponding to the address of the idle loop with the bits of the instruction address register and a sampling strobe enabled one to determine how much time was spent executing in each address. The fraction of time the system is idle is one minus the utilization.

There is a fundamental assumption about inferring CPU utilization and CPU service times with reference to the clock cycle: the frequency of the CPU's clock cycle must be invariant over time. This is normally true for systems that are not subject to a power management regime. Power management is used to conserve energy, to extend the life of a battery between charges, or to reduce the amount of heat generated by the CPU during periods of low utilization. It follows that the system under test should be connected to the power supply and that power management should be disabled when measuring resource usage for the purpose of determining processing time and for the purpose of comparing processing times with one system configuration or another, especially if one is trying to determine whether one software implementation is faster than another. Similar issues might arise with other devices whose speed is varied according to the available amount of power, or to reduce the amount of heat generated if necessary. For example, hard drives in battery-powered laptops may spin more slowly when the power adapter is unplugged, and the clock speeds of processors may be reduced as the battery power declines.

The Windows, Linux, and UNIX operating systems all provide tools to show the average CPU utilization. On systems with single processors, this is simply the utilization of that processor. In multiprocessor systems, one must distinguish between the utilizations of the n individual processors $U_{CPU,k}$ for $k = 0, 1, ..., n-1$ and the overall average processor utilization,

$$\overline{U}_{CPU} = \frac{1}{n}\sum_{k=0}^{n-1} U_{CPU,k} \tag{8.1}$$

With multicore and/or multiprocessors now being available in even the cheapest laptops and netbooks, one should determine how many cores or processors are present before conducting extensive and intensive measurements of the system. Examining the utilizations of the individual processors shows us whether processing activity is equally spread among the processors, or whether the load is focused on only a subset of them. If the load is focused on a single processor and only one application is running, that application is inherently single-threaded. Adding processors to the system will not improve its performance. If the utilizations of the processors are positive but unequal, the load is not being spread among them equally. This could indicate that one running thread is more CPU intensive than another. If response times increase as the load increases, without increasing the

utilizations of the individual CPUs or even of a single one, it is likely that the system is suffering from a software bottleneck because insufficient threads are available to spread the load around, or because some other discrete software resource pool has been exhausted.

On single-processor, multiprocessor, or multicore systems, the average utilization shown by the CPU utilization counter in Windows *perfmon* and by *vmstat*, *iostat*, or *sar* in Linux and UNIX systems is the average utilization among all the processors or cores given by equation (8.1).

In Windows, the utilizations of all individual processors are graphically displayed in the Performance tab of the Task Manager. If Windows *perfmon* is used, the counters showing the aggregate average CPU utilization and the utilizations of the individual processors must be added to the list of active counters one by one. In Linux and UNIX, the utilizations of the individual processors are displayed numerically by the *mpstat* command issued from the command line. They may also be available from a log gathered by the System Activity Reporter *sar* with the correctly chosen options.

Windows *perfmon* also provides counters to display processor utilizations in user mode and in privileged mode. Processes are executed in user mode when they are not interacting with other system components by doing I/O. They execute in privileged mode, which has higher levels of priority and security, when performing I/O and other activity that involves interactions with other parts of the system, such as generating and receiving network packets and paging of virtual memory. Processes executing in privileged mode operate at a higher level of CPU priority than those in user mode and are not time sliced while in privileged mode. The corresponding mode in UNIX is called *kernel* or *system mode*. The CPU time in kernel mode appears in the column marked SYS or sys in the outputs of the standard UNIX measurement tools, including *vmstat*, *iostat*, *mpstat*, and *sar.*

8.4.2 Processor Utilization by Individual Processes

Long response times may be due to excessive processing demand or some other cause, such as a software bottleneck. Examining the CPU utilizations of individual processes is needed to isolate the root cause of long response times as well as to enable one to determine if an increase in a particular kind of activity is likely to exhaust the CPU or some other resource.

Both Windows and UNIX/Linux provide tools to measure the processor utilization by each process, and even by each thread:

- In Windows *perfmon*, it is possible to obtain the CPU utilization attributable to processes whose names are the program names.
- There are at least two ways to obtain per-process utilization in Linux and UNIX. The System Activity Reporter *sar* allows tracking of the processor utilization by a process when the PID (process ID) of the process is known. The PID must be supplied as a command-line argument to *sar* when *sar* is invoked. The IDs of all processes in the system may be obtained by invoking the *ps* (process status) command with option *–au*.
- The UNIX command *ps –elf* causes a line to be generated containing information about every process and thread that is running on the system at the time it is invoked. The information includes the amount of time used by every process to the nearest second since the process started up, as well as the swap space size of the process. To obtain the processing time during a particular time interval, *ps –elf* must be invoked at the beginning of the time interval and at the end. The CPU execution time used by any process is its cumulative processing time at the end of the interval less its cumulative processing time at the start of the interval. The resulting CPU utilization is the difference between the cumulative times divided by the end time minus the stop time, with the numerator and denominator expressed in the same time units. If this ratio is larger than one, the process is multithreaded and has threads executing concurrently on more than one core or more than one processor. To obtain the utilization for a particular process in a multiprocessor or multicore system, one must divide the ratio by the number of processors or the number of cores respectively.
- In UNIX and Linux, the data used by *ps* is also available from the set of files in */proc/<pid>/stat*, where *<pid>* is an integer denoting the identity of a process. Since the *ps* command shows CPU usage only to the nearest second, it will be necessary to obtain per-process CPU data from */proc/<pid>/stat* using system calls if a higher level of precision is desired.
- The UNIX/Linux command *top* displays the top users of processing power and also shows the most recently used CPU of each process. Since *top* itself runs as a process, executing it may cause processes to change processors more often than they

> might otherwise. Moreover, running *top* induces a distortion of its own: it must displace a process from one of the CPUs in order to run. The exact syntax and set of options may vary from one operating system version to the next. The performance engineer or other analyst should consult the *man* page corresponding to the operating environment of the system under test when choosing options for the *top* command and interpreting the output [TopManPage].

Beware: One cannot get a higher level of precision for a single observation than the granularity of the reporting clock. So, if the clock shows times only to the nearest tenth of a second, that is the best precision one can achieve for a single observation.

8.4.3 Disk Utilization

Fetching data from a disk or writing to the disk involves moving the disk arm so that the disk heads are positioned over the correct track (seek time), waiting for the data to be under the heads (rotational latency), and then transferring the data. The disk is said to be busy when any of these three activities occurs. Since data transfer is the goal of disk I/O, one would like to keep the seek time and rotational latency as small as possible. In some systems, it is possible to schedule I/O activity to minimize head movement and rotational latency. Hierarchies of disks, channels, controllers, and storage array networks may complicate the set of measurements needed to determine what areas of the system are to be improved.

In some architectures, there are bits indicating whether the disk arm is moving, whether a transfer is taking place through the disk heads, and whether the device is awaiting the rotation of the disk so that the record is under the heads. Each bit is set to one if the corresponding activity is in progress and zero otherwise. The disk is busy if the OR of these three bits is true. Synchronized sampling of the bits can be used to determine the fraction of time the disk is busy as well as the amount of time that the disk is seeking, transferring, or awaiting disk rotation. Additional bits keep track of whether the current I/O on the disk is for reading or writing. Sampling those bits can tell us the fraction of time that the disk is being used for a read or a write.

Some measurement tools, including *perfmon* in the Windows family of operating systems, can show disk utilizations attributable to reading and writing, as well as the amounts of data transferred to and from the disk. This can tell us a lot about how the disk is being used.

8.4.4 Bandwidth Utilization

Bandwidth utilization is essentially the rate at which bits are used divided by the available bandwidth of the transmission medium. The effect of utilization on performance depends on the network technology and on the media access discipline it uses.

- The original Ethernet uses Carrier Sense Multiple Access with Collision Detection (CSMA/CD) to mediate access to the Ethernet bus. If a collision occurs because two hosts were attempting transmission simultaneously, both hosts must back off for random amounts of time and then reattempt transmission. Offered bandwidth utilization of 25% or more effectively saturates the Ethernet, because repeated retransmissions are a lot more frequent when the bandwidth exceeds that threshold [AlmesLazowska1982]. The available bandwidth is effectively e^{-1} times the nominal bandwidth, a substantial reduction.
- By contrast, switched Ethernet does not have backoffs and retransmissions, so the maximum available bandwidth afforded by switching is essentially the nominal bandwidth.

We need to measure bandwidth utilization to determine if the capacity on one or more links is lightly or heavily used. Examples of counters available in Windows *perfmon* for the network interface include

- Bytes transmitted per second
- Bytes received per second
- Current bandwidth
- Output queue length
- Packets outbound discarded
- Packets sent per second
- Packets received/unicast per second
- Packets received/non-unicast per second

Because the meaning of these measurements is sometimes unclear, and because their meaning and how they are implemented might change from one point release of the measurement instrumentation to the next, the reader is strongly encouraged to review the documentation of each counter online before interpreting the values it produces. If the documentation appears to be ambiguous, controlled experiments in which fixed numbers of packets of known size are transmitted and received

during a fixed time period should be performed so that the meanings of the counters can be verified.

8.4.5 Queue Lengths

For any device, and for many software objects, the current number of queued processes, threads, or transactions is an indicator of congestion. *Perfmon*, the standard graphical performance monitoring tool supplied with various versions of the Windows operating system, allows one to gather the current queue length at time instants fixed intervals apart and the average queue length measured over successive time intervals of fixed length. The measurements do not include the number of jobs currently in service. As we saw in Chapter 3, the instantaneous queue length observed at constant time intervals is not equal to the average queue length, nor is the averaged value of instantaneous queue lengths equal to the average queue length over time in general. There is an exception to this that is mathematically justified. It is known as PASTA: Poisson Arrivals See Time Averages [Wolff1982]. PASTA states that the fraction of customers arriving according to a Poisson process that see a given queue length is equal to the fraction of time that a queue has that length. Hence, the average of the queue lengths seen by customers arriving according to a Poisson process is equal to the average queue length. Observers arriving constant time intervals apart are arriving according to a deterministic process, not a Poisson process, so the average of the queue lengths they see is not the same as the average queue length over time. Therefore, one should check queue length measurements to see if they are instantaneous or time averages. For Windows *perfmon*, the length of time over which all averages are computed is the time interval between the logging of the measurements. This is a configurable value.

8.5 Utilizations and the Averaging Time Window

The measured utilizations provided via operating system functions are samples with the values 0 and 1 (corresponding to idle and busy respectively) that are taken often enough to give the illusion of smooth behavior with values lying somewhere between 0 and 1. The length of the time interval over which one takes observations plays a major role in charting the evolution of the utilizations.

Averaging the processor utilizations over long intervals smooths out spikes and temporary surges in the utilization plot.

Averaging over short intervals allows one to see rapid oscillations in the CPU utilization. These oscillations are not harmful by themselves: they may simply indicate that the processes are engaging in CPU-intensive activity at some times and I/O-intensive activity at others, or that they are alternating between waiting for user input and processing it. To illustrate how different utilization patterns can result in the same average, we have plotted three hypothetical series of utilizations in Figure 8.1. In one of them, the utilization is a constant 0.5. In another, the utilization is 1 or 0 in alternating seconds, giving rise to a square wave. In the third, the utilization is 1 for the first 10 seconds and 0 for the second 10-second period. In all cases, the average utilization over the observation period of 20 seconds is 0.5, or 50%. Thus, radically different processor utilization patterns can yield the same average processor utilization.

The choice of measurement interval lengths has a strong influence on one's perception of how well the system is running. If the measurements are taken at very short intervals, for example, once per second, substantial fluctuations in the processor utilization and other measurements will be observed. By contrast, computing the average processor utilization during a 1-hour period could mask any observation that the system is alternating rhythmically between periods of saturation and

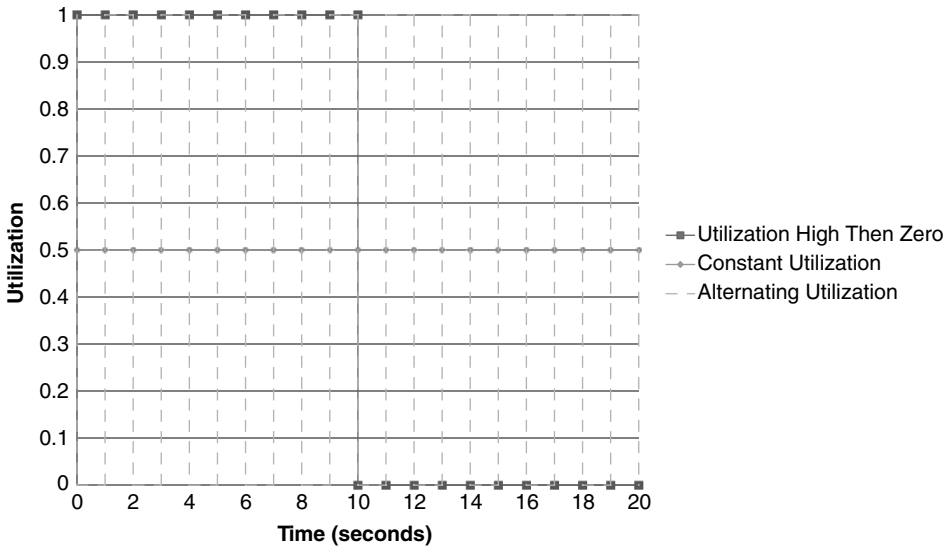


Figure 8.1 *Series of utilizations, all with average 0.5 during the period between time = 0 and time = 20*

periods of zero utilization of a particular resource. Therefore, the length of the time window used for computing average performance measures must be chosen with care.

When concluding that a system under test is saturated, it is important to specify the lengths of the time intervals over which the utilizations were measured. The transient saturation of one or more processors is not necessarily a bad thing: it merely indicates that there are time intervals when it is saturated, and times when it is not. If these time intervals are short, one will be more likely to see short intervals in which the CPU utilization is 100%, as well as intervals in which it is 5% or even zero. From this, one could draw the mistaken inference that the processor is incapable of handling multiple transactions with this sort of traffic pattern, and that something needs to be tuned or modified to reduce the CPU utilization. The author has participated in studies in which concern was expressed about these intermittently high CPU utilizations with only one process running. Until the performance engineer pointed out that allowing more processes to do the same kind of work concurrently would smooth the fluctuations, considerable effort was being invested in reducing the sporadically high utilizations when it was not necessary to do so. The lesson one should draw from this experience is that although it is useful to measure utilizations with short time intervals, there is much to be learned by combining the intervals so that spikes are smoothed over so as to get a broader picture of system usage and capacity.

8.6 Measurement of Multicore or Multiprocessor Systems

It can be shown that the average queueing time for parallel servers fed by a single queue is optimized when the average utilizations of all the processors are equal. A multicore processor or a set of two or more parallel processors fits this description. Usually, operating systems balance the load among processors or cores by routing a queued process to the first processor that becomes free. In some system architectures, cache affinity is used to route a process to the processor at which it was executing most recently before it was preempted, so as to increase the likelihood of data pertaining to it still being in the processor's cache memory. In some situations, one may wish to configure the system to permanently bind a high-priority process to one processor and route all

other processes to the other processors. In that case, the processor utilizations could be unbalanced. The utilizations of individual processors can also become unbalanced if a process is single-threaded and is bound to a given processor by cache affinity. In some environments, the child threads of a process must all be routed to the processor on which the parent process is executing. If a process is multithreaded and subject to this constraint, it will not be able to exploit the other processors, even if they are idle. Early Java virtual machines suffered from this constraint [Killelea2000]. This inability to exploit parallelism undermines a system's load scalability [Bondi2000] (also see Chapter 11). Sometimes the problem is recognized only after measurements have taken place.

Figure 8.2 shows the utilizations of the individual processors in a two-processor system. The loads are clearly unbalanced. Were the loads balanced, the utilizations of both processors would be quite close to the average utilization shown. This plot is based on contrived data, but it is very similar to a plot of actual data taken from a Solaris

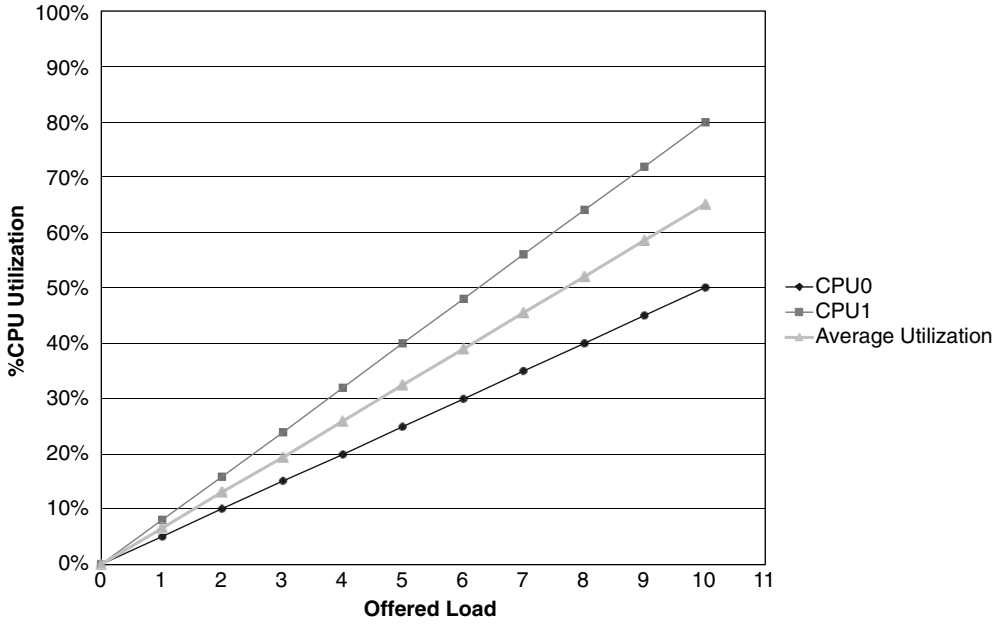


Figure 8.2 *Unbalanced processor utilization as a function of the offered load (synthetic data)*

server with two processors using the *mpstat* command. The server was running a version of UNIX supplied by Sun Microsystems that used cache affinity. The utilizations of the individual processors and the average utilizations are linear with respect to the offered load, and thus they obey the Utilization Law. An inspection of the processor utilizations attributable to individual processes showed that one CPU-intensive process was bound to the more heavily used of the two processors. The CPU utilization attributable to this process was approximately equal to the larger of the two processor utilizations at all load levels. Less-CPU-intensive processes ran on the other processor. We shall examine this further in Chapter 11, which deals with scalability.

Sometimes it is necessary to conduct a pilot test with a contrived process to clear up any questions about the interpretation of measurements and the function of the operating system when the documentation is not entirely clear. When measuring a multiprocessor system, one might be confronted with two questions:

1. How does the operating system balance threads or processes among the processors?
2. What do the utilization counters generated by the operating system actually mean?

One way to confront these two questions simultaneously is to run a pilot test of a program that is known to generate at least as many threads as there are processors and see how they are spread around. The threads should be very CPU intensive and not contain any variables in common. Each thread will contain an infinite loop that executes arithmetic operations, so that there will not be any I/O to cause the thread to give up the CPU. Each thread consists solely of an executing fragment of the form shown in Figure 8.3. Within the loop, a is repeatedly incremented. It is also repeatedly decremented to prevent integer overflow.

If the number of threads is equal to the number of cores or processors, the utilization of each one will approach 100%. If there is one more thread than there are cores or processors, a thread will be seen to bounce from one processor to another. This is manifested by a rapidly oscillating CPU utilization whose maximum value seems to move from one processor to the other.

```
Thread AddAndSubtractRepeatedly()
{
        local int a;
        while (TRUE)
            {
            a++;
            a--;
            }
}
```

Figure 8.3 *Metacode fragment for testing CPU load balancing*

8.7 Measuring Memory-Related Activity

It is essential to measure memory-related resource usage to determine whether there is sufficient main memory to meet performance needs, and to determine whether there is excessive paging. Windows *perfmon* provides counters that can be chosen from a menu in a dialog box to do this. Linux and UNIX provide values of relevant counters as part of the outputs of system commands.

It is necessary to measure overall memory-related activity and the memory usage of individual processes. When a constant transaction rate is applied to the system, steadily increasing swap space usage and/or steadily decreasing free space are indicators of a memory leak. The process that is leaking can be found by examining whether its image size is increasing over time. The image size of a process is the total of the sizes of its program space (sometimes called the *code segment*) and its data space (sometimes called the *data segment*), whether currently loaded in memory or not. In UNIX and Linux, the entire image resides in the swap space, while only a subset known as the *resident set* resides in main memory. The *ps –elf* command generates the current swap space size (column SZ), current resident set size (column RSS), and the cumulative CPU time used since the process started, among other measurements. Executing the command at regular intervals—for example, in a shell script that sleeps for a fixed amount of time and then wakes up to execute the command—enables the values of these measures to be tracked over time. In test environments, a sleep interval of 10 or 15 seconds is useful for processes whose lifetimes exceed the runtime of a performance test, which could last anywhere from 10 minutes to several days depending on the purpose of the test. In production environments, less frequent sampling may be required to reduce the risk of a log file consuming all available disk space. The desired performance measures for each process may be extracted from

a log file using a script written in *awk*, Perl, Python, or some other scripting language of the analyst's choice.

8.7.1 Memory Occupancy

The size of the virtual address space of individual processes is available in UNIX and Linux from the *ps* command. The private memory space of a process is available in Windows from the Task Manager's Processes tab.

The sum total of the sizes of all process images in UNIX and Linux is equal to the size of the swap space, which can be obtained from the *vmstat* command. The analogous quantity in Windows is the number of committed bytes. It is easily seen in the Task Manager. It is one of the fields of the Memory object in *perfmon*, which, unlike the Task Manager, can be configured to save a log of all collected performance measurements to disk.

8.7.2 Paging Activity

It can be hard to attribute paging activity to individual processes if many of them are active concurrently. This is one reason why it is sometimes necessary to measure the performance of processes or applications in isolation. Page reads and page replacements are indicators that insufficient memory is present, or that there is a problem with the way memory has been allocated. Page writes need not be indicators of trouble, since the operating system may routinely write modified pages to the paging device. It is hard to break out disk usage when paging, swapping, and application-related I/O activity are all on the same secondary storage device. To reduce the risk of contention for space and for I/O bandwidth, it is often sensible to use different disks for these purposes. The measurements of individual components of disk usage will then be available by default.

8.8 Measurement in Production versus Measurement for Performance Testing and Scalability

Usually, one needs to collect measurements of a smaller number of resources in production than one does when doing performance or other types of diagnostic testing. Moreover, one tends to collect measurements at longer intervals in production than one does during testing.

For transaction-oriented systems in production, one is mainly concerned with monitoring the evolution of resource usage as a function of load during the course of the day, week, month, or year. For this purpose, it is sufficient to collect measurements of processor, I/O, network, memory, and per-process usage once per minute or even once every 5 minutes. All of these measurements must be collected routinely for capacity planning purposes. The collection of memory data, particularly the size of the swap space in UNIX and Linux or its counterpart in Windows systems, is necessary to prevent the onset of a system crash if there is a memory leak.

The costs of measurement arise as follows:

- There are processing and I/O costs associated with every measurement that is collected. Network costs are also incurred if the measurements are sent across a network to a central collection point such as a network management system.
- Every measurement that is collected has to be stored for future interpretation, unless one is doing "quick and dirty" measurements looking at console logs on a screen.
- The total collection costs increase linearly with the frequency with which measurements are collected. The storage costs increase linearly with the number of measurements that are collected and with the frequency with which they are collected.

When planning a test, one must ensure that sufficient space is available to store the measurements for the entire test run. Otherwise, data may be lost because the storage space is exhausted. This holds whether disk space or memory is used. If pilot measurements show that memory is plentiful and that memory bus contention is not severe, one may wish to consider storing observations in memory and then dumping them to a secondary storage medium, such as a local disk or an external hard drive, at the end of the run. This will reduce the risk of measurement I/O causing spurious delays in the system under test during the test run. This is especially important for measurements taken within the application. On the other hand, if pilot tests show a lot of paging because memory is not plentiful, cleaner measurements might be obtained by writing them directly to disk if this is not done too often. Pilot experiments should be performed to determine which of these is the wiser course. The experiments may also reveal if some other method of storing measurement data should be used, such as a removable memory card or USB stick with low latency. The full test

plan should then be executed using the storage device of choice to store the measurement logs.

8.9 Measuring Systems with One Host and with Multiple Hosts

Many web-based systems consist of multiple logical tiers. In many cases, each tier may reside on its own host. For example, the web server, which serves pages, the application server, which implements the business logic, and the database server may all reside on separate hosts. To increase reliability and preserve system continuity in case of a failure, each tier may reside on a pair of hosts. When the hosts are collocated, they will be connected to each other by a high-speed local area network, such as a switched Ethernet. Transactions arrive through the web server tier. The application-layer tier generates queries to the database server and passes the results back to the web server for presentation to the user. A simplified view of a multitier system is shown in Figure 8.4.

To measure the performance of the system, one must measure the resource usage of each of the hosts in each of the individual tiers, as well as the end-to-end system response times and, where possible, the response times of operations within each of the tiers. Two problems are immediately apparent:

1. Each host and measuring device or load driver has its own clock, so the times may not be synchronized. That could complicate the task of relating measurements and time stamps on different hosts.

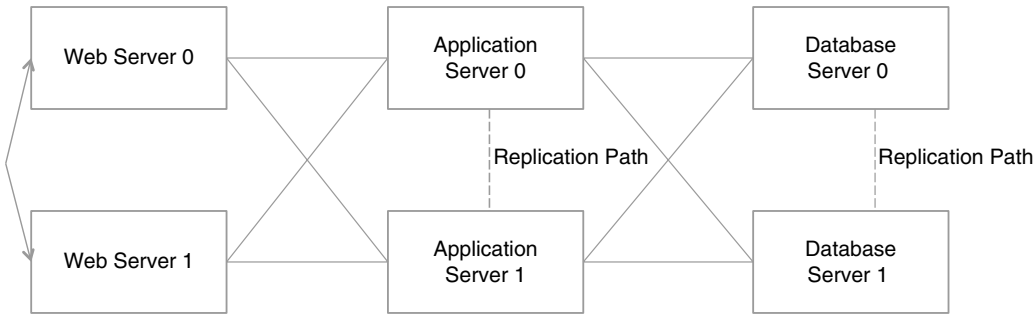


Figure 8.4 *A multitier configuration*

2. Each host and measuring device is potentially collecting large quantities of data from different sources. These must be gathered and reconciled for analysis.

We first consider clock synchronization, and then consider how to deal with multiple sources of measurements.

8.9.1 Clock Synchronization of Multiple Hosts

When plotting measurements from different hosts on a common set of axes, one wishes to be sure that their clocks are aligned so that the events occurring on one host or measurement device can be properly related to events occurring on another. Clocks on all hosts and measurement instrumentation involved in a test should be set to the same time zone and should be synchronized with a common time server. Since it takes different amounts of time for time signals to reach different hosts, some variation between the clocks is unavoidable. Therefore, one should strive to ensure that the clocks are close enough to prevent unnecessary discrepancies in the times of events and measurements. For example, the transmission time stamp on a received packet should always be earlier than the current time at the receiving host. Since clocks drift, resynchronization with the common time server should occur often enough to keep clocks sufficiently synchronized to prevent discrepancies between send and receive times from occurring. We use the term *sufficiently* because the overhead in observing and recording the time itself means that perfection cannot be attained. If the time server is many network hops from individual nodes, the random variation in the elapsed times synchronization messages take to reach their different destinations may induce an error of its own, not to mention delays caused by possible contention within the time server itself.

8.9.2 Gathering Measurements from Multiple Hosts

There are different approaches to gathering measurements from multiple hosts. The simplest method is to initiate measurement on each host in turn, and then gather the resulting log files onto a separate workstation for analysis after each experimental run, perhaps manually. In production systems, software agents can be stationed at each host for the purpose of gathering data. They can then relay the data to a central gathering point when polled by a central data server, such as a network management system. Some agents can be configured to send data to the central data server at regular intervals. They might also be

configured to send data under unusual circumstances, for example, when the memory pool occupancy has attained a designated threshold, indicating that a system crash might be likely. Repeated gathering of data facilitates display on one or more monitoring stations. That allows administrators to observe when a system is overloaded or approaching overload. The administrators can also observe the onset of quiet periods, which might be the best time to take one of a set of identical servers out of service for maintenance or to schedule a backup.

Automated gathering of performance data can be more convenient than manual data gathering, especially if several sources and performance measures are involved. A wide range of tools exists for this purpose. Some are available commercially, and at least one is available as an open-source tool.

Network management stations can be used to gather information centrally from agents on individual hosts. Under a standard network management protocol known as SNMP (Simple Network Management Protocol), the measurement information on each host is stored in management information bases (MIBs) in standard formats. The MIBs follow a standard hierarchy. Data can be collected from the MIBs using polls initiated at the network management stations at configurable intervals, or on demand using a form of *get* command. The SNMP agent on each node might also be configured to send alarm messages to the network management station when a designated performance measure crosses a network threshold.

Munin is an open-source measurement tool that was originally developed in Norway [Munin2008]. A Munin Master centrally gathers performance data from Munin Nodes on individual hosts at regular intervals (every 5 minutes) and uses a web-based interface to generate graphs of resource measurements. It is highly customizable. It can be equipped with plug-ins to gather all sorts of performance data of the user's choice from hosts in various flavors of UNIX and Linux, as well as Windows hosts. It can also gather information from SNMP using the MIBs.

Gathering data across a network and centralizing it while the system is running can be conveniently automated to provide information at regular intervals, usually once every 5 minutes. Some off-the-shelf products are configured to gather the data once every 5 minutes. Sending large amounts of performance data to a central monitoring station could incur considerable overhead, so the frequency with which data is sent must be chosen with care. Sending the data every 10 or 15 seconds, which might be useful for diagnostic, testing, and debugging

purposes, may be informative, but it could also consume enough bandwidth or incur enough cost to interfere with production.

Collecting data on each host using local utilities and then manually moving the logs to a central host might seem cumbersome. It is usually an adequate approach within the test environment and incurs minimal network overhead while a multihost application is running. Sometimes, this approach may be necessary if more elaborate tools cannot be configured to collect performance measurements more often than once every 5 minutes.

8.10 Measurements from within the Application

One may wish to take measurements within an application to determine how often particular data structures are used, as well as the total time for particular actions to be executed. The measurements may have to be crafted and built into the application by hand. This must be done with care, because (1) taking measurements from within the application will slow it down, and (2) logging the measurements will incur memory and/or I/O storage costs, which can mount up very quickly and possibly interfere with the performance of the system under test.

It is essential that the procedure for turning application-level measurements on be simple to reduce setup time and the risk of errors. This could be done through conditional compilation of the measurements or via a command-line option. Conditional compilation requires a lot of test setup time for large systems but has the advantage that the code need not repeatedly do conditional checks to see whether the measurements should be collected, stored, and written out. If a command-line option is used, the reverse is true.

Examples of application-level measurements include counts of the number of transactions of various types, the numbers of times particular data structures are executed, the numbers of times dynamic objects are created and destroyed, and the frequency with which static objects are drawn from a pool and returned to the pool, and how long they are held. Using this data, we can apply Little's Law to determine the average number of objects in use during the course of the run. It is important to count the number of times an object is acquired or created and returned or deleted. If the number of acquisitions in a test period is greater than or equal to the number of deallocations, the object pool may be causing a memory leak. This is undesirable, because pool exhaustion can crash the system and/or cause transactions to be lost.

If you are coding your own statistics collection program for an application-level object pool, you should collect the following data:

- Start and end time of the measurement period in the same time zone as all other measurements
- Average pool size (see Chapter 3 and [LawKelton1982] for how to do this)
- Largest number of allocated objects
- Smallest number of allocated objects
- Largest free pool size
- Smallest free pool size
- Number of pool allocations
- Number of pool deallocations
- Allocation rate = (number of allocations)/(measurement end time – measurement start time)

If you are coding your own statistics collection program to measure a time duration, such as a response time, you should collect the following data:

- Start and end time of the measurement period in the same time zone as all other measurements
- Largest response time
- Smallest response time
- Number of response times observed
- Rate of job completions = (number of completed response times)/(measurement end time – measurement start time)
- Average response time
- Standard deviation of the response time (must have two or more observations)

8.11 Measurements in Middleware

The issues one faces when collecting measurements in middleware are similar to those arising when collecting system-level and application-level measurements. One needs to be concerned about storage costs and processing costs. Off-the-shelf software may be available to collect the resource usage of the various services and applications within the

middleware tier. If performance issues are encountered, one should begin an investigation to find the cause by using external measurements of the processes and resource pools (such as thread pools that communicate between Java applications and databases) to measure the servers and their resource usage.

8.12 Measurements of Commercial Databases

There are many reasons why performance issues can arise in databases. Queries might be poorly written. Unnecessary and repetitive searches might be carried out while processing a query. Tables may not be well organized. Searches within the database might be poorly organized because of inadequate indexing. Inadequate indexing can lead to the need for a long sequence of table searches to find the records of interest. This is an example of the circuitous treasure hunt performance antipattern [Smith2000].

There are commercial tools available for the measurement of Oracle, Sybase, and SQL databases. For example, STATSPACK is used to tune Oracle databases. These tools are documented in many places. An explanation of the use of commercial measurement tools is beyond the scope of this book. Books have been written on this subject [Burleson2002, ShaibalSugiyama1996]. Instead, we briefly discuss what might be measured in a database, and why.

Response times can be degraded even if the CPU utilization by the database server is low. This can occur if tables are locked in their entirety instead of being locked one row at a time. Locking an entire table prevents threads wishing to access different rows from doing so concurrently. This causes prolonged contention for the lock and reduces concurrency. Some care is required in the decision to use row- or table-level locking, because the creation of locks and the mechanisms to mediate their acquisition and release can incur overheads of their own.

If a deadlock occurs in a database, it may be resolved by a deadlock detection algorithm. At the onset of deadlock, there could be a sudden drop in processor utilization until the deadlock is resolved. Deadlocks can occur if two or more processes are contending for a resource that is neither sharable nor preemptible, and if contention occurs in a manner that causes a cyclic dependency between the processes. The same holds for threads. For example, if Thread 1 attempts to lock Table A and then Table B before releasing both tables, while Thread 2 attempts to lock

Table B and then Table A before releasing both, a deadlock will occur if Thread 1 locks Table A and Thread 2 locks Table B, because each will be waiting for the table required by the other. The causes of deadlocks can be very difficult to determine. Deadlock is one of the things one should suspect if response times are increasing while process and I/O utilizations are down.

The following is a list of measurements that might be of interest in examining database performance:

- The number of times each table is searched
- The number of times each table is locked, and for how long
- For databases with row-level locking, the number of times each row in a table is locked, and for how long
- The response times of particular queries

This list is far from complete. Your database administrator and database architect may have their own thoughts about what should be measured.

8.13 Response Time Measurements

In transaction-based systems, the system response time is the performance metric that is most often experienced by the user. In batch systems, which are often used to generate large volumes of bank statements, bills, and the like, there is a need to complete an entire run within a certain amount of time. The complete runtime is also called the response time in mathematical models of system performance like the ones we saw in Chapter 3. When batch jobs contend for the same resources as transactions, it is desirable to run them overnight when the demand for transactions is much less likely. There may be a performance requirement that the entire run be completed within 5 hours.

The system response time includes the response times of individual servers and, in the case of multitier systems, the response times of various actions requested in the individual tiers. When the measured system response time is excessive, it may be necessary to drill down and measure the response times of the individual actions that constitute the transaction as whole, especially those that are likely to be taking the longest. These response times might be measured while the system is under load, or in isolation, perhaps even while conducting unit tests to verify functionality.

The response times of web-based interactions are very often measured by planting hooks in load generators that record the time at which a transaction began and the time at which it ended. The load generators are clients, sometimes called *virtual users*. It is best to run virtual user clients on PCs other than the hosts on which the application is running, so that the load generator does not use resources that could be used by the application under test. The virtual users are represented by programmed scripts mimicking the actions of users. The advantage of running load tests with virtual users is that they can be programmed to perform transactions in a predictable, repeatable way. This facilitates repetition of the experiments with changed system parameters under controlled load conditions. The rate at which transactions are generated and the number of virtual users should be configurable by the test engineer. Several commercial tools are available for this purpose. Typically, the tools will run scripts and gather response times and other performance measurements, including resource usage measurements.

The response times and resource usage of individual layers within the application may be measured by commercial tools or coded into the application directly as discussed earlier. The circumstances under which this level of detail is required may vary. Where possible, the performance characteristics of individual components should be measured before they are incorporated into the application so that problems can be isolated readily, just as one would perform unit testing on pieces of software to verify functionality before integrating them into the whole system.

8.14 Code Profiling

Code profiling measures the number of times every line of code is executed during the course of an experimental run. It slows down execution to the extent that the measured performance of the system as a whole with it turned on is not indicative of the performance in production under the same load. In any case, to be informative, code profiling is best used to examine code execution triggered by one transaction on its own, or a small, fixed number of transactions on their own. Code profiling can tell us if a procedure or method, or a set of lines of code within the method, is executed more often than expected and therefore offers us the increased possibility of gaining insights into how the code

should be tuned. It is also useful for debugging, because it can tell us if code that should have been executed was not or vice versa. While code profiling might offer us insights into how code can be tuned, tuning will not remedy problems that are inherent in the architecture of the system. As we discussed in Chapter 1, architectural decisions are frequently the root cause of performance problems.

8.15 Validation of Measurements Using Basic Properties of Performance Metrics

Whether one obtains measurements from operating system utilities or applications, all measurements must be consistent with one another. The data must always conform to the relationships between performance measures described in Chapter 3. If they do not, investigation is needed.

- All quantities must be greater than or equal to zero.
- Counts of transactions observed in one part of the system must agree with counts observed in another part of the system if all transactions were completed.
- The Forced Flow Law must always hold. This means that if n actions of a certain kind are known to be taken for each transaction, and there were N transactions, the action count in the observation period should be $N \times n$.
- Measured utilizations should always range from 0% and 100%. This is not trivial.
    - The author has seen disk measurements in excess of 100% on a Windows-based system. A web search revealed this to be a known bug. There could be a number of reasons for this, including observations being taken from a controller with more than one disk attached to it. A web search on the string "utilization exceeds 100% perfmon" yields several interesting pages on this subject, including [ITGUYSBLOG] and [MSOFTSUPPORT1].
    - A negative utilization in a real-time system being measured by a colleague was the result of the system clock moving backward instead of being incremented. A conversation with the engineers revealed that this was a known bug.

- Response times (or object holding times), throughputs, and average queue lengths (or average object pool occupancy) should always be related by Little's Law.
- The object pool occupancy must never be greater than the size of the pool or less than zero.

Since average service times or average service demands are obtained from throughputs, it is important that the utilizations and throughputs be gathered correctly.

8.16 Measurement Procedures and Data Organization

Performance measurement is costly because it requires an experimental setup, lab time, and staffing effort. Crucial decisions with significant economic impact may be based on the results. It is therefore essential that measurements be validated, that measurement procedures follow a carefully drafted script, and that measurement and test plans be prepared to answer the questions of interest and to reveal hidden potential problems. Moreover, the results of the performance tests and notes taken before, during, and after the tests should be carefully archived and documented so that the results can be reviewed. This is especially important if any anomalies are found. It is essential to determine whether a strange result is due to a property of the system under test or the manner in which the test was conducted. For example, the system configuration and build number are important aspects of the preconditions of the test that must be recorded. If a program is compiled with debug statements turned on and then subjected to a performance test, the amounts of CPU in system/kernel mode and the amounts of I/O activity will be seen to be much higher than they should be, but the results will be spurious because a system in production would not be compiled in debug mode.

Many of the laboratory disciplines one learns in physics, chemistry, and biology classes are applicable to performance testing:

- An experimental plan must be written describing an inventory of the equipment and samples to be used in the test, the experimental procedures, and the means of collecting the data.
- A checklist should be prepared on which the steps of the procedure will be noted as they occur, together with the times at

which they occurred. A notebook should be used to record the observations as they occur.

- Instrumentation must be validated. In an electrical experiment, one wishes to ensure that a voltmeter works as prescribed. In a chemistry experiment, balances used to weigh out samples must show zero when there is nothing on them. The balances may also have to be calibrated for correctness.
- The test environment must be clean. Only clean test tubes should be used in a chemistry experiment. Bottles and other containers of potentially dangerous reagents should be closed and removed from the test environment. Electrical connections must be tight. There must be no magnetic interference in a physics experiment, unless that is what we are trying to measure.
- Before testing begins, safety checks should be performed, including donning safety glasses and gloves and positioning a fire extinguisher as necessary.
- All staff involved in the experiment should be in position and alert.

Similar rules apply to performance measurement:

- A performance test plan must be written describing an inventory of the equipment and samples to be used in the test, the experimental procedures, and the means of collecting the data. This includes a list of system commands to collect the data with the corresponding settings, a list of scripts to drive those commands, and an explanation of how and where the data will be stored and reduced for presentation afterward. Performance test plans are discussed in the next chapter.
- A checklist should be prepared on which the steps of the test procedure will be noted as they occur, together with the times at which they occurred. A notebook should be used to record the observations as they occur. If performance testing is automated, this functionality should be included in the automated test script.
- Instrumentation must be validated. In the case of standard operating system instrumentation, validation may consist of checks that the presence of an offered load is reflected in a change in resource usage, together with thorough research on known bugs in measurement instrumentation so that these may

be taken into consideration when analyzing the data. *Note:* When hardware probes were used, checks were performed to ensure that they were responding to the voltage changes that are supposed to occur when bits are set to zero or one.

- The test environment must be clean. Nothing should be running on the system that is not related to the test. In particular, the list of running processes should be checked to ensure that only users who are working on the test are logged into any hosts involved in it, and that only processes that are related to the system under test, including those that would be executing in production, are running. Any extraneous users should be forcibly logged off and any extraneous processes should be killed.
- Before testing begins, safety checks of any equipment controlled by the system under test should be performed. Any equipment that is not involved in the test should be disconnected. Staff involved with the equipment controlled by the system under test should be dressed and equipped as safety regulations require, perhaps including donning a helmet, safety glasses, gloves, and/or ear protection and positioning a fire extinguisher as necessary.
- Measurement instrumentation should be turned on before load is applied to the system. Baseline measurements should be noted.
- All staff involved in the experiment should be in position and alert.
- The test load should be applied and the time at which it was started noted.
- Once the test is completed, all test data should be secured away from the test environment. Nothing should be deleted. The configuration of the system should be checked to ensure that it is the same as it was before the test started.
- The correct functioning of the system under test should be noted before and after the conduct of the test, as well as during it.
- A debriefing of the personnel involved in the test should be done as soon as the test is completed. This includes collection of personal observations.
- Analysis of the test data should be performed.

8.17 Organization of Performance Data, Data Reduction, and Presentation

The measurement data should be organized so that data files from different hosts and different measurement instrumentations all have distinct names. The filenames should include the instrumentation, source host, the date of the test, and some encoding of the test case and run number so that individual files can be retrieved easily later, and so that data files are not accidentally overwritten.

8.18 Interpreting Measurements in a Virtualized Environment

The performance of an application in a virtualized or cloud environment is heavily dependent on the resource demands of the other applications that are running in the same environment. This is not astonishing. It is also true of multiple applications running in a mainframe environment or in a cluster of parallel servers. Performance models of resource contention among applications or workloads date back to the 1970s [ReiserKobayashi1975, WilliamsBhandiwad1976, BruellBalbo1980]. In the mainframe environment, each application is essentially a single process within which multiple threads run. In a virtualized environment, each virtual machine mimics a physical one, to the point of providing its own performance counters. The problem is that these counters are inaccurate, in part because the virtual machine's clock may not tell time on the same scale as the "wall" clock. In other words, a second of time on the virtual machine's clock may not be the same as that of the physical machine or the clock on the wall in the machine room. Moreover, the measured resource utilizations of the counters in the virtual machines may not bear much relation to the counters in the host physical machine. We therefore recommend that detailed performance measurement and testing not be conducted in a virtualized environment at all until one is certain that the performance characteristics of the system under test in a physical environment are well understood.

8.19 Summary

The variety of performance quantities to be measured in the hardware, operating system, middleware, and applications and the large selection of measurement tools available and needed to do so indicate that there are many facets to computer and system performance measurement. While the measurement tools are varied, the practices to which one must adhere when using them are constant:

- The validity of all measurement tools and procedures must always be scrutinized.
- Measured values that appear to be unrealistic or peculiar should always be investigated, especially if they violate basic performance laws, such as Little's Law and the Utilization Law.
- Performance measurement and testing should always be done in a clean environment in which only the system being measured is running. This is analogous to using clean test tubes in chemistry experiments.
- The instrumentation used to collect measurements should not interfere with the payload work the system is designed to do.

These principles apply whether the instrumentation is new or in established use. It is essential to adhere to them when measuring new technology for the first time, so that one can determine whether measurements truly reflect the behavior of the system under study. They hold for every aspect of measurement considered in this book.

8.20 Exercises

8.1. A new programming language is introduced to implement monitoring and control systems. Explain how you would verify that executable code that is generated from the source code of this language is capable of running on multiple processors simultaneously.

8.2. An array of disks is addressed by a single controller attached to the I/O bus of a multiprocessor computer. Measurements of the array suggest that it is 100% busy despite the fact that the application is known to be CPU bound and not do much

I/O. Explain how the architecture and the instrumentation of the disk subsystem might be used to account for this discrepancy. How would you research the issue online? How would you corroborate suggestions for resolving the discrepancy that you might find there?

8.3. Measurements of an application that is believed to be I/O bound reveal short bursts of heavy CPU activity when observations of the performance measurements are taken every 2 seconds instead of every 15 seconds. When measurements are taken every 15 seconds, the average CPU utilization is much less than the peaks observed in the 2-second intervals. How would you investigate whether this is cause for concern? (*Hint:* Is the duration of high CPU activity longer than the required average response time for this particular operation?)

This page intentionally left blank

Chapter 9

Performance Testing

Performance testing is essential for avoiding unpleasant surprises in production, such as slow response time, inadequate throughput, and dropped transactions. In this chapter we will learn how performance tests can be structured to verify that the system has desirable scalability properties, such as resource utilizations that are linear functions of the offered load. We shall discuss performance testing practices and procedures and review and interpret actual performance data. This data illustrates how performance testing can be used to uncover undesirable properties of the system, preferably before it goes into production. The chapter concludes with a discussion of performance test automation and the value of automating the analysis of performance measurements.

9.1 Overview of Performance Testing

Performance tests may be conducted at any stage of the software lifecycle. The code that is subjected to performance tests should already have passed functional testing, so that functional problems do not muddy the results of the performance tests, making them hard to interpret.

One reason for running performance tests is to ensure that performance requirements and customer expectations about performance will be met. One must also verify that functional requirements are satisfied during performance testing, even if the system has passed all unit tests and integration tests. This is because functional problems

may be caused by concurrent programming errors that could not occur in unit tests when only one process or one thread is running. Performance testing may also be done to test the limits of system capacity, to obtain measurements for modeling and capacity planning purposes, and to identify workload scenarios or sequences of events that could cause the system to crash.

Performance testing may be used to assess the feasibility of supporting performance requirements, adding load, or adding functionality on a particular platform or with a particular technology. Understanding the performance characteristics of a particular platform or technology is a good way of minimizing the risk that it is inherently incapable of meeting performance needs. For instance, if it is known that an application is likely to invoke system calls or library functions at a given rate, testing the platform's ability to do so before code is developed or ported can be done to mitigate engineering risk. This is worth doing when one is considering porting an existing system to another platform or when the code has not yet been developed [MBH2005].

Of course, performance measurements may be taken of a system that is already in production or that is about to go into production, to be sure that the system is ready for service. Performance testing may be conducted at any stage of the software lifecycle.

While it is sensible to test the performance of those functionalities that have already passed functional tests, the correct functioning of a system undergoing a performance test must be verified to ensure that it is not adversely affected by being under load. Apart from providing the data needed to verify that performance requirements are being met, performance testing can uncover functional problems that cannot occur during unit testing or that may not have emerged during unit testing. For example:

- Concurrent programming errors leading to deadlocks, lack of thread safety, and delays caused by poor scheduling choices cannot emerge during unit testing but may manifest themselves during performance testing.
- Misconfiguration of object pool sizes may lead to very large delays that could be accompanied by low CPU and network utilizations or, in the case of memory pools, by large amounts of paging, or even a system crash.

In this chapter we shall discuss performance testing practices and procedures and explain how measurements can be used to diagnose the causes of system problems, including performance problems. We

shall also use performance modeling principles to structure performance tests to reveal various sorts of performance problems.

Experience shows that the deviation of performance measurements from patterns predicted by performance models is a sure sign of the presence of a performance issue that will diminish the capacity, usability, and even the reliability of the system under study. A deviation of this sort can also be a sign of a functional problem, such as the corruption of data caused by lack of thread safety or incorrectly implemented mutual exclusion on shared objects.

The following are examples of modeling principles, deviations from which indicate the presence of a performance issue or a possible fault in the instrumentation or data reduction:

- In a transaction-oriented system in which the arrival rate is held constant during a given period, the average performance measures taken at regular intervals during the time period should be constant also. Thus, graphs of average utilizations and average response times plotted against time should show little deviation from horizontal lines corresponding to their respective average values. If the average utilizations fluctuate markedly, even though the average arrival rate is constant, there is something amiss in the system. For example, sporadic drops in utilizations could indicate a drop in throughput (despite a constant arrival rate). This could be a sign of system deadlock. Similarly, sporadic spikes in response times could indicate deadlock followed by a timeout.
- The utilizations of resources such as processors, network bandwidth, and I/O devices should increase proportionately as the arrival rate is increased. In other words, the average utilizations at various load levels are linear with respect to the offered loads. This is a consequence of the Utilization Law and the Forced Flow Law. In addition, as the load is increased, the largest utilization should eventually approach 100%. Deviations from this rule indicate a problem in the system. For example:
  - A sudden increase in the utilization of the paging device in the absence of an increase in offered load, possibly accompanied by a large increase in processor utilization in kernel mode, indicates that an excessive amount of paging is going on.
  - A software bottleneck may be present if repeatedly or steadily increasing the offered load fails to cause the utilization of the bottleneck resource to approach 100%. Even though the bottleneck utilization ceases to increase with the load, there

may be an increase in the measured response time as a function of the offered load, together with an increase in the measured response time versus time when the load reaches the point that saturates some software bottleneck. Examples of software bottlenecks include contention for locks and insufficiently sized resource pools.

- Little's Law must hold for measured queue lengths or object pool occupancy, response times, and throughputs. Moreover, the sampled number of response times or object holding times must be equal to or in exact proportion to the number of transactions that are completed. If the number of sampled response times is less than what is required, the pool occupancy will increase along with the object holding time. This is indicative of leaks that could eventually cause a system to crash. All transactions that are initiated in the system must be completed at the same rate at which they occur. This is a consequence of the Forced Flow Law and of the *rate in = rate out* condition for the system to reach equilibrium. If jobs flow into the system faster than they are completed, they will either back up in the system or be discarded or fail. Symptoms of this include average response times increasing over time (perhaps with decreasing sample sizes in successive intervals of equal length), large transaction failure rates (perhaps accompanied by attempts to retry the transactions), and the presence of a large backlog of jobs or transactions that are completed more slowly than the original arrival rate once the arrivals are turned off.

By planning and structuring performance tests to determine conformity with the behavior predicted by performance models, one can identify performance issues that could undermine system scalability or eventually cause system failure, apart from enabling the evaluation of the ability of a system to meet performance requirements.

9.2 Special Challenges

Normally, the integration of the system and integration testing would precede performance testing of the complete system. Still, we have seen instances in which the performance testing team is the very first group to set up, install, integrate, and configure a software system in its entirety and subject it to something like a live load. The system has

never been used before. Installation and configuration entail the use of documentation and tools for the first time ever, as well as the creation of data files and data streams intended to be like those that will be in place when the system goes live.

The performance testing team should not be required to go about its task in isolation from other stakeholders. To facilitate the timely resolution of issues that arise, the performance team should have ready access to the system architect, functional testers, the development teams, and the teams that developed the integration and configuration tools needed to set up the system. All of these stakeholders should be available to provide support in dealing with any bugs that arise. Any necessary changes to the system under test should be implemented while following your organization's change management process, so that the changes and their effects are clearly documented. A configuration change will be less arduous than one involving a change to the code. If a change must be implemented more quickly than a change management process would allow, the lead performance engineer must ensure that all changes and their observed impacts are carefully documented so that the changes can be appropriately logged. This is essential for software auditing purposes. Some customers for the software may require it.

Conversations with product managers and even users may help the performance testers build and implement a performance test plan that has broad credibility. Acquaintance with the domain of application of the system under test is also useful. The performance testing team should have access to the specifications for functional and nonfunctional requirements so that they can determine if observed behaviors are correct, or whether strange behaviors are due to ambiguities or other defects in the requirements themselves. A defect in a requirement or a poorly written specification could lead to unexpected behavior that is sufficient to prevent any further testing from going forward. Systems based on service-oriented architectures are vulnerable to this problem: if one of the services has undesirable performance characteristics, hangs, or goes into an infinite loop, nothing built on top of it will work.

9.3 Performance Test Planning and Performance Models

Figure 9.1 shows an example of a simple performance testing environment in which a system under test is driven by M load generators. Each load generator may be configured to generate transactions at a constant rate, or to await a response to each transaction before generating

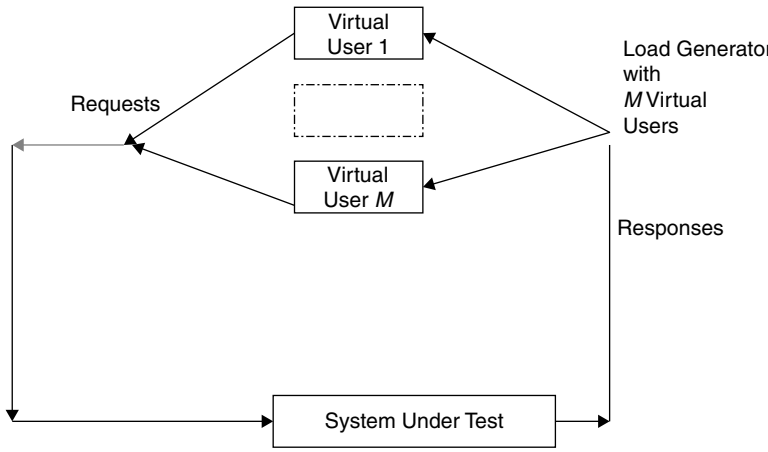


Figure 9.1 *Hypothetical load generator and the system under test*

the next one. In this example, the transaction response time is measured from the instant in time at which a task is initiated to the time at which the load generator receives a response. The response times in other situations may be defined and measured differently. For instance, in a network management system, the time to respond to an alarm or trap might be measured from the time at which the trap message is read from the LAN card to the time it is registered in a database, or the time at which an action is triggered to display an alarm on a console.

From the Utilization Law, we know that resource utilizations attributable to a particular type of transaction are linear functions of the rate at which that transaction occurs. Failure of a system to conform to the Utilization Law is indicative of a malfunction or of a design flaw. Performance tests should be designed to test for linearity of the utilization with respect to the arrival rate of some defined unit of work so that such problems can be revealed. Linearity might break down because of a property of an algorithm in the system under test. For example, a series of transactions that trigger insertion sorts makes processing demands that increase over time, because the cost of an insertion sort is proportional to the square of the number of entries already made. In that case, the processing time per transaction is increasing over time, and the system is not in equilibrium. Under those circumstances, the average service time is not defined and the Utilization Law breaks down.

In many systems, background activity for maintenance consumes resources that are or should be independent of the user's payload. For instance:

- In a telecommunications system, a switch may periodically exchange status messages with the other switches with which it is communicating to ensure that they are still operational. This may occur regardless of the rate at which the switch is processing calls at any instant.
- In a conveyor system, programmable logic controllers may periodically broadcast their status to their neighbors and also send status reports regarding the temperatures of the motors they control and other sensory information. This status information may be a function of the number of devices in the conveyor system, but it is independent of the rate at which entities are being transported by the conveyor, either in the system as a whole or in any segment of it. There is background processing activity associated with monitoring these messages and with recording them.
- A program that monitors resource usage in a computer system itself consumes resources. Consumption might be periodic, for example, when measurements are written to disk. The extent to which this is noticeable depends on the amount of resources consumed and on whether the observations are collected frequently or infrequently relative to the frequency at which averages are collected.
- A fire alarm system must repeatedly check the status of the devices that send messages to it, so that it can send out a notification to maintenance staff if any of them appear to be malfunctioning.
- Garbage collection might occur irregularly in a web application server that is largely implemented in Java.
- A database system may periodically initiate a cleanup process to purge it of records that have been marked for deletion but not yet deleted.

These are all examples of background load of some kind. Suppose that the background load is ongoing rather than intermittent. Denote the background load at device k when there is no payload activity

by b_k, which could be zero. Then, when the transaction rate on the system is λ, the utilization of resource k should take the form

$$U_k(\lambda) = \lambda D_k + b_k \tag{9.1}$$

To verify that our system has this property, we should run performance tests on it at different levels of λ to determine whether U_k is indeed linear in the offered transaction rate λ. One should choose the values of λ so that they cause the utilization of the bottleneck resource to range from 10% to 95%. Pilot tests at various levels of the arrival rate or transaction rate λ should be used to determine those values. Running the system with a single (possibly large) transaction rate λ tells us only whether the system was able to function at that rate or not. It does not tell us about trends in system loading.

The system should be measured for a nontrivial amount of time, such as 5 minutes, without any applied load present, so that one can determine if there is any background activity that consumes system resources. Doing so enables the determination of values of the intercept b_k for each device. One should also investigate whether there is a system-based reason for there to be a background load that runs even when no payload is present. It is important to do this because arbitrarily setting the intercept to zero implies an assumption that there is no background load, while also resulting in a possible incorrect estimation of the demand D_k.

We can evaluate the linearity of utilizations with respect to transaction rate with the following steps:

1. Run the performance tests at increasing load levels, for the same period of time for each load level, and measure the average resource utilizations over each run. If the load is driven asynchronously, the load levels may be chosen by varying the arrival rate or, equivalently, the average time between arrivals. This corresponds to an open queueing system. If, in the actual system, the user thinks between receiving the response to the previous transaction and launching the next one—that is, synchronously—the throughput can be increased by increasing the number of virtual users, and sometimes by reducing the average think time. If the load is driven synchronously, the throughput cannot be controlled, but the device utilizations should be linear functions of the measured throughputs nonetheless.

2. For asynchronous loads, plot the actual offered transaction rates and completed transaction rates against the target rates. If the actual rates are less than the target rates at any load level—that is, if they fall below the line $y = x$—one should check whether transactions are being lost, whether they are indeed being generated at the desired rate, or whether they are backing up somewhere in the system under test because something there is saturated.
3. Plot the utilizations against the actual transaction rates observed (as opposed to the target rates, which could be different).
4. Examine the plots.
   a. If the resulting plots of the individual utilizations appear to be linear or nearly linear in the regions in which no utilization is close to 100%, one can obtain the values of the slope D_k and the intercept b_k by fitting a linear regression line. Statistical packages such as SAS, S, R, and the statistics add-on in Microsoft Excel can all be used to fit linear regression coefficients. In Excel, the function is called LINEST. When using LINEST, a dialog box will appear giving one the option of forcing the intercept b_k through the origin or not. Forcing the intercept through the origin may change the slope; it also implies an assumption that there is no statistically significant background load. An analysis of variance will reveal whether b_k is statistically significant. If b_k is statistically zero, there is no measurable background load on device k. If b_k is not statistically different from zero, or if there is a physical reason for it to be positive, such as a known background task that executes continuously, arbitrarily setting it equal to zero could result in an erroneous estimation of the demand D_k on the kth device. This would lead to errors in the prediction of performance.
   b. If the resulting plots are not linear in the regions in which the utilization is less than 100%, follow a different line of investigation. One must also investigate further if the resource utilization of the bottleneck device levels off at a value below 99% regardless of how much one increases the transaction rate λ. This is a sign of a software bottleneck, since no hardware element is saturated.

Any regression line should be fitted only through consecutive points on the graph that appear to be more or less in a straight line and for which the resource utilization is strictly less than 100%. If the measured utilization of any resource is close to 100% at a particular load level, the system cannot reach equilibrium there, and long-term average values of performance measures for the run will be meaningless. Because the actual throughput is less than the offered throughput, the least throughput causing the utilization to be 100% may be lower than the offered one. Thus, the experimental condition at a point at which the system is saturated is different from the experimental conditions at other points, and the assumptions required for a regression to be valid will not hold if the offending point is included.

9.4 A Wrong Way to Evaluate Achievable System Throughput

Testers sometimes attempt to determine the throughput of a system by creating a very large number of transactions and then throwing them at the system all at once, or as fast as the load generator can send them to the system. The maximum throughput is taken to be the rate at which tasks are completed just before the system crashes, or the number of tasks submitted to the system divided by the amount of time it took to crash. This procedure is uninformative and misguided for a number of reasons:

- It is necessary to establish that the system is running properly and that functional requirements are met at light, moderate, and heavy load levels.
- It is necessary to ensure that the average resource usage and performance metrics are constant at constant loads, that the resource utilizations are linear in the offered load, and that the average response times are increasing with respect to the offered load. The maximum sustainable throughput is identified as the largest one for which the desired response time is attained.
- Using this "bang the system as hard as you can" method, the rate at which transactions are offered to the system

is uncontrolled, so there is no way to determine whether the utilization is linear with respect to the offered load. Moreover, there is no way to determine the arrival rate at which the bottleneck device will be saturated.

- Nor is there any way to determine the maximum sustainable throughput, that is, the maximum transaction rate at which transactions of a given type can be submitted while meeting stated response time requirements.
- Because the transaction rate is uncontrolled, we cannot determine the minimum transaction rate that would cause the system to crash, or the minimum transaction rate for which the average response time exceeds the desired value.

The "bang the system as hard as you can" testing method may provide some information about how systems function under saturation, as may be the case with alarm systems in an emergency situation, but it will not help us to identify performance trends and limitations on scalability the way a test structured to verify linearity of utilizations would. To verify linearity, one must subject the system to constant loads at different levels for sustained periods of time as we discussed in Section 9.3.

9.5 Provocative Performance Testing

"Banging the system as hard as you can" is an example of a method one might call *provocative performance testing*. The intent is to create or reproduce conditions that have provoked or might provoke the onset of a performance-related problem such as large or erratically varying response times or a system crash. Provocation is straightforward when the cause of a program malfunction has been identified and has been shown to occur the same way on the same input data. Such a program is described in the literature of fault tolerance as having a Bohr bug, after Niels Bohr's deterministic model of the atom. Performance problems and bugs that are due to concurrent programming issues can occur nondeterministically or not at all, and they may not be reproducible given the same input data owing to small changes in the execution environment or changes in the order of execution. These are known as Heisenbugs, after the Heisenberg Uncertainty Principle, which states

that the act of measurement changes the state of an electron [Jalote1994]. Provocation of a problem can take many forms:

- A multiprocessor system could be configured to run with all but one processor disabled, or with a fixed number of processors enabled, if the operating system allows for this.
- Deadlock could be provoked in a system that is prone to it on overload with a specific transaction mix by generating an intense load of transactions with a mix that is believed to be problematic. If the problem does not arise on one run, a randomized reordering of the transactions might provoke it on a subsequent run. This enables identification of the traffic ranges in which the problem is likely to occur.
- The consequences of a steady memory leak could be provoked by generating the transactions that cause it at a very high rate.
- Problems can be provoked by configuring a system to make it vulnerable to a chosen problem, and then subjecting the system to a load that brings it about. For example, if a network management system is known to freeze when the failure of a node renders many other nodes unreachable, one can bring on the freeze by taking the offending node out of service and observing what happens next.

If a remedy is proposed for a provoked problem, provocative tests should be rerun to check that the problem does not occur with the intended remedy in place and to check that the remedy is not causing other problems. This is difficult when the problem arises nondeterministically. Repeated experiments with randomization of sequences of actions may go some way toward providing assurance that the problem has been fixed.

9.6 Preparing a Performance Test

Preparation for a performance test entails acquiring a basic understanding of the system's functionality, the points at which measurements can be applied, and how a representative load can be driven into the system so that meaningful measurements can be obtained. It is also necessary to understand whether there are safety or legal constraints that might affect how the system can be tested. For instance, a control

system or a safety monitoring system must be tested in a way that does not undermine the safety of the controlled system or of the monitored environment, and the testing of a financial system may have to be conducted in a dedicated test environment to comply with antifraud regulations. Test equipment must be validated, and the number of load generators needed to offer a given load must be calculated using basic performance modeling techniques. We now elaborate on these points.

9.6.1 Understanding the System

To prepare a performance test, the performance testing team should acquire an understanding of the architecture of the system, how the system will be used, its functionality, the workload that is likely to be placed upon it, how the system should be instrumented to collect performance measures, and how load can be driven through it.

It will be necessary to procure and set up a test platform and a testing environment. The testing platform comprises hardware on which the performance tests will be run and an installation of the test system. The test environment includes the platform and the hardware, the software, the tools to configure the system, the tools needed to populate the system with user and application data, and the tools used to drive the load through it. Use cases must be identified for testing, and scripts developed to drive them.

- If the system is a control system, devices or stubs must be attached to it that are similar to those that will be driven in a live system. For instance, if the system under test is a fire alarm system, alarms or devices that process messages that have been sent to trigger them must be attached to the system under test to capture the resulting effect. The times when the associated events occurred must be noted.
- If the system is a transaction-oriented system or other system involving the use of a back-end database, the database must be populated with data that is very similar to live data. If the test is to be realistic, the size and scale of the database should be of the same order of magnitude as those of the live system or, if the test system is to be scaled down, of an order of magnitude that is in the same proportion to the whole as the test load will be. In addition, test scripts must avoid hitting the same database records repeatedly, unless the production system will do so. Because records are typically cached in memory, the response times of

the second and subsequent queries to a record may be lower than the response time of the first query. This will bias the results.

- The state of a database or other repository of output data before and after a test should be recorded. To enable repeatability and pure comparisons of test runs under different loads or under other conditions, it may be desirable to roll the database back to its state and condition prior to each test. Unfortunately, doing so may be so time-consuming and/or costly that this may not be feasible. For this reason, the usage history of the database should be recorded to determine if database aging is biasing the performance test results. Aging bias might be introduced if many records are marked for deletion without actually being purged. These marked but unpurged records occupy disk space and potentially increase search times. If the tables of which they are a part are loaded into memory, the memory occupancy will also be increased.
- Careful attention must be paid to the configuration parameters of the system and to the state of the build that is used for testing. Many incidents of poor performance in the field and in the lab are caused by incorrect configuration settings. A build with debugging statements turned on will perform much more slowly than one with those statements turned off for production, potentially yielding alarmingly bad performance.

Populating a large database can be a complex exercise. Because of privacy considerations and legal restrictions, one may not be able to populate a test database with live data without taking great pains to obfuscate the identities of those whom it concerns, and perhaps even altering the substantive contents of the records. There may also be complications because the system might have to send transaction logs to an external authority while recording transactions in the system. For example:

- In the United States, the privacy of health care records is governed by stringent regulations under the Health Insurance Portability and Accountability Act (HIPAA) [HIPAA2014]. Thus, any test system must be populated with fake data or with obfuscated live data.
- In countries belonging to the European Union, certain kinds of financial transactions are covered by European Market Infrastructure Regulation (EMIR) [EMIR2014]. The regulations include logging requirements and the clearance of some transactions via a central authority. All of this requires computer actions that might not be included in testing. This could complicate the preparation of the tests and the analysis of the results.

9.6.2 Pilot Testing, Playtime, and Performance Test Automation

Our experience has been that a testing team is more likely to write effective scripts for automated testing quickly if the members have an understanding of how the system will be used, and if they have a chance to play with it while looking at measurements before running planned formal tests. Playtime enables the testers to discover quirks in the user interface that may have been overlooked while the scripts were being written or while configuration was being done. It also enables testers to get a feel for how the measurement instrumentation works and for what patterns might emerge in graphical displays of performance data while the test is in progress, for example, by observing the graphical outputs of *perfmon* and the performance displays in the Windows Task Manager. Since the Processes tab in the Windows Task Manager can be set to display the processes in descending or ascending order of processor usage, the testers also have an opportunity to get a feel for where the processors are spending their time. This is especially useful for systems that have never been in production and have never been tested before. Playtime and pilot testing provide opportunities to uncover bugs that would render a full-blown performance test pointless. Moreover, if the system has never been run before and automated measurement instrumentation and tools for analyzing and reducing measurement data do not yet exist, the experience of manual measurement and data reduction will motivate the testing team to determine what is best automated to relieve them of tedious, repetitive work. A useful by-product of this is that careful automation reduces the risk of human error in any of the automated steps.

9.6.3 Test Equipment and Test Software Must Be Tested, Too

Since test scripts are programs in their own right, they should be tested to verify that they are correctly exercising the desired functionalities and correctly collecting the desired performance statistics. For example, the portion of a script that collects the response time after a mouse click should collect the time from the click to the corresponding response, not the duration of the login session.

The anticipated duration of a login session, the anticipated response time of a transaction, and the anticipated offered transaction rates affect the design and capacity of the performance testing environment as well as of the system under test. Instances of transaction generators must be spread among PCs or other workstations to ensure that transactions

can be generated at the desired rate. If the utilizations of the hardware resources on load generation stations are too high, they may not be able to generate the load at the desired rate. Similarly, it must be assured that the networks delivering load to the system under test have enough capacity to do so, and that production transactions and test transactions do not interfere with one another on the network.

The load generation platforms must have sufficient memory to support the number of instances needed to generate the desired loads. From Little's Law, we know that the average number of pending transactions of any kind is the transaction rate multiplied by the average response time. Because Little's Law is based on average values, we must size the system to accommodate more than the average number of pending transactions, so that memory or software bottlenecks do not degrade the capacity of the test generators. That would reduce the load that one could drive through the system under test.

Finally, the software used to extract and present the measurement data must be tested to ensure that it is generating correct outputs. Some of the checks that must be performed are rudimentary: the printed maximum, minimum, and average values of all quantities must correspond to those in the actual data, and estimates of variances must be non-negative, since they are equivalent to sums of squares. The measurement data itself must be sensible: there should be no negative quantities at all, because execution times are always non-negative, as are utilizations. This is not to be taken for granted. In one specialized system we have seen, descending sequences of time stamps arose because of errors within the operating system or within the data collection mechanism. They were found only because the data analysis software was crashing and generating error messages when computing utilizations. Since the occurrence of the problem was rare, we simply discarded the offending observations and made do with the remainder. If the development or other team is aware of this sort of anomaly, effort can be saved by ensuring that the performance testing team is aware of it.

9.6.4 Deployment of Load Drivers

Software load drivers are programs that use system resources, just like those in the system under test. The measurements of the system under test and the behavior under test should reflect the resource demands of that system only. If the load drivers are run on the same machines as the system under test, each will interfere with the other, and the results of the performance tests might be spurious. Moreover, in transaction-driven

systems, whether these are banking systems or control systems driven by events such as signals from sensors, the production loads will arrive from external sources. Therefore, the test loads should also arrive from external sources. That means that load generators should be deployed on hosts that are separate from those on which the system under test is being run. That will reduce the risk of confounding the test results with resource usage that is an artifact of the way the test was run. Avoiding this muddying of the measurements is essential to making an informed inference about the system based on measurement data. Confounding measurements of the system under test and the load drivers complicates the process of making an informed inference about the system based on the measurement data. For this reason, load drivers should never be run on the same hardware as the system under test.

The number of load drivers and the manner in which they trigger actions play a role in determining the rate at which they will drive work through the system. If transactions are generated asynchronously, that is to say one after another whether or not the previous one has finished, the transaction rate λ is limited by the capacity of the load driver or set of load drivers, or perhaps by the bandwidth of the network connection between the load driver and the system under test. The system will essentially behave like an open queueing network as described in Chapter 3. The throughput of the system will not be constrained by the system response time or by the think time. On the other hand, if the load generator is emulating the behavior of a user who awaits a response to a request and then thinks before initiating the next one, the throughput will potentially be constrained by the system response time, the think time, and the number of emulated users. To see this, recall that the system throughput X, average response time R, and average think time Z are related by the Response Time Law, which is a consequence of Little's Law. Thus, if M is the number of virtual users, we have

$$R = M / X - Z \tag{9.2}$$

Rearranging, we see that the throughput X is given by

$$X = M / (R + Z) \tag{9.3}$$

Thus, a longer think time or a longer response time could drive down the maximum throughput that can be attained. If it is necessary to increase the offered throughput without increasing the number of virtual users M, one must reduce the think time.

There are a number of reasons why one might not be able to increase the number of virtual users. If commercial load drivers are used, the license cost per virtual user might be high. Even if there is no additional monetary cost per license, the number of virtual users running simultaneously may be constrained by the number of load-driving PCs available for the test and by their individual capacities. Therefore, it may be desirable to increase the load that can be offered in the testing environment by reducing the value of the think time Z.

The maximum attainable throughput for a given number of virtual users M is obtained by setting the think time to zero. Thus,

$$X \leq M / R \tag{9.4}$$

Of course, R and X are what we are trying to find by measurement. As we know from our analysis in Chapter 3, the lowest possible value for the response time is obtained with $M = 1$. As we discussed previously in this chapter and in Chapter 3, the system throughput X will be increased by increasing the number of virtual users or logged-in terminals M but only to the point that no server is saturated, that is, to the point that no utilization U_k is close to or equal to 100% for any k.

9.6.5 Problems with Testing Financial Systems

There are special regulatory and security problems with conducting functional and performance tests of financial systems, including those that are used for internal reporting:

- In the United States, and in companies doing business with and/or trading on stock exchanges in the United States, the Sarbanes-Oxley regulations governing truth in financial reporting may impede the creation of dummy accounts and dummy transactions on production systems for testing purposes, as these could be used to generate fraudulent financial reports. It may be necessary to run tests of such systems on a platform that is entirely separate from the one that hosts the production system.
- Dummy account identifiers and passwords in the scripts must correspond to those stored in the system under test. Care must be taken to prevent them from being used to compromise security by allowing unauthorized users (virtual or otherwise) access to an area of the system that they should not have.

- Privacy concerns and regulations necessitate the obfuscation of account identifiers, US Social Security numbers, UK National Insurance numbers, and other identifying information in the test database.
- In the European Union, certain types of transactions must be cleared through a designated authority [EMIR2014]. Emulating the processing costs, network delays, and any remote processing delays due to this clearing may be problematic.
- Some systems allow access only during specific times of the day, such as stock exchange trading hours, and block access otherwise. If the performance test will be run outside these times, arrangements must be made to authorize system access and configure the system under test accordingly.

9.7 Lab Discipline in Performance Testing

Performance testing practice has a great deal in common with the lab discipline needed in physics, chemistry, or biology experiments. Care must be taken to ensure that the experimental environment and results are uncontaminated, that the test procedure and results are carefully documented, and that safety is ensured. Electrical, magnetic, and radio interference must be prevented in physics experiments; clean test tubes are needed in chemistry experiments; and a clean bench and rubber gloves must be used in biology experiments. Measures to prevent fire, air pollution, and other safety hazards are always required. In performance testing, one must ensure that nothing is running on the computer system that is not related to the performance test. If it is not feasible to use dedicated local area networks (and sometimes wide area networks) in the test, one should at least determine that the volume of traffic that is not related to the test is not so large that it would interfere with the test results or that the network traffic attributable to the test is not so large as to interfere with production work.

Accurate note taking is an integral element of lab discipline in the natural sciences. So it should be in performance testing. Parameter and configuration settings should all be logged, as should any observations arising during the preparation for and conduct of the performance tests. Automation helps to ensure that this is done, but it is not a complete substitute for contemporaneous documentation of an experiment.

9.8 Performance Testing Challenges Posed by Systems with Multiple Hosts

Many computer systems are implemented on multiple hosts. Different hosts may support different functionalities. For example, in multitier web-based systems, separate hosts may be used for one or more back-end databases, application servers that implement business logic, and a web server to act as the interface between the users and the business logic and to serve pages. Some hosts might be dedicated to balancing the load among the other hosts. Server farms to support search engines might have hundreds of hosts. The resource usage of all hosts must be measured so that system bottlenecks and areas of low resource utilization can be revealed. Each host may exhibit performance and/or functional characteristics that could have an adverse impact on the performance of the hosts with which it interfaces and even on the application as a whole. Problems and practices relating to testing the performance of web-based systems are described in [Microsoft2007].

Before testing the performance of the entire system from end to end, it may be desirable to test the performance of the individual system hosts, just as one would test the functionalities of individual components before testing the functionality of the entire system. It is much easier to identify problems arising in isolation than it is to find them when components are interacting with one another. This also holds for service-oriented architectures: it is much easier to test one service at a time and identify possible causes of performance problems than it is to test when the service is being invoked in many places in the application code. For a web-based multitier system with a back-end database, it is worth subjecting the database to queries in isolation so that it can be ruled out as the cause of slow response time problems that might be due to the business logic in the adjacent tier or to the inadequate allocation of JDBC threads.

When measuring the performance of complex systems, one is not only collecting response time and hardware resource usage data. One may also be collecting data from within the software platforms that implement the tiers of the system. These include data on locking and table scans of databases and data on waiting times and use case invocations within the business logic of the application server tier. All of this data must be reconciled so that performance issues can be identified. If one is collecting tens of streams of data from multiple hosts, the automated generation of plots of the performance data is essential for analysis. Tools to generate them such as PAL (found at http://pal.codeplex.com) or *gnuplot* (found

at www.gnuplot.info/) may be used to do this. Even with automated plotting tools, the analysis of the performance test results might be an onerous, labor-intensive task. The task can be simplified through the use of software that can identify common behavior patterns in performance data and automatically generate averages from those sets of data for which it is sensible to do so [Bondi2007a].

9.9 Performance Testing Scripts and Checklists

A performance test can involve a large amount of staff time and equipment, with all the costs that that entails. It is therefore essential that the execution of the test be carefully scripted, that the system be properly configured before the test, that all load drivers and measurement instrumentation be carefully calibrated before the test, that there be enough storage space for measurement and other test log files, and that nothing in the system be changed after pretest calibration and preparation.

A well-prepared performance test has much in common with a space shot, an orchestral concert, surgery, an airline flight, or a movie shoot. All personnel are in position before action commences, and all are equipped with scripts and checklists to ensure that all steps are carried out [Gawande2009]. Participating staff are trained on how to proceed and communicate if something unexpected occurs. At least one staff member is responsible for documenting the test as it unfolds. That includes logging the checklists and noting the times at which events occur and the results. This is essential for root cause analysis after the test. Some systems have peculiar security and authorization features. Test preparation must cater to these. Preparing a playbook and following it will go a long way toward ensuring team cohesiveness, the smooth running of the load test according to plan, and reinforcing the usefulness of the test results.

A performance testing playbook might include the following steps. Each step is conducive to ensuring orderly conduct of the test and the production of uncontaminated test results. A playbook along these lines should be followed for each test case.

1. Record the build numbers and release numbers of the system under test, and make sure that they are correct.
2. Check that the correct configuration parameters have been set, and that the proper test data is in place. Test databases must be

properly configured and perhaps rolled back to a desired initial state.

3. Check that the system is empty and idle, and that no transactions are going through it.
4. Check that users are authorized to perform the desired transactions at the time specified.
5. Be sure that only staff and virtual users essential to the test have access to the system. No one and nothing else should be connected to the system under test.
6. Check that only the desired set of processes is active.
7. Turn on all performance instrumentation and be sure that it is running properly.
8. Turn on all logs and make sure they are running properly.
9. Check that there is enough storage space for measurement logs and the logs that are generated by the system under test and by the load generators.
10. Check that the load generators are configured properly.
11. Check that all staff are in position, logged in, authorized to proceed, and quiet.
12. Begin load generation.
13. Check recording status continuously.
14. Terminate load generation at the stipulated time.
15. Copy all logs to at least one other storage location. *Do not erase anything.*

Once consensus has been reached on the contents of the playbook, the steps in it should be automated to the fullest extent possible and their outcomes logged for every test. This will provide an audit trail while reducing the risk of omitting crucial steps or of running tests with incorrect configuration and input parameters.

9.10 Best Practices for Documenting Test Plans and Test Results

A performance test plan contains a specification of actions that will be applied to a system and the test environment, a description of the tools that will be used to apply the actions, a description of

the instrumentation that will be used to measure the system, and an explanation of the expected outcomes. In this respect, a performance test plan is not unlike a requirements specification. The following is a list of desirable characteristics of a performance test plan document:

- The elements of the document must be traceable to requirements documents, product management specifications, architecture documents, and the like.
- The tests and the results they generate must be reproducible.
- The procedure for conducting the tests should be repeatable.
- To assure repeatability, the test environment should be described in detail. Moreover, the criteria for a clean test environment should be specified and steps included in the test plan and playbook to ensure that these criteria are met before each test case is executed.
- The description of the testing parameters and the expected outputs should be linked to the performance requirements document if there is one.

The test plan document should contain sections describing the goals of the tests, the functionalities to be tested, and the workload to be offered to the system under test. Since performance tests are often done on scaled-down versions of the production system, the test plan should contain a section explaining the architectural justification for assuming that the test results will scale up to the larger production system. For example, if a system is to be scaled up by increasing the number of identical servers and each server is completely independent of the others, it may be assumed that the capacity of the system will increase linearly with the number of servers, so long as they do not contend for any data elements in common.

The test plan document should contain a description of the test scripts, together with a path in the file system that shows where they are stored and the scripts used to start them.

There should be a section listing each of the test cases, the preconditions, the desired postconditions, and intended duration of execution of each test case.

There should be a description of the organization of the output files containing the test data and measurement data, including a diagram of the directory structure and a table explaining the file naming convention. It is essential that the file naming convention be consistent and humanly intelligible. Consistent naming enables one to check whether

cases were omitted. It also facilitates automated processing of large numbers of log files by automated scripts. A naming convention that is humanly intelligible enables one to identify the test case to which each file relates. This is the machine equivalent of labeling paper output meaningfully, so that all know the source of the output and the conditions under which it was generated.

9.11 Linking the Performance Test Plan to Performance Requirements

Ideally, a performance test plan should be driven by performance requirements. It could be driven by such considerations as the need to see how a system behaves under stress or by the results of pilot tests that revealed a problem. In some cases, performance tests may be undertaken on small-, medium-, and large-scale systems intended to provide the same functionality under commensurately small, medium, and large offered loads. In that case, the purpose of the performance test may be to determine the largest offered load that can be carried by each size of system while maintaining acceptable levels of performance and system stability. Essentially, one may be reverse engineering the throughput requirements of the system of each size after development has reached an advanced stage or been completed. This sort of prospective testing is useful for facilitating the sizing of configurations for different customer needs, but it carries the risk that the configuration sizes may not be well aligned to market segments. It is preferable to identify the market segments and to size the configurations accordingly before the system is built and tested. On the other hand, if it is found that a system does not meet those requirements once built, one can still use performance testing to reverse engineer the performance requirements so that a suitable market segment can be identified.

The nature of the test cases depends on the performance requirements that are to be validated. If one is attempting to validate, say, that the average response time of a particular transaction shall be 2 seconds or less when there are 100 transactions per second, test cases should be run at 50, 75, 100, and 125 transactions per second so that one can (1) determine that the response time requirement has been met, (2) determine whether utilizations are linear in the offered load, and (3) determine the smallest transaction rate for which the

response time cannot be met. If that transaction rate is less than 100 per second, a remedy must be sought. If that transaction rate is greater than 100 per second, there may be some room to grow the system load or add functionality once the system is in production, or else the system has been overengineered. Testing at a higher rate than the required one allows one to determine whether the system can cope with transient spikes when the transaction rate might exceed the specified rate.

When testing performance, one should also ensure that functional requirements have been met by verifying that outputs are as expected. Unexpected outputs could be, but need not be, caused by concurrent programming errors. The importance of this is underscored by the need to verify that concurrently executing programs have interacted as intended. That is the subject of the next section.

9.12 The Role of Performance Tests in Detecting and Debugging Concurrency Issues

We have already seen that the average values of performance measures in a well-behaved system will follow operational laws. In particular, if the average transaction rate is held constant throughout a performance test, the average performance measures should also be constant when measured with the same time granularity. So, if the average transaction arrival rate is ten transactions per second in successive 10-second intervals, the average response time and resource utilizations should also be constant in successive 10-second intervals, or nearly so. If the performance measures fluctuate wildly or show trends, all is not well with the system.

Adding processors or multiple cores to a system allows multiple threads to execute concurrently. In that case, we expect the average response time to be lower for a given transaction arrival rate, and the maximum sustainable throughput to be higher. If there is no improvement, but the performance of the system is no worse, the chances are that the system is single-threaded. This is especially likely to be the case if the utilizations of the individual processors or cores are unbalanced. Adding processors can also make the performance significantly worse. This is a sure sign of a concurrent programming issue of some kind.

If the average performance measures, including processor utilizations, are constant, but the average response times are longer than

with a uniprocessor or single-core system, it is likely that using multiprocessors is aggravating contention for a shared lock and/or for the memory bus shared by the processors. In particular, contention for the data structure implementing the head and tail of the CPU run queue or ready list could radically slow processing down [DDB1981].

If response times are seriously degraded and/or the introduction of multiprocessing causes performance to fluctuate erratically over time, it is likely that threads or processes are not getting exclusive access when modifying shared objects, or that deadlocks or incidents of indefinite waiting ("After you," "No, please, after you!") are occurring. Each of these problems has its own symptoms:

1. If mutual exclusion is an issue, the shared data may be corrupted. The corruption might trigger error checks and corrections. This might be verified by establishing whether the data has the intended values after the end of the test run. If it does not, mutual exclusion is likely to be the culprit. A thread safety problem is present.
2. If response times and resource utilizations are seen to fluctuate erratically, it is very likely that deadlocks are occurring and then being resolved by timeouts. When a deadlock occurs, the processor utilization attributable to the deadlocked threads drops to zero until a timeout mechanism breaks the deadlock, when processing will resume.
3. Indefinite waiting is a form of deadlock involving increased resource consumption. It is sometimes called *livelock*, because processes or threads attempt to communicate or to use resources repeatedly without making any progress, perhaps causing the starvation of other processes or resources as well.

Each of these concurrent programming issues is an indication of incorrect system function that cannot manifest itself in unit testing. Clearly, functional requirements cannot be met if shared data is corrupted. Throughput and response time requirements cannot be met if processes are blocking one another, whether via deadlock or livelock. Systems in deadlock or livelock cannot function correctly, either, because the threads or processes involved can neither progress nor make use of the data that is meant to be passed from one to the other. An example of livelock appears in Chapter 11. Livelock is discussed further in [HSH2005].

9.13 Planning Tests for System Stability

A performance test is a useful opportunity to monitor a system for signs of instability, because instability can often manifest itself in erratically fluctuating performance measures. A system whose average offered load is constant during an experiment should have constant average utilizations, queue lengths, throughputs, and response times during all intervals following a ramp-up period immediately after the arrival streams have been turned on and preceding a cool-down period after the arrival streams have been turned off. Deviations from this behavior are indicators of instability.

Instability may arise if there are memory leaks. Apart from causing increased paging, which will degrade system response time by increasing the utilization of the processors, the paging device, and the I/O and memory buses, a memory leak might cause a system to crash. If a system is subjected to a homogeneous workload with a constant arrival rate, a memory leak may manifest itself in global performance measures or in performance measures that pertain to one or more individual processes:

- *Evidence of a leak in global performance measurements.* The total memory occupied by the core images of all executing processes will increase steadily with time, with constant or nearly constant slope. In Windows-based systems, it will be seen that the number of committed bytes and the amount of occupied physical memory are increasing with constant slope. The amount of occupied physical memory is easily seen by bringing up the Task Manager after typing Ctrl-Alt-Delete. The committed byte rate is displayed as a number. The number of committed bytes and the percent of committed bytes are available in *perfmon.* In UNIX and Linux systems, the size of the occupied virtual address space is represented by the swap space size. The swap space size is obtainable from the *sar* (System Activity Reporter) and *vmstat* (virtual memory statistics) commands.
- *Evidence of a leak in per-process measurements.* In Windows-, UNIX-, and Linux-based systems, the memory sizes of leaking processes involved in the transactions will grow at a constant rate when subjected to a load with a constant arrival rate if they are leaking memory. In Windows, the memory size of each process can be seen in the Processes tab of the Task Manager or by

collecting the corresponding counter in *perfmon*. In UNIX- and Linux-based systems, the image size can be obtained for all processes at once by invoking the *ps* command with options *–elf*, or by using corresponding system calls. In Linux systems, the virtual image size can be obtained for a task with a given process ID (PID) by invoking the *pidstat* command.

Persistently increasing values of the average response times, queue lengths, and like quantities are also signs of instability, as are sudden unexplained drops or increases in the utilizations of any resource and in the response times of any kind of transaction.

9.14 Prospective Testing When Requirements Are Unspecified

It is possible to plan performance tests when the performance requirements are unspecified or poorly specified. This occurs when a prototypical system has been built without adequately identifying the target market, on speculation, or when the system or system component forms part of a general service whose performance requirements are not understood. Let us illustrate this with a service that is intended to support a transaction-oriented system. The first step is to identify a pilot range of offered transaction rates with which to drive a synthetic system load. Much can be learned about the extent to which the system can be scaled by running the system at some chosen rate λ. If the utilization of the bottleneck device is 90% or more or if the system is saturated, a performance test should be run at rates $\lambda/2$, $\lambda/4$, and so on until bottleneck utilizations well below saturation level are observed. The utilizations should all be plotted against the actual transaction rates to determine whether they are linear in the offered load. Average response times of the service should be plotted to determine the transaction rates that are just to the left and right of the knee in the response time curve, as illustrated in Figure 3.8. The maximum sustainable rate is the one that is just to the left of the knee, provided the associated utilization of the bottleneck resource is less than 80%. This is how we determine the range of sustainable loads as defined in Sections 5.6.3 and 6.2.

Where performance requirements are absent or poorly specified, it may be possible to synthesize the throughput requirements by developing a plausible synthetic workload characterization. This must be

done with extreme care. If the synthesized workload is much larger than will occur in practice, the system may appear to be unfit for service when subjected in testing even if it is actually adequate for the anticipated customer base. If the synthetic workload is too light, the performance test results might provide a false sense of security to stakeholders.

Once a synthetic workload has been devised, it may be useful to run it through some thought experiments to check that it is sensible. Two anecdotes illustrate this:

- The provider of a subscription-based service specified a workload for a customer care center that seemed exaggerated. The workload was based on the assumption that every subscriber would be generating calls at a particular rate. The figures implied that the average customer would have a complaint about the service once every two or three days. A service generating that kind of call volume and customer dissatisfaction would not be in business for very long. From this, one could conclude that the anticipated workload was grossly overestimated.
- The results of performance tests on an engineering tool intended to be accessible to a large user base indicated that the tool was unfit for service at the upper range of transaction rates used in the test but was adequate at the lower rates. The test results showed severe overload of software resources such as buffers and object pools. Project cancellation (which would have resulted in the loss of many jobs) was contemplated until further investigation showed that the heavier workload was unrealistic. The system was made available to a small user base without difficulties. Observations and measurements during production yielded insights leading to better design choices in subsequent versions of the system, allowing the expansion of the user base.

The lesson to be drawn from the first example is that it is possible to exaggerate the extent of the planned workload and incur unnecessary costs by oversizing the system. The lesson to be drawn from the second example is that the workloads used in performance testing should be based on a clear understanding of the initial user base, so that poor performance test results at excessive loads do not scare management into making inappropriate decisions about what to do next.

9.15 Structuring the Test Environment to Reflect the Scalability of the Architecture

Building a test environment that is comparable in scale to that of the production environment could be prohibitively expensive. If the test environment is going to be on a smaller scale than the production environment, it is important to structure it so that the magnitude, distribution, and nature of the test load reflect the structure of the system and how it will be used in the field. One should ask how the system will be scaled in production, and whether the architecture of the test system is truly a scaled-down version of the architecture of the system in production.

When a system is scaled horizontally (sometimes called *scaling out*), the load is distributed among several hosts. These hosts could be performing identical functions. This is particularly easy to do if each host will serve a distinct set of customers or jobs that might be operating on disjoint sets of data. An example of this would be a telephone network in which each switch serves a particular region, and all regions are expected to have similar traffic characteristics. A system is scaled vertically if functions that originally ran inside one host are distributed among multiple hosts. A three-tier web-based system that can be supported on one host when it has a very small customer base could be expanded to support a large customer base by both vertical and horizontal scaling. Vertical scaling occurs when the application layer, the web server layer, and the back-end database layer are moved into separate hosts. Horizontal scaling occurs if the application-layer hosts are replicated to increase capacity. Further horizontal scaling would occur if the web server layer is spread among multiple hosts as well. By contrast, database systems cannot always be easily replicated and deployed on multiple servers. A database that cannot be scaled horizontally must be deployed on a more powerful host with larger amounts of processing power, memory, and disk storage to accommodate growing query rates and amounts of data. The host must be *scaled up* rather than being *scaled out* or scaled vertically [DGLS1999, quoted in MenasceAlmeida2000, p. 99].

The architecture of a scaled-down system should enable the kinds of performance issues to occur that might be anticipated in the field. It should be a microcosm of the production system. A small-scale version of a multitier system that is implemented in a single host with only one

processor will not be able to capture the impacts of various factors affecting performance and scalability. For example:

1. Deploying all tiers of a multitier system on one host will mask the potential impact of network contention and delays.
2. Deploying all tiers of a multitier system on one host will obscure the capability of a multitier system to exploit the possibility of overlapping activities on multiple hosts.
3. If the test host has only one processor, it will not be possible for the test environment to reflect the possible benefits or pitfalls of having a dedicated multiprocessor/multicore environment for each tier in which multiple threads could execute simultaneously.
4. Deployment on a single tier could mask or exacerbate the impact of having an insufficient number of logical connections between different layers, such as JDBC connections between a Java-based application layer and a back-end database. Deployment on a single tier could also mask the effect of having transient or persistent connections between the tiers. An insufficient number of persistent connections can be the cause of a software bottleneck, while a reliance on transient connections could increase response times.
5. If a back-end database is replicated in production, deploying replicates on a single host will not capture the costs of replicating it across a network. Not replicating the database at all will mask the complexities of replication while possibly exacerbating contention issues.

These examples illustrate only a few of the ways that a scaled-down test system could fail to reflect the behavior of a production system. It follows that the configuration of a lab environment for performance testing should be chosen with care.

9.16 Data Collection

When measuring a system to evaluate its performance, it is useful to track how system performance and resource usage measures evolve over time. As we discussed in Chapter 8, the collection of measurements uses the very resources one is trying to measure. For measurements of

systems in production, it is usually sufficient to collect performance data in successive intervals that are 1 or 5 minutes long. Since performance testing is used not only for performance evaluation but to detect anomalies, average measures should be collected in consecutive time intervals of 10 or 15 seconds. The tools that are used to do this should allow the time interval to be configured. Since systems are usually spread across multiple hosts, one should ensure that the clocks on all systems to be measured are set to the same time to the extent that the use of time servers enables this. They should also be set to the same time zone. This will make the analysis of the data much easier. The author's preference is to use GMT or UTC as the time zone, especially in organizations whose hosts and/or load drivers are located in more than one time zone.

9.17 Data Reduction and Presentation

To thoroughly understand how a system is performing and how its various components interact, it is essential to look at the evolution of performance measures and resource usage measures over time. In stable systems, if the average offered load is constant over time, the average performance measures should also be constant. Under these circumstances, trends and oscillations, especially oscillations with wildly varying amplitudes and over wildly varying ranges, are indicators of problems in the system that could potentially reduce its capacity over time or eventually lead to a system crash.

For sizing purposes, however, we also need to look at the average resource usage and performance over time intervals in which the demand, usually in the form of transaction rates, is kept constant. These intervals must be long enough to allow sufficient transactions to be measured for the results to be meaningful. When a load is first applied to an idle system, there is a ramp-up period during which the performance measures increase until they reach an equilibrium level, just as an aircraft on a long journey climbs to cruising altitude. When the load is turned off, resource utilizations should decrease in what might be called a ramp-down or cool-down period of indeterminate or even zero length. If they do not, something has gone wrong in the system! To prevent distortions, one should exclude data from the ramp-up and ramp-down periods from the calculations of the time-averaged performance and resource usage measures. We will illustrate these ideas in the first example in the next section.

9.18 Interpreting the Test Results

9.18.1 Preliminaries

Our approach to analyzing performance data is based upon the premise that a well-behaved system will exhibit characteristics predicted by performance models. Thus, utilizations should be linear functions of the offered transaction rate, and the values of average performance measures should be constant if the average offered load is constant in successive measurement intervals and no resource is saturated. The last two criteria approximately correspond to two of the three necessary conditions for a Markov chain to be in equilibrium, and hence they approximate two of the four conditions for average performance measures to exist. The other two conditions are that the Markov chain be aperiodic and that the Markov chain be irreducible, that is, that every state in the chain should be reachable from every other state in a finite number of transitions. If the arrival process is periodic over any time scale, or any other system property such as the service rate or the number of available servers is periodic, the periodicity should be reflected in the values of the average performance measures. Conversely, if the performance measurements exhibit periodicity, the cause should be established.

9.18.2 Example: Services Use Cases

Here, we consider two services provided by a service-oriented architecture. Both services are invoked at very high rates. In this example, we shall see that the performance of one is healthy and that the performance of the other is not. Figure 9.2 shows the utilizations of the healthy service as a function of the offered transaction rate. The actual throughput is plotted against the right-hand vertical axis, and the measured CPU utilization is plotted against the left-hand vertical axis. The CPU utilization increases as a perfect straight line from 100 TPS to 300 TPS, and then has a sharp corner with a shallower slope from 300 TPS to 400 TPS. The reason for this is that the CPU utilization at 300 TPS is 80%. The offered throughput from 300 TPS to 400 TPS is an increase of 1/3, which would result in an infeasible CPU utilization of (4/3) $\times$ 80%~106.7%. We see a measured utilization of about 99% instead, together with a measured throughput that falls below the target throughput depicted by the line $y = x$. The utilization graph has a sharp change in slope from 300 TPS to 400 TPS. Hence, when fitting a regression line to the utilization graph to estimate the CPU demand per

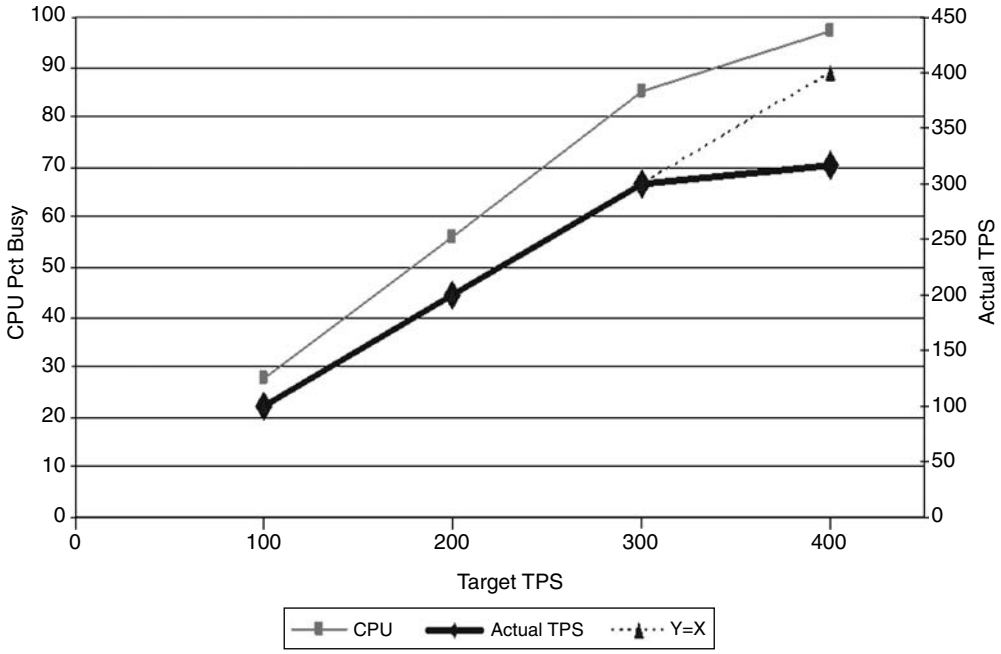


Figure 9.2 *CPU utilization (left axis) and throughput (right axis) of a healthy system versus offered throughput*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305–322. New York: Springer, 2012.

transaction, we should include only the observations for the arrival rates 100, 200, and 300 TPS.

This system appears to be well behaved. Good behavior is indicated by the slow increase in average response time as a function of offered transactions per second shown in Figure 9.3, and the flat utilization and response time curves as functions of time shown in Figures 9.4 and 9.5. This service has a healthy performance profile.

Let us now look at a different service with a much higher transaction rate. Figure 9.6 shows that the CPU utilization peaks at about 45% as the load on the system is increased beyond 1,500 TPS, and then drops off slightly rather than increasing linearly as the arrival rate increases to 2,000 TPS and beyond. The average response time starts rising sharply at 1,500 TPS also, as shown in Figure 9.7. We also see that the throughput levels off below 1,500 TPS even as the arrival rate is increased to 2,500 TPS. When looking at the average utilizations and response times with respect to time

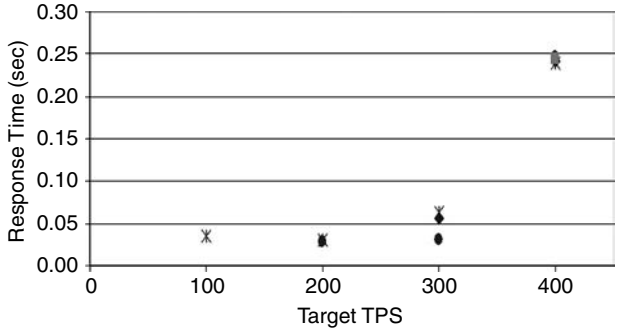


Figure 9.3 *Transaction response time of the healthy system versus offered throughput*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305–322. New York: Springer, 2012.

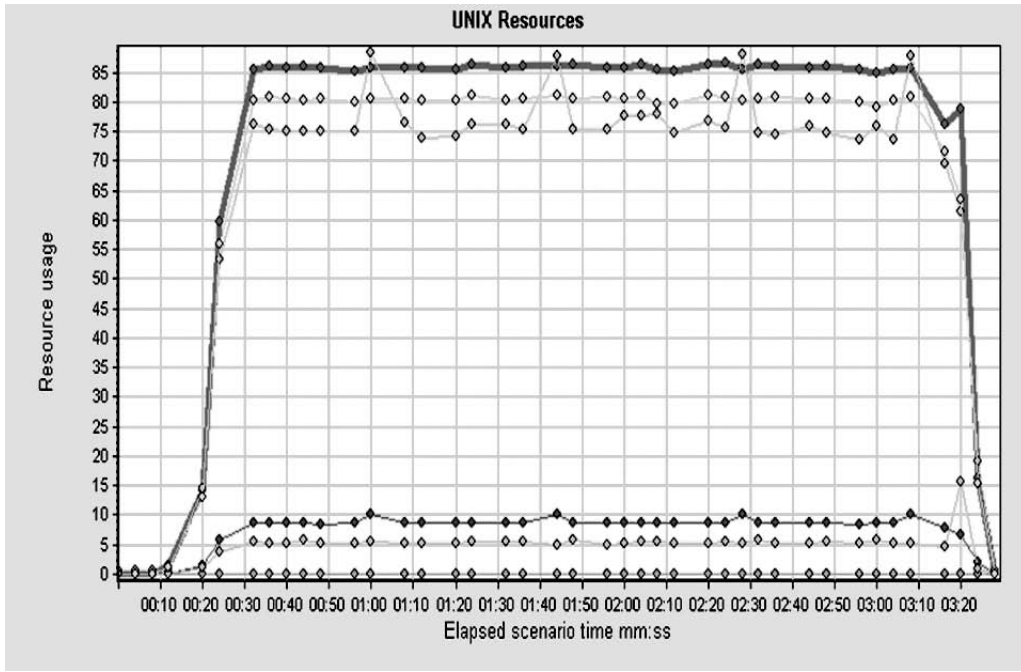


Figure 9.4 *CPU utilization versus time for the healthy system—offered throughput 300 transactions per second*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305–322. New York: Springer, 2012.

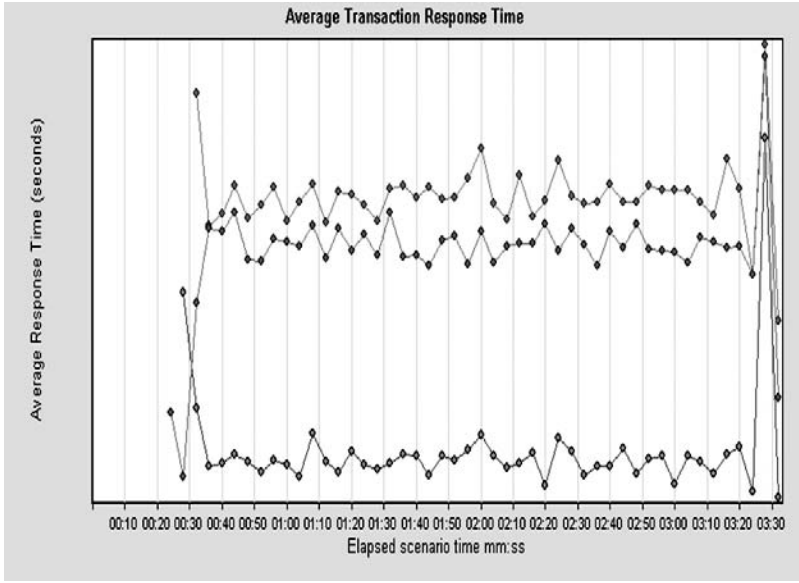


Figure 9.5 *Average response time versus time for the healthy system—offered throughput 300 transactions per second*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305–322. New York: Springer, 2012.

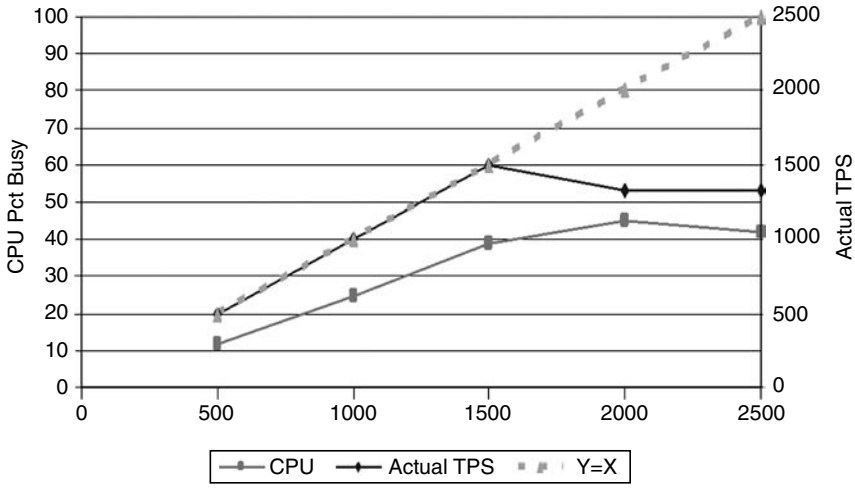


Figure 9.6 *Throughput and CPU utilization of an unhealthy system*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305–322. New York: Springer, 2012.

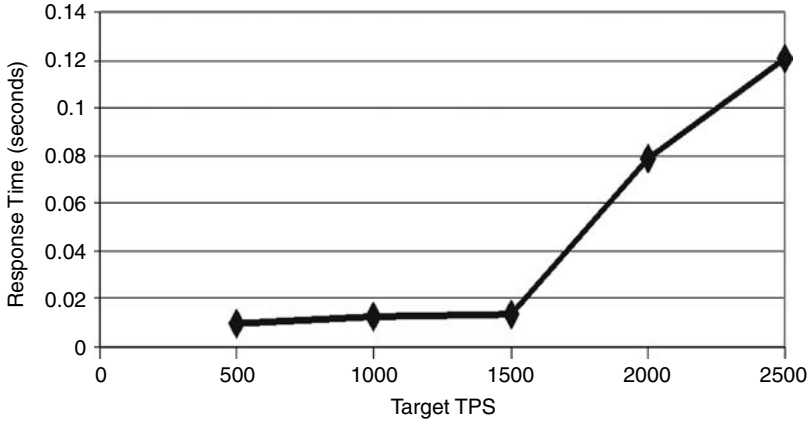


Figure 9.7 *Average response time versus transactions per second*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305–322. New York: Springer, 2012.

when 2,000 TPS are applied, we see that utilization is declining slightly with time as shown in Figure 9.8, while the average response time oscillates wildly with amplitude that increases over time, as shown in Figure 9.9. Moreover, the average CPU utilization for 2,000 TPS in Figure 9.8 is lower than the measured utilization for 1,500 TPS shown in Figure 9.7.

Taken together, these observations are signs of a concurrent programming problem that manifests itself once the arrival rate exceeds 1,500 TPS. Since the CPU is the bottleneck and its utilization is linear in the arrival rate up to this level, with a value of approximately 40%, we surmise that the service could be provided at 2,000 TPS on this platform without difficulty in the absence of the concurrent programming problem. The system would have to be restricted to 1,500 TPS for the system to be operable as implemented, but that would be hazardous, since concurrent programming bugs tend to occur nondeterministically. On investigation, a segment of Java code was found in which the *wait*, *notify*, and *notify_all* operations were being used incorrectly. Replacing this piece of code resulted in more regular performance.

9.18.3 Example: Transaction System with High Failure Rate

In this example, the number of load generators sending transactions to a system is increased at regular intervals, so that the graph of the number of active load generators with respect to time describes a staircase. All load generators are sending identical transactions to the system at the

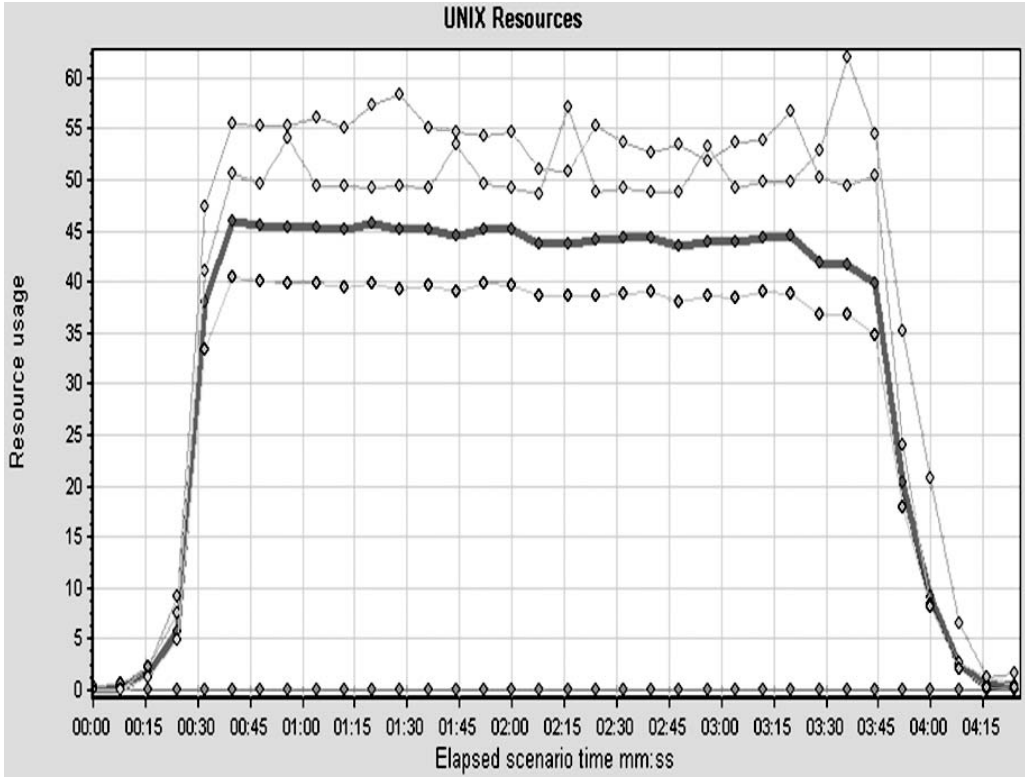


Figure 9.8 *CPU utilization versus time for the unhealthy system, with an offered throughput of 2,000 TPS*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305-322. New York: Springer, 2012.

same rate. The top plot in Figure 9.10 shows the number of load generators (also known as *virtual clients*) as a function of time. In the bottom figure, the upper plot shows the transaction completion rates and the lower plot shows the transaction failure rates as functions of time. It appears that this system is either grossly saturated or dysfunctional, because the number of failed transactions is equal to the number of completed transactions from time to time. Notice also that there is a spike in the rate at which transactions are completed sometime after the introduction of each load generator. A performance test with this sort of result raises questions about whether the throughput requirements of the system and hence the test cases were wisely chosen. It turns out that the code also suffered from thread safety issues. This caused data to be corrupted, contributing to the volume of failed transactions.

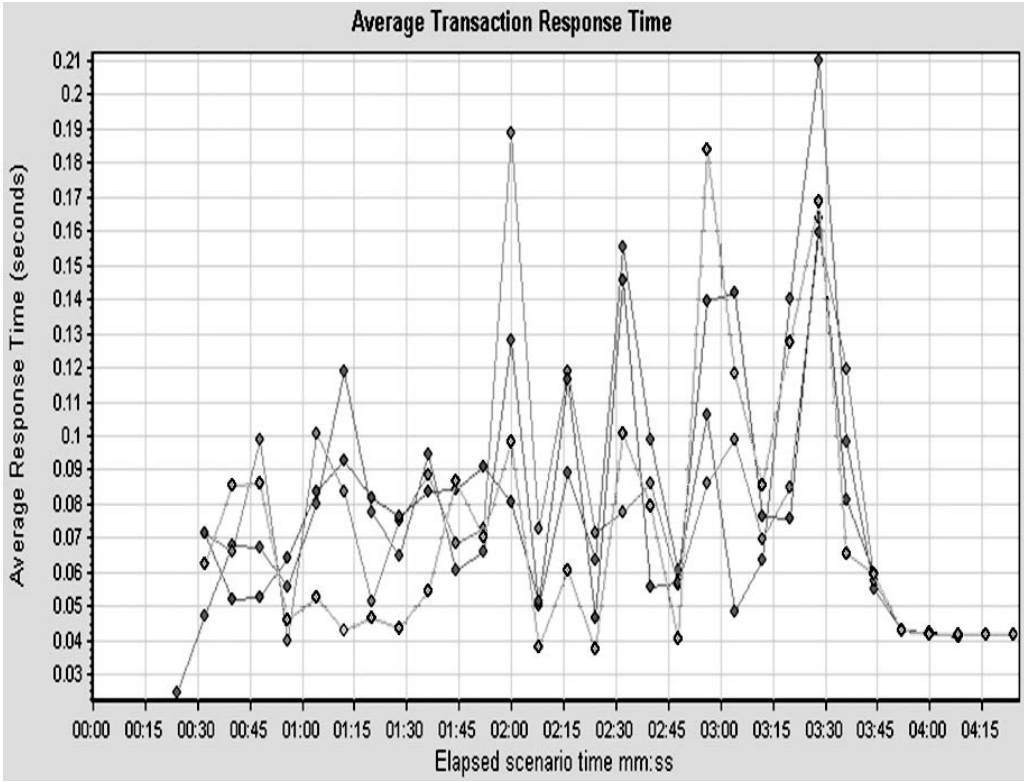


Figure 9.9 *Average response time versus time for the unhealthy system, offered throughput 2,000 TPS*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305-322. New York: Springer, 2012.

9.18.4 Example: A System with Computationally Intense Transactions

Consider a transaction system in which the work items consist of computationally intense transactions. These could be requests to process images as they arrive. Such systems might be used in the oil industry or in medical applications. Testing the performance of this system required the preparation of a large set of representative images, which were submitted for processing to the system under test at fixed intervals. The times between submissions were configurable so that the effects of different arrival rates could be measured. The performance

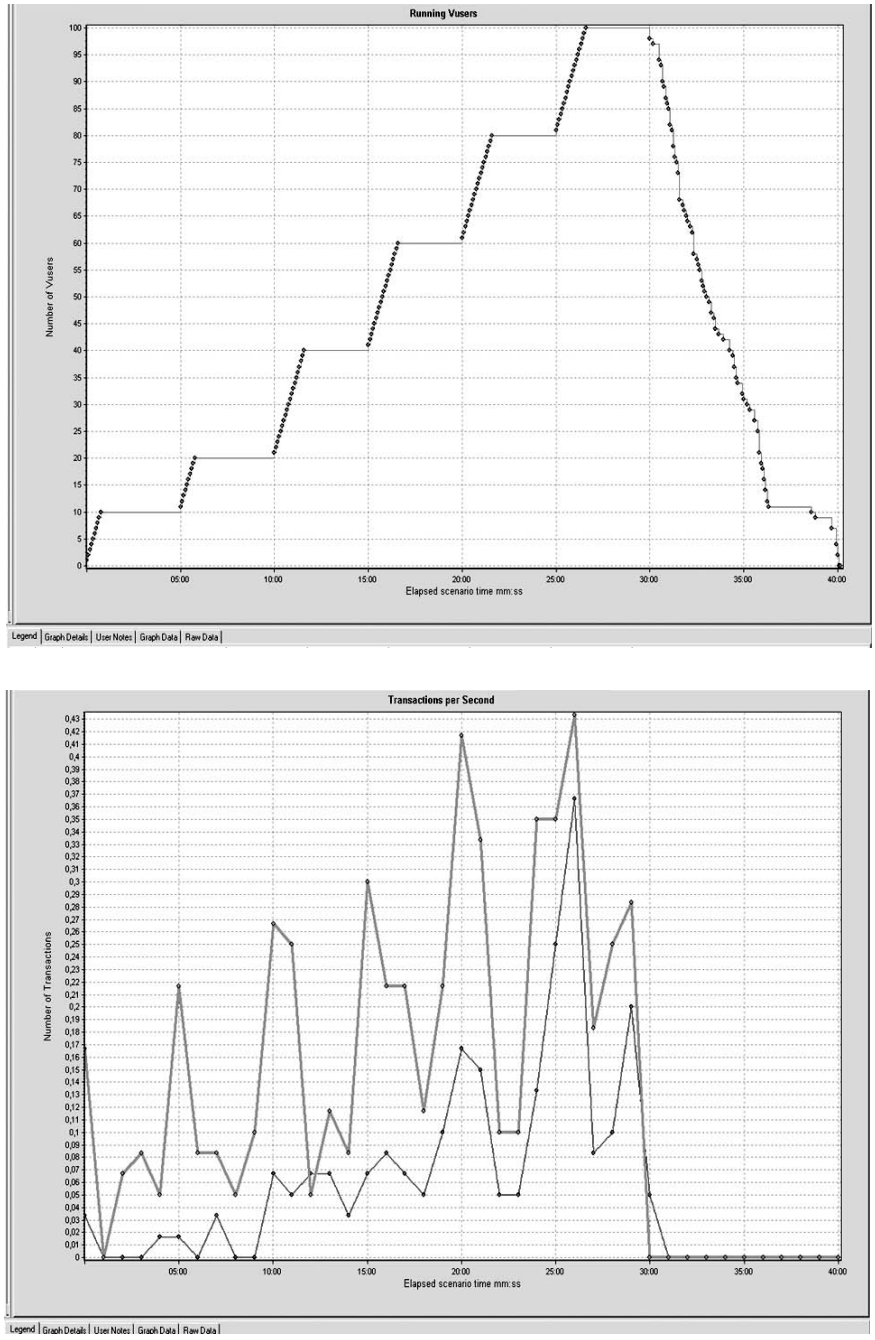


Figure 9.10 *Transaction systems with high failure rate. Top: Number of virtual users versus time. Bottom: Transaction pass rate is the thicker line and transaction rate failure is the thinner line.*

measure of interest was the time from request submission to request completion. The usage of system resources was measured. A performance model similar to the open queueing model described in Chapter 3 was derived from the measurements to allow performance prediction. The modeling assumptions ignored two key features of the system: the use of multiple processors and the use of asynchronous I/O. Asynchronous I/O allows work on the CPUs to proceed even if I/O has not been completed. Measurements of the system showed that the I/O subsystem was the bottleneck. Toward the end of some runs, processor utilizations varied slightly from the average for reasons that the system architect was able to explain. The following is a summary of the results of the performance tests and the modeling effort:

- The average utilizations for each run were linear with respect to the offered transaction rate. The User_Disk is the system bottleneck. It has the highest utilization. This is illustrated in Figure 9.11. This was also predicted by the performance model.
- The CPU load was spread somewhat uniformly among the processor cores, but not exactly so. Still, the load among the cores was not badly skewed. This is illustrated in Figure 9.12.
- Plots of the resource utilizations over time indicated that they were flat under constant load for the most part.
- Average response times were also constant over time, as desired.
- Extrapolating from the highest utilization curve, we see that the maximum achievable throughput of this system with this sort of offered work is 4 normalized work units per second. At that throughput, the utilization of the bottleneck device is 90%. Higher loads would saturate the bottleneck device.
- The performance model yielded pessimistic predictions about the system response time as a function of the offered transaction rate. This is illustrated in Figure 9.13. This may well be because the equation for the global response time (equation (3.24) in Chapter 3) is based on the assumption that processing and I/O for a particular job do not occur at the same time, while the system under test used asynchronous I/O, which allows some processing and I/O to occur for the same job simultaneously. In particular, the measured average response time at the lightest load was less than the predicted response time, even though the load was high enough for a modest amount of contention in the system. Another possible cause of error is that our simple Mean

Value Analysis tool is based on the assumption that there was only one processor. The system under test had two processors, each with four cores, each running at a slower speed than we used for the representation of the single processor in our model. It appears that asynchronous I/O was having the desired impact on performance. Notice that using asynchronous I/O does not increase the maximum achievable throughput because it does not change the utilizations of the devices for a given offered load; it only changes the total time to complete work by allowing processing and I/O to be overlapped for each job.

This example illustrates the following points:

- A performance model can be used to understand the throughput limits, based on the linearity of measured resource utilizations as a function of the offered load.
- Since processing is observed to be spread fairly evenly among the processor cores, the capacity of the system is not artificially limited by the concentration of processing demand on one of them. If the user I/O device were not the bottleneck, the

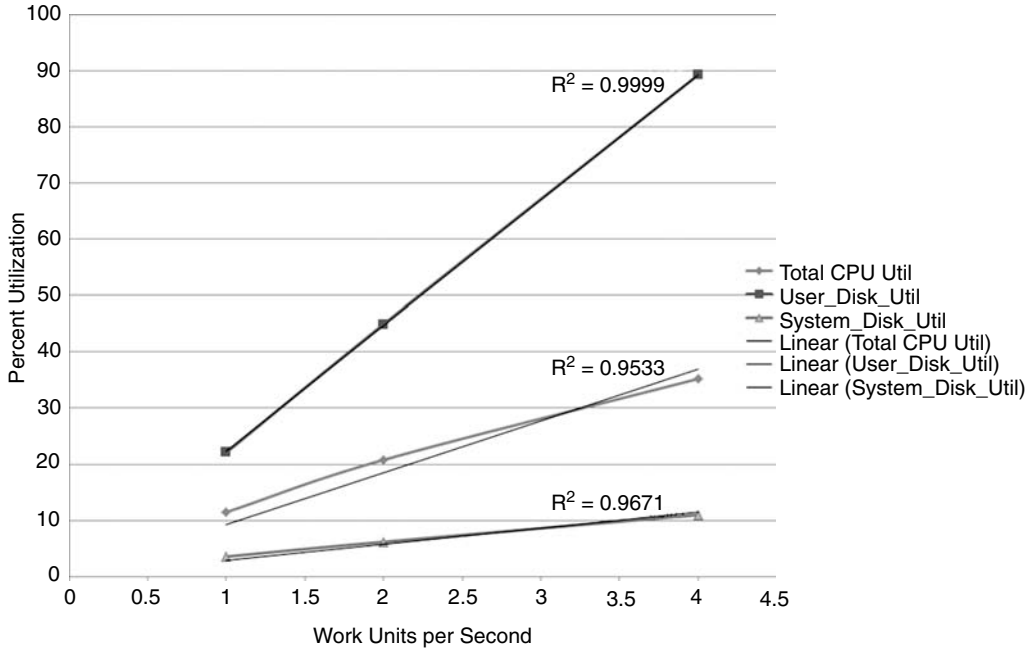


Figure 9.11 *Computationally intense transactions: average processor, user I/O, and system I/O utilizations with regression lines*

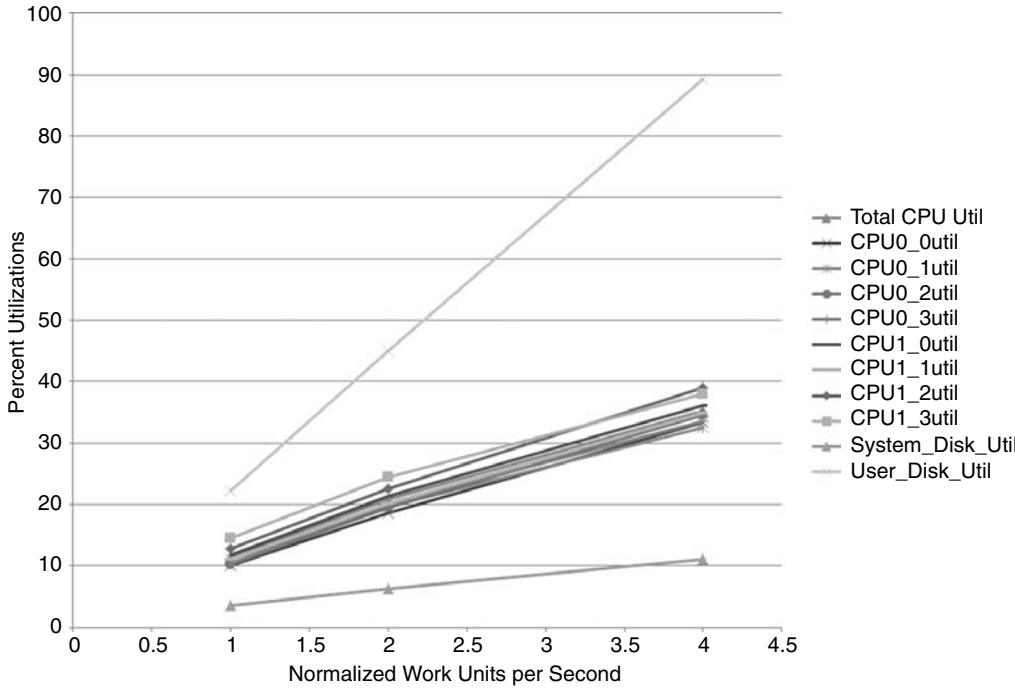


Figure 9.12 *System with computationally intense transactions: utilization of all devices, including individual processor cores*

concentration of demand on one processor would prevent the others from being used and wastefully limit the ability of the system to handle increased loads.

- Informative results can be obtained from a coarse performance model to understand system features (such as asynchronous I/O) and capacity limitations. In particular, the performance model correctly predicted response time trends, including the effect of increasing load on response time, even though it did not capture the reduction in average response time attributable to the use of asynchronous I/O or the effect of parallel processors.

9.18.5 Example: System Exhibiting Memory Leak and Deadlocks

A memory leak can eventually cause a system to halt abruptly unless the process that is leaking is stopped and restarted. In [AvBon2012],

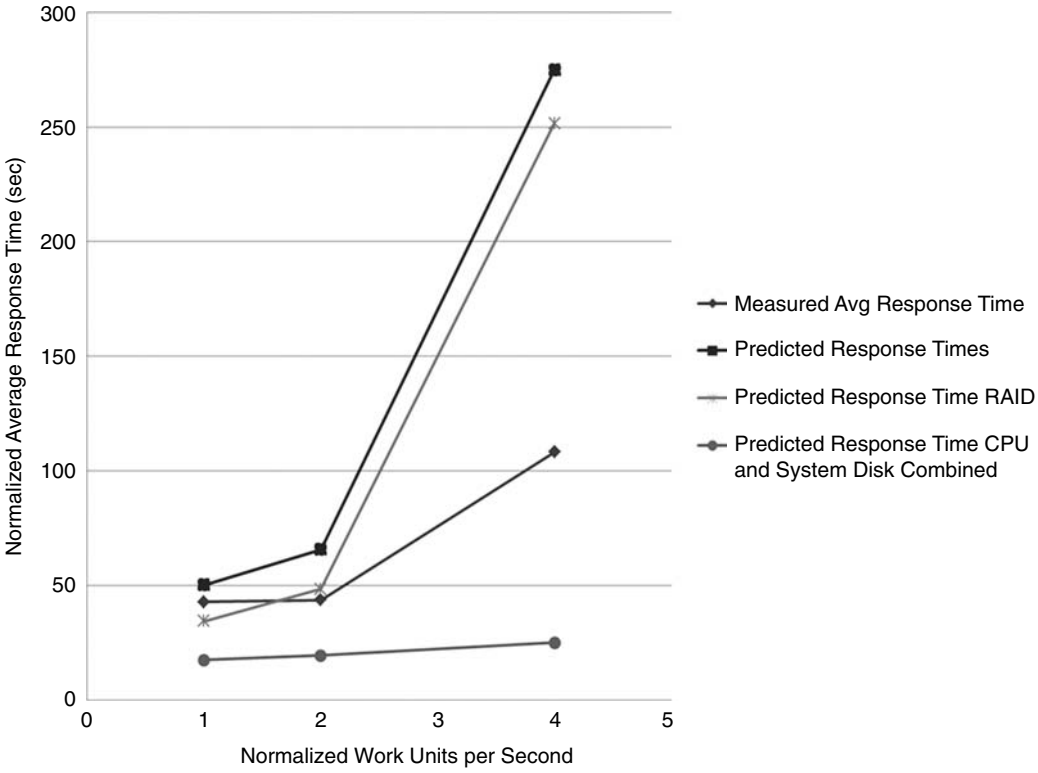


Figure 9.13 *Response times for the computationally intense transactions*

performance data displayed by the Task Manager of a Windows XP–based system indicates that the committed byte rate is increasing over time, while the CPU utilization occasionally drops sharply and then rebounds. The increasing committed byte rate indicates that there is a memory leak, and the fluctuations in CPU utilization are signs of deadlock being resolved by a timeout. A snapshot of the Task Manager display is shown in Figure 9.14. An investigation identified the causes of both the leak and the recurrence of deadlock. Both were remedied, and the test case was repeated on the modified system. The results of the repeated test are shown in Figure 9.15. The latter plots show that the committed byte rate is constant, while the CPU utilization oscillates within a very narrow range about a horizontal line. This line is lower than the average and peak CPU utilizations prior to the remedy being implemented. A comparison of the figures also shows a reduction in I/O activity. Thus, remedying a memory leak and the recurrence of deadlock led to a reduction in resource utilizations and the increased stability of the system at the same time. This is not unusual in our experience.

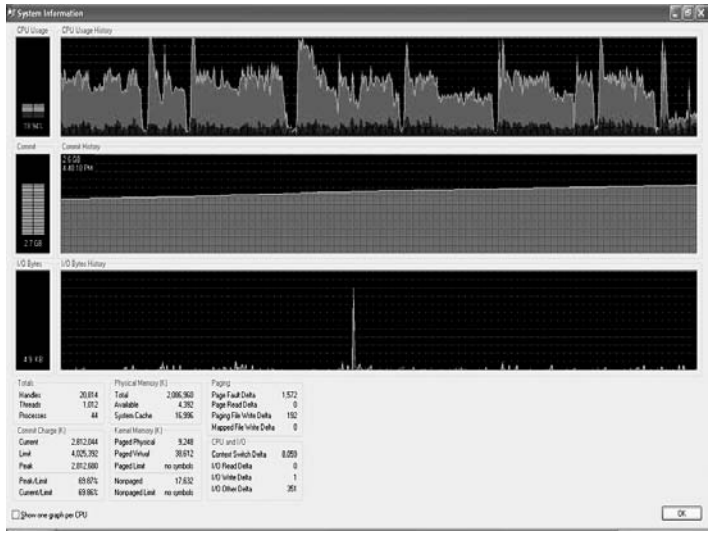


Figure 9.14 *Task Manager performance display showing signs of a memory leak and repeated deadlocks*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305–322. New York: Springer, 2012.

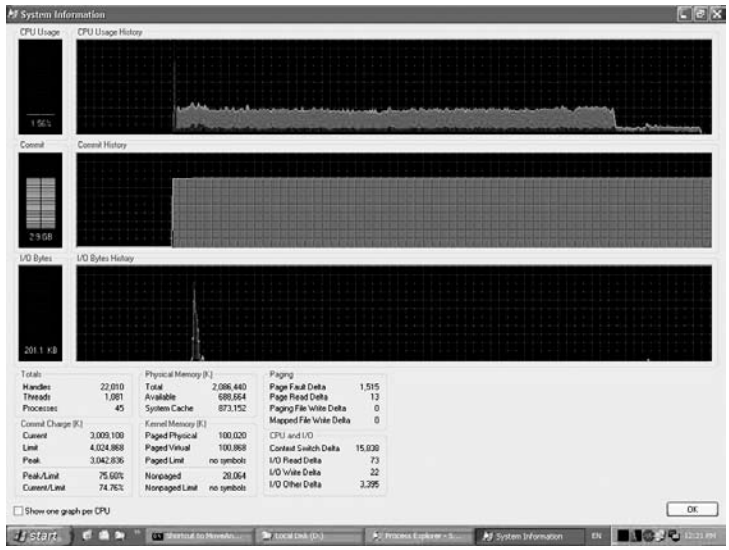


Figure 9.15 *The same system as in Figure 9.14 after remedies were applied*

Source: [AvBon2012] With kind permission from Springer Science+Business Media: Avritzer, A., and Bondi, A. B. "Resilience Assessment Based on Performance." In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, 305–322. New York: Springer, 2012.

9.19 Automating Performance Tests and the Analysis of the Outputs

One benefit of performance test automation is that it frees the performance testing team from the need to repeatedly carry out tedious manual testing tasks that could be prone to error. By automating aspects of a performance test such as system configuration, test parameter setting, verifying that the system is empty and idle, data logging, and verifying whether the desired values of performance measures fall into specific ranges, the performance testing team and the performance engineers are given a verifiable way of ensuring that tests have been carried out as intended. Thus, automated performance testing ensures that the playbook has been executed properly. Moreover, automation enables the repetition of performance tests under verifiable conditions after changes have been made to the system in a controlled manner. Finally, test automation provides an audit trail for the configuration, execution, and logging of the performance test and the test results. Notice that automated performance testing is not a panacea: it is merely an automated implementation of a testing procedure that could be executed manually.

The act of writing the program to automate the playbook provides impetus and opportunities for correcting any ambiguities the playbook might contain. The same holds for the automated verification of the pre- and postconditions of the test.

The conduct of performance tests may be managed with a commercial test case manager. When a commercial test case manager is either unavailable or unsuitable for the application at hand, scripts may be written to drive the execution of successive test cases. The testing team may be motivated to do this by its early experience of conducting pilot performance tests manually [BondiRos2009]. Indeed, it may be worthwhile for the team to execute pilot performance tests manually so as to acquire a solid acquaintance with the system under test and with the environment used for load generation.

When automating the analysis of the outputs, one should first consider the characteristics of the outputs that are desirable. If detection of undesirable characteristics and the verification of desirable characteristics can be done automatically, the undesirable characteristics should be flagged so that the performance engineering team can focus on identifying and remedying their causes. This is especially useful

when testing applications that involve large numbers of hosts, such as a distributed network of servers deployed at the stops of a public transit system. Based on the examples of test results we have presented here, we can identify basic characteristics that might be detected automatically. The following desirable characteristics apply to systems to which a load is applied with an average arrival rate that is constant from one measurement interval to the next:

1. After the load has ramped up to its constant level and before it ramps down at the end of the test, all measured average resource utilizations and average response times should be constant or vary within a very narrow range. No series of measurement data should contain any trends. The graphs of all averages of performance measures should be horizontal between ramp-up and ramp-down.
2. Oscillations in resource utilizations, memory occupancy, and response times are signs of anomalous behavior such as deadlocks resolved by timeouts and should be avoided.
3. The transaction completion rate and the transaction arrival rate should be equal throughout each test run. They should be constant after ramp-up and before ramp-down. If the completion rate is less than the arrival rate, one of the following is likely to be occurring:
    a. Jobs are being lost.
    b. Jobs are backing up in a queue for a saturated resource such as an I/O device or processor.
    c. Jobs are backing up for a discrete resource that has been exhausted, such as an object pool or memory pool.
4. Before the load is applied, resource usage within the system should be constant, and preferably zero.
5. Average response times should fall below the levels specified in the performance requirements at all loads.
6. The average utilizations of all resources computed after ramp-up and before ramp-down such as disks, processors, and network bandwidth should be linear functions of the offered load provided no component of the system is saturated.

Any deviations from these characteristics should be flagged and investigated.

9.20 Summary

In this chapter we have covered a wide range of topics concerning performance testing. While some of the suggested performance testing practices may appear to be mundane or obvious, we have recommended them because our experience has shown that failure to adhere to them seriously diminishes the value of the performance test results. We have shown how performance testing that is structured to verify that the conformance of a system to properties predicted by rudimentary performance models yields essential information about the ability of the system to handle increased offered loads. Among these properties are constant values of average performance measures and resource usage measures under constant average offered loads and linearity of the resource utilizations with respect to the average offered loads. We have described how test beds should be structured to reflect the architecture of the target production system and explained how performance measurements should be collected in a clean environment to ensure that they reflect resource usage by the system under test alone. We have also illustrated how the results of performance tests can be used to identify concurrent programming issues and software bottlenecks and have related these results to the predictions of standard performance models.

9.21 Exercises

9.1. As passengers' luggage passes through an airport conveyor system, the bar codes on luggage tags are read by a scanner that sends the unique identifier of each tag to a database for instructions on how to route the suitcase. The average suitcase passes ten such bar code scanners during its passage through the airport. The luggage belt moves at 2 meters per second, and suitcases are positioned so that the handles bearing the tags are 1 meter apart. There is a database of this kind in each airport. A suitcase is registered in the database either when it is checked in by a passenger departing from the airport or when the manifest of an arriving plane is downloaded into the database. An arriving suitcase may be routed straight to baggage claim or to the loading bins destined for connecting flights. The following is the set of use cases you are asked to assess by

performance testing: (i) the recording of a newly checked-in suitcase in the database, (ii) the querying of the routing database for routing instructions as a suitcase approaches a scanner, (iii) the recording of all suitcases in the manifest of an arriving aircraft.

(a) Explain how you would test the ability of the database to handle queries at various loads associated with each use case being executed in isolation and with all three of the use cases being executed concurrently.

(b) Identify the variables that must be varied from test run to test run to assess the ability of the routing database to respond to queries in a timely manner.

(c) Identify the points in the transaction at which the response time would be measured.

(d) What operating procedures must you consider when formulating the tests? What performance requirements must you consider?

(e) If some of those performance requirements are missing or inconsistently specified, what operational considerations at the airport will you consider to link the test results to domain-specific business and engineering needs?

(f) Explain how you would determine the maximum number of suitcases per hour that could be handled by the routing database. What travel patterns and characteristics of aircraft might determine this?

(g) Describe the set of performance measurements and resource usage measurements you would collect in the course of the test. You may assume that the database is hosted on Windows platforms. (*Hint:* What performance measurements relate specifically to the problem at hand?)

9.2. In Exercise 3.4 we saw the following table of resource demands for a computer system with thinking terminals:

Device Name	Visit Ratio	Service Time (sec)	Service Discipline
CPU	6.0	0.0090	PS
Disk 0	1.0	0.0400	FCFS
Disk 1	4.0	0.0250	FCFS
Thinking terminals	1.0	4.0	IS

(a) Suppose that there are no thinking terminals, and that these are the parameters of an open system in which transactions arrive asynchronously, without awaiting a system response. What maximum throughput should be offered to the system by the load drivers? Explain your results using Mean Value Analysis or otherwise.

(b) Suppose instead that we wish to measure the effect of varying the think time between 0 and 8 seconds. What is the maximum number of logged-in terminals the system can sustain with think times of 0, 4, and 8 seconds? If you wish to double the number of sampled response times obtained when there are four terminals logged in (e.g., so as to be able to have narrower confidence bounds), what combination of think time and number of terminals would you use? Explain your results and justify your choice using Mean Value Analysis or otherwise.

9.3. Performance measurements displayed by the Windows XP Task Manager show that both processors in a dual-core system are 50% busy even when the system is empty and idle before load has been applied. Clicking on the Processes tab shows that there is a polling process whose processor utilization is 100%. The polling process is intended to check if an arriving message queue is nonempty, and then forward any waiting messages to an application process for handling. An examination of the design document reveals that the polling process should repeatedly check the queue to see if a message has arrived, and then forward it for processing elsewhere. The specification contains no statement about how often polling must occur or about how the polling process should behave as long as the queue is empty. Neither does the corresponding functional requirement.

(a) Explain why the displayed CPU utilization of a single process is 100% while the utilizations of both processors are only 50%. What feature of the operating system might make this possible?

(b) What desirable attributes appear to be missing from the functional requirement for the polling process?

(c) Propose a design change to the polling process that will fix the CPU utilization problem and does not impose a theoretical constraint on the maximum polling rate. That is,

there should be no theoretical limitation on the number of times a poll can take place in unit time.

(d) Propose a performance test plan that you will use to check that the remedy works as desired once it has been implemented.

(e) Explain the undesirable impact on the function, capacity, and performance of the system if the problem is not urgently fixed. Also, explain the impact on the performance testing schedule and the time to market of the system. *Note:* Only an explanation in general terms is required.

(f) Identify the list of stakeholders who must be persuaded that the fix should be made, and the list of those who must make the change, bearing your organization's change management process in mind. (*Hint:* It might not be just the developers.)

This page intentionally left blank

Chapter 10

System Understanding, Model Choice, and Validation

To evaluate a system's performance, one must first acquire an understanding of its desired function and of the paths followed by the information that flows through it. The performance of a complex system may best be understood by breaking it down into components whose performance can be engineered separately, and then combining the resulting models into a larger one so that the entire system can be modeled as a whole. This entails understanding the mission of the system, the system's architecture, the information flow through the system, and sometimes the flow of transported entities. At the same time, one needs to obtain both a qualitative and a quantitative feel for the traffic demands and the performance requirements, because these are, or should be, the drivers of the need for capacity and the design and implementation choices made to support them. If a performance model turns out to be inaccurate, the reasons for the inaccuracy should be investigated so that the limitations of the model can be understood.

10.1 Overview

In this chapter we shall study how one goes about modeling a computer system. This entails not only identifying the servers and the parameters of a queueing model as suggested in Chapter 3. To begin with, one must identify the questions to be answered by the model and identify the salient aspects of the system's structure and function. One should identify those portions of the system that could be modeled separately and determine the level of detail to be captured in each component model.

It is not always advantageous to build a detailed model incorporating all facets of the system. A highly detailed model may not be needed to answer basic questions about the capacity of the system or about the effectiveness of a design choice from a performance standpoint. It is often sufficient to focus one's modeling efforts on the foci of load and the principal factors determining capacity and response time. The more detail is captured in the model, the more parameters are required to evaluate it. The values of the parameters may not always be obtainable, and using incorrect values may introduce errors in the performance predictions that might not have occurred with a less detailed model. Moreover, a detailed model may include a considerable amount of information about the state of the system. In general, the more dimensions are used to describe the state of the system, the more computationally expensive the evaluation of the system performance might be. If a question arises about the influence of a detail of system design such as buffer size or the use of a scheduling rule, it may be preferable to address the question in isolation in a separate model. The results of that model can then be used to address the impact of the detail on the system as a whole. For example, if it is determined that a scheduling rule could cause the system to go into deadlock and crash, it is best to address the performance impact of that scheduling rule on its own, and then determine what the performance of the system would be if a deadlock-free rule were used instead.

The application of performance engineering techniques often begins with questions about a system's functionality and the performance needs it must meet. The questions depend on the current status of the system and/or how it might be changed. For example:

- If one is building or procuring a system for the first time, there will be questions about the needed capacity, the desired response times, the functionality to be supported, the technology that is available to implement functional and

performance requirements, and constraints on implementation such as the need to interface with legacy systems or a requirement to use a particular operating system.

- If the system is in production and is perceived to be performing badly, one should ask questions about the reasons for poor performance and what might be done about it. Insights about the causes can be gained from constructing a model of the traffic offered to the system and conducting performance measurements of the system over a weeklong cycle. This allows one to assess performance and resource usage as the traffic varies.
- If one is adding functionality or load to an existing system, there will be questions about whether the existing functionality can be supported and the performance of and demands for the existing and desired functionalities.
- If one is contemplating the introduction or deployment of a new algorithm or technology to reduce response time or increase capacity, the modeler may be asked to predict whether the new technology will have the desired impact on performance before it is introduced.
- Similarly, if one is forced to migrate a system to a new environment, such as an operating system or a new family of disk drives, because the present one will no longer be supported, there will be questions about the ability of the new system to maintain present performance levels or provide better ones.

When formulating performance requirements, one is asking and addressing questions about how good the performance of the system must be and what kind of load it is supposed to carry. These requirements are drivers of the system architecture. The requirements and the architecture determine how the functionality and performance of the system are tested. In addition, requirements and architecture play a role in determining the kinds of questions to be answered by a performance modeling study and hence by the performance models used to carry it out.

In testing the performance of the system as described in Chapter 9, one devises and executes an experimental plan to determine whether performance requirements have been met. One also verifies that the performance and resource usage measures of a system meet requirements, obey basic performance laws, and conform to patterns that are consistent with those expected of stable, well-behaved systems when subjected to controlled loads. Performance models help us make

predictions about the effects of design choices and changes in load or technology, whether these concern systems that have not yet been built or changes to existing systems.

10.2 Phases of a Modeling Study

A modeling study may be thought of as consisting of four phases:

1. A *model development phase*, in which the essence of the information flow is captured and a corresponding queueing network model is mapped out, and special features of the system are identified that might have to be modeled separately. For example, asynchronous I/O is a special feature that we shall examine later in this chapter.
2. A *measurement phase*, in which the actual system is subjected to a controlled test load believed to be representative of what would be seen in production.
3. A *validation phase*, in which the parameters of the model are computed from resource usage measurements and the performance measures predicted by the model compared with the measured values.
4. A *projection phase*, in which one varies the parameters of the model to project the impacts of changes before they are made [LZGS1984].

The first three phases were illustrated in our examination of a computationally intensive transaction processing system in Chapter 9. Our view of the system was formulated in a model development phase. This was followed by a measurement phase, in which we chose measurements to be gathered in performance testing, and then used the measurement data as inputs to a simple performance model. In the validation phase, we found that the resource utilizations predicted by the model were identical to measured results. This was to be expected, because the test data showed that the system was stable and well behaved and that the resource utilizations were linear functions of the offered load. The service demands at each device were obtained by fitting regression lines through the measured utilizations as a function of the measured system throughput, and then calculated from the slopes of the regression lines using the Utilization Law. During the validation phase, it was seen that the predicted response times at all load levels were greater than the

measured ones. Despite that, the predicted response time plot and the measured response time plot had similar shapes, reflecting the onset of saturation as the offered load increased. The discrepancy was at least partially explained by the use of asynchronous I/O to allow I/O and processing of a transaction to occur simultaneously rather than serially.

In the projection phase, a validated model could be used as a baseline for answering questions about what would happen to system performance if transactions involved amounts of I/O activity or processing time that differed from those that were measured, because of changes in the nature of the work. It could also be used to model the effects of adding more I/O devices to ease the load on the existing ones, or the effects of adding faster processors.

Projecting the changed performance of the system is sometimes called what-if analysis, because one is addressing questions like "What if we add an I/O device?" or "What if we add a faster processor?" Because the baseline model we described in Chapter 9 does not capture the effect of asynchronous I/O, parallel CPUs, RAID (redundant array of inexpensive disks) devices, or priority scheduling of any kind, it should not be used to answer questions about the effects of changing them. Of course, the model will tell us something about the values of the response times in the absence of asynchronous I/O, since complete serialization of I/O and processing were assumed. Moreover, because utilizations depend only on processing time and arrival rate, and not on scheduling polices, the baseline model will also tell us whether adding load to the system will cause its capacity to be exceeded.

The inadequacies of the simple model illustrate a critical question often faced by modelers and performance engineers: Is the performance model that has been devised sufficient to address the concerns of the system's stakeholders, or are more accurate models that capture more system details needed? The answer to this question could depend on multiple factors, including the resources and time available to build more sophisticated models, the level of expertise that is available within the organization to address the associated modeling complexities and interpret the results, and the availability of sophisticated modeling tools and the availability of data needed to compute the values of modeling parameters. Purpose-built queuing network models of asynchronous I/O are described in [HeidelbergerTrivedi1982]. If a queueing system is not susceptible to modeling using queueing network models, the use of discrete event simulations may be appropriate [LawKelton1982].

In the remainder of this chapter we examine how one might go about modeling the performance of a fictitious conveyor system. We

then examine the limitations of a simple performance model of the computationally intense system whose measurements we saw in Chapter 9. We go on to discuss how exploring the limitations of a simple model might be used to determine what problem areas should be the subject of further modeling studies, and how the choice of these areas should be reconciled with limitations on the resources available to the clients who commission this work. Since clients' budgets and time to market are usually limited, it is important to focus one's performance engineering efforts where they are most likely to have the largest impact.

Finally, we discuss examples of system behaviors, such as thrashing in paged virtual memory and periodic behavior patterns, that are at odds with some of the assumptions underpinning the basic queueing models we presented in Chapter 3. Ignoring these characteristics can lead to inaccurate performance predictions.

10.3 Example: A Conveyor System

To illustrate how the performance model of a system can be broken into parts, let us consider a conveyor system as described in earlier chapters. It might have the following components involving computations:

- A parcel database that contains records of the origin, destination, and movements of every parcel. This database is queried by the programmable logic controllers (PLCs) to determine how a parcel should be routed on the conveyor belt.
- A set of PLCs that control the movement of parcels on the belt and provide alerts of emergencies such as overheated or jammed components and the pulling of an emergency cord to bring the belt to a stop.
- One or more local area networks to connect the PLCs to each other, to the routing database, and perhaps to a central control or monitoring station if there is one.
- Bar code readers to identify the parcels to the PLCs.
- A system for registering the intake of parcels, including such information as the origin, the destination, the sender, the receiver, all movements including intake and delivery, and any information about special handling such as the requirement for an adult signature on delivery, whether the parcel is fragile, or

whether it should be X-rayed or sniffed by dogs on behalf of law enforcement.

- A monitoring system to keep track of the status of the conveyor's parts, including PLCs, motors, temperature sensors for the motors, jam sensors, and sensors that detect when the red emergency cord has been pulled or emergency stop button pushed.
- Interfaces (human or otherwise) for parcel intake, delivery, and system monitoring.

The PLCs use a local area network to communicate with one another and to send queries to the parcel routing database. The responses to the queries tell the PLCs whether a parcel should be diverted or moved straight ahead.

The load drivers for the parcel routing database are parcel movements, parcel location queries generated by people, and the occurrence of parcel intake and parcel delivery events. The load drivers for the monitoring station are status messages sent by the PLCs, including alarms of any kind.

From the point of view of the PLCs and the systems for intake and delivery, the parcel routing database is a black box whose operational characteristics are the times taken to respond to queries and the contents of the queries. From the point of view of the database, the queries, intake registrations, and delivery registrations are streams of transactions to be processed according to whatever business logic is required. The operational characteristics of the local area network are the times taken to deliver messages.

For the purpose of this exercise, let us assume that the response time requirements of the parcel routing database have been specified, and that the sum of the peak rates at which parcels pass the bar code readers is known. Let us also assume that the message pattern between PLCs is known and that the rates at which PLCs generate queries to the parcel routing database at various times of day are also known. From the standpoint of the PLCs, the query response time consists of the sum of the database response time and the network delivery time of the message carrying the response. The requirement for an upper bound on the value of this sum is determined by the speed of the belt and the distance a parcel must travel between the bar code reader and the next point on the belt at which it might be diverted. The faster the conveyor, the lower this sum can be. Similarly, the closer the bar code reader is to the diversion point, the lower this sum can be. Our task is to determine

whether the sum of the database query response time, the network delay, and the PLC processing time is less than the time it takes for the parcel to travel from the bar code reader to the diversion point. It is usually less costly to determine these delays individually in the lab than it is to build an entire system and see how it performs once it is switched on. It is also easier to model the PLC, database, and network delays separately than to model them as one large system. The results of the respective performance models might then be combined into a whole, just as the system components are combined into a whole after functional and integration testing.

To summarize, we need to formulate the following models to evaluate the performance of the conveyor system:

1. A model describing the status message traffic between the PLCs and the conveyor status monitor. This includes routine status traffic and alarm traffic where needed.
2. A model describing the message traffic directly attributable to the routing and movement of parcels along the conveyor belt. This includes routing queries made to the parcel routing database and their responses.
3. A model describing the message traffic directly attributable to the registration, induction, movement, and delivery of parcels.
4. A model of the local area network used to transport the registration, induction, movement, routing query, and delivery message traffic.
5. A model of the server hosting the routing query database.
6. A characterization of processing delays in the PLCs.
7. A model that integrates the results of the outputs of the various models.

The outputs of the first three models are inputs to the fourth model. The outputs of the third model are inputs to the fifth model. The model of the traffic attributable to the routing of parcels also describes the demand made on the routing query database, that is, the sixth model. All of these outputs cascade into the integrated model mentioned in item 7. The network traffic model is described in [BSA2005].

Let us turn our attention to a model of the parcel routing database. Because we have neither direct traffic data nor performance data for an actual system nor an actual architecture, we recommend the use of a reference architecture and contrived model parameters to build an initial model of the system.

Following the modeling approach discussed in Chapter 3, our initial focus is on the portion of the resource consumption by the database that is driven by transactions triggered by parcel registration and parcel movement. We would begin by examining a reference architecture that is intended to support a reference workload and proceed from there to build a performance model. We would model proposed changes to the architecture by changing the parameters of the performance model and computing the performance measures that would result.

The design of our fictitious system is based on the following considerations:

1. Database records pertaining to parcels on the conveyor belt will be stored in the database cache. They may be removed from the cache once they are no longer on the conveyor belt, either because they have been loaded onto their target mode of transport, such as a plane, train, container, or truck, or because they have been delivered to a pickup point such as a baggage claim belt at an airport.
2. The route to be followed by a parcel will be computed before the parcel is loaded onto the conveyor for the first time at an induction point and stored in the routing database cache as well as on disk.
3. The database will be updated every time a parcel enters a new belt segment, whether because of a diversion or because two segments abut one another. This is done to ensure complete traceability of the chain of custody of the parcel.
4. Updates about the disposition of the parcel will always be written to disk after being written in the database cache, for logging purposes. Updates to the disk will be done asynchronously so that parcel movement is not delayed until the disk update is complete.
5. Records of parcels that are no longer on the belt will be removed from the database cache whenever the cache occupancy reaches a high-water mark. The oldest records will be removed first.
6. Routing queries for parcels in transit are based on the parcel's bar code only.
7. The actual parcel is mapped to the bar code when the parcel is registered. The corresponding record is loaded into the database cache when the parcel is loaded onto the conveyor for the first time. A long time may elapse between registration and intake. For example, a parcel may be registered online when a shipping label is created late in the evening, but intake may not occur until the parcel is brought to a drop-off point the next morning.

8. Modifications to the route can occur while the parcel is in transit, whether because of breakdowns or because of a change in a parcel's destination for whatever reason. The cancellation of the shipment of a parcel that is already on the belt is equivalent to treating the place where the parcel will be retrieved as its new destination.

These activities and behaviors would be reflected in the choice of transaction rates, visit ratios, and service times at the CPUs and the disks in the system hosting the database.

Once we have established that the communication traffic in and out of the routing database server will not saturate any of the network devices on the paths to it, the focus of the performance study of the routing database should be contention for processing power, disk access, and perhaps memory.

Contention for memory and for software objects such as read and write locks will not usually be considered initially unless there is reason to believe that a software bottleneck exists, for example, if the processor utilization does not rise as the query rate increases. The database should have been designed to reduce the risk of lock contention in the first place, for example, by using row-level locking instead of table-level locking where appropriate. Even then, repeated lock contention does not add to processor or disk usage so long as contending threads and processes are put into a sleep state until the locks become available. Even if contending threads do not sleep until the lock becomes available, there is no contribution to disk usage if the object protected by the lock is resident in memory. Still, the designers of the system should be advised to avoid repeated contention, since it slows processing down by stealing both memory and execution cycles.

10.4 Example: Modeling Asynchronous I/O

In our discussion of the computationally intense system in Chapter 9, we did not examine the effect of asynchronous I/O on performance in detail. We explore the modeling issues here. For a detailed discussion of the performance modeling of asynchronous I/O, the reader is referred to [HeidelbergerTrivedi1982].

Asynchronous I/O may be used to shorten system response times when it is not necessary for the process that initiated the I/O to wait until it has been completed before proceeding. This is applicable when

a process is writing to a device and does not need notification of I/O completion for a computation to continue, or if the process has issued a read request and does not need the returned result until reading is complete. Asynchronous I/O in Windows-based systems is described in [WindowsKB2013].

To model the performance impact of asynchronous I/O, one should take the following points into account:

- Asynchronous scheduling of I/O does not change the work that must be done to carry it out. Hence, using it does not change the utilization of the I/O device.
- The use of asynchronous writing entails the buffering of the data to be written until the write operation on the I/O device has been completed. Similarly, data that is read asynchronously from the I/O device must be stored at a known memory location until the process that initiated the request reads it. Sufficient buffer space must be available in memory to hold the data that is to be written or retrieved. Here, we shall focus solely on asynchronous writes.
- An I/O device may perform both synchronous and asynchronous I/O depending on the needs of the application. Since synchronous and asynchronous activities are competing for the same resource, each kind of I/O will tend to increase the device response time of the other kind.

Let us now formulate a performance model. Let k be the index of the I/O device of interest. As depicted in Figure 10.1, we can think of an asynchronous I/O request as an I/O that is spawned by a job that then goes its own way, returning to the CPU to continue its work. By contrast, a synchronous I/O request goes back to the CPU only once it has been completed in the usual way.

As usual, we denote the average service time of the device by S_k. The device has both asynchronous and synchronous visits. Denote the corresponding visit ratios by $V_{k,a}$ and $V_{k,s}$ respectively. The demand on device k is given by

$$D_k = (V_{k,a} + V_{k,s})S_k \tag{10.1}$$

Denote the delay a process experiences when performing an asynchronous I/O by d_k. Denote the response time for any request at device k by R_k. Then, the global response time for the process doing asynchronous I/O and synchronous I/O at this device may be formulated as

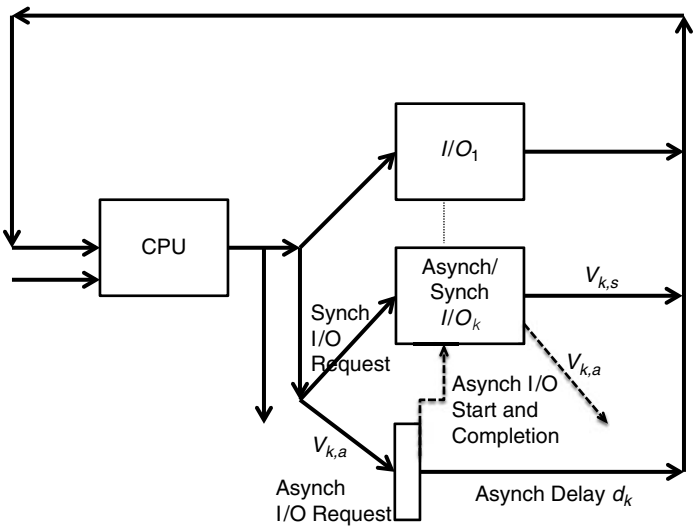


Figure 10.1 *A queueing network model of asynchronous I/O*

$$R_0 = \sum_{i \neq k} V_i R_i + V_{k,a} d_k + V_{k,s} R_k \tag{10.2}$$

The following assumptions are implied by the formulation:

1. The second term assumes that the process suffers a constant delay d_k when performing asynchronous I/O, regardless of the system load. The means for estimating d_k, or, perhaps more easily, $V_{k,a} d_k$, depend on the nature of the system and on the available measurements.
2. The expressions for the individual response times depend on the processing demands at the individual devices and on the system throughput, and on whether the central subsystem is best modeled as an open or closed network of queues.

Notice that if there were no asynchronous I/O, we would have $V_{k,a} = 0$, giving us the same formulas for the device loading D_k and system response time R_0 as usual. If we have purely asynchronous I/O at device k, the system response time becomes

$$R_0 = \sum_{i \neq k} V_i R_i + V_{k,a} d_k \tag{10.3}$$

because return to the CPU is not delayed until completion of the I/O.

Whether the central subsystem is modeled as a closed network of queues with a fixed number of circulating jobs or as an open network with an essentially unconstrained number of circulating jobs, the number of asynchronous I/O requests queued at device *k* is unconstrained, because the jobs generating the asynchronous requests need not wait until they have been completed. For an analysis in a closed network with a fixed number of circulating jobs, the reader is referred to [HeidelbergerTrivedi1982].

Let us now consider the response time and capacity of device *k*. Suppose that the central subsystem is an open network. Then, device k can be modeled as an open queue with exponential service time. The mean queue length of device k is given by

$$\bar{n}_k = \frac{U_k}{1-U_k}, 0 \le U_k < 1 \tag{10.4}$$

where the utilization U_k, which is due to both synchronous and asynchronous activity, is given by

$$U_k = X_0(V_{k,a} + V_{k,s})S_k \tag{10.5}$$

and X_0 is the global system throughput. Using Little's Law and the expression for the average length of an M/M/1 queue, it can be shown that the average amount of time to process I/O requests of either kind is

$$R_k = \frac{S_k}{1-U_k}, 0 \le U_k < 1 \tag{10.6}$$

Notice that the portion of this response time attributable to asynchronous activity contributes to the overall system response time only to the extent that the activity contributes to the delay in handling any synchronous activity. The total time to complete all asynchronous activity for a job, $R_{k,a}$, is the average response time per visit multiplied by the number of asynchronous visits. Thus, we have

$$R_{k,a} = V_{k,a}R_k \tag{10.7}$$

If the activity at device *k* consists solely of writes, the time to complete all work on behalf of a transaction is bounded below by both $R_{k,a}$ and R_0. Asynchronous I/O allows the combined time to complete activity to

be less than or equal to the sum of these two delays. Hence, the total time to complete all activity related to a job of this type R_J is governed by the following inequality:

$$\max(R_0, R_{k,a}) \leq R_J \leq R_0 + R_{k,a} \tag{10.8}$$

Let us now revisit the test data and initial modeling results we saw in Chapter 9. Figure 10.2 shows the predicted and measured response times for the overall response time, and the predicted response times for the individual devices.

Here, we see that the predicted response time of the RAID device is the driver of the predicted overall response time, because it is the bottleneck device, while the predicted values of the combined response time contributions of the CPU and system disks to the predicted response times do not increase much with the load. Because we were not able to capture the response times of the individual devices, we cannot directly validate the predicted values of the response times.

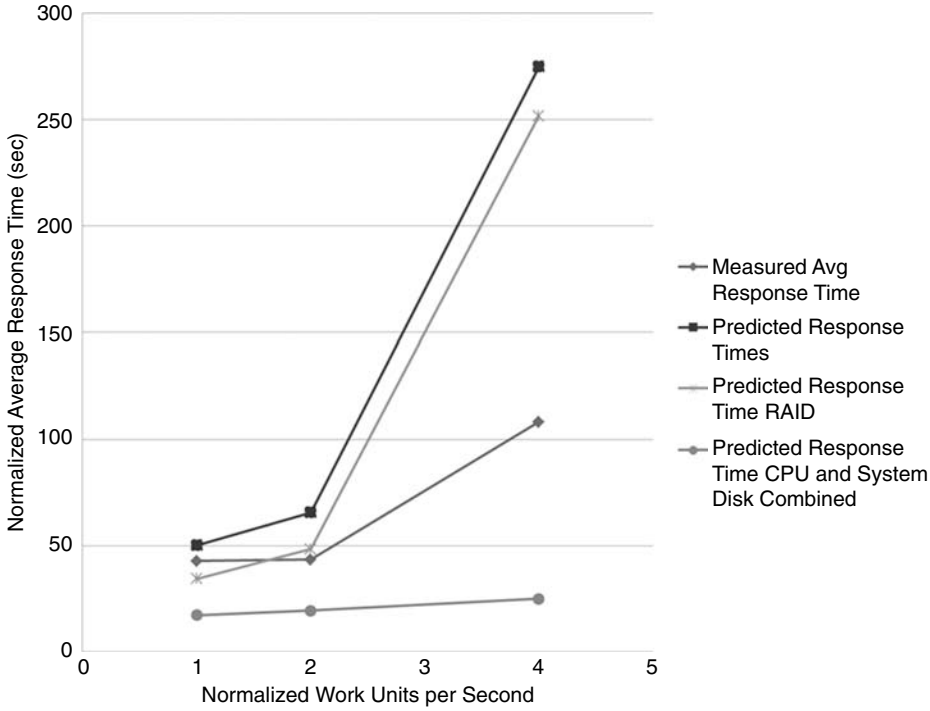


Figure 10.2 *Response times for the computationally intensive system*

Still, the shapes of the response time curves and their placement do yield some insights and some caveats about both the model and the system:

1. While the rise in the measured overall response time under the asynchronous I/O regime is not as dramatic as the predicted rise under the assumption that I/O is purely synchronous, it is large enough for us to suppose that not all of the I/O at the RAID device is asynchronous. If it were, the overall average response time would hardly rise at all, because the combined predicted average response time of the CPU and the system disk does not increase much with the load.
2. Even if the response time perceived by the user does not include the asynchronous portion of the I/O, the I/O will take a great deal of time to be completed, so that any activity that entails reading the information written to the I/O device will be delayed until that I/O is completed. This implies that the overall system response time that should be measured and modeled should include that delay. The performance measurements do include that delay. The modified response time formula in equation (10.2) may not do so, in part because it does not quantify the extent of overlap between I/O and the activity at other devices. Indeed, the extent of that overlap may be a function of the load itself, as is indicated by the widening difference between the overall measured response time and the predicted response times of the CPU and system disks combined.
3. Modeling a RAID device as a single-server FCFS queue might not be accurate. The predicted values of the response times could be too high. Performance models of RAID devices are discussed in [LeeKatz1993], [CLGKP1994], and [TTH2012], among others.
4. As we mentioned in Chapter 9, the system under study has parallel CPUs and parallel cores. This was not captured in our model. Some of the error in the predicted overall response time may be due to this, but not all, because the CPU is lightly loaded compared with the bottleneck device.

In summary, while the original coarse performance model correctly predicts the qualitative nature of the performance of the system and the load at which saturation occurs, and correctly tells us where the

bottlenecks are, it is insufficiently detailed to accurately inform us about the effects of design choices such as asynchronous I/O and multiprocessing on overall performance. To accomplish this, a more detailed and involved modeling effort would be required than was feasible within the scope of the work commissioned by the client.

10.5 Systems with Load-Dependent or Time-Varying Behavior

The performance of systems with load-dependent behavior will not be predicted accurately by performance models with constant, load-independent, and time-independent parameters. Even if the performance model is accurate initially, it may not be later on. The examples that follow show that it is necessary to monitor the performance of a system over time both during initial tests and in production to ensure that performance models remain accurate. If resource utilizations and other performance measures show trends, oscillate, or otherwise vary considerably over time, performance models with constant parameters based on the average values of measurements will not be accurate. Some examples illustrate this point.

10.5.1 Paged Virtual Memory Systems That Thrash

Systems with paged virtual memory may experience repeated ejection and retrieval of pages at a rapid rate if too many jobs are loaded into main memory at once, especially if the size of main memory is too small for the demand. In some cases, the same pages are rejected and retrieved over and over again. This is known as *thrashing*. Thrashing increases the average system response time while seriously degrading the system throughput, because it is accompanied by a massive increase in demand for I/O to bring missing pages into memory while writing modified pages out before ejecting them from memory. In addition, there is a demand for processor power to drive those I/Os and to choose pages for replacement. Once the need to replace pages abates, perhaps because a job has finished execution or been canceled, thus freeing memory, thrashing subsides. For a discussion of thrashing and mechanisms for mitigating or avoiding it, the reader is referred to [CoffDenn1973], [Denning1980], and [PS1985].

10.5.2 Applications with Increasing Processing Time per Unit of Work

There are many cases in which the amount of processing time or I/O activity per unit of work increases over time. In such cases, the performance of the system will degrade. Its capacity will be reduced and the response time per unit of work will increase. The accuracy of performance models of these systems will deteriorate over time if they have constant parameters. For example:

- As the number of records stored in a database grows, the amount of processing needed to carry out each query may increase.
- If a stream of unsorted incoming records is stored in ascending order in a data structure using an insertion sort, the time to insert each record in order could go up with the square of the number of records already present [HS1976].

10.5.3 Scheduled Movement of Load, Periodic Loads, and Critical Peaks

Some systems are subject to periodic or oscillating loads. For example:

- Some monitoring or control systems have pairs of components that are arranged in parallel. To ensure that each component can act as a standby for the other in case of failure, the load is shifted from one to the other at regular intervals, such as hourly. A performance prediction based on the average use of the two components over a prolonged period will underestimate the average response time. The system capacity under this regime may be overestimated by a factor of 2.
- Payroll and online banking systems experience peaks of activity on and shortly after the days that employee payments are issued. Performance engineering for these systems, including modeling and testing, should be done for the period of peak activity.
- We have already seen that the average values of performance measures of alarm systems operating in emergency mode are uninformative. Performance models of the average behavior of such systems will tend not to be accurate or meaningful.
- In the United States, tax returns must be filed annually by midnight on April 15 or the first business day thereafter. The loads on electronic filing systems will increase markedly as the deadline

approaches. A performance model based on the average load during a 48-hour period on April 14 and 15 is unlikely to provide meaningful predictions about their performance.

10.6 Summary

The foregoing examples suggest that combining a coarse performance model and a well-structured performance test is often sufficient for the prediction of qualitative performance trends with respect to the drivers of load and memory footprint. Examples of drivers of load include the offered transaction rate in transaction systems and the rates at which events occur in monitoring systems. Examples of drivers of the memory footprint include the sizes of the executable codes of all running programs, the number of concurrently active transactions, the size of the in-memory portion of any database, and, in the case of monitoring systems, the number of devices being monitored and/or managed.

Investigating the performance impact of particular design choices, scheduling rules, or hardware technologies might entail identifying the areas where delays occur, and then building detailed models of these aspects of the system in isolation. The results might then be incorporated into an integrated model of the system as a whole. Related techniques include hierarchical decomposition [LZGS1984] and layered queueing networks [FAWOD2009, XOWM2005].

For systems that have been measured while in production or while being tested, a preliminary analysis of the measurements may provide an indication of what part of the system may be causing performance issues. For systems that are not yet in production, a traffic modeling effort may be initiated because of early concerns about whether the volume of work will exceed the capacity of one or more system components. For example, in the case of the conveyor system, one may be able to determine from design and protocol specifications that a given rate of parcel movement could cause available bandwidth or I/O capacity to be exceeded. If that is the case, investing in a detailed model capturing a broad range of system functionalities would not be helpful. Instead, one should work with the architects and other stakeholders to determine methods to reduce the amount of message activity to the point that the network bandwidth would be sufficient to handle the anticipated parcel movement rate. Once concerns about network saturation have been addressed, one may turn to concerns about any other bottlenecks that might be unmasked as a result.

As we saw in Section 10.5, various time-varying and/or load-dependent system behaviors can undermine the validity of performance models that are based on long-term average values of performance measures. For this reason, the values of the performance measures of the system over time and at various load levels should be examined to rule out the possibility of aberrant behaviors that not only undermine the validity of performance models but could also be indicators of system instability that might impact performance.

The following is an outline of questions that might be asked when one is deciding whether to embark on a modeling study and what direction a modeling study should take:

1. What do we wish to learn from the model? What questions will it help us answer?
2. How will the model be used in the future?
3. Can the model be used to help stakeholders make necessary decisions about design, implementation, and/or capacity?
4. Can the model be used to help understand whether the system will be able to meet throughput and response time requirements?
5. Are the input parameters of the model available from measurements or otherwise?
6. How sensitive are the model predictions to the values of the parameters?
7. Is the model intended to answer questions about the entire system, or only one part of it? Is there a particular bottleneck in the system whose performance should be examined first?

Deciding what we wish to learn from the model and how it will be used in the future is a precondition for deciding what aspects of the system must be modeled in detail and what aspects can be modeled coarsely. If the model parameters are not available from measurements, validation of the model by comparing it with measurement data will not be possible. In that case, it will be necessary to use best-effort estimates of the model parameters and run the model on various input values to determine if the model's predictions are highly sensitive to them. Sensitivity to model parameters could be a sign that the intended operating range of the system is near saturation, that the system is not robust, or that the model is not robust. The robustness of the model and of the system under study should always be examined carefully.

In our study of the I/O-bound computationally intense transaction processing system in this chapter and in Chapter 9, we saw that our coarse model was adequate for predicting hardware resource utilizations and for showing the absence of software bottlenecks, but that it could not be used to answer detailed questions about the deployment of multiple processors or changes in priority scheduling, since these features were not included. It could be used to answer questions about distributing I/O processing among multiple I/O devices, although it could not be used in its present form to determine whether different RAID schemes would be superior to what is in place at the moment, since those features are not captured in the model. To answer questions about particular system features, it may be worthwhile to build simple models of these features that capture them in isolation, and then incorporate the results and/or the smaller models into the larger model.

10.7 Exercises

10.1. In the computationally intense transaction processing system discussed in this chapter and in Chapter 9, the definition of the transaction response time has not been fully described. The transaction response time could end when the last asynchronous I/O has been initiated, or when the last I/O has been completed.

- **(a)** Give an expression for the transaction response time based on the definition that the response time ends when the last I/O to the asynchronous device has been initiated.
- **(b)** Explain why this definition is unsatisfactory if the goal of the transaction is to transform data for subsequent processing by another computation that cannot proceed until the transformed data is available on the asynchronous I/O device.
- **(c)** Give formulas for upper and lower bounds on the transaction response time that include the final writing of data to the asynchronous device.
- **(d)** Explain the circumstances under which the bounds will hold.

10.2. The status of the conveyor system in this chapter is monitored by a system that receives status messages at regular intervals from the PLCs and from sensors that are connected to them.

Status messages are sent to the monitoring station over the same local area network that is used to send parcel movement control messages and parcel routing queries.

(a) Describe how you would characterize and model the local area network traffic attributable to parcel movements.

(b) Describe how you would characterize and model the local area network traffic attributable to status messages.

(c) Explain how you would assess whether the monitoring traffic limits the capacity of the conveyor system to the point that the desired number of parcels cannot be handled. (*Hint:* Is the combined bandwidth of the parcel movement control traffic and the monitoring traffic close to that of the available bandwidth on the local area network on any of its segments?)

(d) Observations of the conveyor system indicate that the misrouting of parcels is 50% more likely to occur when the ambient temperature in the warehouse exceeds 40 degrees Centigrade, causing some of the conveyor motors and PLCs to overheat. It is also known that overheating will trigger the generation of alarm messages at a rapid rate. Describe how you might use your performance model to explain how overheating could cause misrouting of parcels even if there is no mechanical problem associated with increased temperatures. Explain how you might corroborate this finding through the use of measurements.

(e) Discuss the impact of this misrouting on the query pattern at the routing database, and how you might go about exploring the resulting performance impact of overheating.

10.3. A system with paged virtual memory may show a marked increase in paging activity when the number of jobs in the system is large. Using the bottleneck analysis methods of Chapter 3, explain how a tenfold increase in paging activity with some number N jobs present would drive down the maximum achievable throughput and drive up the minimum response time of the central subsystem. (*Hint:* Consider the visit ratio of the paging device and the resulting visit ratio of the CPU.)

This page intentionally left blank

Chapter 11

Scalability and Performance

Scalability is a highly desirable and commercially necessary feature of many systems. Despite that, there is no universally accepted definition of it. There is no hard-and-fast rule about how to achieve it, although the factors that might undermine it are often readily identifiable and easily understood. In this chapter we shall explore some definitions of scalability. We shall identify practices and system characteristics that are conducive to it and patterns and characteristics that can undermine it. Scalability pitfalls will be explored. We shall show how to plan performance tests to verify scalability and interpret the test results accordingly.

11.1 What Is Scalability?

Scalability is a desirable attribute of a network, system, or process. The concept connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing or shrinking volumes of work gracefully, and/or to be amenable to reduction in size or to enlargement. Usually, one thinks of the need to scale a system up so that it can cope with an increased workload; or control, manage, or monitor larger numbers of devices. In some cases, the ability to scale down a system is desirable when it is being adapted for a niche market or the setting in which the system is intended to operate is inherently

smaller. For instance, there are features of building management systems and alarm systems that one might wish to deploy in small buildings as well as in large ones. Similarly, one might wish to deploy components of a medical system in a rural clinical setting as well as in a large metropolitan hospital, but at far less cost, because the revenue base in the rural area is smaller.

The scalability of a system subject to growing demand is crucial to its long-term success. At the same time, the concept of scalability and our understanding of the factors that improve or diminish it are vague and even subjective. Too often, we know when we don't have enough of it but are not always able to describe why or in what terms. Many systems designers and performance analysts have an intuitive feel for scalability, but the determining factors and the means of quantifying their impacts are not always clear. They may vary from one system to another.

When procuring or designing a system, we often require that it be scalable. The requirement may even be mentioned in a contract with a vendor. The requirement may be ambiguous unless the scope and dimensions of scalability are specified. When we say that a system is unscalable, we usually mean that the additional cost of coping with a given increase in traffic or size is excessive, or that it cannot cope at this increased level at all. Cost may be quantified in many ways, including but not limited to response time, processing overhead, space, memory, or even money. A system that does not scale well adds to labor costs or harms the quality of service. It can delay or deprive the user of revenue opportunities. Eventually, it must be replaced.

The terms *coping* and *excessive* are subjective. This is true of many characteristics and definitions of scalability. Metrics that might describe scalability depend on the context to which they are being applied. The adequacy of scalability depends on the extent to which a system must be scaled, and the dimensions in which it must be scaled. The extent and the dimensions of scalability should be specified in the performance requirements, as we discussed in Chapter 6.

US Supreme Court Justice Potter Stewart once said he could not define obscenity, but that he knew it when he saw it [Jac1964]. Knowing scalability when we see it or when the system apparently fails to provide it is insufficient for engineering purposes. Hill [Hill1990, quoted in Duboc2009] writes that "I examined aspects of scalability, but did not find a useful, rigorous definition of it. Without such a definition, I assert that calling a system 'scalable' is about as useful as calling it 'modern'. I encourage the technical community to either rigorously define scalability or stop using it to describe systems."

Here, we attempt to describe some characteristics of systems that are scalable, and characteristics of systems that make them unscalable. A definition as rigorous and consistent as appears to be desired by Hill may be unobtainable, in part because different types of scalability, such as load, space, and structural scalability, are intertwined [Bondi2000]. Jogalekar and Woodside have explored metrics to describe scalability [JW2000]. They have underscored the need to separate the evaluation of scalability in terms of throughput from the evaluation of system performance in terms of response time. Thus, they implicitly make the point that for a system to be considered scalable with respect to the offered load, the increased offered load must be sustainable in the sense we have described in the context of performance requirements in Chapter 6.

In her doctoral thesis, Duboc [Duboc2009] quotes these and other works on scalability and describes a framework for characterizing and analyzing the scalability of software systems. Duboc et al. [Duboc2008] define scalability as "... the ability of a system to satisfy its quality goals to levels that are acceptable to its stakeholders when characteristics of the application domain ('the world') and system design ('the machine') vary over expected ranges." This is a general definition of scalability that may be applicable not only to the impact of scale on performance, but also to the impact of scale on reliability, dependability, security, and other quality attributes that are relevant to the system of interest.

Our focus in the remainder of this chapter will be on those aspects of scalability that are directly related to performance, namely, load scalability, structural scalability, space scalability, and space-time scalability. While examining characteristics of scalability that affect performance, we shall explore aspects of software architecture, design, and implementation that can affect scalability favorably or unfavorably. We shall link these aspects of scalability to performance engineering topics we have explored earlier in this volume, including performance requirements and modeling. But first, we briefly examine some scaling methods.

11.2 Scaling Methods

When discussing scalability, it is necessary to describe the dimensions in which scaling is to occur, and the method used and architectural choices made to scale a system to meet performance, business, and engineering needs. These needs may arise from increasing or decreasing the supportable traffic volume and from increasing or decreasing the number of objects stored, monitored, or managed by the system.

Adding resources to a system only increases the rate at which it can process units of work if those resources are usable and used. This is not trivial. For example, if the architecture of a system is inherently single-threaded, adding processors or cores to it will not increase its usable capacity, because the single thread can use only one processor or core. On the other hand, replacing the processor with a faster one may increase the sustainable system throughput, provided that the processor is the bottleneck, and provided that the system is not otherwise constrained by memory, memory bus speed, I/O capacity, or other factors. Here, we discuss some methods of scaling a system. In Section 11.3 we shall look at different types of scalability.

11.2.1 Scaling Up and Scaling Out

Replacing a single processor with a faster one is an instance of *scaling up*, while adding multiple processors is an instance of *scaling out*. In general, scaling up means that we replace a slow device or server with a faster one, while scaling out means that we replicate the device or server so that the replicates run in parallel. Scaling out may be implemented by distributing work among different hosts. These may be in the same location, or in different locations.

By way of illustration, consider a web application consisting of three logical tiers: a web page server to draw and serve the web pages, an application server to implement business logic, and a back-end database for data storage and retrieval. Initially, a bench-scale version of this system might be implemented on a desk-side PC with a single processor. If the processor is heavily loaded, it might be replaced with a single faster processor. Time slicing on this system may give the illusion of providing concurrent execution of database queries, the handling of business logic, and the presentation of pages. The inevitable frequent context switching among these functions may slow the system down. Some parallel execution may be achieved by putting multiple processors in the same system unit, but increased memory contention as the database grows may cause the paging device or the memory bus to become a bottleneck. Thus, scaling up the processor or distributing the processing among processors within the same box may not be sufficient to meet the increasing demands on the system.

11.2.2 Vertical Scaling and Horizontal Scaling

The next step is to separate the three functions into separate system units or servers that communicate via a local area network. This is

known as *vertical scaling*, because the information flows vertically from one tier to its neighbor above or below. Web page service, application processing, and database processing each run on dedicated system units known as servers. If the application server is overloaded, its work can be spread among two or more identical servers operating in parallel. This is known as *horizontal scaling* or *scaling out*. The software on each server should be multithreaded, so that multiple activities can occur in parallel on multiple processors. Each web page server can spread the application processing load among the application servers. A load balancer is needed to distribute arriving tasks among parallel web page servers. It may be more difficult to scale the database out, since doing so requires replication of the data. At least one commercial database platform is available to do this. Setting up replication and clustering may require special expertise during implementation, and careful monitoring and administration in production [Niemiec2012]. Moreover, horizontal and vertical scaling may be difficult to achieve if the interfaces between the tiers are not amenable to it. If the interfaces consist of TCP sockets, separation of functions across multiple hosts may be straightforward, even though configuration of the IP addresses must be done with care.

11.3 Types of Scalability

When exploring the architectural and design choices that must be made to support scalability, it is useful to consider the types of scalability that may be required. Load scalability pertains to the ability of the system to cope with increasing amounts of traffic. Space scalability relates to the amount of memory that will be required. Space-time scalability relates to memory requirements as a function of the load. Structural scalability concerns those aspects of system structure that limit scalability, for example, the limit on memory usage imposed by the number of bits in the address space or the limit on the number of outstanding messages in a network connection due to the window size prescribed by a standard.

11.3.1 Load Scalability

We say that a system has load scalability if it has the ability to function gracefully, that is, without undue delay and without unproductive resource consumption or resource contention at light, moderate, or heavy loads while making good use of available resources.

Of course, the terms *light*, *moderate*, *heavy*, *gracefully*, and *undue*, while descriptive, are somewhat subjective. Therefore, they must be viewed in the context of the performance requirements associated with the desired level of scaling. The term *gracefully* is used to express continuity of operation without interruptions, spikes, or troughs in response times, transient drops in throughput, and the like. A system that goes into deadlock or crashes on overload is not functioning gracefully. A system or application that has multiple processors at its disposal and can use only one of them is not making good use of available resources.

It may be easier to describe system characteristics that undermine load scalability than it is to describe those that aid it. In each of the examples that follow, we shall see structural attributes that may diminish scalability, even if they were designed to prevent a different kind of problem.

Some of the factors that can undermine load scalability include (1) the scheduling of a shared resource, (2) the scheduling of a class of resources in a manner that increases the class's own usage (self-expansion), and (3) inadequate exploitation of parallelism. An Ethernet-based network using the CSMA/CD protocol does not have load scalability, because the high collision rate at heavy loads prevents bandwidth from being used effectively. The token ring with nonexhaustive service does have load scalability, because every packet awaiting transmission is served within a bounded amount of time.

A scheduling rule may or may not have load scalability, depending on its properties and the nature of the arriving traffic. For example, the Berkeley UNIX 4.2BSD operating system gives higher CPU priority to the first stage of processing inbound packets than to either the second stage or the first stage of processing outbound packets. This in turn has higher priority than I/O, which in turn has higher priority than user activity. This means that sustained intense inbound traffic can starve the outbound traffic or prevent the processing of packets that have already arrived. This scenario is quite likely at a web server [MogRam1997]. The situation can also lead to livelock, a form of blocking from which recovery is possible once the intense packet traffic abates. Inbound packets cannot be processed and therefore are unacknowledged. This eventually causes the TCP sliding window to shut, while triggering retransmissions. Network goodput, the rate at which data actually reaches its destination, then drops to zero. Even if acknowledgments could be generated for inbound packets, it would not be possible to transmit them, because of the starvation of outbound transmission. It is also worth noting that if I/O interrupts and interrupts triggered by inbound packets are handled at the same level of

CPU priority, heavy inbound packet traffic will delay I/O handling as well. This delays information delivery from web servers.

A system may also have poor load scalability because one of the resources it contains has a performance measure that is self-expanding, that is, its expectation is an increasing function of itself. This may occur in queueing systems in which a common FCFS work queue is used by processes wishing to acquire resources or wishing to return them to a free pool. This is because the holding time of a resource is increased by contention for a like resource, whose holding time is increased by the delay incurred by the customer wishing to free it. Self-expansion diminishes scalability by reducing the traffic volume at which saturation occurs. In some cases, it might be detected when performance models of the system in question based on fixed-point approximations predict that performance measures will increase without bound, rather than converging. In some cases, the presence of self-expansion may make the performance of the system unpredictable when the system is heavily loaded. Despite this, the operating region in which self-expansion is likely to have the biggest impact may be readily identifiable: it is likely to be close to the point at which the loading of an active or passive resource begins to steeply increase delays, because it is close to saturation.

We have already seen that load scalability may be undermined by inadequate parallelism. A quantitative method for describing parallelism is given in [Latouche1981]. Parallelism may be regarded as inadequate if system structure prevents the use of multiple processors or multiple cores for tasks that could be executed asynchronously. For example, a transaction processing (TP) monitor might handle multiple tasks that must all be executed within the context of a single process. If the host operating system allows only one task within the TP monitor to be executed at a time, only a single processor or core can be used. Horizontal scaling across processors or cores is infeasible in such a system. Similarly, single-threaded systems cannot make use of more than one processor, either. In some cases, an application may perform multiple activities. If the most CPU intensive of these can execute units of work only serially, the load among the processors will be unbalanced.

11.3.2 Space Scalability

A system or application is regarded as having *space scalability* if its memory requirements do not grow to intolerable levels as the number of items it supports increases. Of course, *intolerable* is a relative term. We might say that a particular application or data structure is space scalable

if its memory requirements increase at most sublinearly with the number of items in question. Various programming techniques might be used to achieve space scalability, such as sparse matrix methods or compression. Because compression takes time, it is possible that space scalability may be achieved only at the expense of load scalability. Cacéres et al. expand this definition of space scalability to include keeping memory consumption and bandwidth consumption within acceptable levels as the workload increases [CRPH2010]. They suggest that an operating system is *space scalable* if its requirement for physical memory can be reduced by using paged virtual memory management to store unused memory pages on disk. The demand for physical memory is only part of the story: it is possible that the image of the address space stored on disk could become unmanageably large, too. Moreover, if an application has poor locality of reference (as some transaction-oriented systems do), the system may have poor space scalability and poor load scalability because of the recurring and frequent need to write modified pages onto the paging device and bring pages from the paging device into main memory.

11.3.3 Space-Time Scalability

We regard a system as having *space-time scalability* if it continues to function gracefully as the number of objects it encompasses increases by orders of magnitude. A system may be space-time scalable if the data structures and algorithms used to implement it are conducive to smooth and speedy operation whether the system is of moderate size or large. For example, a search engine that is based on a linear search would not be space-time scalable, while one based on an indexed or sorted data structure such as a hash table or balanced tree could be, because the processing cost of a search is potentially much lower than that of a linear search. Notice that this may be a driver of load scalability for the following reasons:

- The presence of a large number of objects may lead to the presence of a heavier load.
- The ability to perform a quick search may be affected by the size of a data structure and how it is organized.
- A system or application that occupies a large amount of memory may incur considerable paging overhead.

Space scalability is a necessary condition for space-time scalability in most systems, because excessive storage requirements could lead to memory management problems and/or increased search times.

A system with poor locality of reference may exhibit poor space-time scalability, because of the paging or communications overhead associated with moving unused data or pages out of memory to make room for other pages. Notice also that poor locality of reference can be inherent in transaction-oriented applications that repeatedly refer to new records, or it can be a characteristic of a system that must progress through multiple functionalities or code segments rapidly in time, and therefore spends only a small percentage of time executing each one.

11.3.4 Structural Scalability

We may think of a system as being structurally scalable if its implementation or standards do not impede the growth of the number of objects it encompasses, or at least will not do so within a chosen time frame. Structural scalability is relative, because the desired extent of scalability depends on the number of objects of interest now relative to the number of objects later. Any system with a finite address space has limits on its scalability. The limits are inherent in the addressing scheme. For instance, a packet header field typically contains a fixed number of bits. If the field is an address field, the number of addressable nodes is limited. If the field is a window size, the amount of unacknowledged data is limited. A telephone numbering scheme with a fixed number of digits, such as the North American Numbering Plan, is scalable only to the extent that the maximum quantity of distinct telephone numbers is significantly greater than the set of numbers currently assigned. In this case, the extent of structural scalability may be constrained by the need to assign three-digit area codes to geographical regions, regardless of the size of the subscriber base in each one. The binding of area codes to regions prevents the use of the entire numbering space, as does the requirement that area codes and the seven-digit local numbers that follow them must not begin with either a 0 or a 1 (www.nanpa.com).

11.3.5 Scalability over Long Distances and under Network Congestion

An algorithm or protocol may be said to be *distance scalable* if it works well over long distances as well as short distances, for example, by meeting bandwidth or message throughput requirements. An algorithm or protocol is *speed/distance scalable* if it works well over long distances as well as short distances at high and low speeds. The motivation for these types of scalability is TCP/IP. Its sliding window protocol

shows poor speed/distance scalability in its original form, because the loss of a packet triggers the closing of the sliding window. Throughput is degraded until recovery from the loss occurs. A long round-trip time, due to distance, network congestion, or both, exacerbates this problem. The degradation is much larger if the maximum available transmission bandwidth is high [JacBrad1988]. Modifications to TCP/IP involving the slow expansion of window sizes after packet loss have been proposed to mitigate this problem [BOP1994].

11.4 Interactions between Types of Scalability

The foregoing types of scalability are not entirely independent of one another. Indeed, poor scalability of one type may, but need not, undermine another. Systems with poor space scalability or space-time scalability might have poor load scalability because of the attendant memory management overhead or search costs. Systems with good space-time scalability because their data structures are well engineered might have poor load scalability because of poor decisions about scheduling or parallelism that have nothing to do with memory management.

Let us now consider the relationship between structural scalability and load scalability. Clearly, the latter is not a driver of the former, though the reverse could be true. For example, the inability to exploit parallelism and make use of such resources as multiple processors undermines load scalability but could be attributed to a choice of implementation that is structurally unscalable. For example, a system whose thread pool cannot be expanded beyond a certain point, perhaps because the number of threads allowed by a software license is capped, has poor structural scalability. It also has poor load scalability, because the constrained number of threads may prevent the full exploitation of parallel processors.

While the types of scalability presented here are not entirely independent of one another, many aspects of each type are. Therefore, though they provide a broad basis for a discussion of scalability, that basis is not orthogonal in the sense that a suitable set of base vectors could be. Nor is it clear that an attempt at "orthogonalization"—that is, an attempt to provide a characterization of scalability consisting only of independent, noninteracting components—would be useful to the software practitioner, because the areas of overlap between the aspects of scalability discussed here are a reflection of the sorts of design choices a practitioner might face.

11.5 Qualitative Analysis of Load Scalability and Examples

In this section we illustrate the analysis of load scalability. Our examples fall into three categories: systems that cannot exploit parallel resources, systems with repeated unproductive cycling through finite state machines, and systems whose poor load scalability can be overcome with the judicious choice of a job scheduling rule. The use of finite state machines to depict system behavior is illustrated in [Shaw1974]. In the context of software performance engineering, small finite state machines have been used to depict the behavior of embedded components [SmithWilliams1998]. Concurrently interacting finite state machines have been studied by Kurshan and coworkers [KatzKursh1986].

By an unproductive cycle, we mean a repeated sequence of states in which a process spends an undesirable amount of time using resources without actually accomplishing the goals of the user or programmer. Classical examples include busy waiting on locks in multiprocessor systems, Ethernet bus contention, and solutions to the dining philosophers' problem that do not have controls for admission to the dining room [Hoare1974, Dijkstra1965]. Other examples include systems whose performance does not degrade gracefully as the load on them increases beyond their rated capacity. Some systems or procedures that are perceived as scaling poorly use resources inefficiently. They may hold one or more resources while being in an idle or waiting state, or they may incur overheads or delays that are tolerable at low volumes of activity but not at high volumes.

We now turn to examples that suggest how load scalability might be improved by reducing the occurrence of unproductive cycles, by modifying a job scheduling rule, or by facilitating the exploitation of parallel resources.

11.5.1 Serial Execution of Disjoint Transactions and the Inability to Exploit Parallel Resources

In Section 11.3.1 we described a system in which tasks are executed serially on one thread, even though they could be executed in parallel because they have no data in common. Another thread does background work. This architectural property became apparent when the performance engineer produced a plot of individual processor utilizations similar to the one in Figure 11.1. The host under test was a Sun

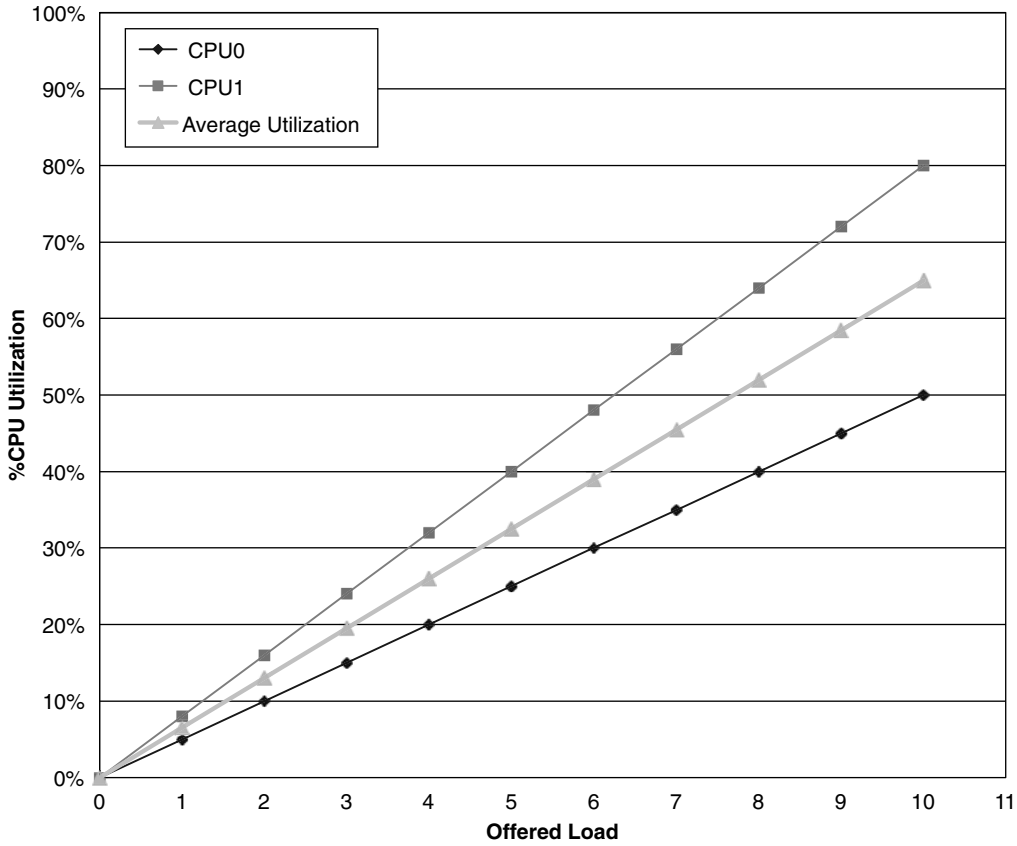


Figure 11.1 *Utilizations of unbalanced parallel processors, based on contrived data*

server with two processors running Solaris. The operating system uses cache affinity to return a thread to the processor on which it last ran before the most recent context switch suspending its execution. Measurements of the system were taken at increasing loads, with each load being run for a long period of time. The processor utilizations were obtained using the *mpstat* command. The total processor utilizations by individual threads could be computed by using the *ps –eLF* command or corresponding system calls to obtain the differences between successive values of the cumulative processing times and dividing these by the wall clock time between the observations. Since the observed utilization of each processor is a linear function of the offered load, there is no apparent software bottleneck. The processor utilizations computed from *ps –eLF* observations corresponded to the utilizations obtained

from *mpstat*. From this and our knowledge of how the scheduling works, we infer that particular threads were bound to particular processors. Under this regime, the maximum achievable throughput is lower than if the work could be evenly spread between the two processors, because the utilization of one CPU is much higher than that of the other. Thus, the load scalability of this system is limited by the single-threaded architecture of the application that executes the transactions and the inherent seriality of the system [Gunther1998]. Figure 11.2 shows that with full load balancing, the maximum transaction throughput could be increased from 8 to 11 transactions per second.

Note: The data in Figure 11.1 and Figure 11.2 has been contrived for the purpose of illustration because the actual results are not publicly available.

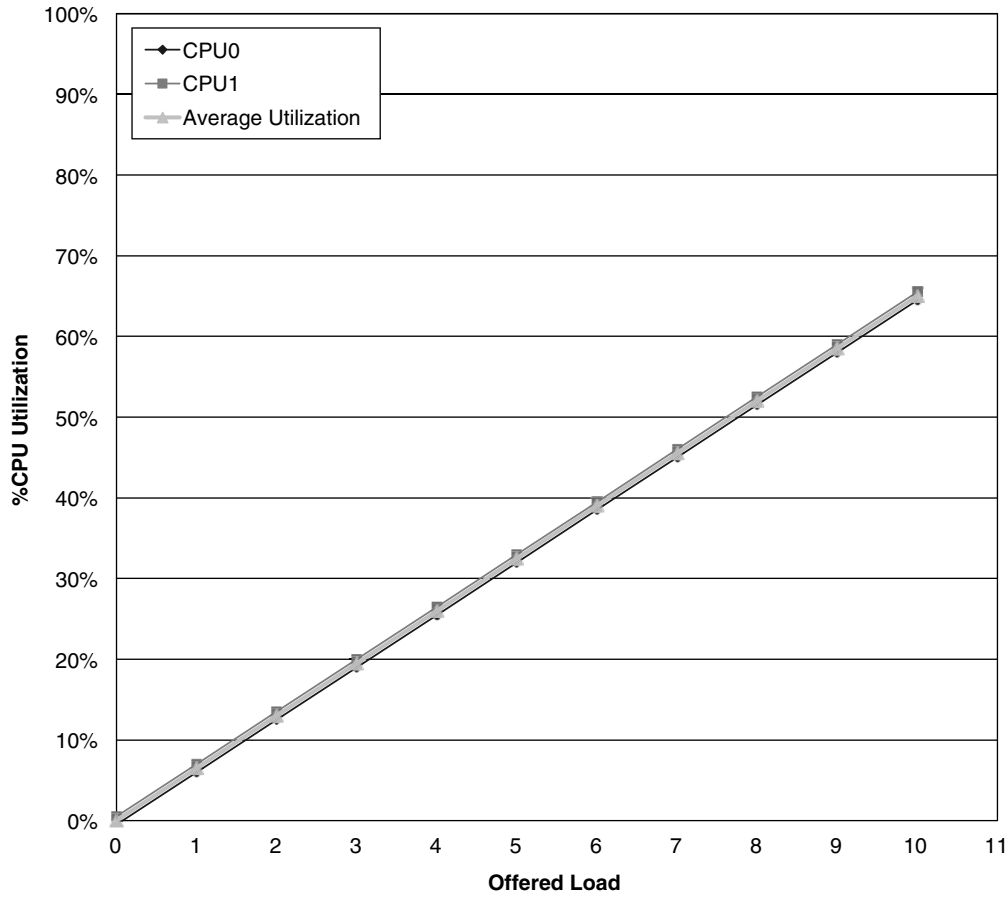


Figure 11.2 *Parallel processors with balanced utilizations, based on contrived data*

11.5.2 Busy Waiting on Locks

In the early days of multiprogrammed computing, contention for shared objects was arbitrated solely with the aid of hardware locks in memory. The lock is set by the first process that finds it unset; this process then unsets it when it has finished with the object protected by the lock. Other processes must repeatedly check it to see if it is unset. This repeated checking steals processing and memory cycles that could otherwise be used for useful work. Repeated cycle stealing, especially in a multiprocessor environment, degrades performance.

Figure 11.3 shows a state transition diagram for this concurrency control mechanism. The first directed graph represents the states of the memory bus, the second the states of a process trying to access the lock. Each attempt to access a lock corresponds to a complete cycle through the memory bus state transition diagram, as well as through a loop consisting of the three states *Get memory*, *Read lock*, *Free memory* in that order. Each process trying to access the lock corresponds to an instance of the second directed graph. Although one cannot break these cycles, one can reduce the frequency with which they are traversed by implementing mutual exclusion with a semaphore [Habermann1976, Dijkstra1965]. By putting a process to sleep until the object to which it seeks access becomes available, the semaphore mechanism simultaneously reduces lock contention, memory bus contention, and the unproductive consumption of CPU cycles. However, frequent context switching in a multiprocessor system with shared memory will still cause heavy memory bus usage, because of contention for the ready list lock [DDB1981]. Thus, the locking mechanism is not load scalable. However, minimizing the use of locks does

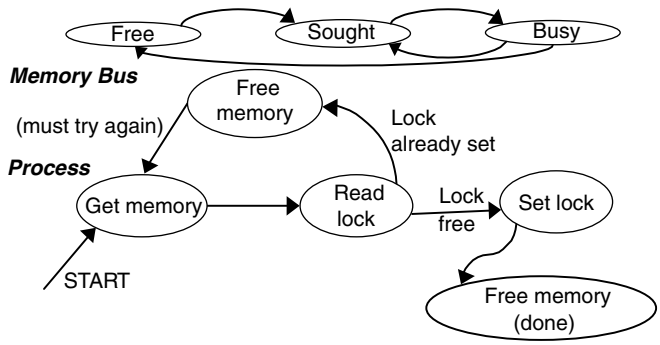


Figure 11.3 *Busy waiting on locks*

Source: [Bondi2000] © 2000 Association for Computing Machinery, Inc. Reprinted by permission.

ameliorate the problem by reducing the occurrence of unproductive cycles spent busy waiting.

11.5.3 Coarse Granularity Locking

Coarse granularity locking can undermine scalability in different ways. It occurs when an object or a set of objects is locked for longer than mutual exclusion on the object(s) of interest is needed. The problem can arise in databases when entire tables or indices are locked, even though only one row in a table is being read or modified. Programmers sometimes resort to coarse granularity locking of multiple objects to avoid divisibility issues, or to ensure compliance with the ACID principle of atomicity, consistency, isolation, and durability.

A measurement study by Horikawa [Horikawa2011] shows how the throughput capacity of a database system can be increased by a factor of 1.6 in a 16-processor system. This was done by systematically identifying coarse-grained locks and replacing them with fine-grained locks. Interestingly, refining locking granularity did not cause the processing load to be spread among the processors any more than before. That can be achieved only by designing the system to execute multiple threads in parallel, preferably on disjoint sets of objects.

The choice of granularity depends on the nature of the workload. Shasha and Bonnet [SB2003] show that row-level locking is useful for online transactions that hold records for short amounts of time. They also report that the overhead of maintaining the locks is low in modern systems. This means that the overhead of managing multiple locks is mitigated by the increased throughput enabled by reduced contention for large objects, provided that the system is suitably multithreaded. On the other hand, if large numbers of records will be locked simultaneously, performance may be improved if page-level or table-level locking is used instead. Some database systems support lock escalation as a means of reducing the amount of locking overhead when the number of records locked in a table by one thread is large, but this can increase the risk of deadlock. The reader is referred to [SB2003] for details.

11.5.4 Ethernet and Token Ring: A Comparison

The delay and throughput of the Ethernet and token ring with cyclic nonexhaustive service were compared in [Bux1981]. Here, we consider their performance with the aid of state transition diagrams. Figure 11.4 shows state transition diagrams for an Ethernet bus and for a single

packet awaiting transmission. The bus repeatedly goes through the unproductive cycle (*Busy*, *Collision*, *Idle*, *Busy*, *Collision*, . . .) when more than one workstation attempts to transmit simultaneously. Similarly, a station attempting to transmit a packet could repeatedly go through the cycle (*Silent*, *Backoff*, *Silent*, *Backoff*, . . .) before finally being sent successfully. It is well known that the performance of the Ethernet does not degrade gracefully as it approaches saturation [AlmesLaz1979]. Repeated traversals of these cycles help explain why. Introducing a bus with higher bandwidth only defers the onset of the problem. Thus, the CSMA/CD protocol is not load scalable.

Let us now examine the token ring. The state transition diagrams for the token and for a workstation attempting transmission of a single packet are shown in Figure 11.4. We see that the token moves between available states cyclically, and that it suffers increased delay in moving from one station to the next only if it is doing useful work, that is, if it is in use. A station attempting to transmit a packet is silent until the token becomes available to it. It then sends the packet without the need for backoff. Notice that the state transition graph for the LAN card contains no cycles. This helps to explain the graceful performance degradation of the token ring as it approaches saturation. The contrast with the Ethernet's cyclic behavior is clear.

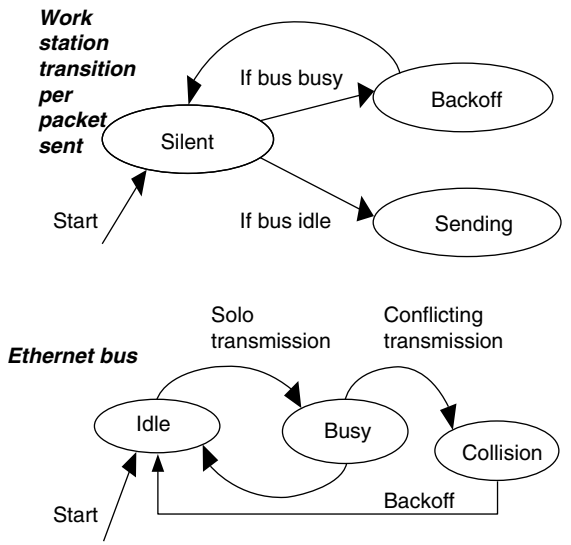


Figure 11.4 *Transition diagram for Ethernet*

Source: [Bondi2000] © 2000 Association for Computing Machinery, Inc. Reprinted by permission.

11.5.5 Museum Checkrooms

This is a simplification of a problem that could occur in replicated database systems [BondiJin1996]. At a museum checkroom like the one at the Metropolitan Museum of Art in New York (the Met), visitors are required to join a common FCFS queue to deposit and collect their coats. The scarce resources are coat hangers (passive) and attendants (active). Coats are placed on hangers, which are hung on carousels, each of which is maintained by a dedicated attendant. An attendant taking a coat performs a linear search on the assigned carousel to find an empty hanger, or to find the visitor's coat. Our objective is to maximize the time the customer can spend looking at exhibits and spending money in the museum restaurant and gift shop by assuring the smooth functioning of the checkroom.

The performance of this system degrades badly under heavy loads. First, the service time increases with the occupancy of the hangers, because it takes longer to find free ones. Second, and more seriously, the system is prone to deadlock at heavy loads, such as during the winter holiday season. In the morning most visitors are leaving their coats; in the evening they are all picking them up. The deadlock problem arises at midday or in the early afternoon, when many visitors might be doing either. Deadlock occurs when the hangers have run out, and visitors wishing to collect coats are stuck in the queue behind those wishing to leave them. Attendants can break the deadlock by asking those who wish to collect coats to come forward. This amounts to the resolution of deadlock by timeout, which is inefficient.

The museum checkroom with a common queue for leaving and collecting coats is a metaphor for an object pool in a computer system in which threads join a common FCFS queue to acquire and free elements in the pool. If deadlock occurs, the system hangs. We have seen this pattern in more than one system. Determining why a complicated system has hung can be extremely hard. Therefore, the use of an FCFS common queue to acquire and release discrete objects should avoided.

With its original service rules, the museum checkroom will function adequately at light loads and with low hanger occupancy but will almost certainly deadlock at some point otherwise. Regardless of the number of attendants and the number of coat hangers, the system is likely to go into deadlock if the volume of traffic is heavy enough. Increasing the number of hangers only defers the onset of deadlock; it does not eliminate the possibility. Increasing the number of attendants to clear a backlog of arriving and departing visitors only allows the

hangers to be filled sooner, bringing on an earlier occurrence of deadlock. Load scalability is further undermined because the holding time of a currently occupied hanger is increased in two ways:

1. The search time for an unoccupied hanger increases as the hanger occupancy increases. This increases the queueing time for those collecting and leaving coats.
2. A customer arriving to collect a coat (and thus free a hanger) must wait behind all other waiting customers, including those wishing to leave coats. This increases the time to free a hanger.

Both of these factors make the hanger holding time *self-expanding*. If the holding time is self-expanding, the product of the customer arrival rate and the hanger holding time—that is, the expected number of occupied hangers—will increase to exceed the total number of hangers even if the customer arrival rate does not increase. This is a sure sign of saturation.

Notice that the impediments to the scalability of this system vary with the time of day. In the morning, when almost all visitors are leaving coats, the impediments are the number of attendants and the somewhat confined space in which they work. The same is true at the end of the day, when all visitors must pick up their coats by closing time. At midday, the principal impediment is the FCFS queueing rule, which leads to deadlock.

For this system, load scalability can be increased with the following modifications:

1. There should be separate queues for those collecting and leaving coats, with (nonpreemptive) priority being given to the former so as to free the hangers as quickly as possible. This priority ordering reduces the tendency of the holding time to be self-expanding. Deadlock is avoided because the priority rule guarantees that a hanger will be freed if someone arrives to collect a coat.
2. To prevent attendants from jostling each other in peak periods, there should also be more than one place at which they can serve museum visitors, regardless of the carousel on which a deposited coat is kept.
3. To prevent visitors from jostling one another, there must be a wide aisle between the queue and the counter.
4. A sorted list of free hangers could be maintained for each carousel, to reduce the time to search for one.

The first modification would be easy to implement within architectural constraints, provided that the floor space is available in the main hall to maintain two separate queues. The second one may also be cheaper, while the third and fourth modifications are less so. The first alone would yield substantial benefits by eliminating the risk of deadlock and reducing hanger occupancy. The usefulness of the second depends on the patience of attendants and visitors alike, especially as closing time approaches. The cost of the fourth modification must be weighed against the reduced cost of keeping fewer attendants around.

The first modification is in use at the Louvre's coatroom in Paris. It continues to function smoothly even when the hangers are heavily occupied.

An enhanced version of the first modification is in place at the Museum of Modern Art (MoMA) in New York. In addition to having separate queues for visitors collecting and depositing their coats, those depositing their coats are directed by attendants to secondary queues for carousels where hangers have recently been freed. Similarly, a visitor collecting a coat joins a specific queue for a subset of the carousels containing the one on which his or her coat has been hung.

While the Louvre's coatroom is free of deadlock, it has another quirk. Unlike its counterparts in New York, it does not accept hats or backpacks, although it does accept umbrellas. Hats and backpacks must be left in a separate checkroom on the opposite side of the entrance hall, for which the visitor is also required to queue. While this may increase the space scalability of the Louvre's coatroom by allowing more coats to be squeezed into a fixed area, it will please only those visitors who arrive bareheaded and empty-handed. Staff will not permit hats to be stuffed into coat sleeves, though scarves are allowed.

There are analogies between these physical systems and computer systems. The visitors correspond to processes or threads. The carousels and hangers respectively correspond to memory banks and memory partitions. A visitor may not enter the galleries with a backpack or an umbrella and must either wear a coat or check it. This is analogous to a task acquiring a memory partition in a computer system. For performance analysis of an open arrival system in which jobs queue for a memory partition before processing and I/O may begin, the reader is referred to [Latouche1981], [AviHey1973], and [Bondi1992]. The attendants correspond to processors. The corridor between the carousels and the counter at which visitors arrive corresponds to a shared memory bus via which all memory banks are accessed. This is an example in which the main (and most easily surmounted) impediment to

scalability lies in the mechanism for scheduling service, rather than in eliminating the unproductive consumption of cycles. At the Met, the corridor (memory bus) is a secondary impediment, because attendants bump into each other while hanging or fetching coats during peak periods, unless each attendant is assigned to one carousel only. In a computing environment, this would be analogous to having a single bus for each memory bank. Finally, scalability is impeded by the narrow doorway between the head of the queue and the counter. This is analogous to there being a unique path from the CPU to the entire set of memory banks, that is, the bus. The Louvre's system of handling coats and umbrellas at one checkroom and hats and backpacks at another is analogous to distributing disjoint functionalities between two distinct servers. The order in which one visits the checkrooms is unspecified, but a visitor wearing a coat and carrying a backpack must visit both, since neither is allowed in the galleries, and since no checkroom accepts both backpacks and coats.

11.6 Scalability Limitations in a Development Environment

It is sometimes useful and convenient to model a system using a markup language in a development environment. Eysholdt and Rupprecht describe a case in which a specification in UML/XML becomes so large that an hour is needed to load it into a modeling tool [EysRupp2010]. While the tool may be adequate for smaller specifications, it would seem to be inadequate for large ones because of the computational effort required to interpret the specification and build an internal representation of it. The specification has grown to the point where the programming environment simply cannot provide a user with a fast enough response. Depending on the implementation of the environment, load, space, and space-time scalability issues may be present. The specification may require more memory than can be practically provided, so a good deal of paging may ensue as the user moves from one section of the model to another. The amount of processing that may be required to build the model internally and translate subsequent changes into an internal representation may be unacceptable. The solution used in this case had two parts: (1) porting the models to a text-based environment which permitted faster processing, and (2) configuring the host system to store

default values of attributes and observing a convention that only values other than the default ones would be specified in the model. While there is a risk of introducing some opacity into the resulting models, avoiding the repeated, explicit specification of default values enabled savings in processing and storage costs. This is a linguistic or modeling equivalent of eliminating unproductive cycles in environments with concurrently executing threads or processes as described in the previous section.

11.7 Improving Load Scalability

Our examples of poor load scalability show that it can have a variety of causes, ranging from access policies that are repetitively wasteful of active resources (e.g., busy waiting) to assignment policies that undermine the "common good" of the system by causing passive resources (e.g., coat hangers) to be held longer than is necessary to accomplish specific tasks.

One way to improve the scalability of a system is to reduce the amount of time it spends in unproductive cycles. This can be done by modifying the implementation so that the time spent cycling is reduced, or by eliminating the cycle altogether through a structural change or a scheduling change.

If a system is not structurally unscalable—for example, if its scalability is not limited by its address space—its load scalability might be improved by mitigating the factors that prevent it from being load scalable, space scalable, or space-time scalable. The first steps to improvement are

- Identification of unproductive execution cycles and their root causes, as well as means of breaking them.
- Understanding the sojourn times (the time spent) in the unproductive cycles, and the means of shortening them.
- Understanding how and whether a system could migrate into an undesirable absorbing state (such as deadlock) as the load increases, and devising scheduling rules and/or access control mechanisms to prevent this from happening.
- Identifying system characteristics that make performance measures self-expanding, and finding ways to eliminate or circumvent them.

- Determining whether scalability is impeded by a scheduling rule, and altering the rule.
- Avoiding structural issues, such as software bottlenecks, that cause idleness of resources even when there is work to be done by them.
- Understanding whether asynchronicity can be exploited to allow parallel execution, and modifying the system accordingly. Notice that when evaluating the increased benefit of parallelism, one must also account for the additional cost of controlling interprocess communication.

Not all of these steps are applicable to all situations. Nor can we be certain that this list is complete. But, once these steps have been taken, one may attempt to identify design changes and/or system improvements that either reduce the sojourn times in the cycles or break the cycles altogether. This must be done with care, because any design change could (1) result in the creation of a new set of cycles and/or (2) result in the creation of a new scheduling rule that induces anomalies of its own. Moreover, a modification might simply reveal the existence of another bottleneck that was concealed by the first one. On the other hand, modifications may well lead to unintended beneficial side effects such as reducing resource holding times and queueing delays. Experimental work by Hashemian et al. [HAKC2013] shows that changing platform configurations from their defaults can increase the maximum throughput that can be supported while maintaining given levels of average response times or percentages of response times exceeding a given level, depending on the nature of the workload. For a TCP/IP-intensive workload, multiple cores were best exploited by binding transactions to particular cores, since this reduces data migration from one cache to another. By contrast, spreading network interrupt handling among all cores rather than confining it to one core resulted in considerable performance improvements, whether the workload was TCP/IP intensive or application (and hence CPU) intensive.

Let us reconsider the load scalability examples in the light of the foregoing:

- Semaphores reduce the occurrence of unproductive busy waiting and hence memory cycle stealing in multiprocessor systems with shared memory. Thus, systems that use semaphores rather than locks are likely to perform better under heavy loads. However,

the solution comes with a cost, the overhead of managing semaphores. The use of semaphores might also expose a previously hidden bottleneck, namely, lock contention for the head and tail of the CPU run queue (ready list) in the presence of too many processors. This is not an argument against the introduction of semaphores, merely a warning about the next bottleneck that might arise.

- Eliminating collisions in an unswitched Ethernet LAN clearly increases capacity for a given bandwidth. Unlike the Ethernet, the token ring provides an upper bound on packet transmission time, albeit at the cost of waiting one's turn as the token moves from one node to the next.
- The museum checkroom example illustrates many facets of scalability. Giving priority to customers collecting coats in the museum checkroom reduces the average number of occupied hangers. This contributes to the reduction of delay by reducing the time to find a free a hanger. This in turn reduces the time the checkroom attendants spend on each visitor, and maybe even the number of attendants required to maintain a given level of service quality. It also prevents deadlock. Indexing the free and occupied hangers reduces search time, though not retrieval time, since the desired occupied hanger is always brought to the front for access. Deploying attendants to the carousel where they are needed instead of assigning them to particular ones increases their utilization. Allowing them enough space to move around freely also reduces customer service time. Dedicating one attendant to coat retrievals reduces hanger occupancy while making scarce hangers available sooner. All of these modifications improve the ability of the system to function properly at increased loads, either by cutting down on active processing time or by reducing the holding times of passive resources.

11.8 Some Mathematical Analyses

Simple mathematical analyses can sometimes be used to explore performance issues in load scalability. In the first example, we explore ranges of parameters to understand the extent to which semaphores

can reduce memory cycle stealing and lock contention. In the second example, we use simple expressions for delays to understand the drivers of self-expansion when there is a common FCFS queue for the elements of a shared resource pool, such as hangers in the museum checkroom.

11.8.1 Comparison of Semaphores and Locks for Implementing Mutual Exclusion

In this section we propose a framework for analyzing the relative performance impacts of different mechanisms for implementing mutual exclusion. In particular, we shall compare locks and semaphores, as in our previous example. A semaphore does not eliminate lock contention entirely; it simply focuses lock contention on the shared data structures accessed by its primitives while preparing to put a process to sleep until a critical section becomes available. This mechanism could be applied to protect data in shared memory or, as might be the case with a database record, on disk. Let L be the lock used to implement mutual exclusion without semaphores, and S be a lock that is used to impose mutual exclusion on the data structures used to implement a semaphore operation. In the database example, L could reside on disk before being loaded into memory as part of a record. The lock used by the semaphore might protect a ready list or queue; in any case it is accessed only by designated primitive operations that are part of the kernel of the host operating system.

Let p_i denote the probability that a lock of type I (= L,S) is accessed and set (or reset) successfully on the current attempt. Let a_i denote the number of successful attempts that must be made to acquire and relinquish control of a critical section. For a simple lock, $a_L = 2$ since a test-and-set or a test-and-reset would be required for acquisition and release respectively. We make the simplifying assumption that the probabilities of success on each attempt to access the lock are identical, and that successive attempts are independent. The probability of success depends on how the lock is used, the number of processors, their loads, and how much context switching is occurring at the time. The need to make at least one attempt and the assumption that successes are i.i.d. (independently, identically distributed) give the number of attempts needed for success a displaced geometric distribution with mean $1/p_i$. If we assume that the costs of accessing locks S and L are the same, and that some constant overhead k is associated with the semaphore

mechanism, the ratio of the costs of the simple locking mechanism to those of the semaphore mechanism is given by

$$f(p_L, p_S) = (a_L / p_L) / [k + a_L / p_S) \tag{11.1}$$

We use semaphores because we expect p_S to be a good deal larger than p_L, but we do not know what values these probabilities take in practice. Therefore, we have evaluated f over a large subset of the unit square, (0,0.95] × (0,0.95], for $a_L = a_L = 2$ and $k = 0,5,10$, as shown in Figure 11.5. The scale of probabilities on the long horizontal axis is repeated to allow the simultaneous display of three surfaces, one for each value of the semaphore overhead analyzed.

As one might expect, when the probability of successfully obtaining a lock is high for both locking mechanisms, there is not much to choose between them. As p_L tends to zero, the cost of the simple locking mechanism is many times that of the semaphore mechanism. This means that the onset of problems with load scalability is much less likely when a semaphore mechanism is used than when a straight locking mechanism is used, and that the performance of an implementation

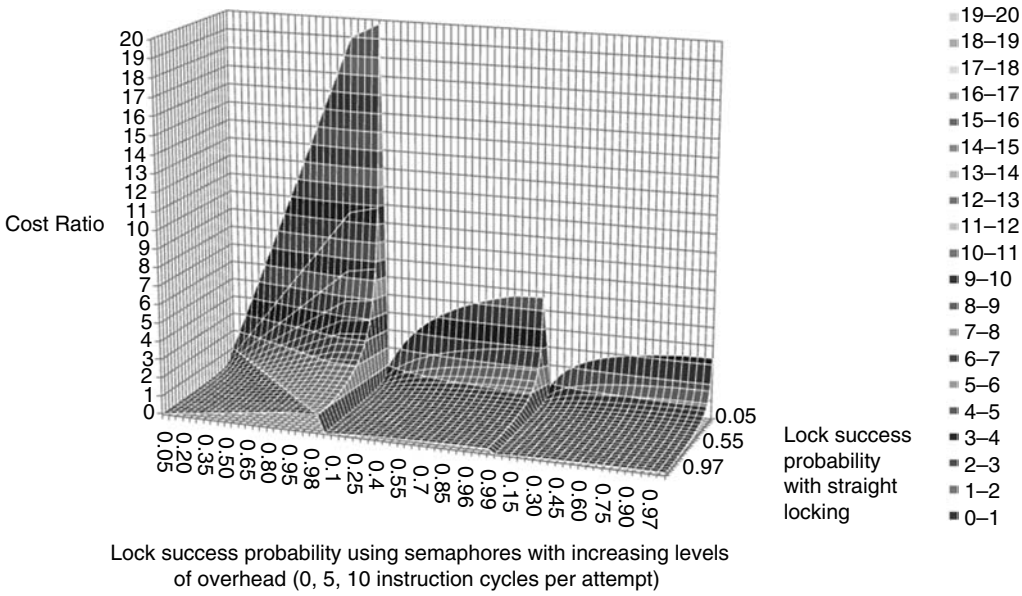


Figure 11.5 *Cost ratios of the expected number of lock attempts with straight locking and semaphores*

using the former is much more robust than that of a system using the latter. To enhance scalability, we would choose the mechanism whose performance is least sensitive to a change in the operating load. In this instance, the semaphore mechanism is the better candidate, as prior literature has led us to expect [DDB1981].

11.8.2 Museum Checkroom

We have argued that the average delay in queueing for a coat hanger will be minimized if those collecting coats are given head-of-the-line (HOL) priority over those leaving their coats, because this minimizes the average hanger holding time. This is also the only discipline that avoids self-expansion of queueing delay with respect to hanger holding time.

Recall that our objective is to maximize the time available to look at exhibitions while avoiding deadlock on contention for hangers. The sojourn time in the museum may be decomposed into time spent waiting to deposit a coat D, exhibit viewing time V, and coat pickup time T. The hanger holding time H is given by

$$H = T + V$$

and the sojourn time in the museum is given by

$$S = D + T + V$$

Since the museum is open for only a finite amount of time each day, M say, we immediately have

$$S, H \leq M$$

Since we cannot control the behavior or preferences of visitors, we cannot control V. Still, one should allow V to be as large as possible by reducing at least one or both of D and T, because this is what the visitor came to do, and because it allows more time to visit gift shops, coffee bars, and so on within the museum.

Because increased hanger holding time H increases the risk of deadlock while increasing both D and T, the queueing times to leave and collect coats respectively, we should first focus on reducing T in order to reduce H, and hence eliminate self-expansion. Under FCFS scheduling, customers collecting coats and leaving them delay each other. This increases H by increasing T. If N denotes the maximum number of hangers and λ the visitor throughput, we must have $\lambda H < N$ for the

checkroom queue to be in equilibrium. Hence, increasing T tends to make the queue unstable by increasing H. Of course, T could also be reduced by speeding up service, but that would only increase the value of λ for which saturation might occur; it cannot prevent deadlock altogether.

Let c denote the number of customers in queue to collect a coat, and a the number of customers in queue to leave a coat at the time the next customer arrives to pick one up. Let s denote the service time. If the queue is FCFS and there is only one attendant, the latest customer must wait

$$T = (c + a + 1)s$$

to obtain a coat. This is an upper bound on T if more than one attendant is present. If those collecting coats are given HOL priority, we have $T \leq (c + 1)s$. This clearly reduces the upward pressure on H, and hence D and T. This shows that using HOL eliminates self-expansion and hence reduces the tendency to saturate the hangers. This policy is optimal with respect to the average waiting time of all customers, regardless of whether they are collecting or leaving coats, because it reduces to shortest hanger holding time next.

Not all customers may understand the optimality of this policy. If customers collecting coats temporarily outnumber customers leaving them or vice versa at certain times of the day, it may be worthwhile to convince the minority that they are not being ignored by ensuring that they are served one of every k times. This rule must be applied judiciously, because it allows self-expansion. Hence, k must be chosen with care. If too many customers leaving coats are served ahead of those collecting them, the resulting self-expansion in the hanger holding time could lead to deadlock, or at the very least an increase in the average values of D and T, as well as of H. A policy for determining k in response to changing conditions is beyond the scope of this book.

11.9 Avoiding Scalability Pitfalls

The preceding examples show how unproductive cycles and poor choices of scheduling rules can undermine scalability. Poor architectural choices and restrictions on protocols can do this also.

The Louvre's use of separate facilities to handle coats and umbrellas on one side of the entrance hall and backpacks and hats on the other may increase the capacity of each by limiting functionality and hence

reducing service time variability and perhaps even space requirements. On the other hand, the system imposes an extra burden on visitors with backpacks and coats by requiring them to queue four times per visit instead of twice (once on arrival, once on departure). Depending on the weather, this could increase congestion in the main hall.

The excessive use of performance antipatterns can undermine system performance and scalability [SmithWilliams2000]. For example:

- A "god class" is a system element or component that interacts with many others, perhaps in a single-threaded way. Such a class can become a hardware bottleneck simply by virtue of the amount of traffic going through it. If it is single-threaded, it will also become a software bottleneck. Domain name servers and authentication servers are potentially susceptible to this problem, the former because all IP addresses must be resolved through them, and the latter because of the volume of activities that must be vetted by them.
- A database query that involves searches through a long sequence of tables is known as a *circuitous treasure hunt*. Scalability is undermined simply by the number of searches involved.
- Where several threads are simultaneously engaged in circuitous treasure hunts involving the same set of tables, scalability is further undermined by the extent to which the repeated locking of one or more tables simultaneously could cause delay by reducing parallelism.
- The use of a common path by two or more classes of jobs, processes, or threads that cannot be present on it simultaneously is sometimes called a *one-lane bridge*, because vehicles can travel on it in only one direction at a time. Those traveling in the opposite direction must wait until vehicles present on the bridge have left it. Moreover, a scheduling rule must be applied to ensure that a stream of vehicles traveling in one direction does not prevent the timely passage of a lone vehicle traveling in the opposite direction.

The adverse impacts on performance and load scalability might be mitigated as follows:

- If data that is "owned" by the god class is only being read and not being modified, the class can safely be replicated and threads distributed among the replicates. For instance, multiple copies of domain name servers and authentication servers could be used, because their contents seldom change.

- To avoid a circuitous treasure hunt, indices might be built to reduce the number of pointers that must be traversed to implement a query.
- To enhance concurrent execution, row-level locking is preferable to table-level locking if there is no dependence between rows, and if the cost of allocating locks to individual rows is not excessive. In any case, it is good programming practice to ensure that the smallest number of abstract objects is ever locked simultaneously. Furthermore, all threads should acquire locks in the same sequence and release them in the reverse sequence to prevent deadlock caused by cyclic dependencies.
- As with the god class, the impact of a one-lane bridge can be mitigated by replicating objects that will not be modified and minimizing the holding time of shared objects that might be modified. This is analogous to shortening a one-lane stretch by narrowing the road as close to the bridge as possible.

These mitigations all involve the redistribution or rescheduling of work so as to avoid queueing delays and to enable the concurrent use of resources. The resources may be active, as is the case with processors, or passive, as is the case with locks, records, memory slots, and other objects that do no processing but are points of contention nonetheless. As we saw in our discussion of asynchronous I/O in Chapter 10, rescheduling and redistribution do not reduce the amount of work that must be done, but they can help to maximize the timely utilization of the resources that are available to do it. That can reduce the time to overall completion.

Redistribution of work to achieve scalability requires care and attention to detail. For example:

- While requiring museum patrons to visit multiple checkrooms to deposit disparate objects may increase the throughput and storage capacity of each checkroom, it could be seen as imposing an undue burden on visitors who have difficulty getting around, such as the elderly. An analogy with our conveyor system example may be that the data associated with a parcel might have to travel with an associated transaction from one node to another, thus increasing network loading.
- Post offices sometimes require all customers to join a single queue to reach any of several windows providing identical services along a single counter. One of the benefits of this is that a customer requiring prolonged service need not delay the next

one in line excessively. On the other hand, delays can be unwittingly introduced if many of the windows are so far from the head of the queue that an elderly customer might take a very long time to walk to the next available one. To see this, consider that the agent at this window will experience enforced idleness until the slow-moving customer arrives. Part of the benefit of parallel service by multiple agents has been lost because of staging delays. The computing analogy here is that a service thread, for which tasks must queue, might not be able to commence execution until all data needed for the work at hand has been loaded into memory.

Redistribution and rescheduling cannot be implemented without suitable architectural and protocol support. If the work associated with a transaction is broken into several tasks that could be executed in parallel, data structures and fields are needed in task control blocks to bind the parallel tasks to the originating transactions so that the results may be delivered to it. Similarly, if priority scheduling is to be implemented, there must be support for marking the tasks as having one priority level rather than another. There must also be distinct logical queues for the different priority levels.

11.10 Performance Testing and Scalability

In Chapter 9 we discussed how performance tests might be planned and executed to verify the scalability of a system. We showed how one verifies scalability by ensuring that systems are well behaved, in the sense that the utilizations of such resources as processors, disks, and bandwidth increase linearly as the offered load is increased. We also illustrated impediments to scalability such as software bottlenecks and the presence of errors in concurrent programming. In the present chapter we have described system characteristics that prevent resources from being fully utilized, that waste them with needless repetition of unproductive cycles of execution, or that have long queueing delays because of serial execution when parallel execution on disjoint data objects is feasible.

When reviewing the results of performance tests to see how scalability might be improved, we should not only be asking how resources are being used, but what might be preventing them from being used if their utilization is low. We should be open to the possibility that the

results will reveal opportunities for improved scalability through the improved use of resources as well as the avoidance of impediments because of wasted resource usage. Once an opportunity for improvement or the cause of an anomaly is suspected, such as busy waiting on locks or inappropriate values for configuration parameters, experiments should be planned that explore how corresponding implementation changes might yield performance improvements while improving load scalability. The results in [HAKC2013] illustrate that the use of default configuration settings in a web-based system can result in reduced resource utilization, while changes to those settings that are tailored to resource usage patterns inherent in the workload can improve resource utilizations considerably without adversely affecting response times. The experiments described in [Horikawa2011] show that changes to locking strategies can improve the usage of multiple cores while reducing wasted CPU cycles, but that there are inherent limits to the extent of improvement. The goal of an experiment aimed at improving scalability should not only be the identification of improvements to the performance of a system with a particular workload. Configuration parameters or design choices should be sufficiently varied to allow the performance engineers and other stakeholders to identify strategies for improvement that are robust when actual workload differs from the one that was tested, too.

11.11 Summary

At the beginning of this chapter, we pointed out that the term *scalability* is vague. Attempts that have been made to define it may differ in semantics or scope, but they all relate to enabling a system or family of systems to gracefully grow or shrink to accommodate changes in demand and the number of objects within their scope. The various definitions have been attempted so that one can determine the extent to which scalability is a desirable quality attribute that confers technical and commercial advantages on a system's stakeholders.

To place scalability on a solid footing, one should view it in the context of performance requirements and performance metrics. As we saw in earlier chapters, performance requirements should be linked to business and engineering needs so that the cost of meeting them can be justified. The performance requirements, whether present or anticipated, can guide us in determining the extent of scalability that is required to meet stakeholders' needs.

It is also useful to identify features of a system that limit its scalability as well as capabilities that will mitigate the limits and perhaps enhance scalability. A scheduling rule that prevents deadlock can also reduce average system response times, while mechanisms that avoid unproductive cycles enable the efficient use of resources, thus potentially increasing overall system capacity. Frequently, the scalability of a system and the ability to reduce its response time are limited not by hardware constraints, but by constraints inherent in the software architecture, such as single threading. Because the demand for a successful system may increase, it is wise to architect it so that its scalability will not be needlessly impeded by a poor choice of scheduling algorithms or by the use of structures that inherently prevent the use of parallel cores or processors, or by the use of performance antipatterns such as god classes. While attending to these aspects of system design and architecture will not guarantee that a system is scalable to the extent needed, doing so will reduce the scalability risk inherent in any system.

11.12 Exercises

11.1. In a performance test, transactions are sent to an online transaction processing system at regular intervals. The CPU utilization of this system increases quadratically with time, while the size of the swap space and the amount of occupied disk space increase linearly with time.

(a) Explain why this system has poor load scalability in its present form.

(b) Identify a simple data structure and algorithm that might be in use in this application. If the development team confirms that this is indeed what is being used, explain what should be used in its place to prevent this problem.

11.2. You are participating in an architecture review of a computationally intense system in which successively arriving transactions operate on completely disjoint sets of data. Upon arrival, an arbitrary number of transactions may go through some preprocessing concurrently. Once the preprocessing is complete, each transaction joins a queue for a software component that can process only one transaction at time, because it is single-threaded. The application is hosted by a system

that supports paged virtual memory and has multiple cores. The paging device and the application data are stored on separate drives.

(a) Explain why this application will have poor load scalability.

(b) Describe the instrumentation you would use to measure the resource usage of the system over time in a Windows environment and in a UNIX or Linux environment.

(c) Suppose that load is offered to the system at 1 transaction per second for 15 minutes, then at 2 transactions per second for the next 15 minutes, then at 4 transactions per second for the next 15 minutes, and so on for 2 hours. Describe how the utilizations of the individual processors might evolve during the test given the description of the computation. Also describe how you might expect the transaction response times to evolve.

(d) Propose a design change that will improve the load scalability of the system.

11.3. The following changes are proposed for a database system: (i) using semaphores rather than busy waiting for all locking; (ii) applying locks so that each process or thread locks the minimum number of objects (tables, rows, etc.) simultaneously, whether the locking is implemented using busy waiting or semaphores; (iii) using row-level locking rather than table-level locking whenever possible.

(a) Explain the performance impact of each of these changes with respect to processor utilization and concurrent execution.

(b) In some database platforms, a semaphore must be created or allocated every time a thread initiates the first access to a critical section. The semaphore is released when no thread is accessing the object being guarded by the critical section. To reduce the overhead associated with the creation and deletion of semaphores, someone suggests that they be kept in a pool for use as needed. Describe the performance impact if the pool is too small. How would you evaluate the trade-off between dynamic creation of semaphores and the use of a pool of fixed size in a system with a high volume of locking and unlocking? Explain what aspects of scalability are involved.

This page intentionally left blank

Chapter 12

Performance Engineering Pitfalls

Choices of scheduling policies or the use of new technologies are sometimes made with the intent of increasing capacity, increasing the sustainable load, shortening average response times, or pleasing one or more stakeholders or constituencies. In many of these cases, the proposed modification will not result in the achievement of the stated performance goal. Indeed, some performance engineering choices may cause undesirable side effects or even worsen performance, while incurring considerable implementation and testing costs. The introduction of priority scheduling can lead to the starvation of lower-priority tasks and have other unintended side effects. Adding processors can sometimes worsen performance. Spawning all tasks of a specific type as threads within a single process or virtual machine can limit parallelism and diminish system reliability. Physical limitations on the potential instruction rates of individual processors will make it ever more necessary to use concurrently executing processes and threads to shorten the total execution times of applications and to increase system throughput. Even then, the individual threads of execution must be implemented with performance consideration in mind if the greatest use is to be gained from available processing, network, and I/O resources. Virtualized environments have performance measurement and engineering issues of their own, which we shall also explore. Finally, we consider organizational pitfalls in performance engineering, including the failure to collect or review data about the performance of systems in production.

12.1 Overview

As performance engineers, we are often confronted with stakeholders who believe that the introduction of a particular scheduling policy or the use of a new technology must inevitably bring about performance improvements. Many performance practitioners have encountered or read about cases in which the well-intentioned use of a scheduling rule has had consequences for performance and service quality that might not have been previously imagined. In previous chapters we described cases in which the introduction of new capacity, such as adding processors, degraded performance or did not provide performance gains that were commensurate with the cost of the new hardware or other technology. In this chapter we shall explore some of these potential pitfalls.

We will see that while priority scheduling can provide performance benefits in some situations, it can be detrimental or be of no benefit in others. It cannot increase the capacity of system resources such as processors, I/O devices, and network bandwidth. As we saw in Chapter 11, it can have unintended consequences. As we saw in Chapter 10, asynchronous activity can shorten total execution times, but it cannot in and of itself increase system capacity. The use of multiple processors and cores can degrade system performance by increasing memory bus contention and lock contention, so the number of processors competing for these resources must be chosen carefully. While the automated garbage collection used in programming languages such as Java might relieve the programmer of the responsibility to free up unused objects, its spontaneous occurrence with its associated processing cost can seriously degrade system performance when capacity is most needed. Finally, we briefly examine the performance engineering pitfalls of virtual machines. These are intended to provide contained environments for execution in server farms. They can also be used to provide contained, isolated environments for functional testing. Their use for performance testing is questionable, because there is no way of mapping resource consumption time in the virtual environment to corresponding values in a physical environment.

12.2 Pitfalls in Priority Scheduling

Priority scheduling has many uses, but increasing the processing capacity of a system is not among them. This is because the payload work

that must be done is always conserved. Indeed, preemptive priority scheduling can increase the processor utilization for a given offered load, because it triggers a context switch to start the execution of a higher-priority task, and a second context switch to restart the execution of the lower-priority task once it has finished using the processor.

- *Pitfall:* Not tailoring priority scheduling to the traffic situation and its requirements.
- *Reason this is a problem:* There may be unintended consequences, including performance degradation. It could lead to deadlock, livelock, or starvation.
- *Mitigation*: Give priority only to activities of short duration.

Priority scheduling can have two benefits. First, there is the obvious one of reducing the average waiting time of the process or thread that is given priority. A second, and no less important, benefit is that discrete resources bound to a high-priority job, such as locks and objects drawn from pools, can be released sooner for use by others. The principle of giving priority to jobs in possession of scarce resources is applicable to the handling of I/O interrupts. These should be given priority for processing over application code so that I/O buffers can be freed quickly. In everyday life, the principle is at play on roundabouts in the United Kingdom (known as traffic circles in the United States). Traffic on the roundabout has priority over traffic attempting to enter it, so that the vehicles that are on it can free space for those wishing to enter. Reversing those priorities could lead to a traffic jam because a stream of entering traffic at an entry point could prevent cars from reaching the following exit point, thus causing a potentially very large backlog at some other entry point. In France, where traffic approaching to the right normally has priority at intersections, drivers entering roundabouts pass signs in the form of an inverted triangle with a red border bearing the legend "*Vous n'avez pas la priorité*" (You do not have priority). Giving priority to traffic on the roundabout rather than traffic from the right avoids congesting the roundabout, thus mitigating the effect of a rule that would be detrimental to the smooth flow of traffic if it were universally applied.

In Chapter 11 we saw how giving high priority to the processing of inbound TCP packets is useful for freeing up space for receive buffers, but doing so delays the transmission of acknowledgment packets. Under extreme loads, the delay is long enough to prevent acknowledgments from being received by the sender early enough to prevent retransmission of the apparently unacknowledged packets on timeout. This leads

to a vicious cycle that could be mitigated, if not entirely prevented, by the use of a fairness scheduling algorithm, such as cyclic nonexhaustive service, that would place upper bounds on the delays in processing the first incoming and outgoing packets. Another side effect of the vicious cycle is the prolonged residence of unacknowledged packets in the sender's transmission buffer. While the transmission buffer might not be viewed as being as scarce a resource as the physical receiving buffer at the server, the penalty of letting it fill up is extra traffic due to retransmitted messages, with effects that could propagate along the possible physical paths between the sender and the receiver. Here, we see that priority inversion is a pitfall because of cyclic dependency.

In the museum checkroom example in Chapter 11, we saw how giving priority to a process that frees an occupied resource reduces the holding time of that resource while eliminating the possibility of deadlock when requests to acquire and free the resource must pass through a common FCFS queue. From the material in Chapter 3, we know that the average resource occupancy in this case is equal to the throughput multiplied by the holding time. Thus, because priority scheduling shortens that average holding time, the constraint on system throughput due to the number of discrete resources is potentially relaxed. The principle of giving priority to the freeing of discrete resources also applies to giving higher processing priority to the completion of I/O requests, since this frees the I/O buffer in a designated memory location in which a piece of data recently read from disk is stored, or in which a piece of data to be written to disk is stored. This is one of the main applications of priority interrupts, and one of the main reasons that operating systems in kernel or privileged mode are given CPU priority. Preemption to complete an I/O via a priority interrupt is an example of Last Come First Served Preemptive Resume (LCFSPR), a discipline that can be found in many operating systems.

One of the reasons that giving CPU priority to privileged operating system functions such as I/O is effective is that they usually use a small fraction of the available CPU time. Priority scheduling is of little use when it is used to give preferential treatment to a function that consumes a very large share of the resource being requested. In that case, lower-priority tasks may be unduly starved of CPU. The starved processes will hold passive resources such as memory partitions and locks much longer than they would otherwise.

Frequent fliers may have had the experience of being on a flight in which almost all passengers have elite or priority status. Under those conditions, no benefit is conferred by having elite status. Indeed, allowing

priority boarding to elite passengers may be counterproductive, since passengers with assigned seats near the front of the plane will block the access of those seated in the rear if they board first. This will increase the total boarding time for all passengers. In that case, allowing passengers whose seats are in the rear to board first will speed departure for all. This is a case where priority scheduling is detrimental to performance, because it can seriously delay the departure of the aircraft.

When considering priority scheduling, one must distinguish between preemptive priority and head-of-the-line (HOL) priority. Under preemptive priority, a high-priority job interrupts the service of a lower-priority job and seizes control of the server until it is done. The higher-priority jobs at this server are oblivious to the presence of the lower-priority jobs. The lower-priority job can resume execution only if no other higher-priority job has arrived in the meantime. Under HOL priority, also called *nonpreemptive priority*, an arriving job of higher priority must wait until the job in service completes execution regardless of its priority level.

- *Pitfall:* Assuming that priority scheduling increases system capacity.
- *Reason this is a problem:* Work is conserved. Giving priority ensures earlier access to a resource for only some classes of jobs or threads.
- *Mitigation:* Assess the total load, and avoid giving priority to a class of jobs that would dominate usage of the resource.

Let us now consider some conservation properties. First, the Utilization Law tells us that the utilization of a server or device is equal to the average throughput multiplied by the average service time. Thus, the total utilization by jobs of all priority levels is invariant with respect to priority ordering. Second, it can be shown that in systems with nonpreemptive priority in which the service times of the queued jobs are not taken into account when deciding who will be served next, the average waiting time among all jobs of all priority levels, weighted by their respective traffic intensities, is invariant with respect to the priority ordering. This is intuitively appealing, because giving higher priority to one job class to reduce its average waiting time increases the average waiting times of jobs of lower priority, as one might expect [Kleinrock1976]. A related conservation rule applies to preemptive priority systems under certain restrictive conditions [Cobham1955, BuzenBondi1983]. As Kleinrock writes about priority queueing [Kleinrock1976], "You don't get something for nothing."

The situation is not so straightforward when the buffer or waiting area is finite. We consider the case of two priority classes with nonpreemptive priority service and common or reserved waiting areas. With a common waiting area for both priority classes, if the mean service time is the same for both priority classes, arrivals are Poisson, and the service time is exponentially distributed, the distribution of the total number of packets or jobs in the buffer is independent of the priority ordering. The throughput of the high-priority packets suffers as their arrival rate increases, because the buffer fills with low-priority packets that crowd the high-priority packets out of the buffer. The throughput of the low-priority packets is also degraded by this crowding. Dedicating some of the buffer space to low- or high-priority packets may not necessarily improve the packet throughput for either priority class [Bondi1989].

12.3 Transient CPU Saturation Is Not Always a Bad Thing

Suppose that the initiation of a transaction results in the CPU utilization rising to 100% for a short amount of time before the utilization falls back. Consider also the possibility that a burst of transactions in an embedded system might result in the utilization of its single processor being 100% for a limited but noticeable amount of time. Some might consider these periods of saturation to be a cause for concern, but this need not be the case.

- *Pitfall:* Averaging performance measures over time intervals that are too short.
- *Reason this is a problem:* Unnecessary concerns might be raised about fluctuations in resource usage that are normal consequences of the workload.
- *Mitigations:* Determine whether the observed irregular behavior prevents performance requirements from being met. Take averages over periods that are long enough to smooth irregularities but short enough to reveal trends.

We first consider the case of a single transaction causing CPU saturation for a limited amount of time. We refer to this time as the *saturation epoch*. The duration of the saturation epoch must be compared with the desired response time of the transaction in question. If the duration of the saturated period exceeds the response time requirement, there is certainly

cause for concern. If no other demands will be made of the processor before the period of saturation has passed, there is no cause for concern if throughput and response time requirements are being met, provided one is satisfied that the processor utilization can be attributed to useful work and is not caused by infinite looping. The duration and reasons for the saturation epoch should be considered before effort is invested in trying to eliminate it. If the CPU is saturated under constant load for the entire length of a test run, the response time may be excessive.

Recall that a process or thread that is not interrupted will hold the CPU until it needs to perform I/O. If no I/O is required, the CPU will be 100% busy. The only thing that matters here is how long this busy epoch will last. There is far more cause for concern if execution halts before the work has been completed, because it could indicate that data is missing or be a sign that interacting threads have gone into deadlock. If the product of the duration of the busy epoch and the desired throughput is less than the number of processors that will be used by the task, there is no cause for concern. Intervention is needed only if the response time requirement is exceeded. Even if multiple processes are contending for the processors, saturation is not a cause for concern unless the throughput and response time requirements cannot be met for the task that is saturating the CPU, or for other tasks, unless other processes or threads will also need to use it, unless an increase in load is anticipated, or unless there is concern about more urgent tasks being deprived of access to resources. In the last case, CPU priority scheduling can be used to allow urgent tasks to execute if the operating system supports it.

The time scale of measurement is important in the evaluation of this situation. Recall that the CPU utilization is taken to be 100% if the idle loop is not executing and 0% if it is. An average utilization of 100% can always be observed if the measurement period is short enough. There are two lessons to be learned here:

- Before embarking on an expensive investigation of why the processor is 100% for a transient amount of time when load is applied, it is worth determining whether the high utilization is preventing any performance requirements from being met. If it is not, other causes for concern should be identified before technical staff members are assigned to finding out why. Otherwise, an incorrect interpretation of performance metrics will lead to a waste of staff time that could be devoted to solving other problems.
- The utilizations should be averaged over longer periods of time. This will smooth out the effect of the utilization spikes.

If the overall level of utilization is still too high, further investigation may be warranted. Measuring over too short a period may raise unnecessary concerns.

12.4 Diminishing Returns with Multiprocessors or Multiple Cores

It is a mistake to suppose that the capacity of a system will increase linearly with the number of processors in a server, or that adding processors will always reduce system response time. Many factors can reduce or even negate the benefit of adding additional processors.

- *Pitfall:* Increasing the number of processors does not necessarily increase system capacity. Indeed, it can reduce it.
- *Reason this is a problem:* Money might be invested with the intent of mitigating a problem without having the desired effect. It might even have a negative effect.
- *Mitigation:* Consider the architecture of the system before making the investment.

The following examples illustrate this pitfall:

- The head of the ready list or run queue, the list of processes that are ready for execution, is shared among all processors and therefore must be protected by a lock to ensure mutual exclusion. Contention for both the lock and the ready list head will cause memory cycle stealing and memory bus contention. In extreme cases, adding processors can make performance worse. This was discussed in Chapter 11 and is discussed further in [DDB1981].
- Each processor has its own cache. If multiple processors share a copy of the contents of a data location, there will be a copy in each processor's cache. These copies must be kept consistent. Memory bus contention to achieve this will cause delays in updating the cache copies, which will slow down execution. Therefore, data sharing among multiple processors should be kept to a minimum [Gunther1998].
- If the executing application is single-threaded, it can use only one processor. The presence of multiple processors in a host will not aid the processing of this application, because it cannot exploit parallelism. This is a sign of a software bottleneck, as we have seen in earlier chapters.

- We have already seen in Chapter 10 that adding processors or using multiple cores can increase the system response time if concurrently executing threads that access shared objects are not properly synchronized, or if thread safety is not properly implemented. The problem becomes apparent because the thread interleaving enabled by using multiple processors enables different sequences of events to occur from those in a single-processor system. Some of these sequences may lead to error conditions that cause retries and even crashes.

12.5 Garbage Collection Can Degrade Performance

Garbage collection is used to free memory blocks containing objects to which there are no longer any valid pointers. It is part of the implementation of interpreted languages such as LISP and languages with built-in memory management such as Java and C#. It can be used in compiled languages that support dynamic storage allocation, including compiled implementations of Java. It is not used in older languages such as FORTRAN which have only static storage allocation. Garbage collection removes the responsibility for the cleanup of unused memory from the programmer. Since control is removed from the programmer as well, there is little or no control over when it occurs. Because collection takes up both processing time and memory, it will degrade performance. In some implementations of Java that allow execution of threads within the same virtual machine on multiple processors, the execution of all of the threads will be suspended while collection is in progress. During this time, the benefit of execution on multiple processors will be completely lost.

The literature on garbage collection is vast. Correct tuning of garbage collection parameters such as the heap size at which collection is triggered depends on the application and on the collection algorithms themselves [BCM2004]. The reader is encouraged to look at what has been written most recently about the implementation used in the system of interest.

12.6 Virtual Machines: Panacea or Complication?

A pitfall with the measurement of resource usage by threads and processes in virtual machines is that the utilization per process is measured relative to the time that the virtual machine possesses the resource

rather than with respect to physical (wall) clock time. The resource usage may or may not reflect contention by other processes running in different virtual machines.

- *Pitfall:* The resource utilizations indicated by virtual machines may not be true indicators of resource utilization.
- *Reason this is problem:* Based on this incorrect data, the system could be modified in a way that makes matters worse.
- *Mitigation:* Measure the system in isolation on a real machine to avoid confounding.

Virtual machines are programs that can mimic diverse operating systems on a single host. For example, they can emulate UNIX, Linux, and Windows environments while keeping each logically hidden from the others. They are useful for functional testing because they provide contained environments that prevent programs from running amok and interfering with the memory address spaces of other programs. If a virtual machine hangs, that is, ceases functioning, it does not disrupt the operation of its host. It merely stops. Other virtual machines on the host can continue to execute. Virtual machines are also seen as attractive because they can collectively make use of idle processing power in a host, even if the applications running within each one are I/O bound. At the same time, the processes within a virtual machine are invisible to those in other virtual machines. This provides privacy and protection in a shared environment which might otherwise not be available.

Scheduling and synchronization of processes and threads can occur only within the context of a virtual machine. Operating system constructs cannot be used to implement interprocess communication across virtual machine boundaries. It might be possible for virtual machines to communicate with one another via TCP sockets or other network devices, but synchronization would be enforced entirely at the application level in that case, and not at the operating system level. It may therefore be difficult to detect bugs attributable to faulty or mismatched communications between processes in different machines by referring to the resource usage measurements of processes and threads alone.

A process might occupy 100% of the processing time available to a virtual machine, but not necessarily 100% of the processing time relative to the wall clock time. If a virtual machine is starved of access to the CPUs by other virtual machines operating on the same host, the physical processor utilizations of a process would be per-process utilization measured within the virtual machine multiplied by the physical processor utilization of the containing virtual machine itself.

Because the operating system within one virtual machine is totally oblivious to the state and presence of processes in other virtual machines, processes and threads can be scheduled only within a virtual machine. Scheduling the execution of virtual machines on the hosts does not control the scheduling of entities operating within them. In some cases, a virtual machine may not even be able to distribute the concurrent execution of threads and processes within it among the physical processors or cores of the host. For example, early Java virtual machines could support multithreading, but the threads could execute only on the processor on which the virtual machine itself was executing. The result was a software bottleneck in which processors remained idle despite the presence of a backlog of threads that were ready to run. In a test environment, the confinement of threads to one processor might be considered an advantage if threads are spawned in an infinite loop without being destroyed. Although their virtual machine might crash, its threads would not compete for all of the processing power within a host, thus reducing the risk of total disruption of the execution of the other virtual machines. Thus, the inability of a virtual machine to exploit parallel processors could be seen as having the deficiencies of its merits.

12.7 Measurement Pitfall: Delayed Time Stamping and Monitoring in Real-Time Systems

In previous chapters we discussed how discrepancies between system clocks on different hosts and clock drift could lead to spurious measurements of response times. In systems that frequently track measurements of such quantities as speed, displacement, temperature, and pressure, there is always a risk that processing delays or contention for memory could lead to perplexing apparent discrepancies between sets of system measurements, or between the measurements and the time stamps.

- *Pitfall:* Time stamps are delayed and do not always agree with the clock.
- *Reason this is a problem:* Spurious inferences about delays may be made.
- *Mitigation:* Check that the granularity of the time stamp is no finer than the delay in recording it. Use a coarser resolution when interpreting it if necessary.

This could occur, for instance, if values are displayed and/or logged less frequently than they change, or if the time stamp is recorded only when the values are displayed and logged rather than when they were stored in memory or registers. For example, a system that tracks the speed, distance traveled, and other properties of a vehicle such as a train, bus, or car might experience delays in recording these observations and bind the observations to a time stamp only when recording occurs. One result of this might be that the measurements of the physical aspects of the system are inconsistent with one another. Thus, if x_i, v_i, and t_i respectively denote the ith observations of position, speed, and time in a moving object, we might find that

$$x_{i+1} - x_i \neq v_i(t_{i+1} - t_i)$$

even if the speed is more or less constant and the difference between the time stamps is small. If the motion of the vehicle is very steady compared with the granularity of the observations, it may be best to take the observed speeds and distances traveled at face value and treat the time stamps as approximate at best. Analogous discrepancies may arise in system measurements that seem to fail to satisfy Little's Law or the Utilization Law over short time intervals. Unless there are forensic considerations such as a crime or crash investigation, it may be worth smoothing the performance measurements by averaging them over longer time periods to overcome the consequences of clock variability.

12.8 Pitfalls in Performance Measurement

As we discussed in Chapters 8 and 9 on performance measurement and testing respectively, it is essential that performance measurements and the instrumentation and statistical tools used to generate them be repeatedly validated. This is especially true of measurement hooks that are crafted on hosting software platforms, such as tools that measure query response times within database systems and hooks that measure response times in load generation scripts.

- *Pitfall:* The time stamps of job completions may be recorded after completion has occurred.
- *Reason this is a problem:* It could create the impression that the measured response time is longer than it should be when in fact it is not.

- *Mitigation:* Ensure that the process that records job completion is not delayed so long that the measurements are late.

As pointed out in [NagVaj2009] and elsewhere, response times are measured on systems that are computers themselves. It is therefore important to ensure that the collection of the response time is not itself delayed, since this will induce a pessimistic bias in the measured results. This can happen if the load generator has a saturated resource such as CPU, I/O, or an object pool, or if the load generation script is running at too low a level of CPU priority. One does not wish to dispatch a team to fix an alleged response time issue that turns out to have its roots in the instrumentation. This is the reason that we insist on resource measurement in the hosts on which the load drivers are run as well as in the hosts on which the system under test is run. We should require measurement of passive resources within the load generators, such as any pools of objects representing uncompleted transactions, as well as processor, disk, and network bandwidth utilizations. The Response Time Law tells us that high occupancy of objects representing transactions and long response times will impede the generation of transactions at a sufficiently high rate. If the main memory of the load-driving host is very heavily occupied, thrashing may ensue, delaying the recording of response times.

12.9 Eliminating a Bottleneck Could Unmask a New One

Improving the performance of a system bottleneck can often cause a system bottleneck to appear elsewhere, because increasing the maximum achievable throughput of the bottleneck device enables the next most loaded device to become saturated. The saturation level of the next most loaded device is the maximum throughput that can be obtained by improving the original bottleneck.

- *Pitfall:* Increasing the capacity of a system bottleneck results in the appearance of a different bottleneck.
- *Reason this is a problem:* The capacity and performance of the system may not be increased as much as is required.
- *Mitigation:* Perform a bottleneck analysis of the system to anticipate where the next bottleneck might be so that the level of improvement can be anticipated.

The following examples illustrate how eliminating one bottleneck can unmask another:

- Eliminating a software bottleneck by increasing the size of a JDBC pool between the application server and the database server in a three-tier web system may reduce queueing for database queries, only to allow another bottleneck to appear in the database server, such as the exhaustion of the CPU as the transaction rate increases, or the emergence of contention for a lock.
- Adding processing capacity to a system with a saturated CPU may allow an I/O device to become saturated.
- Adding memory to a database server may improve response time by allowing more tables to be held in memory, but the result could also be the emergence of an I/O bottleneck.

Each of these improvements results in a reduction in the utilization of the resource that was the bottleneck, to the extent that some other resource might have the largest utilization instead. That is the new bottleneck. We can anticipate that a bottleneck will shift using the bottleneck analysis techniques we discussed in Chapter 3. Recall from Section 3.7.2 that the demand on the ith device is given by

$$U_i = X_0 D_i, \quad i = 1, 2, ..., K$$

The bottleneck device is the one with the largest D_i. Denote the sorted demands D_k by

$$D_{(1)} \le D_{(2)} \le ... \le D_{(K-1)} \le D_{(K)} = D_b$$

where b is the index of the bottleneck device. We know that the system throughput cannot be larger than $1/D_{(K)}$. If the original bottleneck device is improved to the point that $D_b \le D_{(K-1)}$, the system throughput is now bounded above by $1/D_{(K-1)}$. The bottleneck has been shifted to the next most heavily loaded device. This means that the largest gain in throughput is constrained by the capacity of the next most loaded device in the system. It follows that the factor by which the maximum throughput can be increased is

$$[1/D_{(K-1)}]/[1/D_{(K)}] = D_{(K)}/D_{(K-1)}$$

which is greater than one by construction. Hence, if the CPU is the bottleneck device in the original system, doubling its speed (and hence

halving the demand for processing time) will not double the maximum attainable system throughput unless $D_{(K)} / D_{(K-1)} \geq 2$.

12.10 Pitfalls in Performance Requirements Engineering

One of the riskiest pitfalls in performance requirements engineering is not having any idea of the workload that will be placed on the system. Overestimating the workload will lead to unnecessary expense, while underestimating it will lead to unsatisfactory performance and even commercial failure or, if the system is mission critical, a failure to meet safety requirements. If product managers and other stakeholders are hesitant or reticent about providing estimates of the workload, the lead performance engineer would be well advised to draft a reference workload for requirements engineering and testing purposes so that all stakeholders will have some idea of what load the system could support under the assumed normal and stressful operating conditions.

- *Pitfall:* Uncertainty about the workload and its nature, including memory, response time, and throughput requirements.
- *Reasons this is a problem:* The system might be over- or underengineered, causing unnecessary costs, user dissatisfaction, or even endangerment. One is unable to size the system for the correct market segment.
- *Mitigation:* In the absence of knowledge about the workload or performance needs, develop a reference workload and a reference set of performance requirements.

12.11 Organizational Pitfalls in Performance Engineering

In most of this book we have focused on issues in performance engineering related to portions of the software lifecycle occurring before a system goes into production, such as performance requirements, engineering, architecture, development, and testing. Ownership and oversight of the performance of a system must continue while the system is in production to ensure that the system meets throughput and response time requirements as the load changes, and to enable the orderly

planning of additions to the load, additions of capacity to support the additional load, and the orderly planning of additional functions and the capacity needed to sustain them. Data on resource usage and the performance of applications should be collected and above all tracked so that degradation can be detected and service levels maintained. When new applications are added, the measurement instrumentation should be enhanced so that data can be collected about them, too. At least one individual in the organization should be designated to take ownership of these tasks. In the remainder of this discussion we refer to that individual as the capacity manager. The capacity manager should be given the resources and time to undertake those tasks, and access to those who will need to act on performance findings as usage of the system evolves. This individual must have access not only to the people responsible for building and maintaining the system, but also to those who are liable to cause demands to be made upon it. For example, if an advertising campaign is expected to trigger huge numbers of calls to call center agents and/or huge numbers of visits to particular web pages and the back-end systems they access, an effort should be made to estimate what that demand will be so that would-be customers do not go to competing web sites because they do not like long delays, and so that call center agents are not put in the position of saying to prospective customers that "the computer is slow." The capacity manager may have difficulty persuading stakeholders to take emerging performance issues seriously. Diplomatic skills and carefully presented data will always be necessary in this situation.

12.12 Summary

Examples of performance pitfalls in this chapter and throughout this book show that performance pitfalls can occur in many guises. They can be inherent in memory management techniques. They can arise because of organizational decisions or because of misconceptions about the work conservation properties of scheduling rules. They can occur because of inaccurate measurements or because of organizational anxiety that leads to the misinterpretation of why performance measures such as resource utilizations might have high values for a short amount of time. There is no hard-and-fast rule for avoiding these pitfalls. Only healthy vigilance and skepticism in the light of experience and clear analysis can be used to prevent, mitigate, or remedy them.

12.13 Exercises

12.1. Consider a replicated database system in which copies are stored on separate hosts. Strict consistency between the replicates is required. That means that commitment on one copy cannot take place unless it takes place on both copies.

(a) Should the update begin on the busier host or the less busy host? Explain.

(b) Should the process or thread that is beginning updates at the second host be given priority over threads that are doing the first update of some other record? (*Hint:* Think of the museum checkroom problem discussed in Chapter 11 (scalability). For a discussion of this problem, see [BondiJin1996].)

12.2. Explain why giving high priority to a workload that dominates the usage of any resource may not be helpful.

12.3. The average CPU utilization of a quad-processor system is only 25%, yet the utilization of one of the processors is 100% while the other three processors are idle. Absent further information, what do you suspect about the response time? What suspicions do you have about the system architecture? What will you try to find out about the executing processes?

12.4. For the system in Exercise 3.4 with device characteristics like those in the following table, what is the maximum system throughput that can be attained if the speed of the bottleneck device is doubled?

Device Name	Visit Ratio	Service Time (sec)	Service Discipline
CPU	6.0	0.0090	PS
Disk 0	1.0	0.0400	FCFS
Disk 1	4.0	0.0250	FCFS
Thinking terminals	1.0	4.0	IS

This page intentionally left blank

Chapter 13

Agile Processes and Performance Engineering

The use of an agile software development process provides opportunities for the early detection of performance issues that might emerge only after the completion of functional testing of the entire system under a waterfall process. The successful integration of performance testing into an agile process requires meticulous discipline and preparation to ensure that it can be carried out and the results analyzed within the tight time constraints of development sprints. Even if the overall development process of a system is not agile, agile methods can be used to quickly develop performance testing tools and test data so as to facilitate the timely delivery of a product. We examine both scenarios in this chapter.

13.1 Overview

Agile software development processes use sprints lasting a few weeks at most to provide frequent opportunities for iteration between the conception of the software's purpose, design, implementation, and testing. Each iteration occurs during the course of a sprint. The early testing that

sprints afford means that flaws in concepts, architecture, implementation, and function can be detected earlier than they would under a waterfall software development lifecycle. In the waterfall development process, there are distinct stages of software development that follow one another in sequence. Each stage can last many months. The stages include requirements specification, architecture, design, implementation, functional testing, performance testing, and delivery [Boehm1988]. By contrast, in an agile development process, pieces of each stage of the lifecycle occur in short sprints in which smaller pieces of the system are developed.

Performance issues and their origins might be detected and remedied earlier under an agile development process than they would be in a waterfall environment. There are a number of reasons for this:

- The early and frequent performance testing that could occur in an agile process would increase the possibility of performance issues being uncovered early, rather than after the development and functional testing of the entire system have been completed.
- The analysis of performance test results could be reflected in the refactoring and design modifications that might occur in a subsequent sprint.
- Similarly, if performance requirements change early in an agile development process, they can be reflected in changes to the implementation and to the design.

For this to be accomplished effectively within the tight time constraints of a sprint, however, considerable planning is required to ensure that the performance testing infrastructure is in place before the start of each sprint, including load drivers, scripts for automating performance testing, measurement instrumentation, and analysis and data reduction tools [Bondi2007b]. If performance testing cannot be accomplished in the same sprint in which a piece of the software was developed and tested, it must be done in the next sprint.

Agile methods can be used to develop a performance testing environment even if the overall software development process is not agile [BondiRos2009], just as they can be used to develop functional testing software. The performance testing environment is itself a suite of software programs that must be developed and tested before they are applied "in production," that is, to the testing of software components before they can be delivered. An agile process can effectively be used to set up a performance testing environment and execute performance

tests under the supervision of a performance expert even when the team has little or no performance expertise.

13.2 Performance Engineering under an Agile Development Process

The goals of performance engineering in an agile environment are the same as those in a waterfall or other development environment: to ensure that the system can meet performance requirements once those have been specified; to ensure that the software architecture is conducive to meeting those requirements; and to ensure that there is a methodical, repeatable, and traceable performance testing process to verify that performance requirements have been met and to enable the recognition of performance issues. As with functional testing, one key difference between performance testing in a waterfall process and in an agile process is that performance testing is done repeatedly in short sprints in the latter, but often only toward the end of the development process in the former. This means that the performance testing infrastructure must be planned and prepared well in advance of the sprints, perhaps even in the inception phase of an agile process. Another key difference between a waterfall process and an agile process is that stories, and hence performance requirements, may change from one sprint to the next. Since the failure to meet performance requirements usually has its roots in the poor choice of an architecture and/or in poorly specified performance requirements, it would seem that there are better chances of a system meeting performance requirements if there is at least a broad idea of the architecture and performance requirements before the start of the first development sprint. On the other hand, performance issues that are caused by inefficient code implementations or poor design choices can be caught earlier in an agile process than in a waterfall process, because opportunities for performance testing occur earlier in an agile process. Moreover, because inefficiencies and concurrent programming bugs can be introduced when code is refactored, functional and performance testing should always be repeated with the modified code in place.

Whether the system is being developed under an agile process or some other process, the performance engineer will always be faced with the challenge of not being able to execute a performance test until functional tests have executed and passed. It is not worthwhile to execute a performance test on a system that does not do what it is

supposed to be doing in single-user mode. Moreover, as we saw in Chapter 9, performance tests can reveal functional problems with concurrent programming that cannot emerge in unit testing. If functional testing is delayed because development is delayed, performance testing will be squeezed into an amount of time that is insufficient for the careful completion of the task. This problem is even more severe in an agile environment, because the short duration of sprints (two to four weeks) means that a slight delay can still consume a large fraction of the time to the sprint's planned end. It has been the author's experience that a sprint can consist of a series of waterfalls with very short travel times between them, and that the allowable travel times become shorter the further along one is in the sprint. For this and other reasons, an agile process is not a panacea for a weak or poorly trained set of teams. Indeed, the compressed nature of sprints can exacerbate the effects of weaknesses in the development organization. If anything, stronger levels of discipline, maturity, and experience are required under an agile development process than under a conventional one such as the V model or the waterfall model.

Care should be taken to ensure that samples of the data to be used in performance tests be available before the sprint commences. Load drivers, measurement instrumentation, and data analysis tools should be built to enable repeated and repeatable executions with little effort during each sprint. The development of test data should be planned to coincide with or precede the delivery of the portion of software that is going to use it. This is a concern of functional testers also. Indeed, the commonality between functional tests and performance tests is so great that it may be worthwhile for these groups of testers to work together to reinforce and avoid duplication of each other's work. Each team must have a clear understanding of the other's procedures and needs, especially in an agile environment, because time pressures there are particularly intense.

13.2.1 Performance Requirements Engineering Considerations in an Agile Environment

When performance requirements evolve from one sprint to the next, care must be taken to ensure that the essential criteria of sound performance requirements described in Chapters 5 through 7 are maintained, including traceability, correctness, and consistency. In particular, each new performance requirement must be consistent with the ones specified in earlier sprints, as must all changed performance requirements.

When performance requirements are added in later sprints, care must be taken to ensure that they can be met by the parts of the system that have been architected and implemented to date. This is especially true of new functionalities that might be implemented during a sprint. These must coexist with those developed previously. Since new functions may consume new resources, one must verify that the new performance requirements concerning them and the performance requirements relating to previously developed functions are still achievable. This might be predicted by performance modeling. In any case, it must be verified by performance testing. This means that the performance tests done in prior sprints must be repeated in subsequent sprints in which new functionality, workloads, and performance requirements are added, or when code has been refactored. This underscores the need for a heavily automated performance testing environment that enables the repeated and repeatable conduct of performance tests [BondiRos2009]. Repeatability is a precondition for the validity of comparisons of both system performance and function before and after any modifications have been made.

13.2.2 Preparation and Alignment of Performance Testing with Sprints

The level of preparation required for performance testing in an agile process depends on the nature of the system. All the guidelines and practices we have described in previous chapters are applicable here. If anything, performance tests must be planned and executed with even more stringent care than in a waterfall environment because of the tight time constraints imposed by sprints. The size and makeup of any database must be commensurate with that which is anticipated for use in production on similar target hardware. Since functional testing precedes performance testing, the database will be populated by functional testers in any event. The performance engineer and the leader of the performance testing team will be negotiating a schedule with a wide variety of stakeholders.

The schedule for the preparation of a performance testing environment will begin with an inventory of the hardware and software that will be needed to populate the system with application data and system configuration data. Scheduling must be done at two levels: a long-term level to ensure readiness for each sprint, and within sprints. We consider the long-term aspects first:

- Resource measurement instrumentation can be chosen and configured once the host platforms and network elements have been chosen.
- Scripts for reduction, display, and analysis of the resource measurements can be written and tested on like platforms once the operating system, network elements, hardware platforms, and off-the-shelf software such as application servers and databases have been identified.
- The writing of load generation tools with embedded hooks for measuring response times can be done only once the set of test transactions and their expected responses have been identified. In some cases, it may be feasible to insert the measurement hooks in the load-driving scripts for functional testing. Time must be planned to discuss this with the functional testing group.
- Test scripts should be developed for pieces of the application, service components, and platform elements in order of delivery so that functional and performance testing can begin as soon as possible thereafter.

Let us now turn to scheduling within a sprint. Because performance testing of the use cases or functions that are developed within a sprint can occur only once functional testing is complete, it is quite possible that some or even all of the performance testing and analysis will spill over into the next sprint. Within each sprint, the test cases that pose the highest risk to the rest of the system or application should be tested for performance first, so that problems can be corrected as early as possible. Very severe problems, such as overloads at levels well below the target levels, may be the ones to emerge first. Pilot testing might be sufficient to identify an issue that can then be followed up by performance engineers, architects, and developers alike.

13.2.3 Agile Interpretation and Application of Performance Test Results

Data reduction and analysis of performance test results must occur much more quickly in an agile development process than in a more conventional one because of the time constraints imposed by sprints. If a performance problem is encountered, it may be necessary to troubleshoot it by developing performance tests at different load levels and/or with different configuration parameters. This might be deferred to

the next sprint so that all test cases in the current sprint are executed at least once. In the case of large projects, this is something that may have to be discussed in "Scrum of Scrum meetings" at which scrum masters or other representatives of individual scrum teams meet to discuss blocking issues that can be resolved only in cooperation with other teams.

13.2.4 Communicating Performance Test Results in an Agile Environment

Because little time is available for the retrospective sessions that occur at the end of a sprint, the performance engineer should be prepared to condense the test results into a few slides describing the following:

- What the performance test needed to show, and the results that were desired (quantitative and qualitative)
- Bullet items showing which performance requirements were met and, more importantly, which were not
- Performance problems identified, and their potential impact, including risks to the project as a whole
- Next steps, either to address any problems identified or to accommodate the goals of the next sprint

The author's personal experience has been that it may be necessary to step outside the agile/sprint framework to resolve performance issues, especially complex ones that involve major changes to software or, in the case of a service-oriented architecture, the replacement of the entire implementation of a service or use case. The emergence of a severe performance problem that potentially undermines the rest of the system should be addressed rapidly, firmly, and vigorously. It must also be explained and documented clearly. Its resolution may entail explaining the problem and its suspected causes to different sets of stakeholders in terms that each can understand. The performance engineer may have to take the lead in proposing a solution to the problem. Sending e-mail to several stakeholders at once may not be sufficient to obtain timely resolution of a problem, because there will be no sense of accountability. The performance engineer might be well advised to gather insights from several stakeholders and distill them for explanation to the chief architect and product owner, as they will have final responsibility for the choice of a remedy.

13.3 Agile Methods in the Implementation and Execution of Performance Tests

There is a place for agile methods in the development of performance testing suites even if the system under test is being developed under a waterfall or other non-agile process [BondiRos2009]. We can do this because a performance test suite is a software system containing several components that depend only on the operating systems, hardware, and commercial software platforms involved as well as components whose implementation depends on the application. We have already seen that measurement instrumentation for the operating systems, hardware platforms, and off-the-shelf software such as web servers, application servers, databases, and network elements is often readily available and easily customized to measure the system under test. By contrast, load-generating software usually has to be programmed to generate transactions and requests that are meaningful to the application.

The practices one must follow when implementing performance tests in an agile manner are not different from those we have discussed in Chapters 8 and 9. Performance tests must be conducted in a clean environment on a platform that reflects the scale of the tested load and the number of objects encompassed by the system.

13.3.1 Identification and Planning of Performance Tests and Instrumentation

Performance tests should be structured to inform us whether performance requirements are being met and to reveal potential problems that would not have occurred in unit testing. The performance test plan should reflect this. Preparation for performance testing should begin early enough to allow performance testing of key components as soon as they emerge from functional testing. A schedule of sprints should be devised to

1. Formulate the test plan.
2. Identify the instrumentation that is independent of the application and arrange for its timely procurement.
3. Identify and develop the instrumentation and load drivers that depend on the application.
4. Put the instrumentation in place and test it.

5. Document all of the preceding steps.
6. Create tools to verify the correct functioning of the system with multiple users and multiple concurrent activities. This includes ensuring that databases were correctly updated and that no unarbitrated race conditions emerged.
7. Execute one or more rounds of performance tests and document the results.

These steps may occur in separate sprints or in parallel in the same sprint, depending on the skill sets of the participating staff. For example, the identification of resource usage measurement instrumentation can be done in parallel with the creation of tools to verify that the system is functioning correctly with multiple users.

Incremental testing of performance instrumentation and load drivers as they are developed has the following benefits:

- It helps to ensure that correct data is offered to the application at the right rates.
- It allows verification that system resource usage measurements and statistics about response times are correctly collected.
- It provides an opportunity to verify that instrumentation and load generators are correctly calibrated.

13.3.2 Using Scrum When Implementing Performance Tests and Purpose-Built Instrumentation

Ongoing communication among the members of the performance testing team is very useful for resolving blocking issues and keeping the development of the performance test suite on track and focused on its goals. If the team has never done performance testing before, it should be coached and supervised by a performance engineering and testing expert. The author's experience has been that holding scrums shortly before lunch and shortly before the close of the working day facilitates quick tool development and quick correction of mistakes. Knowing that one will have to describe one's progress also places subtle pressure on the testing team to achieve a deliverable at regular intervals. At the same time, shared knowledge of difficulties encountered provides opportunities for self-organization and the formation of small groups to pool expertise and solve problems as they arise. The physical presence of the performance expert among the performance testers provides opportunities for quick discussions about points needing

clarification. If the performance testing team will be the first to integrate the various components of the system as a whole and drive work through it, communication with architects, developers, and requirements engineers will help the team understand what behavior is to be expected while expediting the clarification of any ambiguous requirements. Pilot performance tests should be planned with the Scrum timetable in mind so that there will be a scheduled time when first observations can be shared and the causes of the results identified.

13.3.3 Peculiar or Irregular Performance Test Results and Incorrect Functionality May Go Together

As we discussed in Chapter 9, irregular performance test results and incorrect functionality may go together. Erratic resource utilizations or response times under constant transaction load are indicative of concurrent programming issues like deadlocks and the incorrect implementations of thread safety and mutual exclusion. Incorporating performance testing into an agile sprint provides an opportunity to correct these problems with the developers and other stakeholders before adding more code to the system makes them harder to find. Sometimes unstructured playtime enables the testers to reveal unexpected problems that warrant attention. This was the author's experience of playtime with a testing team that was untrained in performance testing, as described in [BondiRos2009]. The team was the first to put the pieces of the system together and subject it to load. Playtime is discussed in the next section.

13.4 The Value of Playtime in an Agile Performance Testing Process

As with any scientific endeavor, unstructured playtime provides an opportunity to acquire familiarity with system behavior and the responses of instrumentation to changes in system configuration and offered loads. Bondi and Ros describe the beneficial role of playtime in finding a potentially disruptive problem in a complex system during the course of an agile performance testing process [BondiRos2009]. Each release of the system under test was developed according to a V model process [VXT2009], with a progression from concept to requirements to architecture, followed by a progression to design, implementation,

functional testing, and, ultimately, end-to-end performance testing. The system under test was a complex service-oriented XP-based platform that provided services for various applications, implemented in a mix of C and C++.

The performance testing team acquired a better feel for what was happening in the system by playing with it in a spontaneous manner for an hour or so. They could see the performance impact of sending in requests at different rates and try out different *perfmon* counters to see which ones might yield insights and which ones not. The performance engineer found that playtime with the system improved the team's morale and team members' willingness to take the initiative to try out different kinds of tests.

The team was delighted when applying load to the system resulted in a visible increase in CPU utilization, and when turning the load off resulted in a drop. Unfortunately, starting and finishing average CPU utilizations were not zero but 50%, even when the system was supposedly empty and idle. This provided a teachable moment. Under the guidance of the performance engineer, the team learned the value of collecting the utilizations of individual processors when one was shown to be 100% and the other 0% in the absence of applied load. Further investigation showed that one process was the culprit. One of the test engineers recognized that the offending process was repeatedly polling an empty message interface, and that neither the requirements nor the architecture had allowed for any other kind of interprocess communication, including a mechanism that would have allowed the offending process to sleep until the arrival of a message, such as a semaphore.

At this point, the team lost interest in documenting its work and wanted to try out all sorts of things. This sort of playtime was conducive to their getting a feel for how performance measures and instrumentation work. After struggling to record what the team was doing and the results, the performance engineer called a halt and gathered the team members in a conference room. We recorded observations, possible causes, and possible avenues for investigation in separate columns on a whiteboard, transcribed them into a spreadsheet, and developed an action plan for the following business day. The plan included sending e-mails to the system development team and systematically conducting short tests with different parameters under tightly controlled conditions, with careful documentation. The upshot was the identification of some serious bugs in both the software and the underlying requirements. Change requests were written against them on the

following business day. The team could not proceed with the execution of test cases until they had identified a work-around to the most obstructive problem and implemented it.

This experience gave a valuable lesson that obstacles encountered in the execution of a test plan can provide useful insights into the way the system behaves. It also showed the team that performance testing is extremely valuable for provoking concurrency and scheduling problems that could not have shown up in unit testing.

13.5 Summary

The foregoing narrative shows that it is possible to use agile methods to conduct performance engineering activities such as performance testing even when the system under test is not developed under an agile process. Interestingly, the clear separation of the stages of the software lifecycle enables the development and execution of performance tests to be conducted under whatever process works best for the performance testing team. Our experience has been that agile development can be very effective in these circumstances when testing team members are under the guidance of an experienced performance engineer with agile experience. When performance engineering is part of the development sprints, considerable advance preparation of testing tools and data analysis tools is needed to enable timely performance testing under constraints imposed by the short durations of sprints and the likelihood that completion of the functional testing that must precede performance testing will be delayed.

13.6 Exercises

13.1. You are the lead performance engineer in a team that is currently planning the sprints of a large software development effort. You must negotiate time for the preparation of measurement instrumentation and load generators to verify that the performance of the system is sound.

(a) At this stage, the performance requirements of the system are likely to be unstable, assuming that anyone has specified them, which may be unlikely. The functional requirements may not be fully specified either. What instrumentation can you procure or prepare now or in an early sprint to

ensure timely measurement and performance evaluation of the features that are developed in each sprint?

(b) Explain how you might work with functional testers to develop load-driving scripts and scripts to verify that the system has functioned correctly after each performance test. Explain why such verification is necessary, even if the system passes all functional tests before performance testing commences. Explain how early discussions of these points might influence your choice of a commercial software testing package to execute both performance and functional tests.

13.2. The results of your performance tests indicate the presence of some serious problems. Identify the stakeholders who must be informed of this.

(a) Explain how you would formulate an ad hoc test plan to help isolate the cause of the problem. Can you reuse any of the tools that you used to run the original performance tests and analyze the results?

(b) Develop a short presentation to explain the performance problems you have seen. It must be understood by multiple stakeholders, such as architects, developers, requirements engineers, functional test engineers, and the product owner. Contrast what you saw with what you expected to see, and explain what this might mean for the success of the product and what the cause might be. Bear in mind that your presentation has two purposes: obtaining the lab time and staff time needed to run the ad hoc performance tests and triggering insights on the solution of the problem from the other stakeholders. Be sure to indicate that you are prepared to revise your ad hoc test plan to take those insights into account.

This page intentionally left blank

Chapter 14

Working with Stakeholders to Learn, Influence, and Tell the Performance Engineering Story

The effectiveness of performance engineering depends very heavily on one's ability to learn about the system and stakeholders' concerns and to relate one's analysis and recommendations to them in terms they can understand. The first step is to understand what aspects of performance matter to which stakeholders so that their concerns can be documented and articulated. At the same time, the performance engineer must assure all stakeholders that one of the main goals of a performance engineering effort is to ensure that performance concerns and customer expectations are addressed, while providing them with the means and tools to meet them. At every stage, the performance engineer needs to show that the performance effort is adding value to the product, directly or by mitigating business and engineering risks. In this chapter, we describe which aspects of performance may matter to whom and then explore where the

performance story begins. We then go on to describe how concerns might be explained to those with different roles in the software organization.

14.1 Determining What Aspect of Performance Matters to Whom

In previous chapters we have focused primarily on performance engineering methods and practice at various stages of the software lifecycle. Here, we turn our attention to understanding the performance story, developing it, shaping it, and explaining it to others, preferably in terms that they can understand and that address their concerns. Those concerns may depend on the roles of the individuals in their organizations and on the nature of the domain.

Within the last three or four decades, the use of computers has evolved from running intensive calculations in batch mode and interactively handling business transactions to controlling and monitoring systems in every industry, as well as delivering entertainment and services to the home and to handheld devices everywhere. Every domain has its own set of concerns, related performance requirements (whether acknowledged and specified or not), and, in many cases, its own set of regulations and standards to which the equipment and the computers controlling related systems and providing services must conform. The basic metrics one uses to describe performance have not changed much during this time, although the technology, events, and actions to which they relate have. Fundamentally, one is usually concerned with delays, throughputs, and data storage capacity, regardless of the problem domain.

Measurement instrumentation usually lags behind the introduction of new software platforms and sometimes behind new technologies as well, but the underlying methods one uses to dissect observed performance issues have not: they are based on the scientific method and on the design of experiments. Our understanding of how performance issues arise has evolved as the variety of technologies and applications involving the use of computer systems has grown. Yet, similar problems tend to arise with the introduction of each new application, including constraints on processing power, wired network bandwidth, memory, radio bandwidth, and secondary storage size and latency. The basic principles of performance engineering are applicable to them all.

Whether response time and throughput requirements are directly related to safety, as is the case with systems that control power stations, chemical plants, cars, aircraft, and trains, or whether these requirements are largely a question of convenience and competitive differentiation, as is the case with entertainment and gaming systems, performance engineering is needed to ensure that they are met. The performance engineer should proactively engage in the task of identifying the drivers of performance requirements, and then elicit and/or formulate those requirements in concert with product managers and domain experts. The criteria for sound performance requirements are discussed in Chapters 5 through 7. In every case, the requirements should be linked to business, engineering, and regulatory needs.

Some stakeholders will be reluctant to express their performance needs in quantitative terms, whether these terms are measurable or not. This could be due to lack of numerical facility or to a hesitation to make a commitment to a given level of traffic, among other reasons. By contrast, some stakeholders will be engaged in the numerical details of their own aspect of the problem to the extent that the global picture of the performance situation is lost. The performance engineer must be able to both absorb the performance story and share it with stakeholders at both of these extremes and at all levels in between. Regardless of the stakeholders' ability to reason in a quantitative manner, the performance engineer may need to act as a data shepherd to ensure that projections about load, performance requirements, and the capabilities of the proposed hardware and software platforms on which the software will be run are all plausible. Tips on communicating about data may be found in [HuffGeis1954]. Some of these tips relate to the careful choice of scales on axes. The use of graphics to illuminate data is described vividly in [Tufte2006].

14.2 Where Does the Performance Story Begin?

The story the performance engineer must learn and eventually tell depends upon the position of the system of interest in the software lifecycle and how it is perceived by stakeholders. If it is in production, there may already be complaints about its being too slow or apprehension about how it will perform when load and new functionality are added. The architecture phase presents the best opportunity for incorporating system properties that support high throughput and short

response times and enable scalability. In performance testing, expectations about performance will be refined and problems and opportunities for improvement identified provided that the test plan is soundly structured and provided that the measurements are stored and processed in a manner that facilitates analysis.

In many instances, the performance engineer's first encounter with a system will be with stakeholders who are apprehensive about system performance, whose expectations about performance and capacity may be only vaguely described, and whose approaches to resolving performance issues may have gaps or not be well formulated. The performance engineer's first task is to draw out the stakeholders' expectations and observations, and then shape a performance engineering plan based on business and engineering goals and the stakeholders' ability to put resources in place to help carry them out. These resources may include personnel (who may have to be trained), measurement instrumentation, access to a performance testing lab, and lab and staff time to carry out performance tests. They also include the time and efforts of architects, testers, and many others involved in the various phases of the software lifecycle.

One does not want the performance story to begin on the evening news or the front page of a newspaper. Sometimes this is unavoidable, even though it might be foreseeable. A stock market crash, a sale of tickets for a rock concert, unusually cheap air fares, a natural disaster, a terrorist attack on a major city, or, more recently, the introduction of health insurance exchanges to implement the US Affordable Care Act can trigger enough demand on the corresponding web sites to cause them to become excruciatingly slow or cease functioning altogether [Carr2013]. The earlier chapters in this book provide guidance on how one might diagnose the causes of the exchanges' performance issues. Conclusions about the causes of performance problems in the health insurance exchanges or any other system based on news reports alone may be wrong. It could well be that performance issues are due to system malfunctions or misconfiguration rather than to a lack of system resources. System measurements and functional testing under load must usually be used to determine the causes of performance issues.

When interviewing stakeholders before or during a performance engineering effort, the performance engineer should be carefully attentive to what matters to whom and to the potential costs of possible remedies to performance issues. The stakeholders' interests, concerns, and viewpoints are as diverse as their functions in the organization, their training, and their educational background. The performance

engineer may be interacting with product managers, project managers, marketing executives, architects, developers, testers, and many others. The most pressing questions to be borne in mind usually are:

- What aspects of performance engineering matter to whom? How can their interests and concerns be addressed?
- What are the various stakeholders apprehensive about? Is their apprehension justified? Is it based on the right metrics? Is it based on folklore or on experimental evidence?
- When is the system to be delivered? When are enhancements expected to be installed, including new features?
- What costs and constraints would limit the choices of software development efforts or hardware upgrades to address a performance issue? Are there technological constraints, such as the need to interface with existing systems or the need to avoid licensing costs, that would affect the choices made to remedy a performance problem or to ensure that performance requirements are met?
- Within what time are solutions to performance issues needed? When is a major increase in load expected?

Then there are questions related to engineering, regulation, and planned growth:

- What form does the load take? What are the use cases?
- What performance is required of the system? Has this been documented?
- What transaction response times are expected and why?
- What performance measurements are being taken?
- Is the system perceived to be overloaded?
- Are surges of load expected? Will there be large amounts of seasonal variation?
- What performance tests have been run? Were the tests well structured? How do the results look?
- What are the safety concerns related to the system?
- Are there stringent reliability requirements or limitations on downtime? If so, what recovery mechanisms are in place and what are the desired failover and recovery times? Have these been thought through?
- What regulatory needs have to be met by the system?

- What is the planned growth trajectory for the offered load or the number of application-related objects in the system?
- Does the organization have processes in place for performance management, performance engineering, and, in the case of purchasers of the system, capacity management?
- What is the competition up to? What is their edge? What is the performance engineer's client's edge or desired edge?

The performance engineer will develop a picture of a performance story. He or she should then be able to develop a plan of action with stakeholders as the answers to these questions fall into place. The performance engineer must also be prepared to help justify a plan of action on cost grounds as well as engineering grounds.

14.3 Identification of Performance Concerns, Drivers, and Stakeholders

The performance concerns of a system are always tied to system functions and/or business functions. Therefore, the first step in identifying performance concerns is understanding what the system does, how quickly it needs to do it, how it does it if it already exists or is partially developed (i.e., the information flow), and how often it needs to do it.

A system may have foreground and background activities. The foreground activities usually involve taking actions in response to stimuli and, depending on the nature of the application, performing computations. The results of the computations may be required to determine the responses to the stimuli. The computations may be intense but need not be. The actions could be as varied as setting up a telephone call or web connection; performing database transactions; or initiating such time-critical actions as sounding alarms and closing doors in case of a fire, closing grade crossing barriers on a railway, or stopping or slowing a train as it approaches a signal. The computations might include processing or generating images or analyzing large quantities of scientific data for the purpose of information or prediction. Background activities might include archiving, monitoring of the system itself so that processes that halt can be restarted, transaction logging, detecting and generating notifications of intrusions, purging database records marked for deletion, or triggering overload control mechanisms. These lists are not exhaustive.

Because the stakeholders will be the richest source of information about how a system works and how its applications can be measured, it is essential to identify and begin working with them as early as possible in the performance effort. Some of them will be responsible for obtaining the information about architecture and design needed to understand performance issues and how to resolve them, while others will execute measurement plans and performance tests. Many will have a direct line to managers who will decide whether performance engineering is adding value to their product and solving their problems. The performance engineers will be able to reach their goals more effectively and satisfy business goals if they develop good working relationships with those managers and with other stakeholders.

14.4 Influencing the Performance Story

The system stakeholders may turn into the performance engineer's best advocate for implementing his or her recommendations for performance improvement, and they may well come up with excellent recommendations of their own which the performance engineer can then check and even improve. It is much easier to persuade an organization to implement a recommendation if its members had a part in conceiving it. In some cases, the recommendations may be patentable. An inside stakeholder whose work is rightly commended by a performance engineer may well turn into an inside champion for the performance engineer's work and recommendations. This can be very productive for all involved. Moreover, it reinforces the notion that a performance engineer is there to help the engaging organization get where it needs to go and to help empower those who will take it there with his or her expertise.

We now turn to a discussion of the impact of performance engineering at various stages in the software lifecycle and the roles of various stakeholders in performance engineering.

14.4.1 Using Performance Engineering Concerns to Affect the Architecture and Choice of Technology

In Chapter 11 we saw how architectural and design choices could either promote system scalability or undermine it. In particular, we saw that a program that serially processes work on disjoint sets of data objects would not be able to exploit the potential for parallel execution offered

by multiprocessor and multicore systems. The problem is even more complicated with distributed systems because of the communications cost of moving data from one host to another and problems with coordinating activities in different places. The reasons for separating computations may have as much to do with business, legal, or political constraints as with engineering considerations. For instance, the new health insurance exchanges in the United States must query government databases to verify the identities, incomes, employment status, and eligibility of would-be policyholders while keeping sensitive data private. The details of this data are independent of the geographical information and choices of options that insurance companies use to determine the price of insurance coverage and so are held separately.

Let us once again consider a museum checkroom as a metaphor for a computer system with separable concerns. All checkrooms at the Metropolitan Museum of Art in New York accept hats, coats, gloves, scarves, shopping bags, small backpacks, umbrellas, and small parcels. At the Louvre in Paris, coats and umbrellas are accepted at one checkroom, but backpacks and hats must be checked at another. Absent an inquiry of those who made the policy at the Louvre, one can only speculate about the concerns that led to its adoption. The reasons might be historical, or they might have to do with coatroom attendants not wanting to fumble with small objects or deal with a lack of floor space. Similarly, the deployment of distinct functions on different host systems may simplify the implementation by enforcing modularity, but doing so will also increase communication costs. Understanding the drivers that lead to architectural choices is crucial to making recommendations to improve performance, especially when the recommendations include choices that are different from those that have already been made.

As we have seen in earlier chapters, the choice of technologies to implement a system is often influenced by performance considerations. The use of exploratory testing is a useful way of reducing the risk of making a poor choice of technology or of encouraging platform vendors to improve the performance of functions that will be frequently invoked [AvWey1995, MBH2005].

14.4.2 Understanding the Impact of Existing Architectures and Prior Decisions on System Performance

When a system in production or in performance testing is demonstrably failing to meet performance expectations, the performance engineer's first steps should be to review the architecture and information flow through the system and review the measurements that have been taken.

A review of the architecture will enable the identification of additional places where performance measurements should be taken, such as in software platforms like web servers, application servers, and the interfaces to other systems that must be queried so that actions can be completed. The purpose of inserting these extra measurement points is to isolate bottlenecks and, in the case of remote systems, determine whether any component of delay is outside the system owner's control. A review of the measurements and measurement procedures might reveal that the performance concerns are justified, or that they are based on a misunderstanding of the meaning of the data. In either case, the performance engineer must provide a clear and careful interpretation of the measurements so that a correct decision can be made about what needs to be done next, if anything.

At the same time, a review of the information flow should reveal the locations of foci of overload and antipatterns such as god classes and circuitous treasure hunts. God classes are centralized points of activity that result in foci of overload, while circuitous treasure hunts are long chains of activity [SmithWilliams2000].

One may encounter architectural constraints that make it infeasible to implement a performance-enhancing recommendation. Once again using the museum checkroom as a metaphor, there may not be support for implementing a priority scheduling scheme with multiple queues to prevent deadlock between those visitors collecting and those leaving coats. The architecture of the main hall at the Metropolitan Museum does not support this, because only one doorway can be used to enter the checkroom and only one to leave it. Similarly, the process table entries of a simple operating system may not have fields indicating process type or priority level. This makes priority scheduling and distinguishing between processes freeing and acquiring objects from a pool impossible, and deadlock cannot be prevented. This is an architectural or implementation deficiency that prevents a desired performance improvement. The remedy would be to equip the operating system with a priority scheduling mechanism and a method of tagging processes or threads with markers indicating their needs for different levels of priority at different times, according to a well-defined set of rules.

14.4.3 Explaining Performance Concerns and Sharing and Developing the Performance Story with Different Stakeholders

Performance engineering involves partnerships with diverse stakeholders. It is not a soloist's pursuit. We now summarize the issues that are

most likely to be encountered when working with different sets of stakeholders, and how the performance story could be influenced by each one. If the system is in trouble, the performance engineer needs to indicate strongly to all concerned that his or her role is to help enable the team to understand and solve performance problems, and not to apportion blame. Those who feel threatened might not share information gladly.

Apart from organizational considerations, it is important to ensure that there is a common understanding of the performance metrics and the impact of their values on the functioning, usability, and effectiveness of the system. As we discussed in Section 14.2, some metrics may be of greater interest to one set of stakeholders than to another. It is important to ensure that the descriptions of their metrics and their values are mutually consistent so as to reduce the risk of misunderstandings. These misunderstandings could lead to the delivery of a system that fails to meet performance expectations as well as to conflicts within the organization about the meanings of metrics and about customers' expectations about performance. Measures should be taken to avoid them.

14.4.3.1 Architects

The architect is usually, but not always, the sole player in the software project who interacts with all stakeholders and who has a global view of the pieces of the system and how they relate to one another [Paulish2002]. In the author's experience as a performance engineer, and in the experience of a number of fellow performance analysts with whom he has worked, a performance engineer complements this role by examining information flows throughout the system and by identifying bottlenecks as well as performance requirements and other quality attributes.

Usually, it is the architect who has the mandate from the project's customer to see to it that the implementation is performant and scalable, and to require that specific solutions be adopted to achieve that end. The architect therefore has the responsibility for ensuring that information about the performance characteristics of solutions and technologies are well understood and, where they are not understood, that an understanding be acquired through early testing to minimize business risk [MBH2005] or through consultations with trusted colleagues with related experience. This is essential to containing the business and engineering risks inherent in the project.

At the same time, the architect must ensure that solutions are cost-effective, and that they comply with regulatory requirements and legacy constraints. An example of a regulatory requirement would be

the need to ensure that a fire alarm is activated within 10 seconds of smoke being detected [NFPA2007]. An example of a legacy constraint is a requirement to use the same database management system as in an existing implementation to reduce porting and training costs. It is incumbent upon the software architect to ensure that constraints are understood, that performance requirements are gathered, that performance artifacts are specified for each system component, and that performance and scalability are part of the mindset of all stakeholders.

To meet the responsibility for the performance of the system, the architect must take a lead role in ensuring that the performance requirements are carefully elicited in terms that are measurable and testable. This entails ensuring that product management, design, development, and testing managers and their teams all understand the performance requirements and how they will be tested. The architect should give a clear and visible mandate to the performance engineer to facilitate the cooperation of other stakeholders in meeting performance goals.

If accurate performance requirements cannot be elicited, the performance engineer should identify acceptable ranges of performance measures and workloads, based on an investigation of the domain and of business needs. Then, the architect should ensure that consensus for those operating ranges has been reached among domain experts and product managers, among others.

14.4.3.2 Management

While managers may understand that inadequate performance poses a risk to their systems, they may need to be convinced of the value of efforts to mitigate that risk and of the benefits of remedying performance deficiencies or proactively avoiding them altogether. They will also need to be reassured that proposed remedies will be effective or, in some cases, that all is well with the system and that only minor changes are needed to ensure that performance needs are met. Where third-party subsystems, platforms, and hardware are involved, the performance engineer should take the initiative to provide recommendations about the performance requirements of those subsystems and explain how they support the overall performance needs of the system. The performance engineer may also have to understand and explain the limitations of those third-party elements, so as to avoid specifying requirements that are infeasible with the technology available. Moreover, the performance engineer may be asked to assist management in negotiations with third-party suppliers to ensure that performance needs are met.

It is essential for the performance engineering team to have a solid relationship with management, because management will frequently have to give the performance engineer a visible mandate to spend time with other stakeholders to ensure that the performance needs of the systems are met. One reason for this is that stakeholders may feel too preoccupied with features and deliverables to be concerned about performance. Another reason is that stakeholders may fear that a performance engineer's role is to apportion blame for why a system is failing to meet performance expectations. The managers and performance engineering team can help to create an atmosphere in which all are trying to solve a problem and all are part of the solution.

14.4.3.3 Requirements Engineers

Every functional requirement has a performance impact or a performance need, whether or not these have been specified. Similarly, every performance requirement will have an impact on the functioning of a system, on the system's effectiveness and usability from a customer standpoint, and, in the case of commercial systems, on the system owner's competitiveness. Each organization may have its own preferences about whether every functional requirement should have its own performance requirement, or about whether performance requirements should be presented separately. Either way, it may be fruitful for the performance engineering team and the requirements engineering team to coordinate their efforts by storing both sets of requirements in a common requirements management system and by ensuring that performance requirements are managed using the same processes and functional requirements. These processes include requirements elicitation, review, and change control. Using common terminology to describe, evaluate, and write functional and performance requirements will facilitate this. That is the reason that the chapters on performance requirements in this book use the terminology and evaluation criteria that are listed in [IEEE830].

14.4.3.4 Designers and Developers

Designers and developers often intimately know the inner workings of the pieces of the system for which they are responsible. They will be able to tell the performance engineer about algorithms that are inherently inefficient, the use of single threading in a way that impedes the exploitation of parallel processors, and implementations that conform to known antipatterns. They may also describe scheduling rules that either they or the performance engineer may recognize as leading to

deadlock, other concurrency issues, or increased response times. Discussions with developers may help the performance engineer identify suggestions for improvement and give some idea of the effort needed to implement them to other stakeholders.

Developers and designers, among others, may resist involvement with performance engineering efforts because of the fallacious myth that code optimization is evil. As pointed out by Hyde [Hyde2009], early software performance engineering is not the same thing as (allegedly) evil premature code optimization. This is part of a description imputed to Knuth but apparently due to Hoare. Indeed, Hyde points out that Hoare was actually advocating *early* attention to performance considerations in software design (italics added), and that the "evil" was devoting time to micro-optimization of code rather than to designing algorithms for performance. Hyde argues, moreover, that the quotation is greatly responsible for the attitude that performance problems can be solved after the system is built. The point to be shared with developers and others, then, is that careful attentiveness to the performance aspects of algorithms, architecture, coding, and design is essential to assuring the performance of the system. The performance of the system can seldom be radically improved once the system is built, because most performance problems have their roots in incorrect architecture and design.

14.4.3.5 Functional Testers and Performance Testers

Insights into program performance might be gained by sensitizing functional testers to its importance and encouraging them to pass their observations to the performance engineer, the architect, and the functional testing team's coach if there is one. If a functional tester is manually conducting a unit test, he or she will instinctively be aware of an excessive response time and will usually be the first to know about it. In a start-up or agile environment in which information can be exchanged quickly, the performance engineer or even the tester can quickly alert the development team that something is amiss and have the problem rectified. If unit testing is automated, it might be worthwhile to log the initiation and completion times of the test if the automated test manager has that capability. The testers should be encouraged to pass those results to the performance engineer so that performance problems can be quickly identified. The long execution time may be due to a configuration issue, or it may be due to a deficiency in the code. Either way, early intervention may save a lot of aggravation and cost later on. Even if the development process does not permit rapid intervention—for

example, if a rigorous waterfall model is in place—there is much to be gained by logging issues of this nature to facilitate the determination of the cause of the problem later. This is important because isolating the cause of a performance issue becomes more difficult (and hence more costly) as the system becomes more complex and as integration proceeds. Therefore, there is a tangible business value to having a working relationship between the performance engineering and functional testing teams. This relationship is particularly valuable in a compartmentalized organization in which information does not flow easily between teams. It is also valuable for enabling the performance testers to use the functional testers' scripts and test data to eventually run performance tests once the functional tests have been run, and for encouraging the functional testers to make the performance testers and other stakeholders aware of emerging performance issues.

The performance engineer should work with performance testers to identify the form that the test results should take to facilitate analysis and debugging. Commercial performance test drivers may already provide the performance measurements in a form that is susceptible to analysis, including files containing data from which plots can be generated automatically. One should verify that the test drivers are configured to generate measurement log files and plots in the desired format for subsequent analysis. The performance engineer should also ensure that the performance test plan covers the performance requirements, and that it is structured in a manner that allows the revelation of hidden problems such as software bottlenecks and even configuration errors. Configuration errors might manifest themselves in the form of significantly long response times and functional errors. For example, if the error is the incorrect entry of an IP address, a transaction might fail, but only after it has repeatedly been attempted because acknowledgments at the transport or application layers have not arrived.

14.4.3.6 User Experience Engineers

The sequence of steps in which a user must navigate through a user interface may significantly influence or be influenced by the way in which business logic is implemented. The way graphics are rendered at the user interface may have real or perceived performance impacts, and the responsiveness of the business logic under various loads will influence the user's perceptions about the system's effectiveness and usability. The user experience engineers and the performance engineers may wish to discuss what response times are attainable and whether the user interface can be designed to minimize its performance impact on the business logic of the system. They should discuss whether the

interface can be designed for performance testability as well as functional testability. In addition, the user experience engineer's ideas about desired response times might be taken into account when writing performance requirements. Finally, one should not forget the cost of dynamically managing users' performance expectations through the use of progress bars and other notifications. These have resource costs and performance issues of their own.

14.4.3.7 System Administrators and Database Administrators

If a system is already in production, system administrators will be able to tell you what is being measured at the operating system level, where the performance data is coming from, and with what level of granularity. They will be able to describe how the system has been configured and, for example, where load is taking place and what procedures are in place to move data from one disk drive to another as storage capacity limits are approached and when utilizations appear to be high. They may be the custodians of logs that could give insights into performance-related system crashes and surges of activity that resulted in increased delays. This is data that the performance engineer will share with management when decisions are made about implementing recommendations for performance improvements. The systems administrators may also have knowledge about performance complaints that might not have been logged in a problem tracking system.

14.5 Reporting on Performance Status to Different Stakeholders

Graphical depictions of data and trends always communicate a performance story more effectively than tables of data. Tables of data should be used for verification purposes and to generate statistical analyses if necessary. The graphs should be set up to tell stakeholders how the performance of the system evolves during a test run and about the average values of performance measurements taken over specific periods of time. Data and modeling predictions can be used to support recommendations in addition to describing the current performance status of the system and the status of the system before and after various changes were made. The data and the graphs should be presented in a manner that enables stakeholders to understand the strong and weak performance points of the system and to support recommendations for system improvement. The specifics of how the story is told and communicated

depend on the situation and the nature and duration of a performance engineer's engagement with the development organization.

A performance engineer should always convey a nonjudgmental attitude about the quality of the work of the team he or she is supporting. On no account should he or she describe a design decision or architectural decision as silly or sloppy, for example. The emphasis should always be on helping the team achieve the goal of delivering a robust, performant system that will meet or exceed perceived customer expectations, whether these expectations have been carefully described or not.

14.6 Examples

A performance engineer was once called in to evaluate the performance of a system that was close to delivery. There were concerns about where the bottlenecks were and whether the system would support the desired transaction volume. Measurements that had already been systematically collected showed high resource utilizations under a load close to the target load. Response times were higher than desired, yet processor utilization was below saturation, while I/O utilizations seemed high but not intolerable. The testing team had also collected measurements showing that the system spent very small amounts of time in large numbers of sections of the code, through both profiling and observing processor utilizations by threads. Thus, the system had poor locality of reference. The performance engineer and the testing team put together a quick performance test plan to use the instrumentation in the operating system to measure resource utilizations at loads of n, $2n$, $3n$, $4n$, and $5n$ transactions per second. Resource utilizations were linear in the load, indicating that there was no software bottleneck. The cache hit ratio was low. Combining these observations with the knowledge that the system had poor locality of reference, we came to the conclusion that the architecture of the system was sound from a performance standpoint, but that the memory bus and memory cycle times were degrading performance. The test results were presented with all stakeholders in the room. Each one was given the opportunity to relate the observations about the measurements with the parts of the system he or she knew best, and a course of action was then mapped out to resolve the problem.

A performance engineer on a prolonged engagement developed automated testing and analysis tools to provide performance assurance

of a system with a very large capital cost and with a large number of use cases. Tightly integrating the use of these tools into the testing and development processes ensured that performance issues were identified well before the delivery of each release, thus reducing the risk of user dissatisfaction while ensuring the timely remedy of performance issues as they arose. In this case, the performance test results and their analysis were immediately shared with test engineers, development managers, and architects to assure timely solutions of the problems encountered. Presentations containing condensed depictions of the performance data and what they said about the system were given to management from time to time, and the full detailed data was shared with those who were more closely involved with responding to it as needed.

14.7 The Role of a Capacity Management Engineer

Once a system is in production, its performance and resource utilizations should be monitored and tracked on an ongoing basis by a capacity planning engineer or by a performance engineer. The capacity planning engineer has a duty to maintain awareness of how the load and performance of the system are evolving over time and to motivate and possibly direct and coordinate system changes to respond to changing loads. The capacity planning engineer should be informed well in advance when extra traffic or functionality is to be added, for example, because of the introduction of a new service, product, or special offer. He or she can then plan the procurement and installation of new hardware and software resources (such as licenses) needed to cope with the extra demand. Cost-effective procurement requires the addition of resources commensurate with the new load. The lead time for the planned addition of new load should take the time to approve procurement requests and product delivery times into account.

The capacity engineer should be sure that the system owner is continually aware of computer resource usage, lest capacity be taken for granted and the capacity manager's role fade into the background. The best way of doing this depends on the culture of the organization. One way of maintaining visibility is the timely delivery of reports showing the hourly evolution of traffic demand and resource utilizations over the course of an entire week, as well as summary reports showing the demand during the peak hour and the corresponding resource utilizations.

14.8 Example: Explaining the Role of Measurement Intervals When Interpreting Measurements

As we discussed in Chapters 8 and 9, the choice of the length of a measurement interval plays an important role in how data is interpreted. Long measurement intervals tend to smooth out the effects of spikes in measured values such as the CPU utilization, while short ones may yield graphs that reveal and perhaps exaggerate the importance of very short spikes. We illustrate how this story can be told to testers and others who may be concerned about the high utilizations they have seen. Figure 14.1 shows a plot of CPU utilization versus time during a performance test run. The system takes about 11 minutes to ramp up, and then settles down to periodic behavior.

The different curves show the effect of averaging the observed utilizations in contiguous nonoverlapping groups of one, two, and ten intervals. The CPU utilization achieves its peak value for very short amounts of time when the intervals are looked at individually.

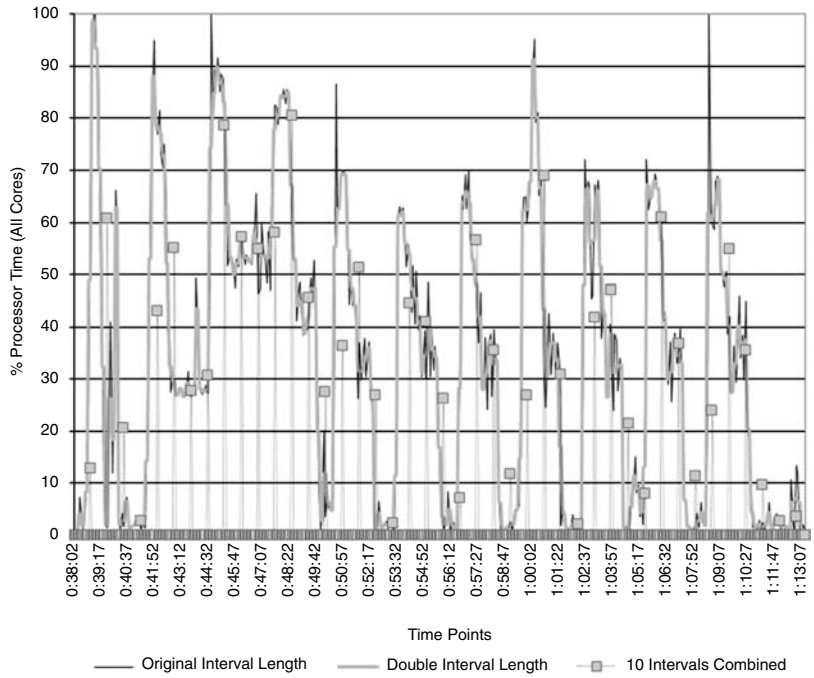


Figure 14.1 *Processor utilizations with original observations, averages of pairs of adjacent observations, and averages of groups of ten observations combined*

The peaks are lower and appear to last longer when the intervals are grouped. The peaks are lower with grouping because the utilizations were averaged over longer intervals, and they appear to be longer because the groups of measurement intervals cover longer time periods. Grouping the intervals masks the peaks. This can be a disadvantage if we are trying to detect the presence of oscillations as was discussed in Chapter 8. It can also be an advantage if one is trying to determine the overall average utilizations for capacity planning purposes and if spikes in delays are not a concern. Figures 14.2, 14.3, and 14.4 show histograms of the distributions of the numbers of intervals with utilizations at each level. The histograms show that the distribution of the number of intervals in which the utilizations achieve specific levels is highly sensitive to their lengths. The wide variation in distributions and in the frequencies with which the peaks occur suggests that peak utilizations should be a concern only when tolerance for variability of the overall response time is very small, especially when the peaks have short duration and when the average utilization is a good deal smaller than the peak. It is

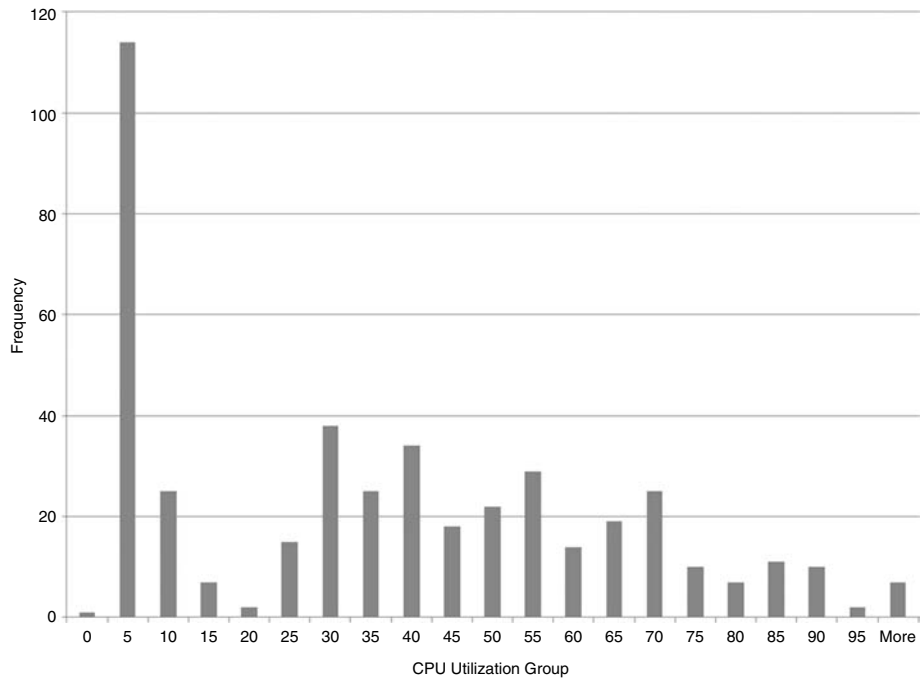


Figure 14.2 *Histogram showing the distributions of the numbers of intervals with different measured utilizations*

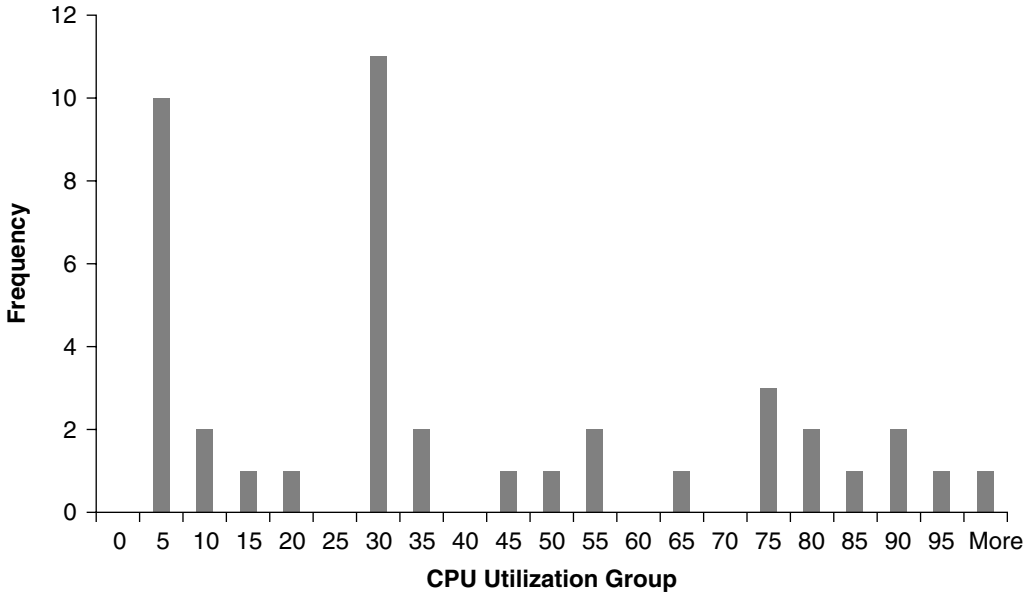


Figure 14.3 *Histogram showing the distributions of the numbers of pairs of intervals with different average utilizations*

important that all stakeholders be aware of this so that effort is not unnecessarily expended on trying to get rid of those peaks. The peaks may occur when a transaction starts or during a particular phase of the transaction and are therefore predictable. The absence of utilization in the presence of load should be a much greater cause for concern, since it indicates a system malfunction, perhaps due to an error in concurrent programming such as deadlock.

In this example, the focus of attention was the value of a single metric, the CPU utilization. The completion times or response times of the tasks at hand, including those of maintenance tasks running in the background, should also have been considered in the discussion. An examination of the response times during the same observation periods might have revealed that the response times did not reach intolerable levels even at the desired throughput. In that case, the variability in the utilizations should not have been of concern. That is one part of the performance story. Another part of the performance

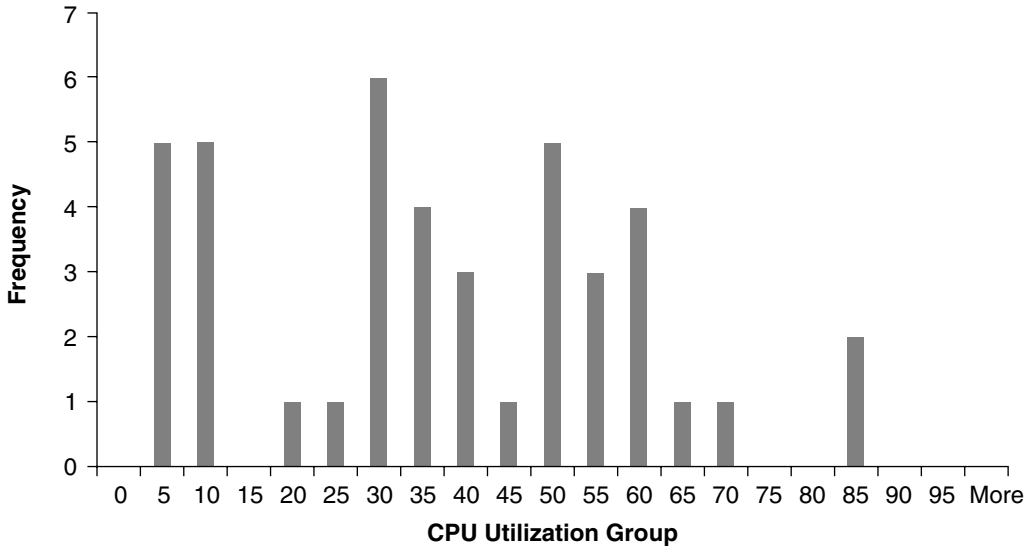


Figure 14.4 *Histogram showing the distributions of the numbers of groups of ten intervals with different average utilizations*

story that may have been overlooked by stakeholders is that the choice of measurement granularity and the choice of the length of the measurement intervals can have a significant impact on one's view of the system. This occurs because short intervals might exaggerate the apparent effect of peak values, while measurement intervals that are too long might mask them altogether. In this case, the actual measurements and histograms of their values were combined to explain this point. The lesson to be drawn from this example is that a careful examination of more than one performance metric and an examination of the evolution of the metrics over time should be done before determining that a problem exists whose resolution could require considerable staffing efforts.

14.9 Ensuring Ownership of Performance Concerns and Explanations by Diverse Stakeholders

If a sense of ownership of performance is not widespread in a software organization, it may be necessary for the performance engineer to foster one so that all stakeholders are sensitive to performance concerns and adjust their efforts accordingly. Three likely hurdles to this are

- A culture in which it is more important to get new features and functionality out the door than it is to ensure that the existing ones are fast and reliable
- A mistaken belief that performance can be tuned once a system is built
- A mistaken belief that performance is someone else's concern in the organization, and that one should focus on one's own assigned tasks

The first of these hurdles may be difficult to overcome, unless there is an emerging social consensus that users prefer having an efficient, small set of features rather than an overwhelmingly large set of features. It may be countered by the knowledge that at least one study shows that poor performance is the biggest risk to a software project [Bass2007]. Overcoming the second hurdle requires the dint of persuasion and socialization among architects. Empowerment and enabling stakeholders to achieve demonstrable successes may be the best ways to overcome the third hurdle, especially if the cost of making a system change and carrying out the testing needed to validate it is not prohibitive.

14.10 Negotiating Choices for Design Changes and Recommendations for System Improvement among Stakeholders

Sometimes, there will be resistance to implementing a remedy for a performance issue on the grounds that it may be too complicated, or that the implementation may incur a high software risk that can be mitigated only by subjecting the modified system to extensive regression testing. Let us return to the museum checkroom example. If the floor plan and space constraints prevent the implementation of separate queues with priority scheduling for those collecting coats to

prevent deadlock, it may be necessary to propose a different remedy that may reduce the risk of deadlock while not preventing it entirely.

1. One alternative method would be to redirect visitors wishing to leave coats to another checkroom when the number of occupied coat hangers reaches a designated threshold. The hanger occupancy level at the alternate checkroom should be very low. Notice that a solution involving the redirection of those collecting their coats is not feasible, since the coats will remain where they were left. Of course, this proposal means shifting the load elsewhere. If visitors leaving coats at the second checkroom and those collecting them go through a common FCFS queue there, the problem will only be shifted away from the first checkroom. It will not go away.
2. A second alternative method might be to allow the checkroom attendant to summon those collecting their coats to the head of the queue when all hangers are occupied. A single queue can be used to access the checkroom, but there must be a way to label the visitors wishing to pick up their coats, so that the attendants can identify them. Moreover, a mechanism must be devised for the attendants to search for visitors wishing to pick up their coats, if there are any in the queue. Of course, this will contribute to the time taken to resume service once deadlock has occurred.
3. A third alternative method might be to do nothing to change the system. Instead, the owner is supplied with guidelines for determining the maximum tolerable load at which deadlock is unlikely, and then sheds any load in excess of that by randomly refusing to accept coats. This method is not recommended because controlling the load to prevent traffic spikes is complex and will not change the sequence in which visitors join the queue or the sequence in which they are served.

Of these alternative solutions, the first simply kicks the problem down the road. Deadlock occurs because the order in which visitors are served is independent of the function they wish to perform. The second method alters the order in which visitors are served only when all the hangers are full but incurs the additional processing cost of identifying those visitors wishing to collect their coats. The third method is not a solution at all, because it will not prevent or resolve deadlock with certainty even if load is shed when the hanger occupancy is high.

How might the performance engineer and the stakeholders negotiate among four choices of actions, of which only the potentially most costly provides a guarantee of deadlock avoidance? The decision should depend on the risks to the users and other system stakeholders associated with the occurrence of deadlock and the time to recover from deadlock if it occurs. Since the occurrence of deadlock depends on both load and the sequence in which resource allocations and deallocations occur, it is difficult to say how frequently deadlock would occur, although the identification of the operation regions in which it is likely is fairly straightforward. Unfortunately, even demonstrating the ineffectiveness of the three alternative methods may not be sufficient to persuade stakeholders to adopt the most reliable method. The assessment of risk depends on the nature of the problem domain. The risks associated with any of the three alternatives might be acceptable in a museum. It is possible that the risks associated with any of them might not be acceptable in a mission-critical system. It is quite possible that the risks associated with any of the three alternatives are acceptable to the supplier, though perhaps not to the buyer, while the development risk associated with the deadlock-free method may not be acceptable to the builders and suppliers of the system, since they must bear the cost of thorough testing and maintenance. In this situation, the performance engineer's best course of action may be to provide a clear written assessment documenting the benefits and deficiencies of each method, being careful to explain the risks associated with the occurrence of deadlock, the circumstances under which it is likely to occur, and the time taken to recover from it. This will at least ensure that all stakeholders are aware of the consequences of any of the choices considered, even if the best one is not used. The report will also provide evidence that the performance engineer has exercised due diligence in this matter.

14.11 Summary

In many ways, the practice of performance engineering is like resolving the story of the three blind men and the elephant. One man touched the elephant's tail and said an elephant was like a rope. One touched the elephant's leg and said it was like a tree. Another touched the elephant's trunk and said it was a like a snake. All three men were correct, but none on his own could fully describe the elephant. One must communicate with multiple stakeholders to gain a solid idea of a system's

purpose, architecture, and function. Depending on the position of the system in the software lifecycle, one must also elicit performance requirements, carefully gather and interpret performance measurements, build models to make performance predictions, and effectively communicate one's understanding of the results to influence the decisions that will be made about the architecture of the system, how much load it can support, and changes that could or must be made to the system to improve its performance. All of this must be achieved with the cooperation of a diverse set of stakeholders. These stakeholders usually possess the information needed to understand a system. They sometimes have the ability to influence the decisions that might be made to implement recommendations provided by the performance engineer, often based on their input.

14.12 Exercises

14.1. The state of Catawba has its own successful health insurance exchange as authorized by the Affordable Care Act. Catawba decides to offer a state-subsidized dental insurance scheme to senior citizens who do not have other coverage, because publicly insured dental care is not provided by their government-run insurance plans. You are a performance engineer who has been called in by the Catawba health department to advise on ensuring the performance of the system with the added workload.

(a) Describe a general high-level plan for a performance process to ensure the smooth introduction of a web-based scheme for applying for the new dental insurance scheme. Your plan should include steps for baselining the load and performance of the existing system, estimating the system demand of the new service, and ensuring the performance of the new service on the same platform as the existing services. Do you need to conduct a full performance study of the existing system before you proceed? Explain.

(b) Identify those parts of your plan that can be executed regardless of changing rules, functional requirements, and changing performance requirements.

(c) Identify those parts of your plan that depend on changing functional and performance requirements.

(d) Devise an outline for a project plan for performance assurance of the new exchange system and identify the stakeholders you must work with from inception to the introduction of the service. The outline should be quite general. It may be needed subsequently, as indicated in the next item.

(e) Two years after the extension of dental care to seniors, the Catawba state legislature decides to offer dental insurance to Medicaid recipients as well (Medicaid is a health coverage scheme for the poor below retirement age). Explain how your responses to questions (a)–(d) can be extended to support this. Identify the stakeholders involved and make the case for your participation in this extension from inception through development, testing, and production.

(f) Make the case for developing a performance test plan for this system. Identify the ranges of workloads that the test plan should cover.

14.2. A university hosts a department of archaeology, a department of natural history, and a medical school. The three faculties decide to jointly produce an openly accessible, widely advertised online archive to display documents and images of their holdings, including static medical images of archaeological and zoological artifacts. The system will also contain links to the catalog of the university's library so that one can research related holdings. The system will be built by the university's IT department with occasional assistance from the computer science department. Funding will be provided by the national government and private foundations. Quick response times are an explicitly but vaguely stated condition for funding. As a performance engineer, you have been asked to join the team that will design and build this system.

(a) Explain why you would have liked to be part of the team that wrote the grant proposal and negotiated the statement of work with the funding agency. Your audience consists of a committee including the head of the office that manages government funding for the university, the heads of the three departments, the head of the library's IT department, and the head of the university's IT department. Make the case for being part of such an effort in the future.

(b) Since the demand for the system is unknown, the architecture of the system will have to be scalable either upward or downward. Before you write performance requirements,

prepare a written list of questions for discussions with the system architect about how the system might be built to cope with unpredictable demand in a cost-effective and easily administrable manner. Bear in mind that one of your objectives is to guide the architect in the formulation of a request for tender for equipment and software (and possibly hosting) to meet the need for flexible sizing.

(c) Identify the stakeholders who have an interest in response time requirements. Explain the possible range of resource requirements and time needed for the delivery of images and captions. What kinds of activities should the response times include? At what points in the delivery chain does a response time begin and end? How will you negotiate response time budgets with the various stakeholders?

14.3. A large manufacturer of automated industrial equipment will demonstrate its technical prowess to the public by hosting a tent at the 210th anniversary of a beer festival in 2020. The tent will feature the automated dispensing and delivery of different varieties of beer and other drinks to customers seated at up to ten tables. Each place at the tables will be uniquely numbered and equipped with a radio-frequency identification chip (RFID). The system will also automatically prepare and deliver sausages and other traditional foods. Delivery will be in closed, uniquely identified containers via a combination of conveyor belts and cable cars suspended from the ceiling of the tent. Order entry and payment will be via apps in customer-owned handheld devices and via touchscreens that are permanently mounted at the tables. The devices will communicate with the order entry system via a dedicated Wi-Fi. Customers will return used containers to one or more drop-off points from which the containers will be sent to a station for washing and reuse. The dispensing of beer will be done in glassed-in areas that are visible to the customers. Since the late delivery of beer and the delivery of cold sausages will lead to bad publicity, a performance engineer has been recruited to work on the project.

(a) Identify possible key components of this system and their potential load drivers.

(b) Outline the performance requirements of a system that texts order status messages to the customers.

(c) Identify the stakeholders in the key components of the system and their potential customers. Describe how you would

outline a performance engineering process for each component. Briefly describe the workloads and the nature of the performance requirements for each one, including the cable car system and the conveyor system. Be prepared to discuss the impacts of the workloads and performance requirements with architects, developers, and testers. Do not forget alarm handling capabilities.

(d) Explain the advantages of prototyping the system with a single table from a performance standpoint.

(e) Since the beer festival is a few years off, explain how your performance process would cope with changes in technology for order entry and payment occurring up to a year before the festival starts.

(f) During performance testing, it is observed that the time from order entry to the delivery of beer is exceedingly slow, even when only two people are sitting at a single table. Describe a test and measurement plan for determining whether the cause of the problem is in the order entry system, the delivery system, or elsewhere.

Chapter 15

Where to Learn More

Although computer technology has changed rapidly, the basic principles of performance engineering have changed little. Rather than focusing on specific technologies and computer architectures, we have sought to acquaint the reader with basic performance engineering principles related to measurement, performance testing, performance requirements engineering, performance modeling, and the link between these and scalability. We have not written about the principles of statistics that can be applied, as these are well documented in other texts. Moreover, we have not covered the application of performance engineering techniques to every conceivable problem domain or operating environment. These environments changed rapidly while the book was being written and will continue to do so. In this chapter we introduce the reader to texts that cover queueing theory, statistics, and simulation and point the reader to papers and books that cover specialized topics in performance engineering. In doing so, we hope to acquaint the reader with the mindset needed to deal with the performance aspects of new technologies and system architectures as they arise.

15.1 Overview

Performance engineering is a rich area that uses techniques, tools, and disciplines from many different fields, depending on the nature of the problem at hand. In this book we have shown how basic performance

modeling can inform thinking about performance and functional testing, scalability, architecture, performance requirements, and the definition and identification of workloads. The disciplines on which performance engineering draws include, but are not limited to, operating systems principles, computer architecture, statistics, and various techniques used in operations research, such as discrete event simulation and optimization. Discrete event simulation is particularly useful for modeling scheduling rules that cannot be modeled simply with analytical queueing models. Optimization techniques can be applied to a wide variety of problems, including deciding how to allocate load among various servers. In this chapter we point to sources for information on topics related to performance engineering that we have not discussed in depth or that we have only mentioned in passing. Many of the references we mention here have also been mentioned earlier in this book, while some have not.

Conference proceedings and journals can be used to track how the field is evolving and to find information about questions that may have already been addressed by others. There are a number of books that focus on the application of methods from statistics and operations research to performance engineering, as well as books containing chapters that address specialized topics such as databases, networking, and performance issues in concurrency.

Performance evaluation has a long and rich history going back to the 1960s. Many of the issues one encounters in computer systems today have analogies to problems that have been encountered before. Some of the issues, such as those related to memory management, concurrency control, and scheduling, keep reappearing in various guises, even though the technology on which they are run or the programming languages and environments in which they occur have changed.

The reader should not view a reference with skepticism or disdain merely because it is "old." It might still be relevant. The early work on computer performance evaluation and modeling was done by practitioners and researchers trained in such disciplines as operations research, physics, mathematics, and statistics. Related work on queueing theory, including models of systems that lose calls, goes back to the 1920s. Moreover, computer performance modeling problems often have analogies in manufacturing and other areas usually addressed by operations research or industrial engineering. For example, the first product form closed queueing network model was used to model carts in a coal mine in the 1950s [Koenigsberg1958]. Similar models have been applied to flexible manufacturing systems [VinSol1985] and to

computer systems [Buzen1973]. A performance engineer should be open to exploring these, too. While the examples in books and the applications in papers reflect the technology of the times in which they were written, many of the principles and much (but not all) of the mathematics they describe have remained the same.

Because it is not feasible to prepare a complete list of articles and books related to computer performance evaluation and engineering, the reference list at the end of the book and the material cited here should be regarded as only a starting point for further investigation.

In the next section we refer the reader to conferences and journals containing articles that are particularly relevant to the subject of this book. In subsequent sections we introduce the reader to books and other references in various related subject areas, including performance analysis, queueing theory, statistics, discrete event simulation, and performance tuning.

15.2 Conferences and Journals

There are a number of conference proceedings and journals that are focused on performance-related topics.

The Computer Measurement Group (CMG, www.cmg.org) hosts a series of conferences that cover a very broad range of performance-related topics. Much of their emphasis is on the application of measurement techniques to systems on a variety of platforms and applications, including commercial and open-source databases and cloud-based applications. Capacity planning issues receive a lot of attention here. The subtitle of the group's 2013 conference was "Measuring the Impact of Virtualization, Cloud, and Big Data." The conference also hosts exhibits by vendors of software of interest to those working in performance evaluation, such as modeling and measurement tools. Regional groups provide a forum for the discussion of performance topics of local interest. CMG also issues a periodical containing articles addressing practical problems in performance engineering.

The Standard Performance Evaluation Corporation (SPEC, http://spec.org) is devoted to the development and maintenance of standardized benchmarks whose performance can be evaluated on a variety of platforms. At the time of writing, it hosts a research group, an open systems group, a high-performance computing group, a graphics and workstation performance group, and several project groups.

The Association for Computing Machinery's Special Interest Group on Performance Evaluation, known as SIGMETRICS (www.sigmetrics.org), is devoted to topics in performance measurement and modeling. It sponsors an annual conference and issues several editions of a bulletin, *Performance Evaluation Review*, each year. Together with its sister special-interest group on software engineering, ACM SIGSOFT, it has sponsored workshops focused on software aspects of performance known as the Workshop on Software and Performance (WOSP).

In 2010, WOSP and SPEC held their first jointly organized conference on performance engineering, known as the International Conference on Performance Engineering (ICPE). These are the URLs of the 2013 and 2014 conferences respectively: http://icpe2013.ipd.kit.edu/ and http://icpe2014.ipd.kit.edu/. (*Disclosure:* The author of the present volume currently serves as the chairman of the steering committee of WOSP.)

The journal *Performance Evaluation* contains many articles on mathematical models to predict the performance of various types of systems, protocols, and scheduling rules. Articles on topics related to performance can also be found in such journals as the *IEEE Transactions on Software Engineering*, the *Journal of the Association for Computing Machinery (JACM)*, *Communications of the Association for Computing Machinery (CACM)*, *Computers and Operations Research*, and many others.

15.3 Texts on Performance Analysis

The present volume emphasizes the relationship between performance engineering and various stages in the software lifecycle. We have shown that there are relationships between the properties of basic performance models, performance requirements, the results of performance tests, and scalability. Prior authors have taken complementary approaches and cover different ground. The following descriptions should be seen as a starting point for further study. This is not a complete list, nor are the descriptions complete.

Neil Gunther's *The Practical Performance Analyst* describes more queueing models than we have here. The book also contains studies of transient behavior in packet switches and circuit switches. There is a

great deal of mathematical analysis of various properties of queueing systems, and applications of performance evaluation techniques to client/server systems and other topics [Gunther1998].

David Lilja's *Measuring Computer Performance* [Lilja2000] contains a detailed discussion on performance metrics, together with applications of statistics to performance engineering. In addition, there are chapters on errors in measurement, simulation modeling and analysis, benchmarking techniques, and basic statistical methods for analyzing measurements.

In *Performance Solutions*, Smith and Williams explore such topics as the application of UML to performance engineering, execution models (which can be used for performance budgeting, among other activities), distributed web-based applications, and embedded systems. Their book also contains an extensive discussion of performance patterns and performance antipatterns [SmithWilliams2001].

Quantitative System Performance by Lazowska et al. is an extensive description of the state of the art in performance evaluation as it stood when the book first appeared. In addition to discussing basic performance models, the book has extensive descriptions of approximate models of local area networks and I/O systems with control channels. These topics are worth exploring, because they deal with such issues as the simultaneous possession of resources (e.g., I/O channels and read/write disk head) that one might encounter today when modeling the simultaneous possession of nested software locks [LZGS1984].

The Art of Computer Systems Performance Analysis by R. K. Jain covers many aspects of probability theory, statistics, queueing theory, modeling, experimental design, and simulation that we have not covered here [Jain1991].

For recent information on performance benchmarking, the reader is referred to the proceedings of the 2009 SPEC Benchmark Workshop [KS2009] and the book edited by John and Eeckhout [JE2006].

Performance Modeling and Design of Computer Systems: Queueing Theory in Action, by Mor Harchol-Balter, contains a comprehensive survey of queueing theory and probability theory applied to performance modeling. In addition to covering queueing systems in equilibrium, Harchol-Balter covers recent topics such as the influence of high variability and probability distributions with heavy tails on the performance of web-based systems. The book also addresses performance issues in large-scale server farms [HarcholBalter2013].

15.4 Queueing Theory

The literature in queueing theory is too rich for us to go into in detail here. Classical texts on queueing theory include *Queues* by Cox and Smith [CoxSmith1961], *Introduction to Queueing Theory* by Cooper [Cooper1981], and the two volumes by Kleinrock, *Queueing Systems, Volume 1: Theory* [Kleinrock1975] and *Queueing Systems, Volume 2: Applications* [Kleinrock1976]. Books on specialized topics in queueing theory abound. For example, queueing models of various complex systems are explored in [Neuts1981], and queues with customers who give up queueing and retry later are explored in [FaTemp1997].

15.5 Discrete Event Simulation

In Chapter 3 we showed how basic queueing models could be used to predict the average performance measures of computer systems under a fairly restricted set of conditions. Those models are analytic queueing models. In analytic queueing models, the performance predictions are based on computations of the values of closed form expressions or on algorithms for solving sets of equations. By contrast, discrete event simulations are computer programs that mimic the logic of scheduling, event handling, and routing combined with scheduled arrivals and service times to estimate performance. Arrivals, service completions, and routing decisions occur based on the values of variables whose values are generated using pseudo–random numbers. Discrete event simulations can be used to investigate the evolution of system performance over time based on decision logic and scheduling rules that may be more complicated and less restrictive than the assumptions that underlie analytical performance models. Simulations can also be driven by traces of time-stamped events that have been recorded in a log to determine resource utilizations, response times, and the like. They can also be used to determine the operating conditions under which undesirable behaviors such as deadlocks occur.

While analytical models are usually used to predict average performance, simulation models can be used to investigate transient effects such as the buildup in resource utilization when load generation starts and the buildup in queues and the decline in utilizations that occur

when part of a system goes into deadlock. Discrete event simulations can be combined with the numerical solution of differential equations to predict system behavior in the presence of complex phenomena [Pritsker1986]. The choice of parameters and statistical distributions to model the scheduling of events is called *input analysis*. The analysis of the simulation results is called *output analysis*. Both involve the use of common statistical methods for fitting model parameters, the planning of experiments, and the statistical analysis of the simulation outputs. Techniques for programming simulations are described in several texts, including [Pritsker1986] and [LawKelton1982]. Essential methods for the analysis of simulation data and modeling are given in [LawKelton1982] and [Fishman2001].

The statistical methods used to plan simulation studies and their outputs are also applicable to the planning and analysis of performance tests. The planning of experiments can be used to reduce the variance in the experimental output so that the underlying factors causing particular performance properties can be more easily identified. Commercial and open-source packages exist to support simulations. The development of basic simulation tools for generating random variates, scheduling events, and gathering statistics on utilizations, queue lengths, and response times can be done as an assignment in a university course.

15.6 Performance Evaluation of Specific Types of Systems

As we mentioned previously, the examples in books on performance evaluation and queueing theory are usually related to technologies that are of interest at the time of writing. The present volume concerns general principles, software engineering aspects, and foundations of performance engineering. Conference papers, journal articles, and other books may cover types of systems of particular interest that we have not covered here. For example, performance aspects of cloud and enterprise computing are specifically addressed in [Gregg2013], and server farms are addressed in [HarcholBalter2013]. Capacity planning, basic modeling, and examples related to e-business are described in [MenasceAlmeida2000], and capacity planning and modeling for web services are described in [MenasceAlmeida2002].

15.7 Statistical Methods

Performance engineers draw on statistical and operations research methods that are essential to any quantitative study. We have not covered them in detail in this book because they are well documented in many of the texts already cited.

Standard spreadsheet packages such as Microsoft Excel contain standard statistical analysis tools. Readers wishing to understand the basis of statistical methods such as linear regression and the fitting of distributions may wish to begin by consulting [VenRip2002], which discusses the underlying principles as well as their application using statistical tools such as S and R. R is widely available. In particular, linear regression analysis should be used to fit equations to obtain service times from measured resource utilizations when the throughputs are known and the relationship between the throughputs and utilizations appears to be linear up to the point of saturation. Thus, regression analysis can be used to fit the values of the products $V_i S_i$ using observed utilizations, observed throughputs, and the Utilization Law, $U_i = X_0 V_i S_i$.

The methods of experimental design may be used to plan performance tests and modeling studies. When confronted with a large number of factors and a complicated parameter space for performance testing and modeling, fractional designs can be used to choose a pertinent subset of the parameter space for measurement and analysis [AndersonMcLean1974].

Standard methods of experimental design are not the only methods one might use to plan performance tests. Avritzer and Weyuker advocate selecting the parameters of performance tests based on a Markov chain representation of the states a system may enter [AvWey1995], with the aim of finding performance issues while the system is in each state or each group of states.

15.8 Performance Tuning

The focus of this book is foundations and principles of software and systems performance engineering. We have not devoted much space to the performance tuning of types of architectures or of specific software platforms because new ones emerge frequently while new releases of existing ones may change how they behave and how they should be configured. There are performance tuning books on Java

[CockcroftPettit1998], Oracle databases [AKP2013], Sun-based systems [CockcroftPettit1998], the cloud [Gregg2013], Network File System (NFS) [Olker2002], and many other topics that are too numerous to list here. Their contents should be regarded as complementary to the basic performance engineering principles we have presented in this volume. We should also reiterate that performance tuning will not compensate for the inadequacies inherent in a poorly conceived performance architecture.

15.9 Summary

Software and system performance engineering encompass many disciplines. New performance issues arise as technology changes, yet the underlying principles of measurement and analysis remain unchanged. Performance issues that were identified and studied many years ago will continue to recur in different guises. The literature we have cited in this chapter and elsewhere in this book covers operating systems principles, performance engineering, database design, queueing theory, statistics, simulation, operations research, and requirements engineering among many others. The author has used all of these disciplines and more to assure the sound performance of the systems on which he has worked. It is hoped that the reader will be able to do likewise.

This page intentionally left blank

References

[AIPWEBSITE] Heisenberg's Uncertainty Principle paper, quoted at www.aip.org/history/heisenberg/p08.htm.

[AKP2013] Alapati, S., P. Kuhn, and B. Padfield. *Oracle 12c Performance Tuning Recipes: A Problem Solving Approach.* APress, 2013.

[AlmesLaz1979] Almes, G. T., and E. D. Lazowska. The behavior of Ethernet-like computer communications networks. Technical Report no. 79-05-01, Department of Computer Science, University of Washington, April 1979.

[AlmesLazowska1982] Almes, G. T., and E. D. Lazowska. The behavior of Ethernet-like communication networks. *ACM SOSP '79, Proceedings of the Seventh ACM Symposium on Operating Systems Principles,* 66–81, 1979.

[AndersonMcLean1974] Anderson, V. I., and R. A. McLean. *Design of Experiments: A Realistic Approach.* Marcel Dekker, 1974.

[ARSTECHNICA2012] http://arstechnica.com/science/2012/06/faster-than-light-neutrino-findings-really-thoroughly-dead/.

[AvBon2012] Avritzer, A., and A. B. Bondi. Resilience assessment based on performance testing. In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M. Vieira, and A. van Morsel. Springer, 2012.

[AvBonWey2005] Avritzer, A., A. B. Bondi, and E. Weyuker. Ensuring stable performance for smoothly degrading systems. *Proc. 5th International Workshop on Software and Performance (WOSP 2005),* Palma, Spain, 2005.

[AvColeWey2007] Avritzer, A., E. Weyuker, and R. G. Cole. Using performance signatures and software rejuvenation for worm mitigation in tactical MANETs. *Proc. 6th International Workshop on Software and Performance (WOSP 2007),* Buenos Aires, Argentina, 172–180, 2007.

[AviHey1973] Avi-Itzhak, B., and D. P. Heyman. Approximate queueing models for multiprogramming computer systems. *Operations Research* 21 (1), 1212–1230, 1973.

[AvTanJaCoWey2010] Avritzer, A., R. Tanikellea, K. James, R. G. Cole, and E. Weyuker. Monitoring for security intrusion using performance signatures. *Proc. First Joint WOSP/SIPEW International Conference on Performance Engineering 2010*, San Jose, California, 93–104, 2010.

[AvWey1995] Avritzer, A., and E. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng.* 21 (9), 705–716, 1995.

[AvWey1999] Avritzer, A., and E. Weyuker. Deriving workloads for performance testing. *Software: Practice and Experience* 26 (6), 613–633, 1999.

[Bass2007] Bass, Len, Robert L. Nord, William Wood, and David Zubrow. Risk themes discovered through architecture evaluations. WICSA 2007, Mumbai, India, January 2007.

[BCM2004] Blackburn, S. M., P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. *Proc. ACM SIGMETRICS 2004,* 25–36, 2004.

[BCMP1975] Baskett, F., K. M. Chandy, R. R. Muntz, and F. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *JACM* 22 (2), 248–260, 1975.

[BhatMiller2002] Bhat, U. N., and G. K. Miller. *Elements of Applied Stochastic Processes*. Wiley-Interscience, 2002.

[Boehm1988] Boehm, B. A spiral model of software development and enhancement. *IEEE Computer* 21 (5), 61–72, 1988.

[Bok2010] Bok, Derek. The Politics of Happiness: What Government Can Learn from the New Research on Well-Being. Princeton University Press, 2010.

[Bondi1989] Bondi, A. B. An analysis of finite capacity queues with common or reserved waiting areas. *Computers and Operations Research* 16 (3), 217–233, 1989.

[Bondi1992] Bondi, A. B. A study of a state-dependent job admission policy in a computer system with restricted memory partitions. *Performance Evaluation* 15 (3), 133–153, 1992.

[Bondi1997a] Bondi, A. B. A model of the simultaneous possession of agents and trunks with automated recorded announcement. In *Proc. ITC15*, edited by V. Ramaswami and P. E. Wirth, 1347–1358. Elsevier, 1997.

[Bondi1997b] Bondi, A. B. A non-blocking mechanism for regulating the transmission of network management polls. *Proc. ISINM97*, 565–580, San Diego, California, May 1997.

[Bondi1998] Bondi, A. B. Network management system with improved node discovery and monitoring. US Patent No. 5710885, issued January 20, 1998.

[Bondi2000] Bondi, A. B. Characteristics of scalability and their impact on performance. *Proc. 2nd International Workshop on Software and Performance (WOSP 2000),* Ottawa, Canada, 195–203, September 2000.

[Bondi2007a] Bondi, A. B. Automating the analysis of load test results to assess the scalability and stability of a component-based SOA-based system. *Proc. CMG 2007*, San Diego, California, December 2007.

[Bondi2007b] Bondi, A. B. Experience with incremental performance testing of a system based on a modular or service-oriented architecture. *Proc. ROSATEA,* Medford, Massachusetts, July 2007.

[BondiBuzen1984] Bondi, A. B., and J. P. Buzen. The response times of priority classes under preemptive resume in M/G/m queues. *Proc. ACM SIGMETRICS 1984,* 195–201, 1984.

[BondiJin1996] Bondi, A. B., and V. Y. Jin. A performance model of a design for a minimally replicated distributed database for database-driven telecommunications services. *Distributed and Parallel Databases* 4, 295–317, 1996.

[BondiRos2009] Bondi, A. B., and J. Ros. Experience with training a remotely located performance test team in a quasi-agile global environment. *Proc. International Conference on Global Software Engineering*, Limerick, Ireland, July 2009.

[BondiWhitt1986] Bondi, A. B., and W. Whitt. The influence of service-time variability in a closed network of queues. *Performance Evaluation* 6, 219–234, 1986.

[BOP1994] Brakmo, L. S., S. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. *Proc. ACM SIGCOMM 1994*, 24–35, 1994.

[BPKR2009] Berenbach, B., D. Paulish, J. Kazmeier, and A. Rudorfer. *Software & Systems Requirements Engineering in Practice.* McGraw-Hill, 2009.

[Browne1981] Browne, J. C. Designing systems for performance. Keynote address, ACM SIGMETRICS Conference, Las Vegas, Nevada, 1981. *Performance Evaluation Review* 10 (1), 1, 1981.

[BruellBalbo1980] Bruell, S. C., and G. Balbo. *Computational Algorithms for Closed Queueing Networks.* North-Holland, 1980.

[BSA2005] Bondi, A. B., C. S. Simon, and K. W. Anderson. Bandwidth usage and network latency in a conveyor system with Ethernet-based communication between controllers. *Proc. IEEE PacRim 2005,* Victoria, BC, August 2005.

[Burleson2002] Burleson, D. K. *ORACLE9i High Performance Tuning with STATSPACK*. Oracle Press/McGraw-Hill, 2002.

[Bush2007] Remarks by President Bush to the Naval War College, June 28, 2007. www.prnewswire.com/cgi-bin/stories.pl?ACCT=104&STORY=/www/story/06-28-2007/0004617850&EDATE=.

[Bux1981] Bux, W. Local area subnetworks: A performance comparison. *IEEE Trans. Communications* COM-29, 1645–1673, 1981.

[Buzen1973] Buzen, J. P. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM* 16, 527–531, 1973.

[BuzenBondi1983] Buzen, J. P., and A. B. Bondi. The response times of priority classes under preemptive resume in M/M/m queues. *Operations Research* 31 (3), 456–465, 1983.

[Carr2013] Carr, D. Obamacare insurance exchange websites: Tech critique. *Information Week,* October 3, 2013. www.informationweek.com/healthcare/policy/obamacare-insurance-exchange-websites-te/240162176?queryText=Affordable%20Care%20Act.

[CLGKP1994] Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys* 26 (2), 145–186, 1994.

[Cobham1955] Cobham, A. Priority assignment—a correction. *Operations Research* 3, 547, 1955.

[CockcroftPettit1998] Cockcroft, A., and R. Pettit. *Sun Performance and Tuning: Java and the Internet, Second Edition.* Sun Microsystems Press/Prentice Hall, 1998.

[CoffDenn1973] Coffman, E. G., and P. J. Denning. *Operating Systems Theory*. Prentice Hall, 1973.

[Cooper1981] Cooper, R. B. *Introduction to Queueing Theory, Second Edition.* Elsevier/North-Holland, 1981.

[CoxSmith1961] Cox, D. R., and W. Smith. *Queues.* Methuen, 1961; reprinted by Chapman and Hall, 1971, and CRC Press, 1999.

[CRPH2010] Cacéres, J., L. M. Vaquero, L. Rodero-Merino, A. Polo, and J. J. Herro. Scalability over the cloud. In *Handbook of Cloud Computing*, edited by B. Fuhrt and A. Escalante, 357–377. Springer, 2010.

[DDB1981] Denning, P. J., T. Dennis, and J. A. Brumfield. Low contention semaphores and ready lists. *Communications of the ACM* 24 (10), 687–699, 1981.

[Denning1980] Denning, P. J. Working sets past and present. *IEEE Trans. Softw. Eng.* SE-6 (1), 64–84, 1980.

[DenningBuzen1978] Denning, P. J., and J. P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys* 10 (3), 225–261, 1978.

[DGLS1999] Devlin, B., J. Gray, B. Laing, and G. Spix. Scalability terminology: Farms, clones, partitions, and packs: RACS and RAPS. Technical Report MS-TR-99-85, Microsoft Research, December 1999.

[Dijkstra1965] Dijkstra, E. W. Solution of a problem in concurrent programming control. *Communications of the ACM* 8, 569, 1965.

[Duboc2008] Duboc, L., Letier, E., Rosenblum, D., and Wicks, T. (2008), Case Study in Eliciting Scalability Requirements, in RE'08: Proceedings of the 2008 16th IEEE International Requirements Engineering Conference, Barcelona, Spain.

[Duboc2009] Duboc, L. A framework for the characterization and analysis of software systems scalability. Doctoral thesis, University College London, 2009. Obtainable at http://discovery.ucl.ac.uk/19413/.

[Eilperin2013] Eilperin, J. CGI warned of HealthCare.gov problems a month before launch, documents show. *Washington Post*, October 29, 2013. www .washingtonpost.com/blogs/post-politics/wp/2013/10/29/cgi-warned-of-healthcare-gov-problems-a-month-before-launch-documents-show/.

[EMIR2014] www.esma.europa.eu/page/European-Market-Infrastructure-Regulation-EMIR.

[Erlang1917] Erlang, A. K. Solution of some problems in the theory of probabilities of significance in automatic telephone exchanges. *Elektroteknikeren*, 13, 1917.

[EysRupp2010] Eysholdt, M., and J. Rupprecht. Migrating a large modeling environment from XML/UML to Xtext/GMF. *Proc. SPLASH* 10, 97–103, 2010.

[EZL1989] Eager, D., J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. on Computers* 38 (3), 408–423, 1989.

[FaTemp1997] Falin, G. I., and J. G. C. Templeton. *Retrial Queues.* Chapman and Hall, 1997.

[FAWOD2009] Franks, G., T. Al-Omari, M. Woodside, O. Das, and D. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Softw. Eng.* SE-35 (2), 148–181, 2009.

[Fishman2001] Fishman, G. S. *Discrete Event Simulation: Modeling, Programming, and Analysis.* Springer Series in Operations Research, 2001.

[Gawande2009] Gawande, Atul. *The Checklist Manifesto: How to Get Things Right.* Metropolitan Books, 2009.

[Gregg2013] Gregg, B. *Systems Performance: Enterprise and the Cloud.* Prentice Hall, 2013.

[Gunther1998] Gunther, N. J. *The Practical Performance Analyst.* McGraw-Hill/Authors Choice Press, 1998.

[Habermann1976] Habermann, A. N. *Introduction to Operating System Design*. SRA, 1976.

[HAKC2013] Hashemian, R., M. Arlitt, D. Krishnamurthy, and N. Carlsson. Improving the scalability of a multi-core web server. *Proc. International Conference on Performance Engineering (ICPE2013),* Prague, Czech Republic, 161–172, 2013.

[HarcholBalter2013] Harchol-Balter, Mor. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Cambridge University Press, 2013.

[HeidelbergerTrivedi1982] Heidelberger, P., and K. S. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *IEEE Trans. Comp*. C21 (11), 1099–1109, 1982.

[Hilkevitch2013] Hilkevitch, J. New O'Hare runway expected to boost traffic, ease delays. *Chicago Tribune,* October 17, 2013. http://articles.chicagotribune.com/2013-10-17/news/ct-met-ohare-runway-1017-20131017_1_new-runway-new-o-hare-second-runway.

[Hill1990] Hill, M. D. What is scalability? *ACM SIGARCH Computer Architecture News* 18 (4), 18–21, 1990.

[HIPAA2014] www.hhs.gov/ocr/privacy/hipaa/understanding/summary.

[Hoare1974] Hoare, C. A. R. Monitors: An operating system structuring concept. *Communications of the ACM* 17 (10), 549–557, 1974.

[Horikawa2011] Horikawa, T. An approach for scalability-bottleneck solution: Identification and elimination of scalability bottlenecks in a DBMS. *Proc. ACM/SPEC Second International Conference on Performance Engineering ICPE 2011*, Karlsruhe, Germany, 2011.

[HS1976] Horowitz, E., and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press, 1976.

[HSH2005] Ho, A., S. Smith, and S. Hand. On deadlock, livelock, and forward progress. Technical Report no. 633, Computing Laboratory, University of Cambridge, 2005. www.cl.cam.ac.uk/techreports/UCAM-CL-TR-633.pdf.

[HuffGeis1954] Huff, D., and I. Geis. *How to Lie with Statistics.* Penguin, 1954; reissued by Norton, 1993.

[Hyde2009] Hyde, R. The fallacy of premature optimization. *ACM Ubiquity Magazine*, article no. 1, February 2009. http://dl.acm.org/citation.cfm?id=1513451.

[IEEE830] IEEE Std 830-1998, IEEE recommended practice for software requirements specifications—description.

[ITGUYSBLOG] www.it-etc.com/2010/04/14/use-perfmon-to-monitor-servers-and-find-bottlenecks/.

[Jac1964] Jacobellis v. Ohio. https://supreme.justia.com/cases/federal/us/378/184/case.html.

[JacBrad1988] Jacobson, V., and R. Braden. TCP extensions for long-delay paths. IETF RFC1072, 1988. http://tools.ietf.org/html/rfc1072.

[Jackson1963] Jackson, J. R. Jobshop-like queueing networks. *Mgt. Sci.* 10, 131–142, 1963.

[Jain1991] Jain, R. K. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 1991.

[Jalote1994] Jalote, P. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1994.

[JE2006] John, L. K., and J. L. Eeckhout, eds. *Performance Evaluation and Benchmarking.* CRC/Taylor and Francis, 2006.

[JW2000] Jogalekar, P. P., and C. M. Woodside. A scalability metric for distributed computing applications in telecommunications. *Proc. Fifteenth International Teletraffic Congress (ITC-15)* 2a, 101–110, 1997.

[KatzKursh1986] Katznelson, J., and R. Kurshan. S/R: A language for specifying protocols and other coordinating processes. Fifth Annual International Phoenix Conference on Computers and Communications, 1986.

[Kerola1986] Kerola, T. The composite bound method for computing throughput bounds in multiple class environments. *Performance Evaluation* 6 (1), 109, 1986.

[Killelea2000] Killelea, Patrick. Java threads may not use all your CPUs. *Java World,* August 11, 2000. www.javaworld.com/article/2076147/java-web-development/java-threads-may-not-use-all-your-cpus.html.

[Kleinrock1975] Kleinrock, L. *Queueing Systems, Volume 1: Theory.* Wiley, 1975.

[Kleinrock1976] Kleinrock, L. *Queueing Systems, Volume 2: Applications.* Wiley, 1976.

[Koenigsberg1958] Koenigsberg, E. Cyclic queues. *Operations Research* 9 (1), 22–35, 1958.

[KS2009] Kaeli, D., and Kai Sachs. *Computer Performance Evaluation and Benchmarking: SPEC Benchmark Workshop,* Austin, Texas, January 2009. *Lecture Notes in Computer Science* 5419. Springer, 2009.

[Latouche1981] Latouche, G. Algorithmic analysis of a multiprogramming-multiprocessing computer system. *JACM* 28 (4), 662–679, 1981.

[LawKelton1982] Law, A. M., and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 1982.

[LeeKatz1993] Lee, E. K., and R. H. Katz. An analytic model of disk arrays. *Proc. ACM SIGMETRICS 1993*, 98–109, Santa Clara, California, 1993.

[Lilja2000] Lilja, David J. *Measuring Computer Performance: A Practitioner's Guide.* Cambridge University Press, 2000.

[Little1961] Little, J. D. C. A proof for the queuing formula L = λW. *Operations Research* 9 (3), 383–387, 1961.

[LZGS1984] Lazowska, E. D., J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance.* Prentice Hall, 1984. Also available online at www.cs.washington.edu/homes/lazowska/qsp/.

[MBH2005] Masticola, S., A. B. Bondi, and M. Hettish. Model-based scalability estimation in inception-phase software architecture. In *ACM/IEEE 8th International Conference on Model-Driven Engineering Languages and Systems, 2005. Lecture Notes in Computer Science* 3713, 355–366. Springer, 2005.

[MenasceAlmeida2000] Menasce, D. A., and V. A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning.* Prentice Hall, 2000.

[MenasceAlmeida2002] Menasce, D. A., and V. A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods.* Prentice Hall, 2002.

[Microsoft2007] Microsoft Corporation. *Performance Testing for Web Applications.* O'Reilly, 2007.

[MogRam1997] Mogul, J. C., and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 15 (3), 217–252, 1997.

[Mossburg2009] Mossburg, Marta. Happiness is no metric for a country's success. *Washington Examiner,* September 18, 2009. http://washingtonexaminer.com/article/33621#.UDvD2qC058E.

[MSOFTSUPPORT1] http://support.microsoft.com/kb/310067.

[Munin2008] Pohl, G., and M. Renner. *Munin: Graphisches Netzwerk- und System-Monitoring*. Open Source Press, 2008.

[NagVaj2009] Nagarajan, S. N., and S. Vajravelu. Avoiding performance engineering pitfalls. In *Performance Engineering and Enhancement,* SET Labs Briefings 7 (1), 9–14, Infosys, 2009. Available online at www.infosys.com/infosys-labs/publications/Documents/SETLabs-briefings-performance-engineering.pdf.

[Neuts1981] Neuts, Marcel. *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach.* Johns Hopkins University Press, 1981; reprinted with corrections by Dover Publications, 1994.

[NFPA2007] NFPA 72, National Fire Alarm Code, 2007 Edition, NFPA, Quincy, MA 02169-7471.

[Niemiec2012] Niemiec, R. *Oracle Database 11g Release 2 Performance Tuning Tips & Techniques.* Oracle Press/McGraw-Hill, 2012.

[Olker2002] Olker, D. *Optimizing NFS Performance.* HP/Prentice Hall, 2002.

[Paulish2002] Paulish, D. J. *Architecture-Centric Software Project Management: A Practical Guide.* Addison-Wesley, 2002.

[Pritsker1986] Pritsker, A. A. B. *Introduction to Simulation and SLAM II, Third Edition.* Halsted Press/Wiley, 1986.

[PS1985] Peterson, J. L., and A. Silberschats. *Operating System Concepts, Second Edition.* Addison-Wesley, 1985.

[ReeserHariharan2000] Reeser, P., and R. Hariharan. Analytic model of web servers in distributed environments. *Proc. 2nd International Workshop on Software and Performance (WOSP 2000)*, Ottawa, Canada, 2000.

[ReiserKobayashi1975] Reiser, M., and H. Kobayashi. Queueing networks with multiple closed chains: Theory and computational algorithms. *IBM J. of R. & D.* 19 (3), 283–294, 1975.

[ReisLav1980] Reiser, M., and S. S. Lavenberg. Mean value analysis of closed multichain queueing networks. *JACM* 27 (2), 1980.

[Rey1983] Rey, R. F., ed. *Engineering and Operations in the Bell System*. AT&T Bell Laboratories, 1983.

[SB2003] Shasha, S., and P. Bonnet. *Database Tuning.* Morgan Kaufmann, 2003.

[SevMit1981] Sevcik, K. C., and I. Mitrani. The distribution of queueing network states at input and output instants. *JACM* 28 (2), 358–371, 1981.

[ShaibalSugiyama1996] Shaibal, R., and M. Sugiyama. *Sybase Performance Tuning.* Prentice Hall, 1996.

[Shaw1974] Shaw, A. C. *The Logical Design of Operating Systems*. Prentice Hall, 1974.

[Smith2000] Smith, C. Software performance antipatterns. *Proc. 2nd International Workshop on Software and Performance (WOSP 2000),* Ottawa, Canada, 127–136, 2000.

[SmithWilliams1998] Smith, C. U., and L. Williams. Performance engineering evaluation of CORBA-based distributed systems. First International Workshop on Software and Performance (WOSP98), 1998.

[SmithWilliams2000] Smith, C. U., and L. Williams. Software performance antipatterns. *Proc. 2nd International Workshop on Software and Performance (WOSP 2000),* Ottawa, Canada, 2000.

[SmithWilliams2001] Smith, Connie U., and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2001.

[SWH2006] Schroeder, B., A. Wierman, and M. Harchol-Balter. Open vs. closed: A cautionary tale. USENIX NSDI'06. http://static.usenix.org/events/nsdi06/tech/full_papers/schroeder/schroeder_html.

[Swift1912] Swift, J. *Gulliver's Travels.* Rand McNally, 1912.

[Thomson1950] Thomson, David. *England in the Nineteenth Century.* Penguin, 1950.

[TopManPage] http://linux.about.com/od/commands/l/blcmdl1_top.htm.

[TTH2012] Thomasian, A., Y. Tang, and Y. Hu. Hierarchical RAID: Design, performance, reliability, and recovery. *J. Parallel and Distributed Computing* 72 (12), 1753–1769, 2012.

[Tufte2006] Tufte, E. *Beautiful Evidence.* Graphics Press, 2006.

[VenRip2002] Venables, W. N., and B. D. Ripley. *Modern Applied Statistics with S, Fourth Edition.* Springer, 2002.

[VinSol1985] Vinod, B., and J. J. Solberg. The optimal design of flexible manufacturing systems. *Int. J. Production Research* 23 (6), 1141–1151, 1985.

[VXT2009] Die Beauftragte der Bundesregierung fuer Informationstechnik. *Das V-Modell XT.* www.v-modell-xt.de/, Version 3, February 2009.

[Webster1988] *Webster's Ninth New Collegiate Dictionary.* Collegiate, 1988.

[Whitt1984] Whitt, W. Minimizing delays in the GI/G/1 queue. *Operations Research* 32 (1), 41–51, 1984.

[WilliamsBhandiwad1976] Williams, A. C., and R. A. Bhandiwad. A generating function approach to queueing network analysis of multiprogrammed computers. *Networks* 6, 1–22, 1976.

[Willis2009] Willis, John. Percent disk time from Windows Perfmon can exceed 100%. www.gulfsoft.com/blog_new/modules.php?op=modload&name=News&file=article&sid=125.

[WindowsKB2013] Microsoft.com Knowledge Base. Synchronous and asynchronous I/O. http://msdn.microsoft.com/en-us/library/windows/desktop/aa365683%28v=vs.85%29.aspx, Build 4/16/2013.

[Wolff1982] Wolff, R. W. Poisson arrivals see time averages. *Operations Research* 30 (2), 223–231, 1982.

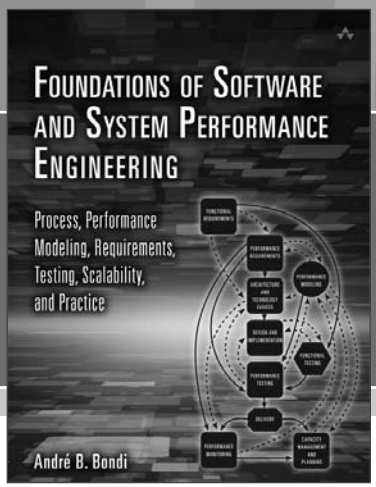
[XOWM2005] Xu., J., A. Oufimtsev, M. Woodside, and L. Murphy. Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates. *SAVCBS '05, Proceedings of the 2005 Conference on Specification and Verification of Component-Based Systems,* Article No. 5. ACM, 2005.

[Zahorjan1983] Zahorjan, J. Workload representations in queueing models of computer systems. *Proc. ACM SIGMETRICS, Conference on Measurement and Modeling of Computer Systems,* Minneapolis, 70–81, 1983.

Index

C

This page intentionally left blank



FREE Online Edition

Safari Books Online

Your purchase of ***Foundations of Software and System Performance Engineering*** includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Addison-Wesley Professional book is available online through Safari Books Online, along with thousands of books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly Media, Prentice Hall, Que, Sams, and VMware Press.

Safari Books Online is a digital library providing searchable, on-demand access to thousands of technology, digital media, and professional development books and videos from leading publishers. With one monthly or yearly subscription price, you get unlimited access to learning tools and information on topics including mobile app and software development, tips and tricks on using your favorite gadgets, networking, project management, graphic design, and much more.

Activate your FREE Online Edition at informit.com/safarifree

STEP 1: Enter the coupon code: MKYUQGA.

STEP 2: New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

Addison Wesley · Adobe Press · Alpha · Cisco Press · FT Press · IBM Press · Microsoft Press · New Riders · O'Reilly · Peachpit Press · Prentice Hall · Que · Redbooks · Sams · SAS Publishing · VMware Press · Wiley · Wrox