Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von

Samuel Kounev, MSc.

Referenten:
Prof. Alejandro P. Buchmann, PhD, TU-Darmstadt
Prof. Dr.-Ing. Dr. h.c. Günter Hommel, TU-Berlin

Tag der Einreichung:	03.05.2005
Tag der mündlichen Prüfung:	03.08.2005

Darmstadt 2005, Darmstädter Dissertation D17

To my father Douhomir (in memoriam),
who bought me my first computer,

my mother Ekaterina,
who encouraged me during my studies,

and to my wife Hanna,
who inspired me to write this thesis.

Abstract

Performance (noun) - the manner in which or the efficiency with which something reacts or fulfils its intended purpose.

– *Random House Webster's College Dictionary*

Distributed component-based systems (DCS) are becoming increasingly ubiquitous as enabling technology for modern enterprise applications. In the face of globalization and ever increasing competition, *Quality of Service (QoS)* requirements on such systems, like performance, availability and reliability are of crucial importance. In order to survive, businesses must ensure that the systems they operate, not only provide all relevant services, but also meet the performance expectations of their users. To avoid the pitfalls of inadequate QoS, it is important to analyze the expected performance characteristics of systems during all phases of their life cycle. However, as systems grow in size and complexity, analyzing their expected performance becomes a more and more challenging task. System architects and deployers are often faced with questions such as the following: Which deployment platform (hardware and software) would provide the best scalability and cost/performance ratio for a given application? For a given platform, what performance would the application exhibit under the expected workload and how much hardware would be needed to meet the service level agreements? In seeking answers to these questions, system developers nowadays often rely on their intuition, marketing studies, expert opinions, past experiences, ad hoc procedures or general rules of thumb. As a result, the overall system capacity is unknown and capacity planning and procurement are done without a sufficient consideration given to the QoS requirements. In the pressure to release new applications as early as possible, more often than not, performance is considered as an afterthought.

In this thesis, a systematic approach for performance engineering of DCS is developed that helps to identify performance and scalability problems early in the development cycle and ensure that systems are designed and sized to meet their QoS

requirements. The proposed approach builds on the fact that if a DCS is to provide good performance and scalability, both the platforms on which it is built and the system design must be efficient and scalable. To this end, we suggest that in the beginning of the system life cycle, the performance and scalability of the platforms chosen are validated using *standard benchmarks* and that *performance models* are then exploited to evaluate the system performance throughout the development cycle. In contrast to the "fix-it-later"approach, this approach helps to identify performance and scalability problems early in the system life cycle and eliminate them with minimal overhead.

Contributions in the Area of Benchmarking The first part of the thesis deals with the problem of measuring the performance and scalability of hardware and software platforms for DCS. The focus is placed on J2EE-based platforms since they are currently the technology of choice for DCS. The shortcomings of conventional benchmarks used in the J2EE industry are described and the need for industry-standard benchmarks is discussed. In order to be useful, benchmarks must meet the following important requirements - they must be representative of real-world systems, must exercise and measure all critical services provided by platforms, must not be biased in favor of particular products, must generate reproducible results and must not have any inherent scalability limitations. In the past five years several attempts were made to develop a benchmark satisfying these requirements and they demonstrated that the latter is an extremely challenging task requiring *far more* than good programming skills.

The first attempt was initiated in 2000 and resulted in the development of the ECperf benchmark. ECperf was developed under the Java Community Process (JCP) with participation of all major J2EE application server vendors. A huge amount of time, money and effort was invested in the specification and development of ECperf, however, while it met most of the requirements discussed above, unfortunately, it failed to meet all of them. We discovered several subtle issues in the design of ECperf, limiting the overall scalability of the benchmark and degrading its performance and reliability. These issues were related to the way long transactions were processed and persistent data was managed. We proposed a redesign of the benchmark in which asynchronous, message-based processing is exploited to reduce data contention in the database and improve the benchmark scalability. This eliminated the scalability bottleneck and addressed the identified performance and reliability issues.

The proposed redesign of the ECperf benchmark was submitted as an official proposal to the ECperf expert group at Sun Microsystems and later to the OSG-Java subcommittee of SPEC, which was responsible for the future of ECperf[1]. The

[1]In September 2002, the ECperf benchmark was taken over by SPEC and renamed to SPEC-jAppServer. SPEC's OSG-Java subcommittee was now responsible for the benchmark.

proposal was approved and the new design was implemented in a future version of the benchmark in whose development and specification the author was involved as release manager and lead developer. The new benchmark was called *SPECjAppServer2004* and was developed within SPEC's OSG-Java subcommittee, which includes BEA, Borland, Darmstadt University of Technology, Hewlett-Packard, IBM, Intel, Oracle, Pramati, Sun Microsystems and Sybase. Even though SPECjAppServer2004 is partially based on ECperf, it implements a *new* enhanced workload that exercises all major services of the J2EE platform in a complete end-to-end application scenario. Thus, SPECjAppServer2004 is substantially more complex than ECperf. SPECjAppServer2004 provides a *reliable* method to evaluate the performance and scalability of J2EE-based platforms and holds a number of advantages over conventional J2EE benchmarks. By modeling a realistic application and not being optimized for any particular platform, SPECjAppServer2004 provides a level playing field for performance comparisons of competing products. Furthermore, SPECjAppServer2004 has been tested on multiple hardware and software platforms and has proven to scale well from low-end desktop PCs to high-end servers and large clusters.

We discuss the way SPECjAppServer2004 evolved and the issues and challenges that had to be addressed in order to achieve the above mentioned goals. While doing this, we present several case studies which show how benchmarks like SPECjAppServer2004 can be exploited for studying the effect of different platform configuration settings and tuning parameters on the overall system performance. The case studies demonstrate that benchmarking not only helps to choose the best platform and validate its performance and scalability, but also helps to identify the configuration parameters most critical for performance. After its release in April 2004, SPECjAppServer2004 quickly gained in market adoption and it currently enjoys unprecedented popularity for a benchmark of this size and complexity. SPECjAppServer2004 became the de facto industry-standard workload for evaluating the performance and scalability of J2EE-based platforms and it is increasingly used throughout the industry.

Contributions in Performance Engineering While building on a scalable and optimized platform is a necessary condition for achieving good performance and scalability, unfortunately, it is not sufficient. The application, i.e. the DCS, built on the selected platform must also be designed to be efficient and scalable. The second major contribution of this thesis was the development of a *performance engineering framework* for DCS that provides a method to evaluate the performance and scalability of the latter during the different phases of their life cycle. This helps to identify design problems early in the development cycle and have them resolved in time. The framework is based on Queueing Petri Net (QPN) models and is made up of two parts. The first part provides a tool and methodology for analyzing QPN models by means of simulation. The second part provides a practical performance modeling methodology that shows how to model DCS using QPNs and use the models for performance evaluation. In the following, we take a closer look at these two parts.

Part 1 - Analysis of QPN Models by Means of Simulation Modeling realistic DCS using conventional models such as queueing networks and stochastic Petri nets poses many difficulties stemming from the limited model expressiveness, on the one hand, and the system size and complexity, on the other hand. We present some case studies which demonstrate this and show how QPNs can be exploited to address these difficulties and allow for accurate modeling of DCS. The QPN paradigm provides a number of benefits over conventional modeling paradigms. Most importantly, it allows the integration of hardware and software aspects of system behavior into the same model. The main problem with QPN models, however, is that currently available tools and techniques for QPN analysis suffer the state space explosion problem imposing a limit on the size of the models that are tractable. This is the reason why QPNs have hardly been exploited in the past decade and very few, if any, practical applications have been reported. A major contribution of this thesis is the development of a novel methodology for analyzing QPN models by means of discrete-event simulation. The methodology provides an alternative approach to analyze QPN models, circumventing the state space explosion problem. As an implementation of the methodology, a simulation tool for QPNs called SimQPN was developed. SimQPN is the world's first simulator specialized for QPNs. It has been tested extensively and has proven to run very fast and provide accurate and stable point and interval estimates of performance metrics. Using SimQPN, now for the first time, QPNs have been applied to model large and complex DCS in realistic capacity planning studies.

An alternative approach to simulate QPN models would be to use a general purpose simulation package. However, mapping a QPN model to a description in the terms of a general purpose simulation language is a complex, time-consuming and error-prone task. Moreover, not all simulation languages provide the expressiveness needed to describe complex QPN models. Another disadvantage is that general purpose simulators are normally not as fast and efficient as specialized simulators. Being specialized for QPNs, SimQPN simulates QPN models directly and has been designed to exploit the knowledge of the structure and behavior of QPNs to improve the efficiency of the simulation. Therefore, SimQPN provides much better performance than a general purpose simulator, both in terms of the speed of simulation and the quality of output data provided.

Part 2 - Performance Modeling Methodology Now that we have a scalable methodology for analyzing QPN models, we can exploit them as a performance prediction tool in the performance engineering process for DCS. However, building models that accurately capture the different aspects of system behavior is a very challenging task when applied to realistic systems. The second part of our performance engineering framework consists of a practical modeling methodology that shows how to model DCS using QPNs and use the models for performance evaluation. The methodology takes advantage of the modeling power and expressiveness of QPN models and provides the following important benefits over conventional modeling approaches:

1. QPN models allow the integration of hardware and software aspects of system behavior and lend themselves very well to modeling DCS.

2. In addition to hardware contention and scheduling strategies, using QPNs one can easily model software contention, simultaneous resource possession, synchronization, blocking and asynchronous processing. These aspects of system behavior, which are typical for modern DCS, are difficult to model accurately using current modeling approaches.

3. By restricting ourselves to QPN models, we can exploit the knowledge of their structure and behavior for fast and efficient simulation using SimQPN. This enables us to analyze models of large and complex DCS and ensures that our approach scales to realistic systems.

4. QPNs can be used to combine qualitative and quantitative system analysis. A number of efficient qualitative analysis techniques from Petri net theory are readily available and can be exploited.

5. Last but not least, QPN models have an intuitive graphical representation that facilitates model development.

The proposed performance engineering framework provides a very powerful tool for performance prediction that can be used throughout the phases of the software engineering lifecycle of DCS. We have validated our approach by applying it to study a number of different DCS ranging from simple systems to systems of realistic size and complexity such as the SPECjAppServer set of benchmarks. We present a case study in which a deployment of the industry-standard SPECjAppServer2004 benchmark is modeled and its performance is predicted under load. In addition to CPU and I/O contention, it is demonstrated how some more complex aspects of system behavior, such as thread contention and asynchronous processing, can be modeled. The model predictions demonstrate accuracy that is by far not attainable using conventional modeling techniques for DCS.

Acknowledgements

Many people have contributed to the work presented in this thesis. First of all, I would like to thank my advisor Prof. Alejandro Buchmann, Ph.D., for his invaluable support, advice and encouragement during the last four years. Without him this work would not have been possible. I would also like to thank Prof. Dr.-Ing. Dr. h.c. Günter Hommel for taking over the part of the second reviewer.

Special thanks go to my friend and colleague Dr.-Ing. Mariano Cilia for his unconditional help and advice during the different phases of my work. His friendship and support have been invaluable to me.

I am also especially thankful to SPEC's Long-time President Kaivalya Dixit (in memoriam) for his incredible support and the opportunity to be involved in a number of industry projects which have helped to give my research practical relevance. Special thanks to my friends and colleagues from SPEC's OSG Java Subcommittee, especially to John Stecher from IBM and Tom Daly from Sun Microsystems who were involved with me in the development of the SPECjAppServer2004 benchmark. Many thanks also to Chris Beer from HP; Ricardo Morin from Intel; Steve Realmuto and Russell Raymundo from BEA; Rafay Khawaja and Shan Appajodu from Borland; Matt Hogstrom from IBM; Akara Sucharitakul, Ning Sun and Shanti Subramanyam from Sun Microsystems; Evan Ireland from Sybase and Balu Sthanikam from Oracle. I really appreciate their cooperation.

I gratefully acknowledge the many fruitful discussions with Dr. Falko Bause from the University of Dortmund and his cooperation in providing me with the HiQPN-Tool. I also acknowledge the cooperation of Jim Rivera, Director of Technology at BEA, who arranged for us to obtain a research license for the WebLogic platform.

I want to thank all members of the Databases and Distributed Systems Group, my colleagues at the Information Technology Transfer Office and those at the Ph.D. Program (Graduiertenkolleg) "Enabling Technologies for Electronic Commerce"for creating such a pleasant and enjoyable working environment. I appreciate their help and support especially during my first year in Darmstadt. Thanks to Dr. Roger Kilian-Kehr, Dr. Ludger Fiege, Lars Brückner, Dr. Peer Hasselmeyer, Jan Steffan, Marco Voss, Christoph Liebig and Dr. Ming-Chuan Wu. I am also especially thankful to Volker Sauer, Dr. Matthias Meixner and Christian Haul who have been of immense help to me in resolving many technical problems. Many thanks also to Marion Braun for her kind support and assistance with numerous bureaucratic matters.

Furthermore, I would like to thank my diploma students Björn Weis, Kai Juse, Christofer Dutz, Bernhard Siedentop, Anja Steih and Andreas Fleischmann for their hard work on many interesting projects in the field of performance engineering. Special thanks goes to Björn Weis for collaborating with me on conducting the case study presented in Section 2.6 of Chapter 2.

In the end, I would like to extend my deepest gratitude to my family. Words fail to describe how much I owe them. My best thanks to my wife Hanna for her endless love and inspiration; to my parents Douhomir and Ekaterina for their immeasurable support and encouragement; to my brother George and my sister Stella for always being there for me; and last but not least to my parents-in-law Peter and Christel Böhringer who have been like real parents to me. Without these people I would not be who I am today and this thesis would never have been written...

Finally, thanks to you, the reader for taking the time to read this thesis. Without you our work would have been in vain.

Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Meaning
2PC	2-Phase-Commit
2PL	2-Phase-Locking
API	Application Programming Interface
AS	Application Server/s
BOM	Bill of Materials
BMP	Bean-Managed Persistence
CB	Component-Based
CGSPN	Colored Generalized Stochastic Petri Net
CMP	Container-Managed Persistence
CPN	Colored Petri Net
DBMS	Database Management System
DBS	Database Server
DCS	Distributed Component-based System/s
DES	Discrete Event Simulation
EJB	Enterprise Java Bean
FCFS	First-Come-First-Serve (scheduling strategy)
FIFO	First-In-First-Out
GC	Garbage Collection
GSPN	Generalized Stochastic Petri Net
HLQPN	High-Level Queueing Petri Net
HQPN	Hierarchical Queueing Petri Net
HTTP	Hypertext Transfer Protocol

Acronym	Meaning
IID	Independent and Identically Distributed (random variables)
IR	Injection Rate
IS	Infinite Server (scheduling strategy)
IT	Information Technology
J2EE	Java 2 Enterprise Edition Platform
J2SE	Java 2 Standard Edition Platform
JCP	Java Community Process
JDBC	Java Database Connectivity
JMOB	Java Middleware Open Benchmarking Project
JMS	Java Messaging Service
JRE	Java Runtime Environment
JSP	Java Server Page
JVM	Java Virtual Machine
LAN	Local Area Network
LCFS	Last-Come-First-Served (scheduling strategy)
LLQPN	Low-Level Queueing Petri Net
LO	Large Order
LQN	Layered Queueing Network
MDB	Message-Driven Bean
MOM	Message-Oriented Middleware
NOBM	Method of Non-Overlapping Batch Means
OLTP	Online Transaction Processing
OS	Operating System
PEPSY-QNS	Perf. Evaluation and Prediction SYstem for Queueing NetworkS
PN	(Ordinary) Petri Net
PO	Purchase Order
PS	Processor-Sharing (scheduling strategy)
P&S	Performance and Scalability
QN	Queueing Network
QoS	Quality of Service
QPN	Queueing Petri Net
RDBMS	Relational Database Management System

Acronym	Meaning
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RR	Round-Robin (scheduling strategy)
RT-UML	UML Profile for Schedulability, Performance and Time
RUBiS	Rice University Bidding System
SLAs	Service Level Agreements
SMP	Session Bean-Managed Persistence
SPEC	Standard Performance Evaluation Corporation
SPEC-OSG	SPEC's Open Systems Group
SPE	Software Performance Engineering
SPN	Stochastic Petri Net
SQL	Structured Query Language
SUT	System Under Test
TCO	Total Cost of Ownership
TPC	Transaction Processing Performance Council
TPM	Transaction Processing Monitor
UML	Unified Modeling Language
WAN	Wide Area Network
WLS	WebLogic Server/s

Chapter 1

Introduction

All solid facts were originally mist.
– *Henry S. Haskins*

A problem well stated is a problem half solved.
– *Charles F. Kettering*

1.1 Motivation

Distributed component-based systems (DCS) are becoming increasingly ubiquitous as enabling technology for modern enterprise applications. In the face of globalization and ever increasing competition, *Quality of Service (QoS)* requirements on such systems, like performance, availability and reliability are of crucial importance. There are numerous studies, for example in the areas of e-business, manufacturing, telecommunications, military, health care and transportation that have shown that, depending on the type of system considered, a failure to meet the QoS requirements can lead to serious financial losses, loss of customers and reputation, and in some cases even to loss of human lives. In order to survive, businesses must ensure that the systems they operate, not only provide all relevant services, but also meet the performance expectations of their users. To avoid the pitfalls of inadequate QoS, it is important to analyze the expected performance characteristics of systems during all phases of their life cycle. The methods used to do this are part of the discipline called *Performance Engineering* [145]. Performance engineering helps to estimate the level of performance a system can achieve and provides recommendations to realize the optimal performance level [112]. However, as systems grow in size and complexity, analyzing their performance becomes a more and more challenging task. System architects and deployers are often faced with the following questions:

- Which platform (hardware and software) would provide the best scalability and cost/performance ratio for a given application?[1]
- What performance would the application exhibit under the expected workload and how much hardware would be needed to meet the Service Level Agreements (SLAs)?

Answering the first question requires being able to *measure* the Performance and Scalability (P&S) of alternative hardware and software platforms. Answering the second question requires being able to *predict* the performance of a given application deployed on a selected platform. The motivation in writing this thesis originated from our observation that, in seeking answers to the above questions, system developers nowadays often rely on their intuition, marketing studies, expert opinions, past experiences, ad hoc procedures or general rules of thumb. As a result, the overall system capacity is unknown and capacity planning and procurement are done without a sufficient consideration given to the QoS requirements. In the pressure to release new applications as early as possible, more often than not, performance is considered as an afterthought. A widespread misconception is that performance problems can be addressed by simply "throwing enough hardware at the system". Therefore, pretty often performance issues are ignored until the final stage of system development (the "fix-it-later"approach [146]). At this late stage, a change of the platform and/or a major application redesign might be required in order to address discovered performance issues. In our opinion, this approach is expensive, time-consuming and professionally irresponsible.

The goal of this thesis is to provide a systematic approach for performance engineering of DCS that helps to identify performance problems early in the development cycle and ensure that systems are designed and sized to meet their QoS requirements. Before we discuss our approach we take a closer look at the problem we just described.

1.2 Problem Statement

Modern DCS are usually built on middleware platforms such as J2EE [158], Microsoft .NET [114] or CORBA [126]. Middleware platforms simplify application development by providing some common services typically used in enterprise applications. Application logic is normally partitioned into components distributed over physical tiers. Figure 1.1 shows a typical architecture of a multi-tiered distributed component-based application. There are three tiers: presentation tier, business logic tier and data tier. The presentation tier includes Web servers hosting Web components that implement the presentation logic of the application. The

[1] In this thesis, we use the terms "system"and "application"interchangeably.

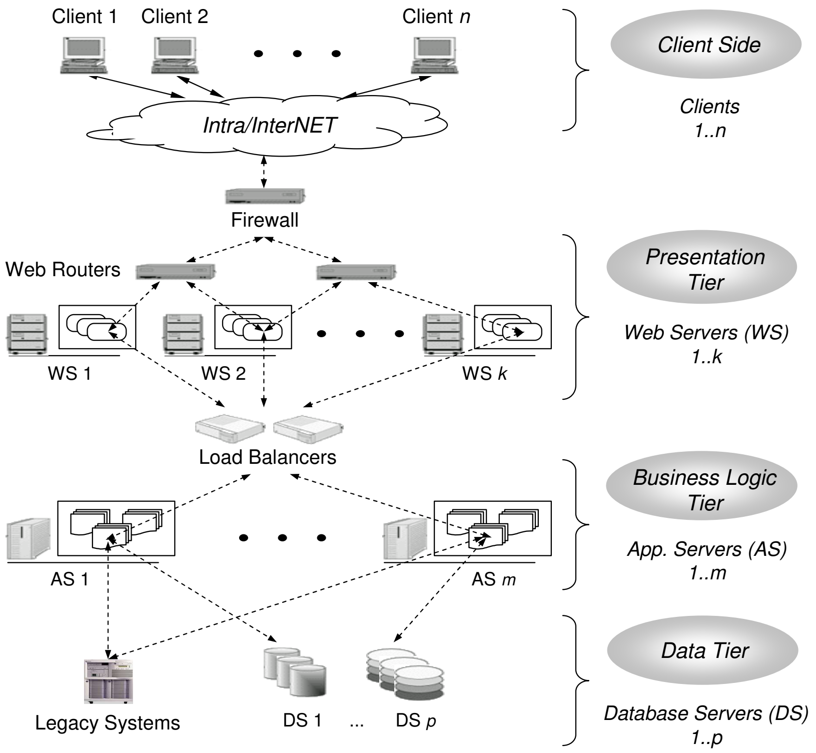


Figure 1.1: A multi-tiered distributed component-based application.

business logic tier includes a cluster of application servers hosting business logic components that implement the business logic of the application. The data tier includes database servers and legacy systems providing data management services. Web routers and load balancers are used to distribute incoming requests over the available Web servers and application servers, respectively. The inherent complexity of such architectures makes it extremely difficult to manage the end-to-end system P&S. System architects and deployers are often confronted with the following questions during the various stages of the system life cycle:

1. Which hardware platform (type of servers, disk subsystems, load balancers, etc.) would provide the best scalability and cost/performance ratio for the application?

2. Similarly, which software platform (OS, middleware, DBMS, etc.) would provide the best scalability and cost/performance ratio for the application?

3. Are the platforms selected scalable or do they have any inherent bottlenecks?

4. Which deployment settings and tuning parameters of the platforms selected (hardware and software) have the greatest effect on the overall system performance? How could the optimal values of these parameters be determined?

5. Is the current system design scalable? Which system components would be most utilized as the load increases and are they potential bottlenecks?

6. What maximum load would the system be able to handle in its current state without breaking the SLAs?

7. What performance would the system exhibit for a given workload and configuration scenario? What would be the average transaction throughput and response time? How utilized would be the various system components (Web servers, application servers, database servers, etc.)?

8. Which system components have the greatest effect on the overall system performance and how much would performance improve if they are optimized?

9. How much hardware would be needed to guarantee that SLAs are met? How many Web servers, application servers and database servers would be required?

1.3 Shortcomings of Current Approaches

Three different approaches have been employed in the industry when seeking answers to the above questions:

- Educated Guess
- Load Testing
- Performance Modeling

In the first approach, system architects and deployers make an educated guess or a simple back-of-the-envelope estimation based on their intuition, past experiences, expert opinions, ad hoc procedures or general rules of thumb [108]. This approach is very quick, easy and cheap, however, the information it provides is usually very rough and can not be relied upon. Therefore, following this approach is extremely risky and is not an option for mission-critical applications. We now take a closer look at the other two approaches and discuss their advantages and disadvantages.

1.3.1 Load Testing

In the second approach, load-testing tools are used to generate load on the system and measure its performance. Sophisticated load testing tools can emulate hundreds of thousands of "virtual users"that mimic real users interacting with the system. While tests are run, system components are monitored and performance metrics (e.g. response time, latency, utilization and throughput) are measured. Results obtained in this way can be used to identify and isolate system bottlenecks, fine-tune application components and measure the end-to-end system scalability. Unfortunately, this approach has several drawbacks. First of all, it is not applicable in the early stages of system development when the system is not available for testing. Second, it is extremely expensive and time-consuming since it requires setting up a production-like testing environment, configuring load testing tools and conducting the tests. Finally, testing results normally cannot be reused for other applications.

Another form of load testing which does not require that the system is available for testing is *benchmarking*. Benchmarking can be applied to test alternative platforms (hardware and software) on which the system can be built. This can be done before system development has started and can help in answering the first four questions discussed in Section 1.2. The way benchmarking works is by running artificial workloads (benchmarks) on the platforms considered and measuring their P&S. However, in order for benchmark results to be useful, the benchmarks employed must fulfill the following five important requirements:

1. They must be representative of real-world systems.
2. They must exercise and measure all critical services provided by platforms.
3. They must not be tuned/optimized for (i.e. be biased in favor of) specific products.
4. They must generate reproducible results.
5. They must not have any inherent scalability limitations.

In the context of platforms for DCS, a number of benchmarks have been developed in the past decade and have been used in the industry (see for example [67, 121, 122, 123, 124, 129, 130, 156, 168]). However, unfortunately most of these benchmarks fail to meet the above requirements because of one or more of the following reasons:

- Workloads used are designed to showcase the performance of particular products (proprietary benchmarks).

- Workloads used are not realistic enough. Important services provided by platforms are not tested at all, or they are tested, but are not stressed enough.
- Individual platform components are tested (e.g. CPUs, secondary storage systems, DBMS software) as opposed to end-to-end platforms (including all hardware and software needed to build a DCS).
- Workloads used have inherent scalability bottlenecks.
- Benchmark results are not reviewed by subject experts to make sure that benchmarks are run correctly and results are valid.

1.3.2 Performance Modeling

In the third approach, performance models are built and then used to predict the P&S of the system under study. Models represent the way system resources are used by the workload and capture the main factors determining the behavior of the system under load [111]. Queueing networks and stochastic Petri nets are perhaps the two most popular types of models that have been exploited in the industry. This approach is usually cheaper than load testing and has the advantage that it can be applied to evaluate alternative system designs at the early stages of system development before the system is available for testing. However, building models of realistic systems that accurately capture the different aspects of system behavior and analyzing the models is an extremely challenging task. Most modeling tools and techniques currently available impose some serious restrictions and when using them the modeler usually runs into one or more of the following problems:

1. The types of models supported do not provide the expressiveness needed to capture all important aspects of system behavior (e.g. queueing network models).
2. Models supported are expressive enough, however, constructing a representative model requires too much time and effort (e.g. simulation models).
3. Constructing a representative model can be done in reasonable amount of time, however, one of the following holds:
    - Once the model is constructed it turns out that it is not possible to analyze it using available analysis methods because the model is too large (referred to as *largeness problem*).
    - Model analysis is possible, but requires too much time and computing power.

- Model analysis can be performed with reasonable overhead, however, results provided are very approximate.

To illustrate the problems above let us consider the two most popular types of models used in practice - queueing networks and stochastic Petri nets. Unfortunately, they both have some serious limitations in terms of expressiveness and often fail to capture important aspects of system behavior. For example, while queueing networks provide a very powerful mechanism for modeling hardware contention and scheduling strategies, they are not that suitable for modeling software contention and synchronization aspects. Stochastic Petri nets, on the other hand, lend themselves very well to modeling software contention and synchronization aspects, but have difficulty in representing scheduling strategies and hardware contention.

Falko Bause has proposed a new modeling formalism called *Queueing Petri Nets (QPNs)* [11] that combines queueing networks and stochastic Petri nets into a single formalism and eliminates the above disadvantages. However, modeling a realistic DCS using QPNs, usually results in a model that is way too large to be analyzable using currently available tools and techniques. The problem is that available tools and solution techniques for QPN models are all based on Markov chain analysis, which suffers the well known *state space explosion problem* and limits the size of the models that can be analyzed (the largeness problem). An attempt to alleviate the problem was the introduction of *Hierarchically-Combined Queueing Petri Nets (HQPNs)* [13] which allow hierarchical model specification and exploit the hierarchical structure for efficient numerical analysis. This type of analysis, called *structured analysis*, allows models to be solved that are about an order of magnitude larger than those analyzable with conventional techniques. However, while this alleviates the problem it does not eliminate it since, as shown in [86], models of realistic DCS are still too large to be analyzable using this approach.

1.4 Approach and Contributions of this Thesis

In this thesis, we develop a systematic approach for performance engineering of DCS that helps to address the questions discussed in Section 1.2. The proposed approach is based on a combination of benchmarking and performance modeling. It builds on the fact that the overall P&S of a DCS is a function of two important variables:

1. The P&S of the hardware and software platforms used.

2. The P&S of the system design.

Therefore, if a DCS is to provide good P&S, both the platforms used and the system design must be efficient and scalable. To this end, we suggest that in the beginning of the system life cycle, the P&S of the platforms chosen are validated using *standard benchmarks* and that *performance models* are then exploited to evaluate the system P&S throughout the development cycle. In contrast to the ”fix-it-later”approach, this approach helps to identify P&S problems early in the system life cycle and eliminate them with minimal overhead.

The performance engineering of DCS is a multidimensional problem. Most current approaches concentrate on a single dimension and fudge the others. The approach we propose addresses multiple dimensions, simultaneously. The contributions of the thesis are now discussed in detail.

1.4.1 Contributions in the Area of Benchmarking

Building on a scalable and optimized platform is a precondition for achieving high P&S. Therefore, we suggest that before an application is built, the alternative platforms that can be used are compared and the one that provides the best cost/performance ratio is chosen. This is especially important for platform components that could have an effect on the system design, such as the type of middleware used. Even if the platform (or parts of it) are predefined and there are no alternatives to consider, the P&S of the platform employed should be validated before starting application development. Otherwise, one might later have to migrate the application to another platform if the selected one turns out not to meet the requirements for P&S.

Thus, standard benchmarks are needed that would provide a *reliable* method to evaluate the end-to-end P&S of hardware and software platforms for DCS. In order to be useful, these benchmarks must meet the requirements discussed in Section 1.3.1, i.e. they must be representative of real-world systems, must exercise and measure all critical services provided by platforms, must not be tuned/optimized for specific products, must generate reproducible results and must not have any inherent scalability limitations. In the past five years several attempts were made to develop a benchmark satisfying these requirements and they demonstrated that the latter is an extremely challenging task requiring *far more* than good programming skills. Maybe the most challenging part is to ensure that the benchmark provides a level playing field for performance comparisons and is designed in such a way that it does not have any inherent scalability limitations, i.e. it should scale well from low-end desktop PCs to high-end servers and large clusters. The first attempt to define a benchmark for J2EE-based[2] DCS platforms was initiated in 2000 and resulted in the development of the ECperf benchmark. ECperf was developed under the Java Community Process (JCP) with participation of all major J2EE application server vendors including IBM, BEA Systems, Sun Microsystems, Oracle, Hewlett Packard,

[2]The Java 2 Enterprise Edition Platform (J2EE) defines an industry-standard for implementing middleware platforms for DCS. J2EE-based platforms are currently technology of choice for building DCS.

Sybase, iPlanet, Borland, Macromedia, Pramati and IONA. A huge amount of time, money and effort was invested in the specification and development of ECperf, however, while it met most of the requirements discussed above, unfortunately, it failed to meet all of them.

In Chapter 2, we present ECperf and conduct a detailed analysis of its design. We discovered several subtle design issues limiting the overall scalability of the benchmark and degrading its performance and reliability. The main problem was in the way long transactions were processed and persistent data was managed. Long transactions were processed synchronously and spanned external network communication which was leading to high data contention in the database. This resulted in a scalability bottleneck which manifested itself clearly when using a database with pessimistic concurrency control and was not that obvious when using databases with optimistic schedulers. Moreover, independent of the type of database used, the identified problem in the design of the benchmark was causing some performance and reliability issues. In Chapter 2, after discussing the problem in detail, we propose a redesign of the benchmark which addresses the discovered issues. In the new design, long transactions are chopped up into shorter ones reducing data contention in the database. External communication is performed *asynchronously* outside of the context of database-intensive transactions. This eliminates the scalability bottleneck and addresses the identified performance and reliability issues.

The proposed redesign of the benchmark was documented and submitted as an official proposal [80] to the ECperf expert group at Sun Microsystems and later to the OSG-Java subcommittee of SPEC[3], which was responsible for the future of ECperf[4]. The proposal was discussed and approved. The new design was implemented in a future version of the benchmark in whose development and specification the author was involved as release manager and lead developer. The new benchmark was called SPECjAppServer2004 and was developed within SPEC's OSG-Java subcommittee, which includes BEA, Borland, Darmstadt University of Technology, Hewlett-Packard, IBM, Intel, Oracle, Pramati, Sun Microsystems and Sybase. It is important to note that even though SPECjAppServer2004 is partially based on ECperf, it implements a *new* enhanced workload that exercises all major services of the J2EE platform in a complete end-to-end application scenario. Thus, SPECjAppServer2004 is substantially more complex than ECperf. After its release in April 2004, SPECjAppServer2004 quickly gained in market adoption and it currently

[3]The Standard Performance Evaluation Corporation (SPEC) is a non-profit corporation whose mission is to "establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers" (quoted from SPEC's bylaws). With currently more than 70 institutional members from across the industry and academia, SPEC has grown to become one of the world's largest and most successful performance standardization bodies.

[4]In September 2002, the ECperf benchmark was taken over by SPEC and renamed to SPECjAppServer. SPEC's OSG-Java subcommittee became responsible for the benchmark.

enjoys unprecedented popularity for a benchmark of this size and complexity. SPECjAppServer2004 became the de facto industry-standard workload for evaluating the P&S of J2EE-based platforms and it is increasingly used throughout the industry. This success is due to the following important benefits that the benchmark provides:

1. The application and workload modeled by SPECjAppServer2004 have been designed to be representative of real-world systems.

2. SPECjAppServer2004 exercises all major services of J2EE platforms and measures the end-to-end platform P&S.

3. SPECjAppServer2004 has been tested on multiple hardware and software platforms and has proven to scale well from low-end desktop PCs to high-end servers and large clusters.

4. Being an industry-standard, SPECjAppServer2004 has not been tuned or optimized for any specific platform and provides a level playing field for performance comparisons.

5. Once a platform has been tested and validated, the results can be used for multiple projects/applications. Official benchmark results are made publicly available and can be used free of charge.

6. Last but not least, prior to publication, official benchmark results are reviewed by subject experts making sure that the benchmark was run correctly and the results are valid.

Doing benchmarking before beginning system development ensures that P&S problems in the platforms used are discovered and addressed in time. Applications can then be developed with confidence that there are no bottlenecks and/or inefficiencies in the platforms employed. However, while the main purpose of benchmarks like SPECjAppServer2004 and its predecessors is to measure the P&S of platforms, they could equally well be used for studying the effect of different platform configuration settings and tuning parameters on the overall system performance. Thus, benchmarking not only helps to choose the best platform and validate its P&S, but also helps to identify the configuration parameters most critical for performance. In Chapter 2, we present some case studies that demonstrate this.

The results from the contributions discussed above were published in [84], [83], [82] and [88].

1.4.2 Contributions in Performance Engineering

While building on a scalable and optimized platform is a necessary condition for achieving high P&S, unfortunately, it is not sufficient. As already noted, the application, i.e. the DCS, built on the selected platform must also be designed to be efficient and scalable. The second major contribution of this thesis is the development of a performance engineering framework for DCS that provides a method to evaluate the P&S of the latter during the different phases of their life cycle. This helps to identify problems early in the development cycle and have them resolved in time. The framework is based on QPN models and is made up of two parts. The first part provides a tool and methodology for analyzing QPN models by means of simulation, circumventing the state-space explosion problem. This allows QPN models of realistic DCS to be analyzed. The second part of the framework provides a practical performance modeling methodology that shows how to model DCS using QPNs and use the models for performance evaluation. We now take a closer look at these two parts.

Analysis of QPN Models by Means of Simulation

Modeling realistic DCS using conventional models such as queueing networks and stochastic Petri nets poses many difficulties stemming from the limited model expressiveness, on the one hand, and the system size and complexity, on the other hand. In Chapter 4, we present some case studies that demonstrate this and show how QPNs can be exploited to address these difficulties and allow for accurate modeling of DCS. The QPN paradigm provides a number of benefits over conventional modeling paradigms. Most importantly, it allows the integration of hardware and software aspects of system behavior into the same model. The main problem with QPN models, however, is that, as discussed in Section 1.3.2, currently available tools and techniques for QPN analysis suffer the state space explosion problem imposing a limit on the size of the models that are tractable. The case studies considered in Chapter 4 show that QPN models of realistic systems are too large to be analyzable using currently available analysis techniques. This is the reason why QPNs have hardly been exploited in the past decade and very few, if any, practical applications have been reported. A major contribution of this thesis, presented in Chapter 5, is the development of a novel methodology for analyzing QPN models by means of discrete-event simulation. The methodology provides an alternative approach to analyze QPN models, circumventing the state space explosion problem. As an implementation of the methodology, a simulation tool for QPNs called SimQPN was developed. SimQPN is to the best of our knowledge the first simulator specialized for QPNs. It has been tested extensively and has proven to run very fast and provide accurate and stable point and interval estimates of performance metrics. Using SimQPN, now for the first time, QPNs have been applied to model large and complex DCS in realistic capacity planning studies. Some of these studies are presented in Chapters 5 and 6.

An alternative approach to simulate QPN models would be to use a general purpose simulation package. However, this approach has some disadvantages. First, general purpose simulation packages do not provide means to represent QPN constructs directly. Mapping a QPN model to a description in the terms of a general purpose simulation language is a complex, time-consuming and error-prone task. Moreover, not all simulation languages provide the expressiveness needed to describe complex QPN models. Another disadvantage is that general purpose simulators are normally not as fast and efficient as specialized simulators, since they are usually not optimized for any particular type of models. Being specialized for QPNs, SimQPN simulates QPN models directly and has been designed to exploit the knowledge of the structure and behavior of QPNs to improve the efficiency of the simulation. Therefore, SimQPN provides much better performance than a general purpose simulator, both in terms of the speed of simulation and the quality of output data provided. Last but not least, SimQPN has the advantage that it is extremely light-weight and being implemented in Java it is platform independent.

The case studies presented in Chapter 4 were published in [85] and [86]. The first paper shows how conventional queueing network models can be used to model DCS demonstrating the difficulties stemming from the limited model expressiveness. The second paper demonstrates how QPN models can be used to address these difficulties and motivates the need for scalable QPN analysis techniques. The proposed methodology for analyzing QPN models by means of simulation has been accepted for publication in [87].

Performance Modeling Methodology

Now that we have a scalable tool for analyzing QPN models, we can exploit them as a performance prediction mechanism in the performance engineering process for DCS. However, building models that accurately capture the different aspects of system behavior is a very challenging task when applied to realistic systems. The second part of our performance engineering framework, presented in Chapter 6, consists of a practical modeling methodology that shows how to model DCS using QPNs and use the models for performance evaluation. The methodology takes advantage of the modeling power and expressiveness of QPN models and provides the following important benefits over conventional modeling approaches:

1. QPN models allow the integration of hardware and software aspects of system behavior and lend themselves very well to modeling DCS.

2. In addition to hardware contention and scheduling strategies, using QPNs one can easily model software contention, simultaneous resource possession, synchronization, blocking and asynchronous processing. These aspects of system behavior, which are typical for modern DCS, are difficult to model accurately using current modeling approaches.

3. By restricting ourselves to QPN models, we can exploit the knowledge of their structure and behavior for fast and efficient simulation using SimQPN. This enables us to analyze models of large and complex DCS and ensures that our approach scales to realistic systems.

4. QPNs can be used to combine qualitative and quantitative system analysis. A number of efficient qualitative analysis techniques from Petri net theory are readily available and can be exploited.

5. Last but not least, QPN models have an intuitive graphical representation that facilitates model development.

The modeling methodology has been submitted for publication at [81] and is currently under review. The proposed performance engineering framework provides a very powerful tool for performance prediction that can be used throughout the phases of the software engineering lifecycle of DCS. We have validated our approach by applying it to study a number of different DCS ranging from simple systems to systems of realistic size and complexity such as the SPECjAppServer set of benchmarks. In Chapter 6, a case study is presented in which a deployment of the industry-standard SPECjAppServer2004 benchmark is modeled and its performance is predicted under load. In addition to CPU and I/O contention, it is demonstrated how some more complex aspects of system behavior, such as thread contention and asynchronous processing, can be modeled. The model predictions demonstrate accuracy that is not attainable using conventional modeling techniques for DCS.

1.5 Thesis Organization

The thesis is organized as follows. In **Chapter 2**, we introduce several popular benchmarks for measuring the P&S of J2EE-based hardware and software platforms. We start with the ECperf benchmark and after introducing it we present a case study which uses the benchmark to evaluate the performance of the major persistence methods in J2EE. We discuss a scalability problem we identified in the design of the benchmark and propose a solution which exploits asynchronous processing to reduce data contention in the database. The case study demonstrates that building on a scalable platform is not sufficient to ensure that a DCS is scalable. After ECperf, we briefly discuss its successor benchmarks SPECjAppServer2001 and SPECjAppServer2002, and then introduce the new SPECjAppServer2004 benchmark. SPECjAppServer2004 is the state-of-the-art industry-standard benchmark for measuring the P&S of J2EE-based platforms. At the end of the chapter, we present a case study with a deployment of SPECjAppServer2004 on the JBoss application server. The case study demonstrates how benchmarks like SPECjAppServer2004 can be used to identify the platform configuration parameters that are most critical for the overall system P&S.

Chapter 3 introduces the performance models that are used in the rest of the thesis. It starts with a brief overview of the conventional queueing network and Petri net models and then introduces the queueing Petri net modeling formalism which is used as a basis for the performance engineering framework developed in this thesis.

In **Chapter 4**, we present two practical performance modeling case studies which illustrate the difficulties that arise when trying to model a realistic DCS and predict its performance. In the first case study, a queueing network model of a SPECjAppServer2002 deployment is built. The difficulties due to the limited expressiveness of queueing network models and the limitations in the available analysis techniques are discussed. In the second case study, a Queueing Petri Net (QPN) model of a deployment of SPECjAppServer2001's order entry application is built. The study demonstrates the modeling power and expressiveness of QPN models and shows how they can be used to integrate hardware and software aspects of system behavior. However, the study also shows that QPN models of realistic systems are too large to be analyzable using currently available tools and techniques for QPN analysis.

In **Chapter 5**, we present SimQPN - our tool and methodology for analyzing QPN models by means of simulation. SimQPN is one of the two major components of the performance engineering framework developed in this thesis. It provides an alternative approach to analyze QPN models, circumventing the state space explosion problem and allowing QPN models of realistic DCS to be analyzed. We validate SimQPN by applying it to study several different QPN models ranging from simple models to models of realistic size and complexity.

Chapter 6 presents the other major component of the performance engineering framework we propose - a practical performance modeling methodology for DCS. The methodology helps to construct models of DCS that accurately reflect both their functional and performance characteristics. It exploits QPN models to take advantage of their modeling power and expressiveness. After discussing the modeling methodology in general, we present a case study in which it is used to model a realistic system and analyze its P&S. The system modeled is a deployment of the SPECjAppServer2004 benchmark. A detailed model of the system and its workload is built in a step-by-step fashion. The model is validated and used to predict the system performance for several deployment configurations and workload scenarios of interest.

Chapter 7 reviews some related work in the area of benchmarking and performance engineering of DCS.

Finally, **Chapter 8** summarizes and concludes the thesis.

Chapter 2

Benchmarking Distributed Component Platforms

The reputation of current 'benchmarketing' claims regarding system performance is on par with the promises made by politicians during elections.

Kaivalya M. Dixit, Long-time SPEC President
The Benchmark Handbook, 1993

2.1 Introduction

The overall Performance and Scalability (P&S) of a DCS is a function of the P&S of the hardware and software platforms on which the system is built and the P&S of the system design. Thus, building on scalable and optimized platforms is a precondition for achieving high P&S. Therefore, we suggest that before a system is built, the hardware and software platforms chosen are tested by means of benchmarks to measure and validate their P&S. The system can then be developed with confidence that there are no bottlenecks and/or inefficiencies in the platforms employed. Benchmarking platforms prior to starting system development not only helps to discover and address P&S problems in time, but also helps to identify the platform configuration parameters that are most critical for the system P&S.

In this chapter, we present several popular benchmarks for measuring the P&S of J2EE-based hardware and software platforms. In addition to this, we present several case studies in which the benchmarks were used to evaluate alternative application designs and study the effect of some platform configuration options and tuning parameters on the overall system performance. The reason we consider J2EE-based platforms is that they are currently the technology of choice for building DCS.

We start with the ECperf benchmark developed in 2000/2001 under the Java Community Process (JCP). After discussing ECperf's business model and application design, in Section 2.4 we present a case study in which the benchmark is used as an example of a realistic application in order to evaluate the performance of the major persistence methods in J2EE. We discuss several subtle issues we discovered in the design of ECperf, limiting the overall scalability of the benchmark and degrading its performance and reliability. After discussing these issues in detail, we propose a redesign of the benchmark in which asynchronous, message-based processing is exploited to reduce data contention in the database and improve the benchmark scalability. This eliminates the scalability bottleneck and addresses the identified performance and reliability issues. The case study demonstrates that building on a scalable platform is not sufficient to ensure that a DCS is scalable, i.e. the system design must also be scalable. After ECperf, we briefly discuss its successor benchmarks SPECjAppServer2001 and SPECjAppServer2002, which are identical in terms of workload and only differ in the version of J2EE they are implemented for.

In the second part of the chapter, starting with Section 2.5, we present SPECjAppServer2004 - the successor benchmark of SPECjAppServer2002, SPECjAppServer2001 and ECperf. It implements a *new* enhanced workload that exercises all major services of J2EE in a complete end-to-end application scenario. SPECjAppServer2004 is the state-of-the-art industry-standard benchmark for evaluating the P&S of J2EE-based platforms. After introducing SPECjAppServer2004, in Section 2.6 we present a case study with a deployment of the benchmark on the JBoss application server[1]. The benchmark is used to study how the performance of J2EE applications running on JBoss can be improved by exploiting different deployment options and tuning parameters offered by the platform. The case study demonstrates how benchmarks like SPECjAppServer2004 can be used to identify the platform configuration parameters that are most critical for the overall system P&S. Even though the results are specific to JBoss, most of the conclusions are generalized to other J2EE application servers.

2.2 The J2EE Platform

Over the past five years, the Java 2 Enterprise Edition Platform (J2EE) has established itself as the technology of choice for developing modern DCS. This success is largely due to the fact that J2EE is not a proprietary product, but rather an industry standard, developed as the result of a large industry initiative led by Sun Microsystems, Inc. The goal of this initiative was to establish a standard middle-

[1] The JBoss application server is the world's most popular open-source J2EE application server.

ware platform for developing enterprise-class DCS in Java. Over 30 software vendors have participated in this effort and have come up with their own implementations of J2EE, the latter being commonly referred to as *J2EE application servers*.

In essence, the aim of J2EE is to enable developers to quickly and easily build scalable, reliable and secure applications without having to develop their own complex middleware services. Here we are talking about services such as transaction management, caching, resource-pooling, clustering, transparent failover, load-balancing and back-end integration to name just a few. Many of these services are also provided by traditional Transaction Processing Monitors (TPMs), but in a rather monolithic and highly proprietary manner. The J2EE platform allows developers to leverage middleware services provided by the industry without having to code using proprietary middleware APIs.

The fact that J2EE application servers are all based on a common standard and that this standard is itself based on Java, ensures that J2EE applications are not only operating-system independent, but also portable across a wide range of middleware platforms - J2EE implementations. This gives J2EE system developers a wide selection of platforms on which they can develop and deploy their applications. The freedom of choice encourages best-of-breed products to compete and establish themselves in the market. However, since product functionality is standardized, the focus is placed on the performance, scalability and the Total Cost of Ownership (TCO) of the platforms. It is exactly here that application server vendors strive to distinguish their products in the market and gain competitive advantage. This raises the following questions that system developers are often confronted with:

- Which J2EE application server would provide the best scalability and cost/performance ratio for a given application?
- Which hardware platform and operating system should be used as deployment platform for the chosen application server?

Obviously, the natural approach to answer these questions is to use benchmarks to measure the P&S of alternative platforms. However, we have all been able to witness how in the recent years different vendors have been running proprietary benchmarks and coming up with contradictory and biased results in their attempts to push their products and beat the competition [3, 120, 122, 124, 163]. It is clear that such results and claims cannot be trusted. What is needed are publicly available industry-standard benchmarks built and maintained by the industry itself with no predominance of any particular vendor. In order to provide useful results these benchmarks must fulfill the following requirements:

1. They must be representative of real-world systems.
2. They must exercise and measure all critical services provided by platforms.

3. They must not be tuned/optimized for specific products.
4. They must generate reproducible results.
5. They must not have any inherent scalability limitations.

The first attempt to come up with a benchmark for J2EE-based platforms that meets these requirements was initiated in 2000 and resulted in the development of the ECperf benchmark.

2.3 The ECperf Benchmark

ECperf is a popular benchmark for measuring the P&S of J2EE hardware and software platforms. It was built by Sun Microsystems in conjunction with J2EE server vendors under the JCP. ECperf is composed of a specification and a toolkit [155]. The specification describes the benchmark as a whole, the modeled workload, the running and scaling rules, and the operation and reporting requirements. The toolkit provides the necessary code to run the benchmark and measure performance.

2.3.1 ECperf Business Model

The ECperf workload is based on a distributed application claimed to be large enough and complex enough to represent a real-world e-business system [155, 161]. The benchmark designers have chosen manufacturing, supply chain management, and order/inventory as the "storyline" of the business problem to be modeled. This is an industrial-strength distributed problem, that is heavyweight, mission-critical and requires the use of a powerful and scalable infrastructure. Most importantly, it requires the use of interesting middleware services, including distributed transactions, clustering, load-balancing, fault-tolerance, caching, object persistence and resource pooling among others. It is these services of application servers that are exercised and measured by the ECperf benchmark.

ECperf models businesses using four domains [155]:

1. Customer domain dealing with customer orders and interactions.
2. Manufacturing domain performing just-in-time manufacturing operations
3. Supplier domain handling interactions with external suppliers.
4. Corporate domain managing all customer, product and supplier information.

Figure 2.1 illustrates these domains and gives some examples of typical transactions run in each of them.

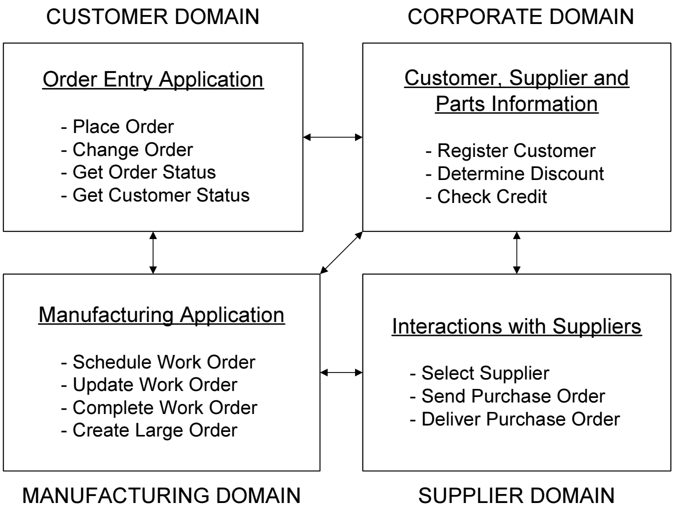


Figure 2.1: The ECperf business domains.

The customer domain models customer interactions using an order entry application, providing some typical online ordering functionality, such as placing new orders (NewOrder transaction), changing existing orders (ChangeOrder transaction) and retrieving the status of a given order (OrderStatus transaction) or all orders of a given customer (CustStatus transaction). Orders can be placed by individual customers as well as by distributors. Orders placed by distributors are called *large orders*.

The manufacturing domain models the activity of production lines in a manufacturing plant. Products manufactured by the plant are called *widgets*. Manufactured widgets are also called *assemblies*, since they are comprised of *components*. The Bill Of Materials (BOM) for an assembly indicates the components needed for producing it. Both assemblies and components are commonly referred to as *parts*. There are two types of production lines, *planned lines* and *large order lines*. Planned lines run on schedule and produce a predefined number of widgets. Large order lines run only when a large order (LO) is received in the customer domain. The unit of work in the manufacturing domain is a *work order*. Each work order is for a specific quantity of a particular type of widget. When a work order is created, the BOM for the corresponding type of widget is retrieved and the required parts are taken out of inventory. As the widgets move through the assembly line, the work order status is updated to reflect progress. Once the work order is complete, it is marked as completed and inventory is updated. When the inventory of parts gets depleted, suppliers need to be located and purchase orders need to be sent out. This is done by contacting the supplier domain. The supplier domain is responsible for interactions with suppliers. A supplier is chosen based on the parts that need to be ordered, the time in which they are required and the prices quoted. A purchase order (PO) is sent to the selected supplier. When the supplier delivers the parts, the supplier domain sends a message to the manufacturing domain to update inventory.

Finally, the corporate domain is a master keeper of all customer, supplier and product information for the whole enterprise. Credit information, including credit limits, about all customers is kept solely in a database in the corporate domain to provide maximal security and privacy. When a new order is placed in the customer domain, the corporate domain is contacted to determine the total amount to be paid taking customer discount policies into account. Before the order is confirmed a credit worthiness check is made.

2.3.2 ECperf Application Design

All the activities and processes in the four domains described above are implemented using Enterprise Java Bean (EJB) components [157] assembled into a single J2EE application which is deployed on the *System Under Test (SUT)*. The only exception is for the interactions with suppliers which are implemented using a separate Java servlet application called *supplier emulator*. The latter is deployed in a Java-enabled Web server on a dedicated machine. The supplier emulator provides the supplier domain with a way to emulate the process of sending and receiving purchase orders to/from suppliers. The supplier emulator accepts a purchase order from the BuyerSes component in the supplier domain, processes the purchase order and then delivers the items ordered to the ReceiverSes component in the supplier domain. This interaction is depicted in Figure 2.2.

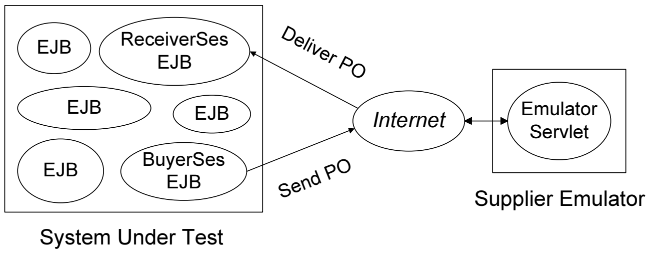


Figure 2.2: Interaction with the supplier emulator.

The workload generator is implemented using a multithreaded Java application called *ECperf driver*. The latter is designed to run on multiple client machines, using an arbitrary number of JVMs to ensure that it has no inherent scalability limitations. A relational DBMS is used for data persistence and all data access operations use entity beans which are mapped to tables in the ECperf database.

The throughput of the benchmark is driven by the activity of the order entry application in the customer domain and the manufacturing application in the manufacturing domain. The throughput of both applications is directly related to the chosen *Injection Rate (IR)*, which determines the number of order entry transactions started and the number of work orders scheduled per second. The total number of order entry transactions (NewOrder, ChangeOrder, OrderStatus and CustStatus) injected per second is equal to the IR. The total number of work orders scheduled per second equals $0.65 * IR$ (of these, $0.05 * IR$ are triggered by large orders placed in the customer domain).

Business transactions are selected by the driver based on the mix shown in Table 2.1. Since the benchmark is intended to test the transaction handling capabilities of EJB containers, the mix is update intensive. The actual mix achieved in the benchmark must be within 5% of the targeted mix for each type of transaction. For example, the NewOrder transactions can vary between 47.5% to 52.5% of the total mix. The driver checks and reports on whether the mix requirement was met.

Table 2.1: ECperf business transaction mix requirements.

Business Transaction	Percent Mix
NewOrder	50%
ChangeOrder	20%
OrderStatus	20%
CustStatus	10%

The summarized performance metric provided after running the benchmark is called *BBops/min* and it denotes the average number of successful Benchmark Business OPerationS per minute completed during the measurement interval. BBops/min is composed of the total number of business transactions completed in the customer domain, added to the total number of work orders completed in the manufacturing domain, normalized per minute. The benchmark can be run in two modes. In the first mode only the order entry application is run, while in the second mode both the order entry and the manufacturing applications are run.

2.3.3 From ECperf to SPECjAppServer2001/2002

In September 2002, the ECperf benchmark was taken over by SPEC and renamed to SPECjAppServer2001[2]. SPECjAppServer2001 became the official industry-standard benchmark for measuring the P&S of J2EE 1.2 hardware and software platforms. In November 2002, SPECjAppServer2001 was ported to J2EE 1.3/EJB 2.0 and a new version of the benchmark was released under the name SPECjAppServer2002. Note that, in terms of workload and application design, both SPECjAppServer2001 and SPECjAppServer2002 were essentially identical to ECperf 1.1 and the only difference between them was in the versions of J2EE/EJB used.

2.4 Evaluating J2EE Persistence Methods

The J2EE platform provides a variety of options for making business data persistent using DBMS technology. However, the integration with existing backend database systems has proven to be of crucial importance for the P&S of J2EE applications because the latter are usually very data-intensive. As a result, the data access layer, and the link between the application server and the database server in particular, are very susceptible to turning into a system bottleneck. In this section, we use the ECperf benchmark as an example of a realistic application in order to evaluate the major persistence methods in J2EE and demonstrate how the data access layer could easily become a bottleneck preventing the system to scale. We discuss several subtle issues we discovered in the design of ECperf limiting the overall scalability of the benchmark and degrading its performance and reliability. The main problem is in the way long transactions are processed and persistent data is managed. Long transactions are processed synchronously and span external network communication which leads to high data contention in the database. This results in a scalability bottleneck which manifests itself clearly when using a database with pessimistic concurrency control. After discussing the problem in detail, we propose a redesign of the benchmark which addresses the discovered issues. In the new design, long transactions are chopped up into shorter ones reducing data contention in the database. External communication is performed *asynchronously* outside of the context of database-intensive transactions. This eliminates the scalability bottleneck and addresses the identified performance and reliability issues.

2.4.1 Persistence Methods in J2EE

Entity beans are the natural method provided in J2EE for modeling persistent business data. They provide an object-oriented model in which business data is

[2]SPECjAppServer is a trademark of the Standard Performance Evaluation Corp. (SPEC). Although released in 2002, the benchmark was named SPECjAppServer2001, because it used the workload of the ECperf 1.1 benchmark, which was released in 2001.

represented as Java objects. The actual data being modeled is stored in attributes of the objects. However, in order to make the entity bean data persistent, the container needs an underlying persistence mechanism. Examples of persistence mechanisms that can be used are relational databases, object databases or file systems. We will be assuming that a relational DBMS is used as persistence mechanism since this is the most typical case. Before the entity bean data can be made persistent, it must be mapped to some data structures in the underlying storage - the database. Data access code (typically SQL) must be provided for storing data in the database and retrieving it. The EJB specification [157] offers two alternatives for defining the data access code of entity beans. In the first case, code is written by the component developer and the bean is said to use *Bean-Managed Persistence (BMP).* In the second case, code is automatically generated by the container and the bean is said to use *Container-Managed Persistence (CMP).* As discussed in [140], both BMP and CMP have their virtues and drawbacks.

There are some fundamental benefits that entity beans bring to the table. First, they allow a clear separation between the business logic and the persistence logic of the application. When using entity beans the developer does not need to know about the underlying persistence mechanism. The data access logic is decoupled from the application logic and application code is much easier to understand and maintain. Second, entity beans allow the container to cache and reuse data in the middle tier and in this way reduce the load on the database. Finally, entity beans can enforce control on the way data is accessed and modified. For example, when updating an attribute on an entity bean, the entity bean may need to perform validation logic on its changes and possibly institute updates on other entity beans in the application. However, even though entity beans bring all these advantages, there are situations when it is not worth to go through the entity bean layer. In particular, when reading large amounts of read-only data for listing purposes it is recommended to consider bypassing entity beans and read data directly through JDBC in session beans. This is sometimes termed *Session Bean-Managed Persistence (SMP)* [101] and in the above situations may lead to significant performance gains. To summarize, the J2EE platform provides three major approaches for managing persistent business data: BMP, CMP and SMP. We will concentrate on BMP and CMP since they are more typical for modern J2EE applications.

2.4.2 Performance Comparison of BMP and CMP

We now use the ECperf benchmark to compare BMP and CMP in terms of their performance and discuss the issues that drive the choice between them. In addition, we provide some guidelines for improving BMP performance.

ECperf offers both BMP and CMP versions of all entity beans used. We conducted experiments, first with BMP and then with CMP, in the deployment envi-

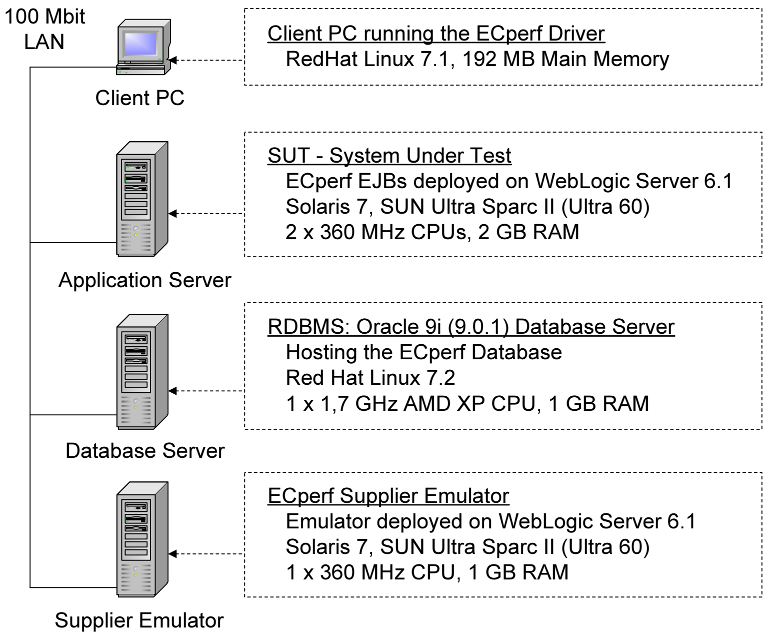

100 Mbit LAN
Client PC
Client PC running the ECperf Driver
RedHat Linux 7.1, 192 MB Main Memory
SUT - System Under Test
ECperf EJBs deployed on WebLogic Server 6.1
Solaris 7, SUN Ultra Sparc II (Ultra 60)
2 x 360 MHz CPUs, 2 GB RAM
Application Server
RDBMS: Oracle 9i (9.0.1) Database Server
Hosting the ECperf Database
Red Hat Linux 7.2
1 x 1,7 GHz AMD XP CPU, 1 GB RAM
Database Server
ECperf Supplier Emulator
Emulator deployed on WebLogic Server 6.1
Solaris 7, SUN Ultra Sparc II (Ultra 60)
1 x 360 MHz CPU, 1 GB RAM
Supplier Emulator

Figure 2.3: ECperf deployment environment.

ronment depicted in Figure 2.3. We were quite surprised that ECperf performed much worse with BMP than with CMP. Monitoring the database server, we noticed that in the BMP version of ECperf entity bean data was updated in the database at every transaction commit even when no changes had been made, i.e. at transaction commit all entity beans accessed by the transaction had their data updated in the database even if they had been accessed in read-only mode. We changed the BMP code to check if data had been modified and only in this case update the database [154]. As a result, throughput soared by a factor of two, but performance was still worse than with CMP.

Figure 2.4 shows the ECperf results that we obtained with our optimized BMP code compared to the results that we obtained with CMP. In these experiments we ran the order entry application under different transaction IRs. The first graph compares average throughput (order entry transactions per minute) relative to the average throughput achieved when running at IR of 10. The second graph reports the average transaction commit time (in ms) of order entry transactions. Results show that CMP performs substantially better as we raise the IR beyond 30. As

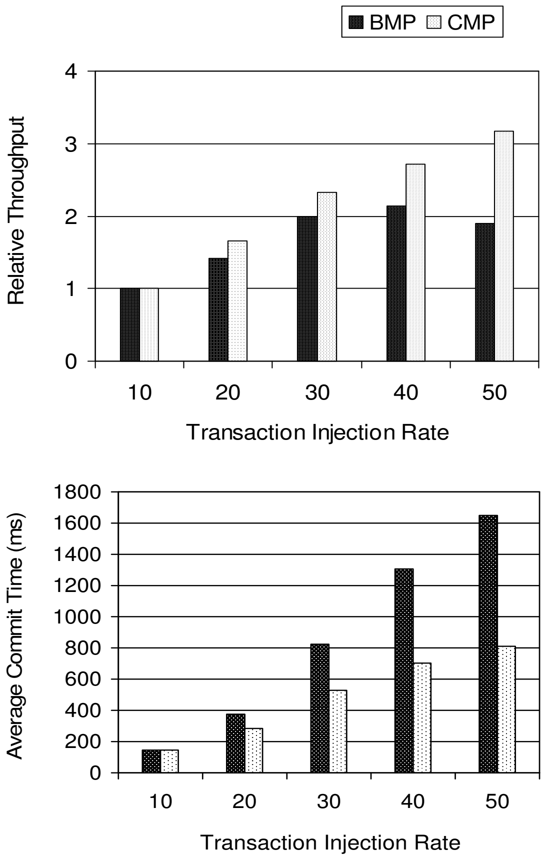


Figure 2.4: ECperf results with BMP vs. CMP.

argued in [154], there are some important reasons for this performance difference. Most importantly, giving the container control over the data access logic, allows for some automatic optimizations usually not implemented in BMP code. For example, the container can monitor which fields of an entity bean are modified during a transaction and make sure that only these fields are written to the database at commit time. This minimizes database access calls and avoids doing unnecessary work. Another optimization that is usually provided is related to the loading of entity beans. With BMP loading an entity bean usually requires two database calls:

1. ejbFind to find the respective database record and retrieve its primary key.
2. ejbLoad to read the entity bean's data from the database.

With CMP these steps are usually transparently combined into a single database access retrieving both the primary key and the data. Similar optimization can also be applied when loading a collection of N entity beans. With BMP this would require N+1 database calls, i.e. 1 ejbFind and N ejbLoads. With CMP the container can combine the N+1 calls into a single call.

Coming back to our results with ECperf (Figure 2.4), we see that the extra database calls when using BMP lead not only to higher response times, but also to the system getting saturated much more quickly. As a result, throughput starts to drop as we increase the IR beyond 40. So, generally speaking, if configured properly CMP usually performs much better than BMP and therefore our recommendation is to use CMP whenever possible. However, there are situations where one may not be able to use CMP. For example, in cases where the container does not directly support the persistence mechanism used, or it supports it, but some complex mappings need to be defined that are not supported. In such cases BMP holds an advantage over CMP, because it not only allows an arbitrary persistence mechanism to be used, but also provides complete control on how beans are mapped to storage structures. If one does not have a choice but to use BMP, there are several straightforward ways to improve its performance [84]. First of all, the N+1 database calls problem described above can be eliminated by using the so-called *Fat Key Pattern* [101]. Second, most containers provide sophisticated caching mechanisms that can be exploited to reduce the load on the database. For example, if an entity is never modified by external applications, one can avoid having to load the entity data at the beginning of every transaction. Once the entity data is loaded from the database, it can be used for multiple transactions without any further database accesses. Finally, BMP performance can be improved significantly by using parameterized prepared SQL statements [125]. This reduces the load on the DBMS by allowing it to reuse cached execution plans for statements that were already prepared. Most application servers provide a prepared statement cache as part of the connection pool manager.

2.4.3 The ECperf Persistence Bottleneck

Apart from running ECperf with Oracle we also conducted some experiments with Informix. However, surprisingly enough, the benchmark was exhibiting quite a different behavior when run with Informix. More specifically, the persistence layer was turning into a system bottleneck and preventing the benchmark to stress the application server and measure its performance. We now take a closer look at the sources of this problem and then proceed to offer a concrete solution to eliminate the persistence bottleneck.

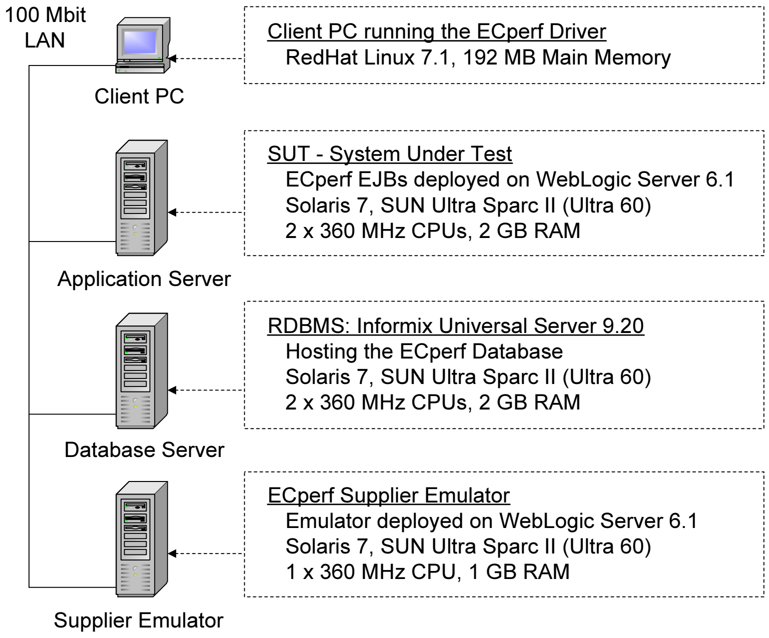

100 Mbit LAN
Client PC
Client PC running the ECperf Driver
RedHat Linux 7.1, 192 MB Main Memory
SUT - System Under Test
ECperf EJBs deployed on WebLogic Server 6.1
Solaris 7, SUN Ultra Sparc II (Ultra 60)
2 x 360 MHz CPUs, 2 GB RAM
Application Server
RDBMS: Informix Universal Server 9.20
Hosting the ECperf Database
Solaris 7, SUN Ultra Sparc II (Ultra 60)
2 x 360 MHz CPUs, 2 GB RAM
Database Server
ECperf Supplier Emulator
Emulator deployed on WebLogic Server 6.1
Solaris 7, SUN Ultra Sparc II (Ultra 60)
1 x 360 MHz CPU, 1 GB RAM
Supplier Emulator

Figure 2.5: Deployment environment with Informix.

Figure 2.5 depicts our second deployment environment with Informix as database server. When we ran ECperf out-of-the-box in this environment we monitored the database and observed very high data contention levels. Unlike Oracle, Informix employs a pessimistic scheduler which uses a locking-based concurrency control technique based on the popular 2-Phase-Locking (2PL) Protocol [170]. Under 2PL data items are locked before being accessed. Concurrent transactions trying to access locked data items in conflicting mode are either aborted or blocked waiting for the locks to be released. When running ECperf we noticed that large amounts of data access operations were resulting in lock conflicts which were blocking the respective transactions. A high proportion of the latter were eventually being aborted because of either timing out or causing a deadlock. As a result, very poor throughput levels were achievable and raising the IR beyond 2 caused a sudden drop in throughput - a phenomenon known as *data thrashing* [160].

The first thing that comes to mind when trying to reduce data contention is to decrease the locking granularity [170]. After setting up Informix to use row-level locks (instead of page-level locks) we observed a significant increase in throughput.

To further optimize the data layer we tried decreasing the isolation level [135].

We configured all entity beans to use the SQL `TRANSACTION_COMMITTED_READ` isolation level although this could compromise data consistency and integrity. However, the ECperf specification [155] does not place a restriction with respect to this. While these obvious optimizations could help to alleviate the identified bottleneck, they could not eliminate it and make ECperf behave as intended.

Getting to the Core of the Problem

After conducting a number of experiments and monitoring the database we noticed the following: the crucial transaction **scheduleWorkOrder** of the WorkOrderSes bean in the manufacturing domain was taking relatively long to complete, while holding exclusive locks on some highly demanded database tables. The transaction first creates a new work order entity, then identifies the components that make up the requested assembly in the BOM and assigns the required parts from inventory. The transaction can also cause calls into the supplier domain in case some parts get depleted and new amounts need to be ordered. Figure 2.6 depicts the execution step-by-step.

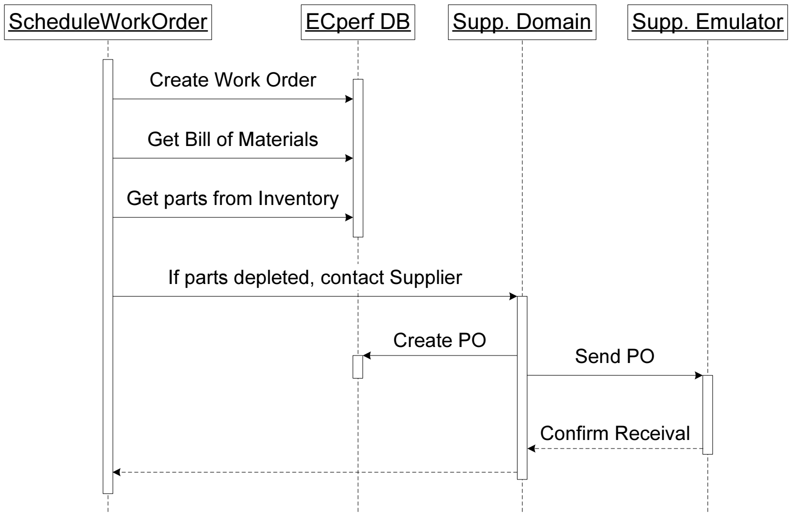


Figure 2.6: The scheduleWorkOrder transaction.

The scheduleWorkOrder transaction proceeds as follows:

1. Create a work order.
   - Insert a row in the `M_WORKORDER` table.
2. Start processing the work order (stage 1 processing).
   - Get the BOM.
   - Assign required parts from inventory.
3. If parts need to be ordered send a purchase order.
   - Insert rows in the `S_PURCHASE_ORDER` and `S_PURCHASE_ORDERLINE` tables.
   - Send the purchase order to the supplier emulator - order is sent in XML format through HTTP.

We identified two problems with this design of the scheduleWorkOrder transaction. First, sending the purchase order (the last step) delays the transaction while holding locks on the previously inserted table and index entries. We monitored the database lock tables and observed that indeed most of the lock conflicts were occurring when trying to access the `M_WORKORDER`, `S_PURCHASE_ORDER` and `S_PURCHASE_ORDERLINE` tables or their indices. This supported our initial suspicion that it was the access to these tables that was causing the bottleneck. The second problem is that the sending step is not implemented to be undoable. In other words once a purchase order is sent to the supplier emulator, its processing begins and this processing is not cancelled even if the scheduleWorkOrder transaction that sent the order is eventually aborted. Indeed, if this occurs, all actions of the scheduleWorkOrder transaction are rolled back except for the sending step. As a result, the respective purchase order is removed from the `S_PURCHASE_ORDER` table, but the supplier emulator is not notified that the order has been cancelled and continues processing it. Later when the order is delivered no information about it will be found in the database and an exception will be triggered. Thus, while the first problem has to do with data contention and performance, the second one concerns the scheduleWorkOrder transaction's atomicity and semantics.

We suggested modifications to the benchmark that aim at eliminating the identified bottleneck. Our approach breaks up the scheduleWorkOrder transaction into smaller separate transactions. This is known as *transaction chopping* in the literature [170]. The main goal is to commit update operations as early as possible, so that respective locks are released. We strive to isolate time-consuming operations in separate transactions that do not require exclusive locks. At the same time we ensure that transaction semantics are correct. We have identified and proposed two different solutions to the problem. In the first one we keep adhering to the EJB 1.1

specification, while in the second one, we utilize some services defined in the EJB 2.0 specification. Here we only consider the second solution since it has some significant advantages over the first one. Readers interested in the EJB 1.1 solution are referred to [80].

Utilizing Messaging and Asynchronous Processing

The problem with the scheduleWorkOrder transaction was that the sending of the purchase order may delay the transaction while holding highly demanded locks. On the one hand, we want to move this step into a separate transaction to make scheduleWorkOrder finish faster. On the other hand, we need to guarantee that the sending operation is executed atomically with the rest of the scheduleWorkOrder transaction. In other words, if the transaction commits, the purchase order should be sent, if the transaction aborts the purchase order should be destroyed and never be sent. In fact, as already discussed above, this atomicity is not guaranteed by the given design of ECperf. However, we believe that this is what the correct behavior of the scheduleWorkOrder transaction should be. We would like to note that in the given situation we do not have the requirement that the sending of the purchase order must be executed to its end before scheduleWorkOrder commits. We only need to make sure that, provided that the transaction commits, the sending operation is eventually executed. This situation lends itself naturally to *asynchronous processing and messaging.*

Messaging is an alternative to traditional *Request-Reply processing*. Request-Reply is usually based on Remote Method Invocation (RMI) or Remote Procedure Call (RPC) mechanisms. Under these mechanisms, a client sends a request (usually by calling a method on a remote object) and is then blocked, waiting until the request is processed to its end. This is exactly our situation above with the sending of the purchase order. This blocking element prevents the client from performing any processing in parallel while waiting for the server - a problem that has long been a thorn for software developers and whose solution has led to the emergence of *Messaging* and *Message-Oriented Middleware (MOM)* as an alternative to Request-Reply. In fact, it is a solution defined long time ago with the advent of the so-called *queued transaction processing models* [18]. In a nutshell, the idea behind messaging is that a middleman is introduced, sitting between the client and the server [140]. The middleman receives messages from one or more message producers and broadcasts these messages to possibly multiple message consumers. This allows a producer to send a message and then continue processing while the message is being delivered and processed. The message producer can optionally be later notified when the message is completely processed. A special type of messaging is the so-called *point-to-point messaging* in which each message is delivered to a single consumer. Messages are sent to a centralized *message queue* where they are processed usually on a First-In-First-Out (FIFO) basis. Multiple consumers can grab messages off the queue, but any given message is consumed exactly once. The Java Messaging Service (JMS) [159] is a standard Java API for accessing MOM infrastructures and the EJB 2.0 specification [157] integrates JMS with EJB by introducing *Message-Driven Beans (MDBs)*. The latter are components that act as message consumers in that they receive and process messages delivered by the MOM infrastructure. For example in a point-to-point messaging scenario MDBs can be used to process messages arriving on a message queue.

Eliminating the Persistence Bottleneck

We wanted to allow the scheduleWorkOrder transaction to commit before the purchase order is sent, but with the guarantee that it will eventually be sent. By utilizing messaging we can simply send a message to the supplier domain notifying it that a new order has been created and must be sent. After this we can commit our transaction and release all locks. A dedicated MDB can be deployed in the supplier domain to handle incoming messages by sending orders to the supplier emulator.

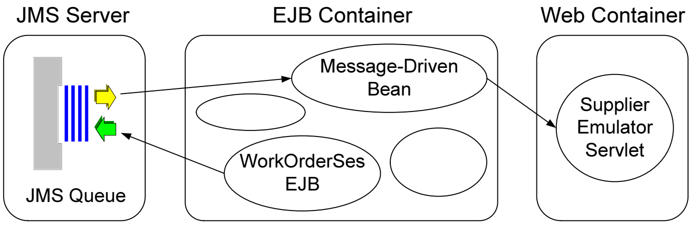


Figure 2.7: Sending purchase orders asynchronously.

We only need to make sure that the operation that sends the notification message to the supplier domain is executed as part of the scheduleWorkOrder transaction. This would ensure that the sending of the message is executed atomically with the rest of the transaction. Furthermore, modern messaging infrastructures provide a *guaranteed message delivery* option which ensures that once a transaction is committed all messages it has sent will be delivered even if respective consumers were down at the time of sending. Taking advantage of this property, we can claim that if the scheduleWorkOrder transaction creates a purchase order and then successfully commits, we have the guarantee that the supplier domain will eventually be notified and the purchase order will be sent out to the supplier emulator. The latter enables us to safely move our sending step into a separate transaction and execute it asynchronously. Figure 2.7 illustrates this solution. The new design of the scheduleWorkOrder transaction is depicted in Figure 2.8.

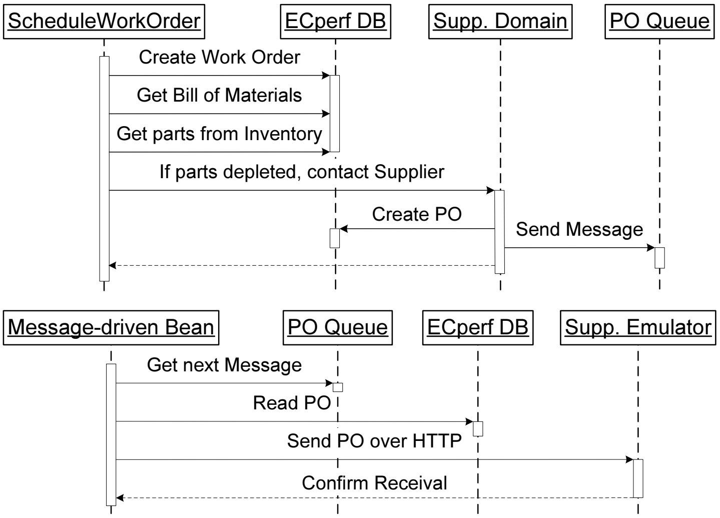


Figure 2.8: New design of the scheduleWorkOrder transaction.

Figure 2.9 compares throughput (BBops/min) achieved with Informix when running ECperf out-of-the-box against throughput achieved after implementing our messaging-based redesign. All data is relative to the throughput that we obtained when running ECperf out-of-the-box at IR of 1. As we can see, throughput increases linearly up to IR of 6 and then gradually starts to drop as we go beyond IR of 10. This is because at this point the application server is saturated (its CPU utilization approaches 100%). Not only does the asynchronous design bring a big performance advantage, but it also eliminates the second problem that we mentioned regarding the atomicity of the scheduleWorkOrder transaction. There is no way for a purchase order to be cancelled after it has been sent to the supplier emulator. A further benefit that we get is related to the application's reliability. Under the original design if the supplier emulator is down at the time a new purchase order is being created, the scheduleWorkOrder transaction will be aborted after timing out and all its work will be lost. With the new design the notification message will be sent successfully and although the sending of the order will be delayed until the supplier emulator comes up, the scheduleWorkOrder transaction will be able to commit successfully.

One might wonder why this bottleneck in ECperf was not noticed earlier. The answer is that everyone had only been testing with Oracle where an optimistic multi-version concurrency control protocol is used. The important advantage of this protocol is that it never blocks read operations even when concurrent updates

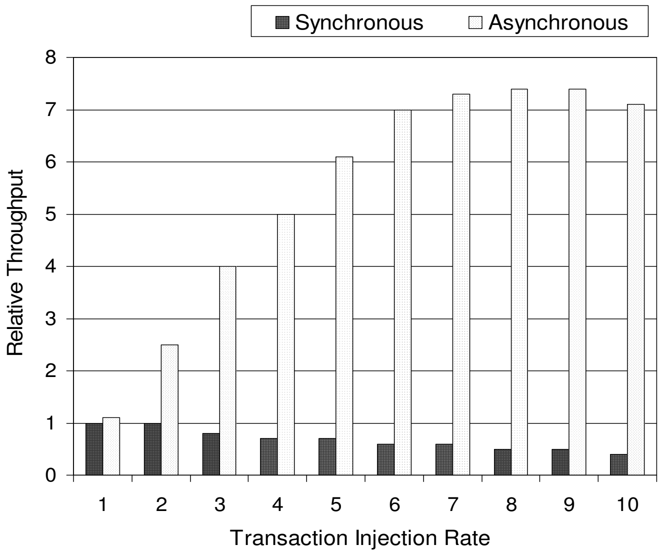


Figure 2.9: Synchronous vs. asynchronous variant of ECperf with Informix.

are taking place. For this particular workload Oracle's protocol obviously proved to perform much better. However, we will now show that even with an Oracle DBMS our redesign brings some significant performance and reliability benefits.

Figure 2.10 compares the synchronous and asynchronous variants of ECperf when run with Oracle. The performance gains of the asynchronous variant are considerably lower in this case, but they increase steadily as we raise the IR. The average CPU utilization of the database server and the application server during the experiments is shown on Figure 2.11. We can see that under the synchronous variant of ECperf the CPU utilization of the application server approaches 100% at IR of 6. This explains why there is hardly any increase in throughput at higher IRs. Under the asynchronous variant, the CPU utilization of the application server is constantly lower because purchase orders are not sent immediately. As as result, the application server is saturated much later, namely at IR of 10 instead of 6. This enables us to achieve higher throughput levels at IRs beyond 6 and explains why the database server's CPU utilization is higher at these IRs. As far as the database server is concerned, there is almost no difference in CPU utilization. This is because whether we send purchase orders synchronously or asynchronously does not affect the amount of work that has to be done by the database server.

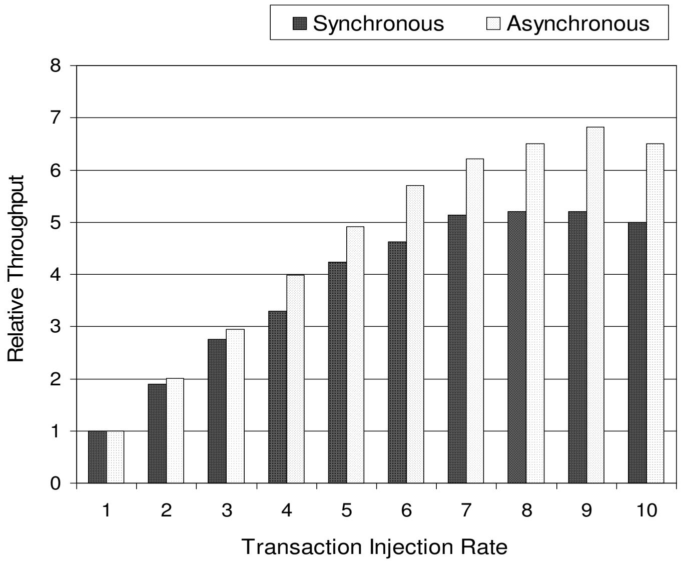
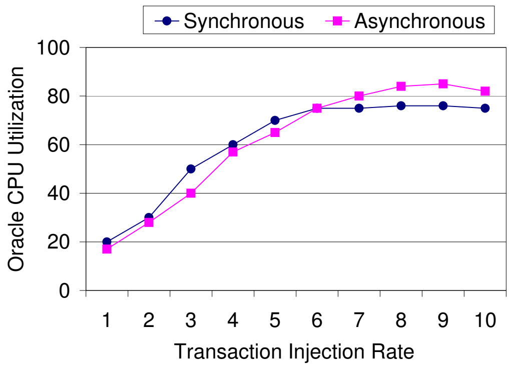


Figure 2.10: Synchronous vs. asynchronous variant of ECperf with Oracle 9i.

However, we must note that the scenario under which we carried out our experiments is not realistic in the sense that the supplier emulator is contacted over a LAN. In real life, contacting the supplier emulator might take much longer if we have to go over a WAN. This would result in much higher network delays and would further degrade the performance of the synchronous ECperf design. In order to illustrate this, we modified the supplier emulator to impose an artificial delay before confirming receipt of the purchase order. We then carried out some experiments under the same transaction IR, but with different length of the simulated network delay. Figure 2.12 shows the impact of the emulator delays on the throughput of the manufacturing application. Obviously, there was hardly any impact on the asynchronous variant of ECperf because the delays were not blocking the transactions in the manufacturing domain. However, this was not the case for the synchronous variant of ECperf where the delays were making the scheduleWorkOrder transaction finish slower and in this way were directly affecting the throughput of the manufacturing application. We can see how quickly the throughput drops as we increase the length of the network delay.

ECperf and its successors SPECjAppServer2001 and SPECjAppServer2002 were intended as DBMS-independent benchmarks and some changes were required in



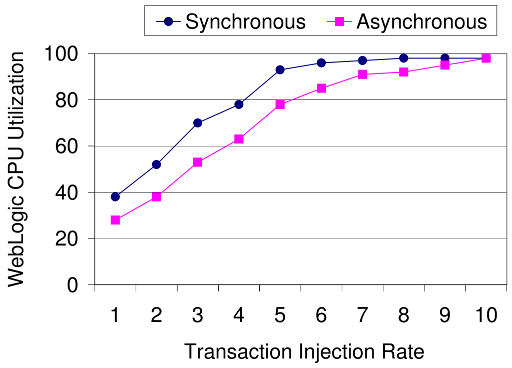


Figure 2.11: CPU utilization of the database server and the application server.

order to make this a reality. The proposed modifications were submitted both to the ECperf expert group at Sun Microsystems and to SPEC's OSG-Java subcommittee where they were discussed and approved. The modifications were implemented in a future version of the benchmark that is presented in the next section.

2.5 The SPECjAppServer2004 Benchmark

SPECjAppServer2004 is the new industry-standard benchmark for measuring the P&S of J2EE hardware and software platforms, i.e. the successor benchmark of SPECjAppServer2002, SPECjAppServer2001 and ECperf. SPECjAppServer2004 was developed by SPEC's Java subcommittee, which includes IBM, Darmstadt

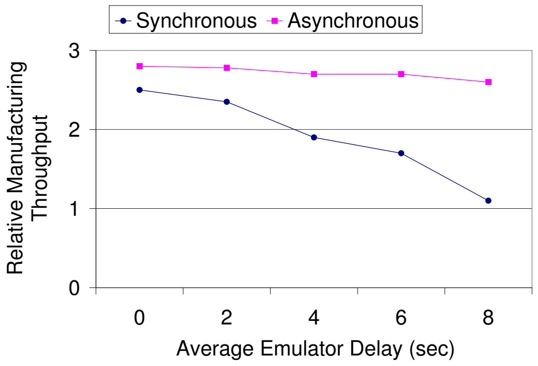


Figure 2.12: Manufacturing throughput as we increase the network delay.

University of Technology, Sun Microsystems, BEA, Borland, Intel, Oracle, Hewlett-Packard, Pramati and Sybase. It is important to note that even though SPECjAppServer2004's business model looks similar to SPECjAppServer2002 (and respectively to ECperf), SPECjAppServer2004 is a new benchmark substantially different from previous versions of SPECjAppServer [153]. It implements a new enhanced workload that exercises all major services of the J2EE platform, including the Web container and the Java Messaging Service (JMS), in a complete end-to-end application scenario. Asynchronous messaging is exploited as proposed in Section 2.4.3 (resp. [80, 83]) to provide maximum scalability and address performance and reliability issues.

2.5.1 SPECjAppServer2004 Business Model

The SPECjAppServer2004 workload has been specifically modeled after an automobile manufacturer whose main customers are automobile dealers [147]. Dealers use a Web based user interface to browse an automobile catalogue, purchase automobiles, sell automobiles and track dealership inventory. As depicted in Figure 2.13, SPECjAppServer2004's business model comprises five domains: *customer domain* dealing with customer orders and interactions, *dealer domain* offering Web based interface to the services in the customer domain, *manufacturing domain* performing "just in time"manufacturing operations, *supplier domain* handling interactions with external suppliers, and *corporate domain* managing all dealer, supplier and automobile information. With exception of the dealer domain, these domains are similar to their respective domains in SPECjAppServer2002 in terms of the services they provide. However, their implementation is different.

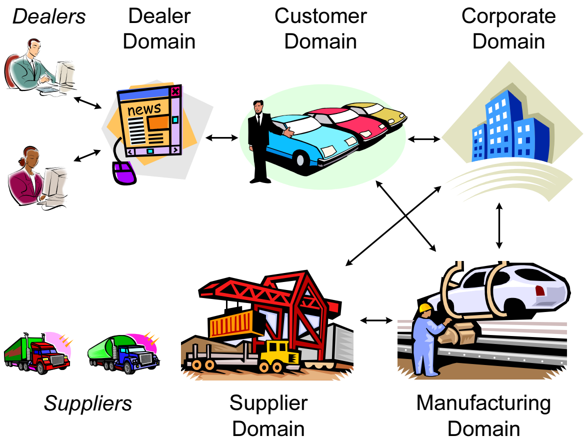


Figure 2.13: SPECjAppServer2004 business model.

The customer domain hosts an *order entry application* that provides some typical online ordering functionality. The latter includes placing new orders, retrieving the status of an order or all orders of a given customer, canceling orders and so on. Orders for more than 100 automobiles are called *large orders*.

The dealer domain hosts a Web application (called *dealer application*) that provides a Web based interface to the services in the customer domain. It allows customers, in our case automobile dealers, to keep track of their accounts, keep track of dealership inventory, manage a shopping cart, and purchase and sell automobiles.

The manufacturing domain hosts a *manufacturing application* that models the activity of production lines in an automobile manufacturing plant. Similarly to SPECjAppServer2002, there are two types of production lines, *planned lines* and *large order lines*. Planned lines run on schedule and produce a predefined number of automobiles. Large order lines run only when a large order (LO) is received in the customer domain. The unit of work in the manufacturing domain is a *work order*. Each work order is for a specific number of automobiles of a certain model. When a work order is created, the bill of materials for the corresponding type of automobile is retrieved and the required parts are taken out of inventory. As automobiles move through the assembly line, the work order status is updated to reflect progress. Once the work order is complete, it is marked as completed and inventory is updated. When the inventory of parts gets depleted, suppliers need to be located and purchase orders need to be sent out. This is done by contacting the supplier domain, responsible for interactions with external suppliers. A supplier is chosen based on the parts that need to be ordered, the time in which they are required and the prices quoted. A purchase order (PO) is sent to the selected supplier. The purchase order includes the quantities of the various parts being ordered, the site they must be delivered to and the date by which delivery must happen. When the parts are delivered, the supplier domain sends a message to the manufacturing domain to update inventory.

2.5.2 Benchmark Design and Workload

All the activities and processes in the five domains described above are implemented using J2EE components (Enterprise Java Beans, Servlets and Java Server Pages) assembled into a single J2EE application that is deployed in an application server running on the *System Under Test (SUT)*. The only exception is for the interactions with suppliers which are implemented using a separate Web application called *supplier emulator*. The latter is deployed in a Java-enabled Web server on a dedicated machine. The supplier emulator provides the supplier domain with a way to emulate the process of sending and receiving purchase orders to/from suppliers.

A relational DBMS is used for data persistence and all data access operations use entity beans which are mapped to tables in the SPECjAppServer database. All entity beans use CMP and follow the guidelines in [83] to provide maximum scalability and performance. Communication across domains is implemented using asynchronous messaging exploiting JMS and MDBs. In particular, the placement and fulfillment of large orders, requiring communication between the customer domain and the manufacturing domain, is implemented asynchronously. Another example is the placement and delivery of supplier purchase orders, which requires communication between the manufacturing domain and the supplier domain. The latter is implemented according to the proposal in Section 2.4.3 (resp. [80, 83]) to address performance and reliability issues.

The workload generator is implemented using a multi-threaded Java application called *SPECjAppServer driver*. The latter is designed to run on multiple client machines using an arbitrary number of JVMs to ensure that it has no inherent scalability limitations. The driver is made of two components - *manufacturing driver* and *DealerEntry driver*. The manufacturing driver drives the production lines (planned lines and large order lines) in the manufacturing domain and exercises the manufacturing application. It communicates with the SUT through the RMI interface. The DealerEntry driver emulates automobile dealers that use the dealer application in the dealer domain to access the services of the order entry application in the customer domain. It communicates with the SUT through HTTP and exercises the dealer and order entry applications using three operations referred to as *business transactions*:

1. Browse - browses through the vehicle catalogue.

2. Purchase - places orders for new vehicles.
3. Manage - manages the dealer inventory (sells vehicles and/or cancels open orders).

Each business transaction emulates a specific type of client session comprising multiple round-trips to the server. For example, the Browse transaction navigates to the vehicle catalogue Web page and then pages a total of thirteen times, ten forward and three backwards. Business transactions are selected by the driver based on the mix shown in Table 2.2. The actual mix achieved by the benchmark must be within 5% of the targeted mix for each type of business transaction. For example, Browse transactions can vary between 47.5% to 52.5% of the total mix. The driver checks and reports on whether the mix requirement was met.

Table 2.2: SPECjAppServer2004 business transaction mix requirements.

Business Transaction	Percent Mix
Browse	50%
Purchase	25%
Manage	25%

The throughput of the benchmark is driven by the activity of the dealer and manufacturing applications. The throughput of both applications is directly related to the chosen transaction *Injection Rate (IR)*. The latter determines the number of business transactions generated by the DealerEntry driver, and the number of work orders scheduled by the manufacturing driver per unit of time. The summarized performance metric provided after running the benchmark is called **JOPS** and it denotes the average number of successful **J**AppServer **O**perations **P**er **S**econd completed during the measurement interval. JOPS is composed of the total number of business transactions completed in the dealer domain, added to the total number of work orders completed in the manufacturing domain, normalized per second.

To ensure that benchmark results are correctly obtained and all requirements are met, the driver makes explicit audit checks by calling certain components on the SUT both at the beginning and at the end of the run. The driver includes audit results with the run results. For a list of the individual auditing activities the driver performs, refer to the benchmark Run and Reporting Rules [147].

The most important J2EE services exercised by the benchmark are the following:

- The Web container, including Servlets and JSPs.

- The EJB container.
- EJB 2.0 container-managed persistence.
- Transaction management.
- JMS and MDBs.
- Database connectivity.

Note that the benchmark also heavily exercises all parts of the underlying infrastructure that make up the application environment, including hardware, JVM software, database software, JDBC drivers and the system network.

2.5.3 Standard vs. Distributed Workload

To satisfy the requirements of a wide variety of customers, the SPECjAppServer2004 benchmark can be run in *standard* or *distributed* mode [147]. In the standard version of the workload, the benchmark domains are allowed to be combined. This means that the benchmark implementer can choose to run a single *deployment unit* that accesses a single database containing the tables of all domains. However, the benchmark implementer is free to separate the domains into their deployment units with one or more database instances.

The distributed version of the workload is intended to measure application performance where the world-wide enterprise that the benchmark models performs business transactions across business domains employing heterogeneous resource managers. In this model, the workload requires a separate deployment unit and a separate DBMS instance for each domain. XA-compliant recoverable 2-phase commits [162] are required for business transactions that span multiple domains. The configuration for the 2-phase commit is required to be done in a way that would support heterogeneous systems. Even though implementations are likely to use the same type of resource manager for all domains, the J2EE servers and resource managers cannot take advantage of the knowledge of homogeneous resource managers to optimize the 2-phase commits.

2.6 Case Study with SPECjAppServer2004 on JBoss

The JBoss application server is the world's most popular open-source J2EE application server. Combining a robust, yet flexible architecture with a free open-source license and extensive technical support from the JBoss Group, JBoss has quickly established itself as a competitive platform for e-business applications. However, like other open-source products, JBoss has often been criticized for having poor P&S, failing to meet the requirements for mission-critical enterprise-level services.

In this section[3], we use the SPECjAppServer2004[4] benchmark to study how the performance of J2EE applications running on JBoss can be improved by exploiting different deployment and tuning options offered by the platform. We start by comparing several alternative Web containers (servlet/JSP engines) that are typically used in JBoss applications, i.e. Tomcat 4, Tomcat 5 and Jetty. Following this, we evaluate the performance difference when using local interfaces, as opposed to remote interfaces, for communication between the presentation layer (Servlets/JSPs) and the business layer (EJBs) of the application. We measure the performance gains from several typical data access optimizations often used in JBoss applications and demonstrate that the choice of JVM has very significant impact on the overall system performance. The exact version of JBoss server considered is 3.2.3, released on November 30, 2003. Even though our study is specific to JBoss, most of the results and findings of this section are generalized to other J2EE application servers.

2.6.1 Experimental Setting

In our experimental analysis, we use two different deployment environments for SPECjAppServer2004, depicted in Figures 2.14 and 2.15, respectively. The first one is a single-node deployment, while the second one is a clustered deployment with four JBoss servers. Table 2.3 provides some details on the configuration of the machines used in the two deployment environments. Since JBoss exhibits different behavior in clustered environment, the same deployment option (or tuning parameter) might have different effect on performance when used in the clustered deployment, as opposed to the single-node deployment. Therefore, we consider both deployment environments in our analysis.

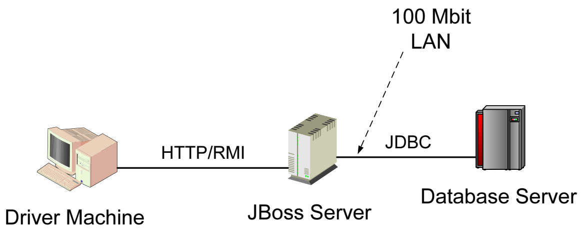


Figure 2.14: Single-node deployment.

[3]The material presented in this section is joint work with Björn Weis [88, 171].

[4]The SPECjAppServer2004 results or findings in this section have not been reviewed by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official Web site for SPECjAppServer2004 is located at http://www.spec.org/jAppServer2004.

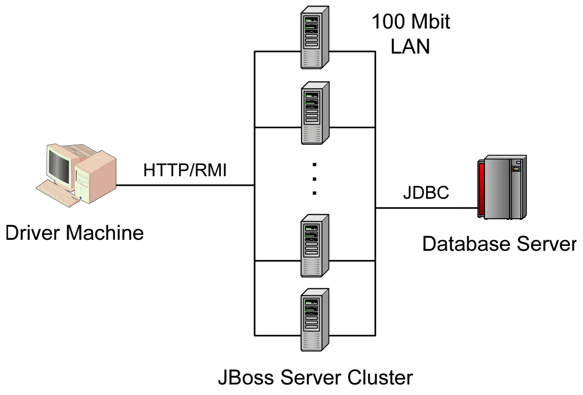


Figure 2.15: Clustered deployment.

Table 2.3: Deployment environment details.

Node	Description
Driver Machine	SPECjAppServer Driver & Supplier Emulator 2 x AMD XP2000+ CPU 2 GB, SuSE Linux 8
Single JBoss Server	JBoss 3.2.3 Server 2 x AMD XP2000+ CPU 2 GB, SuSE Linux 8
JBoss Cluster Nodes	JBoss 3.2.3 Server 1 x AMD XP2000+ CPU 1 GB, SuSE Linux 8
Database Server	Popular commercial DBMS 2 x AMD XP2000+ CPU 2 GB, SuSE Linux 8

JBoss is shipped with three standard server configurations: *"minimal"*, *"default"* and *"all"*. The "default"configuration is typically used in single-server environments, while the "all"configuration is meant for clustered environments. We use the "default"configuration as a basis for the single JBoss server in our single-node deployment, and the "all"configuration as a basis for the N JBoss servers in our clustered deployment. For details on the changes made to the standard server configurations for deploying SPECjAppServer2004, the reader is referred to [171]. In both the single-node and the clustered deployments, all SPECjAppServer2004 components (EJBs, servlets, JSPs) are deployed on all JBoss servers. In the clustered deployment, client requests are evenly distributed over the JBoss servers in the cluster.

The driver machine hosts the SPECjAppServer2004 driver and the supplier emulator. All entity beans are persisted in the database. The DBMS we use runs under SQL isolation level of READ_COMMITTED by default. For entity beans required to run under REPEATABLE_READ isolation level, pessimistic SELECT_FOR_UPDATE locking is used. This is achieved by setting the *row-locking* option in the `jbosscmp-jdbc.xml` configuration file.

We adhere to the SPECjAppServer2004 Run Rules for most of the experiments in our study. However, since not all deployment options that we consider are allowed by the Run Rules, in some cases we have to slightly deviate from the latter. For example, when evaluating the performance of different entity bean commit options, in some cases we assume that the JBoss server has exclusive access to the underlying persistent store (storing entity bean data), which is disallowed by the Run Rules. This is acceptable, since our aim is to evaluate the impact of the respective deployment options on performance, rather than to produce standard benchmark results to be published and compared with other results.

2.6.2 Performance Analysis

We now present the results of our experimental analysis. We look at a number of different JBoss deployment and configuration options and evaluate their impact on the overall system performance. As a basis for comparison a standard out-of-the-box configuration is used with all deployment parameters set to their default values. Hereafter, we refer to this configuration as *Standard* (shortened *"Std"*). For each deployment/configuration setting considered, its performance is compared against the performance of the standard configuration. Performance is measured in terms of the following metrics:

- CPU utilization of the JBoss server(s) and the database server.
- Throughput of business transactions.
- Mean response times of business transactions.

By business transactions, here, we mean the three dealer operations, Purchase, Manage and Browse (as defined in Section 2.5.2) and the WorkOrder transaction running in the manufacturing domain. It is important to note that the IR at which experiments in the single-node environment are conducted is different from the IR for experiments in the clustered environment. A higher IR is used for cluster experiments, so that the four JBoss servers are utilized to a reasonable level. Disclosing the exact IRs at which experiments are run is not allowed by the SPECjAppServer2004 license agreement.

Use of Different Web Containers

JBoss allows a third-party Web container to be integrated into the application server framework. The most popular Web containers typically used are Tomcat [4] and Jetty [117]. By default Tomcat 4.1 is used. As of the time of writing, the integration of Tomcat 5 in JBoss is still in its beta stage. Therefore when using it, numerous debug messages are output to the console and logged to files. This accounts for significant overhead that would not be incurred in production deployments. For this reason, we consider two Tomcat 5 configurations, the first one out-of-the-box and the second one with debugging turned off. It is the latter that is more representative and the former is only included to show the overhead of debugging.

Since the manufacturing application does not exercise the Web container, it is not run in the experiments of this section. Only the dealer and order-entry applications are run, so that the stress is put on the benchmark components that exercise the Web container.

We consider four different configurations:

1. Tomcat 4.1 (shortened *Tom4)*
2. Tomcat 5 out-of-the-box (shortened *Tom5*)
3. Tomcat 5 without debugging (shortened *Tom5WD*)
4. Jetty

Comparing the four Web containers in the single-node deployment, revealed no significant difference with respect to achieved transaction throughput and average CPU utilization. With exception of Tom5WD, in all configurations, the measured CPU utilization was about 90% for the JBoss server and 45% for the database server. The Tom5WD configuration exhibited 2% lower CPU utilization both for the JBoss server and the database server. As we can see from Figure 2.16, the lower CPU utilization resulted in Tom5WD achieving the best response times, followed by Jetty. It stands out that the response time improvement was most significant for the Browse transaction. The reason for this is that, while Purchase and Manage comprise only 5 round-trips to the server, Browse comprises a total of 17 round-trips each going through the Web container. As mentioned, the effect on transaction throughput was negligible. This was expected since, for a given IR, SPECjAppServer2004 has a target throughput that is normally achieved unless there are some system bottlenecks.

The four Web containers exhibited similar behavior in the clustered deployment. The only exception was for the Tom5 configuration, which in this case was performing much worse compared to the other configurations. The reason for this was that, all four servers in the clustered deployment were logging their debug messages to the same network drive. Since, having four servers, means four times more debug information to be logged, the shared logging drive became a bottleneck. Figure 2.17 shows the response times of the three business transactions. Note that this diagram uses a different scale.

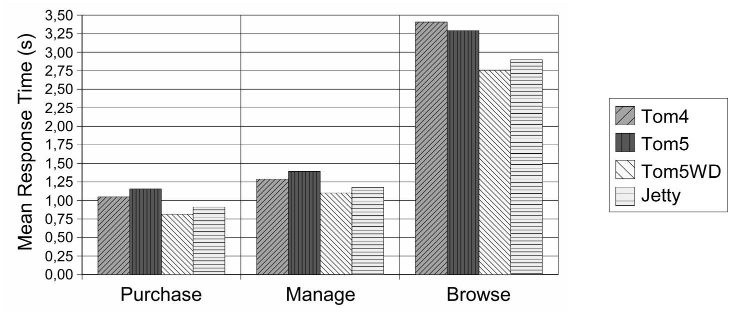


Figure 2.16: Mean response times with different Web containers in the single-node environment.

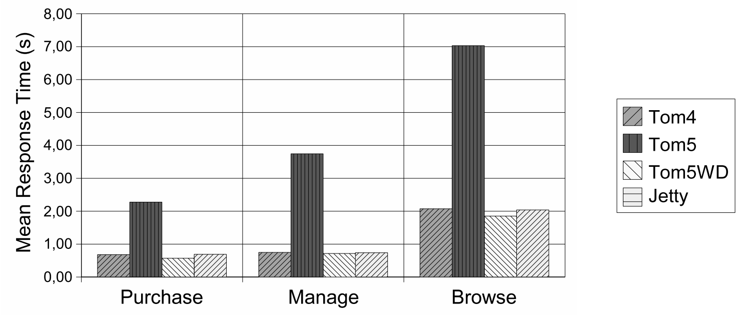


Figure 2.17: Mean response times with different Web containers in the clustered environment.

Use of Local vs. Remote Interfaces

In SPECjAppServer2004, by default, remote interfaces are used to access business logic components (EJBs) from the presentation layer (Servlets/JSPs) of the application. However, since in both the single-node and clustered environments, presentation components are co-located with business logic components, one can alternatively use local interfaces. This eliminates the overhead of remote network communication and is expected to improve performance. In this section, we evaluate the performance gains from using local interfaces to access EJB components from Servlets and JSPs in SPECjAppServer2004. Note that our standard configuration (i.e. Std) uses remote interfaces.

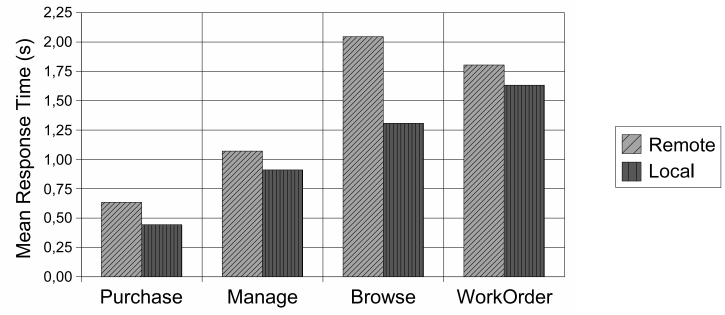


Figure 2.18: Mean response times with remote vs. local interfaces in the single-node environment.

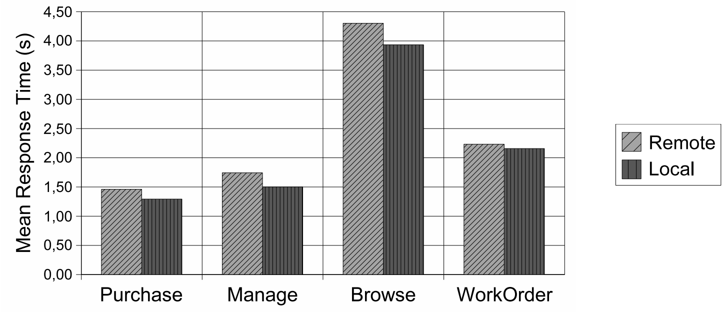


Figure 2.19: Mean response times with remote vs. local interfaces in the clustered environment.

Figure 2.18 shows the transaction response times with remote vs. local interfaces in the single-node deployment. As we can see, using local interfaces led to response times dropping up to 35%. Again, most affected was the Browse transaction. In addition to this, the use of local interfaces led to lower CPU utilization of the JBoss server. It dropped from 82% to 73% when switching from remote to local interfaces. Again, differences in transaction throughput were negligible.

As expected, switching to local interfaces brought performance gains also in the clustered deployment. However, in this case, the delays resulting from calls to the EJB layer were small compared to the overall response times. This is because in clustered environment, there is additional load balancing and synchronization overhead which contributes to the total response times. As a result, delays from calls to the EJB layer constitute smaller portion of the overall response times than in the single-node case. Therefore, the performance improvement from using local interfaces was also smaller than in the single-node case. Figure 2.19 shows the measured response times of business transactions. The effect on transaction throughput and CPU utilization was negligible.

Data Access Optimizations

In this section, we measure the effect of several data access configuration options on the overall system performance. The latter are often exploited in JBoss applications to tune and optimize the way entity beans are persisted. We first discuss these options and then present the results from our analysis.

Entity Bean Commit Options: JBoss offers four entity bean persistent storage commit options, i.e. A, B, C and D [29, 148]. While the first three are defined in the EJB specification [157], the last one is a JBoss-specific feature.

The four commit options have the following semantics:

- Commit Option A - the container caches entity bean state between transactions. This option assumes that the container has exclusive access to the persistent store and therefore it does not need to synchronize the in-memory bean state from the persistent store at the beginning of every transaction.

- Commit Option B - the container caches entity bean state between transactions, however unlike option A, the container is not assumed to have exclusive access to the persistent store. Therefore, the container has to synchronize the in-memory entity bean state at the beginning of every transaction. Thus, business methods executing in a transaction context do not see much benefit from the container caching the bean, whereas business methods executing outside a transaction context can take advantage of cached bean data.

- Commit Option C - the container does not cache bean instances.
- Commit Option D - bean state is cached between transactions as with option A, but the state is periodically synchronized with the persistent store.

Note that the standard configuration (i.e. Std) uses commit option B for all entity beans. We consider two modified configurations exploiting commit options A and C, respectively. In the first configuration (called *CoOpA*), commit option A is used. While in the single-node deployment commit option A can be configured for all SPECjAppServer2004 entity beans, doing so in the clustered deployment would introduce potential data inconsistency. The problem is that changes to entity data in different nodes of the cluster are normally not synchronized. Therefore, in the clustered deployment, commit option A is only used for the beans which are never modified by the benchmark (the read-only beans), i.e. AssemblyEnt, BomEnt, ComponentEnt, PartEnt, SupplierEnt, SupplierCompEnt and POEnt.

The second configuration that we consider (called *CoOpC*) uses commit option C for all SPECjAppServer2004 entity beans.

Entity-Bean-With-Cache-Invalidation Option: We mentioned that using commit option A in clustered environment may introduce potential data inconsistency. This is because each server in the cluster would assume that it has exclusive access to the persistent store and cache entity bean state between transactions. Thus, when two servers update an entity at the same time, the changes of one of them could be lost. To address this problem, JBoss provides the so-called *cache invalidation framework* [89]. The latter allows one to link the entity caches of servers in the cluster, so that when an entity is modified, all servers who have a cached copy of the entity are forced to invalidate it and reload it at the beginning of next transaction. JBoss provides the so-called "Standard CMP 2.x EntityBean with cache invalidation"option for entities that should use this cache invalidation mechanism. In our analysis, we consider a configuration (called *EnBeCaIn*) that exploits this option for SPECjAppServer2004's entity beans. Unfortunately, in the clustered deployment, it was not possible to configure all entity beans with cache invalidation, since doing so led to numerous rollback exceptions being thrown when running the benchmark. The latter appears to be due to a bug in the cache invalidation mechanism. Therefore, we could only apply the cache invalidation mechanism to the read-only beans, i.e. AssemblyEnt, BomEnt, ComponentEnt, PartEnt, SupplierEnt, SupplierCompEnt and POEnt. Since read-only beans are never modified, this should be equivalent to simply using commit option A without cache invalidation. However, as we will see later, experiments show that there is a slight performance difference.

Instance-Per-Transaction Policy: JBoss' default locking policy allows only one instance of an entity bean to be active at a time. Unfortunately, the latter often leads to deadlock and throughput problems. To address this, JBoss provides the so-called *Instance Per Transaction Policy*, which eliminates the above requirement and allows multiple instances of an entity bean to be active at the same time [148]. To achieve this, a new instance is allocated for each transaction and it is dropped when the transaction finishes. Since each transaction has its own copy of the bean, there is no need for transaction based locking.

In our analysis, we consider a configuration (called *InPeTr*) that uses the instance per transaction policy for all SPECjAppServer2004 entity beans.

No-Select-Before-Insert Optimization: JBoss provides the so-called *No-Select-Before-Insert* entity command, which aims to optimize entity bean create operations [148]. Normally, when an entity bean is created, JBoss first checks to make sure that no entity bean with the same primary key exists. When using the No-Select-Before-Insert option, this check is skipped. Since, in SPECjAppServer2004 all primary keys issued are guaranteed to be unique, there is no need to perform the check for duplicate keys. To evaluate the performance gains from this optimization, we consider a configuration (called *NoSeBeIn*) that uses the No-Select-Before-Insert option for all SPECjAppServer2004 entity beans.

Sync-On-Commit-Only Optimization: Another optimization typically used is the so-called *Sync-On-Commit-Only* container configuration option. It causes JBoss to synchronize changes to entity beans with the persistent store *only* at commit time. Normally, dirty entities are synchronized whenever a finder method is called. When using Sync-On-Commit-Only, synchronization is not done when finder methods are called, however, it is still done after deletes/removes to ensure that cascade deletes work correctly. We consider a configuration called *SyCoOnly*, in which Sync-On-Commit-Only is used for all SPECjAppServer2004 entity beans.

Prepared Statement Cache: In JBoss, by default, prepared statements are not cached. To improve performance one can configure a prepared statement cache of an arbitrary size [148]. We consider a configuration called *PrStCa*, in which a prepared statement cache of size 100 is used.

In summary, we are going to compare the following configurations against the standard (Std) configuration:

1. Commit Option A (CoOpA)

2. Commit Option C (CoOpC)

3. Entity Beans With Cache Invalidation (EnBeCaIn)

4. Instance Per Transaction Policy (InPeTr)

5. No-Select-Before-Insert (NoSeBeIn)

6. Sync-On-Commit-Only (SyCoOnly)

7. Prepared Statement Cache (PrStCa)

Analysis in Single-node Environment Figure 2.20 shows the average CPU utilization of the JBoss server and the database server under the different configurations in the single-node environment. Figure 2.21 shows the response times.

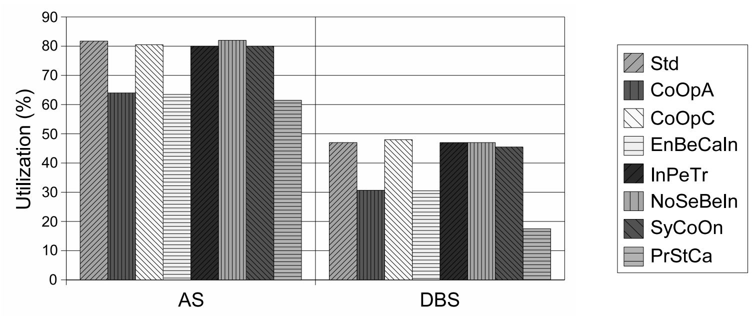


Figure 2.20: CPU utilization under different configurations in the single-node environment.

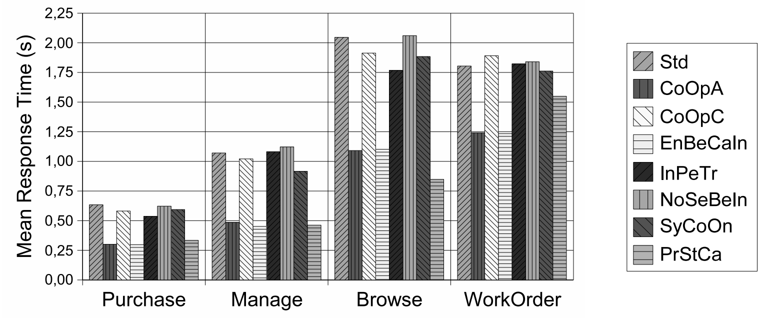


Figure 2.21: Mean response times under different configurations in the single-node environment.

All configurations achieved pretty much the same transaction throughput with negligible differences. As we can see, apart from the three configurations CoOpA, EnBeCaIn and PrStCa, all other configurations had similar performance. As expected, configurations CoOpA and EnBeCaIn had identical performance, since in a single-node environment there is practically no difference between them. Caching entity bean state with commit option A, resulted in 30 to 60 percent faster response times. Moreover, the CPU utilization dropped by 20% both on the JBoss server and the database server. The performance gains from using a prepared statement cache were even greater, i.e. the database utilization dropped from 47% to only 18%!

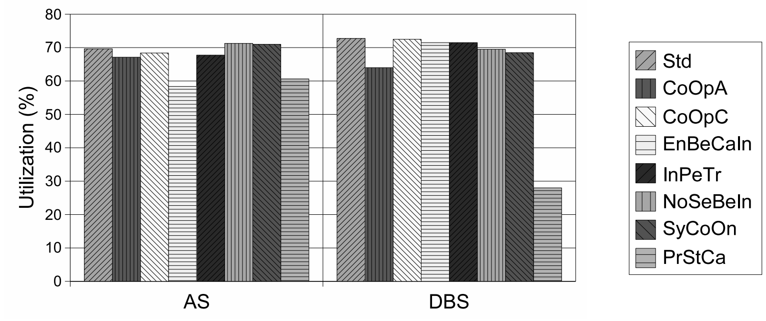


Figure 2.22: CPU utilization under different configurations in the clustered environment.

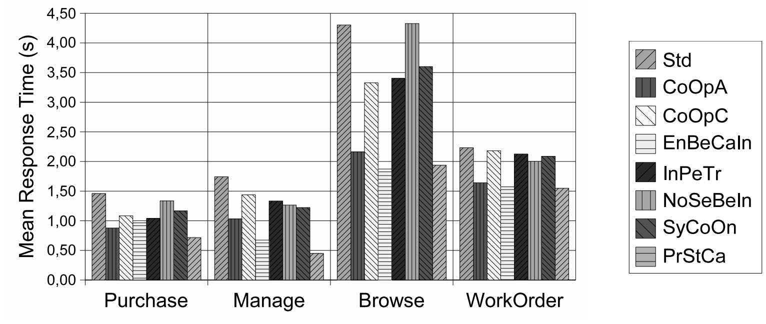


Figure 2.23: Mean response times under different configurations in the clustered environment.

Analysis in Clustered Environment Figure 2.22 shows the average CPU utilization of the JBoss servers and the database server under the different configurations in clustered environment. Figure 2.23 shows the response times of business transactions. As usual, all configurations achieved pretty much the same transaction throughput with negligible differences.

As we can see, the performance gains from caching entity bean state with commit option A (configurations CoOpA and EnBeCaIn) were not as big as in the single-node deployment. This was expected since, as discussed earlier, in this case only the read-only beans could be cached. It came as a surprise, however, that there was a difference between simply using commit option A for read-only beans (CoOpA), as opposed to using it with the cache invalidation option (EnBeCaIn). We did not expect to see a difference here, since cached beans, being read-only, were never invalidated. There was a slight difference, however, not just in CPU utilization, but also in transaction response times, which in most cases were slightly better for EnBeCaIn. Again, the performance gains from having a prepared statement cache were considerable, i.e. the database utilization dropped from 78% to only 27%!

Use of Different JVMs

In this section, we demonstrate that the underlying JVM has a very significant impact on the performance of JBoss applications. We compare the overall performance of our SPECjAppServer2004 deployments under two different popular JVMs. Unfortunately, we cannot disclose the names of these JVMs, as this is prohibited by their respective license agreements. We will instead refer to them as "JVM A"and "JVM B". The goal is to demonstrate that changing the JVM on which JBoss is running, may bring huge performance gains and therefore it is worth considering this in real-life deployments.

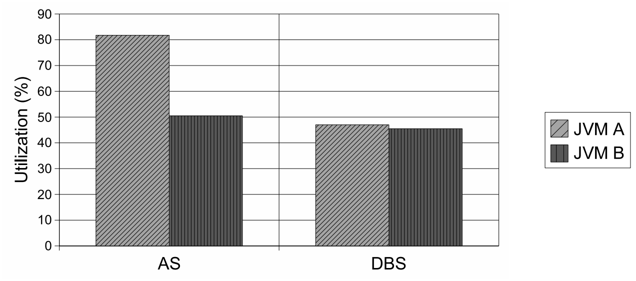


Figure 2.24: CPU utilization under the 2 JVMs in the single-node environment.

Figures 2.24 and 2.25 show the average CPU utilization of the JBoss server and the database server under the two JVMs in the single-node and clustered environment, respectively. Mean response times are shown in Figures 2.26 and 2.27. As evident, in both environments, JBoss runs much faster with the second JVM: response times are up to 55% faster and the CPU utilization of the JBoss server is much lower.

2.6.3 Conclusions from the Analysis

Comparing the three alternative Web containers Tomcat 4.1, Tomcat 5 and Jetty revealed no significant performance differences. Tomcat 5 (with debugging turned off) and Jetty turned out to be slightly faster than Tomcat 4.1. On the other hand, using local interfaces to access business logic components from the presentation layer brought significant performance gains. Indeed, response times dropped by

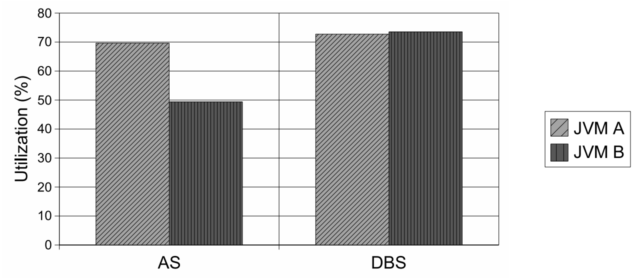


Figure 2.25: CPU utilization under the 2 JVMs in the clustered environment.

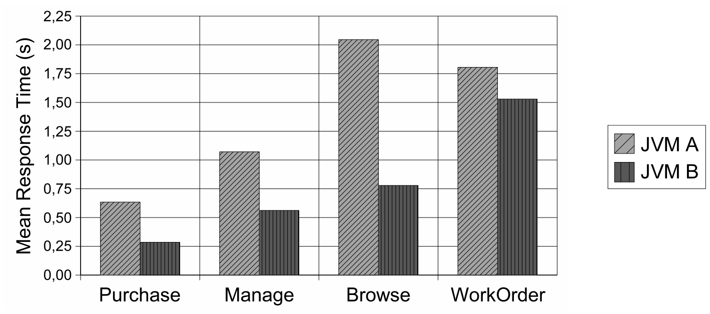


Figure 2.26: Response times under the 2 JVMs in the single-node environment.

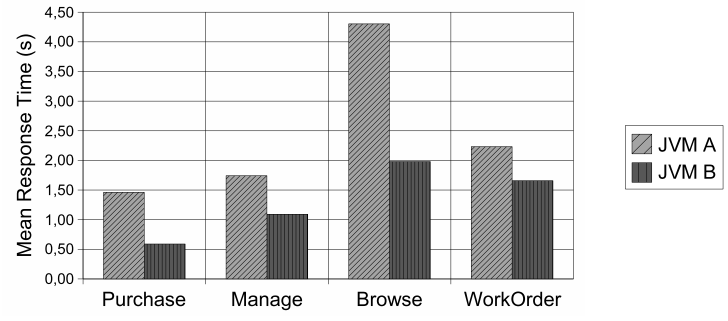


Figure 2.27: Response times under the 2 JVMs in the clustered environment.

up to 35% and the utilization of JBoss server dropped by 10%. Caching entity beans with commit option A (with and without cache invalidation) resulted in 30 to 60 percent faster response times and 20% lower CPU utilization both for the JBoss server and the database server. The performance gains from caching prepared statements were even greater. The other data access optimizations considered led to only minor performance improvements. However, we must note here that, being an application server benchmark, SPECjAppServer2004 places the emphasis on the application server (transaction processing) rather than the database. We expect the performance gains from data access optimizations to be more significant in applications where the database is the bottleneck. Furthermore, in our analysis we considered optimizations one at a time. If all of them were to be exploited at the same time, their cumulative effect would obviously lead to more significant performance improvements. Finally, we demonstrated that switching to a different JVM may improve JBoss performance by up to 60%.

Even though our study was specific to JBoss, most of the conclusions we drew above can be carried over to other J2EE application servers. Conducting tests with several commercial application servers showed similar behavior with respect to the performance gains from caching beans with commit option A, the effect of using local interfaces, the benefits of caching prepared statements and the significance of the choice of JVM.

2.7 Concluding Remarks

Benchmarking platforms prior to starting system development helps to ensure that the platforms selected meet the requirements for P&S. Alternative hardware and software platforms can be compared choosing the ones that provide the best cost/performance ratio. Another benefit of benchmarking is that it helps to identify the platform deployment settings and configuration parameters that are most critical for the overall system P&S. However, in order for benchmark results to be useful, the benchmarks employed must fulfill the following five important requirements:

1. They must be representative of real-world systems.
2. They must exercise and measure all critical services provided by platforms.
3. They must not be tuned/optimized for specific products.
4. They must generate reproducible results.
5. They must not have any inherent scalability limitations.

In this chapter, we looked at several popular benchmarks for measuring the P&S of J2EE-based hardware and software platforms. The most important benchmark considered was SPECjAppServer2004 which is the state-of-the-art industry-standard benchmark for J2EE-based platforms. We looked at the way SPECjAppServer2004 evolved and discussed the problems that had to be addressed in order to make the benchmark meet the above requirements. SPECjAppServer2004 provides a *reliable* method for evaluating the P&S of J2EE-based platforms. It features the following additional benefits:

1. It has been tested on multiple hardware and software platforms and has proven to scale well from low-end desktop PCs to high-end servers and large clusters.
2. Once a platform has been tested and validated, the results can be used for multiple projects/applications. Official benchmark results are made publicly available and can be used free of charge.
3. Prior to publication, official benchmark results are reviewed by subject experts making sure that the benchmark was run correctly and results are valid.
4. Last but not least, the benchmark can be used as a sample (blueprint) application, demonstrating programming best practices and design patterns for building scalable applications.

In addition to considering concrete benchmarks, we presented several case studies that showed how benchmarks can be used to analyze the effect of application design alternatives, platform configuration options and tuning parameters on the overall system performance. In particular, we studied the performance difference between container-managed and bean-managed persistence and showed that in general container-managed persistence utilizes the container's caching services much better than bean-managed persistence. We showed how easily the data access layer of J2EE applications can become a system bottleneck and provided some guidelines on how such bottlenecks can be approached and eliminated. Several subtle issues in the design of ECperf were identified, degrading its scalability, reliability and performance. A redesign of the benchmark was proposed to address the discovered issues. The new design, demonstrated the performance gains and reliability benefits that asynchronous message-based processing provides over traditional request-reply processing. Finally, we studied how the performance of JBoss applications can be improved by exploiting different deployment and configuration options offered by the JBoss platform. Most of the options that we considered are available on other platforms and their effect on system performance does not depend significantly on the platform used. This was confirmed by experimentation and most of the conclusions from the study were generalized to other J2EE application servers.

The case study with ECperf, among other things, demonstrated that building on a scalable platform is not sufficient to ensure that a DCS is scalable, i.e. the system design must also be scalable. In the rest of this thesis, we develop a performance engineering framework for DCS that helps to identify P&S problems early in the development cycle and ensure that systems are designed and sized to meet their performance requirements. The framework exploits performance models for analyzing the system P&S. The next chapter introduces the different types of models that we are going to use.

Chapter 3

Performance Models

Every computer professional, from software developers to database administrators to network engineers to systems administrators, should master the basics of quantitative performance analysis.

Dr. Daniel A. Menascé
2001 A. A. Michaelson Award Recepient

3.1 Introduction

A number of different modeling formalisms have been developed in the past decades that can be used for modeling DCS. Queueing Networks (QNs) and Stochastic Petri Nets (SPNs) are perhaps the two most popular types of models that have been exploited in practical studies. However, as argued in [11], they both have some serious shortcomings. While QNs provide a very powerful mechanism for modeling hardware contention and scheduling strategies, they are not as suitable for representing blocking and synchronization of processes. SPNs, on the other hand, lend themselves well to modeling blocking and synchronization aspects, but have difficulty in representing scheduling strategies. Bause [11] has proposed a new modeling formalism called *Queueing Petri Nets (QPNs)* that combines QNs and SPNs into a single formalism and eliminates the above disadvantages. QPNs allow queues to be integrated into places of Petri nets. This enables the modeler to easily represent scheduling strategies and brings the benefits of QNs into the world of Petri nets.

This chapter provides a brief overview of the conventional queueing network and Petri net modeling formalisms and then introduces the QPN formalism discussing its advantages.

3.2 Queueing Networks

A *Queueing Network (QN)* consists of a set of interconnected queues. Each *queue* represents a service station that serves *requests* (also called *jobs*) sent by customers. A *service station* consists of one or more servers and a waiting area which holds requests waiting to be served. When a request arrives at a service station, its service begins immediately if a free server is available. Otherwise, the request is forced to wait in the waiting area or the service of another request is preempted in case the arriving request has a higher priority. The time between successive request arrivals is called *interarrival time*. Each request demands a certain amount of service, which is specified by the length of time a server is occupied serving it, i.e the *service time*. The *queueing delay* is the amount of time the request waits in the waiting area before its service begins. The *response time* is the total amount of time the request spends at the service station, i.e. the sum of the queueing delay and the service time.

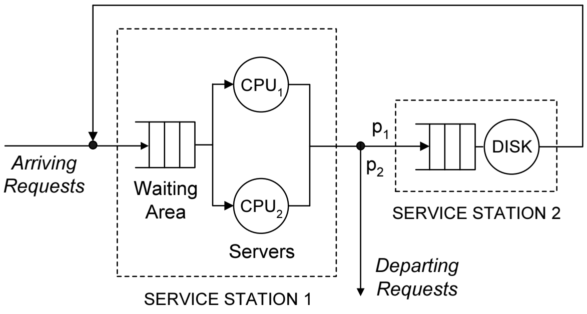


Figure 3.1: A basic QN.

Figure 3.1 shows a basic QN with two queues, i.e. service stations. Arriving requests first visit service station 1, which has two servers (representing CPUs). After requests are served by one of the servers, they move to service station 2 (representing a disk device) with probability p_1 or leave the network with probability p_2. Requests completing service at station 2 return back to station 1. The interconnection of queues in a QN is described by the paths requests may take which are specified by routing probabilities. A request might visit a service station multiple times while it circulates through the network. The total amount of service time required over all visits to the station is called *service demand* of the request at the station. Requests are usually grouped into *classes* with all requests in the same class having the same service demands. The algorithm which determines the order in which requests are served at a service station is called *scheduling strategy* (or scheduling/queueing/service discipline) [24]. Some typical scheduling strategies are:

- FCFS (First-Come-First-Served): Requests are served in the order in which they arrive. This strategy is typically used for queues representing I/O devices.
- LCFS (Last-Come-First-Served): The request that arrived last is served next.
- RR (Round-Robin): If the service of a request is not completed at the end of a time slice of specified length, the request is preempted and returns to the queue, which is served according to FCFS. This action is repeated until the request is completely served.
- PS (Processor-Sharing): All requests are assumed to be served simultaneously with the server speed being equally divided among them. This strategy corresponds to Round-Robin with infinitesimally small time slices and is typically used for modeling CPUs.
- IS (Infinite-Server): There is an ample number of servers so that no queue ever forms. Service stations with IS scheduling strategy are often called *delay resources* or *delay servers*.

There is a standard notation for describing queues, called *Kendall's notation*. Queues are described by six parameters as follows: $A/S/m/B/K/SD$ where

A stands for the request interarrival time distribution.

S stands for the request service time distribution.

m stands for the number of servers.

B stands for the capacity of the queue, i.e. the maximum number of requests that can be accommodated in the queue. If this argument is missing, then, by default, the queue capacity is assumed to be infinite.

K stands for the system population, i.e. the maximum number of requests that can arrive in the queue. If this argument is missing, then, by default, the system population is assumed to be infinite.

SD stands for the scheduling discipline.

The distributions of the request interarrival times and service times are generally denoted by a one-letter symbol as follows:

- M = Exponential (Markovian) distribution.
- D = Degenerate (or deterministic) distribution.

- E_k = Erlang distribution with parameter k.
- G = General distribution.

A deterministic distribution means that the times are constant and there is no variance. A general distribution means that the distribution is not known. If G is used for A, it is sometimes written GI. B is usually infinite or a variable, as is K. If B or K are assumed to be infinite for modeling purposes, they can be omitted from the notation (which they frequently are). However, if K is included, B must also be included to ensure that one is not confused between the two.

A QN in which requests arrive from an external source, get served in the network and then depart is said to be *open*, for e.g. the QN in Figure 3.1. A QN in which there is no external source of requests and no departures is said to be *closed*. In a closed QN it is assumed that for each request class the number of requests circulating in the network is constant. It is also possible that a QN is open for some request classes and closed for others in which case the QN is called *mixed*.

Typical measures of interest for a QN include queue lengths, response times, *throughput* (the number of requests served per unit of time) and *utilization* (the fraction of time that a service station is busy). A relationship, known as *Little's Law* [98], relates the throughput $\mathcal{X}$ of a service station with the average number of requests $\mathcal{N}$ in it and their average response time $\mathcal{R}$. The relation is shown in Equation (3.1) and holds under the condition that the service station is in *steady state*, i.e. the number of requests arriving per unit of time is equal to the number of those completing service.

$$\mathcal{N} = \mathcal{X}.\mathcal{R} \tag{3.1}$$

Another relationship that will be used often relates the service time $\mathcal{S}$ of requests at a station with its utilization $\mathcal{U}$ and throughput $\mathcal{X}$. This relationship is known as *Utilization Law* [51, 109] and is shown in Equation (3.2).

$$\mathcal{U} = \mathcal{X}.\mathcal{S} \tag{3.2}$$

QNs provide a very powerful mechanism for modeling *hardware contention* (contention for CPU time, disk access and other hardware resources). A number of efficient analysis methods have been developed for certain classes of QNs, which enable models of realistic size and complexity to be analyzed. However, QNs are not as suitable for modeling *software contention* (contention for processes, threads, database connections and other software resources), as well as blocking, synchronization, simultaneous resource possession and asynchronous processing. Even though extensions of QNs, such as *Extended QNs* [24], provide some limited support for modeling software contention and synchronization aspects, they are rather restrictive and inaccurate. For further details on QNs, the reader is referred to [24, 166].

3.3 Petri Nets

Petri nets were introduced in 1962 by Carl Adam Petri. An ordinary *Petri Net (PN)* (also called *Place-Transition Net*) is a bipartite directed graph composed of places, drawn as circles, and transitions, drawn as bars. A formal definition is given below [16]:

Definition 3.1 (PN) *An ordinary Petri Net (PN) is a 5-tuple* $PN = (P, T, I^-, I^+, M_0)$ *where:*

1. $P = \{p_1, p_2, ..., p_n\}$ *is a finite and non-empty set of* ***places****,*
2. $T = \{t_1, t_2, ..., t_m\}$ *is a finite and non-empty set of* ***transitions****,*
3. $P \cap T = \emptyset$*,*
4. $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ *are called backward and forward* ***incidence functions****, respectively,*
5. $M_0 : P \rightarrow \mathbb{N}_0$ *is called* ***initial marking****.*

The incidence functions I^- and I^+ specify the interconnections between places and transitions. If $I^-(p, t) > 0$, an arc leads from place p to transition t and place p is called an *input place* of the transition. If $I^+(p, t) > 0$, an arc leads from transition t to place p and place p is called an *output place* of the transition. The incidence functions assign natural numbers to arcs, which we call *weights* of the arcs. When each input place of transition t contains at least as many tokens as the weight of the arc connecting it to t, the transition is said to be *enabled*. An enabled transition may *fire*, in which case it destroys tokens from its input places and creates tokens in its output places. The amounts of tokens destroyed and created are specified by the arc weights. The initial arrangement of tokens in the net (called *marking*) is given by the function M_0, which specifies how many tokens are contained in each place. When a transition fires, the marking may change. Figure 3.2 illustrates this using a basic PN with 4 places and 2 transitions. The PN is shown before and after firing of transition t_1, which destroys one token from place p_1 and creates one token in place p_2.

Different extensions to ordinary PNs have been developed in order to increase the modeling convenience and the modeling power. *Colored PNs (CPNs)* introduced by K. Jensen [69] are one such extension. The latter allow a *type (color)* to be attached to a token. A color function C assigns a set of colors to each place, specifying the types of tokens that can reside in the place. In addition to introducing token colors, CPNs also allow transitions to fire in different *modes (transition colors)*. The color function C assigns a set of modes to each transition and incidence functions are defined on a per mode basis. A formal definition of a CPN follows [16]:

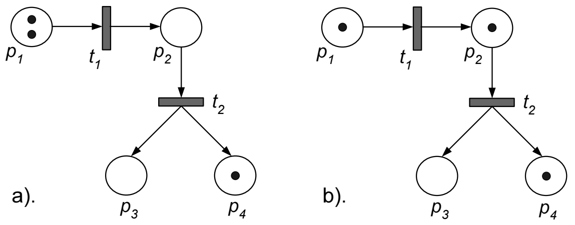


Figure 3.2: An ordinary PN before and after firing transition t_1.

Definition 3.2 (CPN) *A Colored PN (CPN) is a 6-tuple* $CPN = (P, T, C, I^-, I^+, M_0)$ *where:*

1. $P = \{p_1, p_2, ..., p_n\}$ *is a finite and non-empty set of* ***places****,*

2. $T = \{t_1, t_2, ..., t_m\}$ *is a finite and non-empty set of* ***transitions****,*

3. $P \cap T = \emptyset$*,*

4. C *is a* ***color function*** *defined from* $P \cup T$ *into finite and non-empty sets,*

5. I^- *and* I^+ *are the backward and forward* ***incidence functions*** *defined on* $P \times T$*, such that* $I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}]$, $\forall (p,t) \in P \times T$[1]

6. M_0 *is a function defined on* P *describing the* ***initial marking*** *such that* $M_0(p) \in C(p)_{MS}, \forall p \in P$.

Other extensions to ordinary PNs allow temporal (timing) aspects to be integrated into the net description [16]. In particular, *Stochastic PNs (SPNs)* attach an exponentially distributed *firing delay* to each transition, which specifies the time the transition waits after being enabled before it fires. *Generalized Stochastic PNs (GSPNs)* allow two types of transitions to be used: immediate and timed. Once enabled, immediate transitions fire in zero time. If several immediate transitions are enabled at the same time, the next transition to fire is chosen based on *firing weights* (probabilities) assigned to the transitions. Timed transitions fire after a random exponentially distributed firing delay as in the case of SPNs. The firing of immediate transitions always has priority over that of timed transitions. A formal definition of GSPNs follows [16]:

Definition 3.3 (GSPN) *A Generalized Stochastic PN (GSPN) is a 4-tuple* $GSPN = (PN, T_1, T_2, W)$ *where:*

[1]The subscript MS denotes multisets. $C(p)_{MS}$ denotes the set of all finite multisets of $C(p)$.

1. $PN = (P, T, I^-, I^+, M_0)$ is the underlying ordinary PN,

2. $T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$,

3. $T_2 \subset T$ is the set of immediate transitions, $T_1 \cap T_2 = \emptyset$, $T_1 \cup T_2 = T$,

4. $W = (w_1, ..., w_{|T|})$ is an array whose entry $w_i \in \mathbb{R}^+$

- *is a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay, if transition t_i is a timed transition* *<u>or</u>*
- *is a (possibly marking dependent) firing weight, if transition t_i is an immediate transition.*

Combining definitions 3.2 and 3.3, leads to *Colored GSPNs (CGSPNs)* [16]:

Definition 3.4 (CGSPN) *A Colored GSPN (CGSPN) is a 4-tuple $CGSPN = (CPN, T_1, T_2, W)$ where:*

1. $CPN = (P, T, C, I^-, I^+, M_0)$ is the underlying CPN,

2. $T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$,

3. $T_2 \subset T$ is the set of immediate transitions, $T_1 \cap T_2 = \emptyset$, $T_1 \cup T_2 = T$,

4. $W = (w_1, ..., w_{|T|})$ is an array whose entry $w_i \in [C(t_i) \longmapsto \mathbb{R}^+]$ such that $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$

- *is a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay due to color c, if $t_i \in T_1$ <u>or</u>*
- *is a (possibly marking dependent) firing weight specifying the relative firing frequency due to color c, if $t_i \in T_2$.*

PNs are a very powerful tool for both qualitative and quantitative system analysis. Unlike QNs, they easily lend themselves to modeling blocking and synchronization aspects. However, PNs have the disadvantage that they do not provide any means for direct representation of scheduling strategies [10]. The attempts to eliminate this disadvantage have led to the emergence of queueing PNs.

3.4 Queueing Petri Nets

We now give a brief introduction to QPNs, which are a combination of QNs and PNs. A deeper and more detailed treatment of the subject can be found in [10, 11, 12, 13, 16].

3.4.1 Basic Queueing Petri Nets

The QPN modeling formalism was introduced in 1993 by Falko Bause [11]. The main idea behind it was to add queueing and timing aspects to the places of CGSPNs. This is done by allowing queues (service stations) to be integrated into places of CGSPNs. A place of a CGSPN that has an integrated queue is called a *queueing place* and consists of two components, the *queue* and a *depository* for tokens which have completed their service at the queue. This is depicted in Figure 3.3.

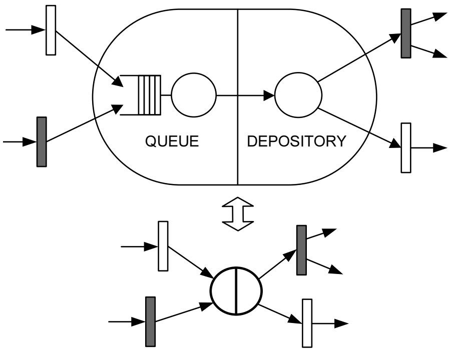


Figure 3.3: A queueing place and its shorthand notation.

The behavior of the net is as follows: tokens, when fired into a queueing place by any of its input transitions, are inserted into the queue according to the queue's scheduling strategy. Tokens in the queue are not available for output transitions of the place. After completion of its service, a token is immediately moved to the depository, where it becomes available for output transitions of the place. This type of queueing place is called *timed queueing place*. In addition to timed queueing places, QPNs also introduce *immediate queueing places*, which allow pure scheduling aspects to be described. Tokens in immediate queueing places can be viewed as being served immediately. Scheduling in such places has priority over scheduling/service in timed queueing places and firing of timed transitions. The rest of the net behaves like a normal CGSPN. An enabled timed transition fires after an exponentially distributed delay according to a race policy. Enabled immediate transitions fire according to relative firing frequencies and their firing has priority over that of timed transitions. We now give a formal definition of a QPN and then present an example of a QPN model.

Definition 3.5 (QPN) *A Queueing PN (QPN) is an 8-tuple* $QPN = (P, T, C, I^-, I^+, M_0, Q, W)$ *where:*

1. $CPN = (P, T, C, I^-, I^+, M_0)$ is the underlying Colored PN

2. $Q = (\tilde{Q}_1, \tilde{Q}_2, (q_1, ..., q_{|P|}))$ where

- *$\tilde{Q}_1 \subseteq P$ is the set of timed queueing places,*
- *$\tilde{Q}_2 \subseteq P$ is the set of immediate queueing places, $\tilde{Q}_1 \cap \tilde{Q}_2 = \emptyset$ and*
- *$(q_1, ..., q_{|P|})$ is an array whose entry q_i denotes the description of a queue taking all colors of $C(p_i)$ into consideration, if p_i is a queueing place <u>or</u> equals the keyword 'null', if p_i is an ordinary place.*

3. $W = (\tilde{W}_1, \tilde{W}_2, (w_1, ..., w_{|T|}))$ where

- *$\tilde{W}_1 \subseteq T$ is the set of timed transitions,*
- *$\tilde{W}_2 \subseteq T$ is the set of immediate transitions, $\tilde{W}_1 \cap \tilde{W}_2 = \emptyset$, $\tilde{W}_1 \cup \tilde{W}_2 = T$ and*
- *$(w_1, ..., w_{|T|})$ is an array whose entry $w_i \in [C(t_i) \longmapsto \mathbb{R}^+]$ such that $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$*
  - *is a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay due to color c, if $t_i \in \tilde{W}_1$ <u>or</u>*
  - *is a (possibly marking dependent) firing weight specifying the relative firing frequency due to color c, if $t_i \in \tilde{W}_2$.*

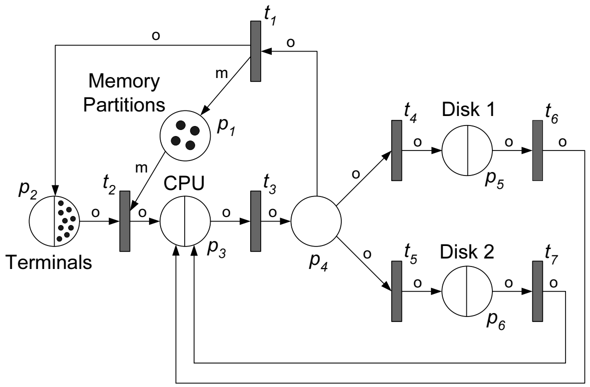


Figure 3.4: A QPN model of a central server with memory constraints (based on [16]).

Example 3.1 (QPN) *Figure 3.4 shows an example of a QPN model of a central server system with memory constraints based on [16]. Place p_2 represents several terminals, where users start jobs (modeled with tokens of color 'o') after a certain thinking time. These jobs request service at the CPU (represented by a G/C/1/PS queue, where C stands for Coxian distribution) and two disk subsystems (represented by G/C/1/FCFS queues). To enter the system each job has to allocate a certain amount of memory. The amount of memory needed by each job is assumed to be the same, which is represented by a token of color 'm' on place p_1. Note that, for readability, token cardinalities have been omitted from the arc weights in Figure 3.4, i.e. symbol **o** stands for **1'o** and symbol **m** for **1'm**. According to definition 3.5 we have the following:*

$QPN = (P, T, C, I^-, I^+, M_0, Q, W)$ *where*

- $CPN = (P, T, C, I^-, I^+, M_0)$ *is the underlying Colored PN as depicted in Figure 3.4,*
- $Q = (\tilde{Q}_1, \tilde{Q}_2,$
  $(null, G/C/\infty/IS, G/C/1/PS, null, G/C/1/FCFS, G/C/1/FCFS)),$
  $\tilde{Q}_1 = \{p_2, p_3, p_5, p_6\}, \tilde{Q}_2 = \emptyset,$
- $W = (\tilde{W}_1, \tilde{W}_2, (w_1, ..., w_{|T|}))$, *where* $\tilde{W}_1 = \emptyset, \tilde{W}_2 = T$ *and*
  $\forall c \in C(t_i) : w_i(c) := 1$, *so that all transition firings are equally likely.*

In [10] it is shown that QPNs have greater expressive power than QNs, extended QNs and SPNs. In addition to hardware contention and scheduling strategies, using QPNs one can easily model simultaneous resource possession, synchronization, blocking and software contention. This enables the integration of hardware and software aspects of system behavior into the same model [15]. While the above could also be achieved by using Layered QNs (LQNs) (or stochastic rendezvous networks) [99, 107, 137, 139, 174], the latter are defined at a higher-level of abstraction and are usually less detailed and accurate. Another benefit of QPNs is that, since they are based on Petri nets, one can exploit a number of efficient techniques from Petri net theory to verify some important qualitative properties of QPNs, such as ergodicity, boundedness, liveness or existence of home states. The latter not only help to gain insight into the behavior of QPNs, but are also essential preconditions for a successful quantitative analysis [11].

3.4.2 Hierarchical Queueing Petri Nets

A major hurdle to the practical application of QPNs is the so-called *largeness problem* or *state-space explosion problem*: as one increases the number of queues and tokens in a QPN, the size of the model's state space grows exponentially and quickly exceeds the capacity of today's computers. This imposes a limit on the size and complexity of the models that are analytically tractable. An attempt to alleviate this problem was the introduction of *Hierarchically-Combined QPNs (HQPNs)* [13]. The main idea is to allow hierarchical model specification and then exploit the hierarchical structure for efficient numerical analysis. This type of analysis is termed *structured analysis* and it allows models to be solved that are about an order of magnitude larger than those analyzable with conventional techniques.

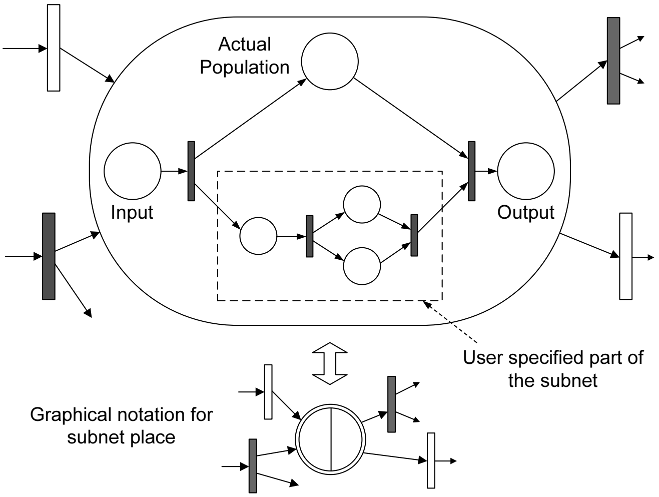

Actual Population
Input
Output
User specified part of the subnet
Graphical notation for subnet place

Figure 3.5: A subnet place and its shorthand notation.

HQPNs are a natural generalization of the original QPN formalism. In HQPNs a queueing place may contain a whole QPN instead of a single queue. Such a place is called a *subnet place* and is depicted in Figure 3.5. A subnet place might contain an ordinary QPN or again a HQPN allowing multiple levels of nesting. For simplicity, we restrict ourselves to two-level hierarchies. We use the term *High-Level QPN (HLQPN)* to refer to the upper level of the HQPN and the term *Low-Level QPN (LLQPN)* to refer to a subnet of the HLQPN. Every subnet of a HQPN has a dedicated input and output place, which are ordinary places of a CPN. Tokens being inserted into a subnet place after a transition firing are added to the input place of the corresponding HQPN subnet. The semantics of the output place of a subnet place is similar to the semantics of the depository of a queueing place: tokens in the output place are available for output transitions of the subnet place. Tokens contained in all other places of the HQPN subnet are not available for output transitions of the subnet place. Every HQPN subnet also contains *actual − population* place used to keep track of the total number of tokens fired into the subnet place.

3.5 Concluding Remarks

The QPN paradigm provides a number of benefits over conventional modeling paradigms such as QNs and SPNs. In addition to hardware contention and scheduling strategies, using QPNs one can easily model blocking, synchronization, simultaneous resource possession and software contention. As will be seen in the next chapter, this lends itself very well to modeling DCS. QPN models bring the following advantages:

1. They combine the modeling power and expressiveness of QNs and SPNs.
2. They allow the integration of hardware and software aspects of system behavior into the same model.
3. They have an intuitive graphical representation that facilitates model development.
4. They can be used to combine qualitative and quantitative system analysis. A number of efficient qualitative analysis techniques from Petri net theory can be exploited.

In the next chapter, we present some practical performance modeling case studies which show how performance models can be exploited for performance analysis of DCS.

Chapter 4

Performance Modeling Case Studies

The best way to learn a subject is to apply the concepts to a real system.
– *Dr. Raj Jain*

I hear, I forget; I see, I remember;
I do, I understand.
– *An Ancient Chinese Proverb*

4.1 Introduction

Performance models are a very powerful tool for performance analysis of DCS. They can be exploited during system development to predict the expected performance of the system under load. This helps to identify performance problems early in the system life cycle and have them resolved in time. However, in order for models to provide useful results, they must be representative. Building models that accurately capture the different aspects of system behavior becomes a more and more challenging task as systems grow in size and complexity. In this chapter, we present two practical performance modeling case studies which illustrate the difficulties that arise when trying to model a realistic DCS and predict its performance. The systems modeled are deployments of the SPECjAppServer2002 and SPECjAppServer2001 benchmarks introduced in Chapter 2.

In the first case study, a Queueing Network (QN) model of a SPECjAppServer2002 deployment is built. The model is analyzed using analytical techniques and used to predict the system performance under several different configurations. Results obtained are validated through measurements on the real system. The study illustrates the difficulties stemming from the limited expressiveness of QN models and the limitations in the available analysis techniques.

In the second case study, a Queueing Petri Net (QPN) model of a deployment of SPECjAppServer2001's order entry application is built. Again, the system performance is predicted for several different configurations and results are validated through measurements on the real system. The study demonstrates the modeling power and expressiveness of QPN models and shows how they can be used to integrate hardware and software aspects of system behavior. However, the study also shows that QPN models of realistic systems are too large to be analyzable using currently available tools and techniques for QPN analysis. New solution techniques and tools are needed which would enable models of realistic size and complexity to be analyzed. Only this would make it possible to exploit QPN models to their full potential.

4.2 Case Study 1: Modeling using QNs

In this section, we build and validate a QN model of a SPECjAppServer2002 deployment and then show how the model can be used to predict the system performance for the purpose of capacity planning. In the course of the study, we discuss the problems that arise from the limited model expressiveness, on the one hand, and from the system size and complexity, on the other hand. We propose different workaround solutions to these problems and illustrate them through practical examples.

4.2.1 Motivation

Imagine the following hypothetical scenario: A company is about to automate its internal and external business operations with the help of e-business technology. The company chooses to employ a J2EE-based platform and develops a J2EE application for supporting its order-inventory, supply chain and manufacturing operations. Let us assume that this application is SPECjAppServer2002 and that the company is testing the application in the deployment environment depicted in Figure 4.1. This environment uses a cluster of WebLogic Servers (WLS) as a J2EE container and an Oracle database server (DBS) for persistence. We assume that all machines in the WLS cluster are identical.

Before putting the application into production the company conducts a capacity planning study in order to come up with an adequate sizing and configuration of the deployment environment. More specifically, the company needs answers to the following questions:

- How many WebLogic servers would be needed to guarantee adequate performance under the expected workload?

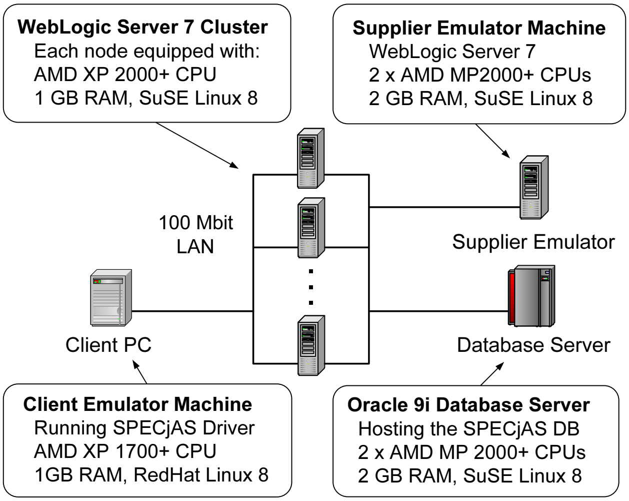


Figure 4.1: Deployment environment.

- For a given number of WebLogic servers, what level of performance would the system provide? What would be the average transaction throughput and response time? How utilized (CPU/Disk utilization) would be the WebLogic servers and the database server?
- Will the capacity of the database server suffice to handle the incoming load?
- Does the system scale or are there any other potential system bottlenecks?

We now build a QN model of the system and show how it can be used to address these questions.

4.2.2 Workload Characterization

The first step in the modeling process is to describe the workload of the system under study in a qualitative and quantitative manner. This is called *workload characterization* [108] and in its simplest form includes four major steps:

1. Describe the types of requests that are processed by the system (the *request classes*).
2. Identify the system resources used by each request class.

3. Measure the total amount of service time (the *service demand*) for each request class at each resource.

4. Specify the number of requests of each class that the system will be exposed to (the *workload intensity*).

As discussed in Chapter 2, the SPECjAppServer2002 workload, which is identical to ECperf, is made up of two major components - the *order entry application* in the customer domain and the *manufacturing application* in the manufacturing domain. The order entry application is processing the following four transaction types:

1. *NewOrder*: places a new order in the system.

2. *ChangeOrder*: modifies an existing order.

3. *OrderStatus*: retrieves the status of a given order.

4. *CustStatus*: lists all orders of a given customer.

We map each of them to a separate request class in our workload model. The manufacturing application, on the other hand, is running production lines. The main unit of work there is a *work order*. Each work order produces a specific quantity of a particular type of widget. There are two types of production lines: planned lines and large order lines. While planned lines run on a predefined schedule, large order lines run only when a large order arrives in the customer domain. Each large order results in a separate work order. During the processing of work orders multiple transactions are executed in the manufacturing domain, i.e. scheduleWorkOrder, updateWorkOrder and completeWorkOrder. Each work order moves along three virtual production line stations, which represent distinct operations in the manufacturing flow. In order to simulate activity at the stations, the manufacturing application waits for a designated time at each station.

One way to model the manufacturing workload would be to define a separate request class for each transaction run during the processing of work orders. However, this would lead to an overly complex model and would limit the range of analysis techniques that would be applicable for its solution. Second, it would not be of much benefit, since after all, what most interests us is the rate at which work orders are processed and not the performance metrics of the individual work-order-related transactions. Therefore, we model the manufacturing workload at the level of work orders. We define a single request class *WorkOrder*, which represents a request for processing a work order. This keeps our model simple and as will be seen later is enough to provide sufficient information about the behavior of the manufacturing application.

Altogether, we end up with five request classes: NewOrder (NO), ChangeOrder (CO), OrderStatus (OS), CustStatus (CS) and WorkOrder (WO). The following resources are used during their processing:

- The CPU of a WebLogic server (WLS-CPU).
- The Local Area Network (LAN).
- The CPUs of the database server (DBS-CPU).
- The disk subsystem of the database server (DBS-I/O).

In order to determine the service demands at these resources, we conducted a separate experiment for each of the five request classes. In each case, we deployed the benchmark in a configuration with a single WebLogic server and then injected requests of the respective class into the system. During the experiment, we monitored the system resources and measured the time requests spent at each resource during their processing. For the database server, we used the Oracle 9i Intelligent Agent, which provides exhaustive information about CPU consumption as well as I/O wait times. For the WebLogic server, we monitored the CPU utilization using operating system tools and then used the *Service Demand Law* [51, 108] to derive the CPU service demand: the service demand $\mathcal{D}$ of requests at a given resource is equal to the average resource utilization $\mathcal{U}$ divided by the average request throughput $\mathcal{X}$, during the measurement interval (assuming that requests of just one type are processed during the experiment), i.e.

$$\mathcal{D} = \frac{\mathcal{U}}{\mathcal{X}} \tag{4.1}$$

We decided we could safely ignore network service demands, since all communications were taking place over a 100 MBit LAN and communication times were negligible. Table 4.1 reports the service demand measurements for the five request classes in the workload model. Figure 4.2 summarizes these measurements in graphical form.

As we can see from Table 4.1, database I/O service demands are much lower than CPU service demands. This stems from the fact that data is cached in the database buffer and disks are usually accessed only when updating or inserting new data. However, even in this case the I/O overhead is minimal since the only thing that is done is to flush the database log buffer, which is performed with sequential I/O accesses. Here we would like to point out that SPECjAppServer2002 and its predecessor benchmarks ECperf and SPECjAppServer2001 use relatively small data volumes for the workload intensities generated. While this keeps the I/O overhead low, as discussed in Chapter 2, it causes increased data contention in the database. As will be seen later, the latter poses some difficulties in predicting transaction

Table 4.1: Request mean service demands.

TX-Type	WLS-CPU	DBS-CPU	DBS-I/O
NewOrder	12.98ms	10.64ms	1.12ms
ChangeOrder	13.64ms	10.36ms	1.27ms
OrderStatus	2.64ms	2.48ms	0.58ms
CustStatus	2.54ms	2.08ms	0.3ms
WorkOrder	24.22ms	34.14ms	1.68ms

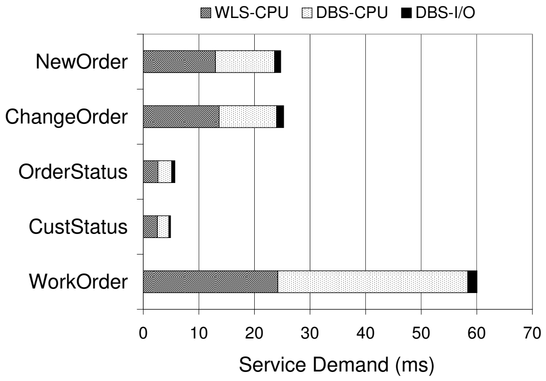


Figure 4.2: Request mean service demands.

response times since data contention does not easily lend itself to analytical modeling using conventional techniques.

Now that we know the service demands of the different request classes, we proceed with the last step in the workload characterization, which aims to quantify the workload intensity. For each request class, we must specify the rates at which requests arrive. We should also be able to vary these rates so that we can consider different scenarios. To this end, we modified the SPECjAppServer2002 driver to allow more flexibility in configuring the intensity of the workload generated. Specifically, the new driver allows us to set the number of concurrent order entry clients emulated, as well as their average *think time*, i.e. the time they "think"after receiving a response from the system before they send the next request. In addition to this, we can specify the number of planned production lines run in the manufacturing domain and the time they wait after processing a work order before starting a new one. In this way, we can precisely define the workload intensity and transaction mix. We will study in detail several different scenarios under different transaction mixes and workload intensities.

4.2.3 Building a Performance Model

We are now ready to build a QN model of the SPECjAppServer2002 deployment. We first define the model in a general fashion and then customize it to some concrete workload scenarios. We use a closed model, which means that for each instance of the model the number of concurrent clients sending requests to the system is fixed. Figure 4.3 shows a high-level view of our QN model. Table 4.2 presents a formal specification of the model queues (i.e. queueing stations) in Kendall's notation. With queues defined in this way, we end up with a non-product-form QN. Note that we could have chosen to model the CPUs of the database server using a single `G/M/2/PS` queue instead of two separate `G/M/1/PS` queues. However, many efficient analysis techniques for non-product-form QNs do not support `G/M/m/PS` queues and therefore we chose the first option so that we have more flexibility when analyzing the model.

Following is a brief description of the queues used:

C : IS queue (delay resource) used to model the client machine which runs the SPECjAppServer driver and emulates virtual clients sending requests to the system. The service time of order entry requests at this queue is equal to the average client think time, while the service time of WorkOrder requests is equal to the average time a production line waits after processing a work order before starting a new one. Note that times spent in this queue are not part of system response times.

$A_1..A_N$: PS queues used to model the CPUs of the N WebLogic servers.

B_1, B_2 : PS queues used to model the two CPUs of the database server.

D : FCFS queue used to model the disk subsystem (made up of a single 100GB disk drive) of the database server.

L : IS queue (delay resource) used to model the virtual production line stations in the manufacturing domain. Only WorkOrder requests ever visit this queue. Their service time at the queue corresponds to the average delay at the production line stations simulated by the manufacturing application during work order processing.

The model is a multi-class QN with five request classes as defined in the previous section. The behavior of requests in the QN is defined by specifying their respective routing probabilities p_i and service demands at each queue which they

Table 4.2: Formal queue definitions.

Queue	Type	Description
$A_1..A_N$	$G/M/1/PS$	WLS CPUs
B_1, B_2	$G/M/1/PS$	DBS CPUs
D	$G/M/1/FCFS$	DBS Disk Subsystem
C	$G/M/\infty/IS$	Client Machine
L	$G/M/\infty/IS$	Prod. Line Stations

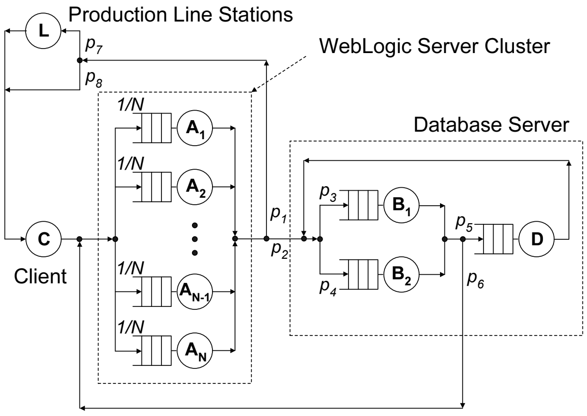


Figure 4.3: QN model of the system.

visit. We discussed the service demands in the previous section. To set the routing probabilities we examine the life-cycle of client requests in the QN. Every request is initially at the client queue C, where it waits for a user-specified think time. After the think time elapses, the request is routed to a randomly chosen queue A_i, where it queues to receive service at a WebLogic server CPU. We assume that requests are evenly distributed over the N WebLogic servers, i.e. each server is chosen with probability $1/N$. Processing at the CPU may be interrupted multiple times if the request requires some database accesses. Each time this happens, the request is routed to the database server where it queues for service at one of the two CPU queues B_1 or B_2 (each chosen equally likely so that $p_3 = p_4 = 0.5$). Processing at the database CPUs may be interrupted in case I/O accesses are needed. For each I/O access the request is sent to the disk subsystem queue D and after receiving service there, it is routed back to the database CPUs. This may be repeated multiple times depending on routing probabilities p_5 and p_6. Having completed their service at the database server, requests are sent back to the WebLogic server. Requests may visit the database server multiple times during their processing, depending on routing probabilities p_1 and p_2. After completing service at the WebLogic server, requests are sent back to the client queue C. Order entry requests are sent directly to the client queue (for them $p_8 = 1$, $p_7 = 0$), while WorkOrder requests are routed through queue L (for them $p_8 = 0$, $p_7 = 1$), where they are additionally delayed for 1 second. This delay corresponds to the 1 second delay at the three production line stations imposed by the manufacturing application during work order processing.

In order to set routing probabilities p_1, p_2, p_5 and p_6 we need to know how many times a request visits the database server during its processing and for each visit how many times I/O access is needed. Since we only know the total service demands over all visits to the database, we assume that requests visit the database just once and need a single I/O access during this visit. This allows us to drop routing probabilities p_1, p_2, p_5 and p_6 and results in a simplified model depicted in Figure 4.4.

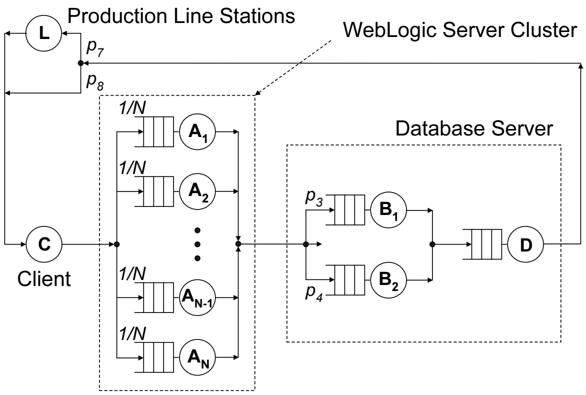


Figure 4.4: Simplified QN model of the system.

The following input parameters need to be supplied before the model can be analyzed:

- Number of order entry clients (NewOrder, ChangeOrder, OrderStatus and CustStatus).
- Average think time of order entry clients - *customer think time*.

- Number of planned production lines processing WorkOrder requests.
- Average time production lines wait after processing a work order, before starting a new one - *manufacturing (mfg) think time.*
- Service demands of the five request classes at queues A_i, B_j, and D (as per Table 4.1).

We consider two types of deployment scenarios. In the first one, large order lines in the manufacturing domain are turned off. In the second one, they are running as defined in the benchmark workload. The reason for this separation is that large order lines introduce some *asynchronous processing*, which in general is hard to model using QNs. We start with the simpler case where we do not have such processing and then we show how the large order processing can be integrated into the model.

4.2.4 Model Analysis and Validation

We now proceed to analyze several different instances of the QN model and then validate them by comparing results from the analysis with measured data. We first consider the case without large order lines and study the system in three scenarios representing low, moderate and heavy load, respectively. In each case, we examine deployments with different number of WebLogic servers - from 1 to 9. Table 4.3 summarizes the input parameters for the three scenarios that we consider.

Table 4.3: Model input parameters for the 3 scenarios.

Parameter	Low	Moderate	Heavy
NewOrder Clients	30	50	100
ChangeOrder Clients	10	40	50
OrderStatus Clients	50	100	150
CustStatus Clients	40	70	50
Planned Lines	50	100	200
Customer Think Time	2 sec	2 sec	3 sec
Mfg Think Time	3 sec	3 sec	5 sec

Scenario 1: Low Load

A number of analysis tools for QNs have been developed and are available free of charge for noncommercial use (see for e.g. [25, 63, 71, 74]). We employed the

PEPSY-QNS tool [25] (Performance Evaluation and Prediction SYstem for Queueing NetworkS) from the University of Erlangen-Nuernberg. We chose PEPSY because it supports a wide range of solution methods (over 30) for product- and non-product-form QNs. Both exact and approximate methods are provided, which are applicable to models of considerable size and complexity. We used the *multisum method* [23] to solve our QN model. However, to ensure plausibility of the results, we cross-verified them with results obtained from other methods such as *bol_aky* and *num_app* [25]. In all cases the difference was negligible.

Table 4.4 summarizes the results we obtained for the first scenario. We studied two different configurations - the first one with 1 WebLogic server, the second one with 2. The table reports throughput (X) and response time (R) for the five request classes, as well as CPU utilization (U) of the WebLogic servers (U_{WLS}) and the database server (U_{DBS}). Results obtained from the model analysis are compared against results obtained through measurements and the modeling error is reported.

Table 4.4: Analysis results for scenario 1 (low load).

	1 WebLogic Server			2 WebLogic Servers		
METRIC	Model	Measured	Error	Model	Measured	Error
X_{NO}	14.59	14.37	1.5%	14.72	14.49	1.6%
X_{CO}	4.85	4.76	1.9%	4.90	4.82	1.7%
X_{OS}	24.84	24.76	0.3%	24.89	24.88	0.0%
X_{CS}	19.89	19.85	0.2%	19.92	19.99	0.4%
X_{WO}	12.11	12.19	0.7%	12.20	12.02	1.5%
R_{NO}	56ms	68ms	17.6%	37ms	47ms	21.3%
R_{CO}	58ms	67ms	13.4%	38ms	46ms	17.4%
R_{OS}	12ms	16ms	25.0%	8ms	10ms	20.0%
R_{CS}	11ms	17ms	35.2%	7ms	10ms	30.0%
R_{WO}	1127ms	1141ms	1.2%	1092ms	1103ms	1.0%
U_{WLS}	66%	70%	5.7%	33%	37%	10.8%
U_{DBS}	36%	40%	10%	36%	38%	5.2%

As we can see from the table, while throughput and utilization results are extremely accurate, the same does not hold for response time results. This is because when we run a transaction mix, as opposed to a single transaction, some additional delays are incurred which are not captured by the model. For example, delays result from contention for data access (database locks, latches), processes, threads, database connections, etc. Our model captures the hardware contention aspects of system behavior and does not represent software contention aspects. While software contention may not always have a big impact on transaction throughput and CPU utilization, it usually does have a direct impact on transaction response time and therefore real (measured) response times are higher than the ones obtained from the model. In [107] and [139] some techniques are presented for estimating delays incurred from software contention. However, they are rather approximative and attempting to apply them to our system leads to technical difficulties stemming from the size and complexity of the latter.

From Table 4.4 we see that the response time error for requests with very low service demands (e.g. OrderStatus and CustStatus) is much higher than average. This is because the processing times for such requests are very low (around 10ms) and the additional delays from software contention, while not that high as absolute values, are high relative to the overall response times. The results show that the higher the service demand of a request type, the lower the response time error is. Indeed, the requests with the highest service demand (WorkOrder) always have the lowest response time error.

Scenario 2: Moderate Load

In this scenario there are 260 concurrent clients interacting with the system and 100 planned production lines running in the manufacturing domain. This is twice as much as in the previous scenario. We study 2 deployments - the first one with 3 WebLogic servers, the second with 6. Table 4.5 summarizes the results from the model analysis. Again we obtain very accurate throughput and utilization results, and accurate response time results. The response time error does not exceed 35%, which is considered acceptable in most capacity planning studies [109].

Scenario 3: Heavy Load

In this scenario there are 350 concurrent clients and 200 planned production lines in total. We consider three configurations - with 4, 6 and 9 WebLogic servers, respectively. However, we slightly increase the think times in order to make sure that the single database server is able to handle the load. Tables 4.6 and 4.7 summarize the results for this scenario. For the configuration with 9 WebLogic servers, the available model analysis algorithms failed to produce reliable response time results and therefore we only consider throughput and utilization for this configuration.

Scenarios with Large Order Lines

We now consider the case when large order lines in the manufacturing domain are enabled. The latter are activated upon arrival of large orders in the customer domain. Each large order generates a work order that is processed asynchronously on one of the large order lines. As already mentioned, this poses a difficulty since QNs

Table 4.5: Analysis results for scenario 2 (moderate load).

	3 WebLogic Servers			6 WebLogic Servers		
METRIC	Model	Measured	Error	Model	Measured	Error
X_{NO}	24.21	24.08	0.5%	24.29	24.01	1.2%
X_{CO}	19.36	18.77	3.1%	19.43	19.32	0.6%
X_{OS}	49.63	49.48	0.3%	49.66	49.01	1.3%
X_{CS}	34.77	34.24	1.5%	34.80	34.58	0.6%
X_{WO}	23.95	23.99	0.2%	24.02	24.03	0.0%
R_{NO}	65ms	75ms	13.3%	58ms	68ms	14.7%
R_{CO}	66ms	73ms	9.6%	58ms	70ms	17.1%
R_{OS}	15ms	20ms	25.0%	13ms	18ms	27.8%
R_{CS}	13ms	20ms	35.0%	11ms	17ms	35.3%
R_{WO}	1175ms	1164ms	0.9%	1163ms	1162ms	0.0%
U_{WLS}	46%	49%	6.1%	23%	25%	8.0%
U_{DBS}	74%	76%	2.6%	73%	78%	6.4%

Table 4.6: Analysis results for scenario 3 (heavy load) with 4 and 6 WLS.

	4 WebLogic Servers			6 WebLogic Servers		
METRIC	Model	Msrd.	Error	Model	Msrd.	Error
X_{NO}	32.19	32.29	0.3%	32.22	32.66	1.3%
X_{CO}	16.10	15.96	0.9%	16.11	16.19	0.5%
X_{OS}	49.59	48.92	1.4%	49.60	49.21	0.8%
X_{CS}	16.55	16.25	1.8%	16.55	16.24	1.9%
X_{WO}	31.69	31.64	0.2%	31.72	32.08	1.1%
R_{NO}	106ms	98ms	8.2%	103ms	94ms	9.6%
R_{CO}	106ms	102ms	3.9%	102ms	98ms	4.1%
R_{OS}	25ms	30ms	16.7%	24ms	27ms	11.1%
R_{CS}	21ms	31ms	32.3%	20ms	27ms	25.9%
R_{WO}	1310	1260ms	4.0%	1305ms	1251ms	4.3%
U_{WLS}	40%	42%	4.8%	26%	29%	10.3%
U_{DBS}	87%	89%	2.2%	88%	91%	3.3%

provide very limited possibilities for modeling this type of asynchronous processing. We now present one approach to integrate large order processing into the model.

Table 4.7: Analysis results for scenario 3 (heavy load) with 9 WLS.

	9 WebLogic Servers		
METRIC	Model	Msrd.	Error
X_{NO}	32.24	32.48	0.7%
X_{CO}	16.12	16.18	0.4%
X_{OS}	49.61	49.28	0.7%
X_{CS}	16.55	16.46	0.5%
X_{WO}	31.73	32.30	1.8%
U_{WLS}	18%	20%	10.0%
U_{DBS}	88%	91%	3.3%

Since large order lines are always triggered by NewOrder transactions we can add the load they produce to the service demands of NewOrder requests. To do that we rerun the NewOrder experiments with the large order lines turned on. The additional load leads to higher utilization of system resources and in this way impacts the measured NewOrder service demands (see Table 4.8). While this incorporates the large order line activity into the model, it changes the semantics of NewOrder jobs. In addition to the NewOrder transaction load, they now also include the load caused by large order lines. Thus, performance metrics (throughput, response time) for NewOrder requests no longer correspond to the respective metrics of the NewOrder transaction. Therefore, we can no longer quantify the performance of the NewOrder transaction on itself. Nevertheless, we can still analyze the performance of other transactions and gain a picture of the overall system behavior. Table 4.9 summarizes the results for the three scenarios with enabled large order lines. We consider one configuration per scenario: the first one with 1 WebLogic server, the second with 3 and the third with 9. As we can see, while the model predictions for transaction throughput and server utilization were very accurate, unfortunately this was not the case for response times. The incorporation of large order lines into the model led to the modeling error for response time soaring up to 75%.

Table 4.8: NewOrder service demands with large order lines running.

TX-Type	WLS-CPU	DBS-CPU	DBS-I/O
NewOrder	23.49ms	21.61ms	1.87ms

Table 4.9: Analysis results for scenarios with large order lines.

	Low / 1 WLS		Moderate / 3 WLS		Heavy / 9 WLS	
METRIC	Model	Error	Model	Error	Model	Error
X_{CO}	4.79	6.4%	19.09	3.5%	15.31	4.5%
X_{OS}	24.77	2.9%	49.46	2.3%	48.96	3.1%
X_{CS}	19.83	2.4%	34.67	2.1%	16.37	1.9%
X_{WO}	11.96	5.7%	23.43	2.6%	29.19	1.2%
R_{CO}	86ms	60.7%	95ms	34.5%	-	-
R_{OS}	18ms	71.0%	22ms	55.1%	-	-
R_{CS}	16ms	74.6%	19ms	59.6%	-	-
R_{WO}	1179ms	16.1%	1268ms	5.0%	-	-
U_{WLS}	80%	0.0%	53%	1.9%	20%	0.0%
U_{DBS}	43%	2.4%	84%	2.4%	96%	1.0%

4.2.5 Conclusions from the Analysis

We used the QN model to predict system performance in several different configurations varying the workload intensity and the number of WebLogic servers available. The model was extremely accurate in predicting transaction throughput and server utilization, and less accurate in predicting transaction response time. The results enable us to give answers to the questions we started with in Section 4.2.1. Depending on the service level agreements and the expected workload intensity, we can now determine how many WebLogic servers we need in order to guarantee adequate performance. We can also see for each configuration which component is mostly utilized and thus could become a potential bottleneck (see Figure 4.5). In scenario 1, we saw that with a single WebLogic server, the latter could easily become a bottleneck since its utilization would be twice as high as that of the database server. The problem was resolved by adding an extra WebLogic server. In scenarios 2 and 3, we saw that with more than 3 WebLogic servers as we increase the load, the database CPU utilization approaches 90%, while the WebLogic servers remain in all cases less than 50% utilized. This clearly indicates that, in this case, the database server is the bottleneck.

4.3 Case Study 2: Modeling using QPNs

In this section, we build a QPN model of a deployment of SPECjAppServer2001's order entry application and then, as in the first case study, we show how the model

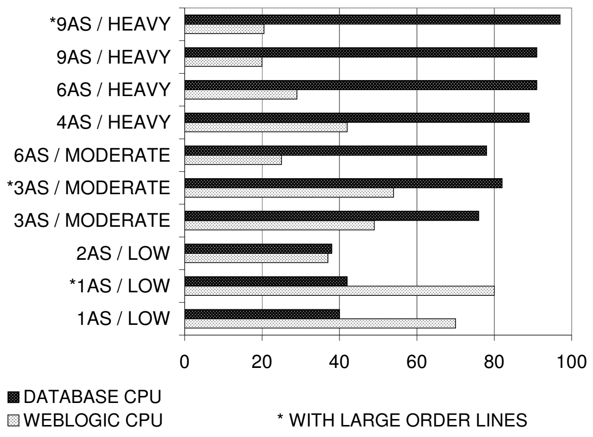


Figure 4.5: Server utilization in different scenarios.

can be used to predict the system performance for the purpose of capacity planning. We demonstrate the modeling power and expressiveness of QPN models and show how they can be used to integrate hardware and software aspects of system behavior. However, we also show that QPN models of realistic systems are too large to be analyzable using currently available tools and techniques for QPN analysis. New solution techniques and tools are needed which would enable models of realistic size and complexity to be analyzed. Only this would make it possible to exploit QPN models to their full potential.

4.3.1 Motivation

Imagine the following hypothetical scenario: A company is about to introduce an online ordering service for its customers and chooses to implement the service using a J2EE application. Assume that this application is the order entry application of SPECjAppServer2001. Before putting the application into production, the company decides to conduct a capacity planning study in order to come up with an adequate sizing and configuration of the deployment environment. We assume that the company initially plans to deploy the application in the deployment environment depicted in Figure 4.6. This environment uses a cluster of WebLogic servers (WLS) as a container for the J2EE application and Oracle 9i as a database server (DBS) for persistence. We assume that all machines in the WLS cluster are identical.

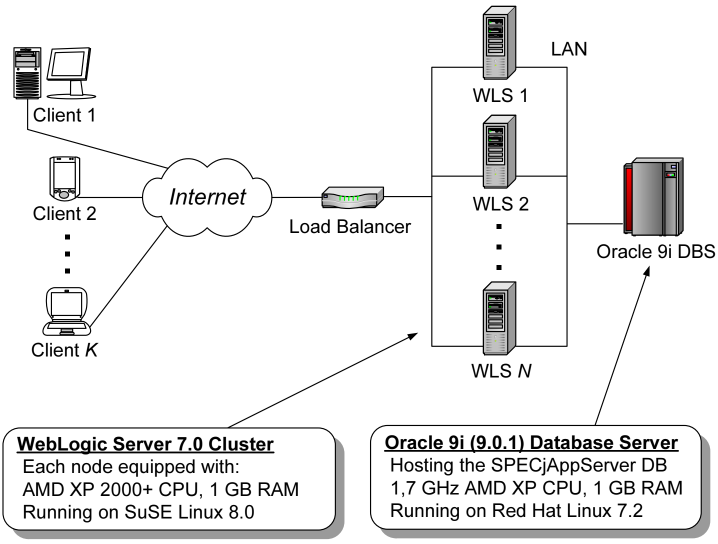

LAN
Client 1
Client 2
Client K
Internet
Load Balancer
WLS 1
WLS 2
WLS N
Oracle 9i DBS
WebLogic Server 7.0 Cluster
Each node equipped with:
AMD XP 2000+ CPU, 1 GB RAM
Running on SuSE Linux 8.0
Oracle 9i (9.0.1) Database Server
Hosting the SPECjAppServer DB
1,7 GHz AMD XP CPU, 1 GB RAM
Running on Red Hat Linux 7.2

Figure 4.6: Deployment environment for order entry application.

The company is interested in finding answers to the following questions:

1. What level of performance does the system provide under load?
2. Are there potential system bottlenecks? Does the system scale?
3. How many WebLogic servers would be needed to guarantee adequate performance?

In addition, the company needs to find optimal values for the following configuration parameters:

1. Number of threads in WLS thread pools.
2. Number of JDBC connections in WLS database connection pools.
3. Number of shared server processes of the Oracle server instance.

We now build a QPN model of the system and show how it can be used to address these questions.

4.3.2 Workload Characterization

As in the previous case study, we first characterize the workload of the system. Since, this time only the order entry application is run, we have four request classes corresponding to the four order entry transaction types: NewOrder (NO), ChangeOrder (CO), OrderStatus (OS) and CustStatus (CS). The following resources are used during their processing:

- The CPU of a WebLogic server (WLS-CPU).
- The Local Area Network (LAN).
- The CPUs of the database server (DBS-CPU).
- The disk subsystem of the database server (DBS-I/O).

Again, in order to determine the service demands at these resources, we conducted a separate experiment for each of the four request classes. In each case, we deployed the order entry application in a configuration with a single WebLogic server and then injected requests of the respective class into the system. During the experiment, we monitored the system resources and measured the time requests spent at each resource during their processing. For the database server, we used the Oracle 9i Intelligent Agent, which provides exhaustive information about CPU consumption as well as I/O wait times. For the WebLogic server, we monitored the CPU utilization using operating system tools and then used the Service Demand Law to derive the CPU service demand. As to network service demands, we decided to ignore them since all communications were taking place over a 100 MBit LAN and communication times were negligible. Table 4.10 reports the service demand measurements for the four request classes. In addition to hardware resources, for each transaction, a WLS thread, a database connection and a database server process are used during its processing.

Table 4.10: Workload mean service demands.

TX-Type	WLS-CPU	DBS-CPU	DBS-I/O
NewOrder	70ms	53ms	12ms
ChangeOrder	26ms	16ms	6ms
OrderStatus	7ms	4ms	0ms
CustomerStatus	10ms	5ms	0ms

4.3.3 First Cut System Model

We are now ready to build a QPN model of the system. We start with a simple model that does not utilize any hierarchical structures and is depicted in Figure 4.7. For now we assume that there is a single WebLogic server in the WLS cluster.

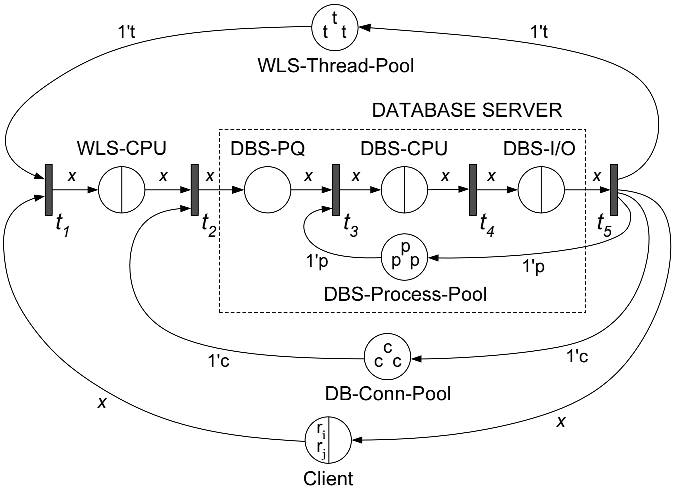


Figure 4.7: Flat QPN system model.

The following types of tokens (token colors) are used in the model:

Token 'r_i' represents a request sent by a client for execution of a transaction of class i. For each request class a separate token color is used (e.g. 'r_1', 'r_2', 'r_3',...). Tokens of these colors can be contained only in places Client, WLS-CPU, DBS-PQ, DBS-CPU and DBS-I/O.

Token 't' represents a WLS thread. Tokens of this color can be contained only in place WLS-Thread-Pool.

Token 'p' represents a DBS process. Tokens of this color can be contained only in place DBS-Process-Pool.

Token 'c' represents a JDBC connection to the DBS. Tokens of this color can be contained only in place DB-Conn-Pool.

In the following we describe the places of the model:

Client Queueing place with IS scheduling strategy used to represent clients sending requests to the system. The service time of this place corresponds to the average client think time.

WLS-CPU Queueing place with PS scheduling strategy used to represent the CPU of the WLS.

DBS-CPU Queueing place with PS scheduling strategy used to represent the CPU of the DBS.

DBS-I/O Queueing place with FCFS scheduling strategy used to represent the disk subsystem of the DBS.

WLS-Thread-Pool Ordinary place used to represent the thread pool of the WLS. Each token in this place represents a WLS thread.

DB-Conn-Pool Ordinary place used to represent the JDBC connection pool of the WLS. Tokens in this place represent JDBC connections to the DBS.

DBS-Process-Pool Ordinary place used to represent the process pool of the DBS. Tokens in this place represent Oracle processes.

DBS-PQ Ordinary place used to hold incoming requests at the DBS while they wait for a server process to be allocated to them.

We now take a look at the life-cycle of a client request in the system model. Every request (modeled by a token of color 'r_i' for some i) is initially at the queue of place Client where it waits for a user-specified think time. After the think time elapses the request moves to the Client depository and waits for a WLS thread to be allocated to it before its processing can start. Once a thread is allocated (modeled by taking a token of color 't' from place WLS-Thread-Pool), the request moves to the queue of place WLS-CPU, where it receives service from the CPU of WLS. It then moves to the depository of the place and waits for a JDBC connection to be allocated to it. The JDBC connection (modeled by token 'c') is used to contact the database and make any updates required by the respective transaction. A request sent to the database server arrives at place DBS-PQ (DBS Process Queue) and waits for a server process (modeled by token 'p') to be allocated to it. Once this is done, the request receives service first at the CPU and then at the disk subsystem of the database server. This completes the processing of the request, which is then sent back to place Client releasing the held DBS process, JDBC connection and WLS thread. The following input parameters need to be supplied before the model can be analyzed:

- Number of requests of each request class in the initial marking.

- Service times of request classes at the queues of places WLS-CPU, DBS-CPU and DBS-I/O (as per Table 4.10). Service times are assumed to be exponentially distributed.
- Average client think time (service time at the queue of place Client).
- Number of WLS threads (tokens 't'), JDBC connections (tokens 'c') and Oracle server processes (tokens 'p') in the initial marking.

4.3.4 Hierarchical System Model

The model described above is a flat QPN model and its corresponding state space grows exponentially with the number of requests in the system. This means that only relatively small instances of the model (i.e. with relatively small number of requests) are analytically tractable. We will now show how hierarchical structuring can be exploited to alleviate the state space explosion problem and enable larger models to be analyzed. The idea is to isolate the database server and model it using a separate nested QPN. In order to do this we replace the database server part of the original flat QPN with a single subnet place which we call "DBS". This place represents the entire database server and is expanded into a low-level QPN containing the original DBS queues of the flat model. Figures 4.8 and 4.9 show the high-level and low-level QPN of the new system model. By specifying the model in a hierarchical fashion we can now exploit structured analysis techniques, which enables us to solve models with much higher number of requests.

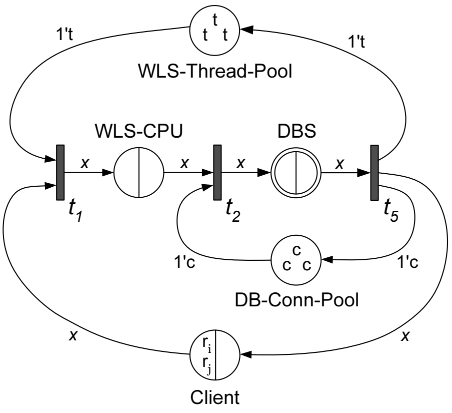


Figure 4.8: Model's high-level QPN.

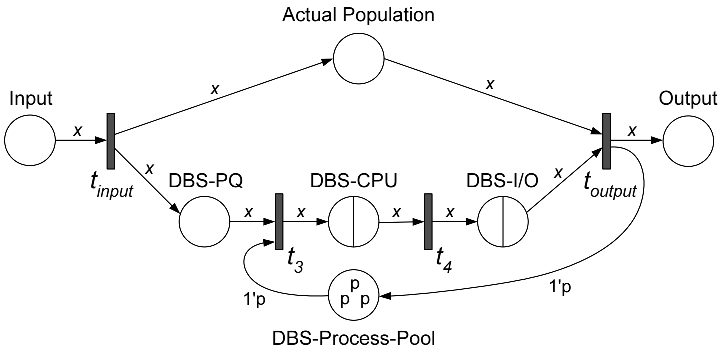


Figure 4.9: Model's low-level QPN.

4.3.5 Model Analysis and Validation

We now proceed to analyze several different instances of the model introduced in the previous section and then validate them by comparing results from the analysis with measured data. We start by taking a look at the mathematical laws and formulas that we use in our analysis. Our presentation is based on the following notation:

$\mathcal{D}$: *Service demand* of a queue, i.e. the amount of time required for a token (request) to be served at the queue.

μ : Average *service rate* of a queue, i.e. the number of tokens (requests) served at the queue per unit of time.

$\mathcal{N}$: Average *token population* of a queue, place or depository, i.e. the average number of tokens (requests) in it.

$\mathcal{U}$: Average *utilization* of a queue, place or depository, i.e. the probability that there is a token (request) in it.

$\mathcal{X}$: Average *throughput* of a queue, place or depository, i.e. the number of tokens (requests) that leave it per unit of time. Note that, at steady state, the rate at which tokens leave a queue/place/depository is equal to the rate at which they enter it.

$\mathcal{R}$: Average *residence time* of a token (request) at a queue, place or depository.

All models that we consider in this section are based on the hierarchical system model presented in the previous section. We use the *HiQPN-Tool* [14] to solve the models. Based on the solution of the model's underlying Markov chain the HiQPN-Tool reports the distribution of the number of tokens at each place in steady state. For queueing places the latter is reported separately for the queue and for the depository of the place. Using the distributions, one can easily derive the average token population $\mathcal{N}$ (which is done automatically and reported by the tool) and the utilization $\mathcal{U}$ of each queue in steady state. The following trivial relations hold:

$$\mathcal{N} = \sum_{i=0}^{\infty} i.p_i \tag{4.2}$$

$$\mathcal{U} = 1 - p_0 \tag{4.3}$$

where p_i is the probability that there are i tokens in the queue.

Note that since places Client, WLS-CPU, DBS-PQ, DBS-CPU and DBS-I/O form a closed chain with respect to the flow of requests in the system, using the *"flow-in = flow-out"* principle from Queueing Theory [166], we can conclude that in steady state the throughput of requests through each of these places must be the same. This applies also to the queues and depositories of the places, i.e. they have the same throughput as the places themselves. Furthermore, this holds both for total request throughput, as well as for throughput of particular request classes. The following trivial relationship holds [109]:

$$\mu = \frac{1}{\mathcal{D}} \tag{4.4}$$

Using this formula we can derive the service rates of the queues in the model for the different request classes based on their service demands provided in Table 4.10. Given that the service rates of all queues are load-independent, we can use the following relation (which follows from the Utilization Law) to derive the throughput $\mathcal{X}$ of a queue in the case of a single request class:

$$\mathcal{X} = \mathcal{U}.\mu \tag{4.5}$$

In the case of multiple request classes we can derive the throughput of each request class using Little's Law. Recall that the latter relates the throughput $\mathcal{X}$ of a queue with the average number of requests $\mathcal{N}$ in it and their average residence time $\mathcal{R}$. The relation is shown in Equation (4.6) below and can be applied both with respect to all requests of the system as well as with respect to separate request classes.

$$\mathcal{X} = \frac{\mathcal{N}}{\mathcal{R}} \tag{4.6}$$

We apply this formula to the Client queue for each request class in the system. The average number $\mathcal{N}$ of requests of each class is reported by the tool. The residence time of requests $\mathcal{R}$ is also known since the queue has IS scheduling strategy and therefore the residence time of all requests is equal to the service time of the queue, which is an input parameter to the model (the client think time). Substituting $\mathcal{N}$ and $\mathcal{R}$ in Equation (4.6) we can calculate the throughput of each request class. Now that we know how to find the throughput of requests in the system, we can derive the residence time of requests at every place, queue or depository using the following equation, which again follows directly from Little's Law:

$$\mathcal{R} = \frac{\mathcal{N}}{\mathcal{X}} \tag{4.7}$$

We are now ready to start analyzing some concrete instances of the model.

Scenario 1: Single Request Class

We start with a simplified scenario in which we have a single request class, the NewOrder transaction. Our goal is to analyze the behavior of the system with respect to this transaction. Assume that we have 80 clients in the system with average think time of 200ms and there are 60 WLS threads, 40 JDBC connections and 30 DBS processes available. We use this data to parameterize the hierarchical system model from Section 4.3.4. Table 4.11 summarizes the analysis results for this scenario. It reports the calculated throughput and residence time of requests at the most important queues and depositories. It also reports the average token population of places WLS-Thread-Pool, DB-Conn-Pool and DBS-Process-Pool. We have used subscripts 'Q' and 'D' to distinguish between queues and depositories of places.

Table 4.11: Analysis results for scenario 1.

PLACE	$\mathcal{N}$	$\mathcal{U}$	$\mathcal{X}$	$\mathcal{R}$ $[ms]$
Client$_Q$	2.85	0.94	14.28	200
Client$_D$	17.14	1.00	-//-	1200
WLS-CPU$_Q$	56.67	1.00	-//-	3967
WLS-CPU$_D$	0.00	0.00	-//-	0
DBS-PQ	0.00	0.00	-//-	0
DBS-CPU$_Q$	3.11	0.75	-//-	218
DBS-I/O$_Q$	0.20	0.17	-//-	14
WLS-Thread-Pool	0.00	0.00		
DB-Conn-Pool	36.67	1.00		
DBS-Process-Pool	26.67	1.00		

The total end-to-end request response time ($\mathcal{R}_{Total}$) is equal to the time needed for a request to make a complete cycle through the queueing places of the system. The latter can be calculated by summing up the residence times of requests at the queues and depositories of all queueing places plus the residence time at the ordinary place DBS-PQ.

$$\begin{aligned}\mathcal{R}_{Total} = {} & \mathcal{R}_{Client_D} + \mathcal{R}_{WLS-CPU_Q} + \\ & + \mathcal{R}_{WLS-CPU_D} + \mathcal{R}_{DBS-PQ} + \\ & + \mathcal{R}_{DBS-CPU_Q} + \mathcal{R}_{DBS-I/O_Q} \end{aligned} \tag{4.8}$$

Note that the above sum excludes the Client queue, since the time spent at it corresponds to the client think time. It also excludes the depositories of places DBS-CPU and DBS-I/O, because requests never wait at them. Table 4.12 compares results obtained from the model analysis with results obtained from measurements and shows the modeling error for the most important performance metrics. The measurements were collected by running an experiment in which the specified workload was injected into the system over a period of 40 minutes. Measurements were taken after the first 10 minutes, which the system needed to reach steady state.

Table 4.12: Modeling error for scenario 1.

METRIC	Model	Measured	Error
WLS-CPU Utilization	100%	100%	0%
DBS-CPU Utilization	75%	65%	15%
NewOrder Throughput	14.28	13.43	6.3%
NewOrder Resp.Time	5399ms	5738ms	5.9%
Thread Queue Length	17.14	18	4.7%

As we can see from Table 4.11, requests spend 1200ms on average at the Client depository waiting for a WLS thread to become available. On the other hand, there are plenty of JDBC connections and DBS processes available and there is no contention for these resources as indicated by the residence times of requests at WLS-CPU$_D$ and DBS-PQ both of which are zero. Since, on average, there are only 3.31 requests served concurrently at the database server we can decrease the number of available JDBC connections and DBS processes significantly without impacting performance in a negative way. In fact, doing this could even improve the overall performance since JDBC connections and DBS processes cost memory and reducing them will increase the amount of memory available for the WebLogic server and database server, respectively.

It would be interesting to see what would change if we decrease the number of available WLS threads to 40. This would limit the number of requests processed concurrently by the WebLogic server. Table 4.13 repeats the analysis for 40 WLS threads.

Table 4.13: Analysis results for scenario 1 with 40 threads.

PLACE	$\mathcal{N}$	$\mathcal{U}$	$\mathcal{X}$	$\mathcal{R}$ $[ms]$
Client_Q	2.85	0.94	14.28	200
Client_D	37.14	1.00	-//-	2601
WLS-CPU_Q	36.67	1.00	-//-	2568
WLS-CPU_D	0.00	0.00	-//-	0
DBS-PQ	0.00	0.00	-//-	0
DBS-CPU_Q	3.11	0.75	-//-	218
DBS-I/O_Q	0.20	0.17	-//-	14
WLS-Thread-Pool	0.00	0.00		
DB-Conn-Pool	36.67	1.00		
DBS-Process-Pool	26.67	1.00		

As we can see, the change does not have any impact on the overall system throughput, since in both cases the WebLogic server is saturated to its full capacity. However, one might at first be misled to believe that because of increased contention for threads, the end-to-end request response time would also increase, which according to the model as well as the measurements is not the case. This is because reducing the level of concurrency in the WebLogic server results in requests being served faster and this compensates the longer time spent queueing for threads. In both cases the WebLogic server is completely utilized and the rest of the system remains unaffected by the change. Table 4.14 shows the modeling error with 40 threads. As we can see, in both cases the QPN model is quite accurate in representing the system.

As demonstrated, QPNs enable us to integrate in the same model hardware and software aspects of system behavior. Using queueing places we can easily model hardware contention and scheduling strategies with the same flexibility as in traditional QNs. However, in addition to this, QPNs also empower us to represent some further aspects of system behavior such as simultaneous resource possession, blocking and contention for software resources (threads, connections and processes). The latter is not possible, at least at this level of accuracy, using conventional modeling paradigms such as QNs and Petri nets. Even though extensions of QNs, such as *Extended QNs*, provide some limited support for modeling software contention, they are way too restrictive and inaccurate to compare with the modeling power demonstrated in the above examples.

Table 4.14: Modeling error for scenario 1 with 40 threads.

METRIC	Model	Measured	Error
WLS-CPU Utilization	100%	100%	0%
DBS-CPU Utilization	75%	65%	15%
NewOrder Throughput	14.28	13.41	6.4%
NewOrder Resp.Time	5401ms	5742ms	5.9%
Thread Queue Length	37.14	40	7.1%

Scenario 2: Multiple Request Classes

We will now look at a scenario in which we have two classes of requests in the system - NewOrder and ChangeOrder. We again use the hierarchical model from Section 4.3.4 as a basis. However, this time we define two types of request tokens (NewOrder and ChangeOrder) so that we can distinguish between the two request classes. Trying to solve the resulting HQPN model for high values of the number of NewOrder and ChangeOrder clients, we ran into state space explosion of the underlying Markov chain. Therefore, we make some simplifications in order to come up with a model that is analyzable. First of all, we assume that there are plenty of JDBC connections and DBS server processes and drop places DB-Conn-Pool and DBS-Process-Pool. In addition, we assume that there are only 20 clients in the system (10 NewOrder and 10 ChangeOrder), the average think time is 1 second and there are 10 WLS threads available. Tables 4.15 and 4.16 report the analysis results and modeling error for this scenario. The modeling error for most performance metrics remains under 10%.

Scenario 3: Multiple WebLogic servers

In this final scenario we generalize our initial setting to allow multiple WebLogic servers to be used. We again use the hierarchical system model from Section 4.3.4 as a basis, but modify the HLQPN to include multiple WLS queueing places - one per WebLogic server. The new HLQPN is depicted in Figure 4.10.

Table 4.15: Analysis results for scenario 2.

PLACE	$\mathcal{N}$	$\mathcal{U}$	$\mathcal{X}$	$\mathcal{R}$ $[ms]$
WLS-Thread-Pool	6.68	0.99		
Over All Request Classes				
Client_Q	16.67	1.00	16.67	1000
Client_D	0.00	0.00	-//-	0
WLS-CPU_Q	2.14	0.76	-//-	128
DBS-CPU_Q	1.00	0.54	-//-	59
DBS-I/O_Q	0.16	0.14	-//-	9
Over NewOrder Requests				
Client_Q	7.45	1.00	7.45	1000
Client_D	0.00	0.00	-//-	0
WLS-CPU_Q	1.64	0.70	-//-	220
DBS-CPU_Q	0.79	0.46	-//-	107
DBS-I/O_Q	0.10	0.06	-//-	14
Over ChangeOrder Requests				
Client_Q	9.22	1.00	9.22	1000
Client_D	0.00	0.00	-//-	0
WLS-CPU_Q	0.50	0.35	-//-	54
DBS-CPU_Q	0.21	0.19	-//-	23
DBS-I/O_Q	0.06	0.09	-//-	7

Table 4.16: Modeling error for scenario 2.

METRIC	Model	Measured	Error
WLS-CPU Utilization	76%	77%	1.2%
DBS-CPU Utilization	54%	64%	15.6%
Avg.free WLS-Threads	6.68	7	4.5%
NewOrder Throughput	7.45	7.47	0.2%
NewOrder Resp. Time	341ms	318ms	7.2%
ChgOrder Throughput	9.22	9.15	0.7%
ChgOrder Resp. Time	84ms	104ms	19.2%

To simplify things we do not include WLS-Thread places for the WebLogic servers in the new model. In fact, if we were to have WLS-Thread places for the

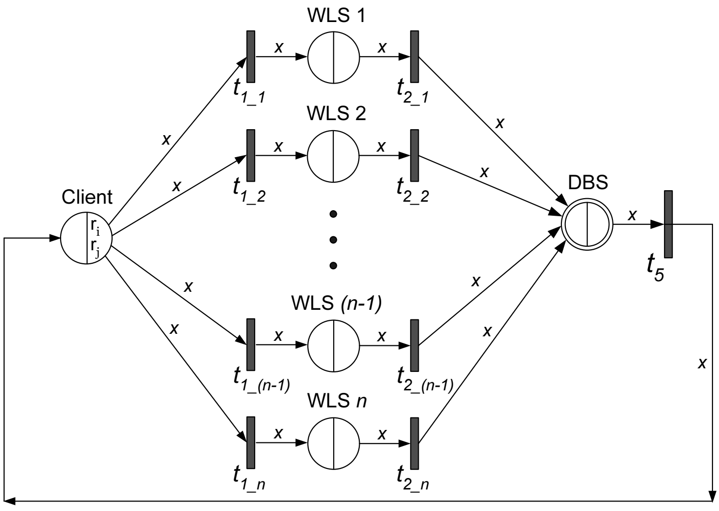


Figure 4.10: High-level QPN model with N WebLogic servers.

WebLogic servers, we would also need some way to distinguish between requests in the DBS subnet originating from different WebLogic servers. This is because we need to know at which WebLogic server to release a thread after completing service at the DBS subnet. One way to implement this is using the notion of *tags* [15] which are automatically added to tokens upon entry into the DBS subnet to keep track of their origin. However, this functionality is currently not supported by the HiQPN-Tool and therefore we do not include any WLS-Thread places in the model. We assume that there are 30 NewOrder clients in the system and that there is no contention for WLS threads, JDBC connections or Oracle processes. The client think time is again 1 second. Tables 4.17 and 4.18 summarize the analysis results for 2 and 3 WebLogic servers. As seen from the results, the modeling error remains under 10% and the QPN models perform quite well as a performance prediction tool.

4.3.6 Conclusions from the Analysis

We used the QPN model to predict the system performance for several different workload and configuration scenarios. In addition to transaction throughput and response times, we were able to predict the number of WLS threads, JDBC connections and Oracle processes used on average during operation. In all cases, the

Table 4.17: Analysis results for scenario 3.

PLACE	$\mathcal{N}$	$\mathcal{U}$	$\mathcal{X}$	$\mathcal{R}$ $[ms]$
For 2 WebLogic Servers				
Client$_Q$	18.28	1.00	18.28	1000
WLS1-CPU$_Q$	1.68	0.64	9.14	184
WLS2-CPU$_Q$	1.68	0.64	9.14	184
DBS-CPU$_Q$	8.07	0.96	18.28	441
DBS-I/O$_Q$	0.27	0.21	-//-	15
For 3 WebLogic Servers				
Client$_Q$	18.42	1.00	18.42	1000
WLS1-CPU$_Q$	0.72	0.43	6.14	117
WLS2-CPU$_Q$	0.72	0.43	6.14	117
WLS3-CPU$_Q$	0.72	0.43	6.14	117
DBS-CPU$_Q$	9.05	0.98	18.42	491
DBS-I/O$_Q$	0.28	0.22	-//-	15

Table 4.18: Modeling error for scenario 3.

METRIC	Model	Measured	Error
For 2 WebLogic Servers			
WLS-CPU Utilization	64%	68%	6%
DBS-CPU Utilization	96%	91%	5%
NewOrder Throughput	18.28	17.56	4%
NewOrder Resp. Time	640ms	693ms	8%
For 3 WebLogic Servers			
WLS-CPU Utilization	43%	44%	2%
DBS-CPU Utilization	98%	97%	1%
NewOrder Throughput	18.42	17.61	5%
NewOrder Resp. Time	623ms	673ms	7%

model predictions were very accurate and the modeling error did not exceed 20%. However, unfortunately, it was not possible to provide exhaustive answers to the questions asked in Section 4.3.1. Trying to analyze the QPN model for a realistic customer population, we ran into the state space explosion problem and the tractable scenarios had to be (artificially) simplified. Thus, using currently available analysis techniques for QPNs, it was not possible to predict the system performance under realistic load conditions and analyze its scalability.

4.4 Concluding Remarks

In this chapter, we presented two practical performance modeling case studies which demonstrated how conventional QN models and the relatively new QPN models can be exploited for performance analysis of DCS. We modeled two realistic systems and discussed the difficulties stemming from the limited model expressiveness, on the one hand, and the lack of scalable analysis techniques, on the other hand.

Using QN models it was not possible to accurately model asynchronous processing and software contention aspects. Moreover, the available analysis techniques were not able to provide reliable response time results when considering large models with increased workload intensity.

We showed that QPN models hold a significant advantage over conventional QN models, since they allow the integration of hardware and software aspects of system behavior into the same model. However, we also showed that QPN models of realistic systems are too large to be analyzable using currently available tools and techniques for QPN analysis. The problem is that the latter are all based on Markov chain analysis, which suffers the state space explosion problem and limits the size of the models that can be solved. This is the reason why QPNs have hardly been exploited in the past decade and very few, if any, practical applications have been reported. Even though HQPNs and structured analysis techniques alleviate the problem, they do not eliminate it. In the next chapter, we develop a new method for analyzing QPN models that exploits discrete event simulation. The proposed method circumvents the state space explosion problem and allows models of realistic systems to be analyzed.

Chapter 5

Analysis of QPN Models by Simulation

Applying simulation to the modeling and performance analysis of complex systems can be compared to using the surgical scalpel, whereby "in the right hand [it] can accomplish tremendous good, but it must be used with great care and by someone who knows what they are doing".

– *R. E. Shannon, 1981*

5.1 Introduction

The Queueing Petri Net (QPN) paradigm provides a number of benefits over conventional modeling paradigms such as queueing networks and stochastic Petri nets. Using QPNs one can integrate hardware and software aspects of system behavior into the same model. As demonstrated in the previous chapter this lends itself very well to modeling DCS. However, currently available tools and techniques for QPN analysis suffer the state space explosion problem imposing a limit on the size of the models that are tractable. In this chapter, we present *SimQPN* - a simulation tool for QPNs developed as part of this thesis that provides an alternative approach to analyze QPN models, circumventing the state space explosion problem. In doing this, we propose a methodology for analyzing QPN models by means of discrete event simulation. The methodology shows how to simulate QPN models and analyze the output data from simulation runs. We validate our approach by applying it to study several different QPN models ranging from simple models to models of realistic systems. We evaluate the quality of data provided by SimQPN by conducting an exhaustive experimental analysis of the variation of point estimates and coverage of confidence intervals reported.

An alternative approach to simulate QPN models would be to use a general purpose simulation package. However, this approach has some disadvantages. First, general purpose simulation packages do not provide means to represent QPN constructs directly. Instead, they require that simulation models are described using a general purpose simulation language. Mapping a QPN model to a description in the terms of a general purpose simulation language is a complex, time-consuming and error-prone task. Moreover, not all simulation languages provide the expressiveness needed to describe complex QPN models. Some simplifications might be required that could lead to less accurate results. Another disadvantage is that general purpose simulators are normally not as fast and efficient as specialized simulators, since they are usually not optimized for any particular type of models. Being specialized for QPNs, SimQPN simulates QPN models directly and has been designed to exploit the knowledge of the structure and behavior of QPNs to improve the efficiency of the simulation. Therefore, SimQPN is expected to provide much better performance than a general purpose simulator, both in terms of the speed of simulation and the quality of output data provided. Last but not least, SimQPN has the advantage that it is extremely light-weight and being implemented in Java it is platform independent.

This chapter is organized as follows: In Section 5.2, we present SimQPN - our simulation tool for QPNs, and discuss its features, design and architecture. In parallel to this, we discuss our methodology for simulating QPN models based on which SimQPN was developed. We look at the methods for output data analysis employed and discuss the specifics of their implementation. Following this, in Section 5.3, we study several different QPN models using SimQPN and validate the results with respect to correctness and accuracy. We evaluate the performance of point and interval estimators implemented in SimQPN. Finally, in Section 5.4, we present some concluding remarks.

5.2 SimQPN - Simulator for Queueing Petri Nets

In this section, we present SimQPN taking a detailed look at its features, design and architecture. In parallel to this, we present our methodology for simulating QPN models based on which SimQPN was developed. The methodology shows how to simulate QPN models and analyze the output data from simulation runs. We look at the methods for output data analysis employed and discuss the specifics of their implementation.

5.2.1 SimQPN Features

SimQPN is a discrete event simulation (DES) engine specialized for QPNs. It is implemented 100% in Java to provide maximum portability and platform-independence. It is extremely light-weight (less than 1 MB) and requires only an installed Java Runtime Environment (JRE)[1]. SimQPN simulates QPNs using a sequential algorithm based on the event-scheduling approach for simulation modeling.

In the first version of SimQPN, the most important features typically used in QPN models have been implemented. As of the time of writing, QPN models with the following restrictions are supported:

- Scheduling strategies for queues are limited to FCFS, PS and IS.
- The following service time distributions are supported for FCFS and IS queues: Beta, BreitWigner, ChiSquare, Gamma, Hyperbolic, Exponential, ExponentialPower, Logarithmic, Normal, StudentT, Uniform and VonMises. For PS queues, currently only exponential service time distributions are supported, which makes it easier to handle residual service times. For the next version of SimQPN it is planned to relax this restriction.
- Empirical distributions are supported in the following way. The user is expected to provide a probability distribution function (PDF), specified as an array of positive real numbers (histogram). A cumulative distribution function (CDF) is constructed from the PDF and inverted using a binary search for the nearest bin boundary and a linear interpolation within the bin (resulting in a constant density within each bin).
- Timed transitions are currently not supported. However, in most cases, a timed transition can be approximated by a serial network consisting of an immediate transition, a queueing place and a second immediate transition.
- Since in practice immediate queueing places are very rarely used, they have been left out from the first version of SimQPN.

5.2.2 Design and Architecture

SimQPN has an object-oriented architecture. Every element (for e.g. place, transition or token) of the simulated QPN is internally represented as object. Communication between objects is mostly implemented through method calls, with exception of some cases, where object data is accessed directly (bypassing accessor methods) to provide better performance. Although the latter is a deviation from the object-oriented paradigm, we have made this compromise because it speeds up the

[1]JRE version 1.1 or higher is required.

simulation significantly. Figure 5.1 shows the major types of objects (classes) used in SimQPN and the relationships among them. At the top level is the Simulator class, which contains the main simulation routine. The PlaceStats and QueueStats objects are used to manage statistics gathered during the simulation. The AggregateStats object is used to manage statistics gathered from multiple simulation runs.

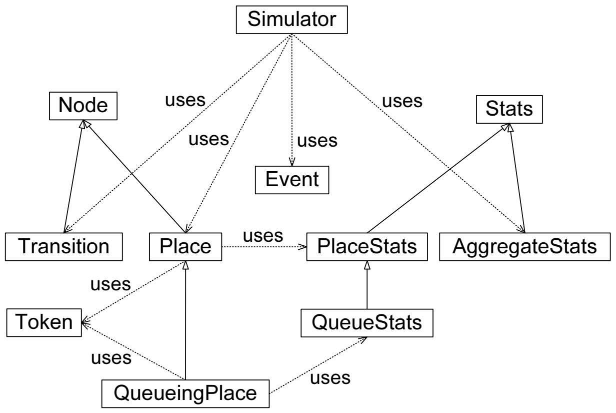


Figure 5.1: SimQPN's object model.

Figure 5.2 outlines the main simulation routine which drives each simulation run. As already mentioned, SimQPN's internal simulation procedure is based on the event-scheduling approach [66, 92]. To explain what is understood by event here, we need to look at the way the simulated QPN transitions from one state to another with respect to time. Since only immediate transitions are supported, the only place in the QPN where time is involved is inside the queues of queueing places. Tokens arriving at the queues wait until there is a free server available and are then served. A token's service time distribution determines how long its service continues. After a token has been served it is moved to the depository of the queueing place, which may enable some transitions and trigger their firing. This leads to a change in the marking of the QPN. Once all enabled transitions have fired, the next change of the marking will occur after another service completion at some queue. In this sense, it is the completion of service that initiates each change of the marking. Therefore, we define *event* to be a completion of a token's service at a queue.

For FCFS queues, a token's service completion event is scheduled (added to the event list) as soon as there is a free server available to serve the token. For IS queues, a token's service completion event is scheduled immediately upon arrival of the token at the queue. Finally, for PS queues in contrast to FCFS and IS queues,

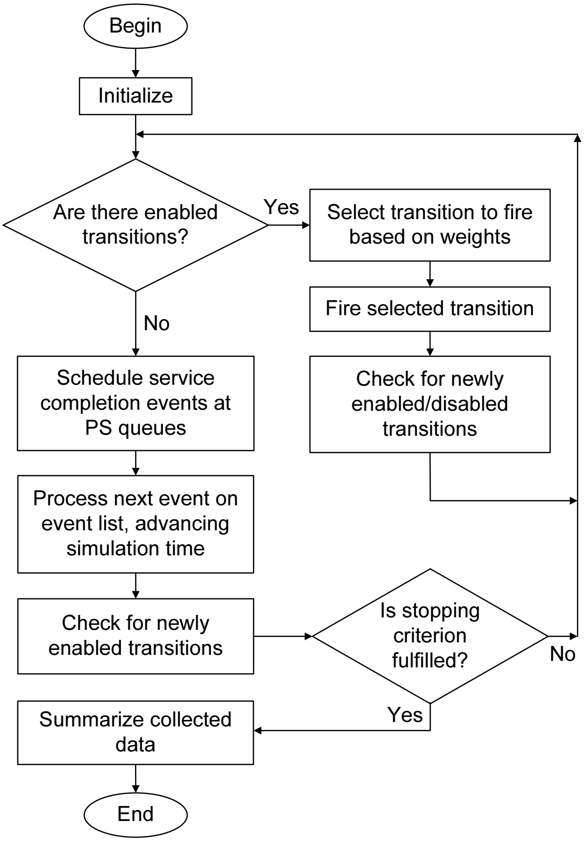

Begin
Initialize
Are there enabled transitions?
Yes
Select transition to fire based on weights
Fire selected transition
Check for newly enabled/disabled transitions
No
Schedule service completion events at PS queues
Process next event on event list, advancing simulation time
Check for newly enabled transitions
Is stopping criterion fulfilled?
No
Yes
Summarize collected data
End

Figure 5.2: SimQPN's main simulation routine.

service completion events are only scheduled after all enabled transitions have fired. This is because service rates at PS queues depend on the token population which may change when transitions fire. By deferring the scheduling of service completion events until after all enabled transitions have fired it is avoided to have to reschedule the events after each change in the token population as transitions fire. Thus, the knowledge of the behavior of the simulated QPN is exploited to save CPU time and improve the efficiency of the simulation. A scheduled service completion event at a PS queue might still need to be rescheduled, however, only in the case where before the time has come for it to be executed, events in other queues cause new transitions to be enabled and their firing triggers a change in the queue's token population. If this happens the next service completion event of the PS queue is rescheduled according to its new token population. Elapsed service times can be safely ignored since PS queues are assumed to have exponentially distributed service times. The next version of SimQPN is planned to also deal with the more complicated case of PS queues with non-exponential service time distributions.

Another way in which SimQPN exploits the knowledge of the structure and behavior of QPNs to improve the efficiency of the simulation is by using an optimized algorithm for keeping track of the enabling status of transitions. Generally, Petri net simulators need to check for enabled transitions after each change in the marking caused by a transition firing. The exact way they do this is one of the major factors determining the efficiency of the simulation [57]. In [118], it is shown how the *locality principle* of colored Petri nets can be exploited to minimize the overhead of checking for enabled transitions. The locality principle states that an occurring transition will only affect the marking on immediate neighbor places and hence the enabling status of a limited set of neighbor transitions. SimQPN exploits an adaptation of this principle to QPNs, taking into account that tokens deposited into queueing places do not become available for output transitions immediately upon arrival and hence cannot affect the enabling status of the latter. Since checking the enabling status of a transition is a computationally expensive operation, our goal is to make sure that this is done as seldom as possible, i.e. only when there is a real possibility that the status has changed. This translates into the following two cases when the enabling status of a transition needs to be checked:

1. After a change in the token population of an ordinary input place of the transition, as a result of firing of the same or another transition. Three subcases are distinguished:

    (a) Some tokens were added. In this case, it is checked for *newly enabled modes* by considering all modes that are currently marked as disabled and that require tokens of the respective colors added.

    (b) Some tokens were removed. In this case, it is checked for *newly disabled modes* by considering all modes that are currently marked as enabled and that require tokens of the respective colors removed.

    (c) Some tokens were added and at the same time others were removed. In this case, both of the checks above are performed.

2. After a service completion event at a queueing input place of the transition. The service completion event results in adding a token to the depository of the queueing place. Therefore, in this case, it is only checked for *newly enabled*

modes by considering all modes that are currently marked as disabled and that require tokens of the respective color added.

SimQPN maintains a global list of currently enabled transitions and for each transition a list of currently enabled modes. The latter are initialized at the beginning of the simulation by checking the enabling status of all transitions. As the simulation progresses, a transition's enabling status is checked only in the above mentioned cases. This reduces CPU costs and speeds up the simulation substantially.

5.2.3 Random Number Generation

SimQPN utilizes the *Colt* open source library for high performance scientific and technical computing in Java, developed at CERN [38]. In SimQPN, Colt is primarily used for random number generation and, in particular, its implementation of the Mersenne Twister random number generator is employed [104]. The latter is one of the strongest uniform pseudo-random number generators known and passes many stringent statistical tests, including the diehard test of G. Marsaglia [102] and the load test of P. Hellekalek and S. Wegenkittl [62, 95, 169]. It has an astronomically large period of $2^{19937} - 1 (= 10^{6001})$ and 623-dimensional equidistribution with up to 32-bit accuracy. By default, SimQPN uses Mersenne Twister for all of its random number streams. However, for situations where one is willing to trade off quality for performance, it offers an alternative medium quality uniform pseudo-random number generator that is a bit faster. In addition to Mersenne Twister, SimQPN also employs Colt's random seed generator to ensure that there is no correlation between seeds used to initialize random number generators.

5.2.4 Output Data Analysis

Modes of Data Collection

SimQPN offers the ability to configure what data exactly to collect during the simulation and what statistics to provide at the end of the run. This can be specified for each place (ordinary or queueing) of the QPN. The user can choose one of four modes of data collection. The higher the mode, the more information is collected and the more statistics are provided. Since collecting data costs CPU time, the more data is collected, the slower the simulation would run. Therefore, by configuring data collection modes, the user can make sure that no time is wasted collecting unnecessary data and, in this way, speed up the simulation.

Mode 1 considers only token throughput data, i.e. for each queue, place or depository the token arrival and departure rates are estimated for each color.

Mode 2 adds token population and utilization data, i.e. for each queue, place and depository the following data is provided on a per-color basis:

- Minimum/maximum number of tokens.
- Average number of tokens.
- Mean color utilization, i.e. the fraction of time that there is a token of the respective color inside the queue/place/depository.

For queues, in addition to the above, the overall queue utilization is reported (i.e. the fraction of time that there is a token of any color inside the queue).

Mode 3 adds residence time data, i.e. for each queue, place and depository the following additional data is provided on a per-color basis:

- Minimum/maximum observed token residence time.
- Mean and standard deviation of observed token residence times.
- Estimated steady state mean token residence time.
- Confidence interval (c.i.) for the steady state mean token residence time at a user-specified significance level.

Mode 4 provides all of the above and additionally dumps observed token residence times to files.

Steady State Analysis

SimQPN supports two basic methods for estimation of the steady state mean residence times of tokens inside the queues, places and depositories of the QPN. These are the well-known *method of independent replications* (in its variant referred to as *replication/deletion approach*) and the classical *method of non-overlapping batch means (NOBM)*. We refer the reader to [1, 2, 8, 92, 132, 134] for an introduction to these methods. Both of them can be used to provide point and interval estimates of the steady state mean token residence time. In cases where one wants to apply a more sophisticated technique for steady state analysis (for example ASAP [149, 150, 151, 152]), SimQPN can be configured to output observed token residence times to files (mode 4), which can then be used as input to external analysis tools (for example [52]).

Both the replication/deletion approach and the method of NOBM have different variants [119, 132, 152]. Below we discuss some details on the way they were implemented in SimQPN.

Elimination of Initialization Bias Since we are interested in estimating steady state parameters of the simulated queueing process, we need to somehow address the well-known *problem of the initial transient* [131, 132, 172]. Both of the above mentioned methods require that the analyzed sequence of observations is stationary, and therefore when using them, the effects of transient system behavior need to be accounted for. A number of different approaches have been proposed in the literature for dealing with this problem, including heuristic, statistical, graphical and hybrid approaches. For a survey of methods refer to [97, 132, 138]. Most methods attempt to estimate the length of the warm-up period and then discard all data collected during it to eliminate initialization bias. One of the simplest and most popular methods is the graphical method of Welch [61, 173], which has met some success [1, 97]. The latter is appealing because it is simple, practical and does not make any assumptions about the type of system modeled. For these reasons, we decided as a start to implement the method of Welch in SimQPN. We have followed the rules in [92] for choosing the number of replications, their length and the window size. SimQPN allows the user to configure the first two parameters and then automatically plots the moving averages for different window sizes. Thus, simulation experiments with SimQPN usually comprise two stages: stage 1 during which the length of the initial transient is determined, and stage 2 during which the steady state behavior of the system is simulated and analyzed. Again, if the user prefers to use another method for elimination of the initialization bias, this can be achieved by dumping collected data to files (mode 4) and feeding it into respective analysis tools.

Replication/Deletion Approach We briefly discuss the way the replication/deletion approach is implemented in SimQPN. Suppose that we want to estimate the steady state mean residence time ν of tokens of given color at a given place, queue or depository. As discussed in [2], in the replication/deletion approach multiple replications of the simulation are made and the average residence times observed are used to derive steady state estimates. Specifically, suppose that n replications of the simulation are made, each of them generating m residence time observations $Y_{i1}, Y_{i2}, \ldots, Y_{im}$. We delete l observations from the beginning of each set to eliminate the initialization bias. The number of observations deleted is determined through the method of Welch as discussed in the previous paragraph. Let X_i be given by

$$X_i = \frac{\sum_{j=l+1}^{m} Y_{ij}}{m-l} \qquad i = 1, 2, \ldots, n \tag{5.1}$$

and

$$\overline{X}(n) = \frac{\sum_{i=1}^{n} X_i}{n} \tag{5.2}$$

$$S^2(n) = \frac{\sum_{i=1}^{n} [X_i - \overline{X}(n)]^2}{n-1} \tag{5.3}$$

Then the X_i's are independent and identically distributed (IID) random variables with $E(X_i) \approx \nu$ and $\overline{X}(n)$ is an approximately unbiased point estimator for ν. According to the central limit theorem [22, 166], if m is large, the X_i's are going to be approximately normally distributed and therefore the random variable

$$t_n = \frac{[\overline{X}(n) - \nu]}{\sqrt{\frac{S^2(n)}{n}}}$$

will have t distribution with $(n-1)$ degrees of freedom (df) [65] and an approximate $100(1-\alpha)$ percent confidence interval for ν is then given by

$$\overline{X}(n) \pm t_{n-1,1-\alpha/2} \sqrt{\frac{S^2(n)}{n}} \tag{5.4}$$

where $t_{n-1,1-\alpha/2}$ is the upper $(1-\alpha/2)$ critical point for the t distribution with $(n-1)$ df [8, 132, 166].

Method of Non-Overlapping Batch Means Unlike the replication/deletion approach, the method of NOBM seeks to obtain independent observations from a single simulation run rather than from multiple replications. Thus, it has the advantage that it must go through the warm-up period only once and is therefore less sensitive to bias from the initial transient. Suppose that we make a simulation run of length m and then divide the resulting observations $Y_1, Y_2, \ldots, Y_m$ into n batches of length q. Assume that $m = n * q$ and let X_i be the sample (or batch) mean of the q observations in the ith batch, i.e.

$$X_i(q) = \frac{\sum_{j=(i-1)q+1}^{(i-1)q+q} Y_j}{q} \qquad i = 1, 2, \ldots, n \tag{5.5}$$

The mean ν is estimated by $\overline{X}(n) = (\sum_{i=1}^{n} X_i(q))/n$ and it can be shown (see for example [8, 92]) that an approximate $100(1-\alpha)$ percent confidence interval for ν is given by substituting $X_i(q)$ for X_i in Equations (5.2), (5.3) and (5.4) above.

SimQPN offers two different *stopping criteria* for determining how long the simulation should continue. In the first one, the simulation continues until the QPN has been simulated for a user-specified amount of model time (*fixed-sample-size procedure*). In the second one, the length of the simulation is increased sequentially from one checkpoint to the next, until enough data has been collected to provide estimates of residence times with user-specified precision (*sequential procedure*). The precision is defined as an upper bound for the confidence interval half length. It can be specified either as an absolute value (absolute precision) or as a percentage relative to the mean residence time (relative precision). The sequential approach for controlling the length of the simulation is usually regarded as the only efficient way for ensuring representativeness of the samples of collected observations [61, 91, 133]. Therefore, hereafter we assume that the sequential procedure is used.

The main problem with the method of NOBM is to select the batch size q, such that successive batch means are approximately uncorrelated. Different approaches have been proposed in the literature to address this problem (see for example [1, 43, 132]). In SimQPN, we start with a user-configurable initial batch size (by default 200) and then increase it sequentially until the correlation between successive batch means becomes negligible. Thus, the simulation goes through two stages: the first sequentially testing for an acceptable batch size and the second sequentially testing for adequate precision of the residence time estimates (see Figure 5.3). The parameters n and p, specifying how often checkpoints are made, can be configured by the user.

We use the *jackknife estimators* [115, 132] of the autocorrelation coefficients to measure the correlation between batch means. A jackknife estimator $\hat{J}(k,q)$ of the autocorrelation coefficient of lag k for the sequence of batch means $X_1(q), X_2(q), \ldots, X_n(q)$ of size q is calculated as follows:

$$\hat{J}(k,q) = 2\hat{r}(k,q) - \frac{\hat{r}'(k,q) + \hat{r}''(k,q)}{2} \tag{5.6}$$

where $\hat{r}(k,q)$ is the ordinary estimator of the autocorrelation coefficient of lag k, calculated from the formula [132]:

$$\hat{r}(k,q) = \frac{\hat{R}(k,q)}{\hat{R}(0,q)} \tag{5.7}$$

where

$$\hat{R}(k,q) = \frac{1}{n-k} \sum_{i=k+1}^{n} [X_i(q) - \overline{X}(n)][X_{i-k}(q) - \overline{X}(n)] \tag{5.8}$$

and $\hat{r}'(k,q)$ and $\hat{r}''(k,q)$ are calculated like $\hat{r}(k,q)$, except that $\hat{r}(k,q)$ is the estimator over all n batch means, whereas $\hat{r}'(k,q)$ and $\hat{r}''(k,q)$ are estimators over the first and the second half of the analyzed sequence of n batch means, respectively.

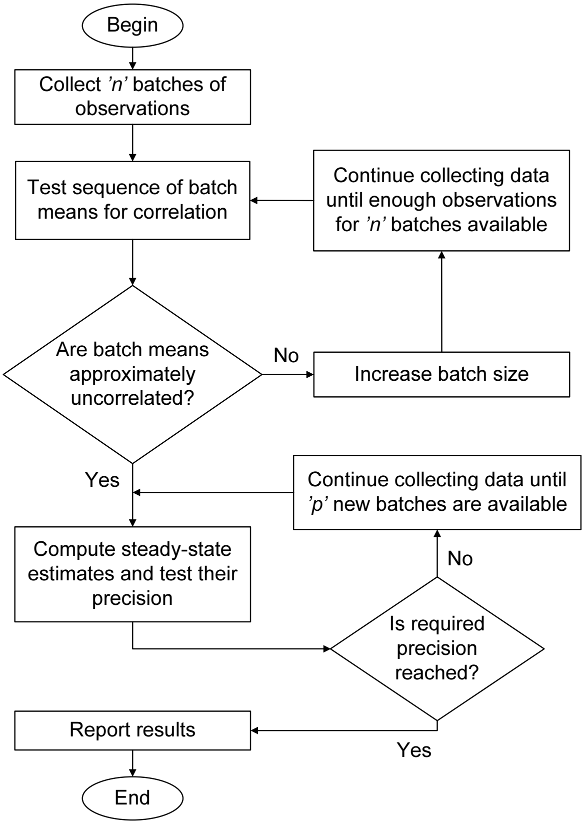

Begin
Collect 'n' batches of observations
Test sequence of batch means for correlation
Continue collecting data until enough observations for 'n' batches available
Are batch means approximately uncorrelated?
No
Increase batch size
Yes
Continue collecting data until 'p' new batches are available
Compute steady-state estimates and test their precision
No
Is required precision reached?
Report results
Yes
End

Figure 5.3: SimQPN's batch means procedure.

We use the algorithm proposed in [132] to determine when to consider the sequence of batch means as approximately uncorrelated: a given batch size is accepted to yield approximately uncorrelated batch means if all autocorrelation coefficients of lag k ($k = 1, 2, \ldots, L$; where $L = 0.1 * n$) are statistically negligible at a given significance level β_k, $0 < \beta_k < 1$. To get an acceptable overall significance level β we assume that

$$\beta < \sum_{k=1}^{L} \beta_k \tag{5.9}$$

As recommended in [132], in order to get reasonable estimators of the autocorrelation coefficients, we apply the above batch means correlation test only after at least 100 batch means have been recorded (i.e. $n >= 100$). In fact, by default n is set to 200 in SimQPN. Also to ensure approximate normality of the batch means, the initial batch size (i.e. the minimal batch size) is configured to 200.

For FCFS queues, SimQPN also supports *indirect estimation* of the steady state token residence times according to the variance-reduction technique in [36]. The latter suggests, first estimating delay times in the waiting areas of the queues, and then adding them to the mean service times to obtain indirect estimates of the total residence times with reduced variance. SimQPN allows the user to configure for each FCFS queue whether direct or indirect estimates should be used (the default is indirect).

5.3 SimQPN Validation and Performance Analysis

In this section, we analyze several different QPN models by means of SimQPN, and then validate the results with respect to correctness and accuracy. We follow the guidelines in [5, 8, 78, 91, 92, 143, 144] and consider models of different size and complexity, in each case, examining the simulation output under a variety of settings for the input parameters. We compare the simulation results with results obtained using other methods, i.e. analytical techniques, approximation techniques or measurements on the system modeled. In some cases, we consider QPN models that may be mapped to equivalent QN models which are analytically tractable. The latter enables us to easily validate simulation results by comparing them against results obtained using analytical techniques applied to the equivalent QN models. This allows us to consider large QPN models that are not analyzable using currently available QPN analysis techniques, but whose equivalent QN models may be analyzed using conventional techniques.

In addition to validating results for reasonableness, we also study the performance of the point and interval estimators implemented in SimQPN. We conduct an exhaustive experimental analysis of the variation of point estimates and coverage of confidence intervals. Before we begin with the presentation of the results, we briefly discuss the method of coverage analysis we employ.

5.3.1 Method of Coverage Analysis Used

Let us consider multiple independent replications (runs) of a simulation experiment. *Coverage* of confidence interval is defined as the probability c that the confidence interval (obtained from a replication) covers the true value ν of the respective parameter being estimated. As usual, if n replications of the experiment have been executed, the coverage c can be estimated by the proportion:

$$\hat{c} = \frac{s}{n} \tag{5.10}$$

where s is the number of replications in which the reported confidence interval contains the true value. The accuracy with which $\hat{c}$ estimates c is usually assessed by the following confidence interval based on the normal distribution:

$$\left(\hat{c} - z_{1-\alpha/2}\sqrt{\frac{\hat{c}(1-\hat{c})}{n}} \;,\; \hat{c} + z_{1-\alpha/2}\sqrt{\frac{\hat{c}(1-\hat{c})}{n}} \right) \tag{5.11}$$

where $z_{1-\alpha/2}$ is the $(1-\alpha/2)$ quantile of the standard normal distribution. This is based on the fact that, while the number of confidence intervals containing the true value ν has a binomial distribution with mean nc, $(\hat{c} - c)\sqrt{\hat{c}(1-\hat{c})/n}$ tends to the standard normal distribution as $n \to \infty$ [65, 133].

In [106, 133] it is argued that, unless certain rules are adhered to, the above point and interval estimators for coverage cannot be relied upon to produce reliable results. It is advocated to conduct coverage analysis sequentially and several rules are formulated for obtaining reliable and statistically accurate results. In [93, 94] the rules are revised and extended, proposing to use an interval estimator of coverage based on the F distribution. We now briefly summarize these rules indicating how they were implemented in our coverage analysis for SimQPN:

- **Rule 1:** Coverage should be analyzed sequentially, i.e. analysis of coverage should be stopped when the *absolute precision* of the estimated coverage satisfies a specified level which is sufficiently small. In our case, we stopped when reaching absolute precision of +/-0.05.

- **Rule 2:** An estimate of coverage has to be calculated from a representative sample of data, so the coverage analysis can start only after a minimum number of "bad"confidence intervals have been recorded. In our case, we required at least 200 "bad"confidence intervals (as suggested in [93]) to be recorded before sequential analysis commences.

- **Rule 3:** Results from simulation runs that are clearly too short should not be taken into account. To implement this rule, after recording 200 "bad"confidence intervals, we calculated the average run length and then discarded all runs shorter by more than one standard deviation than the average run length. As justified in [106, 133], this filters out statistical "noise"and removes significant bias in the results.

- **Rule 4:** An interval estimator which is based on the F distribution of coverage should be used to ensure that the sequential analysis of coverage produces realistic estimates: A $100(1-\alpha)$ lower limit $\hat{c_l}$ and upper limit $\hat{c_u}$ of the confidence interval for the true coverage are given by:

$$\begin{aligned}\hat{c_l} &= \hat{c} - \Delta_1 = \frac{n\hat{c}}{n\hat{c} + (n - n\hat{c} + 1) f_{1-\alpha/2}(r1, r2)} \\ \hat{c_u} &= \hat{c} + \Delta_2 = \frac{(n\hat{c} + 1) f_{1-\alpha/2}(r3, r4)}{(n - n\hat{c}) + (n\hat{c} + 1) f_{1-\alpha/2}(r3, r4)} \end{aligned} \qquad (5.12)$$

where $f_{1-\alpha/2}(r1, r2)$ and $f_{1-\alpha/2}(r3, r4)$ are the $(1-\alpha/2)$ quantiles of the F distribution with $(r1, r2)$ and $(r3, r4)$ degrees of freedom,
$r1 = 2 * (n - n\hat{c} + 1)$, $r2 = 2 * n\hat{c}$, $r3 = 2 * (n\hat{c} + 1)$ and
$r4 = 2 * (n - n\hat{c})$ [93].

In our analysis of coverage, we consider both the interval estimator in (5.12) based on the F distribution, as well as the traditional interval estimator (5.11) based on the normal distribution.

5.3.2 Model of SPECjAppServer2001's Order Entry Application

In Chapter 4 (Section 4.3), we built a QPN model of SPECjAppServer2001's order entry application, analyzed it using analytical techniques and validated it against measurements on the real system. We now consider the same model again, but this time we analyze it through simulation using SimQPN. We then compare results obtained from the simulation with the analytical solution presented in Chapter 4. The model we consider is depicted in Figure 5.4. For a detailed description of places and tokens of the model refer to Section 4.3 of Chapter 4.

The following input parameters need to be supplied before the model can be analyzed:

- Number of requests (i.e. clients) of each request class in the initial marking.
- Service times of request classes at the queues of places WLS-CPU, DBS-CPU and DBS-I/O.
- Average client think time (service time at the queue of place Client).
- Number of WLS threads (tokens 't'), JDBC connections (tokens 'c') and Oracle server processes (tokens 'p') in the initial marking.

In Chapter 4, we studied 3 different scenarios (instances of the model) varying the above parameters. Here we only consider the first one, which is the one with the

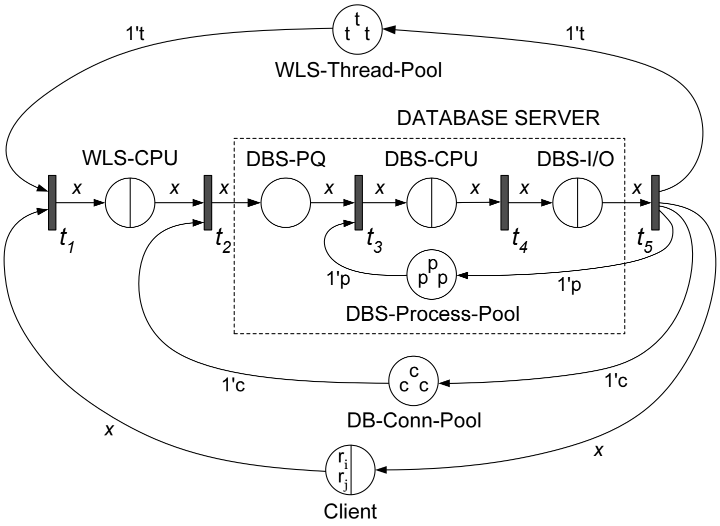


Figure 5.4: QPN model of SPECjAppServer2001's order entry application.

highest number of tokens. The conclusions we draw apply equally well to the other two scenarios. We assume that there are 80 NewOrder clients in the system with average think time of 200ms and there are 60 WLS threads, 40 JDBC connections and 30 DBS processes available.

Tables 5.1 and 5.2 report the results from simulating the model using SimQPN. The method of batch means was used for steady state analysis and the simulation was stopped as soon as the half widths of all 90% confidence intervals for residence times dropped below 5% of the respective point estimates (relative precision stopping criterion). The length of the warm-up period (determined through the method of Welch) was $6 * 10^6$ms (model time) and the total run duration was 9 seconds (wall clock time) on a machine with a 2 GHz CPU. The results are compared with the exact results from the analytical solution in Chapter 4, obtained using the *structured SOR* analysis method. For each queue and depository, the estimated steady state token population ($\mathcal{N}$), utilization ($\mathcal{U}$), throughput ($\mathcal{X}$) and residence time ($\mathcal{R}$) are reported. For residence times, in addition, 90% confidence intervals are provided. We have used subscripts 'Q' and 'D' to distinguish between queues and depositories of queueing places.

Tables 5.1 and 5.2 show the results from a single simulation run. However, as in any simulation, results vary from run to run. To measure this variation and evaluate the confidence interval coverage, we made multiple replications of the above simulation run as described in Section 5.3.1. We stopped as soon as enough data was available in order to provide, for each confidence interval in the simulation results, a 95% confidence interval for the true coverage with absolute half-width less than 0.05. Tables 5.3 and 5.4 present the results from our analysis. We skip the results for throughput and utilization, since the analysis showed that their variation was negligible. As discussed in Section 5.3.1, we include two interval estimates of the true coverage (95% confidence intervals) - the first one based on the normal distribution and the second one based on the F distribution. As expected, the confidence intervals based on the F distribution are slightly wider (i.e. more conservative) than

Table 5.1: Token population ($\mathcal{N}$) and utilization ($\mathcal{U}$) results for the QPN model of SPECjAppServer2001's order entry application from a single simulation run.

	$\mathcal{N}$		$\mathcal{U}$	
PLACE	Anal.	Sim.	Anal.	Sim.
Client$_Q$	2.857	2.864	0.942	0.943
Client$_D$	17.143	17.136	1.000	1.000
WLS-CPU$_Q$	56.676	56.658	1.000	1.000
DBS-CPU$_Q$	3.116	3.134	0.757	0.761
DBS-I/O$_Q$	0.207	0.208	0.171	0.172
WLS-Thread-Pool	0.000	0.000	0.000	0.000
DB-Conn-Pool	36.676	36.658	1.000	1.000
DBS-Process-Pool	26.676	26.658	1.000	1.000

Table 5.2: Throughput ($\mathcal{X}$) and residence time ($\mathcal{R}$) results for the QPN model of SPECjAppServer2001's order entry application from a single simulation run.

	$\mathcal{X}$ [requests/sec]		$\mathcal{R}$ [ms]	
PLACE	Anal.	Sim.	Anal.	Sim. (90% c.i.)
Client$_Q$	14.286	14.309	200.00	200.11 (+/- 00.84)
Client$_D$	14.286	14.309	1199.97	1197.46 (+/- 05.73)
WLS-CPU$_Q$	14.286	14.309	3967.25	3958.87 (+/- 19.04)
DBS-CPU$_Q$	14.286	14.309	218.15	218.97 (+/- 05.25)
DBS-I/O$_Q$	14.286	14.309	14.48	14.51 (+/- 00.05)

the traditional ones based on the normal distribution. Repeating the above analysis for different variations of the model (with modified input parameters) led to similar results in terms of the precision of the point and interval estimates.

5.3.3 Product-form Queueing Network

The next model we consider is a QN model taken from the examples shipped with the *PEPSY-QNS* tool (Performance Evaluation and Prediction SYstem for Queueing NetworkS) [25]. The example we consider is called *e_bcmp2* and is shown in Figure 5.5. It is a closed product-form QN with two request classes. We first translate the QN into a QPN and then analyze the latter through simulation using SimQPN. We compare results obtained from the simulation with the analytical solution provided by PEPSY. Finally, as in the previous case, we analyze the variation of point estimates and the confidence interval coverage.

Table 5.3: Experimental analysis of residence time variation and coverage of 90% conf. intervals for the QPN model of SPECjAppServer2001's order entry application from 1430 runs.

	Variation		Coverage Point/Interval (95% c.i.) Estimates		
PLACE	Mean	St.Dev.	Pt.Est.	Int.Est.(N-dist.)	Int.Est.(F-dist.)
Client_Q	199.99	0.44	0.901	0.901 +/- 0.015	[0.885, 0.916]
Client_D	1199.98	3.18	0.902	0.902 +/- 0.015	[0.886, 0.917]
WLS-CPU_Q	3967.14	11.30	0.896	0.896 +/- 0.016	[0.879, 0.912]
DBS-CPU_Q	218.08	3.50	0.889	0.889 +/- 0.017	[0.871, 0.906]
DBS-I/O_Q	14.48	0.03	0.898	0.898 +/- 0.015	[0.881, 0.913]

Table 5.4: Experimental analysis of residence time variation and coverage of 95% conf. intervals for the QPN model of SPECjAppServer2001's order entry application from 3820 runs.

	Variation		Coverage Point/Interval (95% c.i.) Estimates		
PLACE	Mean	St.Dev.	Pt.Est.	Int.Est.(N-dist.)	Int.Est.(F-dist.)
Client_Q	199.99	0.58	0.948	0.948 +/- 0.007	[0.941, 0.955]
Client_D	1200.04	4.10	0.947	0.947 +/- 0.007	[0.940, 0.954]
WLS-CPU_Q	3967.53	14.67	0.939	0.939 +/- 0.007	[0.931, 0.947]
DBS-CPU_Q	218.05	4.60	0.933	0.933 +/- 0.008	[0.925, 0.941]
DBS-I/O_Q	14.48	0.03	0.945	0.945 +/- 0.007	[0.937, 0.952]

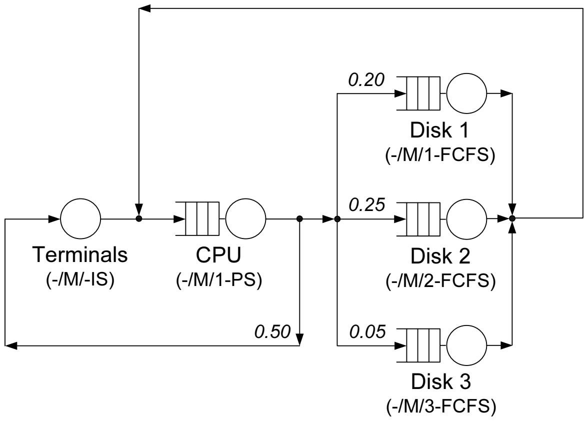


Figure 5.5: Product-form QN.

The mean service times of requests at the various queues of the model are given in Table 5.5 (all times are in milliseconds). Service times are exponentially distributed. There are 10 requests of class 1 and 12 of class 2. Mapping the QN to an equivalent QPN is straightforward and the resulting QPN is shown in Figure 5.6. Basically, every queue is mapped to a queueing place and request classes are mapped to token colors. Connected queues in the QN have their respective queueing places connected through transitions in the QPN.

Tables 5.6 and 5.7 show the results from simulating the product-form QN (more precisely its equivalent QPN) using SimQPN. Again, the method of batch means was used for steady state analysis and the simulation was stopped as soon as the half widths of all 90% confidence intervals for residence times dropped below 5% of the respective point estimates (relative precision stopping criterion). The length of the warm-up period (determined through the method of Welch) was $16 * 10^6$ms (model time) and the total run duration was 65 seconds (wall clock time) on a

Table 5.5: Mean service times of requests at the queues of the product-form QN.

Request Class	CPU	Disk 1	Disk 2	Disk 3	Terminals
Class 1	200	1000	500	20000	10000
Class 2	250	1000	500	20000	10000

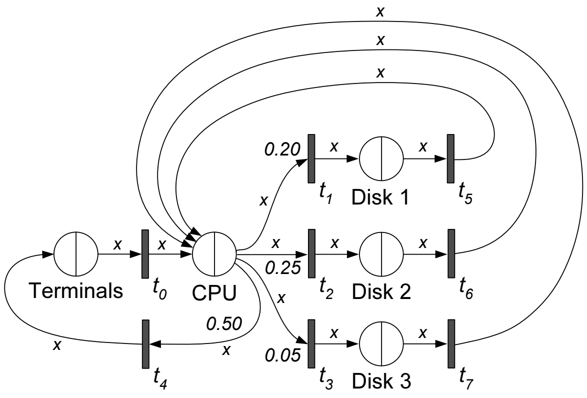


Figure 5.6: QPN equivalent to the product-form QN.

machine with a 2 GHz CPU. The results are compared with the exact results from the analytical solution provided by PEPSY. For each queue, the estimated steady state population ($\mathcal{N}$), throughput ($\mathcal{X}$) and residence time ($\mathcal{R}$) are reported. For residence times, in addition, 90% confidence intervals are provided.

As in the previous case, to evaluate the variation of point estimates and the

Table 5.6: Queue population ($\mathcal{N}$), throughput ($\mathcal{X}$) and residence time ($\mathcal{R}$) results for the product-form QN from a single simulation run.

	$\mathcal{N}$		$\mathcal{X}$ [requests/sec]		$\mathcal{R}$ [ms]	
PLACE	Anal.	Sim.	Anal.	Sim.	Anal.	Sim. (90% c.i.)
	Request Class 1					
CPU	0.592	0.594	1.241	1.243	476.7	477.8 (+/- 001.9)
Disk 1	0.510	0.511	0.248	0.249	2055.0	2056.1 (+/- 012.1)
Disk 2	0.159	0.160	0.310	0.311	513.6	513.8 (+/- 000.3)
Disk 3	2.535	2.535	0.062	0.062	40857.0	40926.5 (+/- 435.3)
Terminals	6.204	6.200	0.620	0.621	10000.0	9985.4 (+/- 018.2)
	Request Class 2					
CPU	0.864	0.866	1.468	1.468	588.7	589.6 (+/- 002.4)
Disk 1	0.604	0.603	0.294	0.293	2056.0	2054.2 (+/- 012.5)
Disk 2	0.188	0.189	0.367	0.367	513.6	513.6 (+/- 000.3)
Disk 3	3.005	3.013	0.073	0.073	40941.0	40947.1 (+/- 417.3)
Terminals	7.339	7.330	0.734	0.734	10000.0	9989.2 (+/- 016.5)

confidence interval coverage, we made multiple replications of the above simulation run and applied the coverage analysis method in Section 5.3.1. The stopping criterion was the same as for the previous model. Tables 5.8 and 5.9 present the results from our analysis. Repeating the evaluation for different variations of the model led to similar results with no degradation in the precision of the point and interval estimates. In Table 5.10, we present the results for one such variation, in which the service times of requests at the "Terminals" queue (i.e. the client think times) were reduced from 10000ms to 5000ms, leading to *overloading* Disk 3. As expected, most affected by this change were the residence times at Disk 3, which increased by over

Table 5.7: Utilization ($\mathcal{U}$) results for the product-form QN from a single simulation run.

	$\mathcal{U}$	
PLACE	Anal.	Sim.
CPU	0.615	0.616
Disk 1	0.542	0.541
Disk 2	0.169	0.170
Disk 3	0.903	0.904
Terminals	1.000	1.000

Table 5.8: Experimental analysis of residence time variation and coverage of 90% conf. intervals for the product-form QN from 2398 runs.

	Variation		Coverage Point/Interval (95% c.i.) Estimates		
PLACE	Mean	St.Dev.	Pt.Est.	Int.Est.(N-dist.)	Int.Est.(F-dist.)
	Request Class 1				
CPU	476.7	1.1	0.888	0.888 +/- 0.012	[0.875, 0.901]
Disk 1	2054.4	6.4	0.902	0.902 +/- 0.012	[0.890, 0.914]
Disk 2	513.6	0.2	0.903	0.903 +/- 0.012	[0.890, 0.914]
Disk 3	40854.1	226.9	0.911	0.911 +/- 0.012	[0.898, 0.923]
Terminals	9999.6	10.1	0.904	0.904 +/- 0.012	[0.891, 0.915]
	Request Class 2				
CPU	588.8	1.3	0.887	0.887 +/- 0.013	[0.873, 0.899]
Disk 1	2056.4	6.3	0.909	0.909 +/- 0.011	[0.897, 0.920]
Disk 2	513.6	0.2	0.894	0.894 +/- 0.012	[0.881, 0.906]
Disk 3	40937.1	226.1	0.907	0.907 +/- 0.012	[0.894, 0.919]
Terminals	10000.3	9.4	0.896	0.896 +/- 0.012	[0.883, 0.908]

177%. This is because, at this load, Disk 3 is completely saturated (its utilization is over 99%), leading to long waiting times in the queue. Residence times at the other queues were not as much affected by the change, since requests have much lower service demands for them (see Table 5.5) and in spite of the heavier load, they were

Table 5.9: Experimental analysis of residence time variation and coverage of 95% conf. intervals for the product-form QN from 4665 runs.

	Variation		Coverage Point/Interval (95% c.i.) Estimates		
PLACE	Mean	St.Dev.	Pt.Est.	Int.Est.(N-dist.)	Int.Est.(F-dist.)
	Request Class 1				
CPU	476.7	1.2	0.936	0.936 +/- 0.007	[0.928, 0.943]
Disk 1	2054.6	6.9	0.947	0.947 +/- 0.006	[0.941, 0.953]
Disk 2	513.6	0.2	0.948	0.948 +/- 0.006	[0.941, 0.954]
Disk 3	40857.8	240.6	0.950	0.950 +/- 0.006	[0.943, 0.957]
Terminals	10000.1	10.8	0.944	0.944 +/- 0.006	[0.937, 0.951]
	Request Class 2				
CPU	588.7	1.4	0.932	0.932 +/- 0.007	[0.924, 0.939]
Disk 1	2056.5	6.7	0.951	0.951 +/- 0.006	[0.945, 0.957]
Disk 2	513.6	0.2	0.944	0.944 +/- 0.006	[0.937, 0.951]
Disk 3	40942.9	238.6	0.946	0.946 +/- 0.006	[0.939, 0.953]
Terminals	10000.1	9.6	0.957	0.957 +/- 0.005	[0.951, 0.963]

Table 5.10: Experimental analysis of residence time variation and coverage of 95% conf. intervals for the product-form QN under heavy load from 4300 runs.

	Variation		Coverage Point/Interval (95% c.i.) Estimates		
PLACE	Mean	St.Dev.	Pt.Est.	Int.Est.(N-dist.)	Int.Est.(F-dist.)
	Request Class 1				
CPU	591.1	3.2	0.933	0.933 +/- 0.007	[0.925, 0.941]
Disk 1	2403.1	13.2	0.943	0.943 +/- 0.007	[0.936, 0.950]
Disk 2	517.5	0.2	0.953	0.953 +/- 0.006	[0.947, 0.960]
Disk 3	72561.5	401.1	0.952	0.952 +/- 0.006	[0.945, 0.958]
Terminals	5000.1	5.3	0.948	0.948 +/- 0.006	[0.941, 0.955]
	Request Class 2				
CPU	726.0	3.9	0.933	0.933 +/- 0.007	[0.925, 0.941]
Disk 1	2405.5	13.1	0.946	0.946 +/- 0.007	[0.938, 0.953]
Disk 2	517.6	0.2	0.948	0.948 +/- 0.007	[0.940, 0.954]
Disk 3	72857.6	398.7	0.950	0.950 +/- 0.007	[0.943, 0.956]
Terminals	4999.9	4.8	0.951	0.951 +/- 0.006	[0.944, 0.958]

still under 70% utilized (the CPU was about 67% utilized, Disk 1 about 60% and Disk 2 still less than 20%). In all cases, the estimated coverage of 90% and 95% confidence intervals did not drop below 88% and 93%, respectively.

5.3.4 Model of SPECjAppServer2002

In Chapter 4 (Section 4.2), we built a QN model of SPECjAppServer2002 that spanned the whole benchmark application. This was a non-product-form model and we were only able to analyze it using analytical approximation methods (more specifically, we used the *multisum method* [23, 25]). However, when increasing the number of customers interacting with the system, even approximation methods started to fail. We now consider the same model again, but this time we analyze it through simulation using SimQPN. We compare results obtained from the simulation with the approximate results presented in Chapter 4. We also consider the cases where approximation methods were failing, and in these cases, we compare the simulation results with measurements on the real system modeled. The QN model from Chapter 4 is depicted in Figure 5.7. It is a closed model with five request classes: NewOrder (NO), ChangeOrder (CO), OrderStatus (OS), CustStatus (CS) and WorkOrder (WO). For detailed information on the model queues and requests refer to Section 4.2 of Chapter 4.

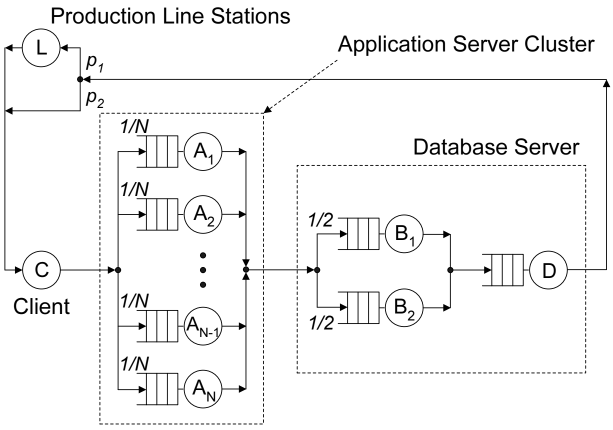


Figure 5.7: QN model of SPECjAppServer2002.

The following input parameters need to be supplied before the model can be analyzed:

- Number of WebLogic servers N.
- Number of order entry clients (NewOrder, ChangeOrder, OrderStatus and CustStatus).
- Average think time of order entry clients - *Customer Think Time*.
- Number of planned production lines generating WorkOrder requests.
- Average time production lines wait after processing a work order before starting a new one - *Manufacturing (Mfg) Think Time*.

Each set of values for these parameters generates a different instance of the model. In Chapter 4, we considered three scenarios (see Table 5.11) representing low, moderate and heavy load, respectively. The number of WebLogic servers was ranging from 1 to 9. Here we only consider the moderate and heavy load scenarios, since they are the largest and most problematic ones as far as analysis is concerned. Again, we translate the QN into an equivalent QPN by mapping queues to queueing places and connecting them through transitions. The resulting QPN is shown in Figure 5.8. Note that, compared to the QPN in Chapter 4 (Section 4.3), this is a huge QPN (considering token population and colors) and even for the simplest scenario (low load) trying to analyze it by means of conventional techniques results in explosion of the underlying state space.

Table 5.11: Model input parameters for the 3 scenarios considered in Chapter 4.

Parameter	Low	Moderate	Heavy
NewOrder Clients	30	50	100
ChangeOrder Clients	10	40	50
OrderStatus Clients	50	100	150
CustStatus Clients	40	70	50
Planned Lines	50	100	200
Customer Think Time	2 sec	2 sec	3 sec
Mfg Think Time	3 sec	3 sec	5 sec

Tables 5.12 and 5.13 summarize the results from 500 simulation runs of the moderate load scenario with 6 WebLogic servers. Each run took approximately 5 minutes on a machine with a 2 GHz CPU. For every request class, the mean and standard deviation of observed throughputs (in requests/sec) and residence times at queues A_i, B_j and D (in milliseconds) are reported. The simulation results are

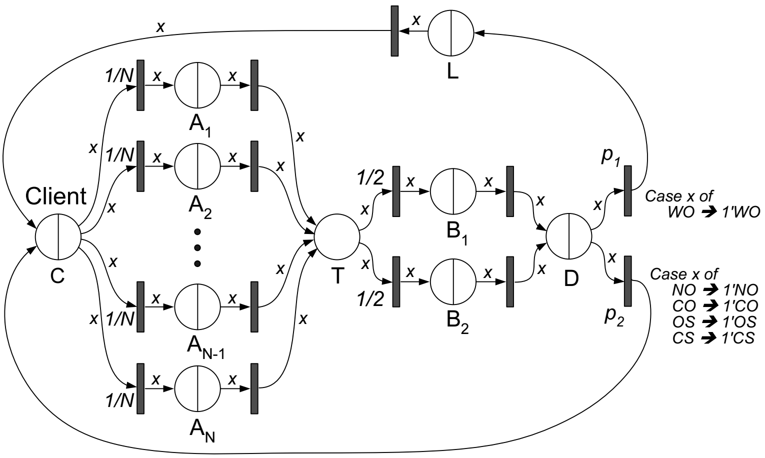


Figure 5.8: QPN model of SPECjAppServer2002.

compared against the approximate results presented in Chapter 4. The latter were obtained using the *multisum* analytical approximation method supported by the PEPSY tool. As we can see, results from the simulation are consistent with the approximate results and have very low variation. Since the exact values of the

Table 5.12: Residence time ($\mathcal{R}$), throughput ($\mathcal{X}$) and utilization ($\mathcal{U}$) results for scenario 2 with 6 WebLogic servers from 500 simulation runs - Part 1.

	A_i (WLS-CPU)			B_j (DBS-CPU)		
Metric	Anal.	Sim.	St.Dev.	Anal.	Sim.	St.Dev.
R_{NO}	17	16.8	0.22	40	39.2	0.73
R_{CO}	18	17.7	0.24	39	38.1	0.73
R_{OS}	3	3.4	0.03	9	9.2	0.16
R_{CS}	3	3.3	0.03	8	7.7	0.13
R_{WO}	31	31.4	0.42	129	124.9	2.69
X_{NO}	4.05	4.06	≤ 0.01	12.15	12.15	≤ 0.01
X_{CO}	3.24	3.24	≤ 0.01	9.72	9.72	≤ 0.01
X_{OS}	8.28	8.28	≤ 0.01	24.83	24.83	≤ 0.01
X_{CS}	5.80	5.80	≤ 0.01	17.40	17.40	≤ 0.01
X_{WO}	4.00	4.01	≤ 0.01	12.01	12.03	≤ 0.01
U	0.23	0.23	≤ 0.01	0.74	0.74	≤ 0.01

estimated parameters are not known (exact analytical solution of the model is not available), coverage analysis for confidence intervals does not make sense in this case.

Table 5.13: Residence time ($\mathcal{R}$), throughput ($\mathcal{X}$) and utilization ($\mathcal{U}$) results for scenario 2 with 6 WebLogic servers from 500 simulation runs - Part 2.

	D (DBS-Disk)		
Metric	Anal.	Sim.	St.Dev.
R_{NO}	1	1.3	≤ 0.01
R_{CO}	1	1.4	≤ 0.01
R_{OS}	1	0.8	≤ 0.01
R_{CS}	0	0.4	≤ 0.01
R_{WO}	2	1.9	≤ 0.01
X_{NO}	24.29	24.31	≤ 0.01
X_{CO}	19.43	19.45	≤ 0.01
X_{OS}	49.67	49.66	≤ 0.01
X_{CS}	34.80	34.80	≤ 0.01
X_{WO}	24.02	24.05	≤ 0.01
U	0.13	0.13	≤ 0.01

Table 5.14: Response time ($\mathcal{R}$), throughput ($\mathcal{X}$) and utilization ($\mathcal{U}$) results for scenario 3 with 6 and 9 WebLogic servers from 500 simulation runs.

	6 App. Servers			9 App. Servers		
METRIC	Anal.	Sim.	Msrd.	Anal.	Sim.	Msrd.
R_{NO}	-	98	94	-	95	81
R_{CO}	-	97	98	-	94	84
R_{OS}	-	23	27	-	22	24
R_{CS}	-	20	27	-	19	25
R_{WO}	-	286	251	-	282	215
X_{NO}	32.22	32.28	32.66	32.24	32.31	32.48
X_{CO}	16.11	16.15	16.19	16.12	16.15	16.18
X_{OS}	49.60	49.62	49.21	49.61	49.64	49.28
X_{CS}	16.55	16.56	16.24	16.55	16.56	16.46
X_{WO}	31.72	31.82	32.08	31.73	31.83	32.30
$U_{WLS-CPU}$	26.5%	26.4%	29%	17.8%	17.6%	20%
$U_{DBS-CPU}$	86.1%	87.7%	91%	86.2%	87.7%	91%

We now repeat the same analysis for the heavy load scenario with 6 and 9 WebLogic servers. The average run duration was 12 minutes. Results are summarized in Table 5.14. For each request class, we consider its total response time and throughput. Note that by *response time* we mean the total amount of time needed for processing a request, i.e. the sum of its residence times at queues A_i (WLS-CPU), B_j (DBS-CPU) and D (DBS-Disk). Unfortunately, available approximation methods fail to provide reliable response time estimates for models of this size. Therefore, this time the analytical results only include throughput and utilization. To validate response time results, we compare them against measurements taken on the real system that the model represents. Note that the expected deviation here is higher, since the model is only an approximation of the system, and as discussed in Chapter 4, it has some inherent limitations. Nevertheless, we see that results obtained from the simulation are close to the actual values measured on the system modeled. Repeating the analysis for the other configurations considered in Chapter 4 led to results of similar accuracy.

5.4 Concluding Remarks

This chapter showed how the problem of analyzing large QPN models can be approached by exploiting discrete event simulation for model analysis. We presented SimQPN - our simulation tool for QPNs, and discussed its features, design and architecture. In parallel to this, we presented our methodology for simulating QPN models based on which SimQPN was developed. The methods for output data analysis used in SimQPN were presented and the specifics of their implementation were discussed. It was shown how SimQPN exploits the knowledge of the structure and behavior of QPNs to improve the efficiency of the simulation.

We validated our approach by applying it to study several different QPN models. In each case, we validated the simulation results by comparing them with results obtained using other methods, i.e. analytical methods, approximation methods or measurements on the system modeled. Models of different size and complexity ranging from simple models to large and complex models of realistic systems were considered. Each model was analyzed for different variations of its input parameters. The variability of output data provided by SimQPN and the coverage of confidence intervals reported were subjected to a rigorous experimental analysis. Results showed that data reported by SimQPN is pretty accurate and stable. Even for residence time, the metric with highest variation, the standard deviation of point estimates did not exceed 2.5% of the mean value. In all cases, the estimated coverage of confidence intervals was less than 2% below the nominal value (higher than 88% for 90% confidence intervals and higher than 93% for 95% confidence intervals). In addition to this, SimQPN proved to be pretty fast in terms of measured CPU running times. A simulation run for the simple models in Sections 5.3.2 and 5.3.3

on a 2GHz CPU took less than one minute on average. Even for the much larger models in Section 5.3.4, CPU running times did not exceed 12 minutes.

SimQPN provides a portable simulation engine for QPNs. Being specialized for QPNs, it is extremely light-weight and fast. It can be used to analyze QPN models of realistic size and complexity, making it possible to exploit the modeling power and expressiveness of the QPN paradigm to its full potential as a performance prediction tool. In the next chapter, we develop a practical performance modeling methodology for DCS based on QPN models. The methodology helps to construct models of DCS that accurately reflect their performance and scalability characteristics and can be exploited for performance prediction in the software engineering process.

Chapter 6

Performance Modeling Methodology

There is no other way to handle the complexity than by breaking it up into manageable pieces.
– *Bruce Schneier*

Make everything as simple as possible, but not simpler!
– *Albert Einstein*

6.1 Introduction

Performance models are a very powerful tool for performance analysis of DCS and are used increasingly during system development to predict the expected performance of the system under load. However, as shown in Chapter 4, building models that accurately capture the different aspects of system behavior is a very challenging task when applied to realistic systems. In this chapter, we present a practical performance modeling methodology for DCS which helps to construct models that accurately reflect the performance and scalability characteristics of the latter. Our methodology builds on the methodologies proposed by Menascé, Almeida and Dowdy in [108, 109, 110, 111, 112], however, a major difference is that our methodology is based on Queueing Petri Net (QPN) models as opposed to conventional Queueing Network (QN) models and it is specialized for DCS. QPN models are more sophisticated than QN models and enjoy greater modeling power and expressiveness. Taking advantage of this, our methodology provides the following important benefits:

1. As shown in the previous chapters, QPN models allow the integration of hard-

ware and software aspects of system behavior and lend themselves very well to modeling DCS.

2. In addition to hardware contention and scheduling strategies, using QPNs one can easily model software contention, simultaneous resource possession, synchronization, blocking and asynchronous processing. These aspects of system behavior, which are typical for modern DCS, are difficult to model accurately using conventional QN models.

3. By restricting ourselves to QPN models, we can exploit the knowledge of their structure and behavior for fast and efficient simulation using SimQPN. As shown in Chapter 5, this enables us to analyze models of large and complex DCS and ensures that our approach scales to realistic systems.

4. QPNs can be used to combine qualitative and quantitative system analysis. A number of efficient techniques from Petri net theory can be exploited to verify some important qualitative properties of QPNs. The latter not only help to gain insight into the behavior of the system, but are also essential preconditions for a successful quantitative analysis [11].

After discussing our modeling methodology in general, we present a case study in which the latter is used to model a realistic system and analyze its performance and scalability. The system modeled is a deployment of the industry-standard SPECjAppServer2004 benchmark, presented in Chapter 2. A detailed model of the system and its workload is built in a step-by-step fashion. The model is validated and used to predict the system performance for several deployment configurations and workload scenarios of interest. In each case, the model is analyzed by means of simulation using SimQPN - our simulation tool presented in the previous chapter. In order to validate the approach, the model predictions are compared against measurements on the real system. In addition to CPU and I/O contention, it is demonstrated how some more complex aspects of system behavior, such as thread contention and asynchronous processing, can be modeled.

This chapter is organized as follows: Section 6.2 provides an overview of the performance modeling methodology and discusses each of its seven steps in detail. Following this, Section 6.3 presents our case study of SPECjAppServer2004 showing how each step of the modeling process is applied in practice. Finally, some concluding remarks are given in Section 6.4.

6.2 Methodology Overview

The methodology we propose includes the following steps:

1. Establish performance modeling objectives.

2. Characterize the system in its current state.
3. Characterize the workload.
4. Develop a performance model.
5. Validate, refine and/or calibrate the model.
6. Use model to predict system performance.
7. Analyze results and address modeling objectives.

It is important to note that the modeling process is iterative in nature and the above steps might have to be repeated multiple times as the system and workload evolve. Each of the steps is now discussed in detail.

6.2.1 Step 1: Establish performance modeling objectives.

The first step is to set some concrete goals for the performance modeling effort. The latter should be stated in a simple and precise manner. Modeling objectives can be classified in the following categories some of which partially overlap:

- *Performance Prediction:* Predict the performance of the system for a given workload and configuration scenario.
- *Performance Verification:* Verify that the system would be able to meet established Service Level Agreements.
- *Capacity Analysis:* Determine the maximum load that the system would be able to handle in its current state.
- *Scalability Analysis:* Study the performance of the system as the load increases and more hardware is added.
- *Bottleneck Analysis:* Find which system components are most utilized and investigate if they are potential bottlenecks.
- *Performance Tuning:* Study the effect of a particular deployment setting or tuning parameter on the system performance and find its optimal value.
- *Performance Optimization:* Find the components with the largest effect on performance and study the performance gains from optimizing them.
- *Cost/Performance Analysis:* Compare different system architectures and configuration alternatives in terms of their cost/performance ratios.
- *Sizing and Capacity Planning:* Determine the amount of hardware that would be needed to guarantee certain performance levels.

6.2.2 Step 2: Characterize the system in its current state.

In this step, the system is described in detail in terms of its hardware and software architecture. The product of this step is a specification that includes the following information:

- System architecture/topology (system tiers, communication links, etc).
- Types of servers used (e.g. Web servers, application servers, database servers).
- Type of hardware used (e.g. server machines, disk subsystems, load balancers, backup systems).
- Type of software used (e.g. operating systems, middleware, transaction monitors, DBMS, application software).
- Communication equipment and network protocols used (e.g. routers, firewalls, switches).

The goal here is to obtain an in-depth understanding of the system architecture and its components. The latter is essential for building representative models.

6.2.3 Step 3: Characterize the workload.

In this step, the workload of the system under study is described in a qualitative and quantitative manner. This is called *workload characterization* [35] and includes five major steps:

1. Identify the basic components of the workload.
2. Partition basic components into workload classes (workload partitioning).
3. Identify the system components and resources (hardware and software) used by each workload class.
4. Describe the inter-component interactions and processing steps for each workload class.
5. Characterize workload classes in terms of their service demands and workload intensity.

The steps above are now discussed in detail.

Identify the Basic Components of the Workload

Basic component refers to a generic unit of work that arrives at the system from an external source [108]. Some examples include HTTP requests, remote procedure calls, Web service invocations, database transactions, interactive commands, batch jobs, etc. Basic components could also be composed of multiple processing tasks, for e.g. client sessions comprising multiple requests to the system, open and closed nested transactions, etc. The choice of basic components and the decision how granular they are defined depend on the nature of the services provided by the system and on the modeling objectives. Since, in almost all cases, basic components can be considered as some kind of requests or transactions processed by the system, we will often refer to them as *requests* or *transactions*[1]. Similarly, basic components comprising multiple processing tasks will be referred to as *composite transactions* and their individual processing tasks will be called *subtransactions*.

Partition Basic Components into Workload Classes

The basic components of real workloads are typically heterogeneous in nature. In order to improve the representativeness of the workload model and increase its predictive power, the basic components must be partitioned into classes (called *workload classes*) that have similar characteristics. The partitioning can be done based on different criteria, depending on the type of system modeled and the goals of the modeling effort. Some typical criteria are quickly described [108, 116]:

- Applications: Classes are derived based on applications involved, e.g. online ordering, inventory management, supply-chain management.
- Functional: Classes are derived based on the functions being served, for e.g. place new order, change existing order, browse product catalogue.
- Objects handled: Classes are derived based on the type of objects handled by the applications, for e.g. a Web server workload can be partitioned by the type of documents accessed: HTML, Image, Sound, Video, etc.
- Resource consumption: Classes are derived based on resource consumption, for e.g. download requests may be divided into three classes based on the size of the file requested: small ($\leq$ 500K), medium (500K-5MB) and large ($\geq$ 5MB).
- Workload intensity: Classes are derived based on the number of concurrent units of work (i.e. requests or transactions) that contend for system resources. Workload intensity will be discussed in detail later in the last step of the workload characterization process.

[1] The term transaction here is used loosely to refer to any unit of work or processing task executed in the system.

- Processing mode: Classes are derived based on the mode of processing: synchronous or asynchronous.

- Geographical: Classes are derived based on geographical orientation, for e.g. HTTP requests may be broken down into classes based on where they originate from: USA, Europe, Asia, etc.

Multiple of the above criteria might be used to partition basic components into workload classes. To improve model representativeness, it is recommended that already at this stage the workload be partitioned in such a way that each workload class is as homogeneous as possible in terms of the load it places on the system and its resources. Note that a workload class might later have to be split into several workload classes if it turns out to exhibit large variability in terms of measured resource usage. This will be revisited when the last step of the workload characterization process is discussed. Workload classes are sometimes also referred to as *request classes* or *transaction classes.*

Identify the System Components and Resources Used by Each Workload Class

In this step, for each workload class the system components and resources used (hardware and software) are identified. For example, an online request to place an order might require using a Web server, application server and backend database server. For each of these servers, the concrete hardware and software resources used must be specified. Hardware resources could be CPUs, disk drives, network links, etc. Software resources could be transaction monitors, database management systems, message-oriented middleware, application software, etc. However, software resources could also be operating system processes, threads, semaphores, database connections, locks, latches, etc. Again, the level of detail here is determined by the system type and modeling objectives. It is distinguished between *active* and *passive* resources [111]. An active resource is a resource that delivers a certain service to transactions at a finite speed. In contrast, a passive resource is needed for the execution of a transaction, but is not characterized by a speed of service delivery. Once a transaction acquires the required number of instances of a passive resource, its execution progresses at a rate that is independent of any characteristic of the resource. Active resources are normally hardware devices, for e.g. CPUs or disk drives, while passive resources are usually software resources (for e.g. processes, threads, database connections or locks), but could also be hardware resources, a common example being main memory.

Describe the Inter-Component Interactions and Processing Steps for Each Workload Class

The aim of this step is to describe the processing steps, the inter-component interactions and flow of control for each workload class. Also for each processing step, the hardware and software resources used must be specified. Different notations may be exploited for this purpose, for example Client/Server Interaction Diagrams (CSID) [110], Communication-Processing Delay Diagrams [108], Execution Graphs [146], as well as conventional UML Sequence and Activity Diagrams [26].

Figure 6.1 shows an example of a CSID describing the inter-component interactions and flow of control for a customer registration transaction. The diagram is made up of nodes (squares and circles) and directed arcs connecting the nodes. Nodes represent clients and servers visited during the execution of the transaction. Every node is identified by a unique number indicated by a label next to the node. Square nodes represent clients, while circle nodes represent servers used during processing. All transactions start at node 1 (called the start node) and end in another square node (called the end node). Directed arcs show the flow of control from one node to the next during execution. The diagram also shows which communication networks are used for inter-component communication and the average sizes (depending on whether such information is available) of the messages exchanged between the components. The latter are shown in the square brackets over the arrows.

Characterize Workload Classes in Terms of Their Service Demands and Workload Intensity

In this step, the load placed by the workload classes on the system must be quantified. Two sets of parameters need to be specified for each workload class:

- Service demand parameters.
- Workload intensity parameters.

Service demand parameters specify the total amount of service time required by each workload class at each resource. For example, the CPU time of order-entry transactions at the application server, their CPU and I/O time at the database server, etc. Note that service demands refer only to times spent receiving service at the resources, i.e. they do not include waiting times. Depending on the level of detail at which the system is modeled, service demands may also be considered for the individual processing steps (subtransactions) of workload classes. For example, the CPU time at the application server for sending a notification to the customer at the end of an order-entry transaction, confirming that the order has been received.

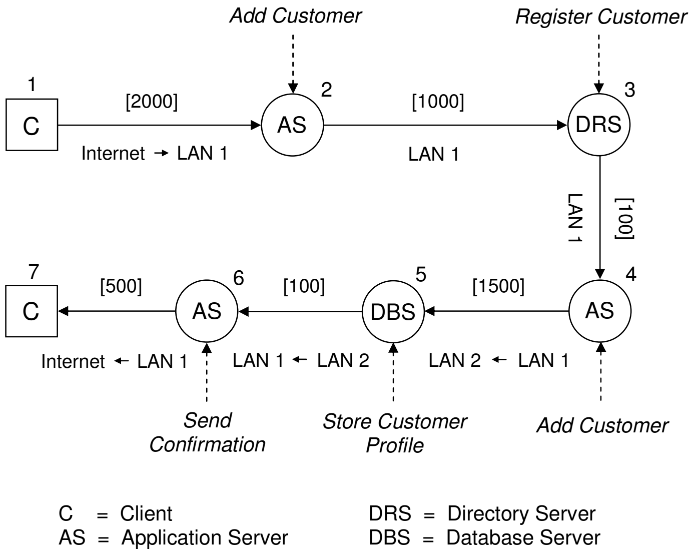


Figure 6.1: Example of CSID for customer registration transaction.

Workload intensity parameters provide for each workload class a measure of the number of units of work, i.e. requests or transactions, that contend for system resources. For example, the number of order-entry transactions executed per unit of time, the average number of batch jobs processed concurrently, etc. Depending on the way workload intensity is specified, workload classes may be classified as open or closed:

- *Open class*: Workload intensity is specified by an arrival rate (λ), i.e. average number of requests that arrive per unit of time. The arrival rate is usually independent of the system state.

- *Closed class*: Workload intensity is specified by the average number of requests (N) served concurrently in the system. Alternatively, workload intensity can be specified by the number of active clients/terminals (M) that send requests to the system and their average think time (Z).

The decision whether a given class is modeled as open or closed (i.e. how its workload intensity is specified) is influenced by several factors, including the nature of the real workload, the amount of information about it available (e.g. measurement data) and the data required as output from the model.

We now discuss some general techniques for obtaining service demand parameters. Most techniques involve running the system or components thereof and taking measurements. In case the system is not available for testing (for e.g. in the early stages of system development), techniques exist for estimating the service demand parameters [113]. In the following discussions, it is assumed that the system is available at least in a prototypical form, so that service demands can be obtained through measurements. The measurement process usually involves three major steps [108, 112, 141]:

1. *Select Variables to Measure*: First, the performance variables to be measured are selected. In most cases, it is not possible to measure service demands directly. Instead, some other parameters are measured from which service demands can be derived. The most typical case is to derive service demands from resource utilization and transaction throughput data. In certain cases, it might be possible to obtain service demands directly using specialized measurement tools.

2. *Collect Measurement Data*: The next step is to gather measurement data for the selected variables to be measured. This is usually done by means of performance monitoring and measurement tools, for example accounting systems, program analyzers, log generators or hardware monitors. Depending on the variables to be measured, several measurement tools may be required at different layers of the system environment, for example operating system, middleware, virtual machine or the application software layer. In some cases, the system might need to be instrumented manually to include performance measurement code.

3. *Analyze and Transform Measurement Data*: In this last step, the measurement data gathered is analyzed and transformed into meaningful information from which service demand parameters can be derived. Measurement tools often gather huge amounts of raw data, corresponding to a detailed log of the system activities during the observation period. This data has to be summarized and represented in a more compact form before it can be used. Two techniques often exploited in this context are *averaging* and *clustering*. If a workload class proves to exhibit large variability in terms of service demands, it is partitioned into several workload classes with more homogeneous populations.

To illustrate the above steps, imagine that we want to estimate the I/O service demand of order-entry transactions of an online ordering system. One way to do this would be to measure the number of I/O operations typically performed by order-entry transactions and multiply it by the average disk service time per I/O operation (assuming that service times of I/O operations performed have low variability). Assume that the system has been monitored for a period of one hour and a hundred thousand observations for the number of I/O operations per transaction have been recorded. The detailed measurement data has to be replaced with a more compact representation in order for it to be usable. One approach would be to compute the arithmetic mean of all observations collected and use it to derive the I/O service demand (averaging). However, if transactions exhibit large variability in terms of the number of I/O operations they perform, averaging all observations would likely produce a model that is not representative of most of the transactions. In such cases, observations are normally grouped into clusters, such that the variability within each cluster is relatively small compared to the variability in the entire set of observations (clustering). Thus, a workload class might need to be split into several workload classes with lower variability in resource usage. For a discussion of clustering techniques refer to [54, 68].

As mentioned, service demand parameters are most typically derived from measured resource utilization and transaction throughput data. The Service Demand Law is used [51]. The latter states that the service demand $D_{i,r}$ of class r transactions at resource i is equal to the average utilization $U_{i,r}$ of resource i by class r transactions divided by the average throughput $X_{0,r}$ of class r transactions during the measurement interval, i.e.

$$D_{i,r} = \frac{U_{i,r}}{X_{0,r}} \tag{6.1}$$

There are two approaches for estimating the service demands using the above relation:

- Conduct a single experiment, injecting transactions from all workload classes concurrently.
- Conduct a separate experiment for each workload class, injecting transactions only from a single class at a time.

If transactions from multiple workload classes are injected concurrently (approach 1), the partial utilization of the considered resource due to each workload class (i.e. $U_{i,r}$) must be determined. However, system monitors normally provide only total resource utilization statistics. Accounting systems or program analyzers are typically used to provide additional data, so that the measured total utilization can be apportioned among the different workload classes. Some methods for apportioning the total resource utilization among the workload classes are discussed in [108, 111, 112]. There are two problems with this approach. First, in almost all cases there is some resource usage (sometimes called *system overhead*), that is either not captured at all, or is captured, but not accounted to any workload class by the employed accounting system or program analyzer. It is usually hard to find a way to distribute the unattributed resource usage among the workload classes in a fair manner. Second, the overhead of accounting systems and program analyzers is often so high that they influence the system behavior significantly. This leads to higher resource utilization, which in turn results in higher service demand estimates not representative of the normal operating conditions. This is particularly true for JVM profilers that are normally exploited in the context of J2EE applications.

In the second approach, transactions are injected from a single workload class at a time, so that the measured total resource utilization is due to a single class and Equation (6.1) can be applied directly. Note that this automatically incorporates the operating system overhead into the service demand of the respective class. However, a potential problem with this approach is that, depending on the workload considered, the transactions from the various workload classes might behave differently when they are run concurrently as opposed to one at a time. In such cases, the interactions between different workload classes would not be captured in the workload model.

Which of the two approaches to estimate service demands is chosen depends on many factors including the type of system modeled, the workload considered, the measurement tools available, etc. In any case, the following should be kept in mind when conducting experiments to estimate service demands:

- Make sure that only transactions from the considered workload class(es) are injected.
- Make sure that the system resources are utilized only by the workload injected, i.e. that no foreign processes are running in the background that could affect the measurements. A common pitfall is to overlook a background process (for e.g. scanlogd daemon on Linux) that produces additional load and skews the measurements.
- Make sure that the monitoring and measurement tools used have negligible overhead, i.e. that they do not affect the measurements taken.
- Make sure that measured resources are reasonably utilized. It is recommended to have them at least 50% utilized, if possible.
- Repeat each experiment at least a couple of times to verify that measurements are stable.
- Run experiments under different load to ensure that service demands are not load dependent. Variations within +/- 5% are normally acceptable.

- Experiments should begin with a *ramp up period* that is long enough for the system to reach *steady state*. The measurement period should start after steady state has been reached and should continue until a reasonable number of transactions (possibly on the order of thousands) have been executed.
- Service demands for heavy loaded resources have greater effect on the model accuracy and therefore more care should be taken when estimating them.
- A good rule of thumb to verify the service demand estimates for a given workload class is the following: Under light load (with little or no concurrency), the measured response time should be roughly equal to the sum of the service demands of the respective workload class at all resources it uses.

This concludes our discussion of the workload characterization step. The output of this step is called *workload model* and is used as input for building a performance model in the next step.

6.2.4 Step 4: Develop a performance model.

In this step, a performance model is developed that represents the different components of the system and its workload, and captures the main factors affecting its performance. The performance model can be used to understand the behavior of the system and predict its performance under load. The methodology presented here exploits QPN models to take advantage of the modeling power and expressiveness of QPNs.

The way a performance model is constructed is based on the output from the previous three steps of the methodology, i.e. the modeling objectives, the system specification and the workload model. One of the greatest challenges in building a good model is to find the right level of detail. A general rule of thumb is: "Make the model as simple as possible, but not simpler!". Including too much detail might render the model intractable, on the other hand, making it too simple, might render it unrepresentative. The information from the workload characterization step is of critical importance when deciding on the level of detail at which different aspects of the system are modeled.

Normally, the modeling process includes the following steps in which the system and workload components are modeled using QPN constructs:

1. Model the system components and resources.
2. Model the basic components of the workload.
3. Model the inter-component interactions.
4. Parameterize the model.

The first step is to map the system components and resources (hardware and software) to respective QPN model constructs. Active resources are usually modeled using queueing places. Depending on the type of resource, queues with different scheduling strategies (queueing disciplines) may be used. For example, for processors (CPUs) usually PS scheduling strategy is used, while for secondary storage systems (e.g. disk drives) and networks, mostly FCFS scheduling strategy is used. Passive resources such as threads, processes, database connections and locks are normally modeled using tokens inside ordinary places. For example, a thread pool can be modeled using an ordinary place whose tokens represent threads. The initial population of the place determines the number of threads available in the pool. Whenever a thread is used, a token is removed from the place and after the thread is released, the token is returned back to the place.

The next step is to model the basic components of the workload, i.e. the transactions (requests) processed by the system. The latter are normally modeled using tokens, exploiting different colors to distinguish between workload classes. Some additional QPN constructs are needed to model the way transactions get started and completed in the system, or equivalently, the way requests arrive and depart from the system. Figure 6.2 illustrates how this is done for open and closed workload classes.

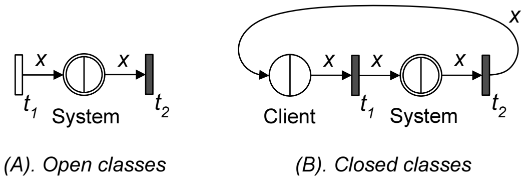


Figure 6.2: Modeling request/transaction arrivals and departures.

For open workload classes, two transitions, one timed (t_1) and one immediate (t_2) are needed. The timed transition does not have any input places and is by definition always enabled. Whenever it fires, it simply creates a new token and deposits it into the system, represented in the figure using a subnet place (a nested QPN). Tokens created correspond to newly started transactions (resp. newly arriving requests). The firing delay is chosen according to the desired transaction injection rate (resp. request arrival rate). The immediate transition, on the other hand, is used to model transaction completions (resp. request departures). It simply destroys tokens, i.e. transactions/requests whose processing has been completed.

For closed workload classes, a different mechanism is employed. A queueing place, called *Client*, with IS queue (i.e. delay resource) and two immediate transitions are used. Two cases are distinguished, based on whether workload intensity is specified as an average number of transactions being processed concurrently (requests being served concurrently), or it is specified as a number of active clients/terminals that start transactions (send requests) and their average think time. In the first case, tokens in the Client place are served immediately (i.e. the service time of the queue is zero) and its initial token population determines the number of concurrent transactions (resp. requests) in the system. In the second case, the Client place (or more precisely its queue) has service time equal to the average client think time and its initial token population determines the number of active clients/terminals starting transactions (sending requests). The immediate transitions are used to move tokens (i.e. transactions or requests) from the Client to the system and back.

Note that a single set of the above described QPN constructs can be used to model multiple workload classes, as long as all of them are of the same type (open or closed). As mentioned, different token colors can be used to distinguish between workload classes. However, it is recommended to use a separate set of constructs for every basic component of the workload.

The next step is to model the inter-component interactions and processing steps for transactions of the different workload classes. The latter were described in detail during workload characterization and their descriptions are used here as a starting point for their modeling. Interactions between system components are normally modeled using transitions connecting the QPN places corresponding to the respective components. Transitions are used to destroy and create tokens at the places of the QPN. A typical case is to destroy a token in one place and then create the same token in another place, which can be seen as moving the token from the first place to the second. This can be used, for example, to model the flow of control from one system component to another when processing a transaction.

For composite transactions, the individual processing steps (subtransactions) can also be modeled using tokens. One way to do this is to use a separate token color for every subtransaction. This is illustrated in Figure 6.3 for open and closed workload classes, respectively. An upper-case X token is used to represent a composite transaction. The individual subtransactions of the latter are represented using lower-case x tokens, where x_i stands for the i-th subtransaction. When the transaction is started (by firing transition t_1), a token x_1 representing the first processing step (subtransaction) is created and deposited into the system. After the subtransaction is completed, its token is destroyed (by transition t_3) and a token representing the next processing step x_{i+1} is created and deposited into the system. This process continues until the last subtransaction (modeled as token x_n) has been processed. Note that, to make things more illustrative, in the case of open classes arriving and departing X tokens are depicted as input and output of transitions t_1 and t_3, respectively.

Figure 6.4 illustrates how allocating and releasing instances of passive resources can be modeled. An ordinary place is used to represent a pool of instances of

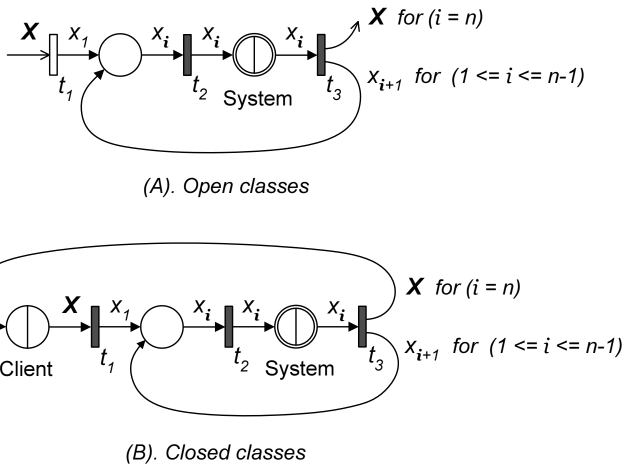


Figure 6.3: Modeling composite transactions.

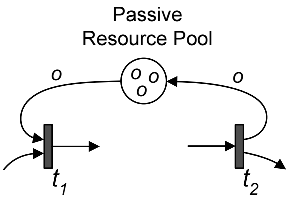


Figure 6.4: Allocating and releasing passive resources.

a passive resource (for e.g. thread pool or connection pool). The instances of the passive resource (e.g. threads or connections) are represented using tokens of color 'o'. Allocating an instance of the resource is modeled by removing a token from the pool (transition t_1). Releasing a previously allocated instance is modeled by returning the token back to the pool (transition t_2). The initial population of the place representing the pool determines the number of instances of the resource available.

The last step of the modeling process is to parameterize the model. This involves providing values for the following model parameters:

- Initial token population of places.
- Service times of tokens at the queues of queueing places.

- Firing weights of immediate transitions.
- Firing delays of timed transitions.

Parameter values need to be supplied on a per workload class basis. The workload model is used as input when assigning values to the parameters. Note that the performance model might be considered for different sets of parameter values.

6.2.5 Step 5: Validate, refine and/or calibrate the model.

The fifth step in our performance modeling methodology aims at ensuring that the performance model built in the previous step reflects the real system and workload to a reasonable degree of accuracy. Before the model can be used for performance prediction, it has to be validated. The model is said to be *valid* if the performance metrics (e.g. response time, throughput and resource utilization) predicted by the model match the measurements on the real system within a certain acceptable margin of error [112]. As a rule of thumb, errors within 10% for utilization and throughput, and within 20% for response time are considered acceptable [111]. Validating the model answers questions such as the following:

- Is the right model for the system being considered?
- Does the model capture all critical aspects of the system behavior, i.e. is it representative enough?
- Are the estimated model input parameters, for e.g. service demands, reasonable?
- Are the assumptions and simplifications made when building the model acceptable?

Model validation is normally done by comparing performance metrics predicted by the model with measurements on the real system. This is done for several different scenarios varying the model input parameters. If the predicted values do not match the measured values within an acceptable level of accuracy, the model must be refined to more accurately reflect the system and workload modeled. Otherwise, the model is deemed valid and can be used for performance prediction. The validation and refinement process is illustrated in Figure 6.5. It is important that the model predictions are verified for a number of different scenarios under different transaction mixes and workload intensities, before the model is deemed valid. Normally, the scenarios considered when measuring the service demands during workload characterization are used as a starting point in the validation phase. After each refinement of the model, all previous scenarios studied, including the ones where the model predictions were accurate, must be re-evaluated.

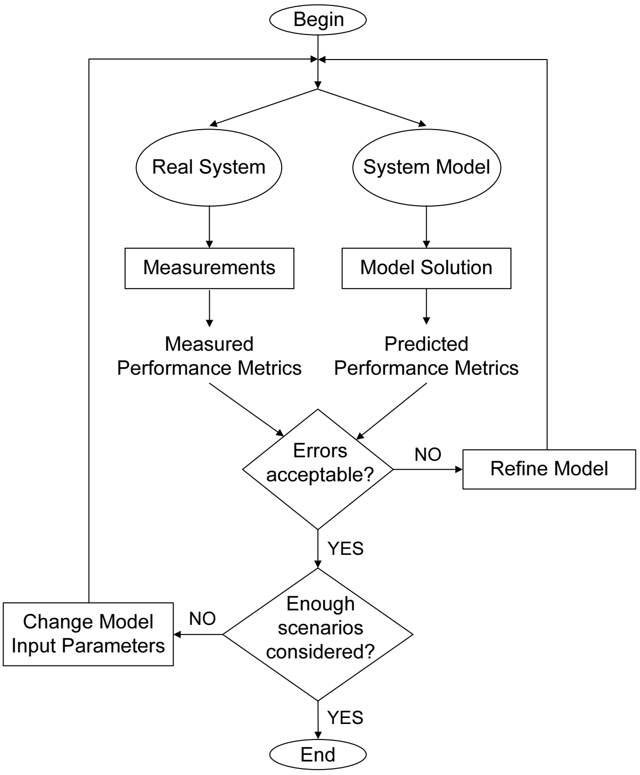

Begin
Real System
System Model
Measurements
Model Solution
Measured Performance Metrics
Predicted Performance Metrics
Errors acceptable?
NO
Refine Model
YES
Enough scenarios considered?
NO
Change Model Input Parameters
YES
End

Figure 6.5: Model validation and refinement process.

The model refinement process usually involves the following activities:

- The model input parameters (e.g. workload intensity and service demand estimates) are double-checked.
- The system is monitored under load to ensure that all critical aspects of its behavior have been captured by the model.
- All assumptions and simplifications made during the modeling process are revisited to make sure that they are acceptable.
- It is considered to increase the level of detail at which system and workload components are modeled.

If after refining the model its results still do not match the measurements on the real system within an acceptable level of accuracy, the model has to be *calibrated.* Model calibration is the process of changing the model to force it to match the actual system [33]. This is achieved by changing the values of some model input or output parameters. The parameters may be increased or decreased by an absolute or percentage amount. Normally, input parameters are changed (e.g. service demands), however, in certain cases also output parameters might be changed. If an output parameter is altered when calibrating the baseline model, it must be altered in the same manner whenever the model is used for performance prediction. After the model is calibrated, the validation procedure must be repeated to make sure that the calibrated model now accurately reflects the real system and workload. For a detailed discussion of model calibration techniques, the reader is referred to [55, 111].

6.2.6 Step 6: Use model to predict system performance.

In this step, the validated performance model is used to predict the performance of the system for the deployment configurations and workload scenarios targeted for analysis. The latter are derived from the modeling objectives.

6.2.7 Step 7: Analyze results and address modeling objectives.

In this last step of the methodology, the results from the model predictions are analyzed and used to address the goals set in the beginning of the modeling study. If after analyzing the results it turns out that some further information is needed, it might be required to go back to the previous step and consider some further workload and configuration scenarios.

6.3 Case Study: Modeling SPECjAppServer2004

Now that we have discussed our modeling methodology in general, we present a practical case study which demonstrates how it can be used to model a realistic DCS and analyze its performance and scalability. The system modeled is our deployment of the industry-standard SPECjAppServer2004 benchmark. Note that in Chapter 4, we modeled previous versions of the benchmark using QN models and QPN models. In both cases, we ran into some serious difficulties stemming from the size and complexity of the system modeled. These problems were addressed in Chapter 5 by means of SimQPN - our simulation tool for QPNs. In this chapter, we include another application of SimQPN, this time analyzing QPN models of SPEC-jAppServer2004. Note that the models considered here span the whole benchmark application and are much more complex and sophisticated. In addition to CPU and

I/O contention, they demonstrate how some further aspects of system behavior, such as thread contention and asynchronous processing, can be modeled.

Consider an automobile manufacturing company that wants to use e-business technology to support its order-inventory, supply-chain and manufacturing operations. The company has decided to employ the J2EE platform and is in the process of developing a J2EE application. Let us assume that the first prototype of this application is SPECjAppServer2004 and that the company is testing the application in the deployment environment depicted in Figure 6.6. This environment uses a cluster of WebLogic servers (WLS) as a J2EE container and an Oracle database server (DBS) for persistence. We assume that all servers in the WebLogic cluster are identical and that initially only two servers are available.

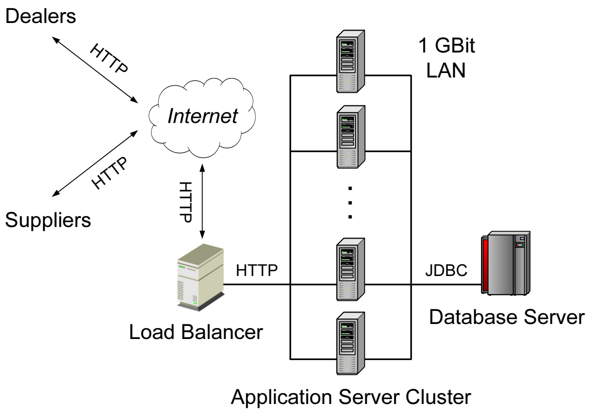


Figure 6.6: Deployment environment.

The company is now about to conduct a performance evaluation of their system in order to find answers to the following questions:

- For a given number of WebLogic servers, what level of performance would the system provide?
- How many WebLogic servers would be needed to guarantee adequate performance under the expected workload?
- Will the capacity of the single load balancer and single database server suffice to handle the incoming load?
- Does the system scale or are there any other potential system bottlenecks?

The following sections show how these questions can be answered by means of the proposed performance modeling methodology.

6.3.1 Establish performance modeling objectives.

Some general goals of the modeling study were listed above. At the beginning of the modeling process, these goals need to be made more specific and precise. Let us assume that under normal operating conditions the company expects to have 72 concurrent dealer clients (40 Browse, 16 Purchase and 16 Manage) and 50 planned production lines. During peak conditions, 152 concurrent dealer clients (100 Browse, 26 Purchase and 26 Manage) are expected and the number of planned production lines could increase up to 100. Moreover, the workload is forecast to grow by 300% over the next 5 years. The average *dealer think time* is 5 seconds, i.e. the time a dealer "thinks"after receiving a response from the system before sending a new request. On average 10 percent of all orders placed are assumed to be large orders. The average delay after completing a work order at a planned production line before starting a new one is 10 seconds. Note that all of these numbers were chosen arbitrarily in order to make our motivating scenario more specific. Based on these assumptions, the following concrete goals are established:

- Predict the performance of the system under normal operating conditions with 4 and 6 WebLogic servers, respectively. What would be the average transaction throughput and response time (for Browse, Purchase and Manage)? How many work orders would be completed per second in the manufacturing domain and what would be the average work order processing time? How utilized (CPU/Disk utilization) would be the WebLogic servers, the load balancer and the database server?

- Determine if 6 WebLogic servers would be enough to ensure that the average response times of business transactions do not exceed half a second during peak conditions.

- Predict how much system performance would improve if the load balancer is upgraded with a slightly faster CPU.

- Study the scalability of the system as the workload increases and additional WebLogic servers are added.

- Determine which servers would be most utilized under heavy load and investigate if they are potential bottlenecks. In particular, verify if the capacity of the single load balancer and single database server would suffice to handle the incoming load.

6.3.2 Characterize the system in its current state.

As shown in Figure 6.6, the system we are considering has a two-tier hardware architecture consisting of an application server tier and a database server tier. Incoming requests are evenly distributed across the nodes in the application server cluster. For HTTP requests (dealer application), this is achieved using a software load balancer running on a dedicated machine. For RMI requests (manufacturing application), this is done transparently by the EJB client stubs. The application logic is partitioned into three layers: presentation layer (Servlets/JSPs), business logic layer (EJBs) and data layer (DBMS). Table 6.1 describes the system components in terms of the hardware and software platforms used. This information is enough for the purposes of our study.

Table 6.1: System component details.

Component	Description
Load Balancer	WebLogic 8.1 Server (HttpClusterServlet) 1 x AMD Athlon XP2000+ CPU 1 GB RAM, SuSE Linux 8
App. Server Cluster Nodes	WebLogic 8.1 Server 1 x AMD Athlon XP2000+ CPU 1 GB RAM, SuSE Linux 8
Database Server	Oracle 9i Server 2 x AMD Athlon MP2000+ CPU 2 GB RAM, SuSE Linux 8
Local Area Network	1 GBit Switched Ethernet

6.3.3 Characterize the workload.

Identify the Basic Components of the Workload

As discussed in Chapter 2, the SPECjAppServer2004 benchmark application is made up of three major subapplications - the dealer application in the dealer domain, the order entry application in the customer domain and the manufacturing application in the manufacturing domain. The dealer and order entry applications process business transactions of three types - Browse, Purchase and Manage. Hereafter, the latter are referred to as *dealer transactions*[2]. The manufacturing application, on

[2]Again the term transaction here is used loosely and should not be confused with a database transaction or a transaction in the sense of the Java Transaction API (JTA transaction).

the other hand, is running production lines which process work orders. Thus, the SPECjAppServer2004 workload is composed of two basic components:

- *Dealer transactions* processed in the dealer and customer domains.
- *Work orders* processed in the manufacturing domain.

Note that each dealer transaction emulates a client session comprising multiple round-trips to the server. For each round-trip there is a separate HTTP request, which can be seen as a subtransaction (or a nested transaction). A more fine-grained approach to model the workload would be to define the individual HTTP requests (subtransactions) as basic components. However, this would unnecessarily complicate the workload model since we are interested in the performance of dealer transactions as a whole and not the performance of their individual subtransactions. The same reasoning applies to work orders, because each work order comprises multiple JTA transactions initiated with separate RMI calls (e.g. scheduleWorkOrder, updateWorkOrder and completeWorkOrder). An alternative approach would be to model the manufacturing workload at the level of JTA transactions or RMI calls. However, this would again unnecessarily complicate the workload model, since we are only interested in the rate at which work orders are processed and not in the performance of the individual work-order-related transactions. This is a typical example, how the level of detail in the modeling process is decided based on the modeling objectives.

Partition Basic Components into Workload Classes

We now partition the basic components of the workload into classes according to the type of work being done. There are three types of dealer transactions - Browse, Purchase and Manage. Since we are interested in their individual behavior, we model them using separate workload classes. Thus, dealer transactions are partitioned into three workload classes: Browse, Purchase and Manage. Work orders, on the other hand, can be divided into two types based on whether they are processed on a planned or large order line. Planned lines run on schedule and complete a predefined number of work orders per unit of time. In contrast, large order lines run only when a large order arrives in the customer domain. Each large order generates a separate work order processed *asynchronously* on a dedicated large order line. Thus, work orders originating from large orders are different from ordinary work orders in terms of the way their processing is initiated and in terms of their resource usage. To distinguish between the two types of work orders, they are modeled using two separate workload classes: *WorkOrder* (for ordinary work orders) and *LargeOrder* (for work orders generated by large orders). The latter will hereafter be referred to as WorkOrder and LargeOrder transactions, respectively. Altogether, we end up with five workload classes: Browse, Purchase, Manage, WorkOrder and LargeOrder.

Identify the System Components and Resources Used by Each Workload Class

The next step is to identify the system components (hardware and software resources) used by each workload class. The following hardware resources are used by dealer transactions (Browse, Purchase and Manage):

- The CPU of the load balancer machine (LB-C).
- The CPU of an application server in the cluster (AS-C).
- The CPUs of the database server (DB-C).
- The disk drive of the database server (DB-D).
- The Local Area Network (LAN).

WorkOrders and LargeOrders use the same resources with exception of the first one (LB-C), since their processing is driven through direct RMI calls to the EJBs in the WebLogic cluster, bypassing the HTTP load balancer.

As far as software resources are concerned, all workload classes use the WebLogic servers and the Oracle DBMS. Dealer transactions additionally use the software load

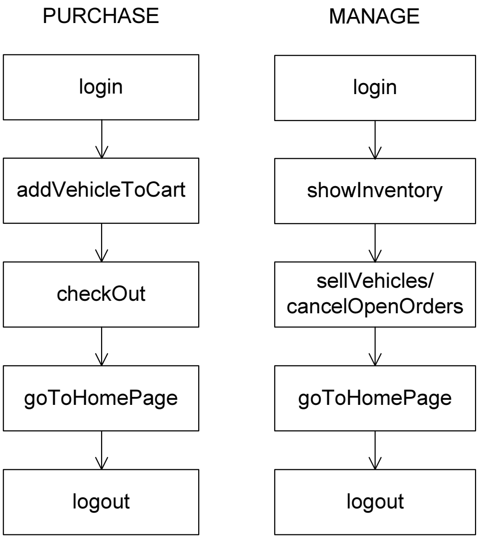


Figure 6.7: Execution graphs for Purchase and Manage.

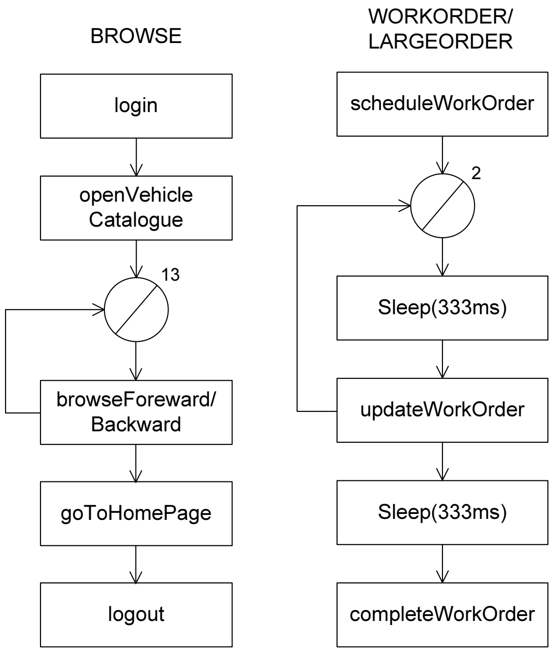


Figure 6.8: Execution graphs for Browse, WorkOrder and LargeOrder.

balancer (HttpClusterServlet), which is deployed in a dedicated WebLogic server. For a transaction to be processed by a WebLogic server, a thread must be allocated from the server's thread pool. If the database needs to be accessed, a database connection from the server's JDBC connection pool must be allocated.

Describe the Inter-Component Interactions and Processing Steps for Each Workload Class

All of the five workload classes identified represent composite transactions. Figures 6.7 and 6.8 use execution graphs to illustrate the processing steps (i.e. the subtransactions) of transactions from the different workload classes.

For every subtransaction, multiple system components are involved and they interact to perform the respective operation. The inter-component interactions and flow of control for subtransactions are depicted in Figure 6.9 by means of client/server interaction diagrams. Directed arcs show the flow of control from one node to the next during execution. Depending on the path followed, different execution scenarios are possible. For example, for dealer subtransactions two scenarios are possible depending on whether the database needs to be accessed or not. Dealer subtransactions that do not access the database (e.g. goToHomePage) follow the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, whereas dealer subtransactions that access the database (e.g. showInventory or checkOut) follow the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7$.

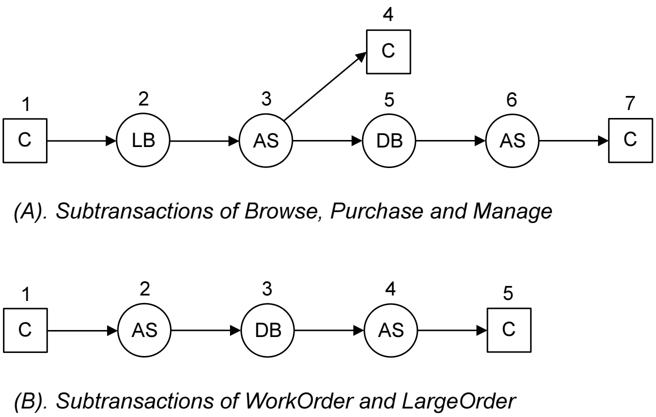

4
C
1
C
2
LB
3
AS
5
DB
6
AS
7
C
(A). Subtransactions of Browse, Purchase and Manage
1
C
2
AS
3
DB
4
AS
5
C
(B). Subtransactions of WorkOrder and LargeOrder

Figure 6.9: Client/server interaction diagrams for subtransactions.

Since most dealer subtransactions do access the database, for simplicity, it is assumed that all of them follow the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7$ in Figure 6.9.

Characterize Workload Classes in Terms of Their Service Demands and Workload Intensity

Since the system is available for testing, the service demands can be determined by injecting load into the system and taking measurements. Note that it is enough to have a single WebLogic server available in order to do this, i.e. it is not required to have a realistic production like testing environment. For each of the five workload classes a separate experiment was conducted injecting transactions from the respective class and measuring the utilization of the various system resources. CPU utilization was measured using the `vmstat` utility on Linux. The disk utilization of the database server was measured with the help of the Oracle 9i Intelligent Agent, which proved to have negligible overhead. Service demands were estimated using the Service Demand Law (see Equation (6.1)) as described in Section 6.2.3.

Table 6.2 reports the estimated service demand parameters for the five request classes in our workload model. Figure 6.10 summarizes this data in a graphical form.

It was decided to ignore network service demands, since all communications were taking place over 1 GBit LAN and communication times were negligible. Therefore, hereafter the network will be ignored in the workload and performance models.

Table 6.2: Workload service demand parameters.

Workload Class	LB-C	AS-C	DB-C	DB-D
Browse	42.72ms	130ms	14ms	5ms
Purchase	9.98ms	55ms	16ms	8ms
Manage	9.93ms	59ms	19ms	7ms
WorkOrder	-	34ms	24ms	2ms
LargeOrder	-	92ms	34ms	2ms

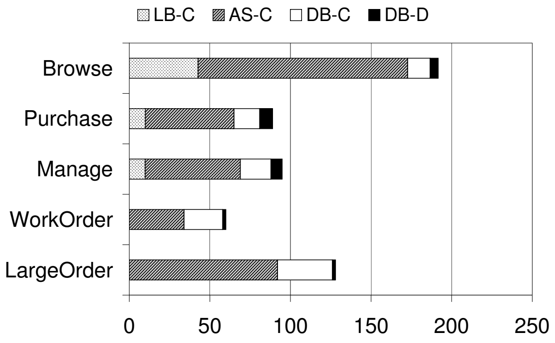


Figure 6.10: Workload service demand parameters (ms).

As evident from Table 6.2, database I/O service demands are much lower than CPU service demands. This stems from the fact that data is cached in the database buffer and, for the most part, disks are accessed only when updating or inserting new data. However, even in this case, the I/O overhead is minimal, since the only thing that is done is to flush the database log buffer, which is performed with sequential I/O. Here we would like to point out that, being an application server benchmark, SPECjAppServer2004 is designed to place the stress on the application server and not on database I/O.

Note that in order to keep the workload model simple, it is assumed that the total service demand of a transaction at a given system resource is spread evenly over its subtransactions. Thus, the service demand of a subtransaction can be estimated by dividing the measured total service demand of the transaction by the number of subtransactions it has. Whether this simplification is acceptable will become clear later when the model is validated. In case the estimation proves to be too inaccurate, one might have to come back and refine the workload model by measuring the service demands of subtransactions individually.

Now that the service demands of workload classes have been quantified, the workload intensity must be specified. For each workload class, the number of units of work (transactions) that contend for system resources must be indicated. The way workload intensity is specified is dictated by the modeling objectives. In our case, workload intensity was defined in terms of the following parameters (see Section 6.3.1):

- Number of concurrent dealer clients (Browse, Purchase and Manage) and their average think time (referred to as *dealer think time*).

- Number of planned production lines and the average time they wait after processing a WorkOrder before starting a new one (referred to as *manufacturing think time* or *mfg think time* for short).

With workload intensity specified in this way, all workload classes are automatically modeled as closed. Two scenarios of interest were indicated when discussing the modeling objectives in Section 6.3.1: operation under normal conditions and operation under peak conditions. The values of the workload intensity parameters for these two scenarios are shown in Table 6.3. However, the workload had been forecast to grow by 300% and another goal of the study was to investigate the scalability of the system as the load increases. Therefore, scenarios with up to 300% higher workload intensity need to be considered as well.

Table 6.3: Workload intensity parameters.

Parameter	Normal Conditions	Peak Conditions
Browse Clients	40	100
Purchase Clients	16	26
Manage Clients	16	26
Planned Lines	50	100
Dealer Think Time	5 sec	5 sec
Mfg Think Time	10 sec	10 sec

6.3.4 Develop a performance model.

It is now shown how to build a QPN model of the system under study and then the model is customized to the concrete configurations of interest. We start by discussing the way basic components of the workload are modeled. During workload characterization, the following five workload classes were identified: Browse, Purchase, Manage, WorkOrder and LargeOrder. All of them represent composite transactions and are modeled using the following token types (colors):

Token 'B' represents a Browse transaction.

Token 'P' represents a Purchase transaction.

Token 'M' represents a Manage transaction.

Token 'W' represents a WorkOrder transaction.

Token 'L' represents a LargeOrder transaction.

The subtransactions of transactions from the different workload classes were shown in Figures 6.7 and 6.8. The inter-component interactions and flow of control for each subtransaction were illustrated in Figure 6.9. In order to make the performance model more compact, the following simplifications are made:

- It is assumed that each server used during processing of a subtransaction is visited only once and that the subtransaction receives all of its service demands at the server's resources during that single visit, i.e. the second visit to the application server (AS) after visiting the database (see Figure 6.9) is dropped, assuming that the subtransaction receives all of its service during the first visit. This simplification is typical for queueing models and has been widely employed.

- Similarly, during the service of a subtransaction at a server, for each server resource used (e.g. CPUs, disk drives), it is assumed that the latter is visited only one time, receiving the whole service demand of the subtransaction at once.

These simplifications make it easier to model the flow of control during processing of subtransactions. While characterizing the workload service demands in Section 6.3.3, we additionally assumed that the total service demand of a transaction at a given system resource is spread evenly over its subtransactions. Together with the above two assumptions, this assumption allows us to consider the subtransactions of a given workload class as equivalent in terms of processing behavior and resource consumption. Thus, we can model subtransactions using a single token type (color) per workload class as follows:

Token 'b' represents a Browse subtransaction.

Token 'p' represents a Purchase subtransaction.

Token 'm' represents a Manage subtransaction.

Token 'w' represents a WorkOrder subtransaction.

Token 'l' represents a LargeOrder subtransaction.

For the sake of compactness, the following additional notation will be used:

Symbol 'D' will be used to denote a 'B', 'P' or 'M' token, i.e. token representing a dealer transaction.

Symbol 'd' will be used to denote a 'b', 'p' or 'm' token, i.e. token representing a dealer subtransaction.

Symbol 'o' will be used to denote a 'b', 'p', 'm', 'w' or 'l' token, i.e. token representing a subtransaction of arbitrary type, hereafter called *subtransaction token*.

To further simplify the model, we assume that LargeOrder transactions are executed with a single subtransaction, i.e. their four subtransactions are bundled into a single subtransaction. Thus, the total service demand of a LargeOrder transaction at a given system resource is assumed to be received at once, during a single visit to the resource. The effect of this simplification on the overall system behavior is negligible, because large orders constitute only 10 percent of all orders placed, i.e. relatively small portion of the system workload. Following these lines of thought, one could consider LargeOrder transactions as non-composite and drop the small 'l' tokens. However, in order to keep token definitions uniform across transaction classes, we will keep the small 'l' tokens and look at LargeOrder transactions as being composed of a single subtransaction represented by an 'l' token.

Following the guidelines for modeling the system components, resources and inter-component interactions presented in Section 6.2.4, we arrive at the model depicted in Figure 6.11.

Following is a brief description of the places used:

C_1 : Queueing place with IS queue used to model the concurrent dealer clients (Browse, Purchase and Manage) conducting dealer transactions in the system. Only 'D' tokens ('B', 'P' or 'M') are allowed in this place and they represent dealer transactions of the respective types. The time tokens spend in this place corresponds to the time a dealer client waits after he has completed a transaction before he starts a new one, i.e. the dealer think time. Therefore, the service time of tokens at the queue of the place is equal to the average dealer think time.

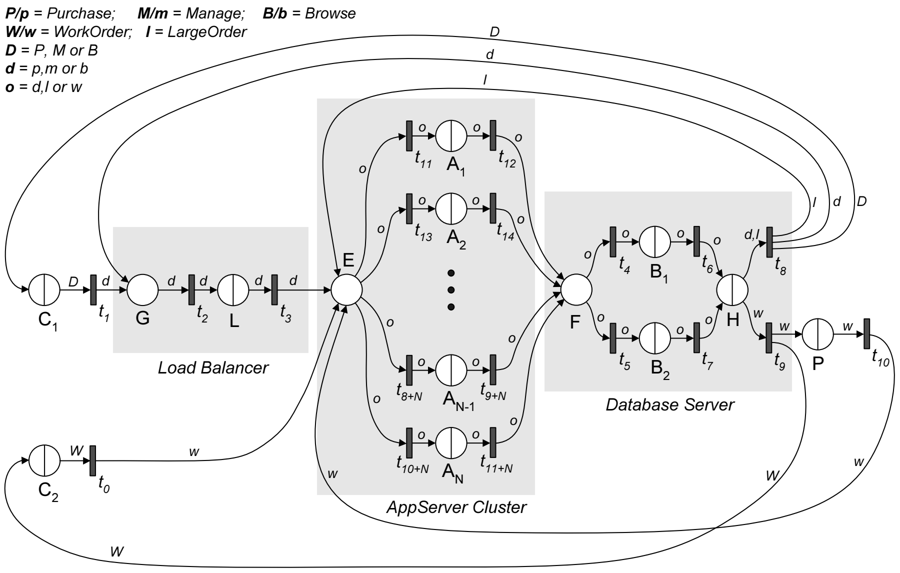


Figure 6.11: QPN model of the system.

C_2 : Queueing place with IS queue used to model the planned production lines in the manufacturing domain driving work order processing. Only 'W' tokens representing WorkOrder transactions are allowed in this place and the time they spend here corresponds to the manufacturing think time. Therefore, the service time of tokens at the queue of the place is equal to the average manufacturing think time.

G : Ordinary place where dealer subtransaction tokens are created when new subtransactions are started.

L : Queueing place with PS queue used to model the CPU of the load balancer machine.

E : Ordinary place where subtransaction tokens arrive before they are distributed over the application server nodes.

$A_1..A_N$: Queueing places with PS queues used to model the CPUs of the N application server nodes.

F : Ordinary place where subtransaction tokens arrive when visiting the database server. From here tokens are evenly distributed over the two database server CPUs.

B_1, B_2 : Queueing places with PS queues used to model the two CPUs of the database server.

H : Queueing place with FCFS queue used to model the disk subsystem (made up of a single 100 GB disk drive) of the database server.

P : Queueing place with IS queue used to model the virtual production line stations that work orders move along during their processing. Only 'w' tokens representing WorkOrder subtransactions are allowed in this place and the time they spend here (i.e. their service time at the IS queue) corresponds to the average delay at a production line station (i.e. 333 ms) emulated by the manufacturing application during work order processing.

We now examine in detail the life-cycle of tokens in the QPN model. As already discussed, upper-case tokens represent transactions, whereas lower-case tokens represent subtransactions. In the initial marking, tokens exist only in the depositories of places C_1 and C_2. The initial number of 'D' tokens ('B', 'P' or 'M') in the depository of the former determines the number of concurrent dealer clients conducting dealer transactions in the system, whereas the initial number of 'W' tokens in the depository of the latter determines the number of planned production lines running in the manufacturing domain. When a dealer client starts a dealer transaction, transition t_1 is fired destroying a 'D' token from the depository of place C_1 and creating a 'd' token in place G, which corresponds to starting the first subtransaction of the dealer transaction. The flow of control during processing of subtransactions in the system is modeled by moving their respective subtransaction tokens across the different places of the QPN. Starting at place G, a dealer subtransaction token ('d') is first sent to place L where it receives service at the CPU of the load balancer. After that it is moved to place E and from there it is routed to one of the N application server CPUs represented by places A_1 to A_N. Transitions $t_{11},\ t_{13}, \ldots, t_{10+N}$ have equal firing probabilities (weights), so that subtransactions are evenly load-balanced across the N application servers. Having completed its service at the application server CPU, the dealer subtransaction token is moved to place F from where it is sent to one of the two database server CPUs with equal probability (transitions t_4 and t_5 have equal firing weights). After completing its service at the CPU, the dealer subtransaction token is moved to place H where it receives service from the database disk subsystem. Once this is completed, the dealer subtransaction token is destroyed by transition t_8 and there are two possible scenarios:

1. A new 'd' token is created in place G corresponding to starting the next dealer subtransaction.

2. If there are no further subtransactions to be executed, the 'D' token removed from place C_1 in the beginning of the transaction is now returned back to place C_1. In case the completed dealer transaction is of type Purchase ('P') and it has generated a large order, additionally a token 'l' is created in place E. Note that, since LargeOrder transactions are assumed to be executed with a single subtransaction, to simplify the model, we create the subtransaction token ('l') directly, instead of first creating a transaction token ('L'). So, in practice, 'L' tokens are not used explicitly in the model.

After a 'D' token of a completed transaction returns back to place C_1, it spends some time at the IS queue of the latter. This corresponds to the time the dealer client "thinks"before starting the next transaction. Once the dealer think time has elapsed, the 'D' token is moved to the depository and the next transaction is started.

When a WorkOrder transaction is started on a planned line in the manufacturing domain, transition t_0 is fired destroying a 'W' token from the depository of place C_2 and creating a 'w' token in place E, which corresponds to starting the first subtransaction of the work order. Since WorkOrder subtransaction requests are load-balanced transparently (by the EJB client stubs) without using a load balancer, they are routed directly to the application server CPUs - places A_1 to A_N. WorkOrder subtransaction tokens ('w') move along the places representing the application server and database server resources exactly in the same way as dealer subtransaction tokens ('d'). After completing their service at place H the following two scenarios are possible:

1. The WorkOrder subtransaction token ('w') is sent to place P whose IS queue delays it for 333 ms, corresponding to the delay at a virtual production line station. Following this the 'w' token is destroyed by transition t_{10} and a new 'w' token is created in place E, representing the next WorkOrder subtransaction.

2. If there are no further subtransactions to be executed, the 'w' token is destroyed by transition t_9 and the 'W' token removed from place C_2 in the beginning of the WorkOrder transaction is now returned back to place C_2.

After a 'W' token of a completed transaction returns back to place C_2, it spends some time at the IS queue of the latter. This corresponds to the time waited after completing a work order at a production line before starting the next one, i.e. the manufacturing think time. Once this time has elapsed, the 'W' token is moved to the depository and the next WorkOrder transaction is started.

All transitions of the model are immediate and with exception of t_8 and t_9 they all have equal weights for all of their firing modes. The assignment of firing weights to transitions t_8 and t_9 is critical to achieving the desired behavior of transactions in the model. Weights must be assigned in such a way that transactions are terminated only after all of their subtransactions have been completed. We will now explain how this is done, starting with transition t_9 since this is the simpler case. Transition t_9 has two firing modes as follows:

A) Mode "$w \rightarrow w$": A 'w' token is moved from place H to place P. This mode corresponds to the case where a WorkOrder subtransaction has been completed, but its parent transaction is not finished yet, i.e. there are further subtransactions to be executed. The parent transaction is delayed for 333 ms at the production line station (place P) and then its next subtransaction is started by depositing a new 'w' token in place E.

B) Mode "$w \rightarrow W$": A 'w' token is destroyed from place H and a 'W' token is deposited in place C_2. This mode corresponds to the case where a WorkOrder subtransaction has been completed, leading to completion of its parent transaction, i.e. there are no further subtransactions to be executed. The 'W' token removed from place C_2 in the beginning of the parent WorkOrder transaction is now returned back.

According to Section 6.3.3 (Figure 6.8), WorkOrder transactions are comprised of four subtransactions. This means that, for every WorkOrder transaction, four subtransactions have to be executed before the transaction is completed, i.e. transition t_9 has to be fired three consecutive times in mode "$w \rightarrow w$" and the fourth time in mode "$w \rightarrow W$". To enforce this behavior, the firing weights (probabilities) of modes "$w \rightarrow w$" and "$w \rightarrow W$" are set to 3/4 and 1/4, respectively. Thus, out of every four times a 'w' token arrives in place H and enables transition t_9, the latter will be fired three times in mode "$w \rightarrow w$" and one time in mode "$w \rightarrow W$" completing a WorkOrder transaction.

Transition t_8, on the other hand, has the following eight firing modes we need to assign weights to:

A) Mode "$b \rightarrow b$": A 'b' token is removed from place H and a new 'b' token is created in place G. This mode corresponds to the case where a Browse subtransaction has been completed, but its parent transaction is not finished yet, i.e. there are further subtransactions to be executed. By creating a new 'b' token in place G, the next subtransaction is started.

B) Mode "$b \rightarrow B$": A 'b' token is removed from place H and a 'B' token is deposited in place C_1. This mode corresponds to the case where a Browse subtransaction has been completed, leading to completion of its parent transaction, i.e. there are no further subtransactions to be executed. The 'B' token removed from place C_1 in the beginning of the parent Browse transaction is now returned back.

C) Mode "$p \rightarrow p$": A 'p' token is removed from place H and a new 'p' token is created in place G. This mode corresponds to the case where a Purchase subtransaction has been completed, but its parent transaction is not finished yet, i.e. there are further subtransactions to be executed. By creating a new 'p' token in place G, the next subtransaction is started.

D) Mode "$p \rightarrow P$": A 'p' token is removed from place H and a 'P' token is deposited in place C_1. This mode corresponds to the case where a Purchase subtransaction has been completed, leading to completion of its parent transaction, i.e. there are no further subtransactions to be executed. The 'P' token removed from place C_1 in the beginning of the parent Purchase transaction is now returned back.

E) Mode "$p \rightarrow P + l$": A 'p' token is removed from place H and a 'P' token is deposited in place C_1. In addition, an 'l' token is deposited in place E. This mode is equivalent to mode "$p \rightarrow P$" above, with the exception that it additionally assumes that the completed Purchase transaction has generated a large order in the manufacturing domain. The LargeOrder transaction is started by depositing an 'l' token in place E.

F) Mode "$m \rightarrow m$": An 'm' token is removed from place H and a new 'm' token is created in place G. This mode corresponds to the case where a Manage subtransaction has been completed, but its parent transaction is not finished yet, i.e. there are further subtransactions to be executed. By creating a new 'm' token in place G, the next subtransaction is started.

G) Mode "$m \rightarrow M$": An 'm' token is removed from place H and an 'M' token is deposited in place C_1. This mode corresponds to the case where a Manage subtransaction has been completed, leading to completion of its parent transaction, i.e. there are no further subtransactions to be executed. The 'M' token removed from place C_1 in the beginning of the parent Manage transaction is now returned back.

H) Mode "$l \rightarrow \emptyset$": An 'l' token is removed from place H. This mode corresponds to the case where a LargeOrder transaction has been completed. Its token is simply destroyed.

According to Section 6.3.3 (Figure 6.7), Browse transactions have 17 subtransactions, whereas Purchase and Manage have only 5. This means that, for every

Browse transaction, 17 subtransactions have to be executed before the transaction is completed, i.e. out of every 17 times a 'b' token arrives in place H and enables transition t_8, the latter has to be fired 16 times in mode "$b \longrightarrow b$" and one time in mode "$b \longrightarrow B$" completing a Browse transaction. To explain how firing weights are assigned to enforce this behavior, let us look at transition t_8' firing modes as 8 possible events (A,B,C,...,H) in terms of probability theory. Each time the transition is enabled, one of the 8 events occurs. The firing weight of a mode is the relative probability that its respective event occurs. Under these assumptions, the above condition for the behavior of the Browse transaction can be expressed in the following way:

$$P(A \mid A \cup B) = \frac{16}{17} \tag{6.2}$$

or alternatively as

$$P(B \mid A \cup B) = \frac{1}{17} \tag{6.3}$$

Applying some basic probability theory leads to

$$\begin{aligned} P(B \mid A \cup B) &= \frac{P(B \ \cap \ (A \cup B))}{P(A \cup B)} = \\ &= \frac{P(B)}{P(A) + P(B)} = \frac{1}{17} \end{aligned} \tag{6.4}$$

which is equivalent to

$$P(B) = \frac{1}{16} P(A) \tag{6.5}$$

Therefore, to enforce the desired behavior of Browse transactions, one needs to simply make sure that the firing weights of modes A and B fulfill condition (6.5). Applying the same reasoning to Manage transactions, we obtain the following condition that the firing weights of modes F and G must fulfill:

$$P(G) = \frac{1}{4} P(F) \tag{6.6}$$

Deriving the condition for achieving the right behavior of Purchase transactions is a little more complicated. The following must be fulfilled:

$$P(D \cup E \mid C \cup D \cup E) = \frac{1}{5} \tag{6.7}$$

Again, applying some basic probability theory leads to

$$\begin{aligned} P(D \cup E \mid C \cup D \cup E) &= \frac{P((D \cup E) \ \cap \ (C \cup D \cup E))}{P(C \cup D \cup E)} = \\ &= \frac{P(D \cup E)}{P(C \cup D \cup E)} = \frac{P(D) + P(E)}{P(C) + P(D) + P(E)} = \frac{1}{5} \end{aligned} \tag{6.8}$$

which is equivalent to

$$P(D) + P(E) = \frac{1}{4}P(C) \tag{6.9}$$

Since, on average 10 percent of all completed Purchase transactions generate large orders, we have the following additional condition:

$$P(E \mid D \cup E) = \frac{1}{10} \tag{6.10}$$

$$\begin{aligned} P(E \mid D \cup E) = \frac{P(E \ \cap \ (D \cup E))}{P(D \cup E)} = \\ = \frac{P(E)}{P(D \cup E)} = \frac{P(E)}{P(D) + P(E)} = \frac{1}{10} \end{aligned} \tag{6.11}$$

which is equivalent to

$$P(E) = \frac{1}{9}P(D) \tag{6.12}$$

Finally, the following condition ensures that arriving tokens in place H are ”processed”by transition t_8 in an order independent of the token color:

$$P(A \cup B) = P(C \cup D \cup E) = P(F \cup G) = P(H) \tag{6.13}$$

This is equivalent to

$$\begin{aligned} P(A) + P(B) = P(C) + P(D) + P(E) = \\ = P(F) + P(G) = P(H) \end{aligned} \tag{6.14}$$

Therefore, in order to enforce the desired behavior of dealer transactions, the weights of transition t_8' firing modes must be chosen in such a way that conditions (6.5), (6.6), (6.9), (6.12) and (6.14) are fulfilled. Note that being relative probabilities, the firing weights do not necessarily need to add up to 1. This leads to a system of seven simultaneous equations (see (6.15)) with eight variables. One possible solution is the following $P(A) = 16$, $P(B) = 1$, $P(C) = 13.6$, $P(D) = 3.06$, $P(E) = 0.34$, $P(F) = 13.6$, $P(G) = 3.4$, $P(H) = 17$.

$$
\begin{aligned}
P(B) &= \frac{1}{16} P(A) \\
P(G) &= \frac{1}{4} P(F) \\
P(D) + P(E) &= \frac{1}{4} P(C) \\
P(E) &= \frac{1}{9} P(D) \\
P(A) + P(B) &= P(C) + P(D) + P(E) \\
P(A) + P(B) &= P(F) + P(G) \\
P(A) + P(B) &= P(H)
\end{aligned}
\tag{6.15}
$$

The proposed performance model has the following input parameters:

1. Initial population of place C_1, i.e. number of 'B', 'P' and 'M' tokens, corresponding to the number of concurrent dealer clients (Browse, Purchase and Manage) conducting dealer transactions.

2. Initial population of place C_2, i.e. number of 'W' tokens, corresponding to the number of planned production lines running in the manufacturing domain.

3. Service time of dealer transaction tokens ('B', 'P' and 'M') at the IS queue of place C_1, corresponding to the average dealer think time.

4. Service time of work order transaction tokens ('W') at the IS queue of place C_2, corresponding to the average manufacturing think time.

5. Service times of subtransaction tokens ('b', 'p', 'm', 'w' and 'l') at the queues of places A_i, B_j and H.

6. Service times of dealer subtransaction tokens ('b', 'p' and 'm') at the queue of place L.

7. Number of application server nodes N.

The workload intensity and service demand parameters from Section 6.3.3 (Tables 6.2 and 6.3) are used here to provide values for the above parameters. A separate set of parameter values is specified for each workload scenario considered. The service times of subtransactions at the queues of the model (items 5 and 6 in the list above) are estimated by dividing the total service demands of the respective transactions by the number of subtransactions they have.

6.3.5 Validate, refine and/or calibrate the model.

The model developed in the previous sections is now validated by comparing its predictions against measurements on the real system. Measurements are collected in the testing environment depicted in Figure 6.12. For details on the hardware and software used refer to Table 6.1. Two application server nodes are available for the validation experiments. The driver machine runs a modified version of the SPECjAppServer2004 driver that allows the user to configure precisely the transaction mix and intensity of the workload injected into the system. Specifically, the modified driver allows the user to set the number of concurrent dealer clients of each class emulated, as well as their average think time. Moreover, the user can specify the number of planned production lines run in the manufacturing domain and the time they wait after processing a work order before starting a new one, i.e. the manufacturing think time.

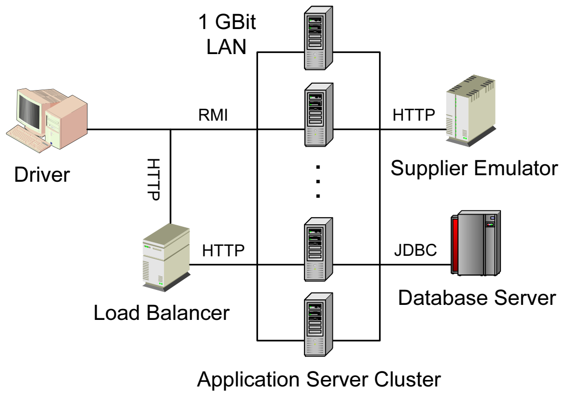


Figure 6.12: System testing environment.

The model predictions are verified for a number of different scenarios under different transaction mixes and workload intensities. The model input parameters for two specific scenarios considered here are shown in Table 6.4.

Table 6.4: Input parameters for validation scenarios.

Parameter	Scenario 1	Scenario 2
Browse Clients	20	40
Purchase Clients	10	20
Manage Clients	10	30
Planned Lines	30	50
Dealer Think Time	5 sec	5 sec
Mfg Think Time	10 sec	10 sec

Table 6.5 compares the model predictions against measurements on the real system. The metrics considered are transaction throughput (X_i), transaction response time (R_i) and server utilization (U_{LB} for the load balancer, U_{AS} for the application server and U_{DB} for the database server). Since LargeOrder transactions are processed asynchronously, they do not have a response time metric. The maximum modeling error for throughput is 8.1%, for utilization 10.2% and for response time 12.9%. Varying the transaction mix and workload intensity led to predictions of similar accuracy. Since these results are reasonable, the model is considered valid. Note that the model validation process is iterative in nature and is usually repeated multiple times as the model evolves. Even though the model is deemed valid at this point of the study, as we will see later, the model might lose its validity when it is modified in order to reflect changes in the system. Generally, it is required that the validation process is repeated after every modification of the model. However, even if the model has not been modified, it is recommended that validation is performed on a regular basis as the system and workload evolve.

6.3.6 Use model to predict system performance.

In Section 6.3.1 the following concrete goals were set for the performance study:

1. Predict the performance of the system under normal operating conditions with 4 and 6 WebLogic servers, respectively.

2. Determine if 6 WebLogic servers would be enough to ensure that the average response times of business transactions do not exceed half a second during peak conditions.

Table 6.5: Validation results.

	Validation Scenario 1			Validation Scenario 2		
METRIC	Model	Measured	Error	Model	Measured	Error
X_B	3.784	3.718	+1.8%	6.988	6.913	+1.1%
X_P	1.948	1.963	-0.7%	3.781	3.808	-0.7%
X_M	1.940	1.988	-2.4%	5.634	5.530	+1.9%
X_W	2.713	2.680	+1.2%	4.469	4.510	-0.9%
X_L	0.197	0.214	-8.1%	0.377	0.383	-1.56%
R_B	289ms	256ms	+12.9%	704ms	660ms	+6.7%
R_P	131ms	120ms	+9.2%	309ms	305ms	+1.3%
R_M	139ms	130ms	+6.9%	329ms	312ms	+5.4%
R_W	1087ms	1108ms	-1.9%	1199ms	1209ms	-0.8%
U_{LB}	20.1%	19.5%	+3.1%	39.2%	40.3%	-2.7%
U_{AS}	41.3%	38.5%	+7.3%	81.8%	83.0%	-1.4%
U_{DB}	9.7%	8.8%	+10.2%	19.3%	19.3%	0.0%

3. Predict how much system performance would improve if the load balancer is upgraded with a faster CPU.

4. Study the scalability of the system as the workload intensity increases and additional WebLogic servers are added.

5. Determine which servers would be most utilized under heavy load and investigate if they are potential bottlenecks.

The system model is now used to predict the performance of the system for the scenarios mentioned above. In order to validate our approach, for each scenario considered we will compare the model predictions against measurements on the real system. Note that this validation is not part of the methodology itself and normally it does not have to be done. Indeed, if we would have to validate the model results for every scenario considered, there would be no point in using the model in the first place. The reason we validate the model results here is to demonstrate the effectiveness of our methodology and showcase the predictive power of the QPN models it is based on.

Table 6.6 reports the analysis results for the scenarios under normal operating conditions with 4 and 6 application server nodes. In both cases, the model predictions are very close to the measurements on the real system. Even for response times, the modeling error does not exceed 10.1 percent.

Table 6.6: Analysis results for scenarios under normal conditions with 4 and 6 app. server nodes.

	4 App. Server Nodes			6 App. Server Nodes		
METRIC	Model	Measured	Error	Model	Measured	Error
X_B	7.549	7.438	+1.5%	7.589	7.415	+2.3%
X_P	3.119	3.105	+0.5%	3.141	3.038	+3.4%
X_M	3.111	3.068	+1.4%	3.117	2.993	+4.1%
X_W	4.517	4.550	-0.7%	4.517	4.320	+4.6%
X_L	0.313	0.318	-1.6%	0.311	0.307	+1.3%
R_B	299ms	282ms	+6.0%	266ms	267ms	-0.4%
R_P	131ms	119ms	+10.1%	116ms	110ms	+5.5%
R_M	140ms	131ms	+6.9%	125ms	127ms	-1.6%
R_W	1086ms	1109ms	-2.1%	1077ms	1100ms	-2.1%
U_{LB}	38.5%	38.0%	+1.3%	38.7%	38.5%	+0.1%
U_{AS}	38.0%	35.8%	+6.1%	25.4%	23.7%	+0.7%
U_{DB}	16.7%	18.5%	-9.7%	16.7%	15.5%	+0.8%

Table 6.7: Analysis results for scenarios under peak conditions with 6 app. server nodes.

	Original Load Balancer			Upgraded Load Balancer		
METRIC	Model	Measured	Error	Model	Measured	Error
X_B	17.960	17.742	+1.2%	18.471	18.347	+0.7%
X_P	4.981	4.913	+1.4%	5.027	5.072	-0.8%
X_M	4.981	4.995	-0.3%	5.013	5.032	-0.4%
X_W	8.984	8.880	+1.2%	9.014	8.850	+1.8%
X_L	0.497	0.490	+1.4%	0.501	0.515	-2.7%
R_B	567ms	534ms	+6.2%	413ms	440ms	-6.5%
R_P	214ms	198ms	+8.1%	182ms	165ms	+10.3%
R_M	224ms	214ms	+4.7%	193ms	187ms	+3.2%
R_W	1113ms	1135ms	-1.9%	1115ms	1123ms	-0.7%
U_{LB}	86.6%	88.0%	-1.6%	68.2%	70.0%	-2.6%
U_{AS}	54.3%	53.8%	+0.9%	55.4%	55.3%	+0.2%
U_{DB}	32.9%	34.5%	-4.6%	33.3%	35.0%	-4.9%

Table 6.7 shows the model predictions for two scenarios under peak conditions with 6 application server nodes. The first one uses the original load balancer, while the second one uses an upgraded load balancer with a faster CPU. The faster CPU results in lower service demands as shown in Table 6.8. With the original load balancer, six application server nodes turned out to be insufficient to guarantee average response times of business transactions below half a second. However, with the upgraded load balancer this was achieved. In the rest of the scenarios considered, the upgraded load balancer will be used.

Table 6.8: Load balancer service demands.

Load Balancer	Browse	Purchase	Manage
Original	42.72ms	9.98ms	9.93ms
Upgraded	32.25ms	8.87ms	8.56ms

Table 6.9: Analysis results for scenarios under heavy load with 8 app. server nodes.

	Heavy Load Scenario 1			Heavy Load Scenario 2		
METRIC	Model	Measured	Error	Model	Measured	Error
X_B	26.505	25.905	+2.3%	28.537	26.987	+5.7%
X_P	4.948	4.817	+2.7%	4.619	4.333	+6.6%
X_M	4.944	4.825	+2.5%	4.604	4.528	+1.6%
X_W	8.984	8.820	+1.8%	9.003	8.970	+0.4%
X_L	0.497	0.488	+1.8%	0.460	0.417	+10.4%
R_B	664ms	714ms	-7.0%	2012ms	2288ms	-12.1%
R_P	253ms	257ms	-1.6%	632ms	802ms	-21.2%
R_M	263ms	276ms	-4.7%	630ms	745ms	-15.4%
R_W	1116ms	1128ms	-1.1%	1123ms	1132ms	-0.8%
U_{LB}	94.1%	95.0%	-0.9%	99.9%	100.0%	-0.1%
U_{AS}	54.5%	54.1%	+0.7%	57.3%	55.7%	+2.9%
U_{DB}	38.8%	42.0%	-7.6%	39.6%	42.0%	-5.7%

We now investigate the behavior of the system as the workload intensity increases beyond peak conditions and further application server nodes are added. Table 6.9 shows the model predictions for two scenarios with an increased number of concurrent Browse clients, i.e. 150 in the first one and 200 in the second one. In both scenarios the number of application server nodes is 8. As evident from the results, the load balancer is completely saturated when increasing the workload intensity and it becomes a bottleneck limiting the overall system performance. Therefore, adding further application server nodes would not bring any benefit, unless the load balancer is replaced with a faster one.

Since the load balancer is the bottleneck resource, it is interesting to investigate its behavior a little further. Until now it was assumed that when a request arrives at the load balancer, there is always a free thread which can start processing it immediately, i.e. there is no thread contention. However, if one keeps increasing the workload intensity, the number of concurrent requests at the load balancer will eventually exceed the number of available threads. The latter would lead to thread contention, resulting in additional delays at the load balancer, not captured by our system model. This is a typical example how a valid model may lose its validity as the workload evolves. We will now show how the model can be refined to capture the thread contention at the load balancer. As discussed in Section 6.2.4, passive system resources such as threads can be modeled as tokens inside ordinary places. In Figure 6.13, an extended version of our system model is shown, which includes an ordinary place T representing the load balancer thread pool. Before a dealer request (i.e. dealer subtransaction token 'd') is scheduled for processing at the load balancer CPU (place L), a token 't' representing a load balancer thread is allocated from the thread pool (place T). After the dealer request has been served at the load balancer CPU, the thread token is returned back to the thread pool. Thus, if an arriving dealer request finds no available thread tokens in the thread pool, it will have to wait in place G until a thread is released. The initial population of place T determines the number of threads in the load balancer thread pool.

At first sight, this appears to be the right approach to model the thread contention at the load balancer. However, an attempt to validate the extended model reveals a significant discrepancy between the model predictions and measurements on the real system. In particular, it stands out that predicted response times are much lower than measured response times for dealer transactions with low workload intensities. A closer investigation shows that the problem is in the way dealer subtransaction tokens arriving in place G are scheduled for processing at the load balancer CPU (place L). Dealer subtransaction tokens become available for firing of the output transition t_2 immediately upon their arrival at place G. Thus, whenever arriving tokens are blocked in place G (because of lack of threads) their order of arrival is lost. After a thread is released, transition t_2 fires in one of its enabled modes ('b','p' or 'm') with equal probability. Therefore, the order in which waiting subtransaction tokens are scheduled for processing at the load balancer CPU does not match the order of their arrival at place G. This obviously does not reflect the way the real system works and renders the model unrepresentative.

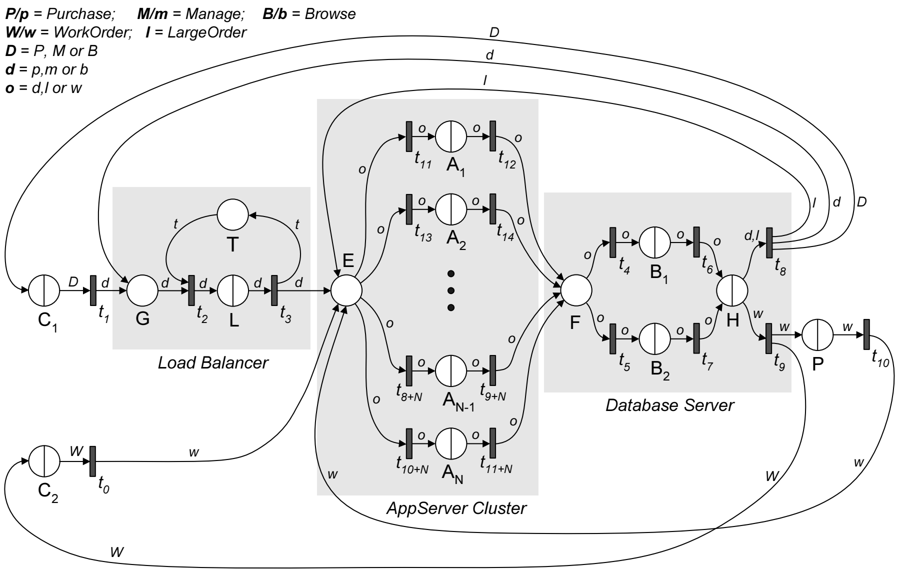


Figure 6.13: Extended QPN model of the system (capturing thread contention at the load balancer).

On average, dealer transactions with high workload intensity would be expected to have more subtransaction tokens blocked in place G (waiting for threads) than dealer transactions with low workload intensities. However, when choosing the next subtransaction token to schedule for processing, all subtransaction types are treated equally irrespective of the arrival order of their tokens. Thus, if one hundred 'b' tokens followed by a single 'p' token arrive at place G and are blocked because of no available threads, the 'p' token would have 50% chance of being scheduled next when a thread becomes available, overtaking the one hundred 'b' tokens. This explains why predicted response times were shorter than measured response times for transactions with low workload intensities.

The above situation describes a common drawback of QPN models, i.e. tokens inside ordinary places and depositories are not distinguished in terms of their order of arrival. In order to address this problem, SimQPN introduces the notion of *departure disciplines* for ordinary places and depositories. The departure discipline determines the order in which arriving tokens become available for output transitions of the place or depository. Currently, two departure disciplines are supported, Normal (used by default) and First-In-First-Out (FIFO). The former implies that tokens become available for output transitions immediately upon arrival just like in conventional QPN models. The latter implies that tokens become available for output transitions in the order of their arrival, i.e. a token can leave the place/depository only after all tokens that have arrived before it have left, hence the term FIFO.

Coming back to the problem above with the way thread contention is modeled, we now change the departure discipline of place G from Normal to FIFO. This ensures that subtransaction tokens waiting at place G are scheduled for processing in the order in which they arrive. After this change, the model passes the validation tests and can be used for performance prediction.

We now consider two additional heavy load scenarios with an increased number of concurrent dealer clients leading to thread contention in the load balancer. The workload intensity parameters for the two scenarios are shown in Table 6.10. The first scenario has a total of 360 concurrent dealer clients, the second 420. Table 6.11 compares the model predictions for the first scenario in two configurations with 8 application server nodes and 15 and 30 load balancer threads, respectively. Table 6.12 presents the same results for the second scenario in a configuration with 8 application server nodes and 20 load balancer threads. In addition to response times, throughput and utilization, the average length of the load balancer thread queue (N_{LBTQ}) is considered. The latter corresponds to the average token population of place G. As evident from the results, the model predictions are very close to the measurements and even for response times the modeling error does not exceed 16.8%. Repeating the analysis for a number of variations of the model input parameters led to results of similar accuracy.

Table 6.10: Workload intensity parameters for heavy load scenarios with thread contention.

Parameter	Heavy Load Sc. 3	Heavy Load Sc. 4
Browse Clients	300	270
Purchase Clients	30	90
Manage Clients	30	60
Planned Lines	120	120
Dealer Think Time	5 sec	5 sec
Mfg Think Time	10 sec	10 sec

Table 6.11: Analysis results for heavy load scenario 3 with 15 and 30 load balancer threads and 8 app. server nodes.

	Heavy Load Sc. 3 with 15 Thr.			Heavy Load Sc. 3 with 30 Thr.		
METRIC	Model	Measured	Error	Model	Measured	Error
X_B	28.607	27.323	+4.7%	28.590	27.205	+5.1%
X_P	4.501	4.220	+6.7%	4.499	4.213	+6.8%
X_M	4.489	4.387	+2.3%	4.494	4.485	+0.2%
X_W	10.784	10.660	+1.2%	10.793	10.800	-0.1%
X_L	0.447	0.410	+9.0%	0.450	0.446	+0.1%
R_B	5495ms	5740ms	-4.2%	5495ms	5805ms	-5.3%
R_P	1674ms	1977ms	-15.3%	1665ms	2001ms	-16.8%
R_M	1685ms	1779ms	-5.3%	1670ms	1801ms	-7.3%
R_W	1125ms	1158ms	-2.8%	1125ms	1143ms	-1.6%
U_{LB}	100.0%	93.0%	+7.5%	99.9%	100.0%	-0.1%
U_{AS}	57.9%	57.8%	+0.2%	57.9%	58.0%	-0.2%
U_{DB}	41.6%	44.0%	-5.5%	41.6%	44.0%	-5.5%
N_{LBTQ}	146	161	-9.3%	131	146	-10.3%

Table 6.12: Analysis results for heavy load scenario 4 with 20 load balancer threads and 8 app. server nodes.

	Heavy Load Sc. 4 with 20 Threads		
METRIC	Model	Measured	Error
X_B	24.978	24.655	+1.3%
X_P	13.305	12.795	+4.0%
X_M	8.867	8.580	+3.3%
X_W	10.766	10.290	+4.6%
X_L	1.333	1.317	+1.2%
R_B	5808ms	5784ms	+0.4%
R_P	1772ms	1996ms	-11.2%
R_M	1782ms	1810ms	-1.5%
R_W	1145ms	1179ms	-2.8%
U_{LB}	100.0%	95.0%	+5.3%
U_{AS}	62.4%	65.4%	-4.6%
U_{DB}	51.7%	61.0%	-15.2%
N_{LBTQ}	151	150	+0.7%

6.3.7 Analyze results and address modeling objectives.

We can now use the results from the performance analysis to address the goals established in Section 6.3.1. By means of the developed QPN model, we were able to predict the performance of the system under normal operating conditions with 4 and 6 WebLogic servers. It turned out that using the original load balancer, six application server nodes were insufficient to guarantee average response times of business transactions below half a second. Upgrading the load balancer with a slightly faster CPU led to the CPU utilization of the load balancer dropping by a good 20 percent. As a result, the response times of dealer transactions improved by 15 to 27 percent, meeting the "half a second" requirement. However, increasing the workload intensity beyond peak conditions revealed that the load balancer was a bottleneck resource, preventing us to scale the system by adding additional WebLogic servers (see Figure 6.14). Therefore, in light of the expected workload growth, the company should either replace the load balancer machine with a faster one or consider using a more efficient load balancing method. After this is done, the performance analysis should be repeated with the new load balancer to make sure that there are no other system bottlenecks. It should also be ensured that the load balancer is configured with enough threads so that there is no thread contention.

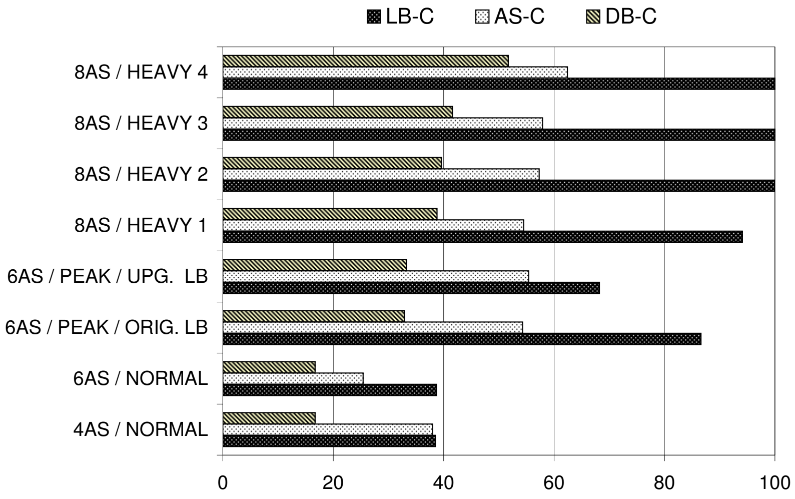


Figure 6.14: Predicted server CPU utilization in considered scenarios.

6.4 Concluding Remarks

In this chapter, a practical performance modeling methodology for DCS was presented. The methodology takes advantage of the modeling power and expressiveness of the QPN modeling formalism to improve model representativeness and enable accurate performance prediction. We presented a detailed case study in which a model of a realistic DCS was constructed and used to analyze its performance and scalability. The model representativeness was validated by comparing its predictions against measurements on the real system. A number of different deployment configurations and workload scenarios were considered. In addition to CPU and I/O contention, it was shown how some more complex aspects of system behavior, such as thread contention and asynchronous processing, can be modeled. The model proved to accurately reflect the performance and scalability characteristics of the system under study. The modeling error for transaction response time did not exceed 21.2% and was much lower for transaction throughput and resource utilization. The proposed performance modeling methodology provides a powerful tool for performance engineering of DCS.

Chapter 7

Related Work

I think there is a world market for maybe five computers.
– *Thomas J. Watson, Chairman of IBM, 1943*

7.1 Benchmarking DCS

In this section, we review some related work in the area of benchmarking and performance evaluation of DCS using artificial workloads.

Standard Benchmarks The Transaction Processing Performance Council (TPC) has released two benchmarks that can be used for benchmarking DCS. The first one is the TPC Benchmark™ W (TPC-W) [165], which has been available since 2000. The second one is the TPC Benchmark™ App (TPC-App) [164], which was released in December, 2004. It is important to note that, unlike SPEC, TPC does not provide implementations of its benchmarks. A TPC benchmark is essentially a specification that defines an application and a set of requirements on the workload that has to be run. The user is expected to implement the benchmark application and workload on the platform to be tested. Having said that, we now take a closer look at the two benchmarks mentioned above.

TPC-W is a transactional Web benchmark. It comprises a set of basic operations designed to exercise business-oriented transactional Web servers. These basic operations have been given a real-life context, portraying the activity of a retail store Web site supporting user browsing, searching and online ordering functionality. The majority of visitor activity is browsing. Some percentage of all visits result in submitting a new order. In addition to using the system as a store-front, it is also used for administration of the Web site.

TPC-App, on the other hand, is an application server and Web Services benchmark. It comprises a set of basic operations designed to exercise transactional application server functionality in a manner representative of business-to-business Web service environments. TPC-App showcases the performance capabilities of application server systems. It does not benchmark the logic needed to process or display the presentation layer (for example, HTML) to the clients. The application portrayed by the TPC-App is a retail distributor on the Internet with ordering and product browsing scenarios. The majority of requests generate order purchase activity with a smaller portion requesting item catalog information.

While, TPC-W and TPC-App may look similar to SPECjAppServer2004, none of them comes close to the scope and complexity of the workload modeled by SPECjAppServer2004. In addition to online order processing, SPECjAppServer2004 models just-in-time manufacturing and supply chain management, which are not covered by either TPC-W or TPC-App. Another difference is that SPECjAppServer2004 measures the performance of an *end-to-end* DCS, encompassing all platform services used to implement the presentation, business and data logic, including dynamic Web page generation, transaction processing, asynchronous messaging, database connectivity, persistence, etc. Both TPC-W and TPC-App, on the other hand, are focused on subsets of these services. For example, TPC-W does not cover asynchronous messaging, whereas TPC-App does not cover dynamic Web page generation.

Proprietary Benchmarks Beside industry-standard benchmarks, a number of proprietary benchmarks for DCS have been developed and used in the industry. For example, IBM has been using Trade3 [67] - the third generation of the WebSphere end-to-end benchmark and performance sample application. The Trade3 workload models an online stock brokerage application and has been designed to showcase the performance and scalability of the key components and features of the WebSphere platform.

Another example is the Java™Pet Store Demo [156] - a J2EE sample application developed under the J2EE Blueprints program at Sun Microsystems. The Java Pet Store application demonstrates how to use the capabilities of the J2EE platform to develop flexible, scalable and cross-platform enterprise applications. Although it was never meant as a benchmark, several companies have used the Java Pet Store to test and showcase the performance of their products. This has led to some contradictive performance comparisons and marketing claims [3, 120, 122, 124, 163].

Two other proprietary benchmarks that have been used to compare middleware platforms are Doculabs' Nile @Bench benchmark [121] and Urbancode's EJB Benchmark [168]. The .NET Framework Community (GotDotNet) hosts a Web site [123] containing resources on evaluating Microsoft .NET vs. J2EE application server technologies. Some free benchmarking suites for CORBA-based systems are available from Charles University Distributed System Group's Web page [40].

Open-Source Benchmarks In the open-source arena, the ObjectWeb open-source middleware consortium has initiated the JMOB (Java Middleware Open Benchmarking) [128] project with the aim to support the development and distribution of open-source middleware benchmarks. Several benchmarks are currently available including RUBiS (Rice University Bidding System) [129] - an auction site prototype modeled after eBay.com, Stock-Online [130] - a simple online stockbroking system, and RUBBoS [127] - bulletin board benchmark modeled after an online news forum like Slashdot.org.

Applications of Benchmarks in Research Numerous research articles have been published that exploit benchmarks for studying different aspects of the performance and scalability of DCS.

In [70] we use the Java™Pet Store application to evaluate the impact on system performance when introducing Web Service interfaces to an originally tightly coupled application. Using two implementation variants of the application, one based strictly on the J2EE 1.3 platform and the other implementing some interfaces as Web Services, performance is compared in terms of the achieved overall throughput, response time and latency.

In [37], Cecchet et al. use the RUBiS benchmark to investigate the combined effect of the application implementation method, container design and efficiency of communication layers on the performance and scalability of J2EE application servers. Five different EJB implementations of the auction site are compared: stateless session beans (SMP), entity beans with CMP, entity beans with BMP, entity beans with session façade beans, and EJB 2.0 local interfaces (entity beans with only local interfaces and session façade beans with remote interfaces). In addition, a Java servlets-only version that does not use EJB is considered. The results of the study are quite interesting, however, unfortunately only open-source application servers have been considered and it is not clear to what extent the conclusions can be generalized to commercial application servers.

Ran et al. [136] developed the Stock-Online [130] benchmark and used it to evaluate competing J2EE platforms and compare their performance and scalability [60]. The benchmark was developed as part of CSIRO's Middleware Technology Evaluation (MTE) project and was later made open-source [128]. It has also been applied to test different J2EE programming idioms and design patterns [59]. Furthermore, based on the benchmark, a J2EE performance tuning methodology was developed.

In [27, 28], Brebner and Gosper use the ECperf benchmark to evaluate the scalability of the J2EE platform in general. They analyze the published ECperf results and find some trends and correlations indicating that the J2EE technology is very scalable, both in a scale-up and scale-out manner. Other observed trends include, a linear correlation between middle-tier total processing power and throughput, as well as between J2EE application server license costs and throughput. However, the results clearly indicate that there is an increasing cost per user with increasing capacity systems, and scale-up is proportionately more expensive than scale-out.

Karlsson et al. [72] use the ECperf benchmark to study the memory system behavior of commercial J2EE-based middleware. They present a detailed characterization of the memory system behavior of ECperf running on both commercial server hardware and in a simulated environment. In [73] they extend their study by comparing ECperf against SPECjbb2000. In [176], Zhang et al. present a comparison of ECperf against IBM's Trade2 benchmark.

Basilio [9] uses the ECperf workload in order to evaluate the performance of the BEA WebLogic JRockit JVM on the Itanium II platform. The internals of JVMs in general and JRockit in particular are deeply studied, and issues like JRockits performance, scalability, reliability and tuning options are tested, compared to other JVMs and analyzed.

Cain et al. [34] characterize the memory system and branch predictor behavior of a Java Servlet implementation of TPC-W. They evaluate the effectiveness of a coarse-grained multi-threaded processor at increasing system throughput using TPC-W and other commercial workloads. The results show a system throughput improvement from 8% to 41% for a two context processor, and 12% to 60% for a four context uniprocessor over a single-threaded uniprocessor, despite decreased branch prediction accuracy and cache hit rates.

Chalainanont et al. [39] use the SPECjAppServer2002 benchmark as a representative workload to investigate the performance of L3 cache of Java Application servers. Shared L3 cache with sizes ranging from 4M to 1G are simulated utilizing the Programmable Hardware-Assisted Cache Emulator (PHA$E). Additionally, the impact of heap size and garbage collection method on the behavior of the L3s under study is analyzed. Heap sizes of 0.75G to 1.5G are simulated. Parallel and generational concurrent garbage collection methods are compared. Finally, the push cache technique that has the priority agent to place updated lines to the cache in place of sending read invalidate requests is evaluated.

Lixin et al. [153] compare the SPECjAppServer2002 and SPECjAppServer2004 workloads in terms of system behavior, execution profile and microarchitecture performance characteristics. Their results show that SPECjAppServer2004 demands significantly more system resources. The after GC heap size increases by more than 2 times, the disk traffic rises by more than 10 times. The new workload also stresses the JVM more than the Java code. The JVM's execution time portion increases by roughly 20% from 6.4% to 7.9%. At the microarchitectural level, the new workload results in about 13% more branches per instruction and a 19% higher branch misprediction ratio.

7.2 Tools/Techniques for QPN Analysis

A large body of work exists on analysis techniques for QN and SPN models. There have been numerous attempts to address the largeness problem, resulting in many exact and approximate analysis techniques. A number of numerical analysis methods, most of them based on model decomposition, have been proposed [30, 32, 47, 49, 56, 76, 77, 96]. Furthermore, many efficient simulation techniques for SPNs are available [57, 75, 79, 118]. Based on the above analysis techniques, numerous analysis tools for QNs and/or SPNs have been developed. Some of the more popular ones are TimeNET [58, 177], Möbius [48, 50], SPNP [46, 64], SMART [45], UltraSAN [142], GreatSPN [44], SHARPE [63], PEPSY-QNS [25], QNAT [74] and RASQ [71].

However, none of the above mentioned tools support QPN models. To the best of our knowledge [167], there is currently only one tool available for modeling and analysis using QPNs. This is the HiQPN-Tool [14] that we used in Chapter 4. The latter allows the hierarchical specification of QPN models and supports a number of methods that exploit the hierarchical structure for efficient numerical analysis. The main idea is to represent the huge generator matrix of the underlying Markov chain by much smaller matrices, each describing a submodel, which are combined using tensor operations [30, 31]. This technique, called *structured analysis*, allows models to be solved that are about an order of magnitude larger than those analyzable with conventional techniques [13]. A salient property of the hierarchical approach is that it does not depend on model symmetries and leads to exact results. The HiQPN-Tool supports a number of structured analysis methods, including Structured Power, Structured SOR and Structured JOR. In addition, some approximation analysis methods are supported. While the structured analysis approach does alleviate the largeness problem, as shown in Chapter 4, it does not eliminate it since models of realistic systems are still too large to be numerically tractable.

7.3 Performance Engineering of DCS

In this section, we review some related work on methodologies for performance engineering of DCS exploiting model-based performance prediction techniques.

Mania and Murphy [100] have been working on a technique that automatically builds performance models from trace files generated by non-intrusive monitoring of a J2EE-based server. The model helps developers to understand the behavior of the system and improve its performance. However, no implementation of the framework is available yet.

Larsson et al. [90] describe a methodology for predicting the behavior of a product software system before it is built. They have proposed the use of a prediction enabled component technology for developing and maintaining a component-based product line architecture in the real-time system's domain. They illustrate predictability of assemblies for the specified component model considering two concrete assembly's properties from a real-time product line's point of view: version consistency and end-to-end deadline.

Dumitrascu et al. [53] have proposed a high-level methodology for predicting the performance of component-based applications. Their methodology is focused on Microsoft .NET-based systems and is based on creating performance profiles for .NET components, assemblies and connection types. A "divide-and-conquer"strategy is used to estimate the performance attributes of an assembly on the basis of the performance attributes of the single components. No implementation of the methodology is available yet.

Chen et al. [41, 42] propose an empirical approach to determine the performance characteristics of component-based applications by benchmarking and profiling. A set of test cases are run to exercise components and measure their behavior. Empirical measures are then used to construct a performance model that describes the generic performance behavior of the respective class of component-based applications (for e.g. J2EE, .NET or CORBA). Applications are viewed as black-boxes and the models used are very simplistic because they do not capture any application-specific behavior. As a result, the information that can be obtained by analyzing the models is very limited.

Balsamo and Marzolla [7, 103] propose an algorithm for deriving simulation models from software architecture specifications. The proposed algorithm generates a process-oriented simulation model from annotated UML use case, activity and deployment diagrams. Simulation provides performance results that are inserted into the UML diagrams as tagged values. The methodology has been implemented into a prototype tool called UML-ψ (UML Performance SImulator). Currently only active resources are supported.

In [105] Mc Guinness et al. use the open-source modeling tool Ptolemy II [17] to build a simulation model of an EJB system. The effect of several model input parameters (e.g. number of CPUs, CPU speeds, number of threads, etc.) on the system performance is analyzed. However, the system modeled is too simple to be representative of real-world systems. Moreover, the model is not validated against the real system to verify if assumptions made when constructing it are valid. Therefore, it is not clear if the results presented can be trusted.

Bertolino and Mirandola [19] propose an original approach, called the CB-SPE, for component-based software performance engineering. CB-SPE builds on, and adapts to a CB framework, the concepts and steps of the SPE technology and uses for modeling the standard RT-UML profile, reshaped according to the CB principles. The approach is compositional in that it is applied first at the component layer for achieving parametric performance evaluation of the component in isolation, and then at the application layer for predicting the performance of the assembled components on the actual platform. In [20, 21] the approach is extended and a concrete implementation is presented.

Wu et al. [175] describe an approach for component-based performance prediction based on performance submodels for each system component, and a system assembly model to describe the binding together of library components and new components into a product. A component can be arbitrarily complex, including a subsystem of concurrent processes. The description pays particular attention to identifying the information that must be provided with the components, and with the bindings, and to providing for parameterization to describe different configurations and workloads. LQNs are exploited as underlying modeling formalism.

A large body of work exists on more general approaches for software performance engineering. In [6] Balsamo at al. present a comprehensive review of recent research in the field of model-based performance prediction at software development time.

While a lot of work has been done to support the performance engineering of DCS, this work is limited in the following ways:

- Most of the approaches proposed have never been implemented.
- To the best of our knowledge, none of the approaches have been validated on real-world DCS of the size and complexity of the ones considered in this thesis.
- None of the approaches have been shown to provide the modeling accuracy and predictive power of the approach proposed in this thesis.

Chapter 8

Summary and Outlook

For which one of you, when he wants to *build* a tower, does not first sit down and *estimate* the cost to see if he has enough to complete it?
– *The Bible, Luke 14:28*

Do not plan a bridge capacity by counting the number of people who swim across the river today.
– *Heard at a Presentation*

In this thesis, we have proposed a systematic approach for performance engineering of DCS that helps to identify performance and scalability (P&S) problems early in the development cycle and ensure that systems are designed and sized to meet their QoS requirements. The proposed approach builds on the fact that the P&S of a DCS is a function of the P&S of the hardware and software platforms it is built on, and the P&S of the system design. Therefore, if a DCS is to provide good P&S, both the platforms used and the system design must be efficient and scalable. To this end, we suggest that in the beginning of the system life cycle, the P&S of the platforms chosen are validated using *standard benchmarks* and that *performance models* are then exploited to evaluate the system P&S throughout the development cycle. In contrast to the "fix-it-later"approach, this approach helps to identify P&S problems early in the system life cycle and eliminate them with minimal overhead. The specifics of the proposed approach and the contributions of the thesis are now summarized.

In the first part of the thesis, we advocated the use of industry-standard benchmarks to evaluate the P&S of the hardware and software platforms used to build DCS. We focused on J2EE-based platforms since they are currently the technology of choice for DCS. We discussed the five most important requirements that benchmarks must fulfill in order for them to provide reliable results: they must be representative of real-world systems, must exercise and measure all critical services provided by platforms, must not be tuned/optimized for specific products, must generate reproducible results and must not have any inherent scalability limitations. Unfortunately, until recently, benchmarks used in the J2EE industry were mostly proprietary and they failed to meet these requirements. There was a need for an industry-standard benchmark that would provide a reliable method to evaluate the P&S of J2EE platforms. In **Chapter 2**, we presented the ECperf benchmark and its successor benchmarks SPECjAppServer2001 and SPECjAppServer2002 which were developed with the goal to address this need. We discussed some scalability issues that we discovered in the design of these benchmarks and proposed solutions to address them. The proposed solutions were published in [84] and [83]. Furthermore, they were submitted both to the ECperf expert group at Sun Microsystems and to SPEC's OSG-Java subcommittee where they were discussed and approved.

The SPECjAppServer2001 and SPECjAppServer2002 benchmarks evolved into a new industry-standard benchmark for J2EE platforms in whose specification and development the author was involved as release manager and lead developer. The benchmark was called SPECjAppServer2004 and it was released in April 2004. This benchmark provided a *new* enhanced workload exercising all major services of J2EE platforms in a complete end-to-end application scenario. SPECjAppServer2004 quickly established itself as the state-of-the-art industry-standard benchmark for J2EE-based platforms and it enjoys extreme popularity and market adoption for a benchmark of this size and complexity. The latter is due to the following advantages that SPECjAppServer2004 provides [82]:

1. It models a realistic application and workload.

2. It exercises all major services of J2EE platforms and measures the end-to-end platform P&S.

3. It has been tested on multiple hardware and software platforms and has proven to scale well from low-end desktop PCs to high-end servers and large clusters.

4. It has not been optimized for any specific platform and provides a level playing field for performance comparisons.

5. It can be used for stress/regression testing and bottleneck analysis.

6. Official benchmark results are made publicly available and can be used free of charge. Prior to publication, results are reviewed by subject experts making sure that the benchmark was run correctly and results are valid.

7. Last but not least, the benchmark can be used as a sample (blueprint) application, demonstrating J2EE best practices and design patterns for building scalable applications.

Doing benchmarking prior to starting system development ensures that P&S problems in the platforms used are discovered and addressed in time. Applications can then be developed with confidence that there are no bottlenecks or inefficiencies in the platforms employed. However, while the main purpose of benchmarks like SPECjAppServer2004 and its predecessors is to measure the P&S of platforms, they could equally well be exploited to study the effect of different platform configuration settings and tuning parameters on the overall system performance. Thus, benchmarking not only helps to choose the best platform and validate its P&S, but also helps to identify the configuration parameters most critical for P&S. In **Chapter 2**, we presented some case studies with ECperf and SPECjAppServer2004 that demonstrated this. The case studies were published in [83] and [88].

While building on a scalable and optimized platform is a necessary condition for achieving good P&S, unfortunately it is not sufficient. The application, i.e. the DCS, built on the selected platform must also be designed to be fast and scalable. The second major contribution of this thesis was the development of a performance engineering framework for DCS that provides a method to evaluate the P&S of the latter during the different phases of their life cycle. This helps to identify design problems early in the development cycle and have them resolved in time. The framework is made up of two parts. The first part, presented in **Chapter 5**, provides a tool and methodology (called SimQPN) for analyzing QPN models by means of simulation, circumventing the state-space explosion problem. This allows QPN models of realistic DCS to be analyzed. The contributions in this area were accepted for publication at [87]. The second part of the framework, presented in **Chapter 6**, provides a practical performance modeling methodology that shows how to model DCS using QPNs and use the models for performance evaluation. The methodology has been submitted for publication at [81] and is currently under review. Being based on QPN models, our approach has the following benefits:

1. QPN models allow the integration of hardware and software aspects of system behavior and lend themselves very well to modeling DCS.

2. QPN models combine the modeling power and expressiveness of the conventional queueing network and stochastic Petri net models and allow complex aspects of system behavior to be accurately modeled.

3. By restricting ourselves to QPN models, we can exploit the knowledge of their structure and behavior for fast and efficient analysis using simulation. This enables us to analyze models of large and complex DCS and ensures that our approach scales to realistic systems.

4. QPNs can be used to combine qualitative and quantitative system analysis. A number of efficient qualitative analysis techniques from Petri net theory are readily available and can be exploited.

5. Last but not least, QPN models have an intuitive graphical representation that facilitates model development.

In **Chapter 4**, we presented two practical performance modeling case studies in which we studied realistic DCS using conventional modeling and analysis techniques. The studies illustrated the difficulties stemming from the limited model expressiveness, on the one hand, and from the limitations in the available analysis methods, on the other hand. The need for more robust and scalable model analysis techniques was discussed and motivated. The two case studies were published in [85] and [86], respectively. In **Chapter 6**, another modeling case study was presented which showed how the problems mentioned above can be addressed using the proposed performance engineering framework. The case study, which was included in [81], can be seen as a validation of our approach.

8.1 Ongoing and Future Work

The SPECjAppServer benchmark suite will continue to be extended and enhanced as the J2EE platform evolves. An important extension will be the inclusion of functionality to exercise Web services which were introduced in J2EE 1.4. Another important change will be the migration of the benchmark to the J2EE 1.5 programming model which would make deployment of the benchmark much easier and reduce the learning curve faced when setting up the benchmark for the first time.

The work on SimQPN will also be continued. A graphical frontend (GUI) to the tool is planned which will make the tool easier to use and more user-friendly. Furthermore, the following enhancements will be made in future versions of the tool:

- Support for timed transitions and immediate queueing places.
- Support for load-dependent service demands.
- Support for deterministic distributions.
- Support for more scheduling strategies.
- Support for hierarchical queueing Petri nets (HQPNs).
- Support for parallel simulation.
- Support for additional methods for determining the length of the initial transient and for output data analysis.

Another area of future work is the extension of the proposed performance engineering framework with modeling templates (patterns) for DCS. Templates would speed up model development by making it possible to reuse models for multiple systems. It would also be interesting to explore the possibilities to automatically generate performance models or parts thereof from higher-level system models defined using UML diagrams. The estimation of service demands at the early stages of system development is another problem that needs further research.

Finally, the proposed performance modeling methodology can be exploited as a basis to build a framework for developing autonomic self-managing systems.

Bibliography

[1] C. Alexopoulos and D. Goldsman. To Batch Or Not To Batch. *ACM Transactions on Modeling and Computer Simulation*, 14(1):76–114, Jan. 2004. 108, 109, 111

[2] C. Alexopoulos and A. Seila. Output Data Analysis for Simulations. In *Proceedings of the 2001 Winter Simulation Conference, Arlington, VA, USA, December 9-12*, 2001. 108, 109

[3] D. Almaer. Making a Real World PetStore. TheServerSide.com J2EE Community, Apr. 2002. http://www.theserverside.com/articles/article.tss?l=PetStore. 17, 178

[4] Apache Software Foundation. Jakarta Tomcat, 2004. http://jakarta.apache.org/tomcat/. 44

[5] O. Balci. Validation, Verification and Testing Techniques throughout the Life Cycle of a Simulation Study. *Annals of Operations Research*, 53: 121–174, 1994. 113

[6] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5), May 2004. 183

[7] S. Balsamo and M. Marzolla. A Simulation-Based Approach to Software Performance Modeling. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 10th ACM SIGSOFT International Symposium on Software Engineering*, 2003. 182

[8] J. Banks, J. S. Carson II, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, Upper Saddle River, NJ 07458, third edition, 2001. 108, 110, 113

[9] G. Basilio. Characterization and Evaluation of the BEA WebLogic JRockit JVM on the Itanium II platform. Master's thesis, Royal Institute of Technology, Department of Microelectronics and Information Technology, SE-100 44 Stockholm, SWEDEN, 2003. 180

[10] F. Bause. "QN + PN = QPN" - Combining Queueing Networks and Petri Nets. Technical report no.461, Department of CS, University of Dortmund, Germany, 1993. 63, 66

[11] F. Bause. Queueing Petri Nets - A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models, Toulouse, France, October 19-22*, 1993. 7, 57, 63, 64, 66, 130

[12] F. Bause and P. Buchholz. Queueing Petri Nets with Product Form Solution. *Performance Evaluation*, 32(4):265–299, 1998. 63

[13] F. Bause, P. Buchholz, and P. Kemper. Hierarchically Combined Queueing Petri Nets. In *Proceedings of the 11th International Conference on Analysis and Optimization of Systems, Discrete Event Systems, Sophie-Antipolis (France)*, 1994. 7, 63, 67, 181

[14] F. Bause, P. Buchholz, and P. Kemper. QPN-Tool for the Specification and Analysis of Hierarchically Combined Queueing Petri Nets. In H. Beilner and F. Bause, editors, *Quantitative Evaluation of Computing and Communication Systems*, volume 977 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995. 90, 181

[15] F. Bause, P. Buchholz, and P. Kemper. Integrating Software and Hardware Performance Models Using Hierarchical Queueing Petri Nets. In *Proceedings of the 9. ITG / GI - Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, (MMB'97), Freiberg (Germany)*, 1997. 66, 97

[16] F. Bause and F. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg Verlag, second edition, 2002. xvi, 61, 62, 63, 65, 66

[17] Berkley University. Ptolemy II - an open-source software for modeling, simulation and design of concurrent, real-time systems, 2004. http://ptolemy.eecs.berkeley.edu/ptolemyII/. 182

[18] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, Inc., 1997. 30

[19] A. Bertolino and R. Mirandola. Towards Component-Based Software Performance Engineering. In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors, *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, Portland, Oregon, USA*, May 2003. 182

[20] A. Bertolino and R. Mirandola. CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Edinburgh, UK*, volume 3054 of *LNCS*, pages 233–248, May 2004. 183

[21] A. Bertolino and R. Mirandola. Software Performance Engineering of Component-based systems. In *Proceedings of the 4th International Workshop on Software and Performance (WOSP 2004), California, USA*, pages 238–242, Jan. 2004. 183

[22] P. Billingsley. *Probability and Measure*. John Wiley & Sons, 3nd edition, 1995. 110

[23] G. Bolch. *Performance Evaluation of Computer Systems with the help of Analytical Queueing Network Models*. Teubner Verlag, Stuttgart, 1989. 79, 123

[24] G. Bolch, S. Greiner, H. De Meer, and K. Trivedi. *Queuing Networks and Markov Chains - Modelling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, Inc., 1998. 59, 60

[25] G. Bolch and M. Kirschnick. The Performance Evaluation and Prediction SYstem for Queueing NetworkS - PEPSY-QNS. Technical Report TR-I4-94-18, University of Erlangen-Nuremberg, Germany, June 1994. http://www4.informatik.uni-erlangen.de/Projects/PEPSY/en/pepsy.html. 78, 79, 118, 123, 181

[26] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999. 135

[27] P. Brebner and J. Gosper. How scalable is J2EE technology? *ACM SIGSOFT Software Engineering Notes*, 28(3), May 2003. 179

[28] P. Brebner and J. Gosper. J2EE infrastructure scalability and throughput estimation. *ACM SIGMETRICS Performance Evaluation Review*, 31(3): 30–36, Dec. 2003. 179

[29] P. Brebner and S. Ran. Entity Bean A, B, C's: Enterprise Java Beans Commit Options and Caching. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms - Middleware, Heidelberg, Germany, November*, 2001. 47

[30] P. Buchholz. A hierarchical view of GCSPNs and its impact on qualitative and quantitative analysis. *Journal of Parallel and Distributed Computing*, 1992. 181

[31] P. Buchholz. A class of hierarchical queueing networks and their analysis. *Queueing Systems*, 1994. 181

[32] P. Buchholz. Adaptive decomposition and approximation for the analysis of stochastic petri nets. *Performance Evaluation*, 2004. 181

[33] P. Buzen and A. Shum. Model Calibration. In *Proceedings of the 1989 International CMG Conference, Reno, Nevada, USA, December 11-15*, pages 808–811, 1989. 146

[34] H. Cain, R. Rajwar, M. Marden, and M. Lipasti. An Architectural Evaluation of Java TPC-W. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 229–240, Jan. 2001. 180

[35] M. Calzarossa and G. Serazzi. Workload Characterization: a Survey. *Proceedings of the IEEE*, 8(81):1136–1150, 1993. 132

[36] J. Carson and A. Law. Conservation Equations and Variance Reduction in Queueing Simulations. *Operations Research*, 28, 1980. 113

[37] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Seattle, Washington, USA*, pages 246–261, 2002. 179

[38] CERN - European Organisation for Nuclear Research. The Colt Distribution - Open Source Libraries for High Performance Scientific and Technical Computing in Java, 2002. http://hoschek.home.cern.ch/hoschek/colt/. 107

[39] N. Chalainanont, E. Nurvitadhi, K. Chow, and S. L. Lu. Characterization of L3 Cache Bahavior of Java Application Server. In *Proceedings of the 7th Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2004. 180

[40] Charles University, Prague. Middleware Benchmarking. Distributed System Group's Web page, 2002. http://nenya.ms.mff.cuni.cz/projects.phtml?p=mbench&q=3. 179

[41] S. Chen, I. Gorton, A. Liu, and Y. Liu. Performance Prediction of COTS Component-based Enterprise Applications. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, 2002. 182

[42] S. Chen, Y. Liu, I. Gorton, and A. Liu. Performance Prediction of Component-based Applications. *Journal of Systems and Software*, Dec. 2003. 182

[43] C. Chien. Batch Size Selection for the Batch Means Method. In *Proceedings of the 1994 Winter Simulation Conference, Lake Buena Vista, FL, USA, December 11-14*, 1994. 111

[44] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1-2):47–68, Nov. 1995. Tool Homepage: http://www.di.unito.it/ greatspn/index.html. 181

[45] G. Ciardo, R. Jones, A. Miner, and R. Siminiceanu. SMART: Stochastic Model Analyzer for Reliability and Timing. In *Tools of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems, Aachen, Germany*, pages 29–34, Sept. 2001. Tool Homepage: http://www.cs.ucr.edu/ ciardo/SMART/. 181

[46] G. Ciardo, J. Muppala, and K. Trivedi. SPNP: Stochastic Petri Net Package. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models (PNPM89), Kyoto, Japan*, pages 142–151, 1989. 181

[47] G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Petri Net Models. *Performance Evaluation*, 18(1):37–59, 1993. 181

[48] T. Courtney, D. Daly, S. Derisavi, S. Gaonkar, M. Griffith, V. Lam, and W. Sanders. The Möbius Modeling Environment: Recent Developments. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST 2004), Enschede, The Netherlands*, pages 328–329, Sept. 2004. Möbius Homepage: http://www.mobius.uiuc.edu/. 181

[49] D. Deavours and W. H. Sanders. "On-the-fly"Solution Techniques for Stochastic Petri Nets and Extensions. *IEEE Transactions on Software Engineering*, 1998. 181

[50] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. Doyle, W. Sanders, and P. Webster. The Möbius Framework and Its Implementation. *IEEE Transactions on Software Engineering*, 28(10): 956–969, Oct. 2002. Möbius Homepage: http://www.mobius.uiuc.edu/. 181

[51] P. Denning and J. Buzen. The Operational Analysis of Queueing Network Models. *ACM Computing Surveys*, 10(3):225–261, Sept. 1978. 60, 73, 138

[52] Department of Industrial Engineering, North Carolina State University. ASAP3 Software For Steady-State Simulation Output Analysis, 2003. ftp://ftp.ncsu.edu/pub/eos/pub/jwilson/installasap3.exe. 108

[53] N. Dumitrascu, S. Murphy, and L. Murphy. A Methodology for Predicting the Performance of Component-Based Applications. In *Proceedings of the 8th International Workshop on Component-Oriented Programming (WCOP 2003), Darmstadt, Germany, July 21-25*, 2003. 182

[54] B. Everitt. *Cluster Analysis*. Halsted Press, New York, 2nd edition, 1980. 138

[55] J. Flowers and D. L.W. A comparison of calibration techniques for queuing network models. In *Proceedings of the 1989 International CMG Conference, Reno, Nevada, USA, December 11-15*, pages 644–655, 1989. 146

[56] J. Freiheit and A. Zimmermann. A Divide and Conquer Approach for the Performance Evaluation of Large Stochastic Petri Nets. In R. German and B. Haverkort, editors, *Proceedings of 9th International Workshop on Petri Nets and Performance Models (PNPM'01), Aachen, Germany*, pages 91–100, Sept. 2001. 181

[57] R. Gaeta. Efficient Discrete-Event Simulation of Colored Petri Nets. *IEEE Transactions on Software Engineering*, 22(9), Sept. 1996. 106, 181

[58] R. German, C. Kelling, A. Zimmermann, and G. Hommel. TimeNET - A Toolkit for Evaluating Non-Markovian Stochastic Petri Nets. *Performance Evaluation*, 24(1-2):69–87, Nov. 1995. 181

[59] I. Gorton and A. Liu. Performance Evaluation of EJB-Based Component Architectures. *IEEE Internet Computing*, 7(3):18–23, May 2003. 179

[60] I. Gorton, A. Liu, and P. Brebner. Rigorous Evaluation of COTS Middleware Technology. *IEEE Computer*, 36(3):50–55, Mar. 2003. 179

[61] P. Heidelberger and P. Welch. Simulation Run Length Control in the Presence of an Initial Transient. *Operations Research*, 31:1109–1145, 1983. 109, 111

[62] P. Hellekalek. On the assessment of random and quasi-random point sets. In P. Hellekalek and G. Larcher, editors, *Pseudo and Quasi-Random Point Sets*, Lecture Notes in Statistics. Springer-Verlag, New York, 1998. 107

[63] C. Hirel, R. A. Sahner, X. Zang, and K. S. Trivedi. Reliability and Performability Modeling Using SHARPE 2000. In *Computer Performance Evaluation / TOOLS 2000, Schaumburg, IL, USA*, pages 345–349, 2000. 78, 181

[64] C. Hirel, B. Tuffin, and K. Trivedi. SPNP: Stochastic Petri Nets. Version 6.0. In B. Haverkort, H. Bohnenkamp, and C. Smith, editors, *Computer performance evaluation: Modelling tools and techniques; 11th International Conference; TOOLS 2000, Schaumburg, Illinois, USA*, LNCS 1786. Springer Verlag, 2000. 181

[65] R. Hogg and A. Craig. *Introduction to Mathematical Statistics.* Prentice-Hall, Upper Saddle River, New Jersey, 5th edition, 1995. 110, 114

[66] G. Iazeolla, A. Lehmann, and H. Van Den Herik (Eds.). Simulation Methodologies, Languages, Architectures, AI and Graphics for Simulation. The Society for Computer Simulation International, La Jolla/USA, Rome/Italy June 7-9, 1989. 104

[67] IBM Software. Trade3 Web Application Server Benchmark, 2003. http://www-306.ibm.com/software/webservers/appserv/benchmark3.html. 5, 178

[68] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley-Interscience, New York, 1991. 138

[69] K. Jensen. *Coloured Petri Nets and the Invariant Method.* Mathematical Foundations on Computer Science, Lecture Notes in Computer Science 118:327-338, 1981. 61

[70] K. S. Juse, S. Kounev, and A. Buchmann. PetStore-WS: Measuring the Performance Implications of Web Services. In *Proceedings of the 29th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems - CMG2003, Dallas, TX, USA, December 7-12*, 2003. 179

[71] M. Kamath, S. Sivaramakrishnan, and G. Shirhatti. RAQS: A software package to support instruction and research in queueing systems. In *Proceedings of the 4th Industrial Engineering Research Conference, IIE, Norcross, GA.*, pages 944–953, 1995. 78, 181

[72] M. Karlsson, K. Moore, E. Hagersten, and D. Wood. Memory Characterization of the ECperf Benchmark. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI 2002), held in conjunction with the 29th International Symposium on Computer Architecture (ISCA-29), Anchorage, Alaska, USA*, May 2002. 180

[73] M. Karlsson, K. Moore, E. Hagersten, and D. Wood. Memory System Behavior of Java-Based Middleware. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, California, USA*, Feb. 2003. 180

[74] H. T. Kaur, D. Manjunath, and S. K. Bose. The Queuing Network Analysis Tool (QNAT). In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, San Francisco, California*, volume 8, pages 341–347, 2000. 78, 181

[75] C. Kelling. *Simulationsverfahren für zeiterweiterte Petri-Netze.* Advances in simulation, scs international, Technische Universität Berlin, 1995. In German. 181

[76] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Transactions on Software Engineering*, 22(9):615–628, Sept. 1996. 181

[77] P. Kemper. *Superposition of generalized stochastic Petri nets and its impact on performance analysis.* PhD thesis, Universität Dortmund, 1996. 181

[78] J. Kleijnen. Theory and Methodology: Verification and Validation of Simulation Models. *European Journal of Operational Research*, 82(1): 145–162, 1995. 113

[79] M. Knoke, F. Kühling, A. Zimmermann, and G. Hommel. Performance of a Distributed Simulation of Timed Colored Petri Nets with Fine-Grained Partitioning. In *Proceedings of the 2005 Design, Analysis, and Simulation of Distributed Systems Symposium (DASD 2005), San Diego, USA*, pages 63–71, Apr. 2005. 181

[80] S. Kounev. Eliminating ECperf Persistence Bottlenecks when using RDBMS with Pessimistic Concurrency Control. Technical report, Darmstadt University of Technology, Germany, Sept. 2001. http://www.dvs1.informatik.tu-darmstadt.de/∽skounev. 9, 30, 36, 38

[81] S. Kounev. A Methodology for Performance Modeling of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 2005. In review. 13, 187, 188

[82] S. Kounev. SPECjAppServer2004 - The New Way to Evaluate J2EE Performance. DEV2DEV Article, O'Reilly Publishing Group, 2005. http://www.dev2dev.com. 10, 186

[83] S. Kounev and A. Buchmann. Improving Data Access of J2EE Applications by Exploiting Asynchronous Processing and Caching Services. In *Proceedings of the 28th International Conference on Very Large Data Bases - VLDB2002, Hong Kong, China, August 20-23*, 2002. 10, 36, 38, 186, 187

[84] S. Kounev and A. Buchmann. Performance Issues in E-Business Systems. In *Proceedings of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet - SSGRR-2002w, L'Aquila, Italy, January 21-27*, 2002. 10, 26, 186

[85] S. Kounev and A. Buchmann. Performance Modeling and Evaluation of Large-Scale J2EE Applications. In *Proceedings of the 29th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems - CMG2003, Dallas, TX, USA, December 7-12*, 2003. This paper received Best-Paper-Award. 12, 188

[86] S. Kounev and A. Buchmann. Performance Modelling of Distributed E-Business Applications using Queuing Petri Nets. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software - ISPASS2003, Austin, Texas, USA, March 20-22*, 2003. This paper received the Best-Paper-Award at ISPASS-2003. 7, 12, 188

[87] S. Kounev and A. Buchmann. SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 2005. To appear. 12, 187

[88] S. Kounev, B. Weis, and A. Buchmann. Performance Tuning and Optimization of J2EE Applications on the JBoss Platform. *Journal of Computer Resource Management*, 113, 2004. 10, 41, 187

[89] S. Labourey and B. Burke. *JBoss Clustering*. The JBoss Group, 2520 Sharondale Dr., Atlanta, GA 30305 USA, 6th edition, 2003. 48

[90] M. Larsson, A. Wall, C. Norström, and I. Crnkovic. Using Prediction-Enabled Technologies for Embedded Product Line Architectures.

In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering, Orlando, USA*, 2002. 181

[91] A. Law and W. Kelton. Confidence Intervals for Steady-State Simulations, II: A Survey of Sequential Procedures. *Management Science*, 28 (5):550–562, 1982. 111, 113

[92] A. Law and W. D. Kelton. *Simulation Modeling and Analysis*. Mc Graw Hill Companies, Inc., third edition, 2000. 104, 108, 109, 110, 113

[93] J.-S. Lee, D. McNickle, and K. Pawlikowski. Confidence Interval Estimators for Coverage Analysis in Sequential Steady-State Simulation. In *Proceedings 22nd Australian Computer Science Conference, Auckland, New Zealand*, 1999. 114, 115

[94] J.-S. Lee, K. Pawlikowski, and D. McNickle. Experimental Coverage Analysis of Interval Estimators for Sequential Stochastic Simulation. In *Proceedings 1st Western Pacific/3rd Australia-Japan Workshop on Stochastic Models, Christchurch, New Zealand*, 1999. 114

[95] H. Leeb and S. Wegenkittl. Inversive and linear congruential pseudorandom number generators in empirical tests. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(2):272–286, Apr. 1997. 107

[96] Y. Li and C. M. Woodside. Iterative Decomposition and Aggregation of Stochastic Marked Graph Petri Nets. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets, Gjern, Denmark*, pages 257–275, 1991. 181

[97] J. Linton and C. Harmonosky. A Comparison of Selective Initialization Bias Elimination Methods. In *Proceedings of the 2002 Winter Simulation Conference, San Diego, California, USA, December 8-11*, 2002. 109

[98] J. D. C. Little. A proof of the queueing formula $L = \lambda W$. *Operations Research*, 9:383–387, 1961. 60

[99] P. Maly and C. Woodside. Layered Modeling of Hardware and Software, with Application to a LAN Extension Router. In *Proceedings of the 11th International Conference on Computer Performance Evaluation Techniques and Tools - TOOLS 2000, Motorola University, Schaumburg, Illinois, USA, March 27-31*, 2000. 66

[100] D. Mania and J. Murphy. Framework for Predicting the Performance of Component-Based Systems. In *Proceedings of the 10th IEEE International*

Conference on Software, Telecommunications and Computer Networks, Croatia-Italy, Oct. 2002. 181

[101] F. Marinescu. *Enterprise Java Beans Design Patterns.* John-Wiley & Sons, Inc., 2002. 23, 26

[102] G. Marsaglia. Diehard Battery of Tests of Randomness. Florida State University, 1995. http://stat.fsu.edu/pub/diehard. 107

[103] M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures.* PhD thesis, Dottorato di Ricerca in Informatica, II Ciclo Nuova Serie, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Jan. 2004. 182

[104] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 1998. 107

[105] D. McGuinness, L. Murphy, and A. Lee. Issues in Developing a Simulation Model of an EJB System. In *Proceedings of the 30th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems - CMG2004, Las Vegas, Nevada, December 5-10*, 2004. 182

[106] D. McNickle, K. Pawlikowski, and G. Ewing. Experimental Evaluation of Confidence Interval Procedures in Sequential Steady-State Simulation. In *Proceedings of the 1996 Winter Simulation Conference, Coronado, CA, USA, December 8-11*, 1996. 114

[107] D. Menascé. Two-Level Iterative Queuing Modeling of Software Contention. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis & Simulation of Computer & Telecommunications Systems (MASCOTS'02), Denton, TX, USA, October 11-16*, 2002. 66, 80

[108] D. Menascé and V. Almeida. *Capacity Planning for Web Performance: Metrics, Models and Methods.* Prentice Hall, Upper Saddle River, NJ, 1998. 4, 71, 73, 129, 133, 135, 137, 138

[109] D. Menascé and V. Almeida. *Scaling for E-Business - Technologies, Models, Performance and Capacity Planning.* Prentice Hall, Upper Saddle River, NJ, 2000. 60, 80, 91, 129

[110] D. Menascé, V. Almeida, R. Fonseca, and M. Mendes. A Methodology for Workload Characterization of E-commerce Sites. In *Proceedings of the 1st*

ACM conference on Electronic commerce, Denver, Colorado, United States, pages 119–128, Nov. 1999. 129, 135

[111] D. A. Menascé, V. A. Almeida, and L. W. Dowdy. *Capacity Planning and Performance Modeling - From Mainframes to Client-Server Systems*. Prentice Hall, Englewood Cliffs, NG, 1994. 6, 129, 134, 138, 144, 146

[112] D. A. Menascé, V. A. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004. 1, 129, 137, 138, 144

[113] D. A. Menascé and H. Gomaa. A Method for Desigh and Performance Modeling of Client/Server Systems. *IEEE Transactions on Software Engineering*, 26(11), Nov. 2000. 137

[114] Microsoft Corp. Microsoft .NET Platform. Specification, 2004. http://www.microsoft.com/net/. 2

[115] R. Miller. The Jackknife: A Review. *Biometrika*, 61:1–15, 1974. 111

[116] J. Mohr and S. Penansky. A forecasting oriented workload characterization methodology. *CMG Transactions*, 36, June 1982. 133

[117] Mort Bay Consulting. Jetty Java HTTP Servlet Server, 2004. http://jetty.mortbay.org. 44

[118] K. H. Mortensen. Efficient Data-Structures and Algorithms for a Coloured Petri Nets Simulator. In *Proceedings of the 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, August 29-31*, 2001. 106, 181

[119] E. d. S. Mota. *Performance of Sequential Batching-based Methods of Output Data Analysis in Distributed Steady-state Stochastic Simulation*. PhD thesis, Technical University of Berlin, School of Electrical Engineering and Computer Sciences, May 2002. 108

[120] MSDN Library. Using .NET to Implement Sun Microsystems' Java Pet Store J2EE BluePrint Application, Oct. 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/psimp.asp. 17, 178

[121] .NET Framework Community (GotDotNet). Microsoft .NET vs. Sun Microsystem's J2EE: The Nile E-Commerce Application Server Benchmark, Oct. 2001. http://www.gotdotnet.com/team/compare/nileperf.aspx, http://www.gotdotnet.com/team/compare/Nile 5, 178

[122] .NET Framework Community (GotDotNet). Microsoft .NET Pet Shop, 2002. http://www.gotdotnet.com/team/compare/petshop.aspx. 5, 17, 178

[123] .NET Framework Community (GotDotNet). Compare Microsoft .NET to J2EE Technology, 2003. http://www.gotdotnet.com/team/compare/. 5, 178

[124] T. Neward. The Pet Store.... Again. O'Reilly Developer Weblogs, Nov. 2002. http://www.oreillynet.com/pub/wlg/2253. 5, 17, 178

[125] B. Newport. Why prepared statements are important and how to use them properly. *TheServerSide.com J2EE Community*, 2001. http://www.theserverside.com/. 26

[126] Object Management Group, Inc. Common Object Request Broker Architecture (CORBA). Specification, 2004. http://www.corba.org/. 2

[127] ObjectWeb Consortium. RUBBoS: Bulletin Board Benchmark, 2002. http://jmob.objectweb.org/rubbos.html. 179

[128] ObjectWeb Consortium. JMOB: Java Middleware Open Benchmarking, 2003. http://jmob.objectweb.org/. 179

[129] ObjectWeb Consortium. RUBiS: Rice University Bidding System, 2003. http://rubis.objectweb.org/. 5, 179

[130] ObjectWeb Consortium. Stock-Online Project, 2003. http://forge.objectweb.org/projects/stock-online/. 5, 179

[131] D. Ockerman and D. Goldsman. The Impact of Transients on Simulation Variance Estimators. In *Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA, USA, December 7-10*, 1997. 109

[132] K. Pawlikowski. Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions. *ACM Computing Surveys*, 22(2):123–170, 1990. 108, 109, 110, 111, 112, 113

[133] K. Pawlikowski, D. McNickle, and G. Ewing. Coverage of Confidence Intervals in Sequential Steady-State Simulation. *Journal of Simulation Practice and Theory*, 6(3):255–267, 1998. Plus: "Erratum"7(1):1, 1999. 111, 114

[134] H. Perros. *Computer Simulation Techniques - The Definitive Introduction*. E-Book, NC State University, 2003. http://www.csc.ncsu.edu/faculty/perros/simulation.pdf. 108

[135] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, 2nd edition, 2000. 28

[136] S. Ran, D. Palmer, P. Brebner, S. Chen, I. Gorton, J. Gosper, L. Hu, A. Liu, and P. Tran. J2EE Technology Performance Evaluation Methodology. In *Distributed Objects and Applications 2002 (DAO'02), Proceedings (addendum) of "On the Move to Meaningful Internet Systems and Ubiquitous Computing", University of California, Irvine*, Oct. 2002. 179

[137] Real-Time and Distributed Systems Group, Carleton University, Ottawa, Canada. Layered Queueing Homepage. http://www.layeredqueues.org/, 2004. 66

[138] S. Robinson. A Statistical Process Control Approach For Estimating the Warm-Up Period. In *Proceedings of the 2002 Winter Simulation Conference, San Diego, California, USA, December 8-11*, 2002. 109

[139] J. Rolia and K. Sevcik. The Method of Layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, 1995. 66, 80

[140] E. Roman, S. Ambler, and T. Jewell. *Mastering Enterprise Java Beans II and the Java 2 Platform, Enterprise Edition.* John Wiley & Sons, Inc., 2002. 23, 30

[141] C. Rose. A measurement procedure for queuing network models of computer systems. *ACM Computing Surveys*, 10(3), 1978. 137

[142] W. Sanders, W. Obal, M. Qureshi, and F. Widjanarko. The UltraSAN modeling environment. *Performance Evaluation*, 1995. 181

[143] R. Sargent. Verification and Validation of Simulation Models. In A. Seila, S. Manivannan, J. Tew, and D. Sadowski, editors, *Proceedings of the Winter Simulation Conference, Lake Buena Vista, FL, USA, December 11-14*, 1994. 113

[144] T. Schriber and R. Andrews. A Conceptual Framework for Research in the Analysis of Simulation Output. *Communications of the ACM*, 24(4):218–232, 1981. 113

[145] C. Smith. Performance Engineering. In Encyclopedia of Software Engineering, J.J. Maciniak (ed.), John Wiley & Sons, 1994. pp. 794-810. 1

[146] C. U. Smith and L. G. Williams. *Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software.* Addison-Wesley, 2002. 2, 135

[147] Standard Performance Evaluation Corporation (SPEC). SPECjAppServer2004 Documentation. Specifications, Apr. 2004. http://www.spec.org/jAppServer2004/. 36, 39, 40

[148] S. Stark. *JBoss Administration and Development*. The JBoss Group, 2520 Sharondale Dr., Atlanta, GA 30305 USA, 3th edition, 2003. 47, 49

[149] N. Steiger, E. Lada, J. Wilson, C. Alexopoulos, D. Goldsman, and F. Zouaoui. ASAP2: An Improved Batch Means Procedure For Simulation Output Analysis. In *Proceedings of the 2002 Winter Simulation Conference, San Diego, California, USA, December 8-11*, 2002. 108

[150] N. Steiger, E. Lada, J. Wilson, J. Joines, C. Alexopoulos, and D. Goldsman. ASAP3: A Batch Means Procedure for Steady-State Simulation Output Analysis. *ACM Transactions on Modeling and Computer Simulation*, 82, 2003. ftp://ftp.ncsu.edu/pub/eos/pub/jwilson/tomacsv37.pdf. 108

[151] N. Steiger and J. Wilson. Improved Batching For Confidence Interval Construction in Steady-State Simulation. In *Proceedings of the 1999 Winter Simulation Conference, Phoenix, Arizona, USA*, 1999. 108

[152] N. Steiger and J. Wilson. Experimental Performance Evaluation of Batch Means Procedures for Simulation Output Analysis. In *Proceedings of the 2000 Winter Simulation Conference, Orlando, FL, USA, December 10-13*, 2000. 108

[153] L. Su, K. Chow, K. Shiv, and A. Jha. A Comparison of SPECjAppServer2002 and SPECjAppServer2004. In *Proceedings of the 8th Workshop on Computer Architecture Evaluation using Commercial Workloads - CAECW-8, San Francisco, February 12*, 2005. 36, 180

[154] A. Sucharitakul. Seven Rules for Optimizing Entity Beans. *Java Developer Connection*, 2001. 24, 25

[155] Sun Microsystems, Inc. The ECperf Benchmark. Specification, Apr. 2002. http://java.sun.com/j2ee/ecperf/. 18, 28

[156] Sun Microsystems, Inc. Java™Pet Store Demo, 2003. http://java.sun.com/developer/releases/petstore/. 5, 178

[157] Sun Microsystems, Inc. Enterprise JavaBeans. Specification, 2004. http://java.sun.com/products/ejb/. 20, 23, 31, 47

[158] Sun Microsystems, Inc. Java 2 Platform, Enterprise Edition (J2EE). Specification, 2004. http://java.sun.com/j2ee/. 2

[159] Sun Microsystems, Inc. Java Message Service. Specification, 2004. http://java.sun.com/products/jms/. 31

[160] Y. Tay, N. Goodman, and R. Suri. Locking Performance in Centralized Databases. *ACM Transactions on Database Systems*, 10/4, 1985. 27

[161] The Middleware Company. The ECperf Web Site at TheServerSide.com, 2002. http://ecperf.theserverside.com/ecperf. 18

[162] The Open Group. Distributed TP: The XA Specification. Specification, 1992. http://www.opengroup.org/public/pubs/catalog/c193.htm. 40

[163] TheServerSide.com J2EE Community. Oracle claims PetStore runs twice as fast on Oracle vs. IBM, BEA, Nov. 2001. http://www.theserverside.com/news/thread.tss?thread_id=10446. 17, 178

[164] Transaction Processing Performance Council. TPC Benchmark™ App (TPC-App). Specification, Dec. 2004. http://www.tpc.org/tpc_app/. 177

[165] Transaction Processing Performance Council. TPC Benchmark™ W (TPC-W). Specification, 2004. http://www.tpc.org/tpcw/. 177

[166] K. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, Inc., second edition, 2002. 60, 91, 110

[167] University of Aarhus. Petri Net Tool Database. Department of Computer Science - DIAMI, 2004. http://www.daimi.au.dk/PetriNets/tools/. 181

[168] Urbancode, Inc. Urbancode EJB Benchmark, 2002. http://www.urbancode.com/projects/ejbbenchmark/default.jsp. 5, 178

[169] S. Wegenkittl. The pLab Picturebook: Load Tests and Ultimate Load Tests, Part I. Technical report, University of Salzburg, 1997. ftp://random.mat.sbg.ac.at/pub/data/pLabReport1.ps. 107

[170] G. Weikum and G. Vossen. *Transactional Information Systems - Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann Publishers, 2002. 27, 29

[171] B. Weis. Performance Optimierung von J2EE Anwendungen auf der JBoss Plattform. Master thesis, Darmstadt University of Technology, Mar. 2004. In German. 41, 43

[172] P. Welch. On the Problem of the Initial Transiant in Steady-State Simulation. Technical report, IBM Watson Research Center, Yorktown Heights, New York, 1981. 109

[173] P. Welch. The Statistical Analysis of Simulation Results. The Computer Performance Modeling Handbook, ed. S. Lavenberg, Academic Press, NJ, 268-328, 1983. 109

[174] M. Woodside, J. Neilson, D. Petriu, and S. Majumdar. The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software. *IEEE Transactions on Computers*, 44(1):20–34, Jan. 1995. 66

[175] X. Wu, D. McMullan, and M. Woodside. Component Based Performance Prediction. In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors, *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, Portland, Oregon, USA*, May 2003. 183

[176] Y. Zhang, A. Liu, and W. Qu. Comparing Industry Benchmarks for J2EE Application Server: IBM's Trade2 vs Sun's ECperf. In M. J. Oudshoorn, editor, *Proceedings of the Twenty-Sixth Australasian Computer Science Conference (ACSC 2003), Adelaide, Australia.*, pages 199–206. ACS Conferences in Research and Practice in Information Technology (CRPIT), Jan. 2003. 180

[177] A. Zimmermann, J. Freiheit, R. German, and G. Hommel. Petri Net Modelling and Performability Evaluation with TimeNET 3.0. In *Proceedings of the 11th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'2000), Schaumburg, Illinois, USA*, LNCS 1786, pages 188–202, Mar. 2000. 181

Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades „Dr.-Ing.“ mit dem Titel „Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction“ selbstständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, den 03.05.2005 Samuel Kounev