INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

**Applying Layered Queueing Modelling
to a Telecommunication System**

by

Christiane Wahba Shousha

A thesis submitted to the Faculty of Graduate Studies in partial fulfillment of
the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

September 1, 1998

© Copyright 1998, Christiane W. Shousha

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-36897-1

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

Applying Layered Queueing Modelling to a Telecommunication System

submitted by Christiane Wahba Shousha, B. Sc.,
in partial fulfillment of the requirements for the degree of Master of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University

September 1, 1998

Abstract

Telecommunication systems software are growing in size and complexity, and consequently the effort of tuning for performance may become very time-consuming and hence costly to the industry. Performance modelling offers an efficient and timely means of estimating software design and implementation alternatives, and is much easier to apply than actually changing the real system. Although it is widely recognized that performance engineering is very important to the telecommunications industry, modelling techniques are not easily accessible to software developers. The most important obstacle is the cognitive gap between the software and the performance domains. This thesis presents the experience of the author in modelling an existing telecommunication product and in using the model results for feeding back recommendations for performance improvement to the software development team. The thesis also introduces the so-called “Software Designer Interface” to bridge the gap between the software designer’s view and the performance model in a first attempt to facilitate the running of performance experiments directly by software designers.

To my husband Ramy and my parents with all my love

Acknowledgments

I would like to thank my supervisor, Prof. Dorina Petriu, who has taught me a lot. You were always there to provide valuable technical guidance and assistance throughout all my work regardless of the busy load that you have. Thank you for always creating time for me even in your most busy schedule and thank you for never letting me feel that I am interrupting your work (although I know I did). I would also like to express my deep gratitude for your encouragement and giving me hope and motivation to go on. I was really honored to work under your supervision!

I would like to thank my advisor Prof. Woodside for his assistance and guidance during my course work, where I learned the principles of performance modelling and giving me the greatest push to start my research with Prof. Petriu.

This research would not have happened without the collaboration with the SEAL group of Nortel. Special thanks to Anant Jainapurkar for his technical assistance and guidance throughout my research and providing me with in-depth industrial insight into telecommunication systems. I would also like to thank Kennedy Ngo and Chung-Horng Lung. It was really a great experience working with the SEAL group.

I would like to thank the RADS group, especially Alex H. and Greg F..

Last but not least, I would like to thank my husband and my parents for always being there for me! Thank You.... You are the greatest !

Table of Contents

Abstract	ii
Acknowledgments	iv
Table of Contents	v
List of Figures	viii
List of Tables	xi
Chapter 1 Introduction	1
1.1 Thesis Motivation	1
1.2 Thesis Goals	4
1.3 Thesis Contributions	4
1.4 Thesis Outline	5
Chapter 2 Background Literature	7
2.1 History of Performance-oriented Software Systems	7
2.2 Capacity Planning	9
2.3 Software Performance Engineering	12
2.4 Performance Models of Software Systems.	14
2.4.1 Queueing Network Models.	14
2.4.2 Polling Review.	19
2.5 Layered Queueing Networks.	20
2.5.1 LQN Model Components.	21
2.5.2 LQN Model Graphical Representation.	24

2.5.3 LQN Assumptions	27
2.5.4 LQN Parameters.	28
2.5.5 Solving LQN Models.	28
2.6 Measurement Tools.	29
2.6.1 Quantify.	29
Chapter 3: Extensions to LQN simulator	31
3.1 Solving Performance Models by Simulation	31
3.2 PARASOL Simulation System.	32
3.3 ParaSRVN Solver.	33
3.4 Extensions to Parasrvn.	35
3.4.1 Polling Servers	36
3.4.2 Queue Length Statistics	38
3.4.3 Enhancements for solving Open Arrival systems	39
Chapter 4: Concrete Industrial Experience	42
4.1 Software Architecture of the System	42
4.1.1 Description of the Critical Path Scenario	46
4.2 Extracting and Building the Base LQN Model	48
4.2.1 Mapping System Components to LQN	48
4.2.2 LQN Model of the System	52
4.2.3 Measurements and Validation	54
4.3 Study 1: Bottleneck Shift Detection on Proposed System	56
4.3.1 Performance Analysis of the Model	57

4.3.2 Performance Analysis for Modified Model	61
4.3.3 Performance Analysis Recommendations	64
4.4 Study 2: Performance Analysis of Base Model and Alternatives	65
4.4.1 Studied Cases	66
4.4.2 Results and Analysis	67
4.4.3 Performance Analysis Recommendations	75
4.5 Study 3:Effect of Polling on Base System.	76
4.5.1 Polling Modifications on the LQN Base Model	76
4.5.2 Performance Study with a very Large Polling Overhead	77
4.5.3 Performance Study with a very Small Polling Overhead... ..	81
Chapter 5: Software Designer Interface.	85
5.1 SDI and SPE	85
5.2 Definition of SDI.	87
5.2.1 SDI Functions from a Software Designer's Perspective	90
5.2.2 SDI Functions from a Performance Analyst's Perspective	92
5.3 SDI vs. SPEX	94
5.3.1 Overview of SPEX.	95
5.3.2 Using SPEX to Design SDI	96
Chapter 6: Conclusions.	99
6.1 Future Work.	100
References.	102

List of Figures

Figure 2-1: Graphical example of a situation requiring capacity planning	10
Figure 2-2: Graphical representation of the capacity planning concept	11
Figure 2-3: Example of a closed queueing network	15
Figure 2-4: Example of an open queueing network	16
Figure 2-5: Server task cycle and entry phases [Woodside95b]	24
Figure 2-6: Sample LQN model of a DBase application	25
Figure 3-1: Pseudo-code for <code>run_simulation()</code> pararvn function	35
Figure 3-2: Pseudo-code for <code>srn_polling_server()</code> ParaSRVN function	37
Figure 3-3: Pseudo-code for <code>srn_polling_worker()</code> ParaSRVN function	38
Figure 3-4: Pseudo-code for <code>srn_polling_queue()</code> ParaSRVN function	38
Figure 3-5: Pseudo-code for manipulating queue length statistics	39
Figure 3-6: Pseudo-code for gracefully reducing asynchronous messages in system	41
Figure 3-7: Pseudo-code for gracefully reducing asynchronous messages in system	41
Figure 4-1: Multiprocessing architecture	43
Figure 4-2: Data multithreading architecture	43
Figure 4-3: Scenario for an incoming request	45
Figure 4-4: Software architecture of the critical path	47
Figure 4-5: Mapping critical sections in LQN	49
Figure 4-6: Mapping communication between stages of the pipeline	49

Figure 4-7: Illegal cyclic LQN model	50
Figure 4-8: Valid cyclic LQN model	50
Figure 4-9: Transforming an open to a closed LQN model	51
Figure 4-10: LQN base model of the system	52
Figure 4-11: Study1: LQN base model of the system	58
Figure 4-12: Utilizations for 1 processor	59
Figure 4-13: Utilizations for 4 processors	60
Figure 4-14: Utilizations for 6 processors	61
Figure 4-15: LQN model of the modified system	62
Figure 4-16: New utilizations for 4 processors	63
Figure 4-17: New Utilizations for 6 processors	64
Figure 4-18: EventHandler Service time on behalf of the request	68
Figure 4-19: Total Response time of a request	71
Figure 4-20: Queueing delay prior to EH service for the four cases	72
Figure 4-21: EH Processor Utilization for the four cases	73
Figure 4-22: Waiting time to acquire ShMem1 semaphore	74
Figure 4-23: Utilizations for 1 processor	78
Figure 4-24: Utilizations for 4 processors	79
Figure 4-25: Utilizations for 6 processors	80
Figure 4-26: Achievable throughput for 1,4 and 6 processors	80
Figure 4-27: Utilizations for one processor	82
Figure 4-28: Utilizations for four processors	82

Figure 4-29: Utilizations for six processors	83
Figure 4-30: Achievable throughput for 1,4 and 6 processors	83
Figure 5-1: Extract from an LQN model input file	95
Figure 5-2: Pseudo-code for SDI main routine	97

List of Tables

Table 3-1: LQN task and corresponding service routines	34
Table 3-2: New LQN task and corresponding service routines	36
Table 3-3: New polling statistics specifications in the LQN input file	37
Table 3-4: New statistics specifications in the LQN input file	39
Table 3-5: New statistics specifications in the LQN input file	40
Table 4-1: Measurements from Quantify and top	56
Table 4-2: Cases studied	67
Table 4-3: Response Time of EventHandler on behalf of the request	69
Table 4-4: Response Time of EventHandler on behalf of the request for cases Case2 and Case4 at the higher request rates	70
Table 4-5: Performance Results for Processor Utilization	73
Table 4-6: Simulation results for the waiting time to acquire ShMem1 semaphore	75

Chapter 1

Introduction

1.1 Thesis Motivation

In a globally networked economy, the market moves fast and the windows of opportunity for various products are very narrow. This implies that the products and solutions have to be delivered to the customers *just-in-time* and be up to the customer expectations. The success of today's industry heavily depends on the ability to interpret market signals and to respond faster than the competition. The industry is continuously challenged by the need to reduce the *time-to-market*, which implies the shortening of the software development cycle. However, in many cases software projects either consume more time/resources than planned or allocated, or the released systems don't meet their performance requirements.

Software performance engineering (SPE) can provide a solution to the latter problem [Smith90]. Performance modelling and analysis can be started from the outset when the design of the system is still being assessed, and then continued through the software development cycle, as more

and more information becomes available. SPE provides performance predictions about the system, before the system is completely implemented. Examples of performance metrics that are useful to software designers are achievable throughput, response time of the system and resource utilization under different loads. Traditionally, such metrics would be collected only by measuring the system after it has been completely implemented. The downside of this approach is that *fixing* performance problems at such a late stage in the software is very costly and difficult. SPE proposes to avoid late *fixes* by building performance models and analyzing them in the early development stages.

Early performance predictions provide the designer with feedback about the current design and recommend exploring alternative solutions if the performance requirements cannot be met, saving the company the cost of implementing a bad design and the subsequent changes to fix performance problems. Building a software performance model is a crucial step of SPE. The model has to capture the collaboration between software processes, their resource requirements and their contention for hardware and software resources.

Performance analysis of the model provides insight on potential bottlenecks of the system and provides the designers with the opportunity to address such concerns before they escalate and become too complex to solve. It is a well-known fact that the earlier performance problems are caught, the less complex and subsequently less costly is the fix. The model can also be used for capacity planning, helping to find the software and hardware configuration that can insure desired performance metrics for different workload levels.

The emerging telecommunications field, where new services are being provisioned, added and deleted dynamically, is a field that changes at a very fast pace. Telecommunication products in this field must be developed in a short time, and they have rather stringent performance requirements. This means that such products would benefit from Software Performance Engineering.

The first motivation of this thesis is to apply the software performance modelling method of Layered Queueing Networks (LQN)¹ and its performance toolset to a real telecommunication product. The construction of the model and its analysis are presented in Chapter 4. We have also extended the toolset with some features that were required in order to analyze the system, as reported in Chapter 3.

Even though it is widely recognized that performance engineering is very important to the telecommunications industry, modelling techniques are not easily accessible to software developers. This is mainly due to the cognitive gap between the software and performance domains, as software developers are not usually trained in performance modelling.

This thesis attempts to reduce the gap between the software development domain and the performance engineering domains by introducing the idea of a Software Developer Interface (SDI), as described in Chapter 5.

1.LQN is a software performance modelling technique developed at Carleton University and is presented in Chapter 2 of this thesis.

1.2 Thesis Goals

The goal of this thesis is to contribute to the integration of performance engineering in the software development cycle and to achieve a better understanding of how to bridge the gap between software development and performance engineering.

The current thesis has explored two fronts:

- the application of an academically developed performance engineering analysis modelling technique to an industrial product and the analysis of the system under different workloads and configurations.
- making performance tools more accessible to software developers who do not have the background of a performance analyst.

1.3 Thesis Contributions

The research contributions of this thesis are the following:

- Applied the Layered Queueing Network modelling technique to an existing telecommunication product.
- Conducted performance analysis of the model for capacity planning and for feeding back recommendations for performance improvement to the software development team.
- Extended ParaSRVN, the simulator tool for the Layered Queueing Modelling with three features:

- Adding a software polling-server.
 - Monitoring and providing statistics for the Task/Entry queue length.
 - Dealing gracefully with open arrival rates that are higher than the rate at which the system reaches saturation.
- Designed a Software Developer Interface to facilitate the running of performance experiments directly by software developers.

Some of the results of this thesis were accepted as an experience paper [Shousha98] at the ACM Workshop on Software and Performance, to be held in Santa Fe in October 1998.

1.4 Thesis Outline

Chapter 2 provides a bibliographical review related to Software Performance Engineering, discusses briefly one of the best known models Queueing Networks, and provides a more elaborate introduction on Layered Queueing Networks, a type of extended queueing networks, which is the software modelling technique used throughout this thesis.

Chapter 3 presents a detailed description of the ParaSRVN simulator for LQN models. It provides information on the underlying PARASOL simulation tool in order to present the new components that were added to the simulator. The new components presented in Chapter 3 are the extension of ParaSRVN to support software polling servers (single and multiservers), the addition of queue length statistics and the enhancement for solving open arrival systems.

Chapter 4 steps through the industrial application of LQN by first describing how the LQN model for the system was built. It then presents three performance studies, two that explore the parameter space under different software and hardware architectures, and the third that analyzes the effect of polling by the software processes.

Chapter 5 presents the so-called Software Designer Interface, which provides a front-end as well as a back-end interface to hide the performance model from the user. The chapter provides first a brief overview of SPEX¹, an experiment controller aimed at the performance analysts, then presents SDI as an extension of SPEX.

Chapter 6 concludes the thesis research, summarizes contributions and looks ahead at future research.

1. SPEX is a software performance experiment controller tool developed at Carleton University.

Chapter 2

Background Literature

This chapter presents an overview of software performance engineering and capacity planning. Performance models widely used, such as Queueing Networks (QNs), are reviewed in Section 2.4. A brief literature survey on the scheduling discipline of polling for QN is presented in Section 2.4.2. Finally, the Layered Queueing Networks (LQN) model, which is the extended queueing network for distributed software systems used throughout the thesis will be detailed in Section 2.5.

2.1 History of Performance-oriented Software Systems

Many software developers in the 1960s and 1970s were experts at performance-oriented design and coding by necessity, due to the reverse hardware constraints (low processing power, small memory). The rapid development in hardware technology has released considerably these constraints. Software developers started building software without worrying about performance, and adopted the *fix-it-later* approach, which assumes that one can tune the software for performance later after it has been already implemented. This led to the belief that the way to increase software system efficiency and improve performance is through *tricky code* [Smith90].

Today's trend in software development is still heavily influenced by the *fix-it-later* performance tuning approach, meaning that many software developers are not concerned with the performance implications of design decisions [Smith90]. They have grown to believe that increased hardware capacity is the remedy to performance problems in many cases. With hardware technology improving, more complex and larger software systems are now the norm. However, counter-intuitive as it may seem even though hardware is getting more powerful, this is not enough to solve all the performance problems.

The dangers of the *fix-it-later* were clearly presented in [Smith90]. The most serious flaw of this approach is that performance problems are detected only at system integration and maintenance stages, hence potentially delaying software delivery. If the software is delivered to customers with poor performance, negative impressions are built that will eventually weaken the business. The performance tuning will require changes to the program code, retesting, and in some cases even redesign. Revisiting many steps from the software development life-cycle takes time and introduces unscheduled delays. Ad hoc tuning leads to increased complexity, increased cost for maintenance and for any future enhancements.

Software Performance Engineering (SPE), proposed by [Smith90] advocates that performance modelling and analysis start at the beginning of the software development and continue throughout the whole life cycle.

Today, performance is vital to many systems. For example, real-time systems, such as shuttle-control-systems or flight-control-systems, have critical response-time requirements that they must

meet in addition to strict reliability and correctness requirements. There is also the managerial cost aspect among the reasons to conduct performance studies. However, in some cases the performance of a company may be perceived directly by the performance of its software products or server systems. Degraded performance may in extreme cases lead to loss of customers.

2.2 Capacity Planning

The capacity of any system is defined by its maximum performance. Planning the capacity of any system is a common-practice activity and is a long-known concept in the business field. For computer systems, however there is usually a strong concern regarding the functionality of the system and very little about its performance [Menasce94].

User dissatisfaction and productivity decrease are among the primary reasons for capacity planning. Budgetary constraints and risks of financial losses are key drivers as well [Lazowka84] [Menasce94].

Capacity Planning is the determination of the *predicted workload* and *time* in the future when the system will saturate and the determination of the most cost-effective way of delaying saturation as long as possible. Modelling Techniques for predicting software system performance will be described later (see Section 2.4).

Figure 2-1 depicts a graphical representation of a typical situation that calls for capacity planning [Menasce94]. The solid curve in the graph shows the predicted evolution of the response time for

the analyzed system, whereas the dashed line is the predicted for a different configuration, for example a faster disk.

Figure 2-1: Graphical example of a situation requiring capacity planning [Menasce94]

The following figure (Figure 2-2) describes the concept of capacity planning. Capacity planning predictions must take into consideration workload evolution, which may be due to existing or new applications, as well as the desired service levels along with the system parameters in order to output the saturation points and recommend cost-effective design solutions [Menasce94].

Figure 2-2: Graphical representation of the capacity planning concept [Menasce94]

Hence, the capacity planning methodology is composed of the following basic five steps [Menasce94]:

- Obtaining a global understanding of the current environment.
- Workload characterization by identifying the basic components and then choosing the characterizing parameters and collecting the necessary data. The workload may be partitioned for different classes of requests, e.g. batch request or interactive requests.
- Validation by solving the system model via simulation or analytical techniques and comparing against actual measurements.
- Calibration of the model of the system to match the observed system behavior. Note that several iterations may be necessary to calibrate and validate the model.

- Prediction of the future performance of the system by applying the forecasted workload on the validated performance model of the system. Different scenarios reflect different features or different workloads or a combination of both. Analyzing different input scenarios, the most cost-effective configuration for the future system is found.

2.3 Software Performance Engineering

Software Performance Engineering (SPE) introduced in [Smith90], is a more encompassing software performance method than Capacity Planning. SPE views performance as an integral part of software development along with functionality correctness and software quality factors such as usability, maintainability etc. SPE is a performance practice for constructing software systems that meet their performance objectives, which spans the whole software life-cycle. The SPE process begins early in the software development cycle and uses quantitative methods to identify whether the designs are satisfactory or should be eliminated in favor of other alternative designs before significant time is invested in the development of an unacceptably performing design.

SPE Benefits

- Increases productivity, when developers' level of effort is not spent in implementations that will later be discarded. SPE methods and tools increase productivity by detecting problems that may potentially delay delivery thereby preventing tricky-code maintenance. SPE advocates that implementation and testing focus on the critical parts of the software.

- Improves quality and usefulness of the resulting software product by avoiding tuning complexity and selecting the suitable design prior to implementation.
- Controls costs of the supporting hardware and software by identifying the required equipment ahead and allowing time for competitive vendor procurement.

SPE Performance Assessment Activities

As shown in [Smith90], with SPE the standard design steps are conducted as usual, then a performance assessment step is conducted and when the predicted performance is satisfactory, development moves to the next step.

The SPE performance assessment steps/activities are:

- Build the Software Execution Model, a kind of flow-graph that follows the execution of the software for a set of frequent scenarios.
- Gather resource requirements for every software module represented in the graph, then aggregate the resource requirements.
- Map the Software Execution model to a System Model that contain both the hardware and the software of the system.
- The System model is then solved, if applicable either analytically or with simulation. Since the precision of the model results depends on the precision of the fed-in resource estimates and since these are difficult to estimate precisely in early steps, a best- and worst-case analysis strategy must be conducted. When there is high uncertainty about resource requirements, estimates about the

lower and upper bounds are used. If the worst-case performance is satisfactory, then designers may proceed to the next step in the software development cycle.

Ongoing activities of the SPE approach are verifying that models correctly represent the software system behavior and to validate model predictions against performance measurements, studying discrepancies, identifying error sources and then calibrating the model as necessary.

2.4 Performance Models of Software Systems

A model is an approximate representation of a system. Mathematical models are used when the observational phenomenon has measured properties. There are two types of models: simulation models and analytical models. Simulation models functionally mimic the behavior of the studied system and consequently are slower to solve. On the other hand, analytical models, do not have operations that mimic the behavior of the system, but capture the system properties by mathematical equations. Analytical models are usually faster to solve than simulations.

2.4.1 Queueing Network Models

Mathematical models, such as Markov models are usually used to model small systems such as a single queue system [Menasce94]. However, computer systems are characterized by series of servers (resources or processing units) that cooperate to accomplish a certain task (i.e. data processing). Such a system is modelled by Queueing Networks (QNs) [Menasce94] [Lazowska84]. QNs are widely used in modelling computer systems for performance analysis and capacity planning purposes.

A QN is a collection of service centers. Each service center represents a resource and is associated with a queue and server(s). Requests are dispatched from the queues according to a specified discipline, such as FIFO or LCFS or polling. The queues are connected by directed links. Customers move from one service center to the next to wait for service and receive service. Once the service is completed, the customer proceeds to the next service center.

Figure 2-3 shows an example of a closed QN, where no arrivals occur and no customers can leave and customer population is constant.

Figure 2-3: Example of a closed queueing network

Figure 2-4 is an example of an open queueing network, where new customers enter and old ones (eventually) leave. In an open QN, the number of customers can vary from zero to infinity. The World Wide Web is an example of a system that can be modelled as an open QN.

Figure 2-4: Example of an open queueing network

Metrics of interest to software performance analysts that can be obtained by solving the steady state solutions of QNs are usually the throughput, mean service time, utilizations, residences times and average queue lengths.

Product Form QNs

A smaller subset of QNs classified as product form QNs are easier to solve. Product form networks established in [Baskett75] are also known as separable networks as the states of individual servers can be separated [Menasce94].

Product form networks are termed product form, due to the inherent property that states that the steady state probability of being in a given state to be a normalized product of the relative device utilizations. Hence, the steady state solution can be easily derived by applying the *local balance* property. *Local balance* states that at steady state, the flow into any state due to the arrival at a queue is equal to the flow out of that same state due to a departure from the queue. Using solution methods such as convolution ([Buzen71]), Mean Value Analysis ([Reiser80]) or REcursion by Chain ALgorithm ([Conway86] [Conway89]), product form networks can be solved to provide exact solutions.

However for a group of networks to be solved with a product form solution. The following assumptions are required for product form solutions [Baskett75]:

- Scheduling disciplines allowed: First In First Out (FIFO), Processor Sharing (PS), Last Come First Served Preemptive Resume (LCFS-PR) and Infinite Server (IS).
- Service centers with a FIFO discipline:
 - Service time must be exponential with the same average mean for all customers.
 - Service rate can only depend on the total number of customers at the server and not on the number of customers of any particular class.
- Service centers with a PS, LCFS-PR disciplines:
 - Service time can have a general distribution.
 - Service rate can depend on the queue length for the class only but not on the queue lengths for other classes may differ.

- Service centers with an IS discipline:
 - Service time can have a general distribution.
 - There is never any waiting for infinite servers and hence no queueing.
- Arrival process: Must be governed by Poisson distribution for open networks. Bulk arrivals are not allowed.

Little's Result

Little's result [Lazowska84] is a quite simple result and applies across a wide range of assumptions. Little's result states that the average number of customers in the system (N) (i.e. the queue length) is equal to the arrival rate of customers at the system (λ) times the average time each customer spends in the system (R).

$$N = \lambda R$$

At steady state, flow in equals flow out and the external arrival rate equals the external departure rate, which is the system throughput. The average time spent by the customer in the system is simply the mean service time of the system ($1/\mu$). With this interpretation, Little's result may be rewritten as:

$$\text{Server Utilization} = \text{Throughput} \times \text{Mean Service Time}$$

$$\text{Server Utilization} = \frac{\lambda}{\mu}$$

$$\text{Mean Response Time} = \frac{1}{\mu - \lambda}$$

Thus, knowing average service time and arrival rate, or queue length and throughput, the mean response time of a system can be calculated using Little's result.

Limitations of QNs in representing Software Systems

QNs are limited in their ability to represent some software execution behavior and be solved efficiently, fast and with exact analytical solutions [Franks95] [Rolia95] [Wooside95a]. QN represent a job as a customer that moves from one service center to the next.

In computer systems, though, jobs may have simultaneous resource possessions. In its simplest forms, a job is using memory while running on the processor. The memory is known as a *passive* resource. *Passive* resources are resources, which are required for processing yet they don't actively provide any service. Other passive resources that are commonly found in software systems are locks, messages that are passed between processes.

There are other features of software execution, such as request execution patterns that vary significantly over the life of the request or routing to servers, that depends on data or on the state of the system or forks and joins. Those features can be represented by QN, yet the performance model solution becomes significantly more complex and hence much harder to solve.

2.4.2 Polling Review

Polling is a technique for allowing the scheduler to check the status of independent queues at certain intervals and according to a certain order of polling.

Up to the present the available literature on providing exact performance solutions for software polling is still scarce. The most important performance measure in polling system is the mean waiting time in the queue. Since the most common application for polling is the token ring network architecture, where at least two polling queues exist, the existing literature addresses mainly this case.

[Takagi90] provides an excellent survey of the literature on queueing analysis and polling models. Relevant papers that propose approximate solution techniques for polling are [Bux83] and [Boxma86]. Bux and Truong approach polling for a single server queueing model by employing an approximation for the second moment based on heuristic extrapolation to any value of n queues from the exact result for two queues [Bux83]. The extrapolation is inexact for a system with one polling queue. However software systems may have only one process that dispatches its request via polling. Boxma and Meister address polling for cyclic service systems with switch-over times by providing approximations for the waiting times [Boxma86].

2.5 Layered Queueing Networks

Layered Queueing Networks (LQN) are extended QNs for modelling distributed software applications. LQN was first independently developed under the name of *Stochastic Rendezvous Networks* (SRVN or SRN) in [Woodside89] [Petriu91b] [Petriu94] [Woodside95a] [Woodside98] and under the name of *Method of Layers* (MOL) in [Rolia87] [Rolia92] [Rolia95]. Many features of the two approaches were joined under the name of LQN [Franks95] [Woodside95b]. LQN modelling has proven to be a convenient way to model distributed software since it provides direct mapping of soft-

ware components to LQN tasks, hence making the models easy to build, understand and modify. The main difference with respect to the traditional QN is the fact that LQN allows for nested services: high-level servers become clients to lower level servers. Therefore, the solution process will expand the service times at higher level servers due to the inclusion of queueing delays at lower level servers. LQN tasks represent logical resources such as processes, critical sections and locks. The LQN toolset provides both analytical and simulation tools for solving layered models [Franks95].

2.5.1 LQN Model Components

An LQN model is built using two basic building blocks: *tasks* and different types of *inter-task communication* requests such as synchronous, asynchronous or forwarding. Processors are modelled as *tasks*.

Task

A task is an entity that models a software process execution demand and executes some work if it's processor is available. A task may be either a *client task* or a *server task*. A *client task*, as it's name implies, is a task that sends requests to other tasks. A server is on the other hand a task that performs work on behalf of requests received from it's clients. A server may also be a client to lower level servers by making requests to lower level servers as part of fulfilling their own work to the higher level client. A client task that never receives requests to execute is known as a *reference task*. Each task may have different classes of workloads on the processor by representing it with several entries. Each entry provides a different service pattern and hence a different execution workload. However, all entries of one task share a common task queue. The task queue scheduling disciplines

supported by LQN, that controls the order in which requests are processed, is First In First Out (FIFO) and Head of Line (HOL).

A server may be a single server, a multiserver or an infinite server. A single server is modelled as a single task, which handles only one request at a time. Concurrency in LQN is modelled by multiservers and replicated servers. A multiserver, refers to a multitude of copies of a task, yet all copies share one common queue for incoming requests. A replicated server, however, is similar to the multiserver, except that each copy task has its own requests queue. An infinite server is modelled as an infinite number of tasks on an infinite number of processors that can handle an infinite number of requests. Network delays are commonly modelled as infinite servers.

Entry

An LQN server may offer a range of services, each one with its own service time and number of visits to other servers. Every such service is modelled as an *entry* of the task (entries are represented as parallelogram *slices*). It is assumed that all the requests for all the entries of a task are queued in a common task queue (not explicitly drawn in the LQN notation).

The execution of a server entry following the reception of a message by the entry is broken into two parts, the first part is the first *phase* which ends when the reply is sent back and the second part is the subsequent *phases* after the reply (see Figure 2-5).

The SRVN approach requires a distinction be made between the different execution phases and each phase is described by its own model parameters. Most simple models can be modelled by

making use of the first phase (rendez-vous) phase only. Second phases can be used to describe concurrent execution. When modelling pipelines, third phases are necessary; phase 1 accepts input and replies to sender with acknowledgment, phase2 executes the pipelined work; phase3 interacts with successor task only once and should be modelled as a deterministic phase with only one call.

Inter-task communication

An arc, in the LQN graph, represents a communication request from a node playing the role of *client* to a node playing the role of *server*. Such a request can be synchronous, asynchronous or forwarded. A synchronous request blocks the client until the server sends back the reply. An asynchronous request, is a request made by the client but the client continues its work in parallel with the server. A forwarded request is very similar to a synchronous request in the sense that the client remains blocked, but the difference is that when the server finishes the request, the responsibility of sending back the reply to the original client is transferred to another server while keeping the original client blocked until it receives the reply.

Figure 2-5: Server task cycle and entry phases [Woodside95b]

2.5.2 LQN Model Graphical Representation

LQN is represented by an acyclic graph (Figure 2-6) whose nodes are either software tasks or hardware devices. Software tasks are drawn as parallelograms whereas hardware devices are drawn as ellipses. Tasks that consist of more than one entry such as the DBMS task represent the different entries. Synchronous requests are represented by full-head arrows, asynchronous requests by half head arrows and forwarding requests by dashed link with a full head arrow (asynchronous and forwarding requests are not represented in the following figure).

Figure 2-6: Sample LQN model of a DBase application

The model presented in Figure 2-6 consists of three layers. The LQN clients; Client 1, Client 2 and Client 3 are at the first top most layer. Requests are made from a top layer to a lower level layer. Strict layering is not enforced. LQN clients can be grouped into more than one class, where each is characterized by a common workload.

Figure 2-6 shows an example of a DBMS system that models three distributed clients making requests to a common server application. Each Client executes on his own processor, whereas the middle server DBMS and the lower-level server Filesystem compete for the same processor, denoted

as Server Processor. Client1 and Client2 don't have any execution demands in phase1, whereas Client3 executes only in phase1. Hence Client1 and Client2 perform their workload after receiving back the replies to their request from the DBMS Task server. A request from any of the clients to a DBMS entry starts by queueing in the DBMS task queue. It is then picked according to the chosen scheduling discipline, and the DBMS entry starts executing on behalf of the request on the Server processor when it is available. The processor incurs an execution demand equivalent to the execution workload of the executing DBMS entry. During the phase1 time interval, the DBMS sends requests to the FileSystem task and remains blocked until the FileSystem replies back.

Note that Client1 has a thinking delay in phase2, which is one of the subtle ways of putting a delay between consecutive requests submitted by Client1. The different phases in the case of the DBMS reflect common database software execution scenario. It is commonly known that database update requests are not always committed and physically updated on-line, but rather a reply is sent to the requester and the updates are performed later. Hence in the case of an Update entry, as in the example of Figure 2-6, upon receiving a request from Client1 and gaining access to the Server Processor, the DBMS task executes for 3 units of time, makes one request for accessing the Filesystem, sends back the reply to Client1 and then executes its second phase for 10 units of time.

2.5.3 LQN Assumptions

LQN Task

Task behavior is cyclic and can be described by more than one phase. The first phase is known as *rendezvous* phase and the subsequent *post-rendezvous* phases are optional. Task phases can be modelled either: a) as stochastic phases with exponentially distributed execution times, random message routing: random order of message destinations and random numbers of messages sent or b) as deterministic phases with generally distributed execution times and deterministic number of messages sent in a random order. A task accepts the first request in the queue according to the chosen request dispatching discipline, regardless of which entry it is addressed to.

Inter-task Communication

Communication between tasks can be rendezvous (synchronous) or send-no-reply (asynchronous). Forwarding is a special case of rendez-vous requests.

Processors

Processors are modelled as pseudo-tasks with FIFO or round-robin preemptive queueing.

Open Arrivals

Poisson Open arrival streams are assumed.

2.5.4 LQN Parameters

The parameter values needed to model the system can be measured when the real system is executing by measuring the execution time of each task and the frequency of calls. Although other parameters can be obtained, the most basic parameters needed to build the LQN model are:

s_{iep} = mean execution time of task i , during phase p of entry e .

y_{edp} = mean or expected number of synchronous (rendez-vous) messages sent from entry e to entry d during phase p of entry e .

z_{edp} = mean or expected number of asynchronous (send-no-reply) messages sent from entry e to entry d during phase p of entry e .

2.5.5 Solving LQN Models

LQN models can be solved with any of the solving tools provided in the toolset [Franks95]. The same input file can be solved by both analytical tools such as *lqns* and simulation tools such as *ParaSRVN*.

The *lqns* analytical solver technique consists of decomposing the LQN model into submodels and then solving individual submodels with Mean Value Analysis. Outputs of some submodel are fed into other submodels. Iteration among the submodels until continuous waiting time results converge for all layers.

The *ParaSRVN* simulation solver mimics the behavior of the system by conducting a discrete event simulation, which is expected to be considerably slower than the analytical solver. However, the simulation is more powerful, in the sense that it allows for more detailed models of the system, whereas the modelling power of the analytical model is limited by mathematical assumptions. The LQN simulator will be further described in Chapter 3.

2.6 Measurement Tools

2.6.1 Quantify

Quantify is a software tool used to measure the execution of C and C++ code [Quantify]. Quantify provides accurate measurements for source code that is compiled under it, but is much less accurate for 3rd party code, such as operating system calls and library routines. Quantify uses information provided by the compiler as well as information collected at run-time. The compiler subdivides the code into basic blocks. Each basic block consists of a sequence of instructions that are always executed sequentially. Using the compiler output, Quantify calculates the execution time based on the number of instructions it executes, the number of machine cycles each instruction consumes and the machine's clock rate (in mips, i.e. Million of Instructions per Second). The run time information collected by Quantify relies on the number of times each basic block has executed. Multiplying that with the number of machine cycles for each block, the total execution time for the block is deduced.

Each function provides metrics for the time that it spent executing its own code as well as the time spent by functions that it called on its behalf.

Measurements can be done at different levels, at the level of basic blocks or the level of functions. It has two options for providing time measurements, *elapsed time* or *kernel time*. To record the elapsed time, Quantify makes use of *gettimeofday()* to measure the elapsed time. For third party calls, however, if a context switch occurs, it will not be detected by Quantify, inflating the time inaccurately of the 3rd party call. To record the user and kernel times, Quantify makes use of calls to *getrusage()*, which is based on sampling the state of the system at every tick of the clock. Hence, its accuracy depends on the clock resolution, and is considered more accurate for longer chunks of codes and less accurate for shorter ones.

Quantify's provides considerably inaccurate measurements under two circumstances. The most obvious in the case of calls to third party code that are made from within the *to-be-measured code*, for which no source code is available to be compiled under Quantify. The second less obvious, yet of no less importance, is the fact that Quantify does not capture caching effects and hence the effect of caching is not reflected in the measurements.

Chapter 3

Extensions to LQN simulator

Chapter 3 presents the extensions contributed by the thesis to Layered Queueing Network simulation solver (ParaSRVN). For this purpose, it provides a review for ParaSRVN and PARASOL, which is the simulation engine on top of which ParaSRVN is built.

3.1 Solving Performance Models by Simulation

The LQN toolset provides both analytical and simulation tools. LQN performance models can be studied with either tool depending on the level of abstraction of the built model. Analytic solution provides a fast solution. On the other hand, simulation performance models are considered to be an important supplement to mathematical models because we can represent more details.

Simulation solvers accept models that are very detailed about the system and do not rely on solving complex mathematical equations, but rather on emulating the operation of the real system over time. The longer the simulation runs, the smaller the confidence interval and the more accurate the results and performance predictions are. The major drawback of the simulation is that it is computationally intensive and is considered slow, especially when compared to analytical solvers. The LQN

simulation solver, known as ParaSRVN has been specifically designed for LQN models. In fact, all LQN tools, accept the same model input file and present their results in the same output format. This is particularly useful, in the case where one may want to run the same model with several solvers. Since the ParaSRVN simulator is implemented on top of PARASOL [PARASOL1] [PARASOL2], we will describe briefly the PARASOL simulation environment before proceeding into the extensions that this thesis has contributed to ParaSRVN.

3.2 PARASOL Simulation System

PARASOL is a discrete event execution-based simulation tool, developed by Prof. J.E. Neilson at the Computer Science Department of Carleton University. PARASOL is designed to simulate distributed and /or parallel computer system models. PARASOL is event-execution-based and does not employ time warp technology or speculative computation to speed up the execution. The PARASOL kernel is written in ANSI C and is therefore highly portable and extendable. Moreover, the PARASOL code and functions can be easily incorporated and invoked from any C program. From a developer's point of view, PARASOL provides the basic primitives for a simulation engine: hardware configuration, task management, inter-task communication, task synchronization and statistics capturing. PARASOL also comes in a C++ flavor, known as PARASOL++. To build a correct simulation is a complex task. PARASOL also provides simple debugging facilities.

The most important active component of the PARASOL simulation environment is the driver, which acts as a scheduler [PARASOL2]. PARASOL provides a complete set of calls to support the

task scheduling process [PARASOL1]. Hardware can be modelled with processing nodes, buses or links. PARASOL tasks can model any software process or hardware resource. A PARASOL task is associated to a processing node. PARASOL extensive set of functions can be used to manage tasks as well inter-task communication, which is port-based. Messages that travel between tasks may be as small as timestamps or any arbitrary amount of user text.

3.3 ParaSRVN Solver

ParaSRVN is built on top of the PARASOL engine to simulate LQN models. The input to ParaSRVN is an LQN model file. It produces several types of output, such as normal or parseable output that may be used with other programs such as SPEX, MultiSRVN or TimeBench. As with simulation solvers, parameters such as total simulation length, number of blocks or maximum confidence interval can be used to control the duration of the simulation.

At start-up, ParaSRVN creates for each LQN task, the corresponding PARASOL task and inserts its first schedule of execution within the PARASOL task-scheduling environment. Along with each PARASOL task, ParaSRVN identifies the associated service-routine for the PARASOL scheduler to execute when this particular task gets invoked. Table 3-1 outlines the LQN task and the corresponding ParaSRVN service routine. All of the listed service routines check for incoming requests

and once received they make calls to a common function called `server_cycle()`, which actually performs the specific entry execution by making appropriate calls to `execute_activity()`.

Table 3-1: LQN task and corresponding service routines

LQN	Corresponding service routine
Client/Reference task	<code>srn_client()</code>
Infinite/Delay server	<code>srn_infinite()</code>
Single server	<code>srn_server()</code>
Multiserver	<code>srn_queue()</code> , <code>srn_worker()</code>
Open Arrival Stream	<code>srn_open_arrivals()</code>

As shown from Table 3-1, an LQN task, known as multiserver has two associated service routines. An LQN multiserver represents n copies of the single server sharing one common request queue. ParaSRVN maps it to a task that performs its service by invoking `srn_queue()` on a separate PARASOL node and the multiple tasks perform their service by invoking `srn_worker()` all on the same PARASOL node altogether.

ParaSRVN, based on the PARASOL simulation system contains a particular function called `ps_genesis()`, which is a special reserved function name required by PARASOL similar to a `main()` function in any C program. PARASOL execution always starts with the `ps_genesis()` function. However, the code of `ps_genesis()` is part of ParaSRVN. Hence, the execution of ParaSRVN starts with `ps_genesis()`, which builds the task list, starts off the PARASOL task driver and calls the ParaSRVN `run_simulation()` function, shown in Figure 3-1, to actually control the simulation and gather statistics.

- *Sleeps for an initial transient delay for which no statistics are gathered*
- *Repeats the following according to specified termination criteria*
 - *Sleeps for duration of block time*
 - *Calls `accumulate_data()` for each task to accumulate statistics*
 - *Calls `accumulate_stat()` for each entry of each task*
 - *Calls `accumulate_stat()` for each phase of each entry of each task*
- *Calls `accumulate_stat()` for each processor*

Figure 3-1: Pseudo-code for `run_simulation()` pararvn function

3.4 Extensions to ParaSRVN

This thesis has added three extensions to ParaSRVN that have been deemed necessary in order to conduct our performance study of the industrial system present in the next chapter. The three extensions are the addition of polling scheduling to ParaSRVN servers, gathering queue length statistics and preventing ParaSRVN from crashing when the system runs out of memory resources. The implementation of the three extensions is described by pseudo-code in the following subsections. New statistics that are gathered are also tabulated in SPEX format (see Chapter 5). Throughout this thesis, we have used SPEX, an experiment controller for the LQN toolset to run and extract specified results.

The thesis has modified the SPEX code to extract the new statistics. The syntax that must be specified in the LQN input file in order to extract those results is presented in a tabular form in each subsection.

3.4.1 Polling Servers

Generally, a process can pick up an input request either by performing a blocked receive on its incoming request queue or by polling the queue. The current implementation of ParaSRVN supports only the first scheduling alternative, where an idle task is blocked on its incoming request queue, waiting for a request to come in. The arrival of the request *awakes* the task. On the other hand, a polling task would perform a poll check on its incoming request queue and if it does not find a request to pick up, it will put itself to sleep for a given period of time after which it starts a new polling cycle. The polling sleep interval may be either set at the application or operating system level depending on the environment. If it is set at the application level then it may provide a degree of flexibility for performance tuning.

Extending ParaSRVN to allow for polling tasks was a requirement that arose from the industrial experience reported in Chapter 4. The extension has added two new types of servers, namely polling server and polling multiserver types. Accordingly, instantiated tasks of these two new types will have the new associated service routines as outlined in Table 3-2.

Table 3-2: New LQN task and corresponding service routines

LQN	Corresponding service routine
Polling Single server	sm_polling_server()
Polling Multiserver	sm_polling_queue(), sm_polling_worker()

Furthermore, with the polling extension the statistics of Table 3-3 are provided in the ParaSRVN output. Table 3-3 provides the syntax to extract them when using SPEX.

Table 3-3: New polling statistics specifications in the LQN input file

Results specification	Results returned
%pst	Rate of polling sleeps
%pstc[95],%pstc[99]	Confidence intervals
%pout	Utilization of task due to polling
%poutc[95],%poutc[99]	Confidence intervals
%tut	Total task utilization, including useful and polling work
%tutc[95],%tutc[99]	Confidence intervals
%tup	Total processor utilization including useful and polling work
%tutc[95],%tutc[99]	Confidence intervals
%poup	Utilization of processor due to polling
%poupc[95],%poupc[99]	Confidence intervals

The following three figures provide a pseudo-code for each of the new polling service routines.

```

-Repeat forever
    -Calls poll_for_message()
    -Calls poll_compute()           //consume CPU time for polling
    -If a message was found then it calls server_cycle()
    -Else calls poll_sleep()

```

Figure 3-2: Pseudo-code for srn_polling_server() ParaSRVN function

```

-Repeat forever

  -Sends a signal to srn_polling_queue task

  -Calls poll_compute()           //consume CPU time for polling

  -Checks if a request has been received from srn_polling_queue task

    -If a message was found then it calls server_cycle()

    -Else calls poll_sleep()

```

Figure 3-3: Pseudo-code for *srn_polling_worker()* ParaSRVN function

```

-Repeat forever

  -Blocks for a signal from any of the srn_worker tasks

  -Calls poll_for_message()

  -Checks if a request has been received from srn_polling_queue task

    -If a message was found then it sends request to srn_worker()

    -Else it sends a NULL signal to srn_worker()

```

Figure 3-4: Pseudo-code for *srn_polling_queue()* ParaSRVN function

3.4.2 Queue Length Statistics

The current implementation of ParaSRVN provides the mean delays for a request from one entry to another. It does not provide average queue length statistics. Although the queue length may

be deduced by calculations, it is recommended to gather it during the simulation for more accurate results.

Table 3-4: New statistics specifications in the LQN input file

Results specification	Results returned
%q	Average number of requests in queue
%q[95],%q[99]	Confidence intervals

Queue length statistics are monitored and captured on an entry basis. However, if the %q is specified at the task information in the LQN file, the average of number of requests for the task is returned. Figure 3-5 provides the changes that were made to ParaSRVN to provide queue length statistics.

```

-in send_synchronous(), send_asynchronous(), do_forwarding():

    -increment queue length statistics

in wait_for_message(), poll_for_message()

    -decrement queue length statistics

```

Figure 3-5: Pseudo-code for manipulating queue length statistics

3.4.3 Enhancements for Solving Open Arrival Systems

Before the enhancement, ParaSRVN currently did not handle gracefully open arrivals at levels close to saturation. If during the simulation, an asynchronous message failed to obtain memory in the

queue, then ParaSRVN used to dump the messages and exit with error status. This was very inconvenient, as the crash is unpredictable and may come after a considerable amount of time was spent running. By crashing the program ended without capturing the latest block statistics.

We have solved this problem by enforcing a graceful reduction in open arrival rates. To achieve this, we have defined two thresholds as outlined in the pseudo-code of Figure 3-6 and Figure 3-7. When a pre-defined hard limit is reached ParaSRVN makes a call to accumulate statistics and exits thereafter. However, before the hard limit is reached, a soft limit is specified which will cause ParaSRVN to start rejecting open arrivals. The actual arrival rate is collected and reported accordingly. The `asynch_counter` keeps track of total number of asynchronous messages present in the system, whereas a positive `throw_counter` indicates that the open arrival rate is being reduced. The `throw_counter` is decremented as requests are being freed and hence messages are being deallocated. Once the `throw_counter` reaches zero again, the system stops rejecting open arrivals.

Table 3-5 provides the specifications for gathering the actual and rejected rate of open arrivals from the parseable ParaSRVN output.

Table 3-5: New statistics specifications in the LQN input file

Results specification	Results returned
<code>%wl</code>	Actual open arrival rate
<code>%wl[95],%wl[99]</code>	Confidence intervals
<code>%wr</code>	Rejected open arrival requests
<code>%wr[95],%wr[99]</code>	Confidence intervals

When generating a new open arrival: in opensrc alloc msg():

- Allocate a new message structure
- Increment asynch_counter for allocated asynchronous messages
- If asynch_counter > Soft_Max_Messages
 - Increment throw_counter
 - If asynch_counter > ALERT_Max_Messages
 - Terminate simulation gracefully by calling awaken_genesis()

After receiving any asynchronous message: in free msg():

- Decrement asynch_counter for allocated asynchronous messages
- Decrement throw_counter if throw_counter > 0

Figure 3-6: Pseudo-code for gracefully reducing asynchronous messages in system

In execute activity()

- Checks if current task is an open arrival source
- If open arrival source and throw_counter > 0
 - Cancel current open arrival request by putting the open arrival task to sleep
 - Record this rejection to compute actual arrival rate.

Figure 3-7: Pseudo-code for gracefully reducing asynchronous messages in system

Chapter 4

Concrete Industrial Experience

This chapter demonstrates the application of Layered Queueing Network Modelling to a real-time telecommunication system. Due to constraints imposed by the proprietary nature of the system, we have not presented here details about the system. The architecture of the system is described in a generic way, and the names (process identifiers) were changed.

4.1 Software Architecture of the System

The starting point for building the LQN performance model is identifying the architecture of the system under study. There are several types of architectural patterns that can be observed in telecommunications system, three of which are discussed here. The notation used to represent software architectures is similar to the notation in [Shaw96]: Software processes are drawn as ellipses, procedures as rectangles and shared memory as double-nested rectangles. Figure 4-1 and Figure 4-2 presents two alternatives of software architecture for telecommunications call-processing systems. The basic components in the two architectures are the stack process (which receives the request messages), the request queue (where messages are saved until processing) and the EventHandlers.

EventHandlers may be multiple process replications (as in Figure 4-1) or multiple threads running in one process (as in Figure 4-2).

Figure 4-1: Multiprocessing architecture

Figure 4-2: Data multithreading architecture

The EventHandler processes in Figure 4-1 are polling the shared request queue for messages to process. In Figure 4-2, an EventDemultiplexer running in a separate thread reads the input queue and distributes the requests to the EventHandler threads when they become free. The inter-communication between EventDemultiplexer and an EventHandler is asynchronous and bi-directional: the first dispatches a request to the second, which processes it and then announces to the EventDemultiplexer that it's free to accept a new request.

The third architecture, shown in Figure 4-3 and Figure 4-4 is more complex. It corresponds to the industrial system studied in this thesis, and is discussed in detail in Section 4.1.1. It consists of an extension of Figure 4-1, where a database process supports the EventHandlers.

As can be observed from the three architectures, the EventHandler is the main event handling process in the system. Having identified the software architecture of the system, the next step is to build a dynamic view of the system by identifying the most frequent scenarios that impact the performance of the system.

Connie Smith [Smith90] has identified seven performance principles that provide guidance for creating responsive software systems and for detection/removal of performance degradation causes. Two of Smith's principles have been applied here; namely the *Processing versus Frequency* and the *Centering* principles. The first principle addresses the amount of work done per processing request and its impact on the number of requests made by advocating to minimize the processing times frequency product [Smith90]. The *Centering* principle is based on the "80-20" rule for the execution of code, which claims that ≤ 20 percent of a program's code accounts for ≥ 80 percent of its computer

resource usage. The frequently requested functions in the programs are referred to as the dominant workload functions. The *Centering Principle* states: “Identify the dominant workload functions and minimize their processing” [Smith90]. Combining the *Centering* and *Processing versus Frequency Tradeoff* principles, we have next sought to identify the most frequently used real-time scenario.

Execution paths can be easily described by use-case maps [Buhr96]. Use-case maps is a notation for visually representing scenarios of execution, that are simple and easy to use at high levels of abstraction. At lower levels of abstraction, i.e. higher details they tend to lose their attractiveness. By consulting statistics/metrics of the usage of the different types of functionalities/services, the most frequently used path can be obtained (shown in Figure 4-3).

Figure 4-3: Scenario for an incoming request

In the case of our studied system, the most frequently used path is the request processing that comes in as a request from the network. All three performance studies of the system were conducted with a confidence interval of $\pm 1\%$ at 95% level.

4.1.1 Description of the Critical Path Scenario

The system of our industrial experience is a real-time telecommunication system, whose critical path scenario exhibits a general pipeline architectural pattern (Figure 4-4). Figure 4-3 shows the different processing that the incoming request incurs while being served by the system. Using the software architecture notation. Figure 4-4 depicts the abstract scenario of an incoming request that is passed through several stages of the pipeline: starting from the StackProcess to the IOEngine to the EventHandler and all the way back. The interprocess communication in the pipeline between the IOEngine and EventHandler is implemented through shared memory. The multiple replications of the EventHandler compete for the unique database and common shared memories. ShMem1 is used as the buffer for incoming requests and outgoing replies and ShMem2 contains request control information. Request control information is in shared storage due to the fact that for certain types of requests a request may start being serviced by a EventHandler and may be continued by another EventHandler. ShMem3 is shared storage for inter-communication between the multiple EventHandler replications and the database.

An incoming request that comes into the system through an external network is received by the Stack process. It is then passed along to the IOEngine process, which does some preliminary processing and puts the request in the shared queue, which we will refer to as ShMem1, that is protected

by a semaphore. The EventHandler picks the request from ShMem1, after acquiring access to the semaphore and starts processing. The main processing is done by the EventHandler, which accesses a real-time database to fetch an execution script for the desired request, then executes the steps of the script accordingly. The script may vary in size and types of operations involved, and hence the execution workload may vary largely from one type of request to another (by one or two orders of magnitude). Accordingly, a performance analysis conducted for a given type of request will not be a good prediction for the performance of the system for different types of requests.

The next step is to map the software/hardware architecture of the system into an LQN model. Note that throughout this thesis, we have focused on analyzing the system for a request that we will call as “type X”.

Figure 4-4: Software architecture of the critical path

4.2 Extracting and Building the Base LQN Model

The real-time execution path that is to be modelled starts from the moment a request arrives to the system and ends after the request was completely processed and a reply was sent back.

The architectural patterns [Shaw96] for the critical path scenario used in the system (i.e., the pipeline pattern involving StackProcess, IOEngine and EventHandler, and the client/server pattern involving EventHandler and Database) provide the basis to build our LQN model, by mapping the system architecture into the corresponding LQN domain.

4.2.1 Mapping System Components to LQN

This section shows the transformations we have applied to the architectural model from Figure 4-4 to build the LQN model shown in Figure 4-10.

LQN's strength and attraction to software performance analysts lies in the fact that it allows the modeller to perform a direct mapping of software processes to LQN components. The software processes shown in the architectural view of the system, as depicted in Figure 4-4, are mapped directly to LQN software components, known as tasks. For example, we begin by representing every process from Figure 4-4 by an LQN task. EventHandler becomes a multi-server, whereas the others are single servers. The message-based communication between the StackProcess and the IOEngine pipeline stages is asynchronous, while that between EventHandler and Database is synchronous.

Software critical sections that are controlled by semaphores, locks or mutexes are modelled as “*pseudotasks*” in LQN, as shown in Figure 4-5, in order to represent their serialization constraints.

Figure 4-5: Mapping critical sections in LQN

The shared-buffer communication between the two pipeline stages **IOEngine** and **EventHandler** is modelled by breaking it into two separate features: a semaphore pseudo-task to model the contention for the shared buffer, and an asynchronous request from **IOEngine** to **EventHandler** as shown in Figure 4-6.

Figure 4-6: Mapping communication between stages of the pipeline

LQN has an implemented protection against cyclic graphs for deadlock prevention purposes, as illustrated by Figure 4-7.

Figure 4-7: Illegal cyclic LQN model

In order to get around that, we split IOEngine into two tasks and add an “*executive*” task to control the thread of execution. IOEXEC insures that the two tasks IOEngineIN and IOEngineOUT cannot execute simultaneously, as modelled in Figure 4-8.

Figure 4-8: Valid cyclic LQN model

Any LQN open model can be turned into a closed model by replacing the open arrival with a fixed number of “*pure*” client tasks. We use this transformation by adding a very large number of clients in order to saturate the system without overflowing the task queues.

Figure 4-9: Transforming an open to a closed LQN model

In the case of open models with asynchronous messaging, we also must substitute the asynchronous requests with *forwarding* requests in the closed version, such that the clients are blocked until the last forwarded request is completed. In the closed version shown in Figure 4-9, a Client sends a synchronous request to IOEngine, which processes the request then forwards it to the next task in the pipeline, namely the EventHandler. IOEngine is ready to continue its activity after forwarding the request but the Client remains blocked until the forwarded request is completed by EventHandler. Note that more than one forwarding is allowed in LQN, in which case the Client waits until the last forwarded request is completed.

4.2.2 LQN Model of the System

By mapping each process of the architectural diagram of the system to an LQN task and performing the special mappings for the shared memories we finally obtain the LQN model for the system as shown in Figure 4-10. Each LQN task is further described by associated entries and each entry has defined request and execution parameters as explained in Chapter 2.

The structure of the LQN performance model was relatively easy to derive from the architectural. Figure 4-10 illustrates the base model for the system. The parallelograms represent the processes, whereas the ellipse depict the hardware resources. The tasks that provide several classes of requests are further detailed by entries.

Figure 4-10: LQN base model of the system

A client submits a request to the system by sending a synchronous request to StackIN. StackIN makes a synchronous call to StackEXEC and is blocked until StackEXEC terminates execution. StackEXEC stands for the execution spent by the request on behalf of the stack. StackEXEC provides a mechanism for synchronizing requests from StackIN and StackOUT, such that they model the behavior of one process and not two.

An incoming request spends some time at the Stack, while the network headers are being removed and the request is being put on the stack. An outgoing reply would incur almost the same delay mainly for removing it from the Stack and preparing it for the network by adding the necessary network headers. StackIN and StackOUT are what we have earlier referred to as *"pseudotasks"* and StackEXEC as *"executive"* task. This is due to the fact that the real system has only one process to handle both incoming requests and outgoing replies.

For similar reasons, IOEngineIN, IOEngineOUT and IOEXEC were added to the model. However, in the case of IOEXEC, the task is further described by two entries, as that process provides two classes of services. Incoming requests are served differently (execution workloadwise) than outgoing replies, and hence each entry is defined by the execution demand for the corresponding class. After StackEXEC replies to StackIN, the second forwards the request to IOEngineIN. By forwarding the request, a reply cascading is in effect. More exactly StackIN does not send the reply back to the Client but rather the reply is forwarded to IOEngineIN, after which StackIN becomes idle. IOEngineIN is responsible for acquiring access to ShMem1 and storing the request info in the shared queue.

The shared memory area ShMem1 has 3 associated classes of workload: write, reserve and read class. Note that since they are part of the same task, the mutual exclusion constraint is satisfied, meaning that a single entry at a time can be executing. This is equivalent to the real system where that area of shared memory can be either accessed by IOEngine or EventHandler under mutex protection. The request is then forwarded further to the EventHandler.

The EventHandler is modelled as a multi-server, with multiple replications but only one queue. The EventHandler makes a call to the Read entry of ShMem1, to model the EventHandler picking up the request info from the shared queue. The EventHandler then makes a call to the “alloc” entry of ShMem2, to model the acquisition of a request control block. It then makes a call to the Database to retrieve the script from the Database. After retrieving the script, it makes several visits (i.e. calls) to the ShMem3 task to model the accesses to the shared memory in the system, that is initialized at system start-up. Finally after the EventHandler has provided service to our request, it will forward the request to IOEngineOUT that will call IOEXECOUT, which in turn will make a call to the “read” entry of ShMem1, to model the reading of an outgoing reply. IOEngineOUT will then forward the reply to StackOUT, which will model the packing for the network by making a synchronized call to StackEXEC. Having reached the last stage of the pipeline and of the cascaded forwarded requests, upon termination of execution of the StackEXEC, StackOUT sends back the reply to the Client.

4.2.3 Measurements and Validation

More difficult than building the LQN model was obtaining the numerical values for the parameters of the model from Quantify measurements and statistics provided by the UNIX command ‘top’.

Quantify is a software tool used to measure the execution of C and C++ code [Quantify]. Quantify provides accurate measurements for source code that is compiled under it, but is much less accurate for third party code, such as operating system calls and library routines. Quantify does not capture caching effect on the measurements (see Section 2.6.1).

top is a UNIX command, that displays and updates information about the top processes in the system. It reports raw CPU percentage used by the different processes. It is also based on the concept of sampling done at the operating system level. The interval of sampling is system-fixed and cannot be altered.

Measurements were run in a lab on two different hardware configurations, for one and four processors. To obtain the workload parameters, we have made use of the measurements using Quantify [Quantify] and the Unix utility “top”. The measurements for Quantify were run at very low loads of around a couple of requests/second, whereas “top” provided us with utilization figures for very high loads of hundreds of requests/second, close to the actual operating point. The service time values used for the model are presented in Table 4.1. We have used the measurements from Quantify to determine the execution model parameters, and the utilization results from “top” to validate our system. The utilizations measured and obtained from the model were very close. A more rigorous validation was hindered by the lack of response time measurements.

Table 4-1: Measurements from Quantify and top^a

LQNTask/Entry	Value
-Reading a request/reply from shared queue -executed twice per request	100 us
ShMem1: Reserv -executed twice per request	120 us
ShMem1: Write - executed twice per request	105 us
ShMem2: Alloc	870.1 us
ShMem2: Free	132.8 us
Dbase	674.88 us
ShMem3 - executed four times per request	157.6 us
StackEXEC - executed twice per request	805.5 us
Non critical logic code	0.1ms
TOTAL SERVICE TIME of the EventHandler on behalf of the request	3.9 ms

a. All tasks that include semaphore operations, have the semop overheads included in the execution times.

4.3 Study 1: Bottleneck Shift Detection on Proposed System

The challenging question posed by the developers was to find the “best” hardware and software configuration that can achieve the desirable throughput for a given mix of services. By “configuration” we understand more specifically the number of processors in the multiprocessor system and the number of EventHandler software replications. In this study, we tried to answer this question by

exploring a range of configurations for a given service mix, determining for each the highest achievable throughput. Then the configurations whose maximum throughput are lower than the required values are discarded. The cheapest configuration that can insure satisfactory throughput and response time at an operating point below saturation will be chosen. Running the LQN model is a more efficient way to explore different configurations under a wide range of workloads than to run and measure the real system for all the cases under consideration.

4.3.1 Performance Analysis of Base Model

Since maximum achievable lossless request rate was the major concern, we have proceeded by closing the model and changing all asynchronous requests to forwarding as explained earlier. The LQN model (shown in Figure 4-11) was built as described in Section 4-1. Model parameters and measurements were gathered as described in Sections 4-1 and 4-2.

Closing the model we have run it to saturation in order to get the maximum throughput. We have chosen to run our experiments on three different hardware configurations: one, four and six processors. We chose one and four processors since the actual system had been run for such configurations, and six processors to see how the software architecture scales up.

A software server in LQN is defined to be utilized during its service and all the services requested from lower servers, as well as any queueing delays while waiting for service from the lower servers.

Figure 4-11: Study1: LQN base model of the system

We have observed, that in the case of running the system on a single processor, the replication of the EventHandler does not increase the maximum achievable throughput as shown in Figure 4-12. This is due to the fact that the processor is the system bottleneck. As known in [Franks96], the replication of software processes brings performance rewards only if there is unused processing capacity (which is not the case here).

Figure 4-12: Utilizations for 1 processor

Interestingly, in the case of four processors, we notice that with more processing capacity in the system IOEXEC and StackEXEC reach critical levels of utilization, affected by serialization constraints in the software architecture which start to surface at higher processing capacity (Figure 4-13). The serialization is primarily due to the fact that the input and output halves of Stack or IOEngine processes do not run concurrently, being executed by a single execution thread. Another serialization is caused by the interprocess communication using ShMem1, for which all IOEngineIN, EventHandler and IOEngineOUT contend for.

Figure 4-13: Utilizations for 4 processors

Furthermore, the increased processing capacity raised the maximum achievable throughput, and consequently StackEXEC and IOEXEC start showing a potential for limiting the system performance at high execution rates. This trend is even more visible the case of a six processor configuration (shown in Figure 4-14). The StackEXEC and IOEXEC effectively saturate faster than the processor, hindering the processor from reaching the maximum utilization of one. This has definitely shifted our bottleneck from hardware to software, where the limitation in performance are due to constraints in the software architecture.

Figure 4-14: Utilizations for 6 processors

We have run further experiments, eliminating the serialization constraints at both the Stack and the IOEngine processes, by removing both StackEXEC and IOEXEC LQN tasks. Yet, no major performance improvement was observed. By examining the percentage of time IOEXEC is actually executing on the processor and its utilization (which includes waiting for lower level servers) we have found that about 90% of IOEngine utilization is spent waiting for the buffer ShMem1 and only the rest of the utilization is due to useful work.

4.3.2 Performance Analysis for Modified Model

In an attempt to reduce the contention for ShMem1, we applied modifications on the deserialized LQN model by splitting ShMem1 into two shared buffers, such that IOEngineIN and

EventHandler contend for one buffer for incoming requests, and IOEngineOUT and EventHandler contend for the other buffer for outgoing replies. The model shown in Figure 4-15 corresponds to the modified software architecture.

Figure 4-15: LQN model of the modified system

Having applied these changes to the model, we solved the resulting LQN model and obtained the utilization results for 4-processor (Figure 4-16) and 6-processor (Figure 4-17) configurations respectively. The maximum achievable throughput has increased by only 2.7% in the case of the 4-processor as opposed to an increase of 10.3% in the case of the 6-processor configuration.

As we can see from Figure 4-17, a new software bottleneck has emerged in the 6-processor case, namely the Database process (which is 100% saturated). The new bottleneck, being caused by a low-level server, has propagated upwards, leading to the saturation of all software processes using it (all EventHandler replications).

Figure 4-16: New utilizations for 4 processors

Figure 4-17: New Utilizations for 6 processors

4.3.3 Performance Analysis Recommendations

Consequently, we have recommended to the development team to use two semaphores instead of one for the input and output message queues in the shared buffer ShMem1, and to split Stack and IOEngine processes into two concurrent processes each. Another important conclusion of our performance analysis is the fact that the new software architecture doesn't scale up for hardware configurations with 6 processors or more.

The final conclusion of this performance analysis study is that *different configurations will have different bottlenecks*, and by solving the bottleneck in one configuration, we shift the problem somewhere else. What performance modelling has to offer is the ability to explore a range of configu-

rations and workload mixes, and to detect causes for performance limitations just by solving the model, before proceeding to changes in the real system.

4.4 Study 2: Performance Analysis of Base Model and Alternatives

Although software designers have access to measurements, they were seeking answers to questions and concerns that measurements alone cannot give, mainly because it is very expensive to conduct measurements for a large range of configurations and workloads.

In our second study of performance modelling and analysis of the system, we have modelled the system as an open model. There was a constraint imposed by the user for the system to run in a “safe” real-time zone, the processor utilization on behalf of the real-time requests should not exceed 80%. The remaining capacity was reserved for non-real time functionality that was not modelled.

We have started from the open base model for a fixed configuration of four EventHandler replications and four processors on the system throughout this second study.

Through some reverse-engineering, we have modelled several variants of the system in order to see the degree of performance improvement these changes would bring to the overall system performance. Some of the changes we have modelled onto the base system are guided by Smith’s principles [Smith90]. We have modelled mainly the aggregation of some accesses to critical sections and moving out some critical section code.

We have observed response times, utilization and other parameters of interest. We have obtained our results on the base model, shown in Figure 4-10.

4.4.1 Studied Cases

The four cases studied in this section have been considered as a result of the discussions with the user, who wanted to verify the performance impact of optimizing critical section code.

Base Case is the base open model for the system (Figure 4-10). The execution parameters have been defined in Table 4.1.

The observation of queueing delays in the *Base Case* led to the study of *Case 1*, which aggregates four shared memory accesses into one and hence eliminates three semaphore acquire/release overheads as well as the associated queueing delays.

By reverse-engineering the critical execution path of the request it has been observed that IOEngine and the EventHandler perform a Reserv before writing to the shared queue. The Write is immediately performed after the Reserv. However, the semaphore is being acquired at both the Reserv and the Write. *Case 2* eliminates that unnecessary semaphore operation to aggregate the Reserv and Write into one semaphore access.

Case 3 moves code outside of the critical section by reducing the execution workload of Alloc to be equivalent to the Free that control ShMem2. By such the holding time of the semaphore is reduced.

Case 4 is obtained by applying all the model changes of the cases 1, 2 and 3 at the same time.

Table 4-2 summarizes the four different cases in the first column. The second column states the reduction in execution demand relative to the execution demand of the base case.

Table 4-2: Cases studied

Studied Cases	Difference in execution time
Base Case	
Case1: Aggregating the 4 ShMem3 accesses into 1 access -> eliminates 3 semaphore acquires/releases and queueing delays	-0.3 ms
Case2: Aggregating the ShMem1: Reserv and Write -> eliminates 2 semaphore acquires/releases and queueing delays	-0.2 ms
Case3: Reducing the ShMem2:Alloc to the same execution measurement as the ShMem2:Free. The difference is moved as non-critical code execution of the EventHandler	0
Case4: Case 1 changes +Case 2 changes +Case 3 changes -> eliminating 5 semaphore acquires/releases and queueing delays -> reducing ShMem2 critical section by making part of the code non-critical	- 0.5 ms

4.4.2 Results and Analysis

Observation 1: Total Service time, i.e. Processing time on behalf of a real-time request

The first observation is that the Base Case has the highest service time, which implies that the proposed changes have a positive effect by reducing the service time.

It is worth pointing out that in the Case 3 we notice a reduction in the service time even though the same total service time for the request is maintained and the change is in moving some code out of the critical section. In Case 3, where the Alloc was initially 0.87 ms and then we reduced it to 0.13ms and moved out of the critical section 0.74 ms, it resulted in a reduction of 0.88 ms in the service time, i.e. a reduction of 14%.

In the other cases, where we have tried to spare semaphore overheads, like waiting on the semaphore and executing the acquire and release statements, we did not observe a big impact on the performance. Nevertheless, it has a positive effect, but because the acquire/release overheads are of the order of 3% of the service time of EH on behalf of the request it does not impact by much.

Figure 4-18: EventHandler Service time on behalf of the request

Service time is a common performance criteria to observe, and the fact that the service time has decreased, this entails the response to a request to be faster and hence our system can support more requests than before.

From Figure 4-18 and the data presented in Table 4-3, a 25% reduction of service time is observed in Case 4. Table results further show that the largest reduction is attributed to the effect of turning critical code to non-critical.

Table 4-3: Service Time of EventHandler on behalf of the request

Case	CPU Processing Time	Difference from Base	Response Time	Difference from Base
Base Case	3.9 ms		6.23 ms	
Case1: Aggregating the 4 ShMem3 accesses into 1 access -> eliminates 3 semaphore acquires/releases and queueing delays	3.6 ms	-0.3 ms	5.72 ms	-0.51 ms
Case2: Aggregating the ShMem1: Reserv and Write -> eliminates 2 semaphore acquires/releases and queueing delays	3.7 ms	- 0.2 ms	6 ms	-0.23 ms
Case3: Reducing the ShMem2:Alloc to the same execution measurement as the ShMem2:Free. The difference is moved as non-critical code execution of the EventHandler	3.9 ms	0	5.35 ms	-0.88 ms

Table 4-3: Service Time of EventHandler on behalf of the request

Case	CPU Processing Time	Difference from Base	Response Time	Difference from Base
Case4: Case 1 changes +Case 2 changes +Case 3 changes -> eliminating 5 semaphore acquires/releases and queueing delays -> reducing ShMem2 critical section by making part of the code non-critical	3.4 ms	-0.5 ms	4.54 ms	-1.69 ms

Table 4-4 presents how long it would take to service those 50 or 100 more requests/second.

Table 4-4: Service Time of EventHandler on behalf of the request for cases Case2 and Case4 at the higher request rates

Case	Request rate/second	Response Time	Difference from Response Time at 600 requests/second	Increase in number of requests/second
Case2: Aggregating the ShMem1: Reserv and Write -> eliminates 2 semaphore acquires/releases and queueing delays	650	6.16 ms	0.16 ms	+ 50
Case4: Case 1 changes +Case 2 changes +Case 3 changes -> eliminating 5 semaphore acquires/releases and queueing delays -> reducing ShMem2 critical section by making part of the code non-critical	700	4.76ms	0.22 ms	+ 100

Observation 2: Total Response time of a request

Figure 4-19 illustrates the total response time per request, which includes all times since the task gets in at the StackProcess and until the last processing by StackOUT is finished and the reply is sent back.

Figure 4-19: Total Response time of a request

Observation 3: Waiting time in the Request Queue before a request starts service

Figure 4-20 shows the queueing delay at the maximum open arrival loads on the system without exceeding 80% of processor utilization. We can observe the queueing delay incurred by a request before it is picked up for service by the EventHandler is in all four cases very close in the average ~0.16ms.

Notice that the curve of Case 2 (aggregation of Reserve and Write) drives the system to a higher load (i.e. number of requests per second) yet in Case 4 (all changes applied) the system goes up to 700 requests per second (0.7 requests/ms).

Figure 4-20: Queueing delay prior to EH service for the four cases

Observation 4: EventHandler Processor Utilization

One of the concerns in the study was to observe that the real-time request processing on the processor does not exceed the 80% limit of maximum processor utilization imposed for the system to run in a safe real-time environment. Notice that at 600 requests/second in the base case we are only 4% less than the targeted 80%. At 600 requests for Case4, the processor utilization is at 68% only. However for Case 4, the system is supporting 700 requests/second without exceeding the 80% limit.

Figure 4-21: EH Processor Utilization for the four cases

Table 4-5: Performance Results for Processor Utilization

Case	Request Rate/ second	Processor Utilization	Request Rate/ second	Processor Utilization
Base Case	600	75.7%		
Case1: Aggregating the 4 ShMem3 accesses into 1 access -> eliminates 3 semaphore acquires/releases and queueing delays	600	71.1%		
Case2: Aggregating the ShMem1: Reserv and Write -> eliminates 2 semaphore acquires/releases and queueing delays	600	67%	650	78.7%

Table 4-5: Performance Results for Processor Utilization

Case	Request Rate/ second	Processor Utilization	Request Rate/ second	Processor Utilization
Case3: Reducing the ShMem2:Alloc to the same execution measurement as the ShMem2:Free. The difference is moved as non-critical code execution of the EventHandler	600	76%		
Case4: Case 1 changes +Case 2 changes +Case 3 changes -> eliminating 5 semaphore acquires/releases and queueing delays -> reducing ShMem2 critical section by making part of the code non-critical	600	68%	700	80%

Observation 5: Waiting Time to acquire the semaphore to perform Reserv and Write

Figure 4-22: Waiting time to acquire ShMem1 semaphore

Table 4-6: Simulation results for the waiting time to acquire ShMem1 semaphore

Case	Request Rate/second	Waiting time	Request Rate/second	Waiting time
Base Case	600	0.22 ms		
Case1: Aggregating the 4 ShMem3 accesses into 1 access -> eliminates 3 semaphore acquires/releases and queueing delays	600	0.24 ms		
Case2: Aggregating the ShMem1: Reserv and Write -> eliminates 2 semaphore acquires/releases and queueing delays	600	0.1 ms	650	0.11ms
Case3: Reducing the ShMem2:Alloc to the same execution measurement as the ShMem2:Free. The difference is moved as non-critical code execution of the EventHandler	600	0.24 ms		
Case4: Case 1 changes +Case 2 changes +Case 3 changes -> eliminating 5 semaphore acquires/releases and queueing delays -> reducing ShMem2 critical section by making part of the code non-critical	600	0.12 ms	700	0.14 ms

4.4.3 Performance Analysis Recommendations

Based on the above results, we have recommended the following to the software designers.

Reduce the size of code that is executed in critical section in general practice and in particular reduce the code executed by the Alloc in ShMem2 so that only necessary operations are performed in critical region.

Aggregate the Reserv and Write of ShMem1, which are executed in sequence into one single operation, which requires a single semaphore lock. In the base model of the system, each of these functions acquires and then releases the semaphore.

Aggregate the four visits to ShMem3, if possible into a single critical region which requires a single lock operation. The main advantages is the elimination of three queueing delays. Functional constraints may prevent this recommendation from being implemented.

4.5 Study 3: Effect of Polling on Base Model

The third study regards the effect of software polling in the system. Software polling as presented in Chapter 2 is a scheduling technique that allows a software process/task to retrieve incoming request from the incoming requests queue at certain intervals. Software polling and interrupt-driven scheduling are the most common techniques found in real-life applications. Section 4.3 and Section 4.4 assumed interrupt-driven scheduling throughout the model. We report two case studies: the first assumes a very high polling overhead on the CPU (10 ms) and the second assumes a much smaller realistic overhead on the CPU (0.02ms).

4.5.1 Polling Modifications on the LQN Base Model

To demonstrate the effect of the polling scheduling discipline on our suggested base model of Section 4.3.1 (Figure 4-11), we have proceeded by modifying the base model such that access to one of the shared memories (ShMem1) is done by polling. On our LQN model of Section 4.3.1, this

means setting polling as the discipline by which EventHandler retrieves its incoming requests (from IOEngineIN) and IOEngineOUT retrieves its incoming requests (from EventHandler).

EventHandler will poll on its incoming requests queue at regular intervals to simulate the polling on the shared memory buffer. If a request is found, then EventHandler will pick it up and start servicing it. However, if no request is waiting then EventHandler will put itself to sleep, releasing the CPU. IOEngineOUT will have a similar behavior with respect to its input queue. IOEngineOUT will poll on its incoming requests queue as well to simulate checking if EventHandler has stored the outgoing reply of an incoming request into the shared memory area. If no outgoing reply is found then IOEngineOUT will put itself to sleep for a pre-defined sleep interval but if an outgoing reply has been found, then IOEngineOUT will proceed with the normal processing (as described earlier in the Chapter) to send it out of the system through StackOUT.

Throughout the following two polling performance studies the value that was changed is the polling overhead on the CPU, whereas the polling interval is kept constant at 10ms for both EventHandler and IOEngineOUT.

4.5.2 Performance Study with a very Large Polling Overhead

This performance study demonstrates an extreme exaggerated case with a polling overhead of 10ms where using polling as a scheduling discipline turns out to negatively affect system performance by consuming more software and hardware resources than the useful work done by the system. We have chosen this case to demonstrate that our changes to the ParaSRVN simulator work properly.

Figure 4-23, which presents the results of running the model for a hardware configuration of one processor and varying the number of EventHandler replications. Note that the utilizations reported for the EventHandler are reported on a per/replica basis. At a first glance, Figure 4-23 shows the presence of a hardware bottleneck, which is the processor. Examining it more closely we notice the utilization of the processor for useful work is about 5% only. However, Figure 4-23 shows that (for the case of four replications of the EventHandler) the EventHandler has a 20% utilization just for polling (which does not include the time that it puts itself to sleep but rather pure CPU polling processing overhead). Hence, the single processor is driven to saturation due to the polling overhead imposed by the multiple replications of the EventHandler. Moreover, the second polling task in the system, IOEngineOUT contributes as well in saturating the processor. However, the IOEngineOUT utilization includes waiting for the IOEXEC task as we are modelling the IOEngine as one process.

Figure 4-23: Utilizations for 1 processor

In the case of four processors, the high polling overhead has similar disastrous effects on performance, as shown in Figure 4-24. The processor utilization for useful work is only 0.05%, meaning that the processor utilization uniquely due to polling overhead processing is ~99.9%. Figure 4-24 shows that IOEngineOUT useful work is the highest due to its servicing outgoing replies generated by the many replications of the EventHandler.

Figure 4-24: Utilizations for 4 processors

Different numbers but same pattern of utilizations is also observed for the six processors case (Figure 4-25).

Figure 4-25: Utilizations for 6 processors

Figure 4-26: Achievable throughput for 1, 4 and 6 processors

Figure 4-26 presents the results for the achievable throughput for the three different hardware configurations. The throughput curve for the four and six processors cases exhibit a similar pattern, where the throughput of the system degrades with more EventHandler replications. However the throughput curve of the one processor case start off at 31.8 requests and increases to 39.7 for 2 EventHandler replication but then decrease dramatically as more EventHandler replications are added into the system.

This study demonstrates that the effect of scheduling (in some cases) is a factor that must be incorporated into the model in order for the performance study to provide accurate predictions. Moreover, we can also conclude that polling overheads considerably large (relative to useful work) are expected to limit system performance.

4.5.3 Performance Study with a very Small Polling Overhead

The model used in this study differs from Section 4.5.3 in the value of the used polling overhead. The model studied in this section uses a very small realistic polling overhead set to a value of 0.02ms to study the effect of polling in the real system.

Figure 4-27 shows the two curves for the EventHandler almost the same, which means that the EventHandler is almost 0% utilized due to polling processing. The Processor curves also demonstrate less than 1% of utilization due to polling. Moreover, comparing the obtained results with results of the study of Section 4.4.1 (which assumes interrupt-driven scheduling) we can conclude that the use of polling does not have a significant effect on performance.

Figure 4-27: Utilizations for one processor

Figure 4-28: Utilizations for four processors

Figure 4-29: Utilizations for six processors

Figure 4-30: Achievable throughput for 1, 4 and 6 processors

Figure 4-30 presents the results for the achievable throughput for the three different hardware configurations. Similarly behaving, the throughput curve for the four and six processors cases exhibit a similarity.

Consequently, we can conclude that the LQN solver does not need to consider polling scheduling of software servers if the polling overhead is considerably small with respect to the execution demands. However, if the polling overhead is significantly larger then polling must be incorporated into the system model to obtain accurate performance predictions.

Chapter 5

Software Designer Interface

This chapter emphasizes the need to make performance analysis more accessible by introducing the concept of a so-called Software Designer Interface. A discussion based on the LQN toolset is presented to demonstrate the concept and further elaborate on the benefits.

5.1 SDI and SPE

Software Performance Engineering, proposed by Connie Smith in [Smith90], advocates that performance modelling should start at the beginning of the software development and be continued throughout the software life cycle. SPE performance assessment activities are recommended to be conducted at each step of the software design cycle.

The actual industrial practice (with some exceptions) with respect to SPE assessment activities, is that performance analysis is conducted by performance analysts who are usually called upon only when capacity planning concerns about the system start to surface. If on the other hand, such assessment activities can be performed by the software designers of the software, SPE activities can be integrated more smoothly within the software life cycle and required hardware equipment or soft-

ware and design optimizations can be identified on time leading to controlled project costs. In any case, SPE practice can only be achieved if the performance assessment activities are conducted by the software designers in parallel with their development.

Our experience with the industry has shown that software developers seldom use any performance engineering tools but rather depend on the services of specialized performance analysis departments in their own company. As we found out, software developers would like to use performance tools tailored to their specific problems to produce on the spot quantitative answers to their questions, as long as they would be spared from learning any performance modelling methods, and would be allowed to express their questions and get the answers in the software domain (as opposed to the performance modelling domain).

However, today's general-purpose performance tools, such as analytic solvers, simulation packages and experiment controllers are built to satisfy the needs of performance analysts, not those of software developers. The inputs and outputs of these tools are expressed in the performance modelling domain, and they produce usually a huge amount of results. A performance analyst has to manage the output files, analyze these results, extract the useful conclusions and present them to the interested parties in a concise, meaningful way (usually as graphs, charts or tables).

Since current performance tools are only intended for the performance analyst community they are too difficult and complex to use by software designers with no performance background. In order to bridge the gap between software development and performance analysis, the thesis proposes

to introduce a simpler application oriented interface provided specifically for the designers, which we will define as the *Software Designer Interface* (SDI).

Our version of SDI does not propose, by any means, to change the performance engineering approach or the way the LQN models are built and solved, but only to add an abstraction layer between developer and the performance model. By adding such a layer we attempt to fill the semantic gap between the software architecture and the performance analysis domain, and to encapsulate the performance model and its analysis. Software designers have certain questions, that may require a number of experiments to be performed and certain results to be gathered, which may involve different models being run and lots of results being generated. SDI will filter these results and provide a succinct report (containing graphs or tables) that answer the developer's questions.

5.2 Definition of SDI

A Software Designer Interface (SDI), as proposed in the thesis, is an application tailored interface intended to be used by software application designers/developers. It uses the architectural view of the system as seen by the software designers and hides the underlying complexity of performance modelling. It can be thought as an extra layer of abstraction from the details of performance modelling to the specifications of the system architecture. More exactly, in our case the software designer's view of the system architecture is the one presented in Figure 4-4, whereas the performance analyst's view is the model from Figure 4-10.

In the case of our modelled system, the software designers wanted to have at their disposal a performance tool with the following characteristics:

- hides the LQN performance model for a given software architecture;
- hides ANY LQN input/output file management.
- accepts input parameters in the software domain (i.e. according to the view from Figure 4-4), as for example: execution times of different processes on behalf of a given type of service, length of various critical sections, number of replications for different processes, etc.;
- controls the execution of the model solver for a given range of request mixes and configurations, and relieves the developers from the tedious task of managing model and result file;
- produces meaningful results (such as throughputs, response time, processor utilization) in a clear form (graphs or tables);
- provides only the desired answers and doesn't overwhelm the developers with an enormous number of results (as the general-purpose performance tools usually do);
- provides a simple diagnosis of performance problems, such as a ranking of potential bottlenecks.

In an ideal case, SDI would be able to translate automatically between the architectural view of the system and the performance domain. However, this is much more complex than can be accomplished within the scope of this thesis. To build a robust/re-usable SDI, we will need a formal method of translation/generation which is generic, yet produces a tailored SDI for each application. Our SDI,

as introduced in this chapter is only a first attempt to shelter the user from the complexity of the performance model. More research needs to be done in the future to produce a complete SDI.

Our version of SDI is manually built by the performance analyst. The performance analyst builds the LQN model as well as the hard-coded scripts that constitute SDI. The role of the scripts is to collect the input from the software designer (which may trigger changes in the LQN model file), to call the LQN experiment controller named SPEX (see next section) on behalf of the software designer, and to extract the results from behind the scenes. This SDI is specific to one LQN model and cannot be dynamically ported to different applications, since different applications require different models. This alpha version of an SDI is similar to a shell that is composed of IO processing scripts that run the LQN experiment controller (SPEX). The performance analyst intervention thereafter is solicited only when new types of questions/capacity planning concerns are requested by the software designers, or when the system architecture changes so drastically that unforeseen modifications of the LQN model become necessary.

SDI takes its input values from the software designer, and modifies the encapsulated LQN model. The software designer is prompted with application specific questions (to be discussed in more detail in the next section). The software designer does not know anything about the underlying LQN models and their parameters, most of which are pre-defined in files and remain unchanged.

The SDI output is rather very rigid, as the selection and type of reports is done only once, when the SDI is coded. A limited flexibility is allowed to the software designers, whereby they may choose the number of results to be plotted on graphs, or may choose to see some results at all.

In conclusion, our version of SDI is a simplistic automated intelligence logic that maps from the software architecture view to the performance domain and vice versa.

5.2.1 SDI Functions from a Software Designer's Perspective

SDI can provide the designers with performance analysis results of different software or hardware architectures under different workloads through a primitive set of functions tailored to the system under study. Recalling the SDI definition, those functions allow the software designer to modify the encapsulated LQN model without knowing anything about the model or the parameters involved. The functions provided to the designers through SDI can be divided into four categories: software architecture changes, hardware architecture changes and results customizers. These functions modify in different degrees the LQN model file (as will be elaborated further in Section 5.2.2).

Functions to change the software architecture can be any of the following:

- Changing number of forked processes for major critical path processes such as Stack, IOEngine, EventHandler or the Dbase;
- Changing in execution times or frequency of major processes located on critical path on a per request basis, such as Stack, IOEngine, EventHandler or the Dbase;
- Changing in semaphore acquire and release overheads;
- Changing in polling overhead or polling sleep interval, if applicable;

Functions to change the hardware architecture can be any of the following:

- Modifying number of allocated processors in processor pool of the modelled system;
- Allocate the Dbase, Stack, IOEngine on a separate processor;
- Restore the Dbase, Stack, IOEngine to run on the processor pool;
- Scale to a higher-speed processing capacity;

Functions to change the workload for the modelled system can be any of the following:

- Study the system at maximum capacity;
- Study the system for various request rates;
- Change the request mix;

Gathering the inputs and presenting the requested results/answers to the software designer is a major requirements specifications for SDI. *Display functions*, the fourth type of functions, are also provisioned to relieve the designer from managing the different output files and prevent him from being swamped by the output of the solvers. The following functions provide the software designer with the capability to customize the output graphs by choosing only the results he wishes to see from a list of options. It is worth mentioning, that list of options is but a subset of what a performance analyst has access to and is again translated in terms of application processes.

Input/Output Functions provided by our SDI, in order not to overwhelm the designers with unnecessary and uninteresting results, is as follows:

- Which process is the system bottleneck under this configuration/workload?
- Throughput of critical path processes such as Stack, IOEngine, EventHandler or the Dbase;
- Response Time of critical path processes such as Stack, IOEngine, EventHandler or the Dbase;
- Utilization of critical path processes such as Stack, IOEngine, EventHandler or the Dbase;
- Utilization of processors in the processor pool or of separate Dbase processor, if applicable.

5.2.2 SDI Functions from a Performance Analyst's Perspective

Since the SDI is responsible for communicating with the application designer in software architecture terms, the software designer is not aware of the underlying LQN logic. SDI does not converse in terms of the LQN performance model, such as in tasks, entries, phases of an entry or asynchronous requests but in term of software processes and modules, that are visible within the development environment.

The SDI functions presented in the last subsection, have different effects on the LQN model, when looked upon from a performance analyst view. These functions can be categorized into three types: functions that modify the architecture of the LQN model in the input file, functions that modify the parameters of the LQN model in the input file and functions that collect/manage the output files.

Changing the number of forked processes, execution times, frequency of execution of major processes, semaphore acquire and release overheads, polling overhead, polling sleep interval, number of allocated processors in the processor pool, scaling to higher processing speed are all functions that change/update a value in the LQN input file. For example, hardware configuration changes requires knowledge of all the tasks assigned to the processor in question.

More complex SDI functions that lead to adding/removing of lines in the LQN input file are as follows: allocating a process on a separate processor, studying the system at maximum capacity, studying the system for various request rates.

Allocating a process (e.g. Dbase, Stack, IOEngine) on a separate processor will add/remove a new line in the processor declaration section (to declare the new processor) and will modify the task allocations in the LQN input files. Modifications that are transparent to the user would modify all tasks that the SDI knows to be part of the process to be allocated to the correct processor. This will involve appropriate modifications in the task declaration section of the LQN input file.

Studying the system at maximum capacity will close the LQN model by adding a new multi-server reference client task to saturate the system and add a request arc from that client task to the Stack entry. Whereas, studying the system with various request rates will open the LQN model by adding an line in the entry declaration section to associate an arrival source to the Stack entry.

The last SDI functions, listed in Section 5.2.1, are the display functions. To extract the requested results, SDI will insert the specification codes for the results (e.g.%u0,%s1,

etc...[SPEX97]) in the appropriate task or entry or processor declaration section of the LQN input file. Then, after the experiment is run, SDI will gather those results from the file produced by the experiment controller, perform some cut and paste and call gnuplot to print the results either on the screen or to a postscript file (readable by Framemaker software on the Unix platform).

In conclusion, the global objective of SDI is to make the running of performance experiments a transparent process to the designer, where the designer is only aware of the changes that they specify in the system architecture and of the output graphs that they request.

Our version of SDI is very primitive, however it brings into perspective the designer needs. Future research will lead to more sophisticated SDI, where the generation process to and from the LQN model can be automated.

5.3 SDI vs. SPEX

Performance awareness among the industry is growing and performance software tools are no longer considered scarce. However, most (if not all) the tools that are available today on the market or in the academic research labs, are not aimed to be used by performance analysts.

The LQN toolset contains an experiment controller named SPEX [SPEX97] [MultiSRVN97]. SPEX is very powerful in controlling LQN experiments and running them on multiple hosts for solving the performance models and extracting the results, but can only be deployed by LQN performance

specialists. Only a performance analyst that understands LQN can use SPEX and understand the outputs.

SDI does not imply replacing the existing experiment controllers, such as SPEX, but rather building on top of them. SDI is an interface that can be viewed as an additional layer on top of the experiment controller.

5.3.1 Overview of SPEX

SPEX is our LQN experiment controller, which can launch any of the LQN solvers, such as `lqns` or `ParaSRVN`. It is built using Perl 5.004 and uses the powerful pattern-matching capability of Perl to parse its input and to extract the results using the parseable output file (with a `.p` extension) produced by the LQN solver. However, the results to extract are specified in the LQN input file by prefixing a `%` to a SPEX argument and following it by a variable name. The following figure Figure 5-1 presents an extract from an LQN input file to illustrate the syntax and complexity of the notation:

```
t T_IO_exec n E_IO_exec_IN E_IO_exec_OUT -1 P_1 %u0 $uTIO_exec
-1
E 17
s E_Clients 1e-18 0 0 -1 %f $fClients
y E_Clients E_HX_IN 1 0 0 -1 %w0 $soarh
F E_HX_IN E_IO_IN 1 -1
z E_Clients E_HX_IN 1 5 -1 %s0 $sclients
```

Figure 5-1: Extract from an LQN model input file

Figure 5-1 presents a minute extract from an LQN input file. The lines that start with the letter ‘t’ are LQN task definitions, whereas the following subsection starting with the letters ‘s’, ‘y’, ‘z’, ‘F’ (as in the figure) are entry declarations. The ‘s’ implies a specification of service time for the three phases of an entry declaration, the ‘y’ a specification of a synchronous rendez-vous rate from entry1 to entry2, the ‘z’ a specification of asynchronous request rate from entry1 to entry2, the ‘F’ determines which of the three phases of an entry is stochastic or deterministic.

The complexity of the LQN file illustrated in Figure 5-1 further implies that to extract trivial results using SPEX, proficiency in LQN syntax and modelling concepts is necessary. As shown in the figure, some of the specifications such as %w are followed by a number and the same for %s and %u. The number denotes the phase of the entry/task for which statistics are required, for example “%u0 \$uTIO_exec” means the utilization of phase 0 of task called T_IO_exec.

The purpose of this extract was to demonstrate the inadequacy how complex is the use of SPEX and how inadequate is SPEX as an interface for software designers. More references on the LQN input file syntax are found in [Franks95] and SPEX specifications is documented in [SPEX97].

5.3.2 Using SPEX to Design SDI

Our version of SDI uses a combination of UNIX shell scripting and Perl. The shell scripts get the input from the user, invoke SPEX to run the experiments, manage the model and result files, extract the desired results and present them in the desired form. The Perl scripts match the lines to be

modified and substitute them or remove them, depending on the SDI function, as described in Section 5.2.2. Figure 5-2 presents the pseudo-code for the SDI's main routine.

- Get input changes from software designers using shell scripts*
- Modify LQN input file calling Perl scripts*
- Manipulate LQN output files using Perl scripts*
- Plot results according to user request using gnuplot*

Figure 5-2: Pseudo-code for SDI main routine

The interface to the user using the shell scripts provides a regular text-based interface with no graphical user-friendliness. The shell scripts role can be compared to the role of the `main()` function in any C program.

However, when SDI functions require modifications to the LQN input file, control logic will build a pattern of the lines to modify and supply it to the Perl script which will match it in the input file. Once the match is successful the line is returned to the shell script that will modify it and request from the Perl script to substitute the matched line with the new line. Another Perl script is also invoked to parse the “.dat” file (produced by SPEX and contains the data to be plotted) and to modify the “.gnu” file (produced by SPEX and controls the way the data in the “.dat” file is plotted) in order to plot the graphs whether on the display or in a postscript file.

In conclusion, our version of SDI is a first attempt to bridge the gap between the software designer's view and the performance model. By using SDI, the developers will be able to run directly performance experiments for different design alternatives, configurations and workload mixes, without having to learn the intricacies of the LQN model and of its experiment controller. This simple SDI is but the first step to put a performance modelling tool in the hands of the software developers, contributing to the effort to integrated SPE into the software development process.

Chapter 6

Conclusions

The first goal of the thesis was to apply an academically developed performance modelling technique to an industrial system. The results of this study have been reported in Chapter 4 of the thesis.

With this occasion, we identified a number of weaknesses in the LQN simulation tool. The thesis extended the LQN simulator “ParaSRVN” with three features, as reported in Chapter 3. The three extensions are: incorporating polling servers, capturing queue length statistics and enhancing the ability to simulate open models.

The first part of Chapter 4 shows how the LQN model was built by applying a series of transformations to the system architecture. The model parameters were obtained by measuring the real system. After describing the base model, the thesis reports three performance analysis studies on the modelled system.

The first study identifies the system bottleneck under different configuration, showing how the bottleneck moves from hardware to software. The second study looked at ways to reduce code that is executed in critical section and to aggregate consecutive critical section work to reduce semaphore overheads. The third study analyzed the effect of polling on the system. The study concluded that polling does not impact performance if the polling overhead is very small relative to the process execution times (which is the case of the system under study). However, polling must be incorporated into the model if the polling overhead is relatively large.

The second goal of the thesis was to put in the hands of the software designers a tool that makes it easier to run performance experiments. The thesis introduced the concept of a “Software Designer Interface” as presented in Chapter 5. Although this SDI is still a first attempt, it will help to gather a set of requirements for a more general-purpose SDI. SDI is introduced as an attempt to reduce the cognitive gap between the software and performance domains to promote the use of performance modelling in the software development community.

6.1 Future Work

The matter of this thesis can be further researched in the following directions:

- Extend the LQN analytical solver to allow for polling servers by implementing an exact mathematical solution for polling server.
- Research and Develop performance diagnosis techniques and tools, such that they be embedded in future SDIs.

- Research on formal transformations between the system architectural view and the performance modelling domain.

References

- [Adebayo97] Omotunde Adebayo. *Performance Analysis of Client-Broker-Systems*. M.Eng. thesis. Carleton University, Ottawa, June 1997.
- [Allen90] A. O. Allen. *Probability, Statistics and Queueing Theory with Computer Science Applications*. Academic Press Inc., 1990
- [Baskett75] F. Basket, K.M. Chandy, R. R. Munz, F. G. Palacios. Open, Closed and Mixed Networks of Queues with Different Classes of Customers. *ACM Journal*. Vol. 22, nb. 2, pp. 248-260, April 1975.
- [Bux83] Werner Bux, Hong Linh Truong. Mean-Delay Approximation for Cyclic-Service Queueing Systems. *Performance Evaluation*. Vol.3, nb.3, pp. 187-196, August 1983.
- [Buzen71] J. P. Buzen. *Queueing Network Models of Multiprogramming*. Ph.D. Dissertation. Harvard University. Cambridge, MA, 1971.
- [Boxma86] O. J. Boxma, B. Meister. Waiting-Time Approximations for Cyclic-Service systems with switch-over times. *ACM Sigmetrics 1986, Performance Evaluation Review*. Vol.14, nb.1, pp. 254-362, May 1986
- [Buhr96] R.J.A. Buhr, R.S. Casselman. *A Use Case Map Approach to High Level Design of Object-Oriented Systems*. Prentice Hall, 1996.

- [Conway89] A. E. Conway, N. D. Georganas. *Queueing Networks - Exact Computational Algorithms: A Unified Theory Based on Decomposition and Aggregation*. MIT Press, 1989.
- [Conway86] A. E. Conway, N. D. Georganas. RECAL - A New Efficient Algorithm for the Exact Analysis of Multiple Chain Closed Queueing Networks." *ACM Journal*. Vol.33, pp.768-791, 1986.
- [Franks95] Greg Franks, Alex Hubbard, Shikharesh Majumdar, Dorina Petriu, Jerome Rolia and Murray Woodside. A toolset for Performance Engineering and Software Design of Client-Server Systems. *Performance Evaluation*, 24(1-2):117-135, November 1995.
- [Franks96] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, C.M. Woodside. Performance Analysis of Distributed Server Systems. In *Proceedings of the 6th International Conference on Software Quality*, pp. 15-26, Ottawa, Canada, October 1996.
- [Friedrich96] R. Friedrich and J. Rolia. Applying Performance Engineering to a Distributed Application Monitoring System. *Proceedings of the International Conference on Distributed Processing*. Dresden, Germany. February 1996.
- [Hills95] G. Hills, J. Rolia, G. Serazzi. Performance Engineering of Distributed Software Process Architectures. Lecture notes in Computer Science, *Springer Verlag*, No. 977, pp. 79-85, 1995.
- [Jain91] Raj Jain. *The Art of computer systems performance analysis; techniques for experimental design, measurement, simulation, and modelling*. New York, Wiley. 1991.

- [Jogalekar98] P. Jogalekar, C.M. Woodside. Evaluating the Scalability of Distributed Systems. *To appear in Proc. of Hawaii Int. Conference on System Sciences*, January 1998.
- [Lazowska84] E. D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance*. Prentice Hall, 1984.
- [LQNS] LQNS command man page.
- [Maly97] Peter Maly. *Applying Layered Modelling to a Bridge/Router Architecture*. M. Eng. thesis. Carleton University, Ottawa, August 1997.
- [Menasce94] D.A. Menasce, V.A.F. Almedia and L.W. Dowdy. *Capacity Planning and Performance Modelling*. Prentice Hall, 1994
- [Metea96] Luminita S. Metea. *Performance Modelling and Measurements of Object-Oriented Software*. M. Eng. thesis. Carleton University, Ottawa, September 1996.
- [Molloy89] Michael K. Molloy. *Fundamentals of Performance Modelling*. Mac Millan Publishing Company, New York, 1989.
- [Murata89] Tadao Murata. *Petri Nets: Properties, Analysis and Applications*. Proceedings of the IEEE, Vol.77, nb.4, pp.541, April 1989.
- [MultiSRVN97] *User's Guide for MultiSRVN Version 4.0*. January 7,1997.
- [Neilson95] John E. Neilson, C. Murray Woodside, Dorina C. Petriu, and Shikharesh Majumdar. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Transactions on Software Engineering*, 21(19):776-782, September 1995.
- [PARASOL1] *PARASOL Users' Manual, Version 3.1*. School of Computer Science, Carleton University.

- [PARASOL2] *A PARASOL Tutorial*. School of Computer Science, Carleton University, John Neilson and Pat Morin
- [ParaSRVN] ParaSRVN command man page.
- [Petriu91a] Dorina C. Petriu. *Approximate Solution for Stochastic Rendezvous Networks by Markov Chain Task-Directed Aggregation*. Ph.d. Dissertation. Carleton University, April 1991.
- [Petriu91b] D.C. Petriu, C.M. Woodside. Approximate MVA from Markov Model of Software Client-Server Systems. *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, pp.322-329, 1991.
- [Petriu94] D.C. Petriu. Approximate Mean Value Analysis of Client-Server Systems with Multi-Class Requests. *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems. Performance Evaluation Review*. Vol.22, nb.1, pp. 77-86, May 1994.
- [Quantify] Pure Software. *Quantify User's Guide*. Pure Software Inc., 1995
- [Reiser80] M. Reiser, S. S. Lavenberg. Mean-Value Analysis of Closed Multichain Queueing Networks. *Journal of ACM*. Vol. 27, nb.2, pp. 313-322, April 1980.
- [Rolia87] J. A. Rolia. Performance Estimates for Multi-tasking Software Systems. Master's Thesis. University of Toronto, Canada, January 1987.
- [Rolia92] J.A. Rolia. *Software Performance Modelling*. Ph.D. Dissertation, CSRI Technical Report #260, University of Toronto, Canada, January 1992.
- [Rolia95] J.A. Rolia and K.C. Sevcik. The Method of Layers. *IEEE Trans. on Software Engineering*, 21(8):689-700, August 1995.

- [Rolia96] J. Rolia and R. Friedrich. Quality of Service Management for Federated Applications. *Proceedings of the 4th International IFIP Workshop on Quality of Service*. Paris, France, March 6-8 1996, pages 259-270.
- [Schmidt95] Douglas C. Schmidt, Charles D. Cranor. *Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-Structured Concurrent I/O*. Proceedings of the Second Pattern Languages of Programs conference in Monticello, Illinois, September 6-8, 1995.
- [Shaw96] Marry Shaw. *Some patterns for Software Architecture*. In J. Vlissides, J. Coplien, and N.Kerth (eds), *Pattern Languages of Program Design2*, Addison-Wesley, 1996, pp.255-269.
- [Sheikh97a] F. Sheikh, C.M. Woodside. Layered Analytic Performance Modelling of a Distributed Database System. *Proc. 1997 International Conference on Distributed Computing Systems*, May 1997, pp. 482-490.
- [Sheikh97b] Fahim A. Sheikh. *Sensitivity Analysis of Performance Predictions for Distributed Applications*. M. Eng. thesis. Carleton University, Ottawa, December 1997.
- [Shousha98] Christiane Shousha, Dorina Petriu, Anant Jalnapurkar, and Kennedy Ngo. *Applying Performance Modelling to a Telecommunication System*. Accepted for publication at the Workshop on Software Performance, October 1998.
- [Smith90] C.U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.
- [SPEX97] Alex Hubbard. *SPEX, Software Experiment Driver, version 1*. Real Time and Distributed Systems Group. Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada.

- [Takagi90] Hideaki Takagi. Queueing Analysis of Polling Models: An Update. *Stochastic Analysis of Computer and Communication Systems*. North-Holland, New York, 1990, pp. 267-318.
- [Trivedi82] K.K. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. Prentice Hall, 1982
- [Vernon1987] M. Vernon, J. Zahorjan, E. D. Lazowska. *A Comparison of Performance Petri Nets and Queueing Network Models*. Proceedings on the International Workshop on Modelling Techniques and Performance Evaluation. Paris, 1987, pp.191-202.
- [Woodside89] C.M. Woodside. Throughput Calculation for Basic Stochastic Rendezvous Networks. *Performance Evaluation*, Vol.9, pp143-160, April 1989.
- [Woodside95a] C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar. The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. *IEEE Transactions on Computers*, 44(1):20-34, January 1995.
- [Woodside95b] C.M. Woodside, S. Majumdar, J.E. Neilson, D.C. Petriu, J.A. Rolia, A. Hubbard, R.B. Franks. *A Guide to Performance Modelling of Distributed Client-Server Software Systems with Layered Queueing Networks*. SCE-95-23, Carleton University, Ottawa, Canada, November 1995.
- [Woodside98] C.M. Woodside. Throughput Calculation for Basic Stochastic Rendezvous Networks. *Performance Evaluation*, vol.9(1), April 1988.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved