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.

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.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI[®]
800-521-0600

**DESIGNING HIGH PERFORMANCE
SOFTWARE SYSTEMS - A
QUANTITATIVE APPROACH**

by

Marin LITOIU

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

Doctor of Philosophy

Carleton University, Ottawa

Ottawa-Carleton Institute for Electrical and
Computer Engineering

04/12/1999

© copyright 1999 Marin Litoiu

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.

0-612-42800-1

Canada

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

**Designing High Performance
Software Systems- A Quantitative
Approach**

submitted by **Marin Litoiu**,
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Chair

Department

Thesis Supervisor

External Examiner

Carleton University
04/12/99

Carleton University

Abstract

DESIGNING HIGH PERFORMANCE SOFTWARE SYSTEMS - A QUANTITATIVE APPROACH

by Marin LITOIU

The work carried out for this thesis develops performance-engineering techniques to be used in the early phases of distributed systems design and implementation. While these techniques are autonomous, they could also be used as a suite in a framework. The final result of framework would be a configuration or a set of configurations for distributed application that satisfies user expectations for various target environments and for a large range of possible workload conditions.

There are four areas we focus on: performance model building, object and process allocation, mapping complex performance requirements onto multi-criteria decision-making problems, and choosing object replication and process threading and replication levels. We consider closed multi-class systems. The framework takes as input performance requirements, hardware topologies for deployments of various scales, and design constraints. An aggregated performance metric called Satisfaction is derived from multi-class performance requirements. Based on a performance model that is built from tracing executable code, the framework builds a set of configurations for each topology. The configurations optimize the Satisfaction for different workloads. Techniques such as non-linear and linear programming are used to reduce the configuration search space. Then configurations are filtered through a multi-criteria decision-making process to select a configuration or a smaller set of configurations that support the topologies for a large range of workloads. Fuzzy set theory is applied throughout the decision-making process. Finally, appropriate threading or replication levels and activation policy are decided for each object. Again, to reduce the complexity of the computations, we make use of performance evaluation techniques and optimization techniques such as non-linear and linear programming.

ACKNOWLEDGMENTS

The author would like to thank his supervisor, Jerry Rolia for his support and suggestions about the research and writing of this thesis.

My thanks to Giuseppe Serazzi, from Politecnico di Milano, Italy, for his useful suggestions.

The author is thankful for the financial support provided by IBM Toronto Lab, through Centre for Advanced Studies, and by the Telecommunications Research Institute of Ontario (TRIO).

Many thanks to my family, Alexander and Milena, for their patience on weekends and evenings, and for their moral support.

To Alexander

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	Performance Engineering of Distributed Applications	1
1.2	General Overview of the Framework	7
1.2.1	Step 1- Define Performance Requirements, Design Constraints and Topologies	7
1.2.2	Step 2- Build a Layered Queuing Model	13
1.2.3	Step 3- Generate Configurations for Medium, High and Very High Population Levels	14
1.2.4	Step 4- Choose a Configuration that Best Satisfies the Performance Requirements	16
1.2.5	Step 5- Determine the Object Order of Multiplicity	17
1.3	Background and Related Work in the Literature	19
1.3.1	Performance Evaluation Techniques	19
1.3.2	Object Allocation	24
1.3.3	Handling Complex Performance Requirements and Multiple Design Alternatives	28
1.3.4	Object Replication	30
1.4	Main Contributions of this Thesis	30
2	PERFORMANCE MODELS FOR DISTRIBUTED APPLICATIONS	34
2.1	Introduction	34
2.2	A Target Model - LQM	35
2.3	Using EXTRA to Characterize Scenarios and Model Structure	37
2.4	Summary	41
3	CHOOSING OBJECT REPLICATION LEVEL AND ACTIVATION POLICY - A QUANTITATIVE APPROACH	43
3.1	Introduction	43
3.2	Maximum Object Utilization in a Multi-Class Distributed System - A Non-Linear Programming Problem	45
3.2.1	Device and Object Utilization in Distributed Multi-class Systems	45
3.2.2	Posing Maximum Object Utilization as a Non-Linear Programming Problem	53
3.2.3	Slack Variables	57

3.2.4	Introducing other Design Constraints	58
3.3	Bottlenecks, Extreme Points, Saturation Sectors and Crossover Points	61
3.3.1	Extreme Points in a Polyhedron	61
3.3.2	Global and Partial Saturation Sectors [Bal97]	63
3.3.3	Properties of Extreme and Crossover Points	65
3.3.4	An Algorithm for Finding Extreme and Crossover Points	73
3.4	Algorithms for Finding Maximum Object Utilization	75
3.4.1	The Search Algorithm - Non-asymptotic Conditions (NLP_NASYM)	76
3.4.2	The Search Algorithm in the Presence of Saturated Software Entities	82
3.4.3	The Search Algorithm in the Presence of Saturated Devices	84
3.5	Complexity of LP_ASYMP and NLP_NASYMP	90
3.6	Case Study - Using Object Utilization to Decide Object Activation Policy	91
3.7	Summary	94
4	OBJECT ALLOCATION IN MULTI-CLASS DISTRIBUTED SYSTEMS	97
4.1	Introduction	97
4.2	The Need for Considering Low, Medium and High Population Levels	100
4.3	Analysis of Response Times and Object Allocation at Very High Population Levels	104
4.3.1	Response Time in Single-Class Systems	106
4.3.2	Response Time in Multi-Class Systems with a Common Natural Bottleneck	108
4.3.3	Response Time in Multi-Class Systems with Multiple Natural Bottlenecks	118
4.4	Allocation Algorithm for Very High Population Levels	125
4.4.1	Fair Response Time - Allocation Algorithm	126
4.5	Finding Configurations under Medium and High Population Levels	128
4.5.1	An Example	130
4.6	Summary	133
5	CASE STUDY: FINDING CONFIGURATIONS THAT BEST SATISFY COMPLEX REQUIREMENTS	136
5.1	Case Study	137
5.1.1	Performance Requirements	139
5.1.2	Finding Configurations	140
5.1.3	Finding Maximum Object Utilization	143

5.2	Mapping Configuration Choice onto a Decision Making Process	147
5.2.1	Alternatives	147
5.2.2	Criteria for Evaluating Alternatives	149
5.2.3	Rating Configurations	149
5.2.4	Aggregating Criteria, Weights and Rates	152
5.2.5	Ranking Alternatives	152
5.3	Summary	154
6	CONCLUSIONS	156
	APPENDICES	164
	REFERENCES	179
	GLOSSARY	185
	INDEX	192

LIST OF FIGURES

Figure 1.....	3
Figure 2.....	8
Figure 3.....	9
Figure 4.....	10
Figure 5.....	12
Figure 6.....	16
Figure 7.....	16
Figure 8.....	18
Figure 9.....	34
Figure 10.....	39
Figure 11.....	40
Figure 12.....	44
Figure 13.....	46
Figure 14.....	49
Figure 15.....	50
Figure 16.....	51
Figure 17.....	52
Figure 18.....	54
Figure 19.....	60
Figure 20.....	63
Figure 21.....	67
Figure 22.....	69
Figure 23.....	70
Figure 24.....	70
Figure 25.....	71
Figure 26.....	74
Figure 28.....	76
Figure 28.....	82
Figure 29.....	86
Figure 30.....	87
Figure 31.....	89
Figure 32.....	93
Figure 33.....	93
Figure 34.....	98
Figure 35.....	101
Figure 36.....	102
Figure 37.....	103
Figure 38.....	106

Figure 39.....	110
Figure 40.....	113
Figure 41.....	114
Figure 42.....	116
Figure 43.....	122
Figure 45.....	124
Figure 46.....	127
Figure 47.....	129
Figure 48.....	129
Figure 49.....	131
Figure 50.....	132
Figure 51.....	133
Figure 52.....	137
Figure 53.....	138
Figure 54.....	140
Figure 55.....	141
Figure 56.....	142
Figure 57.....	144
Figure 58.....	146
Figure 59.....	151
Figure 60.....	167
Figure 61.....	168
Figure 62.....	169
Figure 63.....	170
Figure 64.....	171

LIST OF TABLES

Table 1.....	41
Table 2.....	50
Table 3.....	82
Table 4.....	88
Table 5.....	88
Table 6.....	90
Table 7.....	91
Table 8.....	101
Table 9.....	102
Table 10.....	102
Table 11.....	130
Table 12.....	131
Table 13.....	132
Table 14.....	138
Table 15.....	140
Table 16.....	143
Table 17.....	145
Table 18.....	147
Table 19.....	150
Table 20.....	152
Table 21.....	153

LIST OF APPENDICES

Appendix 1.....	164
Appendix 2.....	165
Appendix 3.....	166
Appendix 4.....	167
Appendix 5.....	168
Appendix 6.....	169
Appendix 7.....	172
Appendix 8.....	173
Appendix 9.....	174
Appendix 10.....	175
Appendix 11.....	176
Appendix 12.....	178

LIST OF THEOREMS

Theorem 1..... 66

Theorem 2..... 68

Theorem 3..... 71

Theorem 4..... 114

Theorem 5..... 115

Theorem 6..... 120

Theorem 7..... 121

Theorem 8..... 121

1 INTRODUCTION

*“Will you walk a little faster?” said a whiting to a snail,
“There’s a porpoise close behind us, and he’s treading on my tail.”
Lewis Carrol, Alice’s Adventures in Wonderland*

1.1 Performance Engineering of Distributed Applications

*In skating over thin ice, our safety is in our speed.
Ralph Waldo Emerson, Prudence*

Distributed systems cover many application domains: telecommunications, multimedia, control systems, transaction processing systems and others. In order for organizations to be part of the global computing infrastructure, it has become necessary for them to adopt software solutions than can be easily adapted when the application domain model changes. Driven by this force, software companies are finding their way to the distributed object technologies. CORBA [Sei96], CBus [CBCo], DCOM [Orf97] and Java RMI [Orf97] are four of today’s object infrastructures. Designers must consider many architectural choices when exploiting these infrastructures and combining them with other technologies. For example, there are middleware environments that provide support for transaction processing; more general-purpose middleware technologies such as DCE [Har94] for wrapping applications and making their interfaces accessible in open environments; and Internet technologies such as HTML/HTTP [Dec95] and Java** [Dec96]. In general, designers must choose combinations of these technologies for their systems in order to ensure short time to market, scalability, access to legacy applications and high performance.

From a designer's¹ perspective, a distributed object application of today (and most likely one of tomorrow) is built from *objects* that discover each other and interact through *method* invocations (for a recent vision of component-oriented design, see [Sip98]). The objects are distributed across two or more logical tiers and grouped in *clusters* or *modules* of related objects. They are instantiated in operating system *processes*, which run on *nodes* that have hardware components such as processors and disks. This model is similar to the reference model for Open Distributed Processing [ISO], to the Object Management Group's Common Object Request Broker Architecture (CORBA), and to the Java RMI model.

In summary, a distributed object model includes the following:

- *Objects*;
- The *methods* of objects;
- *Clusters* (or *Modules* or *Containers*) of *Objects*;
- *Processes* (or *Capsules*); and,
- *Nodes*.

From a customer's² perspective, a distributed object application offers services. Customers make use of services by submitting requests. Customers of the same *class* have request usage patterns that are statistically similar. A customer of the application shares the distributed systems with customers of the same class or with customers that belong to other classes. The number of customers of a class is the *class population*. The *total customer population* (also referred to as the *customer population* or *population level*) is the total number of customers using the system, while the *class population mix* (or *population mix*) denotes the ratio of each class population to the total customer

¹ Here, designer denotes a professional that is involved in defining the architecture, design and implementation of a distributed application.

² Customer is a performance engineering term for user. We will use the terms interchangeably throughout the thesis.

population. A *customer class session* defines the requests and corresponding probabilities that characterize a user session; for example, between log on and log off.

Figure 1. Classes of request in distributed systems: a) one type of request per session; b) multiple types of request per session; c) classes in different applications.

Figure 1 illustrates several types of interactions between the users and a personnel application. There are two services offered by the system: querying information about employees, and updating information. Accordingly, there are two request types: *query* and *update*. Let us assume that, on average, there are N users using the application. Figure 1a shows one pattern of requests. Each user requests a single type of service per session. A user alternates between idle times or “thinking,” and the requests. Since there is only one type of service, a performance engineer can associate the request to a *class*. The average number of users submitting update requests (N_1) is the *update class population*. Similarly, the average number of users submitting query requests (N_2) is the *query class population*. For a model that uses these two classes, N_1/N and N_2/N represent the *population mixes*.

Figure 1b shows a second pattern of requests, in which users may need many types of services during a single session. A user alternates between different types of requests by following a statistical pattern; for example, an *update* request (with probability 0.3) is in most cases preceded by one or more *queries* (with probability 0.7). This statistical pattern qualifies the session for a *class*. Figure 1c shows an application model with two distinct distributed applications that share a common object.

Performance engineering [Smi90] considers the performance of an application during the development phase. It aims to avoid the blunders associated with the “fix it later” approach to performance planning, which assumes that one can tune an application or make small changes to the hardware in order to improve application behaviour. The assumption is not always justifiable in distributed environments, where applications rely on many pieces of hardware and software that reside in separate administrative domains.

There are two key problems that must be addressed by a performance study during system design and development:

- The development of a representative performance model for a system; and,
- The development and support of methods and tools that provide timely feedback to influence the design.

Developing a performance model can be a very labor-intensive process. Applications use software and hardware resources. Software structure in a distributed system consists of many interacting layers, including application objects, threads, middleware, operating system processes, networking protocols, and networks. Performance modeling needs to encompass features that affect performance most. A lack of skill and resources limits the ability of most project groups to consider performance when an application is being designed or developed. Software performance engineering research must help overcome these problems.

Building a performance model is only one step in a performance engineering approach. A model offers insights regarding the system behaviour and offers performance metrics such as utilization, throughput, response times and queue lengths. Once a model is built, we need to elaborate design alternatives and solutions to improve the performance of the system. The belief that analyzing the model will suggest straightforward solutions for performance improvement is incorrect in the case of complex systems. In the context of distributed object technology, the performance of a deployed distributed system depends on several factors:

- *Object resource requirements* or, similarly, the processing power of the hardware that hosts them. This is the domain in which designers expend the most effort following the rule: responsive objects generate responsive distributed systems. Efficient compilers, efficient algorithms, caching and buffering, and careful algorithm design are the main means to get responsive software components;
- *Communication overheads* due to the inter-processor communication for objects residing on the same node but in different operating systems processes, or due to the middleware when the objects reside on different hosts;
- *Network latencies and transfer times* for object interactions that require network resources;
- *Queuing delays* encountered at software components (processes, critical sections, semaphores, and others);
- *Queuing delays* at hardware components, due to other requests that share the system.

In a pure distributed world, the distributed system designer builds a system from pre-developed objects. There are several degrees of freedom for controlling the performance behaviour of these systems by tuning one or more of the factors defined above:

- Allocating the objects to nodes;
- Choosing the threading or replication level for processes and/or objects; and

- Increasing the processing power of the hardware.

Large complex systems require systematic techniques to study these issues, especially when the performance requirements regard many types of services. The work carried out in this thesis aims to develop such techniques. We consider transaction systems. Although the techniques developed are autonomous, they may also be used as a suite in a framework. The final result of the framework is a distributed configuration that satisfies the user's expectations. There are four areas of focus in this thesis:

- Specifying complex performance requirements. In a distributed system that offers many services, each service can have its own performance requirements for various workload conditions (population levels and mixes);
- Performance-model building. Although it is not the main focus of the thesis, a pragmatic way of building performance models has been developed and is documented;
- Object allocation techniques that map objects to nodes;
- Object replica and threading level computation. The goal of this technique is to choose concurrency levels that reduce the queuing delays but also limit the inherent use of memory.

We rely on the idea that the performance of the system is complex, and pay a special attention to the population mixes. Population mixes are hard to predict or enumerate, even for applications as simple as the personnel application shown in Figure 1. As a result, the class population, population levels and population mixes will be present in all aspects of our study, starting with performance requirements and continuing with performance models, object threading and object allocation.

The next sections give details of the proposed framework, along with an overview of related work in the literature. The framework takes as input the performance

requirements, hardware topologies for deployments of various scales and design constraints. Based on a performance model that is built from tracing executable code, the framework builds a set of configurations (including object allocation and object multiplicity) for each topology. Eventually, the configurations are filtered through a multi-criteria decision-making process to select a configuration or a smaller set of configurations that support the topologies. Finally, the appropriate threading or replica level are computed for each object.

1.2 General Overview of the Framework

Divide et impera

The subsections that follow describe the components of the framework. The remaining chapters of the thesis justify and explain these steps in detail. A general overview of the framework proposed in this thesis is presented in Figure 2.

1.2.1 Step 1- Define Performance Requirements, Design Constraints and Topologies

A well-structured performance-oriented design should start with well-defined performance requirements, constraints and topologies of interest. Performance requirements are required for each type of service and for key workload conditions. In this thesis, we assume that these workload conditions are from low, medium and high population areas. Thus, we have:

Figure 2. The general framework for building high-performance distributed applications: NLP- non-linear programming, LP-linear programming, MOL-method of layers, -VHL- very high population levels, HL-high population levels, ML-medium population levels, LQM-layered queuing model; ABA - asymptotic bound analysis.

- *Low population area*: the customer population is low, the queuing delays are very small, and the response time of a class is close to the sum of its resource demands;
- *Medium population area*: the queuing delay at devices or software components contributes significantly to response time; as a result, the response times are very sensitive to the customer populations and population mixes;
- *High population area*: the customer population is such that there is at least one device or software entity that is saturated or very close to the saturation limit. In this case, the response times are linear with respect to the customer population, and non-linear with respect to the population mix. These concepts are explained further in Chapter 4;
- *Very high population area*: the customer population approaches infinity. This is the asymptotic case. It has special analytic properties and receives special attention.

Performance Requirements. The designer expects that the system will work under low, medium, and high population areas and we consider population levels from these areas. Population levels from very high population areas expose particularly interesting analytical modeling issues. These properties are considered in the greatest detail in Chapter 4.

Step 1.1 Define the expected mean response times:

- for each class c , at low, medium, and high populations levels (N^{low} , N^{medium} , N^{high}), define required lower (R_{low}^c) and upper (R_{up}^c) mean response times.

Figure 3. Step 1.1.

The performance requirements are specified as intervals with the $[R_{low}^c, R_{up}^c]$ for each class and for each population level and over all population mixes (Figure 3). Expressions such as “when there are 20 customers in the system, a *query* request should offer a

response time between 500 and 700 ms,” and “*update* should offer a response time between 1.0 and 1.5 seconds,” are examples of performance requirements. In this way, the design expectations help with the search for a good distributed system design. The situation when population levels are defined for each class of request is a particular case and can be solved within the proposed framework.

For quality of service management in transactional or soft real-time systems, we prefer measures such as 90-percentiles instead of mean values. However, predicting higher moments that give percentiles for request response times at a system level is a more difficult modeling task; one that requires models with greater detail than those used to predict mean values. Future detailed models combined with simulation or new analytic techniques could also be used within the framework. In the meantime, we must rely on the mean response time and the fact that response time standard deviation tends to increase with mean queuing delays. To keep the 90-percentile down, we must keep our queuing delays low. After we have confidence that the means are low, we can consider resource-intensive performance evaluation techniques for more detailed studies.

Design Constraints.

Step 1.2 For the medium population level, define total device utilization limits b_i , for each device i , and for high population level define the maximum population level (N^{high}).

Figure 4. Step 1.2

These design constraints (Figure 4) reflect technological or economic aspects of the system. At medium population levels, the constraints are given by device utilization³ limits (b_i). Implicitly, these constraints limit the number of users in the system (as will be shown

³ Utilization is the percentage of time a device is busy. A formal definition of utilization is presented in Section 1.3.

in Chapter 3), so no limit on the population level is required in that case. At high population levels, where a device may become saturated, the high population level (N^{high}) may act as a design constraint. Here, b_i are implicitly assumed to be the physical limits. A justification for these requirements will be given in Chapter 4. We will also show, in Chapter 5, that we can take N^{medium} as the only design constraint for medium population levels.

Although we defined limits only on device utilizations, we may have constraints on the software components as well. We will show in Section 3.2.4 how to handle these. These constraints may come from different sources:

- Not all software entities can be replicated (critical sections, for example), so software queuing delays inevitably occur. The presence of critical sections does not depend on the process or object distribution, but can be affected by them;
- There are software components whose replication level is restricted by design or implementation constraints. For example, a target system may only have the license for some integer greater than 0 instances of a database server process subsystem. Alternatively, it may not be possible to introduce additional instances of a database into a system without making significant changes to its application software. In this case, a replication limit would be 1;
- In many middleware platforms (for example, CORBA-based environments [Sei96]), server processes can be persistent or can be created on demand and then terminated. Those server processes that are used frequently should be persistent in order to minimize startup overheads. If the objects within a persistent server process are reentrant, there is the possibility for the server process to be multi-threaded. However, there can be nucleus (operating system) constraints [Hil95] on threading levels. For example, the constraints can limit the number of file descriptors used by a server process's connections to 64. A threading limit would therefore be 64.

Note that there are no explicit constraints about the memory size requirements. However, they can be considered implicitly in the software utilization limits. By imposing a software utilization limit, we limit replication levels and hence memory consumption.

Topologies.

Step 1.3 Define topologies for low, medium and high population levels.

Figure 5. Step 1.3.

In a distributed model, *networks* interconnect *nodes*. *Nodes*, *networks* and *processes* running on nodes constitute a *topology*. *Nodes* are processing elements, usually consisting of *CPU* and *DISK* subsystems. The topology reflects the physical layout of the system as well as an initial allocation of objects. It should also include performance parameters of the hosts and network, as CPU or device ratios. Topology selection or design is a problem in itself. We do not focus on it in this thesis and consider topology as input, but it is of primary importance when planning for high-performance systems. It is expected that a distributed application will be deployed on many different topologies (Figure 5). For example, different topologies may support increasing numbers of customers. The designer must specify these topologies. The techniques presented in this thesis provide for a small set of configurations that support the topologies. Designers can then focus on those configurations that better support the application.

The performance requirements, design constraints, and topologies are used by Steps 3 to 5 of the framework.

1.2.2 Step 2- Build a Layered Queuing Model

The second step of the framework consists of building a performance model for the application. The main assumption here is that the design of the application and prototype code for the application's key features are available. The model must capture resource demand information about the main scenarios through the application. Performance evaluation techniques are then used to estimate performance metrics such as response time, throughput and utilizations.

The *performance model* we rely on is the *Layered Queuing Model (LQM)*[Rol95]. In queuing theory terms, it consists of queuing centres and delay centres. LQMs are described in detail in Chapter 2. Briefly, they include queues for both software components (processes, critical sections, and inter-process communication components) and hardware devices. Visits between software components and hardware device demands are captured in the models.

For performance purposes, we augment the distributed object model to consider *scenario* information. A scenario is an artifact used during systems analysis and design [Col94] to ensure that proposed designs will satisfy the design requirements. A scenario describes a specific user request that the system supports, and gives its corresponding path through the methods of objects in the system. It has very clear semantics for the user or designer.

From prototype code, we build scenarios from trace files generated by an IBM Visual Age for C++ tool named Executable Trace (EXTRA). The EXTRA files contain a detailed timing record for each function (call/return), each object method (call/return) and each thread (start/end). By parsing the file, we can detect the object interactions and compute the device service demands. Then, we map the scenarios to the LQM. The model can then

be augmented to include features not yet presented in the prototype. The design documents are sources for those features.

All the subsequent steps of the framework use the performance model.

1.2.3 Step 3- Generate Configurations for Medium, High and Very High Population Levels

This step aims to find an allocation of objects to nodes and processes such that it best satisfies the design constraints. In the thesis, we focus on generating configurations for very high, high, and medium population levels⁴ under all population mixes. Very high population configurations highlight interesting properties of the system, and provide a good starting point for searches at lower population levels.

Since the performance requirements regard many types of requests, we should define a function that takes them all into account. For each population level, we characterize an allocation A with regard to a class c by a function that we call $Satisfaction_c^l(A)$. It is defined on the set of all allocations and takes on values in the range $[0, 1]$, for $l = \text{medium}, \text{high}$ population levels such that:

- It is 1 when all requirements are met for all mixes; that is, the lower values for response times are fulfilled under all mixes;
- It is 0 if the upper values are not met for some mixes;
- It is linear function that materializes the rule: the closer the worst case per class response time is to the lower value, the closer the value is to 1.

⁴ We do not investigate further the low population levels. In Chapter 4 we show that existing algorithms and methodologies can be used to deal with these levels.

$$Satisfaction_c^l(A) = \begin{cases} 1 & \text{if } R_{worst}^c \leq R_{low}^c \\ \frac{R_{up}^c - R_{worst}^c}{R_{up}^c - R_{low}^c} & \text{if } R_{low}^c < R_{worst}^c \leq R_{up}^c \\ 0 & \text{otherwise} \end{cases}$$

R_{worst}^c is the maximum mean response time that is encountered by class c requests under the design constraints at the specific population level and over all the population mixes at that population. Methods for finding the maximum response time over all the mixes are defined in Chapter 4. They are based on both non-linear and linear programming techniques. Non-linear programming is an optimization method that deals with the problem of finding the minimum or maximum of an objective function in the presence of equality and inequality constraints. When the objective function and constraints are linear, we have a *linear program*; otherwise, we are dealing with a *non-linear program*. Detailed descriptions of the basic notions used in the following sections are presented in Appendix 1.

The $Satisfaction^l(A)$ of an allocation A measures the overall *Satisfaction* per population level l , and it is defined by taking into account the *Satisfaction* of each class c . More exactly: $Satisfaction^l(A) = \min_c (Satisfaction_c^l(A))$

The allocation process consists of allocation algorithms that allocate the objects across the hosts and across the processes. The result of this allocation is what we call a *configuration*. As will be shown in Chapter 4, the resulting configurations depend on the population level and on the population mixes; in the extreme case, each population level or mix may generate a distinct configuration. A high-level description of a configuration generator is given in the Figure 6⁵ and Figure 7.

⁵ HEURISTIC_FAIR minimizes the maximum demand of a class at bottleneck(s) and is presented in Chapter 4.

Step 3.1 Determine configurations at high population level

Input: LQM, topology, performance requirements, design constraints

Output: one configuration and its satisfaction

Get an initial configuration with the HEURISTIC_FAIR algorithm

REPEAT

- Find R_{worst}^c and compute the satisfaction
- Change the configuration by applying a heuristic(N^{high})

UNTIL satisfaction is 1 or it can not be improved

Figure 6. Step 3.1.

Step 3.2 Determine configurations at medium population level

Input: per class object demands, topology, per class inter-object visits;

Output: one configuration and its satisfaction

Get an initial configuration (from step 3.1);

REPEAT

- Find R_{worst}^c and compute the satisfaction
- Change the configuration by applying a heuristic(N^{medium}, b_i)

UNTIL satisfaction is 1 or it cannot be improved

Step 3.3 REPEAT steps 3.1 and 3.2 for each remaining topology.

Figure 7. Steps 3.2 and 3.3.

1.2.4 Step 4 - Choose a Configuration that Best Satisfies the Performance Requirements

Each configuration that we determined in Step 3 is for a specific topology and population level. Designers could also propose other potential configurations, and it may be necessary to select only one configuration or a small set of configurations that covers all workloads. To do so, the whole set of configurations is compared against the whole set of performance requirements, and the configurations are ordered by following a multi-

attribute decision-making technique. The designer then selects one configuration or a small set to be used in the next phase of the general framework.

The population levels (and eventually the population mix) are separable criteria for our decision-making process, and the corresponding configurations are *alternatives*. Each of the criteria may be weighted to reflect its importance or its probability of occurrence. The weight can come from different sources:

- The designer's estimation of criterion's importance, or
- The measurement of workloads for similar existing systems, to give the most likely operating conditions.

Also, the weights may be expressed in the following different ways:

- As marks between 0 and 1, with 1 indicating a very important criterion and 0 indicating a totally unimportant criterion;
- Probabilistic measures of occurrences; for example, a probability density function;
- Fuzzy estimation of criteria importance such as "important," "very important," and "not important."

The resulting configuration(s) chosen by Step 4 is (are) then completed by computing the multiplicity level (threading or replica level) of the objects. This is done by Step 5 with the goal of reducing software queuing delays.

1.2.5 Step 5 - Determine the Object Order of Multiplicity

When generating configurations, we set the replication of a server (the modeling artifact of LQM) to the following:

- Total number of customers, if it contains objects that have no design constraints (utilization limits);

- The constrained level, if it contains an object that has a utilization limit.

Setting the threading/replica level to the maximum number of customers is a gross overestimate (as will be shown in Chapter 3). A quantitative method to determine the proper level is proposed. This method computes object's order of multiplicity. An object's order of multiplicity is found by computing its *maximum object utilization*, defined as that object utilization beyond which no improvement in response time is perceived under the design constraints. Finding the maximum utilization of an object may be mapped into a non-linear program, with design or physical utilization limits as constraints and with object utilization as an objective function. In the presence of LQM assumptions, this problem has interesting properties, which are explored in Chapter 3. The algorithm that searches for the maximum object utilization is given in Figure 8^{4,5}:

<p><i>Step 5</i> Determine object threading/replica levels Input: Configuration(s), LQM, design constraints (b_i, N_{max}) Output: Threading/replica levels for each object. IF (Asymptotic case) <i>Step 5.1</i> Apply LP_ASYM⁴(N_{max}) ELSE <i>Step 5.2</i> Apply NLP_NASYM⁵(b_i)</p>
--

Figure 8. Step 5.

The result of applying Steps 1–5 is a configuration or small set of configurations that best satisfies the performance requirements under the design constraints, under all possible population mixes.

⁴ LP_ASYM is defined in Chapter 3. It finds the maximum object utilization by combining linear programming techniques with the performance modeling and asymptotic bound analysis.

⁵ NLP_NASYM is defined in Chapter 3. It finds the maximum object utilization by combining non-linear programming techniques with the performance modeling and hill climbing methods.

1.3 Background and Related Work in the Literature

*If I have seen further it is by standing on
the shoulders of the giants.*

- Isaac Newton

The framework of Section 1.2 is based on fundamental results from different fields of both software engineering and more general sciences: performance modeling, linear and non-linear optimization techniques, fuzzy logic, and decision-making. The following subsections provide a review of relevant works in these fields as they pertain to our application domain.

1.3.1 Performance Evaluation Techniques

There are many performance-evaluation techniques that can be applied when studying software systems. Analytic techniques include those based on Markov models [Oliv93] and mean value analysis (MVA) [Rei80]. In general, the Markov-based models are the most expressive but suffer from state space explosion. They are most useful for studying detailed interactions between concurrent entities, but it is usually not possible to apply them to complex systems. Approximate MVA techniques and their corresponding Queuing Network Models (QNMs) do not enumerate state spaces to predict a system's performance behaviour, so they provide for more efficient solutions. Unfortunately, they do not consider contention for resources such as critical sections and software servers.

Several MVA techniques [Sou92][Cha82] have been extended to reflect contention for software in [Rol95]. Other techniques have been developed as well [Woo95]. Contention for software introduces layered queuing into a model. It takes into account the fact that a software server can block itself when requesting a service from another server. The

resulting *QNM*s are referred to as *Layered Queueing Models (LQM)*. A third alternative for performance evaluation is simulation. In general, simulation takes orders of magnitude longer to provide statistically significant performance measures than does analysis based on mean values.

Regarding the work done in this thesis, namely building performance models, Smith [Smi90] describes techniques to capture a model's structure by hand, based on a design. The advantage of this approach is that a skilled analyst can quickly focus on those features believed to affect performance most. However, constructing performance models by hand is a time-consuming and error-prone task. Furthermore, it is a major challenge to ensure that the model remains in step with an evolving design. An alternative approach was proposed by Hrischuk et al. [Hri95]. Their approach exploited a design emulation environment that could emulate software designs. The run-time of the emulator was instrumented to generate a trace of method invocations, and the trace was then processed to give the model's structure. As a design under study evolved, it was necessary only to recapture the traces in order to generate the corresponding model's new structure. Both Smith and Hrischuk et al. associate the method invocations with functions that estimate their CPU and disk requirements. ElRayes et al. [ElR98] also present a technique for building LQMs that further automate the process.

In the approach taken in this thesis, we assume that the code we aim to distribute exists in either prototype (partial implementation) form or final form, but is not yet distributed. This allows us to exploit the component-oriented design of the distributed applications. As a result, the next steps of the framework can be used before the deployment of the distributed application. The component code is executed to capture both the model structure and accurate estimates of resource demands on a reference processor.

Basic Notations. At this point, we introduce some of the notations used in the models described in the thesis. Unless stated otherwise, sets are denoted by bold upper case

characters, and their cardinality is denoted by the same upper case characters in italics. An element of the set will be a lower case letter; when possible, the same letter that denotes the set. Upper case bold characters also denote vectors and matrices. The distinction between vectors, matrices and sets will be apparent from the context.

$\mathbf{P} = \{1, 2, \dots, P\}$: the set of process indices;

$\mathbf{C} = \{1, 2, \dots, C\}$: the set of class indices;

$\mathbf{K} = \{1, 2, \dots, K\}$: the set of device indices, for CPU, networks, and I/O subsystems;

$\mathbf{O} = \{1, 2, \dots, O\}$: the set of object indices;

D_{kc}^o : the demand of object o for the device k due to class c (a demand is the product of the number of visits to the device and the service time per visit);

D_k : the total demand of the class c for the device k ;

$\mathbf{N} = (N_1, N_2, \dots, N_C)$: the class population vector; N_i is the population of class i ;

$\beta_c = N_c / N$, $c = 1, \dots, C$: the population mixes; N is the total customer population;

$\beta = (\beta_1, \beta_2, \dots, \beta_C)$: the population mix vector.

In this thesis, we use closed models to describe systems with a known number of customers that circulate through the system. The performance-evaluation techniques for the models rely on MVA and, as a result, rely on the *product form network* (defined in the next paragraph). We note that, in general, the models do not fully satisfy the product form requirements. However, heuristics are used to overcome the limitations of the product form networks.

Product Form Networks. A *product form network* [Jai91] is composed of service centres (CPU, networks, file systems, and others) visited by classes and includes the following constraints:

- The device service times are exponentially distributed for the first come first served scheduling discipline or generally distributed for other scheduling disciplines (processor sharing, last come last served, delay);
- There are no losses; i.e for each class, the arrivals at service centres equal the departures; and,
- There are fixed routing probabilities between service centres;
- The service rate for a class at a device can be fixed or can depend on the total or per class queue lengths at that device;
- The rate of external arrivals of customers is independent of the number of the customers in the system;
- There are no bulk arrivals.

Fundamental Results for Product Form Networks. Utilization of device i in class c , U_{ic} , is defined as:

$$E 1 \quad U_{ic} = \frac{N_c D_{ic}}{\sum_j D_{jc} [1 + Q_j(N - 1_c)]}$$

where $Q_j(N - 1_c)$ the queue length at device j with one customer of class c removed. The queue length at device i of class c (Q_{ic}) is defined as:

$$E 2 \quad Q_{ic}(N) = \frac{N_c D_{ic}}{\sum_j D_{jc} [1 + Q_j(N - 1_c)]} [1 + Q_i(N - 1_c)]$$

where $Q_j(N - 1_c)$ is the total queue length at device j ; i.e.:

$$E 3 \quad Q_j(N - 1_c) = \sum_{c=1}^C Q_{jc}(N - 1_c)$$

The mean per class response time is:

$$\text{E 4} \quad R_c(\mathbf{N}) = \sum_{i=1}^K D_{ic} [1 + Q_i(\mathbf{N} - \mathbf{1}_c)]$$

The *throughput* of class c , as the number of completions of class c per time unit, is:

$$\text{E 5} \quad X_c = N_c / (R_c + Z_c)$$

where Z_c is the think time of class c . The *utilization law* is:

$$\text{E 6} \quad U_{ic} = X_c D_{ic}$$

The total *throughput* is defined as:

$$\text{E 7} \quad \sum_{c=1}^C X_c$$

A *bottleneck* is any device i that satisfies the relation:

$$\text{E 8} \quad \lim_{N \rightarrow \infty} U_i(\mathbf{N}) = \lim_{N \rightarrow \infty} \sum_{c=1}^C U_{ic}(\mathbf{N}) = 1$$

MVA. The main advantage of the product form assumption is that it allows for efficient algorithms. Mean Value Analysis (MVA) is a recursive technique for computing E 1 to E 7. Approximate MVA techniques [Sou98][Cha82] are iterative techniques that estimate the results of MVA. They are more efficient than MVA, and have heuristics that predict system behaviour even when product form assumption are violated. MOL relies on the approximate MVA and heuristics. These issues are not discussed further in this thesis.

ABA. Asymptotic bound analysis (ABA) is a technique that can be applied to product-form networks. This technique estimates bounds for E 1 to E 7, when E 8 holds. Initially developed for single-class systems, it has been extended to multi-class closed systems [Bal97], [Bal96], [Bal91], [Bal90], [Lue97], [Maj97], [Nei95]. ABA is described in more detail in Chapters 3 and 4. The conditions (class population vector) under which E 8 holds

(there is a bottleneck saturated) are referred to in this thesis as *asymptotic conditions*, while other conditions are referred to as *non-asymptotic conditions*.

1.3.2 Object Allocation

In general, allocation strategies are closely related to scheduling policies. While the latter determine how the CPU and I/O resources are allocated to the processes that are resident on the node, the former determine how to allocate the load across the nodes of a distributed system. Scheduling policies are beyond the scope of this thesis. The reader is referred to [Rie96] for a recent classification of different scheduling and allocation strategies.

The allocation strategies may be grouped into two areas:

- **Dynamic allocation** - the allocation is partially done during run-time by process migration in order to face a variable and unpredictable workload. There are two techniques here:
 - load balancing - the algorithms seek to balance the work among the nodes; and
 - load sharing - the algorithms make sure that there is no idle node in the system.
- **Static allocation** - the allocation is done once, during the system implementation phase, and is based on static assumptions about the workloads and perturbations as well as an exhaustive knowledge about system's parameter distribution and object interactions.

Dynamic allocation may do very well for dynamic workloads, and many papers published in the area deal with this problem [Eag86][Zou88][Lit98][Shi89][Wan85]. Assumptions that simplify the models (e.g. zero overhead for process migration), as well as some practical implementation issues that affect the ability to move objects, make dynamic

allocation difficult to apply in practice. However, portability and platform independence of new technologies such as mobile agents, Java applets and servlets increase the viability of the strategy. Recent results [Ros91][Li98][Lu96] address dynamic allocation and scheduling in multi-class open distributed systems.

Static allocation, in the form in which it is presented today, does not respond to the performance requirements of the distributed systems in a natural way either. That is because it is either too qualitative or too rigid for effective system management. Additionally, traditional static allocation optimizes performance metrics that do not take into account user psychology or the complex workloads that a distributed application may face. When the complexity of the system is low, it is possible to establish a configuration for an application that seems to satisfy its performance requirements. One can do that by considering qualitatively the performance of object placement across the processes and of the processes across the nodes. To make things more practical, some authors have established guidelines or principles for “clustering” objects that respond to both functional and performance requirements. These guidelines are intuitive and try to make use of common sense, and consequently they work well as heuristics for simple systems. As an example, Champeaux [Cha93] suggests the following heuristics:

- *Forced*: identify processes with objects as mandated in non-functional requirement documents;
- *Functional*: identify processes with coarse-grained objects identified in analysis;
- *Structural*: identify processes with objects that are easy to isolate;
- *Service-based*: isolate objects that perform well-known generic services in their own processes;
- *Concrete-class*: allocate objects of a particular concrete class to the same cluster;
- *Link-based*: partition clusters so that as many objects links as possible point to objects residing in the same cluster;
- *Communication-based*: allocate heavily interacting objects to the same cluster;

- *Computation-based*: place any two objects that require large amounts of computational time in different clusters.

Other authors [Smi90][Loo98] suggest:

- *Efficiency principle*: reduce the ratio of overhead to useful work;
- *Locality principle*: place objects as close as possible to their functional sites.

In [Cha93] the guidelines are presented as imperatives. We claim that guidelines are heuristics, some of which may mislead the designer and some of which are too vague to allow the designer to apply them. For example, the first six rules of Champeaux are strongly related to analysis and design, and hence do not claim to address performance issues. The last two make quantitative assumptions, and we claim that they are hard to apply to complex systems without having a performance analysis tool. When the component interaction is complex or the number of components is high or the performance requirements are complex and involve many types of services, a system's behaviour is not intuitive, and such clustering rules do not necessarily yield high-performance systems. Heuristics may give you good performance in some circumstances but very poor performance in others.

There are quantitative static allocation approaches that determine the placement of objects across the nodes. However, they cover only simple models and low population areas⁷. Most of them have been developed for homogeneous, tightly coupled systems, and do not take into account queuing delays and variable workload conditions. Thus, at a low population level, there are exact algorithms [Lo88], heuristics such as clustering [Pri90][Zhu97], simulated annealing [Kir83, Kra86], tabular search and genetic algorithms. Static allocation by using simulated annealing techniques in the presence of

⁷ Low population area includes application from the high performance computing discipline. For a comparison of high performance computing and distributed systems, the reader may refer to [Loo98], the Introduction chapter, page xxiv.

queuing delays, for closed models, fixed population levels and fixed population mixes can be found in [Jog98].

Although static allocation itself does not ensure enough margin for a system's performance, it may be improved considerably by taking into account, at design time, a set of representative workloads, and conducting the allocation such that the system performance remains within acceptable margins for those workloads.

The object allocation approach presented in this thesis combines ideas from static and dynamic allocations. From dynamic allocation, we borrow the idea that the allocation depends on the workload and that many allocations should be considered during the life of a system. A goal of this thesis is to find a small set of configurations that covers many workload conditions. Management systems may help to commute between these configurations, which is simpler (from software engineering perspective) than attempting a full dynamic allocation strategy. However, we analyze the configurations by means of static allocation techniques. A given static allocation is checked against a set of possible workloads and a complex performance metric (*Satisfaction*) by using its corresponding Layered Queuing Model (LQM and MOL).

We consider workloads with two dimensions: population level (N) and population mix (β_i). We remind the reader that the first denotes the total number of customers in the system, while the latter is given by the ratio between the number of customers of a class over the total customers in the system ($\beta_i = N_i/N$, $i=1, C$; i denotes the class and C denotes the number of classes). The details of the approach presented in this thesis are different from those of other dynamic allocation techniques, since we consider closed models as opposed to other approaches [Ros91][Li98][Lu96] that consider open models. Also, the performance metric is different: we use *Satisfaction*, while most of the dynamic approaches use average response time over all workloads. Also, we take into account

design constraints such as device utilization limits and the impact of software queuing delays (all those supported by LQMs).

1.3.3 Handling Complex Performance Requirements and Multiple Design Alternatives

Some work in this area covers multimedia distributed systems, where the user quality of services is mapped into network quality of service [Ste97][Nis97], and the hard real time systems [Lit97, Lit98-1, Lit99]. We are not aware of any literature for transactional distributed systems.

Multi-criteria decision-making is an emerging theory that bases its outcomes on many attributes of the decision process [Ign94]. It also handles uncertainty [Lai94]. A decision-making problem may be defined as $\Gamma = \{A, G, Z\}$, where:

- A is the set of alternatives to be analyzed. The alternatives in our case are the configurations, so we use the terms interchangeably;
- G is a definition of the decision-making problem, which specifies the type of the final result that is expected to be obtained. There are many possible values for G :
 - *choice*, which determines the best alternative;
 - *classification*, which partitions the set of alternatives into sets with predefined profiles (good, acceptable, bad, and others);
 - *sorting* which orders the alternatives from best to worst;
 - *description*, which supplies a description of the alternatives; and,
 - *conceptualization*, which identifies the ideal alternative, not yet present among the alternatives that we scrutinize.
- $Z = \langle A^*, L, E, r(), P, R \rangle$, where
 - $A^* \subseteq A$ is a subset of A and contains those alternatives that are independent;

- L is a set of independent and coherent criteria. Each criterion L_j specifies our case a workload condition, but it can be extended to handle non-performance criteria as well;
- E is a scale associated with L and represents the preference over alternatives (configurations). The scale may be ordinal, interval, relative, or absolute and, in the general case, each criterion may have its own scale. The scale may be obtained by measurement or by reasoning, depending on the type of criteria (measurable or non-measurable);
- $r_i(a_j)$ is the value awarded to $a_j \in A$ by criterion $L_j \in L$. Information about E , L and A^* should be processed together to obtain $r(\cdot)$;
- P is a preference structure that the decision-maker applies to the set of alternatives;
- R is an aggregation function that processes the decision problem data and yields a solution for G .

As we will see in Chapter 4, the most difficult problem to solve is the rating of alternatives against different criteria. To solve this problem, we rely on fuzzy logic. Fuzzy theory has evolved since the 1970's. First introduced by Zadeh [Zad65], fuzzy theory is well known in many fields of science for its ability to model and control complex systems at low cost [Ped93] under uncertainty. This thesis makes particular use of fuzzy logic. An overview of basic concepts of this theory is presented in Appendix 1. The approach is justified by the “fuzzy” appearance of performance requirements. The performance requirements resemble the fuzzy sets with the *Satisfaction* function as membership function and the response time as support. The author is not aware of any other work that combines all these four domains together: performance requirements, system configurations, decision-making and fuzzy logic.

1.3.4 Object Replication

Object replication and threading levels are instruments aimed at improving the performance of a distributed system. Replication of an object on the same host reduces only the software queuing delays, whereas replication on different hosts reduces both software and hardware queuing delays. In this work, we refer to replication on the same host. The work in this area is based on object utilization.

The early results in this area were based on ABA [Bal96]. Hills et al. [Hil95] estimate maximum object utilization under asymptotic conditions with the assumptions of no device or software queuing delays. That is, the object response time, and hence the utilization, was underestimated. However, the methodology used was correct. In the presence of one or more saturated devices in the system, throughputs become constant and are determined by the saturated devices. Response times become linear with respect to the customer population in the system. Since the object utilization components at the bottleneck devices become dominant, the total utilization of an object in the presence of the saturated devices tends to become linear. However, we are interested in studying systems under non-asymptotic conditions and hence non-linear conditions as well. When queuing is taken into account, the waiting times at devices and software entities are non-zero. Furthermore, they are non-linear, with increasing number of customers submitting requests. This causes object utilization to be non-linear with respect to the total population as well.

1.4 Main Contributions of this Thesis

The main contributions of the thesis in the field of performance engineering are summarized as follows:

- The framework as a whole; a unified suite of algorithms is provided to help designers build high-performance distributed applications. The framework covers the design and development cycle of a distributed application;
- Investigation of automatic performance model building from Executable Trace Files. The approach presented in this thesis takes the tracing one step further than other authors, by using a tracing tool that captures both method invocations and their resource demands. We assume that the code we aim to distribute exists in either prototype (partial implementation) form or final form (achievable when reusable components are used). The code can be executed to capture both the model structure and an accurate estimate of resource demands on a reference processor;
- Object allocation under very high, high and medium population levels in multi-class distributed systems. We show that the optimal configurations depend on the total population and on the population mixes:
 - At very high population levels, beyond a specific population, it is always worthwhile to balance per class load across the devices. At this population level, ABA is used. We distinguish between single class systems, multiple classes with a common bottleneck and multiple classes with multiple bottlenecks. For all three types of systems, properties of the response time are derived. Based on these properties, heuristic algorithms are developed for distributing the objects across the nodes and then across the processes;
 - At medium population levels and at high population levels but under a specific population level, we show that the maximum per class response time under the design constraints is at certain points. We will call these points *extreme points* and *crossover points* whose coordinates can be computed;
 - Based on the above computed points, algorithms to navigate through the configurations space are provided. Within the same configuration, we provide exact algorithms for finding the maximum per class response time by combining linear and non-linear programming techniques with performance modeling;

- Results are presented using a *Satisfaction* cost function as well.
- Methods are given for computing the maximum object threading level or number of replicas. We illustrate the properties of object utilization with respect to the per class device utilization and with respect to class population:
 - We show that maximum object utilization under design or physical constraints can be modeled as a non-linear programming problem, with per class device utilizations as control variables and object utilization as an objective function;
 - Algorithms for solving the problem are then developed. They resemble classic search algorithms, but differ from them in the way in which the objective function is computed. Specifically, the object utilization is found by using an LQM and by iterating between per class device utilization space and the population mix;
 - The algorithms are developed for two cases: asymptotic and non-asymptotic conditions. We show that the points in the utilization where more than one device reaches its utilization limits (extreme points or vertices) have interesting properties, and we extend the work carried out by Balbo et al. [Bal97],[Bal96], [Bal91], [Bal90], to an arbitrary number of devices and classes and non-asymptotic conditions.
- Define the mathematical framework for building a configuration that best satisfies the design's overall performance requirements. We assume that the performance requirements are defined to cover a large spectrum of workloads and may be expressed as intervals of preferences. To solve the problem, we model it as a multi-criteria decision problem and solve it with the help of fuzzy logic.

Some of the ideas presented in this thesis have been already published by the author in [Lit99-2], [Lit97-1], [Lit98-2], [Lit98-3], [Kris99]. Other results (especially those presented in Chapter 5) are based on previous work of the author in [Lit97], [Lit98], [Lit98-1], [Lit99], [Lit99-1].

The reminder of the thesis is organized as follows:

Models, sources of information and automatic model building are covered in Chapter 2. Chapter 3 discusses the choice of object threading/replica level, and Chapter 4 discusses object allocation under high population levels. A case study is offered in Chapter 5. The last chapter summarizes the thesis and presents the final conclusions.

As result of the heterogeneous character of this work, the reader should expect a heterogeneous mathematical apparatus: queuing theory (familiar to performance engineers), non-linear programming, fuzzy logic, and decision theory. The appendices describe the mathematical apparatus.

2 PERFORMANCE MODELS FOR DISTRIBUTED APPLICATIONS

You know my method, it is based on the observance of trifles.
-Sir Arthur Conan Doyle

2.1 Introduction

Obtaining a performance model is an essential step in our framework, since the accuracy of the later steps depends upon it, as shown in Figure 2. Figure 9 shows the framework narrowed to the scope of this chapter.

Figure 9. Model builder.

Developing an accurate predictive performance model requires the capture of inter-object visits and the object demands at hardware devices, knowledge of the types of device scheduling, and more. It is a laborious activity. Building a performance model is not the main focus of the thesis; however, the work presented in this chapter offers an effective

solution for capturing resource demands and objects interactions as long the code resides within a single process. This is a very likely situation in the early stages of the design and development.

The work is based on tracing the prototype or component code using a tracing tool. The tracing tool is named EXTRA (Executable Trace Files), and it is commercially available as part of IBM's Visual Age C++ product on several platforms, including Windows NT, OS/2, AIX, and MVS. We note that the work described exploits trace file formats that are not publicly available interfaces.

The EXTRA file contains a detailed timing record for each function (call/return), each object method (call/return) and each thread (start/end). By parsing the file, we can detect the method interactions and compute the device service demands. The time stamps associated with each traced event have a high resolution (the measurements we present were made on an Intel-based OS/2 node with a timestamp resolution of 838 nanoseconds).

The performance model that we build as a result of tracing is a Layered Queuing Model (LQM). This is the subject of the next section. Section 2.3 presents an example of a tracing procedure and describes how we map the tracing scenarios in LQM. Section 2.4 presents a summary of the chapter.

2.2 A Target Model - LQM

LQMs are an extension of Queueing Network Models (QNMs) that take into account both hardware and software resource contention. QNMs consider only hardware resources whereas an LQM is a collection of software servers (processors, critical

sections, Inter-Processor Communication (IPC) paradigms) and devices (node and networking elements) that interact:

- Devices are characterized by:
 - a scheduling policy (Processor sharing, First come first served, Delay, and others described in the MVA literature [Ro195]); and,
 - a service rate.
- Software servers are characterized as follows:
 - each has a replication level to give its number of instances, and a threading level to give the maximum number of requests that can be served in parallel;
 - objects are allocated to software servers, and each object method becomes an entry in the corresponding server;
 - the average number of visits between the methods of objects becomes the average number of visits between the corresponding entries of servers.
- Each entry of each server is characterized by:
 - the average number of visits to each device and the average service demand.

The Method of Layers (MOL) [Ro195] takes the LQM as input and provides the following measures:

- For each server and each entry:
 - the average queuing delay at each device and at each entry of other servers;
 - the average utilization of each device and of each entry of other servers; and,
 - the average response time of each entry.
- For each device:
 - the total utilization; and,
 - the total average queue length.

At design time it is not possible to automatically build all the structure of the model. By using EXTRA, we are able to partially generate the structure of the model. A main limitation is that the tracer is able to trace only one process, so it is not possible to capture directly the IPC paradigms, nor it is possible to have the critical sections automatically detected. It is the responsibility of the designer to reflect them in the model. Other techniques for automating the construction of LQMs are presented in [EIR98].

The following sections illustrate one technique for building LQMs and give by example a more complete explanation of the models used in this thesis.

2.3 Using EXTRA to Characterize Scenarios and Model Structure

A scenario is an artifact used during systems analysis and design to ensure that the proposed designs will satisfy design requirements. A scenario describes a specific request that a system must support, and gives its corresponding path through the methods of objects in the system. We assume that other design techniques are used to help decide on a set of objects and scenarios. We focus on the key scenarios that are expected to affect performance most.

The corresponding performance model is closely tied to the scenarios. As a request passes through the methods of objects, it consumes both hardware and software resources. The hardware resources include:

- CPU demand in milliseconds; and,
- Disk demand in visits. The type of disk gives a mean service time per visit in milliseconds.

The software resources include:

- Critical sections, in terms of visits and the software and hardware demands nested within the critical section; and,

- Contention for access to software servers, possibly with limited threading levels.

For model building, we capture the demands of each scenario as it passes through the methods. A method invocation between objects in different processes causes a remote procedure call (RPC) that relies on middleware and possibly networking protocols.

An EXTRA Example. As an example, consider a prototype for a hypothetical personnel application. This application has three key scenarios that are executed most frequently and are expected to dominate performance. The scenarios are:

- A *query* request that returns information about an employee;
- An *update* request that modifies some fields of a record in the database; and,
- A *delete* request that deletes a record in the database.

By tracing a *query* request scenario, we obtain the timing diagram shown in Figure 10a. From the trace, we observe invocations of the methods of three objects: **Interface**, **Employee** and **Database**. In general, subsequent instances of the same scenario could follow the same trace pattern or, because of data dependencies, they could follow different paths. Furthermore, the differences between time stamps could be different due to processor caching or branching conditions tested by the methods. In the figure, the dotted line illustrates a second traversal of the same scenario that follows the same path but has slightly different service times within the methods.

Two other issues are also of interest. First, EXTRA's instrumentation introduces overheads that affect the time stamps for function and method entry/exit, thread context switching, and trace buffer flushing. Fortunately, EXTRA estimates these overheads, so we post-process the time stamps to diminish the impact of the instrumentation on them. A second issue comes from our measure of CPU time. The CPU time for a method is estimated as the difference between the time stamps for method entry and exit, less any

time spent at other methods. If the process being measured has a context switch due to other work on the measurement node, it can distort our measure.

Figure 10. A trace of the query request (a) and a performance scenario (b): bold face type indicates method interactions; the rest indicates average resource demands or average number of visits.

To reduce the impact of these problems, we apply the following steps: trace on a lightly loaded node, repeat the characterization of scenarios with the same initial conditions, and report the average measures. The later steps are automated. Figure 10b) presents a *performance scenario* that averages the behaviour of the program over 500 query record invocations with different initial parameters. We note that the performance scenario is

quite different from the results of the single invocation. The average number of visits from **Employee::queryRec(long,long)** to **DataBase::getRecByNum(long)** is 2.61, not 1, as illustrated in Figure 10a. This is because the query record behaves differently depending on whether it is invoked by a manager or by a non-manager. A fraction of the requests behave as manager requests, while the rest behave as non-manager requests. The ratio was chosen based on design documents, and could later become a factor in a performance study. The reported CPU and IO service time values are averages.

Figure 11. Mapping a performance scenario in a LQM.

Performance scenarios act as the intermediate language between the design and performance models. They give our object interactions and resource demands. To map a performance scenario onto an LQM, we need further information that must be provided by the analyst (or expert system) that is conducting the performance engineering exercise. The objects must be encapsulated into processes, and the processes mapped onto nodes in a target environment. Figure 11 shows how the software interactions are mapped onto an LQM. Parallelograms represent processes, and circles represent devices. The methods of objects become entries in the processes, and make visits to other entries and devices. Arcs are labeled with the average number of visits.

The number of visits to the CPU by the entry that contains **Interface::getRec(long,long)** entry is 3. This is the sum of the number of visits of **Interface::getRec(long,long)** to methods in other entries, non-CPU devices, and a think centre, plus one additional visit to the CPU. The additional visit takes into account the fact that the method will alternate between the CPU and the other resources. The mean service time at the CPU is chosen based on the method's total CPU time and the number of visits.

Table 1. Scenario parameters.

Scenario	Method (Caller)	Method(Callee)	Number of Visits	(CPU, I/O) service time (ms)
Query	External	Interface::getRec	1	(91.711, 0)
	Interface::getRec	Employee::queryRec	1	(150.0, 0)
	Employee::queryRec	DataBase::getRec	2.61	(51.77, 222.931)
Update	External	Interface::updateRec	1	(21.581, 0)
	Interface::updateRec	Employee::updateRec	1	(150, 0)
	Employee::updateRec	DataBase::addRec	0.01	(336.24, 1344.64)
	Employee::updateRec	DataBase::updRec	0.09	(4.456, 0)
	Employee::updateRec	DataBase::getRecByNum	1	(35.10, 140.283)
Delete	External	Interface::deleteRec	1	(21.460, 0)
	Interface::deleteRec	Employee::deleteRec	1	(152.1, 0)
	Employee::deleteRec	DataBase::delRecByNum	1	(50.87, 203.517)
	Employee::deleteRec	DataBase::getRecByNum	2.40	(106.19, 424.76)

Table 1 documents all three scenarios of the personnel application.

2.4 Summary

In this chapter, we presented the first step of a framework for integrating distributed application design and configuration with performance modeling. Performance scenarios are captured using a tracing tool named EXTRA, which automates the steps needed to describe a model's structure and estimates the resource demands for the methods of objects. The main advantages of this approach are flexibility, accuracy of data sampling, and completeness.

A case study was presented to illustrate the method. It examined structural information and device demands.

3 CHOOSING OBJECT THREADING/REPLICA LEVEL AND ACTIVATION POLICY - A QUANTITATIVE APPROACH

*Entities should not be multiplied unnecessarily. No more things
should be presumed to exist than are absolutely necessary.*

William of Okham (English philosopher, 1280-1349)

3.1 Introduction

The *utilization* of a process or object is the percentage of time that it is busy, and can be interpreted as the average number of instances of the process or object. The utilization can be greater than one for replicated processes/objects and/or multi-threaded processes.

The purpose of this chapter is to give estimates for *the maximum utilization* of a process or object and, based on that, to find the replication or threading levels required for that process or object. Replications beyond this point do not improve user response time. Several algorithms are presented. The algorithms consider the required population levels and all population mixes at those levels, and take into account queuing delays at software components (processes, critical sections, IPC, and others) and devices. To compute object utilization using LQMs and MOL, we consider that each object is encapsulated into one process. If we choose to implement more than one object into a multi-threaded process, the utilization of the process is the sum of the enclosed object utilizations. The processes of finding the number of threads or the number of replicas are treated in the same manner. The difference in the overhead is reflected in the LQM.

Step 5 (Chapter 3): Find the Threading Level for the Chosen Configuration

Figure 12. Object replication: LP-linear programming, ABA-asymptotic bound analysis, NLP-non-linear programming; Straight - algorithm for computing all bottlenecks.

Figure 12 gives an overview of the elements used by the algorithms. The outline of the chapter is as follows:

- Section 3.2 describes the problem of determining maximum object utilization as a Non Linear Programming (NLP) model and reducing the size of the programming model by exploiting the utilization law. The model is valid for both asymptotic and non-asymptotic conditions.
- The properties of the object utilization in extreme and crossover points of the programming model built in Section 3.2 are given in Section 3.3. *Extreme points* identify the potential bottlenecks in the system, while *crossover points* give the population mixes that cause the change of the bottlenecks. The properties illustrated in the section help to implement an algorithm that finds all extreme and crossover points.
- Section 3.4 describes a suite of algorithms, as illustrated in Figure 12, that integrate the model of Section 3.2 with LQMs and the MOL to discover an object's maximum utilization over all workload conditions:
 - Two algorithms use a convex Simplex-like algorithm that navigates through extreme points: in the non-asymptotic case, the search uses NLP

Simplex, while the asymptotic case uses Linear Programming (LP) Simplex combined with ABA;

- An alternative *Straight* algorithm for computing maximum object utilization is also presented. It uses all the crossover points computed by the algorithm developed in Section 3.3.
- Section 3.5 shows how the results can be interpreted and used for planning replication and activation policy for distributed application systems.
- A summary and a description of future work are given in Section 3.6.

3.2 Maximum Object Utilization in a Multi-Class Distributed System - A Non-Linear Programming Problem

3.2.1 Device and Object Utilization in Distributed Multi-class Systems

In this subsection, we define the concept of object utilization and explain the following properties of object and device utilization:

- The non-linearity of total device utilization with respect to per class population;
- The non-linearity of object utilization with respect to per class device utilization; and,
- The linearity of object utilization under asymptotic conditions with respect to the population or population mix.

The continuity of object utilization with respect to per class device utilization is a property that is obvious, and we do not focus on it. Instead, we want to point out qualitatively the strict monotone property and the convexity of object utilization with respect to per class device utilization under non-asymptotic conditions.

These properties⁸ assure us that the object utilization is a convex function, and henceforth we can apply the techniques associated with the NLP.

The next few paragraphs describe the object utilization qualitatively. Readers familiar with the concept may go directly to equation E 9.

We first should mention that the object does its job through devices or other software components. From a queuing perspective, an object is transparent unless it lacks enough replicas. It simply forwards the requests to other queuing centres. If there are fewer replicas than there are requests, the object itself becomes a request queuing centre.

Figure 13. Software entities utilization; a) one thread; b) two threads.

To elaborate on the above statements, let us consider an example where an object is embedded in a process with one replica and one thread and two simultaneous requests at the moment $t=0$. Figure 13a shows how the two requests are handled by the process. For

⁸ We add more quantitative justification for these properties in Appendix 2 and Chapter 4.

simplicity of presentation, we ignore the eventual preemption due to other processes running on the host. *Request 1* is handled first and, up to moment t_1 , CPU processing takes place. Then a disk activity is started within the same request, which ends at t_2 . Within $[t_2, t_3]$, a remote call is performed. At t_3 , the request ends. All this time, *Request 2* waits, although the CPU, the DISK, or both are idle for some periods. This is due to the synchronous behaviour of the request's calls. Even in the case of multiple operating system kernel threads, at t_1 , although the CPU is idle, the second request cannot be handled because the single threaded process is blocked. The same thing happens at t_2 , where both CPU and DISK are idle. *Request 2* processing starts at t_3 and, assuming that it is identical to *Request 1*, it will end at t_4 . A numerical example with $t_1 = 1$, $t_2 = 2$ and $t_3 = 4$ time units gives the response time and utilization of devices for the two requests in the case *a*:

$$R^a_1 = 4; R^a_2 = 8;$$

$$U^a_{CPU} = 2/8 = 0.25;$$

$$U^a_{DISK} = 2/8 = 0.25;$$

Figure 13b shows the same requests but with two threads. At t_2 , *Request 2* is handled by the CPU, at t_3 by the DISK, and then by the remote host. Assuming that the CPU and DISK demands at the remote hosts are identical to those of the local host, it turns out that, for the data defined above, we have:

$$R^b_1 = 4; R^b_2 = 5;$$

$$U^b_{CPU} = 2/6 = 0.33 \text{ and } U^b_{DISK} = 2/6 = 0.33,$$

which means a reduction of 60% in *Request 2* mean response time.

The utilization of a process measures the total busy time of the process, including the time it spends at local devices and the time waiting for returns from remote or local calls.

$$U_p^a = t_5 / t_5 = 1 \text{ and}$$

$$U_p^b = (2 * t_4 - t_1 - (t_4 - t_3)) / t_4 = 9/6 = 1.5$$

In the first case, the process utilization is 1, and this is also the process utilization limit. Hence, the process is a bottleneck. By raising the utilization limit to 2 (the second case), we free the requests that are waiting in the process queue and improve per request response time as well as the average response time; at the same time, the device utilizations increase. It may be seen from this example that an adequate number of replicas may improve the performance of the system. It suggests also that an upper bound for the number of replicas is the number of simultaneous requests through the system, which is less than or equal to the population level. Of course, this is far from being acceptable due to memory and resource consumption. Forwarding requests to other objects where no physical resources are available only introduces unnecessary overhead without reducing the request response time.

The above example may be generalized for an arbitrary number of threads and requests as well as for multiple classes. Each class c makes use of a collection of objects with methods that interact with one another. For each class c , the response time of an object includes its own direct demands and queuing delays at its node's devices plus its nested demands and queuing delays for access to the methods of the other objects that it visits synchronously.

The average response time of an object o for a request of class c , R_o^c , is given in E 9:

$$\text{E 9} \quad R_o^c = \sum_{i=1}^{K_o} (D_{io}^c + W_{io}^c) + \sum_{p \in O_o} V_{o,p}^c (R_p^c + W_p^c)$$

where:

- D_{io}^c is the mean object o demand at device i in within request class c ; and K_o is the number of devices on the object's node;

- W_{io}^c is the mean waiting time of object o at device i within request class c ;
- O_o is the set of objects visited synchronously by object o ;
- $V_{o,p}^c$ is the number of visits of object o to object p within request class c ;
- R_p^c is the mean response time of object p when servicing request class c ; and,
- W_p^c is the mean waiting time at object o when servicing requests from request class c .

Using the utilization law, object utilization is defined as:

$$E_{10} \quad U_o = \sum_{c=1}^C X_c R_o^c$$

where C is the number of classes and X_c is the throughput of class c .

An Example.

Figure 14. Architecture for a personnel application.

For a concrete example, let us consider a hypothetical personnel application. The architecture of the system is shown in Figure 14. Users interact with the application via a browser, which interprets Java applets that interact with the database on the Server node. For simplicity of presentation, we consider that the application has two requests that are submitted most frequently and are expected to dominate the system performance. These requests are:

- *query* request: a query returns information about an employee; and
- *update* request: an update modifies some fields of a personnel record.

Table 2. Total request demands and object demands per request, in milliseconds.

	Query class 1	Update class 2	Browser /query	Browser /update	Employee /query	Employee /update	Database /query	Database /update
CPU_Client Device 1	$D_{11}=20$	$D_{12}=10$	20	10	0	0	0	0
CPU_Server Dev ice 2	$D_{21}=50$	$D_{22}=70$	0	0	30	40	20	30
DISK_Server Device 3	$D_{31}=64$	$D_{32}=56$	0	0	0	56	64	0

The example considers three objects: *Browser* on the *Client Node*, and *Employee* and *Database* on the *Server Node*. Table 2 gives the total CPU and DISK demands of each request and the demands per object per request. The *query* and *update* requests are denoted as classes with subscripts 1 and 2, and *CPU_Client*, *CPU_Server* and *DISK_Server* as are denoted as devices with indices 1, 2 and 3, respectively.

Figure 15. Device utilization is a non-linear function with respect to population.

Figure 15 shows two request classes and two devices at the server side that give a non-linear utilization function at one of the devices. Figure 16 shows that utilization of object

Database is non-linear with respect to per class utilization of a device. The non-linear dependency is more evident with respect to U_{22} . We also note that the object utilization is convex and strictly monotonic with respect to the same variables.

Figure 16. Object utilization is a non-linear function, strictly monotonic and convex with respect to the device 2 per class utilization (non-asymptotic case): U_{21} and U_{22} are device 2 utilization in class 1 and 2, respectively.

Asymptotic analysis. In the presence of one or more saturated devices in the system, the response time component given by the saturated devices becomes linear with respect to the population level as long as the population mix remains constant:

$$E 11 \quad R_w^c = ND^c_b + D^c_b I_c$$

where N is the total population in the system, and the other terms are constant for a given population mix. Mathematical support for this statement is provided in Chapter 4.

However, when the population mix changes, the response time and throughput change and the object utilization are not likely to be linear. Although object utilization is not linear, the

non-linearity is not very accentuated, so only a small error is introduced by considering the object utilization linear under asymptotic conditions (see Figure 17).

It is also worth mentioning that object utilization in the presence of saturated devices may still increase if enough replicas are provided. Recalling the example described in Figure 13, the object forwards the requests to devices or other software entities. The saturated devices bound the throughput, but the object response time is not bounded; it includes the saturated device response time, which grows asymptotically. As result, the object utilization will increase, provided that enough replicas are available.

Figure 17. Object utilization under asymptotic conditions (with respect to population level) is a quasi-linear function.

Although object utilization can increase in the presence of saturated devices, it brings no advantage to overall system performance since the response time is not reduced nor is the throughput increased. On the contrary, by freeing a request that is waiting in a software entity queue, an extra overhead is introduced, which tends to be important when the replica is a process. This stresses once again the importance of computing the maximum

object utilization. By setting utilization limits on each device, we can find the maximum population level under which these limits are obeyed. At the same time, by finding the maximum object utilization, we can set the threading level such that the software entities queue the requests if the population level grows beyond the maximum.

To summarize, we showed that:

- The total object utilization is non-linear with respect to per class device utilization, strict monotone and convex;
- The total object utilization is potentially unbounded, provided that there are enough threads or replicas;
- A threading level lower than a maximum number creates software queuing delays and henceforth a larger response time and a lower throughput; and,
- In the presence of saturated devices, a threading level or a number of replicas higher than a maximum will yield a supplementary overhead such as higher scheduling costs or memory paging faults, which will diminish the overall performance of the system.

3.2.2 Posing Maximum Object Utilization as a Non-Linear Programming Problem

This section starts to map the search for the maximum utilization in a non-linear programming problem. We will identify the objective function and the constraints, and put them in a convenient form to allow for further processing.

Let us assume that no threading limit is imposed for an object o but that there are limits imposed on hardware device utilization or population. We have already shown that U_o is continuous and strictly increasing with respect to per class population as well as with the per class devices utilization. These two properties are very useful in the context of our work. If a function is defined on a convex space, then the monotone property will ensure that the minimum or maximum is reached on the boundary of its space definition.

Moreover, if U_o is linear, then the maximum is reached at so-called extreme points. If U_o is non-linear, the maximum is reached on the boundary of the feasible space. Figure 18 illustrates a convex space and an objective function U_o , with the maximum at D when the objective function is linear and at M when it is non-linear.

Figure 18. Utilization of object o on a convex space $x_1 \times x_2$: ABCD is a convex space; A, C, D are extreme points. A linear utilization must reach its maximum at an extreme point (D, for example) while a non-linear one may reach its maximum anywhere on the boundary (M, for example).

The total utilization of an object o is defined as U_o and may be decomposed into the sum of object per class utilizations.

$$U_o = U_{o1} + U_{o2} + \dots + U_{oc}$$

where U_{oi} is the utilization of o per class i .

We want to find the maximum utilization of the object o such that device utilizations are less than or equal to their specified bounds.

The problem can be expressed for an object o as a non-linear programming problem as follows:

$$\text{E 12} \quad \text{Max} \quad z = U_o = U_{o1} + U_{o2} + \dots + U_{oc}$$

subject to

$$U_1 = U_{11} + U_{12} + \dots + U_{1c} \leq b_1$$

$$U_2 = U_{21} + U_{22} + \dots + U_{2c} \leq b_2$$

....

$$U_K = U_{K1} + U_{K2} + \dots + U_{Kc} \leq b_K$$

$$U_{kc} \geq 0, \quad \forall k \in \mathbf{K}, \quad \forall c \in \mathbf{C}$$

where $\mathbf{C}$, $\mathbf{K}$, are the classes and devices, U_k is the total utilization of device k , U_{kc} is the utilization of device k by requests of class c , and b_k is the permitted utilization of device k .

The solution of this NLP model gives per class utilization of devices that cause the object utilization to reach its maximum within the feasible space imposed by the constraints. This is given by the solution $U^+ = (U_{11}^+, U_{21}^+, U_{31}^+, \dots, U_{12}^+, U_{22}^+, U_{32}^+, \dots)$. We now re-express the problem to introduce the object response time R_o^c into the NLP model. For the time being, we assume that the R_o^c for all the system's objects are known, but discuss their relationship to the NLP model in the next section.

Using the Utilization law, the utilization of object o by a class c is:

$$\text{E 13} \quad U_{oc} = X_c R_o^c \quad \text{and} \quad U_{kc} = X_c D_{kc} ; \quad o \in \mathbf{O}; \quad k \in \mathbf{K}$$

and the total utilization of object o is:

$$E\ 14 \quad U_o = X_1 R^1_o + X_2 R^2_o + \dots + X_C R^C_o$$

The total utilization of a device i is:

$$E\ 15 \quad U_i = X_1 D_{i1} + X_2 D_{i2} + \dots + X_C D_{iC}$$

Within a single class, the forced flow law [Laz84] gives the following relation among utilization and demands:

$$E\ 16 \quad U_{ic}/U_{jc} = D_{ic}/D_{jc} \quad \forall i, j \in \mathbf{K} \cup \mathbf{O}, \quad \forall c \in \mathbf{C}$$

Assuming that there exists a device shared by all classes (for example, a shared network), and taking this device as a reference, the relation E 16 is defined for all classes. If we consider device r as a shared reference device, we have:

$$E\ 17 \quad U_{jc} = U_{rc} D_{jc} / D_{rc} ; \quad \forall j \in \mathbf{K}; \quad \forall c \in \mathbf{C}; \text{ and}$$

$$E\ 18 \quad U_{oc} = U_{rc} R^c_o / D_{rc} ; \quad \forall o \in \mathbf{O}; \quad \forall c \in \mathbf{C}$$

Thus, using device r as reference, we can rewrite E 12 as:

$$E\ 19 \quad \text{Max} \quad z = U_o = U_{r1} R^1_o / D_{r1} + U_{r2} R^2_o / D_{r2} + \dots + U_{rC} R^C_o / D_{rC}$$

subject to

$$U_r = U_{r1} + U_{r2} + \dots + U_{rC} \leq b_r$$

$$U_1 = U_{r1} D_{11} / D_{r1} + U_{r2} D_{12} / D_{r2} + \dots + U_{rC} D_{1C} / D_{rC} \leq b_1$$

...

$$U_K = U_{r1} D_{K1} / D_{r1} + U_{r2} D_{K2} / D_{r2} + \dots + U_{rC} D_{KC} / D_{rC} \leq b_K$$

$$U_{r1}, U_{r2}, \dots, U_{rC} \geq 0$$

The solution of this problem gives $U^+ = (U_{r1}^+, U_{r2}^+, \dots, U_{rc}^+)$, from which we can deduce U^+ using E 17 and E 18.

Note that z is a non-linear function since R_o^c depends on $U_{r1}, \dots, U_{rc}$ in a non-linear manner. Non-linear programming techniques are required to solve E 19. The Convex Simplex-like algorithm [Baz93] can be used to solve for E 19 with $U_{r1}, \dots, U_{rc}$ as decision variables. Several properties of the object utilization with respect to the application of the algorithm are found in Appendix 1 and Appendix 2.

Going back to our example, assume that the maximum permitted utilization of the *CPU_Server* and *Disk_Server* are 0.8 and 0.9. For an object o , we want to find the maximum utilization of o such that the following constraints are satisfied with respect to the *CPU_Server*:

$$\begin{aligned}
 & \text{Max } U_{21} R_o^1 / 30 + U_{22} R_o^2 / 40 \\
 & \text{subject to} \\
 & \quad U_{21} + U_{22} \leq 0.8 \\
 & \quad 1.28U_{21} + 0.8 U_{22} \leq 0.9 \\
 & \quad U_{21}, U_{22} \geq 0
 \end{aligned}$$

Next we consider the use of slack variables needed to solve the programming model.

3.2.3 Slack Variables

The convex simplex method is based on the notion of slack variables. They measure the difference between the right hand side and the left hand side of the inequalities. For this problem domain, the slack variables represent the percentage of idle times of the devices. The number of slack variables equals the number of inequality constraints; in our case, K .

The insertion of slack variables is equivalent to the insertion of a new class $C+1$ in the model, and $U_{1\ C+1}, \dots, U_{K\ C+1}$ are the utilization of devices with respect to the new class. The slack throughput is defined as a function of the reference device demand and the slack utilization at that device:

$$X_{C+1} = U_{1\ C+1} / D_{1\ C+1} = U_{2\ C+1} / D_{2\ C+1} = \dots = U_{K\ C+1} / D_{K\ C+1}$$

Inserting the slack variables in our example adds a new column to the constraints, so we have:

$$\begin{aligned} \text{Max} \quad & U_{11} R^1 / 30 + U_{12} R^2 / 40 \\ \text{subject to} \quad & \\ & U_{21} + U_{22} + U_{23} = 0.8 \\ & 1.28 U_{21} + 0.8 U_{22} + U_{24} = 0.9 \\ & U_{21}, U_{22}, U_{23}, U_{24} \geq 0 \end{aligned}$$

3.2.4 Introducing other Design Constraints

The constraints we have introduced so far are linear with respect to per class utilization of a reference device. Sometimes it is necessary to express software constraints: limits for a method's total utilization to introduce a critical section; a process's total utilization to express a file descriptor limit; or process or device utilization by class for quality of service management. These software constraints are non-linear with respect to the per class device utilizations that we chose as control variables. As long as the utilization limits specified by the software constraints are never exceeded, it is not necessary to include them as NLP constraints. Their non-linear influences are reflected in the analytic queuing model and henceforth are captured by the objective function. If these limits can be

reached, the software under consideration can become a bottleneck and can affect the entire NLP problem, which becomes a non-linear program with non-linear constraints. Changing the software to avoid the software bottlenecks is the first choice for the performance engineering process. If it is possible to do this, as mentioned above, the software constraints will have no impact on the NLP problem. However, if it is not possible, then we suggest the following strategies:

- Change the problem by reducing the device utilization limits (b_k) so that the software utilization limits are never reached;
- Take an optimistic approach and just ignore the software constraints. This strategy will yield overestimated utilization for the object under scrutiny;
- Solve the NLP with non-linear constraints. This requires the use of techniques from the non-linear programming literature [Baz93] to be closely integrated with analytic models to compute the intermediate values of the objective function. Unfortunately, it comes with a high computational cost.

To better explain the above statements, let us consider the device constraints of our example, and then let us assume that the *Database* object cannot have a utilization greater than 2. These three constraints may be represented as shown in Figure 19. The constraints are represented in 2 utilization space. Device 1 and device 2 constraints are linear constraints with respect to U_{21} and U_{22} . They create a convex space, ABDC. U_{Database} is non-linear (we showed before that an object utilization is non-linear with respect to per class device utilization). With the introduction of the *Database* object constraint, the feasible space is no longer convex (Figure 19a). Because of the FE curve, it is now AFEDC. The non-linear constraints do not make non-linear programming unfeasible, but they make it more difficult. However, by lowering the device utilization limits, we will make the object limit unreachable and henceforth, the object constraint may be ignored. This is illustrated in Figure 19b.

a). Object constraint disturbs the convexity of the space.

b). By reducing the device utilization we guarantee the convexity of the space.

Figure 19. Object utilization limits generate non-linear constraints.

Next, we return to finding maximum object utilization for a non-linear program, with object utilization as the objective function and per class device utilizations as control variables. Unfortunately, we do not have a method to directly tune the device utilization. However, we may alter the device utilizations by changing the population mixes and the total population. Thus, we need a technique to map the customer population space into utilization space. The next sections consider this problem. Since the extreme points of the feasible region are very important, we will focus first on them.

3.3 Bottlenecks, Extreme Points, Saturation Sectors and Crossover Points

This section comprises both asymptotic and non-asymptotic behaviour of multi-class distributed systems in the extreme points; i.e., the points in the utilization space where one or more devices reach the utilization limit imposed by the design constraints. The main goal here is to identify the potential bottlenecks in the system and the conditions under which they occur. We do that by generalizing the work of Balbo and Serazzi [Bal97],[Bal96], [Bal91], [Bal90], in the sense that we allow non-asymptotic conditions and more than three classes and three devices. We will also present some of the formulas and concepts developed by Balbo and Serazzi for asymptotic conditions, because they are important and simplify the presentation.

Note that to solve the programming model described in Section 3.2 we do not need to compute all the extreme points, as we will show in Section 3.4. However, in some situations, computing all the bottlenecks may prove useful for analyzing the system performance behaviour.

3.3.1 Extreme Points in a Polyhedron

So far, we have seen that the design constraints may be expressed by a linear system of inequalities:

$$\text{E 20} \quad \sum_{c \in C} U_{jc} \leq b_i \quad j \in \mathbf{K}$$

$$\text{E 21} \quad U_{jc} = \frac{D_{jc}}{D_{ic}} U_{ic} \quad i, j \in \mathbf{K}; c \in \mathbf{C}$$

We did not mention other constraints that are natural and implicitly included in E 20 and

E 21 respectively, such as:

$$\text{E 22} \quad U_{jc} \leq b_j \quad j \in \mathbf{K}; c \in \mathbf{C}$$

$$\text{E 23} \quad U_{jc} \geq 0 \quad j \in \mathbf{K}; c \in \mathbf{C}$$

E 22 is already included in equation E 20, but is worth taking into account for mathematical purposes. We remind the reader that $\mathbf{K}$, $\mathbf{C}$ are sets, and K and C are their cardinality.

The total number of inequalities is $C+K$, and each of them determines a certain half-space⁹. The intersection of all half-spaces generates a polyhedron (the geometric counterpart of the set of all feasible solutions). Each facet of the polyhedron corresponds to one of the constraints. The intersection of at least two facets generates an extreme point or a vertex.

The number of extreme points is less than:

$$\text{E 24} \quad \binom{C+K}{C} = \frac{(C+K)(C+K-1)\dots(K+1)}{C(C-1)(C-2)\dots 1}$$

which is the number of ways in which C constraints can be chosen from a total of $C+K$ constraints. Actually, the above quantity turns out to be a gross overestimate. Using the results proved by McMullen [McM70], there are at most:

$$\text{E 25} \quad \binom{C+K - \left\lfloor \frac{K+1}{2} \right\rfloor}{C} + \binom{C+K - \left\lfloor \frac{C+2}{2} \right\rfloor}{K}$$

⁹ A formal definition of the half-spaces and related notions is presented in Appendix 1.

extreme points.

Figure 20. Extreme points in a two- dimensional space.

In Figure 20, the points A, B, C, D, F, G, which correspond to E 24, represent the set of all extreme points for a two devices and two classes application. However, only four of them (A, B, C and D) belong to the region of feasibility, and this fact obeys E 25.

When devices reach their physical limits, algebraically the problem remains the same. However, there are some things changing in our model and some properties need to be highlighted, especially with regard to class population in those extreme points. This subject is discussed in the next section.

3.3.2 *Global and Partial Saturation Sectors* [Bal97]

Whenever a device utilization reaches the physical limit, that device become saturated. At that point, the population may be increased without changing the device utilization. Thus, an extreme point that corresponds to one or more saturated devices may be generated by more than one population and population mix. It turns out that there is no point-to-point correspondence between population mixes and the per class utilization of a device at that point.

Balbo and Serazzi defined the *saturation sectors* as those subspaces of $\beta = \beta_1 \times \beta_2 \times \dots \times \beta_C$ in which one or more devices are saturated. The points of the above space in which the set of saturated devices change are called *crossover points*.

There is a point-to-point relationship between the saturation sectors and the extreme points of the polyhedron determined by the device constraints; namely, each extreme point corresponds to a saturation sector and vice versa.

In a saturation sector, the following relations hold (for simplicity, we assume in the rest of the thesis that the physical utilization limits are 1):

$$\text{E 26} \quad \sum_{c \in S_c} U_{jc} = 1 \quad j \in S_d$$

$$\text{E 27} \quad U_{jc} = \frac{D_{jc}}{D_{ic}} U_{ic} \quad i, j \in S_d, i \neq j; c \in C$$

where S_c and S_d are some subsets of C and K , respectively. S_d is the set of saturated devices in the saturation sector, and S_c is the set of classes that yields the saturated sector. Let n be the cardinality of S_c and S_d (we build them having same cardinality for operational purposes). We notice that E 26 and E 27 are the equations verified by the extreme points.

We define *global* saturation sectors and *partial* saturation sectors as being those spaces produced by all classes from C (all β_i differ from zero) and a subset of classes (some β_i are zero) respectively.

Because of the correspondence between extreme points and saturation sectors, per class utilization remains constant within a saturation sector. It does not depend on the population mix. The population mixes at the M crossover points of a saturation sector satisfy the following relation:

$$\text{E 28} \quad \beta^i = U^i \quad i=1, \dots, M$$

where U^i and β^i are vectors of size M ; the components of U^i are per class utilization of device $i \in S_d$ within the saturation sector; and the components of β^i are the population ratios at the crossover point i (note the identity between the crossover points and devices: that is because each crossover point is given by a device). E 28 is the result of making the total population (N) approach infinity. So, the partial or global saturation sectors and crossover points can be determined by solving systems of equations E 26, E 27, and E 28, respectively.

Per class throughput at the crossover points is given by:

$$\text{E 29} \quad X_c = \frac{U_{ic}}{D_{ic}} \quad ; \quad \forall c \in S_c; \quad \forall i \in S_d;$$

In some situations, it is useful to compute the extreme points; and in the asymptotic case it is useful to compute both the extreme and crossover points. The next section investigates the properties of extreme points, and highlights rules that help to reduce the complexity of finding crossover or extreme points. Recall that in the asymptotic case, each extreme point has a saturation sector counterpart, and each saturation sector is bounded by crossover points.

3.3.3 Properties of Extreme and Crossover Points

Let

- $\mathbf{B} = \{1, 2, \dots, B\} \subseteq \mathbf{K}$, the set of natural bottlenecks (we assume that they occupy the first B positions in $\mathbf{K}$, without loss of generality);
- $\mathbf{C}_s = \{c \in \mathbf{C} \mid s \text{ is the natural bottleneck of } c, s \in \mathbf{B}\}$, the set of class indices that have the same natural bottlenecks such that the following relations hold:

- a. $C_i \cap C_j = \Phi, \forall i \neq j$
- b. $\cup_{i=1,B} C_i = C$.

To facilitate the analysis, we depict three cases that may be encountered while searching for extreme points:

- Case 1. All the classes from C have an identical natural bottleneck. It can be proved that in this case there is no possibility that another device can become a bottleneck. The proof for the asymptotic case may be found in [Bal96]. The proof for the non-asymptotic case is very similar, and we do not consider it in this thesis. There are a small number of extreme points in this case, and the complexity of finding them is low;
- Case 2. Each class from C has its own natural bottleneck. Depending on the class demands at the devices, some of the remaining devices can become bottlenecks as well;
- Case 3. Some classes from C have a common natural bottleneck, while some have distinct natural bottlenecks. This is a general case, and can be studied by combining Case 1 and Case 2.

The following paragraphs will detail these three cases and show how to find the crossover points associated with the corresponding extreme points.

Case 1. Common Natural Bottlenecks ($B=1$). All classes in C have a common natural bottleneck. Let its index be 1. This case is true when the following relation holds:

$$E\ 30 \quad D_{lc} > D_{jc} \ \forall j \neq l; \ c=1 \dots C$$

Theorem 1. *If $b_1=b_2=\dots b_K$, then there are C extreme points in the system (C crossover points respectively).*

The proof can be found in Appendix 3. In terms of per class utilization, the coordinates of these points are:

$$U_1 = (b_1, 0, 0 \dots 0)^T$$

$$U_2 = (0, b_1, 0 \dots 0)^T$$

.....

$$U_K = (0, 0, 0 \dots b_1)^T$$

An illustration of this case may be seen in Figure 21, for two classes and $b_1=1$.

Figure 21. Crossover points (A and B) in the common bottleneck case (2 classes).

When the bottleneck device saturates, we have [Bal97]:

$$E \ 31 \quad U_{1c} = \beta_c \quad c=1, \dots, C;$$

$$E \ 32 \quad X_c = \beta_c \frac{I}{D_{1c}}$$

within a saturation sector. Note that the relations E 31 and E 32 do not hold unless the bottleneck becomes saturated. To find β_c under non-asymptotic conditions, we have to solve the related LQM, as will be shown later in this chapter. It is obvious that each class

reaches its maximum throughput when it is the only class using the resources (the corresponding β equals 1).

Case 2. Distinct Natural Bottlenecks: Each class has its own natural bottleneck. It assumes implicitly that $K \geq C$ (there are more devices than classes). To keep the presentation simple, let us assume that device c is the bottleneck for class c . Two sub-cases may hold:

Sub-case 1: Consider:

$$E\ 33 \quad D_{cc} > D_{mc} > D_{kc}, \quad m \in \mathbf{B}; \quad c \in \mathbf{B}; \quad k \in \mathbf{K} - \mathbf{B}; \quad m \neq c.$$

Theorem 2. If condition E 33 holds and if $b_1 = b_2 = \dots = b_K$ then there are at most

$$E\ 34 \quad \binom{2C - \left\lfloor \frac{C+1}{2} \right\rfloor}{C} + \binom{2C - \left\lfloor \frac{C+2}{2} \right\rfloor}{C}$$

extreme points in the system. When $b_i = 1, i = 1..K$, then there is one saturation sector in the system (The proof is presented in Appendix 4).

Figure 22 shows the extreme points in the system when the utilization limit is 1 and there are two classes in the system. Device 1 and device 2 are the natural bottlenecks. Due to E 33, the extreme point B is outside of the feasible space, and two other extreme points are generated (C and E). Note also that no non-natural bottleneck device can generate an extreme point.

Figure 22. Extreme points (A, C, E) in the distinct natural bottlenecks case, two classes, $b_1=1$.

When $b_i=1$, it can be also proven (see Appendix 4) that, except for the natural bottleneck devices, no other devices may saturate. Moreover, there is only a global saturation sector, and the relationship between utilization and population mix holds. Figure 23 shows a representation of an extreme point in the corresponding population mix space. The transformation between the two spaces depends on the value of utilization limits. When the utilization limits are 1 (Figure 23b), the point E in per class utilization space is transformed in the segment FG on the new space. The segment FG is defined as a *saturation sector*, and F and G are the *crossover points*. Under the same condition, the segment [A E) and (E B] are mapped on AF and GB, respectively, in a one-to-one point mapping. A more illustrative example is presented in Figure 24 for three classes and three devices. If the utilization limits are less than 1, then a one-to-one point mapping is done between the utilization and the corresponding population mix space (see Figure 23b). A justification of the one-to-one point mapping is given in Appendix 2.

Figure 23. Extreme points in a population mix space. (a) $b_1=1$, and (b) $b_1<1$.

Figure 24. An example of extreme points (left) and saturation sectors (right) in a multi-class system with 3 classes and 3 natural bottlenecks.

Sub-case 2: Consider

$$E\ 35 \quad D_{rr} > D_{mr} \quad m \in \mathbf{B}; \quad r \in \mathbf{B}; \quad m \neq r$$

In this case, a non-natural bottleneck can reach its limits as well, either alone or simultaneously with other devices. This means that it can generate extreme points too.

Figure 25. Extreme points in distinct natural bottleneck case, with a non-natural bottleneck saturated, $b_1=1$.

Figure 25 depicts a situation where devices 1 and 2 are natural bottlenecks, and device 3 may reach its utilization limits although it is not a natural bottleneck. It also means that in the case of saturated devices, non-natural bottlenecks may saturate and may generate many saturation sectors.

Consider S_c , a subset of classes that covers the entire set B . We want to eliminate the devices that do not generate extreme points. Let Ψ_c be a set of devices:

$$E 36 \quad \Psi_c = \{k \in K, \mid \exists j \in K, k \neq j, D_{kr} \leq D_{jr}, \forall r \in S_c\};$$

and

$$E 37 \quad \Gamma = K - \Psi_c$$

Theorem 3. *Devices that belongs to Ψ_c do not generate extreme points.*

Proof. (Appendix 5)

Note also that belonging to Γ is a necessary but not sufficient condition to reach the utilization limits or saturate, independently of the population mix. Some of the devices of Γ are *masked-off* by other devices, in the sense that they are prevented from reaching their

utilization limits or saturate. In terms of utilization, this means that for a subset $S_d \subseteq \Gamma$, a valid solution of an E 26 and E 27 type system of equations is accepted only if it does not contradict the utilization of the remaining devices in $\Gamma - S_d$; i.e.:

$$E\ 38 \quad U_i \leq b_i; \quad \forall i \in \Gamma - S_d$$

For algebraic and geometrical explanations of this assertion, see Appendix 5.

It is also important to note that when all natural bottlenecks generate an extreme point, then a non-natural bottleneck device cannot join that extreme point.

Case 3. Common and Distinct Bottlenecks: This case covers the general situation when a device can be a common natural bottleneck for a subset of classes while other devices can represent distinct or common natural bottlenecks for a class or a subset of classes. It is worth mentioning that this is the most frequent practical situation for a distributed system. The question is how to find the extreme points and eventually the crossover points in this situation.

To solve this problem, we have to use a “divide and conquer” approach that reduces the problem to the previous two cases. Let us assume that B and C_i have been determined (recall that C_i is the set of classes having device i as a natural bottleneck). Let us consider combinations of S_c and S_d , subsets of C and K respectively, having cardinality n where $n=1 \dots B$.

- a. If $\exists s$ such that $S_c \subseteq C_s$ —that is, if the considered classes have a common bottleneck— then there is no need to solve E 26 and E 27, but simply to apply **Case 1**. This suggests that once the crossover points for the natural bottlenecks have been determined, no S_c that has elements belonging to a unique C_s should be considered in the E 26-E 27 equations. The extreme points belonging to these sets are those

- corresponding to the case when each of the classes involved in S_c uses the system in isolation;
- b. If S_d contains only natural bottlenecks, then S_c should contain n classes that belong to different C_s . A corollary and corresponding rule is that there is no need to consider S_c with $|S_c| > B$;
 - c. When studying the possibility that a non-natural bottleneck saturates, another rule should be obeyed along with *Rule b*. This new rule states that a non-natural bottleneck may reach its utilization limits only if it does it in all $n-1$ cardinality class combinations of S_c elements.

This section has presented three possible cases that may be encountered when searching for extreme points. Rules were provided to avoid a high computation cost.

Computing the extreme points is identical in asymptotic and non-asymptotic cases, by solving equations E 26 and E 27 (in the non-asymptotic case the right hand side of equation E 26 is less than 1). In the asymptotic case, we have also crossover points whose coordinates may be found by applying equation E 28. The next section presents an algorithm for finding the crossover points.

3.3.4 *An Algorithm for Finding Extreme and Crossover Points*

A straightforward algorithm for identifying extreme points explores the combinations of classes that cause combinations of devices to reach their utilization limits (Figure 26). We showed that the number of extreme points is combinatorial. The complexity of the above algorithm cannot be different (see Appendix 7). However, for many of the systems that we model, the number of devices and the number of classes are small. It is worthwhile

then, when some conditions apply, to find all extreme points (Recall that extreme points are given by bottlenecks and combinations of bottlenecks).

Crossover_Points_Algorithm()

Input: QLM

Output: Extreme and crossover points

$V = \Phi$; (initially, the set of crossover points V is empty);

$B = \Phi$;

Step 1. Determine the sets C_i : // C_i is the set of classes that have the device i as bottleneck

- $C_i = \Phi$; $i = 1 \dots K$;
- FOR each class $j \in C$,
 - find its bottleneck, $i = \arg \max_{i=1, K} (D_{ij})$;
 - add j to C_i ;
 - add i to B ;

Step 2. Determine the crossover points that correspond to the natural bottlenecks;

Step 3. For each class $j \in C$, add the vector $\beta_j = \mathbf{1}_j$ to V ;

Step 4. Determine crossover points when there are more than 2 natural bottlenecks

FOR $n = 2$ TO B

REPEAT

- select n classes (S_c set) from a subset of $C_1 \times C_2 \dots C_s$ such that they belong to different C_i ;
- determine the devices that cannot reach their utilization limits as Ψ_c :

$$\Psi_c = \{k \in K, \mid \exists j \in K, k \neq j, D_{kr} \leq D_{jr}, \forall r \in C_i\}; \text{ and}$$

$$\Gamma_c = K - \Psi_c;$$

REPEAT

- select n devices, i.e. S_d set, from Γ_c
- solve E 26 and E 27
- if the solution of E 26 and E 27 do not yield over utilization of devices from $K - S_d$,
 - compute β^i , $i = 1, \dots, n$ with E 28 and add β^i to V ;

UNTIL (all combinations are exhausted)

UNTIL (all combinations are exhausted)

Figure 26. An algorithm for computing crossover points.

An important gain is achieved under asymptotic conditions when the maximum object utilization is in a crossover point. We will show in Chapter 4 that per class response times or any linear combination of per class response times reach their maximum or minimum at the crossover points. Also, we may use the extreme points to develop heuristic algorithms that compute the maximum object utilization or response times under non-asymptotic conditions.

3.4 Algorithms for Finding Maximum Object Utilization

There are three obstacles to applying the non-linear and linear techniques to problem E 19:

- We do not possess an analytic expression for the objective function, but a model. To get a value for it, the model must be solved;
- The control variables are not directly settable in the model; that is, we cannot set a vector $\mathbf{U}_r = (U_{r1}, \dots, U_{rc})$ as input variables. Instead, $\mathbf{U}_r$ is an output of the models;
- The non-linear constraints are not analytic functions but outputs of the LQM model.

The following sections present an iterative technique that integrates the non-linear programming results with a queuing analysis to estimate the R_o^c , and a searching technique to find the maximum object utilization and workload conditions that cause it. This technique assumes that the system under study has a corresponding LQM. The LQM is a closed queuing model where each request class has a population and a think time. We distinguish between the asymptotic and non-asymptotic conditions, since the system behaves differently in these two cases.

3.4.1 The Search Algorithm - Non-asymptotic Conditions (NLP_NASYM)

The following algorithm (Figure 27) is used to determine the population vector $\mathbf{N}=(N_1, N_2, \dots, N_c)$, under which U_o is a maximum when we are on the boundary of the feasible space. We assume that the think times associated with each request class are fixed. Initial threading levels for processes are set to the sum of the populations of the request classes to ensure that they cannot cause queuing delays. The algorithm defined by the following steps was referred to in the introductory section as NLP_NASYM. The algorithm combines non-linear programming with performance modeling and evaluation.

NLP_NASYM algorithm

- Step 1.* Formulate E 19, with an initial estimate of R^c_o that does not include waiting times
- Step 2.* Choose an initial point in the feasible utilization space
- Step 3.* Using the Simplex algorithm choose a new value for reference U^*_r to increase U_o
- Step 4.* Repeatedly solve the system's LQM using the MOL to get a vector $\mathbf{N}$ that causes U^*_r and provides a new estimate for R^c_o
- Step 5.* Repeat steps 3 to 4 until U_r can not increase without decreasing U_o

Figure 27. NLP_NASYMP algorithm.

Step 1 follows from Section 3.2, and is not described further. For Step 2, we choose to set the per class device utilizations to zero.

Step 3 is an application of the Simplex algorithm (An implementation of this step is presented in Figure 61, Figure 62 and Figure 63 in Appendix 6). Three actions are taken:

- **Find the search direction** (**BestDirection** in Figure 62) in utilization space by establishing a new goal for U_r , namely U^*_r . The search path is based on pivoting,

using natural and slack utilization variables by computing numeric estimates for the partial derivatives $\frac{\partial U_o}{\partial U_{rj}}$ from the results of the MOL. Specifically, it computes:

$$\frac{\partial U_o}{\partial U_{rj}} \equiv \frac{U_o(\text{MOL}(\text{LQM}, U_{rj})) - U_o(\text{MOL}(\text{LQM}, U_{rj} + \Delta U_j))}{\Delta U_j} \text{ if } U_{rj} \text{ is a decision variable and}$$

$$\frac{\partial U_o}{\partial U_{rj}} \equiv \frac{U_o(\text{MOL}(\text{LQM}, U)) - U_o(\text{MOL}(\text{LQM}, U + \Delta U))}{\Delta U} \text{ if } U_{rj} \text{ is a slack variable.}$$

The direction that produces the greatest increase in objective function is used in the next step. It is worth mentioning that the derivatives of slack variables are computed indirectly by solving a linear system of equations (ΔU is obtained in this way). This is time-consuming and penalizes the efficiency of the algorithm. Heuristic rules can be applied when the only non-basic variables are the slack ones. Note that when a slack variable equals zero, this means that a certain device has reached its upper bound utilization. An increase in a slack variable means a decrease of that device utilization. According to the simplex algorithm, we alter one non-basic variable at a time, which means that all the other devices will remain at the same utilization. If in some classes our object does not use that device, this means that by decreasing that utilization we do not directly affect the object utilization (see the equation for the slack variables). Instead, more room is made at all devices for customers of classes that use the object and hence induce an increase of its utilization. So, the first rule is: *“Decrease the utilization (slack variable) of one of the devices that is not used by object o.”* If such a device does not exist, we may adopt the following rule: *“Decrease the utilization of the device that has the least demand by object o.”* These two rules are applied repetitively up to the point where a variable to modify is found. If no variable is found, then the maximum utilization is reached at that point.

- **Find the maximum distance** (**MaximumDistance** in Figure 63) that does not exceed the constraints on that direction. The maximum distance can be computed by taking the minimum value of the U_{ij} that makes a basic variable fall to zero. Let Δ be the computed maximum.
- **Find how far to go in that direction** yet still have an increase in objective function.

We do that by finding λ :

- $\text{Max } z(\mathbf{U}'_r) = U_o(\mathbf{U}_1 + \lambda \mathbf{d})$
- subject to: $0 \leq \lambda \leq \Delta$

where $\mathbf{d}$ specifies the search direction. This step can be done using a line search method, such as a binary or golden search method.

- determine the starting point of the next iteration as
 - $\mathbf{U}^*_r \leftarrow \mathbf{U}_r + \min(\Delta, \lambda) \mathbf{d}$

Step 4 applies the MOL to search for a population vector $\mathbf{N}$ that achieves $\mathbf{U}^*_r$. The search algorithm uses a *hill-climbing* technique, and is described as follows.

From an initial population vector at the 0-th iteration $\mathbf{N}^0 = (N^0_1, N^0_2, \dots, N^0_C)$ (for example, the zero vector), we search toward the target point $\mathbf{U}^*_r$ with the throughput vector $(X^*_1, \dots, X^*_C) = (U^*_{r1}/D_{r1}, \dots, U^*_{rC}/D_{rC})$ by varying class population and solving the LQM. With each solution of the LQM we have new estimates for R^c_o and U_o . At the k -th iteration, the algorithm chooses the most promising direction for changes to the population vector:

- $\Delta X^t_c = X^*_c - X^t_c$, where X^t_c is obtained by solving the LQM for $\mathbf{N}^t$, where t denotes the t -th iteration
- $d = \max_{c \in C} (|\Delta X^t_c|)$
- $N^{t+1}_d = \begin{cases} N^t_c + \Delta N & \text{if } \Delta X^t_c < 0 \\ N^t_c - \Delta N & \text{otherwise} \end{cases}$

where ΔN is a real number controllable by the analyst.

Open models can give an initial population vector. Unlike a closed model, an open model allows us to directly compute the population and population mixes as well as the object response time under the condition that the per class utilization of devices is known. Based on that, we are able to determine the class populations in the system:

$$\text{E 39} \quad N_{ij}^0 = \frac{U_{ij}^*}{1 - U_i^*} ; N_i^0 = \sum_{j \in R} N_{ij}^0 \quad \text{and} \quad N^0 = \sum_{i \in M} N_i^0$$

where N_{ij}^0 denotes the j class population at device i . To avoid a large computation effort, we use the above determined population as an initial starting point toward a satisfactory solution.

The iterations continue until we are close enough to the per class throughputs that give U_r^* . The chosen measure of the difference between successive solutions is E_t and is defined as the sum of squares of the changes in throughput:

$$\text{E 40} \quad E_t = \sum_c (\Delta X^t_c)^2$$

The search terminates when E_t is less than or equal to a tolerance. If U_r^* cannot be achieved, there exists some bottleneck in the system. Its impact can be identified from the results of the MOL, and is reflected as a constraint in E 19.

Solving E 19, we determine directly:

- $U_{r1}^+, U_{r2}^+ \dots U_{rG}^+$, the solution of E 19;
- $U_1^+, U_2^+ \dots U_K^+$, the total utilization of devices; and
- U_o^+ , the maximum utilization of the object o .

The results of the MOL give:

- The population vector N that causes the maximum utilization of an object to be reached; and,
- The throughput $X_c^+ = U_{rc}^+ / D_{rc}$ per request class c .

The following are the results of several iterations of the NLP_ASYM that leads to the maximum utilization for our example constraints introduced in Section 3.2.2. The object under scrutiny is *Employee*. We use the terminology of Appendix 6.

Iteration 1:

$$\mathbf{U} = \begin{pmatrix} U_{21} \\ U_{22} \\ U_{23} \\ U_{24} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0.8 \\ 0.9 \end{pmatrix}; U_{Employee}=0; \mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1.28 & 0.8 & 0 & 1 \end{bmatrix}$$

$$\mathbf{I}_0 = \{U_{23}, U_{24}\} \quad \mathbf{A}\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \mathbf{Y} = \mathbf{A}\mathbf{N} = \begin{bmatrix} 1 & 1 \\ 1.28 & 0.8 \end{bmatrix}$$

The search direction is to be detected by varying the population for the two system classes. Using **BestDirection** algorithm, with $\Delta U = 0.01$, we find that:

$$\Delta U_{Employee} / \Delta U_{21} = 0.2 \text{ and}$$

$$\Delta U_{Employee} / \Delta U_{22} = 0.94$$

The most promising direction is along U_{22} .

MaximumDistance algorithm gives the maximum allowable distance on the U_{22} direction.

It is $\lambda=0.8/1$, which means that we can go till the extreme point B (Figure 28) is reached.

At that point (Point B in Figure 28), we start the second iteration:

Iteration 2:

$$\mathbf{U} = \begin{pmatrix} U_{21} \\ U_{22} \\ U_{23} \\ U_{24} \end{pmatrix} = \begin{pmatrix} 0 \\ 0.8 \\ 0.0 \\ 0.18 \end{pmatrix}; U_{DataBase}=3.7;$$

$$\mathbf{I}_1 = \{U_{21}, U_{23}\} \quad \mathbf{A}_B = \begin{bmatrix} 1 & 0 \\ 0.8 & 1 \end{bmatrix}; \quad \mathbf{A}_N = \begin{bmatrix} 1 & 1 \\ 1.28 & 0 \end{bmatrix};$$

$$\Delta U_{Employee} / \Delta U_{21} = 0.25 \quad \text{and}$$

$$\Delta U_{Employee} / \Delta U_{23} = 0.11$$

The direction to explore is along U_{23} with maximum distance $\Delta_{max} = 0.55$ (Point D in Figure 13).

MaximumDistance algorithm gives the maximum allowable distance on the U_{11} direction.

It is $\lambda = 0.5$.

Iteration 3:

$$\mathbf{U} = \begin{pmatrix} U_{21} \\ U_{22} \\ U_{23} \\ U_{24} \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.3 \\ 0.0 \\ 0.097 \end{pmatrix}; \quad U_{Employee} = 3.9;$$

$$\mathbf{I}_k = \{U_{21}, U_{22}\} \quad \mathbf{A}_B = \begin{bmatrix} 1 & 0 \\ 1.28 & 0.8 \end{bmatrix}; \quad \mathbf{A}_N = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\Delta U_{Employee} / \Delta U_{23} = 0.00001 \quad \text{and}$$

$$\Delta U_{Employee} / \Delta U_{24} = 0.00002$$

The class populations that are accommodated in the system in this point are 33 and 40, respectively. At this point, we find that there is not any other direction to explore, which means that the maximum has been reached.

Figure 28 shows the feasible space dictated by the constraints of our example, and the point in the decision variable space where the objective function for *Employee* object reaches its maximum (point E) while the object *Database* reaches its maximum utilization at the point D. The objects' total utilization for the points C, D, E, B and the class populations that cause these utilization are presented in Table 3.

Table 3. Request class populations that cause the maximum object utilizations.

	C N=(78, 0)	D N= (63,19)	E N=(33,40)	B N=(0,63)
Employee	1.34	3.44	3.9	3.7
DataBase	7.6	9.8	7.0	5.3

Figure 28. The feasible region of the personnel application. ABDC denotes the feasible area, and AB and BE are the search directions for maximum utilization of the *Employee* object.

3.4.2 The Search Algorithm in the Presence of Saturated Software Entities

The software utilization limits imposed on the software entities make the search algorithm computationally complicated. Not only is the convexity of the feasible region not fulfilled, but the searching process is more difficult to implement. Furthermore a performance engineering process should not plan for software bottlenecks.

There are several methods to deal with such kind of problems, all feasible and tractable when an analytic objective function and analytic non-linear constraints exist. Their mathematical foundations are very well presented in [Baz93]. However, we do not have

analytic expressions for the object utilization or software constraints but models. Note again that to compute the utilization of an object we have to solve a model and go through an intermediate space, the class populations. As described in Section 3.2.4, the following methods are proposed:

Reconsidering the device utilization limits. Note that the goal of finding maximum utilization is to size the threading or replica level of a software component. We do that for performance reasons. These reasons usually require that no device or software entity is saturated under the real-life workload conditions. A saturated software component is more difficult to identify or adjust after the application is deployed. Hence, we should make sure that we do not reach the software utilization limits. Reconsidering the device utilization limits we can ensure that no software component reaches its limits. This technique was illustrated in Figure 19b.

An optimistic approach. The second way of overcoming the software utilization limits is to take an optimistic approach and just ignore the software bottlenecks: that is, we just go on the convex domain border, ignoring those non-linear curves. Ignoring them yields a substantial reduction in the algorithm computation effort. This approach does not eliminate the software bottlenecks but still computes the utilization of those objects that do not have utilization constraints. In the case when the software utilization limits border the feasible space, then the maximum object utilization is an overestimation. Otherwise, the utilization we find is the maximum utilization.

A modified simplex algorithm. As a last attempt to find the object utilization in the presence of the software utilization restrictions, we may adapt the simplex algorithm presented in the previous section. It consists mainly of linearization of the non-linear constraints around the searching points and extending matrix **A** (see Figure 61) to cover the coefficients of the linearization as well. Since there are few software constraints and

many hardware constraints in a system, this operation should be done at those points where the non-linear software limit is reached. The mathematical foundations for this approach are presented in [Baz93].

3.4.3 The Search Algorithm in the Presence of Saturated Devices

The search algorithm for the maximum object utilization under asymptotic conditions is different from the non-asymptotic case. The difference is with regard to the searches in both the utilization and population spaces. Under asymptotic conditions, a point in device utilization space may be obtained from many population mixes and for many population levels. This is trivial, because if the saturation limit is reached for, say, 100 customers, then the same limit is reached for any number higher than 100. However, a saturated device generates near-linear response times, and object utilization may be considered linear. Due to this property, the maximum object utilization is reached at the extreme points (in the utilization space). As a result, we can search the maximum object utilization at the extreme points by following two algorithms: in the first algorithm, we first find **all extreme points** (and crossover points eventually) and then find the objects' utilization at each of these points; in the second algorithm, we navigate through feasible space following a linear programming search.

Computing object utilization at crossover points - Straight Algorithm

Let V be the set of all crossover points of the system computed by the Crossover_Point_Algorithm (Figure 26). If the object utilization is a linear function with respect to device utilization (as in the asymptotic case), then the utilization of an object is reached at one of the extreme points. The extreme points may be also used in the non-

asymptotic cases when an approximate solution is sought. Finding an object utilization is done by adding a new step to the Crossover_Point_Algorithm.

For the non-asymptotic case, this step should be:

Step 5 (Non-asymptotic). FOR each $o \in \mathbf{O}$, DO $U_o = \max_{v \in \mathbf{V}} \text{HillClimb}(\text{MOL}(\text{LQM}))$,

where HillClimb is a mapping method between utilization space and population mix space, as illustrated in Section 3.4.1 and MOL(LQM) is the solver of LQM. We define Crossover_Point_Algorithm plus Step 5 as the *Straight* algorithm.

For the asymptotic case, asymptotic analysis suggests an asymptotic response time at the saturated devices, and consequently an asymptotic utilization for the objects that directly or indirectly make use of the devices. In this case, a supplementary design constraint must act on the population level allowed in the system. Let N^{high} be the maximum population level allowed in the system. The definition of β^v_i gives:

$$E\ 41 \quad N^v_i = \beta^v_i N, \quad i=1,C; \quad \forall v \in \mathbf{V};$$

In this case, it is not necessary to apply the hill-climbing algorithm to find the population mix in extreme points, but to directly solve LQM with the population computed by E 41.

Step 5 (Asymptotic): FOR each $o \in \mathbf{O}$, DO $U_o = \max_{a \in \mathbf{V}} (\text{MOL}(\text{LQM}, N^v_i))$

The complexity of Step 5 is:

Non-Asymptotic: $\Omega(\text{Step 4}) = O(|\mathbf{V}| \Omega(\text{HillClimb}))$

Asymptotic: $\Omega(\text{Step 4}) = O(|\mathbf{V}| \Omega(\text{MOL}))$.

For each object, all elements of V should be checked. In the worst-case scenario, the cardinality of V equals the number of iterations in Step 4.

Linear Simplex Algorithm (Figure 29)

LP_ASYMP algorithm

- Step 1.* Formulate E 19, with an initial estimate of R_o^c that does not include waiting times;
- Step 2.* Choose an initial point in the feasible utilization space;
- Step 3.* Using the simplex algorithm choose a new value U_r^* to increase U_o ;
- Step 4.* Compute the crossover points associated to U_r^* ;
- Step 5.* Compute object utilization in the crossover points determined at step 4, solving LQM with the class populations given by equation E 41;
- Step 6.* Repeat steps 3 to 5 until U_o can not be increased.

Figure 29. LP_ASYMP algorithm.

The simplex algorithm remains far and away the better way of attack when the complexity of the system (in terms of number of classes and devices) is high and the number of objects is small.

Steps 1 and 2 are identical to the non-asymptotic case and are not described further.

Step 3, unlike the asymptotic case, finds the search direction by computing the derivatives of the objective function with respect to the population mixes in the current crossover point. It does not necessarily make direct use of the reference device r , because this device may not be saturated. However, it uses it to project its utilization space into saturated device spaces by using equation E 41. Once a search direction is found, the next point in the utilization space is found by directly applying the simplex methods.

Figure 30. Searching maximum object utilization in a three classes, three natural bottlenecks system.

Step 4 does not use a hill-climbing algorithm. It computes the crossover points and the class populations in these points by applying equation E 41. The class populations are used as inputs for LQM which, solved by MOL, gives the object utilization.

In Steps 4 and 5, the object utilization at the crossover points is computed directly from equation E 41.

As an example, consider a three-class system with three bottlenecks. Figure 30 shows an example of how the search may alternate between the utilization and the population mix spaces. The numbers associated with the extreme points and crossover points give the order of the steps. Usually, after a extreme point in the utilization space has been detected (let's say point 7 in the utilization space), the object utilization is computed for all corresponding crossover points (points 8, 9, 10 in the population mix space).

Example - Asymptotic study. Now we apply the asymptotic techniques to the Personnel example when the devices are allowed to saturate. The upper utilization limit of devices 1

and 2 is equal to 1. Assume that the maximum population is 120. The maximum object utilization is computed by using the two methods described above.

A. Straight Algorithm

Using the Straight algorithm, we first compute all crossover points for the extreme points shown in Figure 31. Then, for each crossover point, the utilization of each object is computed. These are summarized in Table 4. Note that the extreme point D has two corresponding crossover points denoted by the indices 1 and 2. As can be seen, both objects reach their maximum utilization at point C.

Table 5 shows the objects' utilization when the population is increased to 200. As expected, the objects will reach their maximum utilizations in the same extreme points and in the same crossover points (this confirms that the object utilization is linear).

Table 4. Extreme points, crossover points and objects utilization for 120 customers

Extreme Points	Crossover Points($\beta_1; \beta_2$)	U_{employee}	U_{employee}/N	U_{DataBase}	U_{DataBase}/N
B	(1,0)	41.4	0.345	1.92	0.016
D ₂	(0.746,0.253)	37.68	0.314	17.28	0.144
D ₃	(0.583,0.416)	38.64	0.322	10.8	0.09
C	(0,1)	48.24	0.402	29.16	0.243

Table 5. Extreme points, crossover points and objects utilization for 200 customers.

Extreme Points	Crossover Points($\beta_1; \beta_2$)	U_{employee}	U_{employee}/N	U_{DataBase}	U_{DataBase}/N
B	(1,0)	120	0.6	2	0.01
D ₂	(0.746,0.253)	118	0.59	30	0.15
D ₃	(0.583,0.416)	116	0.58	54	0.27
C	(0,1)	128	0.64	74	0.37

B. LP_ASYMP algorithm

The initial point of the algorithm is the origin of the utilization space. The next point in the utilization space is computed by the simplex algorithm and it is:

$$\mathbf{U} = \begin{pmatrix} U_{21} \\ U_{22} \\ U_{23} \\ U_{24} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0.0 \\ 0.18 \end{pmatrix}$$

with U_{11} and U_{12} the natural utilizations and with U_{13} and U_{14} the slack variables. Whenever a slack variable is 0, it denotes a saturated device. Thus, the number of saturated devices gives the number of crossover points. Their coordinates may be found by applying the relationship between population mix and utilization.

Figure 31. The searching path for asymptotic convex algorithm.

In our example, in the extreme point B, we have only one crossover point since only U_{13} is zero. Its coordinates are (0, 1). We should compute the object utilization for $N_1=0$ and $N_2=120$ customers, which gives $U_{DataBase}^1 = 41.4$. The next extreme point to check is D. There are two crossover points there, since there are two slack variables that have the value 0. These are $\beta^1=(0.746,0.253)$ and $\beta^2=(0.583,0.416)$, with the object utilizations

37.68 and 38.64, respectively. Both are less than the utilization in the previous extreme point, which shows that that was the maximum. Figure 31 shows the searching process for this example. Note from this example that each object has its own searching path, which may make the whole search for maximum object utilization costly when the number of objects is large.

3.5 Complexity of LP_ASYMP and NLP_NASYMP

After all these discussions, it is time to motivate our reasons for using the simplex like methods to find the maximum object utilization. Table 6 and Table 7 summarize the main reasons.

Table 6 shows different combinations of number of devices and classes, K and C respectively. The column labeled *Bottlenecks* shows the number of bottlenecks as a function of $C+K$ (see Section 3.3.1 for how to compute this column). The practical complexity of the linear simplex is $3K/2$ [Chv83] and the number of extreme points explored by the algorithm is shown in the last column. We note that this number is much lower compared with the total number of bottlenecks (which is combinatorial, as shown earlier in this chapter).

Table 6. Extreme points searched for each object by simplex algorithm compared with total number of bottlenecks.

K	C	C+K	Bottlenecks	Simplex
2	2	4	4	3
4	4	8	20	6
6	6	12	112	9
8	8	16	660	12
10	10	20	4004	15
20	20	40	40060020	30
30	30	60	4.6E+11	45

Table 7. The number of solved LQMs per object: $N=500; K=10$.

C	# of mixes	Bottlenecks	NLP_NASYMP	LP_ASYMP
2	501	22	36	6
3	125750	67	54	12
4	10510375	137	96	15
5	664114625	318	220	35
6	33039054700	528	400	54

Table 7 shows even more clear the advantages of the simplex approach for ten devices and $N=500$. There we see that the number of bottlenecks is much lower than the number of population mixes. A brute force algorithm will compute the maximum object utilization by solving the associated LQMs for all population mixes, practically an intractable problem. By considering the Simplex like algorithms the number of solved LQMs is nearly polynomial (it should be proportional to the simplex) and certainly much lower than the number of population mixes. There are situation when the number points searched by the simplex may reach the total number of extreme points (or bottlenecks). These cases are well documented in literature [Chv83] and can be avoided. NLP_ASYMP is a non-linear version of simplex and because of the hill climbing technique, the number of solved LQMs is much higher than that for LP_ASYMP. The latter does a direct mapping from utilization to population mix space in C steps (the number of classes C is the maximum number of crossover points per extreme point). The numbers from NLP_NASYM and LP_ASYMP columns have been obtained for the example introduced in the previous sections. We extended the number of classes and devices and varied the demands uniformly within the $[10, 210]$ ms interval.

3.6 Case Study - Using Object Utilization to Decide Object Activation Policy

Replication and activation policies are closely related to the maximum object utilization. The utilization of a process is the sum of its objects' utilizations.

In general, processes and objects with maximum utilization near zero need not be active. Depending on the request arrival distribution, the activation policy should be *shared server* (in which one instance of the object is shared by all requests) or *server per request* (an instance of the object is activated for each request). The latter may be more appropriate if arrivals come in bursts and the startup overheads are low. Terminating a process or object permits the allocation of system resources to other low-frequency processes/objects.

Processes/objects with maximum utilization that is much greater than zero should be active. For these entities, replication and threading levels must be decided. The maximum utilization gives the appropriate replication levels. If the replication levels are lower than the maximum utilization, the customer response times are higher than necessary; if they are beyond the maximum utilization, more memory may be used but there is no improvement in customer response times. Secondary effects from paging overheads may degrade performance. Below, we show the impact of replication level for *Database* and *Employee* objects on query response times.

Figure 32 and Figure 33 show the *query* response times versus population level and the threading level for the *Database* and *Employee* objects. The population level varies from 10 to 82 by keeping the population of the two classes at a constant ratio of 16/63. This corresponds to point D in Figure 28 (see also Table 3). In the figures, the threading level of one of the objects is kept at its maximum value while the threading level of the other object is varied.

Figure 32. Query response time versus the *Database* threading level and population level; (t_1, t_2) denotes the number of threads of *Database* and *Employee* objects. N_1 denotes the class 1 population.

Figure 33. Query response time versus the *Employee* threading level and the population level; (t_1, t_2) denote the number of threads of *Database* and *Employee* objects. N_1 denotes class 1 population.

From the figures, it is clear that too few threads yield high response times since requests suffer queuing delays at the process of the object. Furthermore, having more than the

maximum number of threads does not improve customer request response times. As shown in the figures, the maximum numbers of threads are 4 and 10. These values are given by the search algorithm (see Table 3). We see that the threading level is much lower than the number of customers. This suggests that one thread per session and object is not beneficial from a performance point of view. Even under asymptotic conditions, the threading level, although it is closer to the number of customers, is still low. For our example, under the asymptotic conditions with 120 customers in the system, the threading levels of *Database* and *Employee* are 39 and 18; with 200 customers, the levels are 128 and 74 (see Table 4 and Table 5).

3.7 Summary

In this chapter, we illustrated a methodology for finding the number of object replicas or the object threading level. The method is based on finding the maximum object utilization under design or physical constraints.

First, we illustrated three properties of object utilization:

- A monotone property with respect to class device utilization;
- A non-linear property with respect to the same variables under non-asymptotic conditions; and,
- A linear property with respect to the same variables under asymptotic conditions.

Based on these properties and on physical constraints, we have shown that finding the maximum object utilization is similar to solving a non-linear programming problem with linear constraints. This is based on the fact that the maximum object utilization may be found on the boundary of the feasibility space; in the asymptotic case, the maximum is reached in the extreme points generated by the constraints. These properties dramatically

reduce the search space and allow us to use a systematic procedure in finding the object's maximum utilization by using well-known procedures. A very important reduction in the search space was also accomplished by processing the constraints and expressing them in terms of per class utilizations of a shared device.

However, using per class device utilizations is not enough to solve the non-linear programming problem. To find the objective function, i.e. the object utilization, we have to solve the LQM associated with the system and to map the per class device utilizations to inputs for the LQM. Since the inputs in LQM consist of per class populations, we showed two mapping methods:

- Under non-asymptotic conditions, the mapping may be done by a hill-climbing method;
- Under asymptotic conditions, the mapping may be done using the properties of the extreme points.

The properties of extreme points are illustrated in the chapter so that they may be used for solving the problem. We showed that:

- Under non-asymptotic conditions, there is a one-to-one correspondence between per class device utilizations and per class populations; and,
- Under asymptotic conditions, extreme points from utilization space are mapped in saturation sectors bounded by crossover points.

Based on the mapping between the extreme points and the saturation sectors, we illustrated the properties of extreme points and crossover points that help to compute these points; the crossover points may be used for searching the maximum object utilization.

Additionally, we provided:

- Two systematic search algorithms, for asymptotic and non-asymptotic conditions, based on the convex simplex algorithm;
- A straightforward algorithm for finding the crossover points and object utilization in these points;
- An analysis of the algorithms' behaviour under saturated software components.

A case study was presented to show how these techniques are applied.

4 OBJECT ALLOCATION IN MULTI-CLASS DISTRIBUTED SYSTEMS

But the most common and durable source of factions has been the various and unequal distribution of property.

James Madison- The Federalist.

4.1 Introduction

The goal of this chapter is to offer a quantitative insight into object allocation, its impact on the performance of a distributed system, and to offer methods that systematically deal with the object allocation in the presence of complex workloads. The result of an allocation is a *configuration*. By complex workloads we mean workloads with:

- Many population levels (N); and,
- The whole population mix space for each population level.

The need to consider such workloads is justified by the life cycle of a distributed system; a distributed system supports multiple classes of requests and it should meet its performance requirements for a variety of workloads. In general, total population levels can be specified at the requirement stage, however population mixes cannot. Hence, we consider the whole population mix space.

It should be mentioned that there is not likely to be a single configuration that optimally satisfies all the performance requirements for all workloads. Instead, we search for a small set of configurations that allows for good performance behaviour under a wide range of workloads. Then, multi-attribute decision-making can be applied to choose a design that satisfies not only performance criteria but other non-performance criteria as well.

Figure 34. The framework for establishing the best configuration: ABA-asymptotic bound analysis, LP-linear programming, NLP-non-linear programming- VHL-very high population level, HL - high population level, ML-medium population level.

The approach we take throughout this chapter is described in Figure 34 and it narrows the general framework presented in Chapter 1, Figure 2. The pseudo-code description of the object allocation was also given in the Chapter 1. *Constraints*, *topologies* and the *LQM* are the initial inputs for this step and they are inputs or results from previous stages of the general framework. We have already talked about the *model* and how to build it in Chapter 2. The *topology* reflects the technological view of the application (the physical layout of the system and an initial allocation of processes or objects) and includes performance parameters of the nodes and network.

According to the framework, based on initial data, a scalable (the exact meaning of the term will be defined in Section 4.3.2) configuration is generated for very high population levels (VHL). We will show in this chapter (by using Asymptotic Bound Analysis-ABA) that beyond a certain population, the bottlenecks dictate the response time of a system. Moreover, there is one bottleneck that gives an upper bound for the mean response time asymptote over all population mixes. In the same way, other bottlenecks give the lower bound for the mean response time asymptotes over all population mixes. A scalable configuration may be generated by allocating the objects such that the upper bound asymptote is minimized.

For the population levels less than the above-mentioned threshold, a search is conducted across the configuration space with the goal of maximizing the *Satisfaction* function, as defined in Chapter 1. The approach is explained in greater details below:

- For a given configuration, at high population levels (HL), the worst response time of a class is obtained at a specific crossover point (we defined crossover points in Chapter 3). Determining the specific point is a linear program (LP) search using the system design constraints combined with a performance evaluation of the system's model (MOL and LQM). The search for a configuration at HL, starts with the configuration determined for VHL and continues until a maximum *Satisfaction* is achieved. Each new configuration is generated based on the crossover points that give the maximum per class *Satisfaction*. The performance estimation for each crossover point is done with the associated LQM by using the properties of ABA;
- For a given configuration, at medium population levels, the maximum mean response time of a class (hence the minimum per class *Satisfaction*) is found on the boundary of the feasible space generated by the device utilization limits that act as the design constraints. The search toward the extreme points that give the extreme response times for a class, is conducted by a non-linear program (NLP). Performance evaluation in this case is done through the same LQM but in this case we use NLP to guide the

search. The search could start with the configuration shown satisfactory for HL. Each configuration is generated based on the extreme points that gives the maximum per class mean response times.

The reminder of this chapter is organized as follows:

- Section 4.2 justifies the need to consider multiple population levels; an example is introduced to show that each population level may have its own optimal configuration;
- Section 4.3 discusses the properties of the response time of the distributed systems under asymptotic conditions;
- Section 4.4 presents the allocation algorithms for VHL, developed by using the properties illustrated in section 4.3;
- Section 4.5 describes the search for configuration at high and medium population levels;
- A summary and conclusions are presented in section 4.6.

4.2 The Need for Considering Low, Medium and High Population Levels

This section will show by an example that it is important to consider many workloads when choosing a configuration for a distributed system. In the example, we fix the population mix and vary the population level. The example will show that the configuration can depend on the population level. Figure 35 defines four intervals on the total population variable. The intervals correspond to the four population areas: low, medium, high and very high. The population N^{low} , N^{medium} and N^{high} defined in the performance requirements (see Chapter 1) are chosen to reside in these intervals.

Figure 35. Population areas and population levels from the performance requirements.

Next, we augment the example presented in Chapter 2. Table 8 presents the demands of the objects and Table 9 shows 3 possible configurations. There are three classes of request, 3 heterogeneous nodes, 6 processes and 12 objects. The object interaction is presented in Figure 36.

Table 8. Object demands, total and per class: Sc_i columns indicate the demands per class i; O_i are objects and Cl_i are objects on the client hosts.

Object Name	CPU Demand			DISK Demand		
	Sc_1	Sc_2	Sc_3	Sc_1	Sc_2	Sc_3
Cl ₁	20	0	0	0	0	0
Cl ₂	0	20	0	0	0	0
Cl ₃	0	0	20	0	0	0
O ₁	60.81	60	20	88	88	88
O ₂	50	50	0	80	80	0
O ₃	50.	0	10	80	0	80
O ₄	24.99	25	0	80	80	0
O ₅	10	0	0	80	0	0
O ₆	0	0	99.99	0	0	400
O ₇	75	0	99.99	80	0	400
O ₈	75	75	0	80	80	0
O ₉	0	0	99.99	0	0	400
O ₁₀	75	0	99.99	80	0	400
O ₁₁	75	75	0	80	80	0
O ₁₂	0	75	0	0	80	0

Figure 37. Class 1 response time for three configurations (see Figure 36 and Table 9).

Consider only one class, say Sc_1 , and suppose that CPU factors (factors that divide the demands of Table 8) of Client Node, 2d Tier, 3d Tier are 1, 2 and 4, respectively. The response time for the configurations described in Table 8 are shown in Figure 37.

The response time shows how different configurations are optimal for different areas of population. At low population level, with one customer in the system, configuration A1 offers the best response time. This is due to the fact that there are many objects allocated to the nodes with high processing power. Practically, the queuing delays are insignificant, the response time is given by:

$$E\ 42 \quad R_d = \sum_{i=1}^K D_i$$

The common sense rule to minimize the above sum is:

Heuristic Rule: *“When running at low population levels, place most of the objects on the high speed processing nodes.”*

Systematic techniques that deal with the sum already exist. These include Max Flow/Min Cut algorithms [Gal82][Lo88] or heuristics [Pri90].

When the population level is medium, 10 customers, A3 offers a better response time than A1; the queuing delays start to become important. This is a non-linear area and the performance is very sensitive to the population level.

For high population levels, A3 is the best alternative because some of the work is assigned to the client. This motivates a second common sense rule that says:

Heuristic Rule: *“To be scalable at high population levels, move as much load as possible to the client.”*

When the population mix is taken into account, it is difficult to come up with common sense rules. In this case more complex techniques are needed and the goal of the next sections is to devise such techniques. We start with the very high population levels. The next section introduces the fundamentals of the asymptotic response time of multi-class-distributed system. The section is based on the work of Balbo and Serazzi.

4.3 Analysis of Response Times and Object Allocation at Very High Population Levels

In this section we focus on very high population levels and therefore we study the asymptotic behaviour of the multi-class-distributed systems. The study will be done entirely with hardware devices as source of queuing delays. We do not consider software

queuing delays. This fact relies on the reality that the bottlenecks dictate the behaviour of the system and by assumption we can avoid the software bottlenecks. (In Chapter 3 we showed that we could detect the software bottlenecks and the designer could take action and avoid them). As result, this section is entirely based on the QNMs. Also, we will talk about the asymptotic response times. For simplicity, we do not repeat “asymptotic” term whenever we talk about the response times. However, the reader should be aware of the meaning.

We will show that in the presence of multiple bottlenecks in the system, *per class mean response time* ($R_w^{(c)}$) and *weighted mean response time* (R_w) may be written as:

$$R_w^{(c)} = N * Scalability\ factor + Shift\ Factors * Scalability\ factor$$

$$R_w = N * Scalability\ factor + Shift\ Factors$$

The scalability factor is given by the demands at the bottleneck devices and is the term that has the greatest effect on the mean response times. *Shift factors* are mainly due to the imbalance of the demand on non-bottleneck stations or the influence of the population mixes on the bottleneck devices. The shift factor’s influence is important at small population levels but is small at high population levels.

There are two metrics that we try to minimize at very high population levels: *maximum per class mean response time* over all classes (we call it *fair mean response time*, because per class response times tend to become equal) and the weighted response time. The following sections show how scalability and shifts factors can be used to minimize the above metrics. Also, bounds on these factors are presented. We define *scalability* as the property of accommodating an increased population level with an acceptable increase in response time. The configurations that have this property are called *scalable configurations*.

4.3.1 Response Time in Single-Class Systems

Although it is not the focus of this thesis, we start a discussion with single class systems and show their similarities with multi-class systems. In a single class system, with K devices and bottleneck device b , such that $D_b = \max_{i=1,K}\{D_i\}$, the mean response time is:

$$E 43 \quad R_w = ND_b$$

where R_w is the time required accomplishing a request when there are N users in the system and D_b is the demand at the saturated bottleneck resource.

Suppose that through a wise allocation process, we are able to decrease the demand at device b by Δ by balancing the load across B bottleneck devices in the system. Results for single-class systems with multiple bottlenecks [Lip82] give:

$$E 44 \quad R_w = N(D_b - \Delta) + (B-1)(D_b - \Delta)$$

E 44 shows that, when the population level is beyond a threshold N^* , multiple bottlenecks in the system can diminish the response time.

Figure 38. Asymptotic response time for single and multiple bottlenecks in single class systems.

Figure 38 shows the asymptotic response time for two configurations; the configurations have a single bottleneck and B bottlenecks, respectively. When the population is higher than N^* , R_w^c for the balanced configuration is less than that of the unbalanced configuration. The coefficient of N in E 44, $D_b - \Delta$, can be interpreted as the scalability factor, and $B - 1$ as the shift factor.

4.3.1.1 Scalable Configurations for Single Class Systems

In terms of finding scalable configurations, the problem may be stated as:

$$\begin{aligned} \text{E 45} \quad & \text{minimize } \{D_b\} \\ & \text{subject to a feasible allocation.} \end{aligned}$$

Heuristic Rule: *To obtain a scalable configuration, choose an object allocation that minimizes the demand at the bottleneck device.*

So far we have shown that under asymptotic conditions, single class systems are worth balancing beyond a certain threshold N^* and that a scalable system may be obtained by minimizing E 44. Balancing the load across the devices is a well-known heuristic (see the load balancing definitions in Chapter 1). With E 44, we showed that beyond N^* , load balancing can be an optimal allocation algorithm (we should remember that we talk about asymptotic bounds not about the estimates for mean response times) for single class systems.

4.3.2 Response Time in Multi-Class Systems with a Common Natural Bottleneck

The goal of this section is to:

- Pose the problem of finding scalable configurations in terms of per class demands at bottlenecks;
- Estimate bounds on scalability and shift factors for a given configuration whenever we talk about fair response time or weighted response time.

In a multi-class-distributed systems, a user request must compete with other user requests within the same class or that belongs to other classes. We can write asymptotic limits for mean response time in a form similar to single class systems. In the case of a common bottleneck for all classes, asymptotic mean class response time is bounded as follows [Bal96]:

$$\text{E 46} \quad R_w^c = ND_b^c + D_b^c I_c$$

where D_b^c is the demand of class c at the common bottleneck b and I_c is a correction factor having the units of N . They may be interpreted as the scalability and shift factors. I_c corrects the first term of R_w^c in which we assumed that all customers queue at the common bottleneck device for the same demand, D_b^c . Numerically, I_c is [Bal97]:

$$\text{E 47} \quad I_c = (K-1) - \sum_{i=1, i \neq b}^c \frac{1 - \frac{D_i^c}{D_b^c}}{1 - \sum_{s=1}^K \beta \frac{D_s^c}{D_b^c}} = \sum_{i \neq b} \frac{f_i^c - f_i^*}{1 - f_i^*}$$

where $f_{ic}=D^c/D^c_b$ is the *loading factor* of device i in class c and $f^*_i = \sum_c \beta_c f_{ic}$ is the *average loading factor* for device i . $\beta_c = N_c/N$ is the class c population mix. E 46 and E 47 are obtained by taking the limit of equation E 4 as $N \rightarrow \infty$.

Significance of f^*_i and I_c . It is easy to prove that f^*_i measures the total utilization of device i . From [Bal96], $U_{ic} = \beta_c f_{ic}$. Since total utilization of device i is $U_i = \sum_c U_{ic}$ the equivalence is clear. Regarding I_c , we can rewrite it as

$$\text{E 48} \quad I_c = \sum_{i \neq b} \frac{1}{D^c_b} \frac{D^c_i}{1 - U_i} - \frac{U_i}{1 - U_i} = \sum_{i \neq b} \frac{R^c_i}{D^c_b} - n_i$$

where n_i is the total number of customers queued at device i while the first ratio, R_{ic}/D^c_b expresses the class c population at device i with a mean demand D^c_b . So I_c has the units of population level. It can be positive or negative. To add more insight to the meaning of E 46, we rewrite R_w as:

$$\text{E 49} \quad R^c_w = N^c D^c_b + \sum_{i \neq b} R^c_i + D^c_b ((N - N^c) - \sum_{i \neq b} n_i)$$

This shows that as N grows to infinity, the response time of a customer in class c is the time spent by that customer waiting for N_c customers at device b , plus the time spent by that customer at other devices (second term), minus the time that customer has to wait for customers belonging to the other classes that are waiting at device b . The latter expression is obtained by multiplying the demand at device b by population belonging to classes other than c minus those customers that are queuing at devices other than b .

4.3.2.1 Scalable Configurations with a Fair Response Time

A fair response time configuration gives equal importance to all classes in the system. A fair response time implies the minimization of maximum over c among the relations E 46, where $c=1\dots C$. In terms of scalability, D^c_b is the most important term and is of most interest because it does not depend on the population mix, whereas I_c depends on the population mix and determines a constant positive/negative shift of the asymptotic line ND^c_b .

Figure 39. Asymptotic response time for two classes, common bottleneck.

Figure 39 presents the contribution of scalability and shift factors to the asymptotic mean response time, for two classes. We observe that per class response time depends on the population level N . At large populations, I_c is a small fraction of a response time so the bottleneck demand is dominant while at small population I_c could make a difference.

We may also detect thresholds as we did for single class systems; thus, in Figure 39, with two classes in a system and a common bottleneck, the coordinate N^* determines two regions:

$$\text{E 50} \quad ND_b^1 + D_b^1 I_1 > ND_b^2 + D_b^2 I_2 \text{ when } N < N^* \text{ and}$$

$$\text{E 51} \quad ND_b^1 + D_b^1 I_1 \leq ND_b^2 + D_b^2 I_2 \text{ when } N \geq N^*$$

We must take into account both terms of E 46 when trying to allocate the objects in a way that offers fair scalable mean response times but we also have to consider that I_c is less important than D_b^c .

The allocation problem for very high population levels may be stated as

$$\text{E 52} \quad \text{minimize maximum}_{c=1,C} (ND_b^c + D_b^c I_c)$$

subject to a feasible allocation

Because the problem makes sense for a single N , and we want to cover a large range of populations, we will heuristically decompose the problem into two phases:

$$\text{E 53} \quad \text{minimize maximum}_{c=1,C} (D_b^c)$$

subject to a feasible allocation;

$$\text{E 54} \quad \text{minimize maximum}_{c=1,C} (I_c D_b^c)$$

subject to a feasible allocation that does not affect bottleneck device.

Regarding E 53, since it does not depend on the population mixes, but only on per class demand, its minimization poses no problems. On the contrary, E 54 depends on the population mix. For a given population mix, we can minimize $I_c D_b^c$ and, for all mixes we can estimate the upper and lower bounds. Determining lower and upper bounds is the subject of the next section.

4.3.2.2 Bounds over I_c

From the response time perspective, our goal is to reduce the response time by making I_c negative. It is not possible for all classes because of [Bal96]:

$$E 55 \quad \sum_c \beta_c I_c = 0;$$

However, we can choose an allocation that makes I_c negative for classes that have high bottleneck demands to the detriment of those with low bottleneck demands.

Next we consider some interesting properties of I_c and hence of a class's asymptotic response time.

1. [Bal96] *For a given configuration, the maximum correction factor I_c has an upper bound of $(K-1)$, over all population mixes.*

The above property follows the definition of I_c where each ratio is less than 1.

2. *The actual upper bound of I_c over all population mixes is the maximum of the function I_c subject of population mix constraints:*

$$\max I_c = \sum_s I_{ic};$$

subject to

$$\sum_s \beta_s = 1$$

$$0 \leq \beta_s \leq 1$$

which is a non-linear program with linear constraints and can be solved with one of the known NLP techniques[Baz93].

The terms I_{ic} are defined by:

E 56

$$I_{ic} = \frac{f_{ic} - f_i^*}{1 - f_i^*}$$

and represents the population level of class c at device i , where $c=1...C$, $i=1...K$.

Similarly,

3. The actual lower bound of I_c over all mixes is the minimum of the function I_c subject to population mix constraints:

$$\min I_c = \sum_s I_{ic};$$

subject to

$$\sum_s \beta_s = 1$$

$$0 \leq \beta_s \leq 1$$

which is a non-linear program with linear constraints and can be solved.

The denominator of E 56 are positive, but the numerator may be either positive or negative.

Figure 40. The relative positions of 1, f_{ic} , f_i^* .

The relative position of 1, f_{ic} , and f_i^* and a graphical representation of I_{ci} versus f_i^* are given in Figure 40 and Figure 41 respectively. Representing the components of the last

sum on the f axis, we note that for a given configuration, the only component that is not fixed is $f^*_i = \frac{\sum_{s=1}^K \beta_s \frac{D^s_i}{D^s_b}}$ which depends on the population mix and can be anywhere within the interval $[0, 1]$. Note also that f_{ic} , i.e. D^c_i/D^c_b , is less than 1 since there is only one bottleneck, namely b .

Figure 41. I_{ic} versus f^*_i .

With the above said, we note that:

4. I_{ic} is positive when f^*_i is less than f_{ic} and it is negative when f^*_i is greater than f_{ic} .
5. I_{ic} is a strictly decreasing function with respect to the f^*_i (it can be noticed by computing its derivatives) and reaches the maximum when f^*_i reaches its minimum.

The following two results show how we can compute bounds on the I_c factor without solving the programming models. However, the bounds are not as tight as those from properties 1 and 2.

Theorem 4. *The following relations hold:*

$$\text{E 57} \quad \min_{c=1,C} f^*_i = \min_{c=1,C} \{D^c / D^c_b\}$$

$$\text{E 58} \quad \max_{c=1,C} I_{ic} = \frac{f_{ic} - \min_{c=1,C} (D^c_i / D^c_b)}{1 - \min_{c=1,C} (D^c_i / D^c_b)} \leq 1$$

Proof in Appendix 8.

Theorem 5. *The following relations hold:*

$$\begin{aligned} \max_{c=1,C} f^*_i &= \max_{c=1,C} \{D^c / D^c_b\} \\ \min_{c=1,C} I_{ic} &= \frac{f^c_i - \max_{c=1,C} (D^c_i / D^c_b)}{1 - \max_{c=1,C} (D^c_i / D^c_b)} \end{aligned}$$

The proof is similar to that of Theorem 4.

The upper (lower) bounds on I_c can be obtained by summing the maximum (minimum) of I_{ic} terms. For example, the upper bound built with Theorem 5 is less than that of Property 1 and higher than that of Property 2. The lower bound built with Theorem 5 is higher than that of Property 3.

To summarize, in this subsection we illustrated some properties of the correction factors that allow us to claim that:

- For a given configuration, we can estimate upper and lower bounds for the correction factor over all population mixes;
- For a given configuration, we can compute accurate upper and lower bounds for the correction factors over all population mixes.

4.3.2.3 Scalable Configurations using Weighted Mean Response Time

For some systems, specific classes are more important than others in term of response times. For these systems, we introduce the *weighted mean response time*. It is defined as:

$$\text{E 59} \quad R_w = \sum_c w_c R_w^c$$

where w_c are weights for classes, their sum is 1 and R_w^c is the asymptotic mean response time of class c . We can rewrite E 46 as

$$\text{E 60} \quad R_w = ND_w + I_w$$

where

$$\text{E 61} \quad D_w = \sum_c w_c D_b^c; \text{ and } I_w = \sum_c w_c D_b^c I_c$$

D_w is the *mean weighted demand* at device b and may be interpreted as the scalability factor. I_w measures the contribution of the other devices to the response time and represents the shift factor. It does not depend directly on population level but on the class population weight.

Figure 42. Asymptotic session response time.

A good allocation requires the minimization of the weighted response time which under asymptotic conditions means the minimization of E 60. An intuitive representation of this equation is shown in Figure 42.

Using the same approach as in the fair response time case, the minimization takes place in two phases, as:

$$\text{E 62} \quad \textit{minimize} (D_w)$$

subject to a feasible allocation ;

$$\text{E 63} \quad \textit{minimize} (I_w)$$

subject to a feasible allocation that does not affect bottleneck device allocation.

D_w , the scalability factor, depends on the weights whereas I_w , the shift factor, depends on both weights and population mix. For a given population mix and set of weights, solving E 62 and E 63 does not pose any problems. However, population mix and weights are not likely to be constant throughout the system's life. For this case we can provide bounds. The following section illustrates such bounds.

4.3.2.4 Bounds for D_w and I_w

The same logic as in the fair response time applies to this case. Both weights and mixes are bounded by the 0 and 1, and each sum is 1. These two conditions become constraints in the optimization problem, either linear or non-linear.

For D_w , the upper and lower limits are given by minimum and maximum of linear programming problems as:

$$1. \max D_w = \max_{c=I,C} \{D_w^c\}$$

$$2. \min D_w = \min_{c=I,C} \{D_w^c\}$$

The bounds on I_w are determined by the non-linear programming problems over population and population mix constraints.

The following section illustrates the case when there is more than one bottleneck in the system.

4.3.3 Response Time in Multi-Class Systems with Multiple Natural Bottlenecks

The goal of this section is to analyze the multiple bottleneck case. It was shown in Chapter 3 that for different per class demands, different population mix yield different bottlenecks in the system. We called the bottlenecks of each class *natural bottlenecks* as opposed to *non-natural bottlenecks* that are generated by the population mixes. To allow an analysis of the multiple bottleneck case, we extend the conjecture of Balbo and Serrazi [Bal97] to accommodate more than 2 classes and to cover non-natural bottlenecks as well.

Consider n -class closed queuing network with distinct bottlenecks and let $\mathbf{B}_1$ the set of all bottlenecks (including natural and non-natural bottlenecks):

- *when the system is loaded with a population mix β that yields the simultaneous saturation of the subset $\mathbf{B} \subseteq \mathbf{B}_1$ of bottlenecks (either natural or non natural), the*

asymptotic mean queue lengths of these devices all grow to infinity while satisfying the following relation:

$$\text{E 64} \quad \lim_{N \rightarrow \infty} \frac{n_i(\tilde{N})}{\sum_{i \in B} n_i(\tilde{N})} = \gamma_i; i \in \mathbf{B};$$

Now, we generalize the asymptotic response time to more than two bottlenecks and consider non-natural bottlenecks. We find that, formally, the response time of a multi-class distributed system under high population level is conditioned by the per class bottleneck devices.

Let $\mathbf{B}$ be the set of bottlenecks for a given population mix. Then:

$$\text{E 65} \quad R_w^c = ND_c + D_c I_c + D_c \Phi_c$$

where D_c is a *mixed demand* at the bottlenecks and represents the scalability factor; I_c is the correction factor to the non bottleneck devices and Φ_c is the correction factor due to the influence of other classes to the bottleneck devices. The latter factors can be interpreted as the shift factors. Numerically [Bal96],

$$\text{E 66} \quad I_c = (K - B) - \sum_{i=1; i \in B}^C \frac{1 - \frac{D_i^c}{\sum_{j=1}^K \gamma D_j^c}}{1 - \sum_{s=1}^C \beta_s \frac{D_i^c}{\sum_{j=1}^R \gamma D_j^c}}$$

$$\text{E 67} \quad D_c = \sum_{j \in B} \gamma D_j^c$$

$$\text{E 68} \quad \Phi_c = \frac{\sum_{j \in B} D_j^c}{D_c} \prod_{j \in B} D_j^c$$

Again, these equations are obtained by taking the limit as $N \rightarrow \infty$ of E 4. Note that I_c is different from the common bottleneck case and contains a term, γ_j , not encountered previously. It expresses the ratio of queue length to the device j , j being a device that may saturate under the workload conditions specified by β . Now, the mixed demand at the bottlenecks is given not only by the per class demand but also by the relative queue length at joint bottlenecks. Also, a new factor Φ_c conditions the asymptotic response time. It has the dimension of population and characterizes the imbalance among the joint bottlenecks.

In the same way as in the common bottleneck case, we decompose the allocation problem by minimizing the scalability term (D_c) first and then the correction factors I_c and Φ_c . However, D_c depends on the population mix and therefore we want to find its bounds for both fair and weighted response. This constitutes the subject of the next section.

4.3.3.1 Scalable Configurations with Fair Mean Response Time

Considering the scalability factor D_c , it is easy to prove the following:

Theorem 6. *Let $\mathbf{B}_1^c$ be the set of all bottlenecks of class c . The maximum and minimum demands of class c at the natural bottlenecks in $\mathbf{B}_1^c$ are the upper and lower bounds for the scalability factor across all classes (see Figure 43).*

Proof in Appendix 9.

The relation E 65 is valid for each set $\mathbf{B}$ in $\mathbf{B}_1^c$ and the maximum is one of the natural bottlenecks.

We can estimate an upper bound of I_c , following the same arguments as in the common bottleneck case. Thus:

Theorem 7. The upper bound of I_c , over the entire space of population mix is $K-1$.

The proof is based on the definition of I_c which has the first term $K-B$ and the rest of terms are less than 1 and greater than zero. It turns out that for each set $\mathbf{B}$ that saturate simultaneously, the upper bound is $K-B$. Over set $\mathbf{B}$, $K-B$ reaches its maximum when there is only one bottleneck in the system, hence the maximum is $K-1$.

The bounds on Φ_c are obtained in a similar manner:

Theorem 8. For a given set $\mathbf{B}$ in $\mathbf{B}_1^c$, the upper and lower bounds of Φ_c are reached with the minimum and maximum of D_c , respectively.

The proof follows from the definition of Φ_c . In the same manner as we did previously, we can generalize the theorem to the entire population mix.

As an observation at this point, while I_c is a small term (even its upper bound is small), Φ_c can have significant influence on R_w^c . However, large values for it mean many bottlenecks and therefore a balance between classes. The balance of bottlenecks reduces the upper bound for the scalability factor.

Figure 43. Asymptotic response time under multiple class and multiple bottlenecks.

Figure 43, shows the extreme asymptotes for class c . For any configuration, the scalability factor D_c is always between two asymptotes denoted by OA and OB which are given by the maximum and minimum demands across all bottleneck of class c (the set of all bottleneck of class c is denoted by B^c_1). The actual asymptote for R_c (R_w^c in the figure) is given by adding the shift contribution of shift factors, $I_c D_c$ and $\Phi_c D_c$. The maximum value for I_c is $K-1$, according to Theorem 7. Based on Theorem 8, if we change the population mix and yield an increase in D_c , this decreases the shift factors as indicated by the arrows that cross the named factors. In the figure we notice also that beyond the population level N^* , OA is higher than the asymptote R_w^c .

The above theorems indicate important properties for multi-class systems with multiple bottlenecks:

- A population mix yields an intermediate value for the scalability factor; this value is between OA and OB in Figure 43;
- The bounds on balance factor Φ_c have an opposite direction with respect to the scalability factor;
- The maximum and the minimum bounds of scalability factor of a class c are given by the 2 bottlenecks of class c ;
 - These bottlenecks are among the natural bottlenecks of other classes that may be joint bottlenecks with the bottleneck of class c (conditions under which joint bottlenecks can exist have been presented in Section 3.3).
- The larger the number of bottlenecks, the smaller I_c .

The main conclusion we can draw so far is the fact that minimizing the maximum per class demand at the natural bottlenecks has a long-term benefit with regard to a fair response time. Even if there are non-natural bottlenecks in the systems, the response time attributed to them is also lower.

Example. To illustrate some of the above results, consider the example introduced in Section 4.2. Figure 44 shows the mean response time versus the population level when the population mixes are shown in the Legend. The response times regard only one configuration, A1 (see Table 8 and Table 9). The worst response time is obtained when the only service required is that indicated by Sc_3. The best response time is given when customers require only Sc_2 service. The results confirm that, for the same configuration, the best and the worst fair response times are achieved for natural beta vectors (one class in the system).

Figure 44. Different population mixes for configuration A1. Natural beta vectors bound the asymptotic response times.

4.3.3.2 Scalable Configurations with Weighted Mean Response Time

The asymptotic weighted mean response time for multiple bottleneck case is:

$$E_{69} \quad R_w = \sum_c w_c R^c_w = N \sum_c w_c D_c + \sum_c w_c D_c I_c + \sum_c w_c D_c \Phi_c$$

The coefficient of N in the above expression is the scalability factor, the rest of the terms are the shift factors. Again, all three terms are bounded and their bounds are mostly dictated by the natural bottlenecks. We do not give the bounds for this case, just illustrate the effect of reducing the demands at the natural bottlenecks on the scalability factor.

The scalability factor can be written as:

$$\text{E 70} \quad \sum_c w_c D_c = \sum_c w_c \sum_j \gamma_j D^c_j = \sum_j \gamma_j \sum_c w_c D^c_j$$

Assuming that a perfect balance per class c across all devices is achievable through an object allocation:

$$\text{E 71} \quad \sum_c w_c D^c_1 = \sum_c w_c D^c_2 = \dots = \sum_c w_c D^c_K = \sum_c w_c D$$

it results that the scalability factor may be written as

$$\text{E 72} \quad \sum_c w_c D_c = \sum_c w_c D (\sum_j \gamma_j) = \sum_c w_c D$$

since $\sum_j \gamma_j = 1$

Expression E 72 shows that by reducing the demands at the bottlenecks (we can do that until we equalize them all to D) the scalability factor is reduced as well.

4.4 Allocation Algorithm for Very High Population Levels

This section shows how to use the properties illustrated in Section 4.3. We refer to the notations introduced in the common bottleneck case. The algorithm we present in this section optimizes the allocation with respect to the fair response time for a given population mix. The multiple bottleneck case and the weighted response time may be treated similarly.

4.4.1 Fair Response Time - Allocation Algorithm

It was shown that a good way to obtain scalable systems is to minimize maximum D_c and I_c as it was stated in E 62 and E 63.

We minimize them in 2 phases: in the first stage we minimize the scalability factor D_c (this is beneficial for all ranges of N at very high population levels) and then we minimize I_c (this is beneficial for specific population mixes). Working on the non-bottleneck devices we minimize the latter factor, and do not move the objects already allocated to device b .

The allocation algorithm we propose is shown in Figure 45. This is a heuristic algorithm, other more advanced algorithms may be implemented as well. The objects that can be moved are ordered in non-increasing order of their maximum per class demands at CPUs and DISKs. The list is scanned and each object is then allocated according to the “first fit” rule: the node with the minimum per class resource demand (according to E 62) is chosen for the deployment of the object. In the case of multiple bottleneck or weighted mean response time, the minimum resource demands may be replaced by the corresponding scalability factors.

A second iteration (Step 2) allows for I_c minimization. This step is necessary if we want to optimize the configuration for a specific population mix. We take the following heuristic approach: first we create a list by ordering the objects not assigned to device b in decreasing order of their per class demands at CPUs and DISKs. Then we select the first object from the list and re-deploy that object if an decrease in I_c is noticed. The reason we start with the objects with the largest demand is that by leaving the low granularity objects to the end, we have a better chance to balance the load.

HEURISTIC_FAIR

Input: LQM, β

Output: A configuration;

Step 1. Allocate objects to nodes to minimize the scalability factor

Step 1.1. Create **List**, that contains the objects, with the elements in the non decreasing order of maximum per class demands at CPU and DISKS

Step 1.2. Consider the first unassigned object from the **List**. IF there are no unassigned objects THEN GO TO step 2

Step 1.3. Assign the object to the node with the minimum per class device demand, update the demands on that node, mark the object as assigned, and GO TO step 1.2

Step 2. Reconfigure object allocation to minimize the maximum per class shift factors I_c . Let b be the bottleneck node:

Step 2.1. Build **ListI**, the list of objects not assigned to b . The list is ordered in decreasing order of maximum per class demands at device type

Step 2.2. Select the first element in the list. IF there is no element, THEN GO TO step 3, ELSE

- **FOR** each node $k \neq b$ compute I_k
- $d =$ the index k that gives $\min\{I_k | k \neq b; D_k < D_b^c\}$
- allocate the object to node d

Step 3. For each node, allocate objects to the local processes to minimize the total number of inter process visits

Figure 45. Allocation algorithm for obtaining fair response times.

Once the objects have been allocated to nodes, a new allocation process starts that allocates objects to processes (Step 3). On each node, the objects are enlisted in a list ordered by the total object number of visits. The list is scanned then and at each step, an object is allocated to a process with a goal of minimizing the total number of inter-process visits.

4.5 Finding Configurations under Medium and High Population Levels

Under medium and even high population level, the worst mean per class response time may not be due to demands at natural bottlenecks. By following the same reasoning as we did with object utilization in Chapter 3, we can show that the maximum mean response time may be given by non-natural bottlenecks and joint bottlenecks. In terms of utilization and population spaces, the maximum per class response time resides at a crossover point for the asymptotic case and on the boundary of the utilization space for the non-asymptotic case. However, in this section we will assume that, under non-asymptotic conditions, the maximum mean response time of a class is at an extreme point which is equivalent to considering the response time as linear with respect to the population mixes. Next chapter will show that the above assumption yields a good heuristic.

The algorithms that find the configurations are presented in Figure 46 and Figure 47. At high population levels, the search toward the optimal configuration starts with the configuration found by HEURISTIC_FAIR algorithm. The objects are moved across the system attempting to improve the *Satisfaction* function.

Figure 47 shows the algorithm ChangeConfig. Step 1 of the algorithm identifies an extreme (or a crossover) point (P1) and a class c that yields the minimum Satisfaction over all classes. Step 2 finds the extreme (or a crossover) point (P2) that yields the maximum satisfaction over all classes. Step 3 tentatively moves objects from the hosts that give P1 to unutilized hosts or to the hosts that give P2. Moving the objects should not bring about a substantial increase in the response times of other classes and henceforth a decrease of their *Satisfaction*. Step 3 is implemented in this thesis as a greedy algorithm: the object with maximum per class c demand that is assigned to one of the hosts that give P1 is moved to the host with minimum utilization in class c . In this way the complexity of ChangeConfig remains polynomial: navigating through extreme and crossover points is similar to that shown in Chapter 3; Step 3 also is polynomial. However, more advanced

algorithms such as simulated annealing [Jok98], or genetic algorithms can implement Step 3. This is subject of further work.

```

Allocation Algorithm (type) // type = High Population, Medium Population
Input: LQM, topologies, performance requirements, design constraints
Output: one configuration, and its satisfaction
Step 1. IF type = High Population THEN
    Get an initial configuration with a HEURISTIC_FAIR Algorithm
ELSE
    Get an initial configuration with Allocation Algorithm (High Population)
ENDIF
Step 2. REPEAT
    Step2.1. FOR each class
        • find the worst per class response time by navigating through
          extreme and crossover points (see Chapter 3)
    Step 2.2. compute the satisfaction of the configuration
    Step 2.3. IF type = High Population THEN
        • change the configuration by applying  $ChangeConf(N^{high})$ 
    ELSE
        • change the configuration by applying  $ChangeConf(b_i N^{medium})$ 
    UNTIL satisfaction is 1 or it can not be improved

```

Figure 46. Algorithm for determining configurations High and Medium population levels.

```

ChangeConfig (constraints)
Input: a configuration
Output: a configuration
Step 1. Let P1 be the extreme point in the programming model space that
produces the minimum per class Satisfaction at one of its corresponding crossover
points and let  $c$  denote that class;
Step 2. Search the direction that produces the greatest decrease in the class  $c$ 
response time and let P2 be the ending extreme point of that direction
Step 3. Move an object from the hosts that give P1 to the hosts that give P2

```

Figure 47. Heuristic algorithm for changing configurations.

At a medium population level, the search starts with the configuration of the high population level. Then the objects are moved by working on the extreme points according to the ChangeConfig algorithm.

4.5.1 An Example

To illustrate the above algorithms, we consider a 3 tier application, described earlier in Figure 36, but which is to be deployed on a different hardware environment. There are 3 classes of request, 3 nodes, 6 processes and 12 objects. Except for objects O_3 and O_4 , which can migrate between **2d Tier** and **3d Tier**, all the other objects in this example are pre-assigned as shown in the Figure 36. CPU factors for **Client Node**, **2d Tier** and **3d Tier** are 1, 1 and 4, respectively and disk factors are 1, 1, 2. Table 11 shows the total and per class demand of each object at CPU and DISK, measured on the **Client Node**.

Table 11. Object demands, measured on the client node; O_i are server objects and Cl_i client objects. Class i denotes the scenario Sc_i .

Object Name	CPU Demand			DISK Demand		
	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3
Cl_1	10	0	0	0	0	0
Cl_2	0	10	0	0	0	0
Cl_3	0	0	10	0	0	0
O_1	60	60	20	22	22	22
O_2	50	50	0	20	20	0
O_3	50	0	5	20	0	20
O_4	50	50	0	20	20	0
O_5	4.99	0	0	11	0	0
O_6	0	0	25	0	0	55
O_7	37.5	0	49.99	11	0	55
O_8	37.5	37.5	0	11	11	0
O_9	0	0	49.99	0	0	55
O_{10}	37.5	0	49.99	11	0	55
O_{11}	37.5	62.5	0	11	11	0
O_{12}	0	75	0	0	11	0

Table 12. Three configurations: C1, C2, C3.

Object	C1	C2	C3
O ₃	2d Tier	2d Tier	3dTier
O ₄	2d Tier	3d Tier	2d Tier

The application should satisfy the following performance requirements and constraints:

- At a medium population level, $N^{medium}=25$ customers; $[R'_{Lo}, R'_{Up}] = [R^2_{Lo}, R^2_{Up}] = [R^2_{Lo}, R^2_{Up}] = [1000, 2000] ms$, $U_{disk\ 2d\ Tier} \leq 0.8$, $U_{CPU\ 2d\ Tier} \leq 0.8$, $U_{disk\ 3d\ Tier} \leq 0.85$, $U_{CPU\ rd\ Tier} \leq 0.85$;
- At high population level, $N^{high}=50$, $[R'_{Lo}, R'_{Up}] = [R^2_{Lo}, R^2_{Up}] = [R^2_{Lo}, R^2_{Up}] = [5000, 7000] ms$, *devices may saturate*.

In the example, the performance requirements are identical for all classes; therefore, to maximize satisfaction, we simply minimize the maximum response time over all classes.

In the process of searching the configuration that maximizes the *Satisfaction*, the allocation algorithm generates three configurations described in Table 12 for both medium and heavy population levels.

Figure 48. The path search through extreme points A, B, C for configuration C1.

At a medium population level, the search starts with configuration C1. Figure 48 describes a path through the extreme points when searching for maximum per Class 1 response time in the utilization space. A, B, C are the extreme points computed, with the coordinates described in Table 13.

Table 13. The coordinates of the extreme points A, B, C. $U_{CPU\ 2d\ Tier, i}$ $i=1,3$ represents the utilization of CPU (as reference device) on **2d Tier** by class i . Sl_i , $i=1,4$ are slack variables and measure the distance from the extreme point to the utilization of device i .

	$U_{CPU\ 2d\ Tier, 1}$	$U_{CPU\ 2d\ Tier, 2}$	$U_{CPU\ 2d\ Tier, 3}$	$Sl_{CPU\ 2d\ Tier}$	$Sl_{DSK\ 2d\ Tier}$	$Sl_{CPU\ 3d\ Tier}$	$Sl_{disk\ 3d\ Tier}$
A	0.85	0	0	0	0.518	0.172	0.577
B	0.82	0	0.027	0	0.482	0	0.342
C	0.36	0.48	0	0	0.519	0	0.604

The maximum response time of class 1 is reached at point B, and this point is used in deciding which object is to be moved. Step 3 of the ChangeConfig algorithm (implemented as a greedy algorithm) moves the object O_4 on **3d Tier**. We obtain configuration C2, and the maximum response time of Class 1 is reduced from 2100ms to 1600ms. The evolution of the maximum per class response times for medium load is presented Figure 49. The next configuration C3, found using Step 3, does not increase the overall satisfaction because response time of Class 3 increases.

Figure 49. Maximum mean response time versus configurations C1, C2, C3.

Figure 50. Maximum asymptotic mean response time versus configurations C1, C2, C3.

Figure 50 shows the response time evolution in the case of heavy loads, for each class of request. Again, the optimal configuration is C2. Class 3 dictates the overall satisfaction giving $S=0.5$

4.6 Summary

This chapter illustrated a method for finding configurations that satisfy a given set of performance requirements. We showed that:

- Configurations should cover low, medium and high population levels;
- Each population level may have its own best configuration.

First we investigated the very high population levels and took into account the population mixes. The performance behaviour for these population levels is dictated mainly by the bottlenecks.

- We studied three cases:

- Single class systems;
- Multiple class systems with a common bottleneck;
- Multiple class systems with multiple bottlenecks.
- For all the cases we showed that:
 - The response time may be decomposed into a *scalability factor* which is the demand or a mixed demand at the bottleneck or the bottlenecks, and other factors (we called them *shift factors*);
 - Beyond a threshold, the scalability factor is dominant over the shift factors.
- The above discoveries allowed us to:
 - Find bounds for the *scalability* and *shift factors*;
 - Decompose the minimization of the fair response time or weighted response time in two separate minimization problems;
 - Develop allocation algorithms that balance the load to improve the fair or weighted mean response times.

Then we studied the performance behaviour at medium and high population levels and the object allocation at these levels. At these population levels we searched for a solution that maximize the overall *Satisfaction* function:

- The *Satisfaction* is given by the maximum per class response time over all population mixes; maximum per class response time is found at an extreme point (medium population level) and crossover point (high population level);
- The search for the extreme and crossover points is based on Non Linear and Linear Programming, ABA and properties of LQM already described in the Chapter 3;
- The configuration is changed based on the characteristics of the extreme points;

- An example that implemented the configuration change through a greedy algorithm was presented.

The main contributions of the author in this chapter are:

- The approach as a whole; many concepts and mathematical instruments are brought together to solve a complex problem;
- The properties and bounds over the per class and weighted response times;
- The upper and lower bounds for the fair and weighted mean response times for a configuration;
- Heuristic algorithms for object allocation with justification at high, medium and low population levels.

5 CASE STUDY: FINDING CONFIGURATIONS THAT BEST SATISFY COMPLEX REQUIREMENTS

As long as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality.

-Albert Einstein in “The Tao of Physics”

The goal of this chapter is to present a case study for a complex application. So far, we generated configurations that satisfy the requirements for specific population levels (we considered only three population areas) and all population mixes. It is clear that a large number of configurations may be generated over a larger set of population levels and topologies. Methods are needed to decide the most suitable configuration or to order the utility of the configurations. Usually, not all workloads are equally important in the life of a distributed system. Some of the workloads are more important or more frequent than others. Furthermore the performance is not the only issue in making decisions on distributed system design. Other issues such as security, cost, and reliability should be taken into account.

This section defines guidelines for dealing with this complex problem within a general decision making framework. We chose to use a decision making approach to cope with the semantic complexity of high performance distributed applications. The guidelines take into account the performance aspects of the configurations generated by our algorithms and configurations specified by the designer. Order relationships are defined for configurations according to performance and non-performance criteria. A designer can

select one or more configurations from the ordered set and proceed with the implementation. Supporting more than one configuration helps to deploy the application in many target environments or even reconfigure it at run time.

The reminder of this chapter is organized as follows:

- Section 5.1 presents a case study;
- Section 5.2 maps the design decisions onto a decision making problem, with a focus on performance criteria and solving this problem;
- Section 5.3 presents conclusions.

5.1 Case Study

Figure 51. The initial object allocation in the insurance application.

We consider a multi-tier Insurance Application, as shown in Figure 51. There are 5 classes of request, 6 nodes (including the client node, this system has 12 CPUs and 6 DISKs), 12 processes and 24 distributed objects. Except for the objects O_{13} to O_{24} , which can migrate between nodes, all the other objects are assigned as shown in Figure 51. The non-movable

objects perform generic functions such as session management, naming, persistence or transaction services, and they reside in processes that are non-mutable as well. The figure also shows an initial allocation of movable objects; this allocation constitutes a start configuration for our search.

CPU and DISK factors for the nodes are presented in Table 14. The CPU factors are used to scale the object demands while DISK factors are the service times. The nodes are heterogeneous. For example, the DISK of Node 1 is 8 times faster than the DISK of Client Node, while the CPU of Node 5 is 10 times faster than that of Client Node and 5 times faster than that of Node 2.

Table 14. CPU and DISK factors.

	Client Node	Node 1	Node 2	Node 3	Node 4	Node 5
CPU	1	0.5	0.2	0.1	0.1	0.1
DISK	8	1	0.2	0.1	0.1	0.1

*The object interaction is the same for all classes; only the demands are different.
Average object CPU demand: uniformly distributed within [10, 200] ms.
Average object number of visits to DISK: uniformly distributed within [1, 8] visits.*

Figure 52. Object Interaction.

The pattern of method invocations is the same for all classes of requests and is presented in Figure 52. However, the classes are different: the per class demand of each object at

each CPU and its visits at each DISK are uniformly distributed within $[10, 200]$ ms interval (measured on the Client Node) and $[1, 8]$ visits, respectively. Whenever an object is moved across the nodes, its per class demands at each CPU are scaled with the factors of Table 14 and its demands at each DISK are computed by multiplying its per class visits with the DISK factors of Table 14. The think mean time for all classes of requests is 20 seconds.

5.1.1 Performance Requirements

The application should satisfy the following performance requirements at four population levels from heavy and medium population areas:

- (a) At a medium population level, $N^{medium}=150$, $[R^1_{Lo}, R^1_{Up}] = [R^2_{Lo}, R^2_{Up}] \dots = [R^5_{Lo}, R^5_{Up}] = [0, 10000]$ ms, devices may saturate;
- (b) At a heavy population level, $N^{heavy}=200$, $[R^1_{Lo}, R^1_{Up}] = [R^2_{Lo}, R^2_{Up}] \dots = [R^5_{Lo}, R^5_{Up}] = [0, 50000]$ ms, devices may saturate;
- (c) At a heavy population level, $N^{heavy}=300$, $[R^1_{Lo}, R^1_{Up}] = [R^2_{Lo}, R^2_{Up}] \dots = [R^5_{Lo}, R^5_{Up}] = [0, 60000]$ ms, devices may saturate;
- (d) At a heavy population level, $N^{heavy}=500$, $[R^1_{Lo}, R^1_{Up}] = [R^2_{Lo}, R^2_{Up}] \dots = [R^5_{Lo}, R^5_{Up}] = [0, 100000]$ ms, devices may saturate.

Since the performance requirements are identical for all classes, the maximization of *Satisfaction* is reduced to minimization of the maximum response time over all classes (see the definition of Satisfaction).

5.1.2 Finding Configurations

Table 15. Four configurations.

	O ₁₃	O ₁₄	O ₁₅	O ₁₆	O ₁₇	O ₁₈	O ₁₉	O ₂₀	O ₂₁	O ₂₂	O ₂₃	O ₂₄
A1	Sv. 5	Sv. 5	Sv. 5	Sv. 5	Sv. 12	Sv. 4	Sv.12	Sv.4	Sv.4	Sv.12	Sv.12	Sv.12
A2	Sv. 5	Sv. 5	Sv. 5	Sv. 5	Sv. 5	Sv. 5	Sv.12	Sv.8	Sv.8	Sv.8	Sv.12	Sv.8
A3	Sv. 5	Sv. 5	Sv. 5	Sv. 5	Sv. 12	Sv. 8	Sv.12	Sv.8	Sv.8	Sv.8	Sv.12	Sv.8
A4	Sv. 5	Sv. 5	Sv. 5	Sv. 5	Sv. 4	Sv. 4	Sv. 4	Sv.12	Sv.12	Sv.12	Sv.12	Sv.12

N=200 (requirement b)

N=500 (requirement d)

Figure 53. Maximum asymptotic mean response time versus configurations (N=200, N=500).

In the process of searching for the configurations that maximize the *Satisfaction*, the **Allocation Algorithm** (Figure 46) generates six configurations for the requirement (a), ten configurations for the requirement (b), ten for (c) and seven for (d). Figure 53 and Figure 54 show the response time evolution per each class of request in the case of the population levels (a) to (d). The configurations that maximize the satisfactions for each performance requirements are denoted by A1 to A4 and are presented in Table 15, with A1 corresponding to requirement (a), A2 corresponding to the requirement (b), and so on.

N=150 (requirement c)

N=300 (requirement a)

Figure 54. Maximum asymptotic mean response time versus configurations (N=150, N=300).

Figure 55. Per class satisfaction; N=500.

The evolution for *Satisfaction* for the search when the population level is 500 is presented in Figure 55.

From this case study we noticed several things:

- First, that the behaviour of each class of request evolves differently as objects move from one server host to another, even if their object demand is uniformly distributed in a narrow interval. Figure 53 (d) shows that, except for class 1, configuration 7 offers the best response time for all classes. For class 1, configuration 6 is the best one. However, since we are interested in the configuration that maximizes the minimum satisfaction, we choose configuration 6 as the best alternative at this population level;
- Each population level may have its own configuration that maximizes the satisfaction (see Table 12). This justifies our claim that the configuration depends on population level;
- A good allocation may substantially reduce the response time of the distributed system as a whole and of each class of requests. For example, at N=200, the response time of class 5 was reduced with about 80%, (see Figure 53 (d)). At N=500, the satisfaction was increased by 20% (see Figure 53).

Table 16. The number of solved LQMs: $N=500$; $K=10$.

C	Possible Mixes/Configuration	LP_ASYMP/ Configuration	Allocation Algorithm
5	664114625	420	2625

Regarding the complexity of the Allocation algorithm, we show in Table 16 the total number of LQM models solved per configuration and for all configurations for $N = 500$. We see that the number of models solved is much lower than those solved by a brute force algorithm even for a single configurations.

5.1.3 Finding Maximum Object Utilization

In the process of determining the configurations we kept the server threading level of the processes (in the LQM model) fixed at the maximum number of customers. As we have shown in previous chapters, this is a gross overestimate. Our next goal is to reduce it to a useful level.

In Chapter 3, we presented LP_ASYMP (Section 3.4.2) as the algorithm for asymptotic case, and NLP_NASYMP (Section 3.4.1) for the non-asymptotic case. LP_ASYMP used only the population level as input (it assumes that the device utilization limits are 1). NLP_NASYMP takes only the device utilization limits as input (the population level is an output of the algorithm). In this section we will use only LP_ASYMP because the performance requirements are defined only for population levels and there are no restrictions on the device utilization limits. At medium population levels, this algorithm acts as a heuristic because we do not have saturated devices.

Table 17 and Table 18 show the numerical levels for maximum object utilization at $N=300$ and $N=150$ respectively. To verify the correctness of these values across all population

mixes, we also solve the associated LQMs at those population levels and for population mixes generated with an increment of 0.2 (we call this procedure the experimental algorithm). The results are presented in Figure 56 and Figure 57. *Error* is the difference between the utilization computed by the LP_ASYMP and the maximum utilization computed experimentally. The minimum and average object utilization across the population mixes are also presented in the figures. As long as *Error* is not negative, the experiment validates LP_ASYMP. Negative values for *Error* indicates an error in finding the maximum.

Figure 56. Object utilization, N=300.

Table 17. Maximum object utilization, $N = 300$.

Obj 1	Obj 2	Obj 3	Obj 4	Obj 5	Obj 6	Obj 7	Obj 8	Obj 9	Obj 10	Obj 11	Obj 12	Obj 13	Obj 14	Obj 15	Obj 16	Obj 17	Obj 18	Obj 19	Obj 20	Obj 21	Obj 22	Obj 23	Obj 24
183	75	56	19	31	31	0.2	0.2	13	17	35	24	12	12	12	11	18	7.0	8.8	0.1	0.3	0.2	9.6	0.3

At high population levels (Figure 56), we notice that:

- LP_ASYMP computes the correct values, the expected result according to the justifications we gave in Chapter 3. The *Error* is zero or positive but small relative to the utilization of the object. For example, we see a positive error at Obj 1, Obj 13, and Obj 14. A positive error indicates that the utilization found by the LP_ASYMP is higher than that found by the experimental algorithm, explained by the fact that the experimental algorithm does not cover the entire space (We used a step of 0.2 to cover the population mix space). To cover the entire population mix space a full factorial combination over real number domain would be needed, and is of course intractable;
- Minimum values for object utilization across the population mixes are very low, close to zero (not visible in the figure). Even the average value tends to be much lower than the maximum and about half of the maximum. With this large variability, without systematic techniques such as those presented in this thesis, it would be difficult to assess the maximum utilization of an object;
- The high utilization of some objects, Obj 1, Obj 2 and Obj 3, indicates that we have saturated devices. Setting the threading or replication at such high levels depends on the functional role of those objects. As discussed in section 3.2.1, unless the objects are queuing centres themselves they forward the user requests to devices or other objects (this is the reason they have a high threading level). Ultimately, with no queuing at the objects, the user requests queue only at devices. In this case there is no reason to set a threading level much higher than the number of hardware devices.
- Setting the threading level at a lower level than that computed forces the object to buffer requests, therefore delaying the propagation of request through the system (which is already saturated). This reduces the unnecessary overhead. If the objects are

queuing centres themselves (they contain IPCs) or their role is to buffer the user requests, then the designer could permit higher threading levels.

Figure 57. Object utilization, N=150.

At medium population levels (Figure 57), we may note that:

- The object utilization levels are lower than those at the high population levels, another confirmation of the fact that the threading level depends on the population level;
- There are positive *Errors*, a confirmation that LP_ASYMP finds maximum utilization for most of the objects;
- There are relatively small negative *Errors*, which means that LP_ASYMP fails to find the maximum utilization for some objects. It happens especially at that objects that have a low maximum utilization for example, Obj 7, Obj 21 and Obj 24 have the utilization less than 1 (see also Table 18) and this do not affect the choice object

activation policy or threading level of the process (they are always integers). We remind that LP_ASYMP is used by us as an heuristic, therefore, these errors are acceptable;

- There is still a large difference between the minimum and maximum object utilization, which again suggest that systematic methods are needed to find the maximum.

Table 18. Maximum object utilization, N=150.

Obj 1	Obj 2	Obj 3	Obj 4	Obj 5	Obj 6	Obj 7	Obj 8	Obj 9	Obj 10	Obj 11	Obj 12	Obj 13	Obj 14	Obj 15	Obj 16	Obj 17	Obj 18	Obj 19	Obj 20	Obj 21	Obj 22	Obj 23	Obj 24
32	13	8.1	2.5	1.9	1.7	0.1	0.1	1.0	0.5	1.0	0.5	1.5	1.5	1.4	1.1	1.	0.5	0.3	0.2	0.3	0.2	0.2	0.1

5.2 Mapping Configuration Choice onto a Decision Making Process

This section covers the mapping between configuration choice and a decision making process. There are five steps the decision-maker (designer) should take

1. Set the alternative configurations and their preference levels;
2. Set the criteria and the weight of each criterion;
3. Rate each alternative against each criterion;
4. Aggregate the rates, the criteria preference level and the weights;
5. Rank the alternatives.

Along with each process, we specify the uncertainties that accompany the decision-making process steps for designing a distributed application.

5.2.1 Alternatives

As was pointed out in Chapter 1, the alternatives in the decision making process are denoted by configurations. The configurations we generated satisfy performance

requirements for only several population levels. Furthermore, there remains uncertainty with regard to the set of all possible configurations owing to:

- Lack of imagination on the part of the analyst;
- Large quantity of material to analyze for the analyst and the designer.

The lack of imagination assumes that there are alternatives that may not be seen by the decision-maker, mainly because of the uncertainties in the specifications. In our case, the lack of imagination is combined with the large configuration space. Let us say again that the configurations we generated were for the performance requirements (according to the satisfaction function) and, eventually, for a target topology. There are many degrees of freedom at each step of software and hardware layout design and they may not be all taken into account either because they are not observable or because they are too numerous. A lack of imagination is more involved in hardware layout design, where there are not any formal techniques to enumerate the entire or the useful set of topologies. Instead, the designer chooses a target or several possible targets based on patterns in industry and the constraints of design specifications. Even for a defined topology, in multi-class object allocation each population mix may lead to a different configuration. Being a continuous space it is impossible to generate all configurations. The result is that we should not focus on eliminating uncertainty in this area; it is not possible to generate all alternatives (this is also true in other decision-making fields), but only to establish a set of representative alternatives.

If the number of alternatives is small enough to allow the designer to examine them, a degree of preference may be applied to each alternative in order to substitute criteria that can not be expressed or quantified.

5.2.2 Criteria for Evaluating Alternatives

The criteria we propose focus mainly on the workload conditions. We require configurations that behave well under different workload conditions.

The performance criteria have the dimensions of the workload:

- Population level;
- Population mix.

Thus, the pairs of population level and population mix make separable criteria for our decision-making process. Each one of the criteria may be weighted to reflect its importance or its probability of occurrence. The weight can come from different sources:

- The designer estimation of criteria importance; or
- The measurement of workloads on existent similar systems.

Also, the weight may be expressed in different ways:

- As marks between 0 and 1, with 1 for very important and 0 for totally non-important criteria;
- Probabilistic measures of occurrences as a probability density function;
- Fuzzy estimation of criteria importance as “important” and “very important.”

Other criteria may intervene as well in our decision making. They may regard economic, functional or security aspects.

5.2.3 Rating Configurations

Performance criteria are measurable by solving the LQM for the workload specified by the criteria. Each alternative satisfies each criteria onto a certain degree (we call it rate).

Expressing the rates in absolute value gives us little chance to process the entire decision table when non-performance criteria and hence heterogeneous criteria are considered. It is necessary to express the rates in relative values on a 0 to 1 scale. Two situations must be considered here:

- The performance requirements are expressed in quantitative terms related to per class or weighted throughput and response time. For example, a set of fuzzy constraints may be associated with each criterion, where the criterion is defined as above. Then the criteria are confronted against the constraints by using fuzzy logic operations and a fuzzy rate is obtained and de-fuzzified later to obtain a scalar within the range [0, 1]. An example will illustrate how to deal with the problem, later in this section. The fundamentals of fuzzy logic can be found in Appendix 10.
- The performance requirements are not expressed as design constraints. Using the worst case configuration as a reference, we simply express the rates as the difference between 1 and the ratio between the actual response time and worst-case response time.

Let us go back to our example defined earlier in this chapter. We determined the configurations A1, A2, A3 and A4 that maximize the Satisfaction and cover four population levels and all population mixes. Now, let us assume that there are 4 population mixes (as shown in the Table 19) more likely to happen and we would like to choose the configuration that best satisfies these mixes.

Table 19. Population mixes.

Mix	1	2	3	4
β_1	0.2	0.5	0.1	0.1
β_2	0.2	0.2	0.2	0.2
β_3	0.2	0.1	0.1	0.5
β_4	0.2	0.1	0.1	0.1
β_5	0.2	0.1	0.5	0.1

Performance requirements are expressed as fuzzy sets as shown in Figure 58, following the same definition we used for Satisfaction. The membership function associated to the class's response time is computed by confronting it against the performance requirement fuzzy set as:

$$\mu(R_i) = \begin{cases} 1 & \text{if } R_i \leq R_i^{lo} \\ \frac{R^{up} - R_i}{R^{up} - R^{lo}} & \text{if } R^{lo} < R_i \leq R^{up} \\ 0 & \text{otherwise} \end{cases}$$

where R^{lo} and R^{up} are the lower and upper values for the response times. For the simplicity of presentation we attached neither the class index nor the class population to the constants of fuzzy sets. However, we assume a fuzzy set defined for each class and each population.

The rate $r_i(a_j)$ at which the alternative a_j satisfies the criterion i , is then equal to $\mu(R_i)$.

Figure 58. Performance requirements defined as fuzzy sets associated with per class response time.

Based on the above considerations, the criteria and configurations give the rates shown in Table 20. The table contains the weights of criteria as well. The configurations are evenly important for the designer and the preferences are all 1.

Table 20. The decision table. A 1, A2, A 3, A 4 are configurations: N is the population level: Mix is the population mix (Table 19) and weight rows and columns specify the weight of criteria and configuration preference respectively.

N		150	150	150	150	200	200	200	200	300	300	300	300	500	500	500	500
Mix		1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
	Weight	1	0.5	0.5	0.5	1	0.5	0.5	0.5	1	0.5	0.5	0.5	1	0.5	0.5	0.5
A 1	1	0.746	0.727	0.768	0.74	0.86	0.85	0.8	0.88	0.699	0.688	0.70	0.71	0.49	0.53	0.49	0.676
A 2	1	0.81	0.82	0.84	0.79	0.91	0.92	0.9	0.89	0.71	0.73	0.78	0.68	0.45	0.47	0.52	0.43
A 3	1	0.81	0.82	0.84	0.79	0.91	0.92	0.9	0.89	0.71	0.75	0.78	0.68	0.45	0.73	0.52	0.43
A 4	1	0.76	0.70	0.78	0.76	0.88	0.81	0.9	0.89	0.68	0.63	0.69	0.81	0.47	0.40	0.44	0.71

5.2.4 Aggregating Criteria, Weights and Rates

Another aspect to clarify is the aggregation of the weights, criteria and rates. The goal of the aggregation is to characterize each alternative by a unique measure that replaces the weights, criteria and rates. There are many aggregation functions in the literature. Baas and Kwakernaak [Baa77] give a method that covers the general case: general fuzzy rates and weights. The main disadvantage of this method is its high complexity. Alternatively, the method proposed by Buckley [Buc85] has a much lower complexity. It deals with fuzzy numbers and the complexity of processing fuzzy numbers is low. An overview of the methods is presented in Appendix 11. The use of one method or another is dictated by the type of fuzzy sets involved in the description of weights and criteria.

5.2.5 Ranking Alternatives

The final step consists of providing the output of the decision process by finding an order relation between the fuzzy numbers and using it to solve the decision making process. There

are many ranking methods available but there is no single method that is universally accepted. Two of them are presented in Appendix 12. For a comprehensive list, see [Lit98-1].

Limiting our discussion to fuzzy numbers, the majority of the ranking methods aim to defuzzify them by mapping them on a real number axis where there is a well defined ordering relation.

$$u_i = \{z, \mu_i(z)\}$$

Assuming s fuzzy subsets u_i , defined on a support S_i , $i \in I = 1, 2, \dots, n$, a simple ranking method defines a function F that projects each fuzzy set on the real time axis

$$F: P(I) \rightarrow R$$

where $P(I)$ is a set composed by all subsets of I and F is chosen such that

$$F(u_i) < F(u_j) \text{ implies } u_i < u_j;$$

$$F(u_i) = F(u_j) \text{ implies } u_i = u_j;$$

$$F(u_i) > F(u_j) \text{ implies } u_i > u_j.$$

Going back to our example, by looking at the way in which we determined the rates of Table 20, we see that the rates are fuzzy numbers with membership function equal to 1. According to Kwakernaak's or Buckley's aggregation methods, the utility functions give the numbers shown in Table 21.

Table 21. Utilities.

Configuration	Utility
A 1	0.714735
A 2	0.728777
A 3	0.742308
A 4	0.70401

Both of these methods choose the alternative A3 as the best, followed by A2, A1 and A4.

5.3 Summary

This chapter presented a case study and illustrated a method for finding configurations that satisfy the performance requirements as well as non-performance requirements.

The case study illustrated the scalability of the algorithms presented in Chapter 3 and 4 and compare the results against the experimental findings. We showed that:

- The Allocation Algorithm (Figure 46) at medium and high population levels substantially reduces per class response time and increases the *Satisfaction* with respect to the initial configuration;
- LP_ASYMP (Section 3.4.2) gives accurate estimation of object order of multiplicity at high population levels;
- LP_ASYMP is a good heuristic for medium population levels for the example presented in the thesis.

The process of choosing configurations that best satisfy the design requirements is mapped onto a multi-criteria decision making process:

- Workloads and their importance are criteria;
- Configurations are alternatives;
- The performance metrics associated with workloads are processed as fuzzy sets and mapped into rates;
- The aggregation is based on fuzzy decision making.

The main contributions in this chapter, are:

- The approach as a whole; many concepts and mathematical instruments are brought together to solve a complex problem;

- The idea of selecting a configuration based on a multi-criteria decision making problem;
- The mapping of configuration(s) selection into a multi-criteria decision making problem;
- The original method for computing the rates by applying fuzzy logic.

6 CONCLUSIONS

Roma locuta est; causa finita est (Rome has spoken; the case is concluded)

St. Augustine of Hippo - Sermons.

This thesis presented a framework for supporting the design and configuration of high performance software distributed systems. The framework accompanies the designers throughout the design and development phase. We rely in our work on several aspects and trends of distributed software engineering:

- The next generation of distributed systems will be developed from objects. CORBA, DCOM and Java RMI are 3 examples of developing distributed technologies;
- Because of the component-oriented structure of distributed software, the concern of the designer will move from implementation issues to economic, performance and security aspects;
- The complexity of the new emerging distributed applications will be high in terms of number of components and their interactions, and it will be difficult to assess the performance behaviour of such applications.

The domain of software engineering that will benefit most from our work is likely to be in the area of transaction systems, for example electronic commerce systems with many configuration alternatives and the need for capacity planning.

In Chapter 2, we presented the first step of the framework. Performance scenarios are captured using a tracing tool named EXTRA and then mapped into a Layered Queuing Model (LQM). The main advantages of this approach are flexibility and the accuracy of

data sampling. Model building was not the main focus of the thesis; however, the method presented demonstrates that commercial tools can be used to support automated model building for the kinds of models we considered in this thesis.

In Chapter 3 we illustrated a methodology for finding an appropriate number of object replicas and an appropriate process threading level for system configurations. The method is based on finding the maximum object utilization under given design or physical constraints over all workloads conditions.

Based on the properties of the object utilization and on physical constraints we observed that finding the maximum object utilization is a non-linear programming problem with linear constraints; this implies that the maximum object utilization may be found on the boundary of the feasibility space; in the asymptotic case, the maximum is reached in the extreme points generated by the constraints. These properties dramatically reduce the search space and allow us to use a systematic procedure in finding the object maximum utilization by using well known procedures. A very important reduction in the search space was also achieved by processing the constraints and by expressing them in terms of per class device utilizations.

To find the objective function, i.e. the object utilization, we mapped the per class device utilizations in per class populations, and solved the corresponding LQM. We showed that there are two methods for mapping:

- Under non-asymptotic conditions, the mapping may be using by a hill climbing method;
- Under asymptotic conditions, the mapping may be done by using the properties of the extreme points.

The properties of extreme points are illustrated throughout the chapter 4. We showed that:

- Under non-asymptotic conditions there is a one-to-one correspondence between per class device utilization and per class population;
- Under asymptotic conditions, extreme points from utilization space are mapped in saturation sectors bounded by crossover points.

Based on the mapping between the extreme points and the saturation sectors, we illustrated the properties of extreme points and crossover points that help to compute these points; the crossover points may be used either for searching the maximum object utilization by a systematic method (simplex algorithm) or a brute force algorithm;

Additionally, we provided:

- Two systematic search algorithms, for asymptotic and non-asymptotic conditions, based on the convex simplex algorithm;
- A straightforward algorithm for finding the crossover points, extended beyond three classes;
- Analysis of the algorithms' behaviour under saturated software entities.

Chapter 4 illustrated a method for finding configurations that satisfy the performance requirements. This was a main focus, motivated by the complexity of applications and by the difficulties faced when trying to use the common sense to find high performance configurations. We showed that:

- Configurations should cover low, medium and high population levels;
- Each population level may have its optimal configuration.

We investigated the very high, high and medium population levels and took into account the population mixes. We studied three cases:

- Single class systems;

- Multiple class systems with a common bottleneck;
- Multiple class systems with multiple bottlenecks.

For all the cases we showed that, at very high population levels:

- The response time may be decomposed into a *scalability factor* and *shift factors*;
- Beyond a threshold, the *scalability factor* is dominant over other factors;

The above discoveries allowed us to:

- Find bounds for the *scalability* and *shift factors*;
- Decompose the minimization of the class response time or weighted response time in two separate minimization problems;
- Develop allocation algorithms that balance the load to improve the fair or weighted response times for large areas of population levels;

At medium and high population level, the configurations are searched by using:

- *Satisfaction* as an aggregated cost function;
- Non-linear and linear programming based search algorithms and performance modeling;
- Extreme points or crossover points properties.

As main contributions of the author in the chapter 4, we mention:

- The approach as a whole; many concepts and mathematical instruments are brought together to solve a complex problem;
- The properties and bounds over the class and weighted response times;
- Heuristic algorithms with justification for object allocation at very high, high and medium population levels.

Chapter 5 presents a methodology for mapping the process of choosing an appropriate configuration onto a multi-criteria decision making. We map the configurations and performance requirements as follows:

- Workloads and their importance are mapped into criteria;
- Configurations are mapped into alternatives;
- The performance metrics associated with workloads are processed as fuzzy sets and mapped into rates;
- The aggregation is based on fuzzy decision making.

As main contributions in Chapter 5, we mention:

- The idea of selecting a configuration based on a multi criteria decision making;
- The mapping of configuration(s) selection into a multi criteria decision making;
- The original method for computing the rates by applying fuzzy logic.

Future work will address several areas of interest presented in this thesis: modeling of the performance requirements, performance model building, object allocation, object replication and multi-criteria decision making. Future work will regard these areas in the same context we have used in this thesis, that is the methods are static and used in the design and implementation phase. However, we do not exclude the use of some of the methods at run-time.

In terms of building LQMs, further work will address the limitations of the current method. Discovering the IPCs, critical sections, remote method invocations in trace files are the main focus in the near future. Addressing this issue in languages other than C++, especially Java, is of interest.

Maximum object utilization work presented in this thesis can be applied in many application areas. We investigated object activation only in CORBA. However, other

major distributed object technologies (Enterprise Java Beans - EJB [Sun98], DCOM and others) need quantitative techniques to decide activation and passivation policies and object replication. For example, EJB comes with passivation policies (beside the traditional activation policy). Deciding what policy to use is entirely the choice of the designer and maximum object utilization can help in this regard.

Further work in object allocation should focus on application of the techniques presented in this thesis to specific distributed software and hardware technologies. The trend in distributed objects is to enable objects to be assembled as components in containers [Sun98] and then the containers into the processes. The grouping is based on performance and object policies (activation, transaction, security, passivation, life cycle, externalization and others). Taking these policies into account in the framework presented is subject of further work. A direct application of the techniques presented and subject of further work regards to object allocation on hardware clusters, the common server hardware solution for complex applications.

Multi-criteria decision making in determining the application can be enhanced to take into account the non-performance criteria such security, reliability, and cost/benefit.

Integrating the whole framework within a friendly graphical user interface (GUI) is the subject of further work as a step towards making the tools acceptable by a larger community of developers. Also, specializing the concepts presented in this thesis to different programming models (EJB, CORBA, CBBConnector and DCOM) is subject of further work.

But the most promising enhancement research area will be in the application of the simplex-like algorithms developed in the thesis to any other cost function the researcher might want, as long as the cost function is strictly monotonic and convex. Throughputs,

queue lengths (for finding the buffer size), linear or nonlinear functions of the performance metrics can be optimized in the same way we did with Satisfaction and object utilization.

Trademarks

* Visual Age is a trademark of IBM Corp.

** Java is a trademark of Sun Microsystems, Inc.

Windows is a trademark of Microsoft Corp.

Other product names may be trademarks of their respective owners.

APPENDICES

Appendix 1. Non-linear programming.

Convex Sets, Polyhedral Sets, Extreme Points, Extreme Directions

Notations:

- The n -dimensional real Euclidean space, composed of all real vectors of dimension n , is denoted by E_n ;
- Matrices and sets are denoted by boldface capital letters, such as A , B , and vectors are denoted by boldface lower case letters, such as g and h .

Definitions:

Convex Sets. A set S in E_n is said to be *convex* if the line segment joining any two points of the set also belongs to the set. In other words, if x_1 and x_2 are in S , then $ax_1 + (1-a)x_2$ must also belong to S for each scalar $a \in [0, 1]$.

Convex Hull. If S is an arbitrary set in E_n , the *convex hull* of S , denoted by $H(S)$, is the collection of all *convex* combinations of S . In other words, $H(S)$ is the minimal convex set that contains S .

Polyhedral set. A set S in E_n is called a *polyhedral set* if it is the intersection of a finite closed half-space; that is, $S = \{x: p_i^t x \leq a_i, \text{ for } i=1 \dots n\}$ where p_i is a nonzero vector, p_i^t is its transpose and a_i is a scalar for $i=1 \dots n$.

Extreme point. A vector x from S is called an extreme point of S if $x = ax_1 + (1-a)x_2$ with $x_1, x_2 \in S$ and $a \in (0, 1)$ implies that $x = x_1 = x_2$.

Extreme directions. A non-zero vector d in E_n is called a direction of S if for each $x \in S$, $x + ad \in S$ for all $a \geq 0$. A direction d of S is called an extreme direction if it cannot be written as a positive linear combination of two distinct directions.

Characterization of extreme points

Let $S = \{x: Ax = b, x \geq 0\}$ where A is a $m \times n$ matrix of rank m , and b is an m vector. A point x is an extreme point of S if and only if A can be decomposed into $[B, N]$ such that:

$$x = \begin{bmatrix} x_B \\ x_N \end{bmatrix} = \begin{bmatrix} B^{-1} \\ 0 \end{bmatrix}$$

where B is an $m \times m$ invertible matrix satisfying $B^{-1}b \geq 0$. Any such solution is called a *feasible solution* for S .

Appendix 2. MVA for multiple class - closed models.

```

FOR  $k=1$  to  $K$   $Q_k(0)=0$ ;
FOR  $n=1$  to  $N$  DO
  FOR each feasible population  $\mathbf{n}=(n_1, n_2, \dots, n_C)$  with  $n$  total customers DO
    BEGIN
      FOR  $c=1$  to  $C$  DO
        FOR  $k=1$  to  $K$  DO  $R_{ck}=D_{kc}(1+Q_k(n-1_c))$ 


$$R_c = \sum_{k=1}^K R_{kc}$$


        FOR  $c=1$  to  $C$  DO  $X_c=n_c/(Z_c+R_c)$ 
        FOR  $k=1$  to  $K$  DO  $U_{kc}=X_c D_{kc}$ 
      FOR  $k=1$  to  $K$  DO


$$Q_k(\mathbf{n}) = \sum_{c=1}^C X_c R_{kc}$$


    END

```

Properties

1. R_{ck} is strictly increasing with regard to U_{rc} , where r is a reference device.

To show that, note that it is enough to show that R_{ck} is increasing with regard to X_c (according to Step 3). Let us assume that we have $\mathbf{n}^*$, X_c^* in the system and we want to increase X_l and keep X_c , $c > 1$ constant. We can increase X_l by increasing n_l or decreasing R_l (see the relation in Step 2). If we increase n_l , then we increase R_{ck} (step 1) and R_c because there are more customers to wait for. However, the increase of R_l is smaller than that of n_l (we are in the non-asymptotic case). But the increase of R_c , $c > 1$ yields a decrease of X_c , $c > 1$, (step 2). To increase X_c , $c > 1$, we increase n_c , $c > 1$, which again increases R_{ck} .

To decrease R_l without changing n_l is possible only by decreasing n_c , $c > 1$. But this will decrease the X_c , $c > 1$, which contradicts the assumption that we keep it constant.

2. R_{ck} is strictly increasing with regard to N_c

The explanation for this behaviour is similar to that of for property 1.

3. A vector $\mathbf{U}_r^*$ is given by only a vector $\mathbf{n}^*$

A sufficient condition for this property requires that the function $\mathbf{U}_r$ is a strict monotone function with respect to $\mathbf{n}$. This property was shown in [Bal91]: although per class utilization of device r may not be a strict monotone, the total device utilization approaches 1 in a strict monotone manner.

Appendix 3. Extreme points and bottlenecks- proof of Theorem 1.

When condition E 30 holds, we may prove by contradiction that only natural bottlenecks may reach their utilization limits.

Let m be the non-bottleneck device that saturates. We have:

$$\frac{D_{m1}}{D_{11}}U_{11} + \frac{D_{m2}}{D_{12}}U_{12} + \dots + \frac{D_{mC}}{D_{1C}}U_{1C} = b$$

Since $D_{m1}/D_{11}, D_{m2}/D_{12}, \dots, D_{mC}/D_{1C}$ are smaller than 1, it means that

$U_{11} + U_{12} + \dots + U_{1C} > b$ that contradicts one of the bottleneck constraint.

Appendix 6. Simplex algorithm implementation.

SimplexAlgorithm()

Input Data:

C: the number of classes, including the slack class;

K: the number of devices;

O: the number of objects;

D_{ij}^k : the demand of object k to the device i due the class j ;

$D_{ij} = \sum_k D_{ij}^k$: the total demand of the class j to the device i ;

LQM: the Layered Queuing Model for the system;

$b_i, i=1, \dots, K$, the upper limits of devices utilization;

Output: Maximum utilization of objects;

BEGIN

Step 1. Find a device used by all classes; Let it be r and let $U_{r1} \dots U_{rC}$ be the utilization of this device with respect to the classes $1 \dots C$. Let $U_{rC+j}, j=1, K$ the slack variables.

Step 2. Determine the coefficients

$a_{ij} = D_{ij}/D_{rj}; i=1, K; j=1, C;$

$a_{ij} = 1; i=1, K; j=C+i;$

$a_{ij} = 0$; for the rest

Step 3. Determine the coefficients $c_j = 1/D_{rj}; j=1, C;$

Let $\mathbf{AU} = \mathbf{b}$, the matrix notation of the system constraints

$$\sum_{j=1}^{C+K} a_{ij} U_j = b_i; i=1, K; \text{ where } \mathbf{U} = (U_{r1}, \dots, U_{rC}, \dots, U_{rC+K})^T$$

Step 4. FOR each $o = 1$ TO O DO

BEGIN

Set $t=1$ (t denote the iteration), $U_{rj}^t=0, j=1..K$, such that: $\mathbf{AU} = \mathbf{b}$ and $z^t=0$;

Step 4.1. Select I_t =the set of the K largest components of $\mathbf{U}^t$, and let:

$\mathbf{A}_B = \{a_{ij}; j \in I_t\}, \mathbf{A}_N = \{a_{ij}; j \notin I_t\}$

$\mathbf{Y} = \mathbf{A}_B^{-1}$; $\mathbf{A}_N, y_j$ is a column and y_{ij} is an element of $\mathbf{Y}$.

$\mathbf{d} = \text{BestDirection}(\mathbf{I}_t, \mathbf{U}^t, \mathbf{P}^t)$; The components of $\mathbf{D}$ are as follows:

$d_i = 1$ if $i = m$; m is the direction to search;

$d_i = 0$ if $i \in I_t$ and $i \neq m$

$d_i = -y_{ik}$ if $i \notin I_t$ and $i \neq m$

IF there is no direction to go (all components of $\mathbf{d}$ are null) THEN stop;

ELSE continue with next step.

Step 4.2. $\lambda = \text{MaximumDistance}(\mathbf{I}_t, \mathbf{d}, \mathbf{P}^t)$;

$\mathbf{U}^k = \mathbf{U}^t + \lambda \mathbf{d}$;

GOTO 1.

END

Figure 61. Simplex algorithm for finding the maximum utilization of an object.

BestDirection (I_t , U^t , P^t)

Input: the basis and the parameters of the current point(P^t).

Output: a vector d with the components as follows:

$d_i = 1$ if $i = m$; m is the direction to search;
 $d_i = 0$; if $i \in I_t$ and $i \neq m$
 $d_i = -y_{it}$ if $i \notin I_t$ and $i \neq m$

BEGIN

Step 1. FOR each $j \in I_t$ compute

$\Delta U_0 / \Delta U_j$

$\alpha = \max_{j \in I_t} (\Delta U_0 / \Delta U_j)$ and p an index such that $\Delta U_0 / \Delta U_p = \alpha$;

$\beta = \min_{j \in I_t} (\Delta U_0 / \Delta U_j \cdot U_{ij})$ and q an index such that $\Delta U_0 / \Delta U_q \cdot U_{iq} = \beta$

Step 2. IF ($\alpha = \beta = 0$) THEN

$d_i = 0$, $i = 1..K + C$, and RETURN (d , 0) ;

ELSE continue to step 3.

IF ($\alpha > \beta$) THEN

$d_i = 0$ if $j \in I_t$, $j \neq p$;

$d_i = 1$ if $j \in I_t$, $j = p$;

index= p

IF ($\alpha < \beta$) THEN

$d_i = 0$ if $j \in I_t$, $j \neq q$;

$d_i = -1$ if $j \in I_t$, $j = q$;

index= q

IF ($\alpha = \beta > 0$) THEN

$d_i = 0$ if $j \in I_t$, $j \neq q$;

$d_i = -1$ if $j \in I_t$, $j = q$;

index= q

Step 3. $d_i = -y_{it}$ if $i \notin I_t$;

RETURN (d , index)

END

Figure 62. Computation of a search direction.

MaximumDistance (I_t , d , index, P^t)

Input: the basic variables(I^t), the direction(d), the index, and the parameters of the current point(P^t).

BEGIN

IF $d[\text{index}] > 0$ THEN

IF $(\exists k, \text{ such that } y_k > 0)$ THEN

$\Delta = \text{minimum } (U_{ri} / y_{it}, y_{it} > 0);$

ELSE

$\Delta = \text{Infinite}$

IF $d[\text{index}] < 0$ THEN

IF $(\exists k, \text{ such that } y_k < 0)$ THEN

$\Delta = \text{minimum } (\max_i (-U_{ri} / y_{it}, y_{it} < 0), U_{ri});$

ELSE

$\Delta = \text{Infinite}$

Find λ by solving the following problem:

Max $z(P^{t+1}) = z(P^t + \lambda d)$

subject to

$0 \leq \lambda \leq \Delta$

RETURN(λ)

END

Output: a scalar λ , the distance to go;

Figure 63. Computation of a point in the utilization space.

Appendix 7. Complexity of *Crossover_Points_Algorithm* (see Figure 26).

The time complexity of the algorithm (we will call it $\Omega(\cdot)$) is high because the number of extreme points is combinatorial. However, the programming model size for many distributed systems models is small enough to allow us to perform the crossover points in a short period of time.

We will study the algorithm complexity for each of the three cases discussed above.

Case 1 (all classes have the same bottleneck). In this case, there is no need to perform Step 3;

Complexity of Step 1 is CK ;

Complexity of Step 2 is C ;

So, the complexity of the algorithm is polynomial.

Case 2 (all classes have distinct bottlenecks). This is the most complex situation, and it requires Step 3 to be performed in order to detect all saturation sectors. The complexity is:

$$\Omega(\text{step } 3) = \sum_{i=2}^C \binom{C}{i} \left(iK + \binom{K}{i} i^2 \right); \text{ where } i^2 \text{ is the complexity of solving a system of } i \text{ equations with } i \text{ unknowns;}$$

Case 3. (Common and distinct bottlenecks). The complexity in this case depends on the distribution of common bottlenecks. Intuitively, the number of iterations in Step 3 should decrease considerably due to the application of the rules discussed in Section 3.3.3.

The complexity of Steps 1 and 2 is the same as in case 2. To simplify the presentation, let us assume that the number of classes is uniformly distributed throughout the sets C_b . The complexity is:

$$\Omega(\text{step } 3) = \sum_{i=2}^B \binom{B}{i} \prod_{j=1}^i C_j \left(iK + \binom{K}{i} i^2 \right);$$

Appendix 8. Bounds- proof of Theorem 4.

E 57 can be found by solving

$$\begin{aligned} \min f^*_i &= \min \sum_s \beta_s f_{is} \\ \text{subject of} \\ \sum_s \beta_s &= 1 \qquad 0 \leq \beta_s \leq 1 \end{aligned}$$

which is a linear programming problem (linear objective function and linear constraints). It reaches its minimum when at one of the extreme points, i.e when one variable is 1.

Hence

$$\min f^*_i = \min \sum_s \beta_s f_{is} = \min_s (f_{is}) = \min_s (D^c_i / D^c_b)$$

E 58 is the direct result of the strictly decreasing property of I_{ic} and of E 57.

Appendix 9. Bounds- proof of Theorem 6.

By definition, D_c is linear with respect to the relative queue lengths, the maximum and minimum of the D_c are equal to the maximum and minimum of D_{ic} respectively, i.e:

$$\max(\min) D_c = \max(\min) \sum_{s \in B} \gamma_s D^c_s$$

$$\text{subject to } \sum_{s \in B} \gamma_s = 1$$

$$0 \leq \gamma_s \leq 1$$

whose solutions are in one of the extreme points, i.e:

$$\max(\min) D_c = \max(\min)_{s \in B} \{D^c_s\}$$

Appendix 10. Fuzzy sets, fuzzy rules, fuzzy algorithms.

The fuzzy set. If F is a set and μ_A is a function defined as

$$\mu_A: F \rightarrow [0, 1]$$

then μ_A specifies a set A , called fuzzy set. F denotes the universe of discourse and μ_A a membership function.

The operators $\vee$ and $\wedge$. These operators are defined on the same universe of discourse. If A_1, A_2 are fuzzy sets of a set F then

$$C = A_1 \vee A_2 \Leftrightarrow \forall x \in F \Rightarrow \mu_C(x) = S[\mu_{A_1}(x), \mu_{A_2}(x)];$$

$$P = A_1 \wedge A_2 \Leftrightarrow \forall x \in F \Rightarrow \mu_P(x) = T[\mu_{A_1}(x), \mu_{A_2}(x)];$$

S and T are a conjugate class of t-norm and t-conorm and

- (i) $T(a, 1) = a$;
- (ii) $a \leq b$ and $c \leq d \Rightarrow T(a, c) \leq T(b, d)$;
- (iii) $T(T(a, b), c) = T(a, T(b, c))$;
- (iv) $S(a, b) = 1 - T(1 - a, 1 - b)$;

C and P are called intersection and union of the fuzzy sets A_1 and A_2 .

Examples of t-norm and t-conorm:

(i) minimum:

$$T(a, b) = \min(a, b)$$

$$S(a, b) = \max(a, b)$$

(ii) product:

$$T(a, b) = ab$$

$$S(a, b) = a + b - ab$$

(iii) drastic product

$$T(a, b) = \begin{cases} \min(a, b) & \text{if } \max(a, b) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$S(a, b) = \begin{cases} \max(a, b) & \text{if } (\min(a, b) = 0) \\ 1 & \text{otherwise} \end{cases}$$

The $\times$ operator. Let A be a fuzzy set of F_1 and let B be a fuzzy set of F_2 . Then

$$C = A \times B \Leftrightarrow \forall (x, y) \in G_1 \times G_2 \Rightarrow \mu_C(x, y) = T[\mu_A(x), \mu_B(y)];$$

Fuzzy logic connectives: Logic connectives **AND** and **OR** are defined with the operation intersection and union.

Fuzzy rules of inference. There are three inference rules (**IF A THEN B**) most used in the applications:

Kleen-Dienes: $A \rightarrow B = (1 - A) \vee B$;

Zadeh: $A \rightarrow B = (A \wedge B) \vee (1 - A)$;

Mamdani: $A \rightarrow B = A \wedge B$

Appendix 11. Aggregation functions.

Bass and Kwakernaak [Baa77]. Let $A_1, A_2, \dots, A_m$ be the configurations (alternatives) we want to compare and let $W_1, \dots, W_n$ be the criteria they have to accomplish. Let also assume that the rates may be fuzzy sets and represented by membership functions $u_{ij}(r_{ij})$, where r_{ij} takes discrete real values (we restrict our study to discrete values) and rates how alternative A_i accomplishes criteria W_j . The criteria weight are fuzzy sets as well and expressed by membership functions $u_{wj}(w_j)$, where w_j takes real values. We assume also that membership functions are normal (there is at least a point in the support of the functions where the membership reaches the value 1).

A function g projects R^{2n} on R :

$$g(z) = \frac{\sum_{j=1}^n w_j * r_j}{\sum_{j=1}^n w_j}$$

for each point $z = (w_1, w_2, \dots, w_n, r_1, r_2, \dots, r_n)$;

For each z defined above, a membership function is defined as :

$$u'_{zi}(z) = \left[\bigwedge_{j=1}^n u_{wj}(w_j) \right] \wedge \left[\bigwedge_{k=1}^n u_{ik}(r_k) \right]$$

where i defines the alternative index and $\wedge$ represents the intersection between 2 fuzzy sets (the intersection is defined as a minimum). Function $g: R^{2n} \rightarrow R$, and its associated membership function maps the pairs (z, u'_{zi}) induces a new fuzzy set (r, u_{zi}) where the membership function is defined by

$$u_{zi}(r) = \max_{z: g(z)=r} u'_{zi}(z), r \in R.$$

In this way, each alternative is described by a fuzzy set. Note that considering the whole space of possible combinations generated this fuzzy set and crediting each value involved into the computation.

Processing the initial fuzzy sets with the above formulas is explained by the following reasons:

- for each set of values $w_1, w_2, \dots, r_{i1}, r_{i2}, \dots, r_{in}$ (with i specifying the alternative) the resulting membership function takes the pessimistic approach by reflecting the value with the minimum membership function (either the rate or the weight) and that is why the membership is defined as:

$$\left[\bigwedge_{j=1}^n u_{wj}(w_j) \right] \wedge \left[\bigwedge_{j=1}^n u_{ij}(r_{ij}) \right]$$

Similarly, second expression selects the maximum membership function for the real values of the support that overlap.

Arithmetic mean value [Buc85]. The method proposed by Buckley considers multiple values for the rates and weights since they are the result of a brainstorm. Although it is not the case with performance criteria it

could be the case with nonperformance criteria since they are the subject of human reasoning rather than of a formal computation method.

A "trapezoidal" fuzzy rate as defined by the k-th expert is

$$r_{ij}^k = (a_{ij}^k / b_{ij}^k, c_{ij}^k / d_{ij}^k)$$

In the same way, the weight is

$$W_j^k = (e_j^k / f_j^k, g_j^k / h_j^k)$$

The nice properties of trapezoidal fuzzy sets are given by the '+' si '.' operators which are associative and distributive. Also, it can be easy shown that

$$(a_1/b_1, c_1/d_1) + (a_2/b_2, c_2/d_2) = (a_1+a_2/b_1+b_2, c_1+c_2/d_1+d_2);$$

and

$$(a_1/b_1, c_1/d_1) \cdot (a_2/b_2, c_2/d_2) = (a[L_1, L_2]/b, c/d[U_1, U_2]);$$

where,

- the last fuzzy set may be represented as : zero at the left side of a , $L_1y^2 + L_2y + a = x$ on $[a, b]$, a horizontal line between $(b, 1)$ and $(c, 1)$, $U_1y^2 + U_2y + d = x$ on $[c, d]$ and zero to the right of d (x and y are the horizontal and vertical axis respectively);
- $a = a_1a_2$; $b = b_1b_2$; $c = c_1c_2$; $d = d_1d_2$
- $L_1 = (b_1 - a_1)(b_2 - a_2)$; $L_2 = a_2(b_1 - a_1) + a_1(b_2 - a_2)$;
- $U_1 = (d_1 - c_1)(d_2 - c_2)$; $U_2 = -[d_2(d_1 - c_1) + d_1(d_2 - c_2)]$;

First it should be averaged the rates and weights over the experts:

$$m_{ik} = (1/n) \cdot [a_{i1}^k + a_{i2}^k + \dots + a_{in}^k] \text{ and}$$

$$n_k = (1/n) \cdot [b_{k1} + \dots + b_{kn}]$$

It results that A_i may be described by :

$$r_i = (1/KL) \cdot [(m_{i1} \cdot n_1) + \dots + (m_{ik} \cdot n_k)]$$

The division at L was done because it was assumed that the support of weights is on $[0, L]$.

Appendix 12. Ranking methods.

a. Buckley's method.[Buc85] The numbers are compared using the following order relation:

$A \succ B$ (A is greater than B) if $e_{AB} > e_{BA}$ and $e_{AB} = 1$, where

$$e_{AB} = \max_{a \geq b} [\min(\mu_A(a), \mu_B(b))]$$

If neither $A \succ B$ nor $B \succ A$ then $A \equiv B$.

b. Baas and Kwakernak's method.[Baa77] If A and B are 2 fuzzy sets to be compared, firstly, the fuzzy conditional set (I, μ_{IR}) is determined:

$$\mu_{IR}(i|x_a, x_b) = \begin{cases} 1 & \text{if } x_a \geq x_b, i = A, B \\ 0 & \text{otherwise} \end{cases}$$

Secondly, the fuzzy function (i, μ_i) is computed with the membership function:

$$\mu_i(i) = \sup_{x_A, x_B} [\min((\mu_{IR}(i|x_A, x_B), \min(\mu_A(x_A), \mu_B(x_B))), i = A, B]$$

The number with the greatest μ_I is the greatest fuzzy number and this membership function shows the extent to which this assertion is true.

REFERENCES

- [Baa77] Baas S.M., Kwakernaak H., *Rating and Ranking of Multiple-Aspect Alternatives Using Fuzzy Sets*, Automatica, Vol 13, pp. 47-55, 1977.
- [Bal97] Balbo G., Serazzi G. *Asymptotic Analysis of Multiclass Closed Queuing Networks: Multiple Bottlenecks*, Performance Evaluation Journal, Vol. 30, No. 2, pp. 115-152, July 1997.
- [Bal96] Balbo G., Serazzi G. *Asymptotic Analysis of Multi-Class Closed Queuing Networks: Common Bottlenecks*, Performance Evaluation Journal, Vol. 26, pp. 51-72, 1996.
- [Bal91] Balbo G., Serazzi G., *Asymptotic Analysis of Multiclass Closed Queuing Networks: Multiple Bottlenecks(Part II)*, Internal Report, Politecnico di Milano, 1991.
- [Bal90] Balbo G., Serazzi G., *On The Asymptotic Behaviour of Multiclass Closed Queuing Networks (Part I)*, Internal Report, Politecnico di Milano, 1990.
- [Baz93] Bazaraa M.S., Sherali H.D, Shetty C.M., *Non-linear Programming*, John Wiley & Sons, 1993.
- [Buc85] Buckley J. J., *Ranking Alternatives Using Fuzzy Numbers*, Fuzzy Sets and Systems, No 15, pp. 234-241, 1985.
- [CBCo] *CBConnector*, <http://w3.software.ibm.com/ad/cb/>
- [Cha93] Champeaux, D., Lea D., Faure P., *Object Oriented Systems Development*, Addison -Wesley Publishing Company, 1993.
- [Cha82] Chandy K.M., Neuse D., *A Heuristic Algorithm for Queuing Networks Models of Computing Systems*, CACM, pp. 126-133, February 1982.
- [Chv83] Chvatal V., *Linear Programming*, W.H. Freeman and Company, 1983.

- [Col94] Coleman D. et al., *Object-Oriented Development, The Fusion Method*, Prentice Hall, 1994.
- [Dec95] December J., *HTML and CGI Unleashed*, Sam's publishing, 1995.
- [Dec96] December J., *Presenting Java*, Sam's publishing, 1996.
- [Eag86] Derek E., Lazowska E., Zahorjan J., *Adaptive Load Sharing in Homogeneous Distributed Systems*, IEEE Trans. On Software Eng., Vol. SE-12, pp. 662-675, May 1986.
- [ElR97] ElRayess A., Rolia J., *Automatic Generation of Performance Models Using the Distributed Management Framework*, Proceedings of CASCON '97, November, Toronto, pp. 58-71, November 1997.
- [ElR98] ElRayes F., Rolia J., Friedrich R., *The Performance Impact of Workload Characterization for Distributed Applications using ARM*, Proceedings of the Computer Measurement Group (CMG) '98, Anaheim, California, USA, pp. 821-830, December 1998.
- [Gal82] Gali Z., Micali S., Gabow H., *Priority Queues with Variable Priority and an $O(EV \log V)$ Algorithm for Finding a Maximal Weighted Matching in General Graphs*, Proc. 23rd Annu. Symp. Foundations Comp. Sc., pp. 255-261, November 1982.
- [Har94] Harold W., Lockhart J., *OSF DCE Guide to Developing Distributed Applications*, McGraw-Hill 1994.
- [Hil95]. Hills G., Rolia J., and Serazzi G., *Performance Engineering of Distributed Software Process Architectures*, Lecture Notes in Computer Science, Springer-Verlag, No. 977, pp. 79-85, 1995.
- [Hri95] Hrischuck C.E., Rolia J., Woodside C.M., *Automatic Generation of a Software Performance Model Using an Object Oriented Prototype*, Proceedings of MASCOTS'95, Durham, North Carolina, pp. 399-409, January 1995.
- [Ign94] Ignizio J.P., T.M. Cavalier, *Linear Programming*, Prentice Hall, 1994.
- [ISO] ISO/IEC JTC1/SC21/WG7 N885, *Reference Model for Open Distributed Processing-Part 1: Overview and Guide to Use*, November 1993.
- [Jai91] Jain R., *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991.

- [Jog98] Jogalekar P., *Scalability in Distributed Systems*, PhD thesis, Carleton University, Ottawa, 1998.
- [Kir83] Kirkpatrick, S., *Optimization by Simulated Annealing*, Science, Vol. 220, pp. 671-680, 1983.
- [Kra86] Kravitz S. A., Rutenbar R.A., *Multiprocessor-Based Placement by Simulated Annealing*, Proc. 23rd Design Autom. Conf., pp. 234-245, March 1986.
- [Kris99] Krishnamurthy K., Litoiu M., Rolia J., *Performance Stress Test and Capacity Planning for E-business Applications*, To be published in the Proceedings of the International Symposium on Electronic Commerce, Beijing, China, May 1999.
- [Lai94] Lai H., *Fuzzy Multiple Decision Making*, Springer-Verlag, 1994.
- [Laz84] Lazowska et al., *Quantitative System Performance, Computer Systems Analysis Using Queueing Network Models*, Prentice-Hall, 1984.
- [Li98] Li J., Kameda H., *Load Balancing Problems for Multiclass Jobs in Distributed/Parallel Computer Systems*, IEEE Transactions on Computers, Vol. 47, No. 3, pp. 322-332, March 1998.
- [Lip82] Lipsky L., Lieu C. H., Tehranipour A., Lievfoort A., *On the Asymptotic Behaviour of Time-Sharing Systems*, Communications of the ACM, Vol. 25, No. 10, pp. 707- 714, October 1982.
- [Lit99] Litoiu M., Tadei R., *Fuzzy Scheduling with Application to Real-Time Systems*, To appear in Fuzzy Sets and Systems, 1999.
- [Lit99-1] Litoiu M., Tadei R., *Decentralized Decision Making on Distributed Real Time Systems Using Self Learning Fuzzy Algorithms*, To appear in Fuzzy Sequencing and Scheduling, Springer-Verlag, 1999.
- [Lit99-2] Litoiu M., Krishnamurthy D., Rolia J., *Performance Testing for Distributed Object Applications*, To appear in Proceedings of the ICSE 99, Workshop on Testing Distributed Component-Based Systems, Los Angeles, May 17, 1999.
- [Lit98] Litoiu M., Ionescu T., Labarta J., *Dynamic Task Scheduling in Distributed Real Time Systems Using Fuzzy Rules*, Microprocessor and Microsystems, Vol. 21, pp. 299-311, 1998.
- [Lit98-1] Litoiu M., Tadei R., *Real-Time Task Scheduling with Fuzzy Deadlines and Processing Times*, To appear in Fuzzy Sets and Systems, 1999.

- [Lit98-2] Litoiu M., Rolia J., Serazzi G., *Designing Process Replication and Threading Policies: A Quantitative Approach*, Lectures Notes in Computer Science, No 1462, pp. 15-26, 1998.
- [Lit98-3] Litoiu M., Rolia J., Serazzi G., *Designing Process Replication and Activation, A Quantitative Approach*, Accepted for publication in IEEE Transactions on Software Engineering, 1998.
- [Lit97] Litoiu M., Tadei R., *Real Time Scheduling with Fuzzy Deadlines*, European Journal of Operational Research, No 100, pp.475-481, 1997.
- [Lit97-1] Litoiu M., Khafagy H., Qin B., Rass An.W. and Rolia J., *A Performance Engineering Tool and Method for Distributing Applications*, Proceedings of CASCON'97, pp. 82-95, November 1997.
- [Lo88] Virginia M. L., *Heuristics Algorithms for Task Assignment in Distributed Systems*, IEEE Trans. on Computers, Vol. 37, No. 11, pp. 1384-1397, 1988.
- [Loo98] Loosley C., Douglas F., *High Performance Client Server*, John Wiley & Sons, 1998.
- [Lu96] Lu C., Lau S.-M., *An Adaptive Load Balancing Algorithm for Heterogeneous Distributed Systems with Multiple Task Classes*, IEEE Proceedings of the 16th ICDCS, pp. 629-636, 1996.
- [Lue97] Luethi J., Majumdar S., Kotsis G., Haring G., *Performance Bounds for Distributed Systems with Workload Variabilities and Uncertainties*, Parallel Computing (Special Issue: Distributed and Parallel Systems: Environments and Tools), North Holland, Vol. 22, No. 13, pp. 1789-1806, March 1997.
- [Mcm70] McMullen P., *The Maximum Number of Faces of a Convex Polytope*, Matematika 17, pp 179-184, 1970.
- [Maj97] Majumdar S., Woodside C.M., *"Robust Bounds and Throughput Guarantees for General Closed Multiclass Queueing Networks"*, Performance Evaluation, 1997.
- [Nei95] Neilson J.E., Woodside C.M., Petriu D.C., Majumdar S., *Software Bottlenecking in Client-Server Systems and Rendezvous Networks*, IEEE Transactions on Software Engineering, Vol. 21, No.9, pp. 776-782, September 1995.

- [Nis97] Nishio N., Tokuda H., *Simplified Method for Session Coordination Using Multi-level QOS Specification and Translation*, Proc. 5th International Workshop on Quality of Service, New York, pp. 211-222, May 1997.
- [Oliv93] Oliver I., Hoon C. Trivedi K.S., *Performance Evaluation of Client-Server Systems*, IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 11, November 1993.
- [Orf97] Orfali R., Harkey D., *Client/Server Programming with JAVA and CORBA*, John Wiley & Sons, 1997.
- [Ped93] Pedricz W., *Fuzzy Control and Fuzzy Systems*, John Wiley & Sons, 1993.
- [Pri90] Price C., Saloma M., *Scheduling of Precedence Constraints Tasks on Multiprocessors*, The Journal of Computer, No 4, pp. 32-52, 1990.
- [Rei80] Reiser, M., Lavenberg S.S., *Mean Value Analysis of Closed Multi-Chain Queuing Networks*. Journal of ACM, 27(2), pp. 313-322, April 1980.
- [Rie96] Riedl R., Richter L., *Classification of Load Distribution Algorithms*, IEEE Proceedings of PDP'96, pp. 404-413, 1996.
- [Rol95] Rolia J., Sevcik K., *The Method of Layers*, IEEE Transactions on Software Engineering, Vol. 21, No. 8, pp. 689-700, August 1995.
- [Ros91] Ross K. W., Yao D. D., *Optimal Load Balancing and Scheduling in a Distributed Computer Systems*, Journal of the Association for Computing Machinery, Vol. 38, No 3, pp. 676-690, July 1991.
- [Sei96] Seigel et al, *CORBA Fundamentals and Programming*, John Wiley & Sons, 1996.
- [Shi89] Shin K., Chang Y., *Load Sharing in Distributed Real Time Systems with State-Change Broadcasts*, IEEE Transaction on Computers, Vol 38, No. 8, pp 345- 356, January, 1989.
- [Smi90] Smith C.U., *Performance Engineering of Software Systems*, Addison-Wesley, 1990.
- [Sip98] Siperski C., *Component software : beyond object - oriented programming*, Addison- Wesley, 1998.

- [Sou92] E. de Souza e Silva, R.R Muntz, *A Note on the Computational Cost of the Linearizer Algorithm for Queuing Networks*, IEEE Transactions on Computers, Vol. 39, No. 6, pp 840-842, June 1992.
- [Sun98] <http://www.javasoft.com/products/ejb/>
- [Ste97] Steinmetz R., Wolf L.C., *Quality of Service: Were are We?*, Proc. 5th International Workshop on Quality of Service, New York, pp. 211-222, May 1997.
- [Zad65] Zadeh L.A., *Fuzzy Sets*, Information & Control, No. 8, pp. 338-353, 1965.
- [Zhu97] Zhu W., Liang T-Y, Shieh C-K, *Optimal Task Clustering using Hopfield Net*, IEEE Proceedings, pp. 451-464, 1997.
- [Zou88] Zou, S., *A trace-Driven Simulation Study of Dynamic Load Balancing*, IEEE Transactions on Software Engineering, Vol. 14, No. 11, pp. 1327-1341, November 1988.
- [Wan85] Wang Y.,T., Morris R., *Load Sharing in Distributed Systems*, IEEE Transactions on Computers, Vol. C-34, No. 3, pp. 204-217, March 1985.
- [Woo95] Woodside C.M., Neilson J.E., Petriu D.C., Majumdar S., *The Stochastic Rendezvous Network Model for the Performance of Synchronous Client-Server-like Distributed Software*, IEEE Transactions on Computer, Vol. 44, No. 1, pp. 20-34, January 1995.

GLOSSARY

ABA See Asymptotic Bound Analysis.

Asymptote A bound of a performance metric when the population level is very high.

Asymptotic Conditions those population levels that cause at least a device to be saturated.

Asymptotic Bound Analysis A set of methods for estimating the bounds of performance metrics in a closed model when the population level approaches infinity. Practically, the asymptotic bound analysis is applicable when the population level causes the saturation of one or more devices.

Bandwidth Bandwidth is the amount of data that can be transmitted over a communication link per time unit. It is usually measured in bits per second.

Bottleneck (device or software entity) The first device or software entity that saturates (i.e. it reaches its 100% capacity limit).

Cache A technique to improve hardware and software performance. Copies of the most recently accessed data items are held in a smaller but faster device called cache. Subsequent accesses to the same data are thus speeded up. In client/server applications, server data is cached on the client machine.

Capacity Planning Process of predicting when the future workload level and workload mix will cause the system to reach the utilization limits. Also the process of sizing the system such that the performance requirements are met without reaching utilization limits.

Class A concept that groups the customers that have a similar statistical behaviour.

Class Population Level The number of customers of a class.

Class Response Time The time from when the customer of a class requests a service until that request is answered.

Class Throughput The number of customers of a class served per time unit.

Client A part of the client/server paradigm. The client calls the server via a Remote Procedure Call, a Remote Method Invocation or other communication mechanism.

Usually, the client is the process that presents the data to the end-user. In a 3-tier application, the processes on the second tier are clients for the third tier.

Client/Server A form of distributed processing which consists in splitting the work to be performed by an application between a client and a server. The client requests data, invokes methods or procedure from the server. There is one server per many clients.

Closed Model Queuing model with a fixed number of customers. The number of customers in the system is fixed and the number of system states is finite.

Configuration The result of object allocation to processes and of processes to nodes.

CORBA Common Object Request Broker Architecture. A specification released by *OMG* that defines the architecture for distributed objects and provide a standard mechanism for objects to transparently invoke each other's services independently of programming language and platforms.

CPU Central Processing Unit- the part of the computer that processes the instructions.

Crossover Points Those points in the population mix space where the bottleneck or set of bottlenecks changes.

DCOM Distributed Common Object Model – distributed object architecture developed by Microsoft.

Decision Making A formal methodology that evaluates alternatives in order to classify, order, choose or sort them.

Delay Queue Queue in which the waiting and service times are always constant.

Demand Sum of the service times at a resource (such as CPU, disk or network) over all visits to that resource during one execution of a Scenario.

Distributed Object An object that spans over two execution environments. A proxy (called stub) resides on the client, another proxy (called skeleton) and the object implementation itself reside on the server. The stub and the skeleton hide the communication protocol and the distribution. The client application programmer works with the stub proxy and any method invocation is propagated by the stub to the skeleton and to the object implementation.

Dynamic Allocation The process of migrating the processes or the objects during run time.

EJB See Enterprise Java Beans.

Enterprise Java Beans A specification released by Sun Microsystems that defines the specifications and the architecture for distributed objects implemented in Java. It is also a distributed component model.

Extreme Points Points on the boundary of the feasible space in linear or non-linear programming problems. These points have the property that they can not be between two other points on any hyper-plane that belongs to the feasible space.

Fair Response Time Metric in multi-class distributed models where each class is equally important.

Fuzzy Sets Sets in which each element has membership degree within $[0, 1]$ interval. Unlike the classical sets, where the membership was either 0 or 1, the fuzzy sets make possible any membership degree between 0 and 1.

HTML Hypertext Markup Language – a language for describing and formatting multimedia documents, in particular Web documents.

HTTP Hyper Text Transfer Protocol - an application protocol used by World Wide Web. It defines how client browsers interact with Web Servers.

Internet The set of global networks interconnected by TCP/IP.

Joint Bottleneck a bottleneck generated by one or more population mixes. The joint bottleneck is common to two or more classes.

LAN See Local Area Network.

Layered Queuing Models performance models that consist of two or more QNM arranged in a layered architecture.

Linear Programming Problem optimization problem composed of a function (objective) which is optimized and a set of inequality constraints. Both the objective and the constraints are defined on the same variables and are linear with respect to them.

Linear Programming The branch of applied mathematics concerned with Linear Programming Problems.

Local Area Network a network that serves a small geographic area such as an enterprise or institution.

LQM abbreviation for Layered Queuing Models.

LP - abbreviation for Linear Programming.

Mean Value Analysis Iterative techniques for solving a closed queuing network.

Middleware Software that allows inter-process and inter-object communication across applications and platforms. It consists in set of API and run-time that isolate the process or object from the underlying communication protocol. It also provides intermediate system services (Security, Naming, Transaction and others).

Midware See **Middleware**.

Multi-class (Models or Systems) Models and systems in which customers are partitioned into many classes. Each class has a unique device demand and routing behaviour.

MVA See Mean Value Analysis.

Natural Bottleneck the bottleneck of a class when this class is considered in isolation.

Node A computer interconnected by network into a topology.

NLP Abbreviation for Non-Linear Programming.

Non Asymptotic Conditions The opposite of Asymptotic Conditions. The population levels do not cause any device to saturate.

Non-Linear Programming Problem Optimization problem that consists in an objective function and a set of inequality constraints. At least one of these two elements is non-linear.

Non-Linear Programming The branch of applied mathematics concerned with Non-Linear Programming Problems.

Non-Natural Bottleneck See Joint Bottlenecks.

Object A computer object is a collection of data and methods. Encapsulation, inheritance and polymorphism characterize it.

Object Management Group (OMG) - a consortium responsible for creating CORBA.

Open Model Queuing Model with an infinite customer population. Customers arrive, receive services and exit. Usually the infinite buffer size is assumed and the possible system state is infinite.

Overhead The resource consumption which is not directly part of the main and useful task. Examples include launching and terminating of a process or creating and removing an object.

Performance Scenario A Scenario in which each method is shown with its resource demands.

Population Level The total number of customers in the system, including customers of all classes.

Population Mix The ratio of a class population level and the total population level.

Queue A resource and its associated waiting queue.

Queue Length Number of customers in a queue, including those waiting and being served.

Queuing Network Model (QNM) An interconnected set of queues.

Queuing Theory A body of mathematical methods used to perform analytical modeling. It considers queuing centres associated to hardware and software resources and evaluates metrics such as Response Time, Queue Length.

Response Time Metric that measures the responsiveness of a distributed system. For single class systems, the response time is the interval from when a customer requests a service until the answer to that requests is presented to the customer. See also Class Response Time and Weighted Response Time.

RMI Remote Method Invocation – a distributed architecture and an application protocol that is part of Java.

Satisfaction Metric which measures the degree in which the performance requirements are met. It takes values on [0 1] interval.

Saturated (device or software entity) Device or software entity that is 100% utilized.

Saturation Sector The set of points in the population mix space that causes one or more joint bottlenecks to saturate. When the sector is given by all classes, we have a global saturation sector, otherwise we have a partial saturation sector.

Scalability The capacity of a distributed system to accommodate more workload without diminishing the overall performance of the system.

Scenario The path of the user request through the methods of objects in the system.

Server Process or set of processes, that provides services to clients in a client/server processing model.

Service Time Time spent to a resource whenever a scenario or request visits that resource.

Single-Class (Models or Systems) Models or systems where all customers have the same statistical behaviour with respect to the resource demands and routing.

Software Performance Engineering The process of constructing software systems that meet the performance objectives.

Static Allocation The process of off-line allocating objects to processes and processes to nodes.

Think Time Interval of time elapsed since the user receives the prompt until he submits a new request.

Throughput the measure of the amount of work the system is performing as a whole. It represents the number of customers served per time unit or the number of customers that depart the system per time unit.

Topology The hardware layout of a distributed system. A topology is composed of nodes interconnected by a network.

Transaction An interaction that guarantees the integrity of updated information, protecting it from loss or distortion as a result of the change.

TCP/IP the suite protocol suite used in Internet.

Utilization (of device, object or other software entity) Performance metric which shows what percentage of time a device, object or other software entity is busy.

Web A short name for World Wide Web.

Web server a server in a client/server processing model specialized for Web applications.

Weighted Response Time A performance metric for multi-class systems. It sums the weighted response times of the classes. The weight of each class is given by the class importance in the overall system.

Workload Level See Population Level.

Workload Mix See Population Mix.

World Wide Web A client/server architecture spread over the Internet and which constitutes the infrastructure for exchanging HTML documents.

WWW Abbreviation for World Wide Web.

INDEX

#

β_i , 27, 65

A

ABA, 23
activation policy, 45
aggregation, 29, 152, 154, 160
algorithm
 simplex, 44
 straight, 45
allocation strategies, 24
alternative, 17, 28, 29, 147, 148, 154, 160, 174
applets, 25
Asymptotic, 51
asymptotic conditions, 24

B

bottleneck, 15, 23, 24, 30, 31, 48, 59, 66, 67, 68, 69,
 70, 71, 72, 73, 79, 99, 106, 107, 108, 110, 111,
 112, 114, 117, 118, 119, 120, 121, 123, 124, 125,
 126, 134, 159, 165, 166, 171

C

class, 2, 48
class population, 3
class population mix, 2
clusters, 2
component, 5
configuration, 15
constraint, 55, 58
convex
 space, 54
CORBA, 1
criteria, iii, 7, 17, 28, 29, 32, 98, 137, 148, 149, 150,
 151, 152, 154, 155, 160, 174
critical sections, 37
crossover points, 31, 44, 45, 64, 65, 66, 67, 69, 72,
 73, 74, 75, 84, 85, 86, 87, 88, 89, 95, 96, 99, 128,
 158, 159, 171
Crossover_Point_Algorithm, 1
customer class session, 3

D

DCE, 1
DCOM, 1
decision, iii, 7, 17, 19, 28, 29, 32, 33, 57, 77, 82, 98,
 136, 137, 147, 148, 149, 150, 152, 154, 155, 160
decision-making, 6, 17, 28
decision-making process, 17
delay, 5
demand
 CPU, 37
 disk, 37
design
 constraints, 10
device
 saturated, 52
device utilization, 28, 45, 53, 58, 59, 60, 63, 77, 83,
 84, 85, 94, 99, 158
distributed, 1
dynamic allocation, 24

E

EXTRA, 13
extreme points, 62, 68

F

fair response time, 108, 110, 117, 123, 125
feasible space, 59
framework, 6
function
 convex, 46
fuzzy, 17, 19, 29, 32, 33, 149, 150, 151, 152, 153,
 154, 155, 160, 173, 174, 175, 176, 177, 179, 180,
 181, 182

H

hardware, 5
Heuristic, 26
HTML, 1
HTTP, 1

J

Java, 1, 49

L

linear constraints, 59, 60, 75, 83, 84, 94, 112, 113, 157, 172
linear program, 15
logic, 29
LP, 45
LQM, 18, 20, 32

M

methods, 2
method invocations, 2
metric, 5
middleware, 1, 4, 11
model, iii, 1
MVA, 19

N

natural bottlenecks, 65, 66, 68, 69, 70, 71, 72, 73, 87, 118, 119, 120, 123, 124, 128, 165, 166
NLP, 8, 18, 44, 46, 55, 58, 59, 76, 80, 98, 99
 model, 55
NLP constraints, 58
nodes, 2
non-asymptotic, 1
non-asymptotic conditions, 24
non-linear function, 57
non-linear program, 15, 59, 60, 99, 112, 113
non-linear programming, 55
non-natural bottlenecks, 118

O

object, iii, 1, 4
 demand, 48
 replica, 46
 response time, 49
 waiting time, 49
Object replication, 30
object utilization, 18, 30, 32, 43, 44, 45, 46, 49, 52, 53, 55, 57, 59, 60, 75, 76, 77, 83, 84, 85, 87, 88, 89, 91, 94, 95, 96, 128, 129, 143, 145, 147, 157, 158
objective function, 58
Open Distributed Processing, 2

P

performance, iii, 4
 evaluation, 19
 model, 34

 performance requirements, 7

Performance, 1, 4, 10, 19, 28, 39, 40, 41, 99, 136, 151, 156, 177, 178, 179, 181, 182

population, 2

 high population area, 9

 level, 9

 low population area, 9

 medium population area, 9

 mix, 45

 very high population area, 9

population level, 9, 10, 14, 15, 16, 26, 31, 48, 51, 52, 85, 92, 93, 97, 98, 100, 103, 104, 106, 119, 123, 128, 130, 131, 132, 133, 139, 142, 149, 152, 158, 159

population mix, 3, 9, 17, 27, 32, 51, 64, 69, 71, 85, 87, 89, 97, 99, 109, 110, 111, 112, 113, 114, 117, 118, 119, 120, 121, 122, 123, 125, 148, 149, 152

population mixes. See population mix

product form network, 21

Q

QNM, 19, 20

quality of service, 28

queue

 length, 22

queuing, 5

queuing delays, 11, 30

R

ranking, 153

reference device, 56

requirements, iii

response time, 49

 per class, 22

$r_i(a_j)$, 29

RMI, 1

S

Satisfaction, 14, 15, 29, 32, 99, 128, 150, 159

saturation sector, 64, 65, 67, 68, 69, 1, 166

 global, 64

 partial, 64

scalability, 1

scalability factor, 107, 117, 120, 121, 122, 123, 125, 134, 159

scenario, 13, 38

scheduling, 24

server per request, 92

servlets, 25

shared server, 92

shift factor, 107, 117

shift factors, 108, 122, 134, 159

slack variables, 57

software, 1, 4, 5, 19

software constraint, 58
static allocation, 24
straight algorithm, 1, 88
synchronous, 47, 49

T

thread, 4, 13, 35, 38, 46, 94
throughput, 23
topology, iii, 7, 12, 16, 98, 148
total customer population, 2
transaction, 1

U

utilization, 5, 53
 constraints, 10

law, 49
limits, 59
maximum, 43, 54
maximum object, 18
object, 18
per class, 54
process, 43
space, 59
total, 55

W

waiting time, 49
weight, 17
weighted response time, 108, 116, 117, 124, 125,
 134, 159