INFORMATION TO USERS

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

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

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

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

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

UMI

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

Scalability Analysis Framework for Distributed Systems

By: Prasad P. Jogalekar

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

Doctor of Philosophy

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

August 31, 1997

©1997, Prasad P Jogalekar

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.

0-612-27014-9

Canada

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

"Scalability Analysis Framework for Distributed Systems"

submitted by Prasad P. Jogalekar, M.Tech., in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Supervisor

External Examiner

Chair, Department of Systems and Computer Engineering

Department of Systems and Computer Engineering,
Faculty of Engineering,
Carleton University,
September 1997.

Abstract

Distributed systems are becoming increasingly important in various domains of applications. These systems must be scalable, meaning that they must be economically deployable in a wide range of sizes and configurations. This thesis investigates scalability issues for general distributed systems. A scalability metric is proposed, which measures the cost-effectiveness of scaling any of the system resources, where the effectiveness is measured in terms of the system's throughput and quality of service (QoS) seen by its clients. An optimization framework is developed, which tunes the architectural parameters of the system to maximize the scalability metric. The value of the scalability metric, optimized over a range of scale of interest, indicates whether a system is capable of scaling to the desired size. If so, the optimum configuration is obtained from the result produced by the optimizer tool.

Thus, the framework enables comparative evaluation of design alternatives and scaling strategies.

This metric is tested on idealized systems, and the optimization framework based on the metric is applied to real system prototypes. The results are seen to be giving useful insight into the maximum scaling capacity of the designs and also the guidelines for the re-engineering effort, if required.

Acknowledgements

*Long it has been, steep and winding,
My Ph.D. has been tough, albeit satisfying.
I wish to thank some, from the bottom of my heart,
But for their help, I could not even start....*

*First and foremost, my advisor Murray,
Whose insightful thoughts, saved many-a-disarray.
"Performance" he taught, in more ways than one,
Working with him, has always been fun....*

*Prachi, my wife, wonderful as she is,
Did every sacrifice, to cope with my thesis.
My parents and Kedar - great inspirers for sure,
Despite the long distance, they seemed so near....*

*Last but not the least, it's Mike, Greg & Alex,
Who helped me fix problems, deep and complex.
Lucky I have been, to get a good backing,
Without all this, one can't do much hacking....*

Thankfully,

Prasad

TABLE OF CONTENTS

CHAPTER 1	Introduction	1
	1.1 Motivation	2
	1.2 Thesis overview	3
	1.3 Novel contributions and limitations	4
	1.4 Organization of the thesis	5
 CHAPTER 2	 Scalability Analysis: A Literature Survey and Some Background Topics	 7
	2.1 Scalability analysis of massively parallel multiprocessor systems: literature survey	7
	2.2 Scalability notions for distributed systems	13
	2.2.1 Scalability with respect to hardware resources	14
	2.2.2 Scalability with respect to software resources	14
	2.2.3 Scalability with respect to user traffic	15
	2.2.4 Scalability of algorithms and the embedded logic	15
	2.2.5 Scalability with respect to user classes	16
	2.2.6 Scalability with respect to openness	17
	2.3 Distributed and other general systems	17
	2.4 The need for new scalability metric and methodology	22
 CHAPTER 3	 Design Alternatives and Scalability of Distributed Systems	 24
	3.1 Scalability implications of design choices for a system	25
	3.1.1 Software architecture styles	25
	3.1.2 Flexible architectures and parameterized design	29
	3.1.3 Replication- a possible scalability enabler	34
	3.1.4 Allocation of software objects to CPUs - a possible scalability enabler	38
	3.1.5 DCE and CORBA overview	42
	3.2 The layered queueing models	47
 CHAPTER 4	 The Scalability Metric	 50
	4.1 The scalability metric	50
	4.2 Scalability analysis framework: some concepts and	

	terminologies	56
4.3	Scalability metric applied to idealized cases	58
CHAPTER 5	The Scalability Analysis Framework	69
5.1	Scalability enablers and their optimization	69
5.1.1	Diagnosis of scalability problems	73
5.2	The PRAGmatic Scalable Architecture Designer (PRASAD) tool	74
5.2.1	Simulated Annealing	78
5.3	Handling remote invocation overheads and transaction management overheads	80
5.4	Rule-of-thumb scalability bounds	81
CHAPTER 6	Scalability Analysis of Prototypes of Distributed Systems	88
6.1	The connection management system- version A	89
6.1.1	Performance model of the connection management system (version A)	90
6.1.2	Scalability analysis framework applied to the connection management system (version A)	91
6.1.3	Rule-of-thumb scalability bound	96
6.1.4	Results and discussion	99
6.2	The connection management system- version B	101
6.2.1	Architecture of the connection management system (version B)	103
6.2.2	Scalability analysis framework applied to the connection management system (version B)	106
6.2.3	Rule-of-thumb scalability bound	109
6.2.4	Results and discussion	111
6.3	The call processing system	117
6.3.1	The two half call model	118
6.3.2	The architecture of the call processing system	120
6.3.3	Application of the scalability analysis framework to the call processing system	128
6.3.4	Rule-of-thumb scalability bound	129
6.3.5	Results and discussion	132

CHAPTER 7	Conclusion	141
7.1	Research summary	142
7.2	Possible generalizations of this work	145
7.3	Areas of further research	146

Chapter 1 Introduction

Distributed computing is rapidly penetrating the marketplace for information systems, where a single software application can run on several machines, connected by a high-bandwidth network. A client-server scenario where desktop clients are supported by distributed back-end servers is fast replacing or supplementing the legacy mainframe systems. Systems such as distributed database systems, video-conferencing systems, etc. have already established themselves in modern office environments.

The advances in computer and networking technology as well as the shifting trend towards digitized voice and data have stimulated the growth of distributed software applications. The futuristic distributed applications include digital libraries, multimedia news service, distance learning, remote medicine, distributed network management, internet banking etc. While some of these systems are still evolving - conceptually as well as technologically - tremendous possibilities of new kinds of services are unfolding. Distributed middleware technologies such as DCE and CORBA (which are discussed later) support distribution transparency and application portability, and facilitate the development of such distributed software applications. The advent of high-speed networks and powerful (yet relatively cheaper) computers have provided several technologically and economically viable options for deploying distributed systems. Scalability, which means being able to change the overall system capacity by gradually adding or removing hardware and software resources as required, is an important selling point for distributed systems technology.

While such applications represent the future trends, the underlying technologies pose several intriguing questions about the suitable software architectures to implement such systems and the performance and scalability issues associated with them. The distributed execution platforms typically include a complex service architecture, and a hardware and software infrastructure to support such a distribution of the functions among clients and servers, over fast networking technologies such as the Asynchronous Transfer Mode (ATM). Note that the execution of a service request by a user may involve distributed concurrent execution of various servers, which may in turn act as clients of other servers, transparent to the user. For example, an airline reservation system would typically be implemented as clients invoking back-end transaction server(s) which in turn invoke various name/directory servers and database servers, transparent to the clients. This gives rise to a hierarchy of clients and servers, which interact in complex ways, to execute service

requests by an end user. Such are the types of systems that are of major interest as far as this thesis is concerned.

This thesis aims at developing a framework for assessment of distributed system architectures, in order to be able to be deploy them economically at various scales, while producing satisfactory performance and quality of service.

The distributed client-server systems typically have the following salient features:

- (a) Heterogeneity in protocols, hardware, communication patterns, etc. (b) Geographic separation of the machines on the network. (c) Name/directory service for the clients, in order to find the servers on the network. (d) Security service. (e) Quality of service negotiation. (f) Interoperability and openness, etc.

The above services require a software infrastructure (middleware) which executes on every node in the system, and renders the network transparent to the application programmer. Examples of particular interest include OSF DCE [Shirley94], CORBA [Corba], DPE [Tina] etc.

1.1 Motivation

Distributed systems should be deployable in a wide range of scales, both in terms of their overall size, and their ability to efficiently operate in a diverse range of configurations. The system developers want to develop a single product they can sell in a range of scales to meet a diverse set of customer requirements, whereas the customers want an easily upgradable configuration that could support a change in the operation scale at a minimum extra cost. The “scale” of a distributed system denotes its size, and the notion of size can depend on the context. e.g. a network management system should be able to manage a large number of various types of devices, ranging from edge switches to routers to backbone switches. This enables the users and the system architects to adopt the system design and configuration that suits their need, while maintaining a guaranteed Quality of Service (QoS). Also, it allows room for new management and control policies to be introduced in the network, allowing the legacy and the new management and signalling systems to co-exist.

For distributed client-server systems, scalability can mean being able to scale the throughput and the quality of service, being able to scale the number of servers, being able to scale the number of hardware units, or all of the above. A combination of these factors poses a complex scalability analysis (and optimization) problem. The basic notion of scalability as discussed and

formally defined in this thesis deals with being able to change the population and configuration of various system resources in a cost-effective manner, such that the throughput and quality of service can be enhanced proportional to the investment in doing so. (The resources include hardware, software as well as logical resources such as the embedded algorithms). If this is not achievable for a distributed system, it may need re-engineering of its software/middleware/hardware. If a system is not scalable, the potential section of the market it can appeal to gets limited.

For the target distributed systems (which are client-server systems, consisting of distributed objects communicating across network, typically based on a middleware that allows distribution transparency), the scalability questions can take such specific form as:

(a) Can adding a fast machine to the network buy a proportional improvement in the performance of the given system (after suitably tuning the system parameters)? (b) Can the extra investment in maintaining and administering multiple copies of certain software building blocks, (e.g. the database) be justified against the enhancement in the QoS experienced by the clients? (c) Among several alternatives (such as investing in faster machines versus a large number of slower machines, investing in m number of copies of any given server versus k number of copies of another server, etc.), what is the best scaling path for evolving a given distributed system to support larger scales of operation in future?

These questions can be non-trivial and cannot be answered using the conventional scalability analysis approach used for massively parallel multiprocessor-based systems.

1.2 Thesis overview

This thesis addresses the need for a comprehensive scalability analysis framework for distributed client-server systems, and proposes a unified, formal approach for implementing such a framework. This approach is briefly described below.

From the perspective of the developer/vendor/owner of a distributed system (say, a video-conferencing system), throughput of the service requests that can be handled by the system is an important figure of merit. Based on the requirements (which could span a wide range), it is necessary to be able to scale the system resources to meet the throughput demand with the least possible investment in resources. From a user's perspective, the quality of service is the most significant figure of merit. (e.g. the response time for setting up a video conference should be as close to the user-specified "ideal" response time as possible, and should cost as little as possible.)

A system design is scalable, if both these requirements can be simultaneously met, over a range of system sizes, keeping the provisioning cost as low as possible. This is the sense of scalability that is captured by the scalability metric (that combines both the figures of merit as well as the cost of deployment) proposed herein.

The metric quantifies the judgement of how well a system does (or does not) scale from one configuration to another. From a given reference state, various possible options for scaling can be compared based on the value of the metric in each case.

The metric also allows us to quantify the effectiveness of some design parameters (against others), which can be tuned for maximum scalability benefit (e.g. customizable protocols, number of users, replication levels for various tasks, multithreading, allocation of objects to resources, etc.). These adjustable design and configuration parameters are called *scalability enablers*. For a range of system sizes of interest, these system parameters can be optimized to maximize the scalability metric. This thesis presents a framework for doing this, which has been implemented and tested for some real prototypes of distributed systems. It also allows us *comparative evaluation of various design and scaling policies* over a range of scale, and to analyze the necessity to re-engineer some parts, if the optimized values of the metric fall below par.

1.3 Novel contributions and limitations

The major contributions of this thesis can be summarized as follows:

- (1) Various notions of scalability for distributed systems have been captured in a single scalability metric. A general scalability analysis framework, with new concepts and terminologies, has been presented. The scalability analysis approach proposed herein incorporates the system *design* as well as the system *scaling strategy*, which are integrated into the framework.
- (2) The framework has been applied to a set of idealized, abstract system models whose scalability is understood. This serves to validate the metric and the framework.
- (3) As a part of this framework, an optimization strategy is presented that can comparatively evaluate various designs with respect to given scaling policies. This is done by optimizing the available scalability enablers for each case. This framework also helps to judge the necessity to re-engineer some components of the system, if the choice of scal-

ability enablers for a given scaling policy fails to produce a satisfactory band of values of the optimized scalability metric.

- (4) The optimization framework has been applied to real system prototypes with substantial complexity, to validate the usability of the framework and to test the techniques.
- (5) In order to be able to get the rule-of-thumb scalability bounds for a system for any given scaling policy, a fast bounds analysis approach has been presented.

- Limitations of this thesis:

- The proposed scalability metric applies only to systems with a single user class.
- Some structural properties of certain types of systems may get left out of the analysis.

e.g. if the number of open sockets cannot exceed some number, this scalability limitation is captured by the metric only if it impacts the throughput or the response time of the system.

Other than these limitations, some incidental choices have been made for the analysis presented herein, which do not cause any loss of generality. These are as follows:

- A restricted measure of quality of service, based on the ratio of mean response time and the target (or desirable) response time has been used in the implementation of the ideas so far. The variance of response time and other QoS aspects can be accommodated in the metric (in theory), but have not been tested.

- In the current version of the tool that supports the ideas developed herein, only those systems that can be modeled as layered queueing models can be optimized. (Layered queueing models form a superset of conventional queueing networks.) Note that this is not a limitation of the ideas presented herein, but rather one of the current implementation, because it uses fast analytical methods to solve models of distributed systems.

1.4 Organization of the thesis

Chapter 2 presents a literature survey on various aspects of scalability analysis discussed in the literature, for massively parallel multiprocessor systems as well as some distributed systems, and the quantitative scalability treatment used for massively parallel multiprocessor systems is investigated. Various different notions associated with the term scalability for distributed systems are discussed in this chapter. The scalability analysis cannot be directly extended from mas-

sively parallel systems to distributed systems, as argued herein. However, it provides various useful insights for understanding scalability.

The nature of target distributed system architectures is explored in Chapter 3. Various architecture styles for distributed system designs are surveyed. General usefulness of parametric design and its intuitive effect on enhancing scalability is discussed in this chapter. The middlewares such as DCE and CORBA, that allow distribution transparency to the application objects, and typically influence the architecture of the target systems of this research, are also discussed.

Chapter 4 introduces a new scalability metric for general distributed systems. The metric forms an important basis on which the analysis and the optimization framework is built. The scalability metric has been applied to various idealized systems in order to tally the results produced against intuition.

Chapter 5 introduces some scalability notions and terminologies used by the scalability analysis framework. It also discusses the scalability analysis and optimization tool. The tool (PRASAD) conducts an automatic search in the system's configuration-space for maximizing the scalability metric by tuning the scalability enablers. An algorithm to roughly predict the scalability bounds on a system, which can save computationally expensive runs of the tool and still give some idea about how well the design could scale, is presented.

Chapter 6 discusses application of the scalability analysis framework to some real-life distributed system prototypes. Two different connection management systems are analyzed, using different scaling paths, scale factors, overhead models and cost functions. Scalability analysis of an ATM-based call processing system is also presented.

Chapter 7 summarizes this research work, and discusses future research that could be built upon this work.

Chapter 2 Scalability Analysis: A Literature Survey and Some Background Topics

This chapter presents a brief literature survey on some aspects of scalability analysis, and related background topics. The background material covered in this chapter aims at providing a comprehensive understanding of several qualitative and quantitative scalability notions. Chapter 3 covers the remaining background material required for understanding the analysis approach presented in this thesis.

Section 2.1 surveys the large amount of literature on scalability analysis of massively parallel multiprocessor systems, with symmetrical workload per processor. It illustrates various ways in which scalability can be understood and quantified. The notions of scalability for distributed systems can be quite different, as discussed in section 2.2. Section 2.3 describes scalability analysis of general distributed systems, which is more qualitative than quantitative. We note that there is a need to quantify various notions of scalability of distributed systems in a single metric. As argued in section 2.4, the scalability metrics of massively parallel systems cannot be directly extended to the distributed systems.

This thesis defines such a metric and demonstrates its use in the following chapters.

2.1 Scalability analysis of massively parallel multiprocessor systems: literature survey

Scalability has been actively investigated for massively parallel multiprocessor systems with respect to various algorithm-machine combinations, where the parallel programs execute on processors connected by a bus or multidimensional grid and run to completion. Time to completion is a natural measure of performance in these systems.

Distributed systems of the kind we are considering typically run continuously with many simultaneous users and transactions, and are evaluated by their capacity and the quality of service seen by the users. Therefore, due to the different nature of these systems, the analysis in the

papers listed in this section cannot be directly extended to the distributed scenario. However, these references provide useful insight into different scalability notions, and their quantification.

Many of the papers discussed below derive some form of “speedup” as a result of adding processors to the system. *Scalability of the system is then measured by the ratio of this speedup to the number of processors.* A system that can retain a constant value of this ratio is said to be scalable.

(1) In [Sun93], the authors describe the three models of parallel speedup, namely, fixed-size speedup, fixed-time speedup and memory-bounded speedup. The authors assume a homogeneous distributed-memory multiprocessor architecture such as a hypercube or a mesh.

The authors observe that the scalability limitations for typical applications arise due to uneven allocation (load imbalance) and communication latency. They discuss the use of *fixed-size speedup*, *fixed-time speedup* and *memory-bounded speedup* as a set of possible measures of scalability for an algorithm-machine combination.

While analyzing the *fixed-size speedup*, the amount of work to be executed is kept fixed, and is independent of the number of processors. As more processors are added, the time to complete the execution run is expected to reduce, and this reduction factor is the figure of merit that is captured by this metric. Beyond a point, adding extra processors does not help because of inherent limitations of the algorithm and the algorithm-machine combination is said to be non-scalable.

The fixed-size speedup, considering the communication latency and the uneven workload allocation, is given by

$$S_{fs} = \frac{\sum_{i=1}^m W_i}{\left(\sum_{i=1}^m \frac{W_i}{i} \times \left\lceil \frac{i}{p} \right\rceil \right) + Q(W)} \dots\dots \text{Eqn. (2.1-a)}$$

where

m = the maximum possible degree of parallelism, for the given “parallelism profile” of the algorithm-machine combination to be tested.

W_i = the total work done in states that have the degree of parallelism = i ,

p = the available processors, when not all i processors (i = the degree of parallelism) are simultaneously available, such that $p \leq i$,

$Q(W)$ = the total communication overhead in the units of work

W = the total sequential work required by the application = $\sum_{i=1}^m W_i$.

For analyzing the *fixed-time speedup*, as the number of processors increases, the total workload (W) is also increased, such that completion time remains constant.

Let $W(1, T)$ be the amount of sequential work that takes time T to execute on a single processor. Let $W'(p, T)$ be the amount of scaled work, as the number of processors is scaled, such that the execution time remains constant, equal to T . Note that as extra processors are added, the degree of parallelism with which the work $W'(p, T)$ is executed may increase until the logical constraints in the algorithm prevent further parallelism. This is the point beyond which the algorithm-machine combination loses scalability, and is captured by this metric.

Fixed-time speedup = (the total work done with the scaled degree of parallelism in time T) / (the sequential work done by the algorithm on a single processor in time T)

$$= S_{ft} = \frac{W'(p, T)}{W(1, T)} \text{Eqn. (2.1-b)}$$

Note that in (Eqn. 2.1-a), the total amount of work is fixed, which is not the case in fixed-time speedup (Eqn. 2.1-b).

In the case of *memory-bounded speedup*, the work done is assumed to be a function of available memory. We assume that memory is not a major constraint in most applications of interest (although in a distributed scenario, the available memory could limit the number of threads). Memory-bounded scalability is not discussed further here.

The authors conclude that the performance and scalability of parallel processing is influenced by the inherent parallelism and the communication requirement of the algorithm, by the computation and communication power (bandwidth) of the underlying architecture, and by the memory capacity of the parallel computer system, and the effect of scaling these factors may in fact be correlated.

(2) In [Sun95], the authors analyze performance characteristics of shared virtual memory machines. The authors argue that the *generalized speedup* proposed herein is a better performance metric for parallel machines with shared virtual memory than conventional definitions of speedup

such as [Sun93]. The generalized speedup treats the memory-access issues of parallelization in a different manner than the conventional definitions of speedup, and the authors show that the conventional metrics are its special cases. The authors define *asymptotic speed* of execution to be the time taken to execute a unit step of work, assuming the best memory access scenario (for the given caching, local memory and remote memory architecture), and assume that it is independent of the problem size. The generalized speedup metric, which is the *ratio of the parallel speed of execution to the sequential speed of execution*, is then derived to be:

$$S_{gen}(p) = \frac{W \cdot c'}{\left(\sum_{i=1}^p (W_i/i) \cdot c_p(i, W) \right)} \dots\dots \text{Eqn. (2.2)}$$

where $W = \sum_{i=1}^p W_i$ = the total work on p processors. (The maximum possible degree of multiprogramming is assumed to be less than p),

W_i = the amount of work performed when (any) i processors are active,

$c_p(i, W)$ = elapsed time for one unit operation of work when i processors are active,

c' = the elapsed time per unit operation at the asymptotic speed (used for the sequential time calculation).

If the problem size (work) for both parallel and sequential processing are the same, for the given choice of algorithms, then the generalized speedup is the same as the traditional definition of speedup as in [Sun93]. The authors relate the generalized speedup to *generalized efficiency* (which is equal to the generalized speedup divided by the number of processors) and to *iso-speed scalability*. Iso-speed scalability measures the ability of an algorithm-machine combination to maintain the average generalized speedup (defined in eqn. 2.2) per processor. The iso-speed scalability of the algorithm-machine combination is derived to be

$$\psi(p, p') = \frac{p' \cdot W}{p \cdot W'} \dots\dots \text{Eqn. (2.3)}$$

where p and p' represent the number of processors in any two configurations under consideration, and W and W' denote the respective workloads, such that the generalized speedup defined in equation 2.2 remains constant for these two configurations.

With the help of theoretical and experimental results obtained for a scientific application, the authors argue that the generalized speedup is a distinct and more general metric for evaluating algorithm-machine combinations.

(3) In [Grama93], the authors present “isoefficiency” analysis for parallel algorithms and architectures. They address the following question: at what rate should the problem size increase with respect to the number of processors in order to keep the “efficiency” fixed?

The efficiency of a parallel system has been defined as:

$$E = \frac{1}{1 + \frac{Q(W)}{W \cdot c}} \quad \text{.....Eqn. (2.4)}$$

where $Q(W)$ = the total overheads due to parallelism (depends on the number of processors, p , as well as the problem size, W). These overheads, measured in time units, include idling, communication, contention over shared data, etc.

$W \cdot c$ = (problem size) x (cost of execution of each operation) = sequential execution time.

Increasing the number of processors while keeping the problem size constant typically reduces the efficiency (because $Q(W)$ increases with the number of processors) and increasing the problem size for a fixed amount of overheads increases the efficiency. Efficiency could be maintained constant by increasing both of these appropriately. If W needs to be increased superlinearly with respect to the number of processors in order to maintain constant efficiency, the system is said to be poorly scalable. If W needs to grow only linearly with respect to the number of processors, in order to maintain a constant efficiency, the system is said to be highly scalable. Some parallel algorithms may have lower overheads, but limited concurrency. They perform poorly as the number of processors grows. The authors claim that this effect can be analyzed by the isoefficiency function, as illustrated using some idealized cases.

(4) In [Sarukkai95], the authors present a toolkit for the scalability analysis of a class of deterministic message-passing scientific applications. The tool (Modeling Kernel) generates a model based on the program’s parse tree. The authors observe that the objective of performance prediction is to characterize a program’s completion time and the objective of scalability analysis is to characterize its asymptotic speedup (or any other scalability metric) as a function of the prob-

lem size and the number of processors. The authors leave the choice of the scalability metric to the user. The tool automatically extracts the communication characteristics from message-passing parallel programs.

This paper focuses on the integrated modeling of loops and communication patterns. The factors influencing the program's communication time include the network topology, systems software, communications protocols, message size, temporal distribution of communication events, the set of processors communicating, the data dependencies across processors, etc. The tool described in this paper generates an augmented parse tree from the parallel program and automatically groups each set of related communication operations into "phases", categorizes each phase as synchronous or asynchronous and derives a symbolic expression for each phase's communication complexity.

(5) In [Sivasubramaniam94], the authors comparatively evaluate the techniques for studying parallel system performance, namely, experimentation, analytical modeling and simulation. The authors mention that due to the overheads of parallel processing, both the *available* and the *delivered* computing power have to be taken into account. This is captured by the notion of scalability. Several scalability metrics are summarized in this paper. These include constant problem size scaling, time-constrained scaling, memory-constrained scaling and isoefficiency scaling. The authors identify different types of overheads in a parallel system, including the algorithmic overheads, the software interaction overheads and the hardware interaction overheads.

The performance and scalability evaluation techniques have been broadly classified into two categories: those which study the performance of the system as a whole, and those which study the performance and scalability of specific system artifacts such as locality properties, synchronization primitives, interconnection networks and scheduling strategies. The authors compare various metrics with respect to their accuracy, cost and complexity of evaluation, and ability to capture a broad spectrum of variables.

(6) In [Mellor-Crummey], the authors present fast algorithms for contention-free mutual exclusion, reader-writer control, and barrier synchronization, for shared memory multiprocessor systems. It is shown that contention due to busy-wait synchronization neednot be a major obstacle to scalability and acceptable performance.

Busy-wait mechanisms are often used to protect or separate small critical sections. Typical implementations of busy waiting tend to produce memory and interconnection network contention, introducing performance bottlenecks. The authors describe new algorithms that use distributed data structures to implement protocols for busy-wait synchronization. The authors highlight the importance of local access to shared memory to achieve this, and argue that special purpose hardware for synchronization among processors is unnecessary.

(7) In [Sundaram], the authors use the fixed-time scalability analysis approach, and study the effect of scaling system parameters such as the bus bandwidth and the cache size, as the processor speed scales. The execution characteristics of a number of data-parallel numerical and scientific applications (such as the Fast Fourier Transform, Conjugate Gradient, LU-factorization, oceanographic simulations etc.) are considered. For different classes of applications, different scalability conclusions are drawn.

2.2 Scalability notions for distributed systems

The term scalability has several distinct flavors in the case of distributed systems. These will be described prior to surveying relevant literature on distributed systems scalability (in section 2.3). This is also required for understanding the scalability metric and the methodology discussed in the later chapters of this thesis.

Scaling up a distributed system usually means changing the population of its physical and/or logical resources in order to achieve some enhancement in the system's performance at a justifiable cost.

The software resources in a distributed system include the various executable processes corresponding to the server tasks in the system, the multiple concurrent threads of execution forked within those processes, buffers, etc. Scalability with respect to hardware and software resources of the system is associated with achieving a sufficiently higher amount of concurrency to justify added resources and relatively smaller, or "tolerable" amount of additional overheads like extra network bandwidth demand and extra device utilization. Scalability with respect to embedded algorithms refers to their complexity as well as flexibility while adapting to changes in the system architecture. If the algorithms used are inflexible (e.g. a rigid scheduling mechanism

that can not support jobs beyond a certain number), adding resources would not help to increase the system's capacity and the system may need to be re-engineered.

For any distributed software architecture (such as one of the architectures discussed in section 3.1), one or more of its constituent objects and/or their embedded algorithms can be augmented, causing the system to “scale up” (and conversely, to “scale down”). Depending on what component of the architecture we are dealing with, we get the following broad notions of scalability:

2.2.1 Scalability with respect to hardware resources

Suppose we keep on incrementally adding extra hardware to a distributed system, in order to make the overall execution faster. We can add more nodes, disks and caches until at some point, adding extra hardware does not make “enough” difference, either because the *bottleneck shifts from a hardware resource to some software resource*, (e.g. communication service, security service, trading, transaction management service, etc. which might be either application layer or middleware level) or *because adding extra hardware resources cannot improve the load balance, (some hardware resources get underutilized whereas others get overburdened)*. If adding extra hardware, and then tuning the parameters like the collocation/distribution of the objects can yield sufficient performance improvement (in terms of increased throughput for a fixed QoS and/or enhanced QoS for a fixed throughput), the system is said to be scalable from its previous state to the new state. For a given software architecture and middleware, one would like to know, up to what extent could the hardware be scaled (at best) while obtaining satisfactory performance. This may provide a useful insight into the returns on hardware investment for the given application and middleware.

This kind of scalability is similar to the one addressed in [Char95], discussed in section 2.3.

2.2.2 Scalability with respect to software resources

This type of scalability deals with the application and the middleware layer. In principle, *replication* and *multithreading* of the server objects may improve the overall system performance, by allowing service to more clients concurrently, if the overheads of doing so are not in excess of the benefit derived. A system is said to be scalable in the above sense if, changing the system con-

figuration by adding extra copies of software objects leads to increased system performance (in terms of increased quality of service seen by the clients for fixed throughput -and/or- increased throughput for a fixed quality of service).

A system can be moved from one configuration to another, while enhancing the scalability in the above sense, by appropriate replication and balancing the traffic among the replicas (which are the software resources), and by suitably re-allocating the replicas for overall load balancing among the hardware resources.

2.2.3 Scalability with respect to user traffic

A distributed software application, using the given distributed hardware/software resources, can support several concurrent service requests which are made by various classes of clients, to the “services interface” advertised by the application. e.g. this interface could be a WWW page providing some data to the clients upon request, or it could be an interface of a DCE application, registered in the local cell’s directory. If the arrival rate of the incoming service requests is scaled up, it causes extra workload on the system’s resources and increased queuing within the system. (*The queueing, the concurrency due to different clients’ threads executing simultaneously, and the inherent heterogeneity of distributed systems pose a significant difference from the multiprocessor scenario.*) The response time experienced by the clients becomes unacceptable beyond a certain range of traffic. This describes the scalability limit on the system in the sense mentioned above.

We would like to push this limit as much as possible by “tuning” the parameters - namely, collocation/distribution, multithreading, replication of system resources, caching, etc.

Thus, the system is scalable from one configuration to another in the abovementioned sense, if it can handle more traffic, without causing the response time to cross a client-specified upper bound. This kind of scalability is similar to the one addressed in [Yoshikawa], [Dahlin], [Shiekh97] in section 2.3.

2.2.4 Scalability of the algorithms and the embedded logic

This type of scalability, directly or indirectly, deals with the logic embedded in the algorithms used by the application and the middleware layer. This type of scalability depends on how the embedded logic at the application and the middleware level affects load balancing and com-

munication among the hardware and software resources. Inefficient or inflexible algorithms can cause hot spots in the system irrespective of the available resources to execute them. Consider a database transaction processing system implemented on top of DCE middleware that manages a distributed database. Consider a case where the servers on different machines communicate with each other to maintain consistent data across the cell. The algorithms used by the application logic in this case, to achieve group communication paradigm, have an inherent algorithmic complexity. Thus, it is possible that the logic embedded in the distributed application could contribute to some of the inefficiencies, and may need an excessive number of steps to accomplish the operation.

There could also be some flaws in implementation, for example, unnecessary computations, premature computations, needless library or system calls etc. which should be removed for better scalability, but these parameters cannot be quantified in general as they depend on the specific cases.

This notion of scalability is similar to the one discussed in [Yoshikawa], [Allison96] etc. (section 2.3), and in case of massively parallel multiprocessor based systems.

2.2.5 Scalability with respect to user classes

This type of scalability deals with planning the distributed system for future evolution. In many cases, the distributed software designers have to plan for future versions and extensions of their distributed applications. It means adding different usage patterns to the software, which in turn means changed resource consumption patterns.

The question is: can one avoid re-architecting the software in order to support additional (roughly predictable) future user classes, such that the quality of service (QoS) for all the user classes is acceptable? In other words, can the distributed system be evolved with respect to extra service provisioning capacity? If the answer is yes, then the distributed system is said to be scalable with respect to the usage patterns.

This type of scalability depends on the application. e.g. a network management system may not fully support fault management, for its customers may not want it now, but it may have placeholder interfaces for future requirements of other classes of clients, and sufficient hardware resources planned for this.

This type of scalability can be tackled if any predictions can be made about the resource consumption of the future classes of clients. [Kiczales96-2] suggest an interesting approach for open implementations design guidelines, which may be used to achieve this type of scalability.

2.2.6 Scalability with respect to openness

This type of scalability deals with the flexibility and openness of the interfaces offered by the system. The number of vendors whose products can communicate with the given system for supporting end-to-end functionality measures this kind of scalability. Standards like CORBA 2.x specify inter-vendor operability of the ORBs through the inter-ORB protocol. The more “open” a given distributed service is, the more scalable it is in this sense. There is no need to quantify this kind of scalability, although it is very important from the standards viewpoint, and from the business perspective.

2.3 Distributed and other general systems

The literature survey presented here shows that there has been a lot of discussion about the scalability of distributed systems. This active interest in scalability analysis indicates that there is clearly a problem, but it has been tackled more qualitatively than quantitatively. Moreover, the surveyed scalability analysis lacked a general approach and was more focussed on specific systems. The term scalability can have several flavors in case of distributed systems, as described earlier. Here, a few references are discussed, that use some notion of scalability in the context of distributed/ other general systems.

(1) In [Char95], the authors discuss the scalability of Microsoft’s Windows NT operating system. The authors observe that the performance gained from resources added to a computer system defines how a system scales. The hardware, the operating system, the application software/ middleware, and the flow of work done on the system determine the size of this performance gain. The authors propose that a single number, the *scalability*, can describe the relationship between performance and additional processors.

The authors used an in-memory subset of Microsoft’s SQL server benchmark, to focus on the CPU performance. Scalability analysis is carried out by plotting a graph of performance fig-

ures obtained from this benchmark versus the number of (Pentium 100 MHz) processors. The scalability tests are carried out for 1-6 processors. The authors observe that the software components need to be optimized with respect to the operating system, in order to fully utilize the system resources. (e.g. according to the authors, SQL server 6.0 scales more than SQL server 4.21a, because the former is optimized for use with Windows NT server 3.51.)

(2) In [Allison96], the authors describe a scalable load monitoring service and a scalable resource query service for managing resources distributed across a network. By scalability, the authors mean the relationship between the bandwidth requirements (i.e. the amount of traffic generated on the network) and the number of hosts on the network. In order to obtain information about individual host configurations (number of processors on the host, memory size, operating system details etc.) and the hosts' state (e.g. processor and interface load, spare memory and disk space etc.), two basic protocols are used -

- (a) Supply-driven, where the information is published by the producers, and
- (b) Demand-driven, where the information is passed to the consumers upon request.

In order to enhance scalability, the authors successfully use various techniques that aim at reducing the network traffic, which are as follows:

(1) The state information is disseminated using a supply-driven approach through a load monitoring service, whereas the configuration information is disseminated on demand using a resource query request.

(2) IP multicast is used for periodically publishing load information and for transmitting query requests.

(3) Load information from groups of hosts is summarized into a single number (thus increasing the granularity) before it is transmitted onto rest of the network.

(4) Administrative policy that governs when the nodes can publish their information.

The paper qualitatively uses the network traffic (messages generated by the client for polling the hosts) as an indicator of the inverse of scalability. But no detailed performance comparisons of different possible implementations are presented.

(3) In [Yoshikawa], the authors argue that the existing solutions to provide transparent access to network services need to be modified for the internet paradigm, considering the scalability, fault tolerance and load balancing issues.

The authors observe that the services such as ftp, http and web indexing need to scale gracefully with offered load, and should achieve incremental scalability (with respect to hardware resources on the net), load balancing among servers, dynamic resource recruitment (e.g. during the periods of increased traffic), and fault tolerance.

It is argued that the client, rather than the server, might be the right place to implement transparent access to the network, more so due to the recent advent of Java. e.g. the clients aware of relative load on a number of mirror ftp sites can efficiently connect to the least loaded mirror directly. When a user wishes to make use of a service, a bootstrapping mechanism is used to receive the Java applet corresponding to that service. The Smart Client uses the name resolver to invoke service specific mechanism to retrieve the information. The load balancing mechanism is service specific, but typically involves a dedicated server that keeps track of usage profiles of the service providers, including the hardware performance, network connectivity etc. This information is queried by a smart client, and based on the client's own location in the network and QoS requirements, a suitable server is chosen. In order to keep an applet's state information updated, the authors suggest a lazy mechanism.

The authors survey alternative solutions such as DNS aliasing and HTTP redirect, and suggest that implementing Smart Clients on a platform such as Java leads to more flexibility, fault tolerance and load balancing. The authors' implementation of Smart Clients demonstrates scalable FTP, Telnet and Internet Chat Services.

To demonstrate the scalability available from using Smart Clients for applications such as FTP, the authors measure delivered bandwidth to Smart Client applets running in Netscape Navigator from a varying number of FTP servers. A plot of aggregate bandwidth in MBit/sec versus the number of clients (making simultaneous requests) is presented for each server population.

(4) In [Dahlin], the authors present the xFS caching protocol (for serverless network file service), designed to improve the network file system scalability by taking advantage of the clients' processors, memories and disks. xFS attempts to provide low latency, high bandwidth access to file system data by distributing the functionality of the server among the clients. All files are

stored at the clients and all the data goes from client to client. The server is used only to coordinate the data transfer between the clients.

The authors observe that an important factor in a file system's scalability is the balance between caching and cache-consistency mechanisms. It is argued that in comparison with the other file systems considered, NFS has limited scalability due to its caching policy ("in memory" cache as against caching on local disks), periodically invalidating attributes of files, and its write-through policy that sends all changes to server disk.

xFS reduces the server load by using write ownership and clustering. Write ownership means that the first time a client closes a modified file, the server is notified, triggering an invalidation of all the copies cached on other clients. At that point, the client gets the exclusive write ownership of the file until another client opens it for reading.

The cluster servers isolate the ownership changes and data transfers internal to the cluster from the central server.

Results are obtained using a trace-driven simulation. The authors report that xFS protocol reduces the server load by more than a factor of six compared to the Andrew File System (AFS) without significantly affecting the response time or file availability. The number of messages to and from the server is reduced by 68% (as compared to AFS) and removal of the bottleneck thus leads to scalability of the file system.

(5) In [Sheikh97], the authors present a performance analysis framework, based on extended layered queueing models, to enable design-time modeling of complex large scale distributed applications. The authors analyze how some design parameters for the example system affect the application's QoS and scalability.

This analysis is more valuable if carried out at an early stage during the development cycle, as it allows the comparison of various design alternatives. The application analyzed in this paper is a distributed network management system, with different types of nodes grouped together in LANs (a single management domain) and WANs (collection of various management domains).

The authors study scalability with respect to the number of nodes and the number of domains. End-to-end mean response time is used for evaluating the scalability of the design. The method of layers, used for the analytical modeling of the system, has been briefly discussed in the paper. The results of analytical modeling are validated against simulations. The "scale factors" used in the analysis decide the number of managed devices, and the number of managed domains.

The results include the response time plots versus the number of devices and the number of domains.

This paper discusses a special case of scalability framework developed in this thesis, and provides valuable insights into how layered models can be used for scalability analysis of distributed systems of interest.

(6) In [Rolia95-a] the authors discuss performance and scalability implications of design choices for distributed systems. They observe that the design decisions for a distributed system include the *composition* of software tasks (embedding the functionality into clients and servers) and the *distribution* (the software to hardware allocation policy).

The authors define an application to be “scalable” if it can be easily configured to balance resource consumption across devices for target environments of interest.

Replication and multi-threading have been suggested as the means to improve the distribution aspect of a configuration, in addition to the right choice of communication protocols and security levels. Composition is suggested to be done in such a way, so as to maximize the concurrency between processes, and minimize the communication.

In order to achieve this, the authors suggest utilization analysis of various system components to detect and remove software and hardware bottlenecks. This results in high performance and scalability. The inherent scalability notion used by the authors is that of bottleneck analysis. For performance analysis, the authors suggest a use of mean value analysis, to be validated using simulation and benchmarking.

(7) In [Friedrich95], the authors present a software-based measurement infrastructure for monitoring the QoS and obtaining the model building parameters. The authors rigorously analyze the processing and network traffic overheads of the Distributed Measurement Systems (DMS) tool itself, (which also is a distributed application), and analyze its scalability.

The author’s interpretation of scalability for this system is the number of nodes that can be supported by DMS, without exceeding certain network & CPU utilization. (Note that the number of nodes is proportional to the number of clients of the DMS system.)

The authors claim that certain techniques used in the design of DMS minimize the amount of CPU and network utilization, so as to improve scalability. These include -

- periodically reporting summarized performance data to *collector* objects
- using threshold, to report only the exceptions
- non-polling, bulk transfer algorithms for minimizing bandwidth usage.

The DMS is implemented using the OSF DCE distributed environment (discussed in section 3.1.5). The components of DMS that contribute the most to the utilization of system resources are identified. The scalability advantage for the DMS system over polling-based systems is that if the performance of the system (which is being monitored by DMS) degrades, the amount of information passed by DMS on the network still remains bounded, avoiding an avalanche of diagnostic network traffic.

This paper throws light on scalability concerns for distributed systems of interest for this thesis and their relationship with the architectural level decisions.

2.4 The need for a new scalability metric and methodology

In the light of various types of scalability requirements for distributed systems, such as the ones considered in 2.3, this section discusses the limitations of directly extending the scalability analysis approach used in massively parallel systems.

- Consider the typical scalability questions of interest for distributed systems such as the ones listed in Chapter 1: (1) Can the extra investment in maintaining and administering multiple copies of certain software building blocks (e.g. the database) be justified against the enhancement in the QoS experienced by the clients? (2) Among several alternatives (such as investing in faster machines versus a large number of slower machines, investing in m number of copies of any given server versus k number of copies of another server, etc.), what is the best scaling path for evolving a given distributed system to support larger scales of operation in future?

The scalability metrics for massively parallel multiprocessor systems, surveyed in section 2.1, are not meant to answer questions such as above. The typical scaling path for those systems involves replicating hardware resources such as CPUs or the bus bandwidth, and the scaling path is closely tied to the application level algorithm. Logical and software resources and their replication/multithreading/migration/blocking is beyond the scope of the scalability analysis for mas-

sively parallel systems. Therefore a new metric and a new methodology that embeds many possible scalability notions is necessary. Such a framework is developed and presented in Chapters 4 and 5.

- The distributed systems typically handle a random flow of concurrent requests for service, instead of having planned runs to completion. This leads to queueing at software as well as hardware resources of a distributed system. Thus the performance analysis concepts for distributed systems are not the same as those for massively parallel systems.

Having got a clearer view of what needs to be addressed for analyzing the scalability of distributed systems, we now present a background discussion on the design choices for the target systems and how they may or may not impact their scalability. This is discussed in Chapter 3.

Chapter 3 Design Alternatives and Scalability of Distributed Systems

In Chapter 2, some background topics were covered in order to facilitate better understanding of various qualitative and quantitative notions of scalability of systems. This chapter focuses more on understanding the problem domain (e.g. the nature of the distributed architectures of interest, design alternatives for target systems, etc. and their possible impact on scalability). The scalability analysis framework proposed in Chapters 4 and 5 uses some of the notions discussed herein.

Section 3.1 focuses on the identifying the problem domain (e.g. the target systems and their architectures, and what factors may or may not make them scalable, with appropriate literature survey). Section 3.1.1 discusses various types of software architecture styles, and presents a brief literature survey. This section illustrates how, depending on the nature of the software applications, some architecture styles might be more conducive for scalability than others. Section 3.1.2 highlights the importance of parametric/open design and its impact on scalability. It also discusses the logical framework architecture concepts (TINA, ODP etc.) that promote design of flexible and reusable software architectures. Section 3.1.3 discusses an important scalability enabler, namely, replication, and the way it can affect scalability of software architectures. Section 3.1.4 presents literature survey and discussion on another important scalability enabler - the allocation of software objects to hardware resources. In the proposed scalability analysis framework, the design of software entities is treated independently from how they are allocated to the hardware modules. A near-optimal allocation is automatically found which is used for comparison among design alternatives. The middlewares such as DCE and CORBA, that allow distribution transparency to the application objects, and typically influence the architecture patterns of the target systems of this research, are discussed in section 3.1.5.

Section 3.2 focuses on the background required for understanding the implementation of the scalability analysis methodology. The tool that implements the ideas in this thesis makes a heavy use of the layered queuing models of a given system in order to optimize its configuration. Analytical solution techniques for layered queuing models are surveyed in this section.

3.1 Scalability implications of design choices for a system

This section considers architecture alternatives for distributed systems and how some design choices may affect scalability.

Section 3.1.1 focuses on various architecture styles for distributed software, and presents a relevant literature survey. Section 3.1.2 discusses the importance of parametric design of software and its possible impact on the scalability of systems. Section 3.1.3 focuses on using the replication of software resources as scalability enablers. Section 3.1.4 discusses another important scalability enabler, namely, the allocation of software objects to the hardware resources (collocation/distribution). Section 3.1.5 surveys two important middleware platforms that allow distribution transparency: DCE and CORBA.

3.1.1 Software architecture styles

This section is intended as an introduction to some of the important software architecture styles for distributed systems. Design of a distributed software architecture has a great impact on scalability of the system. Software architectures are higher-level abstractions of the system components. Therefore, choice of a software architecture style has a great impact on design parameters such as communication patterns among the software resources, concurrency among them, bottlenecking due to critical sections, etc. Clearly, the choice of an appropriate architecture is of paramount importance to the scalability of the system implementation.

Distributed systems have distributed hardware, middleware and software components. The hardware configuration of the system can be extremely general, as long as it supports the distributed middleware, e.g. the hardware components can be a collection of heterogeneous machines, disks and caches; connected by a network which could be ethernet, token ring, ATM, etc. Only the performance parameters of the hardware devices (such as the speeds of the CPUs and communication costs between the hardware components) fall into the scope of this thesis. The hardware architectures are not discussed further.

For large distributed systems, major design problems are not only algorithms and data structures, but also the global control structure, communication protocols, synchronization and data access, assignment of functionality to the software units of distribution (building blocks), scalability and performance, etc. This has been addressed in [Garlan 93],[Perry92], etc. A soft-

ware architecture typically consists of software *components* (building blocks) and their *interconnections*.

Designing software architectures is an important step in designing distributed systems because

- software architectures capture high level relationship among various subsystems (which also promotes reuse).
- detailed understanding of the architectures facilitates the right design choice for the given set of requirements.
- the software architectures can describe different components in different views.
- the right choice of architectures can enhance software reuse, robustness, extensibility and scalability.

Some architecture styles based on topology:

The following is a list of some of the commonly found software architecture patterns in distributed systems, which are mainly based on the topology and mechanism of interconnection among its components: [Garlan 93]

- *Pipes and filters*: Filters don't share state with other filters, although they may execute concurrently. Their specifications restrict to what appears on their input and output. A pipeline is a linear arrangement of filters - e.g. programs in a UNIX shell, many signal processing applications, etc.

Compatible filters can be hooked together, and thus, this architecture style can promote reuse of software components. Throughput and deadlock analysis is easier for pipelines. Although suitable for batch processing, such architectures may not be suitable for interactive applications according to Garlan et al.

- *Architectures using event-based implicit invocation*: Instead of invoking a procedure directly, a component in the architecture broadcasts one or more events. Other components register their interest in an event by associating a procedure with the event. The system invokes all the registered processes upon announcement of an event. Thus, an event transparently invokes several procedures in different modules. Examples of such architectures are syntax-directed editors, forms based graphical user interfaces, etc.

Such architectures strongly support reuse, and facilitate easier system evolution due to implicit invocation (components may be easily replaced by keeping the interface the same.) This is an important scalability feature associated with this style of architectures (although the module handling event management can become a bottleneck at high traffic levels).

This architecture style is also called *reactive integration* or *selective broadcast*. The draw-backs of this architecture are (a) - the event announcer has no idea of which components would respond, and in what order, (b) - due to (a), the proof of correctness is difficult and (c) - if some data needs to be passed with the announcement, a shared repository is required, which can have implications on performance and resource management.

- *Architectures using repositories (blackboard architectures)*: This type of architecture maintains a central data repository to represent the current state of the system. A collection of independent components operates on the shared data. The current state of the central data triggers processes. Thus, the blackboard architecture consists of three major parts: (a) independent objects interacting through the blackboard, (b) the blackboard data structure and (c) control (driven by the blackboard).

The scalability features of such an architecture depend on the nature of (a), (b) and (c).

- *State transition systems*, where architectures are described by the states and the transition rules.
- *Client-Server architectures*: Client server architectures are becoming increasingly popular in various domains of software applications, ranging from graphical user interfaces to database management systems. Most modern middlewares such as CORBA, DCE etc. support the client-server paradigm. In general, the client programs (possibly distributed across network) submit requests for some specific services to powerful back-end server software and/or hardware objects, which might be distributed across the network. This style of architectures has been widely discussed in the literature. e.g. [Menasce94].

The popularity of this paradigm can be explained by proliferation of PCs as the front-end clients, supported by powerful back-end servers. Examples of servers include file servers, name and directory servers, database servers, etc.

The Client-server paradigm fits well with the object oriented technology, because the client and server objects manage their own state and communicate with each other at run-time, using a predefined interface.

A server advertises only its interface which is invoked by the client at run time, but the actual implementation of procedures at the server is transparent to the client. In fact, a server may invoke other server(s) without the client's knowledge. Thus, a separation of functionality between clients and servers facilitates greater software reuse, flexibility, and potentially, scalability.

The design issues of client-server systems include the execution load distribution (i.e. the functionality) among the clients and the servers, the communication protocols and overheads, the amount of data transfer between the clients and the servers, bottlenecks due to the provision of name/directory and trading service that allows the clients to negotiate the QoS with the servers, etc.

The servers may be replicated in order to provide fault tolerance and improved performance to the clients. e.g. if a copy of a server is not available due to failure, the client's request can be forwarded to another available copy of the server, which provides the same service.

Thus, an appropriately designed client-server system can lead to better performance, robustness, and economical evolution of the system and hence better scalability.

A more specialized architecture description, namely, the *layered architecture*, can capture a combination of one or more architecture styles such as the ones mentioned above, and is useful for performance modeling, which is instrumental in scalability analysis.

- *Layered architectures*: Layered software architectures fit well into the client-server paradigm of software architectures. Software tasks are organized as a hierarchy wherein each task acts as a server to a task in a layer above, and as a client of a task belonging to a layer below (except for the objects belonging to the topmost or the bottommost layer, which are called pure-clients and pure-servers respectively.)

In strictly layered systems, a client object can make a call to the server object which lies only in the layer that is immediately below the client's layer. Thus, except for the adjacent outermost server layer, all the inner layers are transparent to the client.

Layered architectures are usually organized as pyramids or towers, with increasing levels of abstraction and/or partitioning. Communication protocols usually use layered architectures. Due to standardized interfaces between layers, layered architectures typically enhance the flexibility and the scalability of the design. Moreover, they strongly support reuse of the components.

Drawbacks of the layered architectures: (a) It may be hard to capture some logical interactions into layered models. e.g. peer-to-peer communication between two servers. (b) Performance may get penalized due to layering in some cases.

Section 3.2 discusses the performance modeling of layered architectures, using layered queueing models [Woodside95],[Rolia95].

3.1.2 Flexible architectures and parameterized design

The choice of an architecture is an important factor in the scaling characteristics of a distributed system (This is further discussed in the following sub-sections). The more flexible the architecture, the more is the number of parameters that can be “tuned” to possibly make the system scalable. In other words, *parameterized designs are more flexible and may lead to an increased number of tuning parameters for scalability*. Moreover, the more flexible the design, typically, the less it costs to augment the system’s resources, thus reducing the cost of increasing the system size, thus contributing to the system’s scalability. (The term “scalability” is formally defined in chapter 4.)

We now discuss some more aspects of software engineering that affect the scalability, extensibility and robustness of distributed system designs:

3.1.2.1 Object-oriented design patterns and frameworks:

Design patterns and frameworks capture *architectural themes* which might recur. Patterns capture the static and dynamic structure and collaboration among key participants in distributed software designs. They enable reuse of the architecture and design knowledge but not the code directly.

A *framework* is an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications. Frameworks typically address a particular application domain. Applications are formed by inheriting from and instantiating the framework components.

Object oriented design patterns and frameworks elevate the development to focus on higher-level design issues such as service functionality and policies, service configuration, service concurrency and synchronization, etc. The choice of the design patterns and frameworks can affect the logical complexity of operation and also the scalability of the design.

For further discussion on design patterns, the readers are referred to [Gamma94].

3.1.2.2 Parameterized design of software architectures:

Traditionally, a rule of thumb for software design is “expose the functionality of the module but hide the implementation”. Although such “black box” abstraction of the implementation is usually desirable, it can lead to serious performance and scalability problems when reusing the modules. This is because general purpose implementation strategies of the server’s interface for a wide range of clients are not necessarily optimal for each client.

An *open implementation* approach is a solution to this problem. Whereas black box modules hide all the aspects of their implementation, open implementation modules allow clients some control over selection of their implementation strategy, while still hiding some implementation details. [Kiczales96-1] suggests a “separation of control” strategy to address this. The author suggests a design methodology that separates a module’s *primary interfaces* and *secondary interfaces*. A primary interface supports the functions called by the client whereas a meta-interface (which is coupled to the primary interface) is used by the client to “tune” the implementation underlying the primary interface. The author discusses this with an example of a window system that displays a spreadsheet.

Achieving this kind of separation between the primary and the secondary interfaces for a module is related to the concept of *computational reflection*, which deals with the issue of how modules can provide meta-interfaces for examining and adjusting themselves.

Another type of parametric design of (layered) architectures involves allowing flexible topological parameters of the corresponding layered performance models. For example, the communication topology of the clients and servers of the system, the relative call frequencies, number of replicas of objects, etc. can be parameterized in the design itself, and allowed to be changed by the user at run-time, so that the performance required by the user can be achieved.

As discussed earlier, *the more the number of design parameters that can be tuned, the more likely is the design to be scalable.*

As an example of parameterized topologies of (layered) architectures, the following diagrams have been borrowed from Prof. Woodside's course notes on High Performance Software (Course number 94.411).

(a) Tower

Tuning parameters:

- Multithreading levels of tasks
- Number of replicas of each task
- Synchronous Vs. asynchronous calls

(b) Tower with Critical Section

Tuning parameters:

- Multithreading levels of tasks except the critical section
- Number of replicas of the tasks except the critical section
- All calls to CS have to be synchronous, the rest can be sync./async.

(c) Lattice (communicating towers)

Tuning parameters:

- fraction of requests to neighboring tower's corresponding server
- unbalance between the towers
- multithreading and replication levels of all the tasks
- nature of calls (sync./async.)

(d) Lattice with Critical Section

Tuning parameters:

- same as in (c) except that the calls to CS have to be synchronous and the CS task has to be single-threaded with only one copy.

(e) Hourglass

Tuning parameters:

- parameter for invoking more powerful processor in the centre
- parameter for bypassing the narrow middle and requesting directly downwards
- multithreading and replication levels of all tasks
- unbalance between two halves along the vertical axis, and the nature of calls

(f) Pyramid (or inverted funnel)

Tuning parameters:

- parameter for *distribution of load* - vertically and across layers.
- asymmetry of the graph
- nature of calls, multithreading and replication levels of tasks.

3.1.2.3 Logical framework architectures (ODP,TINA)

The logical framework architectures, standardized by ODP [Odp] and TINA-C [Tina], are finding a widespread appeal in some segments of industry such as the communications industry. These architectures break the different concerns of complex distributed system design into *viewpoints*, thus hiding the effects of distribution by supporting distribution transparency. (e.g. access transparency, transaction transparency, location transparency, failure transparency, federation transparency, migration transparency, group transparency, resource transparency). These standards promote openness and flexibility. The design of the example connection management system analyzed in section 6.1 and 6.2 is based on the TINA architectural concepts. Another example is [Jogalekar95]. These concepts may have important repercussions on the scalability of systems.

The viewpoints defined by the logical framework architectures cover a range of abstraction levels, focussing on the intended features of the system to be designed. These viewpoints characterize the overall system behavior for any given scenario, and have a direct or indirect impact on the scalability of the system. The following five viewpoints have been proposed:

(a) *Enterprise Viewpoint*: This viewpoint focuses on the purpose, scope and policies for the given distributed system to be designed.

(b) *Information Viewpoint*: It focuses on the semantics of information and information processing activities in the system. This viewpoint focuses on the problem domain and defines the information bearing entities (objects), relationships between them, and the constraints and the rules that govern their behavior, including creation and deletion. The information bearing entities are not necessarily related to any machinery that provides the mechanisms to support their behavior, although the design decisions from this viewpoint can affect the scalability.

(c) *Computational Viewpoint*: This viewpoint focuses on the functional decomposition of the system objects into objects which are candidates for distribution. Different types of interfaces based on the invocation methodology are defined, and notions such as trading and notification service are introduced at this level. The division of work between objects and the communication required for their collaboration affects the system's performance and scalability.

(d) *Engineering Viewpoint*: This viewpoint focuses on the organization of an abstract infrastructure, enabling execution of the computational objects. The fact that the objects may be executing on different machines, providing different native computational and communication environments, and communicating in a secure environment, is hidden from the applications.

Appropriate tuning parameters from the engineering viewpoint may facilitate efficient implementation and hence improve performance and scalability.

(e) *Technology Viewpoint*: This viewpoint focuses on the choice of technology to implement the system. The distributed platform of the engineering viewpoint can be built on top of a heterogeneous computing environment as long as it conforms to the specifications for the distributed processing environment.

On this note, it is pertinent to observe that the object model for distributed applications, defined by the Object Management Group (OMG), is fast becoming popular and is described in subsection 3.1.5.

As observed in this section, the choice of the architecture style can greatly affect scalability, due to the embedded concurrency, communication patterns, and the implicit overheads imposed by the underlying mechanisms such as the distribution mechanism. Note that the logical framework architectures don't restrict the topology of interconnection between the communicating objects, which is application-specific.

3.1.3 Replication- a possible scalability enabler

This section discusses the role of replication of software resources (or tasks, or “building blocks”) as scalability enablers. Depending on which task(s) in the client-server hierarchy are replicated, topological features of an architecture can be modified (e.g, from a tower to a pyramid shape, as discussed in 3.1.2.2).

Increased availability and enhanced performance are the two most important objectives of replication in distributed systems [Helal96]. Replication can be used not only for enhancing the availability and overall system reliability, but also to enhance performance and scalability. Whether replication indeed helps achieve these goals depends on the system, and also on some other associated decisions (e.g. degree of replication, allocations of replicas to CPUs etc.) Replication of software resources can be either a part of scaling strategy (in which case, the replication forms a part of the scaling path description as discussed in Chapters 5,6.) - or it can be a part of optimization strategy, given a scaling plan. Whether we want to optimize the replication or make

it a fixed part of the scaling strategy depends on the system, its scaling requirements, and the system designer's judgement.

Software replication can improve load-balancing and concurrency between the system's software as well as hardware resources. For example, replicating the read-only data items enables as many read operations as there are replicas, to be performed *concurrently*. Furthermore, each read operation can choose the replica site that minimizes the relevant costs, e.g. the communication distance. The degree of replication, availability, and performance are inter-related and interact in complex ways. High availability through replication need not always lead to high performance or vice versa. Since the scalability metric proposed in this thesis is not designed to measure availability, we use replication only for performance and scalability.

Consider a database replication scenario. If all the database accesses are for read operations, increased replication usually has a positive impact on performance, at the cost of additional storage requirements and the cost in terms of dollars, for multiple site licenses. Replication also enhances the locality of reference, by allowing many read operations to be satisfied locally, resulting in lower response time as well as reduced network traffic.

In the presence of write operations, the scenario is more complex and the analysis of the gains derived due to replication is non-trivial. An update control and propagation mechanism is required, with the associated communication and processing costs. This is necessary to resolve conflicting concurrent updates and to ensure consistency among the various copies. Replication in such a scenario benefits performance only if the gains derived from faster database operations offset such overheads.

Thus, the main factors that decide whether replication will enhance performance are workload characteristics (read/write frequency, respective service times for these operations, etc.), application requirements, (e.g. transaction support), and the replication control algorithm used. In addition, allocation of the replicas to CPUs (load balancing), execution speed of the respective CPUs, threading level of the individual database copies, the network bandwidth and topology, the networking protocols, etc. play an important role in the overall impact of replication on performance.

Many on-line applications such as banking systems require multiple individual operations on several data items (which maybe replicated), in order to process a single service request from a user. Such database accesses are typically made using *transactions*, which group these operations together. Transactions must possess the following **ACID** properties:

- **Atomicity:** a transaction is either entirely performed, or not performed at all,
- **Consistency:** a transaction must take the database (including the replicas) from one consistent state to another,
- **Isolation:** a transaction is isolated from the update activities of other concurrent transactions, and
- **Durability:** a transaction update applied to the database has to be permanent.

The property of consistency implies that any serial execution of the operations within a transaction can be considered correct. Execution of a concurrent, failure-prone transaction is correct if it can be proved to be equivalent to some failure-free serial execution. As discussed in [Helal96], an execution is serial if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other. If the initial state of the database is consistent and if every transaction preserves the database consistency when executed in isolation, then an execution that is equivalent to a serial execution contains no transaction that observes an inconsistent database.

Concurrency control protocols enforce serializability of concurrent transactions executing in centralized or distributed database systems. [Helal96], [Triantafillou95] refer to several references on various concurrency control protocols.

The replication control protocol proposed in [Triantafillou95] is used for the test cases in Chapter 6 and is discussed below. The authors claim that this algorithm achieves the following goals:

- Try to achieve transaction execution delays as close to one-copy systems as possible, if not less.
- Ensure graceful degradation of availability and other performance metrics in the presence of failures. (Not within the scope of this thesis).
- No performance penalties are seen by the application, corresponding to failure recovery of any of the system's components.

The clients of the replicated task (e.g. the database), before starting the process of transaction operations, obtain the location information that informs them of the database sites that contain up-to-date copies of the inquired objects. This, in other words, is the name/directory service, which is provided to the clients by the *location servers*. The location servers are themselves allowed to be replicated for availability and performance reasons, and the clients can use the fol-

lowing two interface operations on any of the replicas of the location server, which are explained below using the same notation as used by the authors:

- *server?*(list-of-objects) returns list-of(object, list-of-replicas)
- *update-table*(list-of(object, list-of-replicas));

Location server entries are locked using two types of locks, *slocks* which are associated with the *server?* operation and *ulocks* associated with the *update-table* operation. If nearly all the transactions do a read-only operation on the database, the *server?* queries don't have to be queued on u-locks. This is the case with the call processing system's database (discussed in section 6.3), and the overheads associated with blocking on the either of these locks can be neglected. Otherwise, they may introduce a bottlenecking effect at the location server. If the location server does not have "enough" number of replica, the clients may have to wait too long to obtain either the s-locks or the u-locks, and if the location server has too many replica, the non-atomic multicast for synchronizing the state of the "fresh" database objects may introduce excess performance penalties. Therefore, an optimum number of location servers need to be used, depending on the application.

The authors present an algorithm for asynchronous replicated lock acquisition, without introducing serializability problems. The concurrency-control algorithm presented herein belongs to the general category of two-phase locking (2PL) algorithms. (i.e., the lock acquisition phase is distinct from and precedes the lock release phase). The novelty of this algorithm is that the appropriate lock is synchronously obtained at only one replica, called the leader replica. After the operation is executed, the leader replica sends an acknowledgment to the client and (asynchronously) broadcasts lock-acquisition messages to other replicas. Upon receiving such a message, the non-leader replica will (locally) obtain a lock. Thus, the non-leader replicas acquire locks requested by a client in parallel with the client's execution of subsequent operations.

A high-level description of the steps of this algorithm, when a client wants to perform a transaction is as follows:

1. A client submits a *server?* query to the closest location server replica, and obtains a *hint* regarding the location of the *up-to-date replicas* of the desired object.
2. Clients use the hint to select any one of the up-to-date replicas as the leader and submit the operation to it.

3. Clients need only wait for the leader to respond and can then continue with the execution of the next transaction operation.
4. Each leader, after replying to the client, tries to inform “enough” other replicas of locks that need to be acquired and, in case of a write, the operation to be executed. Thus, the non-leader replicas participate in parallel with the execution of the client.
5. During 2PC, each leader ensures that each operation was performed on the required number of replicas, otherwise the transaction is aborted.
6. If the client notices that the set of replicas participating in a write operation is different than the set in the hint provided by the extended location service, the client multicasts the updateable request to the location servers.

In the scalability analysis presented on the test cases in this thesis (Chapter 6), the *database* and the *location server* are the objects of interest for studying the effects of replication. The replication overheads have been accounted for (in terms of extra latencies and execution overheads), but the relatively smaller bottlenecking effect due to the locking algorithms that manage the locking / unlocking of the individual database objects has been ignored. The rationale for doing so is explained in Chapter 6 for the individual cases. Remote invocation overheads have been fully accounted for (in terms of extra execution on the client as well as the server CPUs due to stub overheads; as well as extra latencies). Note that in addition to consistency management overheads, the remote procedure invocation overheads are also interleaved into the problem, if the replicas are remote with respect to the client.

Replication of some software tasks in the system can act as a scalability enabler for the overall system (as a parameter of the design, as discussed in section 3.1.2), if these overheads can be offset by the throughput and QoS gain derived due to replication. *Replication can help achieve a fine-grained control over concurrency and load-balancing among hardware and software resources in the system, and can be a powerful scalability enabler.*

3.1.4 Allocation of software objects to CPUs - a possible scalability enabler

This sub-section presents the association between the allocation problem and the scalability analysis of systems, along with some literature survey. The allocation problem has been an

active area of research for distributed systems, and the problem is important for scalability analysis, due to the following reasons:

- The allocation affects the inter-process communication delays (e.g. remote procedure invocation versus local procedure invocation), and also the (client and server stub) execution overheads associated with it.
- It also affects the execution concurrency between different system resources, when simultaneous requests are received from different clients.
- It affects the load balancing.

Thus, the resource allocation decides the “hot spots” in the system, which in turn affect the scalability of the system. Appropriate allocation of software resources to hardware resources can thus be treated as an enabler for scalability.

Static processor-task allocation is one of the extensively researched areas in the literature. Various solution techniques have been reported. Typically, the objectives are to minimize the inter-process communication, achieve some form of load balancing and/or maximize parallelism. (In case of this thesis, the scalability metric presented in chapter 4 is to be maximized).

The solution techniques can be roughly categorized as graph theoretic methods, mathematical programming methods, methods with statistical guarantees of optimality (e.g. simulated annealing) and heuristic methods. The following is a brief survey of some examples of such analysis.

(1) [Bokhari88] addresses the problem of optimally assigning the modules of a parallel program over a set of processors. The author observes that when the tasks are allowed to execute concurrently, most kinds of allocation problems become NP-complete. This paper discusses variants of the partitioning problem for parallel and pipelined programs. The solution method for pipelines uses the shortest path algorithm to the layered graphs to find the minimum bottleneck path. An algorithm is presented for finding an optimal “sum-bottleneck” path in doubly-weighted graphs. The weights represent the cost being minimized.

(2) [Woodside93] present a fast version of the bin-packing algorithm with heuristic extensions for statically allocating undirected task-graphs to a set of homogeneous, bus-connected processors. The nodes of the graph are associated with the processing time and the arcs represent

communication cost between processors. The authors present various options for initial task sizing (based on the execution and communication times), task ordering criteria and placement criteria. The modified bin-packing algorithm is fast and rewards concurrency between tasks and penalizes the communication overheads. The results of this algorithm were compared for a variety of problems and it was found that the heuristic policy choices produced nearly optimal allocations, for large sized problems.

(3) [Lo88] discusses a heuristic algorithm which combines the recursive invocation of the Max Flow / Min Cut algorithms with a greedy algorithm to find suboptimal assignment of tasks to processors. The author introduces the notion of “interference costs” for the collocated tasks, which indicate resource contention between collocated tasks. The author observes that minimization of the inter-process communication can conflict with load balancing, and the optimality of the solution is affected by their magnitude.

(4) [Chu87] discusses a strategy for module replication and assignment for real-time distributed systems. An analytic model is developed for estimating task response time, by considering such factors as inter-process communication, module precedence relationship, module scheduling, interconnection network delay, etc. Each CPU is assumed to be an independent, M/G/1 queueing system, in order to approximately calculate the response time of each module in the task graph. The objective function is based on the sum of task response times and the delay penalties for the violation of thread response time requirements. Since module assignment as well as its replication affects the task response time, the assignment algorithm iteratively searches for module assignments and replications that reduce the task response time. The computational complexity of the algorithm is observed to be high.

(5) In [Sevcik93], the author identifies effective algorithms for application scheduling and processor allocation in parallel processor systems. The primary objective is to keep the average response times low, taking priorities into account. The algorithms developed in this paper are for large-scale parallel applications, and are directly applicable to uniform memory access systems, and possibly to non-uniform memory access (NUMA) shared memory multiprocessors, message passing multicomputers, and workstation clusters. The author does not consider time-sharing of processors among the applications.

The author presents a survey of some of the prior work on scheduling algorithms that use information about individual application characteristics. e.g. the case when applications are assumed to require a fixed number of processors to execute, for a specified service time. A scheduling model that takes into account the fixed-size speedup (as defined in 2.1.1) for all the applications as a function of the number of processors assigned, and the parallel processing overhead, has been presented. The authors focus primarily on scheduling a batch of applications in the absence of arrivals.

(6) Graph partitioning algorithms can be applied to some versions of the allocation problem. An interesting graph partitioning algorithm is presented in [Oommen96]. The authors present a learning-automata based solution to an NP-graph partitioning problem.

(7) In [Ostlad95], the authors present a dynamic programming algorithm for partitioning a sequence of real numbers into a given number of subsets. This approach might be useful for some versions of task allocation problems, where the cost functions over each of these subsets can be individually calculated (independent of the other subsets), and which have some other mathematical properties in addition, as discussed in the paper.

(8) [Lee91] presents the optimal workload allocation among the stations of a multi-server product-form closed queueing network. The objective function is throughput. The authors prove that the throughput is a pseudo-concave function for some special cases of multi-server closed queueing networks, and use this property for the optimal allocation.

(9) Simulated annealing [Laarhoven87],[Ingber93],[Bilbro92] [Bilbro91] etc. is an increasingly important technique for finding near-optimal solutions for NP-hard problems. This technique allows very general objective functions, and seems to be suitable for the allocation problem under consideration. The objective function can have quite arbitrary degrees of nonlinearity and discontinuities. Simulated annealing gives a statistical guarantee of finding an optimal solution. Moreover, the implementation of the simulated algorithm is relatively easier.

The implementation of the simulated annealing used in this research is discussed in Chapter 5. The drawbacks of simulated annealing include the slower speed in finding an optimal fit in case of some problems. Also, fine-tuning the technique for specific problems can be time consuming. In some cases, a trade-off with the weak ergodic property (such as a faster cooling schedule) can lead to much faster solutions.

3.1.5 DCE and CORBA overview

This section discusses some of the popular middleware platforms, for implementing distributed software architectures. Section 3.1.5.1 talks about the OSF Distributed Computing Environment (DCE) and section 3.1.5.2 talks about the Common Object Request Broker Architecture (CORBA). These distributed environments aim at providing some degree of distribution transparency by providing a communication framework that allows the clients and servers to communicate. The target systems for which this thesis proposes a scalability analysis framework, would typically be implemented using such middlewares as DCE/CORBA. The connection management systems discussed in sections 6.2 and 6.3 are implemented on top of the DCE platform.

3.1.5.1 The OSF Distributed Computing Environment (DCE)

The OSF DCE provides a client-server paradigm for distributed applications, using remote procedure calls. On client as well as server side, automatically generated “stubs” support the remote procedure calls. The client stub communicates with the server stub using the RPC run-time library [Shirley94].

The server’s RPC run-time library receives the remote procedure call and transfers the client information to the server stub. The stub, then invokes the remotely called procedure in the server application. After the server finishes with the procedure, the server stub communicates with the client stub using the RPC run-time library. In the end, the client stub returns to the client application code.

The distributed application development is started by first defining an interface between a client and a server. The interface definition, defined in the file with a “.idl” extension, contains a set of procedure declarations and data types. Every interface is associated with a universal unique identifier (UUID) which is generated by a utility called *uuidgen*.

The entities in the square brackets are optional.

The client:

The client program includes the header file automatically produced by the IDL compiler. While compiling, the client application, the client stub (produced by the IDL compiler) and the DCE libraries (usually a dce and a cma (threads) library) are linked. The IDL supports various data types and pointers of two types (“full” pointers and “reference pointers”). In case of reference pointers, the data can change but not the address itself and they have a performance advantage.

Binding:

In order that the client sends an RPC to the right server, the client creates a “binding”, puts appropriate information in it and lets the RPC run-time library find the server. Binding information includes the communication protocol sequence, the address of the host (typically IP address), and the server process address (endpoint) on the host (typically, port number). The binding can be automatic, implicit (handle held in the global area of client stub), or explicit (has to be included as a procedure parameter). The automatic binding has no security. Client application code has to manage binding handles except when automatic binding is used [Shirley94].

A “partially bound” handle, e.g. a handle that the client process receives from the cell directory service, contains a protocol sequence and the server host address but not the endpoint (address of the server process). The RPC run-time obtains this information from the endpoint map maintained by the server’s host. The dynamic endpoint mapping reduces network management overheads. The cell directory server (CDS) can have information about the specific (application) servers, or database names that have a set of servers, or a search list for the servers.

It is possible to customize a binding handle by adding extra information that might be required by the application. Customized binding information can only be used with explicit or implicit binding methods.

The Server:

The server initialization code makes a sequence of run-time calls in order to enable the server to listen to remote requests. These include [Shirley94]:

- register the interface with RPC run-time library
- create server binding information by selecting one or more protocols
- store the binding information in a name service database
- manage the endpoints in local endpoint map
- wait and listen to the client RPCs

In addition to the interface names, DCE allows the clients and servers to export or import binding information based on the UUID. This feature is useful, for example, when the same server manages a variety of resources. DCE provides “object UUIDs” for uniquely identifying a resource managed by a server. The object UUIDs can be stored in the namespace and in the endpoint map and it could become a part of the binding handle.

Security:

A DCE *principal* is an entity capable of bearing an identity and verifying another principal’s identity. The principals in a DCE cell trust the **security server** of the cell. Each principal has a secret key derived from the principal’s password. This key is known only to that principal and the security server. All principals must first login before accessing DCE resources. DCE uses a secret key based protocol (Kerberos version 5) to perform authentication when a client communicates with a server using an *authenticated RPC*. In addition, DCE provides an *authorization mechanism* using *reference monitors*. A reference monitor can use the client’s identity and group membership and the type of the operation requested, for making an authorization decision.

3.1.5.2 Common Object Request Broker Architecture (CORBA)

The Object Management Group (OMG) came up with a distributed object computing architecture - CORBA. To support distributed object computing, the following two things are required:

- A mechanism to define interfaces between (possibly remote) objects and
- A mechanism for communication between these objects.

For specifying the interface to an object, CORBA defines a specification language called the Interface Definition Language (IDL) - like in case of DCE. An interface definition is similar to a C++ class definition. The interface offered by an object is a collection of its attributes and operations, which can be accessed by its clients. A typical CORBA implementation provides bindings to C, C++, Smalltalk, Java, etc.

For transparent communication between objects, CORBA defines a mandatory protocol that all vendors must support. The Object Request Broker (ORB) is a software component that provides the mechanisms by which objects transparently communicate with each other. The objects might interact with other objects in the same process, in other processes on the same system, or in other processes on a different system through the ORB.

Clients can access the server objects either *statically* or *dynamically*. In case of static invocation, the clients know the IDL interface of the server object at compile time. In case of dynamic invocation, the client object need not know the IDL interface of the server object at compile-time, and calls to the server are built at run-time. CORBA's static interface is very similar to an RPC interface.

To a server, the client's use of static or dynamic invocation mode is transparent. For the dynamic invocation mode, the object interface information is accessible at run-time, in the *interface repository*. This repository contains the information required by a client to build a correct request, including the operation names and parameter types.

That part of ORB which directly communicates with the server object is called the *object adaptor*. It keeps track of, and can start, server objects if required. In order that an object adaptor could instantiate a server on demand, information describing the object implementations is kept in the implementation repository.

Thus, the CORBA architecture can be summarized as follows:

3.2 Layered queueing models

The distributed systems which are of particular interest for this research are client-server systems implemented using middlewares such as the ones in 3.1.5. Such systems can be modeled as layered queueing networks. The wide class of systems that can be modeled as layered queueing models include the systems that use remote procedure calls (RPCs), open distributed processing systems, systems using concurrent object-oriented frameworks, systems based on real-time kernels, etc. The layered queueing models are extensively used in this research for fast (iterative) performance evaluation of client-server systems. Various architectural patterns such as the ones discussed in section 3.1 can be captured in the layered queueing models.

The scalability analysis framework presented in Chapter 5 requires a complete description of the software design as well as the scaling strategy. Layered queueing models effectively capture the former, i.e. the performance related attributes of a broad range of software designs that will be considered under the scope of this thesis. The layered queueing models and fast analytical techniques for their solution have been described in [Rolia95],[Woodside95].

Bottleneck removal is an integral part of improving the scalability of a system. In the (distributed) client-server systems, the bottlenecking can occur at hardware level or in the software. In the synchronous communication paradigm among objects, a client task must wait in a blocked state until the server responds to its request. (e.g. a blocking RPC). The server can itself be a client of other lower-level servers, and may have to block until some or all of them finish their execution. This may introduce a software bottlenecking effect, where clients block for software resources, even when the underlying hardware is not sufficiently utilized. Device bottlenecks (e.g. CPU or a disk or a bus bottleneck) are easier to detect because the bottlenecking device gets saturated while other devices don't. Other kinds of bottlenecking may be due to critical sections, locking of data, blocking of processes, etc., and lead to a more complex behavior due to simultaneous resource possession. This effect, which is not captured by the traditional queueing models, can be analyzed by the layered queueing network solution technique. For analyzing the effect of software as well as hardware bottlenecks in the system, layered queueing models have been used in this thesis.

The layered queueing models also facilitate a general interaction paradigm between computational objects, unlike the traditional queueing models. For example, the communication between clients and servers can be synchronous, asynchronous, or synchronous with some post processing (called "second phase" in the models- i.e. the server unblocks the client early, and does some post-processing.) This allows the capture of a wide spectrum of interaction semantics between clients and servers, and the models sufficiently capture most of the possibilities, except that the requests between objects have to be acyclic.

The scalability analysis framework proposed in Chapter 5 requires the following performance related attributes and "tuning parameters": execution demands of all the entries of all the tasks, allocation map of the tasks to the processors, execution priorities and multiprogramming levels of the tasks, scheduling policies at the software and hardware resources, relative call frequencies between the entries (service interfaces offered by the tasks), etc. These are all captured by the layered queueing models which in turn output the throughputs, utilizations and response times for all the tasks, entries and hardware resources. These are the output parameters required by the scalability analysis framework for optimization decisions.

Thus, the layered queueing modeling has been found to be an adequate modeling framework for the scalability analysis proposed herein.

The toolset discussed in [Franks95] has been used for iterative solution of the layered queueing models, required by the scalability analysis framework discussed in detail in Chapter 5.

Chapter 4 The Scalability Metric

This chapter discusses a new scalability metric for distributed systems. Several scalability metrics used for massively parallel systems were discussed in section 2.1, and as previously observed in section 2.4, these metrics are not suitable for scalability analysis of distributed systems.

Section 4.1 defines the new metric. In section 4.2, the metric is applied to some idealized systems and some theoretical results are derived, which agree with the intuition.

4.1 Defining the scalability metric

Let any given system configuration (with a specified allocation map of its software resources to the hardware resources, replication state of its various software tasks, multiprogramming levels and execution priorities of all the tasks, etc.) be called reference configuration #1. Let one or more attributes of the system be scaled taking it to another configuration #2. If the value of the scalability metric (defined below) is greater than $1 - \epsilon$, it indicates that the system is “scalable” from configuration 1 to configuration 2. In other words, it indicates if this move to another state buys any extra performance at a reasonable provisioning cost.

The metric depends on configuration #1 and configuration #2, and in general can be defined as:

$$\psi = F(\lambda_2, QoS_2, C_2) \div F(\lambda_1, QoS_1, C_1) \quad \dots \text{Eqn(4.1)}$$

where F evaluates the performance and the economy of operation (defined to be the *productivity* of the system) at two different scales of deployment. The reasoning behind the selection of variables in F is that:

- λ evaluates the rate of providing valuable services to users, which is related to revenue capability,
- QoS is a set of parameters which evaluate the quality of the service seen by users,
- C reflects the cost of providing service.

It is clear that F should increase with λ and with better QoS and should decrease with C . The QoS parameters depend on the system requirements and can be quite general, depending on the application. (They could include the mean value of response time, standard deviation, etc.).

Thus, the productivity of a system configuration is the figure of merit that is important in order to quantify and analyze scalability of the system to any other configuration.

The cost function could be expressed as a function of the total number of available flops, the total amount of investment in dollars associated with the system configuration, weighted sum of the number of hardware and software modules, total RAM available in a configuration, etc. This cost can be some arbitrary function of the actual provisioning cost (as against the actual provisioning cost) in dollar terms, and depends on the application, and views of the system planners. e.g. in some of the cases analyzed in Chapter 6, the cost is assumed to be a linear function of the number of CPUs in the system and the number of database replica.

Definition: A system is said to be scalable from state 1 to state 2, if $\psi \geq 1 - \epsilon$. (The choice of the exact value of ϵ depends on the given system and the judgement of the evaluator. In the analysis of the test cases presented in Chapter 6, $1 - \epsilon$ is either 0.8 or 0.9.)

Note that while the scalability metric captures the productivities of two system configurations, it does not restrict the policy by which a system could be scaled. The metric thus allows comparative evaluation of various scaling strategies, which is discussed in Chapters 5 and 6.

We will now discuss some specific forms of equation (4.1), in which QoS is represented by a function of the mean response time T .

Scalability metric based on the “Power” function:

A specific instance of the scalability metric, called *P-scalability*, has been examined in [Jogalekar96], and is briefly discussed below. This metric was used for the scalability analysis of distributed systems earlier in this research, but was later modified as discussed below.

In the abovementioned paper, the “power” measure defined by [Giessler78] was used to derive the expression for productivity.

$$Power = P = (\lambda/T) \quad \dots \text{Eqn(4.2)}$$

where T = the mean response time seen by the clients, for completing a service request.

Treating $1/T$ as a simple measure of the quality of service seen by the clients, the productivity function for a given configuration i was defined in terms of the Power metric as follows:

$$P\text{-productivity} = F_{P,i} = \frac{Power}{Cost} = \frac{\lambda_i}{T_i \cdot C_i}$$

where: $\frac{\lambda_i}{T_i}$ = the power associated with the given configuration identified by index

i , and

C_i = the cost of provisioning for the given configuration i .

When the system is scaled from configuration 1 to configuration 2, the P-scalability metric is then defined as

$$\psi_P = \frac{F_{P,2}}{F_{P,1}} = \frac{\lambda_2}{\lambda_1} \cdot \frac{C_1 T_1}{C_2 T_2} \quad \dots \text{Eqn(4.3)}$$

In the paper referred to above, there is an example of how different scaling policies can be quantitatively compared, using this simple version of the scalability metric.

The terms in equation 4.3 give each configuration a value proportional to throughput, divided by a “penalty” which is a product of delay and the cost of the configuration. If configuration 2 handles k times the throughput for k times the cost and an equal response time, it gives $\psi_P = 1$. This indicates a perfectly “scalable” system. A lower throughput with a lower delay can also give $\psi_P = 1$.

A factor in favor of definition (4.3) is that “power” has been used to evaluate a range of systems, for example by [Kleinrock78], [Kleinrock79], [Giessler78], and has given a useful measure of utility of the system. “*P-scalability*” is however *almost always unrealistic*, because it puts too much emphasis on the value of very small response times. It is not usually true that halving the response time doubles its value, as implied in equation 4.3. This definition of ψ_P assigns infinite value to a zero response time, and halving the response time exactly offsets a doubling of cost in the measure, no matter how small T already is. This metric is therefore not felt to be suitable for the analysis presented herein and a better scalability metric is discussed below.

Derivation of a scalability metric suitable for distributed systems:

QoS functions for specific types of systems are typically represented as a function of parameters like mean response time, jitter, probability of packet loss, variance of response time, etc. The choice of the scalability metric does not limit the scalability analysis framework presented herein, but in order to focus the discussion, this thesis uses a general QoS function based

on the mean response time and the user-specified target response time. In the distributed systems of interest, the users typically expect the service completion time to be close to some target response time, $\hat{T}$, such that below $\hat{T}$ the value of reducing T has diminishing significance. To represent this we will represent T in F by a factor $f(T/\hat{T})$, giving the value, to a user, of a change from $\hat{T}$ to T . For $T > \hat{T}$, f decreases toward zero, and for $T < \hat{T}$, f is near to 1. The other arguments will again be represented as in equation (4.1), giving:

the productivity of configuration $i = F_i = \frac{\lambda_i}{C_i} \cdot f\left(\frac{T_i}{\hat{T}}\right)$ and

$$\psi = \frac{F_2}{F_1} = \left[(\lambda_2) \times f\left(\frac{T_2}{\hat{T}}\right) / (C_2) \right] \div \left[(\lambda_1) \times f\left(\frac{T_1}{\hat{T}}\right) / (C_1) \right] \quad \dots \text{Eqn(4.4)}$$

Various functional forms for $f(T/\hat{T})$ are illustrated in Fig.4.1. Compared to ψ_p , version (a) reduces the impact of T when $T < \hat{T}$. Version (b) modifies the shape of this relationship approaching a threshold function as $n \rightarrow \infty$. The threshold function itself is given by (c). Another way to approach the threshold is given by (d) and (e). An interpretation of the threshold function is a hard limit on T , such that T should be less than $\hat{T}$ for the responses to have any non-zero value. Note that T is the average response time, so this does not impose a hard deadline on every response. We assume a non-zero denominator in cases such as (e). The remainder of this thesis uses version (a).

$$(a) \quad f_a\left(\frac{T}{\hat{T}}\right) = 1 / \left(1 + \frac{T}{\hat{T}}\right)$$

$$(b) \quad f_b\left(\frac{T}{\hat{T}}\right) = 1 / \left(1 + \left(\frac{T}{\hat{T}}\right)^n\right), \quad n \geq 1$$

$$(c) \quad f_c\left(\frac{T}{\hat{T}}\right) = 1 \quad \text{for } T \leq \hat{T}; \quad 0 \text{ otherwise}$$

$$(d) \quad f_d\left(\frac{T}{\hat{T}}\right) = K_T \cdot \left(1 - \frac{T}{\hat{T}}\right) \quad \text{for } T < \hat{T}; \quad 0 \text{ otherwise}$$

$$(e) \quad f_e\left(\frac{T}{\hat{T}}\right) = 1 / \left(1 - \left(\frac{T}{\hat{T}}\right)^\alpha\right) \quad \text{for } T < \hat{T}; \quad 0 \text{ otherwise}$$

FIGURE 4.1

Some QoS functions for the scalability metric ψ

Both the theoretical and practical cases considered in this thesis use the *QoS* function (a) from fig. 4.1. This function rewards the response times $\leq \hat{T}$ in a smooth and bounded fashion, and penalizes the response times $\geq \hat{T}$, leading to a zero *QoS* as the mean response time grows towards infinity.

When substituted in the productivity and scalability expressions, this function has been seen to yield satisfactory results, as will be seen in section 4.3 and in Chapter 5. As long as the mean response time is reasonably close to the target response time, improving throughput of a distributed system is more important than reducing delays, as observed earlier, and the function (a) conforms to this notion.

Substituting in equation (4.4) and simplifying, we get the following expression for scalability, from configuration 1 to configuration 2:

$$\psi = \frac{\frac{\lambda_2}{C_2} \cdot \frac{1}{1 + \frac{T_2}{\hat{T}}}}{\frac{\lambda_1}{C_1} \cdot \frac{1}{1 + \frac{T_1}{\hat{T}}}} = \frac{\lambda_2 \cdot C_1 \cdot (T_1 + \hat{T})}{\lambda_1 \cdot C_2 \cdot (T_2 + \hat{T})} \quad \dots \text{Eqn(4.5)}$$

This is the scalability metric that is used in the rest of this thesis. Before we test this definition of the scalability metric on some of the familiar theoretical systems in section 4.3, following properties associated with the scaling operation, (using this scalability metric), need to be pointed out:

(1) Reflexivity:

An operation of scaling a system is reflexive. That is, a system is scalable (from any given state) to itself. (The definition of scalability implies $\psi = 1$ if nothing is scaled in the system).

(2) Transitivity:

If a system is scalable from state 1 to state 2, and is scalable from state 2 to state 3, then it is scalable from state 1 to state 3. (This can be easily seen, as the productivity expression for state 2 cancels out when $\psi(1, 2)$ is multiplied by $\psi(2, 3)$). This statement holds for *any* state 2. Thus, the value of the scalability metric corresponding to a scale-up from state 1 to state 3 is independent of the path followed in doing so. For example, a system can be scaled either by adding CPUs first and extra database servers later, or vice versa, as long as the final configuration remains the same.

(3) Cost scaling:

The metric remains unchanged if all the factors that contribute to the cost (i.e., the cost in terms of fixed assets, plus the cost incurred for “tuning” the scalability enablers that may allow the system to scale in a cost-effective way - e.g. adding extra database site licenses) are scaled by the same factor. It only depends on the relative cost of the resources of the two system configurations. Thus, theoretically, if a system is deployed at a given scale in two different countries with different currencies, the value of the metric will not change.

If the scalability enablers are inexpensive (relative to the overall system cost), more choices can be available to maximize the value of the scalability metric, by tuning these inexpensive configuration parameters. This is discussed in the following chapter.

(4) Time scaling:

The scalability metric is dimensionless, and scaling the units of time does not affect its value.

The following sections apply the scalability metric (defined by Eqn. 4.5) to various well-known idealized systems and verify that the results are intuitively reasonable.

4.2 Scalability analysis framework: some concepts and terminologies

For scalability analysis, a distributed system description should include a plan for scaling up the system, or the *scaling strategy*, which determines what will be augmented (or down-sized) in a scaled-up (scaled-down) version. *Note that this information is required in addition to the description of the system configuration that is required for the creation of its performance model.* For example, the scaling strategy can be to replicate some specific objects of the system, or simply to increase the number of CPUs. What resources could be scaled in a given distributed system is up to the designers and the system planners, and can be arbitrary. We will suppose that there is a single discrete *scale factor* k that encapsulates some notion of the “size” of the system. (This is easily generalized to multiple scale factors, or a vector scale factor). The scaling strategy is characterized by a set of *scaling variables* defining the changes which will be functions of k . Thus, it can be imagined that the “size” of the system is decided by the value that k takes.

The scaling variables together provide a *scaling path* through the set of possible system configurations, as k is varied. The scaling path can be arbitrarily defined. For example a hypothetical scaling path for an information system might be chosen so that the following three scaling variables are determined as functions of k :

- $Nusers = k$ = the number of active users
- $Datasize = 10000 \log k$ = the assumed path for the size of the database
- $Nproc = k/100$ = the number of processors to be provided

Figure 5.1 illustrates the scaling variables and the scaling path in the space (Datasize, N_{proc}). It shows a curve over $k = 100$ to 300.

FIGURE 4.2

Scaling Variables and the Scaling Path

As k is increased and new processors are added, a host of other variables need to be determined, other than the ones that are explicitly defined to change with k . They could include the allocation of processes to the processors, the replication of processes, including replication of the database, the creation of threads within processes, the memory available for buffers, tuning of the middleware parameters, choice of communication protocols, and priorities of processes. Which of these to use depends on the nature of the distributed system and the views of the planners. We will call all of these configuration variables **scalability enablers**. As k varies the permissible values of scalability enablers may be constrained by *feasibility constraints* derived from the needs of operating the system, including quality-of-service constraints.

The notions scale factor and scalability enablers are used in the following discussion to test the scalability metric using some idealized systems. A discussion of a systematic approach towards optimization of such scalability enablers is postponed until section 5.1.

4.3 Scalability metric applied to idealized cases

One demonstration of the “reasonableness” of a scalability metric is to show that it gives an intuitively consistent result for an example which gives an analytic solution, and which intuition suggests is, or is not, scalable. Eight different idealized cases are analyzed here, using the following metric which has been derived in eqn(4.5).

$$\psi = \frac{\lambda_2 \cdot C_1 \cdot (T_1 + \hat{T})}{\lambda_1 \cdot C_2 \cdot (T_2 + \hat{T})}$$

Example A: A speed-scaled open system with proportional costs.

Here we consider any system architecture and behavior (not necessarily solvable analytically) which has external arrivals and a steady state. Configuration 2 is uniformly sped-up or time-scaled by a factor k , so the input rate and every component (computers, networks, storage) is faster. Then $\lambda_2 = k\lambda_1$, $T_2 = T_1/k$, and a little manipulation of equation (4.5) gives:

$$\psi(k) = \frac{k^2 C_1}{C_2} \cdot \frac{T_1 + \hat{T}}{T_1 + k\hat{T}} = \frac{k C_1}{C_2} \left(1 + \frac{(k-1)T_1}{T_1 + k\hat{T}} \right)$$

If the cost is also scaled up by a factor k then

$$\psi(k) = 1 + \frac{(k-1)T_1}{T_1 + k\hat{T}}, \quad \lim_{k \rightarrow \infty} \psi(k) = 1 + \frac{T_1}{\hat{T}}$$

This example has positive scalability with a bounded increase. If $T_1 \approx \hat{T}$ then ψ levels off around a value of 2. If the cost C_2 rises more rapidly than linearly with k , ψ will eventually decline as $(kC_1/C_2(k))$.

This agrees with intuition. The throughput increase pays for the extra cost in exact proportion, and the shorter response time provides a bonus which is bounded because $f_a (=$

$\frac{\hat{T}}{\hat{T} + T}$) is bounded as T goes to 0.

Example B: Scalability metric applied to M/M/m systems.

Consider the following scaling strategy for an M/M/m system:

Consider a scaling strategy where the number of servers, m , is scaled by a factor k . The provisioning cost, C , is assumed to scale by the same factor. Intuitively, the system should be able to support (at least) k times the throughput while keeping the response time close to the original response time, in order to be classified as scalable. We now show that this condition indeed implies $\psi > 1$.

Substituting the expressions for throughput and waiting time for an M/M/m system in the expression for the productivity,

$$F = \frac{k \cdot \lambda}{m \cdot k \cdot C} \cdot \left(\frac{1}{1 + \frac{1}{\bar{T}} \cdot \left(S + \frac{\rho \cdot P_Q(k)}{k \cdot \lambda \cdot (1 - \rho)} \right)} \right)$$

where λ = the base throughput of the system,

m = the number of servers in the system,

C = cost of each of the m servers,

S = service time of any of the servers,

ρ = utilization of every server,

$P_Q(k)$ = Erlang C formula, which is a function of the scale factor in this case, given by

$$\frac{\left(\frac{(km\rho)^{mk}}{(km)! \cdot (1 - \rho)} \right)}{\left(\sum_{n=0}^{km-1} \frac{(km\rho)^n}{n!} + \frac{(km\rho)^{km}}{(km)! \cdot (1 - \rho)} \right)}$$

As can be observed, for any k , $P_Q(k)$ is bounded above by 1, and bounded below by 0, since it is the probability that an arrival will find all servers busy and will be forced to wait.

As $k \rightarrow \infty$, $F \rightarrow \frac{\lambda}{mC} \cdot \left(1 + \frac{S}{\bar{T}} \right)^{-1}$. Thus, the scalability metric approaches the value

$$\frac{1 + \frac{1}{\bar{T}} \cdot \left(S + \frac{\rho \cdot P_Q}{\lambda \cdot (1 - \rho)} \right)}{\left(1 + \frac{S}{\bar{T}} \right)}$$

and the response time approaches S , which is clearly bounded, and less than the original response time of $\left(S + \frac{\rho \cdot P \cdot Q}{\lambda \cdot (1 - \rho)}\right)$

The scalability limit derived above is clearly > 1 , and the upper bound depends on the parameters of the reference state. It indicates that the system is infinitely scalable according to the proposed scaling strategy, for any system parameters. Thus, the limiting value generated by the metric agrees with the intuitive notion of scalability for this system.

Example C: Closed idealized balanced system scaled in users and speed.

Consider an idealized system with a separable queueing network performance model with K servers, in which the population N and the (single, constant-rate) server speeds are scaled by k . For simplicity, suppose this is a balanced system with equal demand D sec per response at each of the K servers. Then the throughput and response time are $\lambda = \frac{N}{[(N + K - 1)D]}$, $T = \frac{N}{\lambda} = (N + K - 1)D$.

Intuitively, the given system must be scalable using this strategy, meaning that the corresponding scalability metric value must be ≥ 1 .

Substituting into equation (4.5) with $N_2 = kN_1$, $D_2 = D_1/k$ and $C_2 = kC_1$ we get

$$\psi(k) = \frac{k(N_1 + K - 1)}{(kN_1 + K - 1)} \cdot \frac{(N_1 + K - 1)D_1 + \hat{T}}{(kN_1 + K - 1)\left(\frac{D_1}{k}\right) + \hat{T}}$$

$$\lim_{k \rightarrow \infty} \psi(k) = \frac{(N_1 + K - 1)[(N_1 + K - 1)D_1 + \hat{T}]}{N_1[N_1D_1 + \hat{T}]}$$

By inspection this limit is greater than 1.0, but if $K < N_1$ the limit approaches 1.0.

If the system is not balanced then there will be a bottleneck server (having the largest value of D). In this case as k increases, N becomes large, $\lambda \rightarrow 1/D_{max}$ because of the bottleneck. As the servers are all scaled up, $D_{max,2} = D_{max,1}/k$, we have $\lambda_2 = k\lambda_1$ and thus $T_2 = N_2/\lambda_2 = N_1/\lambda_1 = T_1$. Thus for large enough k , $\psi \rightarrow 1.0$ just as in the balanced case with large N_1 .

Example D: System with a single bottleneck.

The usual way to identify non-scalability is to discover a single bottleneck which cannot be speeded up. The effect of such a bottleneck on ψ is straightforward; it limits λ to some value λ_{max} . In a closed separable system it is easy to see that $T \rightarrow N/\lambda_{max}$, so if $N_2 = kN_1$ and $C_2 = kC_1$, then for large k

$$\psi(k) \approx \frac{\lambda_{max}}{\lambda_1} \cdot \frac{C_1}{kC_1} \cdot \frac{N_1/\lambda_1 + \hat{T}}{kN_1/\lambda_{max} + \hat{T}}$$

$$\psi(k) \rightarrow \left(\frac{\lambda_{max}}{\lambda_1} \right)^2 \frac{1}{k^2}$$

This gives a sharp decline towards zero which clearly identifies the non-scalability of the example.

Example E: General closed product-form queueing networks with a single class.

Example C applies the scalability metric to a special case of product form queueing networks, where all the servers have the same demand. In this example, we justify the metric by applying it to general product form queueing networks. The subsection (E.I) studies the scalability when all the devices are scaled by the same factor. Subsection (E.II) deals with a closed queueing system *with a single bottleneck*, and analyze the scalability only with respect to the speed of the bottleneck, by applying the scalability metric in equation (4.5).

(E.I)

Using the formulae for mean value analysis solutions of product form queueing networks, the productivity can be formulated as:

$$F = \frac{1}{C} \left(\frac{N}{\sum_{i=1}^K D_i(1 + L_i(N-1))} \right) \cdot \left(\frac{1}{1 + \frac{1}{\hat{T}} \cdot \left(\sum_{i=1}^K D_i(1 + L_i(N-1)) \right)} \right)$$

where

N = the number of customers in the closed, product form queueing network,

K = the number of queues in the queueing network,

D_i = total resource demand (for all the visits to resource i during a cycle) for any of the customers at queue i ,

$L_i(N-1)$ = the average number of customers at the queue i with one less customer in the system.

C = the cumulative cost of all the servers in the system, with the cost of each individual server proportional to its speed.

Assume that the speed of *all* the servers in the system is increased by a factor k , keeping the user population constant. The cost will also grow by the same factor, due to the assumption that each server's cost is proportional to its speed. If this investment increases the throughput of the system by the same factor, the system would be intuitively scalable. We will examine if it in fact does increase the throughput by the same factor, and if so, whether the scalability metric does take a value ≥ 1 , matching the intuitive notion of scalability.

We will first prove that the queue lengths at any station is not affected by the abovementioned scaling strategy.

Theorem:

When all the servers in a closed product form queueing network are speeded up by the same factor, keeping the customer population constant, the steady state queue lengths at its servers do not change.

Proof:

This conclusion can be directly derived from *Little's result*. It can also be derived using the MVA solution of the system as discussed below. Assume, $L_i[0] = 0$ for all i .

The iteration steps in the mean value analysis [Allen90] are as follows:

$W_{it}[N]$ = the average total time a customer spends at queue $i = D_i(1 + L_i(N-1))$

$$W[N] = \sum_{i=1}^K W_{it}[N] \quad \text{= the cycle time,}$$

$$\lambda[N] = \frac{N}{W[N]} \quad \text{= the system throughput}$$

$$L_{it}[N] = \lambda[N] \cdot W_{it}[N]$$

$$W[1] = \sum_{i=1}^K D_i, L_i[1] = \frac{D_i}{\sum D_i},$$

$$W_i[2] = D_i \left(1 + \frac{D_i}{\sum D_i}\right), L_i[2] = \frac{2D_i \left(1 + \frac{D_i}{\sum D_i}\right)}{\sum_i D_i \left(1 + \frac{D_i}{\sum D_i}\right)}, \text{ and so on.}$$

It can be observed that $L_i[N]$ is dimensionless for any N , and has the same degree of D_i in the numerator as in the denominator. When all the server speeds are simultaneously scaled by a factor k , the scale factors cancel out.

Hence, the result.

The throughput of this system is given by

$$\lambda(N) = \frac{N}{\sum_{i=1}^K \frac{D_i}{k} (1 + L_i(N-1))}$$

Thus, the throughput scales by the same factor k by using this scaling strategy, and hence the system intuitively appears to be scalable with this scaling strategy. We now check that the proposed scalability metric indeed produces a value ≥ 1 .

$$F = \frac{1}{C \cdot k} \left(\frac{k \cdot N}{\sum_i D_i (1 + L_i(N-1))} \right) \cdot \left(\frac{1}{1 + \frac{1}{\hat{T}} \cdot \frac{1}{k} \left(\sum_i D_i (1 + L_i(N-1)) \right)} \right)$$

Case 1: Time scaling: suppose, if the target response time, $\hat{T}$, is also scaled down by the same scale factor, then all the k 's cancel out, giving the same productivity as that of the starting point, and we get scalability = 1 for any k .

Case 2: If $\hat{T}$ is not changed, we get the productivity limit of $\frac{\lambda}{C}$ as $k \rightarrow \infty$. This limit depends on the throughput and the cost in dollar terms of the reference state, and when divided by

the productivity of the reference state, we get the following value of the scalability metric in the limiting case:

$$\psi(k) = 1 + \frac{1}{T} \cdot \left(\sum_i D_i (1 + L_i (N - 1)) \right)$$

Note that the value of the metric is always greater than 1, indicating that the system is scalable as $k \rightarrow \infty$ irrespective of the initial state, although its exact value depends on the initial state (e.g. it depends on the number of customers). Thus, the system can be scaled infinitely, and we conclude that a case that makes intuitive sense is appropriately evaluated by the metric.

(E.II)

Suppose we apply the scalability metric to a closed queueing network with a *single bottleneck*. Intuitively, a good scaling strategy should involve investing in making the bottleneck device faster, and this strategy should lead to effective and economical scaling of the system, up to a point when some other device becomes the bottleneck. The user population remains constant, as a part of this scaling strategy. Note that this scaling path is different from that used in E(I).

Rather than using the actual values of productivity (and scalability) of the system, we will use the *balanced system bounds* [Lazowska84] on throughput and response time to get a balanced system lower bound on the productivity (also, scalability) of the system. (The throughput and response time bounds for queueing networks are later used in section 5.4 to derive a general rule-of-thumb scalability bound.) We will show that as the speed of (only) the bottleneck is scaled (with an assumption that the cost of all the devices remains proportional to their speed), the lower bound on the productivity is proportionately pushed up, in agreement with the scalability expectation, assuming that the bottleneck does not shift to another device during this process.

Let K = the total number of devices in the closed queueing network.

D_{max} = the service demand of the single bottleneck device,

D_{min} = the service demand of the fastest device,

N = the user population.

C_{D_i} = the cost of the device indexed by i ,

$C_{D_{max}}$ = the cost of the bottleneck device.

$$\frac{1}{D_{min}} \cdot \frac{N}{N+K-1} \geq \lambda(N) \geq \frac{1}{D_{max}} \cdot \frac{N}{N+K-1} ,$$

$$D_{min} \cdot (N+K-1) \leq T(N) \leq D_{max} \cdot (N+K-1)$$

Substituting the lower bound on throughput and the upper bound on delay into the productivity expression,

$$F \geq \left(\frac{\frac{1}{D_{max}} \cdot \frac{N}{N+K-1}}{C_{D_{max}} + \sum_i C_{D_i}} \right) \cdot \frac{1}{1 + (N+K-1) \cdot D_{max} \cdot \frac{1}{\hat{T}}}$$

As the bottleneck device is scaled up k times, keeping the user population constant,

$$\therefore F' \geq \left(\frac{\frac{k}{D_{max}} \cdot \frac{N}{N+K-1}}{C_{D_{max}} \cdot k + \sum_i C_{D_i}} \right) \cdot \frac{1}{1 + (N+K-1) \cdot \frac{D_{max}}{k} \cdot \frac{1}{\hat{T}}}$$

Simplifying,

$$F' \geq \left(\frac{\frac{1}{D_{max}} \cdot \frac{N}{N+K-1}}{\frac{\sum_i C_{D_i}}{k}} \right) \cdot \frac{1}{1 + (N+K-1) \cdot \frac{D_{max}}{k} \cdot \frac{1}{\hat{T}}}$$

Note that for all $k > 1$, the lower bound on F' gets pushed up as k increases. Also, the upper bound on the throughput and the lower bound on the delay does not change due to speeding up the bottleneck. Therefore, these bounds get tighter as k increases.

In the limiting case as $k \rightarrow \infty$, the ratio of the lower bounds on F' and F (which might be an indicator of the scalability value) approaches the following limit, clearly > 1 .

$$\frac{C_{D_{max}} + \sum_i C_{D_i}}{C_{D_{max}}} \cdot \left(1 + (N+K-1) \cdot D_{max} \cdot \frac{1}{\hat{T}} \right)$$

This indicator appears to correspond to the intuition in this system.

Example F: Closed idealized system with replication of servers.

As in example C, this system has a separable queueing model. To scale it up, each server is replicated k times and its load is divided equally among the k servers. Assume that there is no

overhead cost associated with the replicas (a major idealization). Again, first consider a balanced version with all demands D equal; then:

$$N_2 = kN_1; K_2 = kK_1; D_2 = D_1/k; C_2 = kC_1$$

Following example C, but with $K_2 = kK_1$ we obtain almost the same limiting ψ :

$$\lim_{k \rightarrow \infty} \psi(k) = \frac{(N_1 + K_1 - 1)[(N_1 + K_1 - 1)D_1 + \hat{T}]}{N_1[(N_1 + K_1)D_1 + \hat{T}]}$$

and again if $K \ll N_1$, $\psi \rightarrow 1.0$. For an unbalanced system the result is also the same.

It is intuitively clear that this is a perfectly scalable system, which agrees with the result for the measure.

Example G: System with Replication and Overhead.

In example F, suppose that for $k > 1$ the demands D are augmented by coordination overhead, for example to maintain overall system state data. The user population is scaled by a factor k . An efficient coordination mechanism might limit this to a slowly increasing amount of overhead, giving a relationship such as

$$D_2 = (D_1/k) + D_0 \log k$$

Then from equation (4.5),

$$\psi(k) = \frac{k(N_1 + K_1 - 1) \cdot D_1}{(kN_1 + K_1 - 1) \cdot (D_1/k + D_0 \log k)} \cdot \frac{(N_1 + K_1 - 1)D_1 + \hat{T}}{[(kN_1 + K_1 - 1)(D_1/k + D_0 \log k) + \hat{T}]}$$

which decreases slowly towards zero as $k \rightarrow \infty$. In this case, a “large value of k ” implies

$$kD_0 \log k > D_1$$

$$k \log k > D_1/D_0$$

Suppose that D_0 , which describes the relative magnitude of the coordination overhead, is 10% of D_1 this limit would be passed at $k \log k = 10$, which (if logarithms are to base 10) is $k = 10$. Then for $k < 10$ the system would be “somewhat” scalable.

Thus, the effect of overheads in terms of limiting the scalability, has been captured by the scalability metric.

Example H: Effect of changing the target response time on a balanced, closed queueing network:

This example is different from the rest of the examples in this section. While the other examples apply the scalability metric to idealized systems using various scaling paths, this example endorses the sensibility of the metric by judging the implications of changing the QoS requirements. The effect of changing the target response time, $\hat{T}$, on the productivity of a balanced queueing network has been analyzed below. The QoS function for a balanced system, as discussed in example E, is

$$1 + \frac{T}{\hat{T}} = 1 + \frac{(N + K - 1)D}{\hat{T}}$$

Where N = the number of clients,

K = the number of (symmetric) queues, and

D = the service demand at each queue.

Assume that the target response time is increased by an amount $\Delta\hat{T}$, and more clients are allowed in the system such that the QoS function does not change. Let the increase in the number of clients be ΔN .

Theorem:

While keeping the QoS function constant, the number of clients in a balanced queueing network can be increased along with $\hat{T}$ such that the allowable percentage increase in the number of clients is at least as much as the percentage increase in $\hat{T}$.

Proof:

$$1 + \frac{(N + K - 1)D}{\hat{T}} = 1 + \frac{(N + \Delta N + K - 1)D}{\hat{T} + \Delta\hat{T}}$$

Simplifying,

$$\frac{\Delta\hat{T}}{\hat{T}} = \frac{\Delta N}{(N + K - 1)} \leq \frac{\Delta N}{N}$$

Hence, the result.

Theorem:

If the number of clients and $\hat{T}$ are increased so as to keep the QoS constant, the productivity of the balanced queueing network also increases.

Proof:

Let $\Delta \hat{T} > 0$, $\Delta N > 0$. The corresponding new productivity value is given by

$$F' = \frac{N + \left(\frac{\Delta \hat{T}}{\hat{T}} \cdot (N + K - 1) \right)}{C \cdot \left(N + \left(\frac{\Delta \hat{T}}{\hat{T}} \cdot (N + K - 1) \right) + K - 1 \right) D} \cdot \frac{1}{\left(1 + \frac{(N + K - 1) D}{\hat{T}} \right)}$$

Let F denote the productivity before increasing $\hat{T}$.

$$\frac{F'}{F} = \frac{\left(\frac{N + \Delta N}{N} \right)}{\left(\frac{N + \Delta N + K - 1}{N + K - 1} \right)} = \frac{1 + \left(\frac{\Delta N}{N} \right)}{1 + \left(\frac{\Delta N}{N + K - 1} \right)} \geq 1$$

Hence, $F' > F$,

Thus, the productivity of the balanced queueing system can be increased by allowing more target response time, and letting more customers in so as to maintain the QoS unaffected.

The abovementioned cases suggest that the choice of the scalability metric makes intuitive sense, and can be applied to complex practical systems as discussed in the following chapters. They also give confidence in the way the metric captures scalability as a function of the scale factor k , in each case.

Some of the idealized cases discussed above provide an important input for the rule-of-thumb scalability bounds analysis presented in section 5.4. It is based on the throughput and response time bounds for product form queueing networks.

Chapter 5 The Scalability Analysis Framework

The previous chapter discussed the scalability analysis metric and its application to some theoretical systems. In this chapter, some new concepts and terminologies are introduced which, together with the scalability metric, are a part of the framework used to systematically assess the scalability of distributed system designs.

Section 5.1 introduces some of the notions associated with defining the scaling plan for the system and optimization of the scalability enablers. Section 5.2 discusses the tool (PRASAD) that embeds these notions and implements an automatic search mechanism in the system's configuration-space for maximizing the scalability metric. The framework also underlines the importance of providing potential scalability enablers in a distributed system design. Section 5.3 discusses how the remote invocation and replication overheads, which can be important in scalability assessment of real-life systems, have been accounted for in the tool. Section 5.4 presents an algorithm to roughly predict the scalability bounds on a system, which can save computationally expensive runs of the tool, and still give some idea about how well the design could scale.

5.1 Scalability enablers and their optimization

Some important concepts and terminologies associated with the scalability analysis framework were discussed in section 4.2. The notion of optimization of the scalability enablers in order to achieve the best possible scalability will be further developed in this section. After having chosen a scalability metric (as discussed in Chapter 4), the next step is to search for a “good” system configuration (obtained by suitably adjusting the scalability enablers), such that the productivity metric is maximized.

Productivity enhancement usually involves removing bottlenecks in the system, which can be achieved by tuning the scaling enablers. It seems only logical that the scalability measure ψ with respect to any two configurations should be calculated using the “best” possible productivity for each configuration. This enables the relative assessment of the productivity of the system in various configurations by removing the sources of variation by optimizing the scalability enablers.

A scalable design can be achieved by providing many powerful scalability-enabling parameters in the design, and leaving their optimization to be done in the field. For analysis purposes the optimization can be carried out much more easily on a performance model than on a real system. The research presented herein uses fast layered queueing model solutions of different system configurations (as scale factor and the scalability enablers get modified), and analyzes the productivity of each one of the configurations, to make the scalability judgement with respect to a given reference state, which is typically the state corresponding to the starting point of the scaling path.

In short, we identify the variables of the distributed system that are automatically fixed once a scaling strategy is fixed, and optimize the system architecture by “tuning” the rest of the variables which still have some degree of freedom. *The less constraining the definition of the “scaling path”, the more flexibility there is for tuning the system.* For instance, in the example illustrated in section 4.2, if the datasize is not automatically determined by k , it leaves one more scaling enabler for tuning.

The evaluation of many possible scaling paths may be needed to be carried out to provide guidelines for choosing the right scaling strategy for a given system. Note that the problem of detecting a bottleneck in a single performance model has now been transformed into a problem of finding the best scaling path, which leads to the least overall bottlenecking and high performance, over a range of system sizes. This, presumably, would be one of the paths with the maximum number of scalability enablers. If a satisfactory path cannot be found, section 5.1.1 discusses how the scalability problems are diagnosed, pinpointing the hot-spots in the models.

If we have two-dimensional scale factors (e.g. the scale factor k_1 indicates the number of CPUs and the scale factor k_2 indicates the number of replicas of the database, and the two scale independently), the value of the scalability metric is a scalar function of (k_1, k_2) . Presumably, the resource cost of the system would also be a function of (k_1, k_2) . In this case, choosing the best scaling strategy amounts to choosing the highest “scalability surface” while the scaling enablers are optimized at each point of interest on the surface.

The rest of this thesis considers only one-dimensional scale factor. In general, the objective of a scalability study is to see if any of the scaling strategies proposed for a given design yields a “satisfactory” value of the scalability metric, say, above $1 - \varepsilon$ for some ε , over a range of scales. Or conversely, if there are many alternative design choices for a given system, we would like to see which choice among them yields a satisfactory value of the metric over a given scaling

path, by appropriately tuning the design parameters at each scale factor. *If this is not possible, we conclude that there is a need to either provide more scaling enablers, and/or to redesign parts of the system.*

A high-level algorithm for designing scalable distributed systems is diagrammed in figure 5.1.

FIGURE 5.1

The algorithm for designing scalable distributed systems.

5.1.1 Diagnosis of scalability problems

The results obtained by optimizing the configurations (using the tool discussed in the following section) give a deeper understanding of the problem areas in a given design and a given set of scaling paths.

As mentioned earlier, this thesis uses layered queueing models of distributed systems to analyze their performance. Along a given scaling path, the optimized model outputs for each scale factor have to be analyzed to assess the causes of the scalability problems. As discussed earlier in section 3.2, the LQNS output includes the throughputs, response times, and utilizations of all the hardware and software resources in the system. If the algorithm in figure 5.1 fails to obtain a satisfactory configuration and a scaling path, or if there is a need to further improve the system's productivity, the following factors can point to some potential areas that may need re-engineering.

- If a specific CPU tends to become a bottleneck in the optimized models along a given scaling path, it may imply a hardware bottleneck. As a remedy, the CPU could be replaced with another high-speed CPU in order to try to improve the productivity of the system.
- If a software task causes its CPU to saturate consistently along many possible scaling paths, it could mean that the task has excessive execution demand causing a load imbalance. The remedy could be to redistribute the functionality among other tasks, to find a cheaper way to replicate/partition the task, or to invest in a faster dedicated CPU for that task. (This is the case with the example discussed in section 6.1.)
- If the hardware devices are consistently underutilized, but the throughput does not grow significantly in the optimized configurations along the scaling path, a software bottleneck might be the cause. The actual task(s) that cause the bottleneck can be found by analyzing the model output files of the optimized models for all the scale factors of interest. The remedy could be to enhance the set of scalability enablers by introducing multi-threading and/or replication for the detected software bottlenecks.
- If the QoS requirements are too stringent, they have an effect similar to a bottleneck. For example, if the CPUs are not fast enough to meet the user-specified target response time, the optimized throughput cannot grow sufficiently with the scale factor. This is because

increasing the throughput may lead to unacceptably large values of response time with respect to $\hat{T}$. (This is the case with the case study presented in section 6.3.)

- If the given set of scalability enablers is inadequate and causes excessive overheads, this could be detected by analyzing the model output files by analyzing the increased device utilizations due to the overheads due to the scalability enablers. (This is the case with the example discussed in section 6.2). The remedy could be either to provide more efficient scalability enablers, or to redesign the system such that the overheads associated with the scalability enablers can be reduced.

The rule-of-thumb scalability analysis presented in section 5.4 also helps to roughly diagnose the scalability problems at a significantly smaller computational cost.

The following section describes the tool that implements the detailed optimization process.

5.2 The *PR*Agmatic Scalable Architecture Designer (PRASAD) tool

The tool PRASAD has been implemented to support the scalability analysis framework discussed in the preceding sections. Before we start using the tool, any (arbitrary) reference configuration of the system is first expressed as a layered queueing model. (Typically, the reference configuration of the system corresponds to scale factor $k = 1$.) A scaling strategy, which is a function of the scale factor, can then be decided. For example, for some of the real system prototypes tested using this tool (discussed in Chapter 6), the number of CPUs in the system is chosen to be equal to the scale factor, and all the remaining parameters (namely, the allocation, replication, multithreading, and the number of clients in the system) are left for optimization, so that the value of the scalability metric is maximized at a given value of the scale factor on a given scaling path. Making either the number of CPUs or the number of replicas of software objects dependent on the scale factor seems to be a general scaling strategy for many distributed systems, as seen in the literature survey in Chapter 2. One of the case studies presented in the next chapter uses the number of database replicas as the scale factor. Other system-specific scale factors are possible, which can

be arbitrary. As long as their effect could be represented in the layered queueing models, they can be handled in this scalability optimization tool.

The tool is written partly in C++ and partly in TCL, and uses the layered queueing network solver [Franks95] iteratively. In its current version, the tool aims at optimizing the scalability metric given by equation (4.5), for any given distributed system which can be described as a layered queueing model. As mentioned earlier, the following variables are used for optimizing the scalability metric (with reference to any given reference configuration):

- the allocation of the distributed objects to hardware resources.
- the number of replica of some specified objects.
- the number of clients in the system
- the number of threads of the server objects.

FIGURE 5.2 Architecture of the tool

Figure 5.2 shows the major components of the tool PRASAD. The arrowheads indicate the direction of control and/or associated data flow. The C++ engine implements the simulated annealing algorithm explained later in this section. The user-defined global values of the variables (such as the values of simulated annealing parameters, the target response time set by the user),

the random number generator (required by the simulated annealing solver) etc., are also parts of the C++ engine. The C++ engine uses the TCL library to share data dynamically with the TCL scripts. This data includes the multiprogramming levels, replication levels and allocation maps of the tasks, which can be changed by the C++ engine. The LQNDEF TCL scripts provide an easy to use interface for accessing the model input files and modifying them.

The database is organized as a collection of UNIX files, which stores the data required for solving the layered queuing network model of the distributed system, and the output generated by the LQNS. The TCL scripts read the model input files, parse them, perturb them as required by the C++ engine, and store them back in the database. After the solver has produced the output by solving the LQN model, it is stored in the database, which is parsed by the TCL scripts to extract the required output parameters to compute the value of the scalability metric. These are accessed and analyzed by the C++ engine for accepting or rejecting the modifications in the system configuration in order to reach the final optimum productivity (hence, scalability).

The implementation of the simulated annealing algorithm is discussed in section 5.2.1.

The following figure describes the solution strategy:

FIGURE 5.3

The optimization algorithm used by PRASAD

The algorithm randomly perturbs the scalability enablers of the system and creates a new model. Any one of the available enablers is randomly chosen at a time - the relative frequencies of the choice of enablers can be adjusted. For example, in the current implementation, reallocation of objects is chosen more frequently than replication or deletion of clones of tasks such as the database. For the newly generated configuration, it then obtains the maximum value of the scalability

metric by optimizing with respect to the number of reference task threads, or the number of “pure clients” in the system, and then accepting/rejecting this system configuration, based on what is called as *Metropolis criterion* [Laarhoven87][Bilbro91]. The energy function to be minimized, which is used in the Metropolis criterion, is the inverse of productivity. The acceptance criterion chooses all the moves that lead to an increased value of productivity (that has been maximized with respect to the number of clients). Otherwise, the moves are accepted with a finite probability, that depends on the productivity of the state and the “temperature” of the system. In order to obtain the maximum productivity point with respect to the number of clients, the *Golden Section* search [Converse70] is used.

The following subsection contains a brief description of the simulated annealing algorithm.

5.2.1 Simulated annealing

The simulated annealing algorithm is based on the analogy between the simulation of the annealing process for solids and the problem of solving large combinatorial optimization problems [Laarhoven87][Ingber93][Bilbro91]. Annealing denotes a physical process in which a solid in a heat bath is heated up to a point when all the particles of the solid become mobile and randomly arrange themselves in a liquid phase, followed by slowly cooling the temperature of the heat bath. All particles then arrange themselves in a low energy ground state of the corresponding lattice. In order to achieve this, the system has to be sufficiently heated to start with, and the cooling has to be slowly carried out so that no faults are developed in the lattice. At the thermal equilibrium at any given temperature, the probability of being in a state with energy E is given by the *Boltzmann distribution*. We will now discuss the basis for the global optimization approach.

Given a temperature T , the *Metropolis criterion* [Laarhoven87] simulates the process of reaching a thermal equilibrium in the following way. Given the current state of the solid (characterized by the position of all its particles), suppose a small, random perturbation is applied to this system (in terms of movement of one of its particles). This will lead to an energy difference, say, ΔE . If this energy difference between the perturbed state and the previous state is negative, i.e., if the perturbation results in a lower energy state for the solid, the move is concluded to be accept-

able for reaching final equilibrium. Otherwise, if $\Delta E > 0$, the probability of accepting the perturbed state is given by

$$\exp\left(\frac{-\Delta E}{K_B \cdot T}\right) \quad \text{where } K_B = \text{Boltzmann constant.}$$

In case of a combinatorial optimization problem, the temperature is interpreted as a “control parameter” that decides the probabilities of acceptance as mentioned above, and energy function is replaced by some cost function, which in this case is the inverse of the productivity.

The basic simulated annealing algorithm is as follows:

```

begin
  Initialize;
  M= 0;
  repeat
    repeat
      Perturb (config. i --> config. j,  $\Delta E_{ij}$ )
      if  $\Delta E_{ij} \leq 0$  then accept else
        if  $\exp(-\Delta E_{ij}/T) > \text{random}[0,1)$  then accept;
      if (accept == TRUE), then Update (configuration j);
    until equilibrium is approached sufficiently closely;
     $T_{M+1} = f(T_M)$ ;
    M= M+ 1;
  until (stop criterion is true, i.e., the system is “frozen”);
end.
```

FIGURE 5.4

The simulated annealing algorithm

In the implementation of PRASAD, for finding the initial temperature of the system, Johnson’s approach has been used, as discussed in [Laarhoven87]. This approach uses the observed energy difference in a few initial trials to derive the initial temperature. At any given temperature, a sufficiently large number of perturbations is generated (which in turn decide the length of the Markov chain obtained at each temperature as described in [Laarhoven87]). This is

done by making the number of perturbations a function of the number of tasks and CPUs in the system. Thus, the bigger the system, the larger is the number of perturbations generated at any given temperature.

The cooling schedule follows a simple rule: when equilibrium is reached at a given temperature, the temperature is halved and the annealing process is continued. If the fraction of the accepted moves is less than 2% for two successive equilibrium iterations, or if the number of iterations exceeds a predefined limit (which depends on the size of the performance model), the optimization algorithm stops.

During the course of the optimization, the tool also stores the best model found up to the given time in execution, which corresponds to the maximum value of productivity found so far. In Chapter 6, the application of PRASAD to real system prototypes is discussed in detail.

5.3 Handling remote invocation overheads and transaction management overheads

When the layered model representing the system is perturbed, say, by moving a task from one CPU to another, not only the allocation map changes, but possibly the overall execution overheads as well. Suppose a task T that was previously collocated with one of its client tasks, is moved to a remote CPU. Then the communicating entries of its clients as well as those of task T have to incur an extra execution overhead, corresponding to the respective client and the server stub execution on the respective CPUs to which these tasks are allocated. On the other hand, if the task T is moved to a CPU on which any of its clients is already executing (note that this was previously a remote CPU), the stub execution overheads for the respective client and the server entries are no longer into picture. What holds for the client tasks of T also holds for its server tasks, in terms of the modifications of the remote invocation overheads associated with the reallocation.

The tool PRASAD accounts for the client and the server stub overheads by providing a remote invocation delay server on each CPU in the system. When a client entry invokes a remote server entry, it also makes the same number of calls to the remote-delay server executing on the local CPU. Similarly, the server entry makes the same amount of calls to its local remote-delay server. This accounts for the execution overheads on each of the CPUs, which amount to additional response times and additional device utilizations.

When a task T is reallocated to another CPU as a perturbation trial, the calls made by its entries to the (new) remote-delay server are appropriately set, taking into account the modified topology map of all the clients and servers of all of its entries. Similarly, all the clients and servers associated with the task T re-adjust their number of calls to their respective remote-delay servers.

The delays due to the consistency management overheads, which are a function of replication, depend on the algorithm used for replica management. These overheads have to be modeled in the layered queueing model of the system (as in case of the remote invocation overheads). For example, in the connection management system (version -B) discussed in Chapter 6, a “transaction overheads” task is introduced to account for the response time overheads due to the replication algorithm discussed in [Triantafillou95]. The “leader” objects, as discussed in section 2.2.3 make calls to various entries of this task during the operation, prepare, commit and abort phases to account for consistency management overheads of the two-phase commit protocol.

Note that when a task clone is created or deleted, the number of local/remote calls received by each of the replica gets modified, assuming even distribution of calls to all the replica. As a result, the total remote invocation latencies seen by client/server entries of all of the clones, which depend on the number of these calls between the client and server entries, may get modified. Hence, whenever a system is perturbed by adding/deleting a task clone, not only do the consistency management overheads change, but also the remote invocation overheads change.

The tool PRASAD takes this into account whenever a task is replicated/deleted. When a task clone is created/deleted and assigned to a CPU, the number of calls by all the client entries to all the clones is modified (assuming an even distribution of the calls from client entries to all of the server clones). Based on this modified number of calls, all the remote invocation delays corresponding to the client and server stubs are appropriately re-adjusted.

5.4 Rule-of-thumb scalability bounds

It would be useful to be able to roughly predict, *at a small computational cost*, the limits on the scale factors beyond which a given system would not scale. If the scalability bound is lower than what the system designers hope for, the design can be concluded to be non-scalable without spending hours of CPU time in the optimization exercise, which was discussed in the previous section. If the bounds analysis predicts an acceptable scale factor range (hence, an acceptable range of system sizes), the actual optimized system configurations for each scale factor, leading to

the maximum values of the scalability metric, can be obtained through the rigorous runs of the optimizer tool (examples discussed in Chapter 6). The bounds analysis presented herein is based on the throughput and response time bounds analysis [Lazowska84] for product form queueing networks (including the special case of balanced queueing networks), which were among the idealized systems discussed in section 4.3.

This rule-of-thumb bounds analysis assumes the following:

- The logistics of the scale factor and the scaling path, (e.g. whether the scale factor indicates the number of CPUs or the number of database replica, etc.) which are a part of the scaling strategy, are known. This is required in order to see what contributes to the cost and execution overheads as the system is scaled.
- The bounds analysis presented herein can be applied to systems with software critical sections only if one of the following two cases holds - (a) the performance penalties caused by the critical sections in the software are negligible in the overall scenario, or (b) the bottlenecking effect of critical sections can be analyzed separately in terms of throughput and response time bounds enforced by the critical section [Majumdar91] which may be a function of the scale factor.

The following factors can potentially limit the scalability of a system, and should be considered either directly or by making a suitable approximation, while computing a scalability limit:

- The cost of the system resources that may increase (with the scale factor) faster than the throughput and/or the QoS.
- The overheads associated with the scaling path (e.g. consistency management overheads associated with replicating a database) may increase quickly, and limit the scalability.
- Logical resources that cannot be replicated (critical sections).

Note that the resource cost is one kind of penalty that accounts for the overall investment in system resources. The performance-related penalties associated with scaling the system resources can be independent of the cost of the system. For example, certain software resources may not be a part of the scale factor (i.e. they do not necessarily decide the system size, and the cost), but may significantly affect the utilization of hardware resources, if replicated, due to overheads such as consistency management, group communication, etc. Adding hardware resources does not require replicating software resources and vice versa. *What matters for the scalability*

bounds are only the mandatory penalties of cost and/or performance overheads that are a function of the scale factor (or the scaling path), and the remaining constraints are relaxed or ignored.

Note that since the scale factor and the scaling path can be arbitrary, performance penalties due to associated overheads can even form an integral part of the scaling path. For example,

- A scale factor that scales the number of CPUs and replicas of a database by the same factor (replication overheads are directly connected to the scale factor here, and are known),
- A scaling path that requires at least R replicas of the database in the system, for any scale factor that indicates the number of CPUs. (Overheads corresponding to at least R database replica are embedded in the bounds analysis in this case).

Algorithm for finding the rule-of-thumb scalability bound:

Step 1. The layered queueing model corresponding to scale factor 1, with the mandatory replication and remote-invocation overheads corresponding to scale factor =1 expressed, is optimized for its productivity with respect to the number of clients.

The effect of overheads is expressed by appropriately invoking the entries of the task(s) representing these overheads. For example, corresponding to the database transactions, the corresponding overhead task entries that account for the *operation*, *prepare*, *commit*, and *abort* phase overheads are invoked by the database entries. This is a modeling artifact used during the bounds calculation as well as during the optimization.

Step 2. The execution demands for all the tasks (which are units of distribution) are obtained by dividing the total utilization corresponding to each task by the reference throughput.

Note that the total demand and average demand calculation (required by the balanced system bounds for the queueing networks) includes the net demand produced due to various overheads, later expressed as a function of the scale factor. (In the simplest form, they can be directly proportional to k .)

Step 3. For the throughput, we consider the *minimum* among (1) the balanced system throughput bound for queueing networks and (2) the asymptotic throughput bound for queueing networks.

This is because, while the balanced system throughput bound is tighter before the throughput saturates, the asymptotic throughput bound (which is the reciprocal of the highest demand)

becomes tighter as the system approaches saturation. For the response time, we use the balanced system lower bound on response time.

For example, total demand, D , can be expressed as $A_0 + A_1 * g_1(k)$, where

- the constant A_0 depends on the total (fixed) demands in the model and $g_1(k)$ indicates how the overheads change with k ,
- Average demand = $D_{avg}(k) = D / g_2(k)$, where $g_2(k)$ indicates how the number of processors in the system changes with k ,
- Response time = $R(k) = D + (N-1) * D_{avg} / (1 + (Z/D))$ which is the corresponding balanced system lower bound,
- Surrogate and other delays including sleep time = $Z(k) = Z_0 + g_3(k)$;
- Cycle time = $T(k) = R(k) + Z(k)$;
- Throughput = $\min \{k * N / T(k), 1 / D_{max}\}$
- Cost = $g_4(k)$, in dollar terms.

Step 4. Substituting in the scalability metric, we get an analytical expression for the value of the scalability bound as an analytic function of k . *Note that this is actually a productivity bound, for each value of scale factor k , normalized with respect to the reference productivity.* This is plotted against k to obtain the scalability bound. The scalability bound is obtained when the (usually dropping) curve hits the user-defined acceptability limit for scalability, which is $1 - \epsilon$. The corresponding scale factor gives the scalability upper bound.

As shall be seen in Chapter 6, the simple scale factor bounds generated by this analysis agree with the solutions obtained by the optimizer. For example, for the connection management system prototype (A) with hypothetical parameters, the predicted scale factor limit of 7 matched the limit observed after optimization. (The exact match can be explained by the fact that both the optimizer and the bounds analysis did not consider any overheads.)

In the case of the connection management system (B), based on an industrial prototype, the scale factor and the scaling path allowed accurate incorporation of the associated overheads into the bounds analysis, and the predicted scalability bound of 8 compares satisfactorily with the scalability limit found by the optimizer (at $k = 5$).

However, in the case of the call processing system, (which is also based on a real prototype), the bound was observed to be rather loose. This was because the scaling path and the scale

factor alone cannot account for the overheads associated with tuning the scalability enablers, and so remote invocation and replication overheads cannot be easily incorporated in the bounds analysis. A maximum number of scalability enablers are used for optimization of the productivity in this case, and most of the enablers are associated with overheads. However, *since the optimum value of these enablers is not known prior to the optimization, the overheads cannot be included in the bounds analysis*. Therefore the bounds analysis, by ignoring the replication and remote invocation overheads, obtained a limit of scale factor = 11, whereas it was actually found to be 3.

Thus, the bounds analysis can be fairly accurate if all the overheads can be captured using the scale factor and the scaling path, but it cannot account for the overheads due to the scalability enablers (which are not the functions of the scale factor), because the enablers in fact have to be optimized “in field”.

Moreover, the bounds analysis is based on balanced queueing networks that cannot model critical sections in software and the bottlenecking effect of software resources. The optimization framework uses a powerful modeling paradigm, namely the layered queueing models, and accounts for these effects. Nevertheless, the bounds analysis may be used to get a rough scalability idea at a low computational cost.

The algorithm discussed in this section can also lead to an *early insight* into the scalability problems associated with the given design and the given scaling strategy, by detecting any excessive overhead and cost components that may adversely affect the value of the productivity bound. This is explained below.

As an example, assume a simple form of the demand and cost functions as follows:

Let $D(k) = g_1(k)$,

$D_{avg}(k) = D / g_2(k) = g_1(k) / g_2(k)$,

Surrogate and other delays, including sleep time, = Z_0 ,

$$\begin{aligned} R(k) &= D(k) + (N-1) * D_{avg} / (1 + (Z/D(k))) \\ &= g_1(k) + (N-1) \left(\frac{g_1(k)}{g_2(k)} \cdot \frac{g_1(k)}{(Z_0 + g_1(k))} \right), \end{aligned}$$

$T(k) = R(k) + Z_0$,

Cost = $C(k)$.

The productivity bound can then be expressed as:

$$F_{bnd}(k) = \frac{\min \left\{ \frac{k \cdot N}{Z_0 + g_1(k) + (N-1) \left(\frac{g_1(k)}{g_2(k)} \cdot \frac{g_1(k)}{(Z_0 + g_1(k))} \right)}, \frac{1}{D_{max}} \right\}}{C(k) \cdot \left(1 + \frac{1}{\bar{T}} \cdot \left(Z_0 + g_1(k) + (N-1) \left(\frac{g_1(k)}{g_2(k)} \cdot \frac{g_1(k)}{(Z_0 + g_1(k))} \right) \right) \right)} \quad \dots \text{Eqn(5.1)}$$

It can be observed that the cost ($=C(k)$), and the *total demand* ($=g_1(k)$) always have a significant impact on the nature of the productivity function. If the cost and/or the net execution demand increase rapidly with the scale factor, the productivity bound declines rapidly with k , aggravating the scalability concerns for the given system.

(1) Consider the case when the system throughput is below saturation. Therefore, the balanced system bound will be used in the productivity metric, which is the left term in the numerator in equation 5.1.

The degree of the numerator is $O(k)$. The degree of the denominator is decided by $T(k) * T(k) * c(k)$. It can be observed that the degree of $R(k)$ and $T(k)$ is at least as much as the demand function, $g_1(k)$. Note that if Z_0 constitutes a major percentage of the response time, then the overall response time does not deteriorate rapidly, when k increases. (If the degree of all the abovementioned polynomials of k is ≥ 1 , then the productivity bound declines rapidly as $k \rightarrow \infty$, and it declines at least as rapidly as k^2 .)

Also, more the degree of $g_2(k)$ (with respect to $g_1(k)$), the less steeper is the response time bound. Therefore, from the viewpoint of the productivity bound, it is advisable to maximize the number of processors that meet the given cost requirement, for a given execution demand. In other words, this bounds analysis suggests that *a scaling path that favors adding a large number of cheaper processors is more likely to be preferred as opposed to the one that favors a small number of expensive processors, provided adding a large number of processors does not cause the communication demand to grow excessively (which is accounted for by $g_1(k)$)*. If we assume that the response time requirements are met in either case, it agrees with intuition, since *this scaling path could lead to increased concurrency of execution*.

In most practical cases, the communication overheads grow as more processors are added. In such cases, the productivity bound expression would capture the trade-off between increased execution demand and increased concurrency.

Thus, if a scaling path favors increasing the number of processors anticipating the increased demand, (such that the overheads do not cause the overall demand to grow as steeply as the number of processors do) it is likely to lead to acceptable productivity bound values over a large range of k , perhaps leading to better overall scalability.

If the total demand does not grow with the scale factor at all, then cost is the only entity that penalizes this rule-of-thumb productivity bound.

(2) When the throughput approaches saturation, the productivity bound declines more steeply with k . The throughput bound gets fixed by the bottleneck device, and the numerator of equation 5.1 is a constant, equal to right hand term inside the curly brackets.

This suggests that a good scaling path should ensure that all the system resources stay below saturation at every scale factor of interest, in order that the productivity bound does not drop to an unacceptable level. In other words, a good scaling path should favor load balancing over a large range of scale factors, which agrees with intuition.

If a scaling path leads to load imbalance over a large range of k , it multiplies the adverse effect of increasing the execution demand with k , based on the productivity bound in equation 5.1. The argument about the positive effect of $g_2(k)$ - i.e. the positive effect of adding a large number of cheaper processing resources, more rapidly than the increase in the processing demand as k increases, still holds.

The optimization process simultaneously achieves the goals of load balancing, minimizing the overheads that grow with the scale factor, and maximizing concurrency. But the productivity bound gives an early insight into obvious scalability problems, such as in a case where the execution overheads and/or the system cost grows faster than the throughput (upper) bound does.

This is illustrated in the rule-of-thumb bounds analysis in Chapter 6, for the three case studies.

Chapter 6 Scalability Analysis of Prototypes of Distributed Systems

In Chapter 4, the scalability metric was applied to various theoretical systems. This chapter demonstrates the application of the overall scalability analysis framework to three systems which are based on real industrial prototypes.

Section 6.1 discusses a connection management system architecture (A), and presents its scalability analysis (using hypothetical model parameters). It presents a comparative evaluation of three different scaling strategies, and illustrates how a distributed system architecture with more “tunable” parameters is likely to be more scalable than the others. This system is an abstraction of the more complex connection management architecture (B) described below.

Section 6.2 discusses another connection management system (B), with realistic (measured) parameters including overheads associated with the replication of database, and using a different scaling strategy than (A). The scaling strategy is focussed on replicating the database. Although the database copies are relatively inexpensive with respect to the total system cost, it is observed that the excessive transaction management overheads due to a large number of write and read transactions suggest redesign of the database architecture than invest in replication, which yields only a marginal gain. Scalability analysis and optimization framework is applied by optimizing the number of customers allowed in the system.

Section 6.3 applies the scalability analysis framework to a prototype model of an ATM based call-processing system. The experiment focuses on studying the scalability of the architecture with respect to the number of processors. The parameters which are optimized (i.e, the scalability enablers) are replication levels of the database, the location server, and the number of users. Unlike the connection management system (B) mentioned above, most of the database transactions are read-only in this case, and the consistency management overheads do not grow excessively with replication. Not only the replication overheads, but also the remote invocation overheads are accounted for, using certain assumptions.

This chapter demonstrates how the analysis framework presented in previous chapters can be applied to realistic distributed systems with considerably complex architectures and overheads,

in order to obtain useful design and deployment guidelines. It also shows that the framework proposed in this thesis is general enough to be applied to various kinds of systems, and can help investigate the design limitations and help guide the improvements in the design, to make it more scalable.

6.1 The connection management system- version A

A case study was done on a prototype system for connection management over an ATM network, which was also used as an example in [Jogalekar97]. Figure 6.1 shows the set of software processes in the design. (This is a simplified model of the connection management system. A more complex version based on real parameters is analyzed in section 6.2). The software tasks implement two phases of connection management:

- allocation and deallocation of bandwidth and end-point identifiers,
- setting up and tearing down of connections using previously allocated resources.

The processes communicate by remote procedure calls, using communications middleware and subsystems which have been left out of the model in order to simplify it for presentation purposes.

FIGURE 6.1

Layered queueing model of the example connection management system (version A).

6.1.1 Performance model of the connection management system (version A)

The User process in Fig. 6.1 calls the Provisioning Agent (PA) for reserving bandwidth for multipoint session(s), setting up and tearing down the connections between two or more of these points, and freeing the reserved resources. PA provides a secure interface to User. It carries out the authentication and authorization checks on client requests, and then forwards them to the network resource manager NRM, and the connection manager CM.

The Network Resource Manager (NRM) invokes the routing manager RM, and chooses the network path to provide the required bandwidth and quality of service. If a feasible path is available, NRM recursively invokes the subnetwork manager SM for setting up the connections in the underlying subnetwork (which could be a single switch in the simplest case). The NRM also updates the database DB accordingly. During the freeing of the network bandwidth, NRM follows an exactly reverse path as above.

The Connection Manager (CM) establishes and releases virtual channels between end user sites. It invokes the routing manager RM before setting up a connection, and when it finds a suitable path, it invokes a transaction that recursively calls the subnetwork connection manager SM. RM finds a path for the connection, using various criteria including QoS. The Subnetwork Manager (SM) establishes and releases connections through the ATM switches between specified endpoints. Any change in the connection map is synchronized with the database state by the object that calls the SM.

The Database (DB) maintains the global state of all the network resources, their allocation map, connections, and the information required for the routing function. With any change in the system state, the database is updated atomically. To scale the system up, the database could be replicated and distributed. In this case, we assume that database access conflicts, race conditions etc. are not a major performance concern under average operating conditions, although with more effort, these could be modeled.

The general scalability questions posed by this system include:

- Can adding a fast machine to the network buy a proportional degree of performance of the above system (after suitably reallocating the objects)?
- Can the extra investment in maintaining and administering multiple copies of the connection manager or the database be justified against the enhancement in the QoS experienced by the clients?
- Among several alternatives, what is the best scaling path for evolving a given distributed system to support larger scales of operation in future?
- In short, does the system produce a satisfactory return on any extra investment in augmenting any of the modules in the client-server hierarchy and how can the system be “tuned” to get the maximum possible return on investment?

6.1.2 Scalability analysis framework applied to the connection management system (version A)

In the following analysis, we attempt to address some of the scalability questions mentioned in 6.1.1 by using different scaling paths and scalability enablers. Let the number of CPUs in the system decide the “size” of the system, by definition. Thus, a scale factor of k for this system would mean that there are k CPUs in the given configuration. Collocation/distribution of

objects, replication of the database, the number of clients allowed in the system and multithreading of all the tasks have been used as the scalability enablers. For this system, “neutral” scalability would be satisfactory, say, even with a modest decline in ψ below unity.

The scaling path for this system is based on adding one processor at a time, starting from a base system with four processors. The path is only carried to seven processors, because at that point, the scalability limit of the system is reached, and our goal of demonstrating the use of the analysis framework is accomplished. The scalability enablers which were optimized at each scale were the number of copies of the database DB, the allocation of tasks to processors, the number of threads used in multi-threaded servers, and the number of users. Including the number of users as a scalability enabler allows us to find the “best” (most productive in the sense of maximum F) number of users for each size of system. Another approach would be to fix the number of users in the path and vary the number of processors. The scaling path is referred to below as path number 3 allows the maximum number of enablers, and it was compared to two relatively rigid scaling paths (1 and 2) that used arbitrary decisions on the database and the allocation.

As mentioned in section 3.2, the performance evaluation was made using an analytic performance model, which was a layered queueing model implemented in the solver LQNS [Franks95]. The workload parameters of the base case are given in Tables 6.1 and 6.2. (These are hypothetical parameter values, used only to make a case, whereas sections 6.2 and 6.3 use actual parameter values measured from the prototypes). However these parameter values show the principle of operation of the analysis just as well as the measured values would. Parameters are associated with entries, as labelled on Fig. 6.1, which are identified with the services offered by each process.

The scalability metric given by equation (4.5) was used, with the target value of the mean response time, $\hat{T} = 150$ sec. The cost C was taken to consist of one unit of cost per processor and 0.1 units per copy of the database (the licensing cost).

**Table 6.1: Mean Execution Times of the Entries
(Remote/Local Procedures)**

Entry (procedure entry point) offered by a task	No. of threads of the task	Execution time (sec)
User (clients)	10	Think time- 60
PA - E1	10	0.4
PA - E2	10	0.3
NRM - E1	1	1.0
CM - E1	5	1.0
RM - E1	10	0.9
RM - E2	10	0.8
DB - E1	4	0.5
DB - E2	4	0.5
DB - E3	4	0.5
SM - E1	5	2.0
SM - E2	5	1.5

**Table 6.2: Relative Call Frequencies Between
Entries**

Remote / local procedure call	Relative Frequency
User --> PA-E1	1.0
User --> PA-E2	10.0
PA-E1 --> NRM-E1	1.0
PA-E2 --> CM-E1	1.0
NRM-E1 --> RM-E1	1.0
NRM-E1 --> DB-E2	4.0
NRM-E1 --> SM-E1	1.0
CM-E1 --> RM-E2	1.0
CM-E1 --> DB-E3	3.0
CM-E1 --> SM-E2	1.0
RM-E1 --> DB-E1	2.0
RM-E2 --> DB-E1	2.0

Table 6.3: Three Scaleup Strategies

Scaling Path	Scale Factor Range (the number of CPUs)	Scaling Enablers
(1) As the scale factor is increased by 1, move the most utilized task to the newly added CPU, and create 1 new copy of the database (upper bound 3) and allocate it to the CPU that was least utilized before increasing the scale factor.	4 -7	-Number of users
(2) As the scale factor increases by 1, add a new copy of the database to the system (upper bound on the copies of the database = 3). Arrange all the tasks in decreasing order of demands, and allocate to the available CPUs in a round-robin fashion.	4-7	-Number of users
(3) Naive Simulated Annealing (Not guided by heuristics) - used to optimize the productivity, F .	4-7	-Number of users, -Multithreading levels of all the software tasks, -Allocation of tasks to processors, -Number of copies of the database (upper bound 3).

Evaluating the three scaling strategies

Each scaling strategy scales up by adding a processor, and then making adjustments. Strategy 1 always adds a new replicated database when it adds a processor, and re-allocates a single task using a load-balancing rule; it then determines the optimum number of users. Strategy 2 also adds a database replica, and then it re-allocates all the tasks evenly over the processors (without regard to load balancing, however). It then optimizes the number of users. Strategy 3 attempts to do an optimal global re-allocation of tasks on the enlarged set of processors, including an option

to add a database replica or not, and also optimizes some other factors before optimizing the users. Because it has more scaling enablers, the last strategy is expected to give the best scalability. However, the optimal allocation problem is difficult and the algorithm used (simulated annealing) only represents a “good try” towards attaining a global optimum, using a standard method. It is interesting to investigate the relative capability of simple, more fixed strategies (1 & 2) as against the (more elaborate) third strategy.

The three strategies are summarized in Table 6.3, and are described in more detail as follows:

- *Scaling path (1)*: For the first strategy, the allocation of the software tasks to the CPUs is hard-coded in the scaling path, for a given initial system configuration, as summarized in Table 6.3, row 1. We optimize the number of users, the only available scaling enabler for this path, such that the scalability metric is maximized. This is done by iteratively searching through the user-population space, based on the golden section search algorithm over a range of users (1-50). The scalability metric is computed by solving the layered queuing model of the system, for any given configuration. This optimization is done for all the values of the scale factor of interest. Above this (optimized) number of users, the QoS cannot be guaranteed, and below this, the operation of the system is uneconomical.
- *Scaling path (2)*: The scaling strategy has been described in Table 6.3, row 2. The tasks are arranged in decreasing order of demands, and allocated in a round-robin fashion. Thus, the only available scaling enabler which can be tuned is the number of users as in case of scaling path (1). As the scale factor changes, the layered models corresponding to this path change according to the scaling path (2), but the optimization of the number of users is carried out in the same way as in case of path (1).
- *Scaling path (3)*: The third strategy is much more flexible and is capable of handling a wide range of scaling enablers, as shown in Table 6.3, row 3. It employs simulated annealing to adjust the enabler values and control the solver LQNS.

In this strategy, the system model is first perturbed by randomly choosing and modifying any one of the scaling enablers other than the number of users. (e.g. reallocation of a randomly chosen object to a randomly chosen processor). The modified system model is then optimized with respect to the number of users, as in case of scaling paths (1) and (2), using a golden-section based search. If the maximum value of the metric is more than the maximum obtained in the pre-

vious iteration, the new system configuration is accepted. Else, it is accepted with some probability as explained in section 5.2.1. This is done until a convergence criterion is met (e.g. the percentage of accepted moves being less than a lower limit).

Although the annealing process could be guided by heuristics for better and faster results (e.g. reallocation of the task with maximum queue length, instead of choosing a task randomly, etc.), even the simpler version of simulated annealing used here was found to be powerful in terms of handling a wide range of scaling enablers (one enabler modified at a time).

6.1.3 Rule-of-thumb scalability bound

This section presents a rough estimate of the scalability limit of this system architecture based on the algorithm presented in section 5.4, and using the layered queueing model parameters of this system, listed in the Tables 6.1 and 6.2.

- The scale factor in the case of this model, indicates the number of CPUs in the system. All the CPUs are assumed to be of the same speed, for simplicity (although the analysis could handle different speeds). The scale factor is an important parameter in deciding the cost of the system.
- The database is the only task that is allowed to be replicated, and the replication overheads are negligible. The number of copies of the database is not a function of the scale factor, although database replication is used as a scalability enabler. The minimum number of database copies in the system is 1.
- We make a cost assumption that the CPUs cost one unit each, and the database replica cost 0.1 unit each.

This scalability analysis mainly focuses on analyzing whether or not extra CPUs buy proportional performance and QoS. In this scalability experiment, the remote invocation and transaction management overheads are ignored (unlike sections 6.2 and 6.3).

We now construct a model corresponding to a scale factor 1, extract the workload parameters from its output, and use the balanced queueing network bounds to compute the scalability bound, expressed as a function of the scale factor. This is explained below.

Step 1. The model corresponding to scale factor 1 is optimized with respect to the number of clients. In the given case, the model has a single CPU on which all the tasks (other than the

tasks that represent delays and the reference task) execute. The optimized throughput (= 0.006715 quality responses per second) is obtained, with productivity = 0.0030633 and optimized number of clients = 1.

Step 2. The execution demands for all the tasks (which are units of distribution, as we increase the scale factor) are obtained by dividing the total utilization corresponding to each task by the reference throughput. Note that if there are tasks that account for the overheads (none here), their demands, as a function of the scale factor, are included in the *total demand* and *average demand* calculation, as required by the balanced system bounds for the queueing networks.

Step 3. We assume that the throughput of the equivalent balanced system can scale at most by a factor k , due to replication of its resources. We use the minimum of the balanced system throughput bound and the asymptotic throughput bound in the scalability bound calculation. For the response time, we use the balanced system lower bound on response time.

In this case, total demand, $D = 68.3$ sec per response,

average demand = $D_{avg}(k) = 68.3 / k$ sec per response,

$D_{max} = 28$ sec.,

response time = $D + (N-1) * D_{avg} / (1 + (Z/D))$ which is the corresponding balanced system lower bound,

cost = $C(k) = k$,

$Z_0 = 60$ sec.

As explained in section 5.4, we get the following expression for the productivity bound. Note that because the overheads donot grow with the scale factor in this case, the only factor causing the productivity to decline is the cost function, which is proportional to k . Intuitively, the productivity bound expression suggests that the system could scale upto a point when the task with the maximum demand saturates its CPU, beyond which point the productivity bound declines rapidly.

$$F_{bnd}(k) = \frac{\min \left\{ \frac{k \cdot N}{Z_0 + D + (N-1) \left(\frac{D}{k} \cdot \frac{D}{(Z_0 + D)} \right)}, \frac{1}{D_{max}} \right\}}{k \cdot \left(1 + \frac{1}{\hat{T}} \cdot \left(Z_0 + D + (N-1) \left(\frac{D}{k} \cdot \frac{D}{(Z_0 + D)} \right) \right) \right)} \quad \dots \text{Eqn(6.1)}$$

Substituting in the definition of the scalability metric, the value of the scalability bound as a function of k has been plotted below for the interval of interest.

FIGURE 6.2 The rule-of-thumb scalability bound for the connection management system (A)

The system configuration described in Tables 6.1 and 6.2, after optimizing with respect to the number of users, is used as an (arbitrarily chosen) reference configuration. The reference configuration has a response time of 124.7 seconds and a throughput of 0.02707 quality responses per second, corresponds leading to the productivity = 0.003605. The optimized number of users is 5. The same reference configuration has been used for the optimization exercise presented in the next section.

The bound is seen to (almost) exactly predicts the scalability limit of path 3, as observed in figure 6.3. The optimally scaled configuration is the output produced by the optimizer tool PRASAD (for each scale factor of interest), as discussed in the following section.

6.1.4 Results and discussion

The results for the scale factors from 4 to 7 (that is, for from 4 to 7 processors) are shown in figure 6.3. As mentioned in the previous section, the reference configuration corresponds to the parameters in Tables 6.1 and 6.2 and has a productivity of 0.0036 (quality) responses per second per unit cost, which was obtained after optimizing the metric with respect to the number of users. *Note that the choice of the reference only normalizes the productivities, but does not change the scaling behavior of the system.*

As observed from figure 6.3, strategy 2 is the worst, and strategy 3 is the best as expected. Strategy 3 achieves a value of the measure greater than unity (and thus perfect scalability) for $k = 4, 5, 6$ and 7. Beyond this range, the way the curve is dropping (which in general suggests that either there is a single bottleneck (the connection manager, CM, in this case) or the QoS function acts like a bottleneck, limiting the throughput, which will limit further scaleup of the system.)

TABLE 6.4. Performance figures at maximum values of the scalability metric

	Path 1			Path 2			Path 3		
	N	R(sec)	λ (sec ⁻¹)	N	R(sec)	λ (sec ⁻¹)	N	R(sec)	λ (sec ⁻¹)
$k = 4$	5 (ref.)	124.7	0.02707	5 (ref.)	124.7	0.02707	10	191.11	0.03982
$k = 5$	7	130.3	0.03512	5	153.6	0.02341	12	171.8	0.05177
$k = 6$	7	142.7	0.03453	5	150.4	0.02366	12	165.1	0.05330
$k = 7$	7	142.7	0.03453	7	143.6	0.03437	12	159.9	0.05456

FIGURE 6.3

Scalability analysis results for the connection management system (version A)

The rule-of thumb bound is based on queueing networks and cannot distinguish between the first phase and the second phase workload (as the layered models can do). In this case, the slight underestimate of the productivity (by 3%) at scale factor = 7 is attributed to the imprecision in the layered queueing model solution, which is an approximation.

The (optimized) number of users in each case indicates that number of active users forming connections, (with only 60 sec. between ending a cycle of connection request and starting the next), which leads to the value of productivity.

In the case of scaling path 1, due to the arbitrary and not very intelligent strategy for migrating objects, some CPUs did not get utilized at all. At $k = 6$, one CPU is not used and at $k = 7$, two CPUs are not used. The cost of the unused CPUs reduces the value of the metric.

For scaling path 2, the round-robin allocation only takes into account the execution demands and not the communication frequency between tasks. Clearly, the scaling path poorly

handles the bottlenecks as new resources are added, and performs poorly as expected. At $k = 7$, this strategy happens to do better load balancing by moving the CM task, which is the bottleneck, to a newly added CPU.

The advantages of using additional scalability enablers and an automated optimization are clear in the success of path 3. The optimization finds a significantly better allocation than in the reference case, and is able to greatly increase the throughput and the number of users. The increased response time is still near the target $\hat{T} = 150$ seconds. If this range of scales is adequate, then we do not need to re-engineer the system, but can deploy these versions over 4 to 7 processors.

The scaled up systems in path 3 show a reasonable trade-off between the additional users and the additional response time.

This experiment was limited to a small range of scales, but the method extends to large systems in a straightforward way. Scaling path 3 is still rather limited in its potential to provide large scale service. More tasks need to be replicated to avoid becoming bottlenecks. The number of replicas and their allocation should be freely adjusted, as we have done through the database server.

Other solutions for scalability enhancement could be to add faster processors, or to redesign the clients and servers to balance the execution and communication costs across the system. To analyze the effect of such changes on the scaling path, the analysis would have to be revisited.

Note that the scalability questions listed in section 6.1.1 can be answered using the scalability analysis framework presented in this thesis. This framework applies to general scaling paths. For example, the effect of scaling the CPU speeds in a similar model has been analyzed in [Jogalekar96].

As observed earlier, the simulated annealing approach presented is robust, flexible, and can be extended to industrial systems of high complexity. This has been illustrated in the following examples. The drawback of the simulated annealing approach is that significant amount of computational effort is involved in the solution. In general, at least a few hundred layered queueing models need to be solved find to find the approximate optimum.

6.2 The connection management system- version B

This section analyzes the architecture of a connection management system based on a real prototype, with real parameters, which is conceptually somewhat similar to the system discussed

in section 6.1, but is different in many ways. The analysis presented in section 6.1 was (mainly) meant to demonstrate the effect of different scaling strategies, and the optimization framework, and ignored the overheads such as remote invocation overheads and transaction management overheads. The model presented in this section contains a different set of software tasks than the ones in (A), obtained by implementing the functionality of the clients and the servers in section 6.1 in a different way. It models the interactions of various objects with the database in more detail, and considers the transaction management overheads not addressed in section 6.1. Also, the scaling strategy used in this analysis focuses on replicating the databases.

Section 6.2.1 presents the architecture of this system (which directly maps onto its layered queueing model), along with brief description of its various tasks. Section 6.2.2 discusses the scaling path, available scalability enablers, the cost assumptions for evaluating the scalability metric and handling of the overheads in the performance model. Based on the description of the system architecture and its scaling path, section 6.2.3 makes a rough prediction of the scalability bound of this system. Section 6.2.4 presents the results obtained by actual runs of the optimizer tool. It can be observed how replication overheads affect scalability even though the replicated resources are relatively inexpensive (which cannot be predicted intuitively).

6.2.1 Architecture of the connection management system (version B)

FIGURE 6.4

The Connection Management System- version B

Fig. 6.4 shows the major components in the connection management system- (B). The architecture of this system prototype was heavily influenced by standards such as G.805 [ITU95]. This connection management system was designed so as to be able to

- set up a virtual private network consisting of the user-specified end-points, allocating the network resources in such a manner as to meet the QoS requirement.
- manage a variety of heterogeneous switching equipment, while setting up end-to-end connections.
- use the allocated resources of the virtual private network and let the user set-up/ tear down connections arbitrarily, among any of the sites. A typical application that uses the virtual private network to set up high-bandwidth connections across any of the given sites is video-conferencing.

The prototype was implemented using a network of workstations running UNIX and DCE middleware (discussed in Chapter 3). The backbone network that is managed using this prototype is based on SONET OC-12 (622 Mbit/s) optical fiber ring. The network elements consist of proprietary switching equipment (edge switches, backbone switches etc.), on which cross-connections can be made/ released as required.

The software client and server tasks (or “building blocks”) in this connection management system are the actual implementations of some of the conceptual objects discussed in section 6.1. The tasks can be roughly classified into three logical layers:

- The topology layer, that deals with the connection topology of the virtual private network (VPN), connecting all the user-specified endpoints (e.g., the User-Network Interface identifiers -UNI's - in the case of an ATM network). In the case analyzed here, the network was supposed to have a star topology with (a maximum of) three client site edge switches, and one central backbone switch (or subnetwork). Point-to-point connections between two of these three sites were used for the demonstration and instrumentation of this system.

Once a virtual private network is established the objects in the topology layer can directly communicate with the lowest layer (called SONET here), in order to set up virtual channels over this VPN.

- The VP layer, that deals with connecting all the sites in a virtual private network with a virtual path. This corresponds to provisioning the network resources to meet user-specified bandwidth and QoS, to support future connections.
- The SONET layer that supports a virtual path by setting up appropriate connections on the SONET ring.

Following is a brief description of the tasks in this system:

- Client: This task represents the users of the connection management system, that set up (or dismantle) the virtual private network and set up (or dismantle) connections on an existing virtual private network. The clients could be the software tasks that manage higher level applications, e.g. a video conferencing system that uses the given connection management system.

The clients interact with the topology layer to set up the virtual private network as well as the connections on it (VC's or the virtual channels). The frequency of setting up/releasing a VPN, which is like a leased line, is much lower than that of setting up/releasing temporary connections by actually using the resources of a virtual private network. (In the given layered queueing model of the system, derived from the above diagram, the ratio of the frequencies of setting up/releasing a VPN and setting up/releasing connections on it is assumed to be 1:50).

- Topo_setup and Topo_delete: these tasks belong to the topology layer discussed above, the former supports setting up VPN as well as end-to-end connections between any two endpoints in the VPN. (A topology layer "trail" is an end-to-end connection seen by the client). The Topo_delete task releases the resources and the connections. The topology layer tasks use the services of the layers below, as well as other tasks such as the Database. The routing functions required by this layer and the layers below are built into the tasks (in their respective "setup" entries), and many of the database interactions are due to the simple routing algorithm implemented in the prototype.

The entries of the Topo_setup and Topo_delete tasks that are marked "VPN" deal with the network resource allocation for the required virtual private network, whereas the entries marked "Trail" deal with setting up/releasing connections on the VPN.

- VP: This task supports the topology layer tasks. Using the services provided by the SONET layer as well as other tasks such as the Database, it sets up/ deletes virtual paths (VPs) that make up the VPN with a required topology.
- SONET: This task manages the fibre-level port-to-port routing and connections required to support the setting up of the VP layer trails, which in turn help set up the VPN.
- Subnet_connect: This task directly controls the network elements that set up the SONET ring.
- Database: From the performance viewpoint, this is the most important task in the system. The database management system is proprietary. The database implemented by the prototype stores a few hundred objects corresponding to various functional layers in the system (many of them required by the standards such as G.805). The database, which is accessed heavily by almost all the tasks in the system, clearly is a potential hot spot in the system. By instrumenting the prototype using a software tool (QuantifyTM), it was verified that the database indeed formed a critical segment in the overall execution of the VPN setup/release as well as connection setup/release.

The database appears to be a clear choice for focussing the efforts for design improvement to achieve high performance and scalability, as well as for high availability and reliability.

The scalability analysis approach as described below focuses on analyzing and optimizing the system's scalability with respect to database replication, while taking in to account the overheads associated with database replication (which turn out to be excessive in this case).

6.2.2 Scalability analysis framework applied to the connection management system (version B)

The layered queueing model parameters for this system were obtained using the instrumentation tool QuantifyTM, from Pure Software Inc. Considerable amount of effort was involved in getting useful information from the call graph produced by this tool at a sufficiently abstract level. (About 7 person-days were required to get results from the field runs, analyze the QuantifyTM generated data, and form a layered queueing model from this data.)

The performance model of the above architecture assumes the replication paradigm discussed in [Triantafillou95], as in section 3.1.3. In order to account for the consistency management overhead (in terms of extra execution), each replica of the database is associated with a replica of the transaction overheads task, that executes on the same CPU as the database replica. Thus, whenever a replica of a database is created, a copy of the transaction overheads server is also created, and both are allocated to a new CPU. The transaction overheads task accounts for the synchronous and asynchronous broadcasting overheads, locking overheads, etc. for consistency management, and the calls made by the database entries to this task during the operation, prepare, commit and abort phase are proportional to the number of database replicas in the system.

The number of calls from any client entry of the database task are equally divided among all the replicas.

The scaling path used for analyzing the scalability of this architecture is different from that used in section 6.1.

In this experiment, we focus on only the replication of the database. The database forms a performance bottleneck in the system. It is assumed that the location server does not pose as serious a performance problem as the database itself, and that it can be replicated in a rather cheap manner as compared to the database, in terms of its dollar cost as well as overheads cost. In other words, whether replicated or not, the location server does not have a significant impact on the cost and the performance of the system. Therefore, the replication of the location server is given a secondary role in the scalability analysis of the system, and for simplicity only a single copy of the location server that can fork unlimited number of threads (one per request) is assumed.

The scale factor in this experiment indicates the number of replicas of the database. Each copy of the database has its own dedicated workstation. The transactions overheads task is also replicated along with the database as mentioned earlier, and each CPU that runs one replica of the database also runs one replica of the transaction overheads task.

The remaining objects in this system are assumed to be executing on PCs, and all the hardware platforms are assumed to support DCE. Most of the communication in this system occurs between the database and the rest of the objects. Since all the database copies have dedicated workstations, collocation of database objects is not a feasible scalability enabler, and in the asymptotic case of replication, most of the communication among the objects in the system occurs using the remote procedure calls. The fixed remote invocation overheads are thus incorporated in the service times of the task entries.

Therefore, only the number of clients allowed in the system is used as a scalability enabler.

The user-specified target response time, for the process of setting up and tearing down the VPN with a frequency of 0.1 per cycle (the “cycle” refers to the execution scenario in the model), and setting up as well as tearing down 5 virtual channels for video conferences among the two user-specified sites is fixed at 15 minutes ($\hat{T} = 15 \text{ min}$).

The provisioning cost for all the components of this connection management system, including one copy of the database server, and one processor per software task, is \$100,000. Each extra copy of the database server (including a new dedicated processor) is assumed to cost an additional \$5000. (Thus, the overall cost is not heavily influenced by the scale factor unlike the other two systems discussed in this chapter. Yet, it is observed how excessive overheads and an inefficient data model affect this system’s scalability.)

The reference configuration of the system has a single database copy, and is optimized with respect to the number of clients, giving a reference productivity of $1.9542\text{e-}02$ per second per unit cost, and a reference throughput of 95 per hour. (Due to the way the scenarios are modeled, this throughput figure corresponds to an accumulated effect of setting up as well as tearing down 9.5 virtual private networks, and setting up and tearing down about 475 virtual channels per hour).

For this system, the lower limit on an acceptable value of the scalability metric is chosen to be 0.8. This is an arbitrary figure close to 1, entirely based on the designer’s judgement.

In case of this system, the number of write transactions is significant. But the granularity of the database objects is small, and a large number of objects are accessed by clients (which is later found to cause performance problems). Assuming that the database accesses are uniformly distributed across this large number of objects, the probability that more than one client has to be blocked on a lock for accessing the same object is assumed to be negligible. Moreover, the logistics suggest that in many cases, different clients are likely to access mutually exclusive objects (e.g. the access groups that are not already in service). Thus, the software bottlenecking effect of locking is ignored in this case, but the execution overheads, which in fact form a major constraint for replication, have been considered.

A powerful feature of the layered queueing network solver (LQNS) can be exploited when the scaling path is symmetric, as it is in this case. The symmetric replication facilitates the use of “replication flag” in LQNS model input file. The LQNS then analyzes all the symmetrically replicated servers using only one replica of each, and applies the results to the overall layered model.

Thus, the layered models can be solved at a constant computational cost, although the actual scale of the system (in terms of the magnitude of replication) can be arbitrarily large. This has been further discussed in [Pan96]. The layered models employed in this section were solved using the replication flag, as well as by creating the detailed system models with all the replica represented separately. The results matched exactly, which are discussed in section 6.2.4.

6.2.3 Rule-of-thumb scalability bound

As mentioned earlier, this scalability analysis accounts for the transaction management overheads associated with the database replication. The transaction management overheads are embedded into the model, with the database entries making calls to the appropriate entries of the corresponding overheads server, in order to account for the two-phase commit protocol overheads. (The entries of the overheads task correspond to the operation phase, prepare phase, for the write operations an additional commit phase and with some probability, the abort phase). These calls are proportional to the number of replicas of the database (i.e. the scale factor).

Thus, accounting for the replication overheads is mandatory for the bounds analysis, because they are a function of the scale factor (and not the optional scalability enablers.)

This scalability analysis mainly focuses on analyzing if extra copies of the database buy proportional performance of the system. In order to find an upper bound on scalability, we consider the transaction overheads (in terms of their CPU demand) as a function of the scale factor. We assume that the remote invocation overheads are aggregated with the entry execution demands, as in the case of the optimization framework described in the following section.

We now construct a model corresponding to scale factor 1, extract the workload parameters from its output (including the workload corresponding to the overheads), and use the balanced queueing network bounds to compute the scalability bound expressed as a function of the scale factor.

Step 1. The model corresponding to scale factor 1, with the mandatory replication and remote-invocation overheads corresponding to scale factor =1 expressed (by appropriately invoking of the entries of the transaction overheads task corresponding to the *operation*, *prepare*, *commit*, and *abort* phase), is optimized with respect to the number of clients. The optimized throughput ($= 2.63916e-5$ per millisecond) is obtained, with productivity (F) $= 1.9548e-5$ units, and optimized number of clients = 23.

Step 2. The execution demands for all the tasks are obtained by dividing the total utilization corresponding to each task by the reference throughput. Note that the total demand and average demand calculation (required by the balanced system bounds for the queueing networks) includes the net transaction overheads demand, multiplied by k . (In the layered queueing models, overheads like those corresponding to the operation phase are actually carried out in the second phase, which cannot be modeled using queueing networks).

Step 3. We use the minimum of the balanced system throughput bound and the asymptotic throughput bound in the scalability bound calculation. For the response time, we use the balanced system lower bound on response time.

In this case, total demand, $D = 14.44 + (k \cdot 22.11)$ sec.,

average demand $= D_{avg}(k) = (14.44 + (k \cdot 22.11)) / (k+5)$ sec.,

$D_{max} = 35.08$ sec.,

response time $= D + (N-1) \cdot D_{avg} / (1 + (Z/D))$ which is the corresponding balanced system lower bound,

$Z_0 = 10$ min.,

cost $= C(k) = 1 + 0.05 \cdot k$,

Substituting, we get the following expression for the productivity bound:

$$F_{bnd}(k) = \frac{\min \left\{ \frac{k \cdot N}{Z_0 + D + (N-1) \left(\frac{D}{k+5} \cdot \frac{D}{(Z_0 + D)} \right)}, \frac{1}{D_{max}} \right\}}{(1 + 0.05k) \cdot \left(1 + \frac{1}{T} \cdot \left(Z_0 + D + (N-1) \left(\frac{D}{k+5} \cdot \frac{D}{(Z_0 + D)} \right) \right) \right)} \quad \dots \text{Eqn (6.2)}$$

As seen in equation 6.2, the overall demand, the number of processors, as well as the cost grow linearly with k . In the limiting case, the productivity bound drops rapidly as k^{-2} . Equation 6.2 suggests that the overhead demands would rapidly cause the devices to saturate, and limit the productivity bound.

Substituting in the scalability metric definition, we get the following plot for the scalability bound.

FIGURE 6.5

Rule-of-thumb scalability bound for the connection management system -(version B)

As observed from figure 6.5, the acceptable scalability limit (0.8) is reached at scale factor 8. This reasonably agrees with the optimization results obtained in the next section (which derives a scale factor limit of 5). Note that figure 6.5 also implies a productivity upper bound at each scale factor, obtained by multiplying the scalability bound by the reference productivity ($= 1.9548e-2$ quality responses per second per unit cost).

6.2.4 Results and discussion

The results are obtained by optimizing the scalability metric with respect to the available scalability enabler (the number of clients), at each scale factor starting with 1. The results are summarized in the following table and figures.

TABLE 6.5.

Scalability analysis results for the connection management system - (B)

Scale Factor	Productivity (Optimized) (sec^{-1} per unit cost) $\times 1\text{e-}2$	Scalability metric value (Optimized)	Throughput (operations per hour)	Normalized* Response Time	Database CPU Utilization		System Cost (units)
					Total	Due to transaction overheads	
1	1.95485	1.0	95	0.3017	92.59	—	1.05
2	2.02031	1.0335	108.62	0.3645	86.86	67.85	1.1
3	1.90128	0.9726	111.18	0.4126	82.43	69.45	1.15
4	1.75662	0.8986	112.01	0.4761	79.77	69.97	1.2
5	1.61546	0.8264	112.26	0.5449	77.98	70.12	1.25
6	1.4861	0.7602	110.95	0.5953	75.77	69.30	1.3
7	1.36948	0.7006	110.94	0.6675	74.84	69.30	1.35

*The normalized response time indicates the response-time normalized with respect to the user-specified target mean response time.

The results show that despite the relatively lower cost of the database, the system cannot gain much productivity by replicating the database, because there are a large number of write transactions, which cause excessive amount of overheads with replication. The ratio of the total number of read operations to write operations is approximately 2:1. Thus, it is evident from the throughput, cost and the response time figures that this system does not scale. The following discussion further illustrates why this system is not scalable using the given scaling strategy, and concludes that excessive consistency management overheads make the system non-scalable, despite cheaper resources. (This paradigm is contrary to the system discussed in section 6.3).

These scalability analysis results suggest that in the presence of a large number of database transactions (especially, the write transactions), the database schema design is of great importance for scalability. A scaling strategy where the overheads dominate the concurrency advantages proves to be highly inefficient. This is illustrated in the following diagrams and the discussion.

(a) Productivity VS the scale factor

(b) Scalability VS the scale factor

(c) (Optimized) throughput VS the scale factor

(d) Percentage utilization of the database CPU VS the scale factor

FIGURE 6.6

Scalability analysis results - Connection Management System (B)

Figure 6.6 (a) shows how the optimized productivity changes with the scale factor. When the number of replica increase from 1 to 2, the productivity increases, and later starts declining as an effect of two things - firstly, the transaction management overheads, which grow with the database replication, and secondly, the increasing provisioning cost (in terms of dollars). In the absence of both of these constraints, theoretically, there won't be a scalability limit. Whether it is the cost in terms of dollars or the effect of overheads in terms of saturation that is predominant can be observed from the optimized throughput, response time and utilization curves, as discussed below.

Figure 6.6 (b) highlights the values of (optimized) scalability metric as a function of the scale factor. The scalability drops to about 85% at a scale factor of 5 and drops below the acceptability limit of 80% beyond this scale factor. Note that the values of the (optimized) scalability metric do not drop drastically until beyond a scale factor of 5, thus falling in the acceptability band (0.8 and above). This is in agreement with the rule-of-thumb bound presented in section 6.2.3.

Figure 6.6 (c) shows how the throughput, optimized at each value of the scale factor, approaches saturation at a scale factor limit of 5. Note that beyond a scale factor of 2, the scalability drops monotonically, but the throughput curve still keeps rising until the saturation point.

We observe that the optimizer meets the user-specified target mean response time, at all the scale factors. *Note that in this case, it is not the QoS function that restricts the throughput, but it is the device saturation.* The metric rewards or penalizes the (normalized) response times according to the QoS function discussed in Chapter 4. As the scale factor increases and the database CPU, which is the bottleneck, gradually moves away from saturation, the optimizer attempts to maximize the throughput, settling for a higher response time.

Figure 6.6 (d) shows how the utilizations change as a function of scale factor. With increasing scale factor, due to better load balancing, we observe that the bottlenecking effect at the database is gradually reduced. The transaction management overheads at each database site are a result of the balance between following two conflicting effects:

- The transaction management overheads at each site are proportional to the inflow of requests, which reduce as a result of replication.
- On the other hand, the transaction management overheads increase with the overall replication level, due to the database consistency management requirements.

An asymptotic value of about 70% utilization due to only these overheads (per database CPU) is reached. At the scale factor 5, about 70% utilization of the CPU is due to the transaction management overheads, and only an additional 10% corresponds to the database operations. This is due to the fact that many of the transactions involve write operations on the database, which have heavy overheads corresponding to the two-phase commit protocol (This is not the case with the call processing system discussed in the next section.)

Thus, it can be concluded that the connection management system discussed herein is scalable up to a scale factor 5, at which point the database sites make up about 30% of the initial system cost. The number of connections for say, a video conference, that can be set-up and deleted per hour (including the setting up and releasing of a new virtual private network per every 50 conferences) can be scaled from 475 to 555 by following this scaling strategy.

It can be observed that the throughput cannot be scaled much, despite adding extra database copies (given that the database is in fact the bottleneck, this would have been a logical scaling choice). The reason, as discussed with the help of figure 6.6, appears to be an excessive amount of transactions (read as well as write) on the database, causing excessive transaction management overheads. The possible solutions are:

(1) The database schema would require re-working and many logically related database objects could be combined resulting in a higher level of granularity, to reduce the overall number of read/write transactions. Due to reduced communication with the database, this can lead to less overall execution demand per response.

(2) It was observed that the routing algorithms at the topology, VP and SONET layer caused much communication between database and other objects. With a more efficient implementation, and perhaps a better tree-search algorithm for the routing function, the database interactions can be reduced and the design can be improved.

(3) Otherwise, a temporary solution may involve supporting only a high target response time to the users (i.e. adopting a lower commitment to QoS), and allowing more users in the system. This can result in higher productivity as explained in Chapter 4, (example H). In such a case, this analysis will have to be re-visited.

(4) In some cases, it might be possible to *partition* the database objects, instead of replicating the whole database. For example, in this case, the ATM objects and the SONET objects could possibly be partitioned into two disjoint parts, after redesigning the database schema. Although this strategy is not general, i.e., may not be applicable in all such systems, it can help increase the

execution concurrency without incurring consistency management overheads, since only a single copy of an object exists in the system at any given time. This may not be acceptable for certain systems that require high availability and fault tolerance, achieved by replication. Moreover, the location service that needs to point to the right database partition can then become a bottleneck.

The usefulness of this strategy should be tested by revisiting the entire analysis.

Thus, the results highlight that merely adding software and hardware resources (even when their cost in dollar terms is not a constraint) need not be a good scaling strategy, if there are a large amount of overheads associated with the scaling path. In order that such a strategy is effective, deep-rooted design changes may be required. The framework presented herein is found to help choose or reject such scaling strategies based on a rigorous evaluation, and it also helps understand the fundamental limitations of a given design in this regard.

6.3 The call processing system

This section describes the architecture of a prototype call-processing system used in digital telephony, based on proprietary message-oriented middleware. This system uses the basic two half-call model for call processing (discussed in section 6.3.1).

The objective of this exercise is summarized below:

- If a real, large-scale system were to be built based on the same technology and design choices as used by the prototype, assess up to what point the design is scalable, and if so, recommend how investment should be made in the hardware and software components of the system for supporting as large a number of users as possible, with a best attempt to meet the required QoS.
- It is obvious that as the number of users grows, the database that maintains the user connectivity information should be partitioned and/or replicated. Replication can improve the response time and facilitate load balancing at the cost of the overheads needed to maintain consistency among the replica. If we choose to use a location-service based replication model for database transactions, similar to the one discussed in [Triantafillou95], we would like to see how the system would scale, and how should the

investment in resources be optimized for supporting as large a number of users as possible, with a best attempt to meet the required QoS.

Before the implementation of the system, it is important to evaluate the effect of architectural choices on the scalability of the system, using the actual performance parameters from the prototype. The scalability analysis framework proposed in this thesis meets this requirement.

Section 6.3.1 discusses the two half-call model, and other architectural features embedded in the system's design. Section 6.3.2 discusses the hardware and software architecture of this system, including some novel schemes for fast message-passing between its software objects. Section 6.3.3 discusses the layered queueing model of this system, and the scalability analysis parameters. Section 6.3.4 applies the rule-of-thumb scalability bounds analysis to this system. Section 6.3.5 presents the scalability analysis results obtained using the tool PRASAD.

6.3.1 The two half-call model:

[McDonald90] discusses the basic design concepts and the software architecture for digital switching systems. The call processing model is partitioned into two logical parts - one corresponding to the call originating site and the other corresponding to the call terminating site. Hence, it is called the *two half-call model*.

The basic functions needed to be performed by the call-processing software are as follows:

- Detecting a request for service
- Interpreting the dialed digits
- Alerting the called party
- Establishing a connection
- Disconnecting a call
- Charging

Other features such as conference calls, call waiting etc. are extensions of the basic call processing functions mentioned above. Call processing begins with detecting a request for service. *Scan programs* poll each trunk for calls from other switches as well as from each subscriber line for a request for service (either call origination or call termination). When a scan program detects a request for origination (a user going off-hook), it stores the terminal number for that user

in the service request buffer. The *executive control program* polls this buffer in turn, and passes the terminal number to the origination program.

The *origination program* prepares the switching machine to receive digits from the originating terminal. This includes identifying the originating terminal's characteristics and connecting a digit receiver object to the terminal. (The terminal's characteristics are obtained from a database, centralized or distributed, using the terminal number as index.)

While the digit scanner collects the dialed digits, the validity of the digits is checked by another program, and when a valid terminating directory number is received, it is later used to access the database to obtain the expected termination of the dialed directory number and to obtain the routing information for the call.

A *call-record* data structure is then instantiated in memory, which exists for the duration of the call, and holds the state information relevant to the call. Call records are usually pooled resources, and are recycled upon call termination.

The call record usually contains (a) the control block - that contains the state of progress of the call, the time-out defaults, scanning rate for digit reception, and possible links to other call records and data structures in the memory (b) the information about the hardware resources used up by the call and (c) a private area for internal usage by the switching processes.

The originating call-setup program then references the switching centre network map, in order to find a path from the receiver to the terminal. When a path is selected, the corresponding resources are marked busy in the network map.

Note that in the case of the call-processing based on ATM, we assume that permanent virtual channels exist between all the switching nodes. Therefore, the call-routing function only needs to look up the right virtual channel (VC) identifier, corresponding to the connection between the originating site and the terminating site.

The originating call set-up program also stores the data associated with the receiver in memory (which contains a link to the call record object and vice versa).

If the terminating line is free, the ringing program at the terminating end gives a ring to the terminating line. Simultaneously, an audible ring is given to the originating site. Then the scan program is activated to look for an answer from the terminating line or a disconnect from the originating line. When a scan program detects an answer, the executive control program deactivates the ringing. At this point, the call is said to be in a stable state and much of the temporary memory

can be freed. The scan program at either end continues to look for the disconnect signal from the terminating lines, and at that time, the appropriate disconnect and billing programs are activated.

We assume that the post-processing such as billing lies outside the call connect/disconnect critical path, and it has not been modeled in the subsequent analysis.

Having discussed the generic call-processing logic, we now discuss another important component of the switching system software - the (location) database. The originating and terminating call setup and release procedures depend on the database for obtaining the attributes of the other side's location.

The database contains the data pertaining to the users as well as the equipment. This includes the information about the lines that terminate local users, the directory number to terminal correspondents, trunks that are available to remote switches, features available to each user, and the routing and billing algorithms. Note that the routing logic, in our prototype, is taken care of at the ATM PVC establishment time and is not a part of the call-processing software.

The location database is usually organized as a set of hierarchical linked lists. The data can be accessed by a tree search based on the directory number as the key. As this database grows in size, it can become a potential performance bottleneck, if it is centralized and accessed simultaneously by several call originating/terminating programs. Therefore, this database can be replicated/ partitioned for better availability and performance. In order to direct the database requests to the right partition/replica, a meta-level location service may be required.

We consider this scenario in the present analysis of the call processing system. The prototype is implemented with a single copy, memory resident database. We use a replication model proposed by [Triantafilou95] in order to make the replication overheads part of our scalability analysis. The replication model proposed therein supports transactions, and is based on a location service paradigm, which fits well with the call processing database requirements.

6.3.2 The architecture of the call processing system:

The system under consideration differs from the traditional call-processing systems used in digital telephony. It is implemented using a message-oriented middleware based on a fast ATM network. The objective is to use the fast ATM technology to minimize the latency, and bypass the overheads such as the RPC stub overheads, TCP/IP protocol overheads, etc. in order to make the

distributed call processing as fast as possible. Thus the communication infrastructure is intended to be bottleneck-free.

The communication architecture used for this system is based on some of the concepts described in [Eicken95]. The U-Net communication architecture uses a virtual view of the network interface that can be directly accessed at user level, allowing direct access to high-speed communication devices.

The latency in communication is caused by the delays introduced by the network, as well as the processing overheads. The processing overheads refer to the time spent by the processor in buffer management, message copying, checksumming, flow-control handling, interrupts, controlling the network interface, etc. Due to availability of fast networking technologies, the performance hot-spots are being shifted from the network to the protocol software. That is, a significant fraction of the end-to-end latency is caused by the protocol overheads, and comparatively little by the network latency. Moreover, sending small messages is uneconomical if the overheads associated with each message are large. Therefore, it is important to reduce the communication overheads when the message size is small, as in case of the given call processing system. (The call processing system uses AAL5 [AAL5] and the message size is typically three ATM cells).

In traditional UNIX networking architectures, the protocol stacks are implemented as a part of the kernel. This disallows the use of dedicated protocols to deal with the application specific requirements, as in case of the call processing system under consideration. In [Eicken95], the authors observe that the lack of support for the integration of application and kernel buffer management introduces high processing overheads, especially so for the connection-oriented protocols.

By removing / bypassing the communication subsystem's boundary with the application-specific protocols, efficient communication protocols can be built. In order to achieve low-latency and high-bandwidth communication between applications, the U-Net model (which is used by this call-processing prototype) removes the kernel from the critical path of sending and receiving messages. This eliminates the system call overheads, and more importantly, allows buffer management at user level. The multiplexing/ de-multiplexing is embedded directly into the network interface. Thus, the network interface is virtualized, and each process has an illusion that it owns the direct interface to the network. This enables abstracting out the network interface for some applications, while still supporting legacy protocols through the kernel.

Description of the software and hardware components:

Fig. 6.7 shows the hardware architecture of the prototype call-processing system. The hosts (IBM Power PCs running AIX, 133 MHz) are connected to an ATM network. Each host has the call processing software that supports the two half-call model, corresponding to call origination as well as call termination.

FIGURE 6.7 Overview of the call-processing system's hardware infrastructure

Any process that wishes to access the network creates one or more objects called *end-points*, allocates memory for storing the messages (called the *communication segment*) and creates a set of send, receive and free message queues with each endpoint.

A point-to-point connection is assumed to exist between every pair of nodes in the network. The ATM permanent virtual channels are established before the call-processing tasks are started on all the nodes, and once the connections are established, the connection establishment delays do not appear in the performance of the call processing system.

To send a message, the user process composes the data in the communication segment and pushes the descriptor for the message onto the send queue. The network interface (embedded in a special integrated device driver - denoted as "H/W support for comm" in figure 6.7) then picks up the message and inserts it into the network. The network interface provides a flag associated with each message in the queue, to indicate if it has been injected into the network.

Each processor in the system has a large amount of memory and cache, and we assume that memory overflow doesn't occur.

Incoming messages are demultiplexed based on their destination. The data is then transferred to the appropriate communication segment and a message descriptor is pushed on the corresponding receive queue. The processes poll the receive queue periodically, to check for any arrivals.

The architecture of the call-processing system attempts to support a "true zero copy" architecture in which the data can be directly sent out to application data structure without intermediate buffering, and where the network interface can transfer the data directly into the user level data structures. The communication segments span the entire process address space and the sender specifies an offset within the destination communication segment at which the message data is to be deposited directly by the network interface. (Note that the special integrated device driver(s) on each node can directly access the hardware network interface, unlike the traditional Unix device drivers that treat the devices as files, with open, close, read and write operations defined on them.)

The error checking and correction due to network errors and/or buffer overflows etc. is handled at the application level, and is outside the scope of the scalability and performance analysis presented herein. But due to the reliability of the communication medium and the availability of large amount of memory and cache to each CPU, the error conditions are not significant for performance analysis.

Fig. 6.8 shows a schematic diagram of the structure of each node in the system.

FIGURE 6.8

Hardware and software layers at each call-processing node

The call-processing prototype used for this analysis was based on only two nodes, and invoked a total of about 80 objects per call, among all the collaborating processes. The performance model discussed in the next section uses an aggregated set of client and server tasks, and all the individual objects need not be modeled.

Dedicated ATM virtual channel circuits (VCCs) are used for call setup, and for carrying the actual voice traffic, another set of VCCs are used. Therefore, the actual voice traffic is not modeled in the scalability analysis herein, but we only deal with the objects that collaborate to set up an end-to-end call connection.

Layered queueing model of the system:

Based on the software and the hardware architecture discussed above, a layered queueing model of this system was constructed, which is shown in figure 6.9. This model represents the tasks involved in call processing at each of the call-processing sites in the system. *We assume that all the call-processing sites are identical, and use this symmetry to analyze the overall system.*

The software objects in this system can be split into two parts: the ones corresponding to the origination of a call and the others corresponding to the call termination, as discussed earlier. The ATM network and the Integrated Device Driver that enables the applications to directly send messages on the ATM network are modeled as delay servers.

Majority of the database transactions initiated by the `Orig_Call_Setup` task are *read-only transactions*, that read the customer-related data for authentication, and the appropriate VC identifier to reach the required termination site. Only when new call processing sites are added, or new VCs are established between the call processing sites, or new services are subscribed by the customer, do the database objects change. These operations are usually carried out in batches, typically off-line, and are infrequent as compared to the call setup/disconnect operations. *Therefore, the abort and commit overheads associated with the database transactions are negligible in this system, although the overheads associated with the prepare operation still impact the performance.*

Because most of the transactions are read-only in this case, and because the relative number of calls to the database is much lower than, say, as in case of section 6.2, we consider the overheads only in terms of the *latencies* associated with database replication, and not in terms of the execution overheads on CPUs. (This can be justified by the fact that in this case, the limiting factor for scalability is not the device saturation. It is the QoS part of the scalability metric which acts like a bottleneck, as shall be observed later. Moreover, the total overhead execution demand accounts for less than 20% of the execution demand of all the tasks, per replica of the database, and it may suffice to consider only the latency effects. All the CPUs, after optimization, are seen to be utilized at most 70%, and they can accommodate this extra execution overhead in the worst case scenario. This also helps to ensure that the LQNS operation is faster and robust, since the number of tasks in the system is thus kept limited, with only one latency server rather than one execution overhead server per CPU. LQNS was found to require excessive amount of memory while handling more than 40 tasks, which can even halt the execution of the solver at times, which needs to be avoided. All the database task entries make appropriate number of calls proportional to the number of replicas to the entries of this single delay task during the operation, prepare, commit and abort phases.

Remote invocation execution overheads are modeled in detail, by providing one remote overhead server per CPU. This not only accounts for the extra latency, but also the extra CPU utilization due to remote invocation of tasks. These tasks are not shown in fig. 6.9 because they are dynamically generated whenever the optimizer alters the allocation map during the run. This is further explained in section 6.3.3.

For each site, we assume that the average outgoing traffic is equal to the average incoming traffic. That is, all the customers are uniformly distributed among all the call-processing sites and

each call processing site is equally likely to receive calls as it is likely to originate calls. Therefore, the system hardware resources at any site are shared by the tasks that originate the call as well as those that terminate the call. This is modeled by connecting the call-origination machinery to the call-termination machinery at the given site itself. All the tasks in the system are modeled to be forking a thread per request (infinitely threaded), in order to account for the performance effects of asynchronous communication between objects.

The scalability analysis framework analyzes the effect of scaling the number of processors at any of the call processing sites. (These processors are assumed to communicate either over a fast ATM connection or over a local bus, such that the communication latency does not exceed 0.1 millisecond.)

FIGURE 6.9

Layered queueing model of the call-processing system

6.3.3 Application of the scalability analysis framework to the call-processing system

To apply the scalability analysis framework, we must maximize the scalability metric over a wide range of scale factor values of interest.

The scalability enablers in the system are:

- Collocation/distribution of objects: The number of objects in the system can grow, theoretically in an unbounded fashion (due to replication), but at least 10 software tasks are free to be allocated to the available number of CPUs (= the scale factor), thus giving a large number of collocation/distribution possibilities.
- Replication: The database and the location server are the only two objects that can be replicated. Every database replica causes increased overheads ((a) in terms of the calls to the overheads server task, which are proportional to the number of database replicas in the system; and also (b) the cost in dollar terms.) However, the location server task is assumed to have negligible replication costs, and can be freely replicated.

In order to account for the remote-invocation overheads, each CPU is provided with a dedicated task that accounts for the latency associated with the remote invocation of a server by a client. Based on the number of calls made by a client to a server that executes on another CPU, both the client and the server entries invoke their local remote-delay servers by making a proportional number of calls. Note that this accounts for the execution overheads on all the CPUs, where remote invocation of services occurs, and it also accounts for the additional latencies seen by the clients due to remote invocation of services.

In the model corresponding to scale factor =1, the database corresponds to only 15% of the total CPU utilization of about 65%, and does not pose a single bottleneck, although it contributes significantly to the system's performance. The number of calls by the database entries to the transaction overheads task, as before, is a function of the number of replicas in the system.

Each database replica is assumed to cost 10% of each CPU. For simplicity in the cost assumptions, all the CPUs are assumed to be identical in terms of their speed.

The user-specified target mean response time for setting up a call is 10 milliseconds. This is used to derive the QoS function. A scalability of 90%, (with respect to the reference system with a single CPU and a single database replica) would be satisfactory in this case. This is an arbitrary

trary figure, based on the designers' and planners' judgement (which is 80% in the previous example).

Because the transactions are mostly read-only for this system, the effect of software bottlenecks due to locking of individual database objects is assumed to be insignificant as compared to that due to the prepare phase execution overheads, and therefore has been ignored.

6.3.4 Rule-of-thumb scalability bound

This scalability analysis mainly focuses on analyzing if extra processors buy proportional performance of the system. Note that, unlike in section 6.2, database replication (with overheads proportional to the scale factor) does not form a part of the scale factor or a mandatory scaling path in this case. Similarly, remote invocation overheads also do not form a part of scale factor or the scaling path. The scaling path leaves all the available design parameters to be optimized "in field", for each scale factor. Therefore, we relax all the overheads associated with the replication as well as remote invocation in order to be able to find an upper bound on scalability (because in the best case scenario, these overheads would be minimized to an insignificant level by the optimizer). Therefore, by relaxing the replication overheads, we assume only a single copy of the database in the layered model used to compute the scalability bound.

We now construct a model corresponding to a scale factor 1, extract the workload parameters from its output, and use the balanced queueing network bounds to evaluate the scalability bound as a function of the scale factor. This is explained below.

Step 1. The model corresponding to scale factor 1 is optimized with respect to the number of clients. In the given case, the model has a single CPU on which all the tasks (other than the tasks that represent delays and the reference task) execute. The optimized throughput ($= 0.304393$ per millisecond) is obtained, with productivity $= 0.16$ and optimized no. of clients $= 182638$. The optimization results presented in the next section use the same reference model.

Step 2. The execution demands for all the tasks are obtained by dividing the total utilization corresponding to each task by the reference throughput. Note that if there are tasks that account for the overheads (none here), their demands would be expressed as a function of the scale factor, in the *total demand* and *average demand* calculation, as required by the balanced system bounds for the queueing networks.

Step 3. We assume that the throughput of the equivalent balanced system can scale at most by a factor k , due to the replication of its resources. We use the minimum of the balanced system throughput bound and the asymptotic throughput bound in the scalability bound calculation. For the response time, we use the balanced system lower bound on response time.

In this case, total demand, $D = 2.1215$ ms,

average demand = $D_{avg}(k) = (2.1215 / k)$ ms,

maximum demand = $D_{max} = 0.5$ ms,

response time bound = $R = D + (N-1) * D_{avg} / (1 + (Z/D))$ which is the corresponding balanced system lower bound,

cost = $C(k) = k$,

$Z_0 = 10$ min.,

throughput bound = $\min(1/D_{max}, k * N / (R + Z))$,

Substituting in the expression for the productivity bound,

$$F_{bnd}(k) = \frac{\min \left\{ \frac{k \cdot N}{Z_0 + D + (N-1) \left(\frac{D}{k} \cdot \frac{D}{(Z_0 + D)} \right)}, \frac{1}{D_{max}} \right\}}{(k) \cdot \left(1 + \frac{1}{\hat{T}} \cdot \left(Z_0 + D + (N-1) \left(\frac{D}{k} \cdot \frac{D}{(Z_0 + D)} \right) \right) \right)} \quad \dots \text{Eqn} \cdot 6.3$$

The total demand in this case is not a function of the scale factor, but is decided by how the scalability enablers tune the system configuration. For this bounds analysis, the total demand is taken to be constant, although the average demand varies inversely with k . The cost is directly proportional to k , and the productivity bound initially increases with k , and later drops as k^{-1} as $k \rightarrow \infty$.

Substituting in the scalability metric yields the plot in figure 6.10.

FIGURE 6.10

The rule-of-thumb scalability bound for the call-processing system

As mentioned earlier, the acceptable scalability limit has been decided to be 0.9. According to fig. 6.10, the upper bound on the scale factor is 11, that yields a scalability metric upper bound in the acceptable band. As observed in the next section, the actual limit of 3 for the scale factor has been rather loosely predicted by this bound. This because of the fact that the remote invocation as well as the database replication overheads have been relaxed in the analysis. These overheads cannot be decided by the scaling path, since they are decided by the values taken by the scalability enablers, which are not known prior to the optimization.

In the previous two systems analyzed in this section, the overheads were decided by only the scaling path. Hence they could accounted for in the balanced system bounds, which resulted in tighter bounds.

6.3.5 Results and discussion

The scalability of the system was analyzed over a scale factor interval of 1-15. The number of CPUs in the system is equal to the scale factor, and the scalability enablers, namely, the collocation/distribution of software objects, the replication levels of the database and the location server and the number of clients are optimized. (Because the threads are assumed to be cheap, we assume a thread-per request paradigm as the best paradigm and model it accordingly.)

TABLE 6.6. Scalability analysis results for the call processing system

Scale Factor	Optimal no. of replica		Productivity (Optimized) (ms^{-1} per unit cost)	Scalability metric value (Optimized)	Throughput (Calls per hour) $\times 10^6$	Normalized* Response Time	System Cost (units)
	Database	Location server					
1	1	1	0.1600	1	1.0958	0.72	1.1
2	1	1	0.1492	0.9325	2.1273	0.88	2.1
3	1	6	0.1444	0.9025	3.0531	0.89	3.1
5	4	2	0.0866	0.5415	3.6485	1.16	5.4
10	2	4	0.0589	0.3683	4.4934	1.11	10.2
15	2	10	0.0387	0.2421	4.4934	1.12	15.2

*The response time is normalized with respect to the user-specified target mean response time of 10 ms.

The *overhead latencies due to database replication*, measured per response, have been summarized below. These results correspond to the optimum replication levels mentioned in Table 6.6.

	Scale Factor = 5	Scale Factor = 10	Scale Factor = 15
Overhead latency (per response) due to database replication (ms)	2.10	0.7	0.7

The *remote invocation overheads* incurred by each CPU are different, and cannot be summarized concisely. The maximum among them are plotted in figure 6.11 (f) below.

(a) Productivity VS the scale factor

(b) Scalability VS the scale factor

(c) (Optimized) throughput VS the scale factor

(d) The number of (optimized) database and location server replica VS the scale factor

(e) The maximum CPU utilization and the load imbalance VS the scale factor

(f) The max. remote invocation overheads incurred, as a function of the scale factor

(g) Search through the solution space for all the scale factors using simulated annealing.

FIGURE 6.11

Results of the scalability analysis of the call processing system

The results show how the optimizer tries to strike a balance between the following:

- Collocation/Distribution of software tasks: Collocation of communicating objects on the same processor reduces the remote invocation overhead (incurred by the client as well as by the server), but may cause a load-imbalance in the overall system.
- Replication (and also removal of the excess replicas, conversely): Replication may help to achieve a fine-grained load balancing among the CPUs in the system, as well as to reduce the response times seen by the client tasks. In many cases, the penalty for replication is not only the associated overheads (in terms of performance) but also the cost in dollar terms (such as the site license fees). Note that the replication of the location server has neither of these penalties associated with it, although the database replication causes increased latency as well as increased cost.

Replication can also affect the remote invocation overheads. For instance, invoking a local replica of a server reduces the remote invocation overheads.

In summary, replication and distribution of objects can improve the overall load balance and response time but may introduce additional overheads, whereas collocation of objects can reduce some of the remote invocation overheads, but may cause load imbalance.

Therefore, a cumulative effect of all these factors is more complex to analyze than the analysis presented section 6.2, which did not have collocation/distribution (with overheads) as a scalability enabler.

The scale factors analyzed in this section are 1,2,3,5,10, and 15; such that the number of CPUs in the system are equal to the scale factor. Fig. 6.11-(a) shows how the (optimized) productivity of the system monotonically drops as a function of the scale factor. This is because the system cost grows with the scale factor more rapidly than the product of throughput and the QoS factor. However, this drop is within only 10% up to a scale factor of 3. After the scale factor of 3, a steeper drop is observed in the productivity of the system.

Figure 6.11-(b) shows how the (optimized) values of the scalability metric change with the scale factor, where the system configuration corresponding to the scale factor 1 is the reference configuration. Since a scalability of up to 90% is acceptable in this case, it can be concluded that the system is scalable up to a scale factor of 3. Figures 6.11-(c) to 6.11-(f) present further results obtained as a result of the optimization.

Figure 6.11-(c) shows the optimized throughput as a function of the scale factor. The scalability limit (at scale factor = 3) roughly corresponds to the knee of this curve. Note that in the overall scaling experiment, the throughput supported by the system has been increased from about a 1.09 million calls/hr to 4.5 million calls/hr, while maintaining a good QoS. But beyond the scale factor 3, the gains derived by scaling system resources are only marginal as against their cost in dollar terms. *In this case, the QoS part of the scalability metric constrains the maximum permissible throughput, rather than device saturation.* (The highest device utilization is around 70% at all scale factors). For scale factors beyond 3, the optimizer opts for a response time which is a little higher than the target response time, in order to allow more throughput.

Figure 6.11-(d) shows the optimized replication levels of the location server and the database server. Note that although most of the transactions in this system are read-only, the consistency management overheads in the prepare phase do impose a penalty. At scale factors 2 and 3, only one database replica of the database as well as the location server is chosen in order to strike a balance between the replication overheads and the load balancing. (An idea about load balance can be obtained from figure 6.11-(e)). At scale factor 3, the optimizer chooses 6 copies of the location server (which are free of any cost), in order to obtain a fine-grained load balancing, by distributing the incoming calls evenly across these replica. But at a scale factor 5, despite the database replication overheads, replicating the database seems to help improve collocation and load balance as seen in figure 6.11 (e). At a scale factor = 5, 4 replicas of the database and only two replicas of the location server are chosen in the optimal solution. Because the optimized replication levels (together with their appropriate allocation) can be unpredictable unless actually generated by the optimizer tool, only a set of discrete points generated by the tool has been plotted, without implying any pattern by joining these points. The highest number of location server replicas (=10) are chosen by the optimizer at a scale factor of 15.

Note that the replication levels are a highly irregular function of the scale factor, because they are obtained as a combined result of optimizing a variety of scalability enablers at each scale factor, and no pattern can be implied, nor can the irregular cases be called as “outliers”.

Beyond a scale factor of 10, there are too many CPUs in the system and almost every task has its own CPU. Note that a task is a unit of distribution, and since not all tasks are allowed to be replicated, there does not exist a fine-grained control over load balancing any more. It is observed that the optimizer creates a large number of location server replicas and tries to balance the load, as expected. Moreover, as the software tasks are spread out across the CPUs, they incur

an increasing amount of remote invocation overheads, as seen in figure 6.11 (f). Thus adding extra CPUs is not always a cost-effective solution for improving the system performance (even though there are no software bottlenecks), because although the workload can be distributed across the CPUs, the workload distribution may become uneven if the tasks have uneven CPU demands, and also the remote invocation overheads can significantly increase the overall workload itself. At a scale factor 10, the optimizer chooses to have only two replica of the database and 4 of the location server, whereas at a scale factor = 15, a large number (14) of the location server replica are chosen with only two database replica.

At scale factors 10 and 15, the optimizer chooses to leave some CPUs unused in order to be able to colocate some of the objects and save on remote-invocation overheads. Due to the existence of unused CPUs (one unused CPU for scale factor = 10, and two for scale factor = 15), the maximum CPU utilization is the same as $U_{\max} - U_{\min}$ as seen in figure 6.11-(e).

Figure 6.11-(f) indicates that as more CPUs are added in the system, the maximum of the remote invocation overheads expressed in terms of % utilization of the corresponding CPU, seems to grow. The maximum remote invocation overheads seem to level off at about 50%.

Figure 6.11-(g) shows the progress of the simulated annealing algorithm, as it searches through the solution space to find the (approximate) global optimum. It was observed that simulated annealing usually generates good quality solutions, although it can sometimes discard good solutions in an attempt to enable the “hill climbing” required for reaching the global optimum. The optimizer does not seem to have converged in a reasonable amount of time for $k=15$, due to the very large size of the solution space and due to non-continuous behavior of the objective function with respect to various available scalability enablers. The tool PRASAD keeps track of the *best solution found upto any point during the execution*, and uses the best model found by the end of execution for further analysis. The results listed in table 6.6 correspond to the best models thus found.

Thus, we conclude that the system is scalable up to a scale factor of 3, at which point it can support about 3.1 million calls per hour. If a the system needs to be scaled beyond this point in a cost-effective manner, the following improvements can be tried:

- As a short-term solution, a higher target response time (more than 10 ms) can be offered to the customers
- The software objects tasks can to be redesigned, in order to try to match the execution and communication demands of all the tasks, thus avoiding execution imbalance in the

system. This may involve redistributing the functionality of the clients and servers such that they cause approximately same execution and communication demand. Such a division may not always be possible, especially in a multi-network scenario, where the ownership of the hardware and the software resources is split between various service providers.

- If the database schema could be modified in such a way that the database could be partitioned in various domains rather than replicating it, the consistency management overheads can be reduced. However, the location service faces an increased execution demand in such a scenario.

Thus the scalability analysis framework and its implementation have been demonstrated using realistic models of systems in this chapter. It can be seen that the optimizer explores various configuration possibilities, and gives a judgement about the scalability limits of a given design. When such information is collected for various scaling strategies, for various alternative designs, a good choice of a design for a given scaling plan (and a good scaling plan for a given design) can be quantitatively decided.

Also, based on the factors that cause the design limitations, guidelines about the changes required in the design to achieve better scalability can be obtained. Typically, these guidelines won't be general enough to be able to derive rules of thumb for *other* types of systems, since the scaling strategies, the scale factors, and even the scalability enablers may be different for different systems. *Hence, it is suggested that this framework be applied to each individual system of interest, rather than extrapolate the observations made in this Chapter to other types of systems. However, the scalability notions presented herein are general and the framework is flexible enough to be applied to any general distributed system.*

Chapter 7 Conclusion

With the evolution of the technologies enabling distributed computing, such systems are fast replacing legacy systems and are becoming more and more popular. The application domains range from office environments to scientific computing. The distributed execution platforms typically include a complex service architecture, and a hardware and software infrastructure to support such a distribution of the functions among clients and servers.

One of the basic requirements for a distributed system is scalability. In fact, scalability is what gives an edge to distributed systems over centralized systems. Unless a distributed system is scalable, that is, unless it is capable of growing in its size and operational capacity while supporting a payload proportional to the investment in its resources, many of the advantages of distribution are lost. Scaling up a distributed system means changing the population of its physical and/or software resources in order to achieve some enhancement in the system's performance at a justifiable cost.

Scalability with respect to hardware and software resources of the system is associated with achieving a sufficiently higher amount of concurrency to justify added resources and relatively smaller, or "tolerable" amount of additional overheads such as extra network bandwidth demand and extra device utilization. Scalability with respect to logical resources such as embedded algorithms refers to their computational complexity as well as flexibility while adapting to changes in the system architecture. Several such aspects of scalability have to be considered in a single framework for application to distributed systems.

Due to the complexity involved in designing and implementing distributed systems, it is often not clear at design time how the system would scale with respect to its traffic handling capacity, its software, and hardware, its communication infrastructure, etc. The dimensions along which a distributed system could scale is a design and capacity planning choice. The implementation of an apparently good design may actually turn out to be non-scalable, because of a lack of an overall scalability picture. The scalability objective is to best utilize the resources of the distributed system, by minimizing the investment and maximizing the throughput and the quality of service.

While scalability analysis exists for massively parallel multiprocessor-based systems, a sufficiently quantitative scalability analysis did not exist for distributed systems. The main contri-

bution of this thesis is to provide a general scalability analysis framework for distributed systems, where “goodness” of the distributed designs can be quantified with respect to scalability, and the system scalability features and limitations can be explored automatically, using optimization. This can provide important guidelines while designing and deploying distributed systems.

Some conclusions and observations derived from of this research work are summarized below.

7.1 Research summary

A large amount of scalability analysis literature exists for a variety of multiprocessor-based massively parallel systems. This literature was surveyed in order to investigate if these scalability ideas can be directly extended to the distributed systems of interest. It was observed that the scalability analysis for such systems is closely tied to the underlying algorithm-machine combination and is inadequate to address the scalability needs of distributed systems, which are more heterogeneous in nature, and are required to handle concurrent requests from many clients simultaneously (e.g. the banking systems, reservation systems, etc.) The scalability analysis cannot be directly extended from massively parallel systems to distributed systems, as explained in section 2.4. However, it provides various useful insights for understanding various quantitative notions about scalability.

Before deriving a scalability metric for distributed systems, the nature of target distributed system architectures was explored, as discussed in Chapter 3. As discussed in some of the literature surveyed therein, certain software architecture styles (e.g. pipelines, blackboard architectures, etc.) can be suited for some class of applications, but may adversely affect the scalability if used for inappropriate type of applications. (e.g. pipelines may not be suitable for interactive multimedia applications according to [Garlan93].) Intuitively, the more flexible a design, and the more parameters it leaves for “tuning” in order to meet the user requirements, the more likely a system is to be scalable. This notion of scalability has to be captured by the scalability analysis framework for distributed systems, along with the effect of scaling a variety of distributed system resources, both software as well as hardware.

The contribution of this thesis is two-fold: firstly, in defining a new scalability metric suitable for distributed systems, and then in defining an optimization framework that allows to search

for a system configuration that optimizes this metric. This ensures that the design limitations are explored only after optimizing the flexible design parameters.

The initial definition of the scalability metric used in this research was based on the “Power” metric [Giessler78]. This scalability metric was applied to a telecommunication connection management system model as discussed in [Jogalekar96].

However, this metric was observed to unduly reward lower response times, no matter how small they already were, thus even settling for configurations with a much lower throughput than what would be otherwise possible. This has been discussed in Chapter 4. Therefore, a new scalability metric for general distributed systems was introduced. The metric forms an important basis on which the analysis and the optimization framework is built. The metric is a ratio of the given system’s *productivity* in any two different configurations. The productivity of any configuration is a function of its throughput, the QoS seen by its clients, and the provisioning cost (in dollar terms) for its software and hardware resources. A system is defined to be scalable from one configuration to the other, if this scalability metric yields a value greater than $1 - \epsilon$.

In order to assess the reasonability of the scalability metric, it was applied to various idealized systems, to tally the results produced with the intuition. The systems analyzed include M/M/1, M/G/1, M/M/m systems, product form balanced queueing networks, closed queueing networks with a single bottleneck, etc. It can be observed that the intuitive notion of bottleneck-removal associated with scalability enhancement is also captured by the metric.

The scalability analysis framework needs to capture not only the design description of a given distributed system, but also the description of its scaling strategy. This is done in the following way.

In this research, the designs of the distributed systems are described at an abstract level, using the layered queueing models of systems. In addition to expressing the distributed system components and their interactions at an abstract level, the layered queueing models can also predict the performance attributes of the systems analytically. The LQNS solver [Franks95] was found to be an efficient and adequate platform to evaluate the productivity of a given system configuration, which also allows computation of the scalability metric when any of the system resources expressed in the layered models are scaled.

The scaling strategy is expressed in terms of a *scale factor* (a factor indicating the size of the system), a *scaling path* (which is defined by the set of possible configurations as the scale factor changes) and the *scalability enablers* (free or partially free tunable design parameters). The

scalability limits of a design can be truly explored only after appropriately tuning the scalability enablers. (e.g. the multithreading levels of the software objects, allocation of software resources to hardware resources, replication levels of various resources which can be freely replicated, choice of communication protocols, etc.). Unless these “tunable” parameters are set so as to maximize the productivity, the true scalability limitations of a design cannot be explored. If these scalability enablers are not optimized, a design or a scaling path can be wrongly judged as non-scalable when it could actually be concluded to be scalable had these configuration parameters been set appropriately. *Hence, optimizing the scalability enablers (which are “tunable” design parameters) is an essential part of the distributed system scalability analysis.*

This optimization framework was implemented using simulated annealing, in the tool PRASAD (the PRAgmatic Scalable Architecture Designer). The practicality of the scalability analysis framework and the usefulness of its implementation was verified by applying it to considerably complex and realistic distributed system prototypes, in Chapter 6. Two different connection management systems were analyzed, using different scaling paths, scale factors, overhead models and cost functions. Scalability analysis of an ATM-based call processing system was also presented. The optimizer performs load balancing and overhead minimization simultaneously.

The results show that it can be non-trivial and counter-intuitive to judge the scalability of distributed systems without such a framework. The analysis not only explores the scalability limits of the design but, if needed, also helps derive conclusions about what aspects of the design may need re-engineering. For example, inefficient data modeling was exposed for the connection management system-B for the given read/write ratio, and the scalability limit was also found. On the other hand, for the call processing system, the throughput was scaled by the optimizer from about 1 million to 4.5 million calls per hour by adding more processors, but beyond a scale factor of 3, it was observed that the user-expected QoS constrains the throughput. Optimum allocation maps and replication levels were found at each step, as the system size grew.

The simulated annealing typically took about 10 hours of computation on an ULTRA SPARC workstation. Thus, though the solution is computationally expensive, the solution technique is general enough to apply to most systems of interest, and it was shown to effectively explore the design and scaling strategy limitations.

A computationally less expensive, but less accurate way of predicting the scalability bounds was also developed in this research. This rule-of-thumb scalability bound is based on the asymptotic as well as the balanced queueing system bounds on throughput and response time. If

the execution workload due to the overheads can be accurately modeled, the bound is seen to be fairly tight. The rule-of-thumb bounds analysis was applied to all the realistic prototypes analyzed by the optimizer in Chapter 6, and the results were seen to be satisfactory. If the scalability upper bound itself is unacceptable, then hours of computation to derive the same conclusion can be saved by this approach. On the other hand, if this upper bound is acceptable, then the optimal configurations at any scale factor can be generated using the optimizer tool.

Thus, the main contribution of this thesis is a comprehensive scalability analysis framework, including a metric, that can be used to optimize distributed architectures and explore their scalability limitations. Taking into account the results produced by the optimization framework, the re-engineering efforts could be focussed on the non-scalable features of the design. This can be valuable at an earlier (prototyping) stage in design - to actually find out the limitations of the design before it is implemented and deployed in large scale. Also, given a design and a scaling plan, a near-optimal configuration found by the optimizer tool PRASAD suggests the most cost-effective operating state for a given investment in resources.

Overall conclusions

This research shows that the scalability analysis framework presented in this thesis works well for real systems with considerable complexity. The optimization is computationally expensive, and could take hours of CPU time on an ULTRA SPARC station. More research is required in making the optimization faster.

If a given distributed system model is topologically symmetric, the modeling paradigm can exploit the symmetry and solve large system models at a (low), fixed computational cost, which can be a particularly valuable feature. This framework and the associated bounds analysis also gives useful insight into the re-engineering of a given system, if required.

7.2 Generality of the framework

Many variations in the scalability analysis problem are possible within the framework developed in this thesis, including:

- **Other cost functions:** The scalability analysis presented herein assumes that memory is a cheap resource, and threads can be added without a significant cost. With a little effort, the cost

function used in evaluating the productivity can be made to account for the cost of threads, bandwidth, and any other resources in the system which might be important for a given case.

- **Enriching the set of enablers:** Intuitively, the more the number of available “tuning parameters” for a design, the more likely is a design to be scalable. The set of scalability enablers which may be used in this framework can be extended or modified, if required by a given system. For example, a parameter could be defined to generate different layered queueing models of the systems based on the choice of communication protocols (say, at the transport layer). Which scalability enablers can be important for a given system is a matter of design judgement, and how they can be incorporated into the performance models is a matter of detail. The scalability analysis framework developed in this thesis is flexible enough to accommodate these generalizations with a little effort.

- **Other performance model solution techniques:** The framework currently uses the layered queueing network solver, LQNS [Franks95]. Although this modeling technique was seen to be adequate, the scalability analysis approach is not tied to it. It is straightforward to carry out the optimization of configurations using other performance analysis tools such as queueing network solvers, Petri nets, etc. A little effort would be required only to interface these solvers with the simulated annealing engine.

- **The QoS parameters:** The quality of service requirements sometimes can be specific to the systems. The QoS functions discussed in this thesis cover a broad notion of QoS, based on the mean response time. But it may be important for some systems to incorporate other QoS attributes in the metric. e.g. some systems may need the variances of response times, or the probabilities of packet loss, etc. to be included in the QoS function. Although this may modify the scalability metric, the optimization notions will be essentially the same.

7.3 Areas for further research

The following areas which have the potential for building upon the current work in a significant manner:

- **Multi-class analysis:** The scalability analysis presented in this thesis primarily addresses single class systems. In case of multi-class systems, the current scalability analysis framework can be applied for only one class at a time, treated independently of other classes of

users. The existing framework can also handle some specialized cases of multi-class systems if the class sizes have a relationship determined by k , and the class throughputs and QoS are combined in a known way. (e.g. the throughputs and the QoS averaged with known weights.) It may be desirable, in some cases, to extend the scalability metric to incorporate a number of classes which do not fall in this specialized category. Such an extension has to be justified using idealized systems, which may take significant amount of effort. Relative optimization of the user population in various classes so as to maximize the multiclass-version of the scalability metric may lead to a different version of the optimization framework than what is discussed herein.

- **Development of heuristics:** Although the basic simulated annealing approach has been found satisfactory, the annealing process may be guided by some heuristics to make the tool more efficient. It may be challenging to come up with a general purpose heuristics for allocation, replication and other scalability enablers in presence of overheads, as can be seen from some of the results presented herein. However, as and when the tool would be applied to several industrial prototypes for scalability analysis, it may be possible to deduce rules of thumb for scalable design and embed some of these rules in the simulated annealing based optimizer. This remains an open research problem.

- **Investigating open architectures and scalability:** The open implementations concept was discussed in Chapter 3. In this case, the servers provide a meta-level service control interface, in addition to the actual service invocation interface, to the clients. How the meta interfaces can be used in general as scalability enablers would make an interesting research topic to build on the present work.

References

- [Allen90] A.O.Allen, *Probability, Statistics and Queueing Theory with Computer Science Applications*, Academic Press, Inc., San Diego, 1990, Chapter 6.
- [Allison96] C.Allison, P.Harrington, F.Huang and M.Livesey, *Scalable services for Resource Management in Distributed and Networked Environments*, WARP Report W1-96, Division of Computer Science, University of St. Andrews, UK. Available at <http://warp.dcs.st-and.ac.uk/warp>
- [AAL5] Specification by ATM Forum, *ATM Adaptation Layer Type 5*, Available at <http://www.atmforum.com/atmforum/index.html>
- [Bilbro91] G.L.Bilbro and W.E.Snyder, *Optimization of Functions with Many Minima*, IEEE Transactions on Systems, Man and Cybernetics, vol. 21, no. 4, July/August 1991.
- [Bilbro92] G.Bilbro, *A New Optimization Strategy*, Internal report, TR-92/10, Centre for Communications and Signal Processing, Department of Electrical and Computer Engineering, North Carolina State University. Revised in IEEE Trans. SMC, vol. 24, no. 4, 1994.
- [Bokhari88] S.H.Bokhari, *Partitioning Problems in Parallel, Pipelined, and Distributed Computing*, IEEE Transactions on Computers, vol. 37, no. 1, January 1988.
- [Char95] O.Char, C.Evans and R.Bisbee, *Operating System Scalability: Windows NT vs. UNIX*, Intergraph Corporation, available at <http://www.ingr.com/ics/wkstas/ntscale.html>
- [Chu87] W.W.Chu and K.K.Leung, *Module Replication and Assignment for Real-Time Distributed Processing Systems*, Proceedings of the IEEE, vol. 75, no. 5, May 1987.
- [Converse70] A.O.Converse, *Optimization*, Holt, Rinehart and Winston, N.Y., 1970, Chapter 4.
- [Corba] The Common Object Request Broker Architecture standard, developed by the Object Management Group, <http://www.omg.org/corba/corbiop.html>
- [Dahlin] M.D.Dahlin, C.J.Mather, R.Y.Wang, T.E.Anderson and D.A.Patterson, *A Quantitative Analysis of Cache Policies for Scalable Network File Systems*, Internal report, Computer Science Division, University of California at Berkeley. Available at <http://now.cs.berkeley.edu:80/Dienst/U1>

[Eicken95] T. von Eicken, A. Basu, V. Buch and W. Vogels, *U-Net: A User-Level Network Interface for Parallel and Distributed Computing*, Proceedings of the 15 th ACM Symposium on Operating Systems Principles, Colorado, Dec. 1995, pp 1-14.

[Franks95] R.G.Franks, A.Hubbard, S.Majumdar, J.E.Neilson, D.C.Petriu, J.Rolia and C.M.Woodside, *A Toolset for Performance Engineering and Software Design of Client-Server Systems*, Performance Evaluation, vol. 24, no. 1-2, Nov. 1995, pp. 117-135.

[Friedrich95] R.J.Friedrich, J.A.Rolia, *Performance Evaluation of a Distributed Application Performance Monitor*, IEEE International Conference on Distributed Platforms, Dresden, Germany, February 1996.

[Gamma94] E.Gamma, R.Helm, R.Johnson and J.Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[Garlan93] D.Garlan and M.Shaw, *An Introduction to Software Architecture*, Proceedings of the ICSE 15 th Intl. Conf. on Software Engg., Baltimore, May 1993, pp 61-99.

[Giessler78] A.Giessler, J.Hanle, A.Konig, E.Pade, *Free Buffer Allocation - An Investigation by Simulation*, Computer Networks, 1978, pp 191-208.

[Grama93] A.Y.Grama, A.Gupta and V.Kumar, *Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures*, IEEE Parallel and Distributed Technology, August 1993, pp 12-21.

[Helal96] A.A.Helal, A.A.Heddaya, B.B.Bhargava, *Replication Techniques in Distributed Systems*, Kluwer Academic Publishers, Boston, 1996.

[Ingber93] L.Ingber, *Simulated Annealing: Practice Versus Theory*, J Mathl. Comput. Modelling, Vol. 18, No. 11, Dec. 1993.

[ITU95] International Telecommunications Union recommendation no. G.805, *Generic Functional Architectures for Transport Networks*, Nov. 1995.

[Jogalekar95] P.P.Jogalekar, G.Boersma, R. MacGillivray, C.M.Woodside, *TINA Architectures and Performance: A Telepresence Case Study*, Proceedings of TINA 95, Melbourne, Australia, Feb. 1995, pp 415-430.

[Jogalekar96] P.P.Jogalekar and C.M.Woodside, *A Scalability Metric for Distributed Computing Applications in Telecommunications*, Proceedings of the 15 th International Teletraffic Congress - Teletraffic Contributions to the Information Age, Editors- V.Ramaswami and P.Wirth, Vol. 2(a), 1997, pp 101-110.

[Jogalekar97] P.P. Jogalekar and C.M.Woodside, *Evaluating the Scalability of Distributed Systems*, Internal report SCE-96-11, Revised January 1997, Department of Systems and Computer Engineering, Carleton University, Ottawa.

[Kaudel92] F.J.Kaudel, *Static Task Allocation for Distributed Computers*, Ph.D. thesis, OCIEE-92-02, Department of Systems and Computer Engineering, Carleton University, Ottawa.

[Kiczales96-1] G.Kiczales, *Beyond the Black Box: Open Implementation*, IEEE Software, January 1996, available at www.parc.xerox.com/spl/projects/oi

[Kiczales96-2] G.Kiczales, J.Lamping, C.V.Lopes, A.Mendhekar and G.Murphy, available at www.parc.xerox.com/spl/projects/oi

[Kleinrock78] L.Kleinrock, *On Flow Control in Computer Networks*, ICC 1978, pp 27.2.1-27.2.5.

[Kleinrock79] L.Kleinrock, *Power and Deterministic Rules of Thumb for Probabilistic Problems in Computer Communications*, ICC 1979, pp 43.1.1-43.1.10.

[Laarhoven87] P.J.M van Laarhoven and E.H.L.Aarts, *Simulated Annealing: Theory and Applications*, D.Reidel Publishing Company, Boston, 1987.

[Lazowska84] E.D.Lazowska, J.Zahorjan, G.S.Graham, K.C.Sevcik, *Quantitative System Performance - Computer System Analysis Using Queueing Network Models*, Prentice-Hall, Inc., New Jersey, 1984, Chapter 5.

[Lee91] H.F.Lee, M.M.Srinivasan and C.A.Yano, *Characteristics of Optimal Workload Allocation for Closed Queueing Networks*, Performance Evaluation, vol. 12, 1991, pp 255-268.

[Lo88] V.M.Lo, *Heuristic Algorithms for Task Assignment in Distributed Systems*, IEEE Transactions on Computers, vol. 37, no. 11, November 1988.

[Majumdar91] S.Majumdar, C.M.Woodside, J.E.Neilson and D.C.Petriu, *Performance Bounds for Concurrent Software with Rendezvous*, Performance Evaluation, vol. 13, 1991, pp 207-236.

[McDonald90] J.C.McDonald, *Fundamentals of Digital Switching*, Plenum Press, New York, 1990, Chapter 4.

[Mellor-Crummey] J.Mellor-Crummey, M.L.Scott, *Synchronization Without Contention*, available at <ftp://ftp.cs.rutgers.edu/pub/papers/Systems>.

[Menasce94] D.A.Menasce, V.A.F. Almeida and L.W.Dowdy, *Capacity Planning and Performance Modeling*, Prentice Hall PTR, New Jersey, 1994, Chapter 3.

[Odp] ISO/IEC 10746-3 / CCITT Recommendation x.903, *Basic Reference Model of Open Distributed Processing - Part 3 - Prescriptive Model*, International Organization for Standardization and International Electrotechnical Committee, November 1992.

[Oommen96] B.J.Oommen and E.V. de St. Croix, *Graph Partitioning using Learning Automata*, IEEE transactions on computers, vol. 45, no. 2, February 1996.

[Olstad95] B.Olstad and F.Manne, *Efficient Partitioning of Sequences*, IEEE transactions on computers, vol. 44, no. 11, November 1995.

[Pan96] A.M.Pan, *Solving Stochastic Rendezvous Networks of Large Client-Server Systems with Symmetric Replication*, Master's thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, September 1996.

[Perry92] D.E.Perry and A.L.Wolf, *Foundations for the Study of Software Architecture*, ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, Oct. 1992, pp 40-52.

[Rolia95] J.A.Rolia and K.C.Sevcik, *The Method of Layers*, IEEE Transactions on Software Engineering, Vol. 21, No. 8, August 1995, pp 689-700.

[Rolia95-a] J.Rolia, V.Vetland, G.Hills, *Ensuring Responsiveness and Scalability for Distributed Systems*, Proceedings of CASCON'95, October 1995, Toronto, Canada.

[Sarukkai95] S.R.Sarukkai, P.Mehra and R.J.Block, *Automated Scalability Analysis of Message-Passing Parallel Programs*, IEEE Parallel and Distributed Technology, Winter 1995, pp 21-32.

[Sevcik93] K.C.Sevcik, *Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems*, Technical Report CSRI-282, Computer Science Research Institute, University of Toronto, Canada, March 1993.

[Sheikh97] Fahim Sheikh, Jerome Rolia, Pankaj Garg, Svend Frolund, Allan Shepherd, *Performance Evaluation of a Large Scale Distributed Application Design*, World Congress on Systems Simulation, Sept. 1997, Singapore.

[Shirley94] J.Shirley, W.Hu and D.Magid, *OSF Distributed Computing Environment - Guide to Writing DCE Applications*, Second Edition, O'Reilly and Associates Inc., May 1994.

[Sivasubramaniam94] A. Sivasubramaniam, U.Ramachandran, and H.Venkateswaran, *A Comparative Evaluation of Techniques for Studying Parallel System Performance*, Technical Report GIT-CC-94/38, College of Computing, Georgia Institute of Technology, Atlanta, September 1994.

[Sun93] X.H.Sun and L.M.Ni, *Scalable Problems and Memory-Bounded Speedup*, Journal of Parallel and Distributed Computing, vol. 19, 1993, pp 27-37.

[Sun95] X.H.Sun and J.Zhu, *Performance Considerations of Shared Virtual Memory Machines*, IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 11, November 1995, pp 1185-1194.

[Sundaram] C.R.M.Sundaram and D.L.Eager, *Future Applicability of Bus-Based Shared Memory Multiprocessors*, Internal report, Department of Computer Science, University of Saskatchewan, Saskatoon, available at <http://www.cs.usask.ca>

[Tina] The Telecommunications Information Networking Architecture Consortium, <http://www.tinac.com>

[Triantafillou95] P.Triantafillou and D.J.Taylor, *The Location-Based Paradigm for Replication: Achieving Efficiency and Availability in Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 21, January 1995, pp 1-18.

[Woodside93] C.M.Woodside and G.G.Monforton, *Fast Allocation of Processes in Distributed and Parallel Systems*, IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 2, February 1993.

[Woodside95] C.M.Woodside, J.E.Neilson, D.C.Petriu and S.Majumdar, *The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software*, IEEE Transactions on Computers, Vol. 44, No. 1, January 1995, pp 20-34.

[Yoshikawa] C.Yoshikawa, B.Chun, P.Eastham, A.Vahdat, T.Anderson and D.Culler, *Using Smart Clients to Build Scalable Services*, Internal report, Computer Science Division, University of California, Berkeley. Available at <http://now.cs.berkeley.edu/SmartClients>

IMAGE EVALUATION TEST TARGET (QA-3)

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

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