Performance Evaluation of Systems Built with Reusable Frameworks

Jinhua Zhang

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

**Master of Science
in
Information and System Science**

Department of Systems and Computer Engineering

**Carleton University
Ottawa, Ontario**

January 2003

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

ISBN: 0-612-83507-3

Our file Notre référence

ISBN: 0-612-83507-3

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.

Canada

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

**Performance Evaluation of Systems
Built with Reusable Frameworks**

Submitted by Jinhua Zhang
in partial fulfillment of the requirements for
the degree of Master of Science

Prof. D. Petriu, Thesis Supervisor

Prof. R. Goubran, Chair, Dept. of Systems and Computer Engineering

Department of Systems and Computer Engineering
Carleton University

January 2003

ABSTRACT

The objectives of the thesis are to get some insights on how the performance properties of a reusable framework influence the overall performance of a system built upon it, and to study the reusability of the framework's performance properties for predictive performance analysis.

A distributed multithreaded case study system, the Document Exchange Server (DES) system, has been developed with ACE frameworks and its performance has been studied with measurements and analytical modeling. The UML performance profile was applied to the DES system. The measured server's execution data was used as input parameters for the profile. A LQN performance model was built and used to assess the performance of the DES Retrieve Document scenario. The reusability of frameworks performance characteristics was discussed in this thesis. This study provides an approach to extend the reusability of frameworks from the software development domain to the performance domain.

ACKNOWLEDGMENTS

First of all, I would like to express my deep gratitude to Professor Dorina Petriu for her guidance, kindness, and generous support as my advisor during my study at Carleton University.

I would like thanks Mr. John Knox and the other staffs of the computer lab at the Department of Systems and Computer Engineering for their assistance during my working on this thesis.

I would also like to thank friends at the University of Ottawa and Carleton University for their friendship and encouragement.

Finally, I would like to thank my parents and sisters for their constantly understanding, encouraging, and supporting me to fulfill my achievements.

TABLE OF CONTENTS

ABSTRACTS	iii
ACKNOWLEDGEMENTS	iv
TABLE OF CONTENTS	v
LIST OF FIGURES.....	viii
LIST OF TABLES	x
CHAPTER 1 INTRODUCTION	1
1.1 Motivation	1
1.2 The Research Problem	4
1.3 Thesis Contributions	6
1.4 Organization of the Thesis.....	7
CHAPTER 2 BACKGROUND	9
2.1 Frameworks and Components.....	9
2.1.1 Frameworks.....	9
2.1.2 Patterns versus Frameworks.....	11
2.1.3 Components.....	12
2.1.4 Black-box, White-box, and Gray-box Reuse	14
2.2 ACE Frameworks Overview	15
2.3 Software Performance and Modeling.....	20
2.3.1 Software Performance	20
2.3.2 Software Performance Engineering (SPE).....	22
2.3.3 UML Performance Profile.....	26
2.3.4 Performance Models	27
2.4 Performance of Reusable Software	31

CHAPTER 3	DESIGN AND IMPLEMENTATION OF A CASE	
	STUDY SYSTEM.....	35
3.1	Introduction	35
3.2	The Document Exchange Server (DES) System.....	35
3.2.1	System Requirements and Use Cases.....	35
3.2.2	Perfromance Objectives	37
3.3	The Design and Implementation of the DES System.....	39
3.3.1	Architecture of the DES Server.....	40
3.3.2	Using ACE Task Framework to Create the Thread Pool	44
3.3.3	Using ACE Socket Wrappers for Connection Establishment and Data Transder	46
3.3.4	Reusing the Jaws I/O Framework	50
3.3.5	Applying the ACE Virtual Filecache System	55
3.3.6	The DES_Request and DES_Response Classes.....	55
3.3.7	Client Applications.....	59
3.4	Summary	59
CHAPTER 4	PERFORMANCE MEASUREMENTS OF THE DES	
	SYSTEM	60
4.1	Introduction	60
4.2	Rational Quantify as a Performance-profiling Tool.....	62
4.3	Performance Measurement Results and Discussions	67
4.3.1	Running Environment	67
4.3.2	The DES Server's Total Execution Time	68
4.3.3	The Execution of transmit_file() of the IO framework	74
4.4	Performance Measurement in a Performance Network	78
4.4.1	Performance Network Set-up	78
4.4.2	Response Time Distribution.....	79
4.4.3	The Average Response Time and Throughput.....	83
4.5	Summary	84
CHAPTER 5	LQN MODEL OF THE DES SYSTEM	85
5.1	Introduction	85
5.2	Applying UML Profile to the DES System.....	87

5.2.1	Annotations for Resources	87
5.2.2	Annotated UML Scenario Model	88
5.3	Performance Analysis of the DES System with LQN.....	91
5.3.1	The LQN Model for the Retrieve Document Scenario of the DES Server	91
5.3.2	Comparing the Results from Measurements and Modeling.....	97
5.3.3	The Impact of the Server's Multithreading on Performance	102
5.3.4	Caching versus Non-caching.....	104
5.4	Reusable Performance Information of Frameworks	106
5.4.1	Frameworks Performance Characteristics in the DES Server.....	106
5.4.2	Understanding the Structure and the Flow of Control in Frameworks.....	107
5.4.3	Studying the Performance of Frameworks at a Right Level of Abstraction	108
5.5	Summary	110
CHAPTER 6 SUMMARY AND CONCLUSIONS.....		112
6.1	Summary	112
6.2	Conclusions.....	113
6.3	Future Work	116
REFERENCES		117

LIST OF FIGURES

Figure 2.1	Application framework component architecture	10
Figure 2.2	A pattern language map in ACE	17
Figure 2.3	The SPE Workflow	24
Figure 2.4	A simple LQN model.....	30
Figure 3.1	Use cases for the Document Exchange System	36
Figure 3.2	Annotated Activity Diagram for the Retrieve Document Scenario	38
Figure 3.3	Architecture of the Thread-Pool Document Exchange Server	41
Figure 3.4	Activity Diagram of the Retrieve Document Use Case.....	42
Figure 3.5	Activity Diagram of the Upload Document Use Case.....	43
Figure 3.6	ACE_Task class relationship	44
Figure 3.7	Class Diagram of Synch_Thread_Pool_Task	45
Figure 3.8	The ACE connection-oriented socket classes relationship	47
Figure 3.9	Class Diagram of the DES_Server.....	48
Figure 3.10	Structure of Jaws IO Framework	51
Figure 3.11	Class Diagram of DES_Request and DES_Response classes	56
Figure 3.12	Retrieve Document Sequence Diagram-Synch_Thread_Pool _Task::svc()	57
Figure 3.13	Retrieve Document Sequence Diagram (cont'd) DES_Handler::read_initial()	58
Figure 4.1	Call graph displayed by the Rational Quantify tool..	64
Figure 4.2	The Functional Detail Window for Synch_Thread_Pool_Task ::svc()	65
Figure 4.3	The Quantify Runtime set up for time method selections	66
Figure 4.4	The server's execution time at a 933 MHz workstation for the Retrieve Document Use Case.....	70

Figure 4.5	Response time vs. document size for cached and non-cached documents in the Retrieve Document Use Case	71
Figure 4.6	The sequence diagram for DES_Response::normaml_response() for Retrieving Document	75
Figure 4.7	(a) The total execution time and (b) the CPU time for JAWS_Synch_IO::transmit_file() of the Jaws IO framework	76
Figure 4.8	The set-up of the alpha measurement network	79
Figure 4.9	The response time distribution for (a) one client (b) 10 concurrent clients in the alpha network. The size of the thread pool of the DES server is 5 and the requested document size is 50 KB.....	80
Figure 4.10	(a) The distribution of response time in increased order; (b) The Histogram of response time for Retrieve Documents scenario for 10 concurrent clients request 50 KB documents at the alpha network	81
Figure 4.11	(a) Response time and (b) Throughput vs. number of clients for a 5-thread DES server at the alpha network	83
Figure 5.1	The deployment of the Document Exchange System	88
Figure 5.2	Activity Diagram with performance annotation for Retrieve Document scenario	89
Figure 5.3	An LQN model for Retrieve Document scenario	92
Figure 5.4	LQN model equivalent with Figure 5.3	93
Figure 5.5	Response time sensitivity to I/O parameters for retrieval of (a) 5 KB and (b) 50 KB documents.....	99
Figure 5.6	Sensitivity of Ethernet packet service time on LQN modeling for retrieval of cached (a) 5 KB and (b) 50 KB documents at the alpha network	101
Figure 5.7	The performance modeling for various number of threads in the DES server for retrieval of cached 5 KB documents	103
Figure 5.8	The DES server's CPU utilization vs. number of threads in the thread pool for the retrieval of cached 5 KB documents.....	104
Figure 5.9	LQN modeling results for the retrieval of 5 KB documents with/without server file-caching strategy	105
Figure 5.10	An LQN Model of resource demands in a multithreaded application	108

LIST OF TABLES

Table 4.1	The DES server's Execution time for non-cached documents at a 933 MHz Pentium workstation	69
Table 4.2	The DES server's Execution time for cached documents at a 933 MHz Pentium workstation	69
Table 4.3	The DES Server's execution time for retrieving non-cached files	73
Table 4.4	The DES Server's execution time for retrieving cached files.....	73
Table 4.5	The CPU execution time and IO time for JAWS_Synch_IO ::transmit_file().....	77
Table 4.6	Response times (in ms) for retrieving 5 KB, 50 KB, 250 KB, and 500 KB documents in the alpha network	82
Table 5.1	LQN service times.....	96
Table 5.2	LQN number or requests	96
Table 5.3	I/O parameters for LQN model for retrieval of cached 5 KB and 50 KB documents	98
Table 5.4	The DES server's CPU utilization vs. number of threads for the retrieval of cached 5 KB documents on the alpha network.....	103

CHAPTER 1

INTRODUCTION

1.1 Motivation

Software requirements include functional and non-functional requirements. Functional requirements specify the services the software promises to provide, and non-functional requirements specify the quality of service (QoS) attributes such as performance, security, reliability, usability, modifiability, etc. Many software projects fail not because they did not carry out the work they promised, but because they could not meet their non-functional requirements [Smith01].

Performance characteristics (such as response time and throughput) are essential quality attributes of every software system. Many performance problems are due to fundamental architecture and design factors rather than to inefficient coding. Intuition about performance problems is not sufficient. In order to assess performance risks, early quantitative assessments are necessary. For this reason, the Software Performance Engineering (SPE) techniques proposed in the late 80's have been increasingly accepted and practiced [Smith90, Smith01]. SPE proposes to use quantitative methods and performance models in order to assess the performance effects of different design and implementation alternatives during the development of a system. SPE promotes the idea that the integration of performance analysis into the software development process, from

the earliest stages to the end, can insure that the system will meet its performance objectives. This would eliminate the need for “late-fixing” of performance problems, a frequent practiced approach that postpones any performance concerns until the system is completely implemented. Late fixes tend to be very expensive and inefficient, and the product may never reach its original performance requirements.

All software systems have performance constraints because every system has its limits on hardware and software resources. Many software systems fail to meet performance objectives when they are initially implemented. Menasce listed several main reasons in [Menasce02] for which developers often fail to produce software systems that meet their performance requirements:

- _ lack of required scientific principle and models for software development;
- _ lack of education in performance in Computer Science curricula;
- _ shortage of IT work force;
- _ single-user mindset of programmers;
- _ small database mindset of programmers.

Software reuse has a positive impact on software quality, software cost, and productivity. Software reuse is the process of creating software systems from existing software rather than building them from scratch [Krueger92]. Even though software reuse has been practiced over many years in one form or another, it is still an emerging discipline. The importance of software reuse stems from its goals to reduce software development cycle-time and to improve software quality. The reusability can be at different levels and different features. It can focus on design, architecture, and code

directly. There are many technologies and approaches targeting software reuse, such as class libraries, patterns, frameworks, and components.

Frameworks facilitate reuse by capturing successful software development strategies [Fayad97, Fayad99, Schmidt97a, Schmidt02b]. Frameworks focus on reuse of concrete designs, algorithms, and implementations in a particular programming language. Frameworks and components are intended for large-scale reuse. They provide generic and important functionality (services) for application software systems in order to avoid re-discovery, re-invention, and re-implementation.

Frameworks facilitate reuse at both design and code level and have the potential to make the performance of the system under construction more predictable. However, there has been little study so far on the performance evaluation of reusable frameworks. This thesis investigate the ways in which the performance characteristics of reusable frameworks can provide valuable information for building predictive performance models for the new software systems developed with the frameworks. This could facilitate early performance prediction of the new systems and make the performance evaluation more accurate, since a good part of the system is already built.

The main challenge for the thesis stems from the fact that the performance characteristics of a system depend on all of its elements: software, operating system, middleware, and hardware. The contributions of these elements are quite difficult to identify and separate. Performance characteristics are highly non-linear in systems with contention for different resources, which happens in most software systems.

The study of the framework performance could benefit the software development as follows:

- Provide reusable performance characteristics for the functionality offered by the framework. Because the frameworks are designed for reuse, their performance characteristic properties would be reusable for many applications.
- Provide a reference point for the application developers to understand how well the framework could perform and what the performance limitations introduced by a reusable framework are. This will help software architects and developers to make decisions at an early stage on choosing suitable frameworks or components for their systems.
- Facilitate the construction of predictive performance models for systems built on top of reusable frameworks, and make the early performance evaluation more accurate.

1.2 The Research Problem

a) Premise

We are starting from the premise that Software Performance Engineering should be integrated into software development from its early stages. Quantitative performance evaluation based on performance models should be used to assess the performance effects of different design and implementation alternatives during the development of a system.

b) Hypothesis

Reusable frameworks have reusable performance characteristics that can be used to facilitate the predictive performance evaluation of the systems built upon them. The

most qualified people to document the performance characteristics of a framework are its developers. The people who use the framework to develop new systems should be able to reuse both the functionality and the performance characteristics of the framework.

c) Challenges

The hypothesis of the thesis raises a series of challenges not yet answered by the research in the field.

- It is very difficult to separate the contributions of each part of a system to its overall performance. As mentioned before, the performance characteristics of a system are highly non-linear and depend on all of its elements: software, operating system, middleware, and hardware.
- It is not clear how to characterize the performance of a framework in a reusable way.
- Performance evaluation is very specific and instance based: we take into account the execution of some run-time component instances on a specific platform under a given workload. On the other hand, frameworks are generic, not completely specialized for a certain application and are intended to be portable to many platforms.

d) Methodology

The methodology for the thesis research is as follows:

- Build a case study system based on a set of reusable frameworks for networked distributed applications. This step was necessary because we did not have an existing system at our disposal.
- Measure the system performance at system-level and at a more detailed level.
- Build a UML model of the system with performance annotations based on the measurements. The annotations are according to the recently adopted UML Performance Profile.
- Build a Layered Queueing Network model of the system and validate it against the measurements.
- Identify what are the framework's contributions to the performance model, and try to express these contributions in a reusable manner.

e) Thesis objectives

The objectives of this thesis are:

- Get some insights on how the performance properties of reusable frameworks influence the performance of the overall system;
- Study the reusability of performance properties of reusable frameworks for predictive performance analysis.

1.3 Thesis Contributions

The contributions of this thesis are divided into two categories.

A. Contributions to Knowledge

- Proposed and conducted an investigation on the reusability of the performance characteristics of reusable frameworks. This study provides an approach to extend the reusability of frameworks from the software development domain to the performance domain. The key to describing reusable performance characteristics is the UML performance profile.

B. Practical Contributions

- Developed a multithreaded distributed case study system, Document Exchange Server (DES) System, using an existing set of frameworks for distributed networked systems, the ADAPTIVE Communication Environment (ACE).
- Studied the performance characteristics of the DES system through measurements.
- Applied the UML Performance Profile to the case study system for making performance annotations to the UML model of the system. Built a LQN performance model based on the annotated UML model. Performance modeling results are compared with the real system measurements. Identified the performance contributions of the framework and proposed a way to describe them.

1.4 Organization of the Thesis

The thesis consists of six chapters organized as follows:

Chapter 2 provides an overview of the background information related to this thesis, such as frameworks and components, performance and modeling, Software Performance

Engineering (SPE), UML performance profile, and the ADAPTIVE Communication Environment (ACE) frameworks for concurrent and network programming, which is used for the case study.

Chapter 3 describes the design and implementation of a case study system named the Document Exchange Server (DES), which is built upon the ACE and an I/O framework. Some design and implementation details are presented, such as ACE_Task framework, IO framework, ACE socket wrappers, and ACE virtual filecache system.

Chapter 4 presents the performance measurements of the DES system using the Rational Quantify profiling tool and a performance measurement network in the RADS lab.

Chapter 5 applies the UML performance profile to the DES system, and studies the performance of the DES system with LQN modeling. The measurement results and LQN modeling results are compared under various workloads, and their difference is discussed.

Chapter 6 summarizes and concludes the thesis and lists directions for future research.

CHAPTER 2

BACKGROUND

This chapter will give a brief overview on the topics related to the thesis study. These topics include frameworks and components, performance and modeling, and the ADAPTIVE Communication Environment (ACE) - a C++ middleware developed for concurrent and networked programming. Some related work on performance evaluation of reusable software components is also introduced.

2.1 Frameworks and Components

2.1.1 *Frameworks*

A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact [Johnson97]. Frameworks define “semi-complete” applications that embody domain-specific object structures and functionality. Application developers can customize the components provided by frameworks to produce user-specific applications. The components of a framework have a generic nature and can be reused as the basis for many similar applications. Complete applications can be composed of by inheriting from or instantiating framework components [Fayad97, Fayad99].

Frameworks target the reusability of design and code in a specific domain. It helps reduce the cost and improve the quality of applications by reifying software designs and pattern languages that have proven effectiveness in particular application domains. The primary benefits of OO application frameworks stem from the *modularity*, *reusability*, *extensibility*, and *inversion of control* they provide to developers [Schmidt97a, Schmidt97b, Schmidt02].

1. Frameworks enhance modularity by encapsulating implementation details behind stable interfaces. Framework modularity helps to improve software quality by localizing the impact of design and implementation changes. This localization reduces the effort required to understand and maintain existing software.

Figure 2.1 Application framework component architecture [Schmidt02b]

2. Frameworks enhance reusability through their interfaces by defining generic components that can be reapplied to create new applications. Reuse of framework components can yield substantial improvements in programmer productivity, and enhance the quality, performance, reliability and interoperability of software.

3. A framework enhances extensibility by providing explicit hook methods that allow applications to extend its interfaces. Hook methods decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context.

4. Frameworks are active and exhibit inversion of control at runtime. Inversion of control allows the framework to determine which application-specific methods to invoke in response to external events. When an event occurs, the framework's dispatcher will invoke hook methods on pre-registered handler objects (callback) and these objects will perform application-specific processing on the events as shown in Figure 2.1.

2.1.2 Patterns versus Frameworks

A design pattern describes a particular recurring design problem that arises in a specific design context and presents a well-proven solution for the problem [Gamma95]. Patterns document the structure and participants in common application program microstructures. Pattern language is a family of interrelated patterns that define a process for resolving software development problems systematically.

Software design patterns capture the intent behind the design of a software system. They standardize piecework to larger units. Design patterns have been identified to avoid dependence on classes when creating objects, on particular operations, on specific presentation or implementation, on particular algorithms, and on inheritance as the extension mechanism [Gamma95]. Design patterns can be seen as a means of transferring design knowledge. Design patterns provide an efficient means of studying

and later on reusing the designs of experienced software engineers. In contrast to methodology that tells us how to do something, a design pattern shows us how to do it.

Object-oriented design patterns and frameworks are intended to alleviate costly rediscovery and reinvention of core software concepts and abstractions. Patterns provide a way to encapsulate the design knowledge that offers solution to standard software development problems. However, patterns could only enable reuse of abstract design and architecture knowledge, and do not directly provide reusable code. Frameworks augment the study of patterns. Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well [Gamma95]. A framework may instantiate a family of design patterns to help developers to avoid the costly reinvention of common software components.

Although they can be thought as a more concrete form of a pattern, frameworks are more like techniques that reuse both design and code [Johnson97]. Frameworks can be viewed as a concrete reification of families of design patterns that are targeted for a particular application-domain. Design patterns can be viewed as more abstract micro-architectural elements of frameworks that document and motivate the semantics of frameworks in an effective way. When patterns are used to structure and document frameworks, nearly every class in the framework plays a well-defined role and collaborates effectively with other classes in the framework.

2.1.3 Components

Software components represent another approach for code reuse. Unlike frameworks, components are more independent units which are mostly meant for “black-

box' reuse. According to [Szyperski98], "a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". The ideal reuse technology provides components that can be easily connected to make a new system.

Components capture the deployment nature of software. A component is an encapsulated, distributable software package with a well-defined interface. It enables practical reuse of software parts and amortization of investments over multiple applications.

Software components have the following characteristics [Leavens00]:

- They are a piece of software.
- They encapsulate specific functionality with interfaces and behaviors by contracts and context dependencies.
- They are ready for plug and play in a component infrastructure.
- They are prepared for cooperation with other components.
- They enable the construction of software systems out of pieces.

Software component technology plays an important role in the development of systems. Examples include desktop applications based on COM/DCOM from Microsoft, Java Beans and Enterprise Java Beans from SUN Microsystems, and the CORBA Component Model standardized by OMG.

2.1.4 Black-box, White-box, and Gray-box Reuse

There are several reuse abstractions for software fragments. The black-box, gray-box, and white-box abstraction refer to the visibility of an implementation 'behind' its interface [Sameting98]. In a black-box abstraction, the clients (users) could only know the interface and its specification. On the other hand, the clients can conduct substantial interference through inheritance in a white-box abstraction. Gray-box reuses are those that reveal a controlled part of their implementation. Black-box reuse is for reusing a software fragment only through their interfaces and specification. White-box reuse refers to using a software fragment through its interfaces relying on the understanding from studying the actual implementation.

Components can be viewed as self-controlled instances of abstract data types (ADTs) that can be plugged together to form complete applications. In terms of OO design, a component is a black-box that defines a cohesive set of operations, which can be reused based solely upon knowledge of the syntax and semantics of its interface. Compared with frameworks, components are less tightly coupled and can support binary-level reuse.

In accordance with the techniques used to extend them, frameworks can also be classified as black-box frameworks and white-box frameworks [Fayad97], and even gray-box frameworks. Black-box frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. White-box frameworks depend on OO language features like inheritance and dynamic binding to achieve extensibility. Gray-box frameworks partially support

inheritance and overriding of pre-defined hook methods, by using patterns to reuse and extend existing functionality.

Frameworks can be used to develop components, whereby the component interfaces provide a Façade for the internal class structure of the framework. Components can be used as pluggable units in a black-box strategy for software building. In general, frameworks are often used to simplify the development of infrastructure and middleware software, whereas components are often used to simplify the development of end-user application software.

Frameworks are more customizable than components, and have more complex interfaces. Programmers have to learn these interfaces before they can use the framework. Learning a new framework is not easy. However, frameworks are powerful since they can be used for many applications that need the services provided by frameworks. A good framework can reduce the amount of effort to develop customized applications by an order of magnitude [Schmidt02b].

2.2 ACE Frameworks Overview

The ADAPTIVE Communication Environment (ACE) is an object-oriented open-source framework that implements fundamental design core patterns and software architectures for concurrent communication software. ACE provides a rich set of reusable C++ wrappers, class categories, and framework components that perform common network programming tasks across a wide range of operating system platforms [Schmidt97b, Schmidt02a, Schmidt02b].

The communication software tasks provided by ACE include:

- event demultiplexing and event handler dispatching;
- signal handling;
- service initialization;
- interprocess communication;
- shared memory management;
- message routing;
- dynamic configuration of distributed services;
- concurrent execution and synchronization.

ACE is targeted for developers of high-performance and real-time communication services and applications. ACE was developed to alleviate the concurrent and networked problems, such as portability, flexibility, and complexity of concurrent, networked programming in various OS systems, and different hardware and software platforms. ACE can be imported into and run portably on dozens of hardware and operation systems. Figure 2.2 shows the pattern language defined and implemented in ACE to support the development of concurrent and networked software systems more effectively [Schmidt00].

The ACE is designed using a layered architecture. The lower layers of ACE are OO wrappers that encapsulate existing OS network programming mechanisms. The higher layers of ACE extend the wrappers to provide OO frameworks and components that cover a broader range of application-oriented networking tasks and services.

1. *ACE OS Adaptation Layer*: The lower layer is the ACE OS Adaptation Layer that shields the higher layers of ACE from platform-specific dependencies associated with OS mechanisms, include:

- multi-threading and synchronization;
- interprocess communication;
- event demultiplexing explicit dynamic linking;
- memory-mapped files and shared memory.

2. *The ACE OO wrappers*: Above the OS Adaptation layer are OO wrappers that encapsulate and enhance existing OS concurrency and network programming mechanisms, such as concurrency, interprocess communication (IPC), and virtual memory mechanisms available on modern operating systems like Win32 and UNIX. Applications can combine and compose these components by selectively inheriting, aggregating, and instantiating the ACE wrapper classes.

The use of OO wrappers improves application robustness by encapsulating OS communication, concurrency, and virtual memory mechanisms with type-secure OO interfaces. This alleviates the need for applications to directly address to the underlying OS libraries that use weakly-typed C interfaces.

3. *The ACE Frameworks*: ACE contains a higher layer network programming framework that integrates and enhances the lower layer OS wrappers. These frameworks are based on a pattern language that has been applied to many networked applications and middleware.

- ACE Reactor and Proactor frameworks implement *Reactor* and *Proactor* architectural patterns. The Reactor and Proactor frameworks automate the detection, demultiplexing, and dispatching of application-defined handlers in response to various types of I/O-based, timer-based, signal based, and synchronization-based events.
- ACE Service Configurator framework implements the Component Configurator design pattern that allows an application to link/unlink its component implementations at run-time without having to modify, recompile, or re-link the application statically. The ACE Service Configurator framework supports the configuration of applications whose services can be assembled dynamically late in the design cycle.
- ACE Task concurrency framework implements various concurrency patterns, such as *Active Object* design pattern and *Half-Sync/Half-Asynchronous* architectural pattern. It helps to enhance the modularity and extensibility of concurrent object-oriented networked applications.
- ACE Acceptor-Connector framework implements the Acceptor-Connector pattern, which decouples the connection and initialization of cooperating peer services in a networked application from the processing they perform after being connected and initialized. The Acceptor-Connector framework allows applications to configure key properties of their connection topologies independently from the services they provide.
- ACE Streams framework implements the Pipes and Filters pattern, which is an architectural pattern that provides a structure for systems that process a stream of

data. The ACE Streams framework simplifies the development and composition of hierarchically layered services, such as user-level protocol stacks and network management agents.

4. *ACE Distributed Service Components*: ACE provides a standard library of network service components and reusable components for common distributed system tasks such as logging, naming, locking, and time synchronization. These components illustrate how to utilize ACE IPC wrappers, Reactor, Service Configurator, Service Initialization, Concurrency, Memory Management, and Streams components. When combined with OO language features, such as classes, inheritance, dynamic binding, and parameterized types, and design patterns, such as Abstract Factory, Builder, and Service configurator, the reusable ACE components facilitate the development of communication services and applications that may be updated and extended without modifying, recompiling, and re-linking.

2.3 Software Performance and Modeling

2.3.1 Software Performance

Performance is the degree to which a software system or component meets its objectives for timeliness [Smith00]. The performance is described as a series of demands and the system response. The performance data is related to the demands and system response, workload, and hardware parameters. The primary performance is measured by average response time and throughput. Response time is typically described from a user perspective and is defined as the time required by the system to serve a user's

request. The throughput is the number of requests that can be processed per unit of time. The maximum throughput is the capacity of the program. The performance parameters are workload dependent. Workload intensities specify the level of usage of the system, and can be open or closed. An open workload is usually specified as by the arrival rate of the requests, and a closed workload by the number of concurrent users.

Performance is an essential attribute of every software system. The failure of a software system can cause various consequences including damaged customer relations, income loss, need for additional project resources, market loss, business failure, and project failure [Smith98].

Functional requirements of a software system are usually clearly stated and understood, but non-functional requirements such as performance are often omitted at the early stage of the software development until it is too late to meet them. Many projects have failed not because they could not provide the required functionality, but because of their poor quality, which in many cases meant that they could not meet their performance requirements. According to [Smith90, Smith 01], many software systems could not be used as initially implemented due to performance problems.

Performance objectives specify the quantitative criteria for evaluating the performance characteristics of the system under development. The objectives could be expressed in response time, throughput, or constraints on resource usage. The well-formed requirements are verifiable. They define the performance profile (which responses), define the arrival process (open, closed, distribution), define the execution environment (OS, processor, networks) and constraints that must be met for either or both of delay and throughput.

To understand and define performance, we need to view a software system in three related ways [Woodside02]:

- scenarios: a scenario describes system behavior for a high-level operation by a sequence of sub-operations.
- resource demands: we need to know how an operation loads the system resources.
- architecture: the architecture describes the organization of objects and modules with interaction relationships.

Performance measures for a system include resource utilization, waiting times, execution demands and response time. A measure could be (a) a required value defined by system requirements or a performance budget based on the requirements, (b) an assumed value base on experience, (c) an estimated value by the calculation from a performance tool, or (d) a measured value.

2.3.2 Software Performance Engineering (SPE)

SPE is a systematic, quantitative approach to construct software systems to meet their performance objectives. It uses quantitative techniques to identify architectural, design, and implementation alternatives that will meet performance objectives [Smith90, Smith02]. SPE is model-based and software-oriented approach. Modeling is central to both SPE and object-oriented development. By building and analyzing models of the proposed software, we can explore its characteristics to determine if it will meet its requirements before we actually commit to build it. In SPE, performance study begins as early as possible in the development process, when a preliminary design exists.

Integration of software performance engineering into the software development process will greatly improve the software quality and lower the risk of software failure. The type of performance models used depends on the purpose of analysis, the level of detail available, and the precision of the input data. Early in the software development process, the knowledge of the software's design and implementation details is insufficient to model the system's performance precisely. Simple models with estimated parameters are used to get rapid feedback on whether the proposed software is likely to meet the performance objectives. Rough performance estimate at an early stage can prevent costly performance mistakes that may make the systems miss its performance requirements by orders of magnitude. As software development progresses, more and more details of the software design and implementation become available, so it's possible to replace the rough estimates of resource usage with more detailed estimates or with measurements of actual usage. Consequently, the accuracy of the performance predictions can improve while the project progresses.

Software architectures need to be decided at the earliest stage in a software development project. Architecture is a specification of the components of a system and the communication between them. Architecture guarantees certain behavioral properties of a conforming system and can be a powerful tool to aid the process of predicting the behavior of a system with that architecture, of managing the construction of a system, and of maintaining it.

The architectural decisions have the greatest impact on software quality. Although a good architecture alone could not guarantee that the product will reach the quality goals, a poor architecture surely can prevent their realizations. Inappropriate

Figure 2.3 The SPE Workflow [Smith01]

decisions or errors at early stages will cost excessively or cause the failure of the project. Most performance failures are due to a lack of consideration of performance issues early in the development process, in the architectural design phase. The “fix-it-later” approach is dangerous; it could not save the project if its major architecture is not suitable to handle the required tasks, no matter how the software is implemented.

The SPE process focuses on the system’s use cases and the scenarios that describe them. Use cases are defined as a part of the requirements definition, and are refined throughout the design process. From a development perspective, use cases and their scenarios provide a means for understanding and documenting the system’s requirements, architecture, and design. From a performance perspective, use cases allow to identify the workloads that are significant from a performance point of view, the collections of requests made by the users of the system. The scenarios allow to identify the processing steps involved in each workload.

The SPE process includes the following nine steps (as shown in Figure 2.3):

1. Assess performance risk;
2. Identify critical use cases;
3. Select key performance scenarios;
4. Establish performance objectives;
5. Construct performance models;
6. Determine software resource requirements;
7. Add computer resource requirements;
8. Evaluate the models;
9. Verify and validate the models;

To be effective, SPE should be an integral part of the software development process. SPE should start as early as possible in the development process.

2.3.3 UML Performance Profile

The Unified Modeling Language (UML) is a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. [Booch99]. UML is widely adopted as the *de facto* industry standard notation, and it greatly facilitates the software development.

The UML Performance Profile is intended for modeling a system from a performance perspective. It enables the construction of models that could be used to make quantitative prediction regarding the performance characteristics, facilitate communication of design intent between developers in a standard way, and to enable interoperability between various analysis and design tools. This profile adds stereotypes, annotations, and tagged values to the UML model to identify the key performance scenarios, their parameters and performance measures (response time or throughput, etc.), hardware and logical resources (CPU demands, I/O times, critical sections, etc.)

The *UML Profile for Schedulability, Performance and Time*, which has been adopted by OMG recently, describes two sub-profiles (one for schedulability and the other for performance) that is intended for general performance analysis of UML models [OMG02]. The profile provides facilities for:

- capturing performance *requirements* within the design context;
- associating performance-related *QoS characteristics* with selected elements of the UML model;

- specifying *execution parameters* which can be used by modeling tools to compute predicated performance characteristics;
- presenting performance results computed by modeling tools or found by measurement.

Tools for performance analysis can be used either to estimate the performance of a system instance by utilizing some performance models or to identify the bottlenecks or critical resources of the system.

2.3.4 *Performance Models*

A performance model should capture the performance characteristics of the entire system: software, operating system, middleware, and hardware. A performance model should capture all the system resources and the competing demands made on resources when executing different scenarios under different workloads. The results of a performance model are system-level measures such as end-to-end delay, response time, throughput, etc. There are many performance-modeling methods and tools available, such as schedulability analysis, simulation, queueing analysis, layered queuing, Petri nets, stochastic process algebra, etc.

Queueing Network (QN) Models have been widely and successfully used as a model for predicting the performance of traditional computer systems for more than three decades [Kleinrock75]. The QN model assumes that the features that affect performance most are the queueing for devices. It uses mean value analysis as a performance evaluation method to estimate the system performance. The problem with the QN model is that it is difficult to represent nested services, so it often fails to capture

important details of logical resources and process communication, or complex interactions among various software and hardware components in distributed and networked system such as client/server systems.

Layered Queuing Network (LQN) is developed as an extension of Queuing Network model for handling the complex interactions in distributed and concurrent software system [Rolia95, Woodside95, Woodside98]. LQN is a performance model as well as a new kind of architectural model and captures the queuing based contention and simultaneous resource possession aspects of a system. LQN models predict the performance of distributed and concurrent software system based on the structure of the software and its use of logical and physical resources. A layered model describes the system architecture as software and hardware modules with resources embedded in them in a layered fashion, and with resource demands as parameters. The advantage of LQN over QN is that LQN can easily represent nested services. A server that serves clients' requests could be in turn a client to other servers, and the returning results from these servers could be used to serve its own clients. The nested services are common for concurrent and distributed applications, which have complex interactions among software components, network, and hardware devices.

Some performance parameters of software are actually demand parameters that describe resource demands put on the system. The design determines the demands. The request arrivals and the demands together are sometimes called the workload.

A LQN model can be built either by analyzing data from an existing system or prototype, or by a conceptual analysis of the system, which can be done as follows:

- a structural analysis, based on a planned software and system structure, which begins by identifying the objects and services.
- a scenario analysis, with a set of major scenarios, which describe the sequence of activities for each scenario.
- a combination of the structural analysis and scenario analysis.

The parameters of an LQN model include:

- clients' population or arrival rates;
- average executions time for software task entries;
- average service time and number of visits for hardware devices;
- average number of visits to other task entries;
- average communication delay;
- scheduling discipline and multiplicity of software and hardware servers.

A LQN model is represented as an acyclic graph. The basic LQN constructs are devices, tasks, entries, and activities. The nodes, also called tasks, represent software entities and hardware devices. Software entities are drawn as parallelograms, and hardware devices are represented as circles. Requests are represented as arcs. Task models a concurrent process or object, or a logical or physical resource. A task has entries, which models a method of an object, through which it offers services to other tasks or users. An entry is drawn as a smaller parallelogram nested in a task. An entry has its own execution time and demands for other services. Servers with more than one entry still have a single input queue, where requests for different entries wait together. Service requests are made to entries, messages are sent to entries, and resource demand

parameters are attached to the execution of entries. The objects that are bound into a task have their demand parameters aggregated together.

Figure 2.4 A simple LQN model

A resource is any entity that can make the execution wait. Resources of all kinds are modeled as tasks, with entries for different services. This includes logical resources such as critical sections or mutexes, locks and control tokens, and hardware devices such as shared buses, processors, disks, and interface controllers.

The LQN tasks are classified into pure clients, pure servers, and active servers. Clients are the nodes with only outgoing arcs; the leaf nodes are normally hardware servers, such as processors, I/O devices, networks, etc. The nodes with both incoming

and outgoing arcs are usually software servers. A software or hardware server node could be either a single or a multiserver. Each server has an implicit message queue.

Figure 2.4 depicts a simple three-tiered LQN model of a client/server system with a database, which is accessed by the system server.

Research on automatic transformation of UML models annotated with performance information into a LQN performance models is ongoing, and it will greatly facilitate the integration of performance evaluation in the software development process [Amer02, Petriu02b].

2.4 Performance of Reusable Software

While the importance of performance analysis in software development has been acknowledged by researchers and software developers, its application to reusable frameworks and software components has not attracted much attention [Yacoub02]. There has been little work on performance studies focused on reusable frameworks in the literature. Most research on frameworks is concerned with the development and reusability of frameworks in design and code domains. Although performance is claimed as one of the benefits from framework reuse, little research has been done directly towards framework performance issues.

Components are intended for reuse as independent software units. The properties of component-based application impose constraints on the tools and methods used in performance analysis on the systems: the source code usually is not available and the application is often distributed in nature. Several studies on the performance evaluation

of components-based systems have been reported [Yacoub02, Sitaraman01, Liu02]. Sitaraman *et al.* introduced a compositional approach for performance analysis [Sitaraman01]. They claimed that reasoning about the functional or performance behavior of a component-based system must be compositional in order to be scalable. Compositional performance reasoning demands that components include performance specification, in addition to descriptions of functional behavior. The authors argued that classical techniques and notations for performance analysis are unsuitable for capturing performance behaviors of generic software components. They fail to work in the presence of parameterization and layering. The performance specification problems are so basic that there are unresolved research issues to be tackled even for the simplest reusable components. Only after the performance specification problems are tackled by practical proposals for sound performance reasoning, software developers could be able to engineer new systems by choosing and assembling components that best fit their performance requirements.

Liu *et al.* describes an approach to predict the performance of N-tier enterprise applications built upon commercial off-the-shelf (COTS) middleware components based on empirical testing and mathematical performance modeling [Liu02]. The models describe generic behaviors of application server components running on COTS middleware technologies. The parameters values in the models are necessarily different for each product because each product has unique performance and behavior characteristics. These values must be discovered through empirical testing. The authors claim that simple test cases used for gathering empirical results are effective. A set of test cases are defined and executed for each COTS middleware product. The results of

these tests make it possible to solve the models for each product, so the performance prediction for a given product becomes possible.

Yacoub [Yacoub02] discussed how component-based system properties influence the selection of methods and tools used to obtain and analyze performance measures. Performance analysis methods and techniques are highly dependent on the properties of the software system to be analyzed. Product line engineering is a special software reuse paradigm that is concerned with the development of applications within the same domain. It is used to facilitate the production of similar applications in the same domain through the composition of common domain components. The development of component-based reference architecture is crucial to the success of a true product line. The author discussed performance analysis of component-based software systems and its automation. Two sets of performance metrics, application-specific metrics and system-specific metrics, were identified in this paper. The combination of application-specific and system-specific metrics is useful for performance evaluation of component-based software systems. The analyst can use these metrics to identify areas of poor performance, and understand resources utilization and how it affects component execution.

As shown by these studies that targeted performance issues of component-based systems, the research in this area is hindered by the lack of (a) suitable measurement tools for practical performance studies, (b) suitable methodologies, and (c) of knowledge about the implementation of software components. In conclusion, more research effort is needed in this area.

Frameworks and components all aim for the reusability at a large scope. The characteristics of these reusable entities would have important impact on software systems that utilize them. The performance analysis and study on frameworks and components could provide the performance characteristics of these reusable entities. These performance characteristics could help the application developers to understand how well these reusable entities would perform under various situations and workloads as well as their limitations.

CHAPTER 3

DESIGN AND IMPLEMENTATION OF A CASE STUDY SYSTEM

3.1 Introduction

This chapter describes the design and implementation of a Document Exchange Server (DES) system case study, which is a framework-based distributed and concurrent client/server system. The DES system is built upon ACE frameworks [Schmidt02b] and an I/O framework [Hu99, Hu00]. In the following two chapters, we will discuss the performance measurements, the application of the UML performance profile to the DES system, and the performance modeling of the DES system.

3.2 The Document Exchange Server (DES) System

3.2.1 System Requirements and Use Cases

The document exchange system is a client/server system. It consists of one document exchange server and multiple clients. The server will run constantly waiting for the requests from clients. Documents are stored on the server machine. There are two types of users: general users and the system administrator. A general user can get the document directory on the server, upload documents to the server, and retrieve the

documents stored at the server. The system administrator can update the existing documents, require the document directory, and access the log files of the server. Each time a client sends a request to the server, the log file in the server will be updated to record this request, and this log file can also be used as a trace record.

Figure 3.1 Use cases for the Document Exchange Server System.

For simplicity, we did not require any security services, assuming that the system is used by trusted users. To increase the efficiency, the server will need to serve several clients simultaneously.

Use cases that represent pertinent system requirements for the DES system are:

- Upload Document Use Case.

- Retrieve Document Use Case.
- Get Document Directory Use Case.
- Update Document Use Case.
- Retrieve Logfile.

The relationship between the use cases is shown in Figure 3.1.

3.2.2 Performance Objectives

The most important use cases for the DES system are the Retrieve Documents use case and the Upload Document use case. These two use cases are the most used features for this system, especially the former. In this case study, we will focus on the Retrieve Document use case as the key performance scenario. To evaluate this scenario, we need a performance specification for the system for various workloads. Since the documents are of various sizes, the performance demands should reflect this variation. For this study, the server will run on a Pentium 266 MHz workstation with 256 MB RAM memory and Windows NT operating system. The clients and the server are connected via a 100 Mbps Ethernet.

The Retrieve Document Use Case: This use case is the most used (> ~90 %) for the DES system and its performance is one of the most critical factors to determine how successful this system could be. The server should be able to process random clients requests for 50 kilobyte documents at an average rate of 100 requests per second, with average response being completed within one second. For 250 kilobyte documents, the server should be able to process 50 requests per second with average of response being completed within one second.

Figure 3.2. Annotated Activity Diagram for the Retrieve Document Scenario

The Activity Diagram is stereotyped as an analysis context <<PAcontext>> (as shown in Figure 3.2). Its start point is stereotyped as workload <<PAclosedLoad>> with tags identifying the intensity of workload and its overall performance measures. For 50 kilobyte documents, \$Nusers = 100; and for 250 kilobyte documents, \$Nusers = 50. Since we will use LQN modeling to assess the performance of the system, we use mean values as performance parameters. The required mean response time for the Retrieve Document scenario should be 1 second in designed high workload (i.e. 100 clients for retrieving 50 KB documents and 50 clients for retrieving 250 KB documents), and the response time predicted by the LQN model will be stored in the variable \$RespT, as shown by the tag:

PArespTime = ('req', mean, (1, 'sec')), ('pred', mean, \$RespT)

We use mean value performance requirements here because the performance for this application is not so critical as some real-time applications such as a video streaming application. For a video application, we can specify the performance requirement using a percentile measure in the tag value as:

`PArespTime = ('req', percentile, 95, (1, 'sec')), ('pred', percentile, $RespT)`

to express the required 95% response time less than 1 second. The LQN solver *parasrvn* has been modified for predicting soft deadlines.

Since the design details are not available at current stage, we will refine the activity of requestDocument by sub-scenario in the next section. The dumbbell symbol indicates the refinement of an activity in another activity diagram.

3.3 The Design and Implementation of the DES System

Software is seldom built completely from scratch. Reusable components will greatly alleviate the effort and time in development, documentation, and maintenance. Reuse is a good feature for object-oriented analysis, design and programming. As an ideal case, existing software documents (architecture, design, and source code, etc) can be adapted to fit new requirements. However, we are far from the goal of making reuse the standard approach to software development.

For the server of the DES System, we adopted part of the architecture, design, and implementation of the Jaws web server, which is built upon ACE frameworks [Hu99]. The Jaws web server is a good example of a well-designed and implemented software system using ACE frameworks. The Jaws web server is designed and

implemented by Douglas Schmidt, the ACE creator, and his group members. We will modify and add some new features to meet our need as document exchange server system. The test client programs will be implemented with either ACE or Java to send requests to the DES server and to measure the end-to-end response time.

3.3.1 Architecture of the DES Server

One of the important design aspects need to consider at the beginning is the capacity of the server. The DES server should be able to process multiple requests concurrently in an efficient manner.

On multithread OSs such as Solaris and Windows NT, applications can use threads to simplify programming and take advantage of parallelism. Multithreading provides a cheaper concurrency mechanism and could increase throughput and better performance, produce more responsive programs. Multithreads offer performance improvement for the cases of higher contention for the server resources (cpu, disk) in that they can avoid single operation block the other requests. If there is not enough contention (i.e. most time only one thread is active), then single thread may provide less latency for fewer overheads in maintaining resources [Hughes97, Orasad97].

Maintaining a pool of threads can improve performance in some circumstances. Instead of creating and destroying many short-lived threads, it is much better holding on a pool of threads. The server launches multiple threads in advance to deal with incoming connections. A request is assigned to an already created thread, and when a thread completes its task, it is returned to the pool, ready for the next task. This will improve performance by amortizing the efforts of thread creation and destruction since thread

creation and destruction does have a significant overhead that is better avoided for short-lived threads.

Figure 3.3 Architecture of the Thread-Pool Document Exchange Server

Maintaining a thread pool could also limit the number of threads in the application. Every system has a response curve with diminishing returns after a certain number of threads are running on it (the knee). Unlimited threads running on a server could lead to exhaustion of the server's resources and finally block the server for service. The best practice is to let the server to queue the request and maintain whatever maximum number of threads is optimal for the server system. Figure 3.3 shows the architecture of the thread pool DES server.

Figure 3.4 Activity Diagram of the Retrieve Document Use Case.

Figure 3.5 Activity Diagram of the Upload Document Use Case

UML activity diagrams address the dynamic view of a system [Booch99]. The activity diagrams in Figure 3.4 and Figure 3.5 show the flow of activities of the system in the Retrieve Document and Upload Document Use Cases.

3.3.2 Using ACE Task Framework to Create the Thread Pool

Multithreading opens a new set of possibilities for solving software performance problems and software organization problems by providing programmers with the ability to utilize concurrency and parallelism in the software design and development phases. However, concurrent programming can be extremely difficult. Multithreaded environments introduce many pitfalls that single-thread environment do not have [Hughes97].

Figure 3.6 ACE_Task class relationship

ACE Task concurrency framework provides a facility for spawning threads in the context of an object and a flexible queueing mechanism for transferring messages between tasks. ACE_Task provides easy, synchronized, deadlock prevention, and thread safe environments for creating multithreaded programs in C++. Networked applications can use ACE Task classes to spawn, manage, and communicate between one or more threads within a process [Schmidt02b].

Figure 3.6 shows the class relationship in ACE Task framework. ACE_Task must be customized by subclasses to provide application-specific functionality by overriding its hook methods. For example, subclasses of ACE_Task could override its open() and close() hook methods to perform application-defined ACE_Task initialization and termination activities, such as spawning and canceling threads and allocating and freeing resources (e.g. connection control blocks, I/O handles, and synchronization locks). ACE_Task classes offer following capabilities:

Figure 3.7 Class Diagram of Synch_Thread_Pool_Task

- Uses an instance of `ACE_Message_Queue` to queue message that are passed between tasks.
- Be used in conjunction with the `ACE_Thread_Manager` to become an active object and process its queued messages in one or more threads of control.
- Instance of `ACE_Task` can be linked/unlinked dynamically via the ACE Service Configurator Framework.
- Instances of `ACE_Task` can serve as concrete event handlers via the Reactor Framework.
- Can be subclassed to create application-specific methods that queue message and/or process messages.

In the DES server, `Synch_Thread_Pool_Task` subclasses the `ACE_Task` to create application-specific synchronous thread pool server. The `ACE_Task::activate()` method uses the `ACE_Thread_Manager` to convert the task into an active object that runs the `svc()` method in each thread of the thread pool.

The class diagram of `Synch_Thread_Pool_Task` is shown in Figure 3.7. Its instance uses an `ACE_SOCKET_Acceptor` to accept request from network.

3.3.3 Using ACE Socket Wrappers for Connection Establishment and Data Transfer

A socket is an endpoint of communication that identifies a particular network address and port number. Network applications use Socket APIs to establish connections and communicate via socket endpoints. When messages are sent, they are queued at the sending socket until the underlying network protocol has transmitted them.

When messages arrive, they are queued at the receiving socket until the receiving process makes the necessary calls to receive them.

Socket is the de facto interprocess communication (IPC) mechanism over TCP/IP to exchange information for network applications. ACE Socket Wrapper façade classes are to encapsulate the functions and data provided by existing non-OO APIs with enhanced type-safety, ensured portability, and simplified common use cases. Figure 3.8 shows the ACE connection-oriented socket class relationships [Schmidt02a].

Figure 3.8 The ACE connection-oriented socket class relationships

ACE_SOCKET_Acceptor class defines a factory that creates new <ACE_Stream>s passively. The <ACE_SOCKET_Acceptor> has its own "passive-mode" socket. This serves as a factory to create so-called "data-mode" sockets, which are what the <ACE_SOCKET_Stream> encapsulates. Therefore, by inheriting from <ACE_SOCKET>, <ACE_SOCKET_Acceptor> gets its very own socket.

Figure 3.9 shows the class relationship of the DES_Server. The DES server class is a subclass of ACE_Service_Object and its init() method will initialize the Synch_Thread_Pool_Task object. The Synch_Thread_Pool_Task object becomes an active object with a number of threads after its activate() method is called. The activate() method uses the ACE_Thread_Manager associated with an ACE_Task to spawn threads:

this->activated (THR_DETACHED | THR_NEW_LWP, threads)

Figure 3.9 Class Diagram of the DES_Server

In each thread of the thread pool, the `Synch_Thread_Pool_Task::svc()` hook method will be invoked. The `svc()` is overridden by `Synch_Thread_Pool_Task` and used to perform application-specific work.

ACE_Task::svc() is the entry point for all threads created in the Task. On the DES server, each thread runs within an infinite loop, synchronously listens to the coming connection request from the Socket_Acceptor. When a connection request arrives, it uses an ACE_Message_Block to store the message, then parses the request, updates the logfile, and processes the request. After the request is served, the thread is recycled and returned to the pool to server another request.

The Synch_Thread_Pool_Task::svc() implementation is as follows:

```

1 int Synch_Thread_Pool_Task::svc (void)
2 {
3   Synch_DES_Handler_Factory factory;
4   for (;;)
5   {
6     ACE SOCK_Stream stream;
7     if (this->acceptor_.accept (stream) == -1)
8       ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "DES_Acceptor::accept"), -1);
9     ACE_Message_Block *mb;
10    ACE_NEW_RETURN (mb,
11                    ACE_Message_Block (DES_Handler::MAX_REQUEST_SIZE + 1),
12                    -1);
13    DES_Handler *handler = factory.create_DES_handler ();
14    handler->open (stream.get_handle (), *mb);
15    mb->release ();
16  }
17  ACE_NOTREACHED(return 0);
18 }

```

Line 3 Create a factory of DES_Handler binding to the synchronous I/O strategy and will be used to create a DES_Handler.

Line 4 Each thread will constantly run this unconditional *for loop* from here.

Line 6 – 8 The ACE_LOCK_SOCKET_Acceptor object is initialized to listen passively at designed ACE_INET_Addr address. On each work thread, acceptor_.accept(stream) is called to initialize the ACE_SOCKET_Stream with a newly accept client connection.

Line 9 –12 Create a new ACE_Message_Block that will be used to store messages in buffers as they are received from the network through the ACE_LOCK_SOCKET_Acceptor.

Line 13 Create a DES_Handler through the factory's create function to handle the request.

Line 14 Call DES_Handler's open () function which will create a receive and a send socket buffer and start a series of operations, such as read the request, parse the request, update log file and process the request, to perform the service for the request.

Line 15 Release the resources held by ACE_Message_Block after finishing the service.

3.3.4 Reusing the Jaws I/O Framework

The Jaws I/O framework can be used for synchronous and asynchronous I/O operations [Hu99, Hu00]. The synchronous I/O describes the model of I/O interaction between a server process and the kernel. In Synchronous model, the kernel does not return the thread of control to the server until the request I/O operation is done. Some operating systems provide asynchronous mode I/O. It de-multiplexes the multiple events in threads without blocking the server. When an I/O operation is initiated by the server, the kernel runs the operation asynchronously while the server could process other request. For our DES server, we only implement the synchronous application.

The structure of the I/O Framework is shown in Figure 3.10. The JAWS_IO_Handler maintains protocol and IO states and is callback driven. An I/O request is passed to its JAWS_IO_Handler, then the JAWS_IO_Handler delegates I/O requests to the JAWS_Synch_IO object.

The Jaws IO framework provides generic input and output operations. The JAWS_Synch_IO::read() method reads the request message from network to an ACE_Message_Block object from the passive mode socket. The JAWS_Synch_IO::receive_file() method uses an ACE_Filecache_Handle object to write to disk the file sent from clients.

Figure 3.10 Structure of Jaws IO Framework

The JAWS_Synch_IO::transmit_file() method performs the operation to send the required document to the peer client through network. It uses a hash table to implement a list of documents, uses the memory map to map the document in the disk to memory and send it to the network through the ACE SOCK_Stream::send_n() function. The implementation of JAWS_Synch_IO::transmit_file() as follow:

```

1 void JAWS_Synch_IO::transmit_file (const char *filename,const char *header,
2                                     int header_size)
3 {
4     ACE_Filecache_Handle handle (filename);
5     int result = handle.error ();
6     if (result == ACE_Filecache_Handle::ACE_SUCCESS)
7     {
8         ACE SOCK_Stream stream;
9         stream.set_handle (this->handle_);
10        if ((stream.send_n (header, header_size) == header_size)
11            && ((u_long) stream.send_n (handle.address (), handle.size ())
12                == handle.size ()))
13        {
14            this->handler_->transmit_file_complete ();
15        }
16        else
17            result = -1;
18    }
19    if (result != ACE_Filecache_Handle::ACE_SUCCESS)
20        this->handler_->transmit_file_error (result);
21 }

```

- Line 4** Query cache for a file with the requested file name and acquire it.
- Line 5** Check the error in the Filecache_handle creation and acquisition.
- Line 8–9** Create a ACE SOCK_Stream and bind it to the SOCK_Acceptor.

Line 10-12 Send the header file and the requested file to socket by SOCK_Stream's send_n ().

Line 14 If send file is successful, involve handler to recycle the thread.

Line 16-20 Otherwise, create an error message and recycle the thread.

The JAWS_Synch_IO::update_logfile() method opens the logfile and uses the ACE_FILE_IO::send_n() method to write the request message to the logfile. ACE provides a family of File classes that encapsulate platform mechanisms for unbuffered file operations in accordance with the Wrapper Façade Pattern. In the ACE File classes, the initialization factory classes, which are used to open or create files, are decoupled from data transfer classes that are used by applications to write and read to an opened file.

The following helper method is used by the I/O Framework to initialize a log file. An ACE_FILE_Connector object is used to open or create the file for writing.

```
1 int SERVER_Synch_IO::create_log_file (ACE_FILE_IO &logging_file,
2                                     ACE SOCK_Stream *logging_peer)
3 {
4     char filename = "logging_server.log";
5     ACE_FILE_Connector connector;
6     return connector.connect (logging_file,
7                             ACE_FILE_Addr (filename),
8                             0,
9                             ACE_Addr::sap_any,
10                            0,
11                            O_RDWR|O_CREAT | O_APPEND,
12                            ACE_DEFAULT_FILE_PERMS);
13 }
```

The `update_logfile ()` method is used to write the client's and request information to the log file.

```
1 void JAWS_Synch_IO::update_logfile (ACE_Message_Block &message_block)
2 {
3     int length;
4     char clientname[MAXHOSTNAMELEN];
5     ACE_FILE_IO log_file;
6     ACE SOCK_Stream stream;
7     stream.set_handle (this->handle ());
8     this->create_log_file (log_file, &stream);
9     ACE_INET_Addr logging_peer_addr;
10    stream->get_remote_addr logging_peer_addr;
11    logging_peer_addr.get_host_name (clientname, MAXHOSTNAMELEN);
12    length = ACE_OS::strlen (clientname);
13    log_file.send_n (clientname, length);
14    log_file.send_n (&message_block);
15    log_file.close ();
16 }
```

Line 5 Create an `ACE_FILE_IO` object to write data to a file opened by `ACE_FILE_Connector`.

Line 6-7 Create `ACE SOCK_Stream` and bind it to the `ACE SOCK_Accept`.

Line 8 Open the `log_file` for writing to.

Line 9-12 Get the remote host name through the `SOCK_Stream`.

Line 13-15 Write the remote host name and the request to the log file and close the log file.

3.3.5 Applying the ACE Virtual Filecache System

ACE implemented a cached virtual file system. A cache is a storage medium that provides more efficient retrieval than the medium on which the desired information is normally located. For ACE file caching mechanism, the cache resides in the server's main memory. A hash table holds the information about entry point into the Cached Virtual Filesystem. On creation, the cache is checked, and reference count is incremented. On destruction, the count is decremented. If the reference count is 0, the file will be removed from the cache. This file cache system avoids unnecessary file copying and uses hashing for locating files. This file cache system will decrease the overhead of the server for accessing the file system [Hu99].

ACE_Filecache_Handle is a brief abstraction over a real files and provides the entry point into the Cached Virtual Filesystem. ACE_Filecache_Handle can be used to open a file for reading or sending to network, and creating a file for writing via underlying Stream objects.

I/O operation is minor in comparison to communication delay. Applying the filecache on the server side is used mainly to improve the scalability of the server rather than to reduce the response time to clients. Using file caching will increase the server's ability to deal with more clients comparing to no filecached situations.

3.3.6 The DES_Request and DES_Response Classes

The DES_Request helper class is used to parse the request String and store the request information. It will also store the request file name, path information, etc. In accordance with the request type, the DES_Response object calls the suitable operation

to perform the required service. *GET*, *MSG*, and *PUT* represent types of requests for retrieving a document, asking for message, and uploading a document, respectively.

Figure 3.11 shows the class diagram of *DES_Request* and *DES_Response* classes.

Figure 3.11 Class Diagram of *DES_Request* and *DES_Response*

In normal response, the operation is determined by the type of request.

```

void DES_Response::normal_response (void)
{
    switch (this ->request_.type ())
    {
        case DES_Request::GET:
            .....
            this->io_.transmit_file(..);
            break;
        case DES_Request::PUT:

```

```

.....
this->io.send_put_message (...);
this->io.receive_file (.....);
break;
....
.....
default:
    this->error_response (...);
}
}

```

Figure 3.12 Retrieve Document Sequence Diagram- `Synch_Thread_Pool_Task::svc()`

Figure 3.13 Retrieve Document Sequence Diagram(cont'd) DES_Handler::read_initial()

The sequence diagram for DES server serving a Retrieve Document scenario is shown in Figure 3.12 and Figure 3.13.

In addition, the ACE Service Configurator framework was used for initializing the DES server. The ACE Service Configurator framework implements the Component Configurator pattern that enhances the system extensibility by decoupling the behavior of services from the time of the implementations of services are configured into application processes [Schmidt02b].

3.3.7 Client Applications

The client programs will send requests to the server for documents. In our performance testing clients, we will collect end-to-end performance statistical data for analysis. These end-to-end studies were conducted in a performance measurement network system so we could collect real requests and data transfers between clients and server. The client-testing program is implemented with Java. Before the request for document is sent out and after the response document is received, Java's `System.currentTimeMillis()` is used to record the time in each client program.

3.4 Summary

In this chapter, the Document Exchange Server (DES) system was developed with ACE frameworks. The design and implementation have been discussed and presented. The applications of several ACE frameworks and classes, such as `ACE_Task` framework, Jaws IO framework, ACE socket wrapper classes, and ACE virtual filecache system in the DES server system are discussed in details. Performance objectives are set up for DES system at the early design stage.

CHAPTER 4

PERFORMANCE MEASUREMENTS OF THE DES SYSTEM

4.1 Introduction

Measurements play an important role in software performance evaluation [Jain91, Lijia00]. Measurements can be used to monitor the performance of the system, to determine if the performance goals have been met, to collect data for performance models, and to verify and validate models. Measurements can also help to identify the bottleneck of a software system [Smith00]. Suitable tools are needed to conduct a measurement study. Without suitable tools, we cannot obtain useful and correct data for analyzing the performance of the system.

Measurements of similar systems may help to develop a frame of reference for estimates of resource requirements (such as CPU cycles, I/O operations, etc.) For a software system built upon reusable frameworks (components), it is expected that the performance characteristics of these reusable parts will help the developers to estimate early the performance of the whole system, before it is built and can be measured. The study of the performance of software systems built upon reusable frameworks and components can provide performance characteristics for the reusable components. A framework or component provides its usage and functionality through its interface. Since

these reusable components are designed for reuse in many similar systems, the framework developers should provide the performance characteristics of these reusable components in addition to the description of their functionality. The framework performance information is valuable since it can help programmers or system architects decide whether the use of the framework (components) for building new systems will meet the performance requirements in addition to the functional requirements. If the framework's developers (the people who know it the best) could provide its performance characteristics as well, this will help the application developers to adopt the frameworks (components) for their programs.

Performance measurements can be done at different levels: at system level (external or black-box view) or at low-level of detail (internal or white-box view). Internal event recorders require the insertion of probes into the software to collect performance data about events of interest. It is a very useful technique for collecting data for performance analysis and as input parameters for performance modeling. The measures of path execution internally will be very helpful for understanding the details of the performance of a program. External measures may mix the path data with other data and make it hard to isolate detailed information.

In this chapter, we will study the performance of the DES server system, which is built upon ACE frameworks, through measurements. The performance of this distributed and concurrent system will be investigated from both the internal and external point of view. The server's internal execution will be investigated using the Rational Quantify profiling tool on a single computer with a single user. System-level end-to-end

measurements will be conducted on a performance measurement network, in a distributed environment where multiple users compete for the same server.

4.2 Rational Quantify as a Performance-profiling Tool

Rational Quantify is a performance profiling tool for C/C++, C#, and Java applications, provided by Rational Software. Rational Quantify delivers repeatable timing data for all parts of an application, including components whose source code may not be available.

Unlike sampling-based profilers, Quantify's reports include very little overhead. Quantify uses a combination of compiler-based information and minimal Object Code Insertion (OCI) to count the instructions for a given execution, and to compute how many machine cycles are required. Unlike gprof, Quantify does not make assumptions about the average cost per function, but measures it directly.

Quantify graphically displays timing data and also provides comprehensive timing data in a tabular format. This information directs users to the areas in the application that are taking the most time to execute. Graphical performance data is displayed as a call graph, which shows what functions were invoked how many times for a particular execution of the program.

For Visual C++, Quantify instruments copies of the executable program and its associated modules. Quantify shows its progress as it instruments the files.

Quantify provides accurate measurements for code that can be compiled under the tool (such as code in the user's modules) and less accurate measurements when the source code is not available (such as system calls and the third party libraries).

For available source code, Quantify uses both compiler and run-time information to collect performance data. The compiler divides the code into a number of basic blocks. A basic block is a sequence of instructions that are executed as an atomic operation. Quantify uses the compiler's output (i.e. the number of instructions in each basic block and the number of machine cycles each instruction needs) to calculate each block's execution time. The number of times of each block's execution is collected by Quantify at run-time.

For unavailable source code (e.g. system calls and third party library), Quantify inserts a code sequence to collect the time elapsed or spent in the kernel and the code. Since the code used to collect time may add some overheads, the measurements may be less accurate.

When Quantify starts profiling, it displays the Run Summary window so that the activity of threads and fibers can be monitored, and other information about the run can be checked. As the code is executed, Quantify records data about its performance.

When the execution terminates, Quantify displays the Call Graph window, which graphically depicts the calling structure and performance of the functions, procedures, or methods in the program. By default, the call graph displays the top 20 functions in the current dataset sorted by function + descendants (F+D) time. Quantify's results include virtually no overhead from the profiling process itself. The call graph will show the graph of function calls with their descendants. The data for each function call includes

Figure 4.1 Call graph displayed by the Rational Quantify tool

the number of calls performed, the total function time either in time scale or in machine cycles, and its descendents' total time. The call graph helps users to understand the calling structure of the program and the major call paths that contributed to the total time of the run. Figure 4.1 is a display of a call Graph from the function named `Synch_Thread_Pool_Task::svc()`, which is the entry point for every thread in the thread pool of the DES server.

Figure 4.2 The Functional Detail Window for Synch_Thread_Pool_Task::svc()

The Function Detail window displays data for a specific function, and data about its callers and descendants, in both tabular and graphical formats. Figure 4.2 shows the function detail window of the function Synch_Thread_Pool_Task::svc(). If debug data was available and the user measured functions at line level, he/she can use the Annotated Source window to analyze a specific function's performance line by line.

Quantify provides several ways to reduce large datasets and display only the data a user is interested in. For example, a user can specify filters by using Filter Manager to remove functions based on module, pattern, or measurement type (for example, all

waiting and blocking functions). One can also focus on a specific subtree of the call graph.

A user can analyze the performance of the program over two runs by merging the separate runs to create a new dataset, or compare two runs with Diff Call Graph. The Diff Call Graph highlights performance improvements in green and regressions in red, to help the user pinpoint performance change. The Diff Function List displays the differences between the two runs, as well as original data from the two runs.

Figure 4.3 The Quantify Runtime set up for time method selections

Quantify uses debug line information to collect line-by-line performance data. It provides several methods to time functions: user time, user + kernel time, kernel time, and elapsed time (as shown in Figure 4.3). Quantify can give line-to-line execution time. Functions can be classified according to the modules they belong to, such as user

modules, system modules, and functions that Block or Wait. The run time options for functions in different modules have various selections, such as elapsed time, Kernel time, User time, ignore, and User time + Kernel time.

4.3 Performance Measurements and Discussions

4.3.1 Running Environment

Both the DES server and Rational Quantify are installed and run in a PIII 933 MHz workstation with 256 MB RAM memory in the Win2000 environment. The client program written in Java runs in another win32 workstation to send requests to the DES sever for retrieving documents through the local Ethernet. The server program runs in Visual C++ 6.0 environment and is started through the Quantify's run program. The port number and number of threads of the thread pool could be specified through the command-line arguments of Quantify's run program. The time scale could be in clock time or CPU cycles.

The server has an initialization phase when it receives the first call after it starts running (for starting the filecache singleton). We are not interested in the initialization phase, so we remove its measurement with Quantify's Compare Run feature. Normally, we average the server execution time of each run over 100 samples (requests) and each selected activity execution time is the average of 10 runs.

The total execution time of the server is collected by setting the timing method in Run time setting as follow: for functions in the user modules choose user time; for functions in system modules choose user + kernel time; for functions that block or wait

choose elapsed time. Pure CPU execution time can be collected, as well, by ignoring blocking and waiting time.

4.3.2 The DES Server's Total Execution Time

In the multithreaded DES system, we applied several ACE frameworks, such as ACE Task framework, ACE Service Configurator framework, and the Jaws I/O framework.

The ACE Service Configurator framework is only involved in the server's initialization and is not involved in the server's services for clients' requests, so it will not affect the server's overall performance. The ACE Task concurrency framework provides a facility for spawning threads in the context of an object and a queuing mechanism for transferring messages between tasks. It adds some overheads in maintaining the thread pool and has little influence on the server's performance. These two frameworks mainly provide mechanisms to facilitate programming with more flexibility and ease of use. They were not included in the key performance scenarios of the DES server. We will focus only on frameworks involved in key performance use cases, such as the reusable Jaws I/O framework, which provides common IO services for the networking application programs. It can be reused in many similar networked systems that require I/O services through sockets.

Table 4.1 The DES server's Execution time for non-cached documents at a 933 MHz Pentium workstation

Document size (KB)	Total, Phase I (ms)	CPU (ms)	I/O (ms)	Phase II (ms)
5	5.27	2.44	2.83	0.16
50	6.42	3.59	2.84	0.17
100	8.48	5.43	3.05	0.17
250	9.82	6.44	3.38	0.18
500	12.34	8.41	3.92	0.19
750	13.80	10.12	3.68	0.18
1000	15.63	12.05	3.58	0.18
1500	19.35	15.15	4.20	0.18
2000	23.90	18.21	5.68	0.17

Table 4.2 The DES server's Execution time for cached documents at a 933 MHz Pentium workstation

Document size (KB)	Total, phase I (ms)	CPU (ms)	I/O (ms)	Phase II (ms)
5	2.79	0.63	2.12	0.15
50	3.47	0.86	2.61	0.17
100	3.64	1.17	2.47	0.19
250	5.32	2.19	3.13	0.20
500	6.92	3.61	3.31	0.17
750	8.21	4.71	3.50	0.15
1000	9.84	5.99	3.85	0.21
1500	12.65	8.05	4.60	0.20
2000	14.65	10.30	4.35	0.17

Since the ACE_Filecache implements file caching, the server execution time for cached or non-cached files will be different. The ACE_Filecache implements the singleton pattern to have only one cache instance. When the file is requested for the first time, it is not cached. The non-cached file is fetched and read from disk before is sent to

the clients through the socket; meanwhile the file reference is inserted into the hash table and is cached. If the file is cached, its reference is in the hash table; searching for this file will be much faster. Figure 4.4 shows that the total DES server's execution time of a cached file is much less for a non-cached file than that of a non-cached file in the Retrieve Document Use Case. It could save about 30 ~ 50 % of server's execution time for a cached file in comparison to a non-cached file of the same size. The server's capability to handle more requests is increased by reducing the service time for each request.

Figure 4.4 The server's execution time at a 933 MHz workstation for the Retrieve Document Use Case

The server execution time consists of two parts: phase I is the time for serving the clients' request; Phase II is the time for cleaning-up the working thread and recycling it back into the available thread pool. Table 4-1 and Table 4-2 show the total DES server's execution time and CPU time for requesting non-cached and cached files of

various sizes. Here the CPU time refers to the CPU active time, and I/O time refers to the time that the thread blocks or waits for I/O operations. The I/O time may also include the time that the thread blocks or waits for entering critical sections. Since Quantify does not provide a method to directly measure the I/O operation time, the I/O time obtained here may not be very accurate.

Figure 4.5 Response time vs. document size for cached and non-cached documents in the Retrieve Document Use Case

File caching on server side is mainly used to improve the server's capacity. To a single client, it may not feel much of an improvement in response time, since a large percentage of the total response time is the packets traffic time through network, especially for large files. Figure 4.5 shows the response time for non-cached and cached documents versus documents size for a single user (no contention).

Table 4.3 and Table 4.4 summarize the DES sever's execution time in CPU cycles for all activities in the activity diagram for retrieve document use case described in Figure 3.4. The total time and CPU time for activities such as *accept*, *read*, *update_logfile*, *parse_request*, and *thread recycling*, are similar for both cached and non-cached files, and independent on the document size. This is due to the fact that the clients' request strings are of similar sizes and processing them takes similar CPU and I/O time. Note that there are some variability in the measurements for these activities given in Table 4.3 and 4.4 due to the way in which Qantify measures the third-party code (see section 4.2).

Significant differences in the execution time for different file sizes are observed for the *send_file* activity, which is mainly performed by the *JAWS_Synch_IO::transmit_file()* operation. Both CPU time and total time increases as the document size increases for both non-cached and cached file. For the same file size, a non-cached file will take more time to process than a cached file. In the next section, we will look closely at the reusable function provided by the IO framework.

The total execution time of the server to serve one request can be calculated by summing of all the activities in one thread service cycle:

$$t_{\text{total}} = t_{\text{accept}} + t_{\text{read, total}} + t_{\text{parse}} + t_{\text{update, total}} + t_{\text{send, total}} + t_{\text{recycle}}$$

The CPU time can be calculated in a similar way:

$$t_{\text{cpu}} = t_{\text{accept}} + t_{\text{read, cpu}} + t_{\text{parse}} + t_{\text{update, cpu}} + t_{\text{send, cpu}} + t_{\text{recycle}}$$

Table 4.3 The DES Server's execution time for retrieving non-cached files (in $\times 10^3$ CPU cycles)

document size (KB)	accept	read		update_logfile		parse_request	send_file		recycle
		total	cpu	total	cpu		total	cpu	
5	201	1,436	125	789	31	28	2,474	1,900	153
50	208	1,278	124	822	32	27	3,671	2,963	157
100	216	1,294	125	894	32	27	5,500	4,676	163
250	189	1,491	127	662	32	27	6,814	5,646	169
500	224	1,890	129	679	33	26	8,715	7,453	174
750	193	1,411	127	839	32	27	10,434	9,083	172
1000	229	1,315	129	696	32	26	12,351	10,852	166
1500	233	1,208	131	699	38	27	15,924	13,734	168
2000	232	1,431	132	898	32	27	19,755	16,606	159

Table 4.4 The DES Server's execution time for retrieving cached files (in $\times 10^3$ CPU cycles)

document size (KB)	accept	read		update_logfile		parse_request	send_file		recycle
		total	cpu	total	cpu		total	cpu	
5	189	1,297	125	605	32	26	495	213	137
50	200	1,151	136	611	32	25	759	413	160
100	202	1,184	129	626	33	25	1,368	707	173
250	219	1,715	126	623	32	25	2,393	1,650	184
500	235	1,416	134	680	33	26	4,114	2,944	161
750	229	1,170	128	677	30	27	5,578	3,990	144
1000	238	1,237	128	686	33	26	7,018	5,180	196
1500	201	1,551	129	629	34	27	9,423	7,132	187
2000	240	1,013	131	663	33	26	11,754	9,203	157

The execution time in second can be obtained as $t = N_{\text{cpu}}/S$, where N_{cpu} is the number of CPU cycles for the operation, and S is the processor speed in Hz. We assume that the difference between the total time (elapsed time) and CPU time gives the time for input and output operations. The input and output operation time: $t_{\text{i/o}} = t_{\text{total}} - t_{\text{cpu}}$.

4.3.3 The Execution Time for transmit_file() of the IO Framework

The activity of the DES server to get the requested document and to send it back to the client through socket is mainly performed by `JAWS_Synch_IO::transmit_file()` from the IO framework. Using the requested file's name and a header string as parameters, this method fetches the file from the filecache singleton and transmits it through socket to the client. The sequence diagram for the `DES_Response::normaml_response()` which uses the `JAWS_Synch_IO::transmit_file()` is shown in Figure 4.6.

Within this method, after the request file is found and opened for read, `ACE_SOCKET_Stream::send_n()` is called to send the file to the socket. There are many reasons that can lead to a network communication failure. If the `send_n()` failed, a `transmit_file_error` message will be sent to `JAWS_IO_Handler` and the thread will be recycled and returned to the available thread pool. If the operation succeeds, a `transmit_complete` message will be sent to `JAWS_IO_Handler` and the thread will be recycled to thread pool.

Figure 4.6 The sequence diagram for `DES_Response::normal_response()` for Retrieving Document

(a)

(b)

Figure 4.7 (a) The total execution time and (b) the CPU time for JAWS_Synch_IO::transmit_file() of the Jaws IO framework

Figure 4.7 (a) shows the total time in millisecond for JAWS_Synch_IO::transmit_file() versus file sizes in kilobytes. The server is running on a 933 MHz workstation. Figure 4.7 (b) shows this function's CPU execution time measured in CPU cycles for transmit_file() versus file size in kilobytes. According to Figure 4.7, we can see that both of the total time and CPU execution time increase as the file size increases. A non-cached files takes more time to process than a cached file. Table 4.5 summarizes the CPU execution time (in CPU cycle) and IO time (in ms) for JAWS_Synch_IO::transmit_file() for various document sizes. The performance data is not dependent on any specific application, so it could be reused in any program where IO framework is applied. The CPU execution time can be converted into clock time by dividing it to the processor's speed. This example demonstrates that not only the framework functionality can be reused, but also some of its performance information.

Table 4.5 The CPU execution time and IO time for JAWS_Synch_IO::transmit_file()

Document Size (KB)	Cached		Non-cached	
	CPU (cpu cycle)	I/O (ms)	CPU (cpu cycle)	I/O (ms)
5	213,000	0.30	1,900,000	0.61
50	413,000	0.37	2,963,000	0.76
100	707,000	0.70	4,676,000	0.88
250	1,650,000	0.80	5,646,000	1.25
500	2,944,000	1.25	7,453,000	1.35
750	3,990,000	1.70	9,083,000	1.44
1000	5,180,000	1.97	10,852,000	1.60
1500	7,132,000	2.45	13,734,000	2.34
2000	9,203,000	2.73	16,606,000	3.37

4.4 Performance Measurement in a Performance Network

The execution time of different code segments is important, but is not enough to understand the overall performance of the system. Also, the measurements with a single client are not enough to understand the overall performance of the system. The DES server is designed for multiple clients asking for services concurrently. The server will behave differently when serving multiple concurrent requests. To understand well the server's performance, it is necessary to measure it under different workloads (i.e., with a different number of competing clients).

4.4.1 Performance Network Set-up

The end-to-end performance measurements were conducted on a measurement network, named "alpha network", in the Real-time and Distributed System (RADS) laboratory of the Department of Systems and Computer Engineering.

The alpha measurement network is composed of 14 Dell 266 MHz workstations with 128 MB RAM, and is running under the NT 4.0 environment. The set-up of this measurement network is shown in Figure 4.8. All workstations are connected through a 100Mbps Ethernet hub. One of these workstations is used as the server where ACE was installed. Other workstations are used as clients. Documents of various sizes, ranging from 5 KB to 2,000 KB, are stored on the server machine. In an experiment, the clients send requests for files of the same size, but with different names (to avoid caching). One of the clients records the response times while other clients concurrently request services. The average response time for one sample is calculated over 10,000 requests.

Figure 4.8 The set-up of the alpha measurement network

4.4.2 Response Time Distribution

Figure 4.9 shows the distribution of response time of 1000 requests in chronological order for 50 kilobytes documents with: (a) one client, and (b) ten concurrent clients on the alpha network. The size of the thread pool of the server is 5. At a higher workload, the response time increases due to contention and the distribution of the response time becomes more uneven.

Figure 4.10 (a) shows the same response time distribution as Figure 4.9 (b) but sorted in increasing order. The average response time for 1000 requests is 47.6 m and more than 95% requests are responded within 101 ms. Figure 4.10 (b) is the histogram of the response time for 1000 requests at the workload of 10 concurrent clients.

(a)

(b)

Figure 4.9 The response time distribution for (a) one client (b) 10 concurrent clients in the alpha network. The size of the thread pool of the DES server is 5 and the requested document size is 50 KB

(a)

(b)

Figure 4.10 (a) The distribution of response time in increasing order; (b) The Histogram of response time for Retrieve Documents scenario for 10 concurrent clients request 50 KB documents at the alpha network

4.4.3 The Average Response Time and Throughput

The average response time for various levels of contention is studied. Figure 4.11(a) shows the response time vs. number of clients for 5KB, 50 KB, 250 KB, and 500 KB documents. The response time increased as the contention increases.

Figure 4.11(b) shows the throughput of the 5-thread DES server versus the number of the clients. According to Figure 4.11, the throughput increases as the number of clients increases at first; then after certain point, the throughput tends to be unchanged as more concurrent clients added. This is due to some of the resources (server or network) are saturated.

Table 4.6 Response times (in ms) for retrieving 5 KB, 50 KB, 250 KB, and 500 KB documents in the alpha network

No. of clients	5KB	50KB	250KB	500KB
1	9	13.1	38.9	72.7
2	12	15.1	44.6	83.2
3	19	19.8	50.1	97.3
4	21	24	60.7	112.2
5	25	28.7	72.5	142.6
6	27	35.3	87.6	160.4
7	33	39.3	108	180
8	29	46.2	113	234
9	39	51.3	137	262
10	36	52.7	157	282
11	49	70.4	173.7	325
12	53	71.5	198	368
13	/	88.6	209	424
14	/	94.1	218	435
15	/	98.6	243	503

(a)

(b)

Figure 4.11 (a) Response time and (b) Throughput vs. number of clients for a 5-thread DES server at the alpha network

4.5 Summary

We have presented the performance measurements of the DES system for cases without and with contention. The DES server's execution times for each activity in Retrieve Document scenario, both for cached and non-cached files, were extracted from the Quantify measurement results. The details of Jaws IO framework's `transmit_file()` execution times are studied versus the document size. Since this performance data is independent on the application, it could be documented with the framework as performance characteristics of the framework. This study provides an approach to extend the reusability of frameworks to the performance domain. The server performance for different numbers of clients and various document sizes was investigated in the measurement network.

The server execution data will be used as parameters for UML profile to document the software performance, and then to build a performance model of the system. The computed results will then compared with the measurement results from the measurement network.

CHAPTER 5

LQN MODEL OF THE DES SYSTEM

5.1 Introduction

As mentioned before, the UML Performance Profile provides facilities for (1) capturing performance requirements for the software system; (2) associating performance-related QoS characteristics to elements from the UML model; (3) specifying *execution parameters* which can be used by modeling tools to compute predicated performance characteristics; and (4) presenting the measured or computed results [OMG02]. In section 5.2, we will apply the UML profile to DES system's Retrieve Document scenario by using the performance values obtained with Rational Quantify in the last chapter.

Measurements are important in a performance study [Smith01]. However, measurements alone are not enough, and they have many limitations.

- Measurements are only feasible in a small range. Internal measures need suitable and powerful tools that can measure the demands of cpu, I/O, and logical resources. External measures need a suitable hardware set-up, so that we can fully explore the software performance and its limitations.

- Concurrency also adds complexity to the performance measurements. Concurrency generates contentions for both physical and software resources. Waiting to access the resources could take a significant portion of the total response time under heavy workloads. As the concurrency level increases, contention for the system's resources would dominate the response time of a process.
- Performance for various workloads may be very time consuming. In today's highly competitive market, there is no time for such thorough tests.
- In some cases, end-to-end performance measurement is not possible. In these cases, performance models are the only options for studying the performance.

Performance models are very useful to evaluate the overall performance of software systems during software analysis and design stages. Modeling could extend to explore those scenarios that are difficult or impossible to measure directly. Moreover, modeling could provide some guidelines on how to achieve higher performance, even if there is no real system implemented yet.

In section 5.3, we will build a LQN model for the Retrieve Document scenario from the UML model and compute the performance of the system at various workloads. The LQN results will be compared with the measurements.

5.2 Applying UML Profile to the DES System

We can apply the UML Performance Profile to add performance annotations to a UML model of the system. We will use the performance data obtained in the last chapter.

5.2.1 Annotations for Resources

Hardware resources are modeled by the stereotypes <<PAhost>> or <<PAresource>>. Hardware resources are represented as nodes in UML deployment diagrams.

A processor is modeled by the stereotype <<PAhost>>, which has associated tagged values that define its scheduling policy, processing rate, context switching time and performance measures, such as utilization and throughput.

Other hardware devices are modeled as <<PAresource>>. The associated tagged values define their capacity, scheduling policy, time to be acquired/released, and performance measures such as utilization, throughput, response time and waiting time.

Software resources are modeled by the stereotype <<PAresource>>. Software resources can be modeled as UML active or passive objects participating in the scenarios. An active object has its own execution thread of control, and models resources such as operating system processes and threads. While objects are used to represent resources, the class operations represent services offered by those resources.

Figure 5.1 shows the deployment diagram of Document Exchange Server (DES) System. It has one application processor for the server and multiple clients. Documents

are stored at the server's disk. The server and Clients are connected through a local area network (LAN).

Figure 5.1 The deployment of the Document Exchange System

5.2.2 Annotated UML Scenario Model

UML diagrams that describe behaviors include sequence diagram, collaboration diagram, state diagram, and activity diagram. The scenarios for performance modeling can be represented either by a sequence diagram or by a hierarchical activity diagram. The Activity Diagram is the closest to the Scenario stereotype from the Performance Profile, in that it explicitly shows alternatives, parallel branches and refinement of an activity by a sub-diagram.

The activity diagram describes the sequencing of activities, with support for both conditional and parallel behavior. An Activity Diagram can use swimlanes to separate each participating component or class. Each activity in the activity diagram is stereotyped as <<PAstep>>. Figure 5.2 shows an activity diagram for the Retrieve Document scenario.

Figure 5.2 Activity Diagram with performance annotation for Retrieve Document scenario

The tagged value `PArespTime` defines the response time of a resource - the total time to complete a service, including the waiting for the resource and its nested resources. `PArespTime` is a measure of the Quality of Service (QoS). `PArespTime` may represent a requirement on the resource, an assumed value, or a result computed by the performance model. In general, the achieved QoS depends on the load placed on the resource.

The quantifier string `'req'`, `'asmd'`, `'pred'`, and `'msrd'` represent required, assumed, predicated, and measured, respectively.

The `PAdemand` tagged value indicates mean measured execution times on the host processor for the associated scenario steps. For example, in the activity of *accept request*, the tagged value:

```
PAdemand = ('msrd', 'mean', (220/$cpuS, 'ms'))
```

indicates that the mean measured value for CPU demand is given by the specified expression in milliseconds.

In the UML performance profile, variables begin with '\$'. The variable `$cpuS` is the host processor speed in MHz. The following variables are file caching mechanism and document size dependent:

`$gcdC` = CPU overhead for getting a document from the disk

`$scdC` = CPU overhead for sending a document to the network

The following variables are application related:

`$RP` = the size of a request message in data packets

`$DocP` = the size of a document in data packets

$\$DocS$ = the average size in kilobytes of a document in physical disk blocks

The numbers of CPU cycles in the annotated activity diagram from Figure 5.2 are obtained from the Quantify measurements from section 4.3. The execution times for activities such as *accept request*, *read request*, *update logfile*, *parse request*, and *recycle thread* are independent on the sizes of requested documents. They are only dependent on the platform and processor speed. The *get document* activity and *send document* activity are determined by the size of document as well as platform and processor speed.

For the Retrieve Document scenario in a win32 system, we assume $\$RP$ as 1 since the request message is almost of the same length. Variables $\$DocS$, $\$scdC$, and $\$cdS$ are document size dependent and can be found in tables 4-4 and 4-5.

DiskProc reads documents from the disk through I/O operations. The I/O operation time depends on the document size. The tagged value

$$PAextOp = ('readDisk', \$DocS)$$

which is attached to the activity *read from disk* indicates the name of the external operation and the number of invocations made in that step. The scenario steps that send message or documents over the network will invoke external operations.

5.3 Performance Analysis of the DES System with LQN

5.3.1 The LQN Model for the Retrieve Document Scenario of the DES Server

1. The LQN Model

Figure 5.3 shows the LQN model for the Retrieve Document scenario of the DES server. The LQN model is constructed manually from the UML model from the previous section. Each software component in Figure 5.2, each processor node, and each network delay are modeled as a corresponding LQN task. The net1 and net2 entries are used to model communication delays over the Ethernet. This is a simplified model since some logical resources (critical sections), which are encapsulated within the ACE wrappers or frameworks, are not included. The programming complexity of frameworks and the limitations of Quantify make the identification of these logical resources a difficult job.

Figure 5.3 An LQN model for Retrieve Document scenario

We also use a simple and coarse granularity disk I/O model by aggregating all I/O operations into one single I/O operation. In this model, the DES server task is

represented as a LQN multi-server. By definition, a LQN multi-server has more than one server copy that serves request from the same queue. The same happens with the threads from a thread pool. The threads in a thread pool are dispatched to serve requests that arrive and are queued at a single port (i.e. the socket port). The network contention in the LQN model is introduced in this model.

The data stream moving in the LQN model is shown in Figure 5.4. The number of calls from server to the net2 is determined by the requested document size and the Ethernet packet size. Figure 5.4 represents an LQN model equivalent with that from Figure 5.3, where the forwarding arcs have been replaced with synchronous requests. This transformation is automatically performed by the LQN solver.

Figure 5.4 LQN model equivalent with Figure 5.3

2. Frameworks Contribution to the DES Performance Model

The thread pool in the DES server is maintained by the ACE Task framework. The resource demands of each thread consist of two parts: V , overhead for dispatching a

thread (V includes context switching, etc.), and W , work done by the thread in the application code. The overall performance of the server will be determined by $(V + W)$. The framework's entities that have influence on the performance of the system would contribute to either V or W , or to both of them.

There are different ways the framework contributes to creating the applications and its performance model.

(a) Architectural features contribute to the overheads V : The architecture of the framework determines the overall architecture of the system. For example, we use the ACE Task framework to facilitate the thread creation and management in the DES server. The thread pool is created at the initialization stage (so the ACE Task thread creation method was not included in the key performance scenario of the server). The selection and dispatching of a thread, and the recycling of the thread back to the pool in each service cycle by ACE Task Framework will add some overheads to V , which is hidden by the framework.

(b) The application specific activities provided by framework methods and application code are included in the application work and contribute to W : an example is in the Retrieve Document scenario, the IO framework's `JAWS_Synch_IO::transmit_file()` will have large impacts on the performance of the DES server.

3. Network submodel

The approach for modeling the Ethernet network in the LQN model from Figure 5.4 model is taken from [Lazowska84]. This approach takes into account the contention for the network by different nodes, and is suitable for systems that are communication-

bound. The parameters for n workstations sharing an Ethernet channel are given as follows:

B = network bandwidth is in bits per second (bps). For an Ethernet with large packets, only 83% of overall bandwidth is useful.

S = slot duration [i.e., the round-trip time propagation time of the channel (time required for a collision to be detected by all stations)], related to the length of the network. S is a parameter of the configuration related to the length of the network. Here we choose $51.2 \mu s$ [Menasce94, Franks00].

P = average packet length, 1518×8 bits.

B/P = the maximum theoretical capacity in packets per second of the Ethernet

$C(n)$ = the average number of slots devoted to contention before a successful acquisition by some station. $C(n) = \frac{1 - A(n)}{A(n)}$, where $A(n)$ is the probability that any of the n stations successfully acquires the channel

during a particular slot: $A(n) = \left[1 - \frac{1}{n}\right]^{n-1}$

$E(n)$ = the proportion of the capacity that is devoted to the useful work when there are n stations desiring to transmit packets and can be expressed as:

$$E(n) = \frac{P/B}{P/B + C(n) \times S}$$

For n stations that share an Ethernet channel, the rate at which the Ethernet delivers packets is given by equation (5.1) [Lazowska84]:

$$\mu(n) = \frac{B}{P} \times E(n) \quad (5.1)$$

We use the average packet size P as 1518 byte (where 1500 byte is data) and the request from the client to the server is one packet. The number of packets from server to the client is given by $\lceil \$DocS/P \rceil$ (rounded to whole number). For a 5 kilobyte document, the number of packets is 4; and for a 50 kilobyte document, the number of packets is 34. The service times for the LQN phases and activities are given in Table 5.1. Table 5.2 shows the LQN number of requests.

Table 5.1 LQN service times

Task, entry, phase or activity	Resource demands (ms)
clientT, default entry	0.5
clientT, phase2	1.5
retrieveThread, phase1	$1.54 + (\$gcdC + \$scdC)/266$
retrieveThread, phase2	0.60
disk	$1.90 + \$cdS$
net1	0.22
net2	$0.22 * \lceil \$DocS/P \rceil$

Table 5.2 LQN number of requests

From	To	Number o of request
clientT	net1	1
	net2	$\lceil \$DocS/P \rceil$
	eS	1
eS	disk	1

We will use the simple LQN model from Figure 5.4 to compute the performance of the DES system for the retrieval of cached documents of 5 kilobyte and 50 kilobyte. We will investigate the sensitivity of the Ethernet service time parameters and the I/O operation times for these two different sizes. We will also study the performance for different numbers of threads, with/without server file caching on the server side.

5.3.2 Comparing the Results from Measurements and Modeling

The resource demands at the server side (such as CPU time I/O time) and the network demands will determine the performance of the system. The resource demands parameters used in a LQN model will affect the modeling accuracy. As performance evaluation is instance-based, the influence may vary for different use cases (scenarios). If a scenario is network dominated (the bottleneck is the network), then network related parameters and the model for the network would have a great effect on the results. On the other hand, if a scenario is server dominated (the bottleneck is the server), the performance results of the model will depend on the server's model and the server's execution parameters.

1. Response time sensitivity to the I/O parameters

It was mentioned before that Quantify has good accuracies for CPU measurements, but it does not provide much help for measuring I/O times. We had to infer the I/O times in chapter 4 as the difference between elapsed time and CPU time. To explore how the measurement accuracy of the I/O time impacts the performance model results for different scenarios, we computed the response times by using different I/O

times for the retrieval of cached 5 kilobyte and 50 kilobyte documents. Table 5.3 shows the I/O parameters used for the LQN model to compute the response time. These times are 0% which is the Quantify measured values and 100% increment of the Quantify measured values.

Figure 5.5 shows the modeling results for the retrieval of cached (a) 5 kilobyte and (b) 50 kilobyte documents from the DES sever and the end-to-end measurements from the alpha network. For the 5 kilobyte case, the I/O time has a large influence on the modeling results. In this case, the server's performance dominates the overall performance since the small documents travel time on the network is a relative small portion. The server performance is influenced, in turn, by the I/O time. The total service time of the server has a large impact on the modeling results.

The situation is different for larger documents of 50 kilobyte, as seen in Figure 5.5(b). A good part of the response time is from the network, so the server's I/O operation parameters have a small impact on the model results. The network utilization from the LQN model shows that the network is the bottleneck for the retrieval of 50 kilobyte documents.

Table 5.3 I/O parameters for LQN model for retrieval of cached 5 KB and 50 KB documents

increment	for 5 KB (in ms)	for 50 KB (in ms)
0%	2.12	2.61
100%	4.24	5.22

(a) 5 KB document

(b) 50 KB document

Figure 5.5 Response time sensitivity to I/O parameters for retrieval of (a) 5 KB and (b) 50 KB documents

2. Response Time Sensitivity to Ethernet Packet Service Time

The system performance is complex and it's the result of the interplay of many hardware and software components [Manesce02a]. To evaluate the performance of a software system, we have to consider the hardware set-up or availability. In some cases, the limitations of the hardware measurement could retard the evaluation of the performance of the software system. For the DES server's Retrieve Documents scenario, if the network is the performance bottleneck, then the overall performance of the system will be mostly determined by the network delays and the server characteristics will have a smaller influence.

In the DES system, we need to model accurately both the network and the server in order to assess the overall the performance of the system. [Lazowska84] provides an approach for modeling the Ethernet. However, the service rate for an Ethernet packet will have different effects on different systems and different scenarios.

The theoretical service time for one 1518 byte packet is given by $P/B = 1518 \times 8 / 100 \times 10^6 = 0.1214$ ms. If there are two workstations, then the service time for a 1518 byte packet over 100 Mbps Ethernet is 0.1726 ms. According to Equation 5.1 and considering the Ethernet bandwidth's efficiency of 83% [Franks00], the service time for one of 12 workstations in the alpha network should be 0.2282 ms, and for one of 15 workstations should be 0.2296 ms. We choose the service time as 0.1214 ms and 0.2248 ms for 5 KB case, and 0.1726 ms and 0.2296 ms for 50 KB case to investigate the impact of the Ethernet packet service time on the overall performance for retrieval of cached documents.

(a) 5 KB documents

(b) 50 KB documents

Figure 5.6 Sensitivity of Ethernet packet service time on LQN modeling for retrieval of cached (a) 5 KB and (b) 50 KB documents at the alpha network

Figure 5.6 shows the LQN modeling results for various Ethernet packet service time parameters for retrieval of (a) 5 kilobyte and (b) 50 kilobyte documents. For the 5 kilobyte case (Figure 5.6(a)), the LQN modeling results are insensitive to the Ethernet packet service time from 0.1214 ms to 0.2248 ms. In contrast, the performance of the retrieval of 50 kilobyte documents varies greatly with the Ethernet packet service times from 0.1726 ms to 0.2296 ms (Figure 5.6(b)). For this case, an accurate model of the Ethernet and a suitable Ethernet service parameter are more important to predict the system performance than the server's execution parameters. By using the LQN model in Figure 5.4, the modeling results would match pretty well with the measurements by choosing Ethernet packet service time as 0.1726 ms.

The discrepancy between the modeling and the measurement can be caused by the following reasons: (1) inaccurate models which did not capture all of the most important performance features of the system, such as inaccurate representation of the architecture, or missing important logical resources; (2) inaccurate parameters used for LQN, either due to measurement errors or to inappropriate tools.

5.3.3 The Impact of the Server's Multithreading on Performance

The CPU utilization of the server is dependent on the use cases and on the workload parameters. Multithreaded server would benefit from the improvement in the server's CPU utilization for cases with higher contention for the server's resources such as CPU and disk. If a program is CPU-bound, multithreading would not help, but worsen the performance by adding some overheads for maintaining the threads. When the number of threads reaches a certain number, the CPU resource will reach its highest

utilization available; thus, adding more threads may not improve the performance any further. Therefore, the numbers of the threads in the thread pool of the DES server can be adjusted in accordance with the workload.

Figure 5.7 The performance modeling for various number of threads in the DES server for retrieval of cached 5 KB documents

Table 5.4 The DES server's CPU utilization vs. number of threads for the retrieval of cached 5 KB documents on the alpha network

no. of threads	1	3	5	7	9	11	13	15	17	19	21
server CPU Utilization	0.569	0.831	0.914	0.951	0.969	0.98	0.986	0.989	0.992	0.993	0.995

Figure 5.7 shows that the performance of the DES server for retrieval of cached documents of 5 kilobyte improves as the number of the threads in the thread pool grows from 1 to 20. However, as shown in Figure 5.8 and Table 5.4, when the number of

threads is larger than 15, increasing the threads in the thread pool does not bring any additional benefits.

Figure 5.8 The DES server's CPU utilization vs. number of threads in the thread pool for the retrieval of cached 5 KB documents

5.3.4 Caching versus Non-caching

One of the advantages of performance modeling is that it can be used to investigate some situations that cannot be directly measured. The DES server adopted the file caching strategy to improve the servers' capacity. The end-to-end measurement in Figure 4.6 shows that there is no obvious improvement in response time for cached and non-cached files for a single client. We could not conduct an end-to-end measurement of the non-cached file performance at high workload. This is due to the fact that, once a file is requested, it will be cached. The following requests for this file will only need a shorter time due to the use of the ACE virtual filecache system.

Performance models provide a way to study how much performance improvement could be done by using file-caching strategy at high contention situations.

Figure 5.9 LQN modeling results for the retrieval of 5 KB documents with/without server file-caching strategy

For this case, we assume that the requested document size is 5 kilobyte and the DES server runs at a Pentium workstation with processor speed at 933 MHz. The parameters in Figure 5.2 were converted into millisecond from CPU cycles by dividing the processor speed in MHz. The total I/O time for non-cached and cached files can be found in Table 4.1 and Table 4.2, respectively.

Figure 5.9 shows the LQN modeling results for the retrieval of 5 kilobyte documents with/without file-caching strategy for various concurrent clients. This model results shows that the use of the cached virtual file technique can improve the server's performance under high contention.

5.4 Reusable Performance Information of Frameworks

5.4.1 Frameworks Performance Characteristics in the DES Server

Frameworks are “semi-complete” applications that embody domain-specific object structures and functionality. The performance characteristics of the frameworks affect the system’s performance via the functionality they provided in the programs, such as multithreading, networking, and input/output operations.

- Multithreading: the multithreading in the DES server is handled by the ACE Task Framework. The CPU demands for multithreading include thread creation and overheads.
 - Thread creation. Since the thread pool is pre-spawned in the DES sever, the thread creation is not a part of the key performance scenario of the DES server.
 - Overheads. There are two kinds of overheads for multithreading here: overheads for recycling threads and overheads for dispatching a request event. This will add some overheads in the performance scenarios.
- Networking: many modeling characteristics of the network come from the hardware platform, not from the framework. For example, the packet size and parameters for the Ethernet model are determined by the platform. On the other hand, the message size is determined by the application and is passed to the framework as a parameter.
- I/O: The I/O operations in the DES server are handled by the Jaws I/O framework. The caching used within the framework is important for the server’s

performance. The CPU and I/O time for cached and non-cached files of various sizes can be reused. There are several factors related to the performance modeling of a detailed I/O system:

- physical block size, which is platform dependent
- physical I/O service time, which is also platform dependent
- cache size, determined by frameworks and OS
- usage patterns, which is application dependent

These parameters may have an impact on the framework performance. If this is the case, they should be taken into account when documenting the performance characteristics of the framework.

5.4.2 Understanding the Structure and the Flow of Control in Frameworks

Performance analysis is based on instances. To measure and model the system correctly, a clear understanding of the structure and the flow of control of the application is an essential. The resource demands of an application include CPU, network, I/O devices, and logical resources (critical sections, locks, mutexes, buffers, etc.) and can be represented as a LQN model shown in Figure 5.10. Logical resources are often hidden behind the interface or methods, especially for programs built with frameworks or components. To predict the performance correctly, performance analysis should capture these resource demands.

Software systems using frameworks can be hard to follow because the framework's "inversion of control" will oscillate the flow of control between the application-independent framework infrastructure and the application-specific methods

callbacks. This increases the difficulty to measure the performance scenarios since the control flow of the system is driven implicitly. Some techniques that facilitate the trace of the flow of the application, such as Use Case Map (UCM) [Buhr96, UCMweb], will help leverage the performance study of systems built upon frameworks.

Figure 5.10 An LQN Model of resource demands in a multithreaded application

5.4.3 *Studying the Performance of Frameworks at a Right Level of Abstraction*

Since ACE is aimed to facilitate network programming, it uses wrappers, patterns and frameworks to hide the excessive details of the platform and native OS APIs. The usage of wrappers, patterns, and frameworks in ACE makes the programming much easier and the programmers do not need to worry about the low-level details. However, software performance is determined by the demands on the physical and logical resources. Some execution details, such as critical sections that must be synchronized for multithreads, are needed for accurate performance evaluation. The omission of such

information from performance measurements will lead to a deviation in performance prediction.

The interface for a framework is the right level for reuse of the frameworks' provided functionality. However, the interface level may not be the right level for performance measurements and modeling. When studying the performance of the software built upon reusable frameworks, we need some details at levels lower than interfaces. Measurements should capture both hardware and software resource demands, such as CPU, I/O time, network time, and critical section time, because these parameters determine the performance of the system. These parameters are needed to build an accurate performance model for evaluating the software performance.

Reusable performance information of frameworks can provide different levels of details. How much framework performance information is needed for modeling the system built with them? Neither too much, nor too little is suitable. Studying the performance at a right level is crucial.

(1) Very detailed performance information for a framework.

The performance of a framework can be very detailed. For example, in a white-box framework, resource demands for every method defined in the framework could be included. There may be too many methods in the framework and it is difficult to measure all of them. Moreover, it would be difficult for the developers to use the information. The problem for this level is that too many details may be unnecessary and could only complicate the study.

(2) The performance information associated directly with APIs defined in the frameworks and the methods that are called directly in the applications. This level of information could be directly reused and documented.

(3) The performance information only at the interface level, such as total CPU and I/O times. The problem for this level is that some logical resource contention within the interface could be hidden and missed.

To make the performance information of frameworks reusable, the information should be associated with the frameworks' usage as well as the structure of the frameworks.

Framework developers could provide valuable low-level information to help performance analysis. With some UML diagrams such as activity diagram or sequence diagram, which are used to describe the behavior of the framework functionality, they can provide information to help to identify physical and logical resources used. The best way is to provide quantitative resource demands such as CPU in machine cycles, number of call to network functions and the message size used, and number of logical I/O operations, etc.

5.5 Summary

In this chapter, UML performance profile was applied to the DES system. The measured server's execution data was used as input parameters in the profile. A LQN performance model was built and used to assess the performance of the DES Retrieve Document scenario. The sensitivity of the model results to the I/O time and network

modeling has been discussed. The performance improvement by multithreading and file caching was also investigated. The end-to-end measurement results and LQN modeling results were compared. The causes for the discrepancy between the two methods were discussed. The right abstraction level for providing reusable performance information for the frameworks was discussed, too.

CHAPTER 6

SUMMARY AND CONCLUSIONS

6.1 Summary

This thesis presented the development and performance studies of a distributed and concurrent multithreaded case study system built upon ACE frameworks.

The Document Exchange Server (DES) system was developed using ACE frameworks. The design and the implementation have been discussed and presented. The applications of several ACE frameworks and classes, such as ACE_Task framework, Jaws IO framework, ACE socket wrapper classes, and the ACE virtual filecache system, in the DES server system are discussed in detail. Performance objectives are set up for the DES system at the early design stage.

Performance measurements of the DES server were done with Rational Quantify tool. The DES server's execution times for each activity in the Retrieve Document scenario, both for cached and non-cached files, were extracted from the Quantify profile. The IO framework's `transmit_file()` execution times were studied under various document sizes it transmitted. Since this performance data is independent on the type of applications, it should be documented as the performance characteristic of the

framework for reuse. This study provides an approach to extend the reusability of frameworks to the performance domain, in addition to the functionality they offer.

The end-to-end measurements of the DES system were performed on the alpha measurement network in the RADS laboratory, which consists of 15 workstations, each with a 266 MHz Pentium processor and 256 MB RAM memory, and a 100Mbps Ethernet connection. The DES server's performance under various workloads and different document sizes was investigated in the measurement network.

The UML performance profile was applied to the DES system. The measured server's execution data was used as input parameters for the profile. A LQN performance model was built on the profile and used to assess the performance of the DES Retrieve Document scenario. The performance improvement by design alternatives for file caching in the server side was also investigated. The LQN modeling results were compared with the end-to-end measurement results under various workloads for different document sizes. The causes for the divergence between the model results and the measurements were also discussed.

6.2 Conclusions

Based on the case study, the following conclusions are made in this thesis:

1. The frameworks performance characteristics are reusable and should be documented with the frameworks via UML performance models and the Performance Profile.

In the DES system, the IO framework's usage is application independent. Its `transmit_file()`'s performance characteristics depend on the file caching strategy and the document size. The performance data such as Table 4.5 and Figure 4.7 could be reused as references for many programs that use this IO framework on the same platform.

Framework developers could provide information to help performance analysis. UML activity or interaction diagrams should be used to describe the software behaviors invoked through the framework interface; these diagrams should be annotated with performance information according to the UML Performance Profile. Stereotypes and tagged values identify the resources used (physical resources, such as CPU, network, I/O devices; logical resources such as software servers, critical sections, locks, semaphores). Also, quantitative resource demands should be provided for the framework functions, such as CPU execution time (in number of cycles), number of calls to the networking functions and message sizes, and number of I/O operations.

2. The performance of a software system built with frameworks may be partially or mainly determined by the framework's characteristics. The level of the framework's contribution depends on the extent of the framework's micro-architectures in the overall architecture of the system.

Frameworks determine partially the architecture of the software system built on top of them. The performance impact of frameworks on the system depends on how much the frameworks are involved in the key performance scenario of the system. As an example in the DES system, the server's execution time and performance of the system for the Retrieve Document scenarios are mainly determined by the IO framework's

transmit_file() performance characteristics, which makes demands to the following devices: CPU, network and disk.

3. Framework performance study should be kept at the right level of abstraction. The performance domain of the framework and the framework's usage domain (functionality offered through frameworks's interface) may not be at the same level. In order to model accurately the performance of a system built upon a framework, some knowledge of the framework is needed. Choosing the right level of abstraction is crucial. If the information about the framework is too abstract, necessary information may be hidden or missing. In contrast, too many low level details may be unnecessary and could only complicate the study. For performance modeling, the resource demands for both physical resources (CPU, IO, networks, etc.) and logical resources (software servers, locks, mutexes, buffers, etc.), are required. The framework documentation should capture the resources demanding characteristics of the frameworks.

4. The Layered Queuing Network (LQN) model is a powerful performance modeling tool, especially suitable for client/server systems. Provided we build a correct model of the system and use accurate resource demand parameters from measurements or other sources, LQN modeling could well predict the performance of distributed /concurrent systems under development.

6.3 Future Work

The following issues need further investigation to improve the performance analysis of concurrent/networked systems built upon reusable frameworks and components:

1. Extend the performance study to more complex software systems built with frameworks to acquire more knowledge and experience on how to effectively extract performance characteristics of frameworks at a right level for reuse. There will be some challenges to isolate the framework's contribution to the performance for more complex systems. Also, investigate how to characterize the performance of the more generic part of the framework, and how to reuse performance characteristics from a platform to another one.
2. Extend the performance study to software systems built with reusable components. Since components are also designed for reuse at large-scale, study the performance reuse for them could provide benefits, too.
3. Applying some scenario-based modeling techniques, such as Use Case Map (UCM) [Buhr96, UCMweb], would help trace the flow of control of the software system built upon frameworks. They would show more clearly the framework's contribution to the execution of key performance scenarios in a software system.

REFERENCES

- [Amer02] H. Amer and D. C. Petriu, "Software Performance Evaluation: Graph Grammar based Transformation of UML Design Models into Performance Models", submitted for publication, 2002.
- [Balsamo01] S. Balsamo and M. Simeoni, "On transforming UML models into performance models", in: *Proc. of Workshop on Transformations in the Unified Modeling Language*, Genova, Italy, 2001.
- [Banks99] J. Banks, J. S. Carson. II, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation*, 3rd ed. Prentice Hall, Upper Saddle River, 1999.
- [Bernardi02] S. Bernardi, S. Donatelli and J. Merseguer, "From UML Sequence Diagrams and Statecharts to analysable Petri Net models", in *Proc. of 3rd International Workshop on Software and Performance (WOSP'02)*, Rome, Italy, July 2002.
- [Buhr96] R.J.A. Buhr, R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996.
- [Calzaro98] M. Calzarossa and R. Marie, "Tools for performance evaluation", *Performance Evaluation*, 33, 1998, pp. 1-3.
- [Booch99] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, 1999.
- [Canevet02] C. Canevet, S. Gilmore, J. Hillston, and P. Stevens, "Performance modelling with UML and stochastic process algebras", in *Proceedings of*

the UK Performance Engineering Workshop (UKPEW'2002), Glasgow, July 2002.

- [Cortelle00] V. Cortellessa and R. Mirandola, "Deriving a Queueing Network based Performance Model from UML Diagrams", in *Proc. of 2nd ACM Workshop on Software and Performance*, Ottawa, Canada, 2000, pp. 58-70.
- [Cortelle01] V. Cortellessa, A. D'Ambrogio, G. Iazeolla, Automatic derivation of software performance models from CASE documents, *Performance Evaluation*, 45, 2001, pp. 81-105.
- [Courtis00] M. Courtois and M. Woodside, "Using Regression Splines for Software Performance Analysis and Software Characterization", *Proc 2nd Int. Workshop on Software and Performance (WOSP2000)*, Ottawa, Canada, Sept 2000, pp. 105-114.
- [Dilley98] J. Dilley, R. Friedrich, T. Jin, and J. Rolia, "Web server performance measurement and modeling techniques", *Performance Evaluation*, 33, 1998, pp. 5-26.
- [Fayad97] M. E. Fayad and D. C. Schmidt, " Object-Oriented Application Frameworks", *Communication of the ACE*, Vol.40, No. 10, October, 1997, pp. 32-38.
- [Fayad99] M. E. Fayad, D. C. Schmidt, R. E. Johnson, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley & Sons, New York, 1999.

- [Franks95] G. Franks, A. Hubbard, S. Majumdar, J.E. Neilson, D.C. Petriu, J. Rolia, C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", *Performance Evaluation* (special issue on Performance Tools), Vol. 24, No.1-2, Nov. 1995, pp.117-135.
- [Franks96] R. G. Franks, S. Majumdar, J.E. Neilson, D. C. Petriu, J. A. Rolia and C. M. Woodside, "Performance Analysis of Distributed Server Systems", *Proc. Sixth International Conference on Software Quality*, Ottawa, Canada, October 28-30, 1996, pp. 15-26.
- [Franks98] G. Franks, C.M. Woodside, "Performance of Multi-level Client-Server Systems with Parallel Service Operations", *Proc. of 1st Workshop on Software Performance (WOSP'98)*, October 1998, pp. 120-130.
- [Franks00] Franks, G., Performance Analysis of Distributed Server Systems, Report OCIEE-00-01, Ph.D. Thesis, Carleton University, Ottawa, Canada (2000).
- [Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.
- [Gomaa00] H. Gomaa and D. A. Menasce, "Design and Performance Modeling of Component Interconnections Patterns for Distributed Software Architectures", in: *Proc. of 2nd ACM Workshop on Software and Performance*, Ottawa, Canada, 2000, pp. 117-126.
- [Hofmeister00] C. Hofmeister, R. Nord, D. Soni, *Applied Software Architecture*, Publisher Reading, MA: Addison-Wesley, 2000.

- [Hrischuk99] C.E. Hrischuk, C. M. Woodside, and J.A. Rolia, "Trace-Based Load Characterization for Generating Software Performance Models", *IEEE Trans. on Software Eng.*, V.25, No.1, 1999, pp. 122-135.
- [Hu99] J. Hu and D. C. Schmidt, "JAWS: A Framework for High-performance Web Servers", *Domain-Specific Application Frameworks: Frameworks Experience By Industry*, John Wiley & Sons, M. Fayad and R. Johnson, Eds., October, 1999.
- [Hu00] J. Hu, I. Pyrali, and D. C. Schmidt, "The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks", *The Parallel and Distributed Computing Practices Journal*, special issue on Distributed Object-Oriented Systems, M. Cobb and K. Shaw, Eds., Vol. 3, No. 1, March 2000.
- [Hughes97] C. Hughes and T. Hughes, *Object-Oriented Multithreading Using C++*, John Wiley & Sons, New York, 1997.
- [Jain91] R. Jain, *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*, John Wiley & Sons, New York, 1991.
- [Johnson97] R. E. Johnson, "Frameworks = (Components + Patterns)", *Communication of the ACE*, Vol.40, No. 10, October, 1997, pp.39-42.
- [Kahkipuro01] P. Kahkipuro, "UML-Based Performance Modeling Framework for Component-Based Distributed Systems", R. Dumke et al., Eds., *Performance Engineering*, LNCS Vol. 2047, Springer, Berlin Heidelberg

- New York, Object Management Group, UML Specification Version 1.4, 2001, pp.167-184.
- [Kleinrock75] L. Kleinrock, *Queueing Systems Volume I: Theory*, Wiley Interscience, 1975.
- [Krueger92] C. W. Krueger, "Software Reuse". *ACM Computing Surveys*, 24, June 1992, pp. 131-83.
- [Law00] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*, 3rd ed , McGraw Hill, 2000.
- [Lazowska84] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance*, Prentice Hall, 1984.
- [Leavens00] C. Y. Leavens and M. Sitaraman, Eds., *Foundations of Component-based systems*, Cambridge University press, 2000.
- [Lijia00] D. J. Lijia, *Measuring Computer Performance: A practitioner's Guide*, Cambridge University Press, 2000.
- [Liu02] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen, "Design a Test Suite for Empirically-based Middleware Performance Prediction", 40th international conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, Conference in research and Practice in Information technology, Vol. 10, J. Noble and J. Potter, Eds., 2002.
- [MacNair85] E. A. MacNair, "An introduction to the Research Queueing Package", *Proc. 17th Winter Simulation Conference*, San Francisco, 1985, pp. 257 – 262.

- [Menasce94] D. A. Menasce, V. A. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling: from Mainframes to Client-Server Systems*. Prentice Hall PTR, Englewood Cliffs, New Jersey, 1994.
- [Menasce02] D. A. Menasce, "Software, Performance, or Engineering?" *Proc. of 3rd International Workshop on Software and Performance (WOSP'02)*, Rome, Italy, July 2002.
- [Menasce02a] D. A. Menasce and V. A. F. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, Upper Saddle River, 2002.
- [Morisio02] M. Morisio, D. Romano, and I. Stamelos, "Quality, Productivity, and Learning in Framework-Based Development: An Exploratory Case Study", *IEEE Transaction on Software Engineering*, Vol. 28, N. 9, Sept. 2002, pp. 876-888.
- [OMG02] Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification," OMG Adopted Specification ptc/02-03-02, July 1, 2002.
- [Orasad97] S. Orasad, *Multithreading programming techniques*, New York, McGraw-Hill, 1997.
- [Petriu97] D. C. Petriu and L. S. Metea, "Performance Modeling and Measurement of Software Systems" *Proc. IEEE Workshop on Emergent Technologies & Virtual Systems for Instrumentation and Measurements, ET&VS-IM'97*, Niagara Falls, May 15-16, 1997, pp. 142-152.

- [Petriu00a] D. C. Petriu and X. Wang, "From UML Description of High-Level Software Architecture to LQN performance models", in: Nagl, M., et al., Eds., *Applications of Graph Transformations with Industrial Relevance AGTIVE'99*, LNCS Vol. 1779, Springer, Berlin Heidelberg New York, 2000, pp. 47-62.
- [Petriu00b] D.C. Petriu, H. Amer, S. Majumdar, and I. Abdul-Fatah, "Using Analytic Models for Predicting Middleware performance", Proc. of the 2nd ACM Int. Workshop on Software and Performance WOSP'2000, Ottawa, Canada, Sept. 2000, pp. 189-194.
- [Petriu00c] D.C. Petriu, C. Shousha, and A. Jalnapurkar, "Architecture-Based Performance Analysis Applied to a Telecommunication System", in: *IEEE Transactions on Software Eng.*, Vol.26, No.11, 2000, pp. 1049-1065.
- [Petriu02a] D. C. Petriu and M. Woodside, "Performance Analysis with UML: Layered Queuing Models from the performance Profile" Book chapter, to be published by Kluwer.
- [Petriu02b] D. C. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", in Computer Performance Evaluation - Modeling Techniques and Tools, T. Fields, P. Harrison, J. Bradley, U. Harder, Eds., Lecture Notes in Computer Science 2324, Springer, 2002, pp.159-177.
- [Rolia95] J.A. Rolia and Sevcik, K.C., "The Method of Layers", *IEEE Trans. on Software Engineering*, Vol. 21, No. 8, 1995, pp. 689-700.

- [Sameting98] J. Sametinger, *Software Engineering with Reusable Components*, Springer, 1998.
- [Schmidt97a] D. C. Schmidt and M. E. Fayad, "Lessons Learned Building Reusable OO Frameworks for Distributed Software", *Communications of the ACM*, Special Issue on OO Application Frameworks, ACM, Vol. 40, No. 10, October, 1997.
- [Schmidt97b] D. C. Schmidt, "Applying Patterns to Meet the Challenges of Concurrent Software", *IEEE Concurrency*, Special Edition on Software Engineering for Parallel and Distributed Systems, Vol. 5, No. 3, August, 1997.
- [Schmidt00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Vol 2: Patterns for Concurrent and Networked Objects*, John Wiley & Sons, Chichester, 2000.
- [Schmidt02a] D. C. Schmidt and S. D. Huston, *C++ Network Programming Vol 1: Mastering Complexity with ACE and Patterns*, Addison-Wesley, 2002
- [Schmidt02b] D. C. Schmidt and S. D. Huston, *C++ Network Programming Vol 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, 2002
- [Sheikh97] F. Sheikh and C.M. Woodside, "Layered Analytic Performance Modelling of a Distributed Database System", *Proc. 1997 International Conf. on Distributed Computing Systems*, May 1997, pp. 482-490.
- [Sitaraman99] M. Sitaraman, "Software Reusability," *Wiley Encyclopaedia of Electrical and Electronics Engineering*, Vol. 19, Ed. J. G. Webster, John Wiley & Sons, 1999, pp. 578-590.

- [Sitaraman01] M. Sitaraman, G. Kulczycki, J. Koron, W. F. Ogden, and A. L. N. Reddy, "Performance Specification of Software Components", *Proc. of the 2001 Symposium on Software Reusability*, Toronto, Ontario, Canada, 2001.
- [Smith90] C. U. Smith, *Performance Engineering of Software Systems*, Readings, MA, Addison Wesley, 1990.
- [Smith98] C. U. Smith and L.G. Williams, "Performance Evaluation of Distributed Software Architectures", *Proc. CMG*, Anaheim, CA, Dec. 1998.
- [Smith99] C. U. Smith and M. Woodside, Chapter in Book: Performance Validation at Early Stages of Software Development, E. Gelenbe Ed. *System Performance Evaluation: Methodologies and Applications*, CRC Press, 1999.
- [Smith00] C. U. Smith, "Building Responsive and Scalable Web Applications", *Proc. CMG*, Orlando, December 2000.
- [Smith01] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison Wesley, 2001.
- [Somadder97] G. Somadder and D.C. Petriu, "Performance Measurements of Multi-Threaded Servers in a Distributed Environment", In *Open Distributed Processing and Distributed Platforms*, J.Rolia, J.Slonim and J.Botsford, eds., Chap. 12, pp. 151-165, Chapman & Hall, 1997.
- [Szyperski98] C. Szyperski. *Component Software: beyond OO Programming*, Addison-Wesley, 1998.
- [UCMweb] Use Case Map Web Page, <http://www.usecasemaps.org/index.shtml>.

- [Wang99] J. Wang and S Keshav, "Efficient and accurate Ethernet simulation", Technical Report TR99-1749, Computer Network Research Group, Cornell University, Ithaca, NY, May 1999.
- [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, 1995, pp 20-34.
- [Woodside96] C. M. Woodside, "Performability Modeling for Multi-Layered Service Systems", *Third Int. Workshop on Performability of Computer and Communications Systems*, Bloomington, Illinois, USA, Sept. 7-8, 1996.
- [Woodside98a] C. M. Woodside, C. Hrischuk, B. Selic, and S. Bayarov, "A Wideband Approach to Integrating Performance Prediction into a Software Design Environment", *Proc. First Int. Workshop on Software and Performance (WOSP98)*, pp. 31-41, October 1998.
- [Woodside98b] C. M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks", *Performance Evaluation*, Vol.9, No.2, 1998, pp.143-160.
- [Woodside02] C. M. Woodside, Lecture Notes for 94.511 High-performance software, The Department of Systems and Computer Engineering, Carleton University, Ottawa, 2002.
- [Yacoub02] S. Yacoub, "Performance Analysis of Component-Based Applications". G. Chastek, Ed., SPLC2 2002, LNCS 2379, pp299-315, Springer-Verlag, Berlin Heidelberg, 2002.