INFORMATION TO USERS

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

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

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

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

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

UMI

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

NOTE TO USERS

The original manuscript received by UMI contains pages with slanted print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI

Automatic Generation of Performance Models for Distributed Application Systems

by

Ming Melvin Qin, M.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Science, Information and Systems Science

Department of Systems and Computer Engineering

Carleton University
Ottawa, Ontario
January 1998

© Copyright
1998, Ming Melvin Qin

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

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

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

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

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

0-612-27066-1

Canada

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

**Automatic Generation of Performance Models for Distributed
Application Systems**

submitted by

Ming Melvin Qin. M.Sc.

in partial fulfilment of the requirements
for the degree of Master of Science, Information and Systems Science

Chairman. Department of Systems and Computer Engineering

Thesis Supervisor Dr. Jerome A. Rolia

Carleton University
January 15, 1998

Abstract

Distributed computing systems are becoming increasingly popular in many organizations. The performances of these systems in terms of response times and cost has a major impact on their effectiveness. To achieve openness and scalability these systems have begun to rely on distributed environment technologies such as the Open Software Foundation's Distributed Computing Environment (DCE) and the Object Management Group's Common Object Request Broker Architecture (CORBA). The resulting systems are complex and have performance behaviour that may defy intuition, because it involves factors in the software design (such as the partitioning of the functionality and the frequency of service requests made to each server) and in the configuration of the distributed system (including distribution of data, replication of services, and the speed of network communication). Predictive performance models and performance evaluation techniques are needed to help manage their behaviour. Predictive models allow the timely evaluation of the performance impact of many different application loads and alternative configurations. Unfortunately model building by hand can be time consuming and error prone. This thesis describes techniques developed to help automate the generation of performance models for operational distributed application systems. It describes the information that is needed to automate model building, the infrastructure that supports the model building, and the methods used to obtain or estimate the required information.

Acknowledgments

I am in debt to many people for their support and advice in completing this thesis. Firstly, I would like to thank my supervisor Professor Jerome Rolia. His patience and guidance made this work possible and enjoyable.

I would also like to thank my colleagues, Asham El Rayess, Randy Lee, Vidar Vetland, and the folks in the RADS lab. Thanks also go to the many Professors and support staff for providing and maintaining an excellent learning and research environment.

I was awarded Ontario Graduate Scholarship and IBM Research Fellowship during the course of this thesis work. Thank you very much, Ontario and IBM.

Last but never the least I thank my parents for their love and encouragements.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	viii
List Of Figures	x
1 Introduction	1
1.1 Distributed Systems Management	1
1.1.1 Distributed application performance management	3
1.2 Problem statement	6
1.3 Contribution to research	6
1.4 Thesis organization	7
2 Background	8
2.1 Distributed environments	8
2.1.1 Distributed Computing Environment	12
2.1.2 CORBA	17
2.1.3 ANSA and ANSAware	22
2.2 Reference Model for Open Distributed Processing	24
2.3 Performance Issues for Distributed Systems	27

2.4	Software performance engineering	28
2.5	Summary	33
3	Performance evaluation: models and techniques	34
3.1	Performance Evaluation Terminology	36
3.2	Queueing Network Model	38
3.3	Layered Queueing Models	41
3.4	Distributed Application Performance Metrics and Management . . .	46
3.5	Performance Measurement for Distributed Applications	48
3.6	Summary	54
4	Information and Infrastructure for Model Building	56
4.1	Information for Model Building	56
4.1.1	Information Sources for Model Building	59
4.2	Infrastructure for model building	60
4.3	Application Program Instrumentation Package	64
4.4	Performance data server	67
4.4.1	The Interface	67
4.4.2	Data analysis	71
4.5	Summary	72
5	Building the Layered Queueing Model	74
5.1	Functional Requirements of Model Builder	74
5.2	Model Building Steps	76
5.2.1	Measurement Time Period	77
5.2.2	Building the Model Structure	79
5.2.3	Deducing Model Parameters	84
5.3	Implementation of Model Building System	86
5.4	Exceptions, model verification, and model validation	90

5.5	Unmanaged work and other types of interactions	91
5.6	Summary	92
6	Case study	94
6.1	An Instrumented Distributed Application	94
6.2	Building the Layered Queueing Model	99
6.3	Predicting System Performance with the LQM	110
6.4	Summary	113
7	Conclusions	114
7.1	Summary	114
7.2	Conclusions	116
7.3	Future Work	117
	References	119

List of Tables

6.1	Prescribed Resource Visit Ratios and Mean Service Times for the Synthetic Application	98
6.2	Experiment Setup Combinations	99
6.3	Measured Visit Ratios and Regressed CPU Service Times for the Client with Relative Errors Compared with Prescribed Values	101
6.4	Measured Visit Ratios and Regressed CPU Service Times for Server 1 with Relative Errors Compared with Prescribed Values	102
6.5	Measured Visit Ratios and Regressed CPU Service Times for Server 2 with Relative Errors Compared with Prescribed Values	103
6.6	Mean Client Response Time Estimates from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model	106
6.7	Mean Response Time Estimates of Server 1, Operation 1 from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model	107
6.8	Mean Response Time Estimates of Server 1, Operation 2 from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model	107
6.9	Mean Response Time Estimates of Server 2, Operation 1 from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model	108

6.10	Mean Response Time Estimates of Server 2, Operation 2 from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model	108
6.11	Resource Utilization Estimates from the Generated Model with Rel- ative Errors Compared with the Prescribed Model	109
6.12	Mean Queue Length Estimates from the Generated Model with Rel- ative Errors Compared with the Prescribed Model	109
6.13	Performance Estimates for the Original System with 40 Clients and 3 Server Threads	112
6.14	Performance Estimates for the Modified System of Option 1 (Two CPU's), with 40 Clients and 3 Server Threads	112
6.15	Performance Estimates for the Modified System of Option 2 (Double Speed CPU), with 40 Clients and 3 Server Threads	112

List of Figures

2.1	Remote Procedure Call (RPC) mechanism	10
2.2	OSF/DCE Runtime components	12
2.3	An example of DCE IDL file	14
2.4	DCE Server Initializing and Client Finding a Server	16
2.5	Object Management Architecture	18
2.6	Example CORBA IDL and client application code	20
2.7	CORBA interfaces	21
2.8	RM-ODP Viewpoints and Software Engineering	25
2.9	RM-ODP Viewpoints and Software Engineering	26
3.1	An open queueing network has external arrivals and departures . . .	38
3.2	A closed queueing network has no external arrivals and departures . .	39
3.3	A Layered Queueing Model (LQM)	43
3.4	Distributed Measurement System Architecture	52
4.1	Infrastructure for Model Building	61
5.1	Notations and Symbols	78
5.2	Algorithm for building LQM model structure with measurement data. Part I: Creating entities and entries for processes and devices.	81
5.3	Algorithm for building LQM model structure with measurement data. Part II: Creating events and connectivity among entities.	82

5.4	Main Components of the Model Building System	87
5.5	Algorithm for fixing LQM model parameters. Part I: accumulate measurement data.	88
5.6	Algorithm for fixing LQM model parameters. Part II: deduce mean values.	89
6.1	A Synthetic DCE Application and the Performance Data Server . . .	95
6.2	Layered Queueing Model (LQM) of the Synthetic Application	97

Chapter 1

Introduction

Distributed computing systems enable the sharing of an organization's information and computing resources. Such systems are complex and have performance behaviours that can defy intuition. Predictive performance models and performance evaluation techniques are needed to help manage their behaviours. Unfortunately model building by hand, as it is usually done, is costly and error prone. This thesis describes techniques developed to help automate the construction of performance models for operational distributed application systems.

1.1 Distributed Systems Management

Distributed computing systems typically consist of a large numbers of heterogeneous computing devices connected by communication networks, various operating system resources and services, and user applications running on them. As these systems get larger and more complex, it becomes more and more difficult to keep them running smoothly, and more things can go wrong. Unavailable services, incorrect services, or inefficient operations of services and applications can potentially interrupt or cripple critical operations. Thus, management support is seen as an important and essential aspect in distributed computing systems[4].

Distributed systems management is a process of ensuring the efficient use of resources and reliable and timely service to users. It includes three classes of system management:

1. *Network management* of network services and devices, such as routers, bridges, and gateways;
2. *Systems management* of system services and resources, such as operating systems, file systems, and I/O devices;
3. *Application management* of user applications, such as configuration, performance measurement, and performance modelling.

The primary objective of distributed system management is to ensure the required behaviour of distributed applications, since it is the applications that are critical to the end users of the system [4]. This is not to say that the management of networks and systems is not important, rather it implies that their management is a critical factor in the overall management objective. Obviously, a malfunctioning network results in poor behaviour of distributed applications operating over the network.

From the perspective of managing the performance of the a distributed application, it is evident that information about the behaviour of the underlying systems and networks is essential. Management of networks, systems, and applications in isolation is too narrow a focus, and that an integrated approach is required. For example, processes migration can be done to reduce the load on a particular host, or process could be migrated to reduced the cost of interprocess communication within a distributed application, with the potential effect of improving the performance of user applications or decreasing network loads.

Distributed systems management also integrates system monitoring, information management, and system modelling techniques:

1. *Monitoring* is the collection of management information.

2. *Information management* refers to the storage and manipulation of management information.
3. *System modelling techniques* provide tools for system performance analysis and help the control of the system.

Most of the research and implemented distributed management systems to date have focused on network management, but in order to provide true manageability for users, all three classes of distributed systems management must eventually be included. In this work, we focus on performance management of distributed applications.

1.1.1 Distributed application performance management

A collection of programs in execution that are distributed over a network and cooperate to accomplish some overall objective is called a *distributed application*. Instances of programs correspond to operating system processes. The procedure of building the software processes from software objects are defined as *composition*. These objects may be the methods or procedures in a distributed application. *Distribution* is defined as the placement of the software processes across nodes in a computer network [31].

Composition must be considered early, since it can have a major impact on design. It is also important to plan support for composition alternatives so that they can be reflected in the architecture and design of the application. Distribution deals with system configuration for specific target environments. A good and flexible distribution can only be available with a good composition. Flexibility in composition and distribution are important. One composition and distribution may be best for a system with tens of users, but a different alternative of the same application may be appropriate for a larger environment with thousands of users. An application is scalable if it can be easily configured to balance resource consumption across devices for target environments of interest.

Performance refers to system responsiveness: either the time required to respond to specific events, or the number of events processed in a given time interval[38]. A typical performance requirement for a system supporting an on-line application is as follows: the system must be able to complete λ requests of type C per second with ninety percent of the requests having a response time less than R seconds. If the requirement is satisfied the application and the network supporting the application can be said to offer a high *quality of service* to the application's customer. Otherwise the system's operational effectiveness will be hampered. Performance management from the perspective of the applications is called *distributed application performance management*. A performance model for a system of applications can be used to detect performance bottlenecks and to conduct "what if" analysis for exploring alternative system configurations.

Ensuring a software system meet its performance requirements is a significant problem. This problem becomes even more difficult to solve for concurrent and/or distributed software systems. The performance behaviour of these distributed applications should be continuously monitored by a distributed application performance management system to verify that their performance requirements are being satisfied and, if necessary, to help locate the source of performance problems. Tools and techniques for distributed application performance management are essential but are currently not available.

Performance issues for conventional centralized systems are moderately well understood. If performance problems occur, experts familiar with a subsystem can improve its behaviour using subsystem tuning techniques. For example, changes can be made to affect operating system, network, or database behaviour, or to speed up or add a small number of resources. To scale upwards towards an organizational or global processing model, new and reengineered applications distribute their functionality across heterogeneous distributed computing facilities. Though *system management* from the perspective of managing individual subsystems will always be important, this strategy does not sufficiently address distributed application system performance problems. Instead we must approach the problem from the perspective

of the applications themselves. Performance-oriented design and management shifts from understanding a collection of independent centralized subsystems to understanding the correlations between distributed application components and the loads they place on the computing facilities available to them.

Performance modelling methodologies, instrumentation, and tool support exist for centralized systems. In such systems the contention for hardware is prominent. Workloads are often characterized in terms of users or groups of users and their use of a relatively small number of hardware resources. For distributed application systems the picture is quite different. Software servers can cause queueing delays and may become bottlenecks so they must be represented explicitly in predictive models along with a characterization of their use of hardware. We are forced to include this greater detail in our performance models. Unfortunately this makes the model building process time-consuming and costly.

There exist several modelling techniques. Some are analytic, like queueing networks and Petri nets, while others are based on discrete event simulation. Analytic techniques can usually provide results much faster than simulations if the analytic models can be solved. For queueing networks we talk about product-form networks or near-product form networks where each queueing centre can be analyzed in isolation and the partial result be combined to produce the overall results. The major problem with Petri nets is the exponential growth of the state space for most realistic models. Simulation models can represent any detail of a system, but the time it takes to develop the model and to run it may be prohibitive. If there are rare events in the simulation model, the simulation may need to run for a long time to obtain statistical significance.

For operational decision making, building models for distributed application systems “by hand” is impractical. New instrumentation and tools are needed to help support and automate model building. Fortunately, there is a trend towards the standardization of distributed system technologies. Most vendors are working towards “open” systems where heterogeneity is possible because of standardized protocols and interfaces. This trend makes it more feasible to include support for the

new instrumentation needed for distributed application performance management.

1.2 Problem statement

The problem this research proposes to solve is:

“Given an operational distributed software application, construct a Layered Queueing Network Performance Model (LQM) [29] which includes structural information and resource demand parameters.”

The technique presented in thesis is a cost-effective method for performance model building that enables the performance management of distributed applications. This technique requires a infrastructure which consists of management agents, management information store, and a model builder. Management agents are running on each machine in a distributed system. They collect management data of the distributed application processes. The data is periodically passed to a configuration database and a management repository. Information about the resource demands (CPU time, I/O counts, etc.) of processes and invocation counts for each operation of each process are used to deduce the resource demands per invocation of an operation. A layered queueing network model (LQM) is built automatically based on information from the configuration data base. LQM models are useful in this context since they are capable of representing contention for both hardware devices and software servers. In this way a “baseline” performance model can be generated in a short time without intervention from the system administrator. The performance model can be used for “what-if” analysis and it may be used for detection of bottlenecks in the distributed system in its current state. We notice that the timeliness of the performance models is very important when used for system management.

1.3 Contribution to research

1. Documented a distributed application performance management architecture and its implementation.

2. Developed techniques to estimate performance parameters for analytic models from performance measurements.
3. Developed a tool that generates performance models from measurements.
4. Identified and supported exceptional conditions in model building process.

1.4 Thesis organization

The structure of the thesis is as follows. Chapter 2 provides background material in distributed computing environments. Chapter 3 describes performance evaluation techniques and the informational requirements for model building and the measurement infrastructure. An instrumentation package that collects data about applications is documented in Chapter 4, as well as a server for performance data and how it is used to estimate some values that are difficult to measure directly but nevertheless are needed for predictive models of distributed applications. Chapter 5 presents the steps of performance model generation and identifies cases that cannot be supported. Examples of model building are given in Chapter 6. Chapter 7 presents some practical experiences and concluding remarks.

Chapter 2

Background

This chapter presents some background material of this thesis. Section 2.1 describes distributed computing environments including DCE, CORBA, and ANSAware. The reference model for open distributed processing is presented in Section 2.2. Performance issues of distributed computing environments are identified in Section 2.3, and Section 2.4 introduces software performance engineering (SPE) methods that address these performance problems.

More details on performance models and performance evaluation techniques that are used in SPE will be discussed in Chapter 3.

2.1 Distributed environments

With advances in communications technologies and development of powerful desktop workstations, computer systems are rapidly changing from a centralized model to a distributed one.

Tanenbaum [40] noted the number one reason for the trend towards distributed system is that these systems potentially have much better price/performance ratio than a single large centralized system would. A next reason for building a distributed system is that some applications are inherently distributed. Another potential advantage of a distributed system over centralized one is higher reliability. Distributed

systems also have advantages over isolated workstations by allowing data sharing, device sharing, and communication.

Coulouris [9] identified six key characteristics which are primarily responsible for the usefulness of distributed systems. These are *resource sharing*, *openness*, *concurrency*, *scalability*, *fault tolerance* and *transparency*. It should be noted that they are not automatic consequences of distribution: system and application software must be carefully designed in order to ensure they are attained.

The single most important difference between a distributed system and a centralized one is the *interprocess communication* between processes. A centralized system consists of only one process which serves user requests, whereas a distributed application built in a distributed system consists of many software processes, distributed over a network, cooperating to accomplish some overall application goals. Distributed applications can be modeled effectively with client-server model. The client-server model embodies a client program (client) and a server program (server), usually running on different systems of a network. The client makes a request to the server, which is usually a continuously running daemon process, and the server sends a response back to the client. *Client* and *server* are relative terms for a particular interaction: in complex systems, server process for one client may itself become the client process for another server.

When a client process issues a request to a server process, there must be interprocess communication mechanism in place to transport the message to the intended receiver and to transport the output to the client when the remote server has finished processing. The *remote procedure call* (RPC) [40] mechanism is the simplest way to implement client-server applications because it keeps the details of network communications out of application code, by using stub code and runtime library. A stub is surrogate code that supports remote procedure calls. During a remote procedure call, stub code is intermediate between application code and the RPC runtime library, and it prepares input and output parameters for transmission. Stub code is usually generated by special compilers, such as Unix *rpcgen* protocol compiler and the *interface definition language* (IDL) compilers in OSF/DCE [23]

Figure 2.1: Remote Procedure Call (RPC) mechanism

and CORBA [21].

Figure 2.1 on Page 10 illustrates how the application code work with RPC mechanism during a remote procedure call. The RPC call in the client code is very much like a local procedure call, because it is a call to a client stub. The client stub packs the parameters into a message and sends the message to the server stub using the RPC runtime library. The server's RPC runtime library receives the remote procedure call and communicates the client information to the server stub. The server stub unpacks parameters and invokes the remote procedure in the server application code, which executes in the server address space. When the server finishes executing the remote procedure, its stub communicates output to the client stub by using the RPC runtime library. Finally, the client stub returns to the client application code.

An essential part of the RPC mechanism is an *interface*, which is a set of remote procedure declarations. Each procedure declaration includes the name of the procedure, the data types of its parameters (if any). An interface definition contains a

set of procedure declarations and data types. Once interfaces are defined, client and server development of an application can occur in parallel and on separate systems of the network.

In order to support distributed application development and to provide connectivity and interoperability among heterogeneous computing systems, a number of distributed environments have been developed. Examples are the Distributed Computing Environment (DCE) [23] offered by the Open Software Foundation, the Common Object Request Broker Architecture (CORBA) [21] by the Object Management Group, and the ANSAware (implementation of the Advanced Network Systems Architecture) [2] by Architecture Projects Management (APM). Besides these commercial products, ISO develops a reference model for Open Distributed Processing (RM-ODP) [42, 43, 44]. It consists of concepts and their formal interpretation for distributed system design, general functions for building an infrastructure and means for bridging heterogeneity between interfaces.

The objective of the middleware environments is to provide the services that distributed applications need and to facilitate the development of distributed applications. In particular, middleware aims to make distributed applications independent of underlying platforms. To accomplish this, it provides runtime services supporting various forms of transparency, such as distribution and location transparency. In addition, it tries to hide many of the details in the underlying system from the developer of the distributed application.

Middleware has several broad characteristics: the services provided are intended to span multiple, heterogeneous, distributed platforms; it is built on top or incorporates existing communication standards; it assumes a transport-independent service layer that provides access to lower-level communication services.

The following sections briefly describe DCE, CORBA, ANSAware, and RM-ODP respectively.

Figure 2.2: OSF/DCE Runtime components

2.1.1 Distributed Computing Environment

Open Software Foundation's (OSF) Distributed Computing Environment (DCE) [23] supports the development and operation of distributed application across heterogeneous platforms. Its runtime components shown in Figure 2.2 on Page 12 allow operating system and network protocol details to be hidden from the application developer. In addition, DCE provides tools to manage its components and tools to automate that part of the development process dealing with distribution of procedure calls.

The DCE *threads service* allows the creation, management and synchronization of multiple threads of control within a single process. It provides an application programming interface to threads used by DCE and by application programmers. The implementation of the thread service depends in part on the operating system of the host. In some operating systems, kernel threads are available and can be used to implement the threads; alternatively, DCE threads can be implemented within the runtime environment of each process.

The DCE *Security Service* provides secure communications and control access to resource within a distributed system, including mechanism for the authentication, authorization and privacy of service requests. The *Distributed Time Service* provides global synchronized time to hosts within the system. The *Distributed File Service* allows users to access and share files anywhere in the system without having to know their physical locations. The directory service functionality is further divided into inter-cell and intra-cell parts. A cell is a fundamental grouping of hosts into some administrative domain. Intra-cell naming is handled by the *Cell Directory Service (CDS)*. Inter-cell naming is handled by the *Global Directory Service (GDS)*, which may either by the X.500 or *Domain Name Service (DNS)* naming scheme.

The DCE *RPC Service* handles the remote procedure calls from a client to a server. The components of the RPC service include the *Interface Definition Language (IDL)* and the RPC runtime. Process interfaces are described using IDL. The description includes a definition of the data types used in the interface and the services provided. The services are specified as procedure prototypes that include the name of the procedure, the names and types of its parameters, and attributes for each parameter indicating whether it is a input, output, or input/output parameter. An interface can have many services and each service is associated with a separate procedure. Each interface must have a *universal unique identifier (UUID)* to distinguish itself from any other interface on the network. UUID can be obtained from the *uuidgen* utility. The *uuidgen* utility generates a unique UUID using time and network address information so that no matter when and where it is generated, it is guaranteed to be unique. The UUID that a client uses must match that of the server otherwise the RPC will not be successful. The interface description for a server is shown in Figure 2.3 on Page 14.

When the interface definition is complete, the IDL file is compiled with the IDL compiler, which creates the following:

- A C language header file that contains definitions needed by the stubs and application code.

```
/******  
** System      : Greetings Application      **  
**                               **  
** Program     : greet.idl                  **  
**                               **  
** Date        : Aug/20/93                  **  
*****/  
[  
  uuid(00802C81-8950-1E1A-9C28-02070107E327),  
  version(1.0)  
]  
interface greetrh  
{  
    const long int STRING_SIZE = 100;  
  
    void greet(  
        [in]          handle_t h,  
        [in, string]  char client_greeting[],  
        [out, string] char server_reply[STRING_SIZE] );  
}
```

Figure 2.3: An example of DCE IDL file

- A client stub file, which will be linked with the client portion of the application.
- A server stub file, which will be linked with the server portion of the application.

The client code need to include the header file produced by the IDL compiler. It is then linked with the client stub and RPC library to produce the client executable. The server executable is created similarly, but it has two portions of code: the actual remote procedures, and the code to initialize the server.

When a client makes a remote procedure call, a *binding* relationship is established with a server. *Binding information* is network communication and location information for a particular server: it includes the following:

1. Protocol sequence, which is an RPC-specific name containing a combination of communication protocols that describe the network communication between a client and server.
2. Server host, which is the name or network address of the server host.
3. Endpoint, which is a number representing a specific server process running on a system. It is typically a port number for TCP or UDP.

A client can find the server in a number of ways, such as managing binding information directly in client and server code, using data files, or using environment variables. However, these methods are not flexible and present maintenance problem. To help clients find servers in a flexible and portable manner, DCE provides a name service to store binding information. The particular name service offered with DCE is the *Distributed Directory Service*.

When it starts up, a server process must export its interface description to the CDS. Figure 2.4 on Page 16 shows the initialization process. A server first registers the interface with the RPC runtime library. The runtime library creates binding information to identify this server process. The server places the binding information in appropriate databases, and places communication and host information in the

Figure 2.4: DCE Server Initializing and Client Finding a Server

CDS. The server also places process information (endpoints) in a special database on the server system called the *local endpoint map*. In the final initialization step, a server waits while listening for remote procedure calls from clients.

Figure 2.4 on Page 16 also shows how a client can find a server when an RPC is made. A remote procedure call in the client application transfers control to the client stub. The client stub finds the server system in the CDS. The RPC runtime finds the server process endpoint in the server system endpoint map, and then uses the binding information to complete the binding of the client to the server. Once the connection has been made, the remote procedure call executes as shown in Figure 2.1 on Page 10. The client stub converts the information into an internal RPC format that the runtime library transmits over the network. This conversion is called *marshalling*. The server runtime library receives the data and transfers it to the server stub, which *unmarshalls* it back to a format the application can use. Once the server finishes executing the procedure, the return arguments are sent back to the client by the stubs and runtime libraries in the same manner.

The interface description included in each server determines what services it can provide. Once the IDL file is compiled and the server code completed, the services are fixed. However, the distribution of the server processes in the network is flexible. The distributed system can be reconfigured by moving the server processes to different hosts, for example from a highly utilized host to a relatively free one, or to a dedicated host. With the CDS, client processes can still locate the servers in a new configuration.

2.1.2 CORBA

The Common Object Request Broker Architecture (*CORBA*) from the Object Management Group (OMG) is an industry consensus standard that defines a higher-level facility for distributed computing [21]. CORBA simplifies distributed systems in several ways.

- The distributed environment is defined using an object-oriented paradigm

Figure 2.5: Object Management Architecture

that hides all differences between programming languages, operating systems, and object location.

- Interoperability of diverse types of implementations is accomplished through well-defined interfaces. CORBA provides a portable notation for defining interfaces called the OMG Interface Definition Language (IDL).
- A comprehensive set of standard OMG IDL interfaces called object services and common facilities is being developed by OMG participants. Whereas CORBA simplifies the distributed software environment, object services and common facilities address the need for interface standards supporting application-level interoperability.

Figure 2.5 on Page 18 shows the OMG's object management architecture with the following components.

- *Object Request Broker (ORB)* - commercially known as CORBA, the ORB functions as a communication infrastructure, transparently relaying object re-

quests across distributed heterogeneous computing environments.

- *Object Services* - these components standardize the life-cycle management of objects. Interfaces are provided to create objects, to control access to objects, and to keep track of objects.
- *Common Facilities* - these provide a set of generic applications that can be configured to the specific requirements of a particular configuration, such as printing, document management, and e-mail.
- *Application Objects* - applications objects comprise all the remaining software including developer's programs, commercial applications, and legacy systems. Application objects can alternate between client roles and server roles. Server objects are called *object implementations*.

OMG Interface Definition Language (IDL) is a technology independent syntax for describing object encapsulations, in which interfaces that have attributes and operation signatures can be defined. A sample IDL is shown in Figure 2.6 on Page 20. It is compiled into header files, stub programs and skeleton programs for each interface. The client program links directly to the stubs. The skeleton program is the corresponding server-side implementation of the OMG IDL interface.

A client accesses an object by issuing requests on the object with an object reference (sometimes called a handle) to the object. The client need not be aware of the location or the state of the object. The ORB is responsible for locating the object implementation that can serve the client requests, handling the connection management between the client and the object, and transmitting data associated with requests and responses.

Figure 2.7 on Page 21 illustrates the communication process between a client and an object. Using an object reference, a client may issue requests on an object either by making a method invocation using a stub, or alternatively by dynamically composing a request using the *Dynamic Invocation Interface* (DII). The reference signifies the type of object to satisfy the request, not a specific object. This allows

```

////////////////////////////////////
//                                     //
// Interface Definition Language      //
//                                     //
// Example - greet                   //
////////////////////////////////////

interface greet
{
    string op1(in long x, inout string y, out string z);
};

////////////////////////////////////
//
// Client (Orbix/INS bindings) code    //
//
////////////////////////////////////

#include 'greet.hh'    // header generated by the IDL compiler
char *result;
long x;
char *y;
char *z;
greet_var myGreet;    // reference to a greet server object

myGreet = greet::_bind(); // Bind to any 'greet' on the network
result = myGreet->op1(x,y,z);

```

Figure 2.6: Example CORBA IDL and client application code

Figure 2.7: CORBA interfaces

the architecture to provide the same interface to the client regardless of where the objects are located or what language the object is implemented in. The ORB core then locates the object, establishes a connection to the object if necessary and delivers the request and all related data from the client to the object implementation. On the object side, the ORB core and the *Basic Object Adapter* (BOA) cooperate to first locate the actual object implementation, and then to deliver the request to it in the form of a method invocation by the skeleton. The object implementation then proceeds to perform the action requested by the client and sends a response back to the client in a similar way.

The first responsibility of a server is to register its implementation with the BOA. The BOA stores this registration to use in future object request. After an object is activated, it may receive client requests through the method call-backs in the OMG IDL skeleton. The BOA is also responsible for the generation of the object reference management, exception handling, the security of interactions, among others.

The ORB core is responsible for locating a server object to execute a request. If the servers are redistributed, the servers will simply inform the ORB core of their new address.

2.1.3 ANSA and ANSAware

Advanced Network Systems Architecture (ANSA) [2] is an architecture for Open Distributed Processing (ODP), and it has been a significant input to the development of the ISO Reference Model of ODP [42, 43, 44]. It is, however, not a design for a distributed computing product. Instead, ANSA specifies a set of components, which forms the basic building blocks and tools of the architecture, and a set of rules, recipes, and guide lines, that helps designers make decisions on how to combine components in designs conforming to the architecture. ANSA is characterized by the properties of its components and the nature of the compositions rules. These can not be expressed in a simple diagram as with other architectures [15]; the architecture allows many different designs.

ANSA uses five complementary projection models (i.e. enterprise, information, computational, engineering and technology) to describe architectural components. Each of the ANSA projection model is object-based. They differ in the way objects are used and in what is modeled. For example, the core of the ANSA computational model is the use of *objects* as units of distribution for management and placement. An object has one or more interfaces that are the points of provision and use of services. Interfaces are of two kinds: *operational interfaces* and *stream interfaces*. Operational interfaces contain a set of named *operations* (i.e. procedures and methods) and have the usual remote procedure call style of interaction.

The ANSAware is an implementation of the ANSA computational model and an example of the ANSA engineering model. It is available for a number of operating systems such as SunOS, HP/UX, VMS, and DOS, allowing users to write applications suitable for heterogeneous distributed environments. ANSAware essentially consists of an infrastructure placed on top of the operating system, and provides a uniform, technology-independent platform upon which application can be executed. Heterogeneity of programming language and operating system is bridged by a RPC mechanism and a common language for abstract interface description called the interface definition language (IDL).

The basic building block of ANSAware is service. Besides services like remote

procedures call and thread support. ANSAware provides three servers.

1. *Factory server* is able to start server objects of a certain service type.
2. *Trader* is another server that provides interface references of servers of a given service type.
3. *Notification service*.

Services are provided at interfaces: an interface is a unit of service provisioning. A client can invoke an operation or service at the interface of a server object in two different ways:

- By interrogation, in which the invoking client waits for the server to perform the operation and return the results (similar to an RPC):
- By announcement, in which the invoking client does not wait for the server to perform the operation and no result is returned (remote process spawn or asynchronous RPC).

An interface reference has to be obtained before a client can use an object service. This reference is obtained by making an service import request to the trader. Interface references that can be forwarded must have already been exported to the trader. The location of the objects or the type of machine they execute on can be changed at run time: the ANSAware infrastructure enables a flexible configuration of application components and provides a uniform way of accessing them.

A node manager is also available for management of the servers and services. It enables creating and deleting server on any node in the system. In addition, cooperations between different trader servers can be established.

More details about ANSA and ANSAware can be found in [2, 15].

2.2 Reference Model for Open Distributed Processing

If the discrete components of a distributed application are capable of exhibiting “openness” – the combination of these characteristics: scalable, accessible, heterogeneous, autonomous and decentralized [41] – then the interaction between them is known as open distributed processing.

We have seen that a distributed environment is a collection of utilities, languages and libraries which support the development and operation of distributed applications. In an ideal open distributed system, it should be possible for distributed applications developed using different distributed environments to interact. This can only be achieved if the distributed environments conform to a common conceptual model, or architecture. The basic Reference Model of Open Distributed Processing (RM-ODP) [42, 43, 44] is a joint effort by the international standard bodies ISO (ISO/IEC 10746) and ITU-T (ITU-T Recommendations X.901 to X.904) to develop a coordinating framework for the standardization of open distributed processing (ODP). The model describes an architecture within which support of distribution, interworking, interoperability and portability can be integrated [26].

To deal with the complexity of distributed systems and with the many relevant perspectives, the RM-ODP framework abstracts or views ODP systems using five view points with specific concerns. These viewpoints are:

- Enterprise Viewpoint (purpose, scope and policies)
- Information Viewpoint (semantics of information and information processing)
- Computational Viewpoint (functional decomposition)
- Engineering Viewpoint (infrastructure required to support distribution)
- Technology Viewpoint (choices of technology for implementation)

A set of concepts, structures, and rules is given for each of the viewpoints, providing a language for specifying ODP systems in that viewpoint. Specifying an ODP system

Figure 2.8: RM-ODP Viewpoints and Software Engineering

using each of the viewpoint allows an otherwise complex specification of an ODP system to be separated into manageable pieces. Figure 2.8 on Page 25 shows how the RM-ODP viewpoints can be related to the software engineering process [26].

The RM-ODP is composed of a descriptive part and a prescriptive part where the descriptive part contains the definition of the concepts, the analytical framework and notation for normalized description of (arbitrary) distributed systems, while the prescriptive model contains the specification of the required characteristics that qualify a distributed system as open. It is based on the concepts and notation of the descriptive model. In addition to the model, RM-ODP provides architectural semantics, specification techniques and formalisms to support both parts of the model.

We consider the computational viewpoint and the engineering view point in this section. They describe behaviour that is to be captured in performance models.

The computational viewpoint is used to specify the functionality of an ODP application in a objected-based and distribution-transparent manner. A computational specification defines the objects within an ODP system, the activities within those objects, and the interactions that occur among objects.

The engineering viewpoint is used to describe the design of distributed-oriented

Figure 2.9: RM-ODP Viewpoints and Software Engineering

aspects of a ODP system: it defines a model for distributed systems infrastructure. The fundamental entities described in the engineering viewpoint are objects and channels. Objects in the engineering viewpoint can be divided into two categories: *basic engineering objects* and *infrastructure objects*. A channel corresponds to binding in the computational specification. The RM-ODP engineering viewpoint prescribes the structure of an ODP system. Figure 2.9 on Page 26 shows the basic units:

- Basic Engineering Objects: these are the building blocks of the application:
- Cluster: a set of related basic engineering objects that will always be co-located:
- Capsule: a set of clusters, a cluster manager for each cluster, a capsule manager, and the parts of the channel which connect to their interfaces:
- Nucleus Object: an (extended) operating system supporting ODP: and

- Node: a computer system.

An implementation can choose to constrain the structure, for example, by allowing:

- only one object per cluster; and,
- only one cluster per capsule.

It should be noted that the RM-ODP functions are still in the early stage of specification and it is premature to expect conformance in currently available products. ANSA is claimed to be ODP conformant. Beitz and colleagues [5] investigated DCE's suitability as a support environment for *open* distributed processing by evaluating how well it meets the general requirements of the RM-ODP and how closely it conforms to the RM-ODP architecture. The investigation reveals that DCE does not provide all of the concepts and functionality needed to support RM-ODP, and hence cannot be considered as conformant with ODP. However, DCE does deliver proven technologies for many aspects of distributed processing and hence might provide a suitable foundation for future ODP-conformant systems.

2.3 Performance Issues for Distributed Systems

Distributed environments like DCE and CORBA offer a number of services and distribution transparency to make the development of distributed applications a much easier task. The composition issues are dealt with during the design of server objects: methods and procedures of computational objects are defined in the interface using *Interface Definition Language*. The distribution issues are managed when installing the servers across the network using the distribution services.

The convenience for developing distributed applications comes at some cost. The complexity of the communication mechanisms provided by the distributed environment means the application developers give up some control of the system. With the inherent complexity of distributed applications, the system may have performance behaviour that can defy intuition.

The many services of the distributed environments also add overhead. Using a distributed environments, a simple Remote Procedure Call from the application developer perspective may involve as much as 10 times more supporting RPC's necessary for carrying out the operation [11]. This cost may be a very important factor in determining the *Quality of Service* of the system, and it must be measurable before it is manageable.

Performance engineering methods and management tool are needed to help develop and manage distributed applications by providing the following services [11].

- Estimating the cost of an object's invocation as a function of resource consumption, target hardware capacity, and channel latency and contention, prior to deployment.
- Creating an efficient, pervasive measurement infrastructure that collects, transports, correlates and analyzes the performance metrics of monitored applications.
- Providing effective performance management that supplements the ODP architecture with abstractions of performance metrics and measurement interfaces, and integrates measurements and models to address system management *what-if* questions before initiating an expensive, risky course of action.

2.4 Software performance engineering

Software performance engineering (SPE) provides guidelines for applying performance modelling techniques throughout the development and management process to help ensure that the resulting software system meets its performance requirements. It can be defined as a combination of a software engineering methodology, predictive performance modelling, and methods for measuring the performance of an operational software system [37]. SPE can be integrated with design methods and can ensure performance requirements can be achieved without sacrificing other

desirable design qualities such as understandability, maintainability, and reusability [38].

Smith describes the increasing importance of the performance analysis of software systems [38]:

“Performance refers to system responsiveness: either the time required to respond to specific events, or the number of events processed in a given time interval. For traditional information systems, performance considerations are associated with usability issues such as response time for user commands. For ‘soft’ real-time systems, performance considerations relate to the ability to adequately handle the external load placed on the system. In ‘hard’ real-time system, performance concerns can become correctness issues. In these systems, failure to meet deadlines or throughput requirements is equivalent to producing incorrect results.”

Smith also notes that performance is an important but often neglected aspect of software development methodologies. In traditional software development methods, performance considerations are treated with a “fix-it-later” approach. This “Fix-it-later” approach advocates concentrating on software correctness: performance considerations are usually deferred until the later phases of the software development process (i.e. integration and testing). If performance problems are discovered at this point, the software is tuned to correct them and/or additional hardware is used.

This “fix-it-later” approach is costly and undesirable. Performance problem may be so severe that they require extensive changes to the system architecture. If the problem is only discovered and changes are made late in the development process, they can increase development costs, delay deployment, or adversely affect other desirable qualities of a design, such as understandability, maintainability, or reuseability.

A better approach is to apply Software Performance Engineering (SPE) techniques to the management of performance concerns in the development of software systems [37, 38]. SPE provides guidelines for applying well-established performance

modelling techniques throughout the development process to help ensure that the resulting software system meets its performance requirements. As an important process in the life cycle of a distributed application system, SPE should start during requirements analysis, continue during design and be applied even after the application is deployed.

We note that as the workloads on a distributed application system change over time in both types of requests and intensity, the software must also be able to be easily reconfigured to meet the new demands placed upon it. Also, SPE is needed for capacity planning to determine if the system can support increased user demand. We assume that as the demands increase, a distributed application system will need to be reconfigured either by changing the number or placement of software servers or by adding more hardware support. Quantitative models are needed to find appropriate solutions.

In order to predict software performance, SPE constructs performance models that can be solved for the indicative performance metrics. The procedures are similar to those used for conventional measurement-based performance evaluation studies (of existing systems). The conventional modelling procedure is as follows:

- study the computer system:
- construct a model (either an analytic or a simulation model):
- measure current execution patterns:
- characterize workloads:
- develop model parameters, and validate the model by solving it and by comparing the model results to the observed and measured data for the computer system:
- calibrate the model until its results match the measurement data: and,

- use the model to evaluate changes to the computer system by modifying the corresponding workload parameters, the computer system resource parameters, or both.

SPE performance modelling is similar: however, because SPE begins when the software does not yet exist, it is not possible to develop the workload parameters from the measurement data. The workload must be estimated and possibly some structure information must also be estimated.

Smith considers two models during system design:

- a software execution model; and.
- a system execution model.

The software execution model represents key features of software execution behaviours and uses an execution graph to describe the software execution in a fashion similar to a flow-chart. Information required to create this model comes from five types of data:

- workload scenarios:
- performance goals:
- software design concept:
- execution environment; and.
- resource usage estimates.

The solution of the software execution model provides workload parameters for the system execution model, which closely resembles the conventional models. The parameters of the system execution model include the visits to the system resources and the total demand placed on the resources. The constructed model is a queueing network model similar to LQMs [29]. Smith advocates the use of Mean Value Analysis and/or simulation to produce the execution model's performance estimates.

that include mean response times and queue lengths, throughput for requests and resource utilization.

As the system is being implemented the resource consumption and performance measures of the modules can be measured using monitoring. System prototypes can also be good source for measurement data. The performance models can be updated with the more accurate resource consumption estimates. Measurement data can also be used to verify and validate the models. Once the system is deployed, or its prototype is available, the measured data can be used to confirm that the performance goals of the system have been satisfied and to identify areas for improvements. The measured data of the existing system can be incorporated into the models.

If a system need to be modified, the performance impact of the proposed changes can be examined using SPE modelling techniques. The system's model can be altered to reflect the changes, and alternatives can be studied efficiently.

Distributed performance management tools can and should play an important role in the SPE process, since its use is not at all limited to managing operational systems. Early in the design cycle when the prototype of an application has been built, the performance management tool can be applied to the prototype system to collect performance measurements and to build performance models. This resulting model can be augmented to consider the not yet built aspects of the system. Also it must be noted that software development is an iterative process, because more features are like to be developed and the design may change over time in order to meet new requirements and the changes in the target environment. With the help of the performance management tools, the performance of existing systems can be better understood, and more accurate performance data can be obtained for the reusable software components. This will definitely benefit future design iterations for the software.

2.5 Summary

This chapter presented background material of this thesis. It first reviewed distributed systems and distributed computing environments, including DCE, CORBA and ANSAware. The Reference Model for Open Distributed Processing (RM-ODP) was then described. Performance issues with distributed applications especially those developed with distributed environments were noted. To ensure well performing distributed applications, Software performance engineering(SPE) methodology that was presented in the chapter should be used throughout the software life cycle. Distributed performance management tools play an important role in the SPE process. As well as being an essential tool for managing operational application systems, they can be applied early in the development cycle to obtain performance estimates for prototypes. The performance data collected for reusable software components can be used in future designs.

Chapter 3 will discuss performance models and performance evaluation techniques that are used in SPE and capacity planning.

Chapter 3

Performance evaluation: models and techniques

The quantitative evaluation of computer performance is needed during the entire life cycle of a computer system. The major quantitative methods can be divided into three main areas, namely *performance measurement*, *analytic performance modelling*, and *simulation performance modelling* [14]. Performance modelling techniques also include *benchmarking* which depends on measurements. The process of performance comparison for two or more systems by measurements is called *benchmarking*, and the workloads used in the measurements are called *benchmarks*. Benchmarks may be real workload on the systems, or in many cases synthetic programs that models real workload [18].

Performance measurement is possible once a system, or its prototype, is built, has been instrumented, and is running. Otherwise modelling is needed to predict performance, as it is the case during design. Modelling is also appropriate for load balancing and capacity planning.

The measurement of system performance is of great practical importance to computer system management. It is required to identify current performance problems and correct them, e.g., by tuning or workload balancing. Another use of performance measurement is to obtain input parameter values for and to validate performance models. The advantage of performance measurement over performance modelling is that the performance of the real system is obtained rather than the performance of

a model of the system. Among the disadvantages of performance measurement are the need for a running system, the difficulty of instrumenting application systems, it is time consuming to get measurements, monitoring overheads can affect system behaviour, and the potential difficulty of modifying the system so that the effect of system change can be studied.

Both analytic and simulation performance modelling are representations of computer system that use a network of interconnected queues to predict the performance of the system. Performance estimates typically include application response time, resource utilization, the queue lengths, and the queueing delays. In comparison to performance measurement or benchmarking, simulation is more popular, but analytical queueing models may offer the most cost-effective technique for computer system performance modelling [1]. Analytic techniques can usually provide results much faster than simulation because they are based on efficient solutions to mathematical equations. However, in order for these equations to have a tractable solution, certain simplifying assumptions must be made regarding the structure and behaviour of the analytic queueing model: simulation models also make assumptions but they are often less severe.

The most simple assumption about the analytic model is a service time homogeneity assumption that requires mean service times to be independent of the number and placement of customers in the model. A more severe restriction is that the model must be product-form or near product-form networks where each queueing centre can be analyzed separately and the partial results be combined to produced the overall results [3]. As a result, analytic models cannot capture all of the details of the system that can be built into simulation models. Nevertheless, for many types of systems carefully constructed analytic models can provide sufficiently accurate estimates of key performance parameters to draw correct conclusions [20].

Whereas analytic techniques have limitation on the range of features that can be modeled, a simulation model can be constructed to an almost arbitrary level of detail. Furthermore, whereas analytic models typically provide only mean values, simulation can provide estimates of distributions and higher moments. In addition,

dynamic or transient behaviour can be studied more easily using simulation: analytic techniques also can be used to study transient behaviours but suffer from state space explosion. However, the more detailed the simulation model the more expensive it is to build, and the more difficult to get parameters and results with adequate confidence intervals.

There are two distinct types of simulation, namely *trace driven simulation*, and the more popular *stochastic discrete event simulation*. The latter is typically a simulation of a queueing model that is driven by sequences of random numbers with user specified distribution. It is usually time consuming to develop a detailed simulation model. If there are rare events in the simulation model, the simulation may need to run for a long period of time to obtain statistical significance.

There are a number of analytic models which produce performance estimates, including *Queueing Network Model (QNM)*, *Layered Queueing Model (LQM)*, and *Petri Nets (PNs)*. The major problem with Petri Nets (PNs) is that it models system states and has an exponential growth of the state space for most realistic systems. Queueing Network Models can be used to estimate the effect of contention for devices, whereas LQMs also incorporate contention for software resources. LQMs are used in this thesis to model distributed applications.

The next section reviews the terminology of performance evaluation, which is taken from [18] and [37]. This is followed by an brief discussion of exact and approximate solution techniques for QNMs, and LQM in later sections.

3.1 Performance Evaluation Terminology

Computer systems can generally be characterized as consisting of a set of both hardware and software resources and a set of tasks, or jobs, competing for and accessing those resource. It is therefore natural to model the system by a network of interconnected queues [14, 18, 20]. A customer is a client which requests services from one or more resources. A customer class characterizes a number of customers that can be viewed as behaving identically.

Performance metrics and model parameters quantitatively describe the average behaviour of a system. The most used terms are as follows, which is taken from [18]:

- *Resources*: The resources in the system are the service centers of the system. The number of resources is denoted by M .
- *Population N* : The number of customers in the system. The customer population N is comprised of the customer population N_c of every customer class c in the system.
- *Think time Z_c* : Time between successive commands for customer class c . The term *class* refers to types of customer. All customers of a single class have the same service demands.
- *Idle time I_i* : The mean time between busy periods of the i th resource, including idle time of zero.
- *Response time $R_{c,i}$* : It is the interval between the end of a request submission and the beginning of corresponding response of the resource i for customer class c .
- *Throughput X* : The rate at which requests are completed by the system, usually given as completions per second for the system as a whole for customer class c , denoted by X_c . The throughput at a single resource i is denoted by $X_{c,i}$.
- *Service time $S_{c,i}$* : The mean service time, per visit, that a customer of class c requests from the i th resource .
- *Service Demand $D_{c,i}$* : The total service demand placed by a customer of class c on the i th resource, per customer request. It is the product of the number of visits by the customer to the i th resource, $V_{c,i}$, and the mean service time $S_{c,i}$. The largest demand at any single resource is denoted $D_{c,max}$.

Figure 3.1: An open queueing network has external arrivals and departures

- *Utilization* $U_{c,i}$: The fraction of time the i th resource is busy serving class c customer requests.
- *Resource arrival rate* $\lambda_{c,i}$: The rate at which class c customers arrive for service at the i th resource .
- *Server type*: This can be: *single server*, *multi-threaded server*, or *delay centre*.

3.2 Queueing Network Model

The simplest way to classify a queueing network model (QNM) is either open or closed, depending if there are external arrivals and departures, as shown in Figures 3.1 on Page 38 and in Figure 3.2 on Page 39. A model that considers customers that arrive and depart from the system is defined as an *open model*. A *closed model* has a fixed number of customers which circulate through the system with a delay that each customer observes before submitting its next request (*think time*). In an extreme case when the number of customers in a closed model becomes very large, it can behave like an open model.

Figure 3.2: A closed queueing network has no external arrivals and departures

Many performance estimates can be evaluated for QNMs using the operational laws [18], which use some simple relationships that do not require any assumptions about the distribution of service times or inter-arrival times. For example, the law of asymptotic bounds can be used to calculate the upper and lower bounds on throughput, X , and response times, R , of a QNM, as shown in Equations 3.1 and Equation 3.2.

$$R \geq \max \{D, N D_{max} - Z\} \quad (3.1)$$

$$X \leq \min \left\{ \frac{1}{D_{max}}, \frac{N}{D + Z} \right\} \quad (3.2)$$

where D is the sum of service demands on all devices in the system, D_{max} is the service demand on the bottleneck device, which is the device with the highest demand, N is the number of customers in the system, R is the system response time, X is the system throughput, and Z is the customer think time.

The asymptotic bounds on throughput and response time give the most optimistic evaluation of the system, and they help to determine whether it is feasible for the system to support its proposed workload.

The bound analysis gives only optimistic and pessimistic performance estimates. A more comprehensive approach is to solve the QNMs analytically for steady-state

performance estimates for a given workload. The solution techniques are classified as either exact or approximate, according to the mathematical accuracy of their solutions with respect to the analytic model's assumption.

Mean Value Analysis (MVA), developed by Reiser and Lavenberg [27], is an exact technique for predicting the performance of QNMs. As the name implies it gives the mean performance measures such as utilization, throughputs, mean queue lengths, and mean response times. The type of queueing network model solved using MVA satisfies product form assumptions. This means that the network should satisfy the following conditions [18, 20]:

- job flow balance:
- one-step behaviour:
- device homogeneity; and,
- all service centres are either fixed-capacity service centres or delay centres with exponentially distributed service times.

The exact MVA algorithm was first developed for product form closed networks with fixed rate and queue length dependent rate servers only, but has been extended to cover a broader range of product forms of state dependent service rate. MVA applies to networks with a variety of service disciplines, such as processor sharing (PS), first-come-first-served (FCFS), and last-come-first-served pre-emptive resume (LCFSPR). The input parameters for a single class MVA are:

- number of customers N :
- mean think time Z :
- number of resources M :
- mean service time S_i per visit to the i th resource; and,
- mean number of visits to the i th resource, V_i .

Mean value analysis is a recursive algorithm. Computation of performance with N customers in the network requires knowledge of performance with $N - 1$ customers. This requires that the QNM be solved for $0, 1, 2 \dots N$ customers successively. Also, within each enumeration the equations must be solved over all resources. For small values of N , this procedure is not computationally too expensive. However, for multiple class networks having a large number of closed classes or a few closed classes with large population, the computational complexity, in terms of space and time, becomes prohibitive [14, 18].

Several approximate solution techniques of queueing networks have been developed to reduce the computational cost of MVA. Schweitzer's [35] approximation is one of the more widely used [18], as well as the Linearizer [8]. The Schweitzer approximation technique avoids the recursion in MVA by using an iterative solution. It appropriately estimate the queue lengths with N customers and computing the response times, R_i and throughput, X . The R_i and X values so computed are used to recompute the queue lengths, N , and if the previous estimate is good, the new computed value will be close to the estimate. The Linearizer produces greater accuracy under the same circumstances but at a higher computational cost.

Queueing network models only take into account the hardware resources as servers. This is adequate for traditional centralized computing systems. To model distributed systems, however, the QNMs must extended to take into account the queueing delays at the software processes as well. Examples of the extended QNMs include *Layered Queueing Models* [28, 29] and the very similar *Stochastic Rendezvous Networks* [45]. This thesis uses Layered Queueing Model, which will be described in the next subsection.

3.3 Layered Queueing Models

Layered Queueing Models (LQM) [28, 29] are an extension of queueing network models developed to address systems of cooperating processes in a distributed and multicomputer environment. LQMs include characterization of the architectural in-

formation of a system, the operating system processes that contribute to the system, their relationships, and their use of hardware resources. Performance evaluation techniques for LQMs, such as *Stochastic Rendezvous Network* algorithm [45] and *Method of Layers (MOL)* [28, 29], take into account contention and thus queueing delays at both hardware resources and software. In addition to questions addressed by QNMs, these models can be used to consider the impact of moving a serving process from one node to another, of replicating a serving process, or changing its level of multi-threading.

Processes that can be said to have statistically identical behaviour form a group or class. Groups use devices but they can also request services from and provide services to other groups. A group can act as a *pure client* by only requesting services from other groups, or as a *pure server* by only providing services to other groups, or as both a client and server, where it receives requests from its clients and process them by generating its own requests to other server groups. Processes in each group have service entries that represent particular services and characterize the resource demand for serving a client's requests. A pure client has one and only one service entry, which is never called by other processes. A serving process can have multiple entries.

Figure 3.3 on Page 43 shows a simple LQM. Each parallelogram is a group of one or more processes and each circle is a device. Service entries are shown as the framed labels within each group. Directed arcs indicate requests for service or visits from the calling group to the serving group or device. Associated with each arc is the number of visits. In this example the server process has two entries, S1 and S2. The two client groups, A and B, can suffer queueing delays when visiting the server process and the device, CPU2.

Model input parameters for LQMs include those of closed separable queueing network models. In addition, the average number of visits a process makes to other processes, and process scheduling discipline must also be specified. The parameters for LQMs are summarized as follows.

- Process identifiers, threading levels, and think times of operating system pro-

Figure 3.3: A Layered Queueing Model (LQM)

cesses:

- Scheduling policies of devices: and.
- For each method (m) of each interface (i) of each process (c):
 - the average number of visits ($V_{c,i,m,k}$) to each device (k) in the network:
 - the average service time ($S_{c,i,m,k}$) per request at each device (k) in the network: and.
 - the average number of visits (V_{c,i,m,s,i_2,m_2}) to each method (m_2) of each interface (i_2) of each of its server processes (s).

The *think time* of a client process is the average time it spends waiting for input from users between consecutive requests. Note that the service times at the methods of serving processes are not specified. These values are a function of the dynamic system state and must be estimated by performance evaluation techniques.

Several types of client-server process interactions can be represented by LQM, with each type of interaction implemented as a type of server with a particular scheduling discipline. The server types allowed in an LQM are as follows.

- *Rendezvous*: It is a first-come-first-served process with two phases of service. A caller waits until the first phase is complete before being released and continuing with their own execution. But the server cannot begin to provide service again until it completes its second phase.
- *Multiple-entry*: FCFS Rendezvous server representing processes that provide different services.
- *Multi-entry Multi-server*: Many identical serving processes share a single queue of customers.
- *SYNC*: Provides a synchronization point between two groups of processes.

Any of the device scheduling and service time distributions permitted in MVA are permitted for LQM. These include delay, process sharing (PS), and FCFS with

exponential service times. Interaction with servers can be synchronous or asynchronous [36].

LQMs can be solved by Method of Layers (MOL), which is based on approximate MVA (Linearizer) [28, 29]. The MOL divides an LQM into two type of models:

- a *software contention model* which describes the process architecture and is used to predict software contention delays; and.
- a *hardware contention model* which describes each group's device usage and is used to predict device contention delays.

The results of the two models are combined to provide performance estimates for the system.

In addition to the metrics provided by MVA for device performance, the MOL produces metrics about each process group. These mean measures include: the average response time R , utilization U , throughput X , and idle time I of a group g process, the average residence time a group g process when visiting group h , the average time a group g process spends acquiring service from other groups, and the average time a group g process spends acquiring service from devices.

Building an LQM for a system requires information about both devices and the process architecture of the system as listed below.

- *Device information*: The schedule discipline of the device and the service rate.
- *Group information*: Group identifier, group type (pure client or server), the server type, the list of entries, the devices used by the group, and group's priority.
- *Entry information*: Number of entries, and phase information for each entry: the mean phase service demand, mean number of visits to other group entries and service demands on devices.

The visits made by the processes can be obtained from a trace of an existing system [17, 16], and by analyzing the individual process based on estimates of conditional branching probabilities of its statement blocks [37]. This information can be

used to estimate average service demand. Measurement based characterizations of components have also been used for estimating service demands and visits [48]. If the system being modeled already exists, system measurement tools, such as profiler and monitor, might also be used to obtain the values of input parameter for the LQM. Performance measurement for distributed applications is an active area of research, and it will be discussed in Section 3.5 on Page 48.

3.4 Distributed Application Performance Metrics and Management

Distributed application performance management (DAPM) is a form of continuous software monitoring. It is responsible for recognizing whether distributed applications are satisfying their performance requirements and locating the sources of performance problems. Performance metrics that can be used to characterize software processes and the requirement for a DAPM system have been defined by Rolia [30].

The requirements for a DAPM system are [30]:

- help to recognize and locate the sources of performance problems quickly;
- maintain a temporally consistent and recent global view of application performance;
- be integrated with network management to correlate application load and responsiveness with network performance;
- cause minimal network and node overhead and be scalable, so that large networks with many applications can be managed, even in the presence of unmanaged applications that are not under the control of the DAPM system; and,
- collect sufficient information for capacity planning and performance modelling.

DAPM augments the performance services of network management which in general is unable to correlate network load directly with specific applications.

A typical performance requirement for a system supporting an on-line application is as follows: the system must be able to complete λ requests of type α per second with ninety percent of the requests having a response time less than x seconds. If the requirement is satisfied the application and the network supporting the application can be said to offer a high quality of service to the application's customers.

When an application is not achieving its performance requirements, the cause of the problem must be identified so that corrective actions can be taken. Potential problems include: high contention for a particular node's processor or input-output subsystems, network traffic congestion, and software bottlenecks. A software bottleneck may arise when a software server's utilization is greater than that of any devices it uses. The software server can become the cause of unacceptable queueing delays even though the utilizations of the underlying devices do not indicate high contention.

Consider a distributed application that is to be managed by the DAPM system. Such an application is defined as a managed application. A request's *response time* can be defined as the end to end latent time required to serve the request. This time includes all queueing delays as well as service demands. When a request is not meeting its performance requirement, the DAPM system must be able to identify the software and hardware bottlenecks most responsible for the problem. It must also be capable of doing so even in the presence of unmanaged applications on the network. The software performance metrics are chosen to simplify this task.

Since a process can provide many operations which have significantly different performance characteristics, metrics for each operation are necessary. The end to end response time of an operation R and its throughput X provide useful information, but are not adequate for pinpointing the source of the performance problems. This is because the operation response time includes both local node computation component and possibly a remote procedure call component that accounts for its own RPCs. So the average response time of each of a process's operations need to

be further partitioned into four parts that can be used to determine whether the process is the source of a performance problem [30]:

1. *waiting time* W of the operation at the process to be served;
2. *node response time* R^{node} which is the time required to satisfy the device service demands at its own node;
3. *remote response time* R^{remote} which is the time required to accomplish remote procedure call made by the operation; and.
4. *network latency time* $R^{latency}$ spent by the operation when receiving RPC packets from the network.

The values of R , W , R^{remote} , and $R^{latency}$ can be obtained via instrumentation and used to compute R^{node} :

$$R^{node} = R - R^{remote} - R^{latency} \quad (3.3)$$

Other metrics such as average queue lengths can be deduced from the response time and throughput measures for each operation using Little's Law [18]. For example the average queue length Q at the operation is defined as

$$Q = XR. \quad (3.4)$$

The software performance metrics presented in this section can be used to recognize and pinpoint the source of performance problems in an application distributed across a network. The metrics support a scalable DAPM system that permits a process to determine whether it is the likely cause of a performance problem without correlating its measures with those of its clients and servers.

3.5 Performance Measurement for Distributed Applications

The software performance management tasks of application design, deployment, bottleneck analysis, and capacity planning require the collection, modelling and

analysis of workload data. The performance measurement data have at least the following uses in software performance engineering [37]:

- Incorporating measurement data from *existing systems* in to the software performance engineering (SPE) models.
- Using experience gained by measuring prototypes to derive performance specification.
- Updating performance specifications (estimates) with measurements of the evolving software to improve model precision.
- Verifying and validating the SPE models.
- Evaluating software performance and identifying areas for improvements.

Computer system performance measurements involve monitoring the system while it is being subjected to a particular workload. Mainframe environments have benefited from comprehensive monitoring systems for decades. Unix systems have had less support but it is improving.

The conventional methods of performance measurement for centralized sequential programs are not adequate in the distributed environment because they address only the performance of individual programs on a single processor. In distributed application environments, software processes can act both as clients and servers to other processes. These processes are virtual resources that must be monitored by distributed application monitoring systems. In this section examples of distributed measurement system are presented.

Building performance measurement system for distributed application involves the collection, interpretation, and manifestation of information concerning the interactions among processes. This requires that the monitor itself must be distributed and should consist of several components. Jain [18] described a layered architecture with seven layers. The lower three layers are:

1. *Observation*: Gathers raw data on individual components of the system (device and software servers):
2. *Collection*: Collects data from various observers. It is possible to have more than one collector on large system: and.
3. *Analysis*: Analyzes the data gathered at various collectors. It may consist of various statistical routines to summarize the data characteristics.

Higher layers include *presentation*, *interpretation*, *console*, and *management* that mostly interact with human operators of the system. A monitor may consist of multiple component from each of the layers. The observers can be implemented using software, hardware, or firmware. Collector, analyzers are usually implemented in software.

The hierarchical architecture of monitors has been implemented in a number of distributed measurement systems. Yang and Miller [47] used a structured and automatic approach. It involves unifying performance information into a single, regular structure that reflects the structure of programs under measurement. A complete picture of the program's execution can be presented at different levels of details in a hierarchical model including: *program*, *machine* or *operating system*, *operating system process*, *procedure*, and *primitive activity levels*. A measurement hierarchy with corresponding hierarchical levels allows the performance of a distributed application be measured by observing its behaviour at different levels of detail. Automatic guidance techniques were also developed to direct users to location of performance problems.

A recent trend is to use an integrated approach in measurement system development. The *JEWEL* performance evaluation tool was developed to be a general purpose measurement system that offers an integrated set of reusable, flexible and adaptable tools with well defined interfaces [19].

Network management protocols, such as *Simple Network Management Protocol (SNMP)* and *Common Management Information Protocol (CMIP)* [39, 33], have been considered to provide services for application management. The CMIP was

developed in 1988, but until recently, has not gained widespread use. It is, however, considered by many as the long-term solution for distributed systems management. SNMP was developed as an "interim solution" for network management over Internet, and has become the *de facto* standard for network management. Software subsystems from operating systems to databases are implementing SNMP access for distributed management.

Another integrated performance measurement and modelling systems is the *Distributed Measurement System* (DMS) [11], under development by Hewlett-Packard for the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) Special Interest Group (SIG) on performance. The DMS attempts to provide a systematic approach to performance measurement and modelling that benefits application design, deployment and management in Open Distributed Processing (ODP) [42, 43, 44]. It is a software-based measurement infrastructure that defines standard performance metrics, instrumentation, and interfaces.

DMS provides an integrated view of application resource consumption across heterogeneous nodes and network channels that node-based tools cannot. It provides an extensible architecture for integrating disparate performance measurement interfaces from a node's nucleus object (operating system) and channels (networking) with the distribution infrastructure. It also provides efficient mechanisms for controlling, collecting and transporting performance data independent of underlying distribution mechanism.

DMS defines standard application programming interfaces (APIs) to ensure pervasive support for performance metrics measurement. The data collected by DMS include statistics for metrics such as the number of times each service is used, service response times, and process resource consumption. Instrumentation is incorporated into programs or libraries to calculate performance metric. Figure 3.4 on Page 52 (reproduced from [11]) shows the DMS architecture and DMS objects which include *sensors*, *observer*, *collector*, *analyzer*, *presenter*, *interpreter*, and *controller*.

DMS *sensor* objects are instances of a general performance metric for a function. The sensors calculate and buffer primitive statistical quantities such as counts.

Figure 3.4: Distributed Measurement System Architecture

summations, or interval times, but defer the computation of more complex statistics such as moments to higher-level DMS objects. A sensor's collection granularity, and thus overhead, is controlled by specifying an *information level*, which control the statistical detail of the collected data.

An *observer* object resides within each instrumented process and provides a sensor control point. It exports a sensor access and control interface named the *Performance Measurement Interface (PMI)*. The observer supports sensor registration and unregistration transmits intervalized data generated by sensors to the collector object. Data are aggregated over pre-defined time-interval called *process period*.

A *collector* is a network-node level object that controls sensor via the PMI and perform local node data management. There is one collector per node. It provide sensor access and control to higher level object through the *Collector Measurement Interface (CMI)*, and accumulates sensor data pushed by all observers on the node through the *Collector Data Interface (CDI)*.

An analyzer object applies statistical routines to compute the distributional characteristics of the collected data, correlates data from all application elements, and prepares data for expert system or human analysis.

An DMS prototype was implemented using OSF/DCE. Sensors and observer were implemented in the DCE runtime library (RTL). The collector was implemented as a daemon process that communicated with all the observers on the node via IPC using the PMI and CDI. The analyzer was implemented as a daemon and it communicated with all the collectors in the network via RPC. The minimum process period supported by DMS is currently 5 seconds, and an interesting feature of DMS is that the process periods can be different for each process. This enables frequently used processes to report metrics more often than lightly used ones.

Information collected by DMS can be used to support the performance modelling of distributed applications. Unfortunately, it does not provide all the information we need for Layered Queueing Models.

DMS counts number of times a process requests services of other processes. The measurements are maintained on a (calling process, (serving process, service)) basis.

instead of ((calling process, service), (serving process, service)) basis. It is feasible to maintain these counters, but when DMS was developed there was a concern that the resulting volume of monitor data could cause too much monitoring overhead. A more recent study has shown that collecting the more detailed information is feasible [12]. A more difficult problem is that most operating systems do not enable resource consumption measurements for individual services. This requires a new and enhanced measurement system, or statistical analysis techniques [32] to calculate the resource consumption for each service.

The model building techniques developed in this thesis are therefore based on DMS but have added capabilities to address the issues mentioned above.

3.6 Summary

This chapter reviewed performance evaluation techniques and distributed application performance management. The major quantitative methods can be divided into three main areas, namely *performance measurement*, *simulation performance modelling*, and *analytic performance modelling*. Analytic performance modelling offers the most cost-effective techniques for computer system performance analysis, though simplifying assumptions must be made regarding the structure and behaviour of the analytic model.

There are a number of analytic models which produce performance estimates, such as *Petri Nets*, *Queueing Network Model*, and *Layered Queueing Model (LQM)*. LQMs are best suited to model distributed applications, as they model contention for software resources as well as contention for devices. LQMs can be either solved with *Method of Layers (MOL)*, and/or simulated, to produce performance estimates.

Performance evaluation techniques are needed in distributed application performance management, which is responsible for recognizing whether distributed application are satisfying their performance requirements and locating source of performance problems. Performance metrics have been defined and can be used to

recognize and pinpoint the source of performance problems in an distributed application.

The software performance management tasks of application design, deployment, bottleneck analysis, and capacity planning require the collection, modelling and analysis workload data. A recent trend is to use integrated approach in measurement and modelling system development. In this thesis, model building techniques are developed to extend the capabilities of a DMS-like monitor.

Chapter 4

Information and Infrastructure for Model Building

This chapter describes the information that is needed for model building, and the infrastructure that supports such model building. A high-level overview of the model-building system is given in this chapter. In the following chapters, more detailed descriptions are given of the model building techniques.

The model used in this thesis is the Layered Queueing Model (LQM), which is described in Section 3.3 on Page 41. Layered queueing models like any other software performance models have two aspects, namely structure and resource demand parameters. The structure describes the architecture of the system, and includes a characterization of the operating system processes that contribute to the system, their relationships, and their use of hardware resources. Parameters detail the number of visits between software components and service time requirements of requests for service as they flow through various components and visit devices. A model building technique must then address the two aspects.

4.1 Information for Model Building

Each program that runs as part of a distributed application has zero or more *interfaces* that define the services it provides (if any) to other programs. The *services* are also called *operations* or *methods*. The operations implement the distributed

application's subprogram in the same manner as in a non-distributed application. Operation interfaces are typically defined by application developer using application development tools, for example in DCE and CORBA, using an interface definition language (IDL) and its compiler. The IDL for an interface looks much like a typical header file (".h") for the C language. Each interface has a unique identifier: in DCE this is referred to as *UUID*. Processes *export* interfaces that they support and *import* interfaces which contain operations that they use. *Non-serving* processes are purely client processes that do not provide services to other application processes and often act as user-interface. Location transparency in distributed systems is achieved using a name server. In DCE this is referred to as the Cell Directory Service, and in CORBA it is the Naming Services. A name server permits connections to be established at the time of execution between (client) processes and the interfaces of serving processes. The interaction between client and server processes is implemented by middleware support for communication. For example, CORBA supports the following types of inter-object communication.

- Synchronous interactions, for remote procedure calls (RPC) where the calling process must send the input parameters and wait to receive the output parameters of the call before continuing its processing.
- Asynchronous interactions, for a non-blocking message send without waiting for reply (using the *oneway* keyword).
- Deferred synchronous interactions, where a calling process sends a message then later checks for a reply (caller is not blocked until it tries to retrieve the reply). These are only permitted with Dynamic Invocation Interface(DII). It must be noted that the DII is far less common than Static Invocation Interface (SII), in the latter case all the methods are specified in advance and are known to the client and server.
- Best-effort, uni-cast communication.

In DCE, blocking RPC service is available, and other types of communications can be mimicked by application developers using DCE thread package. In this thesis, we only consider synchronous (blocking) RPCs.

In order to construct a predictive model of an operational system, it is necessary to collect information about the system's configuration and its resource consumption. This means the need to identify the applications that currently consume resources, the processes that participate in each application, and the connections between these processes. Furthermore, it is also necessary to capture information about the operations of processes, the visit counts between them, and the amount of physical resources used by each operation.

The above metrics describe the parameters of Layered Queueing Models, which models contention for physical resources and for software servers. More details about LQMs are presented in Section 3.3 on Page 41, and only the model input parameters are summarized below:

- Process identifiers, threading levels, and think times of operating system processes:
- Scheduling policies of devices: and
- For each method (m) of each interface (i) of each process (c):
 - the average number of visits ($V_{c,i,m,k}$) to each device (k) in the network:
 - the average service time ($S_{c,i,m,k}$) per request at each device (k) in the network: and,
 - the average number of visits (V_{c,i,m,s,i_2,m_2}) to each method (m_2) of each interface (i_2) of each of its server processes (s).

The *think time* of a client process is the average time it spends waiting for input from users. Note that the service times at the methods of serving processes are not specified. These values are a function of the dynamic system state and must be estimated by performance evaluation techniques.

4.1.1 Information Sources for Model Building

Structural and resource information needed for model building can be found at every stage of the lifetime of an application, from design through to management.

Structural information can be developed from software design and the source code: Smith provides techniques for deducing model structure based on design and code walkthroughs [37]. Hrischuk [17] presents a technique for automatically generating model structure from traces of software execution. With user request described as performance test cases, the technique assumes an executable design and/or implementation environment that can emulate the request. A trace is generated as requests flow through the methods of the objects of the application. The trace is then used to deduce the blocking relationships between software components, which are reflected in a performance model. For this approach, tracing must ensure that all work caused by a request be identified in the trace with the request, communication mechanism between methods be described.

Once the distributed application is deployed, a distributed measurement system and/or tracing mechanism can be used to track requests through an application [4, 11] and deduce the structure relationship between processes.

Resource demand information can also be found at each stage of the development of an application. At design stage, resource demand estimates have to be made based on previously collected information. *Resource functions* [48] can be used for this purpose. They describe the functional relationship between the invocation of a software component and its use of processor, input/output devices, and network resources. The identification of resource functions can be a major undertaking, but they are reusable as long as they are reused in the same environment. If device speeds change, scaling factor can be applied to the resource function to estimate resource demands on the new device. If a device with a different architecture, compiler, or operating system is considered, the existing resource function can only give rough estimates for the resource used by the component in the new environment. The component suitable for characterization include libraries, operating system and middleware services, and frameworks.

Once an application is deployed, resource consumption can be measured by monitoring tools. Techniques and reference architecture have been proposed by the MANDAS project [4] to acquire and manage this information. Operating systems like UNIX monitor process resource consumption, but this cannot be mapped back directly to object and method resource. Application monitoring systems, like DMS [11], are developed to measure application specific information. With these monitors it is possible to count the number of times methods at the interface of a process are used. Unfortunately, there are some model parameters that we require but that cannot presently be measured directly. Specifically resource consumption, such as CPU usage and I/O counts, is still not associated with individual interface methods, rather it is acquired for the operating system process as a whole. Rolia and Vetland [32] consider statistical techniques for estimating these values, based on measured data from a DMS [11] style monitoring system. The techniques are described in more details in Section 4.4.2 on Page 71.

Model building at the management stage needs support from distributed monitoring systems such as DMS, and statistical techniques to estimate parameter values that cannot be measured directly.

4.2 Infrastructure for model building

The capture of parameters for performance models is in itself an important issue. Current operating system monitoring tools can not provide the level of details that the model building system needs. Additional instrumentations are needed to collect parameters for LQMs. This therefore requires the support of a distributed monitoring system in a form of continuous monitor.

A distributed monitor is responsible for monitoring the behaviour of management objects in distributed systems. It can be described as a sequence of virtual layers [18]. A monitoring system has been described in the MANDAS [4] reference architecture for distributed systems management. It consists of management agents and management information repository. Monitored management data are collected

Figure 4.1: Infrastructure for Model Building

by or from management agents and stored in the management information repository. Subsequently, management data may be retrieved from the repository for analysis. Results may be returned to the repository for other uses.

Another integrated monitoring and modeling system is Distributed Measurement System (DMS) [11], which is describe in Section 3.5 on Page 48. The DMS is a software-based measurement infrastructure that defines standard performance metrics, instrumentation and interfaces. The DMS architecture was designed in concert with distributed application modeling requirements, and it has been realized in an objected-oriented prototype based on the OSF Distributed Computing Environment (DCE).

The model building infrastructure use in this thesis is based on DMS. The distributed monitoring infrastructure is a software architecture that facilitates the routine measurement of performance metrics on distributed application processes. The model building system has three main components, as shown in Figure 4.1 on Page 61.

- An instrumentation package that collects measurement data from application processes and forwards the data to a performance data server.
- A performance data server that stores captured data and estimates model parameters that cannot be measured directly.
- A model builder that gathers information from the performance data server and constructs the LQM.

There are two possible mechanisms for software instrumentation: (i) instrumented system software and (ii) instrumented source code. The use of instrumented versions of middleware runtime system and communication libraries is very appealing since performance data can be obtained easily without modifying the application program. However, this requires vendor participation, standardized metrics, and does not provide an easy mechanism for the user to selectively monitor portion of the application. Source code instrumentation works better in these respects: it does not require vendor support, and a variety of routines and macros can be placed by an application programmer at any point within the application's code to collect exactly what the user deems necessary.

Source code instrumentation is used in this work by adding instrumentation code to the programs so their corresponding processes report the data needed for model building. Each process periodically reports a summary of its metrics to a performance data server. Typical reporting intervals may be in the range of minutes. It has been shown [12] that monitoring overheads with this paradigm can be expected to introduce an overhead for most monitoring scenarios below several percent for both processor and network loads.

One of the requirements for a continuous monitor should be to minimize both local overhead at each node and communication over the network, otherwise the total overhead of the monitoring system will deter its use [30]. It is important to distinguish between local monitoring overhead on a node, and overhead caused by management information being communicated over the network. For the purpose of scalability, a primary goal of a distributed application management system should be to minimize its network communication. Local overhead has a lesser impact on scalability. However, if the monitoring overhead for each managed process is too high, application performance or node capacity will suffer. Management roles should be allocated to resources and the management system in a manner that minimize total overhead.

The instrumentation plays the roles of observer and collector. The performance data server plays the role of analyzer and is allocated to some other agent process. To minimize local node monitoring overhead caused by the monitoring system, each software resource should observe its own events and deduce its own metrics. This should decrease the number of local communications required to formulate the set of metrics that must be forwarded to an analyzer.

The “polling” mechanism used in network management is not suitable in a software environment because the management agent may suffer significant queueing delay before it receives service from a busy software resource, and data needed to do the statistical analysis may be lost. It is also likely that the agent may poll idle processes unnecessarily. Instead, a resource should communicate performance metrics periodically if required by the model builder, or only when they have exceeded threshold.

As mentioned in Section 4.1.1 on Page 59, there are some model parameters that we require but that cannot presently be measured directly, such as the CPU usage and I/O counts for individual interface operations in a server process. Section 4.4.2 on Page 71 describes statistical techniques that has been proposed [32] for estimating these values. Such statistical techniques are supported by the performance data server.

The model building tool makes requests to the performance data server to gather information needed to generate the LQM automatically. The Method Of Layers (MOL) [29] can then be used to solve the LQM in order to estimate contention delays.

In the following sections the instrumentation package and performance data server are described in more detail. The model builder is described in Chapter 5.

4.3 Application Program Instrumentation Package

The purpose of the instrumentation package is to measure and collect the information required for model building and forward it to the performance data server. Much of the description in this section also appeared in [25].

The package includes a variety of routines and macros to be placed by an application programmer at certain points within the application's code. The instrumentation of system software was not possible because it was not possible to get access to DCE source code. Though this solution is not ideal, it has been adequate to gather further information easily about software interactions that are needed to test our model building approach. The instrumentation described in this section should eventually be integrated with middleware runtime systems, code generators (such as IDL compilers), or communication class libraries (such as the Adaptive Communication Environment ACE [34]). This approach hides instrumentation details from application programmers and provides more information about events in the middleware.

The behaviour of the instrumentation is considered next. During the start-up period, an instrumented process invokes a routine called `observer_init()` to instantiate and fork an observer thread. The observer thread communicates with the underlying operating system to obtain information that identifies the instantiating process (called the process ID) and the node on which it is executing (called the host Internet Protocol address). Each interface of the process is defined by its *protocol*

sequence. Borrowing from DCE terminology, a *protocol sequence* is a combination of transport protocol identifier (TCP/IP), node address (an IP address), and a port (socket or endpoint) identifier for an interface. For a DCE application, this information can be obtained by querying the binding handles of each imported and exported interface.

The observer thread has a life-cycle of sleep-measure-report. To manage observation overhead, the observer reports measured data to the performance data server with controllable frequency. The observer thread “sleeps” for an interval specified through the `observer_init()` routine. After it “wakes,” it measures the physical resource usage of the process and gathers metrics collected for the operations of interfaces since the previous report, and sends them to the performance data server.

Examples of physical resource requirements include the CPU time, physical read and write counts, and the network communication bandwidth used. The data collected about operations is event-based and contributes to running sums for visit counts and response times. At the time to report, the observer aggregates these measures for the interval and reports them. In this way the reported data is *intervalized* and reported. Note that this is not a pure sampling technique in which the current state of the process is simply observed at the instant of reporting. Important information would be lost with such an approach.

Ideally, hardware resource consumption values would be available for each operation of each interface in each process. Unfortunately, physical resource demands are usually reported by operating systems on a per-process basis. For these systems, statistical techniques can be used to help estimate the measures on a per-operation basis. This is described further in Section 4.4.2. Operating systems with kernel-based threads typically report measures for each thread separately, and may in the future be better able to support the characterization of physical resource consumption at the operation level.

The information needed to characterize operations of interfaces is collected by instrumentation macros in the program code. A block of code, representing an operation, that needs to be measured, is surrounded by a pair of macros that updates

the visit counts and measures response times for the operation. The instrumentation macros also capture the relationships and visit counts between the operations of process interfaces as follows.

Each process (c) exports a set of interfaces (E_c) with their protocol sequences (P_c), and imports a set of interfaces (I_c) with their protocol sequences (P_s). Processes that do not export any interfaces are pure clients. We assume that they are characterized by one virtual interface. A block of code that implements an exported operation of an interface is called an input operation (the name is derived from the process's own point of view), and is surrounded by the following pair of macros:

```
S_IN_OP /* Start INput OPeration */  
E_IN_OP /* End INput OPeration */
```

Similarly, a block of code that defines an imported operation of an interface is called an output operation and is surrounded by the following pair of macros:

```
S_OUT_OP /* Start OUTput OPeration */  
E_OUT_OP /* End OUTput OPeration */
```

The macros characterize operation invocation counts and response times. The pairs also interact with the operating system and/or middleware to obtain information about which processes are connected. This information is discovered by exploiting imported and exported interface protocol sequence information.

By uniquely identifying each instance of an interface using its *protocol sequences*, we are able to determine which of an application's processes are connected together. The invocation counts between interfaces help to characterize the frequency with which they communicate.

All information collected by the macros is stored in a process-wise global data buffer that is shared among the threads of the process. When the observer thread "wakes," it intervalizes the measurement data stored in the buffer and sends it to the performance data server. To minimize communication costs, only information on operations with non-zero visit counts are reported.

A point worth mentioning is the need to protect the process-wise global data buffer against simultaneous access. A mutual exclusion (mutex) mechanism is used to protect the buffer from being accessed by the instrumentation macros while it is being used by the observer thread. This might cause the problem of blocking other threads in the process while the observer thread is copying the data to its own storage arrays for delivery to the performance data server. To minimize the potential impact of the critical section a secondary process-wise global data buffer is provided for each process. The observer maintains a lock on the buffer's mutex only while it is switching pointers of the main and secondary data buffers. The observer can then take its time to view the buffer with the instrumentation macros using the second buffer for data storage.

4.4 Performance data server

The performance data server supports storage for the intervalized process data gathered by the process instrumentation. The data server also performs data reduction and some statistical analysis needed to derive parameters for LQMs. In the following subsections we describe the services provided by the server and how they are used.

4.4.1 The Interface

The performance data server provides an interface that supports the storage, analysis, and reporting of performance data for monitoring and model building. It is a multi-threaded server that supports more than one access method. The interface can be accessed via a TCP/IP socket interface or a DCE Remote Procedure Call (RPC) interface. The two access methods are provided to enable the data server to operate in heterogeneous environments. The performance data server has at least two threads, one to handle DCE RPC, and the other to handle requests via a TCP/IP socket interface. A default well-known port number is used to create a socket. Clients use this port number and the host name of the performance data server to connect to the performance data server. Currently the socket interface

is only able to support one connection at a time, and hence one client at a time. At present this is not a severe limitation since the observer threads and the model builder use the DCE interface, which supports multiple callers. Access to the data server's files are protected by mutual exclusion.

The performance data server must be running before data gathering can begin. It provides ten services to clients. They can be categorized into three types of services.

- Management.
- Data storage
- Data reporting.

Each category is discussed in the following subsections.

Management

Management services are used to initialize data files, to delete data, and to remove the performance data server from the system. They are invoked by management applications only, *not* by managed applications. The `init()` service deletes all existing performance data. If the data files do not exist, the files will be created. `del_period()` deletes all intervalized data in the file belonging to a specified process for a specific period of time. `del_all()` is similar to `del_period()`, but applies to all processes of all applications. `shutdown()` stops the server but leaves the data file and supporting files intact. `uninstall()` removes the data file and any supporting files.

Data storage

The data storage service permits observers to store measurement data. Observers interact with the performance data server using the `put()` service. It enables observers to store the collected data in files. The data can then be obtained by management applications using the data-reporting services supported by the performance data server. Examples of stored data include:

- Time interval for the reported data. in the form of start time (t_s) and end time (t_e).
- Application, process, and interface identification information. A process (c) in a managed application can be identified by the following attributes.
 - The IP address or name of the host machine.
 - The process id and program id.
 - The application id and type.
 - A list of listening ports.

Each export interface (E_c) has the following attributes.

- The interface identification, such as DCE UUID.
 - The operations that are available in the interface. Each operation (m) is identified by its identification within the scope of the interface.
 - Connectivity information. A operation in the export interface (E_c) may require services from operations in the import interfaces (I_c), which are reported in the form of `host IP`, `interface id`, `operation id`.
- Metrics for operations of imported and exported interfaces. The overall process resource usage for a process period include the following attributes.
 - Threading level.
 - number of physical I/O's.
 - CPU usage.
 - Additional information such as page faults.

Also for each operation (m) in the export interface(E_c) of the process (c), the following information is reported.

- Total number of requests started in the period ($V_{c,e,m}$).

- Sum of operation response times ($R_{c,e,m}$), and sums of squared ($R_{c,e,m}^2$) and cubed response times ($R_{c,e,m}^3$).
- The visit counts (V_{c,e,m,s,i,m_2}) to each operation (m_2) in each of the imported interfaces (I_c) of each of its serving processes (S), and the sum of the corresponding response times (R_{c,e,m,s,i,m_2}).

Data reporting

Reporting services provide data for management applications. The services all take a time period as input, in the form of start time (t_s) and end time (t_e).

The `get_applications()` service returns a list of identifiers for applications that were active within the specified time period.

The `get_host_process_ids()` service takes as input an application identifier and returns the list of application processes that reported data during the measurement period. The information about each process (c) includes its host name, process ID and program ID. This information is needed as input to the `get_observation()` and `get()` functions.

The `get_observation()` service takes an initial time as one of its arguments and returns data for the first reported interval for a process (c) that starts after the argument. This service can be used to acquire data for display, or to discover the time at which a process first starts to report.

The `get()` service takes as input a description for a process (c) and a time period (start time t_s and end time t_e). It then aggregates information for the process over the time period, computes confidence intervals, and performs further statistical analysis, as described in Section 4.4.2 on Page 71. The data returned is an aggregate of the intervalized data submitted to the data server by the `put` operation, as follows.

- The regressed CPU demand (D_m), regressed I/O counts (V_{io}) for each operation (m), each with number of observations in regression, coefficient of determination, and confidence interval.
- The average response time ($\bar{R}_m$) for each operation (m).

- Mean measures for the process as a whole, including the average threading level, process CPU demand and process I/O count.
- Accumulated operation invocation counts. For each operation (m) in the export interface (E_c), the accumulated invocation count ($V_{c,e,m}$) is reported, along with the visit count (V_{c,e,m,s,i,m_2}) to each operation (m_2) in each interface (I_c) of each of its serving processes (s).

4.4.2 Data analysis

The collected measurement data is intervalized by the observer within each process and then forwarded to the data server for storage. The `get()` routine, which is typically called by display or model-building applications, aggregates data over a longer time period that spans many reporting intervals. It provides confidence intervals for the measured data.

We note that LQMs require CPU and disk input-output counts for each operation of each interface exported by a server. Data at this level of detail is needed when services provided by application servers require significantly different numbers or types of resources. Unfortunately, this data is not available on most UNIX platforms. Most UNIX platforms report resource consumption on a per-process basis. Sometimes, the resource demand per operation can be estimated using statistical regression techniques [32].

Consider a process that provides n operations with data collected over m periods. Let the number of invocations for each operation i in the j th period be c_{ij} and let r_j be the CPU usage (or disk input or output operations) observed during period j . The data forms a set of linear equations as follows:

$$c_{1j}x_1 + c_{2j}x_2 + \dots + c_{nj}x_n = r_j, 1 \leq j \leq m$$

The solution to this system of linear equations gives the estimated CPU usage (or disk input or output operations) of each operation. That is, if the solution is $(x_1, x_2, \dots, x_n)$, then x_i ($1 \leq i \leq n$) is the estimated CPU usage of the i th operation.

The statistical method presented above has some special cases that are addressed by the performance data server. The number of observations must be greater than or equal to the number of services provided by the given process. If there are not enough observations to perform the regression, the average resource demand for the process per operation is returned for each operation. Another scenario is that the visit counts for several operations may be highly correlated. As an extreme case, visit counts for one operation may always be an exact multiple of the counts for another operation. Regression estimates for strongly correlated operations must be combined. And, if an operation is never invoked, the operation is not included in the regression model. Its resource demands are each set to zero.

4.5 Summary

This chapter has described the information that is needed for model building, and the infrastructure that supports such model building. A high-level overview of the model building system was given.

The model used in this thesis is the Layered Queueing Model (LQM). Like any other performance models, LQM has two aspects, namely structure and resource demand parameters. A model building technique must address these two aspects. An integrated approach is used in this thesis that builds LQMs based on performance measurements.

The model building infrastructure is based on MANDAS and DMS. It is a software architecture that facilitates the routine measurement of performance metrics on distributed application processes. There are three main components, namely an instrumentation package, a performance data server, and the model builder.

The purpose of the instrumentation package is to measure and collect the information required for model building and forward it to the performance data server. It includes a variety of routines and macros to be placed by an application programmer at certain points within the application code.

The performance data server supports storage for the intervalized process data gathered by the process instrumentation. The data server also performs data reduction and statistical analysis needed to derived parameters for LQMs. The interface of the multi-threaded performance data server can be accessed via a TCP/IP socket interface or a DCE Remote Procedure Call interface. The two access methods are provided to enable the data server to operate in heterogeneous environments. The interface services can be categorized into three types of services: management, data storage, and data reporting.

With the support of the instrumentation package and the performance data server, the model builder can construct performance model for instrumented application. The model building techniques are described in Chapter 5.

Chapter 5

Building the Layered Queueing Model

Within the modeling framework, a predictive LQM is built for a specific time period. The model characterizes the mean behaviour of the system for that period. A typical period may be between 15 and 30 minutes. As a goal, it is long enough to give statistical confidence in the measurements yet short enough to describe an interesting part of a system's day, such as a peak busy period. Once data is gathered for a period by the instrumentation and placed in the performance data server, the model builder can be used to generate a corresponding LQM. This chapter presents the functional requirements of the model builder and describes the major components of it, as well as verification and validation steps for the resulting model. Chapter 6 shows how the model builder is used in a case study and how resulting model can be used to explore "what-if" scenarios.

5.1 Functional Requirements of Model Builder

The purpose of the *model builder* is to acquire measured data from the performance data server and construct an LQM that reflects the behaviour of the measured system for a specific time period. The following services must be provided by the model builder:

1. **Model Generation:** For a given instrumented and monitored application, the model builder should be able to collect data from the performance data

server and construct a LQM model for this particular application for a specific period. The model builder should provide services to solve the model for performance metrics and make comparison with measured performance metrics, for instance response time.

2. **Verification and Validation:** The model building process must ensure that the model is correctly constructed and that it is representative of the real system. These two steps are called model verification and model validation, respectively.

The verification of the resulting model can be done by examining its structure and comparing it to the application.

Validation refers to ensuring that the assumptions used in developing the model are reasonable in that, if correctly implemented, the model would produce results close to that observed in real systems. The validation techniques depend upon the assumptions and, hence, on the systems being modeled. Model validation consists of validating the three key aspects of the model[18]:

- assumptions:
- input parameter values and distribution: and,
- output values and conclusions.

Each of the three aspects may be subjected to a validity test by comparing it with that obtained from the following three possible sources:

- expert intuition:
- real system measurements: and,
- theoretical results.

This leads to nine possible validation tests. But in most real situations, in fact, none of the nine possibilities may be feasible. Comparison with real systems is the most reliable and preferred way to validate a simulation model. Statistical

techniques to test whether two systems are identical may be used to compare the simulation model and the measured data.

The model builder should report the measured response times for operations of interfaces. These can be used for display purposes and also for model validation. If confidence interval for data reported by the performance data server are poor, the model builder should be able to detect this and provide warning.

3. **Exception Handling:** The model builder should be able to handle the exceptions returned by the performance data server. For example, if the regression routine of the performance data server fails due to insufficient data, the model builder should be able to use average service values for all methods.

5.2 Model Building Steps

The LQM has two main aspects, *structure* and *resource-demand parameters*. The structure describes requesting and blocking relationships among processes. The parameters fix the mean number of visits and service times of requests as they flow through various operations of process interfaces and use devices. It is assumed that interactions between processes are synchronous RPC. Other interactions and support for them are discussed further in Section 5.5. This section gives an overview of steps needed to build an LQM using measured data.

In an LQM, server processes and devices are represented as *entities*. Each entity has one or more *entries*. An entry is used to characterize the visits made by an operation of an interface to other operations of interfaces and devices. In the model, each device has a single entry that is implied. Each entry has a list of *events*. Each event represents the mean number of visits and service times at a device, for example a CPU, disk, or network, or the mean number of visits to an operation of an interface in some other process. In the model, these are reflected as a visit from the entry of the client entity to an entry of the server entity. Devices have a single entry with no events.

To build the model, one must obtain information about the system of applications from the performance data server and then conducts the following sequence of steps:

1. Choose the measurement time period:
2. Get data about the application processes from the performance store for the time period:
3. Associate processes and devices with entities in the model and assign their populations and scheduling disciplines:
4. Create events to reflect interactions between entries of entities:
5. Fix the mean number of visits and service times for events: and,
6. Fix the think times for entities.

These steps are discussed in detail in the following sections. The notations and symbols used in these sections are shown in Figure 5.2 on Page 78.

5.2.1 Measurement Time Period

For model building a time period of interest should be chosen first. This is because the behaviour of the system changes as the workload on an system changes with time. As an example, a bank's automated teller machine (ATM) system may be most busy during the half hour from noon to 12:30 pm each day. So a time period of interest must be chosen to build a model which represents the average behaviour of the system for that particular time period. The model can then be used for operational performance management. For example, to study the impact of increased loads or alternative application or system configuration on user response times. How to choose the time period depends on the actual system. For this work, users of the model builder need to choose the time period.

In the above example, the performance data server would have to be started before noon, and the ATM application instrumentation enabled sometime shortly

c	A process in an application.
k	A device in the application system.
E_c, ϵ	An export interface of a process c . It is ϵ when used as a subscript.
I_c, i	An import interface of a process c . It is i when used as a subscript.
m	An operation in an export interface.
m_2	An operation in an import interface.
D_c	CPU demand of a process as a whole.
$D_{c,e,m}$	CPU demand of an operation m .
$S_{c,e,m}$	CPU service time of an operation m .
V_c	Invocation counts of a process c .
$V_{IO_{c,e,m}}$	I/O counts of an operation m .
$V_{c,e,m}$	Invocation counts of an operation m .
V_{c,e,m,s,i,m_2}	Visit counts of operation m to operation m_2 in interface I_c of its serving process s .
$R_{c,e,m}$	Response time of operation m .
Δt	Time interval of measurement period.
Z_c	Think time of a client process c .
EP	An entity in an LQM model.
EM	An entry of an entity EP .
EV	An event of an entry EM .

Figure 5.1: Notations and Symbols

before noon. During the time period, observers within application processes periodically submit intervalized data to the data server. The remaining steps of model building can be completed at any time. For this example, it is reasonable to choose a time shortly after 12:30 pm.

5.2.2 Building the Model Structure

This section shows how the model structure can be built from measurement data retrieved from the performance data server. Building the model structure can be divided into three building steps.

- Create entities.
- Create entries of entities.
- Create events that describes visits between entries and to devices.

Each of these three steps is discussed as follows. Simplified algorithms are shown in Figure 5.2 on Page 81 and Figure 5.3 on Page 82.

Creating Entities

In LQM, processes and devices are represented as *entities*. The first thing in building the model is to identify the software processes of a application and the devices they use. When model building occurs, the model builder queries the performance data server to get a list of applications that were active during the measurement time period using the `get_applications()` service of the performance data server, and then for a particular application get the identities of the participating processes using the `get_host_process_ids()` service. Subsequent queries are made to get data for each process using the `get()` service.

Each process (*c*) is identified by an internet protocol address for its host node, a process ID, and a program ID (See Section 4.4 on Page 67). This identifier provides the information for creating entities representing the processes and the host CPU's on which the processes run:

1. Each process running on a host node has a unique process ID, and an entity is created to represent this unique process.
2. The IP address of the host node uniquely identifies its CPU as a device used by the application. So a device entity is created for this node. The scheduling discipline for it is Processing Sharing (PS) as a default.

If subsequent process's host node has the same IP address, no new device entity will be created for this host, and the process is identified as running on the same host sharing the same device.

The disks of each node need to be found from a configuration data store or a lookup table. This is because the I/O device information is not in the performance data store, only the I/O counts of processes are part of the performance information. An entity is created to represent the disk. The service rate should also be found in the lookup table, otherwise a default value is used.

If multiple client processes from the same node have the same program type ID, then they are represented as a single entity with a population greater than one. For server processes, the population of the entity is set to the process's threading level. Entities are then assigned scheduling disciplines. Devices are assigned the processor-sharing discipline as a default. Processes are first-come-first-served servers.

Creating Entries

In LQM, entries represent the operations or methods that servers provide. The operations are specified in the server's export interface (E_c). The interface information of a process (c) can then be used to create entries for the entity representing the process.

Devices have only one entry as implied. Server process can have many entries. A server process exports interfaces it supports, and the interface is stored in the information data server along with other information about the server process (See Section 4.4 on Page 67). Using the `get_host_process_ids()` service of the perfor-

```
Start by initializing the process_to_entity mapping table (T);
get_applications() for a time period (t);
get_host_process_ids() for a specific application (A);

FOR each process (c) in (A)
  IF (c) not in mapping table (T), THEN
    create an entity (EP) for (c);
    get() information about its export interface (Ec);

    FOR each operation (m) in (Ec)
      create an entry (EM) for (m);
    END FOR;

    insert the entity (EP) in the mapping table (T);

    IF the CPU is not in (T)
      create an CPU entity and insert it in (T);
      create an DISK entity and insert it in (T);
    END IF;
  END IF;
END FOR;
```

Figure 5.2: Algorithm for building LQM model structure with measurement data.
Part I: Creating entities and entries for processes and devices.

```
For each process (c) with entity (EP) in the mapping table (T)
    get() information about export interface (Ec);

    FOR each operation (m) in the export interface (Ec)
        FOR each call to an operation (m2) of the import interface (Ic)
            find the entity (EP2) from table (T) for (Ic);
            find the entry (EM2) in the entity (EP2) for (m2);
            create an event to entry (EM2) in entity (EP2);
        END FOR;

        create event to CPU and DISK entities;

    END FOR;
END FOR;

sort the entities and find the client entity;
```

Figure 5.3: Algorithm for building LQM model structure with measurement data.
Part II: Creating events and connectivity among entities.

mance data server, details about the export interface (E_c) can be retrieved. For each operation (m) in the interface (E_c), a corresponding entry is created.

When a server process starts, the observer thread of this process will report the interface operations, with the number of invocations and the services which each interface operation uses for a process period. Such information is used to deduce visit counts and to create events for the entries.

Creating Events

In LQM, an entry has a list of events which reflect the average behaviour of an entity's corresponding process or processes. The events represent the mean number of visits to devices or number of visits to operations of imported interfaces of its serving processes. Using the process information stored in the performance data server, it is possible to establish connectivity among process interfaces and thus create these events.

When a process(c) is represented in the LQM model by an entity (EP_c), each operation (m) of its export interface (E_c) is associated with an entry (EM_m) in (EP_c). When `get_host_process_ids()` and `get()` services of the performance server report information about each operation (m) in the export interface (E_c), the information about visits (V_{c,e,m,s,i,m_2}) to each operation (m_2) of each import interface (I_c) of its serving processes (s) is also reported (See Section 4.4 on Page 67). For each of such visits, the serving process (s) and its corresponding LQM entity (EP_s) can be identified with the process information and process to entity mapping, and the corresponding entry (EM_{m_2}) for operation (m_2) can also be identified by the operation ID within the scope of the process (s). An event can then be created going from the entry (EM_m) in client entity (EP_c) to the entry (EM_{m_2}) in the serving entity (EP_s).

The number of visits (V_{c,e,m,s,i,m_2}) per execution of the export interface operation (m) is the visit count. Physical resource demands for the operation (client entity entry) include the average demands of the operation at the processor and visits to the disk of the entity's node. A system configuration file is used to specify the number

of processors and disks on each node and their service rates. Future performance management systems should make such information automatically available for each node.

5.2.3 Deducing Model Parameters

Within a measurement period of interest for model building, there could be many process periods for the application processes. This is essential for the regression routine in the performance data server. As describe in Section 4.4 on Page 67, the performance data server reports the following parameters for each process (c) for a user specified measurement period.

- Mean measures for the process as a whole, including the average threading level, process CPU demand (D_c) and process I/O count (V_{IO}).
- The mean response time ($\bar{R}_{c,e,m}$) for each operation (m) in the export interface (E_c).
- The regressed mean CPU demand ($D_{c,e,m}$) and I/O counts ($V_{IO_{c,e,m}}$) for each operation (m) for each invocation of (m).
- Accumulated (for the whole measurement period) operation invocation counts for each operation (m), including invocation count ($V_{c,e,m}$) and the visit counts (V_{c,e,m,s,i,m_2}) to each operation (m_2) in each interface (I_c) of each of its serving processes (s).

The model builder then need to go through the process data and deduce parameters to represent the average behaviour of the systems. The parameters are described in Section 4.1 on Page 56, and include the following for each operation (m) of each export interface (E_c) of each process (c).

- The average number of visits ($V_{c,e,m,k}$) to each device (k) in the system.
- The average service time ($S_{c,e,m,k}$) per request at each device (k) in the network.

- The average number of visits ($\bar{V}_{c,e,m,s,i,m_2}$) to each method (m_2) of each import interface (I_i).

CPU Service Demand and I/O Counts

The model builder uses the CPU service demand ($D_{c,e,m,cpu}$) instead of CPU service time ($S_{c,e,m,cpu}$) since the mean CPU demand ($D_{c,e,m,cpu}$) is reported for the measurement period. Visit counts to CPU ($V_{c,e,m,cpu}$) is not measured, otherwise service time ($S_{c,e,m,cpu}$) can be calculated using Equation 5.1.

$$D_{c,e,m,cpu} = S_{c,e,m,cpu} V_{c,e,m,cpu} \quad (5.1)$$

Each class of client can have population that is more than one, i.e., there can be multiple client processes of the same client class running on the same host machine. So the CPU demand of the client need to be the average of the client processes. It is calculated as shown in Equation 5.2, assuming there are (n) client processes. The I/O count is averaged in the same way for each client class.

$$\bar{D}_{c,e,m,cpu} = \frac{\sum_{j=1}^n D_{c,e,m,cpu}}{n} \quad (5.2)$$

Visit Ratios

Invocation counts retrieved from the performance data server are accumulated for the measurement period. Visit ratios ($\bar{V}_{c,e,m,s,i,m_2}$) for each operation (m) need to be deduced from these accumulated counts using Equation 5.3.

$$\bar{V}_{c,e,m,s,i,m_2} = \frac{V_{c,e,m,s,i,m_2}}{V_{c,e,m}} \quad (5.3)$$

Average Response Time

Response time is measured for each operation (m) of each export interface (E_e) of a process (c). The performance data server reports the mean measured response times ($\bar{R}_{c,e,m}$) for operations. The mean process response time can then be calculated using weighted average of ($\bar{R}_{c,e,m}$) using Equation 5.4 and Equation 5.5.

$$\bar{R}_c = \frac{\sum_e \sum_m R_{c,e,m} V_{c,e,m}}{V_c} \quad (5.4)$$

where V_c is defined as,

$$V_c = \sum_e \sum_m V_{c,e,m} \quad (5.5)$$

For a class of clients with population (n), the mean operation response time ($\bar{R}_{c,e,m}$) and mean process response time ($\bar{R}_c$) need to be averaged among all the clients.

Client Think Time

The think times for client processes represent the average time between client requests. Think times are reflected in the model as events to a special think centre device. Each entry of each client entity has an event for a single visit to the think centre. The average think time (Z_c) is calculated as follows: take the difference between the duration of the measurement period (Δt) and process busy times ($\bar{R}_c V_c$), then divide by the number of process entry invocations, as shown in Equation 5.6.

$$Z_c = \frac{\Delta t - \bar{R}_c V_c}{V_c} = \frac{\Delta t}{V_c} - \bar{R}_c \quad (5.6)$$

5.3 Implementation of Model Building System

A prototype model building system has been implemented using OSF/DCE. The communication between the observers and data server is through OSF/DCE, so is the communication between the model builder and the performance data server. The main components of the system are shown in Figure 5.4 on Page 87. Components shown shaded have been described in Chapter 4, and only the rest are described here.

- *User Interface* provide users with simple commands to:
 - control the collection of measurement data of instrumented applications;
 - request model building for given applications and time periods;
 - display model results; and,
 - modify models and perform “what-if” scenario analyses.

Figure 5.4: Main Components of the Model Building System

START

Initialization:

`get_host_process_ids()` for a specific application:

FOR each process c

`get()` measurement data for the measurement period:

 retrieve corresponding entity EP_c for c :

 FOR each entry EM in EP_c

 accumulate $V_{c,e,m}$:

 accumulate $R_{c,e,m} V_{c,e,m}$:

 accumulate V_c :

 accumulate $R_c V_c$:

 FOR each event EV

 set $D_{c,e,m}$:

 set $V_{IO_{c,e,m}}$:

 accumulate V_{c,e,m,s,t,m_2} :

 END FOR

 END FOR

 set *population*:

END FOR

To be continued...

Figure 5.5: Algorithm for fixing LQM model parameters. Part I: accumulate measurement data.

...continued

```

FOR each entity  $EP_c$ 
  calculate response time  $\bar{R}_c$ ;
  calculate client think time  $Z_c$ ;
  FOR each entry  $EM$ 
     $V_{c,e,m,nonCPU} = 0$ ;
    calculate  $\bar{R}_{c,e,m}$ ;
    FOR each event  $EV$ 
      IF to CPU THEN
         $D_{c,e,m} = \frac{D_{c,e,m}}{population}$ ;
      ELSE IF to disk THEN
         $V_{IO_{c,e,m}} = \frac{V_{IO_{c,e,m}}}{population}$ ;
         $V_{c,e,m,nonCPU} = V_{c,e,m,nonCPU} + V_{IO_{c,e,m}}$ ;
      ELSE IF to a server
         $V_{c,e,m,s.t.m_2} = \frac{V_{c,e,m,s.t.m_2}}{population}$ ;
         $V_{c,e,m,nonCPU} = V_{c,e,m,nonCPU} + V_{c,e,m,s.t.m_2}$ ;
      ELSE IF to think centre THEN
         $S_{c,e,m,think} = \frac{Z_c}{V_{c,e,m,think}}$ ;
         $V_{c,e,m,nonCPU} = V_{c,e,m,think}$ ;
      END IF
    END FOR
     $V_{CPU_{c,e,m}} = V_{c,e,m,nonCPU} + 1$ ;
  END FOR
END FOR

END

```

Figure 5.6: Algorithm for fixing LQM model parameters. Part II: deduce mean values.

- *Model Builder* generates the LQM model structure and fix the model parameters. Its output is a LQM data file. The algorithm for constructing the model structure is shown in Figure 5.2 on Page 81 and Figure 5.3 on Page 82. and the algorithm for fixing model parameters is shown in Figure 5.5 on Page 88 and Figure 5.6 on Page 89.
- *Model Solver* solves the LQM model and saves the results to file. Method of Layers [29] is the default solver. though other solvers can also be used.
- *Model Presenter* retrieves LQM model data for display purpose. for instance. to retrieve and compare both measured and estimated response times and utilizations.

The LQM model file contains fields for both analytic and simulation results, as well as model structure and parameters. The tool developed for this thesis uses the fields for simulation results to store the measured performance metrics of the application. Once it is built by the model builder, the LQM model can be solved analytically by invoking the model solver and the results saved in the same model file. The model presenter can then be invoke to extract and display both analytic and measured results. Comparison of the two set of results can be used as a first step in model validation.

5.4 Exceptions, model verification, and model validation

Several problems can arise when relying on an automated model building system. The model building system is designed to recognize and overcome some of these problems.

Separate information about each entry's physical resource demands is not available if the regression performed by the performance data server fails. In this case,

the `get()` calls to the data server simply return the average physical resource demands per completion at the process. These are used for each entry. The remaining visits between operations of interfaces are still characterized correctly.

If the confidence intervals for data reported by the performance data server are poor, a warning message is given. Either not enough measurement data was collected to characterize the system's behaviour, or some significant change to the system took place during the measurement period. Further exploratory work can then be done automatically to determine whether all of the processes were active for the entire measurement period. If not, the resulting information can be used to partition the measurement period into several shorter measurement periods and give models for each. This is a subject of future work.

The instrumentation package reports response times for operations of interfaces. These can be used for display purposes and also for model validation. The performance data server reports mean response times and a confidence interval for a measurement period. These values are reflected in the LQM file along with the model specification. In this way predicted mean response times for the model can be compared with measured response times for that same interval.

5.5 Unmanaged work and other types of interactions

It is noted that there is always likely to be unmanaged work on a system as well. Though they are not presently characterized in the model, unmanaged processes can be treated as client processes that do not import interfaces. *Proxy observers* can periodically forward information about their resource consumption to the performance data server. Information about the processes can be used to create several entities to reflect their combined behaviour. For example, CPU-intensive, disk-intensive, and mixed-work entities can be introduced for each node.

The instrumentation package described in Section 4.3 associates an interaction type with each operation of an interface. For exported interfaces, these include syn-

chronous, asynchronous, and forwarding. Each imported operation also has an interaction type, which includes synchronous, asynchronous, and deferred synchronous. Currently we assume that each operation of each interface is accessed and supported using a synchronous RPC. Model building extensions for the other interactions have been studied [17] and their integration into the model building system is considered as future work.

5.6 Summary

In this chapter the model building techniques were described. The model building system is part of the modelling framework described in Chapter 4, and it requires the support of the performance infrastructure.

To build the model, one must obtain information about the system of applications from the performance data server and then conducts the following sequence of steps:

1. Choose the measurement time period:
2. Get data about the application processes from the performance store for the time period:
3. Associate processes and devices with entities in the model and assign their populations and scheduling disciplines:
4. Create events to reflect interactions between entries of entities:
5. Fix the mean number of visits and service times for events:
6. Fix the think times for entities.

A predictive model is built for a specific time period, which should be long enough to give statistical confidence in the measurements yet short enough to describe an interesting part of a system's day. Once data is gathered for a period by the instrumentation and placed in the performance data server, the model builder can be used to first generate the LQM model structure and then fix the model parameters.

Processes in the application need to be identified, and corresponding model entities created. Corresponding entities for devices also need to be created. Entries are created for operations in the export interfaces of each process entity. Once entities and their entries are created, events for entries can be created using the connectivity information for each entry to represent the visits from operations to devices or other serving operations. Simplified algorithms are shown in Figure 5.2 on Page 81 and Figure 5.3 on Page 82.

Resource demand parameters fix the mean number of visits and service times of requests as they flow through various operations of process interfaces and use devices. Mean CPU demands and I/O counts for operations of processes are reported by the performance data server, and they are assigned to the corresponding entries of the entities. The invocation counts of an operation and its visit counts to operations of serving processes are reported as accumulated sums. Visit ratios must be deduced to represent the average behaviour of the system using algorithm described in this chapter.

Several problems can arise when relying on an automated model building system, such as when the regression performed by the performance data server fails or the confidence intervals for data reported by the performance data server are poor. The model building system is designed to recognize and overcome some of these problems.

It is noted that there is always likely to be unmanaged work on a system as well. Though they are not presently characterized in the model, unmanaged processes can be treated as client processes that do not import interfaces.

A case study is presented in the following chapter to demonstrate how the modelling building system can be used.

Chapter 6

Case study

This chapter demonstrates the model building technique by building a Layered Queueing Model [29] automatically for an instrumented distributed application running in DCE [23] environment.

A synthetic on-line processing system is considered. It is a three-tiered DCE application, with clients and two levels of servers. End users enter commands at the client process to request services from the servers and wait for a response. To serve the client requests, servers may need to use the disk and may request services from each other. Service times are generated as exponentially distributed random variables with specified mean values that mimic application processing times. All the programs have been instrumented to collect measurement data for model building.

The application is run for several different combinations of client populations and server-threading levels. Layered queueing models are built automatically by the model building system and then analyzed with the Method of Layers [29]. The measured and predicted end-to-end response times are compared for validation. Several “what-if” scenarios for the system are explored using the analytic technique.

6.1 An Instrumented Distributed Application

The synthetic distributed application has the structure of a typical on-line transaction processing system with a client/server architecture, as illustrated in Fig-

Figure 6.1: A Synthetic DCE Application and the Performance Data Server

Figure 6.1 on Page 95. Upon receiving user commands, client processes (three instances are shown in Figure 6.1) invoke the operations provided by two server processes, **Server 1** and **Server 2**. **Server 1** also invokes **Server 2** as a consequence of clients' invocations of **Server 1**. Each server provides a single interface which contains a number of operations, in this case each has two operations. Servers can be replicated to provide concurrent execution of the same service. Each server can also have multi-threads, providing concurrency within the same process. The inter-process communication mechanism among the clients and servers is synchronous Remote Procedure Call.

The following services are identified in the processes.

- **Clients** receive commands from users, and then alternate between invoking operations of servers, visiting the processor of their node, and "thinking". Client "thinking" means that the client process is idling waiting for the user to submit a new command.
- **Server 1** has two operations. **Operation 1** handles service request by means

of the node's process and disk, while **Operation 2** also needs service from **Operation 2** in **Server 2**.

- **Server 2** also has two operations, both handle service requests by means of the node's processor and the disk.

The interaction of the computational objects in the synthetic application is further illustrated in Figure 6.2 on Page 97, represented by a Layered Queueing Model. There are six entities representing the software processes and devices:

- **Entity 1** represents the **think centre**, which is a delay centre where the client spends its **think time** before getting the next command from the user and start making request to the servers. Its concurrency level can vary.
- **Entity 2** represents the CPU with process sharing queueing discipline. There is only one CPU entity in this model since all client and server processes run on the same host.
- **Entity 3** represent the **Disk** which has process sharing queueing discipline. There is only one disk for the host.
- **Entity 4** represents **Server 2**. This is a software process with First-In-First-Out(FIFO) queueing discipline and exponential service time distribution. It has two operation in its only interface. Both **Operation 1** and **Operation 2** make service request to the disk, once per invocation.
- **Entity 5** represents **Server 1** which has a FIFO queueing discipline and exponential service time distribution. It has two operations in its only interface. **Operation 1** makes one service request to disk per invocation, while **Operation 2** makes service request to **Operation 2** at **Server 2**.
- **Entity 6** represents the **client** process. Client makes service request to both **Operation 1** and **Operation 2** in **Server 1** and only **Operation 1** in **Server 2**.

Figure 6.2: Layered Queueing Model (LQM) of the Synthetic Application

Calling Entity	Entry	Resource	Visits to Resource	Service Time/Visit (seconds)
Client	Client	CPU	5	0.040
		Thinker	1	16.0
Server 1	Operation 1	CPU	2	0.100
		Disk	1	0.008
	Operation 2	CPU	2	0.035
Server 2	Operation 1	CPU	2	0.100
		Disk	1	0.008
	Operation 2	CPU	2	0.100
		Disk	1	0.008

Table 6.1: Prescribed Resource Visit Ratios and Mean Service Times for the Synthetic Application

The number on each directed edge in the LQM in Figure 6.2 on Page 97 indicates the mean visit ratios from a client process to a serving process or device. Since the central server model is assumed, the visit ratio to the CPU for each operation of a software process is the sum of visit ratios to all other entities plus one. Since the application is synthetic, CPU service times are prescribed for each operation based on software design: they are generated as exponentially distributed random variables with specified mean values that mimic application processing times. Likewise, the client think time is prescribed at 16 seconds. The disk service demand for each visit, however, is not prescribed and it is estimated as 8 milliseconds. These resource usages are tabulated in Table 6.1 on Page 98. It must be noted that the actual visit ratios, service time and client think time will be fixed by the automated model builder from measurement data.

The synthetic application is implemented in DCE environment. The inter process communication mechanism is synchronous Remote Procedure Call. Client and server processes can run on different nodes in the same DCE environment, and the servers

Client Population	Server Threads	Host	Process Period (seconds)	Measurement Period (seconds)
1	1	Host 1	60	3600
5	1	Host 1	60	3600
5	3	Host 1	60	3600
10	1	Host 1	60	3600
10	3	Host 1	60	3600

Table 6.2: Experiment Setup Combinations

are multi-threaded. However all the processes in the experiment run on a single node: this is to eliminate the network delay, which is not currently measured, nor modelled in this particular LQM.

The instrumentation thread in each application process periodically forwards data to the performance data server. The communication between the instrumentation and the data server is also through DCE RPC.

6.2 Building the Layered Queueing Model

Table 6.2 on Page 99 describes several configurations of the distributed application system in terms of the number of client processes and the threading levels of servers. The average client think time is 16 seconds, which means that a user would on average pause for 16 seconds between two requests. The reporting interval (process period) for the observer threads is sixty seconds, and each experiment is run for one hour. All processes for the synthetic application and the data server for the model building program are on the same host running DCE.

Experiments were conducted according to the setups in Table 6.2 on Page 99. Performance data, including invocation counts of export operations, visit counts of import operations, and process CPU demands were collected. The only parameter

that was not measured is the disk service time, which was estimated as 8 milliseconds. The disk visit counts was measured by counting the number of disk read/write operations in the application.

Layered Queueing Models were built automatically by the model building program for each and every experiment. The CPU service times for the client and server operations were derived using statistical regression method from the measured process CPU usage (See Section 4.4.2 on Page 71). The models were then solved using MOL software [29]. Table 6.3 on Page 101 shows resource usage parameters for the client in the automatically built LQM for all test cases, and Table 6.6 on Page 106 shows measured mean client response times $\bar{R}$ along with estimates obtained from solving the LQMs using the MOL. The resource usage parameters for **Server 1** and **Server 2** are shown in Table 6.4 on Page 102 and Table 6.5 on Page 103 respectively.

A LQM of the same application was manually built with all the prescribed resource usage parameter values, and was solved for the same set of combinations of client population and server threading level. The performance metrics obtained from this model were also compared with those from the automatically built model. The comparison shows how sensitive the performance estimates are to variations in the model parameters.

The model must be first be validated to have correct resource usage parameters. Compared with the prescribed mean resource visit ratios and service times shown in Table 6.1, Tables 6.3 to 6.5 show that the model-building infrastructure correctly captures the resource demand values needed for model building. The visits ratios are correctly deduced, with at most 4% error for all experiment setups. This is because the visit ratios are deduced from invocations counts and these counts are accumulated by the instrumentation package accurately.

The service times for the interface methods are expected to have lower accuracy, because, first, CPU measurements obtained by using operating system calls are only estimates of CPU consumption, and secondly, the service times must be statistically regressed from the process CPU usage. However, the service times at resources are in general correctly assigned to each operations. For an example, the **Client**

Calling Entity	Entry	Resource	Test Setup		Visits to Resource	Service Time Per Visit	
			Clients	Server Threads		measured (seconds)	% error to prescribed
Client	Client	CPU	1	1	5.2	0.034	15.0%
			5	1	5.0	0.037	7.5%
			5	3	5.0	0.037	7.5%
			10	1	4.9	0.037	7.5%
			10	3	4.9	0.037	7.5%
		Thinker	1	1	1.0	13.0	18.8%
			5	1	1.0	15.7	1.9%
			5	3	1.0	15.7	1.9%
			10	1	1.0	16.5	3.1%
			10	3	1.0	16.3	1.8%

Table 6.3: Measured Visit Ratios and Regressed CPU Service Times for the Client with Relative Errors Compared with Prescribed Values

Calling Entity	Entry	Resource	Test Setup		Visits to Resource	Service Time Per Visit	
			Clients	Server Threads		measured (seconds)	% error to prescribed
Server 1	Op. 1	CPU	1	1	2.0	0.100	0%
			5	1	2.0	0.100	0%
			5	3	2.0	0.100	0%
			10	1	2.0	0.094	6.0%
			10	3	2.0	0.099	1.0%
	Op. 2	CPU	1	3	2.0	0.039	11.4%
			5	1	2.0	0.032	8.5%
			5	3	2.0	0.034	2.9%
			10	1	2.0	0.043	22.9%
			10	3	2.0	0.037	5.7%

Table 6.4: Measured Visit Ratios and Regressed CPU Service Times for **Server 1** with Relative Errors Compared with Prescribed Values

Calling Entity	Entry	Resource	Test Setup		Visits to Resource	Service Time Per Visit	
			Clients	Server Threads		measured (seconds)	% error to prescribed
Server 2	Op. 1	CPU	1	1	2.0	0.100	0%
			5	1	2.0	0.099	1%
			5	3	2.0	0.097	3%
			10	1	2.0	0.100	0%
			10	3	2.0	0.103	3%
	Op. 2	CPU	1	3	2.0	0.100	0%
			5	1	2.0	0.100	0%
			5	3	2.0	0.105	5%
			10	1	2.0	0.103	3%
			10	3	2.0	0.100	0%

Table 6.5: Measured Visit Ratios and Regressed CPU Service Times for **Server 2** with Relative Errors Compared with Prescribed Values

think time was set at 16 seconds in the experiments, and the measured value is 15.7 seconds with less than 2% error for the case of 5 client instances with either 1 or 3 server threads. The worst case for **Client** think time is at 13 seconds with 18.8% error for single client instance and single server thread. It is also noted that for the same case the **Client** CPU service time is also at a lower value of 0.034 seconds compared to the prescribed mean value of 0.04 seconds. The reason could be that the application was less accurately calibrated to run for the prescribed time for the particular experiment setup, as well as the operating system related CPU usage accounting errors. CPU service times for the servers are correctly assigned, in most case with less than 5% error. For instance **Operation 2** of **Server 2** has a deduced service time of 105 milliseconds against the prescribed mean service time of 100 milliseconds.

The model needs to be further validated for its correctness in providing estimates for the application system in question. This is done by examining the end-to-end client response times, as shown in Figure 6.6 on Page 106. The measured values in all cases are close to the analytical values obtained by solving the LQMs: the analytical values are within the 95% confidence interval of the measurements. The relative errors between the response times estimated with the automatically built model and the prescribed model are also small. In the cases of *single client* or *multi-threaded servers*, the measured and analytical response times are virtually identical. The worst relative error of 13.3% occurs for the case of 10 clients and single thread server.

For this case the CPU utilization is at 48% in the analytic model (See Table 6.11 on Page 109), not counting many unmanaged processes running on the same host. It is interesting to note that for the multi-threaded server with 10 clients, the CPU utilization is also at a high value of 49%, yet the measured and analytical response times are virtually identical.

Table 6.7 on Page 107 to Table 6.10 on Page 108 shows the measured and analytical values of response times of the server operations. Relatively larger errors, up to 30%, are found, and the analytical values are pessimistic except for **Server 1**

Operation 2 in the cases of 5 clients with single thread servers and 10 clients with three-threaded servers. The estimates obtained from the automatically built model are close to those from the prescribed model, with relative error generally below 5%. This supports the assertion that large discrepancies between measured and analytical values most likely have resulted from the way the response time was measured by the instrumentation package and not sensitive to the service demand values.

The technique used in this work is *source code instrumentation* (See Section 4.3 on Page 64), where a variety of routines and macros are placed by an application programmer at certain points within the application's code. The response time is measured by getting a time stamp at the beginning of the server operation block and again getting another time stamp at the end of the block just before the execution returns. The difference of the two time stamps is then reported as response time for this operation. However, the runtime system calls that get invoked to map the incoming call to an operation is not included in the measured response time. When the LQM is built and subsequently solved, the total service time for the operation is taken into account, thus producing a greater analytical response time than measured.

It is also noted that the errors for **Server 1 Operation 2** are generally smaller in comparison to the ones of the other server operations. This particular operation does not visit the disk, instead it invokes **Operation 2** of **Server 2**. This fact makes it reasonable to think that some error also comes from the modelling of disk visits. In fact, estimating the disk service time proved difficult. Even more difficult was counting the disk visits, and the way disk visits were counted in the instrumentation might give incorrect measurements. This is because a disk read/write operation in the application does not necessarily result in a physical I/O to the disk, and yet it is counted as a disk visit. Solving the LQM with these pessimistic disk visits results in greater response time. More investigation in this area is therefore needed.

It has to be stressed that the measured response times are expected to be optimistic simply because of the way how they are measured. The service times for the server operations, however, are more accurately measured, so are the server operation visit ratios. It is the service times and visit ratios that need to be fixed in the

Client Pop.	Server Threads	Measured $\bar{R}$		Prescribed Model $\bar{R}$ (secs)	Generated Model $\bar{R}$		
		(secs)	95% Confidence		(secs)	% error to	
			Interval			measured	prescribed
1	1	0.95	(0.78, 1.12)	0.96	0.97	2.1%	1.0%
5	1	1.20	(1.06, 1.34)	1.17	1.05	12.5%	10.3%
5	3	1.20	(1.06, 1.34)	1.19	1.17	2.5%	1.7%
10	1	1.73	(1.53, 1.93)	1.55	1.50	13.3%	3.2%
10	3	1.56	(1.39, 1.73)	1.54	1.51	3.2%	1.9%

Table 6.6: Mean Client Response Time Estimates from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model

automatically generated LQM, and the accuracy in their measurements ensures the accuracy of an LQM. The measured response times are the artifacts of the model building process and are only used to validate the LQM.

Table 6.11 on Page 109 and Table 6.12 on Page 109 show the utilizations and queue lengths for the CPU, Disk and the two servers in all the experiments. Since these metrics were not measured, comparisons are only possible between the automatically built model and the prescribed model. Good agreement exists between the two models with less than 7% error, except for the case of single client and single server thread combination. The utilization and queue length, however, are relatively small in this case, so even small discrepancies in the actual values would produce large relative errors.

The CPU and Disk utilizations increase with client population in a non-linear fashion; however, they do not seem to be affected by the number of server threads to any great extent. The same is also true for CPU and Disk queue lengths, but the effect of multiple threads is more noticeable especially for higher client population. It is noted that the thread package within the current DCE version is user level thread; with kernel threads the effect could be more significant.

Client Pop.	Server Threads	Measured $\bar{R}$		Prescribed Model $\bar{R}$ (secs)	Generated Model $\bar{R}$		
		(secs)	95% Confidence Interval		(secs)	% error to	
						measured	prescribed
1	1	0.18	(0.17.0.19)	0.23	0.23	21.7%	0%
5	1	0.19	(0.18.0.20)	0.25	0.25	24.0%	0%
5	3	0.20	(0.19.0.21)	0.28	0.28	28.6%	0%
10	1	0.21	(0.20.0.22)	0.28	0.26	19.2%	7.1%
10	3	0.24	(0.22.0.26)	0.36	0.35	31.4%	2.7%

Table 6.7: Mean Response Time Estimates of Server 1, Operation 1 from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model

Client Pop.	Server Threads	Measured $\bar{R}$		Prescribed Model $\bar{R}$ (secs)	Generated Model $\bar{R}$		
		(secs)	95% Confidence Interval		(secs)	% error to	
						measured	prescribed
1	1	0.25	(0.21.0.29)	0.3	0.32	21.9%	6.3%
5	1	0.28	(0.21.0.35)	0.33	0.25	12.0%	24.2%
5	3	0.34	(0.29.0.39)	0.37	0.38	10.5%	2.7%
10	1	0.32	(0.29.0.35)	0.37	0.39	18.0%	5.4%
10	3	0.50	(0.45.0.55)	0.47	0.47	6.4%	0%

Table 6.8: Mean Response Time Estimates of Server 1, Operation 2 from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model

Client Pop.	Server Threads	Measured $\bar{R}$		Prescribed Model $\bar{R}$ (secs)	Generated Model $\bar{R}$		
		(secs)	95% Confidence		(secs)	% error to	
			Interval			measured	prescribed
1	1	0.18	(0.17.0.19)	0.23	0.23	21.7%	0%
5	1	0.20	(0.19.0.21)	0.25	0.24	16.7%	4.0%
5	3	0.22	(0.21.0.23)	0.28	0.27	18.5%	3.6%
10	1	0.23	(0.22.0.23)	0.27	0.27	14.8%	0%
10	3	0.26	(0.24.0.28)	0.35	0.36	27.8%	2.9%

Table 6.9: Mean Response Time Estimates of Server 2, Operation 1 from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model

Client Pop.	Server Threads	Measured $\bar{R}$		Prescribed Model $\bar{R}$ (secs)	Generated Model $\bar{R}$		
		(secs)	95% Confidence		(secs)	% error to	
			Interval			measured	prescribed
1	1	0.19	(0.18.0.20)	0.23	0.24	20.8%	4.3%
5	1	0.19	(0.18.0.20)	0.25	0.25	24.0%	0%
5	3	0.21	(0.20.0.22)	0.28	0.29	27.6%	3.6%
10	1	0.19	(0.18.0.20)	0.27	0.27	29.6%	0%
10	3	0.26	(0.24.0.28)	0.35	0.35	25.7%	0%

Table 6.10: Mean Response Time Estimates of Server 2, Operation 2 from the Generated Model with Relative Errors Compared with the Measured Values and the Estimates from the Prescribed Model

Client Pop.	Server Threads	CPU		DISK		Server 1		Server 2	
		U^* %	Error	U^* %	Error	U^* %	Error	U^* %	Error
1	1	6	20%	0.5	20%	4.0	30%	3.4	26%
5	1	24	4%	2.7	4%	15.0	12%	14.8	3%
5	3	25	0%	2.7	4%	6.5	3%	5.6	4%
10	1	48	2%	4.9	4%	36.0	3%	29.8	4%
10	3	49	0%	5.0	2%	15.4	9%	13.3	1%

Table 6.11: Resource Utilization Estimates from the Generated Model with Relative Errors Compared with the Prescribed Model

Client Pop.	Server Threads	CPU		DISK		Server 1		Server 2	
		$\bar{Q}$	Error	$\bar{Q}$	Error	$\bar{Q}$	Error	$\bar{Q}$	Error
1	1	0.06	20%	0.006	20%	0.04	30%	0.03	0%
5	1	0.26	10%	0.027	0%	0.17	11%	0.16	7%
5	3	0.32	0%	0.027	0%	0.20	5%	0.17	6%
10	1	0.61	6%	0.050	4%	0.49	4%	0.34	6%
10	3	0.79	5%	0.053	0%	0.46	2%	0.40	0%

Table 6.12: Mean Queue Length Estimates from the Generated Model with Relative Errors Compared with the Prescribed Model

As for the two servers, the utilization is inversely proportional to the threading level: utilization is reduced by a factor close to the thread level when changing from single thread to multi-threads. The queue lengths at the servers, however, do not seem to be affected by the threading level, except when client population is high.

Having been validated for its reasonable accuracy in capturing the model parameters and providing performance estimates, the automatically built model can be used to further study the application system. From Table 6.11 on Page 109 and Table 6.12 on Page 109, performance problem can be easily identified. If the end to end client response does not meet performance targets, the utilizations and queue lengths of the resources need to be examined. The most utilized resource, be it hardware or software, is most likely the performance bottleneck. For example, in the case with 10 clients and single-thread servers. The CPU utilization is at 48%, while the server utilizations are around 30%. Changing the servers to three-threads reduces the server utilizations to around 14%, while the CPU utilization is even higher at 49%. The overall end to end client response time, however, has no significant change according to the estimates by the LQM. If a second CPU is added or CPU speed is doubled, the end to end client response will be greatly reduced. These “what-if” scenarios will be discussed in the next section.

6.3 Predicting System Performance with the LQM

The automatically built model can now be used to consider several alternative system configurations in order to answer “what-if” questions. For example, what is the mean response time for the system with 40 clients and server-threading levels of 3?

The mean response time for increased client population can be estimated with the LQM built for the system. The client population in the model is increased to the anticipated level, in this case 40 clients, and all other parameters remain the same. The predictive model is then solved with Method of Layers and the mean performance metrics are shown in Table 6.13 on Page 112. The mean end to end Client response time is now at 18 seconds, which is considered too long. Table 6.13

on Page 112 also reveals that the CPU is highly utilized, at 100%. **Server 1** is also utilized at 100% and has a long queue length of about 18 customers. The performance bottlenecks of the system are thought to be the CPU and **Server 1**.

To improve the performance of this particular system at the given load level, two options are considered:

1. *Option 1*: add a second processor to the system and assign **Server 2** to the new processor; and
2. *Option 2*: upgrade the current processor to double the CPU speed.

These two options are not exclusive and can be implemented together. In this section, however, they will be considered separately. The predictive model is modified to represent the proposed new system.

If Option 1 is to be implemented, a second processor is added to the system and **Server 2** is assigned to that processor. A new entity is created in the LQM and model structure is modified to have **Server 2** entity visit the new CPU. Performance estimates are shown in Table 6.14 on Page 112. The mean end to end response time is reduced by 82% to 3.2 seconds, which is considered acceptable. The original CPU still has a high utilization at 93.7%, which means that performance will severely be affected if there would be other unmanaged process on the same CPU.

If Option 2 is to be implemented, the CPU speed is assumed to be double and the service time is assumed linearly reduced by 50%. The LQM is modified by reducing all service times required from the CPU by 50%. The new model is solved with MOL and performance estimates are shown in Table 6.15 on Page 112. The mean end to end response time is reduced by 89% to 1.9 seconds, which is also considered acceptable. The CPU utilization is also at a high 95.1%.

It is found that both Option 1 and Option 2 can reduce the end to end response time to acceptable level. But in both case the CPU utilization is at a high level. Any other unmanaged load on the CPU may severely affect the performance of the system. Also for Option 1 the network delay can reduce the overall performance of the system.

Performance Metrics	Client	CPU	DISK	Server 1	Server2
Response Time $\bar{R}$, sec.	18.0	N.A	N.A	1.28	1.00
Utilization U	52.5%	100%	11.7%	100%	77.5%
Queue Length $\bar{Q}$	N.A	5.2	0.11	17.95	3.08

Table 6.13: Performance Estimates for the Original System with 40 Clients and 3 Server Threads

Performance Metrics	Client	CPU	CPU2	DISK	Server 1	Server2
Response Time $\bar{R}$, sec	3.2	N.A	N.A	N.A	0.61	0.38
Utilization U	16.5%	93.7%	83.5%	6.1%	83.4%	51.4%
Queue Length $\bar{Q}$	N.A	3.2	1.50	0.06	4.17	1.65

Table 6.14: Performance Estimates for the Modified System of Option 1 (Two CPUs), with 40 Clients and 3 Server Threads

Performance Metrics	Client	CPU	DISK	Server 1	Server2
Response Time $\bar{R}$, sec.	1.9	N.A	N.A	0.44	0.36
Utilization U	10.3%	95.1%	19.8%	64.6%	53.0%
Queue Length $\bar{Q}$	N.A	3.3	0.24	2.37	1.71

Table 6.15: Performance Estimates for the Modified System of Option 2 (Double Speed CPU), with 40 Clients and 3 Server Threads

The behaviour of these alternatives are predicted very quickly using the analytic techniques. Making such changes to the real system and measuring their effectiveness can be expensive and time consuming. Analytic performance models can help prune the number of alternatives for which actual changes and measurements must be made.

6.4 Summary

In this section, the model building infrastructure and technique is demonstrated. A synthetic application was instrumented, and its performance data was collected by the model building program for a number of system configuration setups. Layered Queueing Models were automatically built for the system for each setup, and the LQMs were solved using Method of Layers. Validation of the model was then performed to ensure its correctness in capturing the model parameters and its accuracy in providing performance estimates. It was found that the LQMs were built with correct visit ratios and service times. The end-to-end client response times estimated from the LQMs were within 95% confidence interval of measured values. The response times at individual server operations are less accurately, particularly for multi-thread servers. This may have more to do with the instrumentation methods, where the measurement macros are embedded within the server operations. Better techniques or (future) middleware services for measurements should be used.

The only parameter that was not measured was the disk service time, which was estimated. The disk visits was not directly measured either, instead the application counted its own I/O's, which is pessimistic. These less accurate measurements have some impact on the accuracy of the model. Further investigative work needs also to be done in this area.

Chapter 7

Conclusions

7.1 Summary

This thesis attempts to solve the problem as posed in Chapter 1:

“Given an operational distributed software application, construct a Layered Queueing Network Performance Model (LQM) [29] which includes structural information and resource demand parameters.”

This thesis work started with a review of the background in distributed computing and a survey of the current distributed computing environments. Performance problems that may appear in distributed applications developed with these environments were identified, and performance evaluation models and techniques were described. Among these models and techniques, Layered Queueing Model and Method of Layers are the most suited for modelling distributed applications which have both hardware and software contentions.

The automatic performance model building technique described and demonstrated in this thesis focus on synchronous interactions between concurrent computer programs. It exploits an integrated application management architecture that is based on the DMS [11]. Extensions to DMS were added to support building Layered Queueing Model. An implementation of the architecture was developed that includes:

1. an instrumentation package to gather data from within application processes:
2. a performance data server that acts as an intermediate store and analyzer for performance information: and.
3. a model builder that reflects measured information in models. solves the models for performance estimates. verifies and validates the models. The resulting model is then used to perform management tasks like considering "what-if" scenarios and capacity planning.

This technique is possible once the application or its prototype is operational. In order to collect performance data, the application or its prototype need to be instrumented. The instrumentation can be included in class libraries or runtime environments to provide immediate model-building support for the environment's applications. The instrumentation captures interactions between operations of process interfaces: such interactions are currently not captured by operating systems.

Problems may arise when relying on automated model building system, so the system presented in this thesis is designed to overcome some of the problems. Separate information about each entry's physical resource demands is not available if the regression performed by the performance data server fails. In this case the data server simply return the average physical resource demands per completion at the process, however, the visit counts are still characterized correctly.

The confidence intervals for data reported by the performance data server are used to validate the measurement data. Poor confidence intervals suggests that either not enough measurement data was collected to characterize the system's behaviour, or some significant changes to the system took place during the measurement period. Further exploratory work can then be done, for example, the measurement period can be partitioned into into several shorter periods and give models for each.

There is always likely to be unmanaged workload on the same system where the managed application is running. Although presently not characterized in the model, the unmanaged processes will eventually be treated as client processes that do not

import interfaces: their resource consumption can be collected and forwarded to the performance data server by proxy observers.

Currently this model building technique assumes that each operation of each interface is accessed and supported using synchronous RPC. Other interaction types including asynchronous, deferred synchronous, and forwarding need to be considered and integrated into the model building system. This is considered as future work.

Contributions to research by this thesis can be summarized as below.

1. Documented a distributed application performance management architecture and its implementation.
2. Developed techniques to estimate performance parameters for analytic models from performance measurements.
3. Developed a tool that generates performance models from measurements.
4. Identified and supported exceptional conditions in model building process.

7.2 Conclusions

Gathering performance data “by hand” and reflecting it in performance models is a time-consuming and error-prone process. We believe that automated-performance model building is necessary for the successful performance management of distributed applications. Through this thesis, we exploit a distributed application performance management architecture and its implementation, demonstrated techniques to estimate performance parameters for analytic models from performance measurements, and developed a tool that generates performance models from measurements. Exceptional conditions may occur during the model building process and they can be handled appropriately.

Distributed performance management tools can and should play an important role in the software performance engineering process. Early in the design cycle when the prototype of an application has been built, the performance management tool

can be applied to the prototype system to collect performance measurements and to build performance models. This resulting model can be augmented to consider the not yet built aspects of the system. Also, with the help of performance management tools, performance of existing systems can be better understood, and more accurate performance data can be obtained for the reusable software components. This will definitely benefit future design iterations for the software.

With automated model building and performance evaluation, performance problem can be more accurately identified, and different options to fix the problem can be studied with the validated performance model before their actual execution. Furthermore, automated model building and performance evaluation support system design, configuration and capacity planning by considering the performance impact of:

1. different placements of application functions across software servers, i.e. composition issues;
2. different placements of servers on nodes in a distributed system, i.e. distribution issues;
3. different threading levels for servers; and,
4. increases or changes in workloads or faster hardware technologies.

7.3 Future Work

This thesis only considers synchronous remote procedure calls. Asynchronous interaction should be considered in the extension of this work. Techniques for characterizing unmanaged background load and reflecting it in the model are also needed. Network delay should also be considered as with the background load. Support for confidence intervals is needed from the performance data server. Support for multiple interfaces per process and keeping track of connections between processes are also needed.

The difficulties that were found in carrying out the project are with the support, or more precisely the lack of, provided by operating system and middleware for performance monitoring and analysis. The CPU resource usage can only be obtained for a process as whole, whereas resource usages for each operation of a process is more appropriate. The disk service time and the number of physical I/Os are hard to get as well. Though DCE provides the server operations protocol sequence, i.e., IP address, server identifier, and operation identifier, but it does not currently provide the such details for calling processes (to the server processes). The connections between client and server operations are not readily available. Work is currently in progress for the middleware vendors to provide performance instrumentations [13], which will make performance monitoring and performance modelling much easier.

References

- [1] A.O. Allen. "Queueing Models of Computer Systems". *IEEE Computer*, pages 13-24. April. 1980.
- [2] Architecture Projects Management Ltd.. *Application Programming in AN-SAware*, release 4.1. February 1993.
- [3] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers", *JACM*, vol.22, no.2, pages 248-260. April 1975.
- [4] M.A. Bauer, P.J.Finnigan, J.W. Hong, J.A. Rolia, T.J. Teorey, G.A. Winters, "Reference Architecture for Distributed Systems Management", *IBM Systems Journal* Vol.33, No.3, pages 426-444, 1994.
- [5] A.D. Beitz, P.W. King and K.A. Raymond, "Is DCE a Support Environment for ODP?", in *Open Distributed Processing*, II (C-20), J. de Meer, B. Mahr and S. Storp (Editors), Elsevier Science B.V. (North-Holland), 1994
- [6] R. Buhr and R.S. Casselman, "Timethread - Role Maps for Object-Oriented Design of Real-Time and Distributed Systems", *ACM SIGPLAN Notices*, Vol. 29, No. 10, Page 301, 1994.
- [7] R.J.A. Buhr, G.M. Karam, C.M. Woodside, R. Casselman, R.G Franks, H. Scott, and D. Bailey, "TimeBench: a CAD Tool for Real-Time System Design", *Proceedings of the 2nd International Symposium on Environments and Tools for Ada (SETA2)*, Washington D.C., January 1992.

- [8] K.M. Chandy and D. Neuse. "Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems". *Communications of the ACM*. 25(2):126-134. February 1982.
- [9] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. 2nd Ed., Addison-Wesley, 1994.
- [10] G. Franks, A. Hubbard, S. Majumdar, D. Petriu, J. Rolia, and C.M. Woodside. "A Toolset for Performance Engineering and Software Design of Client-Server Systems". *Performance Evaluation Journal*, Vol.24, pages 117-135, North Holland, November 1995.
- [11] R. Friedrich, J. Martinka, T. Sienknecht, S. Saunders. "Integration of Performance Measurement and Modeling for Open Distributed Processing." *Proceedings of International Conference on Open Distributed Processing (ICODP'95)*. Brisbane Australia, February 1995, pages 341-352.
- [12] R. Friedrich and J. Rolia. "Applying Performance Engineering to a Distributed Application Monitoring System." in *Proceedings of the International Conference on Distributed Processing*, ICDP '96, Dresden, Germany, February, 1996.
- [13] R. Friedrich. "Requirements for Performance Instrumentation of DCE RPC and CDS Services." *Request for Comments:32.0*, OSF DCE SIG, June 1993
- [14] P. Heidelberger and S.S. Lavenberg. "Computer Performance Evaluation Methodology". *IEEE Transactions on Computers*, Vol.c-3, pages 1195-1220, No.12, December 1984.
- [15] A. Herbert. "An ANSA Overview". *IEEE Network*, January/February 1994.
- [16] G. Hills, J. Rolia, and G. Serazzi. "Performance Engineering of Distributed Software Process Architectures". *Lecture Notes in Computer Science*, Springer Verlag, No. 977, pages 79-85, 1995.

- [17] C.E. Hrischuk, J. Rolia, and C.M. Woodside. "Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype". Proceedings of International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95), p. 399 - 409. Durham, North California, January 1995.
- [18] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York, 1991.
- [19] F. Lange, R. Kroeger, M. Gergeleit. JEWEL: Design and Implementation of a Distributed Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 6, November 1992, pages 657-671.
- [20] E.D. Lazowska, J. Zahorjan, G.S. Graham and K.C. Sevcik. "Quantitative System Performance. Computer System Analysis Using Queueing Network Models". Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
- [21] T.J. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley/OMG, 1995.
- [22] J.E. Neilson, C.M. Woodside, D.C. Petriu, and S. Majumdar. "Software Bottlenecking in Client-Server Systems and Rendezvous Networks". *I.E.E.E. Transactions on Software Engineering*, Vol. 21, NO. 9, September 1995.
- [23] Open Software Foundation. *Introduction to OSF DCE*. Prentice-Hall, 1992.
- [24] E. Pozzetti, V. Vetland, J.A. Rolia, and G. Serazzi. "Characterizing the Resource Demands of TCP/IP". B. Hertzberger, G. Serazzi Eds., *Lecture Notes in Computer Science*. Springer Verlag, No. 991, pages 79-85, 1995.
- [25] M. Qin, R. Lee, A. El Rayess, V. Vetland, and J. Rolia. "Automatic Generation of Performance Models for Distributed Application Systems", on the CD-ROM for CASCON '96, Toronto, November 6-12, 1996.

- [26] K.A. Raymond. "Reference Model of Open Distributed Processing: a Tutorial". in *Open Distributed Processing. II (C-20)* J. de Meer, B. Mahr and S. Storp (Editors). Elsevier Science B.V. (North-Holland). 1994.
- [27] M. Reiser and S.S. Lavenberg. "Mean Value Analysis of Closed Multichain Queueing Networks". *Journal of the ACM*. 27(2). pages 313-322. 1980
- [28] J.A. Rolia. "Predicting the Performance of Software Servers". *CSRI Technical Report 260*. University of Toronto, Canada. January 1992.
- [29] J.A. Rolia and K.C. Sevcik. "The Method of Layers". *I.E.E.E. Transactions on Software Engineering*. Vol. 21. pages 689-700. August. 1995.
- [30] J.A. Rolia. "Distributed Application performance. Metrics and Management". in *Open Distributed Processing. II (C-20)*. J. de Meer, B.Mahr and S. Storp (Editors). Elsevier Science B.V. (North-Holland). pages 235-246. 1994.
- [31] J.A. Rolia, V. Vetland, and G. Hills. "Composition and Distribution: Performance Engineering for Distributed Applications". in the Proceedings of IBM Canada/NRC CASCON '95. November. 1995.
- [32] J.A. Rolia and V. Vetland "Parameter Estimation for Performance Models of Distributed Application Systems". in the Proceedings of IBM Canada/NRC CASCON '95. November. 1995.
- [33] M.T. Rose. *The Simple Book*. 2nd ed. Englewood Cliffs, N.J.. PTR Prentice Hall. 1994.
- [34] D.C. Schmidt and T. Suda. "Transport System Architecture Services for High-Performance Communication Systems." *IEEE Journal on Selected Areas in Communication*. Volume 11. pages 489-506. May 1993.
- [35] P. Schweitzer. "Approximate Analysis of Multiclass Closed Networks of Queues". in *Proceedings of the International Conference on Stochastic Control and Optimization*. Amsterdam. 1979.

- [36] F. Sheikh, J. Rolia, P. Garg, S. Frolund, A. Sheppherd, *Layered Modelling of Large Scale Distributed Applications*, in the proceedings of the 1st World Congress on Systems Simulation, Quality of Service Modelling, Singapore, September 1-3, 1997, pages 247-254.
- [37] C.U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, August 1990.
- [38] C.U. Smith and L.G. Williams, "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives", *IEEE Transactions on Software Engineering*, Vol.19, No.7, July 1993.
- [39] W. Stallings, William, *SNMP, SNMPv2, and CMIP: the practical guide to network-management standards*, Reading, Mass., Addison-Wesley, 1993.
- [40] A.S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, 1992.
- [41] V. Tschammer, K. Eckert, J. Hall, G. Schurmann and L. Strick, "OAI - Concepts for Open Systems Cooperation", *Lecture Notes on Computer Science 433 - Progress in Distributed Operating Systems and Distributed Systems Management*, Springer-Verlag, 1989.
- [42] ISO/IEC JTC1/SC21/WG7 N885, "Reference Model for Open Distributed Processing - Part 1: Overview and Guide to Use", November 1993.
- [43] ISO/IEC 10746-2, "Basic Reference Model of Open Distributed Processing - Part 2: Descriptive Model", July 1993.
- [44] ISO/IEC 10746-3, "Basic Reference Model of Open Distributed Processing - Part3: Prescriptive Model", July 1993.
- [45] 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 Transaction on Computers*, Vol.44, No.1, pages 20-34, January 1995.

- [46] C.M. Woodside and S. Majumdar. "Robust Bounds and Throughput Guarantees for General Closed Multiclass Queuing Networks". *SCE Technical Report SCE-94-05*. Carleton University, Ottawa, Canada. January 1994.
- [47] C.Q. Yang and B.P. Miller. Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach. *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, December 1989, pages 1615-1629.
- [48] V. Vetland. "Measurement-Based composite computational work modeling of software". PhD thesis, Norwegian Institute of Technology, University of Trondheim, September 1993.

IMAGE EVALUATION TEST TARGET (QA-3)

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

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