1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Performance Analysis of Distributed Process Architectures

by

Captain Greg Hills, B.Sc Computer Science

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

Ottawa-Carleton Institute for Computer Science

SCS

**Carleton University
Ottawa, Ontario
May 1996**

**© Copyright
1996, Captain Greg Hills**

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-612-13830-5

Canada

Name GREGORY HILLS

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science

SUBJECT TERM

U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0307
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Science 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
General 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce-Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0569
Limnology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Polynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463
Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0745
Sanitary and Municipal 0554
System Science 0790
Geotechnology
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
Performance Analysis of Distributed Process Architectures
submitted by
Captain Greg Hills, B.Sc Computer Science
in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, SCS

Thesis Supervisor

Thesis Co-Supervisor

Carleton University

May 14, 1996

Abstract

Performance engineering is a vital step in the life cycle of a distributed application system. When designing a new system, it is critical to know if the user's performance requirements will be met and whether the proposed distributed system infrastructure can meet the workloads placed on it. We define the utilization constraint of a resource as the maximum number of requests allowed to use the resource concurrently. If workloads place high demands on the system's resources, utilization constraints may be approached causing bottlenecks to arise. Software bottlenecks can limit the application throughput even if the devices are not fully utilized. The composition of operating system processes and their distribution across nodes in a network comprises a configuration of an application. Alternative configurations have different bottleneck characteristics. This thesis describes a method to help a performance analyst find configurations for distributed applications such that software bottlenecks are unlikely to occur. A tool called the Distributed Application System Sizer (DASS) was also developed to help apply the method and is also described.

Acknowledgements

I would like to thank my certain people for whom it would have been impossible for me to complete this thesis. Firstly, I would like to thank my supervisors Professors Jerome Rolia and John Neilson. If it was not for their patience and understanding I might still be writing. I also need to thank Giuseppe Serazzi who created the Asymptotic Bounds Analysis tool on which my thesis is based. A note of thanks must go to Systemhouse Ltd. and specifically Bernie Wong for their help in my case study. The aid they provided was of great importance to my thesis. I must also thank my wife Dana-Lynne. When I started writing this thesis we were simply engaged. She endured the time I spent at school working on my degree and thesis while all the time trying to organize a wedding. Thank you Dana-Lynne for all of your understanding. Lastly, but definitely not least, I would like to thank my parents for their love and guidance. Throughout my life they have insisted on one basic principle - education is the key to success. I have tried to incorporate that principle into my life and I thank them for everything they have done for me.

Contents

Abstract	iii
Acknowledgements	iv
List Of Tables	vi
List Of Figures	vii
1 Introduction	1
1.1 Distributed Environments	2
1.1.1 Distributed Computing Environment	4
1.1.2 CORBA	8
1.1.3 CICS/6000	11
1.2 Structures of Distributed Environments	13
1.2.1 Utilization Limits	15
1.3 Software Performance Engineering	16
1.3.1 Building Models	17
1.3.2 Bounds Analysis and Queueing Models	19
1.4 Existing Performance Tools	20
1.4.1 SES/Workbench	20
1.4.2 A Performance Modeller	21
1.4.3 PARASOL	22

1.4.4	Timebench	23
1.4.5	Tool Summary	24
1.5	Object Based Performance Analysis	25
1.6	Thesis Outline	26
2	Performance Evaluation	28
2.1	Queueing Network Models	30
2.2	Bounds Analysis	31
2.2.1	Asymptotic Bounds	31
2.2.2	Maximum Software Utilizations Using Bounds Analysis	35
2.3	Mean Value Analysis	39
2.3.1	Layered Queueing Models	41
2.4	Performance Evaluation Methodology	42
3	Distributed Application System Sizer	45
3.1	DASS as a Performance Tool	46
3.2	Engineering Structures and Model Parameters	51
3.2.1	Node Parameters	52
3.2.2	Defining the Software Structures	54
3.2.3	Workload Definition	56
3.3	Data Input	57
3.3.1	Add a New Model	58
3.3.2	Edit an Existing Model	64
3.3.3	Modifying a Configuration from a Single View	68
3.3.4	Automated Data Input	70
3.4	Using DASS and ABA Together	71
3.4.1	Outputs of DASS	72
3.5	Summary	74

4	Performance Study of an Example Application	76
4.1	Description of the Application	77
4.2	System Model Definition	79
4.3	Performance Analysis	82
4.3.1	Workload Selection	84
4.3.2	Utilization Analysis	85
4.3.3	Alternate Configuration	87
4.4	Summary	93
5	Case Study	94
5.1	Project Description	94
5.1.1	System Description	95
5.2	Model Description	97
5.2.1	Workload Definition	100
5.3	Alternate Configuration	104
5.4	Summary	109
6	Conclusions	110
6.0.1	Summary	110
6.0.2	Future Research	112
	References	114
	Glossary	118

List of Tables

2.1	Service demands for QNM	33
3.1	Communication costs	55
3.2	Example of request paths	57
4.1	Node service parameters	79
4.2	Composition strategy 1	80
4.3	Resource requirements	82
4.4	Request paths	83
4.5	Communication costs	83
4.6	Configuration 1's Hardware Resource Demands by Request Type . . .	83
4.7	Configuration 1 - Request throughputs at points of interest	86
4.8	Capsule demands by request type in seconds	86
4.9	Composition strategy 2	87
4.10	Configuration 2's Hardware Resource Demands by Request Type . . .	89
4.11	Configuration 2 - Request throughputs at points of interest	91
4.12	Configuration 2's capsule demands by request type in seconds	91
5.1	Case study - node service parameters	98
5.2	Case study - software composition	101
5.3	Case study - method resource requirements	102
5.4	Hardware Resource Demands by Request Type	103

5.5	CFSSU Request throughputs at points of interest	104
5.6	Case study - node service parameters of alternate configuration . . .	104
5.7	Hardware Resource Demands by Request Type	106
5.8	CFSSU Request throughputs at points of interest	107

List of Figures

1.1	Remote procedure call	3
1.2	OSF's Distributed Computing Environment components	5
1.3	Example DCE IDL and client application code	6
1.4	DCE RPC runtime	8
1.5	Example CORBA IDL and client application code	10
1.6	CORBA communication process	11
1.7	CICS/6000 environment layers	12
1.8	Engineering structures of RM-ODP	14
1.9	Execution Models Used By Smith	18
2.1	Network of queues in a closed system	30
2.2	Queueing network model of bounding analysis example	33
2.3	Asymptotic throughput bounds	34
2.4	Possible combinations of request classes	37
2.5	Bottleneck crossover points for a two request type/two device system	38
2.6	Performance Evaluation Methodology	43
3.1	Path of a request through the system	47
3.2	Flow of control of a request	49
3.3	Communications between object's methods	50
3.4	Data Structures of DASS	51
3.5	Main menu screen	59

3.6	New model screen	60
3.7	Add a node screen	61
3.8	Add a capsule screen	61
3.9	Add a cluster screen	62
3.10	Add an object screen	62
3.11	Add a method screen	63
3.12	Add a request screen	64
3.13	Add the method calls screen	65
3.14	Node Entry Form	66
3.15	Capsule entry form	66
3.16	Cluster entry form	67
3.17	Object entry form	67
3.18	Method entry form	68
3.19	Configuration modification screen	69
3.20	Run analysis screen	71
3.21	Included service of resources	73
3.22	Example utilization graph	74
4.1	Computational model of a simple retail store	77
4.2	Configuration 1 of example application	81
4.3	Saturation sectors for configuration 1 of the example application . . .	85
4.4	Simple retail store: capsule utilizations for configuration 1	88
4.5	Configuration 2 of example application	89
4.6	Saturation sectors for configuration 2 of the example application . . .	90
4.7	Capsule Utilizations for Configuration 2	92
5.1	Configuration of WH&DC application	99
5.2	Saturation Sectors for the Canadian Forces Supply System Upgrade .	103
5.3	Canadian Forces Supply System Upgrade Utilization Results	105

5.4	Saturation Sectors for the Canadian Forces Supply System Upgrade	
	- configuration 2	107
5.5	Canadian Forces Supply System Upgrade Utilization Results	108

Chapter 1

Introduction

The performance engineering of computer systems has always been important in the business world. Well performing systems are critical to many business environments. When most of a company's processing was done on a single mainframe, the performance engineering focused mainly on the hardware of the system. If the system was getting slow or needed more capacity, an upgrade to the existing hardware to increase its speed was the usual solution. As companies move away from the centralized systems towards the more flexible distributed computing environments, the focus of performance engineering shifts from strictly hardware to include the software. A *distributed application system* enables application software to exploit the distributed environment it is built on. Software development teams must ensure that the distributed application systems they are developing will satisfy current workloads and those of the future.

Performance engineering is an important process in the life cycle of a distributed application system. It should start during requirements analysis, continue during design and be applied even after the application is deployed. 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. We assume that as the demands increase, a distributed

application system will need more hardware support.

In this thesis, we define the procedure of building the software processes from software objects as *composition*. These software objects may be the methods or procedures that make up a distributed application and they will be described in more detail in Section 1.2. *Distribution* is defined as the placement of the software processes across nodes in a network [1]. The combination of composition and distribution of the software processes defines a *configuration* of the distributed application. We explore techniques to choose software composition and distribution strategies for applications in order to avoid software bottlenecks. A system that can be easily re-configured to support increasing demands without incurring software bottlenecks is defined as being *performance scalable*. Distributed application systems are fast becoming the dominant solution strategy for small and large companies alike. We now consider several features of distributed environments.

1.1 Distributed Environments

The main feature of a distributed environment is the distribution of processing across its nodes. There are several reasons why companies are moving towards distributed application systems [2]. One reason is to share information between users with otherwise standalone workstations. A second reason is the ability to have specialized servers. A company that has a specialized service, for example a print service, may provide a single computer to handle all requests for that service. A third reason to move towards a distributed environment is economy. Having many smaller workstations performing the work may be more cost effective than having a single computer. Distributed environments have several characteristics that allow them to provide the services required and to communicate effectively.

Many distributed applications built in distributed environments can be modelled effectively with client/server interactions. A single distributed application may be comprised of many clients and servers interacting together. The clients and servers usually reside on separate nodes and communicate via a network infrastructure. The

Figure 1.1: Remote procedure call

terms *client* and *server* are relative labels in that a 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, communication is handled by a communication mechanism. An example of a communication mechanism is the remote procedure call (RPC) [3] and is supported by an RPC service. Figure 1.1 on page 3 illustrates the RPC service. Using this service, the client process makes a simple procedure call on its own node which is shown as step 1 in Figure 1.1. The underlying RPC mechanism translates this procedure call into network communications which are received on the server's node, step 2. The server node's RPC service converts the information into a format the server process can use and sends the information to the process, step 3. The server process executes the request and returns the results to the client process, steps 4, 5 and 6.

A key feature of many distributed environments is to have a *location transparency*. With this transparency, a client process does not need to know where the server process resides when communication takes place. This problem is usually supported using a directory name service agent. A directory name service eliminates the need to hardcode the address of server processes or to maintain databases of hosts

and server locations on the client nodes. When a server starts up or shuts down, it informs the directory name service agent which either adds or removes the server from its directory. If a client process needs the location of a server, it queries the directory service for the address information. The server and client processes know the location of the directory name service as it typically resides at a well known address.

A fourth feature of distributed system is the security service. In a non-distributed system, the operating system can often protect the resources. In a distributed environment the network is an accessible resource where messages can be observed or forged. The security service within a distributed environment protects resources from unauthorized access and can also offer privacy through the use of data encryption.

There are many distributed environments available today which implement these features. Middleware environments such as DCE, CORBA and CICS/6000 are examples of distributed environments discussed in the current literature. The following sections describe these distributed environments, how they implement these features, and their impact on application architectures. We believe that other middleware environments have similar characteristics of those discussed in the following sections. The issues of composition and distribution are very prominent in distributed application systems. Choosing which services to provide in each server process and where to place these servers are issues that must be decided before implementation.

1.1.1 Distributed Computing Environment

The architecture of Open Software Foundation's (OSF) Distributed Computing Environment (DCE) [2] is shown in Figure 1.2 on Page 5. The components of DCE include the DCE RPC service which handles the communications between a client and a server. The DCE security service used for secure communications. The DCE directory service named the Cell Directory Service (CDS) to achieve the location transparency for the client/server communications. The DCE distributed time service to coordinate the time between nodes and the DCE distributed file service which

Figure 1.2: OSF's Distributed Computing Environment components

provides a single shared file system for the environment.

To facilitate the communications between the client and the server, DCE provides the DCE RPC service. The components of the RPC service include the Interface Definition Language (IDL) and the RPC runtime. The interface of a server is defined using the DCE IDL. Figure 1.3 on Page 6 illustrates an example of DCE IDL. The IDL file is compiled into three parts by an IDL compiler. These parts are a client stub, a server stub and a header file.

To create a client application, a programmer will write code that makes calls to the routines in the IDL client stub. The client stub, RPC runtime and client code are linked together to produce the client application. The server application is created in the same way. The programmer must provide the body for the operations defined by the IDL.

When a client application needs to make a request to a server, it must be able to find the server's location. This process of connecting to the server process is called *binding*. The location needed by the client consists of two parts, the address of the

```

/*****
** 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 1.3: Example DCE IDL and client application code

machine where the server resides and the network endpoint of the server process. This network endpoint is the address for the process, for example a UNIX port number. The client may obtain the node address of the server in several ways. The value may be hard coded into the client application, it may be available from a file or an environment variable. However, these methods are not very flexible in that if the server's address changes or if the client process moves to another node then the server's correct address may not be available. A more flexible way for a DCE client to obtain a server's address is to use the DCE Cell Directory Service (CDS). On initialization in a DCE environment, each server process must register its interfaces, communication protocols and address with the CDS. A client application can then query the CDS for the node address of a server that has the interface it needs.

The execution of the RPC is transparent to both the client and server processes. Once the connection has been made, the client stub converts the information into an internal RPC format which is then transmitted over the network by the runtime library. The runtime library on the server node receives this information and passes it to the server stub which converts the information into a format that the process can use. This conversion step is called *marshalling* and *unmarshalling*. After the server process has completed the request, the procedure is reversed. Figure 1.4 on Page 8 illustrates the communication procedure between a client and server in DCE.

DCE also provides a facility for secure RPCs. The security service provides mechanism for the authentication, authorization and privacy of the requests between the client and server processes. The secure RPC protects the distributed resources from an unauthorized access. This is accomplished using a server that supports a public key encryption service and encryption algorithms during the marshalling and unmarshalling process.

Composition issues are dealt with during the design of the server processes. The interfaces included in each separate server must be decided before the server is compiled. Different compositions could be supported within a DCE application by recompiling the servers for different sets of interfaces or by letting the servers export

Figure 1.4: DCE RPC runtime

subsets of its interfaces. The distribution issues are managed when deploying the servers across the network. Multiple distribution can be easily configured by moving the server processes from one node to another. The CDS will aid the client processes in locating the servers in a new distribution.

1.1.2 CORBA

The Common Object Request Broker Architecture (*CORBA*) [4, 5] from the Object Management Group (OMG) is a specification that allows client applications to make requests and receive responses from objects. These applications and objects can be located across a network and reside on different machine in a heterogeneous distributed environment. The core of the architecture is the Object Request Broker (*ORB*) which provides the interoperability between the different applications. The ORB core is responsible for locating an object implementation that can service requests made by a client application. The ORB prepares the object to receive a request and communicates the data between the client and object. At a first glance

this architecture may seem like the ORB would become a severe bottleneck in the system, however, it is the case that the ORB itself is composed of many parts which are distributed across the nodes in the environment.

The client performs a request by sending an object reference and specific operation to be performed to the ORB. The object reference signifies the type of object to satisfy the request, not a specific object. This allows 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 client initiates a request by calling a stub routine which is specific to an object type. This stub routine is a defined interface written in an Interface Definition Language (IDL). It is the stub routine that defines the object type and its arguments. Figure 1.5 on Page 10 gives an example of the IDL in CORBA. The IDL gives a consistent interface to the client regardless of where the object is located or what language the object is implemented in.

The objects in the architecture perform the work on behalf of the requests for service. An object's implementation provides its semantics and defines the data needed to service the requests. An object must register its references and definitions so that the ORB Core can transmit a request for service to the correct object implementation. This registration is done through the Object Adapter. The Object Adapter is also responsible for the generation of the object references, method invocation, the security of interactions and the mapping of object references to implementations. When a client sends a request to the ORB, the ORB interfaces with the Object Adapter to relay the request to an appropriate Object for service. Figure 1.6 on Page 11 illustrates the communication process between a client and an object.

Like DCE, the composition and distribution issues can be managed very readily in a distributed application system using CORBA. Composition is managed by deciding which interfaces will be provided in a single server process. Based on this information the server process IDL is then coded and the server object compiled. If a new composition is needed, then the various interfaces to the server objects need to be rearranged and the servers need to be rebuilt. The distribution issue is also dealt with a manner similar to DCE. The ORB core is responsible for locating a


```

/*****
** Interface Definition Language          **
**                                     **
** Example - greet                      **
*****/
interface greet
{
    string op1(in long x, inout string y, out string z);
};

/*****
** Client IDL (Orbix/INS bindings) code  **
*****/
char *result;
long x;
char *y;
char *z;
greet* myGreet;

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

```

Figure 1.5: Example CORBA IDL and client application code

Figure 1.6: CORBA communication process

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

1.1.3 CICS/6000

The Customer Information Control System (CICS) [6] from IBM is an on-line transaction processing environment. Users submit transaction requests to the transaction servers and CICS creates *tasks*¹ to complete the work. CICS/6000 is a similar system that can run on the IBM Risc/6000 and HP UX systems in distributed environments. CICS/6000 systems provide the support for the distribution of both function and data. Client processes that provide the interface for the users to the system may reside on separate nodes than the transaction servers. The transaction servers run the server applications and resource managers providing file, database or communication services. This configuration allows an application to use resources spread over a distributed network.

¹A task can be compared to an object's method. Many tasks may be executed to perform a single request.

Figure 1.7: CICS/6000 environment layers

CICS/6000 is a layered, open architecture using AIX components. Components in the layers are based on elements of the Open Software Foundation's DCE and Encina. CICS/6000 uses the DCE Name service, Directory service, RPC and Time service for all intercommunications between clients and servers. The transaction management service are provided by Encina. Encina provides two key services to CICS/6000, a logging facility for recovery services and a structured file service which is required by all CICS applications. All three layers of this environment - AIX, DCE and Encina - provide a set of services and application programming interfaces (APIs) to aid in the development of a distributed transaction processing system. CICS provides application programmers an interface that accomplishes location transparency that hides the complexity of the underlying distributed environment from the client applications. Figure 1.7 on Page 12 shows the relationship between the different layers of CICS/6000.

1.2 Structures of Distributed Environments

To study the performance of distributed application systems we need a model that reflects the architecture of the distributed environments technologies they exploit. The model must be able to describe the relationship between the software components and the hardware. We want to be able to model a distributed application system that is built using any of the above distributed environments. The engineering structures of the Reference Model of Open Distributed Processing (RM-ODP) [7, 8, 9, 10] provides such a model.

The RM-ODP describes a distributed system as layers of software components which are then allocated over the nodes in the system. Figure 1.8 on Page 14 depicts the software and hardware components of RM-ODP which are:

Basic Engineering Objects (BEOs) : these objects are the building blocks of the application. They contain the *methods* (procedures) that perform the processing in the system;

Clusters : are groupings of related objects. Clusters are an abstraction that help us determine the composition of an application and have no other impact on performance;

Capsules : these are the operating system processes required by the application; and

Nodes : these are the stations in a distributed environment.

The RM-ODP has sufficient abstraction to support the modelling of different distributed environments mentioned previously. A CICS/6000 application is a collection of operating system process that executes user defined tasks. This can be modelled using one capsule for each of the distributed processes with the user programs becoming the BEOs. In DCE and CORBA, application processes are the capsules. Their interfaces define the services that are made available to the objects in other capsules and are defined as clusters in the RM-ODP.

Figure 1.8: Engineering structures of RM-ODP

For performance engineering we must augment the engineering structures of RM-ODP to include request path and resource demand information. The workload of a distributed application system is made of different request types. Each request type has a different path through the objects' methods that make use of the system's physical resources. These paths are similar to Jacobson's *use cases* [11] and Buhr's *timethreads* [12]. The amount of these resources must be quantified and is discussed further in Section 1.3. The augmented model is used to build our workload model for performance evaluation.

The problem now is how to allocate the objects to capsules and how to configure the system to meet its performance objectives. With the engineering structures of RM-ODP, we are able to make the following design choices:

- allocation of objects to clusters (*composition*);
- allocation of clusters to capsules (*composition*); and
- allocation of capsules to nodes (*distribution*).

These design choices affect the performance behaviour of the distributed application system. Furthermore, software and hardware technology constraints limit the concurrent use of the system's software and hardware resources. The next section

describes these technology constraints and how they can impact on the composition and distribution strategy for a distributed application system.

1.2.1 Utilization Limits

The maximum number of concurrent users allowed to use a resource is limited by the technology implementing the resource. This technology constraint is defined as the *utilization limit* of a resource. A hardware resource usually has a concurrency limit of one. At most one customer may use a disk or CPU at any given time ².

The technology implementing a software resource may also impose a utilization limit. The value of this limit depends on the characteristics of the software resource. A critical section within an application that can be used by at most one customer at a given time must be protected. Its utilization limit is one. In some cases, the utilization limit of a process may be greater than one. A database subsystem in a distributed application may allow n customers to be connected at the same time. In this case the utilization limit of the subsystem would be set to n .

At the node and capsule levels, the number of concurrent processes supported by the node or the number of active threads supported by the process may be limited and set to some integer l . This constraint placed on the node and operating system processes limit the number of concurrent users. In this case the utilization limit of the node would be l .

Performance modelling techniques can be used to predict whether a distributed application system will exceed its software or hardware technology constraints. As workloads change, the demands on the system resources increase and the utilization levels of the software resources also increase. If the demands are high enough, the utilization levels will start to approach the utilization limits of the resources and queueing delays will arise. If a software component utilization reaches the limit placed on it by the software technology then it is a software bottleneck and will limit the application throughput. In this case the application may not be performance

²This may no longer be true for more advanced devices, but is offered as a simple example.

scaleable since it can not fully exploit its underlying hardware. To help identify and overcome potential software bottlenecks, distributed application developers can apply software performance engineering [13].

1.3 Software Performance Engineering

The performance engineering of software dates back to the start of modern computing. The goal of software developers was to write small efficient code that satisfied the memory and processing constraints of early computers. As technology advanced, the processing speed of computers increased, main memory became cheaper and virtual memory became available. Performance engineering was considered to be less crucial.

Software development teams often used a lifecycle model where performance evaluation was considered in the late stages of the cycle. One of the first development strategies used in software engineering was the waterfall method. The basic stages in this strategy are *analysis, design, coding, testing, implementation* and *maintenance*. Performance problems were usually detected during integration testing or later.

To try and detect problems earlier in the lifecycle of software systems newer techniques were adopted, these included the iterative or spiral strategies and the rapid prototyping method. Unfortunately all of these techniques adopted a fix-it-later attitude towards the performance aspects of a software system [13]. When the performance problems were discovered, usually in the later stages of development, the common solution was to upgrade the hardware. This approach was expensive and if the software had to be redesigned before deployment the costs were higher still.

In distributed application systems it becomes more difficult to increase the performance of a system with new hardware. Since the software resources are now spread over multiple nodes in a network it is too expensive to simply upgrade all the hardware resources to improve system performance. Furthermore, there are limits to the communication speeds in networks that can not be overcome. Software Per-

formance Engineering (SPE) [13] is an approach that is used today to combat the problems that may arise following the “fix-it-later” ideology.

SPE defines a method that begins at the start of the development process, the *requirements analysis* phase. SPE can be used with any of the software development approaches mentioned previously. It does not stop at the requirements phase but continues through each phase of the lifecycle. This thesis focuses on two aspects of SPE, *model building* and making effective use of results from performance evaluation techniques. Quantitative models are built to describe a proposed system or alternate designs for a system. Performance evaluation techniques such as bounds analysis and mean value analysis are then used to predict performance measures for the models. The effective use of these results helps an analyst to understand the performance impact of alternative design decisions. The next sections discuss these phases in further detail.

1.3.1 Building Models

Smith [13] considers two types of models, the *software execution model* and the *system execution model*. Figure 1.9 on Page 18 shows the relationship between the two execution models. To perform an analysis of a system both the software and system execution models must be built. The software execution models represent the main features of the software execution behaviour and describe the paths the requests follow as they flow through the application. The information required to create the software execution models is obtained from the resource usage estimates, the workload scenario specification and the software design concepts. These concepts are defined as:

Workload scenarios : these are the dominant requests that are performed on the system. The performance characteristics of the system will be defined by the workload scenarios;

Performance Goals : the expected results from the system. The response time, throughput and device utilizations are among the performance measures stud-

Figure 1.9: Smith's Execution Models

ied;

Software design concepts : these are the execution design concepts of the software structure. They include sequential, loops, conditional and recursive execution structures that incur demands on the system resources;

Resource usage estimates : these are the demand estimates that are placed on the resources by the workload scenarios. These include the number of instructions to be executed, input/output operations etc; and

Execution environment : this is the hardware description that the software will be executed on.

The software execution graphs are used to describe the execution structures of the application. For each function or method in the application, a graph is created that is a detailed description of the software execution. The graphs include information about function calls, statements or blocks of code, ie. activities in the system. For

each request, an analyst gives estimates for CPU instruction counts, disk operations, loop counts and branching probabilities. This information is used to calculate the resource usage estimates made by the requests. The different types of requests that flow through the system are given in the workload scenario specification.

The last piece of information to go into the software execution models is the execution environment. The execution environment describes the hardware that the system will be using and the arrival rate or population of customers associated with each workload scenario. The description includes the service rates of the CPUs and the service times of the disks. Once the information has been gathered, the information contained in the software execution model is used to build the system execution model. The parameters of the system execution model include the visits to the system resources and the total demands placed on the resources. Smith [13] advocates the use of queueing network models as the system execution models. Performance evaluation techniques are then used to predict the performance of the modelled system. Analytic estimates include mean response times and queue lengths, throughputs for requests and resource utilizations. Smith also extends these models to include the complex behaviour of the software not well suited for analysis. In these cases, discrete event simulation is recommended for performance evaluation.

1.3.2 Bounds Analysis and Queueing Models

The performance evaluation of the model is the next step in analyzing the behaviour of the system. In this section we introduce two types of performance evaluation, bounds analysis and queueing analysis. Chapter 2 describes these techniques in more detail and how they can be applied in the performance study of a distributed application system.

Bounds analysis requires little computation and provides the upper and lower (asymptotic) bounds for request throughputs and response times. Performance evaluation techniques such as mean value analysis (MVA) and simulation can be used to solve queueing network models to provide the mean response times, mean queue lengths and throughputs of the computer system.

Layered queueing models (LQMs) are an extension of QNMs that take into account the use of software servers in the performance analysis of a distributed application system. [14, 15, 16] describe MVA techniques that provide performance estimates for LQMs. This allows an analyst to calculate mean client response times, and the utilization and average queue lengths at the software resources along with the hardware resources.

1.4 Existing Performance Tools

To conduct a performance study of a distributed application system, an analyst must first describe the system in terms of its resources, demands and requests. Next the models that are built must be solved using one or more of the techniques mentioned previously and then these results must be interpreted. To help the analyst with a performance study there are existing tools to facilitate the process of building the distributed system models and to perform the analytic or simulation study of the performance. Tools such as *SES WorkBench*, *Legent's APM*, *PARASOL* and *Timebench* allow a distributed application to be modelled and examined before it is actually implemented. These tools are described in the following subsections.

1.4.1 SES/Workbench

SES/Workbench [17, 18] developed by *Scientific and Engineering Software, Inc.* is a system level design and simulation toolset that allows a user to model hardware, software, and complete networks using a graphical user interface. SES/Workbench allows a user to analyze information about the flow of requests through the system and study the impact of various software-hardware tradeoffs. System hardware bottlenecks can be identified and system performance can be evaluated.

The hardware information in SES/Workbench identifies node information, such as the CPU service rate, disk service times, and the network connections between nodes. The software configuration describes the functions that are provided by the application being modelled. The functions or methods are created and distributed

across the nodes in the system. The resource usage of the functions are stated in terms of CPU instructions executed and number of disk operations performed. SES/Workbench also allows the user to integrate 'C' code into the software methods to simulate the execution behaviour of the application. The workload specifications define the types of requests in the system, the request's paths through the system and their arrival rates. After the model of the application has been created, the performance of the application can be estimated either analytically using Queuing Network Models or through simulation.

1.4.2 A Performance Modeller

A Performance Modeller (APM) developed by *Legent Corp.* is a modelling tool that uses both analytical and simulation methods to provide measurements for a distributed application. The tool was in beta testing during the writing of this thesis the information presented in this thesis was obtained through the use of the tool and in the limited on-line help that was available. At the current time there is no printed documentation about this application and in late 1995 Legent Corp. was bought out by *Computer Associates* and the fate of this tool is uncertain.

A distributed application is described in APM with the aid of a graphical user interface. The user defines a distributed application system by describing the workload specification, software configuration, and the hardware devices and connections. The hardware description is defined by the user by creating the nodes in the system and then creating the communication links between the various devices. APM has a library of many hardware resources which contain information such as the service rate of the processors in the nodes, the speed of the networks and the service times of the disk drives. If the libraries do not have the information required, the user may define his own devices for the system.

The software specification is done in two parts. Similar to Smith's approach, the first part is to define the processes that will be active in the system. The processes contain the execution statements that will be executed by the requests. The second

part is to create the workloads for the system. The workloads describe the path of the requests in the system.

To create the processes for the distributed application system, the user creates blocks of execution statements. These statements include the resource requirements of the methods and include items such as the number of CPU instructions to be executed and the number of disk operations. The statements may also include loops, conditions and other execution structures. Once the processes have been created, the user distributes them across the nodes in the system.

The second part is to define the workload of the system. The workload is comprised of the requests that will be placed on the system. To create the request path, the user states all processes that are to be used. This is accomplished by creating a graph for each request in the system. The nodes of this graph are the processes that are used by the request. Once the hardware description and workload specification has been entered into APM the user can choose between a mean value analysis or simulation method for calculating the performance measures. The measures include hardware resource utilization, system throughput and queue lengths. The software resource utilizations are not available through APM and it does not take into account contention for software resources on performance.

1.4.3 PARASOL

PARASOL [19] is a simulation tool that can be used to model many types of systems including distributed applications. The user is able to define both the description and behaviour of the hardware and software resources of the distributed application. The distributed application is defined as a network of heterogeneous nodes where the user defined processes execute. Both the hardware nodes and the software processes are created by the user who defines the interactions between them. The user also creates a specialized process called Genesis which is used to define the hardware nodes, the application processes and can also be used to control the simulation.

The hardware is created by using function calls within the Genesis task. The functions *build_node*, *build_bus* and *build_link* are all used to create the hardware

specification of the application to be tested. The nodes of the system are created first with the buses and links built after to connect the nodes. The parameters for the `build_node` function are:

- the node name;
- the number of CPUs;
- a speed factor used to scale execution times on the processors;
- a quantum variable for the round-robin scheduling time slice;
- the queueing discipline; and
- a `stat_flag` set to *TRUE* to collect processor utilization statistics.

The software definition of the distributed application involves the creation of the application processes. The processes are created by calling the *create* function from the Genesis process. To create a process, the user must write a C function for each of the application processes. These functions may contain the code to model the execution behaviour of the system. The function calls *compute*, *hold* and *sync* cause the process to consume CPU cycles to simulate their execution. Communication between the processes is handled by other function calls.

To model an application, the user must write the code within the Genesis process to create the hardware nodes and software processes. The PARASOL driver then uses the Genesis process to create the application model. This is accomplished by the PARASOL driver which builds the first node, *node 0*. The Genesis process is then activated on this node and the PARASOL driver enters an operational loop in which it executes genesis.

1.4.4 Timebench

Timebench [25, 16] is a computer aided design tool that allows a user to prototype real-time systems. Timebench allows a user to design a system through a graphical

user interface and to evaluate the performance of the system using approximate analytic methods. Once fully specified, the user can also generate code automatically for environments such as C/JNX and ADA on SUN stations.

Timebench views software components as blocks of program statements, modules and procedures and associates them with a form of operating system processes called *tasks*. The tasks are then associated with processors in a system. Timebench illustrates RPC in the system as arcs going from one entry in the system to an entry of a second task. The average number of visits one entry makes to another entry can be specified.

Timebench allows an analyst to study the performance of distributed systems by acting as an interface to solvers such as SRVN [28] and MOL [14, 15]. However, the analyst can not easily move entries from one task to another to study the performance implications of different configuration.

1.4.5 Tool Summary

All of the tools mentioned in the previous subsections can be used to model distributed applications. They allow a user to describe a distributed application in terms of its nodes and software processes that are allocated across them. SES/Workbench and Legent's APM do not take into account the software servers when calculating the performance metrics of the distributed systems.

The tools mentioned in this section could but currently do not support the engineering structures of the RM-ODP. The basic engineering objects provide the methods that would be executed by the requests. These would then be grouped into the clusters and capsules within the model and then allocated across the nodes of the system. This thesis describes a methodology that exploits these concepts that will help study the performance of a distributed application system early in the development phase. The Distributed Application System Sizer (DASS) tool was developed for this thesis to allow users to easily reconfigure a distributed system using the constructs of the RM-ODP. The tool is described in Chapter 3.

1.5 Object Based Performance Analysis

The work done in [13] provides performance analysis of the hardware and software system using queueing model analysis. The view taken of the software in creating the execution graphs is procedural. The approach taken in this thesis is to view the distributed system as a group of software objects that can be configured to meet the performance needs of the users. In this thesis this is what is meant by *Object Based Performance Analysis*. We chose to view the system as a group of software objects namely the basic engineering objects, clusters and capsule to allow an analyst to easily create a configuration of a distributed application system. We are trying to find any problems in the composition and distribution of the software components and this object oriented approach gives us the encapsulation necessary for our technique.

We assume that the objects used to describe a system are found using an object based analysis and/or design technique [4, 11]. We base the description of the software components on the RM-ODP as described in Section 1.2. Given these objects and their methods, our goal is to make good use of performance evaluation techniques to find a composition and distribution strategy for a distributed application system. We would like to find a strategy such that the distributed application system will meet its performance objectives over a full range of workload conditions.

To test a configuration over a full range of workloads would take time, so a technique is applied to find a small set of cases that represents the possible workloads. An asymptotic bounds technique [20, 21] is used to find which workloads will cause combinations of hardware resources to saturate. This technique has been developed into a software application called *Asymptotic Bounds Analysis* (ABA) tool by Giuseppe Serazzi at the Politecnico di Milano. The ABA tool takes as input a set of requests in a distributed system and the demands placed at the hardware resources associated with these requests. The output of the ABA tool is the workloads that cause the hardware resources to saturate. This technique and tool are further discussed in Section 2.2.2.

Since, the maximum throughput achievable in any system occurs when the hardware resources are saturated, these workloads become performance test cases used to study a configuration. Once we have determined the maximum throughput achievable in the system we next calculate the utilization levels of the software and hardware components. These utilization values are then compared to the utilization limits as set by the technology to verify if any limits are approached. Our goal is to integrate the ABA tool and the utilization analysis into a single tool to find a subset of configurations, hopefully one, that satisfies the utilization limits over a full range of workload conditions.

1.6 Thesis Outline

This thesis is divided into six chapters. The first chapter has described the main features of distributed environments and the problem of utilization limits in such environments. Several examples of distributed environments that are in use today were given to describe how the issues of composition and distribution are reflected in their architectures. Chapter 1 has also shown how the structures of these distributed applications can be modelled using the engineering structures of the RM-ODP. SPE is described as a technique to help study performance during software design. An outline of the need for Performance Engineering on distributed applications and several tools that are available to perform modelling studies was also given.

Chapter 2 describes performance evaluation techniques that can be used to estimate system performance metrics. The problems that can be addressed by the techniques and in which order they should be applied are also presented. The asymptotic bounds analysis technique is further described, and it is shown how the ABA tool can be used to calculate the maximum throughputs in a system to perform the utilization analysis described in Section 1.5.

Chapter 3 of this thesis describes the Distributed Application System Sizer (DASS). This is a tool developed to integrate the ABA tool and the utilization analysis to study the configuration and distribution of software processes across a

distributed system. DASS can be used to study the composition and distribution strategy of a distributed application system to recognize potential software bottlenecks. Different configurations can be found that reduce the likelihood of or avoid these bottlenecks.

Chapter 4 describes an example distributed application system of a simple retail store. The application's hardware and software components are described and a model is created in DASS. DASS is used in conjunction with a bounds analysis technique to find a configuration that satisfies the utilization limits of the resources.

Chapter 5 describes a case study of work done at SHL Systemhouse Ltd. of Ottawa on a project for the Canadian Department of National Defence. The case study shows how the DASS tool can be used in an existing system to study the design implications of a distributed application system. Conclusions and remarks are offered in Chapter 6.

Chapter 2

Performance Evaluation

Performance evaluation of a computer system is essential in understanding how the system will perform under load. Chapter 1 described the different aspects of distributed application systems and why we would want to study the performance of them. Two methods that are used to study performance are *analytic* and *simulation modelling*. Both analytical and simulation modelling are representations of computer systems that use simplifying mathematical assumptions to solve for performance measures of these systems. The most simple assumption is a service time homogeneity assumption that requires mean service times to be independent of the number and placement of customers in a model. A more severe set of restrictions produce product form (also called separable) models. Though these assumptions are often violated by real systems in practice, the accuracy of the results are usually sufficient for planning purposes [22].

Many analytic methods can be used to better understand the performance questions of a distributed application system. These analytic methods include Queueing Network Models (QNM), Petri Nets (PNs) and Layered Queueing Models (LQMs). Analytic modelling uses algebraic equations to solve the performance metrics of a computer system. Analytic models make assumptions about the workload arrival process and service time distributions to provide estimates for mean response times, resource utilizations and mean queue lengths. This thesis will deal with analytic modelling of distributed application systems. Petri Nets (PNs) are not considered

in this thesis because PNs for the systems we consider require a large number of tokens causing a state space that is too large to deal with.

Simulation modelling uses a program to model the sequence of events or requests of a computer system. The simulating program computes the performance metrics of the model by repeating the simulated execution of the requests that comprise the workload of the application system. Simulation models also make assumptions about the arrival and service time distributions but are better able to predict their behaviour. However, simulation modelling may be very time consuming so whenever possible analytic modelling should be used as a first approach in a performance study.

In this chapter we will look at different applications of analytic solutions for studying the performance of distributed application systems. First we will define the terminology that will be used in the remainder of this chapter. A model that considers customers that arrive and depart from the system is defined as an *open model*. A *closed model* is defined as a model where there is a fixed number of customers. In an extreme case when the number of customers in a closed model becomes very large, it can behave like an open model. The remainder of this thesis will deal only with closed models.

The models that will be considered in the remainder of this chapter are described with the following parameters. The resources in the system are the service centers of the system. The number of resources will be denoted by K and the set of resources is given by $K = \{1, 2, \dots, k\}$. The population of the system is given by N and the system throughput is denoted by X . X is usually given in customer completions per second for the system as a whole. The throughput at a single resource is denoted by $X_j, j \in K$. The customer population N is comprised of request classes denoted by C . The set of request classes is given by the set $C = \{1, 2, \dots, k\}$. Requests enter the queues at the service centers awaiting their time to be served. The demand placed by customers on a system resource $j \in K$ is denoted D_j and the largest demand at any single resource is denoted D_{max} . Demands are usually expressed in seconds.

Figure 2.1: Network of queues in a closed system

2.1 Queueing Network Models

QNM's are an approach to performance evaluation that represents a computer system as a *network of queues*. Figure 2.1 on Page 30 shows a simple network of queues, the service centres each have a queue, shown by the open rectangle, and a resource, shown by the circle. These models can be used throughout the life of a computer system to help during sizing studies and during the evolution of the system itself.

In this chapter, two separate performance evaluation techniques for QNM's are considered [22]. The first technique uses QNM's to calculate the upper and lower bounds on throughput, X , and response times, R . Knowing the upper (asymptotic) bounds on the X and R helps to discover whether it is feasible for the system to support its proposed workload.

The second approach is to solve the QNM's to estimate the mean response times of the given workloads. This approach requires more computation but can be used to determine if the application system will meet its response time goals under anticipated workload conditions. The following sections describe these performance evaluation techniques and how these techniques can be usefully applied in a performance study.

2.2 Bounds Analysis

Bounding analysis is the most simple approach to performance analysis using QNMs. Bounding analysis requires very little computation and few assumptions yet it stills provides the analyst with useful information on the performance of the system. The fact that the analysis can be done quickly and easily lends this technique to be useful as a first cut method in studying a distributed application system. This thesis will focus on one particular type of bounds analysis, namely *asymptotic bounds*.

The lower and upper (*asymptotic*) bounds on the throughput based on the demands at the hardware resources can be calculated using this analysis technique. Based on these asymptotic throughput values an analyst can determine if it is feasible for the system under test to meet its performance objectives. The next section shows how the bounds on the throughput can be calculated for a single class system. There are more sophisticated methods of using bounds to study systems with multiple request classes [20, 21, 22, 28].

2.2.1 Asymptotic Bounds

Consider a system that has a population of customers, N , all from a single request class. As the number of customers in the system increases, the utilization of the resources can increase. The *utilization law* states that the utilization of a device (resource), j , is the throughput at the device, X_j , multiplied by the service demand at the resource, D_j . It is given by:

$$U_j = X_j D_j \quad (2.1)$$

The utilization law requires the service time homogeneity assumption; the average demand per completion is assumed to stay the same regardless of the number of customers at the device.

As we increase the population in the system, the resource with the highest demand will saturate before the others and can be called the system bottleneck. The system bottleneck device will limit the overall system throughput and the demand

at this resource is denoted by D_{max} . The maximum system throughput, X_{max} , can be obtained by setting the utilization of the system bottleneck to 1 and rearranging Equation 2.1 and is given by:

$$X_{max} = \frac{1}{D_{max}} \quad (2.2)$$

At the other end of the scale, when the population N of the system is low we can also calculate the upper bound on the throughput. If the number of customers N is set to 1 there will be no bottleneck. In this case, the system throughput is based on the total time spent in the system. The total system time is the sum of the demands at all service centers and is given by $D_{total} = \sum_{i=1}^k D_i$ plus the think time, Z , of the user. Z is defined as the average time a customer in a closed model does not compete for resources. The system would attain a maximum throughput of $X_{max} = \frac{1}{D_{total} + Z}$, however, since we have N customers in the system, the overall maximum system throughput is:

$$X_{max} = \frac{N}{D_{total} + Z} \quad (2.3)$$

Combining the Equations 2.2 and 2.3 we can get an upper bound on the system throughput over the full population that is bounded by:

$$X_{max} = \min \left\{ \begin{array}{l} \frac{N}{D_{total} + Z} \\ \frac{1}{D_{max}} \end{array} \right. \quad (2.4)$$

Equation 2.4 shows that the upper bound of the system throughput has two parts. The first part for when the system is under light load, and the second for heavy loads. It should be noted that the second part of Equation 2.4 is more significant as it calculates the maximum system throughput.

To demonstrate the usefulness of bounds analysis, a simple queueing network model with a single request class is considered. Figure 2.2 on Page 33 depicts a QNM that has two servers, a CPU and a Disk, with service demands of 3.8 and 5.2 respectively. Bounds analysis can give a quick estimate of which improvement will result in the greatest increase in throughput, upgrading to a CPU that is twice as fast or upgrading to a faster Disk. Table 2.1 on Page 33 shows the service demands of the CPU and Disk for the original QNM and the two upgrade choices.

Figure 2.2: Queueing network model of bounding analysis example

system	Service demands		Upper
	CPU	Disk	Bound on X
Original	3.8	5.2	0.19
CPU upgrade	1.9	5.2	0.19
Disk upgrade	3.8	2.6	0.26

Table 2.1: Service demands for QNM

Figure 2.3: Asymptotic throughput bounds

The D_{max} of the three scenarios are 5.2, 5.2 and 3.8. The sum of the demands, D_{total} , are 9.0, 7.1 and 6.4. Using Equation 2.4 we can calculate the asymptotic bounds of the system throughput for the three scenarios which are shown in Figure 2.3 on Page 34. It is shown that both upgrades will give throughput improvements at low populations but only the upgrade to the Disk will improve the throughput at heavy loads. From this quick performance study using bounds analysis, it can be concluded that upgrading the disk is the better option.

The previous example has shown how bounds analysis can be used quickly and effectively in determining what resource in a system should be upgraded. The decision is based on what upgrade will give the most improved performance. The previous bounds analysis example assumes that we have a single customer request class populating the system to calculate the asymptotic throughput. If we introduced multiple customer classes we would have to calculate the asymptotic throughputs for each of the different customer classes and for different combinations of the customer classes.

This thesis will focus on a technique that provides the asymptotic bounds of multiclass closed systems. Balbo et al. [21, 20] discuss a technique to determine which combination of customer request classes will cause hardware bottlenecks in

a closed system. This technique assumes that the model satisfies the product form assumption [22]. The next section explains this technique in detail and how it can be used to calculate the asymptotic utilization levels for the software resources as described in Section 1.5.

The following notational changes should be noted in order to accommodate the multiple request classes that are now being considered. The demands at the system hardware resources are now given by request class. These demands are given by the matrix of demands, $D_{ij}, i \in C, j \in K$, which gives the demand at resource j by request class i . The throughputs are now denoted by $X_{ij}, i \in C, j \in K$ which gives the throughput of request class i at resource j .

2.2.2 Maximum Software Utilizations Using Bounds Analysis

In this section we consider how to choose a set of throughputs based on the request classes in a distributed system to best accomplish the utilization analysis. A first approach to select a set of throughputs is to simply choose the highest throughput of each request class. This is done by taking the highest resource demand used by each request class and using Equation 2.2 to calculate the maximum throughput of each request class. This approach will find the maximum throughputs of the single request classes and the utilization of the software components can then be calculated with these values. However, we can not be certain that these will correspond to the highest utilization levels as we can not be certain that a combination of request classes will produce a greater system throughput than any single request class.

An extension of the first approach is to test combinations of request classes at random to try and find a maximum system throughput to calculate the highest utilization levels of the software components. The combination of request classes that are needed are the ones where combinations of hardware resources are saturated. Once the hardware resources become saturated we can not achieve a greater system throughput by increasing the customer population. It is at these combinations

that the maximum system throughputs are produced. To determine by hand which combinations of resources may become a system bottleneck, a performance analyst would need to go through all possible request class ratios and determine the system bottleneck resource for each one. This would be very time consuming. A technique is described in [20, 21] where the request class ratios that cause the resources to saturate are calculated all at the same time.

The technique described in [20, 21] takes as input the resource demands of each request class at each resource, D_{ij} , and creates a series of systems of linear equations. This series of equations expresses the bounding relationships between the demands of each request class at the resources and the utilization of the resources. By setting the utilizations of selected resources to one, the solution of each specific system of linear equations gives the mix of customers from each request class that cause that combination of resources to saturate.

Giuseppe Serazzi has integrated this technique into the Asymptotic Bounds Analysis (ABA) tool that allows a user to specify the demand matrix of a distributed system. The ABA tool then calculates the request class mixes that saturate each combination of hardware resources. We can then use these request class mixes as our workload test cases to direct a utilization analysis as described in Section 1.5.

Consider a multiclass closed system that has a set of request classes, $C = \{1, 2, \dots, c\}$, and a set of system resources, $K = \{1, 2, \dots, k\}$. As the total number of customers N is increased, we consider the mix of customers that belong to each request class in C . The mixes of the customer population from each of the request classes is denoted by the vector $\vec{\beta} = \{\beta_1, \beta_2, \dots, \beta_c\}$. The components of $\vec{\beta}$ are derived from the population, N , and are calculated by:

$$\beta_i = \frac{N_i}{N}, i \in C \quad (2.5)$$

Where N_i is the number of customers of request class i . Each of the $\vec{\beta}$ denotes a combination of request classes in the system. The sum of the components in each $\vec{\beta}$ is 1, $\sum_{i=1}^c \beta_i = 1$

The ratio of classes that populate the system determine which hardware resources become bottlenecks as N becomes large. Different ratios of request classes may cause

Figure 2.4: Possible combinations of request classes

different combinations of resources to become the system bottleneck. Figure 2.4 on Page 37 illustrates all possible request class mixes for a system with two types of requests and two resources. For a fixed number of requests in the system, all the possible request mixes are described by the line of equation $\beta_1 + \beta_2 = 1$.

By considering the saturation of each combination of devices, this analysis technique finds the set of vectors $\vec{\beta}$ that are the crossover points of the device bottlenecks. It should be noted that the global throughput changes monotonically, either increasing or decreasing, between two consecutive crossover points if the request type mix is moving in a single device saturation sector. If the mix is in a common saturation sector, where more than one device saturates concurrently, the global throughput remains constant. In all cases, the individual class throughput changes monotonically within a saturation sector. Thus, the extremes for the throughputs of each request class must occur at these crossover points.

These crossover points define the request class mixes that we are interested in. Figure 2.5 on Page 38 illustrates possible crossover points for a system with two types of requests and two resources. In calculating the bottleneck devices of a system, the

Figure 2.5: Bottleneck crossover points for a two request type/two device system

ABA tool also calculates a bottleneck strength for a device which is denoted by γ . In a single saturation sector the bottleneck strength of the saturated device is 1 and all others are 0. This can be seen in Figure 2.4 on Page 37. In the two single saturation sectors we have $\gamma_j = 1, \gamma_l = 0$ and $\gamma_l = 1, \gamma_j = 0$. In the case of the saturation sector of two devices we have $\gamma_j + \gamma_l = 1$. This can be extended for a saturation sector of k devices such that the sum of the bottleneck strengths is equal to one, $\sum_{j=1}^k \gamma_j = 1$. However, at the crossover points, the first case occurs where a single γ is 1. This signifies that we only have to deal with a single resource at the crossover points to calculate the request throughputs.

Using this technique an analyst no longer needs to use ad hoc techniques to test every possible population mix to find the throughputs of the requests that cause combinations of resources to saturate. The outputs of this technique identifies the combinations of resources that can saturate as well as the $\vec{\beta}$ for each crossover point.

The asymptotic throughputs of the request classes at these crossover points can now be calculated. The utilization of a saturated resource U_j is set to 1. Since, we know the demand of the resource attributed by each of the request classes, the asymptotic throughputs of each request class at the bottleneck device, X_{ij} , at the crossover points can be calculated by rearranging Equation 2.1 on Page 31 and is

given by:

$$X_{ij} = \beta_i \left(\frac{1}{D_{ij}} \right), i \in C, j \in K \quad (2.6)$$

where D_{ij} is the demand on resource j attributed to request class i . The D_{ij} is chosen for the resource where the bottleneck strength for that resource is 1, $\gamma_j = 1$. In a multiple saturation sector it is the case that at the crossover point one of the bottleneck strengths is 1 and the others are 0. At each crossover point the $\vec{\beta}$ is used to calculate the portion of the throughput attributed by each of the request classes.

This technique provides the maximum request class throughputs over the population mixes where each combination of hardware resources saturate. The utilizations of the software components can then be calculated based on these request throughputs. Since the maximum throughput occurs at one of these crossover points, the maximum utilization of the software components also occur at some of the points. Using these crossover points to calculate the request throughputs, combinations of request types and individual request types have been considered. By considering the individual and combinations of the request types, it allows us to identify the maximum software utilization levels over all workload conditions.

This technique provides an early method of discarding a system's configuration if the utilization limits, as defined in Section 1.2.1, of the software components is approached or exceeded. This is an efficient method to direct a performance study for a distributed application. The process of integrating the ABA tool into a utilization analysis is further discussed in Chapter 3 and is illustrated in Chapter 4.

2.3 Mean Value Analysis

Asymptotic bounds analysis can be accomplished with very little computation and with few parameters. A drawback of using bounds is the values that are calculated using assumptions that are either optimistic or pessimistic. As a result, the values obtained from the analysis can be loose. In order to determine tighter values of the performance metrics, other analytic techniques such as Mean Value Analysis (MVA) [22] may be used. Like ABA, MVA also requires that the model satisfies

product form assumptions. This thesis does not deal directly with MVA but will give an overview of the technique to show an alternate way of performance evaluation for closed queueing networks.

The basis for MVA is *Little's Law* for systems which is given by:

$$N = XR \quad (2.7)$$

where N is the average customer population in the system, X is the system throughput and R is the average residence time in the system. Equation 2.7 can be applied to a resource, its queue and the average queue length at the resource and is given by:

$$Q_j = X_j R_j, \quad j \in K \quad (2.8)$$

where Q_j is the number of customers in the queue at resource j , X_j is the throughput at resource j and R_j is the residence time at the resource.

For closed models, the exact solution of MVA [23] enumerates all possible combinations of customers at a queue. To solve for a single class closed model with N customers requires that the QNM be solved for $1, 2 \dots N$ customers. Also, within each enumeration the equations must be solved over all resources K . Therefore, the cost of the exact MVA solution increases as the number of customers and resources increase. When there is more than one customer class in the system the solution complexity of the exact MVA technique grows exponentially. This shortcoming in the exact MVA may be overcome by using an *Approximate MVA* (AMVA) technique. The AMVA technique reduces the computation time needed by only approximating the residence time near full customer populations [22].

The use of the MVA and AMVA techniques can provide mean performance metrics for the analyst. The output parameters include system throughput and resource utilization, mean response times, average number of customers in the system and average resource queue length. The computation time needed to solve for either the MVA or AMVA techniques is considerably longer than that of the bounds analysis. For this reason and the fact the measures provided are more accurate, these techniques would be used later in the performance study once the number of test cases

has been reduced.

The above techniques are for closed queueing network models and only take into account the hardware resources as servers. There is an extension of the QNMs described above that take into account the queueing delays at the software processes as well.

2.3.1 Layered Queueing Models

Regular QNMs model a computer system as a network of servers and queues. Typically these servers are the hardware resources within the computer system that handle requests from the software. There is a set of models that also take into account the fact that software processes can handle requests from other processes. In other words, the software processes can themselves become servers. The queueing delays that occur at the software servers must also be taken into account when studying the performance of a distributed application system. These extended queueing models are called *Layered Queueing Models* (LQMs) [14, 15, 16].

These LQMs take as input parameters those of the standard queueing network models as well as information specific to the software capsules. This information includes the number of visits a capsule makes to other capsules as well as the service demands of the software servers. The visits made by the capsules can be obtained from a trace of an existing system [26, 27] or by analysing the performance behaviour of the individual capsules [13].

The resource demands made by a capsule can also be estimated by analyzing the behaviour of the capsule. In [13], it is shown how the break down of the software process along with branching probabilities of the software code can help predict the execution of the process statements. This can help in predicting the service demands for the individual software capsules. An alternative approach is to use system measuring tools, such as profilers or monitors like *Legents TMON*, on an existing system to gather the information that is needed.

Once all of the information needed to build the LQM has been gathered they can be solved using similar techniques as for standard QNMs. MVA techniques that

take into account the software processes as server can be used to solve the LQMs [14, 15, 16]. They provide as output the same performance metrics as standard QNMs with the addition of throughput and utilization of the software servers as well as the average response times and queue lengths for these servers.

2.4 Performance Evaluation Methodology

This section describes a reasonable sequence of steps for the performance evaluation of a distributed application system. This methodology uses the different analysis techniques that have been discussed. Figure 2.6 on Page 43 illustrate the proposed sequence of events. The first level of evaluation would be a *utilization analysis* of the proposed system for some user defined workloads and for each request class separately.

If the utilization limits are not exceeded the asymptotic ABA tool can be used to find combinations of request class throughputs that may give higher device and software component utilizations. The use of the asymptotic bounds analysis technique allows the analyst to study the utilization of the software components over all workload conditions. Individual request classes are considered as well as the combinations of these request classes. This gives the analyst some confidence that the software components in the distributed application system will not become system bottlenecks. This approach considers the utilization of the software resources on the assumption that there are no queueing delays at hardware or software resources. If the utilization levels of the resources approach or exceed their limits then there exists a software bottleneck in the system. The system then needs to be reconfigured and the utilization analysis has to be redone.

Once it has been determined that the utilization limits have not been exceeded and a configuration or set of configurations have been selected, the study can proceed to more computationally intensive techniques. These next analysis techniques involves using a Layered Queueing Model solver. This would provide mean performance estimates of the distributed application system that take into account soft-

Figure 2.6: Performance Evaluation Methodology

ware and hardware queueing delays. Simulation can then be used to provide more confidence in the prediction of the distributed application system's performance. The final phase would be to produce a prototype of the distributed application system before proceeding to the final implementation.

This thesis deals with the first aspect of this methodology, the screening of the initial configurations using an efficient utilization analysis. Reducing the number of configurations that are passed onto the later stages in the design phase can greatly reduce the time and cost spent in designing a distributed application system. The remainder of this thesis will deal with describing this early phase. The next chapter will describe the Distributed Application System Sizer tool. The DASS tool integrates the asymptotic bounds analysis technique through the use of the ABA tool to perform a utilization analysis on the software components of a distributed application system. Chapter 4 illustrates this through the use of an example application and Chapter 5 describes a case study where this was used on an application in development.

Chapter 3

Distributed Application System Sizer

In general, the purpose of performance engineering is to ensure that the hardware provided with a system has sufficient capacity to meet the needs of its users and that loads are acceptably balanced across nodes. When looking at the loads on the hardware resources it must be noted that these loads directly affect the loads of the software resources that uses them. Therefore, in order to evaluate the complete system we also consider the impact of load on the software resources. If the utilization of a software resource approaches its limit as specified in the system's software technology constraints, software queueing delays may arise and limit throughput.

The information needed by the analyst to conduct a performance study of a distributed system must be stored in a way that is easily accessible. The information must be easily read and modified to study different configurations of the system. In this chapter the Distributed Application System Sizer (DASS) is discussed. DASS is a tool that allows the analyst to build a model of a distributed application and to analyse the performance implications of alternate configurations. The information stored about the model is structured according to the engineering structures of RM-ODP and is general enough to permit the modelling of various distributed environments.

Our approach to performance evaluation of a distributed system is to build the distributed system using the objects, clusters and capsules and find a composition/distribution strategy for the system. We then study the impact of workload

scenarios on the utilizations of the components. If there are software components that may become bottlenecks a new composition/distribution strategy is found and the analysis is redone. Our purpose is to find a configuration strategy that avoids bottlenecks for a wide range of conditions.

3.1 DASS as a Performance Tool

As part of this thesis, we have developed DASS to help an analyst perform a utilization study of a distributed application system. The idea is to find a configuration that avoids software bottlenecks across the components in the distributed application. Once a configuration has been found the analyst may continue on to perform more detailed studies using the strategy outlined in Section 2.4. DASS provides the analyst with the ability to:

- manage different configurations (models) of a distributed application;
- compute demands of the various hardware and software resources in the distributed application system;
- automate the interaction with the ABA tool; and
- view the computed data graphically.

Our approach to the performance evaluation of a system is to accept descriptions of those application requests expected to affect performance most [13] and to aggregate their resource consumption by software component. The requests are chosen by the performance analyst and are based on knowledge of the system.

As a request travels through the system, it makes visits to methods within objects and these methods make visits to other methods and demands on the system's resources. The number of times one method visits another is defined as the *visit ratio* for the request. Figure 3.1 on Page 47 illustrates the flow of a request through the system. When the request starts the first method that is invoked is Method 1. Method 1 executes instructions on the CPU and then makes a remote procedure call

Figure 3.1: Path of a request through the system

to Method 2. After Method 2 uses the hardware resources the remote procedure call ends and control is passed back to Method 1. The next procedure call is to Method 3 which uses its hardware resources and returns control to Method 1. Method 1 then finishes and the servicing of the request ends.

The basis of our performance approach are the requests that are executed on the distributed application system. To conduct a performance analysis of the distributed application system we require as input the following visit ratios and service demands on a *per visit basis*.

- The number of visits each object's methods makes to other objects' methods. Only the methods that are used by the request are counted.
- The hardware resource demands made by each of these methods:
 1. The number of CPU operations; and
 2. The number of disk operations.

These demands are later aggregated by request to calculate the resource demands on a *per request class* basis. The hardware resource demands per request class will be the information that DASS passes to the ABA tool to obtain the workload test cases. The software resource demands per request class will be used to calculate the

software utilization levels. The calculation of the software resource demands will be discussed further in Section 3.4.

We also need *hardware technology parameters* and *software technology constraints* to calculate the resource demands of the various requests. The hardware technology parameters are the service rates and times of the hardware nodes. The software technology constraints are defined as the limit of concurrent access to the software components or the utilization limit defined in Chapter 1. These include:

- CPU service rate, *hardware parameter*;
- Average disk service times, *hardware parameter*; and
- Utilization limits for the software resources (objects, clusters and capsules), *software technology constraints*.

To conduct a performance study of a distributed system we also need the *workload scenarios* for the distributed application system. In this thesis we define a *workload scenario* as a set of request types and their given throughputs. Finding appropriate throughputs to use in performance study is an important step in the study of a system. This thesis provides a method to accomplish this which is demonstrated in Chapter 4. In this thesis we consider closed models since the asymptotic bounds analysis technique is used to find these throughputs.

Consider a set of request classes $C = (1, 2, \dots, c)$ and a set of resources $K = (1, 2, \dots, k)$. The throughput X_i of request type i in *requests/second* and its demand $D_{i,j}$ at resource j in *seconds/request* can be used to calculate its utilization of resource j and is given by the utilization law:

$$U_{i,j} = X_i D_{i,j} \quad , i \in C, j \in K. \quad (3.1)$$

For a workload scenario, the utilization law provides hardware and software utilization levels for the distributed system. If we know what workload scenarios produce the maximum throughputs in a distributed application system, the maximum utilizations can be calculated using Equation 3.1. Software resources that approach

Figure 3.2: Flow of control of a request

or exceed their maximum utilization level specified in the software technology constraints can therefore be identified.

It should be noted that in doing the calculation for software resources utilizations, an object's utilization includes its utilization of other objects. This is because a thread of control must be maintained by the calling object while the work is performed by the object it called. Figure 3.2 illustrates a flow of control between objects in a distributed application system. We assume that Object 1 is waiting while its requests are being processed by Object 3. Therefore, the resource demands of Object 3 affect the utilization of Object 1. This is defined as the included service of the software components. The service of the server objects is included in the service of the client objects.

The demands are aggregated per request for all software components when calculating their utilizations. It should be noted that these aggregated software demands are optimistic in that they do not take into account the queueing delays at the nested software servers. When Object 1 makes a call to Object 3 we assume that there are no software queueing delays incurred. This is often not the case but note that in well managed systems the ratio of queueing time to service time rarely exceeds one. This knowledge can be used to better estimate the utilizations but the issue is not addressed further in this thesis.

Figure 3.3: Communications between object's methods

The communication costs of the visits between each object's methods are also taken into account when calculating the demands of the software components. There are different types of communications that can occur between objects in a distributed application system. An object's method may make a procedure call to another method within the same capsule (*intracapsule*) or it may call another method in a different capsule. The second type of call may be broken down even further by distinguishing between calls between capsules on the same node (*intranode*) or on different nodes (*internode*). Figure 3.3 depicts the different communication types. The allocation of the components and the number of visits between methods affect the communication costs of the distributed system. If methods that communicate frequently are placed in different capsules or on different nodes, the costs for the communications between these methods will be different. These varying costs must be taken into account when modelling a distributed system. DASS keeps estimates for these communication costs so they can be computed automatically with each change in configuration. The current version of DASS allocates a fixed communication cost to both the client and server methods when a procedure call is made.

Our task for a given hardware configuration and allocation of software components over the hardware is to test whether the software technology constraints are

Figure 3.4: Data Structures of DASS

satisfied over a full range of workload scenarios, i.e., with the different workload mixes and intensities. If not, we must re-configure the software components until the constraints are satisfied.

To complete the utilization analysis performance study using the method described in Section 2.2.2 DASS must be able to store and manage the information to describe multiple configurations of a distributed application system. It must also interact with the ABA [20, 21] tool to obtain the $\vec{\beta}$ in order to calculate the asymptotic throughputs of the request classes as described in Section 2.2.2. Sections 3.2 and 3.3 will describe the information needed by DASS and how this information is entered into DASS. Section 3.4 describes the interactions between DASS and ABA.

3.2 Engineering Structures and Model Parameters

In Section 1.2 the engineering structures of the Reference Model of Open Distributed Computing (RM-ODP) were described. It was shown through examples how these structures can be used to model several distributed environments. These engineering structures are used within DASS to describe a distributed application system. The information is stored in database tables and links are maintained between the components to support the engineering structures of the RM-ODP. Figure 3.4 shows the relationships between the software and hardware components of DASS. A dis-

tributed environment is described in these terms and the corresponding information is stored within DASS.

The abstraction of the software components from capsules clusters, objects and the objects' methods facilitates easy restructuring of a model within DASS to evaluate alternate configurations. For example, linking objects to clusters gives the ability of moving objects from one cluster to another by redefining the link. If we want to move an entire set of objects that are contained in a single cluster, we only need to move the cluster from one capsule to another. We do not have to move each object individually. This holds true for each of the software and hardware components within the model. The engineering structures of the RM-ODP were chosen for this reason.

The information detailed by the RM-ODP can be broken into two classifications, the hardware (node) and software parameters. The hardware information needed is at the node level which includes the hardware performance measures as well as the technology parameters for each node. The software information includes the descriptions of all the components from the RM-ODP. The software information is augmented by acquiring data on the objects' methods.

Apart from the hardware and software component parameters, the workload definition is also required to perform an analysis of a distributed system. This information is needed to calculate the demands placed upon the system resources as the system is used. The node, software and request information will be discussed in detail in Sections 3.2.1, 3.2.2 and 3.2.3 respectively.

3.2.1 Node Parameters

The first level of information in the description of the application is the hardware that supports the distributed system. As work is done by the requests in the system, demands are placed on the hardware resources. When a method is executed, it does work on the CPU of the node it executes on and it may perform some disk operations. This work is characterized by the number of instructions executed by the CPU or the number of disk reads and writes that are performed by the method. These numbers

must be provided by the performance analyst. Both the CPU service rates and disk service times are stored by DASS in the *node* database. Each node is defined by the node name, node type and the service parameters of its resources.

To calculate the demands placed on the CPU by the requests we need to know the service rate of the CPU. This is chosen as $\frac{1}{IPS}$ where *IPS* is the CPU rating in number of Instructions Per Second. The demand on the CPU is calculated as follows where *Inst* is the total number of instructions executed:

$$D = \frac{Inst}{IPS} \quad (3.2)$$

To classify the demand on all the CPUs in the system by request type, we can extend the equation to:

$$D_{ij} = \frac{Inst_{ij}}{IPS_j}, i \in C, j \in K \quad (3.3)$$

where *Inst_{ij}* is the total number of instructions executed on the CPU of node *j* by request type *i*, and *IPS_j* is the rating of the CPU on node *j*. We note that the number of instructions executed by the method may not be the same on different types of CPUs. If the analysis is to be done to compare different CPUs, then we need to estimate the number of instructions executed on each of the two CPUs.

The demand placed on the disks is also needed for the performance modelling. If a node has more than one disk drive, several optimistic assumptions are made:

Assumption 1 : the read/write operations are balanced across disks; and

Assumption 2 : all disks on a node have the same average service time per read/write operation.

To calculate the disk demand we again extend the Equation 3.2 to encompass demands on all disks. To accommodate the assumptions the total disk service time is averaged over all disks on the node. This is accomplished by taking the service time for each disk and dividing it by the number of disks:

$$D_{ij} = \frac{V_{ij}S_j}{Disk_j}, i \in C, j \in K \quad (3.4)$$

where V_{ij} is the total number of disk operations made on node j by request type i , $Disk_j$ is the number of disks on node j , and S_j is the average disk service time on node j .

The last piece of information needed for the hardware components are the communication costs. The communication of a call from one method to another method requires a certain number of CPU instructions. The number of instructions is determined by several factors. The first factor is the type of communication. Communication by RPC requires a certain number of instructions to execute, whereas a communication using shared memory requires a different amount. A second factor in the number of instructions needed for the communication is the locality of the client and server objects. Communications within a capsule costs less than communications between capsules on the same node, communications between capsules on different nodes may cost even more. Figure 3.3 on Page 50 illustrated the different localities of the communications between objects. The difference in the communication costs may be in the order of tens of thousands of CPU instructions.

The communication costs are added on to the CPU demands of the respective methods, both the calling and called method. DASS automatically adds these costs to make the task of describing a configuration more straightforward. If the configuration is modified, DASS will recalculate the communication costs for the model. The current implementation of DASS adds a fixed overhead for communications between object's methods. Table 3.1 on Page 55 lists the costs of the communications that are added to the service demands of the methods. This overhead is determined by the relative location of the communicating methods, within the same capsule or in different capsules. Future work in this respect is to create a database of the communication costs for various mechanisms [30]. DASS would read this database and use the costs associated with the mechanism used in the model.

3.2.2 Defining the Software Structures

In this section we give a brief overview of the information needed to describe a software model within DASS. Section 3.3 describes in detail how this information

Communication Type	Communication Cost
InterNode	25000
IntraNode	1000
IntraCapsule	0

Table 3.1: Communication costs

is entered into DASS. The software components that are defined within DASS are the *capsules*, *clusters*, *objects* and *methods*. These software components must be defined within DASS in this order. This is due to the relationship between the software components in the engineering structures of the RM-ODP. A cluster is placed within a capsule, therefore the capsule must exist before the cluster can be placed into it.

These software components are the engineering structures of the RM-ODP and are defined in Section 1.2. The operating system processes or capsules are defined and distributed across the nodes in the system. This function is the distribution of a distributed application system. Next the clusters are defined and are allocated to the capsules. The objects are then defined and placed within the clusters. These two functions comprise the composition of the software components within the distributed system. Each of these software components are defined by their name and the component they are placed in.

Once the objects have been entered into DASS their methods can then be defined. There is more information needed to define the methods in the distributed application system than simply the method name and the object to which it belongs. The methods in a distributed application system are the software components that perform the actual work for the requests. The average number of CPU instructions executed and the average number of disk operations performed with each visit to each method must be specified for the request types. The number of instructions executed determines the demand of the request using Equation 3.3, D_{ij} . Similarly, the number of disk operations are the visits to the disk, V_{ij} , used in Equation 3.4. These measurements can be obtained from the application specifications or from

measurement traces of existing systems or prototypes.

Once the methods have been defined, they can be composed into the objects. Figure 3.4 on Page 51 depicts the relationships between these structures and the information defining each of them. The current version of the DASS tool assumes that for each invocation of the methods results in the same number of instructions executed on the CPU and the same number of disk operations to be issued. These values would be dependent on the size and parameters passed to the method and by the type of request invoking the method. These issues can be looked at in future research.

Lastly, a utilization limit as defined in Section 1.2.1 is needed for each software component. This value is the number of concurrent users permitted by the software component. If a software component has no limit on the number of concurrent users then the value can be set to zero, otherwise the limit is set to a value l . This limit will be used in the performance analysis to find a subset of configurations where no software component approaches its utilization limit.

3.2.3 Workload Definition

As a request flows through the system, the object's methods place demands on the hardware resources. The workload on a system is determined by the mix of the different request classes. As the ratios of the different requests are varied, the total demands in the distributed application system change. To conduct a performance study using DASS, the different request classes and workload scenarios must be defined. The request classes are defined within DASS and the ABA tool is used to identify the workload scenarios that will be used to calculate the software utilization levels. This relationship between DASS and ABA is described further in Section 3.4.

The request information needed by DASS includes the objects that are visited by each request type and the number of times each object is visited. This information identifies the flow of the request through the system and is used to calculate the demand on the system's hardware and software resources by request type. The request flow data is stored in DASS in tabular format by describing the client and

Request Name	Client		Server		Number of Visits
	Object	Method	Object	Method	
Request 1	Object 1	Method 1	Object 3	Method 3	1
	Object 3	Method 3	Object 4	Method 4	5
	Object 4	Method 4	Object 5	Method 5	2

Table 3.2: Example of request paths

server object/method pair. Table 3.2 depicts the data for the request path as shown in Figure 3.2. The request in this example starts with Object 1 Method 1 which calls Object 3 Method 3 once. Object 3 Method 3 visits Object 4 Method 4 a total of five times. Each time Object 4 Method 4 is called, it visits Object 5 Method 5 twice. Storing the data in the fashion allows DASS to keep track of all request types for an application.

To calculate the utilization of the resources by request type, all that is needed is the throughput of each request and the demands on the resources by request type. The demands on the system resources are calculated by DASS using the Method and Request tables. By following the flow of each request through the system, DASS calculates the demand on each resource that the request uses. The estimated request throughput can be entered by the user to calculate the resource utilizations.

These estimates may reflect the expected use of the system and are important. However, the purpose of this thesis is to find a subset of configurations that avoid software bottlenecks over all workload conditions. This was discussed on Page 35 in Chapter 2 and is illustrated by an example in Chapter 4.

3.3 Data Input

To describe a configuration within DASS requires information on the hardware and software resources, and the workload scenarios. The current method of entering this data into DASS is a form based textual input. Each of the parameters has a forms entry system where the user can define the hardware and software components, re-

quest types of the distributed application system and the visits between the objects' methods. Each of the input screens are now explained in further detail.

The first screen used to define a model is the main menu of DASS. It is here that a user may *create, copy, edit* or *delete* a model. The choices to run the *analysis* or view the results are also made here. Figure 3.5 on Page 59 shows the main menu with the Model menu item selected. However, the very first step in performing a study on a distributed application system is to create a model for it within DASS. This is done by selecting the *Add model* from the main menu. DASS will pop up a dialog box asking the user to enter the *name* of the model to be created. Once a model has been created, the user may go ahead and define the configuration of the distributed application system. To edit an existing model the user would simply select *Edit Model* from either the Model menu item or Edit menu item from the main menu. The *add model* and *edit model* functions will be described in Section 3.3.1 and 3.3.2. The *copy model* function simply copies all information that has been entered for an existing model over to another model. The *delete model* function simply deletes all information that has been entered for a specific model.

3.3.1 Add a New Model

To add a model the user must create the node and software components in a specific order. After the user has chosen to add a model and has entered the model name, a screen showing the components to add and the order that they should be created in is displayed. This screen is illustrated in Figure 3.6 on Page 60. The node, software and request information must be entered in the order as shown in the screen from left to right and top to bottom. From this screen the user can create the nodes, capsules, clusters, objects and methods that comprise the distributed application system. The flow of the requests are also entered here by using the *request* and *method call* screens.

The forms to enter the node information is shown in Figure 3.7 on Page 61. The user enters the *node name*, *MIPS* rating, *node type*, terminal IO service time (*TIO Svc Time*), disk service time (*DIO Svc Time*), whether or not the terminal IO is

Figure 3.5: Main menu screen

Figure 3.6: New model screen

across a network and the *number of disks* on the node. The terminal IO time is the time it takes to complete screen reads or writes to a terminal. This is meant to be used with distributed systems that may be using dumb terminals and the screen information is read or written to it by a remote host. Once this information has been typed onto the form, the user presses the *Save* button and the data is entered into the node table. If more nodes are to be added to the system the user can press the *New* button to enter another node. If there are no more nodes to be entered the user presses the *Done* button to return the *Add Model* screen.

The user continues to enter the required information to complete the model of the distributed application system. The *Capsule Data*, *Cluster Data* and *Object Data* are very similar in that the only information required for these components are the name of the component and the component to which it will be distributed to or composed in. The functionality of the forms is the same as the *Node Data* form. These forms are illustrated in Figures 3.8, 3.9, and 3.10 on Pages 61, 62, and 62 respectively.

The screenshot shows a Windows-style dialog box titled "Add New Model Information". The "Node Information" tab is selected, indicated by a thick border. The dialog contains the following fields and values:

- Model Name:** Greg
- Node Name:** [Redacted]
- MIPS:** [Redacted]
- Node Type:** [Redacted]
- TIO Svc Time:** 10 msec
- DIO Svc Time:** 10 msec
- TIO Uses Network:** No
- Number Of Disks:** 1

On the right side of the dialog, there is a vertical stack of three buttons: "OK", "Cancel", and "Help".

Figure 3.7: Add a node screen

The screenshot shows the same "Add New Model Information" dialog box, but with the "Capsule Information" tab selected. The fields and values are:

- Model Name:** Greg
- Node Name:** [Redacted]
- Capsule Name:** [Redacted]

The "OK", "Cancel", and "Help" buttons are still present on the right side.

Figure 3.8: Add a capsule screen

Figure 3.9: Add a cluster screen

Figure 3.10: Add an object screen

Method Information

Model Name: Greg

Method Name: Object Name

CPU Instructions: 0

DIOs: 0 Bytes/DIO: 0

TIOs: 0 Bytes/TIO: 0

Description:

Figure 3.11: Add a method screen

To finish the new model definition the user must enter the method information and the request information. The method information is entered using the *Method Data* screen as illustrated in Figure 3.11 on Page 63. The user enters the *Method Name*, the *Object Name* to which it belongs, the number of *CPU Instructions* executed, the number of disk operation (*DIOs*) and terminal reads/writes (*TIOs* performed and the number of bytes per DIO and TIO. An optional description of the method may also be entered on this screen. Again, to enter this method into the method table the user would press the *Save* button. To add another method the user would press the *New* button or would press the to return to the add model screen if there are no more methods to be entered.

To enter the request information and the flow of requests through the distributed application the user will use the *Request Types* and *Method Calls* screens. These two screens are shown in Figures 3.12 on Page 64 and 3.13 on Page 65 respectively. The *Request Types* screen is used to define the different types of requests in the system and the *Method Calls* screen is used to define the flow of the individual requests

The screenshot shows a window titled "Add New Model Information". Inside, the "Request Information" section is active. It contains the following fields and controls:

- Model Name:** Greg
- Request Name:** [Empty text box]
- Request Throughput:** 0
- Req/sec:** [Empty text box]
- Description:** [Empty text box]
- Please Enter the first Method Called by this request:**
- Method Name:** [Empty text box]
- Method Object:** [Empty text box]

On the right side of the window, there is a vertical stack of three buttons: "Save", "New", and "Done".

Figure 3.12: Add a request screen

through the system. The functionality of the screens is the same as the previous screens. To enter the information into the tables the users presses the *Save* button. The *New* button is to enter more information and the *Done* button is used to return to the add model screen. Once all the information has been entered to define a model, the user can press the *Done* button on the add model screen to return to the *Main* screen of DASS.

3.3.2 Edit an Existing Model

A second main function of DASS is to edit an exiting model. Once in the edit screen, the user can edit any of the hardware and software components of the distributed application system. The node information is the first to be entered into the system. Figure 3.14 on Page 66 shows the form used to input the data for the definition of the nodes. The values that are needed to define a node are the *Node Name*, *Model Type*, *MIPS*, *Terminal* and *Disk IO* service times and the number of disk drives on the node. Figure 3.14 depicts a node named *Node 1* that is a *Personal Computer*

Add New Method Information

The Current Model is : Model 1

Model: Model 1

Please fill in the following fields for the new Method Call:

Request Name	Client Method	Client Object	Server Method	Server Object	Visits	Call Type

Existing Method Calls for this Model

Request Name	Client Method	Client Object	Server Method	Server Object	Visits	Call Type

Figure 3.13: Add the method calls screen

386 that is rated at 5 MIPS. It has a terminal access speed of 10 milliseconds and has no disks.

The *capsule*, *cluster* and *object* edit screens are very similar in the fact that the only information they hold is the model name, the name of the specific component and the name of the component that it is allocated to. Figures 3.15, 3.16 and 3.17 on Pages 66 and 67 depict each of these screens.

Once all of the nodes, capsules, clusters and objects have been defined in the model description, the user can then define the methods and allocate them to the objects. Figure 3.18 shows the screen used to enter the method information. The method depicted in Figure 3.18 on Page 68 is part of *Model 1* and is allocated to the *Check Credit* object. The other information that is stored within the tables of DASS tells us that every time the method is executed, 30,000 instructions are performed by the CPU and 2 disk operations are performed. It is not necessary to state what node this method executes on as DASS maintains the links to the other components. This method is allocated to the *Check Credit* object which is allocated to a cluster.

The 'Node Information' form is a graphical user interface window titled 'Node Information'. It contains several input fields and buttons. The 'Model Name' field is set to 'Model 1'. The 'Node Name' field is empty. The 'Model Type' field is set to 'PC 386'. The 'MIPS' field is set to '5'. The 'TIO Service Time' field is set to '1.0000e-2'. The 'TIO Uses Network' field is set to 'Nn'. The 'DIO Service Time' field is set to '0.0000e+0'. The 'Number Disks' field is set to '0'. There are two buttons at the bottom left: 'OK' and 'Cancel'. There are three buttons on the right side: 'Back', 'Forward', and 'Help'.

Node Information

Model Name : Model 1

Node Name : Model 1 Model Type : PC 386 MIPS : 5

TIO Service Time : 1.0000e-2 TIO Uses Network : Nn

DIO Service Time : 0.0000e+0 Number Disks : 0

OK Cancel

Back Forward Help

Figure 3.14: Node Entry Form

The 'Edit Capsule Information' form is a graphical user interface window titled 'Edit Capsule Information'. It contains several input fields and buttons. The 'Model Name' field is set to 'Model 1'. The 'Capsule Name' field is set to 'Capsule 1'. The 'Node Name' field is set to 'Node 1'. There are two buttons at the bottom left: 'OK' and 'Cancel'. There are three buttons on the right side: 'Back', 'Forward', and 'Help'.

Edit Capsule Information

Model Name : Model 1

Capsule Name : Capsule 1 Node Name : Node 1

OK Cancel

Back Forward Help

Figure 3.15: Capsule entry form

The screenshot shows a window titled "Edit Cluster Information". Inside, the text "Model Name : Model 1" is displayed. Below it, there are two labels: "Cluster Name :" followed by a text box containing "Cluster 1", and "Capsule Name : Capsule 1". Under the "Cluster Name" text box are two buttons: "Previous" and "Next". To the right of these buttons is a vertical stack of three buttons: "OK", "Cancel", and "Help".

Figure 3.16: Cluster entry form

The screenshot shows a window titled "Edit Object Information". Inside, the text "Model Name : Model 1" is displayed. Below it, there are two labels: "Object Name :" followed by a text box containing "Object 1", and "Cluster Name : Cluster 1". Under the "Object Name" text box are two buttons: "Previous Record" and "Next Record". To the right of these buttons is a vertical stack of three buttons: "OK", "Cancel", and "Help".

Figure 3.17: Object entry form

Edit Method Information

Model Name : Model 1 **Object Name :**

Method Name : Check Credit **Method CPU Inst :** 30,000.00

Method TIOs : 0.00 **Method Bytes/TIO :** 0.00

Method DIOs : 2.00 **Method Bytes/DIO :** 512.00

Method Description :

Figure 3.18: Method entry form

That cluster is allocated to a capsule which is on a node. DASS follows this link to find out what node the instructions and disk operations are executed on.

The previous screen have shown how to edit the configuration of an existing model by editing each of the components separately. This is fine if the user only want to move a single object from one cluster to another, but if the user wants to entirely reconfigure the model then it would be very tedious. DASS has the capability to reduce the amount of work needed to modify an existing model.

3.3.3 Modifying a Configuration from a Single View

The user also has the capability to view the components of the entire configuration on one form. This allows the configuration to be easily modified. Figure 3.19 shows this screen. It is accessed from the *Edit Model* screen. This screen is useful to view all of the embedded components at the same time and to reallocate the software components to create an alternate configuration.

The top level component on the screen is the hardware node. The software

Modify the Configuration of a Model

Model Name : Model 1

Node Name :

Capsule Name

Capsule 1

Cluster Name

Cluster 1

Object Name

User

Method Name	Method Description	CPU Inst	TIOs	Bytes/TIO	DIOs	Bytes/DIO
Client		5,000.00	5.00	1,024.00	0.00	0.00

Figure 3.19: Configuration modification screen

components that are distributed on that node are listed in the tables on the screen. To view the other nodes in the distributed application system the user would simply click on the *Next* button in the upper right corner of the screen. All capsules that are on that node would be displayed in the *Capsule Name* table. To scroll through the capsule, if there are more than shown on the screen, the user can use the scroll bar on the right side of the table.

The other tables will show only the software components that are composed within the next higher level component. For example, in Figure 3.19 *Cluster 1* is composed within *Capsule 1*. To view the clusters that are composed within *Capsule 2*, the user would have to select that capsule by clicking on it with the mouse or by using the scroll bar. This is the same for all of the other software component tables. The user can scroll through any of the components that are part of the shown configuration.

To modify the configuration, the user simply selects the components they wish to move and change its location. This is accomplished by double clicking with the left

mouse button to select the component to be moved. After selecting the component, a pop-up window appears showing the components current composition and a list of available components to which it may be moved. The user simply selects the new location of the component from the list supplied and DASS automatically updates the configuration.

This forms based textual input is a basic method of entering data. It takes time and effort to complete a configuration of a distributed application system. A second techniques of entering the data that can be considered for future research is using traces of real systems or prototypes. A *trace* is defined as a record of events in the order as they occur in system. Trace events can be generated with each procedure call and often include timestamps. The sequence of the events and the difference between the timestamps can be used to obtain the visits, V , between methods and the demand, D , on the system resources.

3.3.4 Automated Data Input

Hrischuk et al. [26] describe a technique that can provide test cases and estimates for model parameters. This technique describes an object-based systems using a prototype to record object interactions in the system's execution. These traces are a record of the system's execution for the request types. A single trace is recorded for each request type in the system. From these traces, *workthreads* are built for each request type. A workthreads gives the resource consumption at each device made by a request. DASS could automatically create a configuration of a distributed application system using the information supplied by this technique.

WARTS [27] is a second tool that could be used to provide execution traces to DASS. The Wisconsin Architectural Research Tool Set (*WARTS*) is a set of tool that can provide profiles and traces of computer applications. The WARTS toolset contains a program tracing system called *QPT*. *QPT* inserts code into an application executable that records the execution of blocks. These counts are then analysed to calculate the execution costs of the application methods. *QPT* can also generate a full program trace of the application that can be used with the execution costs

Figure 3.20: Run analysis screen

to build an applications configuration within DASS. Once the application has been described, the performance characteristics of the configuration can be studied.

3.4 Using DASS and ABA Together

Once a model has been created and a configuration has been developed for the distributed application system the user can perform the utilization analysis. This section will describe how DASS and ABA can be used together to perform a utilization analysis of the distributed application system. Section 2.2.2 on Page 35 defined the inputs that the ABA tool needs to calculate the $\vec{\beta}$ that define the customer request ratios where combinations of hardware resources saturate. These $\vec{\beta}$ will be used as our workload test cases in the utilization analysis. DASS can produce the information required by the ABA tool by using the data defining the distributed application system. Figure 3.20 depicts the screen where the data analysis is executed from within DASS.

The *Resource Demand by Request Type* function calculates the demand on the hardware resources aggregated by the request classes in the system. This is accomplished by scanning through the request table and identifying all of the requests for a specific model. Next DASS scans through the *method call* table to identify all of the methods that this request visits and the number of disk operations and CPU instructions that are executed by this method. Once all of the information is read from the tables, DASS calculates the demands placed on the hardware resources using the service times that are found in the *node table*. The hardware resource demands are calculated by using Equations 3.3 on Page 53 and 3.4 on Page 53.

This information is the input required by the ABA tool to compute the $\tilde{\beta}$. DASS exports this data into a file which can be read into the ABA tool. The $\tilde{\beta}$ are then computed and are written to a file that can be imported into DASS. DASS takes these population vectors and calculates the maximum request throughputs and uses these values to calculate the maximum resource utilizations. The asymptotic request throughputs are calculated by using Equation 2.6 on Page 39. The demand on each of the hardware resources has already been calculated in the previous step. Since the $\tilde{\beta}$ have been computed by the ABA tool, DASS can calculate the Asymptotic request throughputs. This will be illustrated by an example application in Chapter 4.

Once the asymptotic throughputs of the request classes have been calculated the user can then see if any of the software components exceed or approach their utilization limits. This will be described in the next section.

3.4.1 Outputs of DASS

The user can calculate the utilization limits on any of the software components. Figure 3.20 on Page 71 shows the *Resource Demand by Method*, *Resource Demand by Object* and *Resource Demand by Capsule*. Each of these buttons produces tables which contain the demand on system resources aggregated by the specific software component. These demands include the service of the resources used by the software component. Figure 3.21 on Page 73 illustrates the included service of the resources. Capsule 1 makes use of the hardware resources on its node but it also makes use of

Figure 3.21: Included service of resources

Capsule 2 on Node 2. The service demand produced by Capsule 2 is also included in the service demand of Capsule 1.

If the user would like to know the utilization levels on the capsules in the system, they would press the *Resource Demand by Capsule* button. DASS calculates the resource demands aggregated by capsule, the throughputs using the workloads as defined by the ABA tool, and uses Equation 3.1 to calculate the capsule utilization levels.

The results can then be viewed graphically within DASS. Figure 3.22 on Page 74 shows a capsule utilization graph produced by DASS. This graph depicts a configuration with a workload comprised of three request types. The utilization levels of a capsule are given for several workload scenarios. Some of the workloads consist of all three types of requests while others have only one or two request types. If the capsule shown in Figure 3.22 had a limiting utilization of five due to the software technology constraints, we would have to modify its configuration to satisfy this constraint under workload scenarios 11 and 12. This could be done by moving some of the objects from the overly utilized capsule to a less utilized capsule. To compute other performance metrics, such as mean request response times and resource queue lengths, the data available from DASS can be used with other performance tools.

Figure 3.22: Example utilization graph

The data contained within DASS could in the future be used to generate a layered queueing model (*LQM*). The resource demands, visit numbers between methods, service rates and workloads are input parameters for LQM techniques such as the Method of Layers (*MOL*) [15] and SRVN [28]. These techniques can be used to estimate mean system response, resource utilizations and mean resource queue lengths.

3.5 Summary

This chapter has described the Distributed Application System Sizer. DASS is a tool that can be used in a utilization analysis of distributed application system to find a configuration that reduces or eliminates software bottlenecks. These software bottlenecks can limit the maximum throughput attainable in a distributed system by creating delays at software servers instead of the hardware servers.

The data structures of DASS were described and links between the different software components were explained. It was also shown how the engineering structures of the Reference Model of Open Distributed Computing are accommodated by the table structures in DASS.

The information needed for DASS to perform the analysis was described. This information was broken down into the hardware and software categories. The hardware information needed included the node parameters including the CPU service rates and the disk service times. The software information included the software components and the composition that described the configuration of the distributed application system. The detail information needed to describe the objects' methods was also given.

The operations of DASS were also described. The data input screen used within DASS were shown and explained and how a user enters a models configuration was explained. The process of how to modify a configuration was also explained. The utilization analysis of DASS was explained. By creating different configurations of a distributed application system and running the analysis on each of them, a user can find a configuration that reduces or eliminates potential software bottlenecks. The next chapter illustrates the process of using DASS with bounds analysis to choose workloads for the utilization analysis as described in Section 1.5.

Chapter 4

Performance Study of an Example Application

Chapter 2 introduced several performance evaluation techniques for predicting the performance behaviour of distributed application systems. A methodology for the use of these techniques was given. The methodology started with a simple bounds techniques to reduce the design space. Once the number of test cases has been reduced, more advanced MVA and simulation techniques could then be explored.

Chapter 3 described the Distributed Application System Sizer that can be used to find a composition/distribution strategy to reduce bottlenecks at software resources. The DASS tool is used in conjunction with the Asymptotic Bounds Analysis (ABA) technique described in Section 2.2.2. The DASS tool is intended to be as a first step in analyzing a distributed application system. This chapter will illustrate the process of using DASS by analyzing an example application. The application will be described and a model of the distributed application system will be created in DASS. A utilization analysis of this model will be performed to determine the performance impact of alternative configurations on software component utilizations and customer class throughputs.

Figure 4.1: Computational model of a simple retail store

4.1 Description of the Application

A simple retail store will be used as an example application to illustrate the use of DASS for performance evaluation. This first section will describe the distributed system application that will be analyzed and the subsequent sections will illustrate steps taken in DASS. We will begin the description with a high level computational view of the application and finish with the composition and distribution of the components to produce a configuration of the system as defined on Page 2.

The computational model of the distributed system application describes the fundamental objects within the application. There is no allocation of the objects to hardware devices. The computational model illustrates the relationships between the objects and their overall functionality. The retail store application has six objects in the system which are *client (user)*, *purchase*, *return*, *check-credit*, *inventory* and *warehouse* objects (U,P,R,C,I,W respectively). Figure 4.1 shows the high level representations of the objects in the application.

The basic functions of the computational objects are:

Purchase object : this object processes the information received for purchasing merchandise. The object must verify the customers credit before completing the purchase.

Credit object : this object verifies the customers credit rating with the retail store.

Return object : this object processes the information during the return of merchandise.

Inventory object : this object processes information from both the purchase and return objects. The inventory object communicates to the warehouse on behalf of the others.

Warehouse object : this is the merchandise data storage for the retail store. The inventory object sends the data from the purchase and return objects to the warehouse.

Each of the computational objects in Figure 4.1 on Page 77 must be expanded to include the methods that will perform the actual work of the requests in the system. After the methods are created they must be composed into objects which are allocated to clusters. The clusters are then composed into the capsules. In a real application each object would have many methods, however, to keep this example relatively simple each of the computational objects will have only one method. The names of these methods are:

- Purchase;
- Return;
- Inventory;
- Credit; and
- Warehouse.

The types of requests that will be performed on the system can be used to describe the workload for the distributed application system. There are two types of requests which can be performed in the retail store. These are *purchase* and *return* merchandise. For this application, the purchase request can be further broken into

Node	CPU Svc Rate (<i>MIPS</i>)	Number of Disks	Disk Svc Time (<i>sec</i>)
Node 1	5	<i>NA</i>	<i>NA</i>
Node 2	5	<i>NA</i>	<i>NA</i>
Node 3	10	3	0.01
Node 4	5	3	0.01

Table 4.1: Node service parameters

two classes of request for performance modelling. These classes of request are *small purchase* and *large purchase*. With a small purchase request a client buys an average of two items and with a large purchase request a client buys an average of ten items. These two requests are specified separately to account for the difference in computing resources needed to complete the requests. Within a return merchandise request a client returns an average of one item.

4.2 System Model Definition

To create a performance model of the distributed application system we have to move from the computational view of the application to an implementation model. In order to accomplish this we have to define a composition of the software components in the distributed application system. This composition is an extension of the computational objects as shown in Figure 4.1 on Page 77 to include the software components as described in Section 1.2 on Page 13. These software components are then distributed across the nodes to form a configuration.

The first step in defining a configuration of the retail store is to create the nodes in the system. In this example there are four nodes. Nodes 1 and 2 will be the user terminals in the distributed application system. Nodes 3 and 4 will be the stations that will contain the software components of the system. Table 4.1 on Page 79 shows the hardware specifications of the nodes.

The next step in the definition of the application model is the composition of

Capsule	Cluster	Object	Method
Capsule 1	Cluster 1	User	User
Capsule 2	Cluster 2	User	User
Capsule 3	Cluster 3	Purchase	Purchase
Capsule 3	Cluster 3	Return	Return
Capsule 3	Cluster 4	Credit	Credit
Capsule 4	Cluster 5	Inventory	Inventory
Capsule 4	Cluster 5	Warehouse	Warehouse

Table 4.2: Composition strategy 1

the software components. The Purchase and Return objects will be composed into a single cluster, the Credit in a second cluster and the Inventory and Warehouse objects will be composed into a third cluster. These clusters will be named *Cluster 3*, *Cluster 4* and *Cluster 5* respectively. The composition of the capsules is the following, *Capsule 3* is composed of Clusters 3 and 4, and *Capsule 4* is composed of Cluster 5. The capsules identify the operating system processes of our system. Table 4.2 shows this composition strategy.

Once the nodes have been defined and the composition of the software components has been completed, the configuration of the system can be entered into DASS. The final part of the configuration is the distribution of the capsules to the nodes. Nodes 1 and 2 are client nodes which submit the requests to the system. Node 3 has Capsule 3 allocated to it and Capsule 4 is distributed to Node 4. Figure 4.2 on Page 81 shows the configuration of the example application.

Before the performance evaluation of the system can be done, the resource requirements of the objects' methods have to be defined in DASS. We also need the path of the requests as they flow through the distributed application system in order to calculate the resource requirements by request type. This information would be available from the system designers or from traces of existing systems. For the example Table 4.3 on Page 82 shows the resource requirements for the object's methods in the application. In this example, we assume the resource requirements for each

Figure 4.2: Configuration 1 of example application

Object	Method	CPU inst	Disk IOs
User	User	5000	0
Purchase	Purchase	55000	3
Return	Return	55000	3
Inventory	Inventory	35000	0
Check_credit	Check_credit	30000	2
Warehouse	Warehouse	15000	2

Table 4.3: Resource requirements

method is the same for each request.

The path of the requests is outlined in Table 4.4 on Page 83. During a purchase request, the purchase method calls the check_credit method and then calls the inventory method. The inventory method calls the warehouse method to update the warehouse database, the number of calls between the inventory and warehouse methods corresponds to the number of items being purchased. A return merchandise request executes much the same way as a purchase request, the only difference being there is no call to the check_credit method.

4.3 Performance Analysis

Each type of request causes visits between methods, and in turn each of the methods in the application makes demands on the system's resources. Using these resource demands and visits between methods for each type of request, the DASS tool calculates the demands on each of the different software resources and node devices by request type. Factors that affect the demands on system resources are the resource requirements of the methods and their need for communication. In this example, we assume that each communication between methods on different nodes (*internode*) require 25000 CPU instructions per method and communications between methods on the same node (*intranode*) require 1000 CPU instructions per method. The communication costs are shown in Table 4.5. Table 4.6 shows the demands on hard-

Request Name	Client		Server		Number of Visits
	Object	Method	Object	Method	
Small Purchase	User	User	Purchase	Purchase	1
	Purchase	Purchase	Check_credit	Check_credit	1
	Purchase	Purchase	Inventory	Inventory	1
	Inventory	Inventory	Warehouse	Warehouse	2
Large Purchase	User	User	Purchase	Purchase	1
	Purchase	Purchase	Check_credit	Check_credit	1
	Purchase	Purchase	Inventory	Inventory	1
	Inventory	Inventory	Warehouse	Warehouse	10
Return	User	User	Return	Return	1
	Return	Return	Inventory	Inventory	1
	Inventory	Inventory	Warehouse	Warehouse	1

Table 4.4: Request paths

Communication Type	CPU Instructions
Internode	25000
Intranode	1000

Table 4.5: Communication costs

ware resources of nodes 3 and 4 aggregated by request type. For the remainder of this example the hardware resource demands at nodes 1 and 2 will not be considered. These nodes support user interfaces and can easily be upgraded by adding more user workstations to the network.

Request Type	Resource			
	1	2	3	4
Type	CPU3	CPU4	Disk3	Disk4
Small Purchase	0.014	0.019	0.017	0.013
Large Purchase	0.014	0.046	0.017	0.067
Return	0.011	0.015	0.010	0.007

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Table 4.6: Configuration 1's Hardware Resource Demands by Request Type

4.3.1 Workload Selection

To determine which workloads to use in the calculations of the utilization limits, the Asymptotic Bounds Analyser (ABA) tool is used. As mentioned in Section 2.2.2, ABA takes as input the request classes and the hardware resource demands of these requests. These values are computed by the DASS tool, as described in Section 3.4 on Page 71, from the model that has been created and are shown in Table 4.6 this example. The output obtained gives us the points at which each device or combinations of devices saturate to give as the asymptotic throughputs of the distributed application system. These are the points at which we expect extreme values for the utilization of the software components.

Figure 4.3 on Page 85 shows the graph produced by the ABA tool depicting the saturation sectors for configuration 1 of the example application. The sectors outline the areas where the devices or combination of devices are saturated. The points at which the sectors meet are our points of interest in determining the asymptotic utilization values of our software components. Using these points, the asymptotic throughputs of the request types can be calculated. Using the utilization law in Equation 3.1 on Page 48 we can calculate the asymptotic throughputs of the request types at our points of interest.

We know that $U = 1$ since the devices at these points is saturated and we know the demand D_i of request type i at the device, the throughput X_i of request type i can be calculated by $X_i = \beta_i \left(\frac{1}{D_i} \right)$. The asymptotic throughputs of all request types can be found at all of the saturation points by following this method. Table 4.7 gives the asymptotic request throughputs for the configuration of the distributed application system. The first $\vec{\beta} = (0.597, 0.403, 0.000)$ signifies that 59.7% of the customer population is comprised of the *small purchase* request class and that 40.3% is comprised of the *large purchase* request class. The saturated devices at this point are the CPU and disk on Node 4. However, the bottleneck strength for the CPU at this point is $\gamma_{cpu} = 1$ and for the disk is $\gamma_{disk} = 0$ so we use only the demand

Figure 4.3: Saturation sectors for configuration 1 of the example application

values of the CPU. From Table 4.6 we can see the resource demands at the CPU on Node 4 by each of the request classes. The throughputs achieved by each of the request classes is calculated by:

$$\text{Small Purchase} \quad X_{\text{small}} = 0.597 \left(\frac{1}{0.019} \right) = 31.42 \quad (4.1)$$

$$\text{Large Purchase} \quad X_{\text{large}} = 0.403 \left(\frac{1}{0.046} \right) = 8.76 \quad (4.2)$$

The other request throughputs are calculated in the same fashion.

4.3.2 Utilization Analysis

To calculate the utilization of the software components we use the procedure outlined in Section 3.4.1 on Page 72. DASS uses the resource demands and request path information to compute the resource demands on the hardware resources such as CPUs and disks. These demands can then be computed for any of the software resources such as capsules. Table 4.8 gives the capsule demands aggregated by request type. The utilization law, Equation 3.1, is then used to discover the utilization of the resources for the given throughputs.

Intersection point	Small purchase throughput	Large purchase throughput	Return merchandise throughput
(0.597, 0.403, 0.000)	31.42	8.76	0.00
(0.000, 0.544, 0.456)	0.00	11.82	30.37
(1.000, 0.000, 0.000)	52.63	0.00	0.00
(0.000, 0.000, 1.000)	0.00	0.00	66.67
(0.413, 0.587, 0.000)	31.74	8.77	0.00
(0.000, 0.793, 0.207)	0.00	11.83	29.59
(0.000, 1.000, 0.000)	0.0	14.93	0.00

Table 4.7: Configuration 1 - Request throughputs at points of interest

Request Type	Capsule 3	Capsule 4
Small Purchase	0.062	0.032
Large Purchase	0.143	0.113
Return	0.043	0.022

Table 4.8: Capsule demands by request type in seconds

Capsule	Cluster	Object	Method
Capsule 3	Cluster 3	Purchase	Purchase
Capsule 3	Cluster 3	Return	Return
Capsule 3	Cluster 3	Inventory	Inventory
Capsule 3	Cluster 4	Credit	Credit
Capsule 4	Cluster 5	Warehouse	Warehouse

Table 4.9: Composition strategy 2

The utilizations can be compared with their limits as specified in the software technology constraints. For this configuration, ABA found 7 workload cases as shown in Table 4.7. Figure 4.4 on Page 88 shows configuration 1's asymptotic capsule utilizations of the simple retail store. For this example we assume that Capsules 3 and 4 have utilization limits of four and two respectively. This configuration approaches or exceeds its limit for Capsule 4 in scenarios 1, 2, 5 and 6. These results indicate that there is the possibility that Capsule 4 could become a software bottleneck. To try and find a configuration for the distributed application system that avoids any potential bottlenecks the model can be reconfigured and the analysis repeated.

4.3.3 Alternate Configuration

To modify the example application, the Inventory object will be moved from Cluster 5 to Cluster 3. This change is shown in Table 4.9 and is depicted Figure 4.5 on Page 89 which illustrates the new configuration. In order to calculate the new utilization levels of configuration 2 we need the resource demands by request type. These demands will be passed to the asymptotic bounds analysis tool to calculate the request type customer population ratios that will be used to calculate the asymptotic request throughputs for the new configuration. The resource demands by request class are shown in Table 4.10 on Page 89.

The bounds analysis done by the ABA tool produces a device saturation graph

Figure 4.4: Simple retail store: capsule utilizations for configuration 1

Figure 4.5: Configuration 2 of example application

Request	Resource			
	1	2	3	4
Type	CPU3	CPU4	Disk3	Disk4
Small Purchase	0.020	0.016	0.017	0.013
Large Purchase	0.040	0.080	0.017	0.067
Return	0.014	0.008	0.010	0.007

Table 4.10: Configuration 2's Hardware Resource Demands by Request Type

Figure 4.6: Saturation sectors for configuration 2 of the example application

that is shown in Figure 4.6 on Page 90. It has the same shape as the graph for the first configuration, however the bottlenecks have shifted. In the first configuration the bottleneck devices were the CPU and disk on node 4. In configuration 2 the bottleneck devices are now the CPU on node 3 and the CPU on node 4. These results are as expected because we have shifted some of the work from Node 4 to Node 3. This shift in the bottleneck devices gives us the asymptotic throughputs shown in Table 4.11.

The nested capsule demands needed to calculate the utilization levels for the second configuration are shown in Table 4.12. Using these values and the asymptotic request throughputs we can calculate the maximum utilization levels of the capsules. These utilization levels are illustrated in Figure 4.7. The utilization limits for the two capsule are 4 and 2 respectively and it can be seen that the limit is never exceeded for any of the workload scenarios. However, under scenario 5, the limit is reached so there is a possibility that a bottleneck may arise.

Intersection point	Small purchase throughput	Large purchase throughput	Return merchandise throughput
(0.869, 0.131, 0.000)	43.69	3.27	0.00
(0.000, 0.290, 0.710)	0.00	7.26	50.03
(1.000, 0.000, 0.000)	50.25	0.00	0.00
(0.000, 0.000, 1.000)	0.00	0.00	70.42
(0.732, 0.268, 0.000)	45.76	3.35	0.00
(0.000, 0.594, 0.406)	0.00	7.43	50.74
(0.000, 1.000, 0.000)	0.00	12.5	0.00

Table 4.11: Configuration 2 - Request throughputs at points of interest

Request Type	Capsule 3	Capsule 4
Small Purchase	0.070	0.033
Large Purchase	0.204	0.147
Return	0.036	0.015

Table 4.12: Configuration 2's capsule demands by request type in seconds

Figure 4.7: Capsule Utilizations for Configuration 2

4.4 Summary

It can be seen that both configurations provide similar maximum throughputs for each of the request types. Under Configuration 1 the maximum throughputs for the small purchase, large purchase and return requests are 52.63, 14.93 and 66.67 respectively. Under Configuration 2 the maximum throughputs are 50.25, 12.5 and 70.42. From these results either configuration would be adequate if we were looking at the throughput rates alone. However, if we also look at the utilization levels of the capsules we can conclude that Configuration 2 would be a better choice. This is based on the fact that it never exceeds either of its capsule's utilization limits whereas in Configuration 1, the utilization limit of Capsule 4 is exceeded under scenarios 1, 2 and 5 and is approached in scenario 6.

In general, if resource utilization levels approach or are greater than their limits, we expect software queueing problems and possibly software bottlenecks. We can then use DASS to reconfigure the system. Options are to replicate or reallocate capsules, clusters, or objects or to partition them. We iterate between DASS and the bounds analysis until a configuration is found that satisfies its throughput and software technology constraints over all workload conditions. If no single configuration satisfies all workloads combinations, the system's software architecture should be chosen to support the changes needed to easily reconfigure the software to support the various workload conditions.

Chapter 5

Case Study

In Chapter 4 it was shown how DASS can be used in a performance study. A small example application was described in DASS and it was shown how a composition and distribution strategy can be found for a distributed application system over a full range of workloads. To show the merit of this technique and DASS a case study of a distributed system being developed by SystemHouse Ltd. (SHL) will be given. The project and the application being developed will be described followed by the steps taken in the analysis of the system. This case study will follow the involvement from the definition of the workloads through the modelling of the system and then the analysis.

5.1 Project Description

SHL is designing a distributed system that will automate the Supply System of the Canadian Forces (CF). The name of the project is Canadian Forces Supply System Upgrade (CFSSU). The Supply System of the CF is a large, complex system that incorporates many aspects of materiel handling. These include distribution and requirements planning (DRP), local procurement (LP) and warehousing and distribution centre (WDC) system. DRP is mainly administered at the National Headquarters (NDHQ) where the materiel planning for the whole of the CF is done. DRP consists of the initial purchasing of materiel as well as the distribution amongst all

units within the CF. The second aspect of the Supply System is local procurement. The commander of any unit within the CF has the mandate to locally purchase materiel, up to a set amount, without requiring authority from NDHQ. The amount that the unit can purchase locally varies and depends on the type of unit. A base is able to purchase more materiel than a small station but less than a command.

The third component of the Supply System are the warehousing and distribution centre system. The WDC are the materiel depots that receive and issue all materiel for the CF that is not locally purchased by the individual units. There are two main supply depots within the CF that manage all of the materiel that is to be shipped to the bases across Canada. For supply reasons, Canada is divided into two regions, East and West, and each of these depots manage the materiel for their respective regions. CFD Edmonton controls the materiel for the western region and CFD Montreal is allocated to the eastern region. The depots receive the materiel, store it in the depot and then issue the materiel according to orders sent in by the bases. This thesis will do a case study on the warehousing and distribution control system that is being installed at the supply depot in Montreal, CFD Montreal.

5.1.1 System Description

The main purpose of the WDC system at the supply depots is to maintain a database of all the materiel stored in the warehouse. Other information in the database includes inbound orders and shipments and outbound orders. The system being designed and installed by SHL is a distributed system that makes use of client/server technology. The system consists of SparcStations as clients and SparcServers as the database server and support servers. The server and clients are networked together over Ethernet. The description of the system has been obtained from meetings between the project management office (PMO) of the CFSSU and the performance engineering (PE) team of SHL. The PMO has supplied SHL with the system functions and the basic descriptions of the different stages in of the requests.

There are two main functions accomplished by the WDC system, these are the receipt request and the issue requests. The receipt request function of the WDC

system is initiated when an inbound shipment arrives with materiel to process. This function is virtually a continuous action. The initial step is to receive the materiel from the shipments and to process the inbound orders by matching the received materiel to these orders. The processing of the materiel is handled by bar code scanning equipment and a graphical user interface (GUI) to help the user match the shipment to the orders in the database. The user scans the received materiel into the system and all orders requiring that equipment is retrieved from the database. The user fills as many orders with the received materiel as possible and the partially or unfilled orders remain open in the system. The received materiel is then sent to the storage cells to be placed on the shelves until needed for an outbound order. The storage cells rescan the materiel to determine what storage bin can be used for the equipment. This storage bin selection is automatically done by the WDC system. The materiel is then taken to the appropriate storage bin and the operator can finalize the receipt request.

The issue request is initiated by an outbound order. The user receives an outbound order list from the system and verifies that all the materiel needed for the order is available. The user picks the equipment needed for the order and generates a task for the storage cell to retrieve the materiel from the storage bins. The retrieved materiel is sent to the packing cell where it is scanned to match the materiel to the outbound orders. The materiel is then sent to the appropriate packing bay to be put into the parcel for the specific order. Once all of the materiel for the order has been packaged, the user finalizes the issue request.

The functionality of the WDC system will be supplied by a commercial off the shelf (COTS) software package. Uniteq's WH&DC software package will be used to automate the functionality of the warehouses in the Supply System. The WH&DC package is a distributed application system that incorporates a GUI for the client workstations, support server processes to handle the requests and utilizes a database system for the materiel and order processing. WH&DC includes support for major database systems such as SYBASE, Oracle and FoxPro.

The WH&DC package can be broken down into three components, the client

process, the support servers and the data server. The client process is a GUI terminal where the users can initiate the materiel receipt and order processing.

The support servers are the background processes that process the information sent by the client GUI. These servers include the pick server, task server, putaway server and receive server. The pick server handles the requests to pick materiel to fill the outbound orders. The task server is used to confirm the put, pick and load tasks. The confirm put task occurs when the materiel received from an inbound shipment is put into the storage bins. The confirm pick task is initiated when the user has retrieved the materiel needed to fill an outbound order. The task server completes a confirm load task after an outbound order has been filled and is ready to be shipped out. The receive server processes the data information from the inbound shipment to the database. When the user scans in the materiel and associates it to the inbound orders, the database is updated with the shipment. The putaway server handles the task of assigning the incoming materiel to a storage location. It retrieves the data and locates an open storage bin that can accommodate the materiel.

Since the software package used in the WDC system is a COTS package, the composition of the processes can not be changed. However, the distribution issue of the processes can be assessed by DASS and the threading levels of the processes can be chosen. The next section looks at creating a model of the system and evaluating the distribution issues of the different servers in the distributed application system.

5.2 Model Description

The first step in the evaluation process is to create a model depicting the configuration and execution of the system. The WH&DC software package is to be installed on Sun SPARCstations and SPARCservers. The client process will be executed on a SPARCstation5 with the support servers and the data server running on SPARCserver20s. The client process will not be modelled, each user has a dedicated workstation. The SPARCserver to be used in the system are rated at 167.4 MIPS and have a disk service time of 15 ms. The hardware configuration states that the

Node	CPU Service Rates (MIPS)	Disk Service Times (sec)
Client Node	78	N/A
Support Node	167.4	0.015
Data Node	167.4	0.015

Table 5.1: Case study - node service parameters

support servers will be located on a single SPARCserver and the data server will be installed on a second SPARCserver. Table 5.1 lists the node service parameters of the hardware components in the case study. The software components descriptions can be entered into the model once the hardware components have been defined. The composition issues of the processes can not be addressed since the server processes can not be modified. However, they can be modelled.

The configuration of the distributed application is depicted in Figure 5.1 on Page 99. There are four support server objects namely the pick, task, putaway and receive servers which are all composed into a single capsule. The pick server object has one interface with a method called *Generate_pick*. This method is executed when processing an outbound order. It processes the items in the order and informs an operator to retrieve the required materiel from the storage bins.

The task server object is composed of three methods, the *Confirm_put*, *Confirm_pick* and *Confirm.load*. The *Confirm_put* method is executed when the materiel from an inbound shipment is put away into the storage bins. The *Confirm_pick* method is executed during the processing of an outbound order. The user must inform the system when all of the materiel needed for a specific order has been retrieved from the storage bins. The third method *Confirm.load* is executed when all of the materiel has been packaged up and is ready to be shipped out. The Putaway and Receive server objects both have one method each. The Putaway server's method *Receipt_put* coordinates the task of finding suitable storage bins for the materiel in the inbound shipment. The Receive server's method *Process_receipt* updates the data server with the materiel that was received on the inbound shipment. Each

Figure 5.1: Configuration of WH&DC application

of these objects are composed into separate clusters which are composed into the single support server capsule.

The data server node has a single object that is composed of all the methods on that node. This object is composed into a single cluster which comprises the data server capsule which handles the order and materiel data for the system. The data object has many interfaces to update and retrieve the data in the system. The interfaces are *create_inorder*, *process_inshipment*, *associate_orders*, *process_putaway*, *create_wave*, *create_outorder*, *create_outshipment*, *pick_generate*, *pick_confirm*, *load_confirm*, *process_load*. The *create_inorder* method updates the data files when an inbound order is created. The *process_inshipment* processes the data received from a shipment and the *associate_orders* method associates the materiel received from an inbound shipment to a particular order. The *process_putaway* updates the storage bin inventory database with the locations to which the materiel from the inbound shipment has been allocated. The *create_wave* method creates an outbound wave that will include any outbound orders. The *create_outorder*, *create_outshipment*, *pick_generate*, *pick_confirm*, *process_load* and *load_confirm* methods are executed during an outbound shipment to process the information on the data server. Table 5.2 illustrates the composition of the software components and Table 5.3 lists the individual processes, their methods and their resource demands.

5.2.1 Workload Definition

The system description in Section 5.1.1 described the main functions of the WH&DC package. There are two dominant functions in the system which are to process the receipt and to process the issue of materiel. The process of receiving materiel can be broken into two separate requests - *process receipt*, and *receipt putaway* - and the process of issuing materiel is one system request, *process issue*.

This request flow information together with the resource demands made by each of the methods are used by DASS to calculate the information to be used in the bounds analysis. The resource demands by request type is shown in Table 5.4. This information is passed to the asymptotic bounds analysis tool which calculates the

Capsule	Cluster	Object	Method
Support Server	Pick cluster	Pick object	Generate_pick
Support Server	Putaway cluster	Putaway object	Receipt_put
Support Server	Receive cluster	Receive object	Process_receipt
Support Server	Task cluster	Task object	Confirm_put
			Confirm_pick
			Confirm_load
Data Server	Data cluster	Data object	Process_putaway
			Associate_orders
			Process_inshipment
			Create_inorder
			Create_outorder
			Create_wave
			Process_pick
			Pick_confirm
			Process_load
			Create_outshipment

Table 5.2: Case study - software composition

Method	Millions of CPU Instructions	Disk Operations
Generate_pick	107.14	0
Receipt_put	214.27	0
Process_receipt	190.84	0
Confirm_put	159.03	0
Confirm_pick	611.01	0
Confirm_load	297.97	0
Process_putaway	617.46	13
Create_inorder	390.04	15
Associate_orders	167.4	12
Process_inshipment	1050	53
Create_outorder	97.09	13
Create_wave	636.12	16
Process_pick	266.17	10
Pick_confirm	573.46	20
Process_load	532.45	9
Create_outshipment	210.92	39

Table 5.3: Case study - method resource requirements

Request	Resource		
	1	2	3
Type	Support Server CPU	Data Server CPU	Data Server Disk
Process Receipt	1.14	9.602	1.2
Receipt Putaway	2.23	3.689	0.195
Process Issue	6.07	13.837	1.605

Table 5.4: Hardware Resource Demands by Request Type

Figure 5.2: Saturation Sectors for the Canadian Forces Supply System Upgrade

bottleneck devices based solely on the hardware resources. The bottleneck devices are depicted in Figure 5.2. As the figure shows, resource 2 which is the CPU on the data server is the only bottleneck device.

The request ratios obtained from the bounds analysis as shown in the saturation sector graph are given in Table 5.5 along with their corresponding throughputs. These throughputs are then used to calculate the utilization levels of the capsules. Figure 5.3 on Page 105 shows the capsule utilization levels for each of the workload scenarios. The highest utilization of the Data Server capsule is 1.1234 and the Support Server capsule has a maximum utilization level of 1.07335. The Data Server and Support Server capsules are both multithreaded and each has a utilization limit

Intersection point	Process Receipt throughput	Receipt Putaway throughput	Process Issue throughput
(1.000, 0.000, 0.000)	0.104	0.000	0.000
(0.000, 1.000, 0.000)	0.000	0.271	0.000
(0.000, 0.000, 1.000)	0.000	0.000	0.072

Table 5.5: CFSSU Request throughputs at points of interest

Node	CPU Service Rates (MIPS)	Disk Service Times (sec)
Client Node	78	N/A
Support Node	167.4	0.015
Data Node	334.8	0.015

Table 5.6: Case study - node service parameters of alternate configuration

greater than 2. Since both of the maximum utilization levels are well below the utilization limits for each of the capsules we can conclude that there will not be any software bottlenecks in this system.

5.3 Alternate Configuration

From the previous example we can see that the utilization limits are not exceeded but the request class throughputs are quite low. To try and increase the request class throughputs but still maintain the utilization integrity of the capsules we could change the configuration of the distributed application system. Since we can not modify the composition or distribution of the COTS package, we can try to increase the speed of the bottleneck device. For this configuration of the system we will assume that we have installed a multiprocessor machine with two processors as the data server.

Table 5.6 depicts this increase in CPU speed. We have assumed that by doubling the number of CPUs in the node we have doubled the speed of the machine. This in

Figure 5.3: Canadian Forces Supply System Upgrade Utilization Results

Request	Resource		
	1	2	3
Type	Support Server CPU	Data Server CPU	Data Server Disk
Process Receipt	1.140	4.801	1.2
Receipt Putaway	2.230	1.845	0.195
Process Issue	6.070	6.919	1.605

Table 5.7: Hardware Resource Demands by Request Type

fact would not be the case since there would be additional overhead in the scheduling of the processes. However, for this example we have assumed a linear increase in CPU speed.

With this change in configuration we have to recalculate the hardware resource demands by request class. These new demands will be used by the ABA tool to provide DASS with the population mix vectors, $\tilde{\beta}$, that will be used to calculate the asymptotic throughputs of the request classes. Table 5.7 shows the new hardware resource demands. With the increase in CPU speed we would expect that the bottleneck would shift from the CPU on the data server to another device.

Figure 5.4 depicts the saturation sectors for the second configuration of the WH&DC system. As can be seen from the figure, under certain workload conditions the bottleneck has shifted from the data server CPU, resource 2, to the CPU on the support server, resource 1. This is expected because the demand for the *Process Issue* request is still the greatest on the data server CPU whereas the demand for the *Receipt Putaway* request is greatest on the support server CPU.

The next step in this utilization study is to calculate the asymptotic request throughputs by using the $\tilde{\beta}$ given by ABA. For this configuration, ABA calculated 7 workload scenarios. Table 5.8 on Page 107 gives the request class ratios with their corresponding throughputs. The capsule utilizations corresponding to these throughputs is shown in Figure 5.5 on Page 108. Since the utilization levels are again well below the limits for each of the capsule we can conclude there will not be any software bottlenecks even though we have increased the request throughputs.

Figure 5.4: Saturation Sectors for the Canadian Forces Supply System Upgrade - configuration 2

Intersection point	Process Receipt throughput	Receipt Putaway throughput	Process Issue throughput
(0.051, 0.949, 0.000)	0.045	0.426	0.000
(0.000, 0.448, 0.552)	0.000	0.201	0.091
(0.000, 1.000, 0.000)	0.000	0.448	0.000
(0.215, 0.785, 0.000)	0.045	0.425	0.000
(0.000, 0.370, 0.630)	0.000	0.201	0.091
(1.000, 0.000, 0.000)	0.208	0.000	0.000
(0.000, 0.000, 1.000)	0.000	0.000	0.145

Table 5.8: CFSSU Request throughputs at points of interest

Figure 5.5: Canadian Forces Supply System Upgrade Utilization Results

5.4 Summary

The results from both of the configurations show that there will not be a problem with the software components in the CFSSU. The most critical problem is the need for faster processors or a multi-processor machine. The maximum utilization levels of the software components based on the asymptotic throughputs show that the utilization levels will not be reached. The utilization levels for the support server and data server capsules are both greater than 1 but they are multi-threaded capsules so they are able to handle these requests.

However, after increasing the CPU speed on the data server node we can see an increase in the request class throughputs. The maximum request throughputs in configurations 1 and 2 for the *Process Receipt*, *Receipt Putaway* and *Process Issue* are (0.104, 0.271, 0.072) and (0.208, 0.426, 0.145) respectively. From these results it would be beneficial to increase the speed of the data server node to achieve the higher throughputs since the utilization levels are not approached. To study the system even further the analyst can then continue on with a more detailed analysis using an MVA technique to predict the mean system response times and average queue lengths.

After doing this study, the performance of this distributed application system came under scrutiny after an implementation test bed was set up at CFD Montreal. They found no problems with the software configuration of the system but the system throughput was low and response times were too slow. This was mainly due to the high demands at the data server. Had this study been completed earlier the results may have prevented the Department of National Defence from incurring that expenditure.

Chapter 6

Conclusions

The key contribution of this thesis are the development of the Distributed Application System Sizer (DASS) tool and a novel use of bounding analysis. These two ideas together are intended to facilitate the performance engineering process. This novel use of bounding analysis allows an analyst to reduce the number of configurations of a distributed application system at an early stage in the design phase. This can reduce the time and cost spent in the design of the distributed application. Using these techniques helps an analyst to find a suitable set of workload scenarios that can be used to study software behaviour over a wide range of workload conditions. A summary of the work and possible future research are provided in the following sections.

6.0.1 Summary

Distributed environments are increasingly being used by companies to try and increase the availability and performance of their computing base. These distributed environments all have the same main features. The first feature is a *communication mechanism*. An example of this is the remote procedure call. A second common feature is *location transparency*. A *security service* is the third feature that is shared between the distributed environments.

A distributed application system is built on these environment using the concepts of *composition* and *distribution*. Composition is the concept of building the

software processes and distribution is the process of allocating these processes across a distributed system. The combination of composition and distribution form a *configuration* of a distributed application system. The main purpose of this thesis is to demonstrate a tool and technique that can be applied early in the design phase of a system to find a small set of configurations that will support the workloads placed on the system.

Several distributed environments were discussed and how each of them share the above features and how they deal with the issues of composition and distribution is described. To effectively study distributed application system built on any of these environments, a software model that fully reflect their architecture was needed. The Reference Model of Open Distributed Processing is described and it is shown how its components can be used to model a distributed application system. The components of the RM-ODP are:

- Basic engineering objects;
- Clusters;
- Capsules; and
- Nodes.

The notion of a utilization limit on the components within a distributed application system is defined. A hardware resource has a utilization limit of one. Usually, only one person at a time can access a disk or CPU at an given time. A software resource, on the other hand, may have a utilization limit of greater than one. A multithreaded process may have, for example, at most 5 active threads. The utilization limit of this process would be five. The utilization level that we compute is the average number of users for this process. This utilization level must be lower than the number of threads that exist for this process.

A performance evaluation process was described giving a proposed sequence that could be used in the performance evaluation of a distributed application system. The process starts with a utilization analysis based on an asymptotic bounds technique.

The utilization analysis computes the maximum utilization levels of the software components in a distributed application system. If any of the software components exceed their utilization limits, there exists the possibility of a software bottleneck. The distributed application system can then be reconfigured and the utilization analysis can be repeated until a configuration can be found that avoid any potential software bottlenecks.

The next step in the performance evaluation process is to use a layered queueing model (LQM) solver to predict the performance of the distributed application system. The LQM solver uses a mean value analysis technique to compute performance measures that include the mean system throughput and response times, and the average resource queue length and mean resource utilizations. A brief overview was given on the MVA technique.

The Distributed Application System Sizer tool was developed to automate the utilization analysis of a distributed application system. A model of a distributed system is created within DASS that includes the software structures and hardware information described by the RM-ODP. This information is augmented with service demand values and request class data. This information is passed to the Asymptotic Bounds Analyser which calculates the request class ratios that create the hardware resources to saturate. The asymptotic request throughputs occur at these points. This information is passed back to DASS which calculates the asymptotic throughputs. Using these values, the maximum utilization levels of the software components can be calculated. If there are any software components that approach or exceed their utilization limits a software bottleneck may arise. The model can be reconfigured and the analysis can be repeated. This process is repeated until a configuration strategy is found that avoids or eliminates any possible software bottlenecks.

6.0.2 Future Research

There are a couple of areas where future work can be carried on from this research. The first deals with the area of building models within the DASS tool. Section 3.3 described the method of entering data into DASS using a form based entry system.

Future work in this area would be to modify a tracing system that would output the information needed by DASS to build the system models.

The second area of future research is to interface with other analytic techniques. The performance evaluation process outlined in Chapter 2 detailed the use of layered queueing models once a configuration or set of configurations has been found. The information in DASS can be directly converted to create a model in a LQM solver. The future work could automate this process to complete the performance evaluation process such that it starts with the utilization analysis and finishes with the mean value analysis.

It was also stated in Chapter 4 that we do not take into account the delays at the software servers when calculating the demands of the objects. An objects' demand in a distributed application system includes its demands on the system resources such as the processors and disks but it also includes other software servers. In this thesis we assumed the minimum demand time for the objects in the system which assumes a zero queuing delay at the software servers. In a real system, the queuing delays at the software systems would not be zero and must be accounted for. Future research should look at techniques to take these queuing delays into account when calculating the demand placed on the system resources by the objects.

Other areas of future research can include looking at memory and networks as resources in a distributed application system. The current version of the DASS tool looks at only the CPU and disks in the distributed system as the hardware resources. The use of memory in database application can decrease the time spent in accessing the data from the disks and can affect the demands of the requests. Also, the type of network used for the communications can also affect the demands of the resources. These factors could be looked at in future work.

References

- [1] J.A. Rolia, V. Vetland, G. Hills, "Composition and Distribution: Performance Engineering for Distributed Applications". in the Proceedings of IBM Canada/NRC CASCON '95, November, 1995.
- [2] Open Software Foundation, "Introduction to OSF DCE", Prentice-Hall, 1992.
- [3] A.S. Tanenbaum, "Operating Systems, Design and Implementation", Prentice-Hall, 1987.
- [4] Doug Smith, "An Overview of CORBA", Washington University.
- [5] T.J. Mowbray, R. Zahavi, "The Essential CORBA: Systems Integration Using Distributed Objects", Wiley/OMG, 1995.
- [6] IBM, "AIX CIC/6000 Customization and Operation", International Business Machines Corporation, September, 1993.
- [7] ISO/IEC JTC1/SC21/WG7 N885, "Reference Model for Open Distributed Processing - Part 1: Overview and Guide to Use", November 1993.
- [8] ISO/IEC 10746-2, "Basic Reference Model of Open Distributed Processing - Part 2: Descriptive Model", July 1993.
- [9] ISO/IEC 10746-3, "Basic Reference Model of Open Distributed Processing - Part3: Prescriptive Model", July 1993.
- [10] K.A. Raymond, "Reference Model of Open Distributed Processing: a Tutorial" *Open Distributed Processing, II (C-20)*

- [11] I. Jacobson, "Object-Oriented Software Engineering", Addison-Wesley, 1992.
- [12] R. Buhr, 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.
- [13] C.U. Smith, "Performance Engineering of Software Systems", Addison-Wesley, August 1990.
- [14] J.A. Rolia, "Predicting the Performance of Software Servers", *CSRI Technical Report 260*, University of Toronto, Canada, January 1992.
- [15] J.A. Rolia, K.C. Sevcik, "The Method of Layers", I.E.E.E. Transactions on Software Engineering, Vol. 21, p. 689-700, August, 1995.
- [16] G. Franks, A. Hubbard, S. Majumdar, D. Petriu, J. Rolia, C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", Performance Evaluation Journal, Vol. 24, p. 117-136, North Holland, November 1995.
- [17] SES, "SES/Workbench User Manual", Scientific and Engineering Software, Inc., Austin, Texas, 1995.
- [18] Data Management Review, "Review of SES/Workbench", Powell Publishing Inc., September, 1995.
- [19] J.E. Neilson, "PARASOL Users' Manual", School of Computer Science.
- [20] G. Balbo, G. Serazzi, "Asymptotic Analysis of Multiclass Closed Queuing Networks: Common Bottlenecks" Performance Evaluation Journal, North Holland, 1995
- [21] G. Balbo, G. Serazzi, "Asymptotic Analysis of Multiclass Closed Queuing Networks: Multiple Bottlenecks", *Technical Report 93-094*, Politecnico di Milano, EECS Dept., 1993.

- [22] E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik, "Quantitative System Performance, Computer System Analysis Using Queueing Network Models". Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
- [23] M. Reiser, S.S. Lavenburg, "Mean Value Analysis of Closed Multichain Queueing Networks", IBM Research Report RC 70 23, Yorktown Heights, N.Y., 1978.
- [24] G. Hills, J. Rolia, G. Serazzi, "Performance Engineering of Distributed Software Process Architectures", Lecture Notes in Computer Science, Springer Verlag, No. 977, pages 79-85, 1995.
- [25] 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.
- [26] C.E. Hrischuk, J. Rolia, C.M. Woodside, "Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype", Proceedings of International Workshop on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95), p. 399 - 409, Durham, North California, January 1995.
- [27] M.D. Hill, J.R. Larus, A.R. Lebeck, M. Talluri, D.A. Wood, "Wisconsin Architectural Research Tool Set", Computer Sciences Department, University of Wisconsin, 1995.
- [28] C.M. Woodside, S. Majumdar, "Robust Bounds and Throughput Guarantees for General Closed Multiclass Queueing Networks", *SCE Technical Report SCE-94-05*, Carleton University, Ottawa, Canada, January 1994.
- [29] J.E. Neilson, C.M. Woodside, D.C. Petriu, 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.

- [30] E. Pozzetti, V. Vetland, J.A. Rolia, 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.

Glossary

ABA : Asymptotic Bounds Analysis

AMVA : Approximate Mean Value Analysis

BEO : Basic Engineering Object

CDS : Cell Directory Service

CICS : Customer Information Control System

CF : Canadian Forces

CFD : Canadian Forces Depot

CFSSU : Canadian Forces Supply System Upgrade

CORBA : Common Object Request Broker Architecture

COTS : Commercial off the Shelf

DASS : Distributed Application System Sizer

DCE : Distributed Computing Environment

DRP : Distribution and Requirements Planning

IDL : Interface Definition Language

LP : Local Procurement

LQM : Layered Queueing Model

MOL : Method of Layers

MVA : Mean Value Analysis

NDHQ : National Defence Headquarters

ODP : Open Distributed Processing

OMG : Object Management Group

ORB : Object Request Broker

OSF : Open Software Foundation

PE : Performance Engineering

PMO : Project Management Office

QNM : Queueing Network Model

RM-ODP : Reference Model of Open Distributed Computing

RPC : Remote Procedure Call

SHL : SystemHouse LTD.

SPE : Software Performance Engineering

SRVN : Stochastic Rendezvous Networks

WDC : Warehousing and Distribution Centre

END

2 1 - 1 1 - 9 6

FIN