"Load Management Systems for Component-based Middleware"

Octavian Ciuhandu, B.E.

Masters of Engineering in Electronic Engineering

Dublin City University

Supervisor by: Dr. John Murphy

School of Electronic Engineering

May, 2004

Acknowledgements

Primary thanks to my parents for their continuous and priceless support.

Thanks go out to all my friends who helped and supported me throughout these years. Special thanks to Adi, Ada, Carmen, Doru and Val.

I would also like to thank my supervisor, Dr. John Murphy without whom this work could not be have been finished.

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Engineering in Electronic Engineering is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____ (Candidate) ID No.: 51166780

Petre Octavian Ciuhandu Date: May 19th, 2004

Table of contents

List of figures v

List of tables vii

1 Introduction 1
- 1.1 Performance of Distributed Systems . . . 2
- 1.2 Technical Approach . . . 2
- 1.3 Thesis Outline . . . 3

2 Introduction to Distributed Systems 5
- 2.1 Distributed Systems Communication Models . . . 9
- 2.2 Characteristics of Middleware-based Systems . . . 11
- 2.3 Component-based Distributed Systems . . . 12
- 2.4 Performance of Component-based Distributed Systems . . . 16

3 Literature Survey 19
- 3.1 Introduction . . . 19
- 3.2 Distributed Object Computing Middleware Layers . . . 19
- 3.3 Load Distribution for Distributed Systems . . . 26
  - 3.3.1 Load Management Services Characteristics . . . 26
  - 3.3.2 Load Management Services Classification . . . 28
  - 3.3.3 Load Management Services Requirements . . . 30
  - 3.3.4 Load Management Services Components . . . 34
  - 3.3.5 Load Management Services Design Challenges . . . 35
- 3.4 Quality Of Service for Distributed Systems . . . 37
  - 3.4.1 Static QoS Limitations . . . 39
  - 3.4.2 New QoS Techniques . . . 39
  - 3.4.3 Related QoS Research . . . 41

4 Problems in Existing Load Management Systems 43
- 4.1 Introduction . . . 43
- 4.2 Load Management-Related Issues . . . 43
- 4.3 QoS-Related Problems . . . 46

5 A New Load Management Framework 48
- 5.1 Introduction . . . 48
- 5.2 The Framework . . . 48
- 5.3 The Load Monitoring Module . . . 52
- 5.4 The Load Prediction Module . . . 53

5.5 The QoS Control Module 54
5.6 The Load Evaluator Module 55
5.7 The Load Distribution Module 57
5.8 Characteristics of the QoS Enabled Load Management Service 58

6 Application Server Simulation Models **62**
6.1 Introduction 62
6.2 Hyperformix WorkBench Modelling and Simulation Environment 63
6.3 Single Server Environment 66
6.3.1 Introduction 66
6.3.2 The Model 67
6.4 Multiple Server Environment Simulation 75
6.4.1 Introduction 75
6.4.2 The Model 76
6.4.3 Round Robin Request Distribution 78
6.4.4 Weighted Round Robin Request Distribution . . . 79
6.4.5 Load Balanced Request Distribution 80
6.4.6 Introduction of Service Levels Request Distribution 81
6.5 Two-Layered Algorithm for Service Levels Model 84
6.5.1 The Approach 84
6.5.2 The Implementation 85

7 Simulation Model Results **89**
7.1 Introduction 89
7.2 Simulation Methodology 89
7.3 Evaluation of the Load Distribution Algorithms 90
7.4 Service Levels Simulation Results 91
7.5 Two-Layered Algorithm Simulation Results 95
7.5.1 Round Robin Algorithm vs. Two Layered Load Balanced Algorithm, Without Priorities 95
7.5.2 Round Robin Algorithm vs. Two Layered Load Balanced Algorithm, With 10 Percent High Priority Transactions and Weight 5 99
7.5.3 Round Robin Algorithm vs. Two Layered Load Balanced Algorithm, With 10 Percent High Priority Transactions and Weight 10 102
7.5.4 Round Robin Algorithm vs. Two Layered Load Balanced Algorithm, Without Priorities, adjusting service time 106
7.5.5 Round Robin vs. Two Layered Load Balanced With Priorities, adjusting service time 112
7.5.6 Round Robin vs. Two Layered Load Balanced With Priorities, Balanced Load 118
7.5.7 Round Robin vs. Two Layered Load Balanced With Priorities, Load Influence 123
7.5.8 Conclusions 130

8 Conclusions **133**
8.1 Future Work 137

List of figures

2.1 A General Distributed System. 6
2.2 A middleware overview. 8

3.1 Distributed Object Computing Middleware Layers. [60] . . 20
3.2 Load Management System Architecture. 34

5.1 The Proposed Framework. 49
5.2 Two separate clusters load-managed. 51
5.3 Two clusters load-managed, sharing resources. 52
5.4 Failover Protection Operations (1). 59
5.5 Failover Protection Operations (2). 60

6.1 Available WorkBench Nodes. 63
6.2 Simulation Model Overview. 68
6.3 Workload Module. 69
6.4 BrowseCar Module. 70
6.5 BuyCar Module. 71
6.6 EntityBean Module. 72
6.7 SessionBean Module. 74
6.8 Database Module. 74
6.9 System Module. 75
6.10 Simulation Model for Multiple Servers Scenario. 76
6.11 Reviewed Simulation Model. 85
6.12 Reviewed Simulation Model, Application Server Logic. . . 85
6.13 Reviewed Simulation Model, Database Server Logic. 86

7.1 Average response time evolution for different distribution algorithms. 91
7.2 Average response time evolution for separate server vs. different weights. 93
7.3 Round Robin Average Response Time, no priorities 96
7.4 Load Balanced Average Response Time, no priorities . . . 98
7.5 Round Robin Average Response Time, 10 percent high priority transactions, Weight 5 99
7.6 Load Balanced Average Response Time, 10 percent high priority transactions, Weight 5 101
7.7 Round Robin Average Response Time, 10 percent high priority transactions, Weight 10 104
7.8 Load Balanced Average Response Time, 10 percent high priority transactions, Weight 10 105

7.9 Round Robin Average Response Time. 107
7.10 Load Balanced Average Response Time V1. 109
7.11 Load Balanced Average Response Time V2. 111
7.12 Round Robin Average Response Time. 114
7.13 Load Balanced Average Response Time, Weight 5. 115
7.14 Load Balanced Average Response Time, Weight 10, V5. . . 117
7.15 Round Robin Average Response Time, Uniform Load. . . . 118
7.16 Load Balanced Average Response Time, Uniform Load. . . 121
7.17 Load Balanced Average Response Time, Uniform Load, With decision overhead. 122
7.18 Round Robin vs. Load Balanced Average Response Time, Uniform Load 124
7.19 Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,1) 125
7.20 Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,2) 126
7.21 Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,5) 128
7.22 Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,10) 130
7.23 Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,20) 131

8.1 Initial System Configuration. 134
8.2 QoS-Enabled System Configuration. 134
8.3 Large System Configuration. 135

List of tables

7.1 Algorithm Response Time Comparison 91
7.2 Separate vs. Shared Server Weight Influence 92
7.3 Separate vs. Shared Server High Priority Percentage Influence 94
7.4 Round Robin Server Average Response Time, no priorities 96
7.5 Load Balanced Server Average Response Time, no priorities 97
7.6 Number of Transactions Processed, Round Robin Case, no priorities 97
7.7 Number of Transactions Processed, Load Balanced Case, no priorities 97
7.8 Round Robin Server Average Response Time, Weight 5 . . 100
7.9 Load Balanced Server Average Response Time, Weight 5 . 100
7.10 Number of Transactions Processed, Round Robin Case, Weight 5 100
7.11 Number of Transactions Processed, Load Balanced Case, Weight 5 100
7.12 Round Robin Server Average Response Time, Weight 10 . 103
7.13 Load Balanced Server Average Response Time, Weight 10 103
7.14 Number of Transactions Processed, Round Robin Case, Weight 10 103
7.15 Number of Transactions Processed, Load Balanced Case, Weight 10 103
7.16 Round Robin Server Average Utilization, no priorities, adaptive 107
7.17 Round Robin Server Average Response Time, adaptive, no priorities 108
7.18 Load Balanced Server Average Utilization, no priorities, adaptive 109
7.19 Load Balanced Server Average Response Time, adaptive, no priorities 110
7.20 Number of Transactions Processed, load balanced, adaptive, no priorities 110
7.21 Load Balanced Server Average Utilization, no priorities, adaptive 110
7.22 Load Balanced Server Average Response Time, adaptive, no priorities 111
7.23 Number of Transactions Processed, load balanced, adaptive, no priorities 112

7.24 Round Robin Server Average Response Time, adaptive, 15 percent high priority 113
7.25 Round Robin Server Average Utilization, adaptive, 15 percent high priority 113
7.26 Number of Transactions Processed, Round Robin, adaptive, 15 percent high priority transactions 113
7.27 Load Balanced Server Average Response Time, adaptive, 15 percent high priority, weight 5 114
7.28 Load Balanced Server Average Utilization, adaptive, 15 percent high priority, weight 5 115
7.29 Number of Transactions Processed, Round Robin, adaptive, 15 percent high priority transactions 116
7.30 Load Balanced Server Average Response Time, adaptive, 15 percent high priority, weight 10 116
7.31 Load Balanced Server Average Utilization, adaptive, 15 percent high priority, weight 10 116
7.32 Number of Transactions Processed, Round Robin, adaptive, 15 percent high priority transactions 117
7.33 Round Robin Server Average Response Time 119
7.34 Round Robin Server Average Utilization 119
7.35 Number of Transactions Processed, Round Robin 119
7.36 Load Balanced Server Average Response Time, No Decision Overhead 120
7.37 Load Balanced Server Average Utilization, No Decision Overhead 120
7.38 Number of Transactions Processed, Load Balanced, No Decision Overhead 120
7.39 Load Balanced Server Average Response Time, With Decision Overhead 122
7.40 Load Balanced Server Average Utilization, With Decision Overhead 123
7.41 Number of Transactions Processed, Load Balanced, With Decision Overhead 123
7.42 Server Average Response Time, Lognormal(05,01) 124
7.43 Server Average Utilization, Lognormal(05,01) 125
7.44 Number of Transactions Processed, Lognormal(05,01) . . . 126
7.45 Server Average Response Time, Lognormal(05,02) 127
7.46 Server Average Utilization, Lognormal(05,02) 127
7.47 Number of Transactions Processed, Lognormal(05,02) . . . 127
7.48 Server Average Response Time, Lognormal(05,05) 128
7.49 Server Average Utilization, Lognormal(05,05) 129
7.50 Number of Transactions Processed, Lognormal(05,05) . . . 129
7.51 Server Average Response Time, Lognormal(05,20) 130
7.52 Server Average Utilization, Lognormal(05,20) 131
7.53 Number of Transactions Processed, Lognormal(05,20) . . . 132

Abstract

This thesis proposes a new approach to the design of reflective load management services for middleware, tackling the main problems in existing load management services. The system is designed using a modular architecture. The two key benefits of the approach are that modules can be dynamically activated and deactivated as required, enabling the minimisation of the overhead introduced by the system, and that new modules with enhanced functionality can easily (and dynamically) be introduced into the system.

The system comprises of a load monitoring module, a load prediction module, a load evaluation module and a load distribution module. Each module has a clearly defined role in the system and a well-defined interface. The load evaluation module offers the possibility of dynamically changing the distribution algorithm. The modularity of the system is further extended to the monitoring, workload prediction and load distribution components, so that new monitors and algorithms can be added at runtime. A novelty of the proposed approach consists in the inclusion of QoS in the load management system, thus making it transparent to the managed applications. This approach offers increased flexibility and reusability because QoS can be added to existing (non QoS-aware) applications without the hurdle of changing the code. The response time metric is used for QoS level differentiation. An important characteristic of our load management service is that it is transparent to distributed application developers. The design of the load management system ensures high availability by including a simple load distribution mechanism in the distribution module.

Another novelty of this approach is the automatic selection of the optimal load distribution algorithm at runtime, according to current system state and workload. It is considered that the most important performance metric for system performance is the response time. An important achievement of such a reflective load management service is that it adapts itself at runtime to workload/environment changes without user intervention.

A simulation model was created to evaluate existing load distribution algorithms. The model was extended to offer simulation scalability (e.g. the number of servers can be easily changed) and to support the evaluation of the newly proposed load distribution algorithm. The influence of the workload on the performance of the distribution algorithm was also investigated.

The research approach employed carrying out an extensive literature survey in order to identify the main problems in existing load management services. These problems represented the ground for the framework proposed in this thesis. It is beyond the scope of this thesis to validate the entire framework thus only the key elements of the framework have been investigated in detail and validated using simulations.

1.1 Performance of Distributed Systems

Two definitions for the performance in the context of software systems can be found. On one hand, it denotes the speed at which a computer operates, either theoretically (e.g. using a formula for calculating number of theoretical instructions per time unit) or by counting operations or instructions performed (e.g. millions of instructions per second) during a benchmark test. On the other hand it also denotes the total effectiveness of a computer system (i.e. throughput, individual response time, availability). In this thesis the second definition of system performance is used.

While in the past performance problems were typically addressed by throwing in hardware upgrades, the last years forced a dramatic change in the approach for dealing with performance problems due to high pressures for costs reduction, and observations that this approach cannot guarantee that it will solve complex systems performance problems.

Due to the continuous increase in the number of users and in their performance expectations, new distributed systems require a broad range of features, such as service guarantees, dependability, predictable performance, secure operation and fault tolerance.

Most existing workload management services fail to meet these requirements, mostly due to the substantial amount of recent technological developments in this domain (of component-based distributed systems).

1.2 Technical Approach

The thesis proposes a new approach to the design of workload management services for component-based distributed systems, such as EJB. A workload management service is a service that, considering the existing hardware (available servers) and the distributed application, must dis-

tribute the incoming workload to obtain maximum performance.

The proposed workload management system is reflective, i.e. it adapts itself to runtime environment changes, and is based on a modular architecture. The reasoning behind selecting a modular architecture is the important requirement of minimising the computing and communications overhead introduced by the workload management system.

A simulation environment was created for testing the most important features of the proposed workload management service. The simulation models are created using a general-purpose modelling and simulation tool, since this approach offers the possibility of early exploration of the solution space.

1.3 Thesis Outline

Chapter two introduces the area of research, presenting the distributed systems concepts and some basic terminology used in the following chapters. Starting with a general description of what distributed systems are, we continue by narrowing the domain to component-based distributed systems, since it is the target domain of our research. We conclude this chapter by introducing performance terminology in the context of our research.

Chapter three provides a review of the current state of the art in the domain. It starts with a detailed presentation of the existing middleware layers. The rest of the chapter is divided into two main parts, the load distribution and the QoS related work and problems. The first part details the characteristics, classification, requirements and problems for the load management services, giving also an overview of the components and design challenges for these services. The latter part discusses issues associated with delivering end-to-end QoS to the users of the system and

presents newly proposed QoS techniques and related research.

Chapter four details the identified performance problems for component-based distributed systems, providing the reasoning for this research. The existing research mentioned in the previous chapter is analyzed and the differences between previous proposals and the new approach are discussed.

Chapter five presents the proposed framework. All framework components are presented in detail with the interfaces and relationships between them. The functionality of the framework and its characteristics are also detailed.

Chapter six introduces the work done for demonstrating the need for the proposed framework and for evaluating the performance improvements the framework can offer. Validating the entire framework involves a considerable effort and it is not feasible to achieve this in this single thesis. However, the most important aspects of the framework are examined through simulations. The reasons for choosing a simulation environment rather than an implementation one are presented together with the simulation models used for achieving these goals. The load distribution algorithms evaluated are presented in detail as well as the proposed approach for the introduction of service levels.

Chapter seven presents in detail the results obtained using the simulation models described in chapter six. These results are analysed and discussed and the merits of the different approaches to workload management under different conditions determined.

Chapter eight summarizes the work. A summary of the obtained results is presented as well as the conclusions for the framework. Possible future work is also outlined, presenting other observed problems the proposed framework can address as well as proposed solutions for solving them.

Chapter 2

Introduction to Distributed Systems

With the increasing availability and affordability of network devices and internet services, more and more machines and systems are becoming interconnected. As a result, users rely on information and services available from other machines, with which they are connected, rather than having everything stored and available locally. This leads to an increasing number of people accessing different network-based services. The required infrastructure is realized using different types of distributed systems.

Distributed Systems are composed out of networked processing units and other devices, cooperating in order to provide a (set of) required service(s). An office workstation is usually connected to different distributed systems, like file servers, database servers, printers, backup devices and World Wide Web facilities. A general distributed system is presented in Figure 2.1.

While the hardware of a distributed system might be considered important, it is software that largely determines its characteristics. Distributed systems have some similarities with traditional operating systems since they act as resource managers for the underlying hardware, allowing mul-

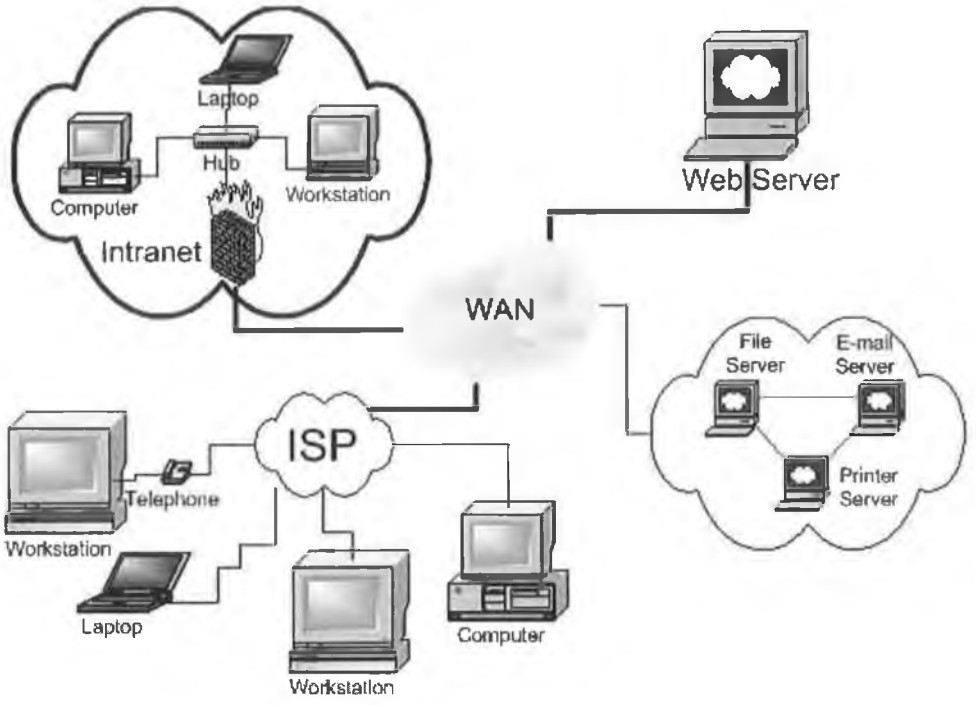

Figure 2.1: A General Distributed System.

tiple users and applications to share resources (such as CPUs, memories, peripheral devices, the network, and data of all kinds). Distributed systems hide the heterogeneous nature of the underlying hardware by providing a virtual machine on which applications can be easily executed. Operating systems for distributed computing systems can be roughly divided into two categories: loosely-coupled systems and tightly-coupled systems. While loosely-coupled systems can be considered a collection of computers, each running their own operating system, for tightly-coupled systems the operating system essentially tries to maintain a single, global view of the resources it manages. However, these operating systems work together to make their own services and resources available to the others [46].

Loosely-coupled *network operating systems* (NOS) are used for heterogeneous multicomputer systems. NOSs implement protocol stacks as well as device drivers for networking hardware. Although managing the

underlying hardware is an important issue for a NOS, the distinction from traditional operating systems comes from the fact that local services are made available to remote clients. While scalability and openness represent the main advantages of these operating systems, their disadvantage is that they do not provide a view of a single and coherent system [46].

A tightly coupled operating system is generally referred to as a *distributed operating system* (DOS) and is used for managing multiprocessors and homogeneous multicomputers. In comparison with NOSs, DOSs are modular, extensible and strive for a high degree of transparency and often support data and process migration. The main difference between the two is that DOSs support a transparent view of the entire network, in which users normally do not distinguish local resources from remote resources [46]. Like traditional uniprocessor operating systems, the main goal of a distributed operating system is to hide the intricacies of managing the underlying hardware such that it can be shared by multiple processes. The drawback of this type of operating systems is that they are not intended for handling a collection of independent computers.

The question comes to mind whether it is possible to develop a distributed system that has the best of both worlds: the scalability and openness of NOSs and the transparency and related ease of use of DOSs. The answer is the introduction into a DOS of an additional layer of software , used in some NOSs to hide the heterogeneity of the collection of underlying platforms and to improve distribution transparency. Many modern distributed systems are constructed by means of such an additional layer of what is called middleware. Middleware is a software that facilitates interoperability by mediating between an application program and a network, thus masking differences or incompatibilities in network transport protocols, hardware architecture, operating systems, database systems, remote procedure calls, etc [46].

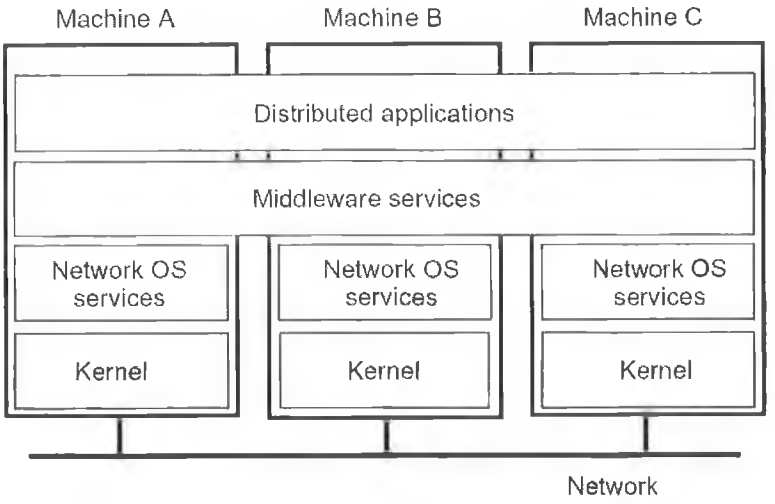

Figure 2.2: A middleware overview.

Middleware itself does not manage an individual node, as seen in Figure 2.2. Each local system forming part of the underlying NOS is assumed to provide local resource management in addition to simple communication means to connect to other computers.

An important goal is to hide the heterogeneity of the underlying platforms from applications, many middleware systems offering a more-or-less complete collection of services and discouraging the use of anything else but their interfaces for those services.

Distributed systems vary in their degree of centralization [46]. Traditionally distributed systems tend to be fairly centralized. The server/client paradigm is the prime example. All essential computation and all services are placed on the server; the client is little more than an input/output device, providing mainly the GUI. Another example are server clusters which typically rely on centralized coordination, the fact that all machines reside within one administrative domain, uniformity of connection (e.g. the same latency between all participants), centralized maintenance (e.g. code upgrading), monitoring (e.g. failure-detection) and control (e.g. the ability to stop a runaway processor).

The contrast to the traditional centralized approach is fully decentralized systems, recently popularized as peer-to-peer systems [55]. With the improvement of technology for dealing with fully decentralized systems, the door for various hybrid solutions where the simplicity of centralized control is optimally mixed with the scalability of decentralization is opened.

2.1 Distributed Systems Communication Models

To make development and integration of distributed applications as simple as possible, most middleware is based on some model, or paradigm, for facilitating distribution and communication. Examples of middleware models are the file model, distributed file systems model and RPC model.

A relatively simple model is treating everything as a file [46]. All resources, including I/O devices such as keyboard, mouse, disk, network interface, and so on, are treated as files. Essentially, whether a file is local or remote makes no difference. An application opens a file, reads and writes bytes, and closes it again. Because files can be shared by several processes, communication reduces to simply accessing the same file.

A similar approach is followed by middleware centered around distributed file systems. In many cases, such middleware is actually only one step beyond a network operating system in the sense that distribution transparency is supported only for traditional files (i.e., files that are used for merely storing data, as opposed to the first model). For example, processes are often required to be started explicitly on specific machines. Nevertheless, middleware based on distributed file systems has proven to be reasonable scalable, which contributes to its popularity [46].

Another important middleware model is the one based on *Remote*

Procedure Calls (RPCs) [46]. In this model, the emphasis is on abstracting network communication by allowing a process to call a procedure of which an implementation is located on a remote machine. When calling such a procedure, parameters are transparently shipped to the remote machine where the procedure is subsequently executed, after which the results are sent back to the caller. It therefore appears as if the procedure call was executed locally: the calling process remains unaware of the fact that network communication took place, except perhaps for some loss of performance.

As object orientation techniques came into vogue, it became apparent that if procedure calls could cross machine boundaries, it should also be possible to invoke methods of objects residing on remote machines, in a transparent fashion. This has now led to various middleware systems offering a notion of distributed objects. The essence of distributed objects is that each object implements an interface that hides all the internal details of the object from its users. An interface consists of the methods that the object implements, no more and no less. The only thing that a process sees of an object is its interface.

Distributed objects are often implemented by having each object itself located on a single machine, and additionally making its interface available on many other machines [46]. When a process invokes a method, the interface implementation on the process's machine simply transforms the method invocation into a message that is sent to the object. The object executes the requested method and sends back the result. The interface implementation subsequently transforms the reply message into a return value, which is then handed over to the invoking process. As in the case of RPC, the process may be kept completely unaware of the network communication.

2.2 Characteristics of Middleware-based Systems

There are a number of services common to many middleware systems. Invariably, all middleware, one way or another, attempts to implement access transparency, by offering high-level communication facilities that hide the low-level message passing through computer networks. The programming interface to the transport layer as offered by network operating systems is thus entirely replaced by other facilities, using a higher abstraction level. How communication is supported depends very much on the model of distribution the middleware offers to users and applications. RPCs and distributed-object invocations are examples of higher abstraction level communications. In addition, many middleware systems provide facilities for transparent access to remote data, such as distributed file systems or distributed databases. Besides RPCs and distributed-object invocations, transparently fetching documents as is done in the Web is another example of high-level (one-way) communication.

An important service common to all middleware is that of naming. Name services allow entities to be shared and looked up (as in directories), and are comparable to telephone books and the yellow pages. Although naming may seem simple at first, difficulties can arise when scalability is taken into account. Problems are caused by the fact that to efficiently look up a name in a large-scale system, the location of the entity that is named must be assumed to be fixed. This assumption is made in the World Wide Web, in which each document is currently named by means of a *Universal Resource Locator* (URL). A URL contains the name of the server where the document to which the URL refers is stored. Therefore, if the document is moved to another server, its URL ceases to work.

Many middleware systems offer special facilities for storage, also re-

ferred to as persistence. In its simplest form, persistence is offered through a distributed file system, but more advanced middleware have integrated databases into their systems, or provide facilities for applications to connect to databases.

In environments where data storage plays an important role, facilities are generally offered for distributed transactions. An important property of a transaction is that it allows multiple read and write operations to occur atomically. Atomicity means that the transaction either succeeds, so that all its write operations are actually performed, or it fails, leaving all referenced data unaffected. Distributed transactions operate on data that can be spread across multiple machines. Especially in the face of masking failures, which is often hard in distributed systems, it is important to offer services such as distributed transactions. Unfortunately, transactions are hard to scale across many machines.

Finally, virtually all middleware systems that are used in non-experimental environments provide facilities for security. Compared to network operating systems, the problem with security in middleware is that it should be pervasive. In principle, the middleware layer cannot rely on the underlying local operating systems to adequately support security for the complete network. Consequently, security has to be partly implemented in the middleware layer itself. Security has turned out to be one of the hardest services to implement in distributed systems, due to the extensibility requirements of middleware systems.

2.3 Component-based Distributed Systems

One frequent question is what is the rationale behind component software. Traditionally, closed solutions with proprietary interfaces addressed most customers' needs. Heavyweights such as operating systems and database

engines are among the few examples of components that did reach high levels of maturity. Large software systems manufacturers often configure delivered solutions by combining modules in a client-specific way. However, the interfaces between such modules tend to be proprietary, at most open to highly specialized independent software vendors (ISVs) that specifically produce further modules for such systems. In many cases, these modules are fused together during a linking step and are no longer distinguishable in deployed solutions.

In many current approaches, components are heavyweights. For example, a database server could be a component. If only one database exists, it is easy to confuse the instance with the concept. For example, the database server might be seen together with the database as a component with persistent state. This instance of the database concept is not an actual component. Instead, the static database server program is a component, and it supports a single instance: the database object. This separation of the immutable level from the mutable instances is the key to avoiding massive maintenance problems. If components could be mutable, that is, have state, then no two installations of the same component would have the same properties. The differentiation of components and objects is thus fundamentally about differentiating between static properties that hold for a particular configuration and dynamic properties of any particular computational scenario.

Distributed object computing extends an object-oriented programming system by allowing objects to be distributed across a heterogeneous network, so that each of these distributed object components interoperate as a unified whole. These objects may be distributed on different computers throughout a network, having their own address space outside of an application, and yet appear as though they were local to an application.

While objects have been around for some time, components are on

the upswing. Often object-oriented programming is considered to be sold in new clothes by simply calling objects "components." The emerging component-based approaches and tools combine objects and components in ways that show they are really separate concepts.

Most standard-driven approaches originate in industry consortia. The prime example here is the effort of the *Object Management Group* (OMG). However, the OMG hasn't contributed much in the component world and is now falling back on JavaSoft's *Enterprise JavaBeans* (EJB) standards for components, although it's attempting a CORBA Beans generalization. The EJB standard still has a long way to go; so far it is not implementation language-neutral, and bridging standards to Java external services and components are only emerging.

The separate existence and mobility of components [13], as witnessed by Java applets or ActiveX components, can make components look similar to objects. Often the words "component" and "object" are used interchangeably. Constructions such as "component object" are used as well. Objects are said to be instances of classes or clones of prototype objects. Objects and components both make their services available through interfaces. Language designers add further complexity by discussing namespaces, modules, packages, and so on.

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [64].

A component must be independently deployable and thus it needs to be well-separated from its environment and from other components. Therefore it must encapsulate its constituent features. Also, since it is a unit of deployment, a component can never be partially deployed.

If a third party needs to compose a component with other components,

the component must be self-contained. (A third party is one that cannot be expected to access the construction details of all the components involved.) Also, the component needs to come with clear specifications of what it provides and what it requires. In other words, a component needs to encapsulate its implementation and interact with its environment through well-defined interfaces and platform assumptions only. It's also generally useful to minimize hard-wired dependencies in favor of externally configurable providers.

Finally, a component without any persistent state cannot be distinguished from copies of its own. (Exceptions to this rule are attributes not contributing to the component's functionality, such as serial numbers used for accounting.) Without state, a component can be loaded into and activated in a particular system-but in any given process, there will be at most one copy of a particular component. So, while it is useful to ask whether a particular component is available or not, it isn't useful to ask about the number of copies of that component. A component may simultaneously exist in different versions. However, these are not copies of a component, but rather different components related to each other by a versioning scheme.

Contextual component frameworks, such as *Enterprise Java Beans* (EJB), have emerged out of the need to facilitate the development of easily evolvable and modifiable enterprise applications. Dynamic recomposition is the ground on which these features are being built. Applications can be built using third-party components and deployed on third-party platforms. Companies increasingly rely on component-oriented technologies, such as EJB [48] and *Commercial Off-The-Shelf* (COTS) components, in order to build large scale applications, reduce system development costs and capitalize on third party expertise.

2.4 Performance of Component-based Distributed Systems

Performance problems have been typically addressed by addressed by adding more resources, especially as systems got smaller and cheaper. The performance problem has suffered dramatic changes in the last few years and this not only because systems' complexity increased as the world we live in becomes' a web-based world. This approach is not valid anymore due to the increased pressure on costs reduction, which lead to better profit and businesses that survive.

The response times of an application can not be considered good or bad unless unless compared against a standard, eg setting a targeted response time, throughput, latency or other performance indicators.

That process is an art, according to [26]. "It all gets into service-level agreements, he says, because if you don't have some idea of what performance you expect, then when a user says things are too slow, you'll say, 'Well, what's too slow?' and then..."

In the "new world", system management services must be able, beside managing systems, to prioritize applications, spread the workload across a number of servers.

Software performance is the process of predicting (at early phases of application lifecycle) and evaluating (at the end) whether the software system satisfies the user performance goals. This process is based on the availability of software artifacts describing suitable abstractions of the final system. Examples of such artifacts are identified requirements, proposed software architectures, design documents and specifications.

Different approaches exist for integrating the performance analysis and prediction into the software development cycle, targeting various stages of the software system development process. Examples are Architec-

ture Tradeoff Analysis [9] [43], Performance-Critical Systems or COTS Based Systems initiatives (Software Engineering Institute initiatives [47]), J2EE [21] and general [37] design patterns-based approaches [50] or anti-patterns based designs, prototyping and trace-analysis based approaches [12], UML-based performance modelling and analysis efforts like OMG UML Profile, for Schedulability, Performance, and Time [5] [39] [17] and the Software Performance Engineering (SPE) initiative [14] [44].

Since the performance is a runtime attribute, suitable descriptions of the software's runtime behavior are required for performance analysis.

System performance at runtime is being extensively researched, the important directions being:

- **performance evaluation**, using system and application monitoring (use of monitors to collect data and detect existing bottlenecks). In the case of complex, large-scale systems, developed based on contextual component frameworks, runtime is the only place where the performance of the system can be evaluated.

- **performance optimization**, which implies runtime system changes. Some means of achieving runtime optimization are load distribution, component instances migration and replication, caching (at different system levels) and pooling, as well as adaptive, reflective, self-optimizing, self-repairing and evolving systems.

- **performance prediction**, based on recorded monitoring information

A systems' performance can be analyzed at design time, at deployment time, or be left for optimizations at runtime. In general, early performance analysis leads to better results [18] but it also implies that some overall system knowledge is available. For many component-based systems this is not the case since knowledge about components interconnections, patterns

of communication and underlying platform is available only at runtime. These conditions are generally not applicable to any approach involving the use of a component framework in which binding occurs late. Current efforts [4] employing traditional performance analysis methods, such as LQN [65] on EJB systems, make the assumption that system structure is known. The same limitations apply to deployment time optimizations. All that is known are type dependencies, that could lead to some optimisations, such as co-locating type dependent implementations. Furthermore, if the system utilization patterns change, deployment time optimizations might fail to deliver any performance improvement [45].

For a component-based system, previously mentioned conditions include knowledge about components' interconnections, patterns of communication, and underlying platform. In an EJB system, neither are completely known until runtime.

Considering the issues noted above, it can be inferred that EJB systems, due to their dynamic and unpredictably evolving nature, may benefit less from early-design approaches and more from runtime optimizations.

Chapter 3

Literature Survey

3.1 Introduction

In this chapter we present the relevant work being carried out in component-based distributed systems load management area. An introduction to Distributed Object Computing Middleware with its layers is presented first. Next, existing load distribution approaches and existing work concerning Quality of Service (QoS) for Component-based Distributed Systems will be presented. The two apparently separate areas are being presented since it is considered that grouping them together can lead to good performance improvements.

3.2 Distributed Object Computing Middleware Layers

Distributed object computing (DOC) middleware architectures are composed of relatively autonomous software objects that can be either collocated (i.e. in the same room/builging, connected using high-speed networks) or distributed throughout a wide range of networks and interconnects [60]. The clients invoke operations on target objects for performing

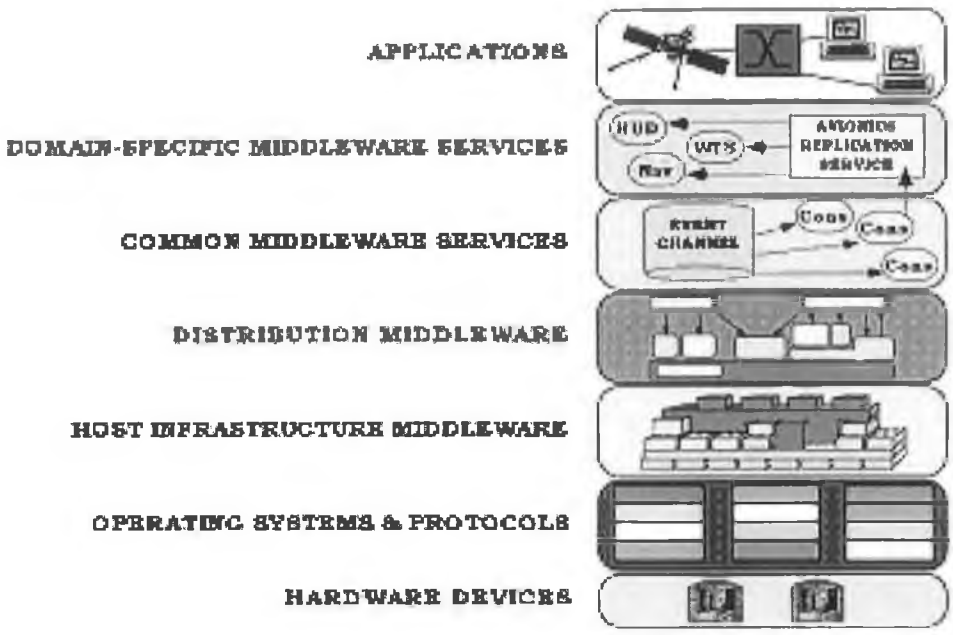

Figure 3.1: Distributed Object Computing Middleware Layers. [60]

interactions and invoking the functionality needed to achieve the application's goals. The aggregation of these simple, middleware-mediated interactions forms the base of large-scale distributed systems deployment [59].

Distributed object computing middleware can be decomposed into multiple layers [60], as shown in Figure 3.1:

- **Host infrastructure middleware**: enhances and encapsulates native operating system communication and concurrency mechanisms
- **Distribution middleware**: defines higher level distributed programming models
- **Common middleware services**: services define a high-level, domain-independent services
- **Domain-specific middleware**: define services tailored to the requirements of particular domains

```
**********************************************************************
User name:   MunteanG (111)              Queue:  EE_OLYMPUS/C230_LASER
File name:                               Server: C230_LASER
Directory:
Description: Octavian-Thesis.pdf
             May 24, 2004                                  12:55pm
**********************************************************************

              M   M                t                          GGGG
              MM MM                t                        G
              M M M u   u nnnn  tttt   eee   aaa  nnnn  G
              M M M u   u n   n  t    e   e     a n   n G GGG
              M   M u   u n   n  t    eeeee  aaaa n   n G   G
              M   M u   u n   n  t  t e     a   a n   n G   G
              M   M  uuuu n   n   tt   eeee  aaaa n   n  GGGG


**********************************************************************

**********************************************************************
```

The ***host infrastructure middleware*** enhances and encapsulates native operating system communication and concurrency mechanisms for creating reusable components [60], such as acceptor-connectors, monitor objects and component configurators. Examples of infrastructure middleware are:

- ***Sun's Java Virtual Machine (JVM)*** [42] provides a platform-independent way of executing code. It abstracts the differences between different operating systems and hardware architectures. The JVM is responsible for interpreting the java bytecode and for translating it into an action or system call.

- ***.NET CLR.*** [66] is Microsoft's platform for XML web services, designed for connecting devices and information in a common but still customizable way. The .NET common language reference (CLR) represents it's infrastructure middleware. CLR provides an environment for code execution that manages the running code and simplifies software development providing automatic memory management, a security system, simplified deployment, cross-language integration and interoperability with existing code/systems.

- ***Adaptive Communication Environment(ACE)***[61] is a highly portable toolkit written in C++ which encapsulates native operating systems network programming capabilities (e.g. connection establishment, event (de)multiplexing, interprocess communication, (de)marshalling, concurrency and synchronization). The main difference between ACE, CLR and JVMs is that ACE is always a compiled interface, rather than bytecode interface, removing another level of indirection and optimizing runtime performance.

The ***distribution middleware*** defines higher level distributed programming models whose reusable component and APIs extend the pro-

gramming capabilities incorporated by the host infrastructure middleware [60]. It enables programming distributed applications similar to standalone ones by invoking operations on target objects without any hard-coded dependencies (e.g. location, programming language, operating system, communication protocols or hardware). The core of distribution middleware are the request brokers:

- OMG's ***Common Object Request Broker Architecture***(CORBA) [29] relies on a protocol called the Internet Inter-ORB Protocol (IIOP) for remote objects. Everything in the CORBA architecture depends on an Object Request Broker (ORB). The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely. Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. The client can now make method calls on the object reference as if the CORBA server object resided in the client's address space. The ORB is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client. A CORBA object interacts with the ORB either through the ORB interface or through an Object Adapter - either a Basic Object Adapter (BOA) or a Portable Object Adapter (POA). Since CORBA is just a specification, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is an ORB implementation for that platform. Major ORB vendors like Inprise have CORBA ORB implementations through their VisiBroker product for Windows, UNIX and mainframe platforms and Iona through their Orbix product.

- Microsoft's ***Distributed Object Model***(DCOM) [10] is often called 'COM on the wire', supports remoting objects by running on a protocol called the *Object Remote Procedure Call* (ORPC). This ORPC layer is built on top of DCE's RPC and interacts with COM's runtime services. A DCOM server is capable of serving up objects of a particular type at runtime. Each DCOM server object can support multiple interfaces each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space. As specified by COM, a server object's memory layout conforms to the C++ vtable layout. Since the COM specification is at the binary level it allows DCOM server components to be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL. As long as a platform supports COM services, DCOM can be used on that platform. DCOM is now heavily used on the Windows platform. Companies like Software AG provide COM service implementations through their EntireX product for UNIX, Linux and mainframe platforms; Digital for the Open VMS platform and Microsoft for Windows and Solaris platforms.

- Sun's ***Java Remote Method Invocation*** (RMI) [35] is a distribution middleware technology that relies on a protocol called the *Java Remote Method Protocol* (JRMP). Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object have to be written in Java. Each Java/RMI Server object defines an interface which

can be used to access the server object outside of the current Java Virtual Machine(JVM) and on another machine's JVM. The interface exposes a set of methods which are indicative of the services offered by the server object. For a client to locate a server object for the first time, RMI depends on a naming mechanism called an RMIRegistry that runs on the Server machine and holds information about available Server Objects. A Java/RMI client acquires an object reference to a Java/RMI server object by doing a lookup for a Server Object reference and invokes methods on the Server Object as if the Java/RMI server object resided in the client's address space. Java/RMI server objects are named using URLs and for a client to acquire a server object reference, it should specify the URL of the server object as you would with the URL to a HTML page. Since Java/RMI relies on Java, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is a Java Virtual Machine (JVM) implementation for that platform.

- W3C's standard ***Simple Object Access Protocol*** (SOAP) [71] is a distribution middleware technology based on a lightweight and simple XML-based protocol that allows applications to exchange structured and typed information on the Web. SOAP is designed for enabling automated Web services based on a shared and open Web infrastructure. SOAP applications can be written in a wide range of programming languages, used in combination with a variety of Internet protocols and formats (such as HTTP, SMTP, MIME), and can support a wide range of applications, from messaging systems to RPC.

The ***common middleware services*** define high-level, domain-independent services that allow application developers to concentrate on

programming business logic [60]. While distribution middleware focuses on managing end-system resources in support of an object-oriented distributed programming model, common middleware services focus on allocating, scheduling, and coordinating various resources throughout a distributed system using a component programming and scripting model [59]. Some typical middleware services are:

- OMG's ***CORBA Common Object Services*** (CORBAservices) [28] defines a variety of services, including event notification, logging, multimedia streaming, persistence, security, transactions, fault-tolerance and concurrency control. It provides domain-independent interfaces and capabilities that can be used by various distributed applications.

- Sun's ***Enterprise Java Beans*** (EJB) [49] technology allows developers to create n-tier distributed systems by linking pre-built software services (beans). Since EJB is based on top of java technology, the EJB service components can be only built using the java language.

- OMG's ***CORBA Component Model*** (CCM) [30] defines a superset of EJB services and capabilities that can be implemented in any programming language supported by CORBA.

- Microsoft's ***.NET web services*** [66] complements the lower-level .NET middleware capabilities and allows application logic to be packaged into components accessible using standard higher-level internet protocols (such as HTTP). However, unlike the traditional component technologies, .NET web services are accessed using web protocols (e.g. HTTP and XML) rather than object model-specific protocols.

The ***domain-specific middleware services*** define services tailored to the requirements of particular domains, like telecommunications, e-commerce, health care, process automation or aerospace, targeting vertical markets [60]. They represent the least mature middleware layer, partly due to the lack of standards [59].

3.3 Load Distribution for Distributed Systems

Distributed environments are based on some type of object/component model describing general design principles of the middleware, its services and applications. These environments are generally composed of components, accessed using well-defined interfaces. Components are addressed using references. The main difference from object-oriented programming is that the invoker component (client) and the target component (server) can reside on different, even heterogenous hosts. The heterogeneity and the distribution itself are hidden by the middleware, so that from the developer's point of view distributed systems are developed in a similar fashion with ordinary object-oriented applications [41].

As application components are distributed over a number of hosts, the slowest host can determine the overall performance of the whole application, thus load imbalance is a significant problem of the middleware. For dealing with this problem, load management services have been created, in order to compensate the load imbalance by distributing the load across the available hosts.

3.3.1 Load Management Services Characteristics

Several characteristics of the middleware have been found to be important when load management services are being evaluated [41]:

- **Distribution:** Since the interacting components are distributed over different machines or even geographical locations, the load management system has to be implemented in a distributed way for guaranteeing optimal results as well as scalability.

- **Heterogeneity:** Current middleware-based applications can have two levels of heterogeneity: On the one hand the runtime environment can be heterogenous in respect to the hardware and operating system and on the other hand the components can be heterogenous in respect to their implementation. This will have a severe influence on the persistence of distributed component instances, in respect to the load management system.

- **Transparency:** The component models imply a certain level of transparency. Location transparency requires that the client is unaware of the actual location of the component it is accessing. Access transparency implies that all components are accessed in a uniform manner, independent of their implementation or runtime environment. The same transparency requirements must be fulfilled by the load management systems.

- **Granularity:** Component-based distributed applications are usually more fine-grained than other categories of applications (offering similar services). While this eases the load distribution and increases the efficiency of the load management system, it also complicates the load distribution strategies, since single components as well as whole applications have to be taken into account.

- **Openness:** In this context, openness means that once a client receives a reference for a component, it may request services from that component at any time and furthermore it can also pass this reference to other clients. Because of this complete openness, the

workload associated with a component is potentially unlimited, so load management systems have to provide mechanisms for handling this kind of overload.

3.3.2 Load Management Services Classification

Based on their implementation, load management services can be classified into [41]:

- **Application level load management** (services integrated into the application): the application developer does most of the work, implementing the load management functionality.
- **System level load management** (services integrated into the runtime system): hides the complexity of load management from the application developer, integrating the load management functionality in the runtime environment (operating system or middleware).
- **Service level load management** (separate services): Represents a hybrid of the previous two approaches, since the load management is done by a separate service interposed between the application and the runtime environment.

Another possible classification criteria is based on workload distribution. While application level load management services generally distribute data, since at this level the application internals, algorithms and structures are known, most system level load management services distribute processes, having no knowledge about application internals. Since component-based distributed systems are the focus of this work, it seems natural that component instances should be the natural load-distribution entity.

Analyzing the three classes of load management services in the light of the previously mentioned middleware characteristics, the following ob-

servations can be made [41]:

- Application level load management needs to be implemented by the developer for every application. This can lead to conflicting behavior when different applications with possible contradictory load management strategies have to share the same runtime environment. Moreover, this type of load management cannot successfully fulfill the transparency requirements implied by various component models.

- System level load management cannot fulfill the transparency requirements, either. Moreover, it is not powerful enough for enabling efficient implementation of a variety of load management strategies since due to it's location at operating-system level it can not know the content of what it is distributing.

- Service level load balancing seems to be the best approach. It fulfills the transparency requirements aforementioned thus easing the development of load manageable applications. By managing the whole distributed environment using a unified strategy it avoids possible conflicts caused by contradictory strategies.

A cost-effective solution is employing load balancing services based on distributed object computing middleware, such as The Common Object Request Broker Architecture (CORBA) [29], JAVA Remote Method Invocation (RMI) [35], Distributed Component Object Model (DCOM) [10] and Microsoft .NET [16]. The load balancing services distribute the client workload among the existing back-end servers in an equitable way, in order to obtain the best possible response time, given a particular load.

3.3.3 Load Management Services Requirements

Load Balancing features required for satisfying the demanding requirements of complex distributed systems will be presented below. Server transparency, decentralization and the requirements for taking into account component state are detailed [56].

Server Transparency is an important requirement. Since a single server can become overloaded, thus representing a system bottleneck, an adaptive load balancing service can be used to[56],[41]:

1. distribute client requests among a group of servers in a equitable way

2. actively monitor and control the load on the servers

An adaptive load balancing service must communicate with the servers so it can force them to either accept or reject requests. *One solution* is to ensure that the application accepts load balancing requests from the load balancing service, beside the client requests. Most of the distributed applications are not designed with such capabilities nor should necessarily be since it will complicate in an unwanted way the responsibilities of application developers.*Another solution* might be using the adaptive load balancing transparently on the server-side of the distributed applications and installing a feedback/control mechanism in the server, without altering the application software.

Decentralized load balancing ensures failover protection. For a group of distributed servers being load balanced, in addition to the load information sent from each server to the load balancing service, control requests (e.g. for discarding an incoming request) are sent from the load balancing service to the servers [56].

Load balancing services are often centralized, a single load balancing server managing client requests and server loads and performing all load

balancing tasks for each distributed application. Although these systems are easier to design and implement, they represent a single point of failure to the system, thus affecting system reliability and scalability. The solution is using a federated architecture, in which a distributed set of load balancers collectively form a single logical service. The main advantages of this architecture are:

1. there is no single point of failure
2. there is no single bottleneck

All load distribution decisions are being taken in a collaborative way, so that each load balancer can communicate with other load balancers in order to determine the best load distribution. While taking the decisions in a collaborative way eliminates the possibility of having a bottleneck it also presents a set of disadvantages:

1. Increased network overhead since all servers will require the continuous updates of the load information from all other servers.
2. Increased computational overhead due to the replicated processing for taking the decisions.
3. Complicates the load distribution algorithms since all instances will have to reach a decision in a collaborative way (or individually taken decisions have to be correlated).

The option of having an elected coordinator solves the problems mentioned above. Though, its main disadvantage is that it introduces the possibility of failure, from the moment the elected coordinator fails until the remaining instances detect its failure and select a new coordinator.

Stateful distributed applications The current state of the application is used when servicing a client request.

For enhancing the reusability and making the solution more generic, a load balancing service should be able to balance loads across servers hosting the stateful distributed application. It has to ensure that the state held by all servers is consistent, a task that is non-trivial in heterogenous environments. The load balancing service needs a priori knowledge of the state contents for transferring it to other servers. These requirements make the automatization of load balancing for stateful distributed applications a very complex task.

- *Diverse load monitoring granularity:* While a server has multiple objects running on it, each of them requiring load balancing, multiple servers might be running at a single location and these need to be load balanced also.

  Every instance of a load monitoring component utilizes resources, thus instantiating a load monitoring component for each component instance may not scale. Also, the load balancing decisions made for a group of component instances can severely influence the decisions made for another group. The solution is using a shared load monitor component for a group of component instances that have a common load metric. While this can significantly reduce the amount of resources needed, it also complicates the load monitoring implementation. Generalizing this, a load monitor component hierarchy can be created to further reduce the number of communication messages for communicating the load information, leading to a reduced network bandwidth requirement, which can be important in some circumstances [56].

- *Fault tolerant load balancing:* Distributed applications have high availability requirements. They must be available to the clients at all times, thus they must be fault tolerant.

Centralized load balancing services introduce a single point of failure in the system. Decentralization of load balancing services leads to better fault handling. Being distributed applications at their turn, however, they are susceptible to the same failure types as the distributed applications they load balance. This problem needs further analysis and new solutions need to be developed.

- *Extensible load balancing algorithms:* Load conditions on a distributed application can suffer dramatic changes at some moments of time in a day. These times are in general not known a priori. Also, the number of servers that service client requests can also vary in time.

  Most load balancing services (e.g. SCO Unixware, Windows NT WLBS, CORBA TAO ORB, MagicRouter NOW Project, IBM Interactive Network Dispatcher, Jonas Application Server) only support a very restricted set of load balancing algorithms that might not be adequate at all times during the lifetime of a distributed application. Another problem is having these algorithms configured in a static way in the load balancing service. The load balancing service can not predict the situation where several new servers are added/detached to/from the group. Also, a poorly designed load balancing strategy could fail in handling degenerate conditions, such as unstable server loads. An improved load balancing strategy should perform the following:

  1. Consider past load trends when predicting future load conditions.
  2. Take advantage of the sophisticated algorithms designed specifically to restore system equilibrium when it is perturbed by external forces, like additional client requests or transient loads

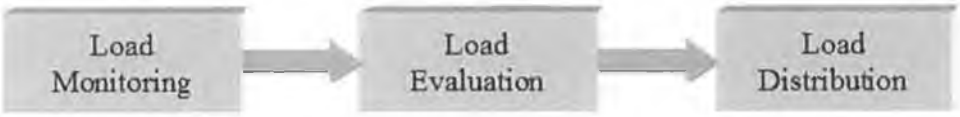

Figure 3.2: Load Management System Architecture.

generated by other applications.

3. Take decisions based on multiple load metrics

- *On demand server activation:* While a load balanced distributed application starts with a given number of servers, depending on availability of resources (such as CPU, network bandwidth), this number may increase or decrease over time. This requires the load balancing service to have a means of adding/detaching servers on demand.

3.3.4 Load Management Services Components

A load management system has, in general, three components:

- load monitoring component
- load evaluation component
- load distribution component

These components work at different abstraction levels while performing different tasks, thus easing the implementation of the load management system (See Figure 3.2).

The load monitoring component must provide in a dynamic way two categories of information:

- information on resource utilization and availability (resources may be shared with legacy applications)
- information on application components and their resource usage

The load distribution component must distribute the existing and incoming load. The available mechanisms for load distribution are:

- initial placement: deals with creating component instances on hosts with sufficient computing resources for efficient component execution
- replication: deals with creating copies of an existing component instance, called replicas. The following service requests will be divided among the original component instance and all replicas.
- migration: deals with moving an instance of a component from one host to another that offers better execution.

The load evaluation component must take decisions about load distribution using the information received from the load monitoring component.

3.3.5 Load Management Services Design Challenges

Two of the most important issues are enhancing the feedback and control loop and supporting modular strategies for load distribution [57].

Enhancing feedback and control

Adaptive load balancing services have to determine the current load conditions on all registered replicas. The load balancer should have no apriori knowledge about the particular type of load metric used. Moreover, the type of metric should be unimportant, only the magnitude of the load being considered while making the load distribution decisions. This will ensure that the combination of load monitors can be changed at runtime and that new types of load monitors can be added at runtime without the need of reconfiguring the system [57].

The replica load sampling procedure should be completely transparent to the replicas to avoid an obtrusive design that would not be feasible (altering application code - generally unavailable) and would not scale well.

Supporting modular load distribution strategies

In some applications load evolution may be predictable. In others it may not be possible to predict it [57].

Because some load analysis techniques are not suitable for all use-cases while others, more general, do not yield optimal performance under some conditions, it would be useful to analyze the set of replica loads in different ways, depending on the situation. As example, in certain situations it can be useful to analyze the workload history for a group of components for predicting high load conditions, while in other situations, where the duration of the complete request is very short, the analysis might exceed the required processing time.

When required, it must be possible to change the load analysis algorithm dynamically, without shutting the system down (critical requirement in some cases). Some applications may even require different load analysis algorithms at runtime, for adapting to new workload patterns.

All adaptive load distribution services must take into account possible hazards in the load analysis algorithms:

- **Thundering herd:** This situation appears when a low loaded server becomes available. The load distribution service can start forwarding all requests to that server immediately and by the time the new load is reported the server might have become a overloaded one. This effect is minimized by maximizing the rate at which the loads are reported and analyzed (ideally this rate should be equal with the rate at which the requests are forwarded). Limits for the fre-

quency of receiving and analyzing the reports are being imposed by the computational and possibly network overhead introduced [57].

- **Balancing paroxysms:** This situation appears when the number of servers is small. For example, considering only two servers, if a naive load distribution service is trying to keep the load uniform, it will continuously shift the load to the less loaded server, which will then become the more highly loaded one. If the load distribution service uses migration, this problem can become even more severe, the service continuously moving some component instances between the two machines [57].

3.4 Quality Of Service for Distributed Systems

Many domains rely heavily on predictable computing and networking services for performing their jobs, e.g. aerospace, health, military and manufacturing. Next generation distributed applications require a broad range of features, such as service guarantees and adaptive resource management for supporting a wider range of QoS aspects, such as dependability, predictable performance, secure operation and fault tolerance [36].

Information technology is becoming highly comoditised and there is a growing acceptance of the network-centric paradigm, where distributed applications, with a range of QoS needs are constructed through the integration of separate components, connected by various forms of communication services. This interconnection ranges from

- very small and tightly coupled, to
- very large and loosely coupled (global communications systems).

The result of these trends is:

- Focus shifting from programming to integration.
- Component QoS proves to be insufficient, end-to-end QoS support being needed.

While individual technologies have touched the problem of delivering end to end quality of service (QoS) for specific domains or usage patterns, these achievements have not substantially contributed to the broad domain of QoS enabled distributed applications [11]. For offering end to end QoS, an architecture that integrates and coordinates the existing QoS technologies is needed. The integration and coordination must take place

1. across all system resources
2. at all system levels
3. on all time scales of system development, deployment and operation

All systems are increasingly required to use COTS components. The newly available COTS components allow clients to invoke operations on distributed components, ignoring details such as component location, programming language, operating system platform, communication protocols or interconnects and hardware [68]. Nevertheless, the lack of support in these components for:

- QoS specification and enforcement features
- integration with high-speed networking
- technology, performance, predictability and scalability optimizations [19]

results in a very limited development rate for advanced distributed applications[38].

3.4.1 Static QoS Limitations

Static strategies for allocating scarce or shared resources to system components have traditionally been used for real-time applications. The reasons behind this are:

- insufficient system resources for more computationally-intensive dynamic approaches
- need of simplifying analysis and validation, essential for remaining on schedule and budget

These static methodologies and techniques are inflexible and can not support the next-generation QoS-enabled distributed applications requirements. These applications will have QoS requirements that can vary significantly at run-time, increasing the demands on the end to end system resource management. This makes it hard to (1) create effective resource managers using the existing statically constrained resource allocators and schedulers and (2) achieve reasonable resource utilization.

3.4.2 New QoS Techniques

The decisions for managing QoS are made throughout applications lifecycle, at design time, configuration/deployment time and at runtime. The runtime requirements are the most challenging since at this stage the shortest time scale for decisions is available. For managing the increasingly stringent QoS demands, the middleware has to be more adaptive and reflective. Adaptive middleware offers the possibility of modifying the functional and QoS-related properties

- statically, for leveraging the capabilities of specific platforms, enabling functional subsetting and minimizing hardware/software infrastructure dependencies or

- dynamically, for optimizing system responses to changing environments or requirements.

Reflective middleware also permits automated examination and adjustment of the offered capabilities. A reflective system can adapt itself to internal/external changes without any user intervention.

New QoS techniques, such as adaptive reconfiguration [36], dynamic scheduling [15] and multi-resource scheduling [72] are necessary and appropriate extensions to the existing static resource allocation techniques. Statically allocated priority banding [15] can be extended with preemptive thread priorities. The techniques for hybrid static-dynamic scheduling [63] offer a way of preserving the off-line scheduling guarantees needed for critical operations while improving overall system utilization.

The following problems have to be dealt with for ensuring system correctness, performance, adaptability and adequate resource utilization:

- **Diverse inputs:** next-generation distributed applications must simultaneously use a wide range of sources of information while sustaining real-time behavior.

- **Diverse outputs:** next-generation distributed applications must simultaneously produce diverse types of outputs, whose resolution quality and timeliness is decisive to other systems they interact with.

- **End-to-end requirements:** Many next-generation distributed applications will have to manage distributed resources in order to enforce the end-to-end QoS requirements, while operating in heterogenous environments

- **System configuration:** resource utilization and management as well as internal concurrency must be controllable throughout the network, end-systems, middleware and applications

- **System adaptation:** next-generation distributed applications must be capable of:

    1. reflecting on situational factors as they dynamically appear in the runtime environment
    2. adapting to these factors while preserving the integrity of key activities

3.4.3 Related QoS Research

While interconnecting real-time systems and also interconnecting real-time systems with non-real-time systems, a need of supporting more flexible and configurable scheduling techniques arises [15]. Advanced architectures are being designed and constructed for modern high-performance routers and switches in order to support novel approaches for providing QoS [20] [67]. Real-time applications demand QoS assurance at both end-system and network resource level. Only in this way true end-to-end QoS can be obtained. Several research efforts are targeting the CORBA middleware ORB (TAO) [36] [15]. AQUA (Adaptive QUality of service Architecture) [40] is a end-system level resource management architecture where the applications and the operating system cooperate to dynamically adapt to resource requirements/availability variations. Different QoS architectures and models have been proposed to address the end-to-end QoS challenge. IETF has several ongoing efforts for defining an architecture and proposing the necessary protocols and infrastructure requirements. Differentiated services (DiffServ) [34], integrated services (IntServ) [33] and Integrated Services over Specific Link Layers (ISSLL) [27] are some of the efforts made in this direction.

Integrated Services (IntServ), defined in RFC 1633 [6], provides QoS transport over internet using resource reservation protocol (RSVP) [23] for signalling resource requirements. ISSLL provides QoS transport over

IP over specific networking technologies. Differentiated services (DiffServ) [69] addresses the implementation and scalability issues associated to IntServ. It uses service classes for aggregating the flows, instead of keeping a per flow state and QoS requirements are being specified out-of-band, removing the requirement of a signalling protocol. A few bits in the IP header are used for specifying the service class.

There have been some attempts to design and implement a unified QoS API for developers to leverage the network and end-systems QoS available features [7] [11].

Chapter 4

Problems in Existing Load Management Systems

4.1 Introduction

In this chapter we focus on the problems detected in the presented related work for load distribution and quality of service, problems that we address in our research. In the first section the load management related problems are being presented in details, followed by the QoS related problems detailed description, presented in section 2.

4.2 Load Management-Related Issues

Current application server implementations offer the possibility of improving request response times by using clusters of servers to handle the requests. In order to distribute the incoming requests among the servers in the cluster, two methods are typically used:

- simple (hardware/OS level) request distributors
- application server-level transaction distribution control mechanisms.

Application servers offer a higher level of clustering implementation, namely at application level. This offers the system the possibility of controlling transaction location, as well as duration and termination.

In the context of our research, a cluster is a group of application servers running on a number of workstations. The application server group transparently runs the distributed applications as if the group was a single entity. Clusters provide mission-critical services, to ensure minimal downtime, maximum scalability and optimal performance.

Simple request distribution usually performs well when the components are simple and take a short time to execute. The most widely used options are *Domain Name Server* (DNS) round robin and hardware load balancers. The main disadvantage of DNS round robin is that it cannot guarantee equal client distribution across all servers in the cluster (if cooperating DNSs don't analyze the complete list of returned addresses). Hardware load balancers solve this problem, but set-up and configuration is complex and costs are high. The solution is the use of request distributors at application level, as part of the middleware.

General problems related to existing middleware-level load management services and suggestions for future research directions have been presented in the past [60]. Some adaptive load management services have been proposed for the CORBA platform [41] [53] [57].

Load balancing middleware is largely used for improving the scalability and overall throughput in distributed systems. However, many middleware load balancing services are simplistic, being targeted only for specific environments and use-cases. This causes limitations that make it very difficult to use a load balancing service for anything other than the application it was designed for. This lack of generality leads to continuous re-development of application-specific load balancing services [53]. Beside the increase in the development and deployment costs of distributed

applications, this also increases the possibility of obtaining non-optimal load balancing results since the tested and proven optimizations can not be reused directly.

The main problems detected in existing middleware load balancing services are:

1. lack of server-side transparency
2. centralized load balancing
3. lack of support for stateful replication
4. fixed load monitoring granularities
5. lack of fault tolerant load balancing
6. non-extensible load balancing algorithms
7. simplistic replica management

Most of the existing middleware load balancing services provide just the functionality required for supporting simple applications. One example is stateless distributed applications, that often use a simple load balancing service integrated in the naming service. In this case, for each client request the naming service returns a reference to a different object. This type of load balancing supports only a static and non-adaptive form of load balancing that severely limits its applicability to distributed systems, where more sophisticated middleware load balancing is needed.

Adaptive load balancing services can consider dynamic load conditions when making decisions, thus leading to the following important benefits:

- Can be used for a large range of distributed systems, since they are not designed for a specific type of applications.

- The cost of developing load balancing services for specific types of applications is eliminated, since a single load balancing service can be used for many types of applications.
- The development efforts are changed from a particular aspect of a specific type of application to the load balancing service in general. This can improve in time the quality of optimization used in the load balancing service.

The first generation of adaptive middleware load balancing services does not provide any solutions for the key problems mentioned [41] [54] [57]. Moreover, their limited functionality cannot satisfy the optimization requirements of complex distributed applications. With the growth in distributed application complexity, the load balancing requirements necessitate more advanced functionality, like:

- fault toleration
- adding new load balancing algorithms at runtime
- create replicas on demand, for handling bursty clients

Including these functionalities in the load management system can considerably improve the performance of the system.

4.3 QoS-Related Problems

There is no integrated end to end QoS solution available. The existing approaches have not focused on providing vertically (network to application) and horizontally (end to end) integrated solutions [11]. Determining the mapping of earlier QoS research onto a suitable system architecture is crucial for offering next generation QoS enabled distributed applications. For End-to-end QoS support, an environment with visible, predictable, flexible and integrated resource management strategies within and between

the components is needed. Delivering end-to-end QoS requires support from all layers:

- the network substrate
- the platform operating systems and system services
- the programming system in which these applications are developed
- the applications themselves
- the middleware used for integrating all the elements together.
- the application itself

Chapter 5

A New Load Management Framework

5.1 Introduction

In this chapter we present the proposed framework for approaching the problems mentioned in the previous chapter. First a general view of the framework is presented and then the functionality of every framework module is described in detail. Finally, the main characteristics of the proposed framework are detailed. The steps considered to be required for implementing the framework are also presented.

5.2 The Framework

A new QoS enabled load management service for component-based middleware is presented.

The proposed QoS enabled service for component-based middleware addresses the requirements and problems mentioned in the previous chapter.

The load management framework consists of several modules, with well-defined interfaces. The modules of our QoS-enabled load manage-

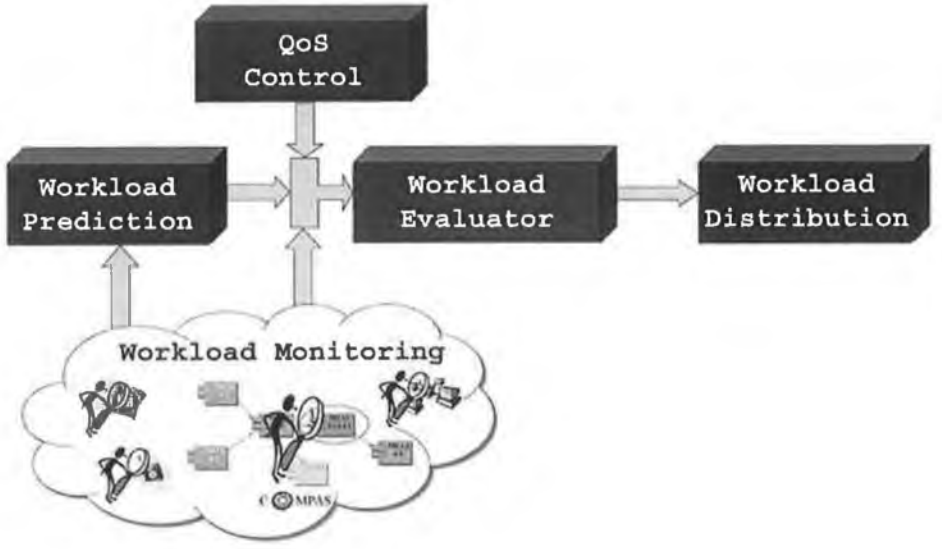

Figure 5.1: The Proposed Framework.

ment service (see Figure 5.1) are as follows:

- **Workload monitoring module:** gathers load information and computes a load description vector for all replicas. The vector is forwarded to the load evaluation module.

- **Workload prediction module:** examines the dynamic characteristics of incoming requests and based on this it creates a model. Based on the model, system load predictions for the near future are made and reported to the load evaluator.

- **QoS control module:** is responsible for ensuring the end-to-end quality of service required (or local QoS if end-to-end QoS is not supported by the existing infrastructure). The QoS is provided in the form of service levels (like premium, standard and best-effort), at application level, not at transaction level.

- **Workload evaluator module:** analyzes the information received from the load monitoring module and the load prediction module. The load introduced by the transaction currently being distributed is

not taken into account since for component-based systems a transaction can follow different paths through the system (with highly different system load) based on the execution context. The received information is used for changing the current load distribution policy for optimal performance in respect to application requirements. The information received from the QoS control module is used for modifying/tuning the selected load distribution algorithm so that the required QoS level is enforced.

- **Workload distribution module:** uses the policy received from the load evaluation module for distributing the incoming requests.

Most existing load management services are centralized, a front-end server managing all client requests and server loads. While this type of system is easier to design and implement, it introduces a single point of failure in the system. Our approach is to have an instance of the load management service active on every server. The load distribution decisions can be taken in a collaborative way, all instances of the load management service participating in this process. Another approach to this problem is having an elected coordinator service, all other instances periodically reporting to it and testing its state. The key design issues for the proposed load management service are that its modules are exchangeable at runtime and the service can be extended by adding new load distribution algorithms and new monitors at runtime.

The main difference from existing load management systems is the platform independence, at design and functionality level. This is achieved through the modularization of the load management system, which offers the possibility of implementing some monitors and the distribution algorithms in a standardized and platform-independent way.

When considering farms of servers hosting multiple distributed applications, the possibility of activating/deactivating servers at runtime is an

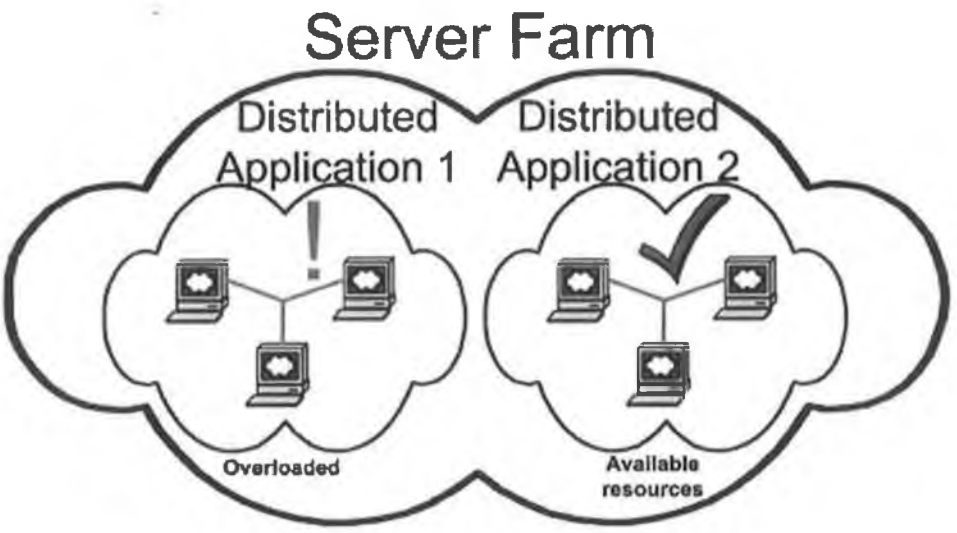

Figure 5.2: Two separate clusters load-managed.

important requirement for optimal performance achievement. In figure 5.2 the initial configuration is presented, where the two application clusters are unaware of each-other and one of them is overloaded while the other one has available resources. The proposed optimization is presented in figure 5.3, where the two clusters share their resources and both can process their load without failures.

It is considered that any viable load management service must accept pluggable load monitoring modules that can be selected according to the requirements of the distributed application being load managed. Most of the proposed load management services rely on specific CORBA functionalities, which would make them very hard if not impossible to use on other middleware platforms. We propose a load distribution service that uses no specific functionalities for any middleware platform. Parts of this service (like the load distribution algorithms descriptions) are platform-independent.

The load management service can be extended with new features at runtime, by activating the required module(s) (e.g. the load prediction module). At the same time, the modules used in the system can be up-

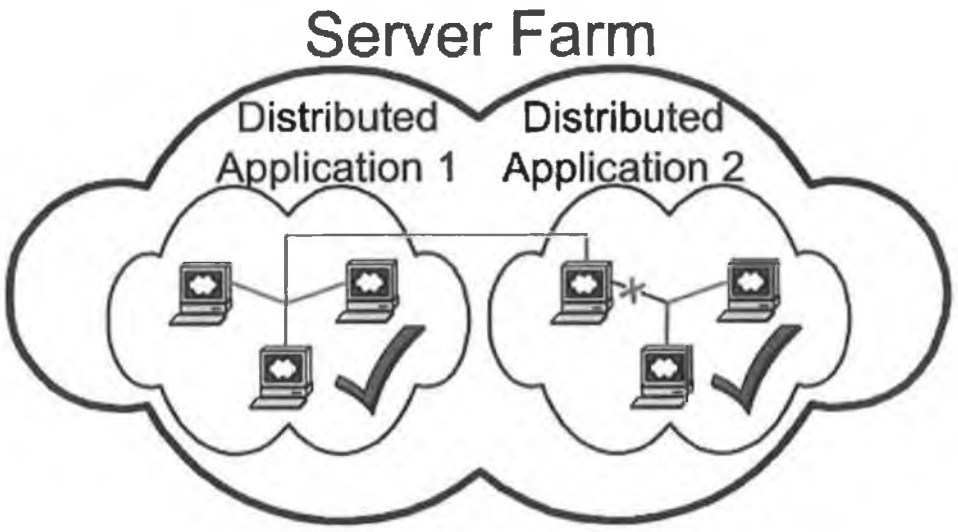

Figure 5.3: Two clusters load-managed, sharing resources.

dated and newly developed load distribution algorithms or load monitors can be added. The modular architecture also facilitates the correction of an initial configuration error, like selecting wrong or not enough types of load monitors, without the need of shutting down the application.

If some features are not required (e.g. for managing the load on a simple distributed application) the unnecessary modules can be removed from the system. This will ensure that the load management service uses as few resources as possible for achieving the required response times and performance.

5.3 The Load Monitoring Module

For some systems, observing only a few basic load metrics, such as CPU usage, disk access, available memory and network bandwidth are sufficient for making good load distribution decisions. For more complex systems, the performance could be influenced by resource contention (e.g. database access) and other complex factors thus component instances have to be monitored in order to select optimal load distribution. While a server has

multiple component instances running on it, each of them requiring load balancing, multiple servers are active and each of them needs to be load balanced. Every load monitor instance requires resources, thus instantiating a large number of load monitors is not a valid approach. In addition, the load distribution decisions taken for a group of component instances can severely influence the decisions taken for other groups. Our proposed solution is the use of pluggable load monitoring modules (e.g. CPU, memory, network, component instances monitors). These load monitoring modules can be activated, deactivated or changed at runtime, according to system requirements. While changing a load monitoring module at runtime affects the performance of the system during the exchange time (the information from the monitor is not available) it is considered that this is a low cost for the overall system performance improvement when the required set of monitors is used. Detecting the information that needs to be monitored is a complex task and particular to every distributed application and thus it is considered that no automatic mechanism or distributed system analysis can achieve good performance. Only runtime distributed application analysis can offer the required information and thus runtime intervention from the system administrator is required.

Different performance monitoring and prediction frameworks are being proposed and developed ([4] [25] [8] [70] [58] [62] [22]). Parts of these frameworks could be used as plug-ins for our load management service.

5.4 The Load Prediction Module

For some distributed systems, workload time evolution is fairly deterministic, thus predictions for its evolution in the near future can be easily made. For other systems, creating such a model is a very difficult if not impossible task.

A load prediction module that will use the information it receives from the load monitoring modules is proposed. The received data offers information on the runtime environment and the client workload time evolution will also be gathered.

This information constitutes the entries for a modelling algorithm that will create a model. The model created will be automatically and continuously refined and its predictions compared with reality. While these predictions remain close to reality, they are forwarded to the load evaluation module. If differences over a certain threshold are detected, the prediction mechanism is invalidated.

The predictions will be revalidated when the difference from reality drops back under the imposed threshold and maintain so for a defined time period. This ensures that for systems where the workload has a predictable pattern models for predicting it will be created.

The load prediction module component can be upgraded (replaced) when new prediction methods become available and, if not needed (for small distributed applications), it can be disabled.

5.5 The QoS Control Module

A key design issue is that for the proposed service the QoS will be implemented as a part of the load management system, being totally transparent for the distributed application being load managed.

The QoS Service levels are at application level, i.e. different users accessing the distributed application can avail of different service levels but one user can not avail of different QoS service levels for different operations made. The latter option could be introduced if complex frameworks for monitoring and modelling the application [2] are incorporated in this service so that application architecture would be known but this is this

does not represent the main focus of the present work.

The QoS levels and their performance guarantees are specified in a standardized format at deployment time but can be changed during runtime. This is an important requirement since it offers the possibility of changing the QoS guarantees without the need of changing any code in the distributed application.

Based on the specified QoS levels, the QoS module establishes for every newly created connection the end-to-end QoS requirements (for the required service level for that particular user).

If end-to-end QoS requirements are not available for a connection (due to the infrastructure not fully supporting QoS) local QoS requirements can be enforced and the connection is continuously monitored. If increased connection delays are detected the algorithm can adapt the enforced QoS policies trying to maintain the service level agreement. While this can not offer full hard end-to-end guarantees it offers the best solution for this situation.

5.6 The Load Evaluator Module

Load conditions on a distributed application can suffer dramatic changes at some moments in time, generally not known apriori [24]. Most load distribution algorithms are designed targeting specific workload conditions, for which they realize an optimal distribution (e.g. Round Robin, Weighted Round Robin). A load distribution algorithm might not be able to handle degenerated load conditions (like unstable servers) and the use of specifically designed algorithms for restoring system equilibrium is required [31]. Thus, a key requirement for any load management system is the possibility of changing the load distribution algorithm.

The selection has to be made dynamically, at runtime, for ensuring

high availability. New load distribution algorithms can be developed and existing algorithms optimised for particular workloads [31] so a good load management system has to offer the possibility of adding them to the system. For complex distributed systems, the workload can change during runtime in such a way that the load distribution algorithm used can no longer provide an optimal policy for load distribution.

Our solution to this problem is dynamically changing the algorithm. For automatic load distribution algorithm selection it is required that all algorithms include a standard description for the workload type for which they are most suitable. These descriptions are compared by the load prediction module with current and predicted workload and the best load distribution algorithm is activated.

Since complex load distribution algorithms can have complex tuning parameters, for the initial configuration it is considered that the algorithm also includes a set of fixed values for the tunable parameters (i.e. the same algorithm with different tuning parameters is considered as different algorithms). As a further extension of this service, the possibility of adaptive parameter tuning can be investigated but it is beyond the current focus of the current work. In order to ensure that the modules of the load management system are exchangeable, the load evaluation module must have no apriori knowledge about the particular metric/combination of metrics used for load monitoring. Moreover, this should be completely unimportant, only the magnitude of the metric being considered while optimizing the load distribution policy. This is a critical requirement for allowing dynamic load distribution algorithm and load monitoring modules changing.

The system will use the predictions from the load prediction module for optimizing its load distribution and/or activating new servers before predicted workload peaks reach the system. This will minimize the re-

sponse time and optimize the load distribution and systems response for high-priority service levels. When increases in the percentage of users requesting high-priority service levels is predicted the system can preemptively adjust its configuration. In this way it will be ready for optimal response when the new requests will enter the system.

The load evaluator module is also responsible for activating/deactivating servers, according to the information received from the load monitoring and load prediction modules. This is a common situation for hosted applications, where a farm of servers hosts a number of distributed applications. The service will maintain a list of available servers and according to the requirements servers can be activated / deactivated in a completely transparent way.

5.7 The Load Distribution Module

The load distribution module uses the policy selected by the load evaluator module when forwarding the incoming requests to the servers. The most important feature of this module is failover protection. A good load distribution module should be always available and should be able to distribute the incoming requests even if other modules of the load distribution service fail.

The solution proposed is the inclusion of a simple load distribution algorithm, like round robin, in the load distribution module. If the load evaluator module fails for some unexpected reason, the load distribution module should revert to a simple algorithm for load distribution, like round robin, in order to ensure that the system will keep on running albeit with degraded performance.

The load distribution module has to take into account all levels of load management, namely initial placement, migration and replication.

Initial placement represents the creation of new component instances on servers where enough resources are available for efficient execution. Note that this does not refer to placement of component instances at application deployment time; rather it refers to placement of component instances during runtime when the load management system determines that extra component instances are required.

Migration of running instances deals with the movement of existing component instances to another server that offers the required resources for more efficient execution. In the case of stateful components the state must be continuously synchronized.

Replication of component instances involves the creation of new component instances from an existing one. The new instances must be identical with the source. This applies only to stateful component instances, for stateless ones only simple instantiation being required. A complex problem in this case is mirroring the state of the original component instance to the newly instantiated ones. The state must also be continuously kept synchronized among all existing servers.

5.8 Characteristics of the QoS Enabled Load Management Service

The solution to the problem of having a centralized front-end server managing all client requests, which represents a single point of failure, is the use of a federated architecture. A set of load management service instances will form in a collective way a single logical entity. Our approach is to have an instance of the load management service active on every server.

Two possibilities of taking decisions are available. Although taking the decisions in a collaborative way (See Figure 5.4, 5.5) eliminates the

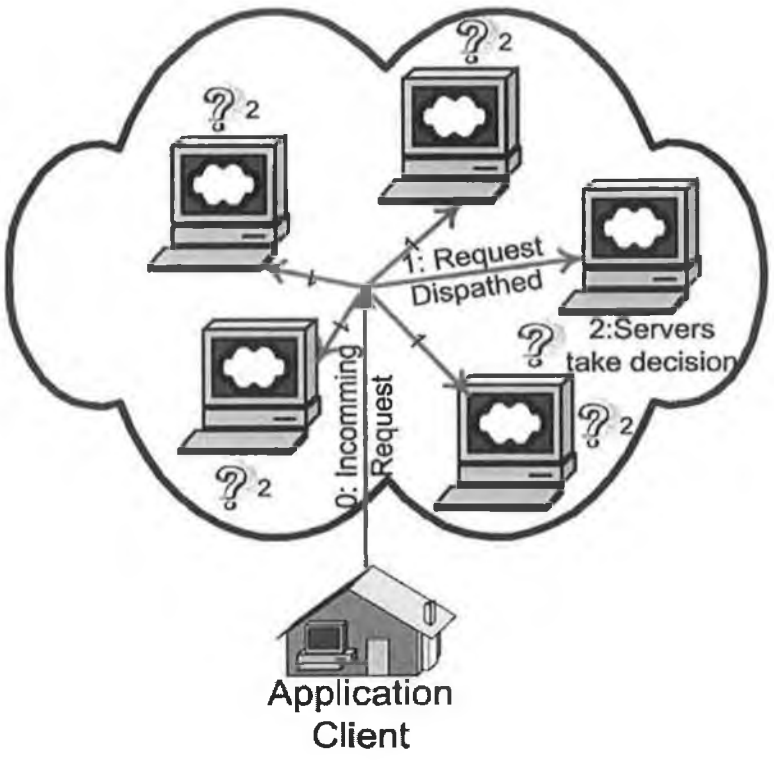

Figure 5.4: Failover Protection Operations (1).

possibility of having a bottleneck it also presents a set of disadvantages:

- Increased network overhead since all servers will require the continuous updates for the load information from all other servers.
- Increased computational overhead since the load prediction and evaluation modules will have instances on all machines.
- Complicates the load distribution algorithms since all instances will have to reach a decision in a collaborative way (or the individual decisions have to be correlated).

The option of having an elected coordinator solves the problems mentioned above. Though, its main disadvantage is that it introduces the possibility of failure, from the moment the elected coordinator fails until the remaining instances detect its failure and select a new coordinator.

The QoS controller will send information about the active QoS level

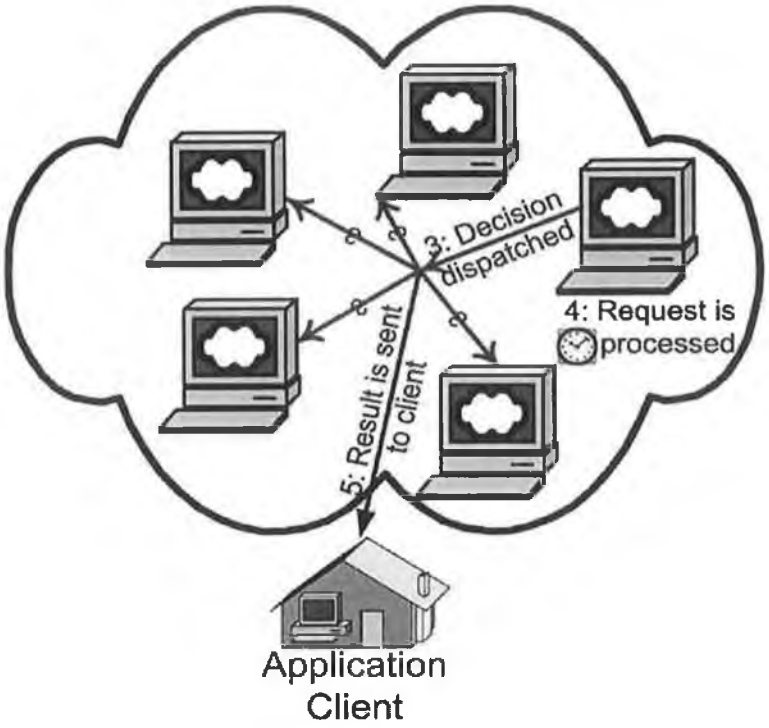

Figure 5.5: Failover Protection Operations (2).

requirements to the load evaluator module. It is the task of the load evaluator module to adjust the load distribution algorithms so that the required response times are satisfied. The load evaluator module receives also information for deciding when the system load has reached its maximum for offering the required QoS levels and new incoming transactions must be dropped.

The main advantage of this approach is that the load management system could control in a unified way the load on the distributed application. This will, on one hand, offer the guarantees needed by the high priority users while efficiently using the available resources for serving low priority users. It is demonstrated that, using this approach, while maintaining the guarantees for high priority users, the performance for low priority users is substantially improved [52].

The load management service should offer the possibility of connecting to other similar services. The group of services, each managing the

****

User name: MunteanG (111) Queue: EE_OLYMPUS/C230_LASER
File name: Server: C230_LASER
Directory:
Description: Octavian-Thesis.pdf
May 24, 2004 12:55pm

****

```
M   M                t                          GGGG
MM MM                t                        G
M M M u   u nnnn  tttt   eee   aaa  nnnn  G
M M M u   u n   n  t    e   e     a n   n G GGG
M   M u   u n   n  t    eeeee  aaaa n   n G   G
M   M u   u n   n  t  t e     a   a n   n G   G
M   M  uuuu n   n   tt   eeee  aaaa n   n  GGGG
```

****

****

workload of a distributed application, should cooperate and share available resources in order to maximize the performance of all distributed applications. The group could maintain a shared list of available servers and could temporarily transfer the control of a server or a group of servers from a low loaded system to a high loaded system.

Chapter 6

Application Server Simulation Models

6.1 Introduction

This chapter presents the work carried out for supporting the proposed QoS-Enabled Load Distribution Framework and the performance improvements it could add to existing component-based distributed systems. We will introduce the simulation models we have used for evaluating existing load distribution algorithms and possible optimizations.

Our simulation models were created using the Hyperformix Work-Bench 4.0 [32]. The tool has been selected because it is a general-purpose modelling and simulation tool, it offers a graphical programming language for performance modelling and it was powerful and easy to use to construct simulation models. It is suitable for prototyping design issues for performance because it offers early possibilities for exploring the solution space.

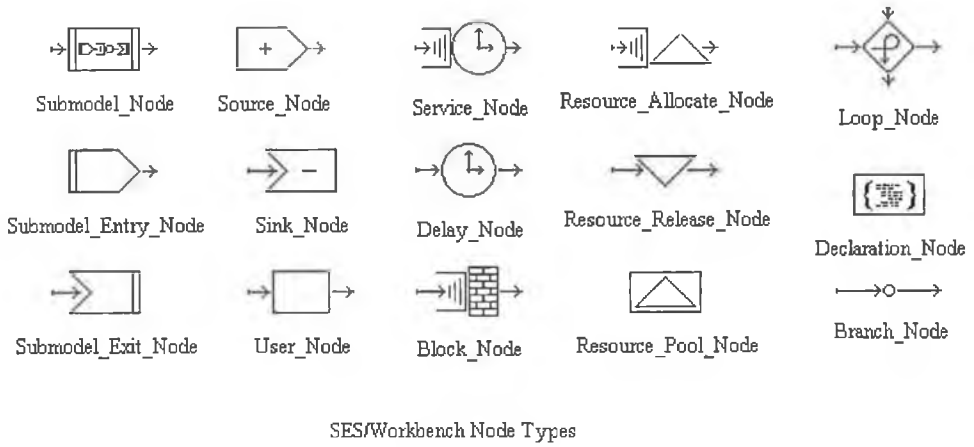

Figure 6.1: Available WorkBench Nodes.

6.2 Hyperformix WorkBench Modelling and Simulation Environment

Figure 6.1 presents the available nodes for modelling a system using Hyperformix WorkBench.

The modelling and simulation environment offers two levels for creating the models. On the upper level, only global declarations, pools of resources and system modules are allowed. On the lower level, the behaviour/functionality of every module is represented.

The **Submodel Node** allows model decomposition. It encapsulates model functionality and is similar to a subroutine or function in a program. When used on the upper level it represents a module that is part of the system being modelled while on the lower level it represents a reference to a previously defined module. For a module to be referenced, it is required that it has a **Submodel Entry Node**. The transaction can return from the referenced module to the node from which it originally entered in the submodel through a **Submodel Exit Node**.

The **Source Node** generates transactions, which, for the purpose of this work, represent the incoming transactions.Using a Source node is a flexible way to generate one or more transactions repeatedly at specified

intervals. The Source node offers complete control over the number of transactions created and the intervals at which they are created.

The **Sink Node** represents the exit point for the system. When a transaction has finished it's work in the system, it finishes by exiting the system through a sink node. Generally, all transactions have to leave the system when they have finished processing. Transactions arriving at a Sink node simply disappear from the model. Any resources held by transactions are returned to their owning resource pools.

The **User Node** allows the use of C and SES/sim languages to specify arbitrary computations to execute. When transactions arrive at a User node, the transactions execute statements specified in the node method.

The **Service Node** represents a device designed to perform a specific function for many users. This node is used to model an active resource (that is, one that performs processing on transactions), typically a hardware device, such as a disk drive. When a transaction arrives at a Service node, it enters the nodes queue. If a server is available (one that is not processing another transaction), the transaction enters service and departs after it has received its requested service time. If a server is not available, the transaction waits in the queue until the server becomes available or the transaction is selected for service according to the queuing discipline specified for the node.

The **Delay Node** can be used to delay a transaction for a specified amount of time. When a transaction arrives at a Delay node, it waits for the time specified (without queuing) and then proceeds.

The **Block Node** provides a mechanism for making a transaction wait until an arbitrary condition is satisfied. A transaction arriving at a Block node first evaluates the block until condition. If the condition evaluates to true and a server is available the transaction leaves the node. If the condition evaluates to false, the transaction enters the queue and waits

until it is reevaluated with an update statement or it is interrupted.

The **Resource Node** is used to declare one of several types of passive resources, depending upon the specific needs of the model. Each passive resource pool is represented by a Resource node. Transactions do not flow through Resource nodes. Transactions manipulate resources by passing through Allocate, Create, Destroy, and/or Release nodes. Resource pool declarations are visible according to the hierarchical level of the definition: resource pools declared on the Module window are visible to all submodels in that module, while resource pools declared in a submodel are visible only within that submodel.

The **Allocate Node** allocates passive resources to transactions. A transaction arriving at an Allocate node requests resource elements from one or more resource pools. When the resource element arrives, the transaction enters the nodes queue. If both a server and the requested resource elements are available, the transaction receives the elements and immediately leaves the node. If a server is not available, the transaction waits in the queue until it is selected for service according to the specified queuing discipline for that node. After access to a server is granted, the transaction waits at the server until the requested resource elements become available, or until the transaction is interrupted or preempted.

Transactions arriving at a **Release Node** relinquish some or all of the resources they hold. One of the following operations can be performed:

- Workbench returns resources to the pool from which they originated
- Workbench returns resources to some other resource pool
- Workbench immediately destroys resources

The **Loop Node** can be used to repeatedly route transactions through a particular section of a model. Transactions loop through that section until some termination condition is satisfied. Transactions can be looped

under for, while, or do { } while conditions. The Loop node has two entry points, Initial and Return, and two exit points, Continue and Terminate. A transaction enters at the Initial entry point and begins executing the loop. The transaction enters the body of the loop at the Continue exit point and returns from the loop body at the Return point. Until the termination condition is satisfied, the transaction continues to cycle through the body of the loop and exits the node at the Continue point and returns at the Return point. When the termination condition is satisfied, the transaction exits the Loop node at the Terminate point.

The **Declaration Node** can be used to declare constants, variables, and routines using C. Declaration nodes are independent nodes; that is, they require no arcs. Transactions do not flow through Declaration nodes.

The **Branch Node** performs no processing on transactions, however, it does have a number of important uses:

- It can be used to visually simplify models and clarify routing of transactions
- It can provide an anchor for the connection of a statistical response arc
- It can provide a location from which interarrival statistics can be collected
- It can be used to change topology arc specifications between nodes

6.3 Single Server Environment

6.3.1 Introduction

For obtaining a deeper knowledge of the internal functionality of an application server and for validating the assumptions that are the base of

the proposed load management system, a complex model for simulating in detail the functionality of a single application server has been realised in collaboration with several other colleagues. The model is used for predicting the system response time and throughput as the system workload changes.

This work was part of a larger project. The piece of this larger project that constituted part of this work is described in the following sections.

6.3.2 The Model

A simulation model of a car shop application was created. The application allows car browsing and buying. The browsing operation retrieves all fields from the database, displays them and permits the client to select a car model and buy it. The buy operation involves a credit-card check and decreases the number of car units available in stock.

The initial model was restructured, using the already developed modules, to separate all layers visible in a J2EE application. The new structure clearly separates distributed application layers and the client workload generation layer. As seen in Figure 6.2, there are six layers:

- Execution environment, represented by the Hardware and Resource layer.
- Application server, represented by the J2EE Bean Type Layer
- Distributed application level, represented by the J2EE Applications Beans and J2EE Applications Logic layers
- Workload, represented by the Client Workload Generation Layer

The distributed application has two layers since the modelling and simulation tool does not permit true hierarchical layering thus the components used by the application can not be modelled inside the distributed application logic layer. The simulation tool offers only two layer levels:

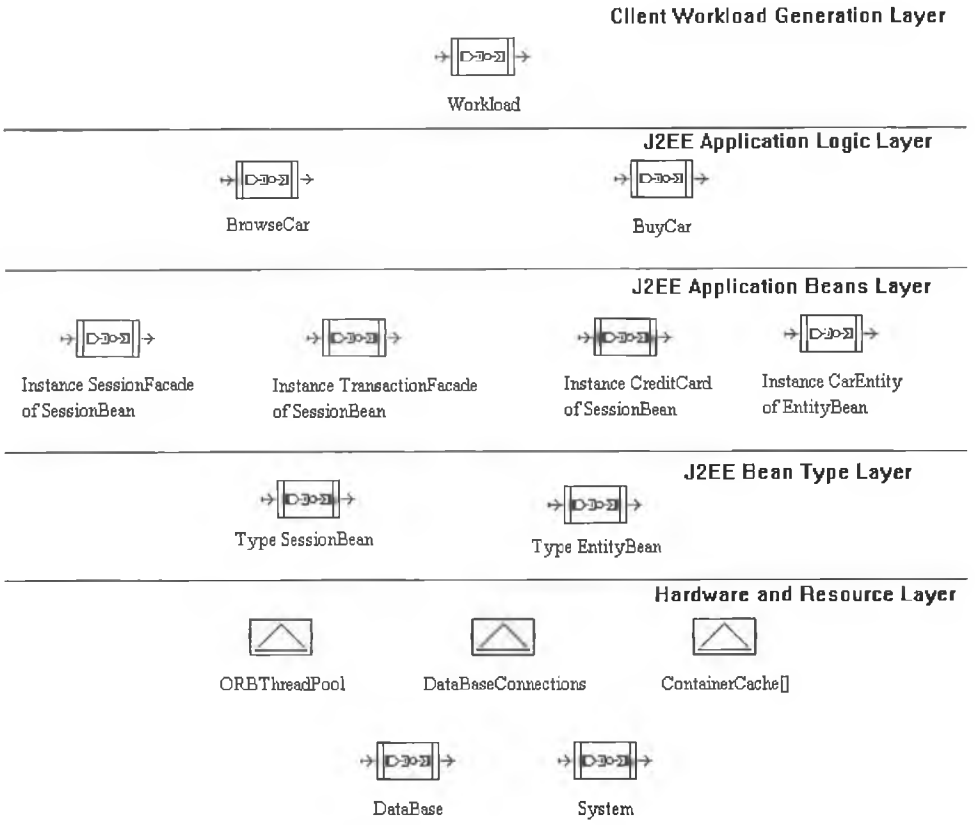

Figure 6.2: Simulation Model Overview.

- the modules level, represented in Figure 6.2
- the design level, inside a module. The Workload module is represented in Figure 6.3.

The simulation model is highly parameterized, to ensure easy adaptation to the environment and workload being analyzed. All layers of a distributed application are taken into account and specific parameters are provided. For the hardware layer, the CPU and disk times are variables and depend on the specific operation being performed. The total available memory is also a model parameter. At the application server layer, the number of threads, the container cache, the maximum number of simultaneous database connections and the size of the bean pool are parameterized. The distributed application has a number of call sequences, and for each call sequence, the resource usage (CPU, disk, memory) are

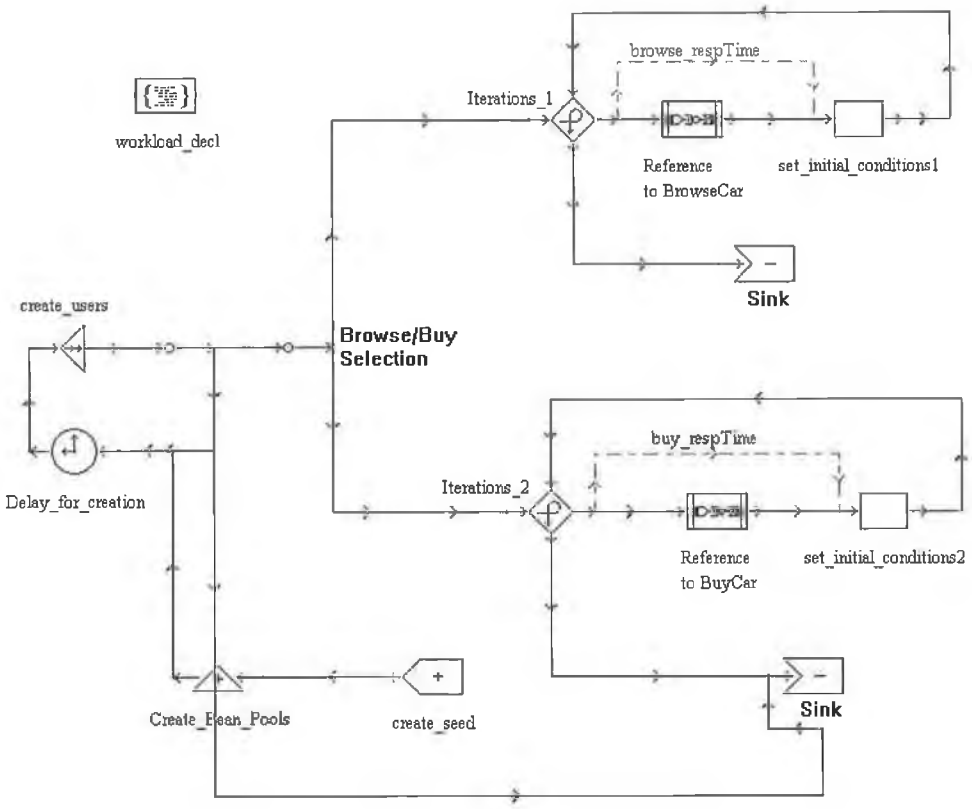

Figure 6.3: Workload Module.

parameters that are set for every call in the sequence. The workload model also simulates a load generator. The most important characteristic of the load generator is that a user performs the same type of operation for a specified number of times (it can not perform one operation and later another one), as it can be seen from Figure 6.3. The users are introduced in batches, i.e. a fixed number of users at a time. The workload parameters are the number of users, the think time, the number of users per batch and the delay between batches.

In the initial version of the model, the workflow was broken into small independent flows (at bean call level) that were activated using resources. This approach, while shortening the execution time for a simulation, does not completely conform to the real workflow and did not allow following the path taken by a transaction through the system. At the same time,

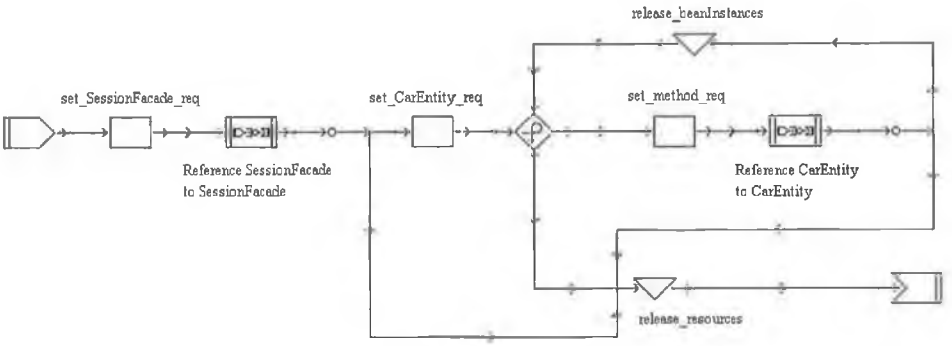

Figure 6.4: BrowseCar Module.

the existing workflow was not completely following the transaction paths in the real system. The workflow was redesigned, according to the requirements, so that any initiated system transaction can be traced and the exact path of the transaction (with all session/entity bean calls) can be observed, as seen in Figure 6.3,6.4 and 6.5.

From Figure 6.3 it can be observed that any generated user will arrive, according to a specified probability, to the node Iterations_1 or Iterations_2. The create_seed node generates only one transaction, used for starting the system. The generated transaction initializes the bean pools sizes and activates the user generation. The Delay_for_creation node ensures that the specified inter-batch delay is respected while the create_users node generates the batches of users. The initial transaction (used for starting the system) exits the system through the provided connection to the sink node, while all generated users are directed towards the second junction. According to the specified probability, the users split into two categories, users that only browse the catalogue of cars (the upper loop) and users that buy cars (the lower loop). The two loops reference the corresponding modules that contain the application logic for the specified operations (BrowseCar and BuyCar).

The BrowseCar (Figure 6.4) and BuyCar(Figure 6.5) modules incorporate the application logic for the two operations. The application is

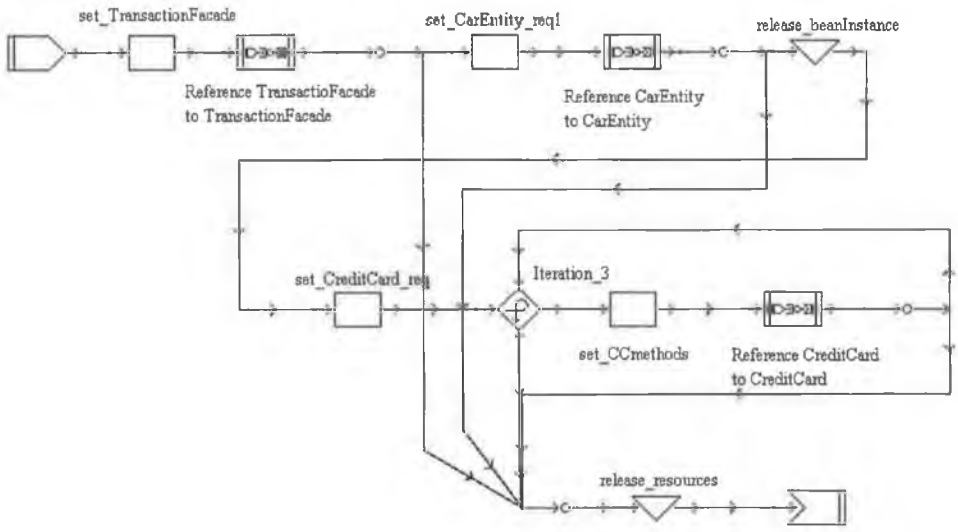

Figure 6.5: BuyCar Module.

designed using the session facade design pattern, represented by the SessionFacade and TransactionFacade references. The transaction parameters for the CarEntity component (the parameters mentioned in previous section) are set in the set_CarEntity_req(1). Since the real application that is modelled accesses all cars and displays a list with all available car models, the loop in the BrowseCar module ensures that the required number of accesses to the car entity beans is performed. After the information from an entity bean is retrieved, the entity bean is released so that other accesses to the bean (from other clients) can be performed. Before finishing the transaction, all resources are released.

The BuyCar(Figure 6.5) module accesses only the entity bean containing the information for the desired car, retrieves its info and then releases the bean to be accessible by other clients. The credit-card is verified and when the transaction finishes all resources are released.

The EntityBean type module (Figure 6.6) is used for instantiating the entity beans used by the distributed application. It implements the two types of calls possible (local and remote). For remote calls it allocates a thread, the transaction accesses the system (allocating the thread requires

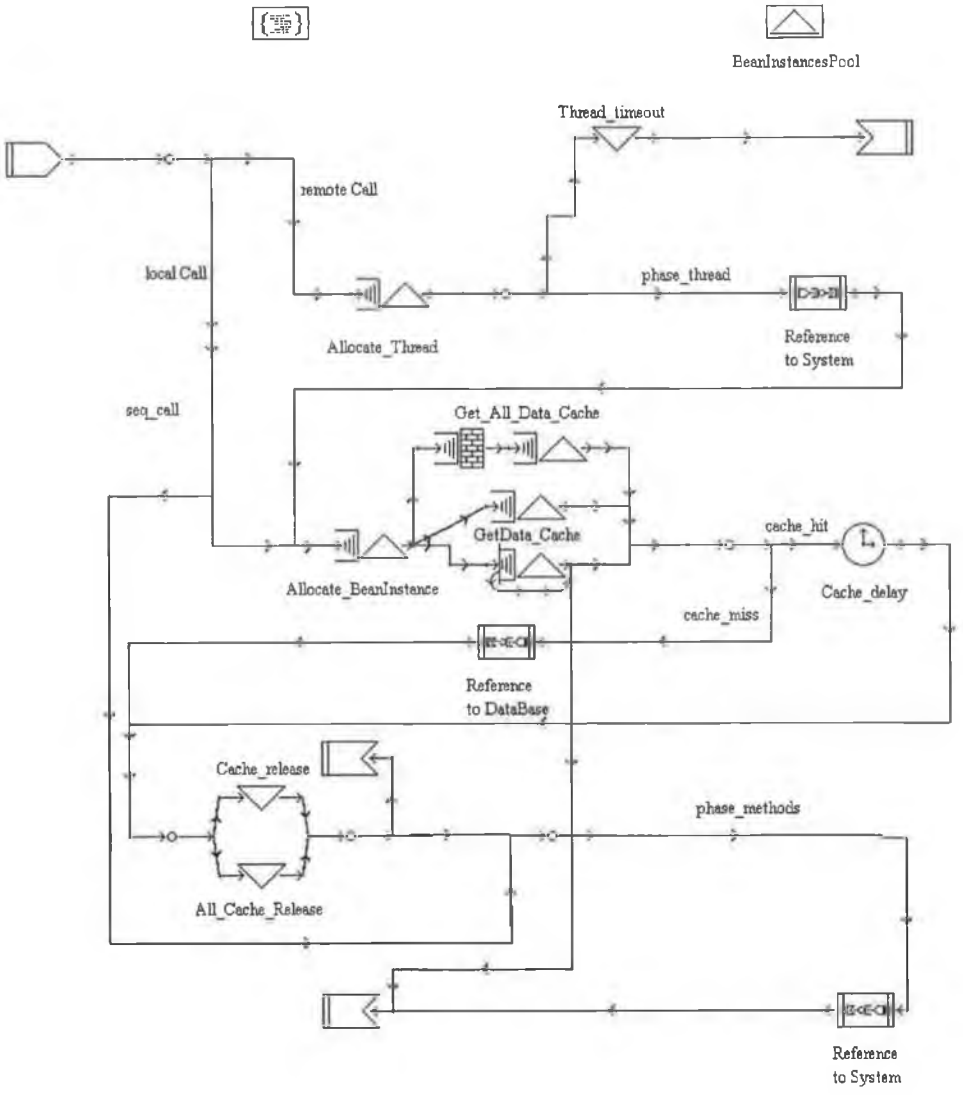

Figure 6.6: EntityBean Module.

system activity).

For ensuring system scalability and correct functionality for large databases, a cache mechanism is included. It allows cache misses and then it accesses the database module to retrieve the information and cache it. Thus the cache behavior was modelled. For the cache hits, a small delay can be introduced using the Cache_delay node. In the case of the small application tested the delay was negligible and thus set to 0. In order to model the entity beans access and the cache in a realistic way, I had to introduce a new parameter, carType, had to be introduced, that

represents the database record that is accessed. This is used to ensure that if concurrent requests to the same entity bean are tried only one will gain access and the rest will have to wait until the one that obtained the access releases it. It is also used for modelling the cache locks. The Get_All_Data_Cache and GetData_Cache nodes offer the required functionality. The cache is modelled as an array of resources. In this case cache dimension is equal to the number of records in the database but the dimension is parameterized thus easily changeable. The number of resources available in every array entry is one. In the cache, the lock is obtained at bucket level, i.e. set of entries, so a bucket is modelled as one resource. Every resource request will result in taking the only available resource from the specified array entry thus any subsequent request to that entry will be blocked until the resource becomes available or will be marked as rolled-back). I have implemented a different behavior for the read and write operations:

1. For write, a finite queue (with 0 queue length) was added so any write operation that can not get a lock for the entity bean is marked as being rolled-back (roll-back category).

2. For read, an infinite queue was used since the read has no roll-back.

3. For the findAll() method, where the application must get a lock on all buckets and after processing must release them all, initially a loop was designed. While testing the configuration, it was determined that it leads to deadlocks since by the time one transaction finished looping to lock all buckets, another transaction could have accessed some of the buckets that are still available. The solution implemented was changing the code in the node used for allocating the resources so that all buckets are allocated at once, as it is done in a real system.

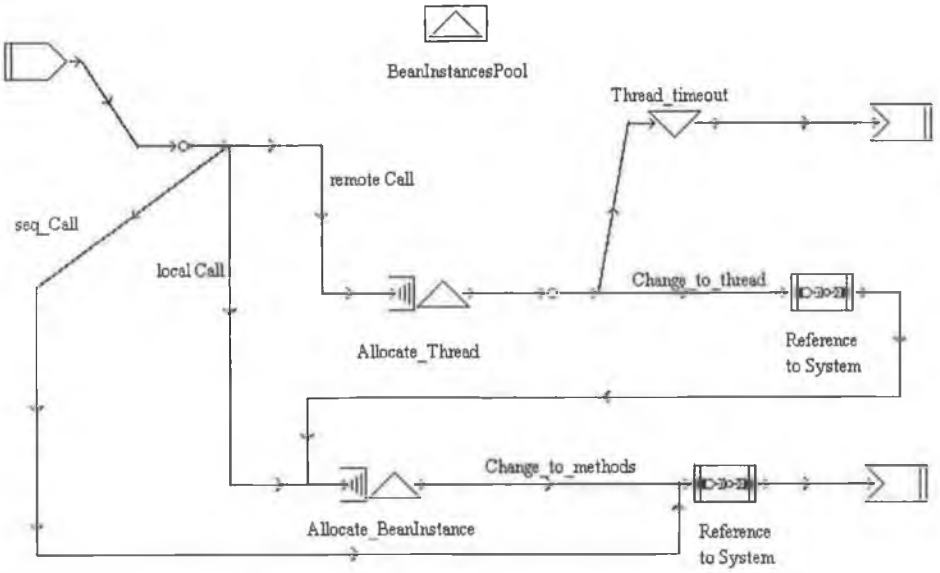

Figure 6.7: SessionBean Module.

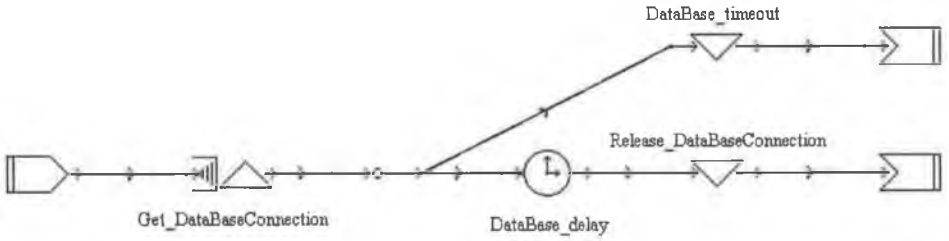

Figure 6.8: Database Module.

Since the database used for gathering results from the real application had small dimensions, from the point of view of the application server caching all data is cached, so that the cache hit probability after a short warmup is 1, and there is no database access.

The session beans type module is similar with the entity bean type module except it does not have any cache. The same separation is local and remote calls exist, so that only the remote calls will allocate a new thread (the local ones use the thread allocated when they were initiated). In both nodes, thread allocation timeouts exist.

The database module (Figure 6.8) allocates a connection from the pool of available connections, introduces a delay in the transaction execution

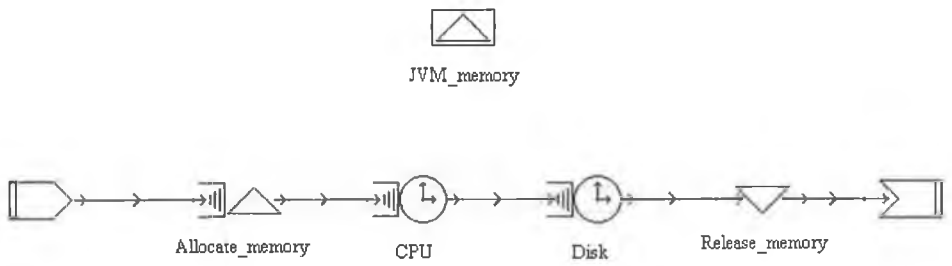

Figure 6.9: System Module.

and then releases the connection resource. As for the threads in the entity and session bean types, a connection timeout is included, so that if a transaction waits longer than a specified threshold, the connection is considered to be failed. The timeout was included for system scalability.

The system module (Figure 6.9) models the hardware resources (i.e. CPU, disk and available JVM memory). The CPU and disk nodes are service nodes, with the service time specified by the transaction being processed.

Due to the requirements for model validations (modelling a real application), all values for the parameters are deterministic and have been collected by monitoring the real application at runtime (using the JProbe monitoring tool). The results of this model validation can not be included in the thesis due to an academic restriction on access to them at this stage.

6.4 Multiple Server Environment Simulation

6.4.1 Introduction

The performance of different categories of load distribution algorithms had to be investigated, for detecting the use-cases in which a particular category of algorithms is most suitable.

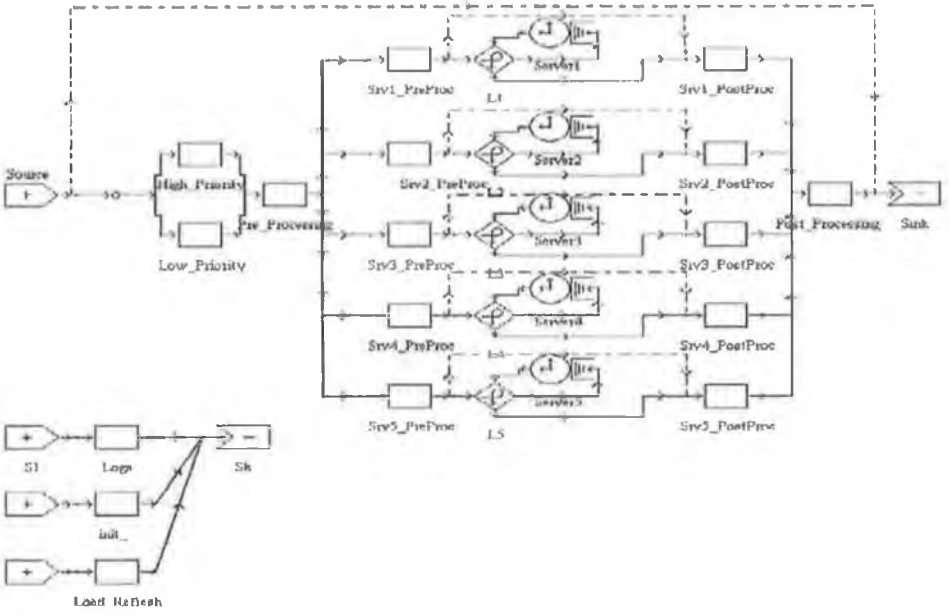

Figure 6.10: Simulation Model for Multiple Servers Scenario.

For evaluating the performance of different categories of load distribution algorithms, a simulation model was created. The model offers the possibility of using different load distribution algorithms and different workload models.

6.4.2 The Model

A model was created (See Figure 6.10) to simulate some of the most frequently used categories of request distribution algorithms.

The model consists of five service nodes representing application servers and a source node, representing a transaction generator.

The transaction generator is labelled *Source*. It generates transactions in bursts, every time unit. The transactions are generated using an interarrival rate that follows a **Poisson** distribution function. The Poisson distribution was selected since it is the most frequently recomended distribution function for modelling requests interarrival rates in the literature.

In the Pre_Processing node a service demand (load) factor is associated with every transaction, according to a **Lognormal** distribution (selected

based on the literature survey as being the most appropriate). At the same time, various counters are being set for collecting the required statistics.

The servers are labelled *Server1* to *Server5*. Each server has a service time associated, dynamically updated, according to the total load on the server (each time a transaction enters or leaves the server). The service time is computed according to the formula:

$$Service\_Time = \frac{WeightFactor}{MaxSrvLoad - CrtSrvLoad}$$

$CrtSrvLoad$ represents the total workload of the server and is computed as the sum of workloads for all transactions currently on the server

$MaxSrvLoad$ represents the maximum possible workload on the server and it is determined by simulations in order to ensure that the value of service time is a positive number,

$WeightFactor$ is used in order to adjust the final value, depending on the inter-arrival time and the number of new transactions generated. It allows the system to simulate transactions that have high workload associated as well as light transaction workloads. During the performed tests, a value of 1 was assigned for $WeightFactor$ and thus the formula used in the simulation tool is:

$$Service\_Time = \frac{1}{MaxSrvLoad - CrtSrvLoad}$$

By using this formula for the service time the depth of detail required for the simulation model is limited. It incorporates the observed behavior of the underlying infrastructure (software and hardware) and thus eliminates further (deeper) extension and parameterization of the model. The formula is based on real experiments (see [1] [3]). Based on those experiments, the dependency between the load of the system and the service time was determined to follow the aforementioned equation.

The loops *L1* to *L5* are used for ensuring that the processing time is accurate for all transactions. Every transaction loops for a number of times equal with the workload associated to it. In order to simulate the parallel processing of incoming transactions the number of servers in all service nodes has been set to a very large value (a limit never reached in the tests). Since the service nodes do not permit the specification of different service times for different servers all transactions will finish the processing after they spend a fixed amount of time (*service time*) in the system. To overcome this limitation we have considered that this time represents the time required for processing one unit of load for a transaction and the load of the transaction represents the number of units it has to process. The service time is at the same time dynamically adjusted according to the total load on that node (the sum of load values carried by all transactions in that node).

For the initial tests the *Low_Priority* and *High_Priority* nodes were not used.

6.4.3 Round Robin Request Distribution

Introduction

The Round Robin algorithm was selected for performance evaluation since it is representative for the category of simple load distribution algorithms, which have no knowledge about the distributed system runtime environment or workload they are distributing. It was preferred to the random one since the random algorithm implementation and thus its performance can vary from one load management service to another (due to different random number generators). The implementation differences would be an obstacle in the simulation as well since it would be more difficult to compare the results obtained using the random number generator offered by the modelling and simulation tool with the ones obtained from an load

management service.

The Approach

Round Robin is a simple algorithm, with no knowledge about the content being distributed or the state (load) of the destination server. It distributes the requests one for each server, in a cyclic manner.

The advantage of this approach is that the decision time is the shortest possible since there no conditions to be considered when distributing the transaction.

Due to its simplicity, the round robin algorithm is applicable in a very limited set of situations since it can not properly handle varying incoming workload nor a set of servers with different processing power or with different workload requirements.

The Implementation

The algorithm was implemented as a simple parameter increment with a test checking if the value is greater than the number of servers and if so resetting it to an initial value of 1. The check is performed after the transaction is distributed and thus it does not delay the distribution of the current transaction.

6.4.4 Weighted Round Robin Request Distribution

Introduction

The Weighted Round Robin algorithm was selected for performance evaluation since it is an extension of the Round Robin algorithm and it is representative for the category of simple load distribution algorithms (no knowledge about the distributed system runtime environment or workload they are distributing). It is more widely spread than the Round Robin

algorithm since it allows for hardware imbalances between servers to be taken into account when distributing the incoming requests.

The Approach

The Weighted Round Robin algorithm is very similar to the Round Robin algorithm, the only difference being that it allows weight assignation to servers. The Round Robin algorithm can be considered a particular case of Weighted Round Robin, where all weights are equal. Every time a request enters the system, it is sent to the server with the highest weight.

The weights are used in general to fairly distribute the load among a set of servers having different processing power. In the tests the processing capacity of one server was doubled and a weight of two was assigned to it.

The Implementation

The Round Robin implementation was changed so that the maximum weight is always detected and decremented with one unit. If two or more servers have the same weight and it is the maximum weight, the first one is selected as a target. An extra test verifies if the weight vector is null and then re-initializes it with the assigned weights.

6.4.5 Load Balanced Request Distribution

Introduction

The Load Balanced algorithm was selected for performance evaluation since it is representative for the category of complex load distribution algorithms, with knowledge about the distributed system runtime environment and/or workload they are distributing. Considering the different implementations available, very similar, the ideal implementation of the algorithm was used in the initial tests. The initial implementation did

not take into account the delays for making the decision and the server workload information is updated after every event (i.e. transaction arrival/departure). For later experiments workload information refresh frequency and the decision delay have been introduced as parameters and thus the actual performance of the algorithm can be investigated.

The Approach

The Load Balanced algorithm forwards every incoming transaction to the least leaded server. In the model used for tests the delay for taking the decision was not considered and the system updates the load vector after every new transaction enters the system.

The Implementation

A load vector was used for registering the load on every server (*CrtSrvLoad*, see section 6.4). The values are updated every time a transaction is forwarded to a server, thus the delay in vector update specific to real systems, is not taken into account. This decision was taken since the purpose of these tests is to analyze the performance improvements of load-aware distribution algorithms, without the introduced delays.

6.4.6 Introduction of Service Levels Request Distribution

Introduction

The model was restructured in order to support service levels such that particular requests can be prioritised. The service-levels support was required for evaluating the advantages of the QoS component of the framework. The performance of separate servers for every service level is compared with the proposed approach, sharing the servers between all service

levels.

The model

While in the first simulation setup, the model had no knowledge of priorities and was used to evaluate the transaction response times, in the second setup, the best performing algorithm (load balanced, see section 6.4.5, 7.3) was chosen and the concept of user classes has been introduced.

Two user classes are defined using priorities. All transactions are marked when they enter the system, as either low or high priority, and the request distribution algorithm has been adapted to take such information into account. In this model, the high-priority transactions are executed on only one of the five servers available (*Server1*). For the low-priority transactions two cases have been taken into consideration:

- the remaining four servers are used for processing all low-priority transactions, using the selected algorithm for distribution
- all five servers are used for processing low-priority transactions and modified request distribution is used for distribution

From the Source Node, the transactions are sent, according to a selected probability to the High_Priority node, the rest being sent to the Low_Priority node. These nodes are used for labelling the transactions and for selecting the destination server. The Pre_Processing and Post_Processing nodes (and the Srvi_PreProc and Srvi_PostProc nodes) are used for setting additional parameters and collecting the required statistics.

The influence of three factors is examined:

1. The ratio of low priority transactions to high priority transactions.
2. The number of low priority transactions served by the server processing the high priority transactions, relative to the number served

by the other servers. This is controlled by a weighting factor (i.e. the higher the weight the lower the number of low priority transactions processed on that server). It should be noted that the case of a server dedicated exclusively to serving high priority transactions can be considered as a limiting case for the weighting factors, i.e. the weight of the server processing high priority transactions is much higher than the weights for the other servers.

3. The overall system load.

In order to better quantify the improvements, high priority transactions were generated representing 5% and 10% of the total number of transactions.

Separate Server for High Priority Clients Case

The first server (*Server1*) is used for processing all high priority transactions and the remaining four servers are used for processing the low priority transactions. The load balanced algorithm is used for selecting the destination server for the low priority transactions.

For distributing the low priority transactions the same implementation of the algorithm described in section 6.4.5 was used. For the high priority transactions no distribution algorithm is required since all of them are processed on *Server1*.

Shared Server for All Service Levels

For this case, the load balanced algorithm had to be modified in order to offer the possibility of controlling the load on servers. This was necessary since the server that will process the high priority transactions should have a lower load than the others. A weighting factor was introduced for every server. The values resulting by multiplying the weighting factor of

a server with the workload currently being processed on that server are compared when selecting the least-loaded server and thus the higher the weight, the lower the number of transactions being processed.

The first server processes all high priority transactions and some low priority transactions, while the other four servers served only the low priority transactions. For the results presented in section 7.4, a value of 1 was used as the weighting factor assigned for all servers processing only low-priority transactions.

6.5 Two-Layered Algorithm for Service Levels Model

6.5.1 The Approach

Using the simulation model presented previously the influence of request content awareness over the performance of the load distribution algorithm was examined. The influence of sharing the resources for all service-levels was also evaluated.

For complex systems, where the system load information will be represented by a vector of values instead of a single value, the overhead introduced by the decision mechanism can become significant, especially for systems with a single point of decision (see section 5.8). In order to minimize the delay, a two layered distribution algorithm is proposed. The first layer will be a weighted round robin implementation. The second layer will implement a more complex load analysis algorithm and will be used for continuously tuning the weighting factor. This system has the advantage that it combines the reduced distribution mechanism delay of simple algorithms with the performance of the complex algorithms. It represents a further step for validating the proposed framework. The two

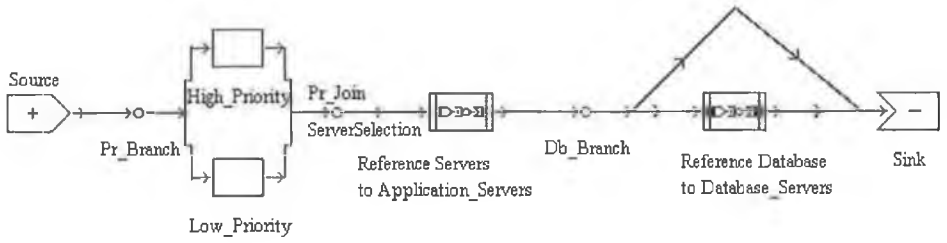

Figure 6.11: Reviewed Simulation Model.

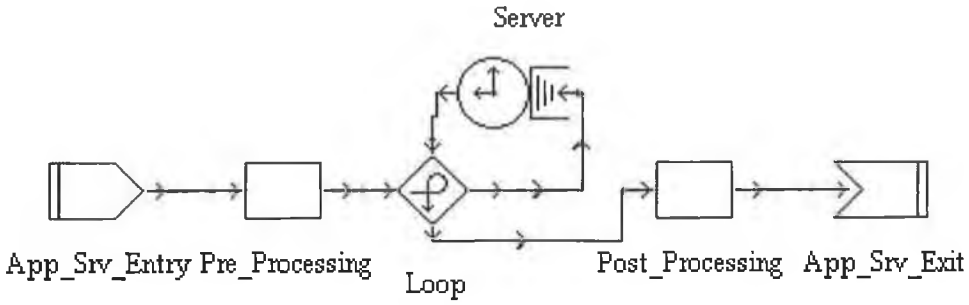

Figure 6.12: Reviewed Simulation Model, Application Server Logic.

layers of the proposed algorithm correspond to the load distribution module: the first layer of the algorithm (weighted round robin) corresponds to simple distribution algorithm included in the distribution module for ensuring high availability and the second layer of the algorithm represents the algorithm selected by the load evaluator module (it offers the required mechanism for changing the distribution algorithm at runtime).

6.5.2 The Implementation

The model was redesigned so that it allows system scalability, see Figure 6.11. instead of the five servers, an array of Application Server nodes (see Figure 6.12) was implemented, having the array dimension a simulation parameter. An array of Database Server nodes (see Figure 6.13) was also defined for simulating the database delays.

The distribution mechanism is implemented in the *High_Priority* and *Low_Priority* nodes, when transactions are labelled. Every time a new

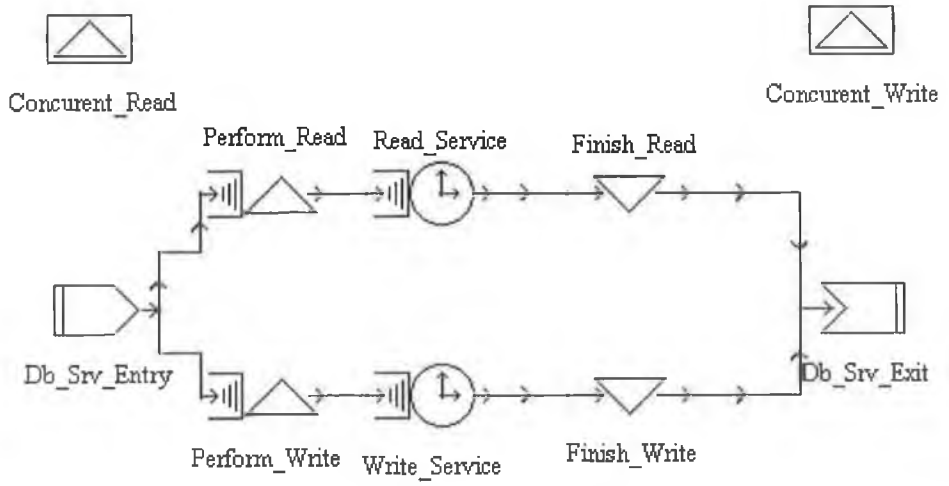

Figure 6.13: Reviewed Simulation Model, Database Server Logic.

transaction enters the system, the weighted round robin mechanism is used for selecting the destination server. The weights are automatically adjusted with a certain frequency (the frequency is a simulation parameter). The update is not done every time a transaction enters the system in order to simulate the way a real system is running.

The model is designed in such a way that it prevents any server from reaching an unstable state. The $MaxSrvLoad$ from

$$\frac{1}{MaxSrvLoad - CrtSrvLoad}$$

(see 6.4.2) is dynamically updated so that under no circumstances $CrtSrvLoad$ could get higher. If the $CrtSrvLoad$ reaches a higher value than $MaxSrvLoad$, $MaxSrvLoad$ is modified to be equal with $CrtSrvLoad$ plus 100. The value of 100 was determined by simulations and selected in such a way that the affected node will eventually be able to process the workload it has, albeit with severely degraded performance, since no possibility of dropping existing workload was included (as some overloaded real systems would do). In order to ensure that the processing capacity for all servers in the system remains the same, the $MaxSrvLoad$

is updated at the same time for all servers.

The purpose of the model and simulations is to determine the behavior of the load managed system under high loads. Thus if the $MaxSrvLoad$ keeps on increasing, this goal would not be achieved (since after the system processes the existing workload it keeps taking advantage of the higher processing power). In order to avoid this, a new function has been defined for reducing the $MaxSrvLoad$ value once the high load is processed. The function evaluates the difference between $MaxSrvLoad$ and $CrtSrvLoad$ and once it gets over 125 (thus the load on the server decreases) it is reduced to 75. Transaction load is based on a uniform distribution function with a minimum of 5 and a maximum of 10.

Selecting the optimal weight for the servers is the most important task. The weight is split in two parts, a fixed value part and a variable one. The variable one adjusts the overall weight taking into account current system load while the fixed part ensures that the total number of transactions is evenly distributed between the servers. The fixed part of the weight has a value of 1 for the servers processing only low-priority transactions. For the servers processing high priority transactions the following equation has been determined for computing the associated weight:

$$\frac{P}{K} + \frac{100 - P}{N - (1 - W) * K} = \frac{100}{N}$$

where:

- P represents the percentage of high-priority transactions
- N represents the number of servers
- K represents the number of servers processing high priority transactions
- W represents the weight for the server

The percentage of high-priority transactions P must fulfill the condition

$$P < K * \frac{100}{N}$$

so that the servers processing high priority transactions are not overloaded.

Since the weight has to be determined given a specific percentage of high-priority transactions and a number of servers, the previous equation can be rewritten as:

$$W = \frac{100 * K^2 - N * P * K}{N^2 * P - 2 * N * P * K + 100 * K^2}$$

The weight obtained using this equation represents the lower limit for the weight assigned to the high priority server. It does not follow a linear path since when high priority transactions enter the system, the processing for adjusting the weights (the variable part) adds an additional overhead that must be taken into account.

In the model it is considered to be a parameter and it is initialized when the simulation starts. The dynamic weight, computed at runtime considering the load on all servers and it is validated by comparing it with the fixed weight. If the dynamic weight is lower than the fixed one, the value of the fixed weight is used. The average percentage of high priority transactions (parameter of the simulation) was used in the simulations when the fixed weight was computed, in order to reduce the complexity of the model. This choice was made because a uniform distribution was used for selecting the high priority transactions.

Chapter 7

Simulation Model Results

7.1 Introduction

This chapter presents the results obtained for the evaluated algorithms, using the simulation models and configurations described in the previous chapter. Inferred conclusions from analyzing the results are also presented.

7.2 Simulation Methodology

For all presented results a set of three runs was done and the average of the three runs is presented in the graphs.

The simulation time was selected to have a fixed value so that the number of transactions processed and their distribution can be compared. The value of the simulation time is selected so that it is substantially higher than the one required by the simulation tool for obtaining a confidence level of 99% with accuracy of 1% of the mean. Several tests were performed using all algorithms in order to make sure that the selected simulation time maintains a confidence level of 99% with accuracy of 1% of the mean, at most.

Three runs have been performed for all simulations to further increase

the confidence in the obtained results.

7.3 Evaluation of the Load Distribution Algorithms

The first set of results is based on the simulation model presented in section 6.4.

The following configuration is used:

- five servers are considered,
- the client requests are generated using a poisson distribution with a mean value of 0.2 time units (TU),
- the load associated with every client request is generated using a Lognormal distribution with a mean value of 3 and a standard deviation value of 3,
- the percentage of high priority transaction is set to 0,

The delays introduced by the decision mechanism were excluded from all algorithm evaluation tests since the purpose of the tests was to determine the performance improvements in the transaction processing time. These delays can be estimated by evaluating the complexity of the decision algorithm.

For the round robin algorithm, the destination is always known when the transaction enters the system. After it is dispatched, the destination server number is increased with one unit and a test is done in order to see if the resulted value is greater than the number of servers, in which case it is changed to point to the first server.

The results are presented in Figure 7.1 and Table 7.1. The performance of the load balanced distribution algorithm offers an improvement of 34.8% in the response time.

Algorithm	Response Time [TU]
Round Robin	1.02
Weighted Round Robin	0.712
Load Balanced	0.665

Table 7.1: Algorithm Response Time Comparison

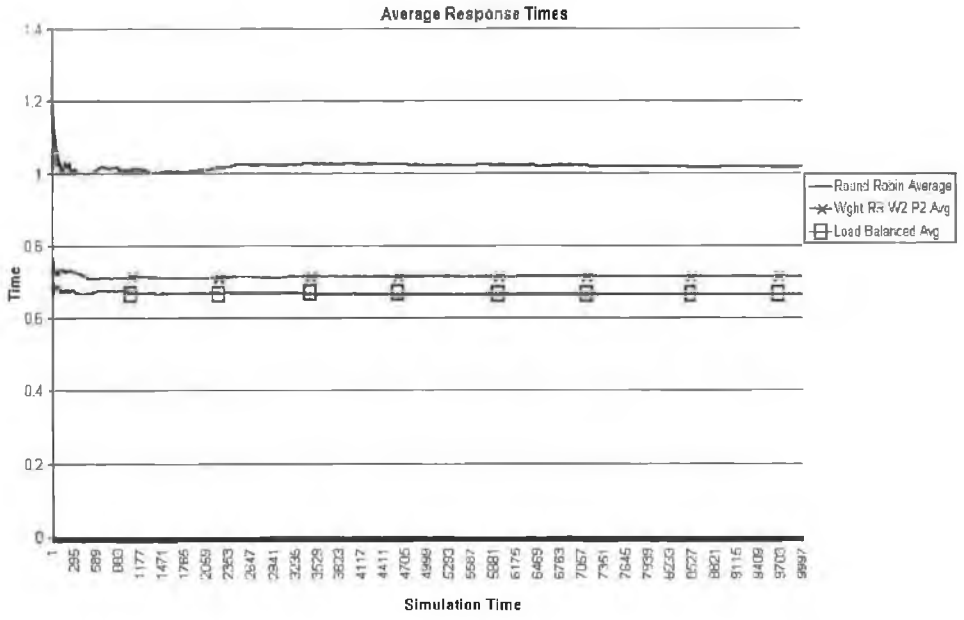

Figure 7.1: Average response time evolution for different distribution algorithms.

The weighted round robin algorithms' complexity is comparable with the complexity of the load balanced algorithm (both have to locate a minimum/maximum value in a vector with the same size). While the load balanced algorithm introduces an overhead by computing and sending load information to all server it clearly outperforms the weighted round robin algorithm.

7.4 Service Levels Simulation Results

In order to ensure better response times for the high priority transactions, several tests were performed, with different weighting factors assigned to

The Case	Priority	Response Time [TU]
Separate	High	0.273
Separate	Low	2.292
Weight 2	High	0.456
Weight 2	Low	0.9
Weight 3	High	0.425
Weight 3	Low	0.97
Weight 4	High	0.415
Weight 4	Low	1.015

Table 7.2: Separate vs. Shared Server Weight Influence

the first server (weight 2, 3 and 4) while the other four servers had the weight 1. Considering the case when the weighting factor was 2, the expected improvement would be that instead of running all low priority transactions on the four servers (25% on each machine), they would be distributed so that the four machines would each receive 22.22%, while the fifth one (serving high priority transactions) would have to serve 11.11% of the low priority transactions.

For the high priority transactions no distribution algorithm is required since all of them are processed on *Server1*.

A first set of tests was done for evaluating the influence of the weighting factor for the first server over the response time, with the percentage of high priority transactions fixed at 5%. The results are shown in Figure 7.2 and the steady-state values are presented in Table 7.2.

The weighting factor is used for controlling the load and thus the response time of the first server. As the weighting factor increases, the load on the server decreases, leading to better response times for the first server. At the same time, this will lead to less workload transfer from the remaining four servers to the first one, thus a lower improvement in the response time for the low priority transactions.

Compared with the separate server situation, for weight 2, an increase of 66.4% (0.183 Time Units-TU) in the average response time for high

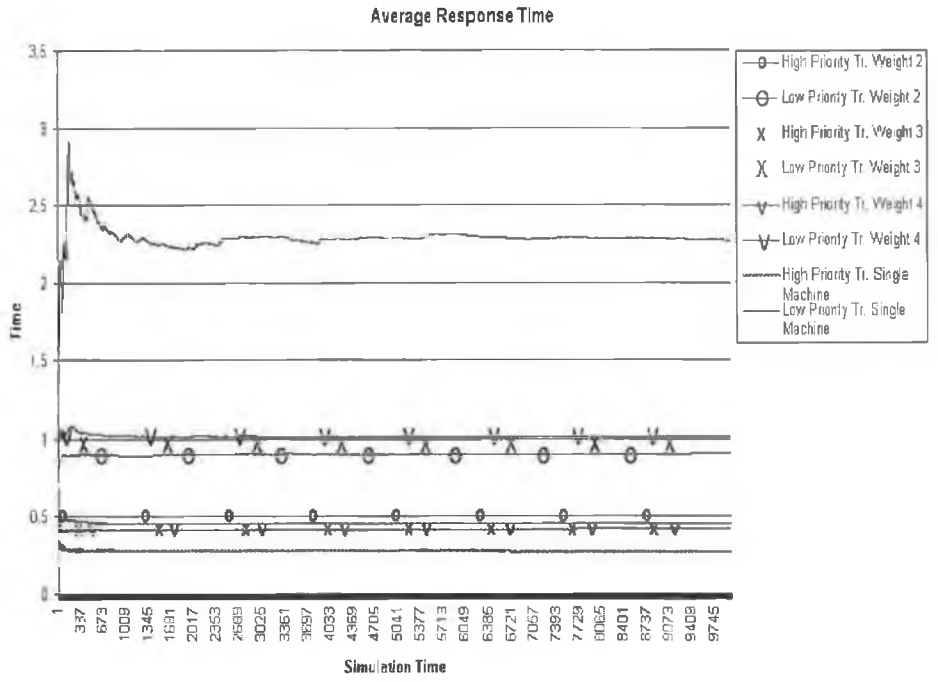

Figure 7.2: Average response time evolution for separate server vs. different weights.

priority transactions leads to an decrease of 60.73%(1.392 TU) in the average response time for low priority transactions.

For weight 4, an increase with 52% (0.142 TU) in the average response time for high priority transactions leads to an decrease with 55.73% (1.278 TU) in the average response time for low priority transactions.

It is seen that while the different approach has a severe influence (considering the difference from the single server to the weight 2), the weighting factor has to be tuned with care in order to ensure that the response time for the high priority transactions remains in the specified margins while the maximum number of low priority transactions will be processed by *Server1*.

The second set of tests was done for evaluating the influence of high priority transactions percentage over the performance improvements, considering the weight assigned to the first server fixed (equal to 2). The

The Case	High Priority %	Priority	Response Time [TU]
Separate	5%	High	0.274
Separate	5%	Low	2.292
Separate	10%	High	0.347
Separate	10%	Low	1.623
Separate	15%	High	0.425
Separate	15%	Low	1.325
Weighted	5%	High	0.456
Weighted	5%	Low	0.9
Weighted	10%	High	0.506
Weighted	10%	Low	0.875
Weighted	15%	High	0.576
Weighted	15%	Low	0.846

Table 7.3: Separate vs. Shared Server High Priority Percentage Influence

steady-state results are shown in Table 7.3.

The percentage of high priority transactions influences the performance of the Weighted Least-Loaded, by reducing system load imbalance in the separate server case, thus reducing the need of a load distribution mechanism. For 20% high priority transactions, the separate server system would act like the separate server system since one fifth of the transactions (the workload received by every server) is high priority transactions. Thus, it is expected that the best improvements are obtained for low percentages of high priority transactions.

It can be observed that, for 5% high-priority transactions, an increase of 66.4% (0.182 TU) in the average response time for the high priority transactions leads to a decrease of 60.73% (1.393 TU) in the average response time for the low-priority transactions.

For 10% high-priority transactions, a 45.82% increase (0.159 TU) in the average response time for the high priority transactions leads to a 46.09% decrease (0.748 TU) in the average response time for the low-priority transactions.

In the case of 15% high-priority transactions, the 35.53% (0.151 TU) increase in the average response time for the high priority transactions

leads to a 36.15% (0.479 TU) decrease in the average response time for the low-priority transactions is observed.

7.5 Two-Layered Algorithm Simulation Results

The unstability protection mechanism (presented in section 6.5) has been disabled for a first set of tests in order to obtain accurate results that can validate the system. This was necessary since if the system adapts the maximum server load in a dynamic way the results for the round robin and load balanced can not be directly compared (the system uses different service times during the same simulation period for different algorithms and thus the system throughput varies).

7.5.1 Round Robin Algorithm vs. Two Layered Load Balanced Algorithm, Without Priorities

A first set of simulation tests has been done, considering no high priority transactions, for validating the model.

The test results are also used for evaluating the performance improvements of the two layered load distribution algorithm in comparison with the round robin algorithm. The following configuration was considered for the model described in section 6.5.2:

- five homogeneous servers are considered,
- the client requests are generated using a poisson distribution with a mean value of 0.025 time units (TU),
- the load associated with every client request is generated using a log-normal distribution with a mean value of 5 and a standard deviation value of 10,

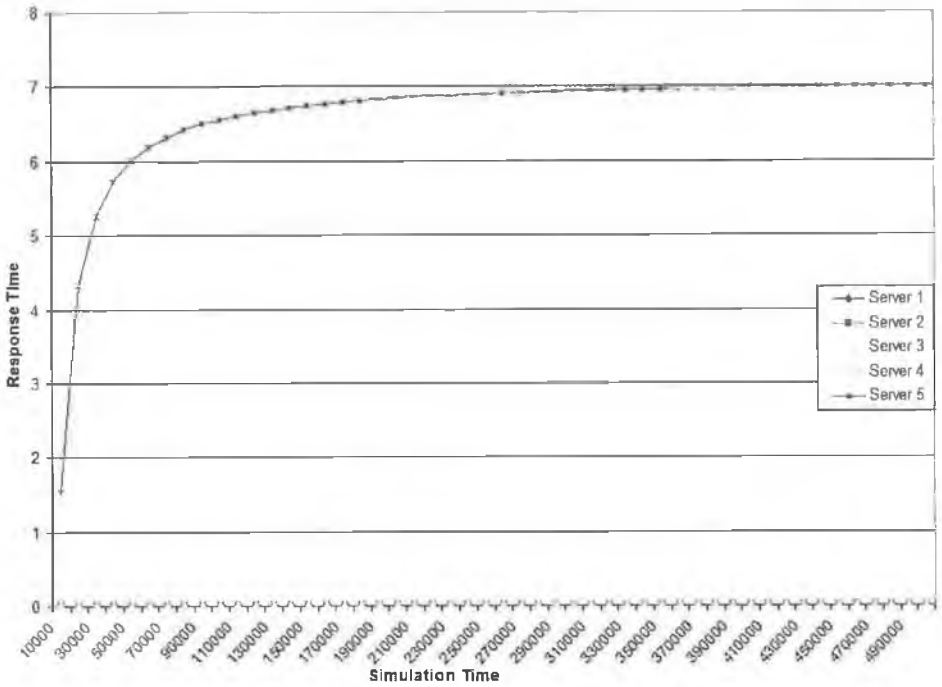

Figure 7.3: Round Robin Average Response Time, no priorities

Server	Response Time [TU]
Server 1	0.014
Server 2	0.014
Server 3	0.014
Server 4	0.014
Server 5	6.549

Table 7.4: Round Robin Server Average Response Time, no priorities

- the percentage of high priority transaction is set to 0,
- for the two layered load balanced algorithm the load is refreshed every 0.25 TU
- every fifth transaction has its load increased three times (for simulating a worst-case scenario for the round robin algorithm)

The results are presented in figure 7.3 and 7.4. The average response times for all servers are presented in table 7.4 and 7.5. The total number of transactions processed by the servers is presented in table 7.6 and 7.7.

Server	Response Time [TU]
Server 1	0.021
Server 2	0.02
Server 3	0.019
Server 4	0.019
Server 5	0.02

Table 7.5: Load Balanced Server Average Response Time, no priorities

Server	Number of Transactions
Server 1	3998923
Server 2	3998923
Server 3	3998923
Server 4	3998923
Server 5	3998863
Total:	19994155

Table 7.6: Number of Transactions Processed, Round Robin Case, no priorities

Server	Number of Transactions
Server 1	10277495
Server 2	6164209
Server 3	2620616
Server 4	776032
Server 5	169059
Total:	20007411

Table 7.7: Number of Transactions Processed, Load Balanced Case, no priorities

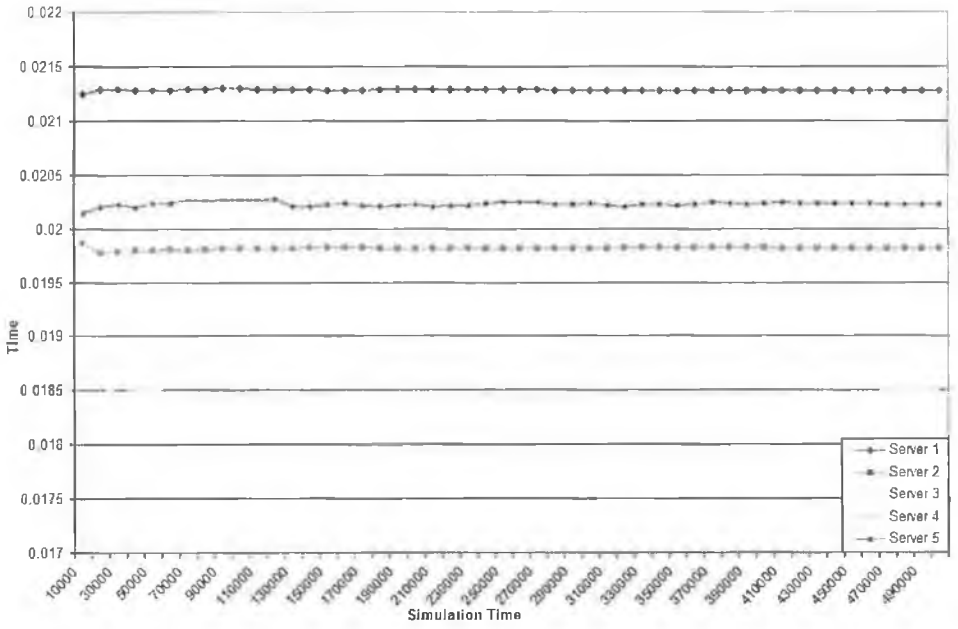

Figure 7.4: Load Balanced Average Response Time, no priorities

From the results presented in table 7.4 it can be seen that the first four servers are underutilized while the last one is overloaded.

The situation changes when the load balanced algorithm is used, as seen in table 7.5. The average response time for the load balanced algorithm is about the same for all servers.

In the round robin case, the number of transactions (see table 7.6) the same number of transactions reached all servers but the fifth server, being overloaded, introduces a large delay in processing the load and thus it served a slightly lower number of transactions. After the service has reached its maximum service time (the overload limit) the number of transactions precessed in parallel keeps on increasing. Due to the particularity of the system that all transactions are processed in parallel the total number of transactions processed is similar for all servers (the difference in the response time causes the small difference).

In the load balanced case, the number of processed transactions (see table 7.7) varies with the server since the system is underloaded and if

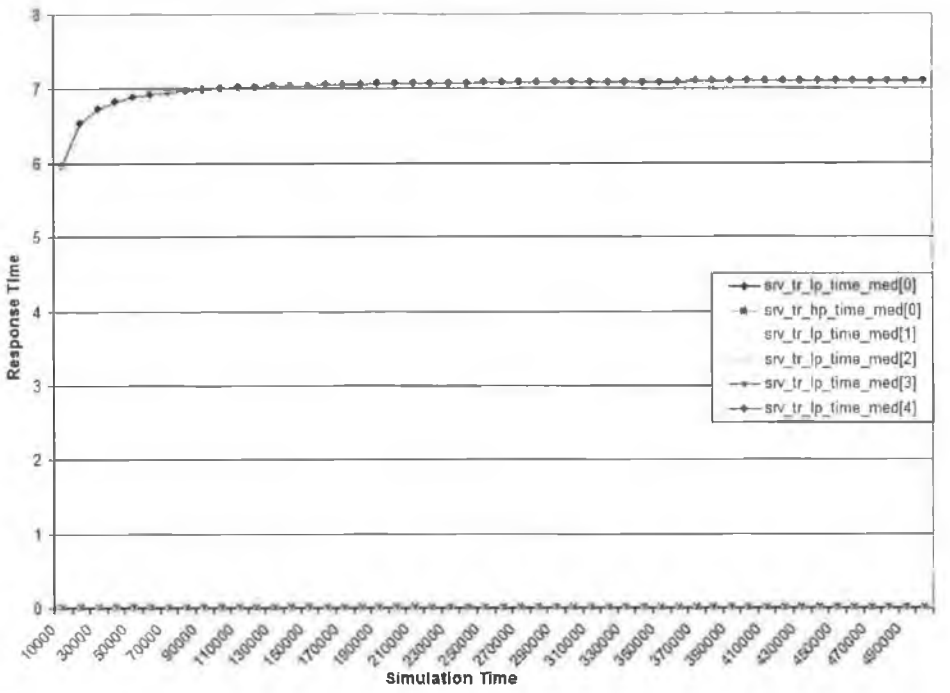

Figure 7.5: Round Robin Average Response Time, 10 percent high priority transactions, Weight 5

all servers are idle, the new transactions are distributed starting with the first server.

7.5.2 Round Robin Algorithm vs. Two Layered Load Balanced Algorithm, With 10 Percent High Priority Transactions and Weight 5

For the second set of tests, the same configuration was used for the model, only the percentage of high priority transactions was set to 10 instead of 0. A weight of 5 was assigned to the first server. This is expected to lead to an even distribution of load on all servers, according to the formula presented in section 6.5.2.

The results are presented in figure 7.5 and 7.6. The average response times for all servers are presented in table 7.8 and 7.9. The total number of transactions processed by the servers is presented in table 7.10 and

Server	Response Time [TU]
Server 1	0.014
Server 2	0.014
Server 3	0.014
Server 4	0.014
Server 5	6.969

Table 7.8: Round Robin Server Average Response Time, Weight 5

Server	Response Time [TU]
Server 1 High Priority	0.015
Server 1 Low Priority	0.021
Server 2	0.02
Server 3	0.019
Server 4	0.018
Server 5	0.02

Table 7.9: Load Balanced Server Average Response Time, Weight 5

Server	Number of Transactions
Server 1 High Priority	1998189
Server 1 Low Priority	3600150
Server 2	3600149
Server 3	3600149
Server 4	3600149
Server 5	3600101
Total:	19998887

Table 7.10: Number of Transactions Processed, Round Robin Case, Weight 5

Server	Number of Transactions
Server 1 High Priority	2000386
Server 1 Low Priority	9340990
Server 2	5700197
Server 3	2239582
Server 4	601917
Server 5	117916
Total:	20000988

Table 7.11: Number of Transactions Processed, Load Balanced Case, Weight 5

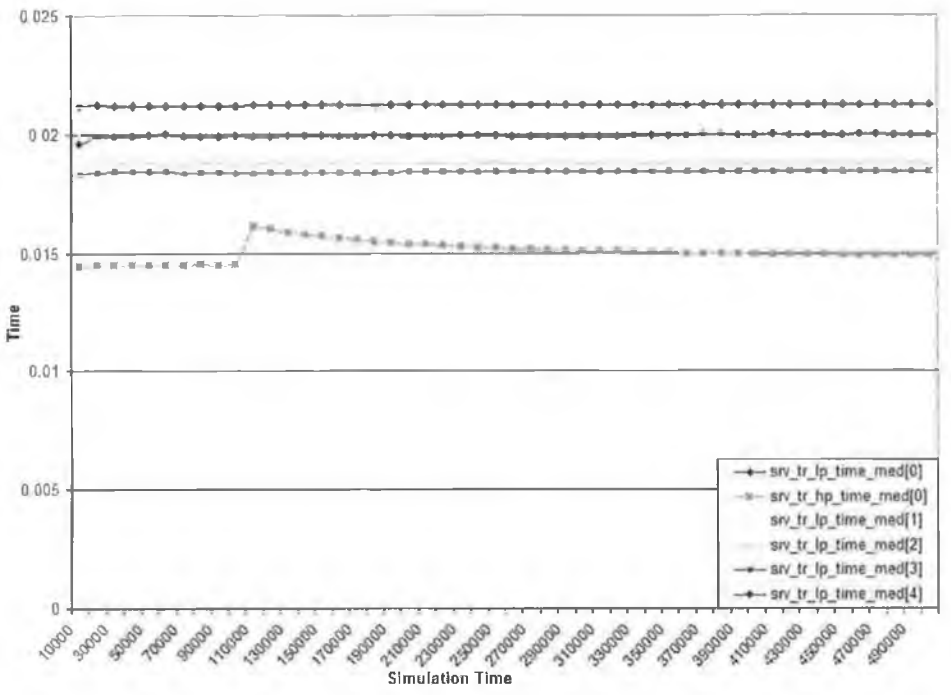

Figure 7.6: Load Balanced Average Response Time, 10 percent high priority transactions, Weight 5

7.11.

From the results presented in table 7.8 and 7.9 it can be seen that the behavior in the previous section is maintained, as expected.

For the round robin case (table 7.8) the first four servers are underutilized while the last one is overloaded.

The situation changes when the load balanced algorithm is used (table 7.9), the system response time becomes balanced. The average response time for the load balanced algorithm, for the low priority transactions, is about the same for all servers.

While the average response time for the low priority transactions increase from 0.14 TU to about 0.20 TU in the load-balanced case, for the fifth server it decreases significantly, from 6.969 TU to 0.2 TU. For the high priority transactions, the response time maintains approximately the same level, with a very slight increase to 0.15 TU.

In the round robin case, the 19998887 transactions are evenly dis-

tributed to the five servers (see table 7.10) on average the same number of transactions reaching all servers. The fifth server, being overloaded, introduces a large delay in processing the load and thus it served a slightly lower number of transactions. After the service has reached its maximum service time (the overload limit) the number of transactions precessed in parallel keeps on increasing.

In the load balanced case, the number of processed transactions (see table 7.11) varies with the server. The system is underloaded and thus, if all servers are idle, new transactions are distributed starting with the first server. The difference between the number of transactions processed by the first and the last server can be used as an indication of the load of the system. If the system is underloaded the difference is high (as in this case) while if the system is highly loaded the difference should be small.

The obtained results are consistent with the results presented in section 7.4.

7.5.3 Round Robin Algorithm vs. Two Layered Load Balanced Algorithm, With 10 Percent High Priority Transactions and Weight 10

Another set of tests has been done using the same configuration for the model, with 10 percent of high priority transactions. The weight assigned to the first server was increased to 10. This is expected to lead to an even distribution of load on all servers, and to a complete elimination of extra delays for the high priority transactions (in the load balanced case an extra 0.001 TU was added, in average, to the high priority transactions response time).

The results are presented in figure 7.7 and 7.8. The average response times for all servers are presented in table 7.12 and 7.13. The total number

Server	Response Time [TU]
Server 1	0.014
Server 2	0.014
Server 3	0.014
Server 4	0.014
Server 5	5.794

Table 7.12: Round Robin Server Average Response Time, Weight 10

Server	Response Time [TU]
Server 1 High Priority	0.014
Server 1 Low Priority	0.021
Server 2	0.02
Server 3	0.019
Server 4	0.018
Server 5	0.02

Table 7.13: Load Balanced Server Average Response Time, Weight 10

Server	Number of Transactions
Server 1 High Priority	2000525
Server 1 Low Priority	3600234
Server 2	3600234
Server 3	3600234
Server 4	3600234
Server 5	3600187
Total:	20001648

Table 7.14: Number of Transactions Processed, Round Robin Case, Weight 10

Server	Number of Transactions
Server 1 High Priority	1998768
Server 1 Low Priority	9339070
Server 2	5702613
Server 3	2238904
Server 4	601506
Server 5	117581
Total:	19998442

Table 7.15: Number of Transactions Processed, Load Balanced Case, Weight 10

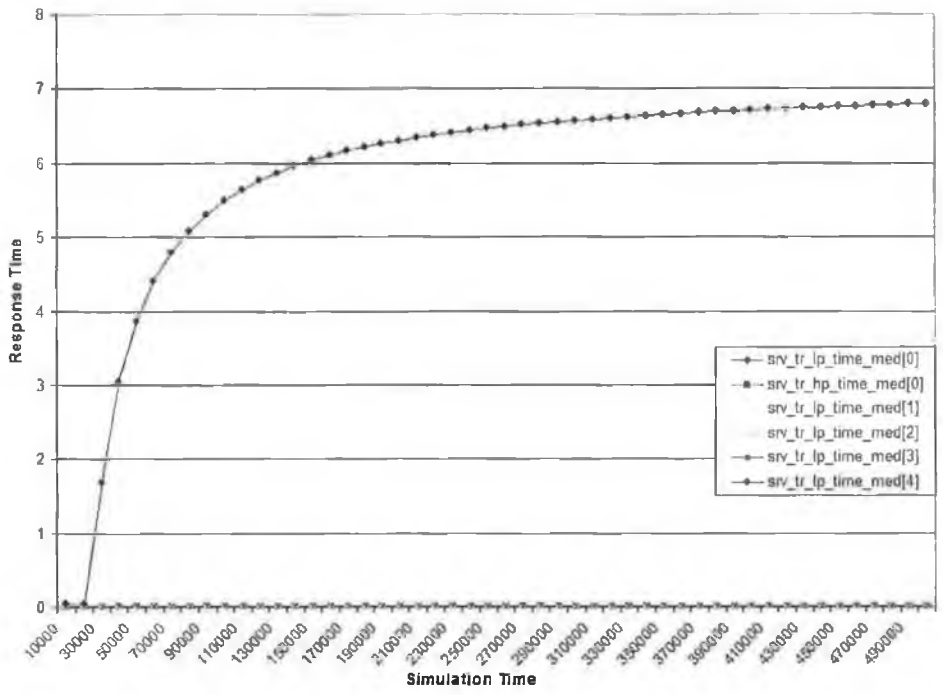

Figure 7.7: Round Robin Average Response Time, 10 percent high priority transactions, Weight 10

of transactions processed by the servers is presented in table 7.14 and 7.15.

A behavior similar to the first two cases can be observed from the results presented in table 7.12 and 7.13.

For the round robin case (table 7.12) the first four servers are under-utilized while the last one is overloaded, as in the previous scenarios.

When the load balanced algorithm is used (table 7.13), the system response time becomes balanced. The average response time for the load balanced algorithm, for the low priority transactions, ranges between 0.18 and 0.21 TU thus is fairly well balanced.

While the average response time for the low priority transactions increase from 0.14 TU to a value between 0.18 and 0.21 TU in the load-balanced case, for the fifth server it decreases significantly, from 5.794 TU to 0.2 TU. At the same time, for the high priority transactions, the response time maintains a similar level to the previous set of tests, having the same value as for the round robin case, 0.14 TU. It ca be concluded

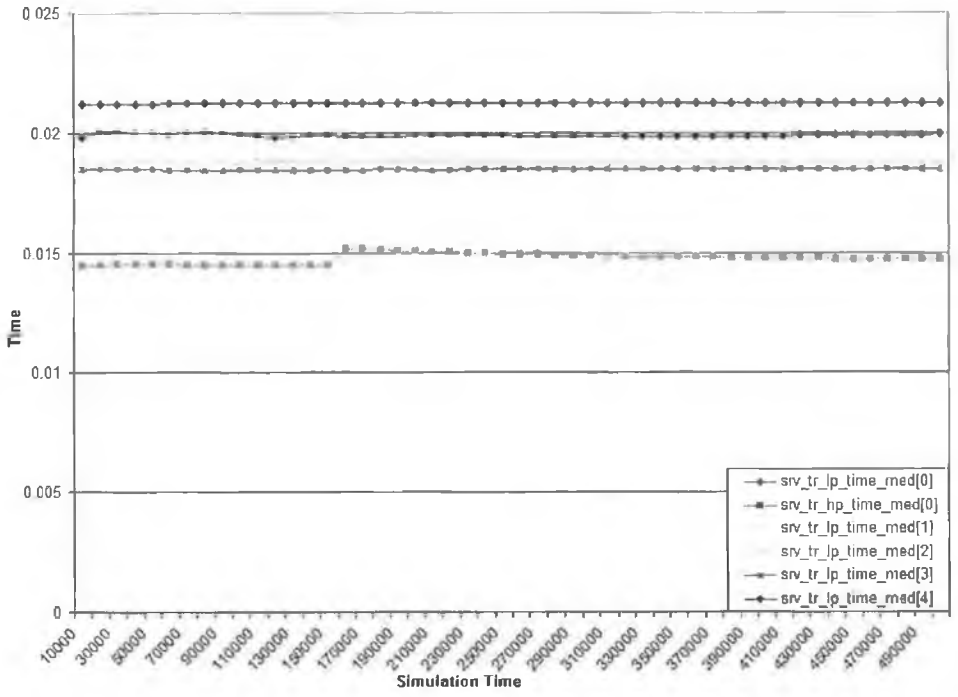

Figure 7.8: Load Balanced Average Response Time, 10 percent high priority transactions, Weight 10

that if an optimal value is determined for the weight, the influence of the algorithm over the high priority transactions can be almost completely eliminated.

In the round robin case, the 20001648 transactions are evenly distributed to the five servers (see table 7.10) on average the same number of transactions reaching all servers. Due to the overload, the fifth server introduces a large delay in processing the load and thus it serves a slightly lower number of transactions. After the service has reached its maximum service time (the overload limit) the number of transactions waiting to be precessed (in the queue) keeps on increasing.

In the load balanced case, the number of processed transactions (see table 7.11) varies with the server. The total number of transactions serviced reduces as the server number increases. This is due to the fact that the system is underloaded and thus, if all servers are idle, new transactions are distributed starting with the first server. The difference between

the number of transactions processed by the first and the last server can be used as an indication of the load of the system. If the system is underloaded the difference is high (as in this case - 9400000 compared with 117500) while if the system is highly loaded the difference should be small, taking into account that the load in this case follows a lognormal distribution.

7.5.4 Round Robin Algorithm vs. Two Layered Load Balanced Algorithm, Without Priorities, adjusting service time

The unstability protection mechanism (presented in section 6.5.2) has been enabled for the following tests in order to increase the overall load on the system. The load increase is achieved by adjusting the MaxSrvLoad parameter, as presented previously. The mechanism will detect the server with the highest load and use it as a reference, considering it as close to its overload limit.

A first set of tests has been done with no high priority transactions to revalidate the model and evaluate the performance improvements of the two-level load-balanced algorithm presented in section 6.5 over the simple round-robin algorithm, using the adaptive service time and thus a higher load. The following configuration was used: five servers, the client requests are generated using a poisson distribution with a mean value of 1/40 time units (TU), the load associated with every client request is generated using an lognormal distribution with a mean value of 5 and a standard deviation value of 10 and the MaxSrvLoad is reevaluated every 2 TU. In order to simulate the worst case for the round robin algorithm, every fifth transaction load is increased three times. While for the round robin algorithm this means that one server (the fifth one) will receive on

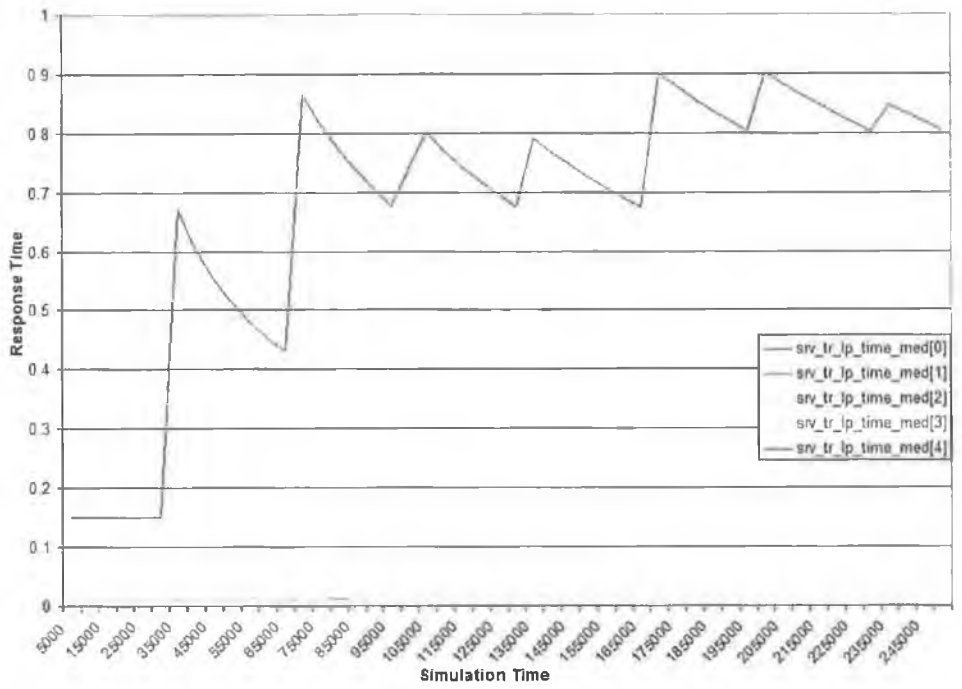

Figure 7.9: Round Robin Average Response Time.

Server	Utilization
Server 1	0.147
Server 2	0.095
Server 3	0.095
Server 4	0.095
Server 5	6.69

Table 7.16: Round Robin Server Average Utilization, no priorities, adaptive

average three times as much load as the others, for the load balanced algorithm the load should be mode equally distributed.

The results are presented in figure 7.9, 7.10 and 7.11.

The round robin case (Figure 7.9) the system is unbalanced since the fifth server receives a much higher load and thus it's average response time is much higher than for the other four servers.

It can be seen that the last server had a much higher load than the other four. The fifth server was overloaded since while for the first four servers the average utilization was 0.108 while for the fifth one the average

Server	Response Time [TU]
Server 1	0.013
Server 2	0.013
Server 3	0.013
Server 4	0.013
Server 5	0.72

Table 7.17: Round Robin Server Average Response Time, adaptive, no priorities

utilization was 6.69.

There was a number of 9006210 transactions processed, distributed uniformly among the five servers (every server served 1801242 transactions).

The average response times are presented in table 7.17. It is seen that while for the first four servers the response time is small, for the last one it is 55 times higher. The 0.72 TU is the higher limit for the system response time, due to the overload protection mechanism (unstability prevention). For this reason, the response time, alone, is not enough to compare the performance differences between the algorithms and the server utilization is required. In this case, the service time was constantly on a high limit thus the actual service time for the fifth node would be 1/125 multiplied with the transaction load (since the MaxLoad is considered to be, at overload limit, CrtSrvLoad + 125).

Two subsequent sets of tests ware realized for evaluating the two-layered load balanced algorithm.

The first test for the load balanced case (Figure 7.10) proved a large reduction of the load imbalance. In this scenario, the refresh rate for the load of the servers (and thus for the weight assigned to the servers) was set to 2 time units, that could be translated to the system distributing an average number of 80 transactions between two successive load updates.

It is noticed that the maximum server utilization dropped from 6.69 to

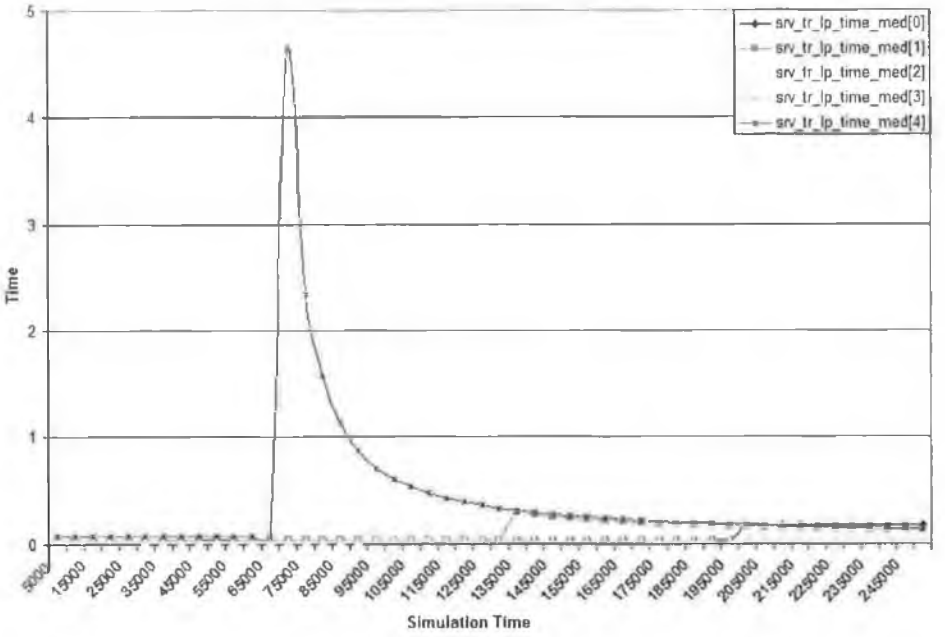

Figure 7.10: Load Balanced Average Response Time V1.

Server	Utilization
Server 1	2.893
Server 2	1.179
Server 3	0.369
Server 4	0.221
Server 5	0.363

Table 7.18: Load Balanced Server Average Utilization, no priorities, adaptive

2.893 (see table 7.18). For both load balanced tests the load on the first server remains higher than for the rest because the fixed weight assigned to it (see section 6.5.2) had a value of 2, thus the first server would receive a higher load than the other servers.

It is seen that due to the fact that the first server had to process a higher load it had the highest utilization and response time. The fixed weight assigned to it was set to 2, a limit below the recommended minimum value of 5, according to the formula presented in section 6.5.2.

In this case, the MaxSrvLoad was modified only when the simulation started and when the fifth server received a very high load at once (as

Server	Response Time [TU]
Server 1	0.666
Server 2	0.105
Server 3	0.037
Server 4	0.037
Server 5	0.402

Table 7.19: Load Balanced Server Average Response Time, adaptive, no priorities

Server	Number of Transactions
Server 1	2187070
Server 2	2342814
Server 3	2361599
Server 4	1456043
Server 5	652474
Total	9000000

Table 7.20: Number of Transactions Processed, load balanced, adaptive, no priorities

seen in Figure 7.10). The overall server service time was lower and this is why the first and last servers seem to perform worse than in the round robin case. If the same values for service time would have been used, the service times would represent at most 60% of the obtained value.

For the second set of tests, the load (and thus server weight) refresh rate was reduced to 1 TU, averaging a number of 40 transactions between two successive updates. The results are similar to the ones from the first set.

The utilization of the first server reduced to 1.992 while the utilization

Server	Utilization
Server 1	1.992
Server 2	1.809
Server 3	0.368
Server 4	0.263
Server 5	0.177

Table 7.21: Load Balanced Server Average Utilization, no priorities, adaptive

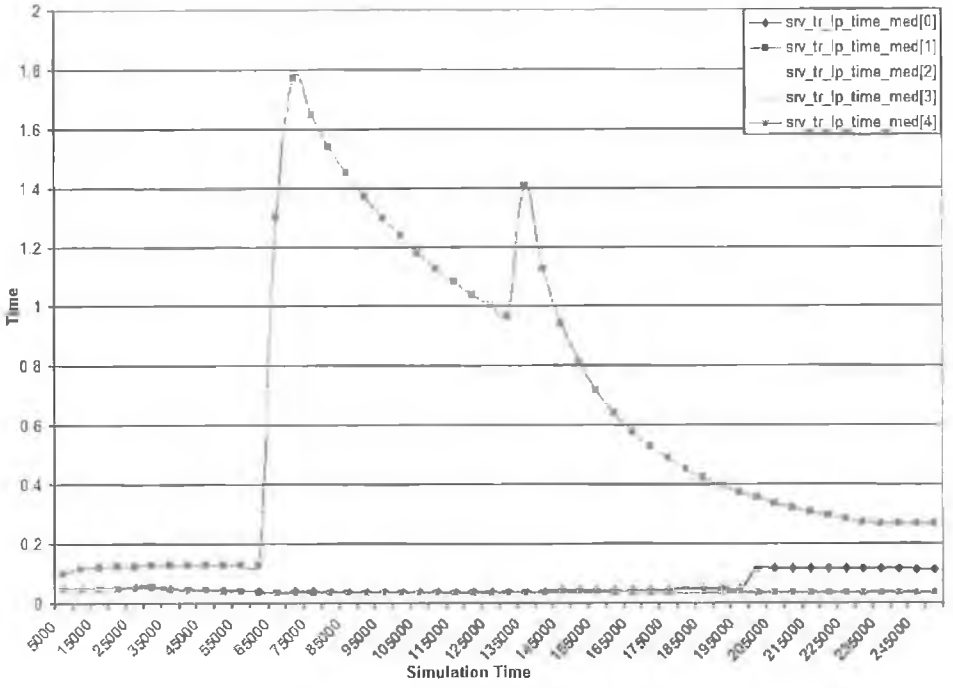

Figure 7.11: Load Balanced Average Response Time V2.

Server	Response Time [TU]
Server 1	0.059
Server 2	0.629
Server 3	0.04
Server 4	0.04
Server 5	0.04

Table 7.22: Load Balanced Server Average Response Time, adaptive, no priorities

of the second server increased with 0.63 (see table 7.21). For the other three servers, the utilizations maintained the low levels from the previous set of tests. In this case the first two servers have a similar utilization, although the number of precessed transactions differs (the first server processed 700000 transaction more than the second one). This is due to the load associated with the transactions and to the small weight assigned to the first server.

In this case, the second server was the overloaded one. This is due to the adaptive service time, since it received the highest load and the adap-

Server	Number of Transactions
Server 1	2127470
Server 2	1410611
Server 3	2420980
Server 4	1802686
Server 5	1238453
Total	9000000

Table 7.23: Number of Transactions Processed, load balanced, adaptive, no priorities

tive mechanism adjusted the load so that it will be close to the unstability limit.

Compared to the distribution of transactions in the first set of tests (with the adaptive service time disabled) in this case it can be noticed that the load was more evenly distributed (since the overall system load was higher).

7.5.5 Round Robin vs. Two Layered Load Balanced With Priorities, adjusting service time

A second set of tests was done, using a higher percentage of high-priority transactions (15%) and the load imbalance was increased so that every fifth transaction has its load increased 10 times. Based on the minimal recommended weight formula (see section 6.5.2), for the load balanced case two situations were considered: the server processing high priority transactions has a weight of 5 assigned for the first case and a weight of 10 for the second case. The system load (and thus weight) refresh frequency was increased to 1/20 TU.

It can be seen (from Table 7.24) that the fifth server has an average response time of 2 TU, 800 times higher than the average response time of the other four servers and the utilization for the first server is about 550 times higher than for the other servers (see Table 7.25).

Server	Response Time [TU]
Server 1 High Priority	0.0025
Server 2 Low Priority	0.0025
Server 3	0.0025
Server 4	0.0025
Server 5	2.028

Table 7.24: Round Robin Server Average Response Time, adaptive, 15 percent high priority

Server	Utilization
Server 1	0.031
Server 2	0.017
Server 3	0.017
Server 4	0.017
Server 5	14.304

Table 7.25: Round Robin Server Average Utilization, adaptive, 15 percent high priority

Server	Number of Transactions
Server 1 High Priority	1501385
Server 1 Low Priority	1700319
Server 2	1700318
Server 3	1700318
Server 4	1700318
Server 5	1700304
Total	10002962

Table 7.26: Number of Transactions Processed, Round Robin, adaptive, 15 percent high priority transactions

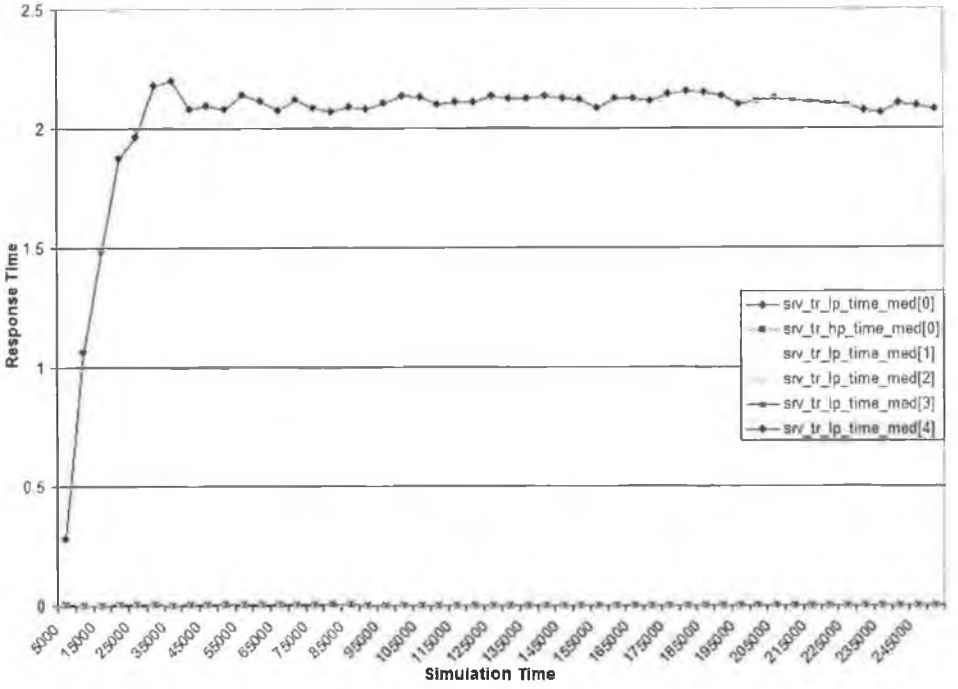

Figure 7.12: Round Robin Average Response Time.

The number of transactions (see Table 7.26) is evenly distributed among the five servers and, as expected, the overloaded server has a larger waiting queue (and thus slightly fewer transactions processed).

For the first load balanced test, a fixed weight of 5 was assigned to the first server. The load imbalance reduced so that the response time imbalance dropped from 800 times to a maximum difference of 18.054 times and an average difference of 5.473 times.

As it can be seen in Table 7.28, the utilization difference between the

Server	Response Time [TU]
Server 1 High Priority	0.158
Server 1 Low Priority	0.139
Server 2	0.668
Server 3	0.077
Server 4	0.082
Server 5	0.037

Table 7.27: Load Balanced Server Average Response Time, adaptive, 15 percent high priority, weight 5

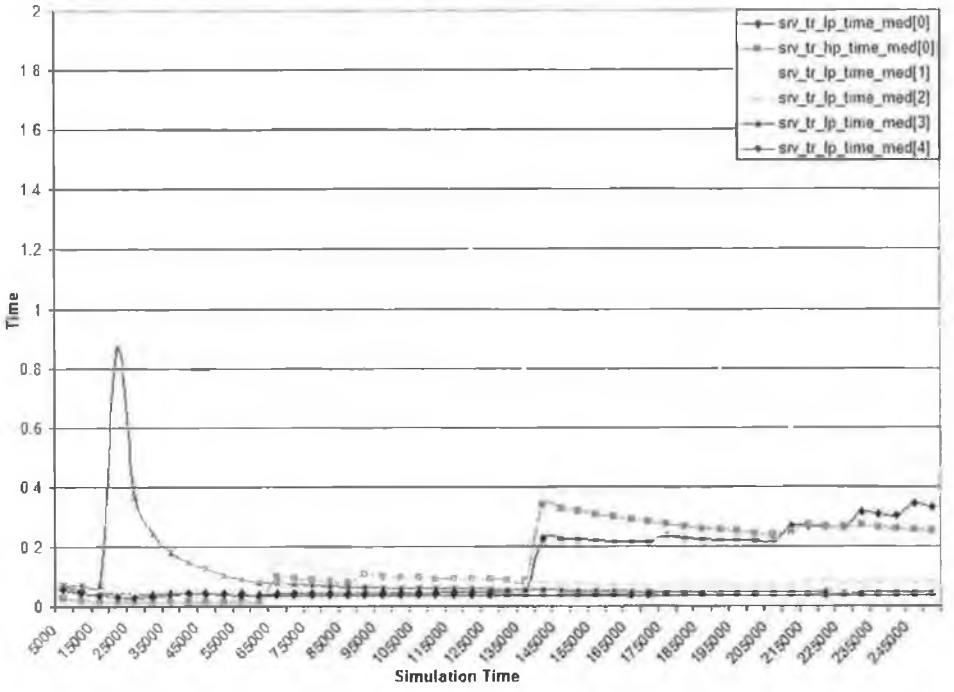

Figure 7.13: Load Balanced Average Response Time, Weight 5.

Server	Utilization
Server 1	3.466
Server 2	3.005
Server 3	0.704
Server 4	0.338
Server 5	0.194

Table 7.28: Load Balanced Server Average Utilization, adaptive, 15 percent high priority, weight 5

servers also decreased from 550 times to a maximum of 17.86 times, while the average utilization is 7.924 times lower than the maximum value.

Due to the different load on the transactions they are not distributed evenly among the servers, differences of up to 10% of the total number of transactions being noticed among the servers (see Table 7.29) A second set of tests was done, increasing the fixed weight assigned to the first server from 5 to 10.

In this case, the system imbalances reduce even further. For the response time, the maximum average response time imbalance is 17.449

Server	Number of Transactions
Server 1 High Priority	1501053
Server 1 Low Priority	1478939
Server 2	1421455
Server 3	2221511
Server 4	2079634
Server 5	1297193
Total	9999785

Table 7.29: Number of Transactions Processed, Round Robin, adaptive, 15 percent high priority transactions

Server	Response Time [TU]
Server 1 High Priority	0.132
Server 1 Low Priority	0.239
Server 2	0.855
Server 3	0.068
Server 4	0.050
Server 5	0.049

Table 7.30: Load Balanced Server Average Response Time, adaptive, 15 percent high priority, weight 10

times, lower than the 18.054 times value obtained using the weight of 5. The average response time imbalance has a value of 5.686 times, close to the 5.437 times value obtained using a fixed weight of 5. It is noticed that the maximum imbalance among the servers load has been further reduced when the weight was increased.

The utilization imbalance between the five servers has also reduced, from a maximum of 17.86 times to a maximum of 16.518 times. Com-

Server	Utilization
Server 1	2.097
Server 2	4.113
Server 3	0.676
Server 4	0.454
Server 5	0.249

Table 7.31: Load Balanced Server Average Utilization, adaptive, 15 percent high priority, weight 10

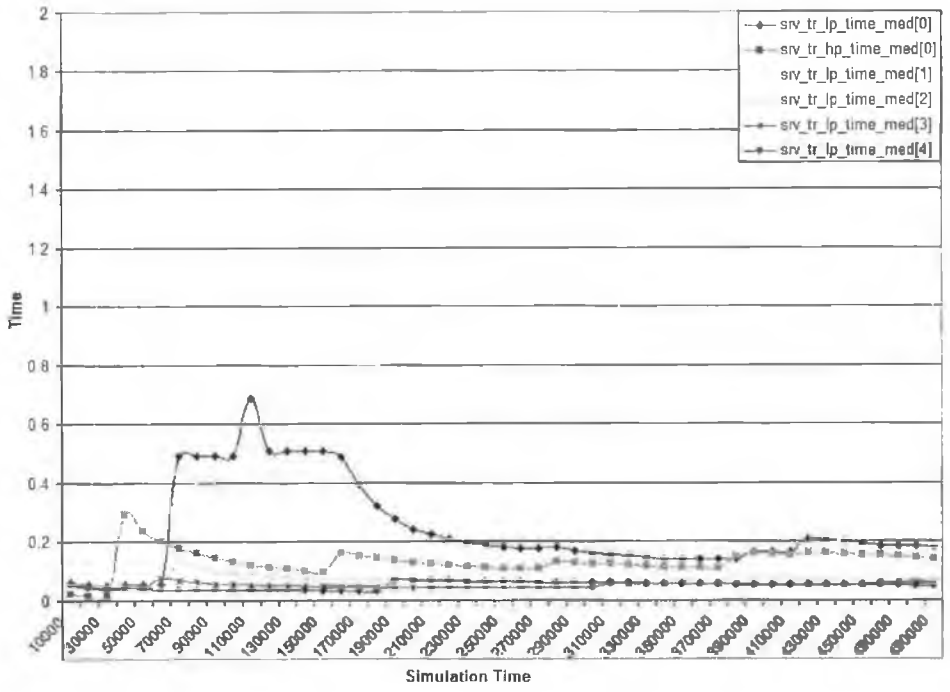

Figure 7.14: Load Balanced Average Response Time, Weight 10, V5.

paring the average utilization, the imbalance also reduced from a value of 7.924 times to a value of 6.095 times.

The simulation time for this set of tests was doubled and thus the total number of transactions being processed is twice the number of transactions being processed in the previous test (using a weight of 5). It can be seen that the number of transactions processed by the servers varies with the server, ranging from over 6 million transactions (for the first server) to

Server	Number of Transactions
Server 1 High Priority	2942505
Server 1 Low Priority	3279534
Server 2	1961834
Server 3	4962627
Server 4	3910594
Server 5	2541953
Total	19599047

Table 7.32: Number of Transactions Processed, Round Robin, adaptive, 15 percent high priority transactions

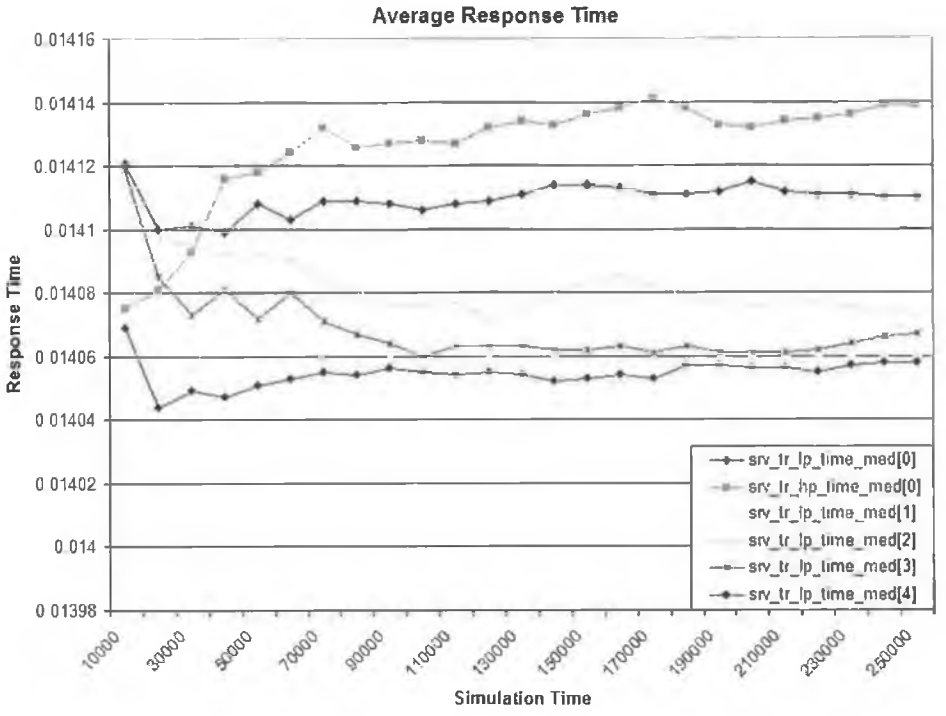

Figure 7.15: Round Robin Average Response Time, Uniform Load.

slightly less than 2 million transactions (for the second server).

7.5.6 Round Robin vs. Two Layered Load Balanced With Priorities, Balanced Load

Another set of tests was performed to evaluate the performance improvements of the two algorithms considering uniform workload, situation where we have expected the round-robin algorithm to be the best (since it adds the minimum distribution overhead).

The simulation conditions were kept similar to the previous experiments. The transactions are generated using a Poisson distribution, with a mean value of 0.025 TU. The load is generated based on a Lognormal distribution, with a mean of 5 and a standard deviation value of 10. For the load balanced case, the overhead of the decision algorithm was modelled as an increase with one unit of the load associated with the transaction.

Server	Response Time [TU]
Server 1 High Priority	0.014
Server 1 Low Priority	0.014
Server 2	0.014
Server 3	0.014
Server 4	0.014
Server 5	0.014

Table 7.33: Round Robin Server Average Response Time

Server	Utilization
Server 1	0.158
Server 2	0.101
Server 3	0.101
Server 4	0.101
Server 5	0.101

Table 7.34: Round Robin Server Average Utilization

Due to the light load, the response times for all servers are the same, and the utilization is also the same (excepting the first server, which also processes high priority transactions).

It can be seen (Figure 7.15, 7.16) that if the overhead is ignored, the two algorithms have the same performance. The response time and utilization is the same for both situations. The total number of transactions is evenly distributed between the five servers. Due to the fact that the total load on the system was low, the high priority transactions did not influence the distribution for the low priority transactions.

Server	Number of Transactions
Server 1 High Priority	998917
Server 1 Low Priority	1800025
Server 2	1800025
Server 3	1800024
Server 4	1800024
Server 5	1800024
Total	9999039

Table 7.35: Number of Transactions Processed, Round Robin

Server	Response Time [TU]
Server 1 High Priority	0.014
Server 1 Low Priority	0.014
Server 2	0.014
Server 3	0.014
Server 4	0.014
Server 5	0.014

Table 7.36: Load Balanced Server Average Response Time, No Decision Overhead

Server	Utilization
Server 1	0.158
Server 2	0.101
Server 3	0.101
Server 4	0.101
Server 5	0.101

Table 7.37: Load Balanced Server Average Utilization, No Decision Overhead

Server	Number of Transactions
Server 1 High Priority	1001069
Server 1 Low Priority	1799167
Server 2	1799167
Server 3	1799167
Server 4	1799167
Server 5	1799166
Total	9996903

Table 7.38: Number of Transactions Processed, Load Balanced, No Decision Overhead

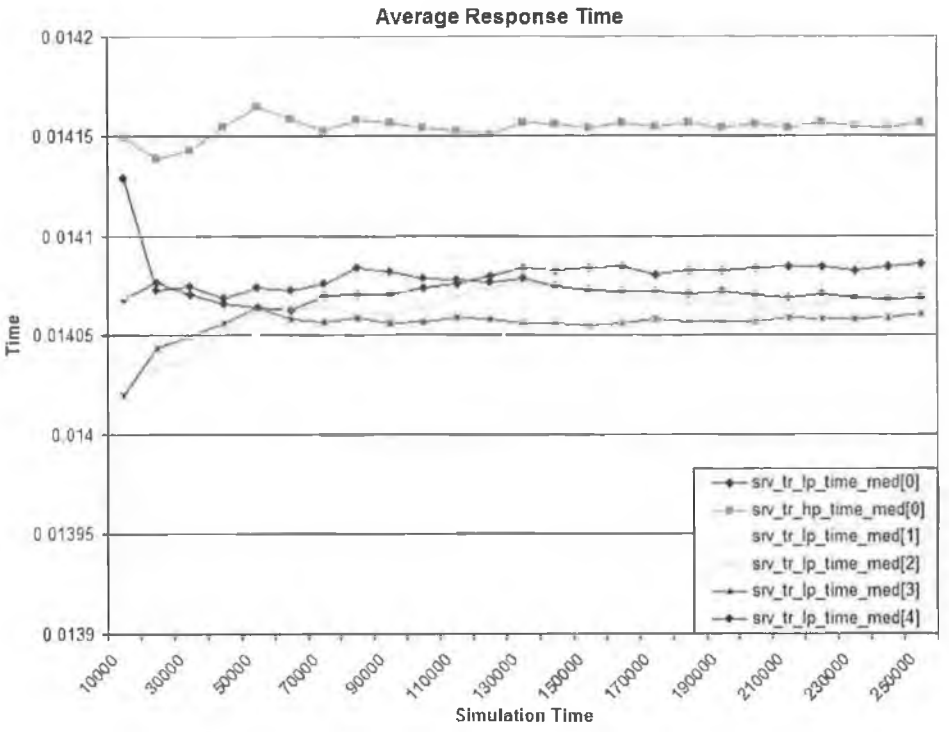

Figure 7.16: Load Balanced Average Response Time, Uniform Load.

Another test has been done taking into account the overhead introduced by the load distribution algorithm. It was considered that the overhead is similar to generating a new transaction, assigning it a load equal to the decision overhead and processing it on the server that represents the entry point in the system (the performance of the federated architecture configuration was not evaluated). The worst-case was selected for the tests, so that the shared server (processing both high- and low-priority transactions) is also considered to be the entry point thus having to process these additional transactions.

The load increase was considered to be 1 unit of load, which would represent an average of 20% of load. This apparently large value was selected since the transaction load on the system is very low, the server utilization having an average of 10% to 12% (thus, the real added load on the system is around 2%).

It can be seen from Table 7.39, in comparison with Table 7.33, that

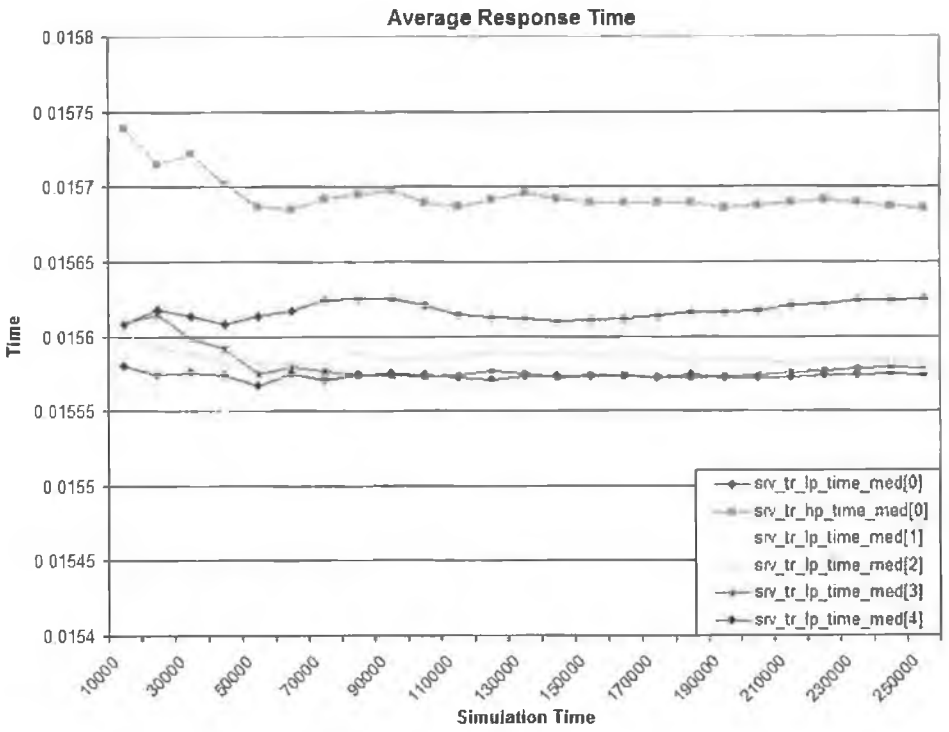

Figure 7.17: Load Balanced Average Response Time, Uniform Load, With decision overhead.

the average response time for the load balanced case, when considering the overhead, increases with 11.43%, from a value of 0.014 TU to a value of 0.0156 TU.

While the utilization for servers two to five remains the same, for the first it increases with 36.7%, from 0.158 to 0.216 (as seen from Tables 7.34, 7.40).

The total number of transactions processed by the system is about the

Server	Response Time [TU]
Server 1 High Priority	0.0156
Server 1 Low Priority	0.0157
Server 3	0.0156
Server 3	0.0156
Server 4	0.0156
Server 5	0.0156

Table 7.39: Load Balanced Server Average Response Time, With Decision Overhead

Server	Utilization
Server 1	0.216
Server 2	0.101
Server 3	0.101
Server 4	0.101
Server 5	0.101

Table 7.40: Load Balanced Server Average Utilization, With Decision Overhead

Server	Number of Transactions
Server 1 High Priority	999646
Server 1 Low Priority	1799723
Server 2	1799722
Server 3	1799722
Server 4	1799722
Server 5	1799722
Total	9998257

Table 7.41: Number of Transactions Processed, Load Balanced, With Decision Overhead

same, and the distribution between the five servers is the same for both algorithms. The high priority transactions are all processed by the first server while the low priority ones are equally distributed between the five servers.

The tests demonstrate that when the system receives requests with similar load, a more complex algorithm will offer lower performance than a simple one. Thus, these results (as seen in Graph 7.18) represent a valid argument supporting the need of a load evaluator module (as presented in section 5.6).

7.5.7 Round Robin vs. Two Layered Load Balanced With Priorities, Load Influence

The influence of the workload distribution over the performance of the load distribution algorithms was investigated. The same type of distribution function was used (Lognormal) and the mean value is kept constant having

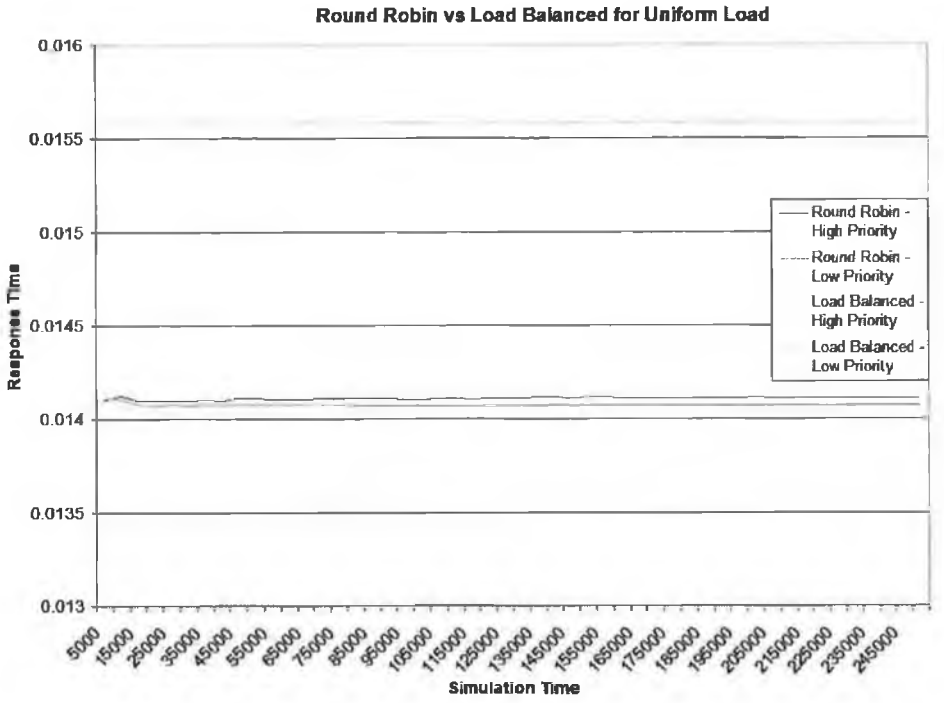

Figure 7.18: Round Robin vs. Load Balanced Average Response Time, Uniform Load

a value of 5 and the standard deviation value varies between 1 and 20 (using values 1, 2, 5, 10 and 20)

Load dispersion influence over the distributed algorithms' performance

Server	RR Resp. Time [TU]	LB Resp. Time [TU]
Server 1 High Priority	0.0077	0.0203
Server 1 Low Priority	0.0075	0.0199
Server 2	0.0057	0.0138
Server 3	0.0102	0.0101
Server 4	0.0065	0.0071
Server 5	0.0049	0.0103

Table 7.42: Server Average Response Time, Lognormal(05,01)

It can be seen (from Graph 7.19) that for small load dispersion the round robin performs much better than load balanced algorithm, due to it's lower overhead. For a standard deviation of 1, for high priority

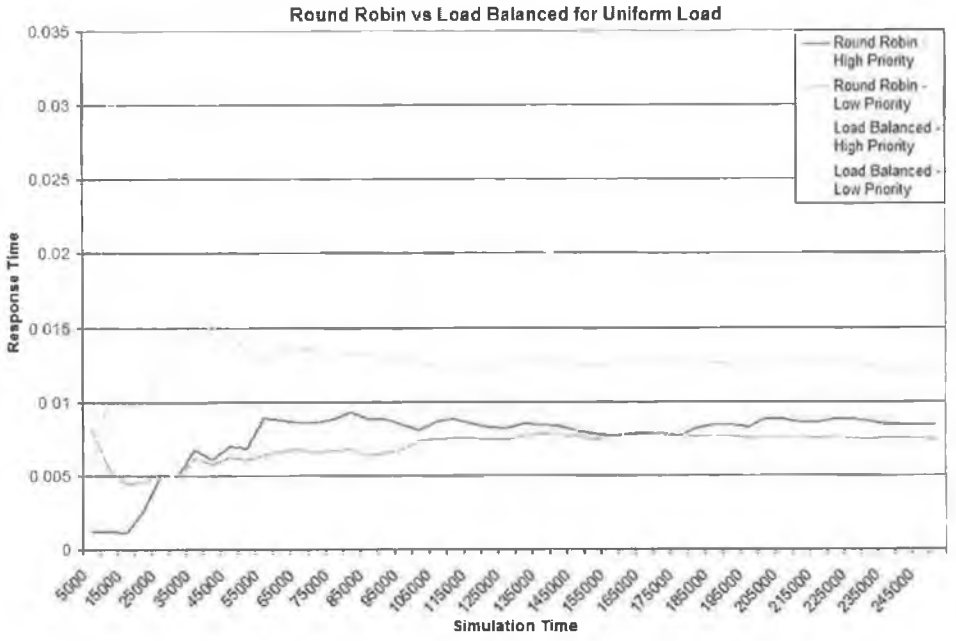

Figure 7.19: Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,1)

transactions the round robin average response time represents 37.93% of the average response time for the load balanced case (see Table 7.43). For the low priority transactions it is 56.86% of the average response time for the load balanced case.

Server	RR Utilization	LB Utilization
Server 1	0.092	0.204
Server 2	0.040	0.084
Server 3	0.070	0.072
Server 4	0.052	0.071
Server 5	0.045	0.077

Table 7.43: Server Average Utilization, Lognormal(05,01)

While the utilization for the load balanced case is substantially higher than for the round robin case (more than 2 times higher for the first server - see Table 7.43), the number of transactions being serviced by the servers is equally distributed in both cases (see Table 7.44) and from monitoring the simulation we can conclude that the load balanced case has the same

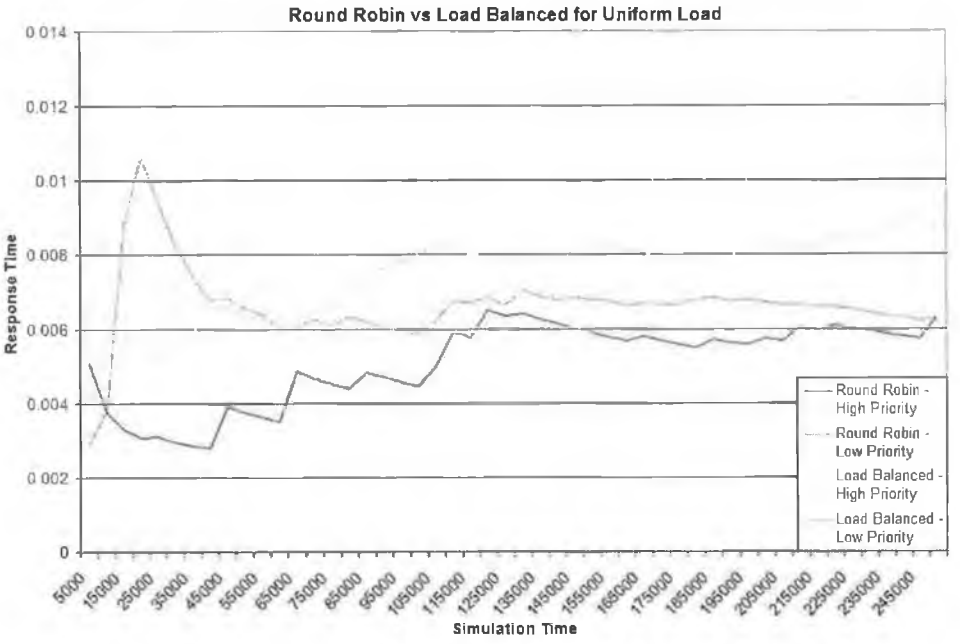

Figure 7.20: Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,2)

functionality as the round robin algorithm.

Server	RR Transactions	LB Transactions
Server 1 High Priority	1000033	1000507
Server 1 Low Priority	1799068	1799886
Server 2	1799068	1799886
Server 4	1799068	1799885
Server 5	1799067	1799885
Total	9995372	9999935

Table 7.44: Number of Transactions Processed, Lognormal(05,01)

The same observations can be made for the tests using a standard deviation of 2, but the differences tend to be smaller. For high priority transactions the round robin average response time represents 52.13% of the average response time for the load balanced case (see Table 7.46), while for the low priority transactions it is 88.98% of the average response time for the load balanced case.

The utilization for the load balanced case is higher than for the round robin case (approximately 2 times higher for the first server - see Table 7.46

Server	RR Resp. Time [TU]	LB Resp. Time [TU]
Server 1 High Priority	0.0049	0.0094
Server 1 Low Priority	0.0051	0.0089
Server 2	0.0067	0.0072
Server 3	0.0084	0.0076
Server 4	0.0073	0.0064
Server 5	0.0056	0.0071

Table 7.45: Server Average Response Time, Lognormal(05,02)

Server	RR Utilization	LB Utilization
Server 1	0.069	0.130
Server 2	0.047	0.064
Server 3	0.039	0.072
Server 4	0.054	0.084
Server 5	0.042	0.051

Table 7.46: Server Average Utilization, Lognormal(05,02)

but lower differences for the other servers) and the number of transactions being serviced by the servers is equally distributed between the five servers in both cases (see Table 7.47)

Server	RR Transactions	LB Transactions
Server 1 High Priority	1002499	1000193
Server 1 Low Priority	1800130	1799976
Server 2	1800130	1799975
Server 3	1800129	1799975
Server 4	1800129	1799975
Server 5	1800129	1799975
Total	10003146	10000069

Table 7.47: Number of Transactions Processed, Lognormal(05,02)

For the tests using a standard deviation of 5, similar results are obtained (see Graph 7.21). In this set of tests, the exact same results are obtained for the low and high priority transactions, as expected using a fixed weight component of 5 (see Section 6.5.2), excepting the slight change for server 5. For the low priority transactions, the round robin average response time represents 84.44% of the average response time for

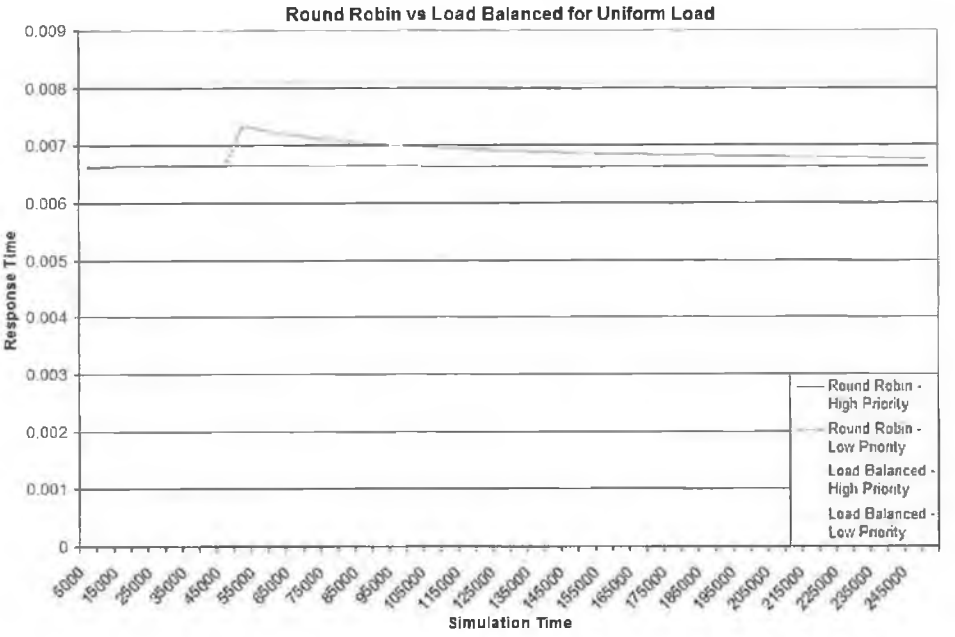

Figure 7.21: Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,5)

Server	RR Resp. Time [TU]	LB Resp. Time [TU]
Server 1 High Priority	0.0066	0.0081
Server 1 Low Priority	0.0066	0.0081
Server 2	0.0066	0.0081
Server 3	0.0066	0.0081
Server 4	0.0066	0.0081
Server 5	0.0078	0.0081

Table 7.48: Server Average Response Time, Lognormal(05,05)

the load balanced case (see Table 7.48). For the high priority transactions a slightly higher difference is noticed, the round robin average response time representing 81.48% of the load balanced average response time.

The utilization differences between the two cases reduces with the increase in the standard deviation, thus in this case for the round robin case it represents 83.9% of the load balanced case (see Table 7.49. The number of transactions being serviced by the servers is equally distributed between the five servers in both cases (see Table 7.50)

All information for the test using the Lognormal distribution with a

Server	RR Utilization	LB Utilization
Server 1	0.074	0.091
Server 2	0.048	0.058
Server 3	0.048	0.058
Server 4	0.048	0.058
Server 5	0.053	0.058

Table 7.49: Server Average Utilization, Lognormal(05,05)

Server	RR Transactions	LB Transactions
Server 1 High Priority	1000135	1000284
Server 1 Low Priority	1799993	1799963
Server 2	1799993	1799963
Server 3	1799992	1799963
Server 4	1799992	1799963
Server 5	1799992	1799962
Total	10000097	10000098

Table 7.50: Number of Transactions Processed, Lognormal(05,05)

mean value of 5 and a standard deviation of 10 are presented and discussed in section 7.5.6.

Using a standard deviation of 20, the obtained results support the observation that the increasing the standard deviation reduces the performance for the round robin algorithm while improving the performance of the load balanced algorithm (see Graph 7.23). For the low priority transactions, the round robin average response time represents 98.91% of the average response time for the load balanced case (see Table 7.51). For the high priority transactions a slightly higher difference is noticed, the round robin average response time representing 94.69% of the load balanced average response time, thus the behavior of the two algorithms becomes fairly similar.

The utilization differences between the two algorithms is small in this case, the round robin algorithm representing 94.66% of the load balanced algorithm (see Table 7.52). At the same time, the number of transactions being serviced by the servers follows the already observed behavior, being

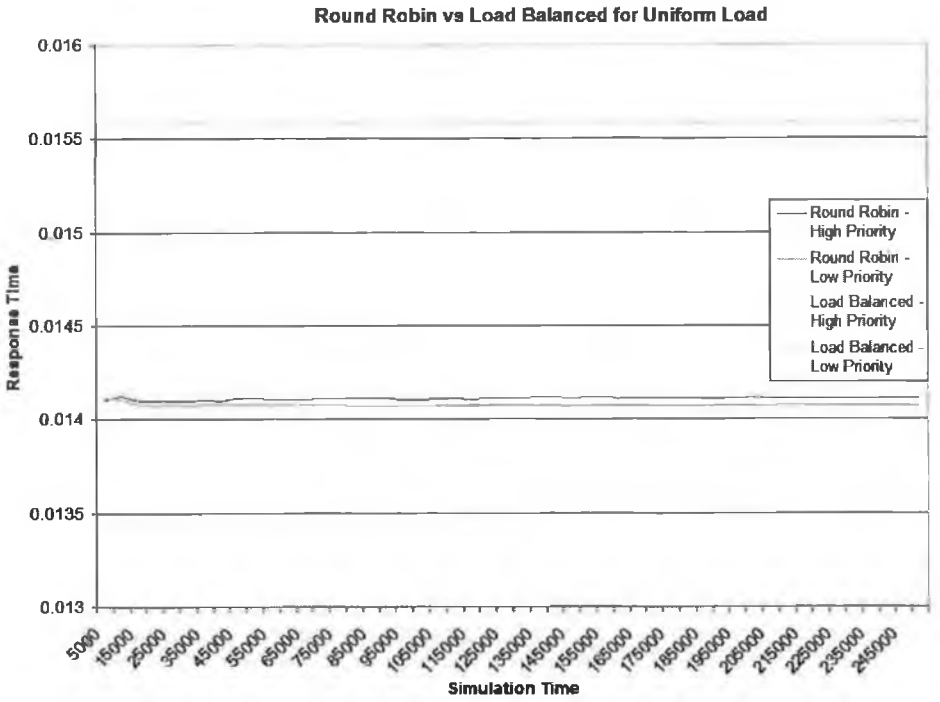

Figure 7.22: Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,10)

equally distributed between the five servers for both algorithms (see Table 7.53)

7.5.8 Conclusions

Evaluating the simulation results presented in Graphs 7.19, 7.20, 7.21, 7.22 and 7.23, it can be concluded that the load dispersion highly influences the performance of the algorithm and thus it supports the assumption

Server	RR Resp. Time [TU]	LB Resp. Time [TU]
Server 1 High Priority	0.0303	0.0320
Server 1 Low Priority	0.0298	0.0315
Server 2	0.0296	0.0312
Server 3	0.0296	0.0312
Server 4	0.0296	0.0312
Server 5	0.0296	0.0312

Table 7.51: Server Average Response Time, Lognormal(05,20)

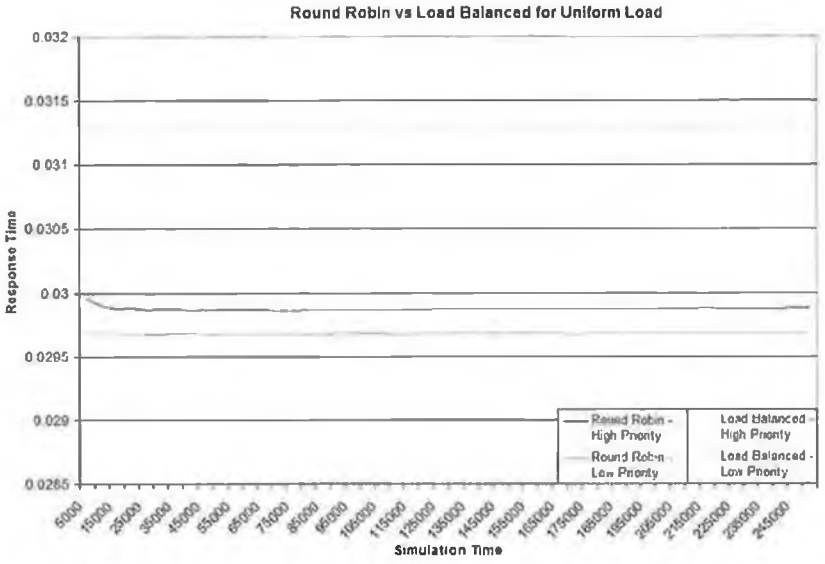

Figure 7.23: Round Robin vs. Load Balanced Avg. Response Time, Lognormal(5,20)

Server	RR Utilization	LB Utilization
Server 1	0.336	0.355
Server 2	0.213	0.225
Server 3	0.213	0.225
Server 4	0.213	0.225
Server 5	0.213	0.225

Table 7.52: Server Average Utilization, Lognormal(05,20)

that different algorithms perform better under different workload and the proposed framework (see section 5.6). At the same time, these simulations prove that the load dispersion should be one of the important factors monitored by the load management system for optimal load distribution algorithm selection.

Server	RR Transactions	LB Transactions
Server 1 High Priority	1000478	999375
Server 1 Low Priority	1799878	1799795
Server 2	1799878	1799795
Server 3	1799877	1799795
Server 4	1799877	1799794
Server 5	1799877	1799794
Total	9999865	9998348

Table 7.53: Number of Transactions Processed, Lognormal(05,20)

Chapter 8

Conclusions

The demand for improved throughput and greater dependability and scalability for distributed applications is increasing. Distributed systems throughput and scalability degrade significantly when the servers are overloaded by client requests. For alleviating such bottlenecks, load management services have to be used.

The problems detected in the existing load management services are presented and a new adaptive load management system is proposed. The most important load distribution requirements addressed are server transparency, decentralization, fault tolerance requirements and the need for activating/deactivating servers on demand, at runtime (for multiple distributed applications sharing a farm of servers).

The proposed framework has the advantage that it addresses scalability problems from small servers with no QoS-enabled services (figure 8.1) to the introduction of QoS (figure 8.2) and then to the large systems hosted on farms of servers (figure 8.3).

Another important feature of the proposed load management system is the addition of a end-to-end QoS service that is transparent to the application. The possibility of using diverse load monitoring granularity and of selecting/extending the load distribution algorithms at runtime is

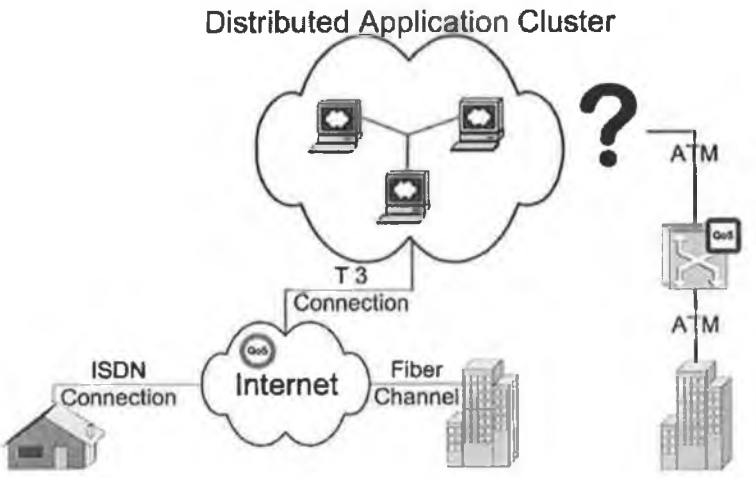

Figure 8.1: Initial System Configuration.

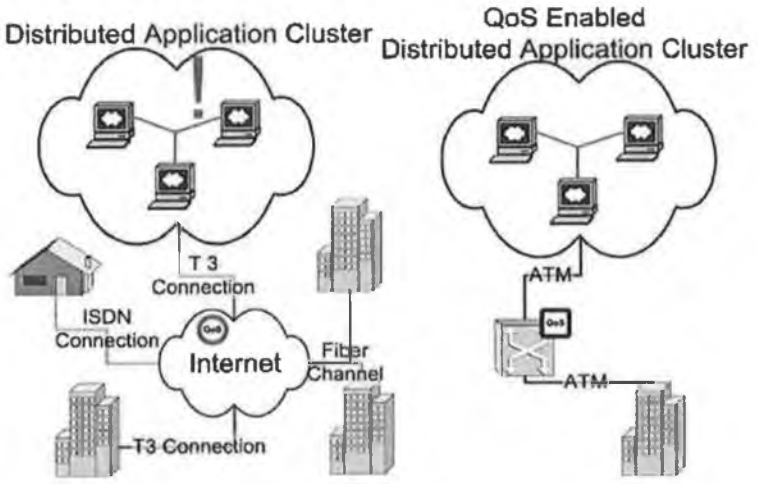

Figure 8.2: QoS-Enabled System Configuration.

guaranteed by the modularity of the framework.

A simulation model was created for testing available load distribution algorithms [52] [51] as presented in chapter 6.4.

The results confirmed the assumptions that the load aware algorithms can provide much better performance under most conditions and that sharing a server between different QoS service levels improves the response time for the lower priority levels. At the same time, the weight used for controlling the load on the shared server is very important, fine-tuning it guaranteeing that the response time remains between the specified limits

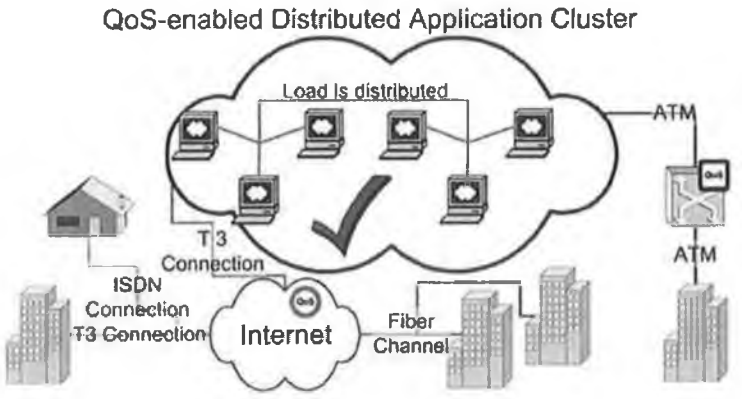

Figure 8.3: Large System Configuration.

while maximizing the number of lower-priority transactions serviced. It was considered that the server is not overloaded. If necessary for avoiding server overload, low priority transactions can be dropped.

An important factor for determining the number of servers required for processing the high-priority transactions is the percentage of high-priority transactions. By monitoring/predicting it the number of servers assigned for processing the high-priority transactions can be dynamically adjusted.

Through the simulations presented in the previous chapter the possibility of minimizing the influence of low-priority transactions over the response time for the high-priority ones (for the shared server) was shown. Thus, the results support the inclusion of QoS in the load-management system and servers sharing between different QoS service levels as proposed in section 5.2.

The proposed two-layered algorithm, required for the distribution algorithm replacement mechanism (described in section 5.6) was validated using simulations (see section 7.5). It was shown that the performance of the proposed algorithm is similar to the performance of the classic load distribution algorithms and that for some types of workloads a simple algorithm performs much better than a complex one (by minimizing the introduced overhead). These two conclusions support the proposed

modularity of the framework, especially the separation of the load evaluator and load distribution modules, as well as the need for multiple load distribution algorithms.

As seen in sections 7.3, 7.4 and 6.5, the performance improvements obtained are substantial and support the proposed approach.

Two of the most important conclusions that can be made from the experimental results are:

- Server weight has an important component directly related to the percentage of high priority transactions percentage. The weight must take this percentage in account.
- If a server becomes overloaded the distribution algorithm can bring it back to normal working conditions only after long periods of time, regardless of the load control/update frequency, thus affecting the overall response times for unacceptable periods of time. Thus, the use of special algorithms for such situations is required.

An important characteristic of this research approach is that it is general, i.e. a general model for the problem is developed, using a general-purpose simulator. The other available options are a very detailed model based on a dedicated simulator or an implementation. The advantages of this approach are flexibility, technology independence, and the fact that a general model is easier to develop than a very detailed one. At the same time, using a more general model, the results obtained are more general as well and thus could be used in other similar areas.

By decomposing the problem into subproblems and addressing them separately a better understanding of the problem can be achieved. Unnecessary technical details and limitations are not included in the model. For example the model the lower levels communication protocols was not

included in the model, like the influence of RMI/IIOP, database functionality details, or the overall performance of the network.

8.1 Future Work

An interesting problem, in the context of our work, is deciding when to activate/deactivate servers. This however, is an issue for further research; the emphasis here is focused on the more complex problem of determining the best way for managing the existing (constrained) resources.

For the two-layered algorithm (see section 6.5), the frequency of weights updates is a very important factor and it is very important to detect the optimal value. This value differs from application to application and an automated mechanism for detecting the best tradeoff point between the accuracy and the overhead should be analyzed.

Possible future work can address the following research problems:

- Load distribution algorithms optimization
- Automatic load distribution algorithms parameter tuning
- Component instance migration applicability
- End-to-end QoS framework
- The possibility of using existing monitoring modules [4] for the load monitoring module can be evaluated.

The mentioned research problems are discussed below.

Load Distribution Optimization

An important task is finding an optimal way of describing the situations where a particular algorithm is most suitable. The description has to be generic, using a standard, platform independent description language.

The same language can be used for describing the load distribution policies, so that the algorithms can be platform independent and reusable. Load distribution algorithms for handling degenerated conditions, such as unstable server loads can also be evaluated using the existing simulation model.

An application server cluster could be created, using an open-source application server, for implementing the proposed algorithms, evaluate their performance and further validate the existing simulation model.

Load Evaluator Parameters Adjustment

The load distribution algorithms have a series of parameters that can be adjusted, some of them having significant influence on system performance. In this thesis, the assumption that a load distribution algorithm with different values for its parameters is considered as a distinct algorithm was made. This assumption could be eliminated since it is not close enough to reality and in some situations, parameter tuning could lead to optimal performance, thus changing the algorithm completely will not be required. The possibility of creating a mechanism for automatic load distribution algorithm parameter tuning can also be evaluated.

Migration

Besides the initial request distribution algorithms, migration is a common way of distributing the load in a cluster of servers. While it is very common in process level load distribution, for component-based distributed systems it might not be applicable in some situations.

Migration is applied based on the assumption that the execution time on the overloaded server is higher than the migration time plus the execution time on the server it was migrated. In component-based distributed systems, due to its specific fine granularity, the execution time for a com-

ponent instance call is usually very short and at the same time, migrating the state of a component instance can be a very complex issue due to the concurrency of the system.

We envisage that component instance replication is the solution for these systems. Instead of migrating a component instance, a new instance of the same component is created on another server and the state of the first instance is mirrored.

The simulation model could be extended in order to take into account component instance state and evaluate the applicability of component instance migration and replication.

QoS Related Issues for a Practical Implementation

The existing QoS approaches can be extended and used to create an end-to-end QoS environment. The most important contribution to this area should be the unified service that will group most existing QoS approaches, situated at different system levels and offer a unified configuration point. The specified QoS configuration will then be propagated at all system levels in the required manner so that the end-to-end QoS goal will be achieved. The simulation model can be extended so that QoS enabled connections can be simulated.

Publications

- Octavian Ciuhandu and John Murphy, *"A Reflective QoS-enabled Load Management Framework for Component-Based Middleware"*, Proc. of Benchmarking Workshop, in conjunction with 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), California, USA, October 2003.

- Octavian Ciuhandu and John Murphy, *"Modular Quality of Service-enabled Load Management Service for Component-based Distributed Systems"*, Poster at 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), California, USA, October 2003.

- Octavian Ciuhandu and John Murphy, *"A Modular QoS-enabled Load Management Framework for Component-Based Middleware"*, Proc. of Communications Abstractions for Distributed Systems workshop, in conjunction with The 17th European Conference on Object-Oriented Programming (ECOOP), Germany, July 2003.

- Octavian Ciuhandu and John Murphy, *"Transaction Distribution Algorithms with User Classes for Distributed Application Performance Optimisation"*, Proc. of 10th IEEE International Conference on Software, Telecommunications and Computer Networks, (SoftCOM), Italy, October 2002.

- Octavian Ciuhandu and John Murphy, *"Component distribution algorithms for distributed application performance optimization"*, Proc. of IT&T 2002: Information Technology and Telecommunications, pp. 27-35, Waterford, Ireland, October 2002.

References

[1] J. Murphy A. Mos. New methods for performance monitoring of j2ee application servers. In IEEE, editor, *IEEE 8th International Conference on Telecommunications (ICT)*, June 2002.

[2] J. Murphy A. Mos. Performance management in component-oriented systems using a model driven architecture approach. In *Proc. of 6th IEEE International Enterprise Distributed Object Computing (EDOC)*, Sep. 2002.

[3] J. Murphy A. Mos. Performance monitoring of java component-oriented distributed applications. In IEEE, editor, *IEEE 9th International Conference on Software, Telecommunications and Computer Networks (SoftCOM*, October 2002.

[4] J. Murphy A. Mos. Understanding Performance Issues in Component-Oriented Distributed Applications: The COMPAS Framework. In *Seventh International Workshop on Component-Oriented Programming (WCOOP)*, June 2002.

[5] Analysis and Design PTF. UML Profile, for Schedulability, Performance, and Time. Specification, Object Management Group, Sept. 2003.

[6] D. Clark B. Branden and S. Shenker. Integrated services in the internet architecture. RFC 1633, IETF, 1994.

[7] A. Adamson B. Riddle. A QoS API Proposal, May 1998.

[8] A. P. Mathur B. Sridharan, S. Mundkur. Non-intrusive testing, monitoring and control of distributed CORBA objects. In IEEE, editor, *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 195–208, June 2000.

[9] M. Klein R. Kazman L. Bass J. Carriere M. Barbacci and H. Lipson. Attribute-Based Architecture Styles. In *Software Architecture, Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 225–243. Kluwer Academic Publishers, 1999.

[10] Don Box. *Essential COM*. Addison-Wesley Pub Co, 1st edition edition, 1998.

[11] F. Kuhns C. D. Gill, D. L. Levine and D. C. Schmidt et al. Applying Adaptive Middleware to Manage End-to-End QoS for Next-generation Distributed Applications. In *Proceedings of the 1st International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems, IEEE*, Phoenix, Arizona, Nov. 1999.

[12] C. M. Woodside C. Hrischuk, J. Rolia. Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype. In *MASCOTS*, pages 399–409, 1995.

[13] D. Gruntz C. Szyperski and S. Murer. Component Software. Beyond Object-Oriented Programming, 2002.

[14] L. G. Williams C. U. Smith. Software Performance Engineering: A Case Study with Design Comparisons. In IEEE, editor, *IEEE Transactions on Software Engineering*, volume 19(7), pages 720–741, 1993.

[15] Douglas C. Schmidt Christopher D. Gill, David L. Levine. The design and performance of a real-time CORBA scheduling service. *Real-Time Systems*, 20(2):117–154, 2001.

[16] Microsoft Corp. Application Architecture for .NET: Designing Applications and Services. Technical report, 2002.

[17] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from uml diagrams. In ACM, editor, *Proceedings of the second international workshop on Software and performance*, pages 58–70, 2000.

[18] L.G. Williams C.U. Smith. Performance and Scalability of Distributed Software Architectures: An SPE Approach. *IParallel and Distributed Systems*, 13(2), 2002.

[19] D. L. Levine D. Schmidt and S. Mungee. The design and performance of real-time Object Request Brokers., 1998.

[20] D. Decasper, G. Parulkar, and B. Plattner. A scalable, high performance active network node, 1999.

[21] Dan Malks Deepak Alur, John Crupi. *Core J2EE Patterns - Best Practices and Design Strategies.* Prentice Hall PTR, 1st edition edition, June 2001.

[22] Empirix. Bean test.

[23] R. Branden et al. Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification. RFC 2205, IETF, Sept. 1997.

[24] T. Ewald. Use application center or com and mts for load balancing your component servers. http://www.microsoft.com/msj/0100/loadbal/loadbal.asp, 2000.

[25] R. Kroeger F. Lange and M. Gergeleit. JEWEL: Design and implementation of a distributed measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657–671, Nov. 1992.

[26] David Gabel. Performance Management - High performance: Be proactive, not reactive. Technical report, TechReport, Oct. 2001.

[27] ISSLL Working Group. A framework for integrated services operation over diffserv networks. RFC 2998, IETF, 2001.

[28] Object Management Group. CORBAServices: Common Object Service Specification. Technical report, Dec. 1998.

[29] Object Management Group. The Common Object Request broker: Architecture and specification 2.4 ed. Technical report, Oct. 2000.

[30] Object Management Group. CORBA Component Model Specification, V.3.0. Technical report, June 2002.

[31] C.-C. Hui and S. T. Chanson. Improved strategies for dynamic load balancing. *IEEE Concurrency*, 7(3):58–67, July-September 1999.

[32] Hyperformix. SES/Workbench modeling and simulation tool. http://www.hyperformix.com.

[33] IETF. Integrated services (intserv). http://www.ietf.org/html.charters/intserv-charter.html. Technical report, 2000.

[34] IETF. Differentiated services (diffserv). http://www.ietf.org/html.charters/diffserv-charter.html. Technical report, 2003.

[35] SUN Microsystems Inc. Java Remote Method Invocation Specification (RMI). Technical report, Oct. 1998.

[36] Richard E. Schantz John A. Zinky, David E. Bakken. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

[37] E. Gamma R. Helm R. Johnson and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, Oct. 1994.

[38] V. Kachroo, Y. Krishnamurthy, F. Kuhns, R. Akers, P. Avasthi, S. Kumar, and V. Narayanan. Design and Implementation of QoS enabled OO Middleware, 2000.

[39] P. Khkipuro. Uml based performance modelling framework for object-oriented distributed systems. In Springer-Verlag, editor, *Lecture Notes in Computer Science No. 1723*, volume UML 99 – The Unified modelling Language, Beyond the Standard, pages 356–371, 1999.

[40] K. Lakshman, R. Yavaykar, and R. Finkel. Integrated CPU and Network I/O QoS Management in an Endsystem.

[41] Markus Lindermeier. Load Management for Distributed Object-Oriented Environments. In *International Symposium on Distributed Objects and Applications*, pages 59–, 2000.

[42] Tim Lindholm and Frank Yellin. *The JavaTM Virtual Machine Specification.* Second Edition edition, 1999.

[43] R. Kazman M. Klein M. Barbacci T. Longstaff H. Lipson and J. Carriere. The Architecture Tradeoff Analysis Method. In IEEE, editor, *Fourth International Conference on Engineering Complex Computer Systems*, volume Proceedings of ICECCS'98, pages 68–79, August 1998.

[44] Connie U. Smith Lloyd G. Williams. Performance Evaluation of Software Architectures. In ACM, editor, *Proceedings of the first international workshop on Software and Performance*, volume Workshop on Software and Performance, pages 164–177, 1998.

[45] J. Murphy M. Trofin. A self-optimizing container design for enterprise java beans applications. In *International Workshop on Component-Oriented Programming, part of ECOOP 2003*, 2003.

[46] Andrew S. Tanenbaum Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 1st edition edition, Jan. 2002.

[47] Carnegie Mellon. Software Engineering Institute.

[48] Sun Microsystems. Java 2 Platform Enterprise Edition.

[49] Sun Microsystems. Enterprise Java Beans Specification, 2.0 - Final Release. Technical report, Sept. 2001.

[50] A. I. Verkamo J. Gustafsson L. Nenonen and J. Paakki. Design Patterns in Performance Prediction. In *Workshop on Software and Performance*, pages 143–144, 2000.

[51] J. Murphy O. Ciuhandu. Component distribution algorithms for distributed application performance optimization. In *Proc. of ITT Annual Conference*, 2002.

[52] J. Murphy O. Ciuhandu. Transaction distribution algorithms with user classes for distributed application performance optimization. In IEEE, editor, *Proc. of 10th IEEE International Conference on Software, Telecommunications and Computer Networks, (SoftCOM)*, Oct. 2002.

[53] D. Schmidt O. Othman. Optimizing Distributed System Performance via Adaptive Middleware Load Balancing. In *ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*, Snowbird, Utah, June 2001.

[54] D. Schmidt O. Othman, C. O'Ryan. An Efficient Adaptive Load Balancing Service for CORBA, 2001.

[55] Andrew Oram and Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, Inc., 2001.

[56] Jaiganesh Balasubramanian Ossama Othman and Douglas C. Schmidt. The design of an adaptive middleware load balancing and monitoring service. In *Third International Workshop on Self-Adaptive Software, Arlington, VA, USA, June 9-11, 2003.*

[57] O. Othman, C. O'Ryan, and D. Schmidt. The Design of an Adaptive CORBA Load Balancing Service, 2001.

[58] Precise. Precise/Indepth for the J2EE platform.

[59] R. Schantz and D. Schmidt. Research advances in middleware for distributed systems. In *World Computer Congress, August 2002.*

[60] R. Schantz and D. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications, 2001.

[61] Douglas C. Schmidt and Stephen D. Huston. C++ Network Programming: Mastering Complexity Using ACE and Patterns, 2002.

[62] Quest Software. Performasure.

[63] D. Stewart and P. Khosla. Real-time scheduling of sensorbased control systems, 1991.

[64] Clemens Szyperski. Workshop on component-oriented programming, 1996.

[65] Z. Luo T. K. Liu, S. Kumaran. Layered Queuing Models for Enterprise JavaBean Applications. Technical report, IBM Research, 2001.

[66] Q. Lam T. Thai. *.Net Framework Essentials.* 2nd edition edition, Feb. 2002.

[67] Kenneth J. Turner. An architectural description of intelligent network features and their interactions. *Computer Networks and ISDN Systems*, 30(15):1389–1419, 1998.

[68] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[69] S. Blake D. Black M. Carlson E. Davies Z. Wang and W Weiss. An architecture for differentiated services. RFC 2745, IETF Diffserv Working Group, Aug. 1998.

[70] R. Weinreich and W. Kurschl. Dynamic analysis of distributed object-oriented applications. In IEEE Computer Society Press, editor, *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31),Software Technology*, volume 7, pages 386–395, Jan. 1998.

[71] World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. Technical report, May 2000.

[72] J. Huang R. Jha W. Heimerdinger M. Muhammad S. Lauzac B. Kannikeswaran W. Zhao and R. Bettati. A real-time adaptive resource management system for distributed mission-critical applications, 1997.