

Fine-Grained Object Based Load Distribution

An Experiment with Load Distribution in Guide-2

Master Thesis

Christian Damsgaard Jensen
DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 København Ø
Denmark
email: sigfried@diku.dk

October 1995

Abstract

Load distribution is commonly recognized as a useful technique to improve the performance of a distributed system. Much of the previous work on load distribution is performed in the context of process based operating systems executing on autonomous workstations.

In this thesis we examine load distribution in the context of fine-grained object based operating systems and describe the design and implementation of a load distribution facility in one such system. Our load distribution facility distributes groups of objects to avoid that some nodes in the system are idle while others are overloaded.

We use this load distribution facility to experiment with fine-grained load distribution based on objects. These experiments show that performance may be harmed by distribution of small groups of objects, but that distribution of larger groups of objects may significantly improve the performance of the system.

Resumé

Belastningsudjævning er en anerkendt metode til at forbedre ydeevnen i distribuerede systemer. Hovedparten af den hidtidige forskning indenfor dette område har fokuseret på traditionelle procesbaserede operativsystemer, hvor hele processer flyttes for at udjævne belastningen i et netværk af arbejdsstationer.

Dette speciale behandler belastningsudjævning foretaget i distribuerede objektbaserede systemer og beskriver design og implementation af en belastningsudjævner i sådan et system. Denne belastningsudjævner benytter grupper af objekter til at fordele arbejdet i systemet, således at ingen maskiner er ledige samtidigt med at andre maskiner er overbelastede.

Belastningsudjævneren benyttes til at eksperimentere med finkornet belastningsudjævning baseret på objekter. Disse eksperimenter viser at systemets ydeevne kan forringes når små grupper af objekter benyttes til belastningsudjævning, mens større grupper af objekter kan medføre betragtelige forbedringer af ydeevnen.

Preface

This thesis is in partial fulfillment of the Cand. Scient. degree at the University of Copenhagen. The thesis has been supervised by Eric Jul.

The work presented in this thesis has been done in two phases. The first phase includes the design and implementation of a load distribution facility in Guide-2, and was completed during a one year visit to the Bull-IMAG research laboratory in Grenoble, France. Unfortunately problems with the load distribution facility and with the Guide-2 system in particular prevented us from doing the evaluation of the load distribution during this visit. The evaluation that constitutes the second phase of this project, has therefore been done from Denmark, through the Internet, on machines located in France.

We expect the audience for this thesis to be familiar with basic operating system concepts, and have some fundamental knowledge about object-oriented programming and object based systems¹.

Acknowledgements

I would like to thank my supervisor Eric Jul for his help and support during my stay in Grenoble, and his guidance during the writing of this thesis.

I would also like to thank professor Sacha Krakowiak for inviting me to work in his laboratory, and for his advice and helpful comments on my work during the stay in Grenoble.

I am grateful to the entire team at Bull-IMAG for their assistance when I experienced problems with the system, their general interest in my work, and their great patience with my slow progress in learning French.

Thanks also to my colleagues in the System Support Group (EDB-afdelingen) at the Department of Computer Science at the University of Copenhagen for their support and their help in the final phases of this project.

I thank my fiancée, Lena, for her patience during my stay in Grenoble and for her help and support after my return to Denmark.

Finally, I am grateful to my parents for their financial support during my stay in Grenoble.

¹The terms object-oriented and object based will be used interchangeably throughout this thesis

Contents

1	Introduction	15
1.1	Motivation	15
1.2	Context of this Thesis	16
1.3	The Goals of this Thesis	16
1.4	Contributions	17
1.5	Overview of the Thesis	17
2	Load Distribution	19
2.1	General Concepts	19
2.1.1	Load Sharing or Load Balancing	20
2.1.2	Static or Dynamic Load Distribution	21
2.1.3	Initial Placement or Process Migration	21
2.1.4	Source Initiative or Server Initiative	22
2.2	The Load Distribution Facility	22
2.2.1	Performance Metrics	22
2.2.2	Components of a Load Distribution Facility	23
2.2.3	Policies and Mechanisms of Load Distribution	24
2.3	Information Gathering Policy	25
2.3.1	Hardware Load Indicators	25
2.3.2	Software Load Indicators	26
2.3.3	Choice of Load Indicators	27
2.4	Information Exchange Policy	27
2.5	Initiation Policy	28
2.6	Selection Policy	29
2.6.1	Resource Estimation	29
2.6.2	Process Discrimination	29
2.6.3	Random Selection	30
2.7	Location Policy	30
2.7.1	Defining Load Thresholds	30
2.7.2	Static and Dynamic Thresholds	31
2.7.3	Location Using Thresholds	31
2.8	Transfer Policy	32
2.8.1	Initial Placement	32
2.8.2	Process Migration	33
2.9	Summary	34

3	Objects and Load Distribution	35
3.1	Object Terminology	35
3.2	Objects and Distribution	36
3.3	Granularity of Objects	37
3.4	Encapsulation of Data in Objects	37
3.5	Interaction among Objects	38
3.6	Shared Objects	38
3.7	Duration of Method Invocations	38
3.8	Summary	38
4	Previous Work on Object-Oriented Load Distribution	41
4.1	Emerald	41
4.1.1	The Emerald System	41
4.1.2	Explicit Load Distribution in Emerald	41
4.1.3	Automatic Load Distribution in Emerald	42
4.1.4	Conclusions for Emerald	43
4.2	Bellerophon	43
4.2.1	System Overview	44
4.2.2	Load Distribution Overview	44
4.2.3	Information Component	44
4.2.4	Control Component	44
4.2.5	Conclusions for Bellerophon	45
4.3	Amadeus	45
4.3.1	System Overview	45
4.3.2	Load Distribution Overview	46
4.3.3	Experiments with Load Distribution in Amadeus	46
4.3.4	Conclusions for Amadeus	47
4.4	Ellie	47
4.4.1	System Overview	47
4.4.2	Load Distribution in Ellie	47
4.4.3	Conclusions for Ellie	49
4.5	Summary of Conclusions	49
5	Granularity of Object-Oriented Load Distribution	51
5.1	The Scale of Object-Oriented Load Distribution	51
5.1.1	Light-Weight Load Distribution Mechanisms	51
5.1.2	Integration with Other System Components	52
5.1.3	Increasing the Granularity of Load Distribution	52
5.2	Grain-Size of Load Distribution	52
5.2.1	Fine-Grained Load Distribution	53
5.2.2	Performance Benefits	53
5.2.3	Overhead of Load Distribution	54
5.3	Clusters of Objects	54
5.3.1	Grouping Objects into Clusters	55
5.3.2	Explicit Clustering	56
5.3.3	Automatic Clustering	56
5.4	Summary of Object-Oriented Load Distribution	57

6	The Guide-2 System	59
6.1	Introduction to the Guide Project	59
6.2	Overview of Guide-2	60
6.2.1	Generic Object Model	61
6.2.2	Generic Execution Model	62
6.2.3	Object Management	64
6.3	The Guide-2 Virtual Machine	64
6.3.1	The Object Machine	64
6.3.2	The Segment Machine	65
6.3.3	The Cluster Manager	65
6.3.4	The Execution Machine	65
6.3.5	Mach 3.0	66
6.4	Distribution in Guide-2	66
6.4.1	Jobs	66
6.4.2	Activities	67
6.4.3	Clusters	67
6.5	Summary	68
7	The Guide-2 Load Distribution Facility	69
7.1	Previous Work on Load Distribution in Guide	69
7.1.1	Simulations of Load Distribution in Guide-1	70
7.1.2	Explicit Load Distribution Facility in Guide-1	72
7.1.3	Lessons Learned	73
7.2	Design Goals	73
7.2.1	Performance Goals	74
7.2.2	Efficiency Goals	74
7.2.3	Implementation Goals	75
7.3	Granularity of Load Distribution	76
7.3.1	Distribution of Jobs	76
7.3.2	Distribution of Activities	77
7.3.3	Distribution of Clusters	77
7.3.4	Distribution of Objects	78
7.3.5	Granularity of Load Distribution in Guide-2	78
7.4	General Architecture	79
7.4.1	Load Balancing	79
7.4.2	Dynamic	79
7.4.3	Initial Placement	79
7.4.4	Source Initiative	79
7.4.5	Autonomy	79
7.4.6	System Integration	80
7.5	Information Component	80
7.5.1	Information Gathering Policy	80
7.5.2	Information Exchange Policy	81
7.6	Control Component	81
7.6.1	Initiation Policy	81
7.6.2	Selection Policy	81
7.6.3	Location Policy	81
7.6.4	Transfer Policy	82
7.7	Summary	82

8	Implementation	85
8.1	Elvis Architecture	85
8.2	Elvislib	86
8.2.1	Initialization Code	86
8.2.2	Invocation Code	87
8.2.3	Integration into the Cluster Manager	87
8.3	Elvis	87
8.3.1	Information Component	87
8.3.2	Control Component	89
8.4	Modifications to the Guide-2 System	91
8.5	Summary	91
9	Evaluation	93
9.1	Evaluation Goals	93
9.2	Evaluation Strategy	94
9.3	Evaluation Environment	96
9.3.1	Hardware Configuration	96
9.3.2	Running on Mach 3.0	96
9.3.3	Evaluation over the Internet	97
9.3.4	The Experimental Nature of Guide-2	97
9.4	Software Evaluation	97
9.4.1	System Overhead	98
9.4.2	Resource Consumption	99
9.4.3	Software Design Goals	101
9.4.4	Conclusions and Perspectives	102
9.5	Performance Evaluation	103
9.5.1	Average Workload Test	103
9.5.2	CPU-Intensive Workload Test	116
9.5.3	Conclusions and Perspectives	118
9.6	System Evaluation	118
9.6.1	Designing the Experiment	119
9.6.2	Performing the Experiment	119
9.6.3	Implications of the Experiment	123
9.6.4	Conclusions and Perspectives	124
9.7	Summary of Conclusions	124
10	Conclusions	127
10.1	Fine-Grained Load Distribution	127
10.2	Guide-2 Load Distribution	127
10.3	The Experiment	128
10.4	Future Work	129
10.4.1	Further Experiments	129
10.4.2	Load Indicators	129
10.4.3	Fine-Grained Load Distribution	129
10.4.4	Further Work on the Guide-2 Load Distribution Facility	130
10.5	Summary	130
A	Elvis Configuration Guide	131
A.1	Configuration Parameters	131
A.2	Sample Configuration File	132

List of Figures

2.1	Load Distribution with two thresholds.	31
3.1	Applications define subsets of a common distributed object space.	37
5.1	Fine-grained load distribution allows a evenly balanced load in the system.	52
5.2	Expected absolute speed-up by load distribution as a function of the object run-time.	53
5.3	The effect of the overhead of load distribution.	54
6.1	The Guide-2 system.	60
6.2	The generic object model.	61
6.3	The execution structures of Guide.	63
6.4	The structure of the Guide-2 virtual machine.	65
7.1	The host model used by Lunati.	70
8.1	Cluster mapping with load distribution.	86
9.1	Histograms of application run-times	106
9.2	Histogram showing the extreme nature of the measured run-times. The histogram shows all processes with run-times less than 500 ms. Ordinates have been cut off at 25000, so that more data points may be distinguished .	107
9.3	Logarithmic histogram of the measurements from figure 9.2	108
9.4	Model of execution for UNIX and Guide	110
9.5	Execution times of an UNIX application as a function of the number of iterations	110
9.6	Calculating starting cost of Guide application with linear regression	112
9.7	Execution times of a Guide application as a function of the number of clusters	112
9.8	Calculating the time needed to map a cluster with linear regression	113
9.9	Execution times of a Guide application with a fixed number of clusters as a function of the number of iterations	113
9.10	Structure of clusters in the test applications	114
9.11	Speed-ups achieved by load distribution	122
9.12	Run-times of applications with different cluster sizes	123

List of Tables

9.1	Execution times of an application with and without the overhead of the load distribution facility.	98
9.2	Calculation of the overhead introduced by the load distribution facility. . .	99
9.3	The measured CPU requirements of selected system services	100
9.4	Memory requirements of different Guide-2 services (with the name of the service in parenthesis)	100
9.5	Characteristics of the observed UNIX workload.	108
9.6	Characteristic properties of the three classes.	109
9.7	UNIX run-times of the application classes	109
9.8	CPU requirements of the application classes under UNIX	111
9.9	CPU requirements of the application classes in Guide-2	111
9.10	Main properties of synthetic applications	114
9.11	Run-times of application classes	115
9.12	Run-times of application classes	115
9.13	Distributed run-times and speed-ups achieved by load distribution on a typical Guide-2 workload	116
9.14	Distributed run-times and speed-ups achieved by load distribution on a light Guide-2 workload	116
9.15	Measured speed-ups for workloads generated by a varying number of applications.	117
9.16	Measured speed-ups for workloads generated by a varying arrival rate of applications.	117
9.17	Run-times and Speed-ups for three applications started with a one minute interval	120
9.18	Run-times and Speed-ups for six applications started with a one minute interval	121
9.19	Run-times and Speed-ups for nine applications started with a one minute interval	121
9.20	Run-times and Speed-ups for six applications started with a 30 seconds interval	121

Chapter 1

Introduction

This thesis describes the implementation of a load distribution facility in the Guide-2 Distributed Operating System. Guide (Grenoble Universities Integrated Distributed Environment) is a prototype implementation of the COMANDOS model of a distributed environment. COMANDOS (Construction and Management of Distributed Open Systems) is a cooperative project under the ESPRIT European programme aiming at providing an integrated environment for distributed applications [Cahill 93].

This chapter starts by giving a motivation for working with load distribution. In 1.2 the framework for the load distribution facility is defined, followed in 1.3 by the goals of the facility. This is followed by a statement of contributions of the work in 1.4, and an overview of the thesis is presented in 1.5.

The reader of this thesis is assumed to be familiar with distributed computing and distributed operating systems as well as with object-oriented programming systems. We do not assume the reader to be familiar with the Guide-2 system, which we describe in chapter 6.

1.1 Motivation

During the past fifteen to twenty years many computer installations have changed from centralized main-frame systems to workstations connected through a fast local area network. This shift occurred when workstations became powerful enough to satisfy most people's computing requirements.

The change from centralized to distributed computing has several explanations, but the most important is probably the development of hardware prices. The prices of main-frame systems have remained fairly constant, while the prices of workstations have dropped by a magnitude. Together with this decrease in prices the performance of workstations have increased by another magnitude. Another important factor is the networking technology that has been able to keep up with, and is possibly even about to overtake, the increase in processing speed, which makes the simultaneous utilization of several workstations advantageous. All in all this has made the network of workstations a cheap source of immense computing power.

In order to facilitate sharing of resources on the network (e.g. files, computing power or printers), an integration of the workstations into a common networked system is needed. In such an integrated environment it is frequently observed that, even though the workstations are powerful enough for most jobs, it happens that some of the hosts become heavily loaded, while others remain idle or only lightly loaded. This suggests that performance gains can be achieved by offloading work from the heavily loaded hosts onto the

less loaded ones. This sharing of computing power is the main objective of any kind of load distribution facility.

1.2 Context of this Thesis

This thesis studies load distribution in a small to medium scale distributed operating system, based on workstations connected by a (relatively) high speed network.

A network of autonomous computers are said to constitute a distributed operating system when the following conditions are met:

1. Each computer in the network is running its own incarnation of the operating system.
2. All the incarnations cooperate to create an illusion of a single computer.
3. The failure of a single computer in the network, does not bring the entire system down.¹

The Guide-2 distributed operating system, as implemented by a kernel called Elliott that runs on top of the NORMA version of Mach 3.0 on a network of Intel-386 and Intel-486 based PCs connected by an Ethernet.

Guide-2 is an object-oriented distributed operating system developed specifically to explore the use of shared objects as a base for co-operating applications in a distributed system. It currently supports programming in the Guide language and C++. When an object is not available locally, remote invocations will take place to access the object. This distribution is transparent to the user, which means that Guide-2 meets the two first conditions above.

Objects however are currently stored on a single node, which means that the third condition can not be met. Work is under way to remedy this shortcoming, which is why Guide-2 will be referred to as a distributed operating system throughout this thesis.

1.3 The Goals of this Thesis

The main goal of this thesis is to investigate the impact of fine-grained distribution on the performance benefits of load distribution.

This goal has been divided into three sub-goals: an examination of fine-grained (e.g., object based) load distribution based on the differences between process and object based systems, an implementation of load distribution in Guide-2 that allow us to experiment with fine-grained load distribution, and an experiment with this load distribution facility that gives a first indication of the performance benefits of load distribution in fine-grained systems.

An Examination of Object-Oriented Load Distribution

By examining the literature and selected implementations of load distribution mechanisms, the implementation issues (policies and mechanisms) of load distribution will be stated. These issues will then be examined with regard to the current demands of object-oriented systems. This leads to the definition of the primary areas of interest when implementing a load distribution facility in an object-oriented environment.

¹Leslie Lamport is quoted for the following definition: "A distributed system is one that stops you from getting any work done when a machine you've never even heard of crashes."

An Implementation of Load Distribution in Guide-2

It is a specific goal of this project to develop a working load distribution facility for Guide-2, which allow us to examine the impact of fine-grained distribution (e.g., distribution of objects) on the performance benefits achieved by load distribution.

An Experiment with Fine-Grained Load Distribution

The implementation of load distribution in Guide-2 should be used to confirm or deny the following hypotheses:

1. Load distribution offers general performance benefits in a distributed system with a fine-grained unit of distribution, such as the Guide-2 system.
2. Load distribution may significantly improve the performance of some applications, when the load on the nodes in the system is unbalanced.
3. The granularity of distribution is important for the size of these performance benefits.

The confirmation or denial of these hypotheses is the major goal of the performance evaluation in chapter 9.

1.4 Contributions

This thesis has made the following contributions:

1. A general study of load distribution has been performed, which allow us to design and implement a load distribution facility in Guide-2.
2. Work has been done extending load distribution concepts from the traditional process-oriented operating systems to the emerging object-oriented distributed operating systems.
3. We have developed a load distribution facility that allows experiments with fine-grained load distribution in the context of the Guide-1 object based operating system. Experiments with this load distribution facility has showed that:
 - Load distribution may degrade performance of applications with short run-times when the overall system load is light to moderate.
 - Large compute intensive applications may experience substantial performance benefits from load distribution, when the general activity in the system is high.
 - The granularity of load distribution is an important performance parameter in the construction of a load distribution facility.

1.5 Overview of the Thesis

The thesis is divided into three major parts. The first part consisting of chapters 2–5, examines load distribution as it is defined in the context of process based systems, and identifies some of the differences between process and object based systems that has to be taken into account when an object-oriented load distribution facility is implemented. An examination of process based load distribution is presented in chapter 2. Chapter 3 defines

our terminology in object based systems and identifies some important differences between process and object based systems with respect to load distribution. Previous work on load distribution in object based systems is examined in chapter 4. Different aspects of the finer granularity of distribution in most object based systems are examined in chapter 5.

The second part describes the implementation of a load distribution facility in the distributed object-oriented operating system Guide-2. This part consists of chapters 6–8. Chapter 6 describes the Guide system in general and the specific details needed to develop the load distribution mechanism. The chapters 7 and 8 describes respectively the design and implementation of the load distribution facility.

The third part consists of a single chapter: chapter 9. This chapter describes the evaluation of the load distribution facility and our experiments with fine-grained load distribution.

The conclusions of this thesis are found in 10.

Chapter 2

Load Distribution

Despite the large amount of literature on load distribution, a uniform theory has not emerged in the field. The designers of load distribution facilities are left with only a collection of seemingly unrelated schemes to choose from, often described in conflicting terms. This lack of uniformity in assumptions and idiosyncratic features of different schemes makes it unclear what performance benefits are potentially gained.

We do not attempt to provide a complete and coherent theory of load distribution in this chapter, in fact our main purpose is only to provide a framework that will allow us to discuss different aspects of load distribution. However, we do discuss some central issues in load distribution and present a view of load distribution based on the functions provided by a load distribution facility.

We start with a presentation of some general concepts in 2.1 that allow us to discuss different issues in load distribution. We then present a functional view of a load distribution facility in 2.2 that serves as a basis for the following presentation. The facility consists of two components: the information component and the control component that implement a number of different policies. These policies are described in 2.3–2.8.

The information gathering policy discussed in 2.3 defines what information is used to describe system load. This is followed by the information exchange policy in 2.4, which defines how this information is exchanged among the nodes in the system. The initiation policy that defines when load distribution should be considered is described in 2.5. The selection policy defines what work should be relocated to perform load distribution (e.g., which process to migrate); this policy is described in 2.6. The location policy defined in 2.7 determines which node should receive the relocated work, and the transfer policy that defines how work is relocated (e.g., are active processes migrated?) is discussed in 2.8. A summary of this chapter is given in 2.9.

2.1 General Concepts

Most of the work in load distribution to date has been done in the context of process-based operating systems, as opposed, for example, to object-oriented operating systems. This means that much of the terminology is defined in relation to processes. Rather than re-inventing terminology, this process-based terminology is used throughout this chapter. Some of this terminology is ambiguous and will be clarified in the following examination of general concepts in load distribution.

2.1.1 Load Sharing or Load Balancing

Although the terms load sharing and load balancing have been used interchangeably in the literature, they have also been used to distinguish between two important goals for load distribution. To avoid misconceptions by overloading existing terminology, we use the term load distribution throughout this thesis. We define load distribution as any scheme that redirects work in the system in order to improve performance.

The term load sharing is often used to describe load distribution schemes where work is directed to otherwise idle workstations, while load balancing often describes schemes that attempts to distribute the work evenly among the nodes in the system. We use these definitions throughout this thesis.

The success of a load sharing scheme depends on the number of idle workstations in the system. The assumption that idle nodes generally exist is supported by numerous observations reported in the literature [Theimer 85, Nichols 87, Douglass 91, Mutka 92] along with our own observations on the systems at the Department of Computer Science at the University of Copenhagen.

Load sharing has been successfully implemented in Sprite [Douglass 91], and Condor [Litzkow 87] using similar schemes. Local activity is monitored on all hosts, and those where no local activity has occurred for some time are regarded as idle and work may be offloaded to them. When users return to their workstation that may now be executing an alien process, this process is stopped and execution is resumed on another idle host. This transfer uses the process migration facility in Sprite, while Condor relies on a checkpointing scheme and a shadow process that may be resumed from the last checkpoint.

Load balancing is implemented in Utopia [Zhou 91] which is build to run on top of standard UNIX systems. It has also been the topic of numerous simulations [Eager 86, Zhou 88, Svensson 90]. In the following we outline the load distribution schemes described by Zhou [Zhou 88], to illustrate the diversity of load distribution schemes, but also because these schemes are frequently quoted in the literature and two of them (GLOBAL and CENTRAL) are used later in this thesis.

GLOBAL In GLOBAL one host, designated as the load information center (LIC), receives load information from all other hosts every p seconds, and assembles them into a load vector, which is broadcast to all other hosts. If the load on one host has remained the same, no information needs to be sent to the LIC.

The location policy of GLOBAL is as follows: when a job has been selected, the local copy of the load vector is searched for the host with the lowest load, if this load is lower than the current local load by some Δ^1 the job is sent to that host. If there are several hosts with the same lowest load, one of these hosts are chosen arbitrarily.

DISTED DISTED is similar to GLOBAL, but instead of reporting the load information to a central LIC, all hosts broadcast their own load periodically to all other hosts, so their local version of the load vector can be updated. The location policy of DISTED is identical to the location policy of GLOBAL.

CENTRAL In the previous two algorithms, location decisions were made independently on each host based on their local copy of the load vector. In CENTRAL these decisions are made at the LIC that still receives periodic load information updates from all hosts.

¹The optimal value for Δ depends on the expected workload.

When a host has selected a job for load balancing, it sends a request to the LIC along with the current value of its load. The LIC selects the host with the lowest load as target, and instructs the originating host to send the job there. The LIC then increments the load of the target host by 1 to reflect the additional load of the transferred job.

All the load balancing algorithms described above, assumes that global load information is available for the location policy, as opposed to the following algorithms that use no system state information.

RANDOM Only local load information is used in RANDOM. When the local load is high, a random node is selected to receive the job.

THRHLD When a job has been selected for load balancing, a number of randomly selected hosts, up to a probing limit L_p (probing limit), are polled and the first host with a load below a load threshold T_l is selected as target. If no host is found, the job is processed locally.

LOWEST LOWEST is similar to THRHLD except that a fixed number of hosts L_p are polled and the host with the lowest load is selected.

RESERVE This is a server-initiative algorithm based on job reservation. When a job terminates the local load is examined, and if it is below some threshold T_l , other hosts are probed to register R reservations at R hosts with load above T_l . The outstanding reservations are stored in a stack so that, when a job has been selected for transfer, the most recent reservation is used. If the load drops below T_l all reservations are cancelled.

As can be seen from the algorithms above a wide spectrum of information may be used, and many different location policies may be implemented based on this information.

2.1.2 Static or Dynamic Load Distribution

In static load distribution no information about the current state of the system is collected. Information about the different processes' demand for resources (i.e. CPU, memory, disk and possibly other external devices) are assumed to exist before the processes are executed or they are simply not available to the load distribution facility. This means that load distribution must be based on simple heuristics or instructions from the programmer, the latter case is also called explicit load distribution.

Dynamic load distribution assumes no *a priori* knowledge about the resource requirements of processes. This means that the load distribution facility must keep track of the system state, e.g. what hosts are up and what their loads are, and make the assignment decisions based on this information. Dynamic load distribution facilities therefore have the ability to react to imbalances in the system, by offloading work from overloaded hosts to those hosts that have less load.

2.1.3 Initial Placement or Process Migration

A load distribution facility that migrates running processes in order to offload work from overloaded hosts is called *pre-emptive*. This allows load distribution to improve on previously taken scheduling decisions by reassigning processes that do not perform well. There

is a price to be paid by process migration (e.g., the address space has to be transferred from one host to the other), so great care should be taken to weigh this price against the potential benefits.

Load distribution schemes that are not pre-emptive, make the assignment decisions when processes start up, they are therefore known as *initial placement* load distribution schemes. They do not pay the price of process migration, and in the case when programs reside on a file-server, almost no extra price is paid for remote execution (except for the overhead involved in the assignment). Initial placement schemes have been implemented in Utopia and are studied in the simulations of load balancing schemes listed above.

Simulations by Eager et al. [Eager 88] indicate that the performance benefits of migrating active processes may be limited compared to initial placement. Furthermore, the previously quoted work with load sharing in Sprite concludes that migration is unlikely to offer any improvement in a similar workstation environment, i.e., where idle workstations generally exist. We therefore believe that the performance benefits achieved by process migration are questionable, and that process migration should not be implemented or indeed used for the sole purpose of load distribution.

2.1.4 Source Initiative or Server Initiative

Another characteristic of a load distribution facility is who initiates load distribution. In *source initiative* algorithms the overloaded host decides to offload processes to another host, whereas algorithms where the underloaded hosts searches for more work are called *server initiative*.

In the terminology of above source initiative algorithms can be either load sharing or load balancing and may be used with both pre-emptive, and initial placement schemes. Server initiative algorithms are mostly load sharing (i.e. they only probe for work when they become underloaded).

The question of source initiated versus server initiated load distribution is examined by Wang and Morris [Wang 85], who evaluated ten different load distribution schemes with respect to source or server initiative. The evaluation was performed in part by mathematical techniques and in part by simulations. They conclude that server initiative algorithms have the potential to out perform source initiated algorithms, when the inter process communication (IPC) overhead is negligible.

The examination of load distribution concepts performed above was organized as four choices that has to be made when a load distribution facility is designed. These choices are: load sharing vs. load balancing, static vs. dynamic load distribution, initial placement vs. process migration, and source vs. server initiative in the load distribution facility.

2.2 The Load Distribution Facility

In the following we describe some of the issues that has to be addressed, when a load distribution facility is designed. We also present a functional view of a load distribution facility with a definition of the different policies involved in load distribution.

2.2.1 Performance Metrics

Load distribution may improve performance in a number of different ways, measured by the throughput, the system utilization, or the average run-times of applications.

System Throughput The system throughput is defined as the number of applications that can be processed by the system in a given period of time. Applications with a high demand for resources are often few, so throughput may often be optimized by allocating a few hosts for the most resource demanding application, and have the majority of the hosts executing the applications that require few resources. This way resources for the majority of applications will be plentiful and a large number of applications can be executed, while the few resource demanding applications will suffer from resource saturation on their few allotted hosts.

System Utilization The utilization of the system is measured by the fraction of hosts that are used for a given workload. Low system utilization may indicate the distribution of work among the hosts in the system is bad, or that resources are plentiful, i.e., the same amount of work could have been done in the same time with a smaller system.

Response Time The response time is measured by the average run-time of applications, i.e., the time it takes from applications are launched until they finish. Over the last decade applications have become increasingly interactive, so the duration of computational tasks (possibly from the same application) should be measured instead. However, this is difficult to measure without explicit support from the operating system kernel, so the average run-time is used to measure response time throughout this thesis.

Optimizing each of these performance metrics will favour different types of applications, although all applications may benefit. By optimizing throughput as described above, small jobs benefit at the expense of applications with long run-time. System utilization defines how well parallelism is exploited in the system. If the level of parallelism in the system is significantly higher than the number of machines, i.e., the number of runnable threads is much higher than the number of machines, optimizing utilization will also optimize system response time. However, if the level of parallelism is smaller than or comparable to the number of machines, there is no guarantee that optimizing utilization will improve the performance of applications, although multi threaded applications should benefit under these circumstances.

System throughput and system utilization both focus on the efficiency of the system, while the response time indicates how long users have to wait for their computations to complete. We therefore chose to focus on the response time as the performance metric in the rest of this thesis.

2.2.2 Components of a Load Distribution Facility

The load distribution facility may be divided into two major components, called the information component and the control component, based on what they manage.

This separation of information and control is inspired by E. G. Talbi [Talbi 95], because it provides a convenient way to describe a load distribution facility, but also because it allows a more complete classification of load distribution schemes than previous classifications (e.g., the taxonomy proposed by Casavant & Kuhl [Casavant 88]).

The responsibilities of these components are described in the following.

Information Component

The information component is responsible for collecting and distributing the information used in load distribution. This information generally describe the system state, but it

may also include information about applications, e.g., the expected resource requirements of the application. Information about application requirements generally requires a priori information about the application, which is rarely available before the input is given, so we only consider system state information in this chapter.

Control Component

The control component is responsible for the relocation of work in the system. This includes determination of when load distribution should be performed, what application should be selected for relocation, where the application should be transferred to, and how the transfer of control is done.

These components are described by the different policies that are used to implement the component. These policies are described in the following.

2.2.3 Policies and Mechanisms of Load Distribution

As with most previous work, our description of load distribution is a concoction of previously studied techniques and novel approaches. The policies presented here constitute a superset of what is normally found in the literature, so they form an adequate basis for describing existing load distribution schemes and general discussions of load distribution.

The following list constitutes our superset of the policies found in the literature. The actual division into separate policies may vary, but these different tasks are generally identified.

1. Information gathering policy (what is load.)
2. Information exchange policy (how to distribute the load information.)
3. Initiation Policy (when to perform load distribution.)
4. Selection policy (which processes to move.)
5. Location Policy (where to move processes.)
6. Transfer policy (how to move processes.)

These policies should preferably be implemented by separate mechanisms, so that policies can be changed by replacing the mechanism, if the precondition for the policy change. This means that the policies define the components, while the mechanisms simply implement the chosen policies. We therefore limit our scope to the description of different load distribution policies.

In reality, existing mechanisms will often play a part in shaping the policies, e.g., the information gathering policy will often be determined by what information is easily obtained. Furthermore, few of these policies may be chosen independently, because choices made for one policy may limit the possible choices of other policies, e.g., if random selection is chosen for the location policy, system state need not be maintained, which reduces the importance of the information component.

The first two policies define the information component, while the policies 3–6 define the control component. Each of these six policies are described in the following.

2.3 Information Gathering Policy

The information gathered by the load distribution facility are generally called load indicators although they may only give a very indirect indication of the load, e.g., knowing the rate of dropped or retransmitted network packets originated at the host indicates whether or not the communication capacity has been exhausted, but does not directly indicate the machines ability to handle more processes². Based on these load indicators, estimates can be made regarding the anticipated response time for executing processes on that host. These estimates are typically based on the assumption that the load will not change dramatically within a limited period of time. However, this is generally not true, as shown by Ferrari and Zhou [Ferrari 86], but these estimates need not be very precise as long as the different hosts can be compared and the host or set of hosts with the shortest execution time for a given process can be found.

The traditional concept of “load” (e.g., the UNIX load obtained through `uptime(1)`) describes the contention for the CPU, but this is only one of the sub-systems in the computer, and all the other sub-systems — e.g., disk, RAM, the system bus, the network and other input/output devices — may be equally important, because saturation may degrade performance. Apart from these very hardware oriented load indicators other indicators that describe the performance of the system software may be of interest. Such indicators are the number of context switches, the number of created processes, the ratio of virtual to physical memory, the number of messages send from or received by this host, and so on. These different possibilities will be the subject of further study in the following.

2.3.1 Hardware Load Indicators

The CPU has traditionally been regarded as the expensive unit in a computer system, so all the other sub-systems has been designed to keep the CPU working optimally (i.e. DMA, caches, intelligent I/O-controllers, etc.). It should then be safe to use CPU utilization indicators — such as the current/average length of run-queue or CPU idle time — as a universal load indicator. It is only necessary to consider a modern PC (486-66DX with an ISA-bus) to see that this is not always true. The speed of the CPU and the disk has overtaken the speed of the bus, which is artificially low for compatibility reasons, so the bus is in effect the bottleneck for disk intensive processes. The comparison of the relative speed-ups of CPUs and operating systems by John Ousterhout [Ousterhout 90], shows that the CPUs have gotten much faster than operating systems have, which suggests that importance of the CPU as the single contended resource may no longer be true. In fact, he concludes that memory bandwidth is not keeping up with CPU speed and suggests that this will possibly worsen in the future. It is therefore useful to consider the other possible hardware indicators from time to time to make sure that the balance between the different hardware sub-systems has not been upset.

We examine most of these alternative load indicators in the following, and outline their importance with respect to load distribution.

The hardware load indicators for the internal memory are the amount of available free memory and the page fault/swapping-rates. The amount of available memory indicates if more processes can be added to this machine without it has to start swapping. Swapping to disk is much slower and therefore generally unwanted.

²A process need for different resources like CPU, memory or disk remain constant regardless of distribution. The network has a special status, being the medium of distribution, but also because the need for this resource depends on the distribution of processes.

Most of the modern virtual memory systems over-commit physical memory (i.e. more virtual memory is in use than physically exists at the machine, physical memory is then backed by disk), so there may not be any free memory at all. Instead of using the available memory as a load indicator, other indicators like the page fault/swapping rate that tells us how much the physical memory is over committed has to be considered.

By avoiding swapping the CPU can allocate more of its time to actual computational work, so an indication of the chance for swapping to occur can be a useful parameter for the location policy.

As already mentioned the bus can be a serious bottleneck in a computer system. This is especially true in the world of “Open Systems” where hardware has to comply to standards that may be outdated. In most cases these standards will be updated before performance degradation is noticeable, as can also be seen with the EISA, MCA, VESA, and PCI bus technologies introduced on the PC market. The bus utilization should therefore rarely provide information beneficial to the load distributor.

Using the utilization of an I/O-device as a load indicator is only interesting when the device is replicated on all hosts, and all the replicas serves the needs of the process (e.g. temporary disk space). Otherwise the process will have to pay the cost of using the loaded device, wherever it is, by waiting for a remote agent located on the same machine as the desired device, instead of waiting for the device itself. With the above limitations and since many processes use several I/O-devices (e.g. disk and display) we do not believe that this type of load indicator is very useful for a load distribution mechanism.

The utilization of the network indicates whether communication is currently a good idea or whether processes should be kept locally or not (it is often faster to run processes on an overloaded node if it avoids using a congested network). A previous project investigating the communication time between nodes on a network [Jensen 93] showed that it can be very difficult to predict communication times, even though they are fairly stable, because of sudden spikes in the communication time. This suggests that a network utilization indicator is not generally useful, but may be useful when the network is overloaded.

2.3.2 Software Load Indicators

The hardware load indicators describe the resources currently available on a given machine, while the software load indicators describe the activity on that machine.

The system-activity can be measured by the number of context switches or the number of system calls. High rates of context switches or system calls generally means that many processes are active on the system. It also means that a lot of time is spent by the operating system administering these events — which again means less time for useful work. The system activity is the software correspondent to the CPU-utilization of the hardware.

A good measurement for the memory-utilization is the ratio between virtual memory in use and actual physical memory. This ratio expresses directly the over-commitment of physical memory.

Almost all activity in the system uses the system bus, but there is no single event that may be measured to quantify the saturation of the system bus.

There are no good software measurements for the network-utilization. The number of messages sent from or received at the machine, or the number of free message-buffers tells about the network interface, but not the state of the network itself. It is only when these resources become saturated that they have an impact on the system performance. The saturation of these resources means that the system cannot communicate with other machines in the distributed system and should rarely occur. The use of these statistics in

the load indicator is therefore questionable.

2.3.3 Choice of Load Indicators

The question is now, what load indicator or combination of load indicators to use. Two different approaches have been taken to answer this question. The result of these works will be summarized in the following.

Based on workload traces from a network of diskless Sun-3 workstations, simulations by Thomas Kunz [Kunz 91] shows that CPU related load indicators are best (i.e., length of run-queue and number of context switches). More surprisingly, he claims that using a combination of several load indicators does not improve the performance of the load distribution facility. This contradicts the common intuition that more detailed information makes better decisions. His claim is based on limited data, in fact he only combined two CPU related load indicators (run-queue length and system call rate), which does not enhance the knowledge about the system but only confirms it, therefore results comparable to the ones for the single load indicators would be expected, which is indeed the case. This leads to the following rule for construction of combined load indicators: “If the load indicators are seen as vectors that span a space of system information (e.g., the run-queue and the disk queue span the two dimensional space of CPU and disk demand) it is important that the chosen combination of load indicators are a minimal set, i.e., that no smaller set of load indicators will span the same space”. This definition is not very formal, but it gives a useful way of thinking about how combined load indicators should be constructed.

A queuing network model has been used to give an analytical answer to this question [Ferrari 86]. They also conclude that a single CPU related load indicator is best. The combination of the resource-queue lengths for the CPU and the disk was investigated, but they found that the added information about the disk-queue did not make any significant difference on the calculation of the response time.

Two independent works have reached the same conclusion that the CPU related load indicator is best. Kunz also confirms the common assumption that the CPU is still the contented unit, and he claims that the use of combined load indicators do not improve the result of the load distribution. However, this last result is based on very limited research that we believe has reached the wrong conclusions, because he combines two indicators for the CPU. Furthermore Ousterhout claims that memory bandwidth may become a bottleneck in the foreseeable future, which indicates to us that further research in this area is needed, and that combined load indicators should be reexamined each time the balance among system components change.

2.4 Information Exchange Policy

The goal of the information exchange policy is to make the gathered load information available to the load distribution facility where and when process assignments needs to be made.

The quality of load information depends on what load indicators were gathered (as discussed above) and how often they are updated. The update frequency is the source of two conflicting interests: A high frequency means that information is more up to date, but more system resources are consumed in gathering and distributing this information; resources which could otherwise have been used for computing.

Where and when load information is needed depends on the load distribution policy and how the systems state is maintained. It is possible not to maintain any system state; whenever load information is needed the other workstations are polled. Such schemes have the advantages of up to date information and high fault tolerance, but the price is more network communication, especially when the load is high — leading to possible congestion of both CPU and network. These schemes will not be discussed further, and the following assume the maintenance of some system state.

System state can be maintained on a scale from fully *centralized* where one host keeps all load information to fully *global* where all the hosts in the system keeps their own copy of the system state. These two extremes do not scale very well, so intermediate levels like hierarchical schemes or buddy systems are needed for large scale distributed systems. These systems are outlined in the following.

Hierarchical schemes may be defined in many ways, but they share some common features. The hosts are divided into a hierarchy, where each host generally has access to load information about other hosts at the same level of the hierarchy (either through a centralized or a global scheme), and the hosts that form the inner nodes of the tree also hold the information about the next level below themselves.

In buddy systems, each host has a set of hosts called buddies, where it can offload work and (possibly another set of buddies) from whom it may receive work. The host needs to know the load of the systems it can offload work to, and to report its own load to the systems it can receive work from.

Other schemes have been suggested, where the recipients of the load information are chosen at random, and statistical analysis is used to optimize the quality of information with the minimum of information exchange. These schemes will not be discussed further here.

In general, the choice of information exchange policy depends on the scale of the system and the interconnection of the network connecting the machines. The above mentioned strategies are a subset of the possible strategies, and appropriate policies must be chosen for each system.

2.5 Initiation Policy

The goal of the initiation policy is to avoid the overhead of executing the load distribution facility, when performance benefits are unlikely, i.e., when the local load is low.

The initiation policy is normally triggered either periodically or by some event in the system. Some randomness may be included in the length of the period (i.e., not all periods have the same length) to avoid phase locking between applications and the load distribution facility. Phase locking occurs when the system load changes periodically with the same period used by the load distribution facility. In this case a seemingly stable load may hide the fact that large variations in the system load exist. The events that trigger the initiation policy are either external (e.g., requests for work from underloaded nodes) or in some way supported by the kernel (e.g., the kernel monitors the local load and schedules the initiation policy when the load exceeds some threshold or simply the creation of a new process may trigger the initiation policy).

When the initiation policy is triggered, it will examine the local load to determine if performance benefits from load distribution are likely, and initiate the load distribution facility if the local load is high. If the initiation policy decides that such performance benefits are unlikely, the rest of the load distribution facility need not come into play.

2.6 Selection Policy

The goal of the selection policy is to select the process that, when executed remotely, will either benefit the most or be most advantageous to the system as a whole — depending on the overall goal of the load distribution facility.

In general, the resource demands of the processes are not known until after they are executed, so they will either have to be estimated in some way or other methods must be used to select the process that should execute remotely.

2.6.1 Resource Estimation

In pre-emptive load distribution, resource estimates have to be made for all running processes. The quality of the load distribution depends on the quality of these estimates.

A traditional way of estimating the remaining resource demands for a process is to rely on the fact that when all processes are examined, they will statistically be half way through execution. The remaining resource demands are then assumed to be equivalent to the resources consumed so far.

Another way is to ask the user to give some estimate of the different resource demands required by the process. This was done in older commercial batch systems like Sperry's OS-1100. The problem with this approach is that once users understand how the load distribution facility works, they are likely to give an estimate that will give their own processes the most advantageous conditions leading to inflated resource estimates.

A third way of estimating the resource demands is based on previous execution times for the same program. This approach is implemented in Sprite [Osser 92]. This work shows that monitoring the usage of CPU, number of system calls that must be resolved on the users "home computer" (e.g., system calls like `gethostname(2)` in UNIX) and calls accessing local devices, allows metrics to be constructed that help identify the processes that should always be executed locally.

In general, these estimates will not be very precise, so only significant differences should be taken into account.

The resource estimation in initial placement algorithms are somewhat simpler because it only has to consider whether or not the new process is likely to benefit from remote execution. This simpler version of resource estimation is called "the problem of process discrimination", i.e., which processes should be allowed to execute remotely. Process discrimination can also be used to limit the scope of the selection policy, thus making the resource estimation cheaper because fewer processes need to be examined.

2.6.2 Process Discrimination

Since not all processes will benefit from remote execution, those that do not have to be detected and should always remain local. Examples of such processes are processes with high demand for local resources (i.e., are I/O-bound on local hardware or are communicating a lot with local processes), and processes that only have a short demand for CPU-time.

Heavy use of local resources is hard to detect and the detection methods will depend on the system, so only the case of short-lived processes will be considered here.

User supplied information is a simple way of discriminating processes. Processes are typically divided into two categories, those that may benefit from remote execution and those that should always remain local. This method has been implemented by the Utopia system [Zhou 91] from the University of Toronto. Utopia uses an initial placement algorithm that performs process discrimination based on information from a per user con-

figuration file, or alternatively from system wide configuration files that tell whether the application is eligible for load distribution or should always be executed locally.

A method for automatic detection of short-lived processes is proposed by Anders Svensson [Svensson 90]. A filter called History is introduced into the selection policy to determine whether processes should be executed remotely or not. Based on the names of programs and collected statistics about their previous behavior, short-lived processes are detected and only longer running processes are considered for remote execution. The method has been evaluated by trace driven simulation on a network of 16 diskless Sun-3 workstations (the traces were collected on the 16 workstations at Lund University during a period of four months, and have then been simulated on a network queueing model of this environment).

This work shows that a simple filter can limit the number of processes considered by the load distribution policy to 5% without performance loss compared to passing all processes on to the load distribution policy. Performance can be improved by restricting the load distribution policy to the 20% longest running processes i.e. by using this filter to discriminate against 80% of the processes.

It also shows that the effect of the filter increases with the costs of transferring processes. This means that only longer running processes should be considered for remote execution, when transfer costs are high.

2.6.3 Random Selection

A last possibility is to select the process randomly. This scheme is simple and easy to implement and will generally perform the selection much faster than the schemes outlined above. The success of this scheme depends on the number of processes that will benefit from load distribution compared to the number of processes that will not. If this ratio is high enough, random selection could provide good results.

2.7 Location Policy

The goal of the location policy is to locate the remote host, where the selected process is offered the best conditions. This host will be called the destination or the target for load distribution.

When global system state is kept (load indicators are stored on all nodes), the target host is selected based on this information, otherwise a set of hosts are requested to submit load information according to the policies of the information component.

As previously mentioned more than one candidate for the destination is likely to exist, so the location policy has to select one of these hosts. If load information is stored locally, several decisions could be based on the same indicators, so some heuristic for updating these indicators or some randomness must be introduced in the location policy to avoid overloading a host by repeated selections.

Location of an appropriate host is normally based on one or more thresholds. These thresholds are either predefined (static) or defined in terms of the overall load of the system (dynamic).

2.7.1 Defining Load Thresholds

Determining a static threshold is not simple, because it should reflect the anticipated load of the system. If the threshold is too high, no load distribution will take place because all

hosts appear underloaded, and if the threshold is too low, all nodes may appear overloaded and too many resources are consumed by distribution of processes.

Hosts with a load near the predefined threshold constitutes another problem. Sometimes they will appear underloaded and may accept work, that will overload them, forcing them to offload work that will bring them back among the underloaded hosts. This phenomenon is generally known as thrashing.

Both problems are resolved by introducing a second threshold as illustrated in figure 2.1, this in effect creates a third category of normally loaded hosts between the underloaded and the overloaded hosts.

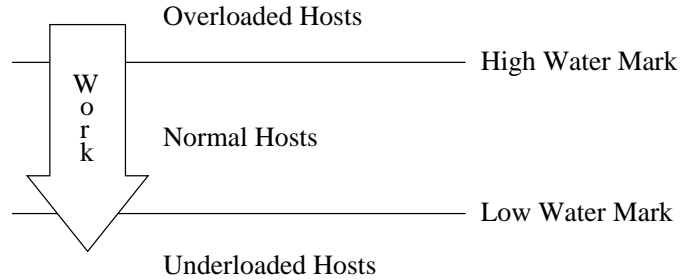


Figure 2.1: Load Distribution with two thresholds.

The load spectrum for the normal loaded hosts should be large enough to ensure that offloading work from an overloaded host will not directly make it underloaded. This use of thresholds implies that the high water mark is used by the initiation policy to determine whether load distribution should be considered and the low water mark should be used by the location policy to identify underloaded nodes.

2.7.2 Static and Dynamic Thresholds

Static thresholds have a problem when the entire system is overloaded, because no load distribution will be performed even though great imbalances can exist in the system. However, it is not clear that load distribution should be performed under these circumstances, because the resources consumed by the load distributor, may be better used to finish some of the processes.

If imbalances exist in the system, dynamic thresholds will define some of the hosts as underloaded, even though contention for system resources may exist on the target host, and therefore always allow load distribution to be performed. If no targets can be located, it is because no load distribution should be done, i.e. system resources are more contented on the other hosts.

2.7.3 Location Using Thresholds

Considering the difference between the load on a machine and the high water mark as a measure of available resources at that host, assignments can be made using the same algorithms used in memory management, i.e. “first fit”, “next fit”, “best fit”, “worst fit” or a buddy system (described in [Tanenbaum 92]). In the following these methods will be studied in the context of host location.

First fit is defined as the first host satisfying the resource needs of the process is selected. When the resource needs are not known, the first underloaded host is selected, which may

in turn overload the host.

Next fit is a variation of first fit, that keeps a list of currently underloaded hosts. Each time the location policy is invoked, it remembers which host was assigned last time and starts from here. This is one way of making sure that the same host will not be selected several times in a row.

Best fit selects the host where the available resources best matches the needs of the process. This method can not be used, when the resource needs of the process are unknown.

Worst fit selects the least loaded host, providing optimal conditions for the process and leaving the largest possible resource gap, that may then be used by another process. This is the worst approach in memory management, but commonly regarded as the best in load distribution.

Buddy systems as defined in memory management does not make sense in this context, but another definition of buddy system may replace it. The hosts are divided into small groups of buddies that resolve the assignments within themselves. This approach has been proposed by Peter Dickman [Dickman 92b] for Bellerophon and a variation of this approach is known as the Execution Territory [Raverdy 95]. If no underloaded hosts exist in the buddy group, this group is merged with another buddy group until an underloaded host is found or all hosts are in the same group. When several underloaded hosts exist in a buddy group it may be split in two simplifying the location policy, but also limiting the hosts that needs to be updated by the information policy.

In Conclusion: Using the algorithms developed for memory management gives a convenient way of describing location algorithms. However, the relative merits of these algorithms has yet to be determined.

2.8 Transfer Policy

The transfer policy deals with the transfer of control and data from the source-host to the destination-host. This transfer is quite different in initial placement schemes from the process migration of pre-emptive schemes. These different transfer policies will be studied separately below.

2.8.1 Initial Placement

This is the simplest case, where distribution is handled by the remote execution facility and the file sharing facility of the underlying distributed operating system.

The remote execution facility should enable the system to start processes remotely, that will run in the same context as if they had been started locally, i.e. terminal I/O should be connected to the display and keyboard of the source host, file access should be transparent, and possible user customizations of the run time environment should be available on the destination host.

If a global transparent file store is available, this facility can be used to transfer the process image to the destination host, otherwise the program will have to be transferred by the available Remote Procedure Call (RPC) mechanism.

2.8.2 Process Migration

Process migration is used by pre-emptive load distribution schemes to migrate active processes from nodes where they perform poorly to nodes where they may perform better.

In general the process migration mechanism must handle the transfer of process control, process image and external process references. This transfer is discussed in the following.

Process Control Transfer of control is done by stopping the process, transferring the process control structures (often called the process control block or PCB) and restarting the process on the new host. The process control block consists of all the information stored in the kernel about the process. This information typically contains the basic state of the process, identification, creator, priority, and information about the image of the process e.g. page table entries, and the stack with stack-pointer. Some or all of this information may have to be translated on the receiving host, e.g. it is unlikely that the process will be assigned the same set of physical memory pages, so the page table entries will have to be changed.

Process image The image of the process can be copied before, while or after the process is stopped. Stopping the process and transferring the image is the naïve approach most easily implemented. Unfortunately it means that the process is stopped for the duration of the copying, which for a 2 MB process on a 10 Mbit Ethernet may be around 6 seconds, where the process exists in a kind of Limbo not responding to external events like receiving messages, which in turn could make other processes time out believing that this process is dead. In this case the system will have to take care of these external events, making sure that messages will not time out and that they will become available on the target host.

Copying most of the image before the process is stopped, known as *pre-copying*, has been implemented in the V system. The idea is to limit the time the process is stopped by first copying all of the image to the target. When this copying has finished the local image is examined and the memory-pages that have been updated is again copied to the target. This process is continued until few pages have been updated or the number of copied pages remain constant, then the process is stopped and the control is transferred. Calculations have shown [Theimer 85] that pre-copying a 2 MB process image reduces the time a process needs to be stopped from 6 seconds to 0.03 seconds.

Another scheme that reduces the time a process is stopped, is to stop the process, transfer control and then rely on *demand paging* from the source host to transfer the image. This has an advantage over pre-copying with the respect to the amount of data transferred, because it only copies the part of the image that the process actually accesses and then only once. This scheme is implemented in the Accent system [Zayas 87] and measurements show that on average only 42% of the image needs to be transferred. The problem with demand paging from the source host is that the process still consumes resources on that host, and that it becomes vulnerable to crashes on the source host.

External References Other processes may hold references to the migrated process, so these references will have to be updated, or forwarding references will have to stay behind, so these processes will be able to communicate with the migrated process.

In general forwarding references are often unwanted because such residual dependencies generates load on the source host, and it makes the process vulnerable to crashes or reboots of that machine. On the other hand it is very difficult to identify all processes that may

hold references to the process, so some way of updating these references without necessarily involving the source host will have to be devised.

One scheme that involves forwarding addresses and broadcasts when these turn out to be invalid has been implemented in the Emerald system as described in [Jul 89]. Another scheme is presented in [Theimer 85], but a common feature is that there is some unique way of identifying the migrated process.

2.9 Summary

In this chapter we defined some general concepts of load distribution and examined the following characteristics of a load distribution scheme:

Load Sharing vs. Load Balancing

Static vs. Dynamic Load Distribution

Initial Placement vs. Process Migration

Source Initiative vs. Server Initiative Algorithms

All of these characteristics have to be addressed when a load distribution facility is designed.

We also presented a view of a load distribution facility based on the functions it has to perform. This view divides the load distribution facility into two components: the information component and the control component.

The information component implements two policies: the information gathering policy that defines what information is collected and the information exchange policy that defines how information is distributed in the system.

The control component implements four policies: the initiation policy that defines when load distribution is considered, the selection policy that defines what work is redistributed, the location policy that defines where the work is distributed to, and the transfer policy that defines how the work is redistributed.

In our examination of the information gathering policy we presented a novel view of load indicators by dividing them into hardware load indicators (resource indicators) and software load indicators (activity indicators). This allows a consistent comparison of individual and combined load indicators, based on the system component the indicator measures.

The remaining policies are examined based on previous work on load distribution.

Chapter 3

Objects and Load Distribution

In the previous chapter we examined the theory of load distribution that has emerged from the experiences gained by doing load distribution in process based operating systems.

The object technology is fairly new and a uniform terminology is still missing in the field. We therefore start in 3.1 by defining some of the object related terms used in this thesis.

The work on load distribution presented in chapter 2 originated in process based operating systems. We examine some of the differences between process based and object based systems in 3.2, and identify some of the new issues in load distribution that arises in object based systems.

Most of the distributed object-oriented operating systems use the individual object rather than entire applications as the unit of distribution. This means that load distribution in object-oriented systems may be very different from load distribution in traditional process based systems. We have identified five such differences described in 3.3–3.7. These differences are: the finer granularity of objects that are described in 3.3, the encapsulation of data and code described in 3.4, the interaction among objects that is described in 3.5, shared objects are described in 3.6, and the duration of method invocation is described in 3.7. We do not analyse these differences in any great detail here, but simply list them so that they may be considered when the load distribution facility is designed.

A short summary of this chapter may be found in 3.8.

3.1 Object Terminology

The main concepts of object-orientation used in this thesis are defined briefly in the following. Our terminology is based on the one presented by Krakowiak [Krakowiak 93], but new terms have been added and some terms defined by Krakowiak are excluded because we do not use them in this thesis.

Object

An instance-object (or simply an object) is an encapsulated entity including data and code. The code is organized as a set of methods that define the permissible operations on the data.

The objects have interfaces that define those methods and data that are visible to the outside users of the system.

Class

A class describes the implementation of an object, i.e., the internal representation of data and the code for the methods. The class works as generator for the instance-objects, and are themselves represented by objects (called class-objects) in the system.

Object Binding

Binding is the operation where a specific piece of code is associated with a method. It is often done on a per object basis, i.e., a class-object is associated with an instance-object. We also use the term resolving references to refer to the binding of an object.

Binding may either be static which means that references are resolved at compile-time, or dynamic which means the references are resolved when the object is first referenced at run-time.

Persistent Objects

An objects is called persistent if the lifetime of the object is not limited by the application that created it. Support for persistent objects implies that secondary storage may be used to hold the objects when they are not in use, and some naming service is required to globally identify persistent objects, i.e., pointers or addresses may not be used to reference persistent objects.

The use of secondary storage to back objects do not imply any fault tolerance in our definition of persistence.

Granularity of Objects

Different object based systems have natural support for objects of different sizes ranging from a few instructions and a few bytes of data to full blown servers managing megabytes of data.

We define the granularity of objects in a system as the smallest natural size (with respect to both the number of instructions and the amount of data) of objects in that system.

A system where this granularity is small is called fine-grained while systems with large objects are called coarse-grained.

Granularity of Distribution

The granularity of distribution is defined as the smallest unit of execution that may be independently placed in the system. This unit of execution is called the unit of distribution in the system.

The unit of distribution need not necessarily also be the unit of load distribution, but the granularity of load distribution is always equal to or coarser than the unit of distribution.

3.2 Objects and Distribution

In the process based systems considered in the previous chapter, the unit of scheduling and the unit of distribution are the same (the process), so it is natural to make this the unit of load distribution. This is not the case in object based systems where applications are implemented by objects that may be distributed among several nodes in the system.

One view of object based systems illustrated by figure 3.1 — is that they implement a globally distributed object space, and that applications define subsets of that space.

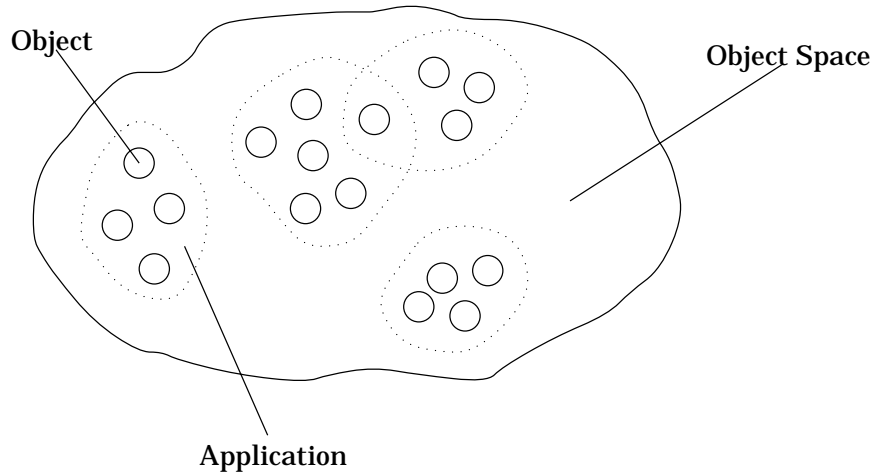


Figure 3.1: Applications define subsets of a common distributed object space.

This view of object based systems makes the choice of unit of load distribution less obvious, but also other aspects of objects and distribution have to be considered.

In the following we identify some of the problems introduced by objects in the field of load distribution. We comment on the problems, but generally do not try to solve them.

3.3 Granularity of Objects

Different granularities of objects exist, but objects are typically much smaller than traditional processes (an investigation in Guide-1 [Freyssinet 91] showed that more than 85% of all objects were less than 512 bytes).

Apart from the size, objects may be viewed as servers in a traditional client/server architecture. They manage some local resource (the data encapsulated in the object) that may only be modified through well-defined interfaces (methods or operations).

This analogy shows that objects have many features in common with the processes of traditional operating systems (a local address space and well defined interfaces for communication), although they do not necessarily allocate system resources (e.g. stack and kernel structures) normally associated with processes.

This similarity indicates that many of the lessons learned about load distribution from traditional process based operating systems, may be applied to object based systems, although the differences between process based and object based systems are fundamental.

The difference in granularity is probably the most significant difference among the two types of systems, with respect to load distribution, and many of the other differences identified below relate to the finer granularity in most object based systems.

3.4 Encapsulation of Data in Objects

A process can be regarded as a program containing a sequence of operations on some external data. This means that programs and data need to be looked up separately in the system and co-located in order for the process to run.

Objects on the other hand encapsulate data and only allow modifications through well defined interfaces (methods or operations). This means that data and code are located in the same operation.

This unification of data and code makes object migration much simpler than process migration (e.g. no references to open files need to be translated), which makes a migratory load distribution mechanism conceptually cleaner in an object based system.

3.5 Interaction among Objects

Processes define the outer scope of procedures or functions or even threads operating on the same data to achieve some common goal. This makes processes well defined and self contained entities.

Objects on the other hand are defined by the data they encapsulate and their interfaces to other objects. They may only implement a small part of the application and has to rely on other objects for the rest of the work.

This interaction among objects suggest that not only the individual object, but also those objects with which it interacts should be considered when load distribution decisions are made.

3.6 Shared Objects

One of the reasons for introducing object-oriented programming is the ability to reuse previously defined objects in several applications. Furthermore, it is common for object based applications to communicate through shared objects, as illustrated by figure 3.1.

Object reuse and sharing means that several applications may be actively using the same objects at the same time. It is therefore possible that object migrations that may lower the local load and improve the performance of some applications will hurt the performance of other applications because operations on shared objects have become remote.

Placement of objects should therefore not be based on the needs of a single application, but all applications that reference the object should be taken into account.

This shifts the focus in object-oriented load distribution from the applications to the object.

3.7 Duration of Method Invocations

Processes are the unit of execution (i.e., they implement the entire application), so off-loading them results in a permanent improvement of the local load. Objects only constitute a part of the application, and the thread of control may pass through many objects, so offloading work (e.g., by migrating objects or placing newly bound objects on another machine) may only improve the load for a short time, until the method invocation returns. This makes load distribution in object-oriented systems much more difficult, and the benefits of load distribution less certain.

3.8 Summary

This chapter defined our use of object terminology, and examined some of the differences between process based and object-oriented systems with respect to load distribution. We

did not analyse these differences in any great detail but limited ourselves to identifying them.

The following differences were identified: the granularity of distribution based on objects, the encapsulation of data and code, interaction among objects, sharing of objects, and the duration of method invocations.

Chapter 4

Previous Work on Object-Oriented Load Distribution

This chapter presents a study of previous work on load distribution in object based systems. We want to determine the unique problems with object based load distribution previously identified and the solutions (if any) proposed to solve these problems.

The following systems are studied: Emerald, Bellerophon, Amadeus, and Ellie. The presentation of the different systems is followed by the conclusions in 4.5.

4.1 Emerald

Emerald is an integrated language and object support system developed at the University of Washington [Hutchinson 87, Jul 88].

We present two approaches to load distribution in Emerald. First we describe the language components in Emerald that allows the application programmer to perform explicit load distribution, and then we focus on an implementation of an automatic load distribution facility in Emerald implemented by Morten Lehrmann [Lehrmann 93].

4.1.1 The Emerald System

Emerald is implemented as a kernel that runs in the context of a process on a host system (in this case UNIX). Inside this UNIX-process several threads of control may execute in parallel managed by a round-robin scheduler.

The object model allows both active and passive objects. Active objects contain a process which is activated when the object is initialized. Passive objects are manipulated by the processes of the active objects.

Applications are started by activating the process in an active object. This process may then invoke methods in other objects and possibly create other processes, that may run in parallel with the first one.

4.1.2 Explicit Load Distribution in Emerald

Object mobility is an integral and important part of the Emerald language [Jul 89]. It is expressed explicitly by statements in the Emerald language, by arguments to a parameter list, or by attaching objects together. This control over the location of objects allows the application programmer to implement explicit load distribution, using one or more of the facilities outlined below.

Migrating Objects Two statements are used to migrate objects in Emerald, these are **fix** and **move**. Both statements take two parameters, the first parameter tells which object to migrate (we call this object the migrant, in order to simplify notation), and the second parameter is an object on the node where the migrant should be migrated to.

The **fix** statement moves the migrant to the node of the other object and forces it to remain there, until it is released by an **unfix** statement, or a **refix** statement that is an atomic **unfix** and **fix** (presumably with a different object) operation.

The **move** statement is only a hint to the system that the migrant should be moved to the node of the second parameter. There is no guarantee that the object will actually move, but also no constraints on subsequent migration.

Parameter Objects The default method for passing objects as parameters is *call-by-reference*, this means that all operations on the parameter object is performed on the node where the object resides. It may be more efficient to migrate the parameter object to the node of the callee, which may be done as *call-by-move* or *call-by-visit*. In both cases the parameter object is moved to the node of the callee, and while it remains there in the first case, it is returned to the original node in the second case.

Attached Objects It is possible to build clusters of objects that are treated as one unit with respect to mobility. These clusters are created with the **attached** primitive at compile-time that instructs the system to always trigger relocation of the other objects, when one of the objects is relocated. Attachment of objects is transitive, so that large clusters may be build, and it is furthermore the only permanent way of co-locating objects in the Emerald language.

4.1.3 Automatic Load Distribution in Emerald

The automatic load distribution facility designed, and implemented by Morten Lehrmann uses two algorithms called the Load Distribution Algorithm (LDA) that offloads work from overloaded nodes, and the Remote Invocation Reduction Algorithm (RIRA) that tries to co-locate interacting objects by migrating the caller object to the callee when the load on the callee node is low. These algorithms will be described further in the control component, but first we focus on the information component used by the LDA.

Information Component The Emerald kernel runs on top of a UNIX system, so both Emerald and UNIX activity have to be considered by the information component. The chosen load indicator is the sum of the UNIX load (the length of the run-queue averaged over 60 seconds) and the current number of runnable threads on the local Emerald kernel.

The load indicators are requested from all nodes whenever load distribution is being initiated: a broadcast is sent to all nodes which in turn reply with their local load indicators.

Control Component (LDA) The LDA uses a sender-initiative migratory load distribution algorithm. It implements the following three policies: the Transfer Policy, the Selection Policy, and the Location Policy.

The Transfer Policy determines when load distribution is initiated. Load distribution is considered on each node every 60 seconds and is initiated if both the local load is higher than a threshold and a runnable Emerald thread exists in the local kernel. The

load threshold is dynamically determined, so that it express the relative load of the node compared to the rest of the system.

The Selection Policy determines which thread (object) to migrate. This policy chooses one of the runnable objects at random.

The Location Policy determines where the object is migrated to. This decision is based on the information obtained by the request for load information. When the request is broadcast, the local load is included with the request, this way nodes with a higher or only slightly lower load need not reply. After a specified timeout of one second, the answers are compared and the node with the lowest load is selected. If more than one node exist, the node that replied last is chosen because this information is most recent.

The randomly chosen object is then migrated to that node.

Control Component (RIRA) The RIRA collects statistics about object invocation patterns and uses them in an attempt to co-locate objects that interact much. The RIRA consists of an statistics gathering part and a movement decision part.

The statistics gathered for each object consist of the number of invocations the objects have performed of objects on each node in the system.

The RIRA is activated when the LDA has been idle for some time, i.e., when the overall activity in the system is low. The RIRA takes effect when remote invocations are performed, if enough invocations to that node has been recorded in the statistics, the calling object is migrated to that node and the statistics are cleared.

This algorithm should eventually locate all the interacting objects on the same node.

Experiments with Automatic Load Distribution Several experiments have been performed with the load distribution facility, but two deserve to be mentioned here. The first experiment with a CPU-intensive benchmark shows a modest improvement in performance with the LDA. The second experiment uses a varying workload to determine if the dispersion and retraction mechanism of the LDA and RIRA offers any performance benefits. The conclusions from this experiment are that the LDA alone hurts performance, while the combined mechanism offers performance benefits, as long as the variation in workloads are limited.

4.1.4 Conclusions for Emerald

Emerald includes powerful language constructs to build and migrate clusters of objects. The strong support for co-locating objects indicates that the designers of Emerald expect this to be an important performance parameter.

The modest speed-up in the experiment with automatic load distribution indicates that the unit of load distribution is too fine (it takes too long to offload work from a node). The success of the second experiment suggests that the dispersion and retraction mechanism should be considered for object-oriented load distribution. It also underlines the importance of locality of reference.

4.2 Bellerophon

Bellerophon has been developed as part of a Ph.D. project at the Cambridge University Computer Laboratory [Dickman 92a] to investigate scalable object management mechanisms for object support in networks of autonomous computer systems.

4.2.1 System Overview

The Bellerophon system appears to be a single widely distributed pool of objects, where only a part is known to each application. Distribution is transparent to applications, but the transparency may be broken in the following way: each distinguishable location has an associated immovable object, and the co-locator and contra-locator, described later in this section, may be used to ensure that an object remains at the desired location.

The object model of Bellerophon supports two main concepts, references and the objects they refer to. Objects have identity and may be referenced and located separately in the system. References are the only way of accessing objects offering both transparent access to and location of objects.

Two special forms of references are defined by Bellerophon called co-locators and contra-locators. Co-locators are used to keep objects together, e.g. to improve efficiency, while contra-locators may be used to keep objects apart, e.g. for redundancy of replicates or to ensure physical distribution. Objects with mutual co-locators located on the same node will remain together, i.e., if one of the objects are migrated the other will also be migrated. Objects with mutual contra-locators placed on separate nodes will remain apart, i.e., an attempt to migrate one of the objects to the node of the other will fail.

4.2.2 Load Distribution Overview

The Bellerophon load distribution facility uses structural information about object invocation patterns, obtained from the garbage collector, to identify clumps of objects that should be migrated to the same node.

In this scheme all objects in the system are examined, but only the clumps are considered for load distribution.

4.2.3 Information Component

Two types of information are gathered in Bellerophon, these are information about object invocation patterns and information about the state of the system.

The following information is gathered about the invocation patterns of each clump of objects in the system. References and objects have attached counters that indicate the number of messages passed. Frequent interactions are indicated by a high message count of one reference compared to the total for the referenced object, and lead to the combination of the containing clumps. These message counts are gathered by the garbage collector that regularly examines all objects in the system, in order to detect local and distributed collections of objects that are no longer in use by any application.

The system state is described by multiple load indicators (e.g. average CPU and memory loads) that are compared individually.

4.2.4 Control Component

Load distribution is only considered within a subset of the entire Bellerophon system, which improves scalability with respect to nodes in the system. Each node maintains a set of so called “buddies”, nodes that are selected locally as candidates for load distribution, or have selected this node for that role. Buddy sets are dynamic and selections are made with a bias toward nodes where remote dependencies already exist.

The load distribution facility is executed at slightly-randomized, but essentially fixed, intervals to avoid the risk of phase-locking between an application and the load distribution facility. The buddy set is then examined for load imbalances, and load distribution is

considered if it appears worthwhile, i.e. if sufficiently large imbalances exist in the buddy set.

The load distribution scheme in Bellerophon tries to identify clumps of objects that should remain together and migrate these clumps to nodes so that imbalances in the system may be rectified. The cohesiveness of a clump (the ratio of internal to external activity) and its size is the primary determinants for the clumps usefulness for load distribution. In addition the weight of the references to a candidate node and the absence of contra-locators are factors when the clump is considered for load distribution.

If suitable clumps for the existing imbalances can be found, these clumps are migrated to the candidate node.

4.2.5 Conclusions for Bellerophon

Experiments with Bellerophon have also shown that load distribution can be integrated with the distributed garbage collector without prohibitive additional costs (the overhead of the combined mechanism is within the same order of a magnitude albeit somewhat higher).

4.3 Amadeus

Amadeus [DSG 91] is a prototype implementation of the COMANDOS¹ platform, developed at Trinity College Dublin. Amadeus runs on top of ULTRIX on a network of workstations.

The Amadeus load distribution facility [Tangney 91, Tangney 92] is included because of the close similarity between Amadeus and Guide-2, for which we intend to implement a load distribution facility.

4.3.1 System Overview

The execution unit of the Amadeus system is a distributed address space called a job, which roughly corresponds to an application. Within each job several separate activities may execute concurrently, possibly spreading onto several nodes. Thus, a job may be composed of several local address spaces called contexts, one on each node where the activity is present. Objects in the Amadeus system are passive, and are manipulated by the activities.

Clusters of objects Objects are clustered to improve performance and limit the allocation overhead of disk blocks and memory pages. The cluster is both the unit of mapping and distribution in Amadeus, and therefore also the finest possible granularity of load distribution.

Clusters are created by the programmer with the `StartNewCluster()` command that also changes the default cluster to the newly created cluster. Objects are created in the default cluster so all subsequently created objects will be placed in the new cluster. `StartNewCluster` returns a unique cluster identified (CID) that may later be used to change the default cluster(`ChangeDefaultCluster(CID)`).

¹The execution structures and object model of Amadeus is very similar to Guide-2. We therefore limit the system overview of Amadeus to an absolute minimum and refers to the description of Guide-2 in chapter 6.

Dynamic Binding of Objects The first time that an activity invokes a method in an object, the cluster in which the object resides is mapped into the virtual address space of the current context of the activity. To ensure data consistency, objects can only be mapped on one node at a time, which means that not all objects can be mapped locally. If the object cannot be mapped locally, a remote invocation will have to be performed. This is done to another context belonging to the same job, but on the node where the object resides, in turn creating this context if it doesn't already exist.

4.3.2 Load Distribution Overview

The Amadeus load distribution facility supports distribution of activities and clusters. The type of distribution is decided when the system is initialized.

Activity distribution is performed when activities are created. The load distribution facility is then invoked to assign a preferred node to the activity. The activity will map clusters on the preferred node, unless they are already mapped in another context. This ensures that the majority of the activity's work will be performed on the preferred node.

Cluster distribution is performed when clusters are being mapped, in this case, activities may execute on several nodes, depending on where the clusters are mapped.

4.3.3 Experiments with Load Distribution in Amadeus

The experiments with load distribution in Amadeus focus on applications that are structured with a number of cooperating activities.

Distribution is transparent to the application programmer so the load distribution facility is free to place activities on different nodes resulting in physical parallelism. Furthermore activities may be placed on the nodes with the lowest load.

Two experiments with activity distribution are reported in [Tangney 91]. These experiments are carried out using the same CPU-intensive benchmark, structured with a number of workers (activities) computing in parallel. The objects of each worker is stored in a separate cluster.

The benchmark performs a number of floating point calculations, and may be controlled with the following parameters.

- The number of times the calculations are done, i.e., the size of the computation.
- The number of workers involved in the computation, i.e., the granularity of parallelism.
- How frequently the workers should synchronize with each other, i.e., the computation to synchronization ratio.

The first experiment investigates granularity of parallelism for which the system offers effective speed-up. The numbers of workers and synchronization are fixed while the amount of work and the number of nodes are varied to measure the speed-ups. This investigation shows that effective speed-ups are obtained when the workers run for 1 second or more, i.e., the preferred granularity for load distribution in Amadeus is fairly coarse grained.

The second experiment investigates the influence of communication on the performance obtained by load distribution. The amount of work, and the number of workers are fixed while the number of synchronization are varied. This investigation shows acceptable performance with modest synchronization, while the performance suffers when the computation to synchronization ratio gets too high.

4.3.4 Conclusions for Amadeus

Two load distribution schemes are proposed for Amadeus: distribution of activities and distribution of clusters. Activity distribution is driven by the thread of control while cluster distribution is driven by the location of data.

No experiment with cluster distribution has been reported, but the experiments with activity distribution shows that the load distribution facility is rather coarse (the activity should have a run-time of 1 second or more to achieve a speed-up).

Correct clustering is important. The experiments in Amadeus indicated that modest amounts of communications (in this case synchronization between master and slave) are acceptable but once the communication to synchronization ratio gets too high performance suffers.

4.4 Ellie

The Ellie language [Andersen 91a, Andersen 91b] is designed as a general purpose, fine-grained, object-oriented programming language intended for programming of Multiple Instructions Multiple Data (MIMD) type parallel computers. Parallel computers exist in a multitude of different configurations, ranging from a few processors to thousands of processors, so Ellie defines fine-grained object and execution models to allow applications to exploit whatever hardware is available.

4.4.1 System Overview

The fine granularity of the Ellie execution model means that applications are implemented by a large number of small processes (possibly only a few instructions). Parallelism among these processes is determined by the application programmer, while the distribution is left to the system, so when the number of processes is much larger than the number of processors, several processes may execute on the same processor.

Parallelism is expressed by the means of bounded and unbounded remote procedure calls (RPC and URPC). Bounded RPCs are synchronous method invocations, while unbounded RPCs are asynchronous calls that return a future result object immediately and allows the calling process to continue. The future result object is an abstract object with a state that is undefined until the URPC returns. If the object is accessed in the undefined state, the process is blocked until the URPC returns and the state is defined.

The object model allows both active and passive objects. Processes are associated with active objects and may invoke methods in both active and passive objects. Objects may reside inside other objects, in this case the innermost object is called a local object, while the outermost object is called the origin object.

Creation and management of the large number of processes in an application involves a large overhead that is unnecessary when the processes are executing on the same node on behalf of the same application.

4.4.2 Load Distribution in Ellie

The load distribution scheme proposed for Ellie [Andersen 92a, Andersen 92b] employs a fine-grained dynamic load distribution scheme. The following problems with this scheme are identified: Load information is most likely out of date when processes are active for a short period, and the overhead of distribution may be comparable to — or even larger than — the computation time of the process. The proposed solution to these problems

is to adapt the granularity of the application to the granularity of the host computer, by using information about the application obtained at compile-time, i.e., first compile-time analysis of the program is used to adapt the granularity of distribution to the target computer and secondly a dynamic load distribution scheme is used to place and possibly migrate these units of distribution.

Only a part of this two phased scheme has been implemented. Instead of the compile-time analysis, an annotation workbench has been implemented that allows the programmer to supply the load distribution information. No information component is implemented so only static load distribution is possible in Ellie. The annotation workbench has been used to experiment with static load distribution based on load distribution information supplied by the programmer. This will be described in more detail later.

Although the compile-time analysis has not been implemented some elements of the analysis has been considered. We will take a short look at the compile-time analysis, before we describe the annotation workbench and some of the results it has produced.

Compile-Time Analysis The purpose of the compile-time analysis is to provide the necessary information for the grain-size adaption.

The following hints have been considered for Ellie:

1. Place new object at callers node, hereby eliminating communication overhead.
2. Place new object at origin objects node, hereby eliminating communication overheads for accessing the origin object.
3. Place new object at parameter objects node, hereby eliminating communication overheads for accessing the parameter object.
4. Attach new object to caller, with possible migration of caller object.
5. Attach new object to origin object, with possible migration of origin object object.
6. Attach new object to parameter object, with possible migration of parameter object.

To produce these hints the compiler must analyse the dependencies between objects, and build clusters of closely dependent objects that should reside on the same node. Hint 1–3 concern the initial placement of objects, while hint 4–6 are for building clusters of attached objects.

Annotation Workbench The annotation workbench allows programmers to supply the information that the compile-time analysis is supposed to provide. This is done by placing annotations in the beginning of object constructors, to tell the load distribution mechanism where to place the object. The following annotations are possible:

1. Hint: Caller placement (callers node)
2. Hint: Origin placement (origin objects node)
3. Hint: Parameter placement (parameter objects node)
4. Hint: Migrate caller object to new object
5. Hint: Migrate origin object to new object

6. Hint: Migrate parameter object to new object
7. Claim: Bounded reduction (reduce the degree of parallelism by reducing URPC to RPC)
8. Claim: Local reduction (create the future result object locally)
9. Claim: Included reduction (physical combination)
10. Relax: Nodes Nearby (not necessarily the same node)

The annotations 1–3 correspond to the hints 1–3 from the compile-time analysis and the annotations 4–6 correspond to the hints 4–6. These annotations are only used to control the placement of objects. The claims are used to adapt the granularity of objects and processes to the target computer, i.e., they may be used to experiment with different granularities of “virtual objects”. The last annotation is used to relax a hint or claim for placement on the same node.

The annotation workbench has been used to investigate different load distribution strategies with a single application, that calculates the twelfth Fibonacci number, executed on a network of 16 transputers organized in a wrapped mesh. Different annotations generate different granularities, but the variation of granularity within the application is large. This means that the experiment does not give an indication of the impact of different granularities on the benefits of load distribution.

A comparison of run-times for the application, shows the shortest run-time when only the recursive Fibonacci calls are distributed, while the run-times grow longer when more work is distributed (additions, subtractions and integer comparisons). This indicates to us that the overhead of fine-grained distribution is too high to allow benefits of distribution. Another result of this experiment is that the use of the future result object allows physical parallelism and may improve the performance with an order of a magnitude (from 1.3 to 15 seconds).

4.4.3 Conclusions for Ellie

Ellie supports very fine-grained objects and distribution (down to the size of a single arithmetic operation), but the experiment with load distribution of the Fibonacci application indicates that this granularity is too fine. The importance of clusterization has been acknowledged and different ways of controlling the initial clustering are proposed.

4.5 Summary of Conclusions

The modest speed-ups in the first experiment with Emerald suggest that individual objects may be too fine-grained. The second experiment underlines the importance of locality of reference.

The implementation of load distribution in Bellerophon is very interesting, because it shows that intricate load distribution schemes may be implemented without a prohibitive overhead as long as they are integrated with other system components. However, no performance measurements are reported, so we do not know how well the scheme works.

The experiments in Amadeus show that clusterization is an important issue, and that remote invocations may hurt performance.

Ellie implements a static load distribution facility, and is primarily included in this survey because the annotation workbench shows the type of hints that a programmer may provide to support initial clustering performed by the compiler.

Co-location of objects is an important issue. All the surveyed systems support mechanisms to keep objects together at all times. Furthermore experiments with the RIRA in Emerald and synchronization in Amadeus both support this conclusion.

The grain-size of individual objects may be too fine-grained, to warrant performance benefits from load distribution.

Chapter 5

Granularity of Object-Oriented Load Distribution

In this chapter we examine two of the new issues in object-oriented load distribution raised by the finer granularity of distribution in object based systems. This examination is based on discussions of the two previous chapters.

We first examine the increase in scale caused by using fine-grained objects as the unit of distribution. This is followed by an analysis of the impact of fine-grained distribution on the expected performance benefits achieved by load distribution. We propose to solve the problems identified above by grouping objects into clusters, which decreases the scale and increases the granularity so that techniques originated in process based load distribution may become directly applicable in the object-oriented environments.

The increase in scale is examined in 5.1, the impact of fine-grained distribution is examined in 5.2, and the solution by clustering objects is examined in 5.3. A summary of this chapter may be found in 5.4.

5.1 The Scale of Object-Oriented Load Distribution

Object based applications may consist of several thousands of objects depending on the size of the application and the granularity of the object support system. If all objects are eligible for distribution, the size of the load distribution problem (deciding which objects should execute where) explodes in size. Dealing with this explosion in scale is a major concern when designing an object oriented load distributor.

We have identified three ways of dealing with this increase in scale, namely making the mechanism extremely light weight, integrating the load distributor with other system support (e.g., with the garbage collector like in Bellerophon) so the cost of the mechanism is shared by several services or increasing the granularity of load distribution, e.g., grouping objects into clusters, as in Amadeus, so the cost of the mechanism is amortized by several objects. Each of these solutions will be outlined in the following.

5.1.1 Light-Weight Load Distribution Mechanisms

With an increase in scale of the load distribution problem, the amount of work done for each object has to be proportionally smaller to ensure the same improvements in performance, unless the change in granularity by itself improves the performance of applications.

5.1.2 Integration with Other System Components

Another way of dealing with the increase in scale is to integrate the load distribution facility with any system component that references all objects in the system.

This approach is used with object migration in Bellerophon where the load distribution facility is integrated with the garbage collector that regularly examines all objects to locate objects that are no longer in use by any application.

Other system components could be the name server or the object storage facility in object based systems with persistent objects. These components are typically used when objects are initially referenced by an application, so it seems natural to use them in an initial load distribution scheme.

5.1.3 Increasing the Granularity of Load Distribution

Increasing the granularity of load distribution is probably the simplest approach. Many distributed object based systems already include support for groups of objects (attached objects, co-locators, or clusters). The technology behind this support is well known, and it should be easy to implement it in systems that do not provide this support. Apart from load distribution, this support may also be used to ensure locality of reference and thereby improving performance of distributed applications.

Considering groups of objects for load distribution increases the granularity of load distribution, but it also means that all objects should belong to a group in order to benefit from load distribution. We examine the problem of defining these groups in 5.3.

5.2 Grain-Size of Load Distribution

Fine-grained object-oriented systems, often offer fine-grained distribution, i.e., each object may be located separately. The experiments in Emerald indicated that this granularity may be too fine to allow performance benefits by load distribution. In the following we examine the possible influence of grain-size on the performance benefits offered by load distribution.

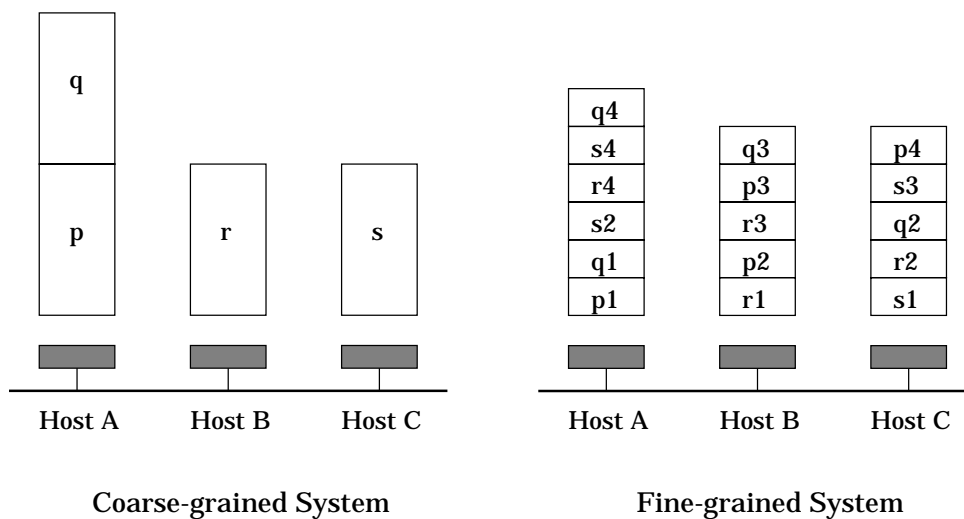


Figure 5.1: Fine-grained load distribution allows a evenly balanced load in the system.

5.2.1 Fine-Grained Load Distribution

A fine-grained unit of load distribution makes it possible to balance the work evenly among the nodes in the system, which is illustrated by figure 5.1.

The more even distribution of work, means that the system resources will be shared among the applications in a fair manner, and that the average run-times of applications will be shorter because of lower contention for the system resources (this last point is not illustrated by the figure above).

The granularity also determines the frequency of load distribution with an initial placement scheme, where a fine granularity allows frequent load distribution.

5.2.2 Performance Benefits

The performance benefits of load distribution is primarily gained from executing objects on nodes with less contention for the CPU, i.e., on nodes where the objects will be scheduled more often. This suggests that the speed-up (here defined as the differences in distributed and non-distributed run-times) will be a linear function of the objects run-time, i.e., when the objects are scheduled twice as often, they will execute twice as fast. This linearity is illustrated by figure 5.2 below.

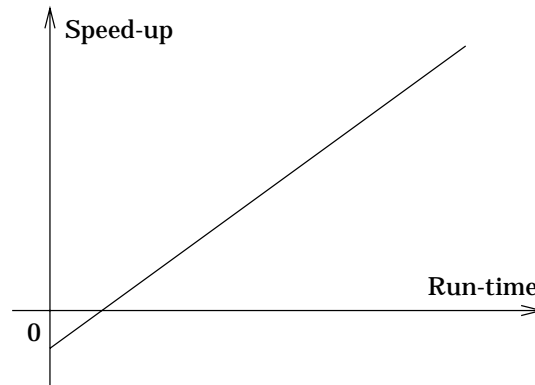


Figure 5.2: Expected absolute speed-up by load distribution as a function of the object run-time.

The initial negative speed-up above illustrates that the performance benefit may be smaller than the overhead of performing load distribution. This overhead is examined further in 5.2.3.

The linear behaviour should not be taken literally, because memory contention and external devices are not considered. However, we believe that the CPU is still the main source of contention, so the cumulated effect of these “minor” factors should not change the overall increase in speed-up with increasing run-times.

This suggest that a mechanism that discriminates against small objects by only considering larger objects for load distribution could significantly improve the performance of an object-oriented load distribution facility. One such mechanism is the History filter described on page 30.

5.2.3 Overhead of Load Distribution

Performing load distribution involves an overhead, first to choose a candidate node for load distribution and then to actually distribute load to that node.

The overhead of the placement policy is inevitable, although it may be small, and the lower bound for this overhead depends on the system. The overhead of distributing load may be totally avoided in some cases, e.g., using an initial placement scheme with a central object store. In this case cost of moving the object from the object store to the node of execution has to be paid anyway.

The effect of the cumulated overhead is illustrated by figure 5.3.

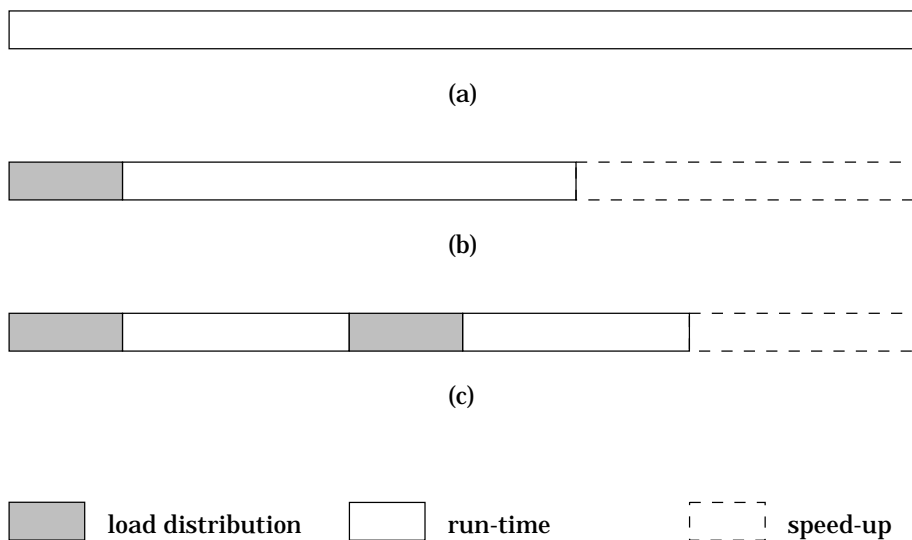


Figure 5.3: The effect of the overhead of load distribution.

The figure shows the benefit of off-loading an application that runs on a node where resources are scarce (a) to a node (b) with twice the amount of resources, so that the run-time is halved. The overhead of load distribution means that the execution time of the application is not halved. The same application distributed to the same node is shown in (c), but the grain-size of load distribution is halved, so the overhead of load distribution is paid twice resulting in a smaller speed-up.

This reduction in speed-up indicates to us that the overhead of load distribution has to be considered when the granularity of load distribution is determined.

Fine-grained load distribution implies that the overhead of load distribution is paid more often, and may in the end turn out to slow applications down. This happens when the overhead of load distribution becomes large compared to run-time of the unit of load distribution.

5.3 Clusters of Objects

One solution to the problems identified above is grouping objects into clusters that are used as the unit of load distribution. Clusters increase the granularity of load distribution, thereby limiting the problem of scale in object-oriented load distribution and increasing the possible performance benefits by load distribution.

Efficient clustering of objects is a new area of research with many unanswered questions, and solving this problem is beyond the scope of this thesis. Instead, we limit our scope to identifying and examining some of the problems that must be addressed when objects are clustered for the purpose of load distribution.

We first examine the problem of deciding which objects to group together in a cluster. We then examine how this clustering of objects may be performed either explicitly by the application programmers or automatically by the object support system. In both cases clusters may either be dynamic, which means that objects may be regrouped into new clusters or migrated between existing clusters, or static which means that objects remain in the same cluster throughout their lifetime but new clusters may be created and objects added to existing clusters.

5.3.1 Grouping Objects into Clusters

Although we do not claim to know how objects should be clustered, some of the issues in clustering objects have been identified and will be presented in the following.

Locality of Reference

Objects in the same cluster are scheduled together, so operations on objects that reside in the same cluster are ensured locality of reference, while operations on other objects may be remote.

Locality of reference is especially important when operations among the objects are frequent or large pieces of data are passed as parameters to the operations. These types of invocation patterns have to be identified and the objects co-located.

Determining invocation patterns may be difficult without executing the application and gathering statistics. This is especially true for the data sizes of parameters when method parameters contain polymorphic object types or other objects with dynamic types (e.g. strings).

Statistics gathered about the interaction among objects should be based on the frequency of operations instead of a simple reference count, e.g., counters could be zeroed periodically. This is important for applications with long run-times, where reference counts may eventually grow large although objects rarely interact.

Grain-Size of Load Distribution

The grain-size of load distribution was examined in 5.2. Which grain-size to aim for is still an open issue, but if an optimal grain-size exists then clusters should obviously be adapted to this grain-size.

The examination in 5.2 and the experiments in Emerald together with our investigation into the influence of different grain-sizes presented in 9.6.1, suggest that clusters should probably be fairly large.

Exploiting Physical Parallelism

A third concern is exploiting parallelism in applications. Objects in the same cluster are scheduled on the same node, so obviously objects that are supposed to execute in parallel have to reside in different clusters.

This means that clustering should be based on techniques to determine possible parallelism in applications. Such techniques generally involve manipulation of large graphs, so it should not be performed at run-time. Instead, parallelism should be identified during a

compile-time analysis of the application, that may either directly cluster objects or generate instructions like the co-locators and contra-locators from Bellerophon, to be used by the runtime system.

Parallelism should obviously be weighted against the other two problems defined above. One of the results of the experiments with Ellie was that some parallelism could successfully be sacrificed as long as sufficient parallelism existed in the system to utilize all nodes.

5.3.2 Explicit Clustering

Explicit clustering means that the application programmer is responsible for identifying clusters of objects. This method is proposed for Amadeus, Ellie, and the explicit load distribution in Emerald.

The large number of systems that support some form of explicit clustering indicates that it is fairly simple to implement.

Dynamic Clustering

Dynamic clustering means that objects may be regrouped at all times, i.e., objects are allowed to migrate between different clusters. This means that two units of distribution will be used actively at the same time: objects and clusters.

The primitives to support grouping and regrouping of clusters should allow the application programmer to create a new cluster, migrate an object between two existing clusters, and merge two clusters. Objects should be created in an existing cluster, or in a new cluster that may be merged with other clusters.

Static Clustering

The application programmer defines the objects that should be placed in the same cluster. The decision to cluster objects should be based on his knowledge of the object invocation patterns, so that locality of reference and the size of the parameters passed among objects in different clusters is optimal.

One problem is that static clusters are eternally growing entities, because objects may only be added. Garbage collection is impossible if objects are persistent, so clusters may eventually grow large enough to make distribution prohibitive or objects have to be placed in less optimal clusters to avoid this problem. However, it is difficult for the application programmer to know exactly when the cluster has grown too large, so the problem cannot be totally avoided.

5.3.3 Automatic Clustering

Automatic clustering is transparent to the application programmer. The system is responsible for identifying and building clusters of objects. This approach has been proposed for Bellerophon.

Dynamic Clustering

The system may use statistics about invocation patterns to perform dynamic clustering. This way objects that interact frequently have high probability of being located in the same cluster, i.e., the locality of reference is likely to be high. Automatic adaption of the cluster size to an optimal grain-size may not be an issue, but it should be possible to do so.

The statistics gathered about objects can not be used to determine parallelism in applications. Here we have to rely on the application programmer or the compiler to provide information (e.g. co-locators and contra-locators in Bellerophon) to ensure that objects that should execute in parallel are placed in different clusters.

Static Clustering

Doing static clustering automatically means that clusters are build by the system before the application is executed, i.e., clustering is based on information obtained through compile-time analysis of the application.

The compile-time analysis should be able to take advantage of parallelism within the applications, but other problems are associated with this approach. First of all object invocation patterns are difficult to establish at compile-time, because the input to the application is not known. It may be possible to identify objects that interact often, but not necessarily objects where large parameter data is passed in method invocations. The problem of growing clusters can be identified by the system and objects be placed in alternative clusters.

5.4 Summary of Object-Oriented Load Distribution

Object-oriented load distribution raises a number of new issues, some of which have been identified in this chapter. These issues are summarized below.

Scale If individual objects are considered for load distribution, the number of units of distribution that has to be considered by the selection policy may increase by several orders of a magnitude.

Grain-Size The performance benefits of load distribution should more than pay the overhead of the load distribution facility and the cost of relocating objects. Both these costs have a lower bound, so the grain-size of load distribution should be large enough to warrant sufficient performance benefits.

Clustering Objects may be grouped into clusters, so that the overhead of load distribution is amortized by all the objects in the cluster. This may solve the two problems defined above, but poses the new problem of identifying the objects that should be grouped in the same cluster.

Most of these issues relate directly to the finer granularity of distribution offered by most object based systems. In fact, we claim that the proper response to this fine granularity is the main issue in object-oriented load distribution. The examination of these issues presented in this chapter indicates that the granularity of load distribution should be coarser than the individual object.

Chapter 6

The Guide-2 System

This chapter describes the Guide project and the Guide-2 system. It is important to understand the goals of the Guide project and the type of applications it aims to support in order to define a proper role for the load distribution facility.

We therefore start in 6.1 by giving a short introduction to the goals and the history of the Guide project.

We present a general description of the Guide-2 system in 6.2, with special focus on the object and execution models defined for Guide. These object and execution models are supported by a layered virtual machine, known as the Elliott kernel.

We do not present a full description of the Guide-2 system and the Elliott kernel, but describes Elliott in sufficient detail to identify the different possibilities for distribution in Guide-2 (section 6.3).

These possibilities are examined in 6.4, which allows the discussion of load distribution in Guide and the design and implementation of the Guide-2 load distribution facility described in chapter 7.

6.1 Introduction to the Guide Project

The main focus of the Guide project is to explore distributed computing in an object-oriented environment, based on a set of possibly heterogeneous workstations connected by a local area network. The Guide-2 system is a general purpose object support system, but it has been especially designed for the office environment, i.e., it may not be optimal for large scale number crunching like weather forecasting or graphical animation in video games.

One of the major goals of the Guide project is to provide location-transparent interfaces to objects and high-level services, e.g. visualization of object invocation patterns, which may be used to optimize, debug, and gradually evolve applications.

The project started in September 1986 and has since then passed through two phases. A first version of Guide, now known as Guide-1, was implemented as an integrated language and run-time environment on top of Unix. The Guide language [Krakowiak 90, Nguyen Van90] defined in the context of this project is an object-oriented programming language that supports programming of distributed applications. Each application may define several threads of control executing in parallel (possibly on different machines) and communication among these threads is done through shared objects. The distribution of applications is transparent to both users and the application programmer.

The Guide-1 implementation provides an adequate test bed for investigating the development of distributed applications, and Guide-1 is still used by the people working

with object-oriented distributed computing and high-level services, but it suffers from limitations both in functionality and performance. To overcome these limitations a second version known as Guide-2¹ has been implemented on top of Mach 3.0.

Although the Guide language and the Guide-1 object model are still supported by Guide-2, the second version has been designed to provide generic support for multiple object-oriented programming languages and object models at the same time. This provoked the separation of language and object support system illustrated by figure 6.1.

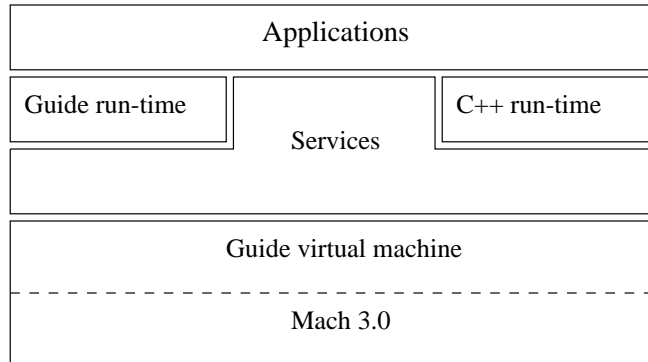


Figure 6.1: The Guide-2 system.

Applications written in an object-oriented language supported by Guide-2, runs on top of a language specific run-time system, but they may also access general services provided by the service layer of Guide-2. The language specific run-time libraries interface with the generic object support functions of the virtual machine and the system services, to provide object support facilities with the appropriate language semantics.

6.2 Overview of Guide-2

The Guide-2 system illustrated in figure 6.1 consists of three major components: language specific runtime systems, system services, and the Guide-2 virtual machine. We give a short overview of each of these components, followed by a presentation of the generic object model that may be used by the language specific runtime systems to define their specific object models. This is followed by a presentation of the generic execution model and the object management facilities supported by the virtual machine.

Language Run-times The language specific runtime systems map the language specific structures and behaviour to the generic system defined by the primitives of the system services and the virtual machine.

Two languages are of particular interest to the Guide project: the Guide language, and C++. The Guide language is interesting because it constitutes the basis for the definition of the virtual machine, i.e., the interfaces of the virtual machine has been developed especially with this language in mind. C++ is especially interesting because of its great popularity.

System Services The system services are themselves object based applications built on top of the virtual machine, possibly in one of the supported object-oriented languages.

¹The terms Guide-1 and Guide-2 will be used to describe the specific implementations, while the general term Guide will be used to describe the entire project or properties common to both implementations.

An example, of a system service, is the name-server that associates human comprehensible names with system level object references.

There is no inherent difference between Guide-2 applications and the system services, from the point of view of the virtual machine, because the services are object based just like the applications. The main difference between applications and services is that applications are managed by users, while the services are managed by the System Manager, who decides their protection, and their modes of sharing, naming, and access.

Virtual Machine The Guide-2 virtual machine has been designed to provide support for development and execution of applications structured in an object-oriented way. The system is designed to concurrently support several languages which requires the Guide-2 virtual machine to be very general. Only the basic storage and execution structures, expressible in different languages are defined. Although the Guide-2 virtual machine provides the basic support for management of these specific structures, it assumes no knowledge about the organization of the actual execution structures. We describe the virtual machine in greater detail in 6.3.

6.2.1 Generic Object Model

The generic object model defines three entities: instance-objects (objects), class-objects (classes), and code libraries. Objects are defined similar to the definition in 3.1, i.e., the encapsulation of data and methods that operate on that data. They are instances of classes that describe the internal representation of data and implement the code for the methods. The actual code is stored in the code libraries that are invisible to the users unlike objects and classes.

Explicit references are maintained between these entities, i.e., objects have references to their class, classes have references to the code libraries that implement the code for the methods, and code libraries may have references to other code libraries that implement part of the code.

The relationship between objects, classes, and code libraries is illustrated in figure 6.2.

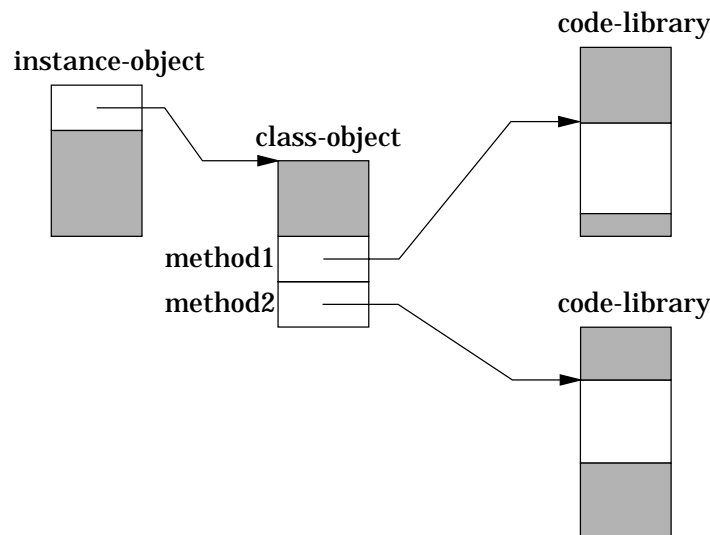


Figure 6.2: The generic object model.

The classes are organized in a hierarchy according to the relation *is-a-subclass-of*. The details of this organization and the effect it has on the applicable operations for the objects depends on the language in use. The virtual machine does not define a model for the organization of classes, but provides mechanisms that permits the implementation of different models — particularly for multiple inheritance (i.e., that a class may be sub-class of several classes).

Objects are named by universal references, which is some unique information that allows the system to identify and find it. The system provides several kinds of references that differs by their scope and the duration of their life time. This allows the management of objects known in different environments and with different life time. Particularly the universal references (independent of addressing context) allows the management of persistent objects.

Persistent objects has a lifetime that is independent of the application that created the object. This means that the objects need to reside on secondary storage, when they are not in use by applications.

Objects are passive entities managed by the structures defined in the execution model.

6.2.2 Generic Execution Model

The generic execution model of Guide-2 defines two units of execution: jobs and activities. A *job* is a distributed address space where objects are made accessible to applications that are generally implemented by a single job. *Activities* are threads of control that execute inside a job. Each job may have several activities executing on each of the machines where the job is present. The number of hosts where a job is present and the number of activities and objects in the job, may evolve dynamically throughout the life time of the job.

The distributed address space of a job consists of a collection of local address spaces called contexts. A *context* is a piece of homogeneous address space (virtual memory) present at one host.

This decomposition of jobs into contexts, instead of using one large distributed shared memory, is motivated by the lack of a sufficiently large uniform address space (Guide-2 is implemented on a 32-bit architecture), but also by protection concerns because it prevents incorrect written code from corrupting data outside the local context.

A distributed job has at least one context at each host, where the activities can execute. Activities executing in the same context share the same virtual address space.

When jobs are created an object and a method in this object have to be specified. After creation the job consists of a single context that contains the initial object and an activity, called the *principal activity*, which executes the specified method in the initial object. This activity may then in turn create other activities in the job. The execution of an activity is a sequence of synchronous calls of methods in objects.

Activities are also distributed entities and a method call may change the context if the object resides on a remote host. The activity will make an *extension* into a context belonging to the same job on that node, creating it if one is not available. The method call will then be performed as a local method call from the extension. This mechanism is called *diffusion*. The execution structures of Guide are shown in figure 6.3, where the activity a2' is an extension of a2, i.e., a2 has diffused from Host A to Host B.

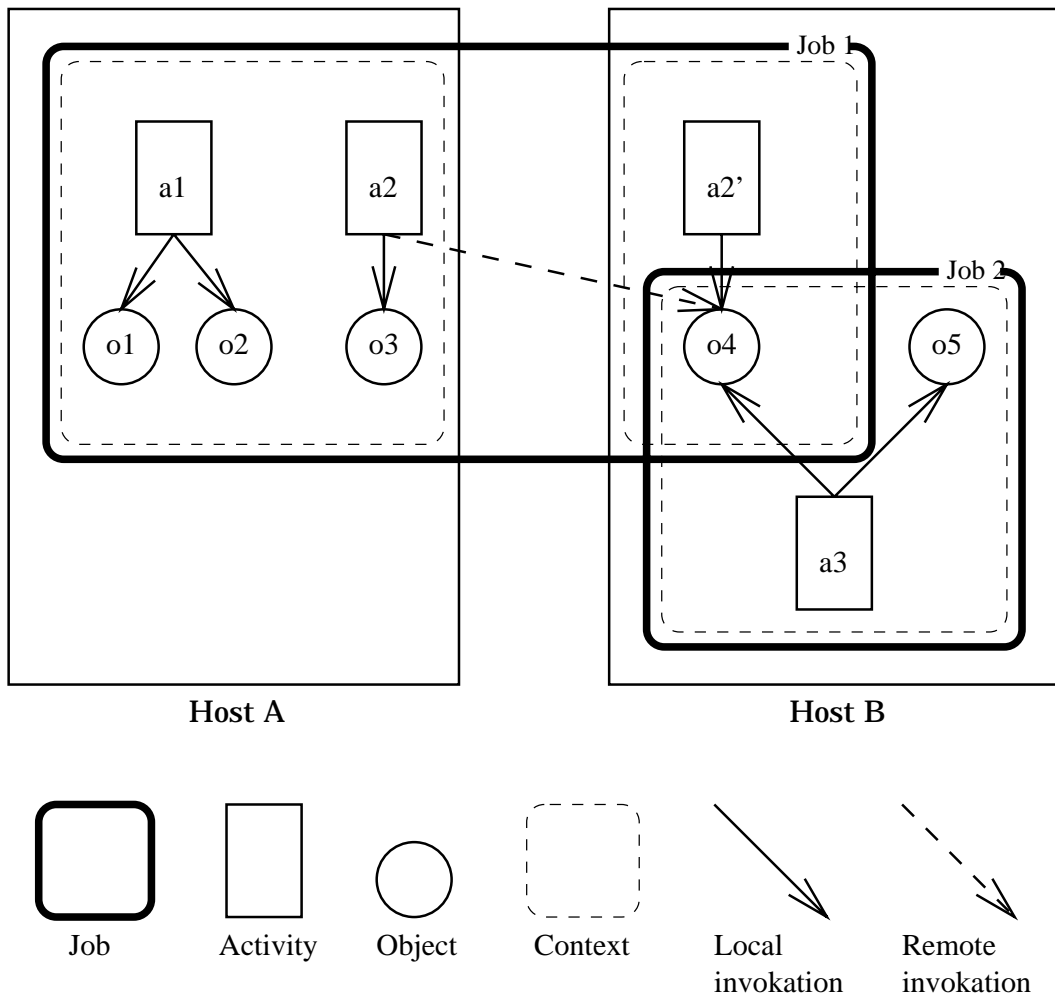


Figure 6.3: The execution structures of Guide.

Figure 6.3 shows two jobs started on separate machines. The first job has two activities, a1 and a2 executing on Host A. The activity a2 has diffused to a2' on Host B to access the object o4 used by the second job on that host. It also illustrates how communication between activities is done by sharing objects.

This method of communication is used both for activities in the same job, but also for activities in different jobs. The system supports mechanisms that allows implementation of different policies for object sharing (e.g. one copy or replicated objects, access synchronization, maintaining coherence if there are several objects). These policies can be programmed in generic classes, that are common to several applications, and a default policy has to be specified. The default policy for Guide language objects is to have a single copy.

The virtual machine provides mechanisms to control where objects are located. With these mechanisms different execution policies can be implemented (migration, load distribution). The location of objects are transparent by default.

6.2.3 Object Management

Apart from the generic models defined above, two properties of the Guide-2 system are of general interest, these are the clustering of objects (objects are co-located in clusters to amortize the cost of object management) and the use of dynamic binding (objects are not created or fetched from secondary storage before the first reference). These properties are described in the following.

Clusters of Objects

Experiences with Guide-1 [Freyssinet 91] showed that Guide objects are generally small, so using individual objects as the unit of distribution means that each object has to pay the cost of object management and distribution. Therefore a clustering scheme is used in Guide-2, where clusters of objects (collections of logically related objects belonging to the same owner) define the unit of distribution. In this way clusters may amortize the cost of object management and distribution, if most object references are local to the cluster.

Dynamic Binding of Objects

The first time an activity references an object (invokes a method in the object), the cluster containing the object is brought into the address space of the activity (this mechanism is called mapping a cluster into the context).

To ensure data consistency, clusters can only be mapped on one machine at the time, which means that not all clusters can be mapped locally, e.g., when two activities on separate nodes communicate through a shared object, the activity that reference the object first may map it locally, while the other has to diffuse to the node where the object is mapped.

This way the activity may be forced to change context several times during execution, i.e., distribution may be said to be data driven. However, it is important to note that the activity may remain on the same node as long as it references objects that are strictly private.

6.3 The Guide-2 Virtual Machine

The virtual machine provides the basic system support needed for the Guide-2 system. It is designed as a hierarchy of co-operating machines, each supporting different aspects of the object and execution models defined above. The structure of the virtual machine is shown in figure 6.4. This modular design has the advantage of allowing the system to evolve by substituting new components with the same interface, e.g. instead of the existing centralized integrated storage server, another distributed storage server is currently under implementation.

Each of the components shown in the figure will be described in sufficient detail to support the discussions of distribution in Guide-2 in 6.4, and the design of the load distribution facility in chapter 7. A more comprehensive description may be found in publications from Bull-IMAG [Krakowiak 92a, Krakowiak 92b, Chevalier 93] and internal working documents from the Guide-2 project [Cayuela 92, Lacourte 93].

6.3.1 The Object Machine

The object machine provides management facilities for objects classes and code libraries. It provides primitives that allow compilers to create classes and code libraries, as well as

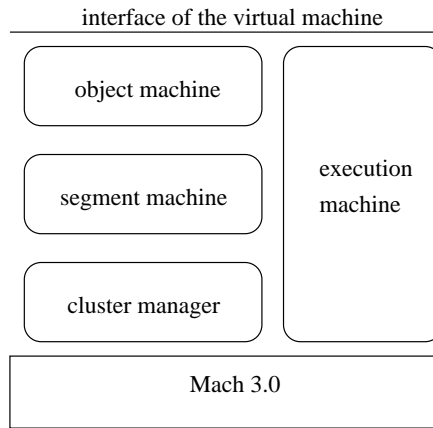


Figure 6.4: The structure of the Guide-2 virtual machine.

run time support for object creation and invocation.

Apart from creation it also manages the references to objects, their classes, and the code libraries. Objects, classes and code libraries are stored in segments of memory supported by the segment machine.

6.3.2 The Segment Machine

The segment machine emulates a segmented memory like Multics [Organick 72], where each segment is a unit of contiguous virtual memory that may vary in size. This type of memory directly supports an object based system that manages logical units of arbitrary sizes.

In the absence of real segmented memory, a segment is made available by mapping it into the address space of a context. The actual access (that may need the segment read from disk or from a remote host) relies on mechanisms provided by the kernel, i.e., an external pager in Mach 3.0. Inside the context, the mapped segment is represented by a virtual memory address. Objects are made available to contexts, by mapping the segment or segments that support the object.

6.3.3 The Cluster Manager

The cluster manager is composed of two layers: the upper layer maps the segments of the segment machine into clusters and manages sharing of these clusters between contexts, and the lower layer is in charge of the clusters stored on disk. The lower layer of the cluster manager is also known as the storage server, which is implemented by an external pager to Mach 3.0.

6.3.4 The Execution Machine

The execution machine implements the execution structures defined in 6.2.2. The primitives provided by the execution machine allows creation, activation and deletion of jobs and activities, and the creation of contexts.

It is also the execution machine that is responsible for remote method invocations.

Most of the functionality of the virtual machine is supported by libraries that are linked into the object based applications by the compilers. However, large parts of the execution

machine and the entire storage server are supported by separate servers, i.e., the Guide-2 daemon (gd) and the Mach 3.0 external pager (elixir) respectively.

6.3.5 Mach 3.0

Mach 3.0 is a micro kernel developed by the Open Software Foundation (OSF) and Carnegie Mellon University [Loepere 93a, Loepere 93b]. The micro kernel is present on all nodes in the system and provides the basic system services and communication primitives needed to build an operating system. This means that many of the system services normally associated with operating systems execute outside the kernel, e.g., paging is done by pagers executing in user space.

The Mach 3.0 execution model defines a local address space called a Task² where several Threads (threads of execution) may execute in parallel. The similarity between Mach Tasks and Guide-2 context is no coincidence; contexts are implemented by Mach Tasks, and activities are implemented as Threads executing in these Tasks.

All communication in Mach goes through ports that are endpoints of communication. Ports are uniquely identified in the system and the location of ports are transparent.

The NORMA version of Mach 3.0 from OSF used in the Guide-2 project, supports a distribution transparent name service used to look up system references, e.g., to look up ports.

Apart from the components shown in figure 6.4, a UNIX like operating system called OSF/1 runs on top of the Mach 3.0 micro kernel at the same time as Guide-2. OSF/1 is used to launch the Guide-2 system and to access external devices like disks and displays.

6.4 Distribution in Guide-2

Distribution in Guide-2 is transparent per default. However, the execution structures described above offer several possibilities to actively control distribution in Guide-2. We have identified three possibilities (granularities): jobs, activities, and clusters, which will be examined in the following.

This examination focus on the system support for active distribution, while the suitability of these granularities for load distribution is discussed in 7.3.

6.4.1 Jobs

It is important to understand the steps needed to create a job, in order to understand how distribution may be based on jobs.

As previously mentioned jobs are implemented by a collection of Mach Tasks, possibly executing on different hosts. Applications are in the current implementation of Guide-2, started from the OSF/1 system as an ordinary process, i.e., it is OSF/1 that created the Task that constitutes the first context of the job. This process transformed into a Guide-2 application when the execution machine is invoked to create and transfer control to the principal activity. This activity may now create new activities in the same context (i.e., new Threads in the same Mach Task) or new contexts with new activities on remote hosts.

Distribution based on jobs must happen before the first context is created, i.e., before the process is started from OSF/1. This type of distribution is similar to `rsh(1)` in UNIX, and `rsh` may in fact be used to perform initial distribution of jobs in Guide-2.

²Mach defines Tasks and Threads, so to distinguish the Mach usage of the words, we have chosen to capitalize Mach Tasks and Threads throughout this thesis.

6.4.2 Activities

Activities defines the treads of control in Guide, so they are the natural choice for explicit distribution. In the following we examine how the Guide-2 system support such explicit control of distribution in the form of activities.

Creation of an activity is a local operation directly supported by the execution machine with the `act_Create` primitive. This call accepts a call block describing what method in what object the activity shall start executing, and the parameters to that method invocation.

Activities may then extend themselves into other contexts using the `act_RemCall` primitive of the execution machine that accepts references to a context and an activity (the current local activity), and a call block as described above.

In order for an activity to diffuse to another host, an activity belonging to the same job must already exist on that host. If this is not the case a context will have to be created with the `ctxCreate` call that accepts references to the job, the node, and an identification of the user executing the application as parameters.

The current implementation does not directly support remote creation of activities, but the necessary functionality, to explicitly controlling distribution of activities, is already present with the `ctxCreate` and `act_RemCall` primitives. However, combining these primitives requires an existing activity that is left behind, so the functionality should rather be integrated with the `act_Create` call by expanding the parameters to include node and user identification.

6.4.3 Clusters

The lifetime of jobs and activities are limited by the application that created them, while clusters may contain persistent objects. The distribution of clusters should therefore not be determined at creation, but rather each time the cluster is mapped. We therefore examine the steps required to map a cluster to determine how this may be explicitly controlled.

Clusters are mapped when an object in the cluster is referenced for the first time by an object invocation through the object machine.

All objects are initially unbound, i.e., they share a common reference called a ghost handle. When a ghost handle is first referenced, dynamic binding of the objects takes place by descending through the hierarchy of the virtual machine shown in figure 6.4. The segment that holds the object is determined in the segment machine and the cluster that holds that segment is determined by the upper layer of the cluster manager.

The upper layer of the cluster manager requests the storage server (the lower layer) to map the cluster in the current context . Mapping is done if there is no protection violation and if the cluster is not currently mapped on another node (the storage server maintains a list of all clusters currently in use to ensure that clusters are only mapped on one node at the time). If the mapping cannot take place, an error is returned to the upper layer of the cluster manager and propagated up through the system where appropriate actions are taken. If the reason for the error is that the cluster is mapped on another node, the activity has to diffuse to that node in order to bind the object. Distribution in Guide-2 may therefore be said to be driven by data (objects).

Using clusters to control the distribution, means that object references should be resolved as far as the identification of the containing cluster.

6.5 Summary

This chapter starts with an overview of the Guide project, and a presentation of the overall goals of the project.

The Guide-2 system that constitutes the context of this work, is outlined and the object and execution models of Guide-2 are presented along with two aspects of object management in Guide-2: clustering, and dynamic binding of objects.

The virtual machine that implements the Guide-2 system is presented and each of the components that constitute the virtual machine (the object machine, the segment machine, the cluster manager, and the execution machine) are outlined. This system description is extended with a short overview of Mach 3.0, which define the Mach terminology used in this thesis.

The last part of this chapter is devoted to an examination of how explicit control of distribution is supported in the Guide-2 system. We examine the existing system support for distribution of jobs, activities, and clusters, which provides a basis for the design of the load distribution facility described in chapter 7.

Chapter 7

The Guide-2 Load Distribution Facility

This chapter describes the design goals and the design choices of the load distribution facility developed for Guide-2. The actual implementation is outlined in 8.

Other work has targeted load distribution in the Guide project, but this is the first implementation of load distribution in Guide-2. However, the previous work in Guide-1 may provide useful information for the design of the Guide-2 load distribution facility. These previous projects are therefore examined in some detail in 7.1.

We define the goals of the load distribution facility in 7.2, before we start the actual design.

Our discussion of object based load distribution in chapter 5 showed that the granularity of load distribution is one of the most important design issues. We discuss the granularity of the Guide-2 load distribution facility in 7.3.

In chapter 2 we defined four issues that had to be addressed when a load distribution facility is designed. These issues are whether load sharing or load balancing should be performed, whether load distribution should be dynamic or static, whether initial placement or migration should be performed, and whether source or server initiated load distribution should be performed. These issues along with other issues of special relevance to the Guide-2 implementation are discussed in 7.4.

We then discuss the two components of a load distribution facility. The information component is discussed in 7.5, and the control component is discussed in 7.6.

A summary of this chapter is found in 7.7.

7.1 Previous Work on Load Distribution in Guide

Load distribution has been the topic of two other projects in Guide. These projects are examined in the following to see what lessons have been learned.

The first project is a simulation of two different load distribution schemes and two different granularities (activities and objects) by Jean-Michel Lunati [Lunati 88]. The second project by Michel Christaller [Christaller 91] implements the basic mechanisms for explicit load distribution in Guide-1¹.

¹Guide-1 does not support clusters, instead individual objects are read from disk and stored in one large Virtual Object Memory (VOM). The diffusion mechanism used by Lunati and Christaller is therefore based on individual objects.

7.1.1 Simulations of Load Distribution in Guide-1

This first work on load distribution in Guide was done before the first prototype of Guide-1 was ready. Based on simulations performed on a queueing network model, Lunati predicts that general performance gains by load distribution in Guide should be possible.

We have chosen to describe the parameters of the network queueing model in some detail, so that possible problems may be identified.

Context

Lunati uses the QNAP2 modeling software² and the host model illustrated in figure 7.1 to model the combination of two different load distribution schemes and two different granularities of load distribution in Guide.

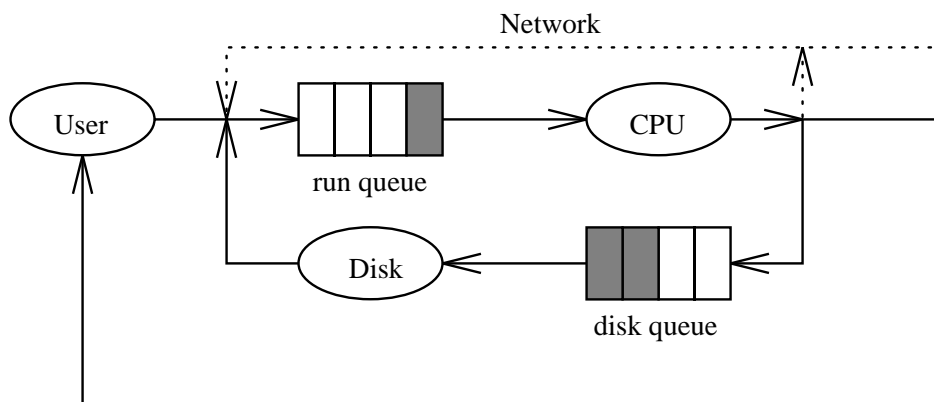


Figure 7.1: The host model used by Lunati.

Only one type of job is modeled in the simulation. This job has a principal activity that starts five more activities to execute the application. Each activity is a separate thread that does not communicate with the other threads. It goes through a sequence of five method invocations on previously unbound objects.

When an activity is created it is added to the run queue. It then references an object that initially has a 50% chance of being available in memory, otherwise it has to be fetched from disk. If the object is stored on disk, the activity waits for the disk access to finish and is put back into the run-queue. If the object is already in main memory, the activity executes for the allotted amount of time and is put back into the run-queue. This cycle is repeated for each of the five objects that the activity references.

The network queueing model is based on a $M/M/1$ network queueing model, which means that job arrival and job execution times are exponentially distributed and only one resource is considered.

The exponential distribution is defined by the mean, so the means of the arrival rate and the execution times are parameters in the experiment. Two different means for the arrival rate, 10 seconds and 30 seconds, are used in the simulations, so that different workloads may be simulated. The mean execution time is arbitrarily fixed at 200 ms. (different studies indicates that 60%–65% of all applications run for less than 500 ms.). The rest of the parameters of the simulations are listed below, so the simulated system may be compared to the current environment:

²QNAP2 (Queueing Network Analysis Package, version 2) was developed as a joint venture project between CII Honeywell Bull and INRIA in 1982

Sending a request on the network: 1 ms.

CPU overhead to send the request: 10 ms.

Transferring an object on the network: 5 ms.

CPU overhead to transfer an object: 50 ms.

Total network time to transfer object and invoke method: 10 ms.

Total CPU overhead to transfer and invoke an object: 50 ms.

Time to access an object on local disk: 20 ms.

CPU overhead to manage the disk access: 20 ms.

CPU overhead to transfer load information: 10 ms.

The network transfer time is based on a 10 Mbit Ethernet, while the disk and CPU requirements models a SUN3 workstation. These parameters are very similar to the environment used by Guide-2.

Load Distribution Schemes

Lunati performs an analysis of the two most promising load distribution schemes proposed by Zhou [Zhou 88], CENTRAL and GLOBAL (these load distribution schemes are described in 2.1.1), with distribution of activities and objects.

GLOBAL The load information is periodically sent to a central load information center (LIC) that creates a vector describing the load on all the hosts in the system and redistributes this load vector. An experiment that varied the load update period from 1–5 seconds, showed that the shorter periods performed the best.

Each host performs load distribution by examining the local load vector and selecting the host with the lowest load, if this load is lower than the local load. A random host is chosen when several hosts exist in this category.

We use the following notation GLOBAL< *activity* > and GLOBAL< *object* >, to designate the GLOBAL algorithm based on activities and objects respectively.

CENTRAL The load information is periodically sent to a central load information center (LIC) that stores it locally.

Load distribution is performed by sending a request (with the actual load of the node) to the LIC, that selects a node based on the centralized information.

We use a notation similar to the one above to designate CENTRAL< *activity* > and CENTRAL< *object* >.

Conclusions

The analysis of Guide-1 presented by Lunati concludes that diffusion provides a limited form for pre-emptive activity migration at no additional cost.

The simulations show that performance benefits may be achieved by distribution of objects, and the comparison of the four load distribution schemes shows that despite the cost of communication, GLOBAL< *object* > is the best algorithm.

His results are interesting because they indicate that performance benefits are possible in Guide-1, the performance benefits may be achieved without object migration, and that the finest granularity of load distribution offers the best performance benefits.

7.1.2 Explicit Load Distribution Facility in Guide-1

The implementation of load distribution in Guide-1 by Christaller, defines an extension to the Guide language to perform explicit load distribution. This extension is composed of an addition to method invocations that allows a remote node for the invocation and a dynamic location mechanism that appears as a system service (i.e., the location mechanism is referenced as an ordinary object, which is always available on the machine).

Information Component

The information component uses the load information available from `rwhod(8c)` updated approximately every two minutes. The information in `/usr/spool/rwho/whod.*` is consulted periodically and a local cache of information is maintained by the load distribution facility. Each time an object is placed on a remote node, the cached information for this node is also updated. This means that frequently interacting machines have fairly up to date information, while the information for the rest of the machines may be two minutes old.

Control Component

The control component supports the extensions to the Guide language, that allows explicit distribution of method invocations.

Initiation Policy & Selection Policy Since load distribution is explicit, both policies are controlled by the application programmer. However, Christaller acknowledges the importance of filtering objects by the selection policy, so that objects that are unlikely to benefit from load distribution are executed locally.

Location Policy The local load is examined before the search for a target node is started, and the local node is preferred if the load is low. If the local load is high or rising, the load information is searched for the least loaded node.

If the information from this node has not been updated since the last periodic update, the node is contacted and the load information is updated. Unless the load has increased dramatically, this node is then selected as the target for load distribution, otherwise the search for an underloaded node is repeated.

If no underloaded nodes exist in the system, the local node is selected.

Transfer Policy The extension to the object invocation, allows the programmer to select a remote node for the method in the following way:

```
object.method(parameters)@node;
```

Where *node* is a character string that uniquely identifies the node, e.g., the host name. This allows static load distribution, but dynamic load distribution is possible by invoking the load distribution facility through the object interface.

The implementation does not allow direct invocation of the load distribution facility in the place of *node*, so it must be invoked in the following way:

```
node := LoadDistributionFacility.LocateNode;  
object.method(parameters)@node;
```

Unfortunately this syntax does not allow distribution within the `co_begin ... co_end` structure, because the Guide language does only allow method invocations and not blocks of instructions between the `co_begin` and the `co_end`. This means that the construction may not be used to exploit parallelism in the Guide language.

Conclusions

A partial load distribution facility was implemented, so that the difficulty of implementing such a facility could be estimated. The implementation shows that system support for explicit load distribution in Guide-1 is easily integrated into the system.

Several applications are implemented and their algorithms have been parallelized in order to benefit from load distribution. However, no performance measurements are presented in the report.

7.1.3 Lessons Learned

The simulations by Lunati use a system model that is fairly close to Guide-2, which suggest that performance benefits should also be possible in Guide-2. Furthermore, they indicate that the finest possible granularity offers the best performance benefits.

We expect most of the system parameters of the Guide-2 system to be similar to the parameters used in the simulations by Lunati, except for the cost of remotely mapping an object, which we expect to take longer in Guide-2 (we are mapping an entire cluster). The definition of the workload parameters is largely arbitrary, and we believe that these parameters may be rather different, so the results are not directly applicable to our load distribution facility. However, the results of these simulations may give a first indication that should later be verified.

The simulations by Lunati have also shown that the shortest possible load update period should be used.

The implementation by Christaller is not of direct interest to our work, because it implements explicit load distribution. However, it is interesting to note that both the previous projects emphasize the support for multi threaded applications. This suggest that we should expect multi threaded applications, and that the load distribution facility should provide support for this type of applications.

7.2 Design Goals

One of the main goals of this thesis is to investigate the possible performance benefits of load distribution in an object-oriented operating system. The primary design goal of the load distribution facility is to support this investigation, which means that other design goals may be sacrificed in order to support this goal.

Apart from this overall goal, three sets of goals for the load distribution facility needs to be defined. These are the performance goals, the efficiency goals and the implementation goals described below.

The two first sets of goals define what kind of performance gains should be expected from the load distribution facility and the amount of resources that may be consumed in the pursuit of these goals. These two sets are described further in 7.2.1 and 7.2.2 below. The third set of goals, described in 7.2.3, defines all other requirements that the load distribution facility should fulfill.

7.2.1 Performance Goals

The performance goals must define, which of the performance metrics, mentioned in 2.2.1, should be optimized. These metrics are: system throughput, system utilization, and system response time (defined by the average run-time of applications). We choose to focus on the response time, because it is the parameter that is most visible to the users of the system.

It is unlikely that all applications will benefit from load distribution, so the balance between performance gains and possible performance losses has to be defined. The goals stated below define how the load distribution facility should weigh performance gains against possible losses.

First of all, the load distribution facility should at least be neutral to the majority of applications. This means that big performance gains by a few applications at the cost of many others are unacceptable.

Our second goal is to limit the degradation in performance for those applications that do not benefit from load distribution.

The last requirement is that load distribution should result in an overall improved performance, i.e., the performance gains should out-weigh both the resources consumed by the load distribution facility and the performance losses by the applications not benefiting from load distribution.

Performance Benefits The performance of the majority of applications must not degrade.

Low Degradation Those applications that suffer, must not suffer heavily.

Overall Improvement The overall result of load distribution should be an improvement in performance.

These ambitions are very modest, but until more experience with fine-grained object-oriented load distribution is gained, it is impossible to say what kind of performance gains should be expected.

7.2.2 Efficiency Goals

In our examination of the impact of fine-grained objects on load distribution in chapter 5, we found that the overhead of the load distribution facility is an important performance parameter because of the increase in scale of the load distribution problem and the smaller absolute speed-ups expected from fine-grained units of distribution. Furthermore, the Guide-2 load distribution facility has been introduced into an already existing system, which means that the load distribution facility should not introduce a large overhead in any part of the system, thus possibly invalidating otherwise valid assumptions about costs of services.

Apart from the overhead of the load distribution facility, another aspect of efficiency has to be considered, namely the amount of system resources consumed by the load distribution facility. This amount of resources should always be limited, but the consumption

of system resources should be weighted against the performance improvements by load distribution, thus higher costs may be justified by higher performance gains. The only ways of determining the optimal amount of resources to allocate for load distribution are through simulations or experiments with the system. Neither are available to us at this stage, so we simply choose to minimize the amount of resources consumed by the load distribution facility. This may not be optimal, but it leaves most resources available for applications.

These goals for the efficiency of the load distribution facility are summarized below.

System Overhead The limitation of the overhead is of prime concern in the design of the load distribution facility. The overhead imposed by the load distribution facility should be negligible.

Resource Consumption The ratio of the performance benefits to the amount of resources consumed by load distribution, should be as high as possible. It is impossible to estimate this ratio without simulations or experiments, so we simply choose to minimize the divisor in the equation.

7.2.3 Implementation Goals

The execution environment of Guide-2 and the general design of the system, impose some limitations on the design of the load distribution facility. These limitations together with general design goals unrelated to the performance and efficiency of the load distribution facility constitute the implementation goals.

Like other Guide-2 system services (e.g., Elmer³ and Elisa⁴), the load distribution facility should be developed as an independent entity. Since load distribution is not vital to the system it should be able to run without load distribution enabled, and to survive the crash of the load distribution facility. Load distribution takes place when the load distribution facility is running, otherwise applications execute without it.

Making the load distribution facility an independent entity has other advantages. First of all it makes it easy to replace by new enhanced facilities, as long as the interface remain the same. It is also an advantage in the implementation phase, where the load distribution facility is easier to debug, when no real changes are made to the already running system.

Only a few implementations of load distribution in object-oriented systems have been reported in the literature, and the impact of object-orientation on the existing theory of load distribution is not fully documented. This mean that the load distribution facility should be configurable and easily extendible, in order to support experiments with different policies and to accommodate new load distribution schemes that may emerge in the field.

One important purpose for building distributed operating systems is fault tolerance, so the load distribution facility should also be able to survive the failure of any machine in the system. This is not a main concern for the Guide-2 load distribution facility, but algorithms should be as robust as possible.

The Implementation goals of the Guide-2 load distribution facility is summarized below.

Transparency Load distribution should be transparent to users and the application programmers, i.e., no special actions should be taken when applications are launched,

³Elmer is the Guide-2 execution visualization tool, that makes it possible to follow the execution of applications in the distributed environment.

⁴Elisa is the persistent storage administration tool, used to configure volumes and define which cluster manager to use.

nor should statements in the code of the application be required for the application to benefit from load distribution.

Extendibility The load distribution facility should be modular and easy to extend, in order to accommodate the emerging technologies in the new field of object-oriented load distribution.

Configurability The behaviour of the load distribution facility should be easy to change by configuration. This allows experiments with different load distribution policies, but also modifications to support different application environments.

No kernel modifications The implementation of the load distribution facility should require no modifications to the underlying micro kernel, and only require few *additions* to the Guide-2 system.

No application modifications Existing applications should not be redesigned to benefit from load distribution. We do accept that applications should be recompiled, and that applications written with load distribution in mind benefit more from load distribution than other applications.

Along with the goals stated in 7.2.1 and 7.2.2 these goals form the design goals for the Guide-2 load distribution facility.

7.3 Granularity of Load Distribution

The single most important choice when designing the load distribution facility, is choosing the granularity of load distribution. In 6.4 we identified three possible granularities for actively controlling distribution in Guide-2: jobs, activities, and clusters.

Any of these granularities may be used in our implementation and indeed the jobs and activities of the execution model seem obvious candidates because they define the threads of execution in Guide-2. However, both jobs and activities are themselves distributed activities and may eventually diffuse to other nodes.

Diffusion is determined by the location of objects, i.e., where the cluster containing the object is mapped, so load distribution may also be done by making sure that object invocations are remote, i.e., by mapping the containing cluster on a lightly loaded node.

A rudimentary object migration scheme has been implemented that allows migration of objects between clusters, so this possibility should also be considered when the granularity of load distribution is decided. These four granularities are examined in greater detail in the following.

7.3.1 Distribution of Jobs

Jobs are distributed address spaces and are as such passive entities. The active entities in Guide are the activities, but when a default policy for creation of activities is added to the job, the initial placement of the job becomes important. In the current implementation activities are started in the context of its creator, meaning that the principal activity will be created in the first context of the job, and that all subsequent activities will be created in this context unless the job diffuses onto other hosts.

Load distribution of jobs resembles load distribution in traditional process based operating systems, allowing only very coarse grained load distribution. This means that the possible advantages of fine grained object-oriented systems are not explored.

In the case where the job and the application are the same, load distribution can be implemented by any existing load distribution facility for UNIX, e.g., the Utopia system mentioned earlier.

7.3.2 Distribution of Activities

Load distribution based on activities has a finer granularity than facilities based on jobs, and activities are generally used to define the parallelism expressed in the application.

The Guide activities are again distributed entities invoking methods in objects that may reside locally or on remote hosts. Placement of activities becomes important, when a policy for making the objects available to activities is defined. This policy determines when objects are called locally and when diffusion is needed. Guide uses dynamic binding that maps the invoked object into the current context of the activity, when a method in the object is first invoked (if the object is already bound in another context, the activity will have to diffuse into that context). Without load distribution this means that activities remain local as long as only objects in clusters that are not previously mapped on another node (local objects), are referenced.

The principal activity is created in the job, when the application is initialized. New activities may be created by demand, e.g. by using the `co_begin . . . co_end` structure of the Guide language. The distributed nature of activities means that they may diffuse onto several hosts in the system, thereby dividing the load proportionally between these hosts. It is important that the initial object of each new activity reside in a separate cluster for the distribution of activities to work, otherwise they would immediately diffuse to the host holding the initial object, upsetting the work of the load distribution facility.

Load distribution at the activity level is especially useful in parallel programming, where the programmer expresses parallelism explicitly and the system then uses different activities to support this parallelism in the most efficient way. When no parallelism exists in the application, distribution of activities degenerate into load distribution of jobs.

7.3.3 Distribution of Clusters

The granularity of load distribution gets even finer at the cluster level. When objects are bound the entire cluster gets mapped into the context of the calling activity. Clusters can only be mapped by one host at a time, so if the cluster is already mapped by another host, the activity will have to diffuse into a context on that host.

Instead of mapping all clusters locally, some clusters could be mapped on an under-loaded host, thereby forcing the activity to diffuse to this host. This is in effect a hybrid between pre-emptive and non-pre-emptive load distribution, because only the part of the activity which currently utilizes the CPU migrates to the target host, and only while objects in that cluster are invoked. Furthermore, all subsequent invocations of methods in objects residing in that cluster will be redirected to the target host.

There are two different scenarios for the arrival of a mapping request. In the first scenario a request to map a new cluster comes from an activity that has run for some time, and in the second the mapping request comes from a newly created activity.

If the activity has run for some time it will appear in the local load information and the extra cluster is unlikely to increase the load on the host (objects may start new activities, but the mapping of the first cluster by the new activity brings us into the second scenario). If the current load is high the diffusion of the activity will offload the host for the duration of the method invocation, but if the local load is low the added overhead of remote invocation is likely to degenerate performance.

In the second scenario the interval from an activity is created to the initial cluster is mapped is so short that the activity will not appear in the local load information. If the mapping request comes from a newly started activity it is likely to increase the load of the host, and therefore load distribution should be considered. If several new activities are created simultaneously, each of their initial objects should reside in different clusters, thus creating many mapping requests. In this case some of the clusters should be mapped on remote hosts, in order to keep the current host from being overloaded.

The only way to safely distinguish between the two scenarios above is to keep track of all activities on the host, thus creating a heavy weight mechanism.

For performance reasons objects that interact should reside in the same cluster. This means that mapping of clusters by an activity will be relatively scarce, except for the case when many activities are created. A cheap way of detecting this scenario with high probability would be to count the number of clusters mapped locally within some interval of time. If this number is high, e.g., it exceeds some pre-determined threshold, it is probably because several activities have been created and then some of these clusters should be mapped on remote hosts.

7.3.4 Distribution of Objects

Load Distribution based on distribution of objects implies migration of objects between clusters. A rudimentary support for this type of object migration exists in the current implementation, however it relies on a forwarding scheme of references that means all the clusters an object visits have to be mapped to resolve references. This makes the cost of the object migration scheme prohibitive, so we do not consider it any further.

7.3.5 Granularity of Load Distribution in Guide-2

The units defined in the execution model are themselves distributed entities. This means that they are ill suited for non-pre-emptive load distribution, but the default policies for creating activities within jobs and mapping clusters in the current context of the activity somewhat remedy this limitation.

The storage structures defined in the object model are stationary entities. The placement of objects determines where the execution of their methods will take place. This makes these structures more promising vehicles for load distribution in the Guide-2 system.

The cluster level for load distribution has been chosen, partly because of the reasons stated above, but more important because it fits with the overall goal of experimenting with the finer granularity offered by object-oriented systems (the job level degenerates to the application level, which is examined in Utopia [Zhou 91] and the activity level is mostly interesting for applications written with parallel execution in mind). Furthermore the simulations done by Lunati indicates that the finest possible granularity should be chosen, so it is possible that fine-grained load distribution may improve the performance of the system. Another advantage of cluster level load distribution is that it can be implemented with limited modifications to the existing Elliott kernel, because the mechanism for diffusion when clusters are mapped remotely already exists.

By distributing clusters or objects, activities are forced to diffuse onto more hosts thus creating more contexts, and thereby fragmenting the job. If jobs become too fragmented performance will suffer from the higher rate of remote invocations and context switches. The difference between this fragmentation and the fragmentation introduced when distributed applications communicate through shared objects, is that no functional

justification for this fragmentation exists, but hopefully it will be justified by improvements in performance.

7.4 General Architecture

In 2.1 we identified four important issues that the implementation of a load distribution facility has to address: load sharing vs. load balancing, static vs. dynamic load distribution, initial placement vs. pre-emptive load distribution, and source vs. server initiative.

We have chosen to implement a load distribution facility that performs load balancing, is dynamic, uses initial placement, and is based on a source initiative scheme. We give a short justification for these choices in the following, where we also discuss two previously unmentioned issues: autonomy of the load distribution facility, and the level of integration into the system that should be required of the load distribution facility.

7.4.1 Load Balancing

Load sharing may be viewed as a special case of load balancing, where only idle hosts are considered as possible targets. The success of a load sharing scheme therefore depends on the availability of idle hosts in the system.

Guide-2 is primarily aimed at small to medium scale distributed systems, where few idle hosts are expected, so we choose to implement a load balancing scheme that does not restrict us to idle hosts, but also allows work to be offloaded to under utilized machines.

7.4.2 Dynamic

The resource requirements of applications are not known before they are executed, and the workload of a single host is not easily modeled without the knowledge of the applications running on the host.

It is therefore unlikely that a static load distribution scheme should out perform a dynamic scheme, which is why the approach has been chosen.

7.4.3 Initial Placement

A non-pre-emptive load distribution scheme has been chosen, because it can be implemented on top of the existing diffusion mechanism. This makes the implementation of the load distribution facility much simpler.

7.4.4 Source Initiative

The choice of an initial placement algorithm suggests the use of source initiative, but a server initiative scheme like RESERVE described in 2.1.1 may also be used. However, this scheme becomes unnecessarily complex because reservations must be cancelled when work arrives at the previously underloaded node, or the node has to be polled to determine whether it is still underloaded. Source initiative algorithms are conceptually simpler and is therefore chosen.

7.4.5 Autonomy

The Guide-2 load distribution facility is distributed, i.e. it is having a local agent running on each host. These local agents will be responsible for executing the load distribution

policies, as well as maintaining their part of the global system state, i.e. updating the local load information.

The choice of a distributed architecture is partly motivated by the conclusions by Lunati that GLOBAL< *object* > is the best, partly by the concern for fault tolerance.

7.4.6 System Integration

The load distribution should be an external component in the system for several reasons. First of all because it makes implementation much easier, when interactions with other system components are limited. It makes it possible for machines not to participate in load distribution, e.g., if they are running critical applications. Furthermore, it makes modification of the load distribution facility much easier.

Another advantage of the load distribution as an external entity is that it may be used with different granularities without modifications, which means that activity migration may be implemented by the same load distribution facility by changing its location in the system. This requires some work and is beyond the scope of this thesis.

The defined architecture for the Guide-2 load distribution facility should be dynamic, doing load balancing, using a source initiative initial placement algorithm, autonomous, and implemented as a separate component in the Guide-2 system.

7.5 Information Component

The load information gathered is not necessarily the information exchanged. It is only necessary to update the information, when significant changes in the current load occurs. This saves network communication when the load remains fairly constant.

7.5.1 Information Gathering Policy

In 2.3.3 the different load indicators were discussed and a single CPU related load indicator was preferred based on research by Kunz and by Ferrari & Zhou.

Kunz examines several CPU related load indicators, e.g., the current number of runnable threads and the average number of threads runnable within the last minute (i.e., the UNIX load number). Kunz concludes that the first choice was far superior, probably caused by the short run-time of most processes. The environment studied by Kunz was a network of diskless workstations with a file-server, where processes are “transferred” by sending the program name to the remote host, which executes the command. This situation is very similar to Guide-2 where clusters are mapped by the storage server on the host that requires the cluster. The cluster can then be accessed by activities on that host. This means that the results of Kunz should be directly applicable to the Guide-2 environment.

We suspect that a load indicator combining information about CPU and memory may prove to be better than the single source indicator, but is not considered for two reasons. First of all because it complicates the design of the location policy by making comparison of nodes more complicated, and second, but more important, because it is unrelated to fine-grained load distribution, and therefore beyond the scope of this thesis. However, it is an obvious extension to the system, so the design should ease the implementation of this extension.

7.5.2 Information Exchange Policy

Global system state is kept by distributing the load information from all local agents to the load distribution facilities running on the different hosts.

The quality of the load information depends on the frequency of information exchange. A high exchange frequency provides more accurate information but also requires more network communications. The frequency of exchange of load information should either be configurable, which allows experiments with different update frequencies, or the system should be made adaptive so that the load information is only updated when a significant change has occurred.

We choose to focus on the load distribution scheme defined by GLOBAL, where the period p is supplied at run-time, but in fact both approaches are implemented, and the behavior of the system is determined when it is configured.

7.6 Control Component

We have previously identified the five policies that the control component should implement. In the following we define these policies to conform with the design goals stated in 7.2 and the general architecture defined in 7.4.

7.6.1 Initiation Policy

The decision to implement an initial placement scheme means that the initiation policy is triggered by the arrival of a request to map a cluster.

When the initiation policy is triggered the local load is examined, and load distribution is initiated if the load exceeds a predefined threshold.

7.6.2 Selection Policy

The choice to implement a non-pre-emptive load distribution scheme, reduces the role of the selection policy. The selection is already made because only the cluster which is requested for mapping is considered. This means that the selection policy should only decide whether the cluster is suited for remote mapping.

Discrimination of short termed processes and processes demanding local hardware was discussed in 2.6. In Guide-2 such discrimination should target clusters with short lived objects or objects that communicate a lot with other clusters currently mapped locally.

Different applications may access the objects in a cluster in different manners, which means that the variation in execution time of objects in the same clusters may be significant. It is therefore important to re-examine the average run-time proposed for History, to determine whether it is an adequate metric for the weight of clusters, as it were with processes.

Under all circumstances, discrimination may improve performance, so although it is not considered for the current implementation, the design should make the addition of such functionality possible.

7.6.3 Location Policy

The location policy uses a double threshold policy as illustrated by figure 2.1, to identify the best possible target for load distribution. This means that hosts are divided into three classes: underloaded, normally loaded, and overloaded hosts. If the load interval allowed

for underloaded hosts is small (i.e., the `lowWaterMark` is chosen close to zero) load sharing will be performed, otherwise load balancing is performed.

The policy described in 2.7 only allows work to be offloaded from overloaded hosts to underloaded hosts, to avoid thrashing. However, thrashing is not a problem with initial placement policies, so we also allow work to be offloaded from normally loaded hosts to underloaded hosts. This allows us to experiment with load distribution without seriously overloading the hosts (this is needed in our investigation of load distribution with an average workload presented in 9.5.1). We maintain both thresholds because we expect the benefits of fine-grained load distribution to be small, so our design should be easy to change so that load distribution is only considered when the local load is high.

Work is offloaded to underloaded hosts, and when more underloaded hosts exist a choice has to be made. Because the load information is likely to be slightly out of date, all underloaded hosts are considered equal. The target is therefore chosen by a simple round-robin mechanism, preventing that the same host is chosen several times in a row. This method should avoid that hosts become flooded in most situations.

Initializing an application often references many objects in a short time (this is known as the *hot spot* problem [Dickman 91]), possibly provoking several mapping requests. If this happens on an overloaded host these clusters will be mapped on remote hosts. If only a few underloaded hosts exist the round-robin is not sufficient to prevent flooding so the number of clusters that may be mapped on each underloaded host should be limited. When this limit has been exceeded for all the underloaded hosts, the round-robin should also include the normally loaded hosts.

7.6.4 Transfer Policy

The transfer policy takes advantage of the existing mechanisms for diffusion when clusters are already mapped on remote hosts.

The load distribution facility is introduced as an extra layer in the cluster manager placed between the upper and the lower layers. When the required cluster has been identified by the upper layer, the load distribution facility is invoked to select a host for mapping. This host is used as a parameter in the mapping request to the storage server.

If the mapping is successful, i.e., the cluster may be mapped on that host, a fake error message is propagated up through the layers of the virtual machine saying that the cluster is unavailable, because it is already mapped by the same user on a remote host. The transfer may then be treated as an ordinary remote object invocation, using the existing diffusion mechanism.

This transfer policy is quite heavy because the storage server is called on both nodes, but it has two advantages: it is easy to implement because it relies on existing mechanisms for most of the thread management, and it requires little integration into the system.

7.7 Summary

Two previous studies in load distribution were examined, and simulations based on a network queueing model of the Guide-1 system suggested that a decentralized load distribution scheme (GLOBAL) will perform better than a centralized scheme (CENTRAL). Furthermore, the simulations indicated that fine-grained load distribution based on objects, should be preferred over the coarser granularity of activities.

We decided to use the response time, measured by the average run-time of applications, as the performance metric that we wish to optimize with our implementation of

a load distribution facility. The invocation overhead and the resources consumed by the load distribution facility must be minimized, and we presented a list of miscellaneous implementation goals, including: transparency, extendibility, configurability, and limitation of modifications to the system and applications.

Having stated these overall goals for the implementation the following granularities were examined with reference to load distribution: jobs, activities, clusters, and objects. We chose the cluster as our unit of load distribution, as it offers the greatest opportunities to experiment with fine-grained load distribution.

We presented an overall architecture for the load distribution facility, based on the load balancing scheme GLOBAL presented by Zhou [Zhou 88], and we described the information component and the control component through the six policies that constitute the load distribution facility. In the following we present a short summary of these policies.

Information Gathering Policy We use a single CPU related load indicator.

Information Exchange Policy The load indicator is periodically send to a central load information center, that redistributes a vector of the collected load indicators to all hosts.

Initiation Policy Load distribution is initiated when a previously unmapped cluster is referenced for the first time.

Selection Policy We do not consider discrimination of clusters, i.e., all clusters are eligible for distribution.

Location Policy We define a location policy based on two thresholds that divide the hosts into underloaded, normally loaded, and overloaded hosts. Underloaded hosts are preferred, but normally loaded hosts may be used if no underloaded hosts exist. If several hosts exist in a given category, a round robin mechanism is used to avoid that a single node is overloaded.

Transfer Policy The transfer policy should rely on the existing diffusion policy for transfer of control.

The implementation of the load distribution facility based on these policies is described in the next chapter.

Chapter 8

Implementation

This chapter contains all the implementation specific details of the Guide-2 load distribution facility called “elvis” (elvis is an acronym for the Elliott Load Visualization and Information Service). We do not give a full presentation of the implementation, but aim to describe it in sufficient detail, so that a proficient programmer may understand, maintain, and further develop the load distribution facility with a limited effort. The presentation should therefore be regarded as a navigation instrument and an aid while reading the actual code.

We start by giving an overview of the architecture of the load distribution facility in 8.1. This is followed by presentations of the implementations of the client library in 8.2, the daemon in 8.3, and the necessary modifications to the Guide-2 system to facilitate load distribution in 8.4. A short summary of this chapter is presented in 8.5.

8.1 Elvis Architecture

The Load distribution facility may be integrated at several places in the virtual machine, but we have chosen to implement it as an extra layer between the upper and the lower layers of the cluster manager.

This location is motivated by the choice of clusters as the unit of load distribution, which makes it natural to wait until the mapping of a new cluster is required, and our wish to implement an autonomous load distribution based on GLOBAL, which means that the load distribution facility may not be invoked from the currently centralized storage server.

The upper layer of the cluster manager is implemented by a library that is linked into every application, like most of the object support functions of the virtual machine. We implement the load distribution facility by an addition to this library that invokes the load distribution facility as an external component, which is in accordance with the design goals stated in 7.4.5.

The added functionality should initialize the application part of the load distribution facility, e.g., look up the reference for the external component in the name server, and invoke the external component of the load distribution facility. This functionality is implemented by a library with a single function that should be known to the cluster manager.

This means that the load distribution layer in the virtual machine consist of two parts: a client library that is linked into every application, and an external elvis server, that implements the policies of load distribution defined in chapter 7. This architecture is illustrated by figure 8.1, which shows the steps required to map a cluster when load distribution is enabled and the default mapping scheme without load distribution.

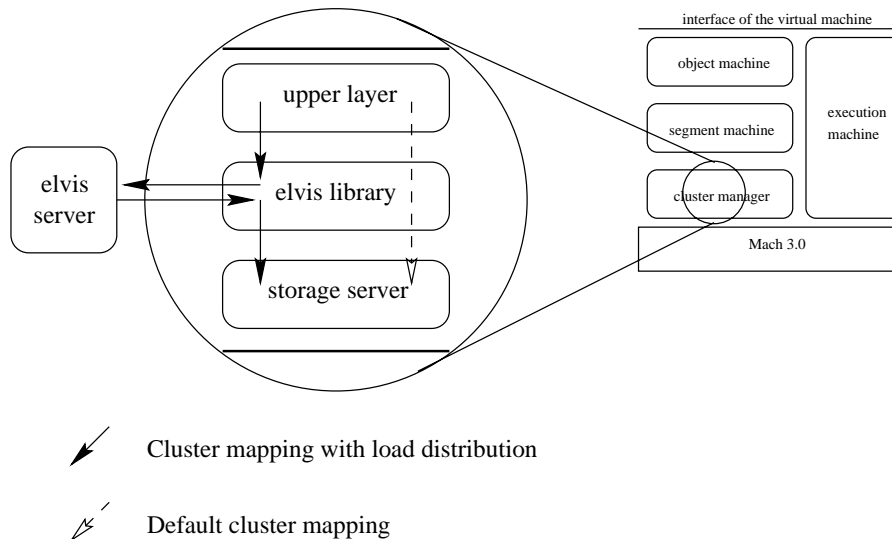


Figure 8.1: Cluster mapping with load distribution.

The external server is implemented as a multi threaded Task in Mach 3.0, with the following three threads: the update server that gathers load information, the display server that shows the load information in a X11 window, and the message server that handles communication with the elvis library and the other elvis servers in the system.

The extra layer of the load distribution facility adds functionality to the cluster manager, when load distribution is enabled, but is otherwise invisible to the application.

8.2 Elvislib

The client library contains code to initialize the load distribution facility used by the library itself and the elvis server. It also contains code that enable invocation of the elvis server through IPC. We give an outline of these functions in the following, along with a short description of how the load distribution facility has been integrated into the virtual machine described in chapter 6.

8.2.1 Initialization Code

The library has two functions that are used to initialize the load distribution facility: `elvis_userConfig(OUT port)`, and `elvis_getConfig()`. These functions are outlined in the following. Instead of listing the type of each parameter we use the terms IN, OUT, and INOUT to denote input parameters, output parameters and parameters that are used for both input and output. This terminology is used throughout this chapter.

`elvis_userConfig(OUT port)` This function is used to initialize the client applications. It calls two other functions: `adm_getConfig()` that configures global system parameters, and `elvis_getConfig()` that configures global elvis parameters. The function returns a Mach port that should be used to invoke the elvis server. If a centralized load distribution scheme is used the port of the central server is returned, otherwise the port of the local server is returned.

`elvis_getConfig()` This function reads the configuration file and initializes global variables according to the configuration parameters in this file. A description of the current configuration parameters and their possible values are found in appendix A.

8.2.2 Invocation Code

The elvis server is invoked by a single IPC call, which is described in the following.

`elvis_locateNode(OUT node)` This function asks the elvis server for a target node. If this call returns an error (i.e., no elvis server daemon has been started) it prevents further invocation of the load distribution facility and returns the local node, otherwise the target node from the elvis server is returned.

The function `elvis_locateNode(OUT node)` is the only function that needs to be called by the virtual machine. It initializes the load distribution facility the first time it is called.

8.2.3 Integration into the Cluster Manager

The load distribution facility is invoked from the function that maps a cluster into the current context (`clu_Map(...)`). First the load distribution facility is invoked to locate the best node for mapping the cluster. If this node is identical to the current node the unmodified code that maps a cluster locally is executed, otherwise the storage server is probed with the probe-call ("`elixir_Probe(...)`" that has been added to the storage server for the purpose of load distribution), to determine whether remote mapping is possible. If this is the case diffusion is initiated as described in 8.3.2, otherwise the cluster is mapped on the node returned by the storage server, e.g., the local node.

The library implementing the functionality described above is called "libelvislib.a" in accordance with the Guide-2 naming scheme.

8.3 Elvis

The elvis server daemon ("elvis") is started on all nodes that participate in load distribution. This means that no work is off loaded to and from nodes, where elvis is not started. The mechanisms implementing the different policies are described in the following, and the architecture of the elvis server is outlined.

8.3.1 Information Component

In the following we outline the mechanisms that implement the policies of the information component defined in 7.5 based on the GLOBAL load distribution scheme. This definition was rather detailed, so we limit the following description of the mechanisms to an outline of the functions used to implement the component.

The support for global state has been extended with support for centralized stated as defined in CENTRAL, and the ability to avoid keeping state as defined in RANDOM. This support was primarily integrated because it was easy to implement and allows larger flexibility. However, this support for alternative system state management is insignificant to this project, and is only mentioned here for completeness, so we will therefore not mention it any further in this chapter.

We have also experimented with an algorithm that prevents the update of load indicators, unless the load has changed significantly. This algorithm has not been tested thoroughly and is therefore only outlined in this chapter.

The information gathering mechanism is described in 8.3.1, the information exchange mechanism is described in 8.3.1, and the update restriction algorithm is outlined in 8.3.1.

Information Gathering Mechanism

In 7.5.1 we decide to use a single CPU related load indicator, e.g., the number of runnable threads on a node.

Information about the number of runnable threads is provided by the Mach 3.0 system call `host_info` that returns the average number of runnable threads during the last 5, 20 and 60 seconds. The number returned by the system call is multiplied by 1000, compared to the traditional UNIX load.

We choose the 5 second average, because it is the one closest to the actual number of running threads. The actual number of runnable threads could be calculated from this average in the following way: the algorithm that calculates the average is run every second, so by executing the system call twice, with a one second interval, the algorithm may be run backwards to calculate the number of runnable threads. However, this introduces a delay in the load gathering mechanism and is therefore not considered any further.

Information is gathered at fixed intervals p , where p is determined by a configuration parameter, and is distributed among the hosts in the system as described in the following.

Information Exchange Mechanism

In the following we describe the mechanism that implements the information exchange policy defined in 7.5.2.

The optimal distribution strategy, on a broadcast network such as the Ethernet, would be to broadcast the load information requiring N messages to keep N hosts updated. Unfortunately Mach 3.0 does not facilitate broadcasts, so another solution must be used. Having each host updating all the other hosts would require N^2 messages, so a solution where each load distribution facility sends the information to a central load distribution facility, which redistributes the information is chosen. This approach only requires $2N$ messages to be send, but unfortunately introduces a single point of failure. Failure detection protocols and voting schemes to elect a new central server can be introduced to retain fault tolerance, but it is considered beyond the scope of this work.

The information exchange scheme outlined above is similar to GLOBAL, where the local load is examined periodically with every period p , and load information is sent to the central LIC, which we incidentally have chosen to call the global master. The global master collects the load from all the participating nodes and assembles a load vector that is distributed to all participating nodes.

The primitives used to update load in the system is outlined below.

`elvis_updateLoadEntry(IN node, IN load)` This function is used to update the load indicator for a single node. It is intended to be used for communication to the global master, but it may also be used to communicate among the other nodes in the system.

`elvis_updateLoadVector(IN loadVector)` This function updates the entire load vector. It is intended to be used by the global master to update the other nodes.

These two primitives are sufficient to support all the information exchange policies that we can envision.

Load Update Restriction Algorithm

We have also implemented an algorithm that limits the amount of messages needed to exchange load information by only updating the system state when significant changes have occurred. The local load is still examined periodically, so this algorithm only limits the number of messages to and from the global master. This algorithm is outlined in the following.

First of all, we want to avoid sending messages when the load on the machine is insignificant. We therefore introduce a threshold, so that the local load is only sent to the global master when it is above the threshold. This prevents idle machines from sending messages, so it is important that a message is sent when the load drops below this threshold and the machine may become idle.

We also want to avoid updating the global master, when the load remains stable, i.e., unless the current load is very different from the last load indicator sent to the global master. Two configurable parameters (`risingLoadUpdate` and `fallingLoadUpdate`) are used to determine whether the global master should be updated or not. These parameters define how many percent the current load is allowed to vary from the last reported load before the global master is updated. An update is sent to the global master if the current load is more than `risingLoadUpdate` percent higher or `fallingLoadUpdate` percent lower than the last reported load indicator. We need two parameters because the increase in load from 10 to 20 constitutes a change of 100% while the decrease in load from 20 to 10 only constitutes a change by 50%, although the same absolute change in load is registered.

We have seen this algorithm work, but it is not used in our experiments for the following reason: the load distribution facility has a tendency to hang after 2–6 hours (this is probably caused by a resource leak that means that port rights in Mach gets exhausted) and it is impossible to determine whether the load is stable or the system is hung without probing the system and thereby influencing the performance measurements. When load indicators are updated periodically this error state is easily identified, because the global master does not update the other nodes, so we focus on periodic updates of load information throughout the rest of this thesis.

8.3.2 Control Component

In the following we describe the mechanisms that implement the policies of the control component defined in 7.6.

Initiation Mechanism

The initiation policy defined in 7.6.1 requires the local load to be examined and the load distribution facility should only be invoked if the local load is high. Load information is currently maintained by the kernel, and by the elvis server.

We choose to use the information from the elvis server although it may be slightly more costly than the system call to Mach. This means that we have no separate initiation mechanism, but rather implement it as an initial phase in the location policy that always return the current node when the load is low.

Selection Mechanism

The selection policy defined in 7.6.2 defined all clusters as eligible for load distribution.

This is not entirely true because when a cluster containing instance-objects has been mapped on a node, the clusters containing the class-objects and the code-libraries should

also be mapped on the same node. Clusters are always mapped so that instance-clusters are mapped first and the other type of clusters are mapped afterwards.

The `elixir_Probe` call provides information that allows the identification of the class-clusters and the code-libraries, so selection is performed based on this information. This means that selection is performed *after* the load distribution facility has been invoked.

Location Mechanism

The location mechanism implements a location policy based on a double threshold scheme as defined in 7.6.3. This scheme is extended with the responsibilities of the initiation policy, i.e., if the local load is low, no load distribution should be performed.

The following functions are used to implement the location mechanism.

`elvis(IN myNode, OUT bestNode)` This function locates the best target for load distribution for a mapping request originating at `myNode`. This target is located based on information from the local copy of the load vector in the following way: If the current load is less than the lower threshold the local node should be selected, otherwise a lightly loaded node should be sought. If no lightly loaded nodes exist a normal loaded node is sought and if no such node exist the entire system is overloaded, which means that the local node should always be preferred. If several nodes exist in a desired category, a round robin algorithm is used to avoid overloading a node with repeated mapping of clusters.

`elvis_getNode(IN node, OUT load)` This function returns the load for the node supplied as input parameter. The load information is looked up in the local copy of the load vector.

`elvis_getAllNodes(OUT loadVector)` This function returns the local copy of the load vector.

The first function implements the current location policy, while the last two functions are meant for future extensions or other policies build on top of the existing `elvis` engine.

Transfer Mechanism

The transfer policy defined in 7.6.4 uses the existing diffusion mechanism for load distribution.

Transfer is based on the output from the `elixir_Probe` call. The storage server may either reply that mapping is possible on the requested node, or that mapping is impossible because the cluster is already in use by some user on another node (possibly the local node).

The reply that mapping is possible means that the storage server has made a reservation for the cluster on that node, i.e., all subsequent calls within a fixed timeout will be told that the cluster has already been mapped on that node. The reply parameters are then rewritten to say that local mapping is impossible because the cluster has already been mapped by ourselves on the remote node. This reply is returned to the upper layers of the virtual machine, and the activity has to diffuse to that node in order to invoke the object. The mapping on the remote node will succeed because the cluster has been reserved for that node. The reservation is necessary because the mapping request on the remote node also calls `clu_Map` that invokes the load distribution facility, which might suggest another node, so unless something is done this could go on forever.

The parameters returned by the `elixir_Probe` call in case the cluster is already in use are identical to the reply from the `elixir_Map` call that actually maps the cluster. It is therefore possible to return the parameters unmodified to the upper layers so that the activity may diffuse to the node where the cluster is mapped.

8.4 Modifications to the Guide-2 System

Only two modifications were needed to the Guide-2 system: the addition to the upper layer of the cluster manager, and the `elixir_Probe` call added to the storage server.

The probe call is primarily needed to reserve the cluster for a given node.

8.5 Summary

This chapter described the implementation of the Guide-2 load distribution facility called `elvis`.

`Elvis` consist of a library that is linked into the upper layer of the cluster manager, and a separate server that implements the load distribution policies.

The cluster manager calls a function in the `elvis` library to invoke the load distribution facility. This call initializes the load distribution facility and receives the best node from the external server.

The storage server is probed to determine whether the cluster may be mapped on that node, and if this is the case the activity diffuses to that node where the cluster will be mapped.

Chapter 9

Evaluation

This chapter describes our evaluation of the load distribution facility. This evaluation is accomplished by a series of experiments performed to establish certain goals. Before we present the design and evaluation of these experiments a proper framework for the evaluation is defined. This framework consists of the goals of the evaluation as well as the evaluation strategy.

The environment of the evaluation imposes restrictions on the evaluation so it must be described, to allow the decisions we make throughout this evaluation to be understood in their proper context.

The load distribution facility has been added to an already existing system, so the evaluation should focus on three different areas, namely the load distribution facility itself, the effects of the load distribution facility on the system, and how the design of the system effects load distribution. We call these three areas the software evaluation, the performance evaluation and the system evaluation respectively.

This chapter has been organized, so that 9.1 states the main goals of the evaluation. The evaluation strategy is defined in 9.2, which is followed by a description of the evaluation environment in 9.3. Having defined the framework of the evaluation the rest of the chapter will present the design and evaluation of our experiments. This starts with the software evaluation in 9.4, followed by the performance evaluation in 9.5, and the system evaluation in 9.6. Each of these evaluations contain their own conclusions, and a summary of these conclusions is given in 9.7.

9.1 Evaluation Goals

This evaluation has several sets of goals. Some of these goals are of general nature and should be considered throughout this evaluation, while others relate specifically to the three areas of evaluation defined above.

Few implementations of load distribution in object-oriented systems have been made, and even fewer performance measurements have been reported, so one of the main goals of this evaluation is to gain and report experiences with fine-grained object-oriented load distribution.

Another main goal is to design the experiments pessimistically, so that the reported benefits of load distribution are smaller than should generally be expected. This goal is impossible to evaluate, but it is crucial to the design of the experiments.

The rest of our goals relate to the three areas of evaluation defined above.

Software Evaluation The purpose of the software evaluation is to determine whether the design goals stated in 7.2 has been achieved, and to identify areas where improvements of the load distribution facility are possible.

We do not attempt to perform an exhaustive evaluation of the load distribution facility, in order to obtain an optimal configuration of this mechanism. The configuration of the load distribution facility used throughout the evaluation in this chapter is based on a few experiments, so it should not be assumed to be optimal.

Performance Evaluation The performance evaluation estimates the impact of load distribution on the performance of the Guide-2 system, so that the main hypotheses from in 1.3 may be confirmed or denied.

We do not perform an exhaustive evaluation of the possible performance benefits, but focus on a few important workload scenarios.

System Evaluation The system evaluation focus on the Guide-2 system and identifies areas where modifications may improve the performance of the load distribution facility, either by modifying the system, e.g. the Elliott kernel or the compilers, or by guidelines for the application programmers to follow.

9.2 Evaluation Strategy

The experiments of this evaluation has been designed according to the technique described by Jain [Jain 91]. This technique is outlined in the following along with some of our general decisions.

Jain identifies the following eight steps in measuring system performance.

1. Define the goals of the evaluation.
2. Define the system under test.
3. Identify the parameters that influence the performance of the system under test.
4. Select the factors that will be varied in the test from the identified parameters.
5. Select the evaluation technique.
6. Select the workload.
7. Design Experiments.
8. Analyse data.

These steps will be described further in the following.

Definition of Goals Defining the goals of the evaluation is probably the most important step of them all, but according to Jain one that is often neglected. Without exact goals the experiments may result in ambiguous or misleading experiments.

Define the System Under Test The system under test (SUT) is defined in the second step. Not all parts of the system are directly relevant to the performance evaluation, so those that are must be identified, e.g. the protection mechanisms, the object visualization server (elmer), and the transaction support of the storage server are not important for the performance of applications in Guide-2.

Identifying the Performance Parameters All the parameters that may influence the performance of the system under test should be identified and listed. Typical parameters in this evaluation may be system related like the time it takes to map a cluster or workload related like the arrival pattern of applications on a node.

Select the Factors of the experiments Those parameters from the list above that will be varied in the experiment are called the factors of the experiment. The number of factors in the experiment and the number of different values, they may have, determine the size of the experiment. It is important to choose the right number of parameters and the right number of values for these parameters, so that the goals of the experiment may be fulfilled, without exceeding the resources available for the experiment.

Selecting Evaluation Technique Different techniques of evaluation may be appropriate depending on what the goals of the evaluation are. We have identified four different techniques that may be used for systems performance evaluation, these are:

1. Small test applications with high demand for system resources (macro-benchmarks)
2. Test programs that evaluates a certain aspect of the system (micro-benchmarks)
3. Existing applications
4. Test-suites that mimics the behaviour of job traces collected from a similar system.

Examples of goals where different techniques may be appropriate are the measurement of a single system parameter, where a micro-benchmark should be used or a stress test where a macro-benchmark is most appropriate.

Select the Workload The workload is defined by a list of service requests to the system, which in our case is a list of applications that should be executed and the arrival rate of these applications. This workload should be defined so that it will vary the factors of the experiment in a way that may establish the goals of the evaluation.

Design the Experiment This step covers the selection and/or creation of the applications needed to generate the workload selected above, along with the actual execution of these applications and the registration of their results.

As there are few applications written for Guide-2, we generally have to create the test-applications ourselves. These applications should be simple to write and simple to execute, so one of our main design criteria for the test applications is simplicity.

We have no way of measuring time from within Guide-2 applications, without modifying the system, so we choose to register the run-time of applications by the elapsed wall-clock time.

This approach only gives an indirect measurement of the run-times, because we are also measuring the accuracy of the system clock, and the fluctuations in the execution time caused by the scheduler or by retransmission of messages on the network. This pollutes our measurements, so it will have to be taken into account when the experiment is designed.

Fortunately the effect of this pollution may be avoided by repeating the experiments as described below. It is therefore only cases, where the pollution may bias the results of the experiment in a single direction that will be considered.

Analysing Data The output of measurements are generally subject to variation introduced by statistical fluctuation. This variation must be analysed in order to avoid conclusions based on insignificant measurements.

A proper analysis of this variation requires great skills and experience in statistical analysis, so we choose another way of controlling the variation of our measurements.

Instead of performing a variance analysis, we repeat our experiments and rely on the central limit theorem, which states that “*the sum of a large number of independent observations from any distribution tends to have a normal distribution*”. The mean of this normal distribution will be the “true value” of the experiment. We may therefore compare two sets of measurements by comparing the mean values of each set.

9.3 Evaluation Environment

Before we describe the experiments of this evaluation, the environment of these experiments should be described. This is important because the conditions under which the experiments have been performed have had a big influence on the evaluation.

9.3.1 Hardware Configuration

The experimental environment consisted of three Zenith 33MHz 486 PC's connected by a 10 Mbit Ethernet. One of these PC's is dedicated to the storage server, so only two machines are available for user applications.

We dedicate a machine to the storage server to eliminate the effects of the difference in time it takes to map a cluster, when the application is either running on the same machine as the storage server or on a remote machine (a small experiment showed a factor of two in favour of mapping clusters locally on the same machine as the storage server as opposed to mapping them remotely). This significant difference in mapping times would introduce a random element in our measurement of run-times, where a fortunate placement of applications may appear as a significant performance improvement.

The use of a dedicated storage server reduces the number of the machines available for load distribution, but we asses the advantage of a uniform run-time on all nodes to be higher than the loss of the third node. The total number of PC's available for our experiments is rather small, but unfortunately no more machines could be used.

We had exclusive access to the PC's for the duration of the experiments, while the network was shared with the other users at Bull-IMAG.

9.3.2 Running on Mach 3.0

The PC's are all running Mach-NORMA, which is an enhanced version of Mach 3.0 developed by OSF. This version of Mach caused some problems with the evaluation that will be described in the following.

The main problem with this system is the granularity of the clock, that is available to the users though system-calls. This granularity is 10 ms., which is very coarse (it is the same size as the time-slices used for scheduling in UNIX). This coarse granularity means that experiments should be executing for a long time and that the operations should be repeated many times in order to achieve a satisfactory accuracy when measuring time.

Furthermore, we experienced frequent system failures, and hung systems that may in part be attributed to the Mach-NORMA kernel. These errors meant that the systems had to be rebooted every two or three days, and that numerous experiments had to be repeated.

9.3.3 Evaluation over the Internet

Although the design and implementation of the load distribution facility, has been done during a stay at Bull-IMAG, the evaluation has been done after returning to the University of Copenhagen, which means that the tests have been performed over the Internet from Denmark to the machines located in Grenoble. This had a significant influence on the possibilities for performing the evaluation.

The Internet connection from Denmark to Grenoble has at times been unreliable, with frequent network failures at Nordunet in Stockholm and with France Telecom in Lyon. These frequent network failures have meant that many experiments only ran partially and had to be repeated. This has forced us to limit the number of experiments in this evaluation.

9.3.4 The Experimental Nature of Guide-2

Guide-2 is a research prototype, so it is not as stable as should be expected from releases of commercial operating systems. Furthermore the system was constantly evolving and several system components was modified while the load distribution facility was implemented.

This means that the evaluation may unveil bugs that should either be fixed or worked around. We have generally chosen the second approach although it may limit the scope of our evaluation and in some cases influence the generation of the evaluation workload.

As indicated by the name Guide-2 is the second incarnation of the Guide system. Since many of the goals were already fulfilled by the first incarnation not everything has been fully implemented in Guide-2, e.g. the compiler for the Guide language does not support all features of the language specification. This state of the system imposes some serious restrictions on the experiments that may be performed, e.g. the inability to supply online arguments to applications means that all external references must be hard-coded into the applications, making the implementation of test applications a very time consuming task.

9.4 Software Evaluation

The software evaluation should determine whether the design goals stated in 7.2.2 and 7.2.3 have been fulfilled. These design goals are:

Limitation of system overhead is a prime concern in the design of the load distribution facility. The overhead imposed by the load distribution facility should be negligible.

Limitation of resource consumption is another important design issue. Excess resource consumption by the load distribution facility could be better used in the applications.

Must be easily extendible in order to accommodate the emerging technologies in the new field of object-oriented load distribution.

Must be easily configurable to facilitate experimentation with different load distribution policies.

No kernel modifications should be required of the underlying micro kernel, and only limited modifications to the Elliott system should be required.

No modifications of existing applications should be required for them to benefit from load distribution. It is however acceptable that applications written with load distribution in mind benefit more from load distribution than other applications.

The first two goals will be evaluated in 9.4.1 and 9.4.2 respectively, while the rest of the goals are discussed together in 9.4.3.

9.4.1 System Overhead

The system overhead is defined as the additional time required to map a cluster when the load distribution facility is used. It is measured as the difference in run-times of an application executed first without load distribution and then with the load distribution facility enabled. We only enable the load distribution facility on a single node, so that no remote execution will take place.

By comparing the run-times we are also comparing the time used initializing the application, mapping clusters and fluctuations in run-times caused by scheduling and network load. However, these factors are the same regardless of whether the load distribution facility is enabled or not, so we may therefore compare the two sets of measurements by comparing the mean values of each set.

Another problem is the granularity of the clock supplied by the system, which is very coarse i.e., the granularity of the clock available to programmers through system calls to Mach-NORMA is 10 ms. The time to map a cluster locally on the machine running the cluster-manager is 8 ms. [Chevalier 94], which is within the same order of a magnitude, i.e., several clusters will have to be mapped in each application in order to get an acceptable accuracy of the measurements.

We have experienced some problems with the storage server, when the same application maps many clusters (more than 10) in quick succession, so we will use an application that only maps 10 clusters. This is not optimal but the calculated time for mapping 10 clusters should still be an order of a magnitude larger than the granularity of the clock.

The experiment is performed by executing the application 100 times on a node different from the one running the storage server. Then the load distribution facility is enabled on that node alone, and the experiment is repeated. The results of these experiment is shown in the table below.

Mapping 10 clusters	
Without load distribution	19825.9 ms
With load distribution	20943.8 ms
Difference	1117.9 ms

Table 9.1: Execution times of an application with and without the overhead of the load distribution facility.

We divide this difference in run-times by the number of clusters in the applications, to calculate the overhead of the load distribution facility on a single cluster. This may then be compared with the original mapping time of a cluster, when the load distribution facility is disabled, found on page 113 and the mapping time of a single cluster with the load distribution facility enabled may be calculated, as shown in table 9.2.

The overhead of the load distribution facility increases the mapping time of clusters by a factor of almost 2.5. This is far more than we hoped for, but considering that we do both

Original mapping time	77.3 ms
Overhead of load distribution	111.8 ms
New mapping time	189.1 ms

Table 9.2: Calculation of the overhead introduced by the load distribution facility.

a local RPC to get the best node from the load distribution facility, and a remote RPC to probe the storage server about the intended mapping, this overhead is not so surprising.

The new mapping time may roughly be broken up into the following components, invoking the load distribution facility ($\frac{1}{2}$), probing the storage server (1), and the original time it takes to map the cluster (1).

The time taken to invoke the load distribution facility, can be improved by keeping the load indicators in memory shared by the load distribution facility and the applications, thus avoiding the local RPC.

The probe-call to the storage server reuses most of the code used to validate a request to map a cluster. It is possible to improve the time it takes to probe the storage server by optimizing the storage server code. Such improvements are however expected to be very modest. Another way of limiting the overhead is to integrate the load distribution facility with the storage server. This is only possible in the current implementation, where load distribution is performed at the cluster level, so it would mean a loss of flexibility to integrate the two components.

9.4.2 Resource Consumption

Different system resources are consumed by the load distribution facility, when it is running. These resources could otherwise have been used by applications, so it is important that the amount of resources consumed by the load distribution facility is minimal.

The load distribution facility consumes three different resources, when it is gathering load indicators, distributing these indicators and making placement decisions, namely CPU-time, memory and network bandwidth.

In the following we will estimate the amount of each resource that is consumed when the load distribution facility is running. These estimates will either be based on measurements obtained from the underlying system (Mach or OSF/1) or they will be calculations based on knowledge of system properties and algorithms.

CPU-Time

The amount of CPU-time consumed by the load distribution facility is directly obtainable from the OSF/1 system used to launch Guide-2 services and applications with the command `time(1)`. This command tells us the amount of CPU-time (user-level + system-level) used by a process, so using `time` on the load distribution facility for a fixed period of wall-clock time will tell us how much CPU-time have been consumed in that period.

This amount of CPU-time will depend on the frequencies of gathering and distributing load indicators as well as the number of placement decisions made by the load distribution facility. Such information may be very useful when the load distribution facility is configured in its final version, but our purpose is only to determine whether the load distribution facility is lightweight, so the scale of the CPU-time consumption is sufficient.

We have measured the CPU-time consumed by the load distribution facility over a period of 10 minutes, where a light workload of 20 applications was executed.

We report the CPU-requirements of different Guide-2 services on a one minute basis. We have registered both the time spend by the service in user space (User time) and the time spend by Mach-NORMA (System time) on behalf of the service. The total CPU requirements of the service per minute may be found by adding these two numbers.

Guide-2 Service	User time	System time	Elapsed time	CPU requirement
Guide Daemon	0.14 s	0.04 s	0.18 s	0.30 %
Storage Server	0.16 s	0.20 s	0.36 s	0.60 %
Object Visualization	0.39 s	0.11 s	0.50 s	0.83 %
Load Distribution Facility	1.54 s	0.43 s	1.97 s	3.28 %

Table 9.3: The measured CPU requirements of selected system services

The load distribution facility is the only service that is running regardless of the activity in the system. It is therefore not surprising that it is the service that requires most CPU resources with the extremely light test workload used in this experiment.

Memory

We want to measure the amount of physical memory used by the load distribution facility. This memory usage is obtained directly from the system with `ms(1)`, which is a system utility for Mach 3.0. This utility gives us both the amount of virtual memory (VM) demanded by the application as well as the resident set size (RSS), i.e., the amount of physical memory currently allocated to the process. The virtual memory size express the total size of the application with all the libraries that have been linked into it regardless of whether they are needed, while the resident set size express the amount of memory actually used by the application. The resident set size measures the actual memory requirement better, as long as physical memory is available and neither paging nor swapping occurs.

We have measured the memory requirements of different Guide-2 system services while physical memory was plentiful, and listed the services in order of increasing resident set size in table 9.4 below.

Guide-2 Service	Resident Set Size	Virtual Memory
Storage Server (kelixir)	0.48	2.55
Load Distribution Facility (elvis)	1.29	7.58
Execution Visualization (elmer)	1.34	3.68
Storage Configuration (elisa)	1.44	3.78
Guide Daemon (gd)	1.52	3.90

Table 9.4: Memory requirements of different Guide-2 services (with the name of the service in parenthesis)

The virtual memory requirement of the load distribution facility is about the same size as most Guide-2 services with respect to resident set size, while the virtual memory requirement is about twice as large. The larger virtual memory may be attributed to the way the load distribution facility has been implemented. We reused existing code written for other parts of the system, which means that the load distribution facility had to be linked with many libraries.

Network Bandwidth

Network bandwidth is consumed when load information is updated in the system and when the storage server is probed for a possible mapping.

In every load update period, all active nodes send their load to the global master that constructs a load vector from this information and distribute it to all active nodes in the system.

Each update requires $n - 1$ updates to be sent to the master and $n - 1$ load vectors to be distributed to the nodes, where n is the number of active nodes in the system including the master. The size of the RPC header of a Mach message is 40 bytes so the size of an update message is 44 bytes and the size of a message containing a load vector is 56 bytes. The network bandwidth consumed by updating the load information in one load update period is $(n - 1) \times 100$ bytes.

The experiments described in this chapter has been performed with a load update period of 5 seconds and two nodes, so the bandwidth consumed is

$$\frac{100 \text{ bytes} \times 8 \text{ bits/byte}}{5 \text{ s}} = 160 \text{ bits/s}$$

which is 0.0016% of the total bandwidth of an Ethernet.

The network bandwidth consumed by probing the storage server depends on the number of times the load distribution facility is invoked. Each time the storage server is probed 176 bytes are send across the network ($2 \times (40 \text{ bytes for the Mach RPC header} + 48 \text{ bytes of parameters})$). This uses 0.014% of the total bandwidth so we may perform 71 placement decisions per second without exceeding 1% of the total bandwidth.

All of our calculations indicates that the network bandwidth consumed by the load distribution facility is negligible.

9.4.3 Software Design Goals

The software design goals stated in 7.2.3 are easy to evaluate, but most of this evaluation is difficult to document. We choose to document this evaluation by simply describing our experiences working with the load distribution facility. Each of our stated software design goals will be evaluated separately below.

Extendibility

The extendibility of the load distribution facility is best illustrated by the number of alternative policies that have been implemented for the load distribution facility. We have implemented two different ways of keeping state, (either globally on all nodes or centralized), two policies for updating load information (ether periodic or depending on the activity on the node), and two node selection policies (either random or round robin among the least loaded nodes).

The highly modular design of the load distribution facility, has made the addition of new policies very easy.

Configurability

The behaviour of the load distribution facility is controlled by a large number of parameters described in appendix A. We have not worked much on the configuration of the load distribution facility, but our limited experience indicates that the available parameters are sufficient to adapt the load distribution facility to different workload scenarios.

The only option that we could have wished for is the ability to designate a single node to be selected by the load distribution facility. However this option is only useful, when the load distribution facility is tested, so it is not needed with respect to the load distribution.

Kernel Modifications

The load distribution facility has been implemented in two parts, a library that is linked into the applications and a separate server. The only modification needed in the existing system was the addition of an extra call (the probe call) to enquire the storage server whether a remote mapping was possible and to avoid the ping-pong effect.

Application redesign

The load distribution facility has been implemented in the context of the virtual machine without changing the semantics of the existing interfaces, so existing applications may be used without modification to the code. Recompile of the applications is however necessary for the applications to take direct advantage of the load distribution facility, indirect benefits may arise without recompilation if enough new applications use the load distribution facility to be scheduled on other machines.

9.4.4 Conclusions and Perspectives

The overhead introduced by the load distribution facility is much larger than we had hoped for, almost increasing the time needed to map a cluster by a factor of 2.5. This overhead may be decreased by optimizing the load distribution facility, but we should probably expect the overhead of an optimized load distribution facility to double the time it takes to map a cluster. This overhead is largely caused by probing the storage server to avoid the ping-pong effect and is specific to the implementation of load distribution at the cluster level.

The load distribution facility consumed more than 3% of the CPU resource, which is more than we expected. It is however difficult to say whether it is too much, because the consumed CPU time should be weighted against the performance benefits of load distribution. Furthermore the consumption of the CPU resource may easily be decreased by increasing the load update period.

The memory requirements of the load distribution facility is about average with respect to the resident set size and we believe that the virtual memory required by the load distribution facility may be decreased to the level of the other system services by restructuring the existing libraries. Such restructuring is however far beyond the scope of this thesis.

The load distribution facility is only consuming a small fraction of the total network bandwidth.

All our experiences with the load distribution facility indicates that our software design goals have been met. It is possible that further configurability may be wanted when the load distribution facility is optimized for use in a given environment, but we believe that the current options for configuration covers the needs of most environments.

We have achieved all the design goals for the load distribution facility, except the limitation of system overhead. A large part of this overhead may probably be eliminated by integrating the load distribution facility with the storage server. This integration would conflict with the flexibility of the load distribution facility and is beyond the scope of this thesis.

9.5 Performance Evaluation

The goal of the performance evaluation is to measure the impact of the load distribution facility on system performance. These measures may be used to evaluate the efficiency of the implementation and the algorithms of the developed load distribution facility, but the main goal of this evaluation is to confirm or deny the hypothesis, stated in 1.3. These hypotheses are:

1. Load distribution offers general performance benefits in a distributed system with a fine grained unit of distribution, such as the Guide-2 system.
2. Load distribution may significantly improve the performance of some applications, when the load on the nodes in the system is unbalanced.
3. The granularity of distribution is important for the size of these performance benefits.

The first hypothesis will be evaluated by the experiment with an average workload described in 9.5.1 below. The second hypothesis will be evaluated in 9.5.2 with respect to a workload where the applications are CPU-intensive. The third hypothesis has been the topic of a prior investigation [Jensen 95] that will be described in the system evaluation in 9.6.1. The conclusions of the performance evaluation are found in 9.5.3.

9.5.1 Average Workload Test

Guide-2 is a research prototype which means that the number of people who are actively using the system is very small. It also means that only a few applications have been developed for or ported to the Guide-2 system. The absence of actual users and applications, means that it is impossible to base this evaluation on actual job traces or to create a model of an average workload based on such traces, so we need another way of modeling an average Guide workload.

The main part of this test is devoted to the construction of a workload model based on information from other sources that may be conceived as a probable Guide-2 workload. The model should tell us what parameters to use as factors of the experiment and the levels of these factors, e.g. it should tell us the number of applications to use, the sizes of these applications (both with respect to data-size and execution time), the arrival pattern of applications on a node, the possible interactions among applications and the internal structuring (with respect to clustering and invocation patterns) of the applications.

With this information we may create a test-suite of artificial applications that mimics the behaviour of an average workload in Guide-2, i.e., enable us to build a small set of applications that have the same characteristics as the original applications. The implementation of these applications requires many resources (all references and computational behaviour needs to be hard-coded into each application) so it is important that the number of synthetic applications be kept small.

Before we define this model any further the first three steps of the evaluation (stating the goals, defining the system under test, and identifying the performance parameters) should be taken.

The goal of this evaluation is to investigate the possible performance benefits of load distribution on a typical Guide-2 workload.

The system under test is the Elliott kernel and the Elvis load distribution facility. We have previously decided not to focus on the configuration of the load distribution facility, so the SUT may be restricted to the elements of the Guide-2 system directly involved in

starting and executing an application, which means that the services offered by the SUT are:

1. Map a cluster locally
2. Extend the job to a remote node
3. Invoke an object in a cluster

The service of starting an application is mainly composed of 1 and 3.

We have identified the following three top level parameters that influence the performance of the SUT.

1. local workload
2. remote workload
3. network load

We select the local workload as the only factor of our experiment for the following reasons. By only generating workload on one node, the other node will act as an idle workstation that previous work mentioned in 2.1.1 has shown generally exist in a workstation environment. A previous study of communication times between workstations [Jensen 93], shows that the variations in network load is highly unpredictable, and modeling and generating an average network load is beyond the scope of this thesis. Instead we perform the measurements at night, when nobody else is using the network at Bull-IMAG.

The following parameters influencing the performance of the local workload have been identified.

1. The arrival rate of applications
2. The sizes of applications, with respect to run-time
3. The amount of time applications use for computing, terminal I/O, and referencing persistent object (corresponding to disk I/O in traditional operating systems)
4. The sizes of different clusters within applications
5. The invocation patterns between objects in different clusters

The arrival rate and run-times of applications reflect the users need for computational resources. We may obtain information about these parameters by examining application traces from another workstation environment, e.g. UNIX. Although UNIX and Guide are fundamentally different, information from a UNIX environment, reflects the computational requirements of users in a networked workstation environment, which may then be translated into computational requirements in Guide-2. It is, however, important to note that applications tend to be adapted to the underlying system, so the results obtained from this type of workload generation should be read with the widest possible margin.

We do not need the model to consider terminal I/O because this is not supported by the current implementation of the Guide runtime system.

As previously stated only few applications have been written for Guide-2, so it is impossible to know exactly how Guide-2 applications will be structured. Furthermore we have found no references in the literature describing the structure of typical applications written in fine-grained object-oriented languages. We therefore structure our test applications based on information obtained through interviews with people at Bull-IMAG that have worked with both incarnations of the Guide system.

Application Arrival Rates

Our observations of a small number of personal workstations used at DIKU has shown two main types of application arrival rates.

The first and most frequent type is an idle workstation, where applications arrive with large intervals (more than 5 minutes). These applications do typically not require many resources, so the resources locally available will be sufficient for this type of applications. This type of workload is unlikely to gain from load distribution, and it will not be considered any further.

The second type of workload is a short burst of applications, where 15–30 applications arrive within the same minute¹. This is the type of workload that we want to examine with our experiments.

We simulate this burst with 20 applications arriving at random times within the same minute. This number of applications has been chosen because it fits our observations and our goal of limiting the number of applications at the same time.

Application Run-Times

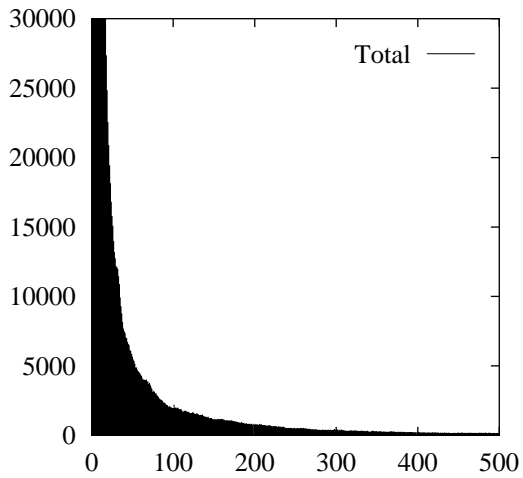
We have examined all the accounting logs from 1994 of five UNIX workstations placed at the Department of Computer Science at the University of Copenhagen in order to build a model of the typical workstation users computing requirement. To ensure that the data set represents the computing requirements of users, all applications run by UNIX system users (the users root, daemon, news, etc.) have been filtered from the data set. The run-times of jobs recorded in these accounting logs are summarized by the histograms in figure 9.1, where both axes have been cut off because the data points otherwise would be indistinguishable from the axes.

The distribution of run-times shown in the histograms in figure 9.1 is very similar, so we concentrate on the measurements from a single host throughout the rest of this evaluation.

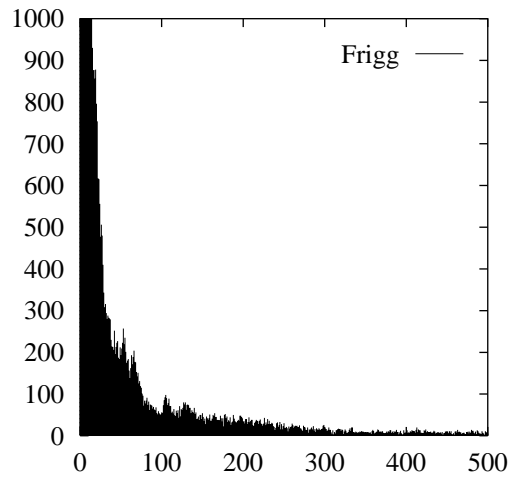
The chosen workstation is a HP9000s735/99 (Frigg) running the HP-UX 9.05 version of UNIX. It is used both as a workstation for a researcher at DIKU and for executing jobs that are expected to run for a longer time. A histogram of the number of jobs that have been executed on this workstation during 1994 is shown in figure 9.2, where the X-axis have been cut off at a run-time of 500 ms. and the Y-axis have been cut off at 25000 jobs, so that some detail may be shown.

The most important characteristics of this data set are summarized in table 9.5. All values are given in UNIX accounting ticks, awarded the running process every $\frac{1}{60}$ second. This means that processes may appear with a zero tick run-time, if they have run for less than one whole accounting tick.

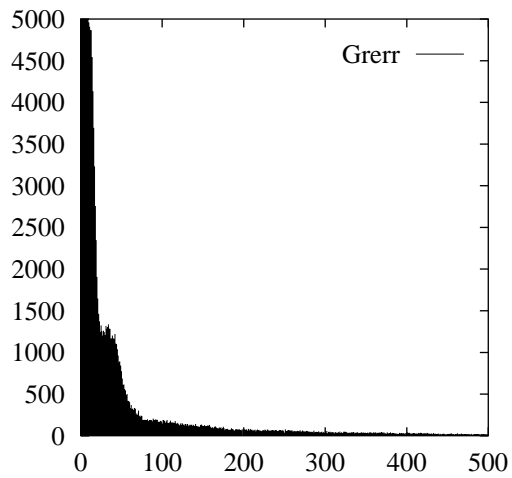
¹The observations were made with `lastcomm(1)` which only registers the minute when the process terminates. Our investigation of UNIX run-times summarized in table 9.5 shows that more than 99% of UNIX applications run for less than one minute, so we assume that processes arrived within the same minute that they terminated.



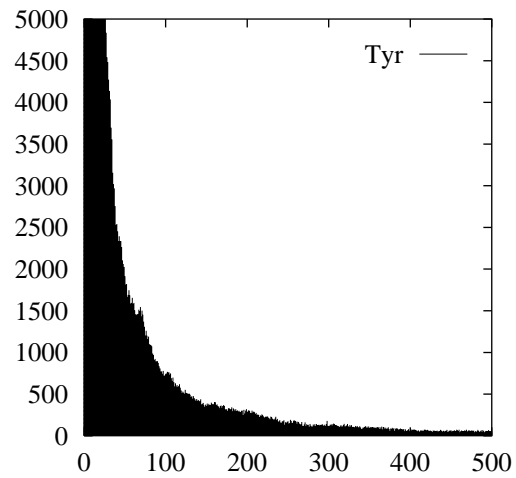
(a) Measurements from all hosts



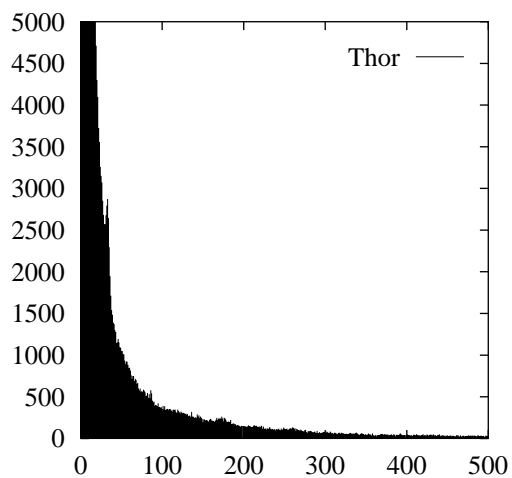
(b) Measurements from Frigg



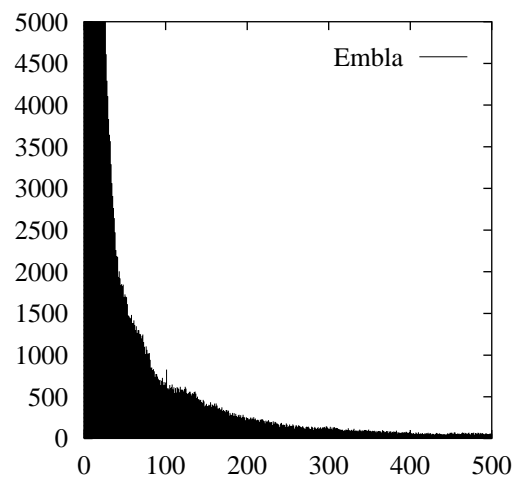
(c) Measurements from Grrr



(d) Measurements from Tyr



(e) Measurements from Thor



(f) Measurements from Embla

Figure 9.1: Histograms of application run-times

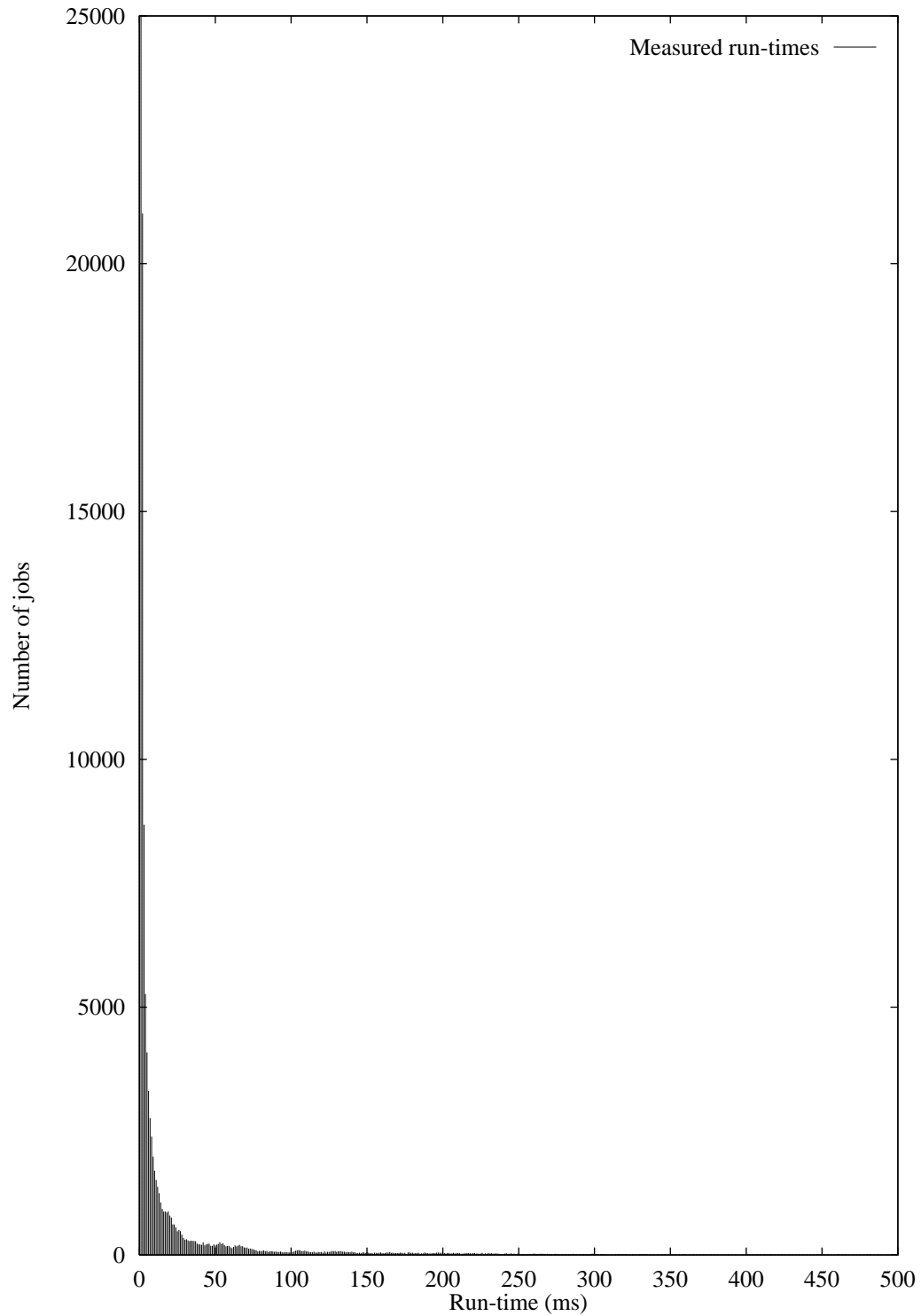


Figure 9.2: Histogram showing the extreme nature of the measured run-times. The histogram shows all processes with run-times less than 500 ms. Ordinates have been cut off at 25000, so that more data points may be distinguished

Property	Value (ticks)
Minimum	0
First Quartile	0
Median	0
Third Quartile	1
80 Percentile	2
85 Percentile	4
90 Percentile	13
95 Percentile	65
99 Percentile	2220
Maximum	87556096
Mean	1630

Table 9.5: Characteristics of the observed UNIX workload.

These number together with figure 9.2 suggest that the run-times are distributed according to an exponential distribution. However this is not the case, which may be easily seen from figure 9.3, where the histogram is plotted with a logarithmic scale on the y-axis.

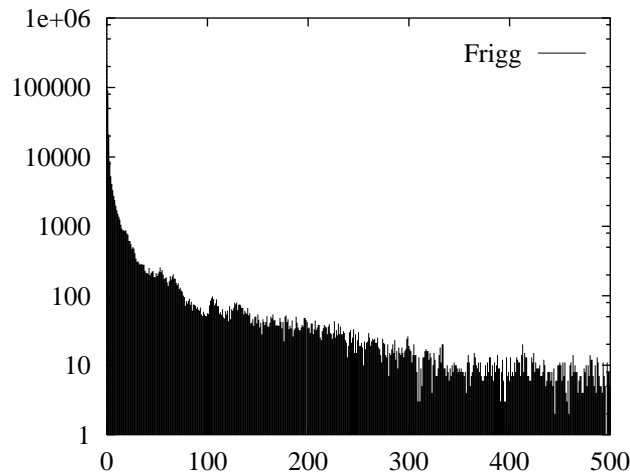


Figure 9.3: Logarithmic histogram of the measurements from figure 9.2

An exponential distribution plotted with a logarithmic scale is linear, while the histogram of figure 9.3 has retained the nature of the exponential distribution. The distribution is clearly not exponential and we have no other hypothesis regarding the distribution, so we have to analyse the measurements without the tools of statistical analysis.

Our main difficulty in modeling the measurements is the tails of the distribution (the values around the minimum and the maximum of the distribution), especially the more than 65% of all jobs that have been registered with zero run-times. This is obviously not possible, so these run-times will have to be modified, so they better reflect the applications they are supposed to model.

Since we cannot model the distribution statistically, it is impossible to determine the weight that should be attributed to the applications with run-times in the interval $[0, 1[$ by calculating the integral of the distribution, so we have to use a simpler method.

The only good choices, when nothing is known, are either to use the mean of the interval (0.5) or the minimum time needed to start an application. This start time is determined

experimentally on page 111 to be 11.1 ms, which corresponds to 0.666 accounting ticks. The time needed to start an application is greater than the mean of the interval, so we choose to use this value instead of the zero run-times in all our calculations.

The other tail is simply cut off, i.e., we discard the 1% longest running jobs because they mainly consist of applications run by a Ph.D. student doing research in algorithmics and combinatorial optimization. This type of applications are very rare in the office environment that Guide-2 has been developed for.

It seems natural to divide these observed run-times into three classes, light-weight, medium-weight and heavy-weight applications. We choose this small number of classes because it seems to fit the observations well, but also because it means that fewer test applications have to be designed for the experiment (all applications in the test-suite needs to be written and compiled separately).

The class of light-weight applications represents those 85% of the observations with a run-time from 0 to 4 ticks. The medium-weight applications represent the 10% of the applications with run-times between 4 and 65 ticks while the heavy-weight applications represent the remaining 4% of the applications with run-times between 65 and 2220 ticks.

We will now represent each of these classes with an artificial application that has the average run-time of the class, but before we start to design these applications, we need to determine how well they fit the observations. We therefore calculate the mean, variance, standard deviation and coefficient of variation (C.O.V.) for all three classes. The C.O.V. is the ratio of the standard deviation to the mean, i.e., $C.O.V. = \frac{\text{standard deviation}}{\text{mean}}$.

Property	Light-Weight	Medium-Weight	Heavy-Weight
Mean	0.9458	8.0018	183.2174
Variance	0.3692	6.3047	111604.6
Standard Deviation	0.6076	2.5109	334.0727
C.O.V.	0.6424	0.3138	1.8234

Table 9.6: Characteristic properties of the three classes.

We can determine how well the mean fits the class by looking at the C.O.V., this number expresses the dispersion independent of the values of the observed entities. The C.O.V. of the medium-weight class is very low, so the mean is a good representative for this class. The C.O.V. of both the light- and heavy-weight classes are slightly higher, but since both classes represent the “tails” of a distribution the fit must be said to be satisfactory.

The measured run-time of the three classes of applications are shown in table 9.7.

Application Class	Mean Run-Time	
Light-Weight	0.9458 ticks	15.76 ms
Medium-Weight	8.0018 ticks	133.36 ms
Heavy-Weight	183.2174 ticks	3053.62 ms

Table 9.7: UNIX run-times of the application classes

Now that we have divided the applications into three categories, the execution requirements of these categories must be translated to Guide applications. UNIX applications go through two phases, where the first phase includes the creation of the process, allocation of kernel structures and the loading of the single address space into memory, while the

second phase is the execution of the application code. Guide applications will have an extra phase of cluster mapping because of the dynamic binding of objects. The different phases of the two systems are illustrated in figure9.4.

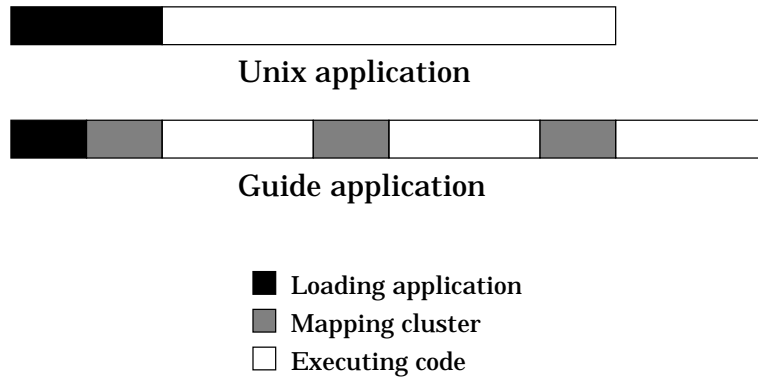


Figure 9.4: Model of execution for UNIX and Guide

The amount of work being done in the starting phase of an application is fairly constant, so it is safe to assume that the time it takes to do this work is also constant. This makes it possible to calculate the load time by running an application several times, each time executing more iterations of the same work loop. If the execution times of these measurements are fairly linear, the time needed to load an application will be found at the intersection between this line and the y-axis, and it can be calculated by linear regression.

We have executed a simple application 20 times with 1000, 5000, 10000 and 20000 iterations. The execution times of these applications are shown in figure 9.5.

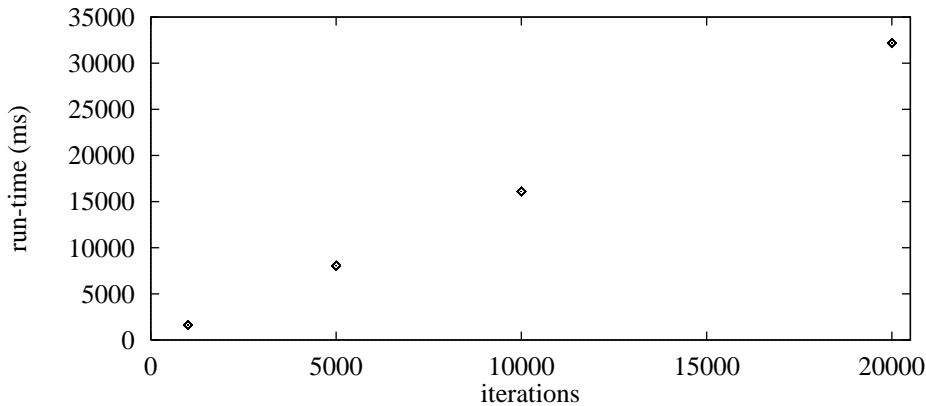


Figure 9.5: Execution times of a UNIX application as a function of the number of iterations

The data points are clearly linear so we may use linear regression (least squares method) to find the gradient as:

$$\beta_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

and the intersection with the y-axis as:

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

This way we find the time required to start an application in the UNIX system to be 11.1 ms, which should be subtracted from the run-times shown in table 9.8, in order to give the actual CPU requirement of the application classes in the UNIX system.

Application Class	CPU Requirement
Light-Weight	4.66 ms
Medium-Weight	122.26 ms
Heavy-Weight	3042.52 ms

Table 9.8: CPU requirements of the application classes under UNIX

The first step of transforming the observed CPU requirements from UNIX into a Guide-2 workload is to consider the difference in processor speed between the two systems. We used a simple benchmark to determine the difference in integer performance, and found that the workstations at DIKU were 2.5 times faster than the PC's running Guide-2. We therefore have to multiply the run-times from table 9.8 by 2.5 to get the expected CPU requirements of the application classes in Guide-2 listed in table 9.9.

Application Class	CPU Requirement
Light-Weight	11.50 ms
Medium-Weight	305.65 ms
Heavy-Weight	7606.30 ms

Table 9.9: CPU requirements of the application classes in Guide-2

We have now established the CPU requirements of the test applications in Guide-2. and must now determine the starting time of applications in Guide-2 and the time it takes to map clusters.

The time needed to start a Guide application is found in a slightly different way from the one used in UNIX, as illustrated in figure 9.6. Instead of iterating a loop as above we vary the number of clusters that are mapped, so that the work and the number of clusters will be proportional. This way we may find the time needed to start an application with linear regression as before.

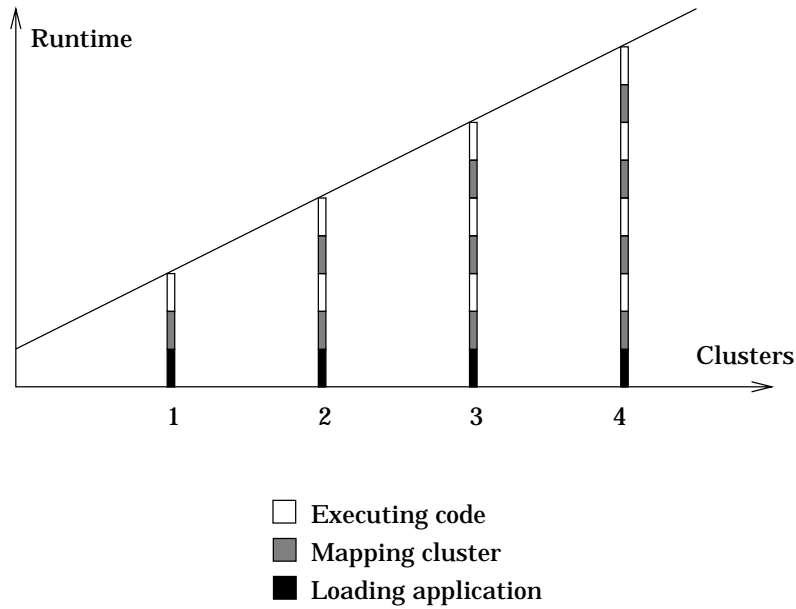


Figure 9.6: Calculating starting cost of Guide application with linear regression

We have executed a simple application with 1, 2, 4, 6, 8, and 10 clusters. Each application was executed 20 times and the run times of these executions are shown in figure 9.7.

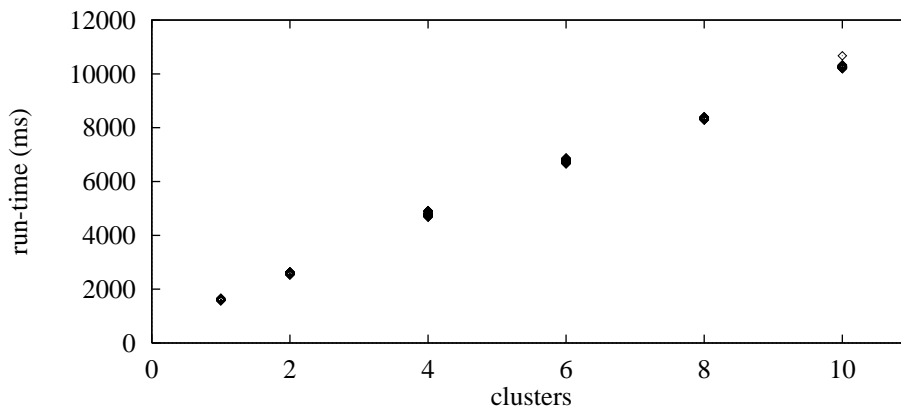


Figure 9.7: Execution times of a Guide application as a function of the number of clusters

These data points are clearly linear so linear regression may be used to find the gradient and the intersection between the line and the y-axis. The time needed to start an application with the Guide-2 application launcher `eliott` is 750.8 ms.

Having found the time needed to start an application in Guide, we may find the time needed to map a cluster using the same method as we used to find the time needed to start a UNIX application as illustrated in figure 9.8.

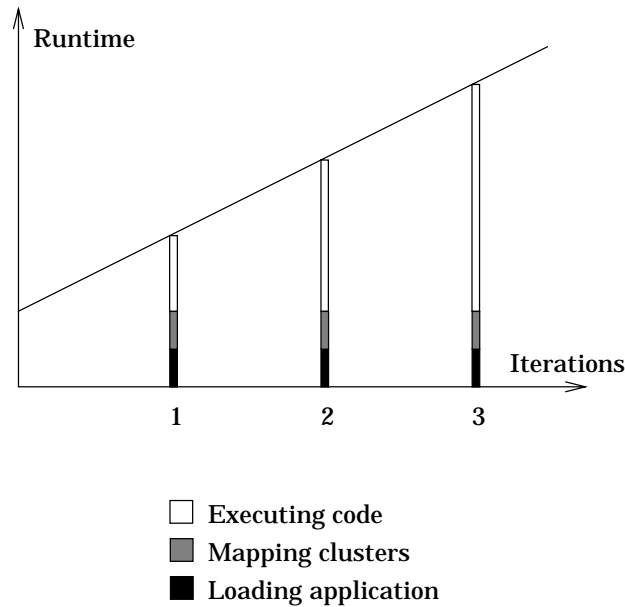


Figure 9.8: Calculating the time needed to map a cluster with linear regression

Using the same number of clusters, we varied the number of iterations in our application, and ran it with 10000, 100000, and 1000000 iterations (three distinct values are the minimum required to establish a line). We ran each application 20 times and the run-times are shown in figure 9.9.

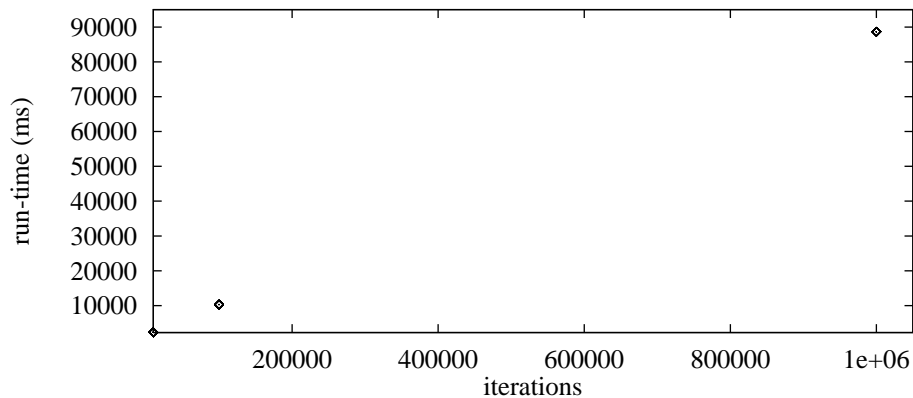


Figure 9.9: Execution times of a Guide application with a fixed number of clusters as a function of the number of iterations

Again the measurements are linear, so linear regression may be used to find the sum of the times needed to load the application and to map the clusters. We find the time needed to start an application and map 10 clusters is 1523.8 ms.

The time needed to map 10 clusters may now be found by subtracting the start-time from this number, and the time needed to map a single cluster by dividing the result by 10. This way we find the time needed to map a single cluster is 77.3 ms.

We have now found all the timing related properties needed to model the run-times of the synthetic applications. These properties are summarized in table 9.10 below.

Property	Time
CPU-time of light-weight applications	11.5 ms
CPU-time of medium-weight applications	305.6 ms
CPU-time of heavy-weight applications	7606.3 ms
Start-time of applications	750.8 ms
Mapping-time of cluster	77.3 ms

Table 9.10: Main properties of synthetic applications

Before we can calculate the run-times of the synthetic applications, their internal structure and the number of clusters must be determined.

Internal Structure of Test Applications We interviewed Jean-Yves Vion-Dury and André Freysinnet, who both have a large experience with Guide applications², who told us to expect the applications to use a fairly low number of clusters (less than 20) and to have a high locality of reference within each cluster.

With these guide lines we have decided that the number of clusters in an application should be proportional with the run-time of that application and that the invocation pattern in general will be balanced trees. Furthermore we decide that the locality of reference within clusters should be absolute, i.e., after the first method in the first object of a cluster is invoked, no other methods of objects in the cluster will be invoked from outside the cluster.

This have lead us to the internal structure of our test applications illustrated in figure 9.10.

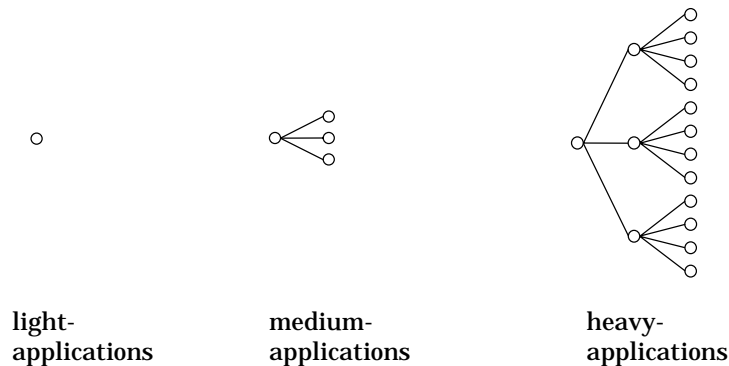


Figure 9.10: Structure of clusters in the test applications

We have now determined all the properties needed to define the synthetic applications. The run-time of an application class is found from the following formula

$$T_{\text{RUN}} = T_{\text{START}} + T_{\text{CPU}} + n \times T_{\text{MAP}}$$

Where T_{RUN} is the run-time of the synthetic application, T_{START} is the start-time of applications, T_{CPU} is the CPU requirement of the application class, n is the number of clusters in the application class, and T_{MAP} is the time needed to map a cluster.

²Jean-Yves has implemented an interactive object browser/debugger for Guide-1, while André has been involved in Guide-1 and implemented the execution machine for Guide-2

The run-times of the synthetic applications calculated from this formula are shown in table 9.11 below.

Application Class	Run-time
Light-Weight Applications	839.6 ms
Medium-Weight Applications	1365.6 ms
Heavy-Weight Applications	9593.9 ms

Table 9.11: Run-times of application classes

Performing Experiments

The three classes of applications designed above was implemented, so that the run-times of the synthetic applications were as close to the desired run-times of the classes as possible. The average run-times of the synthetic applications are shown in table 9.12, along with their standard deviation and the coefficient of variation.

Application Class	Average Run-Time	Standard Deviation	C.O.V.
Light-Weight Applications	836 ms	47.66	0.057
Medium-Weight Applications	1392 ms	55.14	0.040
Heavy-Weight Applications	9604 ms	225.50	0.023

Table 9.12: Run-times of application classes

The difference between the run-times of the test-suite shown in table 9.12 and run-times of the workload model shown in table 9.11 is very small. The difference between the medium class of applications is slightly higher than for the others, but in all cases the differences in run-times are smaller than the standard deviation, normally used to define the variability of sampled data. The variation in the run-times of the test-suite is rather small, as indicated by the small values of the coefficients of variation. We therefore find that the test-suite is a good representation of our workload model.

Much effort has gone into designing the evaluation workload, so that it may be conceived as a typical Guide-2 workload. It makes little sense to introduce variation into this workload, because the workload then no longer reflects our expectations about the typical Guide-2 workload.

We have however introduced a single element of variation, by scaling the arrival interval of the applications from 1 to 5 minutes. This second experiment is performed to investigate the influence of load distribution on a very light workload, as described on page 105.

We first execute the designed test-suite, where the applications arrive at random intervals within one minute. The random pattern of arrival was performed once and for all, so that changes in this pattern does not influence the results.

We have registered the average run-times for the entire test-suite as well as for each of the application classes. These run-times and the speed-ups³ are shown in table 9.13.

³The speed-up is calculated as $\text{speed-up} = \frac{\text{non-distributed runtime}}{\text{distributed runtime}}$.

Application Class	Distributed Run-Time	Speed-Up
All applications	41526 ms	0.88
Light-Weight Applications	1172 ms	0.83
Medium-Weight Applications	5842 ms	0.61
Heavy-Weight Applications	15756 ms	1.03

Table 9.13: Distributed run-times and speed-ups achieved by load distribution on a typical Guide-2 workload

The numbers show a general degradation in performance, except an insignificant improvement for the class of heavy applications. The main reason for this degradation is probably the overhead introduced by the load distribution facility, which is an order of a magnitude larger than the CPU-requirements of the majority of applications (111.8 ms overhead vs. 11.5 ms CPU-requirement for the light applications). The improvement in performance for the heavy applications indicates to us that performance improvements should be possible with a heavier workload.

We now execute the light workload, where the applications arrive with an interval five times the one used above. The distributed run-times and the speed-ups by load distribution are shown in table 9.14 below.

Application Class	Distributed Run-Time	Speed-Up
All Applications	35020 ms	0.92
Light-Weight Applications	1051 ms	0.98
Medium-Weight Applications	1855 ms	0.85
Heavy-Weight Applications	13634 ms	0.85

Table 9.14: Distributed run-times and speed-ups achieved by load distribution on a light Guide-2 workload

The performance was degraded for all application classes in this experiment, which we again attribute to the overhead introduced by the load distribution facility. The degradation of all classes in this experiment, indicates to us that the generated workload was so light that all clusters were mapped locally and that such degradation should probably be expected for all light workloads.

9.5.2 CPU-Intensive Workload Test

The purpose of this experiment is to investigate the performance benefits that may be achieved by a CPU-intensive workload in an unbalanced system. In our environment (with only two nodes) this means that a large CPU-intensive workload should be generated on one node, while the other is left idle.

The main performance parameters of this experiment are run-times of these CPU-intensive applications, their arrival rate, and the number of applications that are executing in parallel.

The size of the applications mainly determines how long the experiment will take, while the number of applications executed in parallel determines the saturation of the CPU and the arrival rate determines how fast this resource is saturated. We have therefore chosen only to make the last parameters into factors in our experiment.

We use the naïve implementation of the Sieve benchmark⁴, developed for our investigation of the importance of the granularity of load distribution.

The experiment is divided into two experiments in the following way. First we perform an experiment where one factor is varied, while the other is fixed. Then we perform a second experiment where the other factor is varied, while the first factor is fixed. This method of evaluation only examines a small part of the entire parameter space, so it is inappropriate for a general evaluation of the performance benefits achieved by distributing large applications, however the purpose of this experiment is only to establish what kind of performance benefits may in fact be achieved under favourable conditions.

First the number of applications started in parallel is varied, while the application arrival rate is one application per minute. This fairly low arrival rate is chosen, because the 5 seconds load average used as load indicator only responds slowly to the change in load. The results of this experiment is shown in table 9.15.

Number of applications	Measured speed-up
3	1.56
6	1.94
9	2.17

Table 9.15: Measured speed-ups for workloads generated by a varying number of applications.

These numbers indicate that the performance benefits of load distribution in Guide-2 increases with an increasing workload. The speed-up is achieved by changing virtual parallelism, where several applications executing in parallel on the same node, to physical parallelism where the applications are physically distributed on the nodes in the system.

The super-linear speed-up achieved with nine applications on two nodes is fairly common in parallel computing where the speed-up generally decreases with an increasing number of nodes. In our case we attribute this effect to saturation of other resources than the CPU, e.g the over-saturation of internal memory is halved when the workload is distributed, so swapping may possibly be avoided.

We now vary the arrival rate of six applications. The arrival rate will have to be fairly low, to allow the load indicator to respond to the increased load. The output from this experiment is shown in table 9.16.

Application arrival rate	Measured speed-up
1 per 30 seconds	1.94
1 per 60 seconds	1.94
1 per 90 seconds	2.04

Table 9.16: Measured speed-ups for workloads generated by a varying arrival rate of applications.

This experiment shows an almost constant speed-up for workloads generated with different arrival rates. Contrary to what should be expected the speed-up is highest for the lightest workload, but the variation in the speed-ups is very low, so we do not attribute this increase any greater significance.

⁴The sieve benchmark calculates primes using Erathostenes formula.

Both experiments performed above indicate that substantial performance benefits may be achieved by load distribution on a workload generated by many large applications. The increase in speed-up when the number of applications is increased and the fairly constant speed-up when the arrival rate is varied, indicate that the number of applications is the most important of the two parameters.

9.5.3 Conclusions and Perspectives

We have performed an evaluation with a workload constructed to resemble an average Guide-2 workload. The construction of this workload was based on information from many different sources, both within and outside the Guide-2 system. This approach is far from optimal, but it is the best we could do in a system with few applications and users.

Our measurements show that load distribution with a typical Guide-2 workload most likely will degrade the performance of the applications. This is mainly because the overhead of our load distribution facility is too large compared to the CPU-requirements of typical applications.

For the purpose of this project we developed a general and flexible load distribution facility that may be used several different places in the system, and tried it on the finest possible granularity of distribution in the Guide-2 system. The cost of generality is often performance, so it may seem paradoxical to use a mechanism with a large overhead in a setting where this overhead hurts the most, but we also wanted the ability to experiment with other types of load distribution, e.g. distribution of activities.

Several ways of improving the performance of the load distribution facility by specializing it to the fine granularity of clusters have been suggested, and it is possible that these improvements are enough to ensure performance benefits by load distribution. We do not feel confident that this is the case, and suggest that more radical measures are considered.

The problem is that the CPU-requirements of the majority of applications are small compared to the overhead of the load distribution facility, and that the large applications are divided into clusters with smaller CPU-requirements. The majority of clusters are likely to have small CPU-requirements, which means that they will suffer from load distribution instead of gaining by it.

We therefore suggest that only large clusters are considered for load distribution, by introducing a filter like the one purposed by Anders Svensson in [Svensson 90]. This filter works by having the run-times (CPU-requirements) of clusters registered in a data base and using this information to decide whether the load distribution facility should be consulted. If we decide that the cluster level is best for load distribution in Guide-2, and integrate the load distribution facility with the storage server, this information about run-times may be stored with the cluster on disk.

The second experiment, where only long running applications are distributed, show very good performance improvements by load distribution, so we believe that an improved load distribution facility that only considers clusters with high CPU-requirements is likely to offer general performance improvements in the Guide-2 system.

9.6 System Evaluation

The system evaluation will mainly be based on an experiment and observations done either in chapter 5 or during our work with the load distribution facility.

We focus on a single issue in Guide application design, namely the granularity of distribution. The granularity of distribution in Guide-2 is the cluster, so we must examine the

influence of different granularities on the performance of load distribution and determine how clusters should be managed to ensure good performance of load distribution.

The granularity of load distribution has been considered more or less explicitly by previous research in object-oriented systems [Dickman 91, Dickman 92b, Andersen 92a], however none of this research has provided measurements that may indicate to the developers of object-oriented systems and applications which granularity to aim for.

The purpose of this experiment is to provide a first insight into this important design issue.

We focus on sequential applications, a class of applications that are fairly common in administrative data processing such as banking (calculating and crediting interests to accounts), text processing (spell-checking and global word substitution in large documents), all forms of lexical analysis, and many other areas of data processing.

9.6.1 Designing the Experiment

The goal of this experiment is to determine the influence of the granularity of load distribution, when large sequential applications are subject to load distribution.

The system under test in this experiment is the same as the one defined in 9.5.1.

In order to design a test application, we need to identify the parameters that determines the result of this experiment. These are:

1. The arrival rate of applications at a node
2. The run-time of applications
3. The size of the working-set of the applications
4. The sizes and run-times of the clusters in the applications

The arrival rate of applications determine the load on the system, so it must be a factor of the experiment that is varied to generate different workloads.

The run-time of the applications should be large enough to allow us to experiment with a wide variety of different cluster sizes. We simplify the workload generation by only considering one large type of application, this way the arrival rate alone determines the load on the system.

The size of the applications working set determines the saturation of main memory, which implies swapping and paging that may interfere with our measurements if it does not happen consistently, so we fix this parameter throughout the experiment by defining applications large enough to ensure memory saturation under all circumstances.

The sizes of the clusters within the application is the primary factor of this experiment. This factor is varied by implementing the same application with a different number of equal sized clusters.

9.6.2 Performing the Experiment

We have implemented a simple artificial application with a few desirable properties to facilitate this investigation. The application is strictly sequential accessing each element in its large working set one time only. It has been designed so that it may be easily divided into smaller units of equal size, without interfering with the overall amount of work done by the application.

Since there is no parallelism within the application, there is no inherent need for load distribution in the application. The need for load distribution arises when several applications are started in parallel.

We have performed a series of three experiments where the workload is varied by increasing the number of applications that arrive within a one minute interval⁵. This series of experiments is followed by an experiment, where the workload is increased by having the applications arrive more frequently, i.e. one every 30 seconds. The purpose of this last experiment is to confirm the results of the first series of experiments, when another parameter (the application arrival rate) is used to vary the workload.

For each experiment we measure the average run-time of the applications (wall clock time), first without load distribution and then with the load distribution facility enabled. The measured run-times may be used to calculate the speed-up as follows:

$$\text{speed-up} = \frac{\text{non-distributed}}{\text{distributed}}$$

We report the calculated speed-ups along with the run-times of the applications executed with load distribution.

In our first experiment we start 3 applications — all alike — with a one minute interval. This is repeated with the applications divided into 1, 2, 4, 8, and 16 equal sized clusters. The results of this experiment are shown in table 9.17 below.

Number of clusters	Distributed run-time (ms)	Speed-Up
1	286405	1.48
2	285256	1.56
4	379736	1.24
8	350044	1.35
16	367578	1.26

Table 9.17: Run-times and Speed-ups for three applications started with a one minute interval

The numbers show a weak tendency for the speed-up to decrease when the granularity gets finer. They also indicate that the distributed runtime of the applications increases with fine-grained load distribution. Last but not least the speed-up with two clusters is the best (and the run-times are shorter) so it appears that keeping the entire application in one cluster is too coarse-grained. It is interesting to note that the applications with four clusters had the longest run-times and the lowest speed-up. We believe that this is a feature of the application and the specific workload, and that such anomalies should be expected for all applications with some specific workload, i.e., for each application exists a workload, where this application will behave poorly.

The second experiment is similar to the first, except the workload is increased by starting six applications with a one minute interval. The results of this experiment are shown in table 9.18 below.

⁵The 5 second average used as load indicator responds slowly to the changes in load, so the time between starting the application has to be rather long (30–60 seconds).

Number of clusters	Distributed run-time (ms)	Speed-Up
1	521066	1.83
2	428749	1.94
4	480083	1.83
8	542929	1.71
16	565089	1.50

Table 9.18: Run-times and Speed-ups for six applications started with a one minute interval

This experiment confirms the observations from the first experiment, except there are no drop in performance for applications with four clusters. We can also see that the benefit of offloading work increases when the load increases (not really that surprising).

The third experiment uses a higher workload with 9 applications started with a one minute interval. The results of this experiment are shown in table 9.19 below.

Number of clusters	Distributed run-time (ms)	Speed-Up
1	541900	2.13
2	558815	2.17
4	665471	1.99
8	762099	1.80
16	853883	1.47

Table 9.19: Run-times and Speed-ups for nine applications started with a one minute interval

Here the one cluster application has the shortest run-time while the application with two clusters has the best speed-up. We have no good explanation for this, but the overall tendency of the experiment shows the same increase in run-times and the same decrease in speed-ups as the first two.

The last experiment is performed to see if the same tendencies appear when the workload is increased by shortening the delay between applications. In this experiment six applications are started with a 30 seconds interval, and the results are shown in table 9.20 below.

Number of clusters	Distributed run-time (ms)	Speed-Up
1	559244	1.67
2	507851	1.94
4	561930	1.82
8	717747	1.54
16	725408	1.38

Table 9.20: Run-times and Speed-ups for six applications started with a 30 seconds interval

This experiment shows exactly the same behaviour as the first series, except for the anomaly with four clusters, so we have not continued the investigation any further in this direction.

The speed-ups from the experiments above are illustrated by figure 9.11⁶.

⁶It may seem intuitive that these speed-ups are continuous, but our experiments are much too limited to support this claim, so the curves should only be used for visualization.

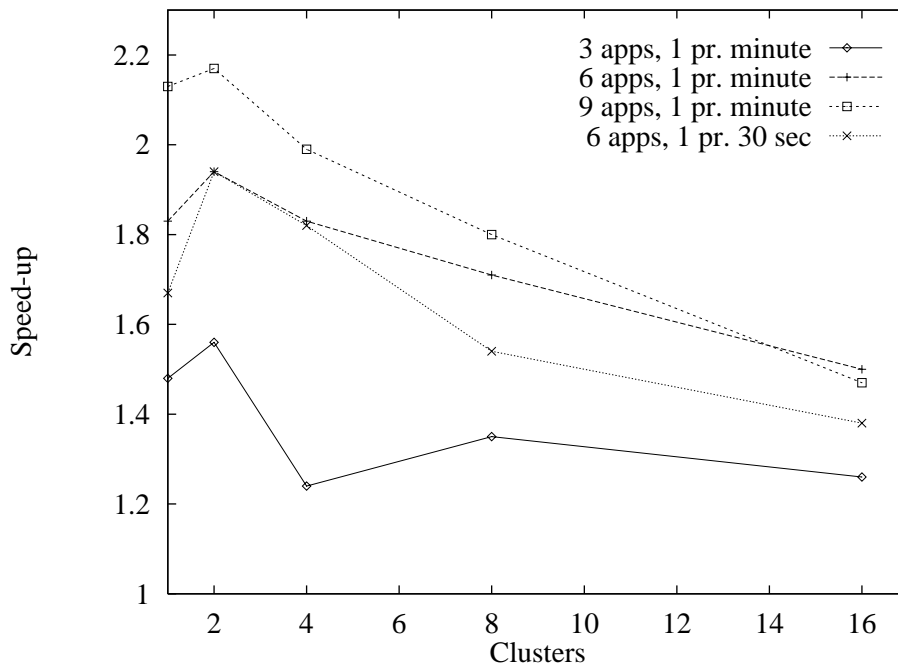


Figure 9.11: Speed-ups achieved by load distribution

It is not enough to focus on the speed-up alone, the actual run-time of the distributed application should also be considered. These run-times are illustrated in figure 9.12.

The numbers illustrated by figure 9.11 show that the performance benefits generally decrease when the granularity of load distribution becomes finer, although all applications have the largest speed-up with two clusters instead of one.

Speed-up is not everything, we also have to consider the run-times of the applications illustrated by figure 9.12. These run-times show the same tendencies for the performance improvements as the speed-ups, i.e., the distributed run-times generally increases when the granularity becomes finer.

All experiments have the largest speed-up with two clusters, and all except the workload with 9 applications also have the shortest runtime with two clusters. This means that the system overhead incurred by clusterization has been amortized by load distribution, and that the granularity of the 2-cluster application is better suited for load distribution than the entire application used by the 1-cluster application. This is a consequence of the initial placement strategy implemented by our load distribution facility that only allows load distribution to be performed when the clusters are initially mapped. In the experiment with one cluster this means that load distribution were possible each time an application was launched (i.e., 6 times), while load distribution also was possible half way through the application when two clusters were used.

Our observations indicate that the coarser granularities of load distribution are most likely to offer the best performance with load distribution, but also that the granularity can become too coarse so that load distribution is rarely done.

Future experiments, using a mixed set of applications instead of a single heavy-weight application, will enable us to verify these results for more common usage patterns. Increasing the number of workstations should allow us to determine the relation between the optimal number of clusters and the number of active workstations.

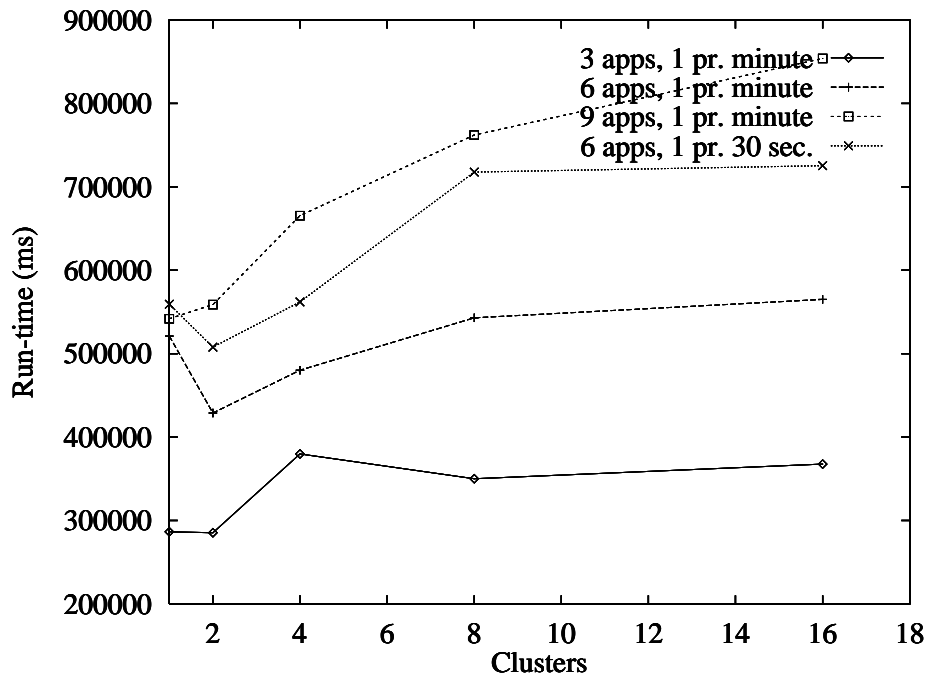


Figure 9.12: Run-times of applications with different cluster sizes

9.6.3 Implications of the Experiment

The experiment with different granularities of load distribution in sequential applications indicated that coarse granularity (few clusters) offers the best performance for load distribution. In this experiment fine-grained load distribution is not penalized by remote invocations, because the locality of reference (within the work space) is high in sequential applications, so we expect the performance of fine-grained applications to degrade when the object access pattern becomes more random, i.e. with less regular sequential applications. Furthermore, and regardless of load distribution, large clusters act as object pre-fetch that also improves the performance of coarse-grained applications.

The importance of the granularity of distribution raises the question of how clusters are managed, so that a good granularity for load distribution is achieved.

Explicit Management of Clusters

In the current implementation clusters are invisible to the application programmer. The default policy of creating objects in the current cluster, gives the programmer some control over the way clusters are managed, and although this control is sufficient for most purposes it is not very intuitive and requires knowledge about the implementation from the application programmer.

The question is now whether this storage transparency (invisibility of clusters) should be broken, by introducing a set of tools to manage the clusters, or it should be maintained by adding functionality to the Guide-2 system that optimizes clusterization.

Locality of Reference

References that are kept within the same context (the same address space on the same machine) are called local references, and the locality of reference is an important performance

parameter in a distributed system because local objects normally are invoked faster than remote objects. Locality of reference is guaranteed among objects in the same cluster, so large clusters generally implies high locality of reference

Parallel programming

parallel activities are possible within the same application (a Guide job), through the `co_begin ... co_end` construction in the Guide language.

It is important that objects invoked in parallel reside in different clusters, so that the expressed parallelism actually becomes physical parallelism. If one or more objects reside in the same cluster, the activities invoking methods in these objects will have to diffuse to the same node where the cluster is mapped. This may imply that a finer granularity should be used, so that physical distribution becomes possible.

Supporting parallel applications may thus lead to a conflict of interests where large clusters are wanted for the high locality of reference, while small clusters may be wanted to allow physical distribution.

This suggest the use of co-locators and contra-locators as defined by Bellerophon described in 4.2 where object interactions are co-locators and parallel invocations are contra-locators. Another, and simpler, possibility is to make clustering explicit and leave it up to the application programmer as it is done in Amadeus. The latter approach has the further advantage of not requiring object mobility, which is currently not supported by Guide-2.

9.6.4 Conclusions and Perspectives

We have shown that the granularity of load distribution is very important for the performance benefits offered by load distribution. This granularity is defined by the clusters in Guide-2 and different issues in the management of clusters have been considered.

Explicit clustering of objects break the transparency of the storage structures and should preferably be avoided. However, the price of maintaining this transparency is the implementation of object mobility among clusters, a mechanism for gathering statistics about object interactions and parallelism, and a mechanism that clusters objects according to these statistics. Explicit clustering is much simpler, so we recommend this, until such time may be available as to implement the support needed for automatic clustering.

We suggest a solution like the one described for Amadeus in 4.3, where the Guide language (and the underlying Guide-2 system) supports constructions to manage clusters. These constructions should allow the application programmer to create a new clusters, and to change the current cluster.

9.7 Summary of Conclusions

We have developed a load distribution facility that is general, extendible, configurable, do not require modifications to neither the kernel nor the applications, and consumes a limited amount of system resources. Unfortunately the price of generality has been a large overhead in invoking the load distribution facility.

We have shown that our load distribution facility is unlikely to improve the performance of an average workload in Guide-2 and that this inability to improve performance is largely caused by the overhead of load distributing small applications. Load distribution may however significantly improve the performance of large, long running applications.

We have also shown that the granularity of load distribution is an important performance parameter, and that coarse grained load distribution most likely will offer the best performance benefits.

Chapter 10

Conclusions

This chapter presents the conclusions of this thesis, and evaluates whether or not the goals stated in 1.3 are achieved.

These goals include an examination of fine-grained load distribution evaluated in 10.1, an implementation of load distribution in Guide-2 evaluated in 10.2, and a set of hypotheses about the benefits of load distribution in fine-grained systems evaluated in 10.4. Future work with fine-grained load distribution is proposed in 10.4.4, and a summary of this chapter may be found in 10.5.

10.1 Fine-Grained Load Distribution

Our examination of object based systems shows that most of the techniques from load distribution in traditional process based operating systems are directly applicable in object based systems. However, important differences exist between process based and object based operating systems which raise a number of new issues.

We identify three issues that must be addressed when an object-oriented load distribution facility is designed. The first issue is the scale of the load distribution problem, which increases dramatically because most applications require a large number of objects. The second issue is the generally smaller grain-size of objects, which means that the expected performance benefit by relocation of each object is small. The last issue has to do with the interaction among objects in the same application, which means that some of these objects should preferably remain co-located to ensure locality of reference.

All of these issues relate to the finer granularity of distribution in typical object based systems. We therefore believe this fine granularity of distribution to be the key difference between object based systems and traditional process based load distribution.

10.2 Guide-2 Load Distribution

We have designed and implemented a load distribution facility in the object based system Guide-2 that allows us to experiment with different aspects of fine-grained load distribution.

Our experiment with an average workload is based on run-time measurements of more than 18,000,000 executions of many different applications in the UNIX environment at the Department of Computer Science, University of Copenhagen. This experiment shows that load distribution may degrade performance of applications with short run-times when the overall load in the system is light to moderate. Applications with longer run-times show

no difference in performance under the same circumstances. We attribute a large part of this degradation in performance to the overhead of the load distribution facility and the distribution of clusters with very short run-times, so although we cannot prove it, we still believe that performance benefits are possible if these problems are resolved.

An experiment where large sequential applications were used to investigate the importance of the granularity of load distribution shows that coarse-grained applications perform better than fine-grained applications, but that the granularity of load distribution must be fine enough for the load distribution facility to exploit the possibilities for parallelism, i.e., we should not necessarily opt for the most coarse-grained unit of load distribution.

Our evaluation also shows that the time required to map a cluster more than doubles when load distribution is performed, i.e., our load distribution facility is rather heavy. This explains the degradation in performance in our average work load experiment. Most of this increased overhead can be explained by the probe call to the storage server, which is required to make the reservation for the cluster on the target node.

It is therefore natural to experiment with a centralized load distribution scheme, where the load distribution facility is integrated with the storage server of Guide-2 thus avoiding the probe call completely. This is possible in the current implementation of Guide-2, which has a centralized storage server, but we do not know whether or not this will be possible in the forthcoming distributed version of the storage server.

The current implementation of the load distribution facility is based on clusters of objects, which is the finest granularity of distribution in Guide-2. However, the external server of the load distribution facility could also be used in the distribution of jobs and activities without modification.

10.3 The Experiment

One of the goals of this thesis defined in 1.3 is the confirmation or denial of the following hypotheses:

1. Load distribution offers general performance benefits in a distributed system with a fine-grained unit of distribution, such as the Guide-2 system.
2. Load distribution may significantly improve the performance of some applications, when the load on the nodes in the system is unbalanced.
3. The granularity of distribution is important for the size of these performance benefits.

Our experiments neither confirm nor deny the first hypothesis. The average workload experiment supports a denial of the hypothesis, but our evaluation of the load distribution facility shows that the overhead is rather high, which means that performance benefits from fine-grained load distribution are unlikely. In 10.4.4 two methods to decrease the overhead of the load distribution facility are proposed, which should be implemented and the experiments repeated in order to reevaluate the hypothesis. However, our analysis of fine-grained load distribution in chapters 3–5 and the granularity experiment indicate that the performance benefits of fine-grained load distribution is likely to be smaller than the benefits of load distribution in coarse-grained systems.

The second hypothesis is confirmed by our experiment with granularity, where CPU-intensive sequential applications with a long run-time and the optimal granularity of load distribution experience super linear speed-up by load distribution using two nodes. We do not expect all longer running applications to experience the same speed-up, but the

experiment shows that substantial performance benefits are possible for an important class of applications.

The third hypothesis is also confirmed by our experiment with granularity, which shows a generally improved performance when the granularity becomes more coarse-grained (e.g., the experiment with nine applications shows a range in speed-ups from 1.47 to 2.17, depending on the granularity of load distribution).

10.4 Future Work

We have touched upon several topics of load distribution in this thesis. In the following we describe some of the natural extensions to this work.

10.4.1 Further Experiments

The inability to confirm or deny hypothesis 1, indicates that further experiments are necessary. Apart from implementing the changes proposed in 10.4.4 and reevaluating the average workload experiment, experiments with different workloads may help confirm or deny the hypothesis.

10.4.2 Load Indicators

A new paradigm for load indicators is presented in chapter 2, where each load indicator is associated with the system component it describes (e.g., the length of the run-queue indicates the load of the CPU). These indicators were divided into hardware load indicators (indicators of the available resources for the system component) and software load indicators (indicators of the activity involving the system component). We then formulated the principle of minimum span for the construction of combined load indicators, i.e., that each system component is only measured by one load indicator.

We propose to use this paradigm in an evaluation of the respective merits of these load indicators (e.g., repeating the simulations presented by Kunz in our context). We would also like to evaluate the benefits of combined load indicators that are constructed from the minimum span principle proposed in this thesis.

The evaluation of load indicators is not a direct extension of this work, and may be performed in another system or by simulations using a queueing network model.

10.4.3 Fine-Grained Load Distribution

Our experiment with the granularity of load distribution focuses on the size of clusters in large sequential applications. We here assume that clusters can be build so that most of the object references are resolved within the cluster.

One of the reasons for implementing operating systems based on objects is that objects provide a convenient abstraction for hiding the location of data and code from the application programmer. An investigation of clustering and how clusters may support object based applications is a natural extension to this work.

Clustering of objects raises a number of other problems that should also be investigated: what objects should be put in the same cluster, how are these clusters constructed (by the users or the system), what size should clusters typically have (this is an extension of the granularity experiment presented in this thesis), should clustering be static or should objects be allowed to regroup into new clusters, and what are the performance implications of inter cluster references.

We believe that the granularity of load distribution is the key issue in object-oriented load distribution, and that clustering of objects is a good way to limit the overhead of the load distribution facility and ensure performance benefits from load distribution. It is therefore important to examine all the questions above, so that a good way of clustering objects may be found.

10.4.4 Further Work on the Guide-2 Load Distribution Facility

The work with the existing Guide-2 load distribution facility may continue along two separate lines: evolving and optimizing the existing load distribution scheme based on clusters, or changing the load distribution facility to distribute activities or jobs.

We believe that a substantial improvement in performance may be achieved by the following changes to the existing load distribution facility. Changing the load distribution scheme from GLOBAL to CENTRAL prevents the extra probe call to the storage server, thereby decreasing the overhead of the load distribution facility. This overhead may be decreased even more by moving the initiation mechanism from the external server to the elvis library (e.g., by keeping the load indicators in a memory segment shared by all applications), thereby avoiding the overhead of always invoking the load distribution facility.

We have seen that large clusters, in sequential applications, have the largest speed-up, so implementing a filter that discriminates against clusters with short run-times that prevents the overhead of remote invocation to exceed the run-time of the cluster should also improve performance.

So far we have focused on fine granularity. However, it is possible that distribution of activities, or jobs may improve the performance of the load distribution facility. This supposition is supported by our experiment with granularity, which showed generally better performance with coarse-grained load distribution. These coarser granularities of load distribution should therefore be considered in the industrial version of Guide.

10.5 Summary

We have achieved all of the goals stated in chapter 1.3 except the confirmation or denial of the hypothesis that load distribution may offer general performance benefits in fine-grained object based systems. We have proposed modifications to the existing load distribution facility, that should allow us to reevaluate this hypothesis.

We have also proposed several areas of future work, apart from the reevaluation described above, that build on the work presented in this thesis.

Appendix A

Elvis Configuration Guide

Elvis has been designed with emphasis on flexibility and is highly configurable. The different configuration options are explained below and a sample configuration file is given in appendix A.2.

A.1 Configuration Parameters

masterNode The masterNode identifies the host that serves as the central relay for load information. This host is identified by its hostname or its Elliott node number from the 'nodes' configuration file, e.g. the host guam can be specified either 'guam' or '4'.

loadUpdateFrequency The loadUpdateFrequency indicates the time in seconds, between examinations of the local load. If the load is updated periodically this is also the period used for updates of the central server.

lowWaterMark The lowWaterMark is the threshold between underloaded and normally loaded nodes. The unit is the same as the `avenrun` value returned by the Mach 3.0 `host_info()` call, i.e., the average length of the run-queue during the last 5 seconds multiplied by 1000.

highWaterMark The highWaterMark defines the threshold between normally loaded and overloaded hosts. It uses the same unit as the lowWaterMark above.

risingLoadUpdate The risingLoadUpdate is used by the adaptive load update scheme, to specify how many percent the load should increase before the central server is updated.

fallingLoadUpdate The fallingLoadUpdate serves the same function as the risingLoadUpdate, but for falling load. This has been introduced for the following reason: If the change in load from 2000 to 4000 is significant, it will be an increase of 100% when the load is rising, but only a drop in load by 50% when the load is falling.

loadUpdateThreshold The loadUpdateThreshold is also used by the adaptive load update scheme, to suppress updates when the load is low. A rise in load by 1000% from 33 to 330 is insignificant because the load of the host is still very low.

mapLocal The mapLocal parameter tells how many clusters should be mapped locally if the host is underloaded, before remote mapping is forced.

loadUpdatePolicy The loadUpdatePolicy indicates which information exchange policy elvis should employ. The currently recognized values are:

NO_UPDATE which is not currently used, but it has been included so static load distribution schemes may be implemented on top of the current implementation.

PERIODIC_UPDATE indicates that periodic updates of the central server should be performed.

VARIABLE_UPDATE indicates that the central server should be updated whenever significant changes in the load has occurred. This is also referred to as the adaptive load update scheme.

The default information exchange policy is periodic updates.

systemStatePolicy The systemStatePolicy describes how system state is maintained in the system. The currently recognized values are:

NO_STATE means that no state is kept in the system. This means that the location policy may either be NO_LOAD_BALANCING or RANDOM.

GLOBAL_STATE means that state is kept on every node in the system. Load distribution is performed based on the local copy of the load vector.

CENTRAL_STATE means that system state is kept on one node only. This implies that load distribution should query this node to perform the location policy.

The default system state policy is global state.

nodeLocationPolicy The nodeLocationPolicy describes the location policy. The currently recognized values are:

NO_LOAD_BALANCING always chooses the local node.

RANDOM chooses a node at random.

ROUND_ROBIN locates the node based on a round robin scheme on the least loaded category of nodes (i.e., underloaded nodes when they are available otherwise normally loaded nodes).

The default location policy is round robin.

A.2 Sample Configuration File

This configuration file was used in our experiments with the load distribution facility. The character '#' means that the rest of the line is a comment. The system load is specified as the Mach 3.0 avenrun parameter (i.e., similar to the UNIX load multiplied by 1000).

```

#
# Configuration file for elvis
#

# name of the node where the LIC is running
masterNode                morane

# load indicators are gathered every 5 seconds
loadUpdateFrequency       5

# threshold for underloaded nodes
lowWaterMark              800

# threshold for overloaded nodes
highWaterMark             2400

# the LIC should be updated when the load increases by 25%
# this parameter is not used because of the periodic update
risingLoadUpdate          25

# the LIC should be updated when the load drops by 15%
# this parameter is not used because of the periodic update
fallingLoadUpdate         15

# the LIC should not be updated as long as the load is
# under 300
# this parameter is not used because of the periodic update
loadUpdateThreshold       300

# four clusters may be mapped locally during an update period
# before load distribution becomes mandatory
mapLocal                  4

# load is updated periodically
loadUpdatePolicy          PERIODIC_UPDATE

# system state is kept on all nodes
systemStatePolicy         GLOBAL_STATE

# the loacation policy should be based on a round robin
# algorithm on the least loaded category of nodes
nodeLocationPolicy        ROUND_ROBIN

#
# End of configuration file
#

```


Bibliography

- [Andersen 91a] Birger Andersen: “Ellie Language Definition Report”. *Ph.D. Thesis (partial), DIKU Report no. 91/3, Department of Computer Science, University of Copenhagen, Denmark, June 1991 (2nd ed.)*. Also appeared in *ACM SIGPLAN Notices, Vol. 25, No. 11, pp. 45–64, November 1990 (1st ed.)*.
- [Andersen 91b] Birger Andersen: “Grain-Size Adaption in the Fine-Grained Object-Oriented Language Ellie”. *Ph.D. Thesis (partial), DIKU Report no. 91/5, Department of Computer Science, University of Copenhagen, Denmark, November 1992*.
- [Andersen 92a] Birger Andersen: “Load Balancing in the Fine-Grained Language Ellie”. *Presented at the ECOOP '92 Workshop on Load Balancing in Object Oriented Systems, June 1992*.
- [Andersen 92b] Birger Andersen: “Load Balancing in the Fine-Grained Object-Oriented Language Ellie”. *Proceedings of the Second International Workshop on Object Orientation in Operating Systems, IEEE Proceedings, September 1992*.
- [Cahill 93] V. Cahill, R. Balter, X. Rousset de Pina and N. Harris (Editors): “The Comandos Distributed Application Platform”. *Springer-Verlag 1993*.
- [Casavant 88] Thomas L. Casavant & Jon G. Kuhl: “A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems”. *IEEE Transactions on Software Engineering, vol. 14, No. 2, February 1988*.
- [Cayuela 92] J. Cayuela, P. Y. Chevalier, A. Duda, A. Freyssinet, D. Hagimont, S. Lacourte, P. Ledot, M. Riveill, and X. Rousset: “La Machine Guide-2”. *Internal working document K16.archi, Bull-IMAG, February 1992*.
- [Chevalier 93] P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte, X. Rousset de Pina: “Experience with Shared Object Support in the Guide System”. *Proceedings of the fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS), San Diego, California, September 1993*.
- [Chevalier 94] Pierre-Yves Chevalier: “Persistance et disponibilité dans les systèmes répartis: application à Guide”. *Ph.D. Thesis, Univesité Joseph Fourier – Grenoble I, October 1994*.
- [Christaller 91] Michel Christaller: “Mécanismes et Politiques de Répartition de Charge: Application à GUIDE”. *Rapport du DEA Informatique, Unité Mixte Bull-IMAG/Systèmes, June 1991*.

- [Dickman 91] Peter Dickman: “Effective Load balancing in a Distributed Object-Support Operating System”. *Proceedings of the International Workshop on Object-Oriented Systems, pp. 147–153, IEEE 1991.*
- [Dickman 92a] Peter Dickman: “Distributed Object Management in a Non-Small Graph of Autonomous networks with Few Failures”. *Ph.D. Thesis, University of Cambridge Computer Laboratory, 1992.*
- [Dickman 92b] Peter Dickman: “Matching Dynamic Object Clumps to Current Load Imbalances: A Proposal”. *Presented at the ECOOP '92 Workshop on Load Balancing in Object Oriented Systems, June 1992.*
- [Douglis 91] Fred Douglis & John Ousterhout: “Transparent Process Migration: Design Alternatives and the Sprite Implementation”. *Software – Practice and Experience, Vol. 21, No. 8, pp. 757–785, August 1991.*
- [DSG 91] Distributed Systems Group: “Overview of the Amadeus Project”. *Distributed Systems Group, Dept. of Computer Science, Trinity College Dublin, Technical Report A390, May 1991.*
- [Eager 86] Derek L. Eager, Edward D. Lazowska, and John Zahorjan: “Adaptive Load Sharing in Homogeneous Distributed Systems”. *IEEE Transactions on Software Engineering, Vol. 12, No. 5, May 1986.*
- [Eager 88] Derek L. Eager, Edward D. Lazowska, and John Zahorjan: “The Limited Performance Benefits of Migrating Active Processes for Load Sharing”. *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems, pp. 63–72, Santa Fe, New Mexico.*
- [Ferrari 86] Domenico Ferrari & Songnian Zhou: “A LOAD INDEX FOR DYNAMIC LOAD BALANCING”. *Proceedings of 1986 Fall Joint Computer Conference, Dallas, Texas*
- [Freyssinet 91] André Freyssinet: “Architecture et Réalisation d’un Système Réparti à Objets”. *Ph.D. Thesis, Université Joseph Fourier – Grenoble I, September 1991.*
- [Goscinski 91] Andrzej Goscinski: “Distributed Operating Systems The Logical Design”. *Addison-Wesley 1991*
- [Hutchinson 87] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul: “The Emerald Programming Language Report”. *Technical Report 87-10-07, Dept. of Computer Science, University of Washington, Seattle, October 1987.*
- [Jain 91] Raj Jain: “The art of Computer Systems Performance Analysis”. *Wiley 1991*
- [Jensen 93] Christian D. Jensen & Allan K. Schougaard: “Indføring af kommunikationstid som afstandsmål mellem datamaskiner på et netværk”. *Department of Computer Science, University of Copenhagen. – in Danish.*

- [Jensen 95] Christian D. Jensen: “An Investigation of Grain Sizes of Load Distribution in an Object Based System”. *Proceedings of Journées de Recherche sur Le Placement Dynamique et la Répartition de Charge : Application aux Systèmes Répartis et Parallèles, Paris, May 1995.*
- [Jul 88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black: “Fine-Grained Mobility in the Emerald System”. *ACM Transactions on Computer Systems, Vol. 6, No. 1, pp. 109-133, February 1988.*
- [Jul 89] Eric Jul: “Object Mobility in a Distributed Object-Oriented System”. *Ph.D. dissertation, DIKU Rapport 89/1, Department of Computer Science, University of Copenhagen.*
- [Krakowiak 90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, “Design and Implementation of an object-oriented, strongly typed language for distributed applications”. *Journal of Object-Oriented Programming (JOOP), Vol. 3, No. 3, pp. 11-22, September/October 1990.*
- [Krakowiak 92a] S. Krakowiak: “Spécifications d’objectifs et orientation générales pour le système Guide-2”. *Rapport Bull-IMAG 12-92, Bull-IMAG, Grenoble, June 1992.*
- [Krakowiak 92b] S. Krakowiak & X. Rousset de Pina: “Architecture du système Guide-2 Résumé des choix de conception”. *Rapport Bull-IMAG 13-92, Bull-IMAG, Grenoble, June 1992.*
- [Krakowiak 93] S. Krakowiak: “Issues in Object-Oriented Distributed Systems” (invited paper). *International Conference on Decentralized and Distributed Systems, (IFIP WG 10.3), Palma de Mallorca, September 1993.*
- [Kunz 91] Thomas Kunz: “The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme”. *IEEE Transactions on Software Engineering, Vol. 17, No. 7, July 1991.*
- [Lacourte 93] Serge Lacourte: “Interface de la machine virtuelle Eliott”. *Internal working document K30.imv-eliott, Bull-IMAG, February 1992.*
- [Lehrmann 93] Morten Lehrmann: “Load Distribution in Emerald: an Experiment”. *Masters Thesis, Department of Computer Science, University of Copenhagen, June 1993.*
- [Litzkow 87] M. J. Litzkow, M. Livny, and M. W. Mutka: “Condor: A Hunter of Idle Workstations”. *Computer Science Technical Report, No. 730, Computer Science Department, University of Wisconsin-Madison.*
- [Loepere 93a] Keith Loepere, Editor: “OSF Mach Kernel Principles”. *Open Software Foundation, Inc. & Carnegie Mellon University, 1993.*
- [Loepere 93b] Keith Loepere, Editor: “OSF Mach Kernel Interfaces”. *Open Software Foundation, Inc. & Carnegie Mellon University, 1993.*

- [Lunati 88] Jean-Michel Lunati: “Migration de processus et partage de charge dans les systèmes repartis — Application au système Guide”. *Rapport du DEA Informatique, Institut National Polytechnique de Grenoble, Laboratoire de Génie Informatique September 1988.*
- [Mutka 92] Matt W. Mutka: “Estimating Capacity For Sharing in a Privately Owned Workstation Environment”. *IEEE Transactions on Software Engineering, vol. 18, No. 4, April 1992.*
- [Nguyen Van90] Hiep Nguyen Van, Michel Riveill, and Cécile Roisin: “Manuel du Langage Guide (V1.5)”. *Rapport Bull-IMAG 3-90, Bull-IMAG, Grenoble, December 1990.*
- [Nichols 87] David A. Nichols: “Using Idle Workstations in a Shared Computing Environment”. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles, Austin, Texas, pp. 5-12.*
- [Organick 72] E. I. Organick: “The Multics system: an examination of its structure”. *MIT Press 1972.*
- [Osser 92] William Osser: “Automatic Process Selection for Load Balancing”. *University of California, Santa Cruz, Master Thesis, June 1992.*
- [Ousterhout 90] John K. Ousterhout: “Why Aren’t Operating Systems Getting Faster as Fast as Hardware”. *Proceedings of USENIX Summer Conference 1990, Anaheim, California.*
- [Raverdy 95] Pierre-Guillaume Raverdy & Bertil Folliot: “Presentation of the Execution Territory: a Two Levels Load Balancing Mechanism”. *Proceedings of European Research Seminar on Advances in Distributed Systems (ERSADS), Grenoble — l’Alpe d’Huez, April 1995.*
- [Svensson 90] Anders Svensson: “History, an Intelligent Load Sharing Filter”. *Proceedings of the 10. International Conference on Distributed Computing Systems, pp. 546—553, 1990.*
- [Talbi 95] E. G. Talbi: “Régulation de charge dans les systèmes distribués at parallèles”. *Proceedings of Journées de Recherche sur Le Placement Dynamique et la Répartition de Charge : Application aux Systèmes Répartis et Parallèles, Paris, May 1995.*
- [Tangney 91] Brendan Tangney & Annrai O’Toole: “An Overview of Load Balancing in Amadeus”. *Proceedings of the ISMM Conference on Parallel and Distributed Computing and Systems, Washington, October 1991.*
- [Tangney 92] Brendan Tangney & Andrew Condon: “Some Issues in Load Balancing in Amadeus”. *Presented at the ECOOP ’92 Workshop on Load Balancing in Object Oriented Systems.*
- [Tanenbaum 92] Andrew S. Tanenbaum: “Modern Operating Systems”. *Prentice-Hall 1992, pp. 84-87.*

- [Theimer 85] Marvin M. Theimer, Keith A. Lantz and David R. Cheriton: “Pre-emptable Remote Execution Facilities for the V-System”. *Proceedings of the 10th ACM Symposium on Operating System Principles (1985)*, *Operating Systems Review*, Vol. 19, No. 5, pp. 2-12.
- [Wang 85] Yung-Terng Wang & Robert J. T. Morris: “Load Sharing in Distributed Systems”. *IEEE Transactions on Computers*, Vol. c-34, No. 3, March 1985
- [Zayas 87] Edward R. Zayas: “Attacking the Process Migration Bottleneck”. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles, Austin, Texas*, pp. 13-24.
- [Zhou 88] Songnian Zhou: “A Trace-Driven Simulation Study of Dynamic Load Balancing”. *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, September 1988.
- [Zhou 91] S. Zhou, X. Zheng, J. Wang and P. Delisle: “Utopia: A load sharing system for large, heterogeneous distributed computer systems”. *Technical Report CSRI-257, University of Toronto, November 1991*.