Performance Monitoring in Service-Based Computing

PhD Thesis

Doris Ressmann

Software Technology Research Laboratory
De Montfort University

2003

Abstract

Service-based computing is a new computing paradigm where computing is seen as a utility, similar to gas, electricity, etc. An application is not physically shipped to the customer but made available for remote usage. A service-based application consists of services executed from all over the world on the machines where they are installed. This brings a number of design problems into being compared to traditional local execution. This thesis addresses some of these including naming, searching, accounting, binding different services to form new services and finally guaranteeing the quality of service a user would expect, whereby the focus is set on performance monitoring in such a large scale global distributed system. It is an important requirement for Quality of Service (QoS). Since the distribution of services is supposed to be transparent from the user, any kind of delay in the interaction of these services needs to be kept to a minimum. To achieve a control of these interactions the performance needs to be monitored and in case of under-performing a reconfiguration needs to be triggered. Three different architectures for performance monitoring have been designed, whereby the difference between these architectures is the location of the performance monitor within the distributed system. Each of these is able to detect and reconfigure hardware failures and delay in the execution. A hardware failure means a complete shut down of either a network connection or a machine where services are being executed. A delay on the other hand, results in the application not being able to terminate within the time that the user has originally agreed with the service provider.

The algorithms discussed in this thesis estimate the start and end times of application parts running on an heterogeneous Network Of Workstations (NOW) taking into account that the executing machines are slowing down with the increasing load of parallel tasks. Additionally to the estimated execution times information about instructions during which two application parts share same resources are stored. This information is then used to optimise the distribution of the application parts

using a dynamic algorithm which has similarities to Ant Colony Optimization algorithms (ACO-algorithms) and is also described in this thesis.

Finally a Grid Performance Software (*GriPS*) has been designed, developed and tested. It is written in Java and has been used to create experimental results to compare and analyse the different performance monitoring architectures. GriPS simulates a large global distributed system by using connection data collected from CAIDA's skitter initiative [8]. Within the simulation there exist actually three different distributed systems. One for the application itself, one for the performance monitoring within the models, additionally the location broker can be located on a different machine as well.

Acknowledgements

I wish to express my most profound gratitude to:

- My supervisors Professor Dr. Hussein Zedan, Dr. Amelia M. Platt, Dr. Michael J Morse and Mr. Steve M. Rumsby for their invaluable advice, support and encouragement.

- British Telecom (BT) for their financial support and the help in outlining the subject.

- Helge Janicke, Niels-Peter de Witt and Karsten Wolke for hours of discussions and the collaboration during their 6 month project with me.

- Prof. Dr. Karl Hayo Siemsen for initiating my contact to De Montfort University and his encouragement throughout my research.

- All our colleagues at the STRL for providing such a stimulating and friendly working atmosphere.

- My friends Rolf Peters, Oana Spulber, Gilles Compienne, Carola Deppe, Kara McKechnie and Ursula Augsdörfer for all the help and encouragement.

- My family for the constant belief in my abilities

To all of you
Thank you

Declaration

I declare that the work described within this thesis was originally taken by me between January 1999 and February 2003. It is submitted for the degree of Doctor of Philosophy at De Montfort University.

Publications

- Doris Ressmann, Amelia Platt, Steve Rumsby; Architecture to Support Performance Monitoring in Object Based Distributed Systems; Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001); Italy, Bologna; October; 2001

- Doris Ressmann, Amelia Platt, Steve Rumsby; Performance Monitoring of Large Global Distributed Systems; Proceedings of 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2001); Florida, Tampa Bay; October; 2001

- Doris Ressmann, Performance Monitoring in Service Based Computing: the Early Warning Approach; Proceedings of Informatiktage 2002, Germany, Bad Schussenried; November; 2002

Contents

Abstract i

Acknowledgements iii

Declaration iv

Publications v

List of Figures xiii

List of Tables xiv

List of Listings xv

List of Acronyms xvii

1 Introduction 1

1.1 Background to Distributed Systems 1

1.2 Background to Research 2

1.3 Environmental Constraints 3

1.4 Outline of thesis 5

2 Future Distributed Environment 7

2.1 Service-Based Computing 7

2.1.1 Issues Needed for Service-Based Computing 9

2.2 The Structure of Distributed Service-Based Applications 11

2.3 Distributed Environment for Software Developer 18

2.3.1 Essential Requirements in a Distributed Environment 18

2.3.2 Class Names and Class Function Names 19

2.3.3 Model to Structure Class Information to Facilitate Searching . 21
2.4 Distributed Environment for Application Execution 27
2.4.1 Binding 27
2.5 User Requirements Through Contracts 30
2.6 Summary 34

3 Architectural Design for Performance Monitoring and Reconfiguration 35
3.1 Introduction 35
3.2 Centralised Model for Performance Monitoring-Client 36
3.2.1 Advantages 39
3.2.2 Disadvantages 39
3.3 Distributed Architecture for Performance Monitoring 41
3.3.1 On a Per Host (Network) Basis 41
3.3.2 On a Per Class Basis 46
3.4 Prediction and trade offs of the Behaviour of the Models 51
3.5 Algorithms for Performance Monitoring 52
3.5.1 Measurements and their Implementation 52
3.6 Summary 57

4 Estimating the Execution Time of a Globally Distributed Application 58
4.1 Introduction 58
4.2 Representation of the globally distributed application 59
4.2.1 Directed acyclic graph 59
4.2.2 Tree-structure as a specialised DAG 62
4.3 Representation of the resources 64
4.4 Estimating times 65
4.4.1 Sequential execution 66
4.4.2 Parallel execution without slow down 67
4.4.3 Parallel execution affecting machines 68
4.4.4 Generalisation of the times estimation 86
4.5 Summary 91

5 Optimisation of the Execution Time of a Globally Distributed Application **92**
5.1 Introduction . . . 92
5.2 Distributing to slower machines . . . 93
5.3 Idea . . . 94
5.4 Structure . . . 95
5.4.1 Calm down . . . 96
5.4.2 Push back . . . 98
5.4.3 Expel . . . 98
5.4.4 Decide where to go . . . 99
5.5 Generalisation of the optimisation . . . 100
5.6 To push or not to push – cycles . . . 101
5.7 Adjusting the parameters . . . 103
5.7.1 Adjusting the greediness and fear . . . 104
5.7.2 Adjusting the evaporation factors . . . 107
5.8 Comparison with random distributions . . . 107
5.9 Properties of the algorithm . . . 109
5.10 Summary . . . 110

6 The Simulation Model **111**
6.1 Introduction . . . 111
6.2 The Graphical User Interface . . . 112
6.2.1 Machine Panel . . . 113
6.2.2 Application Structure Panel . . . 114
6.2.3 Connection Panel . . . 115
6.2.4 Model Panel . . . 116
6.2.5 Object Diagrams Panel . . . 117
6.3 The Implementation . . . 118
6.3.1 Simulation of a Large Global Distributed System . . . 118
6.3.2 The Simulated Application . . . 118
6.3.3 Performance Monitoring Client . . . 119
6.3.4 Performance Monitoring Server . . . 122
6.3.5 Simulated Application Object . . . 123
6.4 Reconfiguration . . . 124
6.4.1 Reconfiguration due to a hardware failure . . . 124

6.4.2 Early Warning Performance Monitoring 126
6.4.3 Creation of Results 127
6.5 Scalability 127
6.6 Summary 128

7 Case Study (e-learning) 129
7.1 Introduction 129
7.2 Requirements 130
7.2.1 Hypothesis 133
7.3 Application 133
7.4 Analysis 138
7.4.1 Centralised Model 138
7.4.2 Host Model 141
7.4.3 Class Model 143
7.4.4 Comparison of the three models without a failure 145
7.5 Failure recovery 147
7.5.1 Failure recovery after a time-out 147
7.5.2 Early Warning Recovery of a Performance Failure 157
7.5.3 Failure Recovery for a time-critical Application 157
7.6 Summary 157

8 Conclusions and further research and development 159
8.1 Vision 159
8.2 Achievement 160
8.2.1 Performance Monitoring Client and Server 160
8.2.2 Algorithm for Time Estimation 161
8.2.3 Algorithm for Optimisation of Execution Time 161
8.2.4 Grid tool for Performance Monitoring 161
8.2.5 Evaluation 161
8.3 Further Work 162

A Data structures 164
A.1 Data tables for the time estimation in the DAG 165
A.2 Distributions for the 24 Vertex DAG 188

B Grid Performance Software **190**
B.1 The Graphical User Interface 190
B.1.1 The GUI Structure 192
B.1.2 The menu bar 192
B.1.3 The different panels 194
B.1.4 The Settings Dialog 205
B.2 Diagram Settings 212
B.2.1 Diagram Scaling 212
B.2.2 Graph Policies 213
B.2.3 Axis & Grid 214
B.2.4 Misc 215
B.2.5 MetaPost and TEX 215
B.3 Example Files 216
B.3.1 The Connection-Data file 216
B.3.2 The Replacement file 218

C Optimization Software **219**

References **224**

List of Figures

2.1 *The structure of a distributed application* 12
2.2 *The structure of a distributed application object based* 13
2.3 *A remote method invocation* 14
2.4 *Replicated Location Broker* 23
2.5 *logical class server hierarchy in physical network* 24
2.6 *hierarchy of class servers* 25
2.7 *contract diagram* 31

3.1 *Interaction between the PMC and PMS in centralised model* 37
3.2 *Centralised Architecture for Performance Monitoring* 40
3.3 *Server Side Performance Monitoring in Centralised Model* 41
3.4 *Interaction between the PMC and PMS in host model* 43
3.5 *Distributed Architecture for Performance Monitoring on a per host-(network) Basis* 45
3.6 *server and client side performance monitoring on a per host- (network) basis* 46
3.7 *Interaction between the PMC and PMS in class model* 48
3.8 *Distributed Architecture for Performance Monitoring on a per class basis* 50
3.9 *server and client side performance monitoring as part of an class* . . 51

4.1 Sample Network of Workstations 59
4.2 Application structure 61
4.3 Layers for the execution 62
4.4 simplified DAG (left), tree structure (right) 63
4.5 DAG for a sequential application 66
4.6 Time flow (greedy) 75

4.7 Time flow for the DAG example 89

5.1 Timeflow (more distributed) 94
5.2 Without push back 102
5.3 Push back with $\lambda_2 = 0.2$ over 50 iterations 102
5.4 Push back with cycle over 100 iterations 103
5.5 Totally greedy behaviour ($\alpha = 1, \gamma = 0$) 104
5.6 Initial greedy solution ($\alpha = 1, \gamma = 0$) 105
5.7 Found totally greedy solution ($\alpha = 1, \gamma = 0$) 105
5.8 Totally greedy behaviour ($\alpha = 1, \gamma = 0$) 106
5.9 Totally frightened solution ($\alpha = 0, \gamma = 1$) 106
5.10 Random distributions versus the algorithm choices 108
5.11 Random distributions versus the algorithm choices 108

6.1 General overview of the Grid Performance Software 112
6.2 The Overview of the Graphical User Interface 112
6.3 The Machine Panel 113
6.4 The Application Structure Panel 114
6.5 The Connection Panel 115
6.6 The Model Panel 116
6.7 The Object Diagrams Panel 117
6.8 *Class Diagram for the Performance Monitoring Client* 121
6.9 *Class Diagram for the Performance Monitoring Server and SimulatedApplicationObject* 123
6.10 *example application* 125

7.1 *e-learning distributed over the entire globe* 131
7.2 *e-learning service connections* 132
7.3 *The e-learning example application* 134
7.4 *First year study course* 136
7.5 *The connection speed between the used machines* 137
7.6 *Sequence Diagram of an e-learning course in the centralised model* . . 139
7.7 *Sequence Diagram of an e-learning course in the host model* 142
7.8 *Sequence Diagram of an e-learning course in the class model* 144
7.9 *Chart about the total execution time* 146
7.10 *First year study course in case machine sienna failed* 148

7.11 *Sequence Diagram of an e-learning course in the centralised model when one host fails during execution* 150
7.12 *Sequence Diagram of an e-learning course in the host model when one host fails during execution* 152
7.13 *Sequence Diagram of an e-learning course in the class model when one host fails during execution* 154
7.14 *Chart about the total execution time in the failure case* 156

A.1 Best possible distribution 188
A.2 Worst found solution 189

B.1 *The Graphical User Interface Service Panel* 195
B.2 Adding a machine 196
B.3 The Connection Panel 199
B.4 Adding a connection 200
B.5 The Application Panel 202
B.6 Setting the LocationBroker 206
B.7 Setting of object parameter 207
B.8 Setting the timer 208
B.9 Setting the speed diagram settings 209
B.10 Setting default values for connections 210
B.11 Setting default values for services 211
B.12 Scaling values of the diagram 212
B.13 Policy values of the diagram 213
B.14 Axis and grid values of the diagram 214
B.15 Miscellaneous values of the diagram 215

C.1 Screen shot of the test application 220

List of Tables

2.1 *class location* 13
2.2 *scheduling* 13
2.3 *The Advantages and Disadvantages of local execution* 17
2.4 *The Advantages and Disadvantages of remote execution* 18
2.5 *A possible classification / numbering containing the functionality* . . . 20

4.1 Machine speeds/ services 74
4.2 greedy schedule 74
4.3 Description of a collision tuple 76
4.4 Collisions for the *greedy* example 81
4.5 Machines and distribution for DAG example 89
4.6 Iteration sample 90

5.1 Distribution (more distributed) 93

7.1 Distribution of e-learning services 134
7.2 E-Learning Scheduling 134
7.3 Connections used in this example 135
7.4 Results for the Centralised Model 140
7.5 Results for the Host Model 141
7.6 Results for the Class Model 143
7.7 Results for the Centralised Model in the failure case 149
7.8 Results for the Host Model in the failure case 151
7.9 Results for the Class Model in the failure case 153

Listings

2.1 Contract Agreement 32
2.2 Ponder Posetive Authorisation Example 33
2.3 Ponder Negative Authorisation Example 33
2.4 Ponder Obligation Example 34
3.1 Centralised Model 38
3.2 Host Model 44
3.3 Class Model 49
3.4 Time - Out Failure 54
3.5 Failure detection triggered from location broker 54
3.6 Time-critical applications 55
6.1 The Early Warning Algorithm 127
B.1 Fake instruction 190
B.2 Windows SpeedServer start 190
B.3 Unix SpeedServer start 191
B.4 Windows LOB start 191
B.5 Unix LOB start 192

List of Acronyms

ACO-algorithm Ant Colony Optimization - algorithm

ATM Asynchron Transfer Mode

BT British Telecom

CAIDA Cooperative Association for Internet Data Analysis

COM Component Object Model

CORBA Common Object Request Broker Architecture

DAG Directed Acrylic Graph

DARPA Defense Advanced Research Projects Agency

DCOM Distributed Component Object Model

DMU De Montfort University

DTD Document Type Definition

GriPS Grid Performance Software

GSM Global System for Mobility

GUI Graphical User Interface

HLR Home Location Register

LOB Location Broker

NAB Network Address Binding

NOW Network of Workstations

NSF National Sience Foundation

NTP Network Time Protocol

OLE Object Linking and Embedding

OMG Object Management Group

PMC Performance Monitoring Client

PMS Performance Monitoring Server

QOS Quality Of Service

RAID Redundant Array of Independent Disks

RMI Remote Method Invocation

RPC Remote Procedure Call

SAObject Simulated Application Object

VCR Video Cassette Recorder

VHS Video Home System

XML EXtensible Markup Language

Chapter 1

Introduction

1.1 Background to Distributed Systems

Distributed Systems have been the focus of much research activity, for a number of decades. A Distributed System is defined by Coulouris et al [10] as one 'in which hardware and software components located at networked computers communicate and co-ordinate their actions only by passing messages'. Most distributed systems in existence today, are based on the client/server paradigm. In this paradigm the client (personal computer or workstation) makes a request for some service (e.g. email) while the server (typically a high specification computer) responds to the request. The client and server may be physically separated but connected via a communications network. The client typically implements the user interface ("thin" client) and may perform some or all of the application processing which are dedicated (e. g. cray).

While client/server applications are in theory distributed, nevertheless the distribution tends to be relatively simple; indeed Coulouris describes existing distributed systems as *simple* [10]. For instance, the server is located at a well-known address and thus the client always knows where to send the request. Furthermore, the application is distributed across only two cluster computers and thus two (logical) locations. However, recent developments in software construction and communications have created new opportunities for more sophisticated and flexible distributed systems. Object technology allows applications to be constructed using small software components called classes; a class is the template for an object. A class models a small part of a system and is comprised of data and set of operations on that

data. By designing classes that are independent components in their own right. it is possible to use classes in more than one application. Indeed class reuse is considered to be one of the greatest benefits of object technology [17]. Taking this one step further, it is possible to construct applications from existing classes that can be widely distributed. This distribution is facilitated by the existence of very high speed local area networks (LANs) and wide area networks (WANs) which are capable of delivering the Quality of Service (QoS) required by the application.

1.2 Background to Research

This thesis is concerned with some aspects of the design and management of large scale, global distributed systems that can support many millions of users and a wide variety of services. Telephony, videoconferencing, Internet access, television and entertainment, etc are some obvious services which must be supported. Anything which can be encoded in bits and transmitted across a network is considered a potential service in this future distributed system. Note that in the context of this research, a **service** is provided through the execution of an application and we use the terms service and applications synonymously.

The work was sponsored partially by the Future Distributed Systems Group of British Telecom (BT). BT initially outlined a number of major problems unique to these future distributed systems and it was agreed that performance monitoring was a major requirement in this new environment. The role of performance monitoring is to monitor the performance of the system and, if necessary, reconfigure some parts of the system. In terms of distributed applications, performance monitoring specifically monitors the performance of the executing application using the QoS defined by the user at runtime. The performance is considered *unacceptable* if the QoS is not satisfied. Performance monitoring must consider both what aspects of the performance are to be monitored and also how this can be realised. The system must also allow for the application to be partially reconfigured (which will typically mean that underperforming classes are substituted) while it is running.

From an early stage it was clear that performance monitoring could not be considered in isolation from the rest of the environment. For instance, performance monitoring relies on knowledge of how distributed applications execute. Thus in considering performance monitoring, assumptions and proposals had to be made

regarding the runtime environment of this distributed system; these are discussed in this thesis. It should be noted that because research into this type of distributed system is very new, there was no body of knowledge on which to build. Thus, all the work for this thesis has broken virgin ground, drawing on good practice and design used in related system software.

The reader who is familiar with distributed systems will be aware of the contribution COM (Component Object Model), OLE (Object Linking and Embedding) and DCOM (Distributed Component Object Model) which are Microsoft's counterpart to CORBA (Common Object Request Broker Architecture) [42] have made to the development of distributed systems in recent times. CORBA is a standard for distributed systems defined by the Object Management Group (OMG), based loosely on an object based client/server paradigm. The main aim of CORBA was to define a distributed platform which provided transparency of hardware, software (programming language and system software) and location between client and server objects. A feasibility study carried out prior to the start of this project indicated that while CORBA had many strengths, it was not a suitable platform for a large scale, global distributed system [38]. This is also the view held by BT [32].

1.3 Environmental Constraints

A Grid Performance Software (*GriPS*) has been designed, developed and tested. It is written in Java and requires a JRE version 1.3 or higher to run. This software has been used to create experimental results to compare and analyse different performance monitoring architectures. GriPS simulates a large global distributed system by using the minimal, maximal and average delay between different location around the world. These connection data were originally collected, eight times a day (at 01am, 04am, 08am, 10am, 01pm, 04pm, 07pm and 10pm) with the help of an automated *ping* command to the following Internet Domains:

charity	internet.vsnl.net.in
web.mit.edu	www.acm.org
www.arab.net	www.canada.com
www.china.com	www.csu.edu.au
www.fho-emden.de	www.indiaworld.com
www.japantimes.co.jp	www.kaau.edu.sa
www.latimes.com	www.man.ac.uk
www.ncl.ac.uk	www.ox.ac.uk
www.ru	www.southafrica.net
www.state.ny.us	www.state.tx.us
www.uchile.cl	www.washingtondc.gov
www.whitepages.com.au	www.yahoo.com

However it was soon discovered that the information collected was not sufficient, because no information about the delay between these location were created. CAIDA, the Cooperative Association for Internet Data Analysis collects, monitors, analyzes, and visualizes several forms of Internet traffic data concerning network topology, workload characterization, performance, routing, and multicast behavior. These analyses serve a variety of disciplines/purposes, including research, policy, education, and visualization.

As such the data used in this research was collected as part of CAIDA's skitter initiative, http://www.caida.org Support for skitter is provided by DARPA, NSF, and CAIDA membership.

For the example in Chapter 7 four machines with the following specifications are used.

real machine name	specification of the machine	Simulated IP address	Location of Simulated machine
forquet	HP model C3000 (PA-8500 CPU), 2GB memory, O/S: HP-UX 11.00	128.8.7.4	College Park, MD, US (University of Maryland)
ossi	Pentium 450Mhz, 256MB memory, Win NT4.0	216.168.227.250	Herndon, VA, US (Verisign)
garfield	Sunblade 150, UltraSparc IIi 650Mhz, 256Mb memory, Solaris 8 (108528-16)	203.181.248.27	Tokyo, Kanto, JP (APAN)
sienna	Sunblade 150, UltraSparc IIi 550Mhz, 128Mb memory, Solaris 8 (108528-16)	193.0.0.11	Amsterdam, North Holland, NL (RIPE)

1.4 Outline of thesis

The present work is organised as follows:

Chapter 2

Describes the requirements for service-based computing and the different perspectives a user, service provider and a developer will have. Furthermore it covers design problems in *future distributed environments.* It demonstrates how an object-based application can be distributed over a vast amount of machines within a large global distributed system. Furthermore several options are given to search for these objects. Additionally it is explained how these remote objects are finally put together to build one service-based application.

Chapter 3

Covers the *architectural design for performance monitoring and reconfiguration.* Out of many possible solutions where in the distributed system the performance monitor is located, three solutions are being introduced. Finally this chapter explains what these performance monitors will have to measure and the expected differences between the architectures. It will not be possible to choose one of these models and announce it to be *"the best"*, since it depends strongly on the individual conditions, like the type of application, the level of distribution and other environmental parameters. The reason for the different performance monitoring architectures is to find an appropriate architecture for an application. It is to predict that not all three models behave in the same way.

Chapter 4

Is all about *estimating the execution time of a distributed application*, which will allow an early detection of a performance problem during execution. This idea is evolved stepwise from a sequential to a parallel executing application. Finally an algorithm to optimise the object distribution is developed.

Chapter 5

Describes *the simulation model* which is written in Java, requires a JRE version 1.3 or higher to run, and simulates a large distributed system. First the graphical user interface is described with all its features, secondly the underlying technology.

Chapter 6

Uses a *Case Study about e-learning* to create results which allow to compare the different architectural models.
At the end it is described how the different models react in case of a failure. And how they reconfigure the application.

Chapter 7

Finally, in the last chapter, the conclusions and further research are given.

Chapter 2

Future Distributed Environment

2.1 Service-Based Computing

A traditional application need to be bought, installed, updated and maintained by the user. On the other hand service-based computing is a new computing paradigm where an application is not sold, but made available for remote usage. It is executed from all over the world on the machines where it is installed. A service-based application comprises several perspectives. The user who wants to use the service, the service provider who advertised and sells the service, a location broker who finds several object providers, each of which own one or more objects, which build together the service. When viewing a service-based application it is to bear in mind that each of these roles have a different point of view.

The user's perspective The user sits on his/her own PC and uses the service-based application only when needed. S/he has no knowledge about the location of this application and has the advantage that the latest version of service is always available. As long as the appearance and the compatibility, to earlier versions, stay in a way that the user can easily adjust to these changes, the user will have all the benefits. However when these conditions are not fulfilled the user might get disappointed and does not want to use this service any longer. One of the main advantages for the user is the fact that s/he only has to pay for the time the service has been used. Very much the same as we use a telephone service nowadays. In general the user will get the impression that every program would be installed on the same machine and will not be able to recognise that in reality every service can

be located at a completely different place. The management of such a service-based application is completely transparent unless there is a fault, from where it is not possible to recover, in this case the user would get an error message.

The service provider's perspective The service provider constructs out of several widely distributed objects a service, whereby the objects remain on the server of the object provider and a location broker is responsible for finding all these remote objects. The service provider only realises the connection between these objects. These newly combined objects build a software as a service. The service provider collects the cost for the use of every single object and every network connection, adds a service charge for its own so that the user then will have to pay only for the time this service has been used.

The location broker The location broker is responsible for finding the location of specified objects on any object provider around the world. The location broker is being used by the service provider who assembles out of different objects a service as well as by the developer of new objects to allow interaction with other objects.

The object provider's perspective The object provider is not able to see the overall picture of how these objects are being used. On the server is a functionality which is able to collect information how long each object is being used by whom and how much this provider is able to charge for it. The objects provided are being advertised at a location broker together with their interfaces, so that developers and service providers are able to use objects from other providers to create a particular functionality.

The developer's perspective The developer of an object, which is specially developed for such a distributed application, will have to take care that each object can be executed remotely. The interface has to be precisely specified to allow other parts to interact and to use objects developed from different teams. The advantage is that each object can be updated separately as long as the new version is fully compatible with the old version. The developer will also have to bear in mind that some kind of monitoring will be installed between every possible remote call.

2.1.1 Issues Needed for Service-Based Computing

A service-based application needs the following issues paramount in their development compared to a traditional centralised application.

- security
- accounting
- naming
- location management
- network configuration
- fault management and
- performance monitoring; including scheduling-related issues

Security When distributing an application and running software on remote locations security is a big issue. There might be a risk of harmful attacks on the remote machines. These attacks might have the form of eavesdropping, masquerading, tampering and denial of service. Cryptography provides the basis for the authentication of messages as well as their secrecy and integrity; carefully designed security protocols are required to exploit it [10]. A policy based specification, e.g. used in Ponder [11, 12, 13] gives a way to control the access and specifies what activities a subject is permitted or forbidden to do, to a set of target objects. However it is very hard, if not impossible to ensure a system is secure in the sense that no intruder can find access to this system. A very secure machine would be a standalone machine without any network connection, where nobody has access and it is powered by its own source. It is incidental that security measurements do not only avoid unauthorised access but also establish how to deal with an intruder once it is detected. Neumann et al [36] introduces a way to allow a network administrator to realise when an intruder has accessed the system, and follow every step the intruder is doing.

In addition, the system might need a protection against Spam. An increase of network traffic needs to be detected and dealt with. E.g., in case the network connection is intensively used for all kind of messages, just for the sense of decreasing the amount of traffic able to be transferred over this part of the network.

Another aspect of security is a form of self harm protection. If the system itself is malfunctioning, this needs to be detected and dealt with.

The aspects of security are not considered in detail in this thesis, and will be a subject of further work.

Accounting To execute an application as a service in a distributed manner, the way the charging for the application as well as for the resources used are operated will have to change. In service-based computing the user will be charged in a **'pay per use'** basis. The charging will operate in much the same way as charging for telecommunication services, in that customers will receive an itemised bill.

A similar accounting system existed already in the sixties in the hight of mainframe computers. At this time computational power was limited to few powerful machines at the time, and the user had to connect for his/hers computation. Obviously this computation was not for free and different accounting systems were in operation. Diamond et al [14] describes different billing systems used for mainframe computations. Some systems billed the user with a flat rate or for the connection times. Other used billing schema's for the transaction or the resources used.

Naming and Location Management For service-based computing it is an important factor to find the remote locations of the services involved. This creates a major problem especially in widely distributed systems. It is crucial that every service can be uniquely identified, either by its name or by its functionality. However the naming only solves one problem to find a remote location. After identifying exactly what object we are looking for, the location where it resides has to be found.

Network Configuration A service-based application is likely to be distributed over different types of networks. Every single network will have to be configured in its own way. As King et al [25] defines, to name but one, this configuration management exists to collect and monitor configuration information so that the effects of changes in hardware and software can be managed. This allows re-routing in case of failures. Furthermore all these networks need to interact with each other.

Fault Management Fault management involves a five step process. First the fault needs to be detected, and located. The location of the fault can result in a restoration of the services. A solution to avoid the fault needs to be found, e.g. an

alternative service will be used. Following the location and restoration the root of the problem need to be identified. Why did this fault happen? Is there any way to avoid that this fault will happen again? Finally the problem needs to be resolved.

Performance Monitoring and Scheduling Performance monitoring is an important requirement for Quality of Service. Since the distribution of objects is supposed to be transparent for the user any kind of delay in the interaction of these objects need to be kept to a minimum. To achieve a control over these interactions the performance needs to be monitored and in case of under-performing a reconfiguration needs to be triggered.

Crucial to the above is tasks/object scheduling. A large number of scheduling techniques with underpinning theoretical foundation exist. Key references include [37, 29, 31, 24, 6]. Chapter 4 deals with these issues and its relation with our performance monitoring techniques.

2.2 The Structure of Distributed Service-Based Applications

In describing distributed applications, it is important to differentiate between the terms classes and objects. A class can be described as the template for an object; it *defines* the code (in the form of methods) and data for the object. The object comes into being (is instantiated) when the class code is executed. The instantiation of an object allocates memory for the data defined by the class and thus allows the object data to be manipulated. The relationship between class and object is similar to the relationship between program and process. Where the program represents the source-code and the process the execution of this code. In this thesis the term object is used only when the execution of classes are being discussed.

It was stated earlier that in this distributed environment applications are built using existing classes, wherever possible. Furthermore, the classes can be widely, indeed globally, distributed. Figure 2.1 shows one way in which a distributed application can be viewed. In this Figure it is assumed that these classes are located at 6 different addresses in this distributed environment and thus in this instance the application is widely distributed. Table 2.1 shows the distribution of the classes.

It can be seen from Figure 2.1 that the execution of the classes comprising the

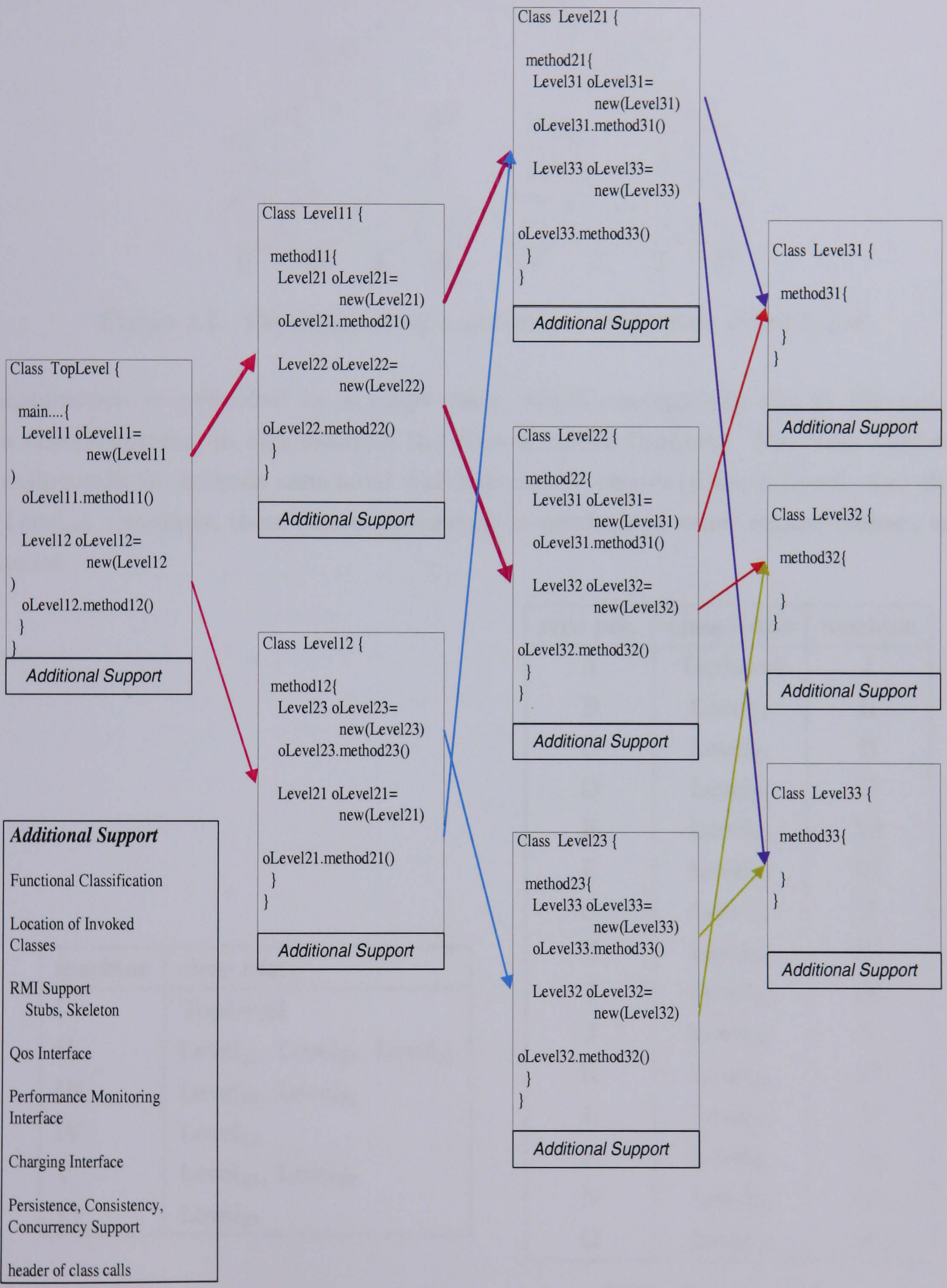


Figure 2.1: *The structure of a distributed application*

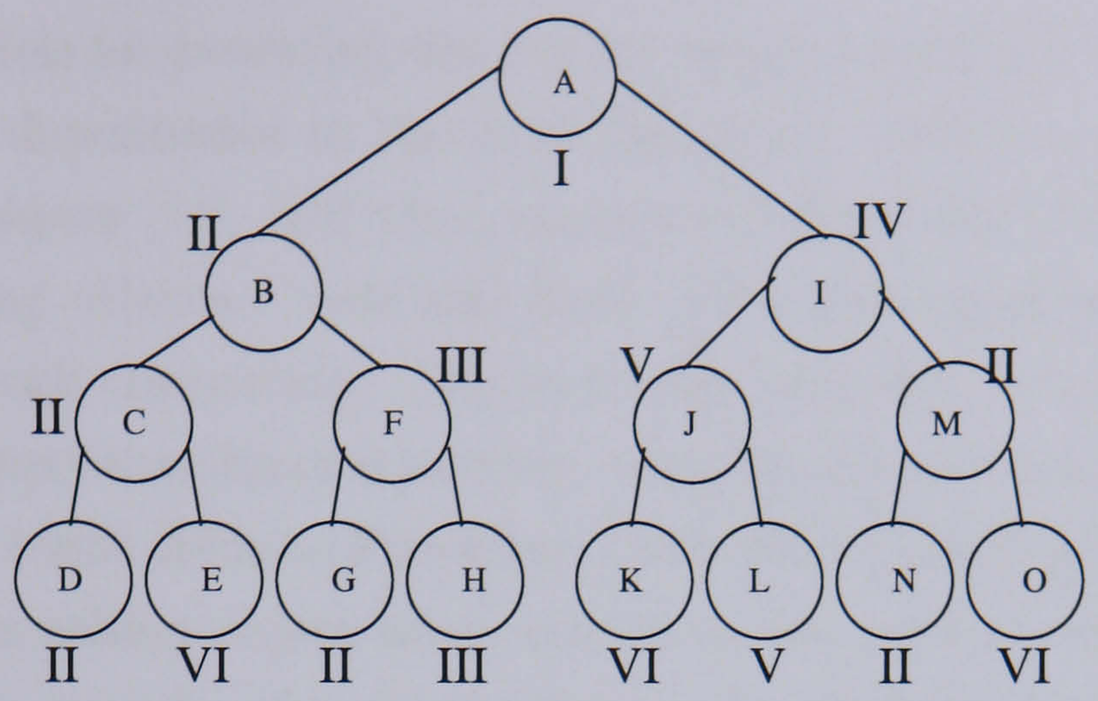


Figure 2.2: *The structure of a distributed application object based*

application is controlled by a single class, which conceptually sits at the top of a class hierarchy; in this example the class is called *TopLevel*. The class *TopLevel* makes calls to methods associated with two remote classes (Classes $Level_{11}$ and class $Level_{12}$). Similarly, these classes make calls to methods in other remote classes, and so on.

machine	class name
I	TopLevel
II	$Level_{11}$, $Level_{21}$, $Level_{31}$
III	$Level_{22}$, $Level_{32}$
IV	$Level_{12}$
V	$Level_{23}$, $Level_{32}$
VI	$Level_{33}$

Table 2.1: *class location*

tree pos.	class name	machine
A	TopLevel	I
B	$Level_{11}$	II
C	$Level_{21}$	II
D	$Level_{31}$	II
E	$Level_{32}$	VI
F	$Level_{22}$	III
G	$Level_{31}$	II
H	$Level_{32}$	III
I	$Level_{12}$	IV
J	$Level_{23}$	V
K	$Level_{33}$	VI
L	$Level_{32}$	V
M	$Level_{21}$	II
N	$Level_{31}$	II
O	$Level_{33}$	VI

Table 2.2: *scheduling*

During runtime each class allocates memory and as such becomes an object.

There are two models for executing the remote method calls. Either the appropriate class code can be downloaded to the local (invoking) machine, using for example, serialisation techniques [17], and then execution takes place locally. Serialisation works by converting objects - code and data - into byte streams and then sending them over a network connection. Alternatively, execution can take place at the remote machine where the object is located, using Remote Method Invocation (RMI) (e.g. in procedure terms Remote Procedure Calls (RPC)) techniques and the results are returned to the calling object when execution has properly terminated.

The application introduced in Figure 2.1 can be illustrated as a tree, to demonstrate its runtime behaviour (see Figure 2.2). The capital letters show the position of each object in the tree in preordered enumeration. Whereby the *TopLevel* class becomes the root-object (in the following also called the root-node) and is situated at position A. The root-object might call both its children at position *B* and *I* in parallel. It can be seen that out of class Level_{21} actually two objects are instantiated, one at position *C* and the other at position *M*, both of them are located on *machine II*. Table 2.2 shows such distribution.

To be able to call a remote method every remote class is instantiated on the remote machine and registered with a *registry*. When the local object calls a method from a remote object it gets the handler from the remote registy and is then able to invoke this remote method. (see Figure 2.3)

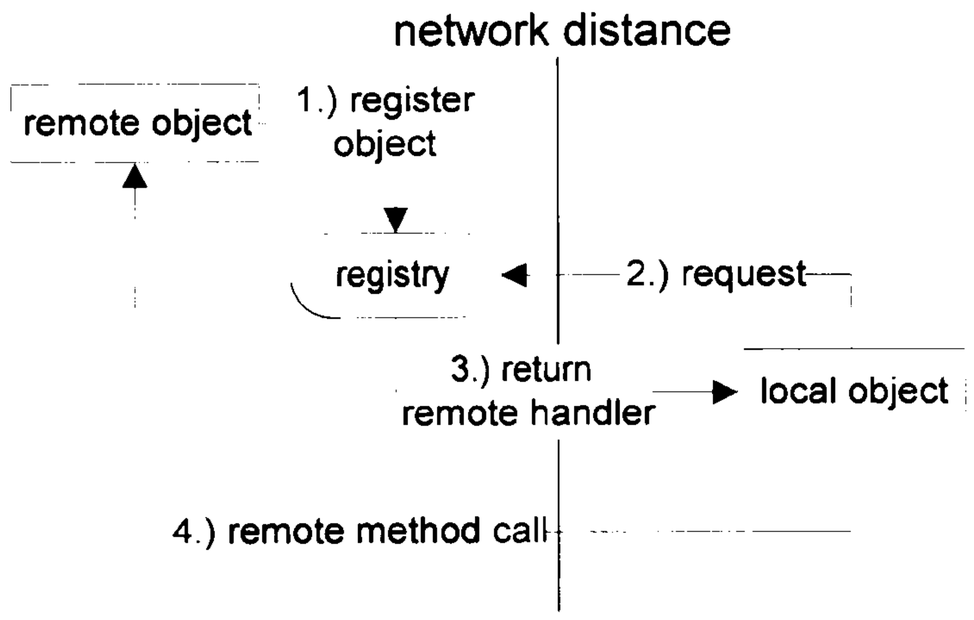


Figure 2.3: *A remote method invocation*

We are considering the remote execution model because there are a number of disadvantages associated with the execution locally and additional advantages to be gained from the remote execution model. A summary of these advantages and disadvantages for the local execution can be seen in Table 2.3 and for the remote

execution in Table 2.4 each of them are explained in the following.

The traditional way of executing an application on one local machine, as opposed of executing it directly on the remote location, has the advantage that neither network delay nor the availability of remote objects will influence the execution. When an object is executed on a remote machine and the results are transferred back to the invoking object the network in between these two objects will delay the arrival of the results. However whether this delay is acceptable depends on the amount of delay, which is dependable of the network speed and the congestion in this part of the network.

Depending on the size of the class code compared to the size of the results created, it can be faster and more efficient to execute a class either on the local machine or on the remote machine and simply transfer the smaller part over the network. In case the size of the results is big in contrast to the size of the class code, the class code would be downloaded to the local machine and execution takes place on the local machine. On the other hand when the size of the results is fairly small it is feasible that the execution takes place on the remote machine, especially when the distance to the invoking class is high and the network speed low.

A disadvantage of the local execution will be when a class uses features not available on the local machine. For example a java class needs a virtual machine running on the machine where it is executed. In case the virtual machine is not already installed it has to be done locally and somebody has to have the knowledge and the rights of how to install it. For example, a user who is only interested in the results, created from the remote class, has not necessarily the administration rights to install any software on the local machine.

Also it is very difficult to measure how often a piece of software has been executed once it is installed. Traditionally this problem is overcome by the fact that the user will have to buy the software, install it on the local machine, and then use it unlimited times. However the disadvantage of this method is the fact that a user might spend lots of money on a product which does not completely fulfil his or her requirements. The remote execution of objects changes the way the charging for resources used is operated, for instance the cost of using the object itself and the cost of the resources required for execution. In particular, they will not buy software outright or pay a renewable license fee to gain access to resources, rather they will be charged for the service used on a 'pay per use' basis. The charging will operate in much the same way as charging for telecommunication services, in that customers

will receive an itemised bill. Thus the distributed system must automatically collect billing information (this is shown as 'Billing Support' in the Additional Support in Figure 2.1).

A further disadvantage originates when a new version of a locally installed application becomes available. The maintenance of such an application will take time and uses again the network resources. An update or even a complete new version will have to be downloaded. All these disadvantages are avoidable when the application is maintained remotely.

The additional advantages of a remote execution is when the execution of an object needs specific hardware only available on the remote machine. For example an object which has to convert a Video Home System (VHS) video tape in a "mpeg" file format will need to use the features of an VHS Video Cassette Recorder (VCR). The VHS tape and the VCR is not available on every machine in the network. After the conversion has taken place this video file would be in a format which can be read without the need of the hardware and therefore is portable to be viewed on any machine.

An application which needs a large amount of processor power will slow down the processing speed of the machine where it is executed. However when this application is distributed all over the network, every part would use a tiny bit of the processing power on each machine, and the overall load will not be disturbing.

LOCAL execution	
advantages	disadvantages
• no network delay during execution • when the local machine is running the application is running • depending on the application the time to download an application can be less than the transmission of the results	• the time to download an application can be unacceptably high • installation may be required and security becomes an issue • it is difficult to control access to the class once a copy has been downloaded. For instance, class static variables which are shared between all instantiated objects for a class are more difficult to control. • the user will have to update and maintain the application regularly

Table 2.3: *The Advantages and Disadvantages of local execution*

REMOTE execution	
advantages	disadvantages
• the object may require hardware which is only available where it is located • the processing load is distributed • easy access control • the user has no maintenance requirements to fulfil • depending on the application the time to transmit the results might be less than to download the whole application	• delay during execution • depends on the availability of the remote machine and the network connection during the whole execution

Table 2.4: *The Advantages and Disadvantages of remote execution*

2.3 Distributed Environment for Software Developer

2.3.1 Essential Requirements in a Distributed Environment

To understand the problems of developing distributed object-based applications it is useful to consider object based applications where all the class libraries are stored locally. Typically a development environment tool such as Jbuilder [23] will be used. Among other things the developer will need support from the development tool to browse the class libraries in order to obtain precise information relating to the functionality and interface of the class. If a distributed environment is considered then clearly the class libraries will be distributed, possibly over the entire globe. Thus it is not practical for the developer to find class details by browsing. Instead the developer must find classes by *searching*. In some instances the developer will

know the name of the class, in which case the search is carried using the class name. However, a more realistic scenario is that the developer will know only the *function* that the class must provide. In this instance the search must be carried out by using the name of the function required. For instance, the search may be for a class providing a *sort* function. In summary, classes other than local standard library classes will typically be found using the class function search facility in contrast to searching by class name.

In searching for classes, either by name or function, there are two related problems to be solved. One relates to the naming of the classes (or the class function) and the other relates to structuring the distributed systems to facilitate the search. These two problems are considered below.

2.3.2 Class Names and Class Function Names

Classes must be given globally unique names when they are made available for use. Clearly replications of a given class must have exactly the same name. These replications will exist at various locations identified by some addressing scheme, for instance the one being used by the Internet. Note that the address is not part of the class name, and therefore cannot be used in combination with the class name to uniquely identify a class. Thus to find a class in a distributed environment, only the class name is used in the search. The address, together with the class interface details, is the information returned by the search. The need for unique names for classes is widely recognised and solutions are discussed in e.g. [4, 41, 44] to name but a few.

In contrast, the name of a class functionality will not be unique because it is expected that more than one class will provide the same identical functionality. For instance, two different classes may provide a sort method. It is important that classes providing the same functionality are given the same name in reference of their functionality. This could be achieved by using a function classification scheme. One such classification scheme is shown in Table 2.5. The structure of this classification is similar in principle to the Domain Name Service (DNS) used in the Internet [43].

Application or Object?	General	Detail 1	Detail 2	Detail 3	Number
Application	Financial				
	Leisure	Entertainment			
		Gardening			
		Games	Race		
			Card	Snap	
				Bridge	Different bridge card game applications
				Poker	
				etc	
			Adventure		
			etc		
	Medical				
	Design				
Object	Sport				
	Mathematics	Statistic			
		Algorithm	Sort	Quick Sort	
				Merge Sort	Different Merge Sort objects
				Bubble Sort	
		Calculations			
	Multimedia				
	etc				

Table 2.5: *A possible classification / numbering containing the functionality*

The first characteristic of this classification describes if the element is an object

or a whole application. The second characteristic describes a general specification. In the example shown in the table there are four different kind of applications and three different kind of objects. The applications are divided by functionality: financial, leisure, medical, design -application. The objects are divided into sport, mathematics and multimedia -objects. The next characteristic describes the next level of detail. Whereby the leisure application is divided into Entertainment, Gardening and Games. The mathematical object is divided into statistics, algorithm and calculations. The second level of detail divides the games in the category of leisure application into race, card and adventure -games and the object mathematics algorithm into sort algorithms. The third level of detail then divides the card games into snap, bridge, and poker games as well as the sort objects into quick, merge and bubble -sort algorithms. The last characteristic consist of a number to differentiate different bridge card games and to differentiate different merge sort objects.

Before a class is made available for public use, it would be classified (allocated a name to specify its functionality) according to the classification shown in Table 2.5. Clearly the actual classification would have to be standardised and administered by a centralised body. However, authority could be delegated for sub-classes, perhaps to the industries which typically produce software for this sub-class. For instance, the world financial institutions could assume responsibility for standardising the classification of functions relating to financial applications. The function classification information could be encoded using markup description languages such as eXtended Markup Language (XML). XML provides a way of describing the attributes and content of information and is applicable to all types of information including that held electronically. For example the Metalab group [30, 33] uses Metadata, which is a particular XML Document Type Definition (DTD), as a way of cataloguing free software for LINUX. Developers catalogue and register their software and subsequently users looking for LINUX software can search the register. In the past it was difficult to locate LINUX Software, because although there was a huge amount of free software available, there was no standardised way to find it.

2.3.3 Model to Structure Class Information to Facilitate Searching

Having considered the problems of class naming and functional classification, the way in which this information is structured/distributed to facilitate searching must

now be considered. Ultimately the objective is to provide a structure, which allows classes matching a given name/functionality to be found. There are a number of ways in which this can be achieved and a simple taxonomy is to consider approaches where the class summary details are stored separate from the actual class code and where they are one and the same information.

2.3.3.1 Class Code and Class Summary Details are Separate

With this scheme the class code is made available by the developer and is typically stored on a server (hereafter called a class-server) at the developers site. Class summary information (name function, address, etc) are registered with one, or more location broker which exist at different sites. There are typically a small number of location brokers and these will be shared between distributed systems. A search engine is used to find the appropriate class at a location broker in much the same way as search engines are used currently. The search engine is likely to be integrated with the application development tools. Assuming that at least one location broker has knowledge of the class then the summary information together with the class address is returned and the development tool now uses this information to communicate directly with the appropriate class-server. The user can now browse the class documentation as if it was stored locally.

With this approach the main problem is in deciding how to structure and distribute the location broker. One, centralised location broker is not feasible, for a number of reasons. The reliability of centralised systems has always been a drawback. Moreover it would clearly represent a bottleneck because all accesses would be through this one broker. It is possible to replicate the broker, over a number of sites and this would solve these problems, however, it is unlikely that a location broker would have the resources to store (and retrieve) details of all classes, efficiently. The maintenance of replicated data is an additional problem. All the replicated location brokers will have to have a consistent view of all available classes. Figure 2.4 shows how a replicated location broker could be implemented.

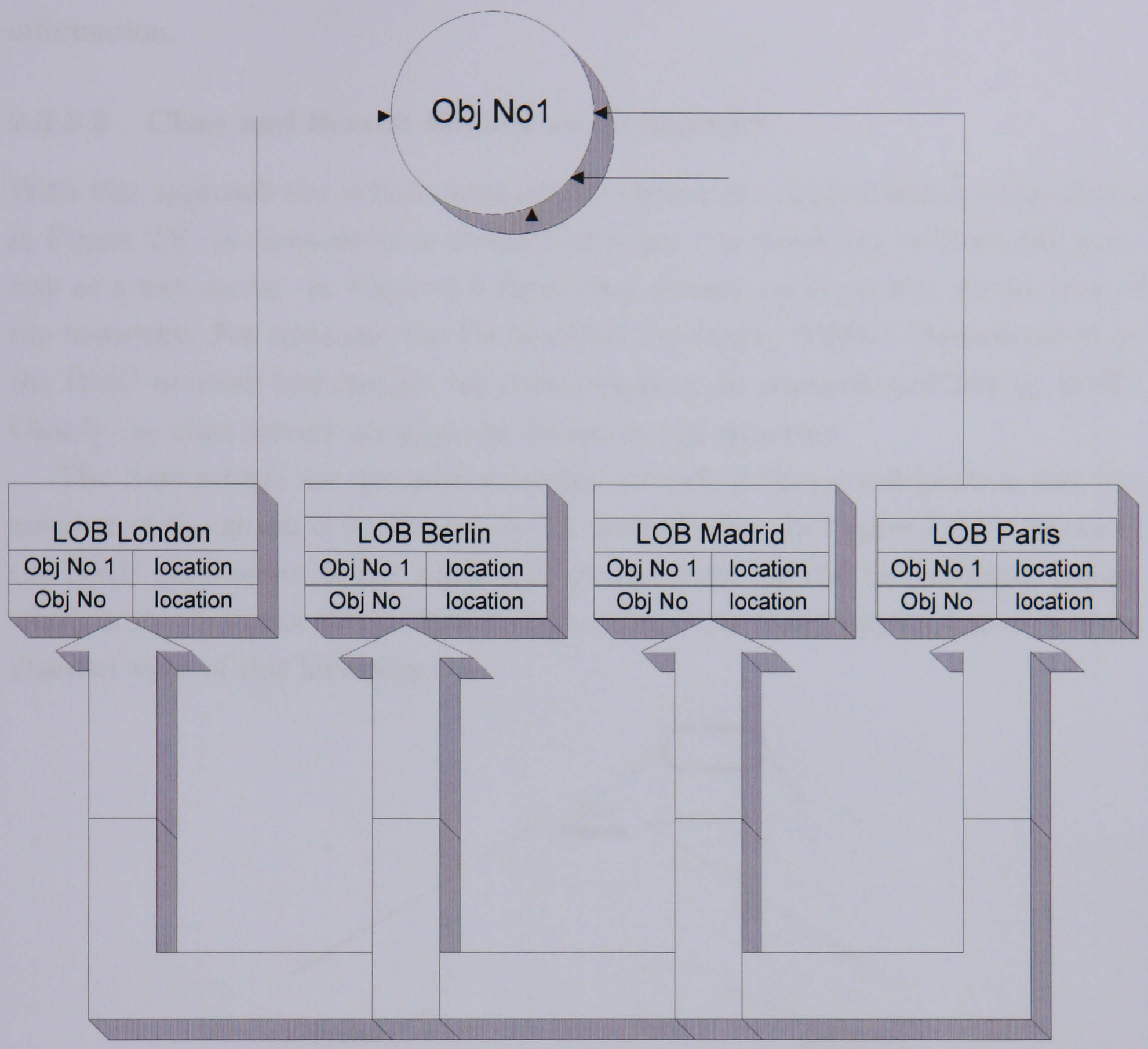


Figure 2.4: *Replicated Location Broker*

The only realistic solution is to consider the location broker as a logical entity, which can be partitioned and distributed throughout the distributed system. This is how the Home Location Register (HLR) is defined by the Global System for Mobility (GSM) standard [19]. The partitioning could for instance be by functionality, according to the classification discussed above. For example, all entries for the financial classification could be stored in one place. The cost of providing and maintaining this location broker could be borne by the professional body responsible for maintaining the classification scheme and recovered by charging for access to the

information.

2.3.3.2 Class and Search Details are Integrated

With this approach the various sites can be viewed as a logical hierarchy as shown in Figure 2.6. A class-server is resident at every site (network) in much the same way as a web server. In Figure 2.6 these class servers are located at the bottom of the hierarchy. For instance, the De Montfort University (DMU) Class-server is on the DMU network and contain the class details made available publicly by DMU. Clearly the class servers are physical devices in the structure.

The class servers are grouped according to their geographical location and one member of the group is nominated as the group leader. In Figure 2.6 for instance, the DMU class-server is the nominated group leader for the Leicestershire region which is shown in the shaded box in Figure 2.6. Note that Figure 2.5 gives a more abstract view of this hierarchy.

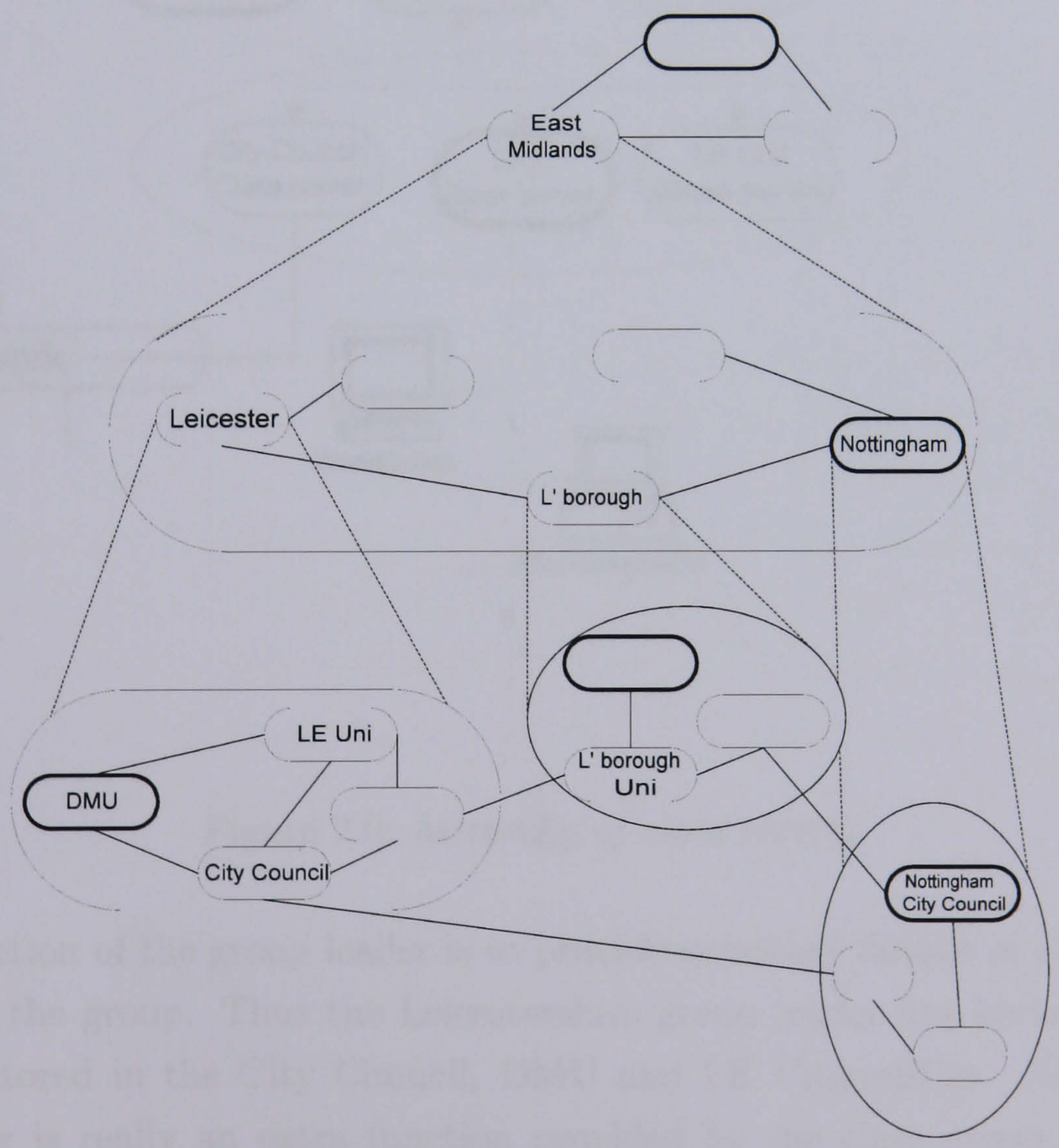


Figure 2.5: *logical class server hierarchy in physical network*

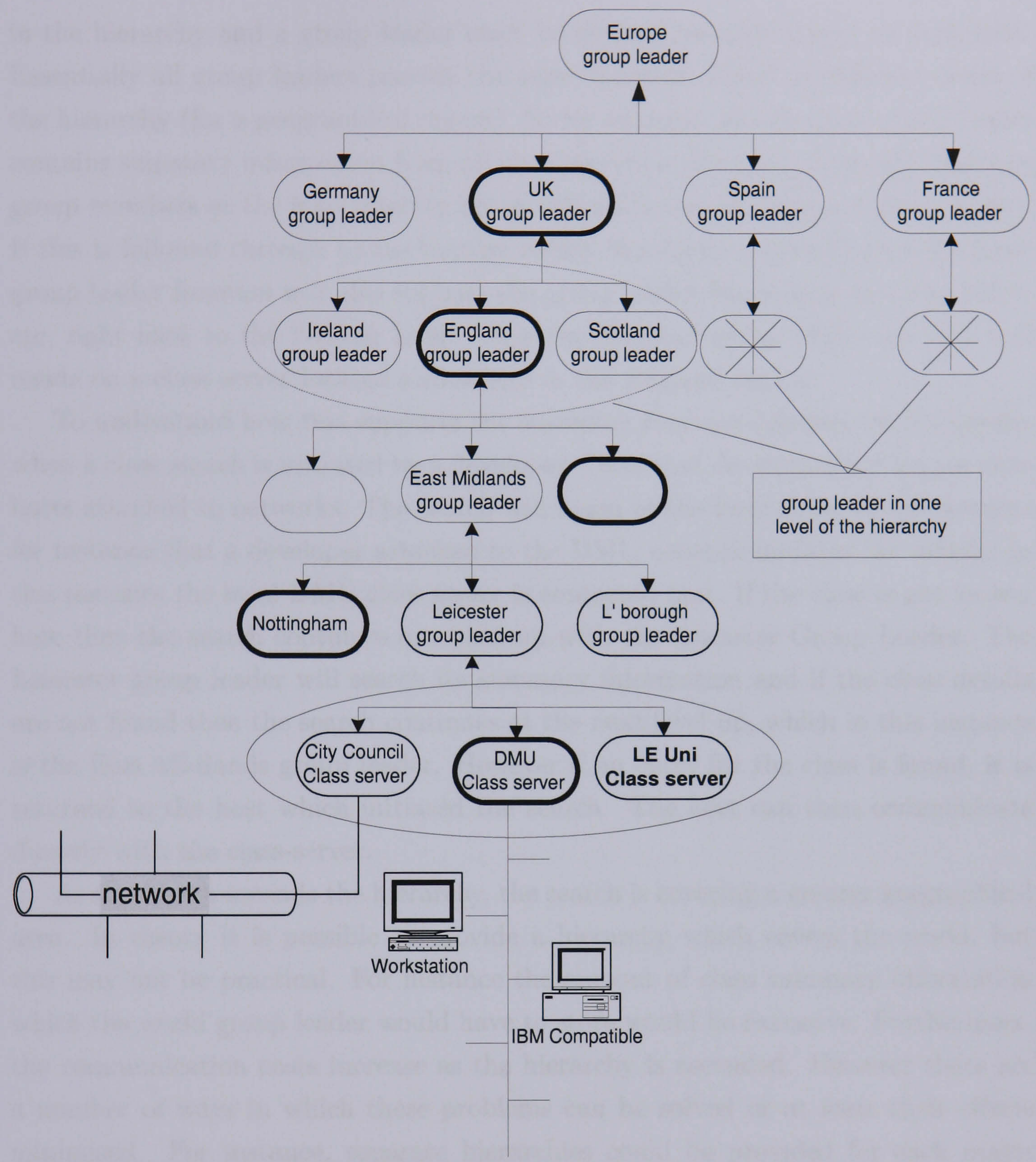


Figure 2.6: *hierarchy of class servers*

The function of the group leader is to provide summary details of all the classes available in the group. Thus the Leicestershire group leader has knowledge about all classes stored in the City Council, DMU and LE Universities. Note that the group leader is really an extra function provided by the class server, and clearly not all class servers will be group leaders. Any number of levels can be created

in the hierarchy and a group leader must be elected for each group, at each level. Essentially all group leaders provide the same function, albeit at different levels of the hierarchy (for a geographical region). So for example, the England group leader contains summary information from all class servers in England. Note also that the group members at the level directly below will nominate the England group leader. If this is followed through to the bottom clearly the device supporting the England group leader function will also support the group leader function at the level below etc, right back to the bottom level. Thus the England group leader function will reside on a class-server located somewhere in the England region.

To understand how this supports the searching function consider what happens when a class search is initiated by a developer. Note that developers are logged onto hosts attached to networks. The search will begin at the local class server. Assume for instance that a developer attached to the DMU network initiates the search. In this instance the local DMU class server is contacted first. If the class is not known here then the search continues one level up with the Leicester Group Leader. The Leicester group leader will search its summary information and if the class details are not found then the search continues at the next level up, which in this instance is the East Midlands group leader. However if an entry for the class is found, it is returned to the host which initiated the search. The host can then communicate directly with the class-server.

As the search ascends the hierarchy, the search is covering a greater geographical area. In theory it is possible to provide a hierarchy which covers the world, but this may not be practical. For instance the amount of class summary information which the world group leader would have to store would be excessive. Furthermore, the communication costs increase as the hierarchy is ascended. However there are a number of ways in which these problems can be solved or at least their effects minimised. For instance, separate hierarchies could be provided for each major function classification (discussed previously). Clearly separate group leaders would exist for each of these functions. The various function group leaders could be located on different class servers. This has the advantage of spreading the processing load and the number of levels required in the hierarchy will be reduced. One major advantage of the hierarchical method is that it is possible to limit the extent of the search. It is possible for instance to limit the search to at most 2 levels, or to a geographical region, for instance Europe. Hierarchies are similar to all tree like data structures and therefore have many other advantages. Indeed Coulouris [10] states

that hierarchical algorithms/structures are the only ones which are scalable.

2.3.3.3 Summary

This section reviewed two very different ways in which the class information could be structured in the distributed system to facilitate searching. Other structures are possible. The main objective is to choose a structure which will always allow class information stored in the distributed system to be found, efficiently.

2.4 Distributed Environment for Application Execution

As was mentioned earlier a distributed application comprises a set of distributed classes. Typically classes make calls on methods contained in other remote classes and thus the application can be viewed as a hierarchy; Figure 2.1 showed such a hierarchy. Before the application can be executed the classes comprising the application must first be located and then bound together. By binding in this context we mean the process of finding where in the network the classes are stored (and is defined as Network Address Binding (NAB)). There are two obvious approaches to binding. Binding can be carried out when the user invokes the application just before the application begins execution. Alternatively, binding can be carried out during execution. These two approaches are discussed and analysed below.

2.4.1 Binding

2.4.1.1 Network Address Binding at Invocation Time

Static binding is carried out just before execution of the application begins. Such as setting up a virtual circuit in a connection oriented network such as ATM [1]. In ATM, for instance, the connection is built before any information is transmitted and the user is given the opportunity of declining the connection (or renegotiating) if the application QoS cannot be guaranteed.

Clearly the name of all classes comprising the application must be known and this information can be extracted from the class source code, starting with the class at the top of the application hierarchy. The top class is typically the name of the class that the user provides for the runtime system. In Figure 2.1 for instance, the Class

TOPLevel makes calls to classes $Level_{11}$ and $Level_{12}$. Thus it needs to be identified if classes $Level_{11}$ and $Level_{12}$ reside on the local machine otherwise the location must be found in the distributed system. A *partial bind* table must be constructed by the class-server for the class *TOPLevel* with entries for class $Level_{11}$ and class $Level_{12}$ and their corresponding addresses. Class-servers for the second level classes (in this example class $Level_{11}$ and class $Level_{12}$) must be accessed next and the same process of constructing a partial bind table must be executed. This recursive process must be carried out until addresses have been found for all class references, at all levels. As a result of this binding process every class in the hierarchy will have identified all the remote class calls and corresponding class addresses, before the execution of the application and this information is stored in the set of distributed bind tables.

Note that the network conditions, just before the choice between each of the candidate classes is made, are known and, if there is a choice of candidate classes (replications stored at different locations) then the network conditions can be used to help in the selection process. However, it should be remembered that network conditions can change rapidly, thus it is questionable whether this information should be used in the class selection process, particularly if the application has a long execution time. One disadvantage of this scheme is that it delays the start of application execution. Furthermore, all class references must be resolved, but at run time not all classes are necessarily invoked. Thus this approach is potentially inefficient.

Note also that it is possible to bind at compile time. However, the disadvantage of this approach is that the time lapse between binding and execution may be long, meanwhile the class may have been removed from the location. It would not be practical to maintain bind tables to reflect the movement of classes.

2.4.1.2 Network Address Binding at Run Time

With this approach classes are located and bound at runtime. Execution of the application is started and a search for a class takes place only when it is invoked. The actual process of finding the class locations however is identical to that used in NAB at invocation time.

NAB at run time has a number of advantages. Firstly, only classes that are actually invoked are located. Secondly, because classes are found just prior to their invocation it is possible to take current network performance into consideration when choosing between copies of the same class located in different places. However, a

major disadvantage is that the application execution is interrupted while classes are found. It is not possible to bound (predict) this extra delay and therefore it is not possible to guarantee the application Quality of Service (QoS) requirements.

Problem Handling in Network Address Binding Whenever the location of a class has to be found to bind its network address there exist a problem if no corresponding location is found for this class. Obviously a distributed application will not function when one of the needed classes does not exist. In this case a message has to be sent to the service provider. The service provider might find another way of creating the service with the help of different classes with similar functionalities, otherwise the user of this service will be informed and the execution is cancelled. Furthermore the service provider will inform all involved parties within its knowledge. These parties, for example, are all location brokers responsible for this class as far as the service provider has knowledge about these brokers and class servers previously providing this class.

2.4.1.3 Summary

Because of the limitation with all of these structures, we integrate NAB at invocation time and NAB at run time. First an initial distribution is created at invocation time just before the start of the execution, and in case of any failure a reconfiguration is triggered. This reconfiguration initiates a new search and network address binding during execution.

Having defined how the searching for objects can be implemented and how they can be brought together to built one entity, we need to look what other aspects are needed for service-based computing.

2.5 User Requirements Through Contracts

Every user has different requirements. One might want a fast execution time and is willing to pay for this service, whereby another user prefers to get the service for the lowest possible cost, which automatically results in longer delays.

User's requirements in service-based computing paradigm need to be negotiated with the system through a contract in much the same way as that between utility providers and customers. A contract is modelled by a specialised state transition diagram [7] called a contract diagram (see Figure 2.7). The actions to be taken in every possible state of affairs are defined in contract clauses, including the contract's *fulfilment*, *cancellation* and *violation*. Each results in another state of affairs coming into effect, ultimately the termination of the contract. The clauses oblige, permit or forbid each party to perform processes in terms of the contract, where permission and prohibition are types of authorisation. The appropriate process may be instantiated in terms of the contract and, when completed, results in a new state of affairs.

The ability of XML [45] to structure a document, and to tag its contend with semantic labels, provides the opportunity to integrate policies and the process by which they are enforced. For example Listing 2.1 is a fragment of a contract with a user who might want to use Service A.

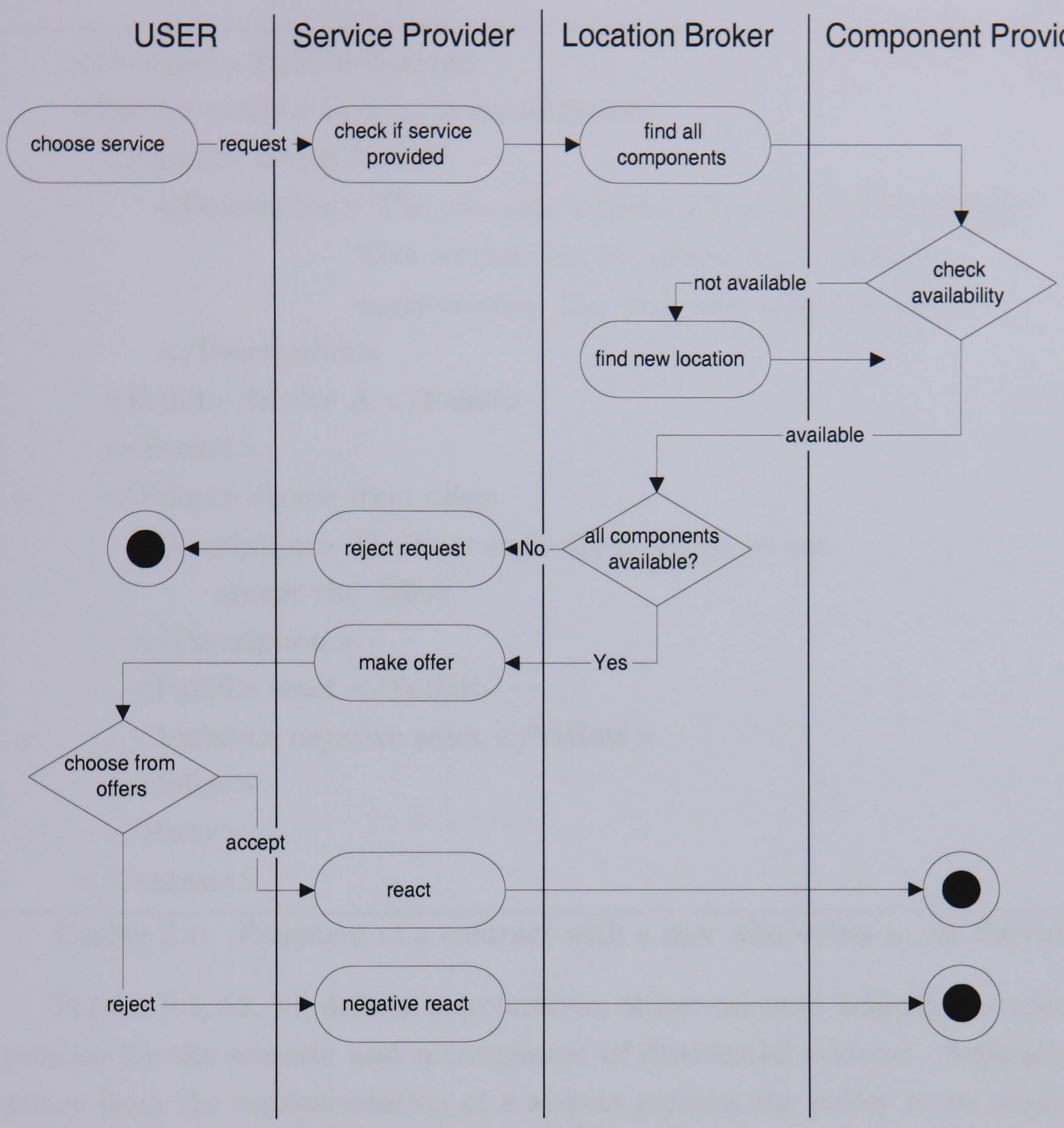


Figure 2.7: *contract diagram*

```
<Contract> Service Contract
<Party> entity.role.party.ServiceProvider
  <Permit> USER NAME
     <Description> The user may request if Service A is available.
                   This service can be offered with different
                   requirements, like cost and speed factors
     </Description>
  <Fulfill> Service A </Fulfill>
  </Permit>
  <Oblige> choose from offers
    <Description> The User shall either accept or not
          accept the offers
    </Description>
    <Fulfill> react </Fulfill>
    <Violate> negative react </Violate>
  </Oblige>
  </Party> ....
</Contract>
```

Listing 2.1: Fragment of a contract with a user who wants to use Service 1

Ponder [11, 12, 13] defines a declarative, object-oriented language for specifying policies for the security and management of distributed systems. Separating the policy from the implementation of a system permits the policy to be modified in order to dynamically change the strategy for managing the system and hence modify the behaviour of a system, without changing the underlying implementation. Ponder supports an extensible range of policy types, however only the two main important policies for this thesis are stated below:

Authorisation policies are essentially security policies related to access-control and specify what activities a subject is permitted or forbidden to do, to a set of target objects. Any request made by a subject can be defined in terms of an action on an object. A positive authorisation policy defines the action that a subject is permitted to perform on a target. A negative authorisation policy specifies the actions that a subject is forbidden to perform on a target. Negative authorisation are supported by many security platforms and can be

used to temporarily remove access rights from subjects. This way the access to a distributed service-based application can be controlled and managed. Example:

```
inst  auth+                    selectionProcedure{
      subject                  \user\sort;
      target<sortFunction>     \unlimited;
      action                   choose(), execute();
}
```

Listing 2.2: Ponder Positive Authorisation Example

Members of the domain sort, which is a subdomain of user are authorised to choose from a "table of contents" any function within the sortFunction domain and execute them on the remote location.

```
inst  auth-                    selectionProcedure{
      subject                  \user\sort;
      target<sortFunction>     \unlimited;
      action                   download(), remove(), disable();
}
```

Listing 2.3: Ponder Negative Authorisation Example

Members of the domain sort, which is a subdomain of user are forbidden to download, remove or disable any function within the sortFunction domain.

Obligation policies specify what activities a subject must do to a set of target objects and define the duties of the policy subject. Obligation policies are triggered by events and are normally interpreted by a manager agent at the subject. These obligation policies can help to built a service-based application because every possible action occuring during execution can be specified. As such the runtime behaviour can be fully defined. Negative obligation policies are not equivalent to negative authorisations. The main difference lies in the fact that obligation policies are interpreted by subjects while authorisation policies are interpreted by access control components on the target host. Thus, negative obligation policies act as subject based filters specifying actions that managers 'must refrain' from performing.

Example:

```
inst  oblig                 reconfigure{
      on                    timeout(locationA.objectB);
      subject               s=PerformanceMonitor;
      target<children>      t=locationA.objectB.child[],
                            size=locationA.objectB.child.size();
      do                    for(int i=0; i<size; i++){
                                locationA.objectB.child[i].stop();
                            }
                            locationB.objectB.start();
}
```

Listing 2.4: Ponder Obligation Example

This policy is triggered when a timeout occurs for objectB at location A. The PerformanceMonitor will stop the execution of all objects originally instantiated by objectB at locationA and will restart objectB on the new locationB.

2.6 Summary

This chapter has introduced the different perspectives a service-based computation can be viewed of. In addition it states the issues needed for service based applications compared to traditional centralised applications. It introduces an example structure of such an application and two ways to implement the searching for application parts have been proposed. Once the distributed objects have been found their network addresses have to be bound together to build one entity. Three different architectures have been introduced to bind the network addresses. Finally it is stated how user requirements can be specified through contracts to specify the requirements for the performance monitoring introduced in chapter 3.

Chapter 3

Architectural Design for Performance Monitoring and Reconfiguration

3.1 Introduction

The performance of an application must be monitored during execution to ensure that the required Quality of Service (QoS) is satisfied. Failure to meet the necessary QoS will necessitate a reconfiguration (or partial reconfiguration) of the application. This may occur, for instance, if a class server fails and a replication of the class located at a different class server must be substituted. Thus performance monitoring and reconfiguration are inextricably bound together. In the following we discuss various architectures that can support program execution, performance monitoring and reconfiguration.

We distinguish between two separate but related application performance monitoring activities that must be undertaken in a distributed environment. The Performance Monitoring-Client (PMC), which invokes another class, must implement the performance monitoring activity. In contrast, the Performance Monitoring-Server (PMS) which is invoked, plays a more passive role and is merely expected to respond to requests and instructions issued by the PMC. For instance the PMC must have the functionality to interrogate the PMS server side, for example, to find out current values of given parameters. Furthermore, the PMC must be able to send commands to the PMS, for instance a command to change the execution priority. The PMC

can, in theory, be located anywhere. However, because the PMS must be able to respond to requests and commands from the PMC, clearly it is best that it resides on the class server where it is executing. In this thesis, the PMS functionality is included in the **Additional Support** for each class. The Additional Support as shown in Figure 2.1 contains information about the functional classification, location of invoked classes and the PMS, since each PMS is only responsible for the execution of one particular class. A number of alternative architectures for the PMC are presented below.

3.2 Centralised Model for Performance Monitoring-Client

As mentioned earlier the classes comprising a distributed application are located and executed at class servers throughout the distributed environment. One simple way in which the PMC can be designed in this environment is to locate it at the host that invokes the application; Figure 3.1 presents this arrangement. In particular, it shows the PMC integrated with user host. The reason to place the PMC on the user-host is, when the host with the PMC fails the whole application would fail. In case this failure is on the user host s/he will notice and has to restart anyhow. Another alternative is to have one or better two replications of the PMC, synchronised with the original and able to overtake the whole functionality in a failure case. There exist numerous techniques in the literature on how to mirror hard disks e.g . Baek et al [5] has presented a hierarchical Redundant Array of Independent Disks (RAID) architecture with multiple controllers. It can offer performance and reliability without exhaustive disk utilization, thus annihilates the drawbacks of traditional disk arrays.

Furthermore it can be seen in Figure 3.1 and Listing 3.1 that the user first negotiates a service with a service provider. After this the two parties have agreed on a contract the service provider contacts one or more location broker to find all needed class locations. The found network addresses are bound together (NAB at invocation time as described in section 2.4.1) and saved in a bind table. Then the execution is instantiated by contacting and sending the bind table to the PMC on the user host. This PMC is responsible to control and monitor the execution of the whole application.

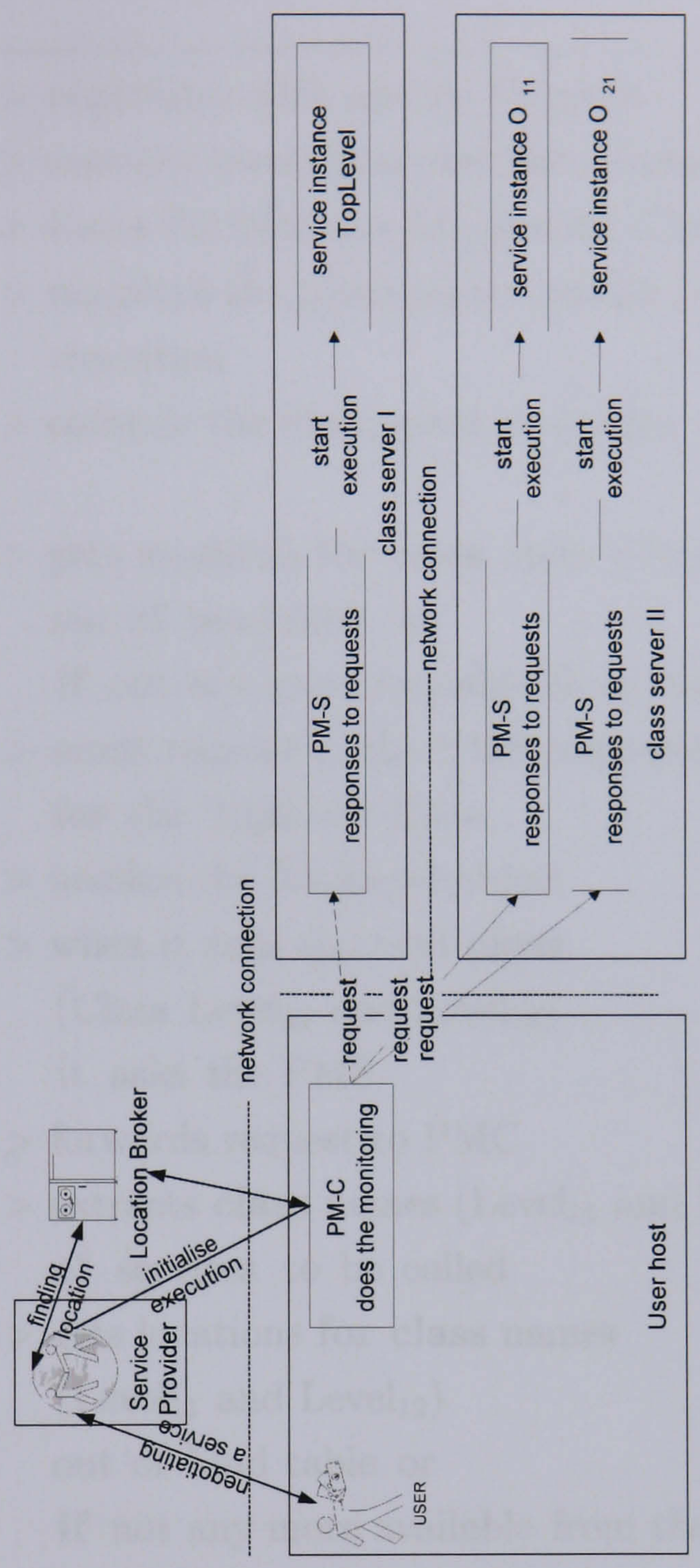


Figure 3.1: *Interaction between the PMC and PMS in centralised model*

```
user                  -> negotiates with service Provider
service provider      -> contacts location broker for all class locations
service provider      -> starts Performance Monitoring Client on user's host
PMC                   -> monitors the performance within the distributed
                         execution
PMC                   -> controls the distributed execution

PMC                   -> gets locations for class name (TopLevel)
                         out of bind table or
                         if not any more available from the location broker
PMC                   -> sends request to the PMS responsible
                         for the TopLevel Class
PMS                   -> invokes the TopLevel object
TopLevel object       -> when it calls the next class
                         (Class Level11 and Level12)
                         it asks the PMS
PMS                   -> forwards request to PMC
PMC                   -> extracts class names (Level11 and Level12)
                         of services to be called
PMC                   -> gets locations for class names
                         (Level11 and Level12)
                         out of bind table or
                         if not any more available from the location broker
PMC                   -> sends request to the PMS responsible
                         for the Class Level11 on class server II
PMC                   -> sends request to the PMS responsible
                         for the Class Level12 on class server IV

and so on ...
```

Listing 3.1: Centralised Model Scenario

When taking the example application shown in Figure 2.2 the PMC extracts details of the TopLevel class from the bind table (class name and location) sends a request to start the execution for the TopLevel class to the responsible PMS on class server I (using RMI techniques). The PMS invokes the TopLevel object, which starts executing. As soon as this object reaches the point where it would have to

call objects Level_{11} and Level_{12} it sends a message to its PMS with this request. The PMS forwards this request to the PMC which gets the location for these objects out of its bind table. It then sends a request to the PMS responsible for class Level_{11} on class server II and in parallel a request to the PMS responsible for class Level_{12} on class server IV. Both these PMSs will invoke the corresponding objects and so on. In this way, all class invocations are made centrally by the PMC.

The PMC typically will be implemented as a separate program and thus will run as a process on top of the operating system. Figure 3.2 shows a wide area network environment in the form of a set of class servers interconnected via a mesh of WAN routers. Note that the classes incorporate Additional Support which, as stated earlier, includes the PMS. This interface augments the class with the ability to respond to requests from the PMC for performance information. Figure 3.2 also shows how the classes comprising the example application shown in Figure 2.1 can be distributed across class servers. For instance one class server supports classes O_{11}, O_{21}, O_{31}. In this example, the classes for the example application have been distributed over 6 class servers as shown in Table2.1, but clearly any pattern of distribution is possible. Figure 3.3 gives a more detailed view of a class-server and attempts to clarify a number of points. Such as the close connection between the PMS and the object. The PMS is part of the class and communication from/to other classes is under the control of the centralised PMC.

3.2.1 Advantages

The function of the PMC relates specifically to a single execution of a given application. Thus the PMC has an overall view of the performance of the application and thus can more easily assess whether the performance satisfies the QoS requirements specified by the user at run time. Furthermore, failure recovery and maintenance are under central control.

3.2.2 Disadvantages

Much communication is involved; this can be seen visually in Figure 3.2. In particular, all classes must report back to the PMC when execution has finished (or when there are problems). Clearly the communication overhead depends on both the location of the classes relative to the PMC function together with the number of performance messages which must be exchanged during execution. The invocation

of classes is also inefficient because the class server issuing the invocation must filter it through the PMC. In case the PMC fails and no replication exist, all information will be lost.

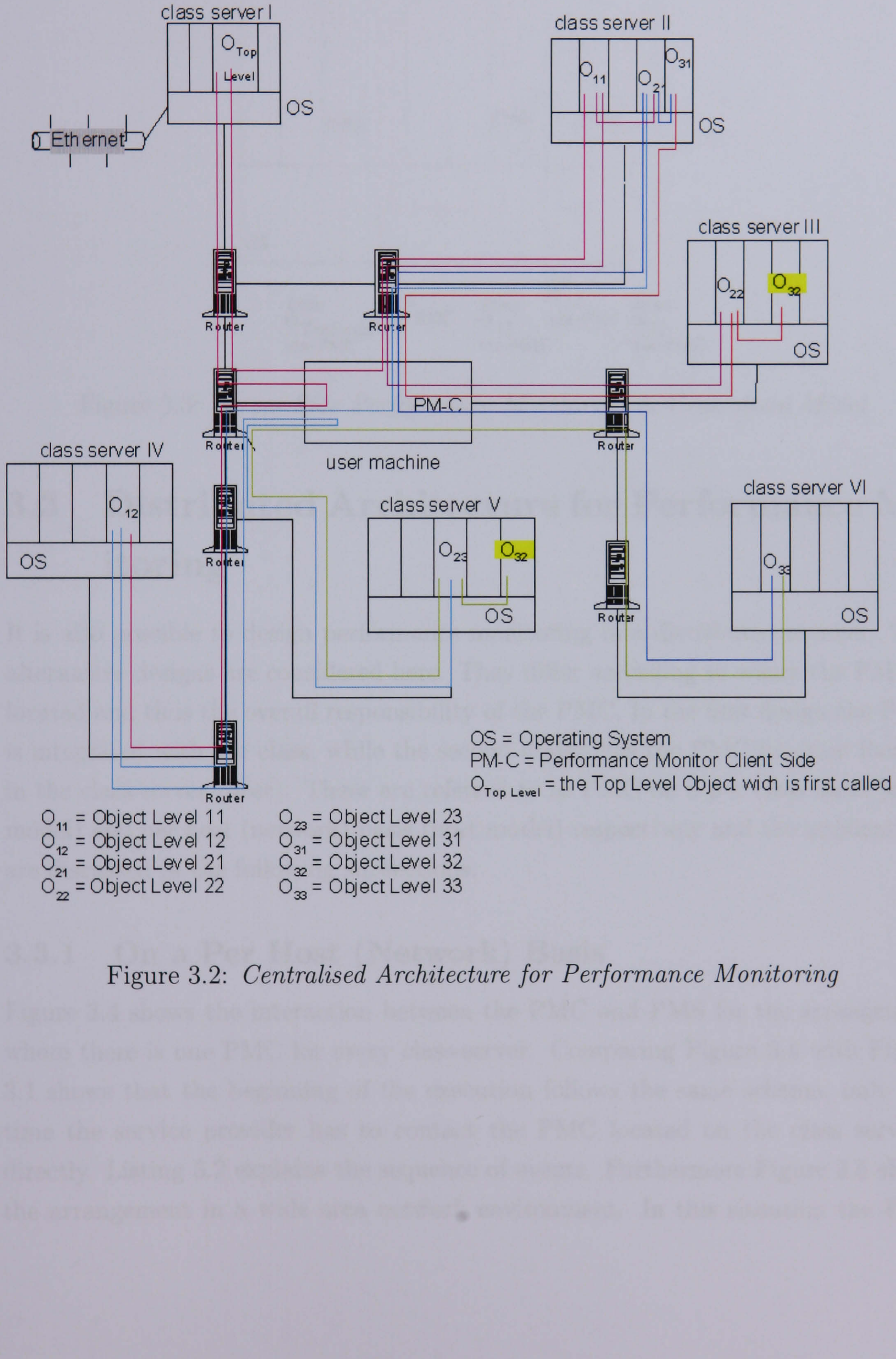


Figure 3.2: *Centralised Architecture for Performance Monitoring*

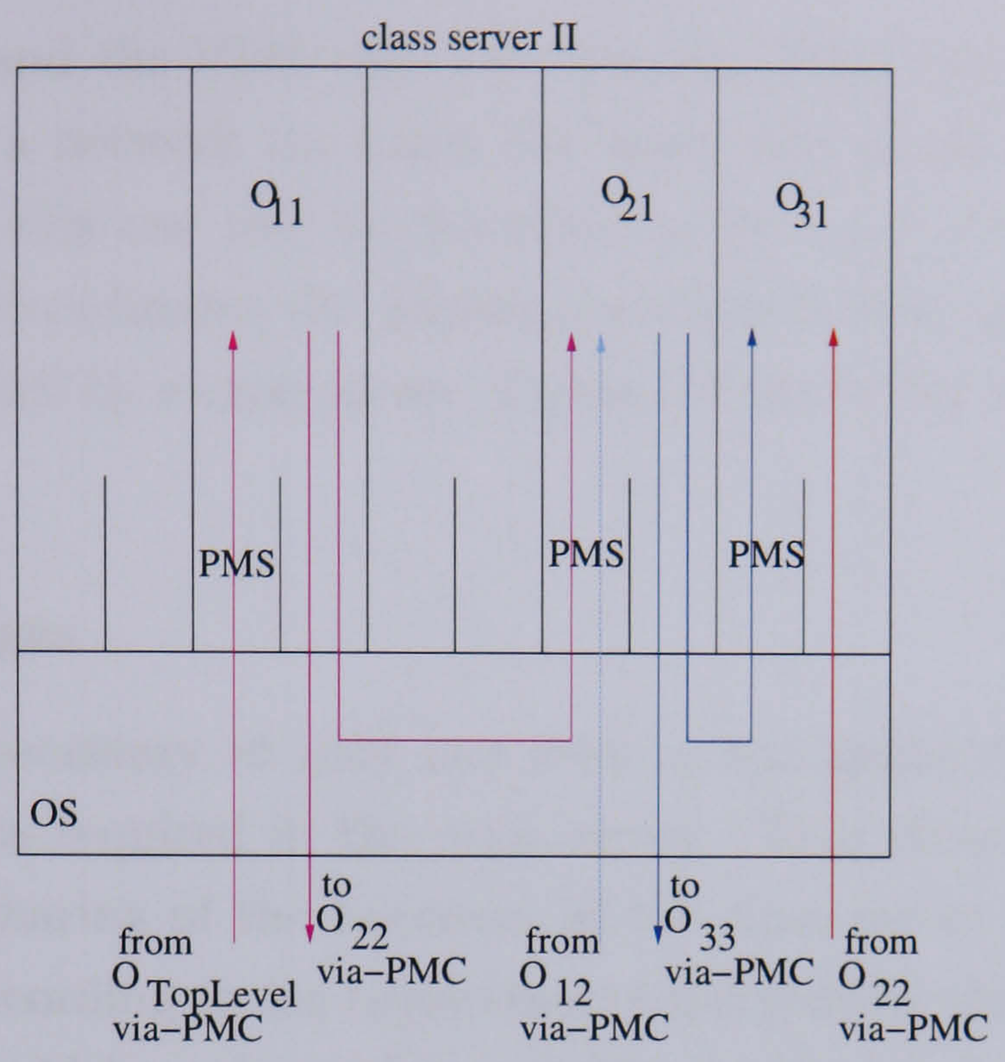


Figure 3.3: *Server Side Performance Monitoring in Centralised Model*

3.3 Distributed Architecture for Performance Monitoring

It is also possible to design performance monitoring in a distributed manner. Two alternative designs are considered here. They differ according to where the PMC is located and thus the overall responsibility of the PMC. In the first design the PMC is integrated with the class, while the second design has the PMC function located in the class-server (host). These are referred to as PMC on a per class basis (class model) and per host (network) basis (host model) respectively and the implications are discussed in the following subsections.

3.3.1 On a Per Host (Network) Basis

Figure 3.4 shows the interaction between the PMC and PMS for the arrangement where there is one PMC for every class-server. Comparing Figure 3.4 with Figure 3.1 shows that the beginning of the execution follows the same schema, only this time the service provider has to contact the PMC located on the class server I directly. Listing 3.2 explains the sequence of events. Furthermore Figure 3.5 shows the arrangement in a wide area network environment. In this situation the PMC

runs as a **process** and the PMS runs as a thread. Note however, if there is only one class server on a network (in much the same way as there is one web server on a network) then this can also be described as on a per Network basis. In this instance the PMC co-ordinates the performance monitoring activity for all remote class invocations issued by a class server. Figure 3.6 shows the class-server in greater detail.

3.3.1.1 Advantages

Communication is necessary at only one level in the application hierarchy. Only one PMC function is required in the class server. This allows the PMC function to co-ordinate the sharing of the resources of the class server between the various executing classes, according to the requirements specified by the user. Note that the operating system could be enhanced to provide this function. As long as there is no reconfiguration required, the monitoring will not create any additional network traffic.

3.3.1.2 Disadvantages

The major disadvantage of this design is that the PMC has to co-ordinate classes associated with different applications. No global knowledge relating to the application is available. Furthermore the first host accommodating a PMC in this application hierarchy creates a weak point for failures. When this machine fails there is no reconfiguration possible unless this PMC is mirrored to at least one different location, so that in case the first machine fails the mirrored PMC can take over the monitoring smoothly.

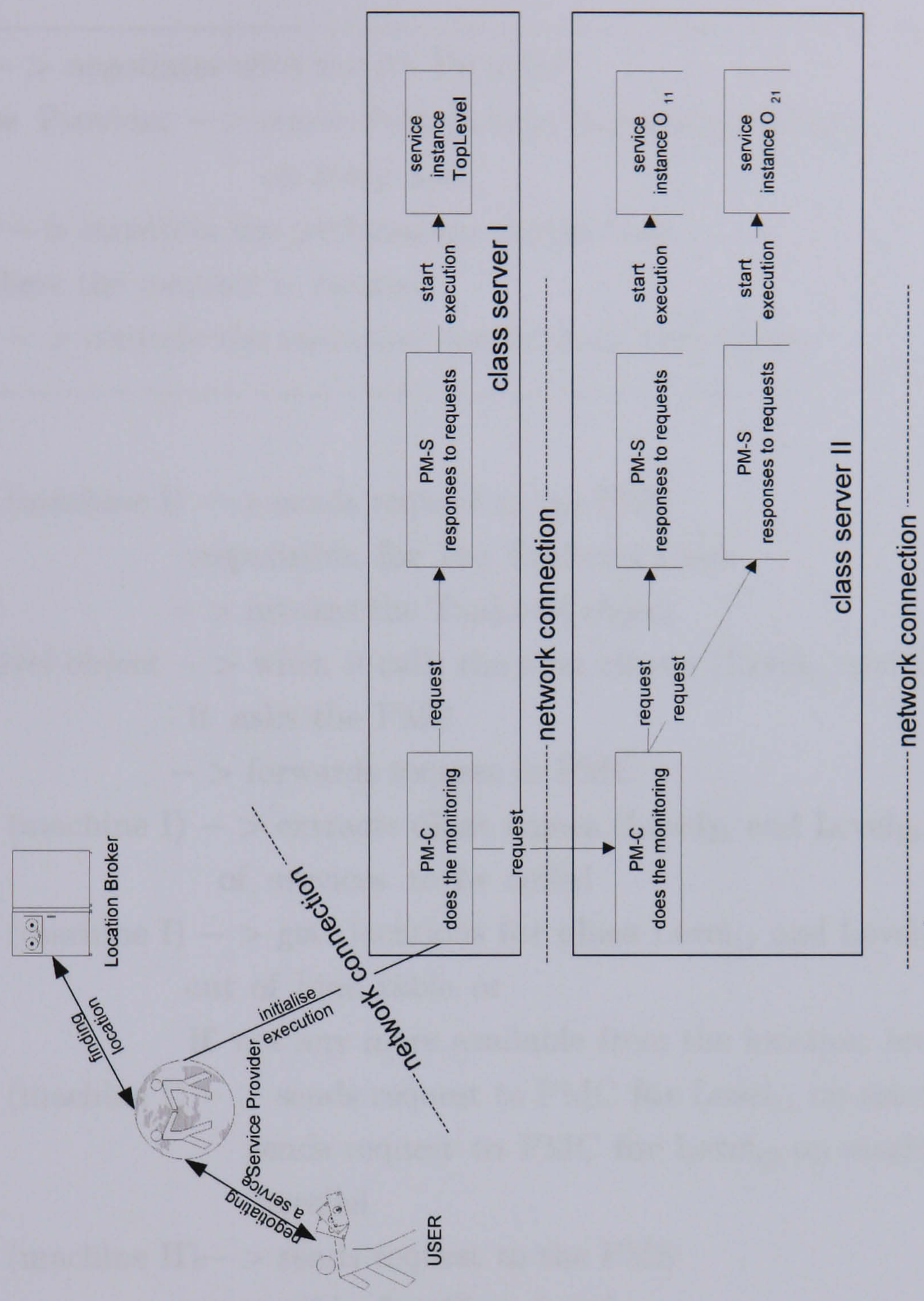


Figure 3.4: *Interaction between the PMC and PMS in host model*

```
user -> negotiates with service Provider
service Provider -> starts Performance Monitoring Clients
                    on every host
PMC -> monitors the performance on the host
    where the monitor is located
PMC -> controls the execution started from this client
-------------------------------------------------

PMC (machine I) -> sends request to the PMS
                   responsible for the TopLevel Class
PMS             -> invokes the TopLevel object
TopLevel object -> when it calls the next classes (Level11 and Level12)
                   it asks the PMS
PMS             -> forwards request to PMC
PMC (machine I) -> extracts class names (Level11 and Level12)
                   of services to be called
PMC (machine I) -> gets locations for class Level11 and Level12
                   out of bind table or
                   if not any more available from the location broker
PMC (machine I) -> sends request to PMC for Level11 on machine II and
                   sends request to PMC for Level12 on machine IV in
                   parallel
PMC (machine II)-> sends request to the PMS
                   responsible for Class Level11

PMC (machine IV) -> sends request to the PMS
                   responsible for Class Level12

and so on ...
```

Listing 3.2: Host Model Scenario

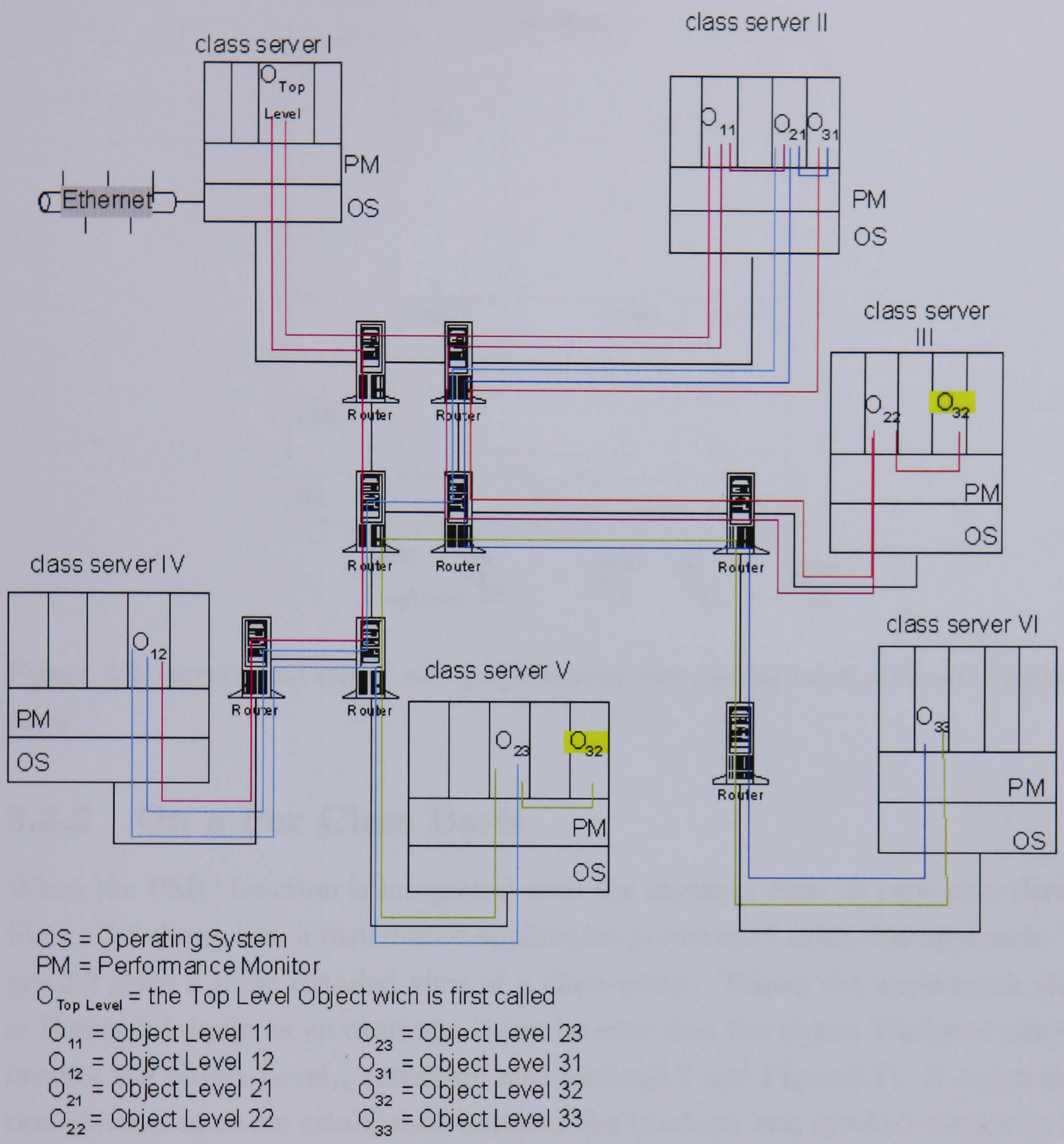


Figure 3.5: *Distributed Architecture for Performance Monitoring on a per host-(network) Basis*

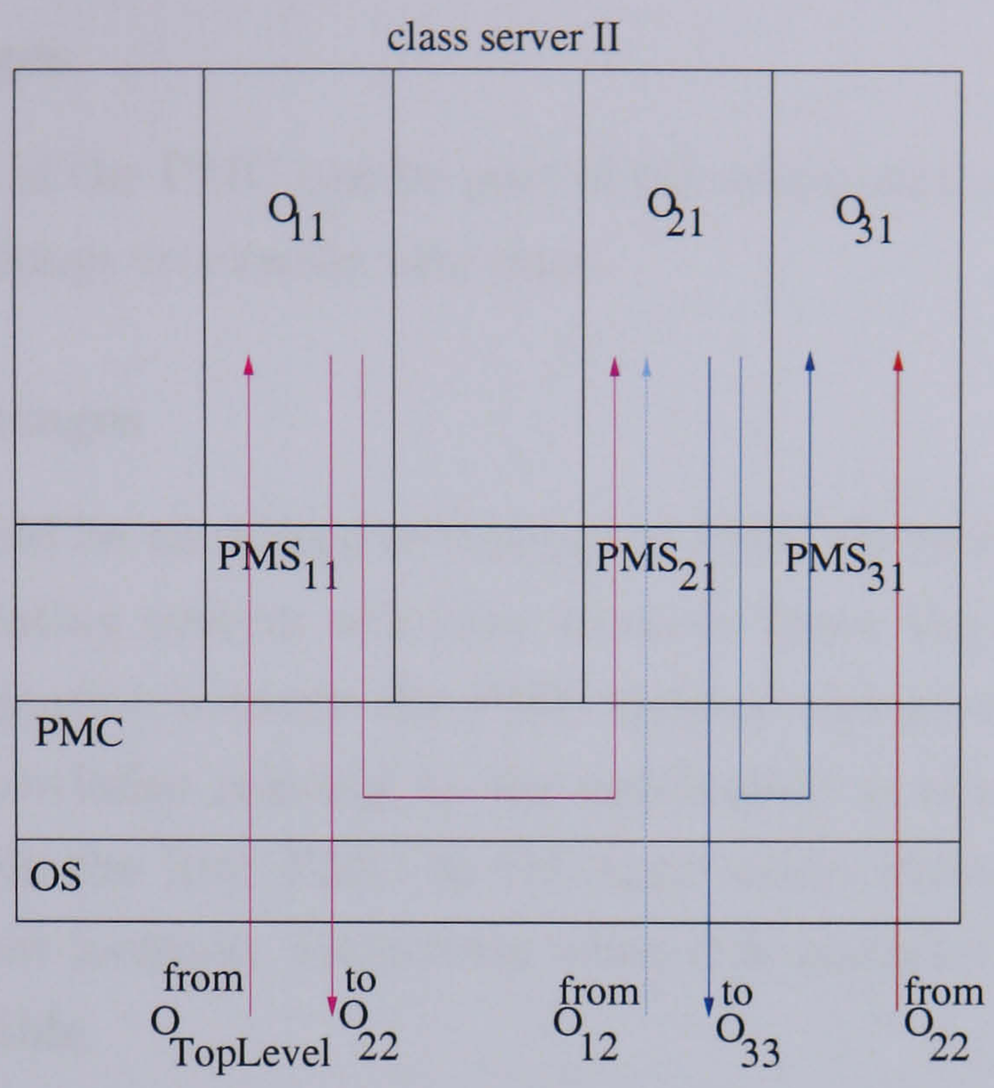


Figure 3.6: *server and client side performance monitoring on a per host- (network) basis*

3.3.2 On a Per Class Basis

When the PMC function is integrated with the invoking class, it runs as a **thread**. Figure 3.8 shows how a distributed application is executed using this approach. Figure 3.9 gives a more detailed view of a class-server. Taking the application shown in Figure 2.1 again as an example, it can be seen that the object TopLevel remotely invokes the object Level_{11} (compare also Listing3.3 and Figure3.7). A communication channel must be established between the invoking and invoked classes so that message relating to the performance can be exchanged between them. This is shown in Figure 3.8 (the pink connection at the top of the diagram). Note that when a class invokes another class on the same class server then no communication channel is needed. For instance O_{11} invokes O_{21}. With this arrangement both the application execution and the performance monitoring is distributed. Note also, in this instance the bind table is also distributed. Listing 3.3 gives this example in pseudo code.

3.3.2.1 Advantages

The implementation of the PMC can be part of the additional support for each class, since every PMC belongs to exactly one class.

3.3.2.2 Disadvantages

Potentially there could be an excessive number of PMC threads executing in a class-server, and the operating system will have to co-ordinate the sharing of resources. There is no communication between the PMC threads and no co-ordination between them. No global knowledge relating to the application is available. As well as in the other two models the first PMC in the application hierarchy will have to be mirrored to a different location. Otherwise when this machine fails there will be no reconfiguration possible.

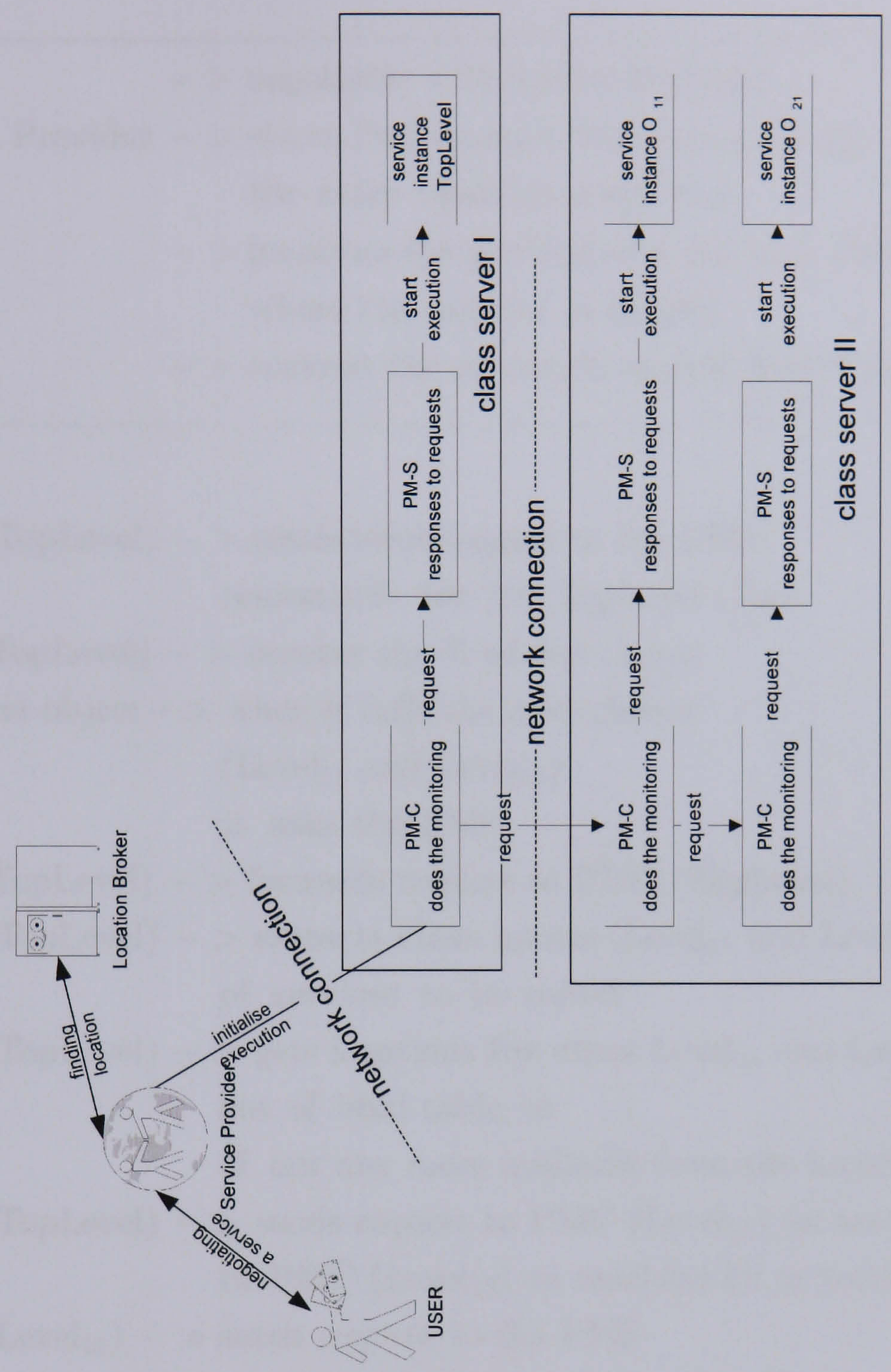


Figure 3.7: *Interaction between the PMC and PMS in class model*

```
user                    -> negotiates with service Provider
service Provider        -> starts Performance Monitoring Clients
                           for every class on every host
PMC                     -> monitors the performance for each class
                           where the monitor is located
PMC                     -> controls the execution started from this client
-------------------------------------------------------------------

PMC (TopLevel)          -> sends local request to the PMS
                           responsible for the TopLevel Class
PMS (TopLevel)          -> invokes the TopLevel object
TopLevel object         -> when it calls the next classes
                           (Level11 and Level12)
                           it asks the PMS
PMS (TopLevel)          -> forwards request to PMC (TopLevel)
PMC (TopLevel)          -> extracts class names (Level11 and Level12)
                           of services to be called
PMC (TopLevel)          -> gets locations for class Level11 and Level12
                           out of bind table or
                           if not any more available from the location broker
PMC (TopLevel)          -> sends request to PMC (Level11) on machine II and
                           to PMC (Level12) on machine IV in parallel
PMC (Level11)           -> sends request to the PMS
                                responsible for the Level11 Class
PMC (Level12)           -> sends request to the PMS
                                responsible for the Level12 Class
PMS (Level11)           -> invokes object Level11
PMS (Level12)           -> invokes object Level12
and so on ...
```

Listing 3.3: Class Model Scenario

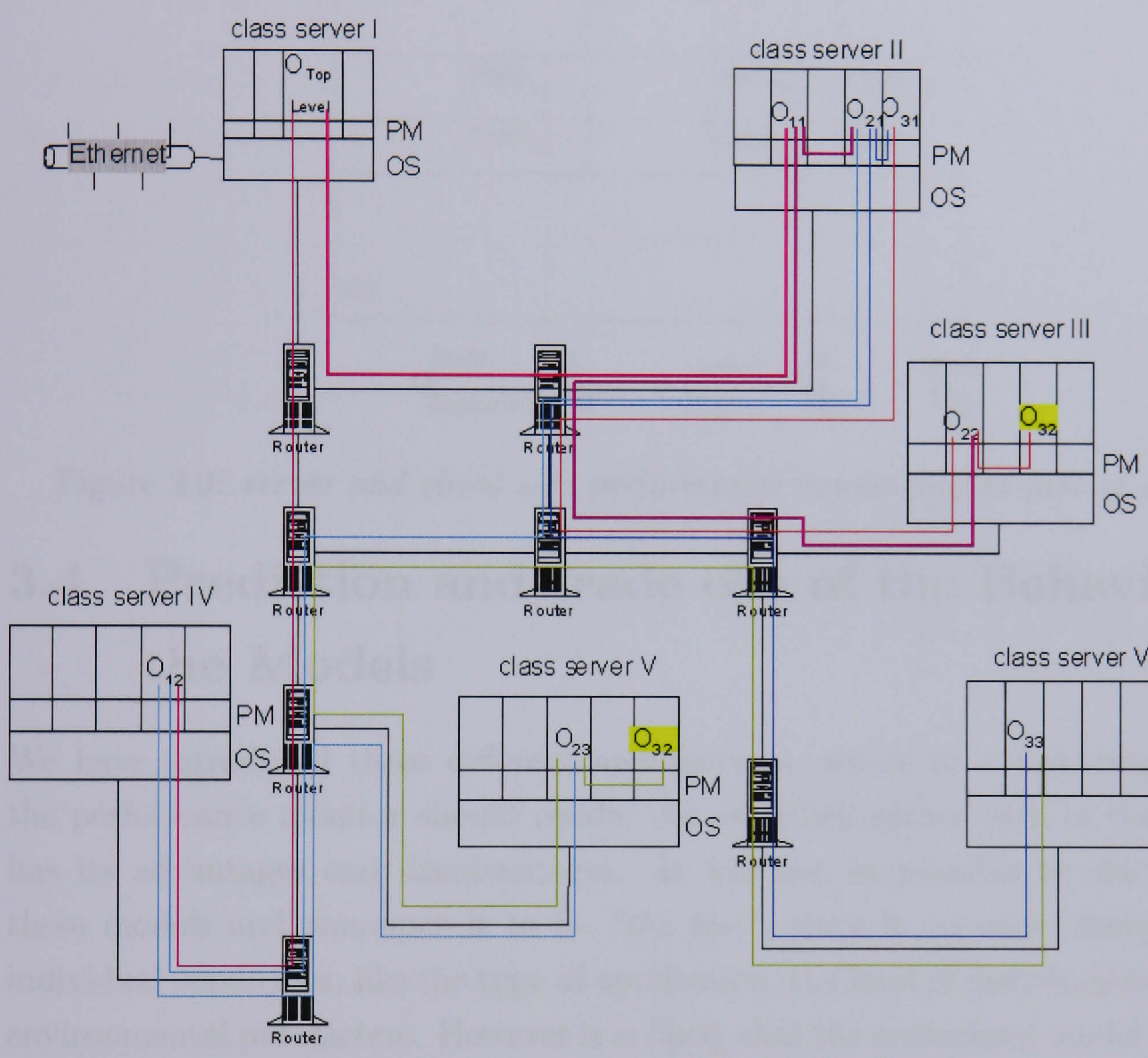


Figure 3.8: *Distributed Architecture for Performance Monitoring on a per class basis*

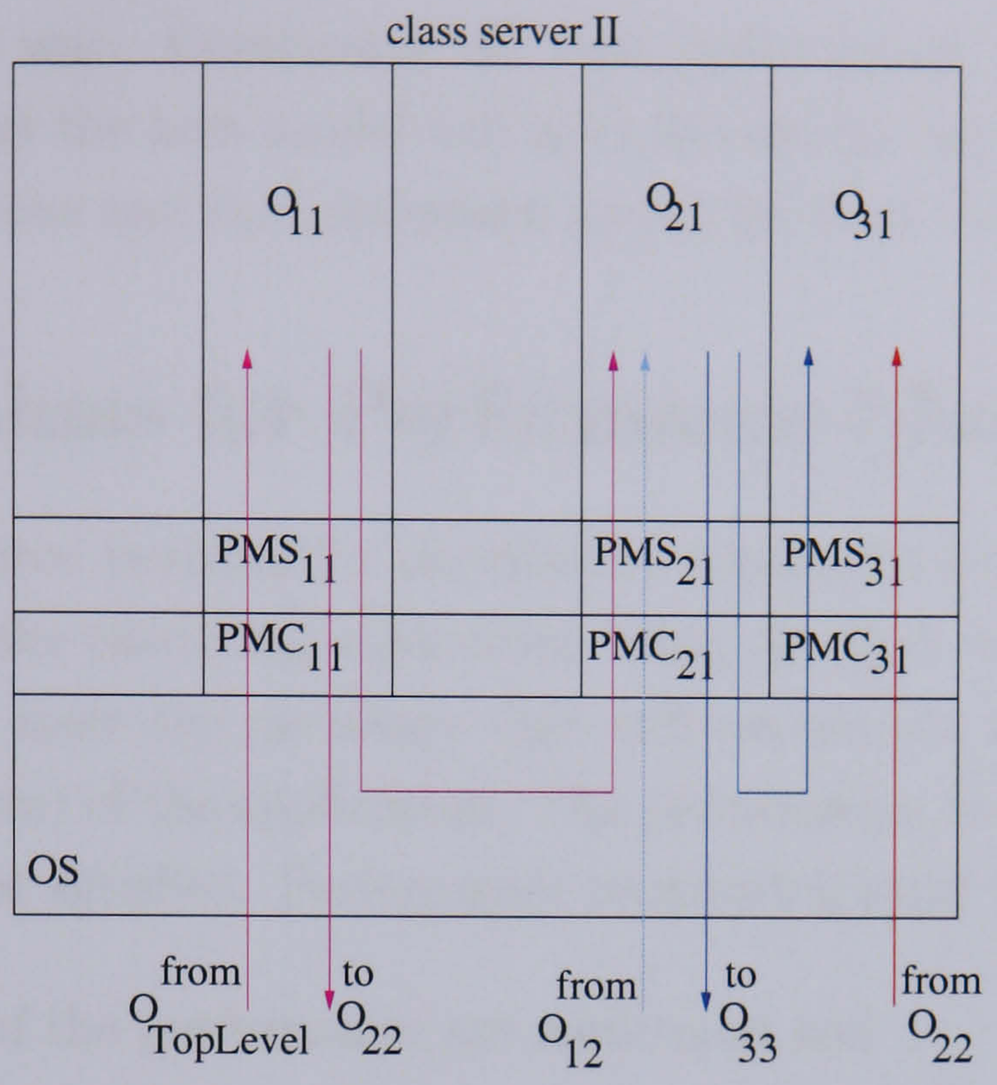


Figure 3.9: *server and client side performance monitoring as part of an class*

3.4 Prediction and trade offs of the Behaviour of the Models

We have introduced three different architectures, where in a distributed system the performance monitor should reside. As described earlier each of these models has its advantages and disadvantages. It will not be possible to choose one of these models and announce it to be *"the best"*, since it depends strongly on the individual conditions, like the type of application, the level of distribution and other environmental parameters. However it is likely that the centralised model becomes a big bottleneck for physically widely distributed applications. Furthermore the host model should not create any extra traffic and performance monitoring is feasible for every kind of application, whereby not as accurate as the centralised model, because each monitor has no overview of the whole application. Last but not least the class model loses a further part of the overview of the whole application and as such has to rely even more on information about failures from outside its own PMC. The only advantage for this model is, no extra installation is required, since the PMC can be included in the additional support of each class.

The reason for the different performance monitoring architectures is to find an appropriate architecture for an application. It is envisaged that not all three models

behave in the same way. Comparing the past performance versus future performance it is likely that the host model will have the fastest response time for widely distributed applications and the centralised model for local execution.

3.5 Algorithms for Performance Monitoring

A performance monitor (written for distributed applications) specifically monitors the performance of the executing application using the QoS defined by the user at runtime. Failure to meet the necessary QoS will necessitate a reconfiguration (or partial reconfiguration) of the application. The performance is considered unacceptable if the QoS is not satisfied. Performance monitoring must consider:

- what aspects of the performance are monitored and
- how this can be designed.

The system must also allow for the application to be partially reconfigured (which will typically mean that under-performing objects are substituted) while it is running.

3.5.1 Measurements and their Implementation

When a service has a level of real time constraints, there are a number of non-functional as well as functional requirements to be considered and monitored. As for the non-functional requirements it is to monitor if an object is available and how reliable it will be over the next period of time.

In addition a user of such a service-based application will have some functional requirements. In contrast to an application installed on the local machine, a service-based application will bring a different perspective on how to calculate the cost a user (human or application) will have to pay to use this service. In addition the user will be interested in some kind of feedback. One might want to know of the progress, whereby another user, just wants to get the results as soon as possible and is not interested in this detailed information. All this is negotiated within the contract described in section 2.5. As soon as a network is involved in the execution a factor of delay is created and needs to be monitored especially when real time critical services are involved.

The factor of delay can be separated in number of remote calls served, the corresponding network speed for these remote calls, processing capabilities of the remote servers, magnitude of inter process communication, and an overall worst case execution time.

In order to define the different techniques for measuring performance, we need to differentiate between the time frames used for measuring the parameters. This implies that measurement implementations for each parameter, before execution, will be different from the implementation used, during execution of the remote calls.

Availability In case a host machine no longer provides a particular service, it will be unregistered at the location broker responsible for this service. If this happens prior to execution an alternative location is chosen. However when it happens during execution a penalty for the provider of the host machine occurs, since the contract is broken. Furthermore the location broker is able to inform every performance monitor monitoring this particular service on the particular host and an alternative location can be chosen.

However whether a host providing a service is available, prior to execution, is easily discovered out when the performance monitor sends a request "Are you alive?" In case an acknowledgement comes back the execution can start otherwise a replication of this service has to be detected. However even when a service was available prior to execution it might fail during the execution. There are three different types to monitor the availability during execution. The traditional way is not an option in a large global system, because it would create far too much unnecessary traffic. For this instance a kind of watchdog would keep sending requests to the service and wait for the reply. As soon as this reply is missing the watchdog would know that the service is not any more available and has to find a replacement. The second option is a very cheap implementation, which can be used as long as no time critical applications are involved. In this case the performance monitor starts a timer as soon as a service is called. When the reply does not arrive within the limit a replication of the same service has to be started. Listing 3.4 shows this algorithm in pseudo code. In addition every performance monitor will register with the location broker responsible for the services used. This location broker detects early whether a machine has failed and sends a message to all listening performance monitors, to inform about the failure of a host machine. A pseudo code demonstrating this scenario can be seen in Listing 3.5. Finally a more expensive but very accurate option,

for time critical application is called N-version programming [2]. This solution implies starting replications of the same service on different location at the same time. The results are taken from the service finishing first, as Listing 3.6 demonstrates. However the performance of every single machine used will go down as we find in chapter 4. To improve each of these solutions breakpoints [21] can be injected inside of the object code and partial results can be created. It will then be possible to restart an object right from the last valid state within it.

```
performance monitor -> start timer A
performance monitor -> start remote service 1
service 1           -> execute
<the machine I where service 1 is executing fails>
performance monitor -> timer A expires
performance monitor -> start reconfiguration
```

Listing 3.4: Failure detection triggered by time-out

```
performance monitor -> start timer A
performance monitor -> start remote service 1
service 1           -> execute
<the machine I where service 1 is executing fails>
location broker     -> detects failure of machine I where service 1 is
                       located
location broker     -> send message to all performance monitors
                       interested in machine I
performance monitor -> receives message machine I has failed
performance monitor -> start reconfiguration
performance monitor -> ...(find alternative location
                       start timer B
                       start service 1 on machine II)
performance monitor -> timer A expires
performance monitor -> reconfiguration already started
                    -> timer A is ignored
```

Listing 3.5: Failure detection triggered from location broker

```
  performance monitor -> start service 1 on machine I, II and III
                                  in three  different  threads
  <service 1>            -> execute on machine I, II and III
        <machine I fails>
5 <service 1>            -> still executing on machine II and III
  performance monitor -> receives results of service 1 from machine III
  performance monitor -> stopps the execution of service 1 on machine II
```

Listing 3.6: Failure handling for time-critical applications

Reliability Failure can occur, either due to software or hardware failure. Here we assume only hardware failure. The reliability of a host machine prior to execution can be predicted on the basis of statistical data about the failure of the object maintained by the service provider. During every execution, these records can be updated and used for the next invocation of the object. In addition to detect a delay early during runtime, the estimated start and end time for every object is compared with the actual time by the performance monitor, just before the call takes place. Whereby it is to bear in mind, that here exist a problem of different times around the globe and a synchronisation of timeservers will have to be implemented. Mills [34] explains the Network Time Protocol (NTP) used for Internet Time Synchronisation and Lamport [27] describes one example to implement a time server by synchronising all the node clocks, using the available information about Universal Time.

Before the start of the execution the distribution and estimated execution time for every object is calculated. To reduce the execution time for the whole application it is advisable to execute parts in parallel. However for objects executing in parallel, it is not always advisable to use the fastest available machine, since every execution will decelerate the machine speed (further information are given in Chapter 4), this results in a requirement to distribute the parts of the application running in parallel. The algorithm to calculate the estimated execution time for every object is a challenging problem, since the behaviour of a machine, when changing the load, needs to be predicted in advance.

For software failures, various techniques forward and backward error recovery may be adopted [39].

Number of Remote Calls The number of remote calls made by an application prior to execution can be estimated in two ways. One of these is using statistics maintained by the location broker for an earlier execution of the same distribution. The other way is a prediction by the location broker, of the number of remote calls possible for a particular object invocation. Both of these solutions are only manageable with a certain level of precision. The location broker knows every object location, however it is not able to predict the exact behaviour during runtime. Some decisions like which object is to be called next are only made during runtime e. g. one object calls the next only in case of one event happening and in case of another event a third object will be called. Furthermore it might be necessary to reconfigure the application during runtime by using replicated objects on different locations and the whole structure of distribution changes. This would automatically change the number of remote calls. However calculating the number of remote calls during runtime is complex and an easy way to do it is to increment a counter for every remote call made and review these statistics after the execution is over.

Network Delay The network delay can be measured as a function of delay between the call made and results obtained. Prior to execution, network delay can be predicted on the basis of data collected from previous invocations. Chapter 4 describes how to estimate the execution time for every object used within the application. The same estimation can be used for the delay created by a network connection. During execution, the PMC saves the time before it calls a remote object and as soon as this call return. The difference between these two times is the execution time plus the network delay. Before the start of the execution this delay has been estimated and can be used for an early warning algorithm (as described in section 6.4.2). When an object finishes after the estimated end time it is clear that a performance problem exist and a reconfiguration of each following object can be initiated.

Cost The cost of the execution depends on the way the performance will be monitored and if a failure occurs during execution. In case an N-version programming algorithm is being implemented, there exist a cost for every replication executed. Even though that only one solution will be used at the end, the other distributions will create network traffic and use system resources. Furthermore additional service providers might be involved and need to be paid.

In case a failure occurs one must additionally differentiate between a hardware failure (every object on the same computer will have to be migrated and restarted), and a performance failure (only the next objects to be executed have to be migrated). Before the execution such a cost can always only be estimated, however during runtime it can be added up, so that after the execution has terminated, the exact cost can be determined.

3.6 Summary

This chapter has introduced three performance monitoring architectures. Whereby in each of these architectures the monitor is divided into two parts. The Performance Monitoring Client (PMC) implements the monitoring activity. In contrast the Performance Monitoring Server (PMS) plays a more passive role and responses to requests and instructions issued by the PMC. The PMS is always a part of the object which has to be monitored, whereby the difference between the architectures is the location of the PMC. In the centralised model there exist only one PMC for each application, whereby the other two models have even a distributed architecture for the PMC. In one model there exist one PMC on every host used and in the last model there exist for every PMS exactly one PMC. The advantages and disadvantages for each of these architectures and a prediction of the behavioural difference between the models have been discussed. Furthermore several algorithms to monitor the availability, reliability, number of remote calls and the network delay created by these remote calls are presented.

Chapter 4

Estimating the Execution Time of a Globally Distributed Application

4.1 Introduction

This chapter describes an algorithm, that estimates start and end times of distributed application parts that run in parallel on a heterogeneous network of workstations (NOW).

Objects are distributed to workstations inside a network (LAN or WAN), where they represent data structures combined with methods operating on the data. The workstations' behaviour is simulated by machine and operating system dependent values (such as maximum speed and behaviour for pseudo parallel tasks) described in Section 4.3.

The communication between machines is only of interest when communication times between workstations are known values, and when estimating the execution time. Though the described algorithm gives an idea of how connection load can be simulated. Figure 4.1 shows a NOW, that is used in one of the later examples.

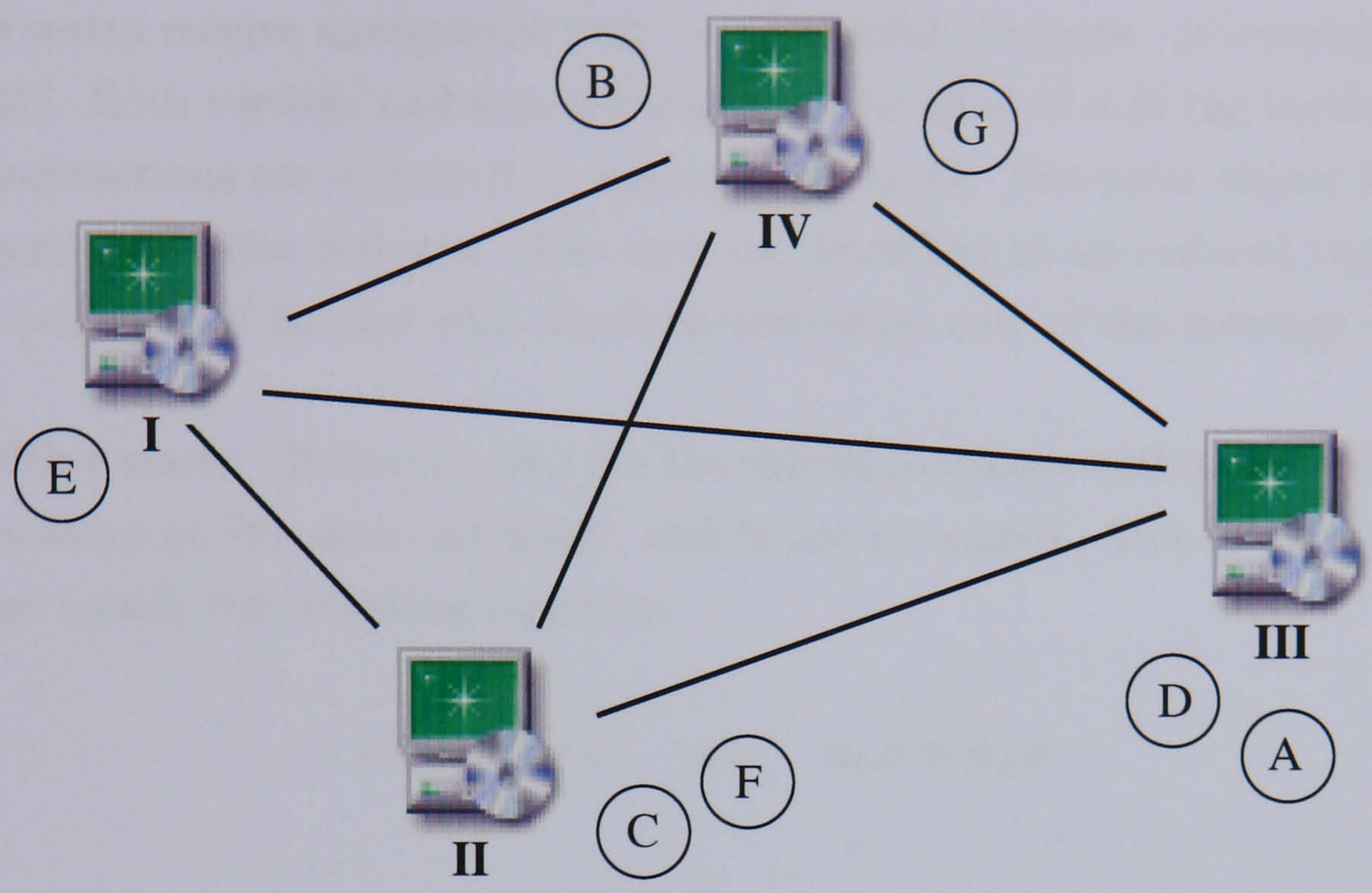


Figure 4.1: Sample Network of Workstations

The machines throughout this chapter are identified by Roman numbers, where uppercase letters denote objects. Figure 4.1 depicts the object distribution and connections between four machines.

The algorithm for estimating execution times is gradually evolved from a restricted application structure (tree structure, Section 4.4.3.1) to an application that is represented by a directed acyclic graph (DAG, Section 4.4.4), which allows asynchronous calls, parallel execution within objects and synchronisation. Section 4.2 describes the DAG and the application structure.

4.2 Representation of the globally distributed application

4.2.1 Directed acyclic graph

The application is represented as a directed acyclic graph (DAG) [9]. The DAG consists of vertices and arcs. A vertex represents an atomic part of the application, where the arc represent communication between these parts. Atomic means that during the execution of the vertex neither calls are made from the vertex itself nor

does the vertex receive messages or wait for other kind of events – it is computational closed [46]. Both vertices and arcs are labeled. The label $O\ n$ at the vertex denotes that n instructions are executed in service instance O. The term object is used as a synonym for service instance. The arcs are described as an ordered tuple of two vertices (n_i, n_j) and labeled with the communication cost of the message $c(n_i, n_j)$.

Arcs with ϵ costs If the $c(n_i, n_j)$ is ϵ the thread continues within the object itself, therefore stays on the same machine[1], and is not measured. Arcs labeled with an ϵ cost must satisfy the following equation:

$$c(n_i, n_j) = \epsilon \quad \Longrightarrow \quad n_i.o = n_j.o$$

Where $n_i.o$ denotes the service instance of vertex n_i.

ϵ arcs also indicate that the object stays alive and is used by the same or another thread later. A new object is created, if no ϵ arc enters the vertex. The object is destroyed after a vertex, that has no ϵ arc leading from it, has executed and sent messages.

Synchronisation in the DAG Each vertex n_i waits for all messages $\forall n_j(n_j, n_i)$to arrive, and then executes its instructions on the assigned machine. After all instructions have been executed all messages are sent asynchronously at the same time.

Figure 4.2 shows an example of a DAG. Note that, to keep the example simple, we have only one thread executing instructions on one object (service instance) at a time. We used the name of the object to identify the thread in Figure 4.3.

[1]persistent storage of objects and migration of objects during their lifetime is not discussed in this thesis.

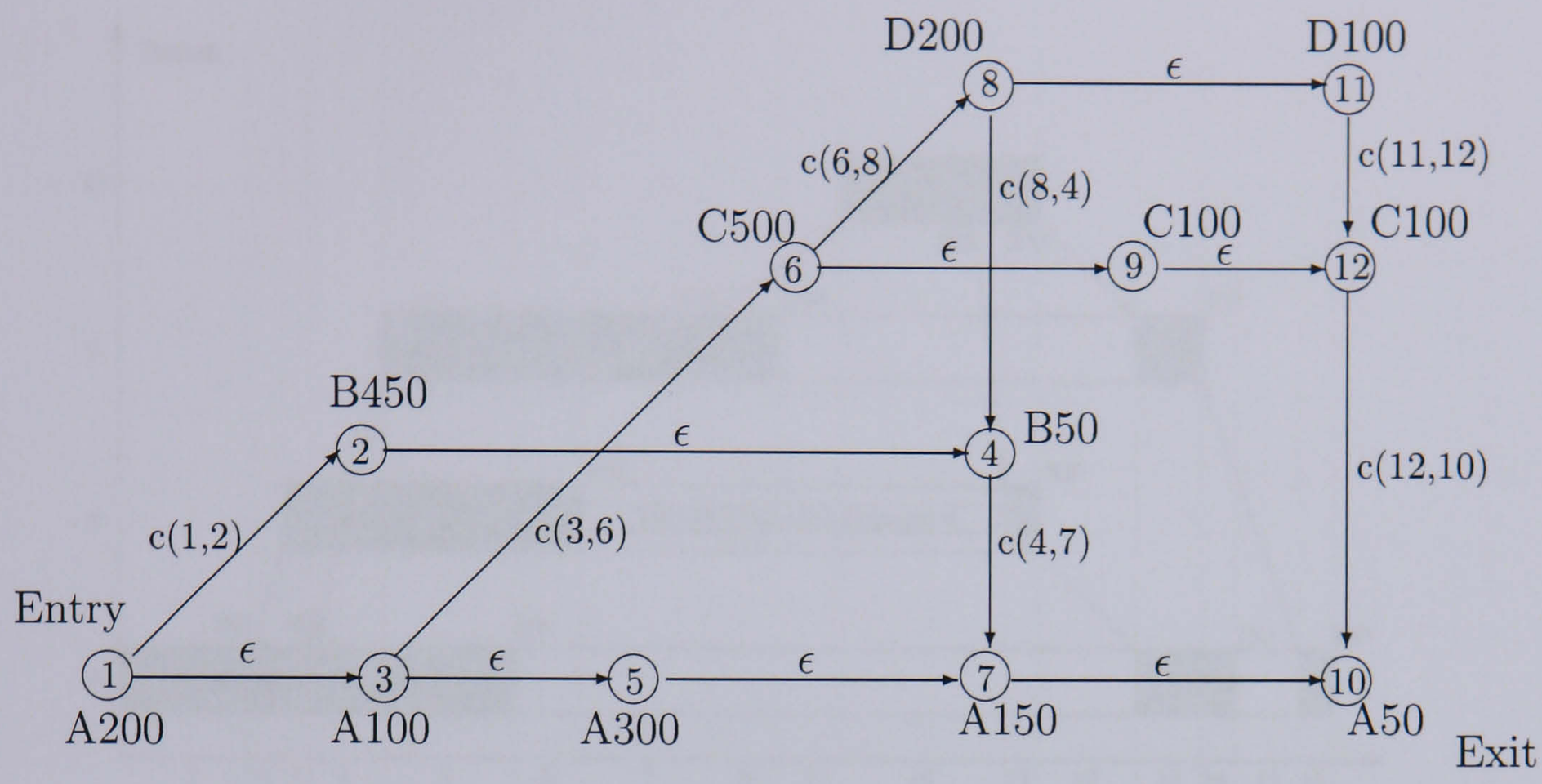


Figure 4.2: Application structure

In this example the entry vertex executes 200 instructions, then splits of a new thread, which calls n_2. Meanwhile it executes 100 more instructions (n_3) and then forks again to call n_6 and in parallel execute 300 instructions more (n_5). n_2 executes 450 instructions and then synchronises with vertex n_8 to execute n_4. n_6 executes 500, forks to call n_8, executes 100 instructions more in n_9 and waits (n_{12}) for the separated thread to join again after this has executed n_{11}. Meanwhile n_4 received the synchronisation message from n_8 and joins after 50 instructions in vertex n_7 its parent thread. So does the other remaining thread in n_{10}, which is the exit vertex of this graph.

Figure 4.3 shows for all the threads, their execution and waiting phases.

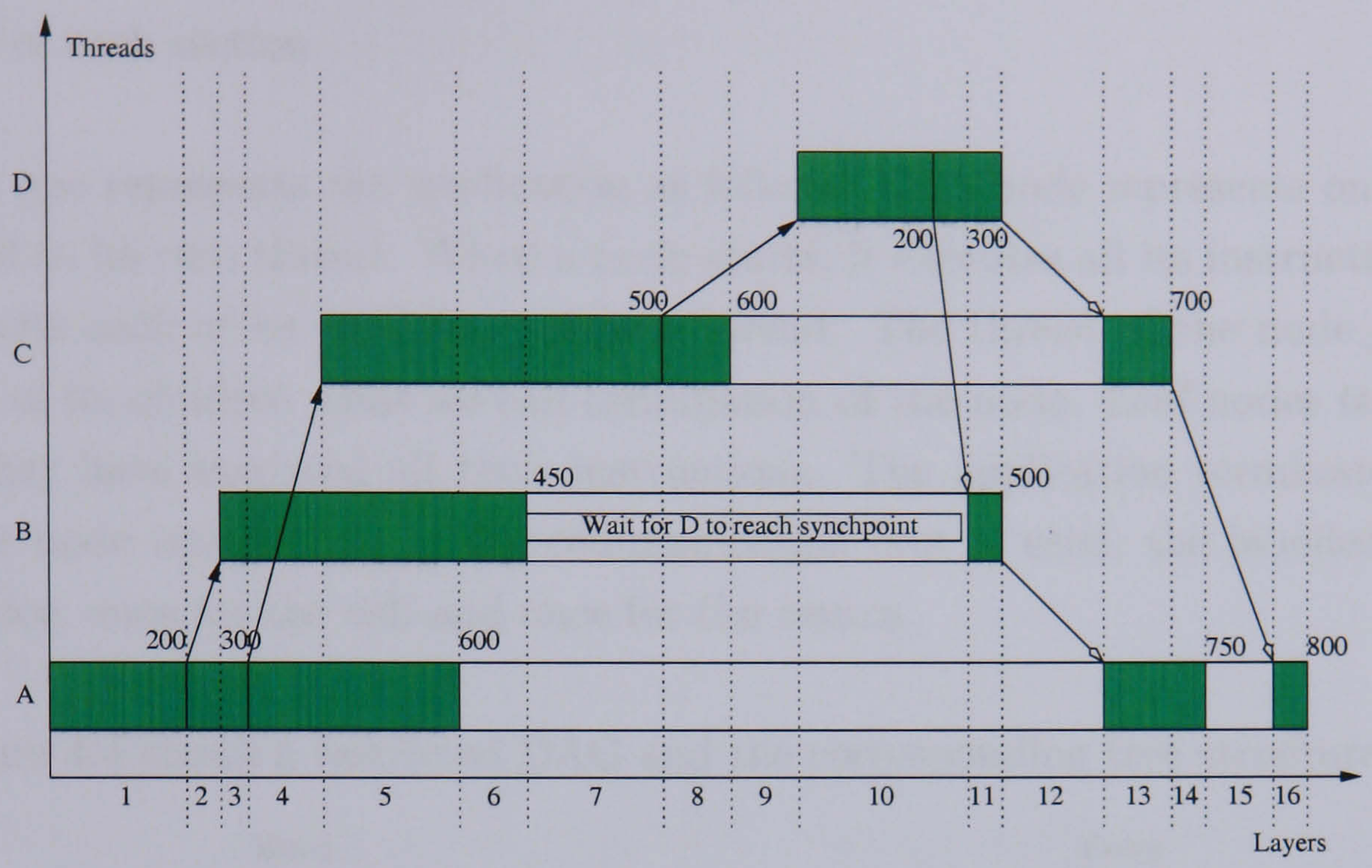


Figure 4.3: Layers for the execution

The filled parts of the bars means that the thread is executing instructions, while the outlined bar means that the thread is currently waiting. The dotted vertical lines define level borders. From one level to the next the conditions in the system are changing, because the number of parallel processing threads changes, however within one level no changes can occur. The algorithm to estimate the execution time of the distributed application is mainly finding these times and using them to compute the total time of the application – which is the end time of the last level.

The DAG consists of vertices and arcs. n_i identifies the vertex, where the ordered tuple (n_i, n_j) is used to address the arcs connecting the vertices n_i and n_j.

Costs are the communication time of a call which is $c(n_i, n_j)$ and the execution time of a node n_i which is denoted by $w(n_i)$, where the number of instructions executed by the node is $n_i.i$.

4.2.2 Tree-structure as a specialised DAG

To introduce the algorithm on simple examples, a more restricted data structure is chosen to represent the application. The algorithm will be evolved using the tree structure, which is described below and is generalised to take a DAG as input at

the end of each section.

The tree represents the application as follows: Each node represents one object executed in its own thread. When a node starts, it executes all its instructions and then starts each of its children in a new thread. The thread of the node joins all threads of its children what we call termination of the node. Leaf nodes terminate when they have executed all their instructions. The application terminates when the root node terminates. If the communication cost is used, the labelled cost is used twice, once for the call and once for the return.

Figure 4.4 shows a restricted DAG and the corresponding tree structure.

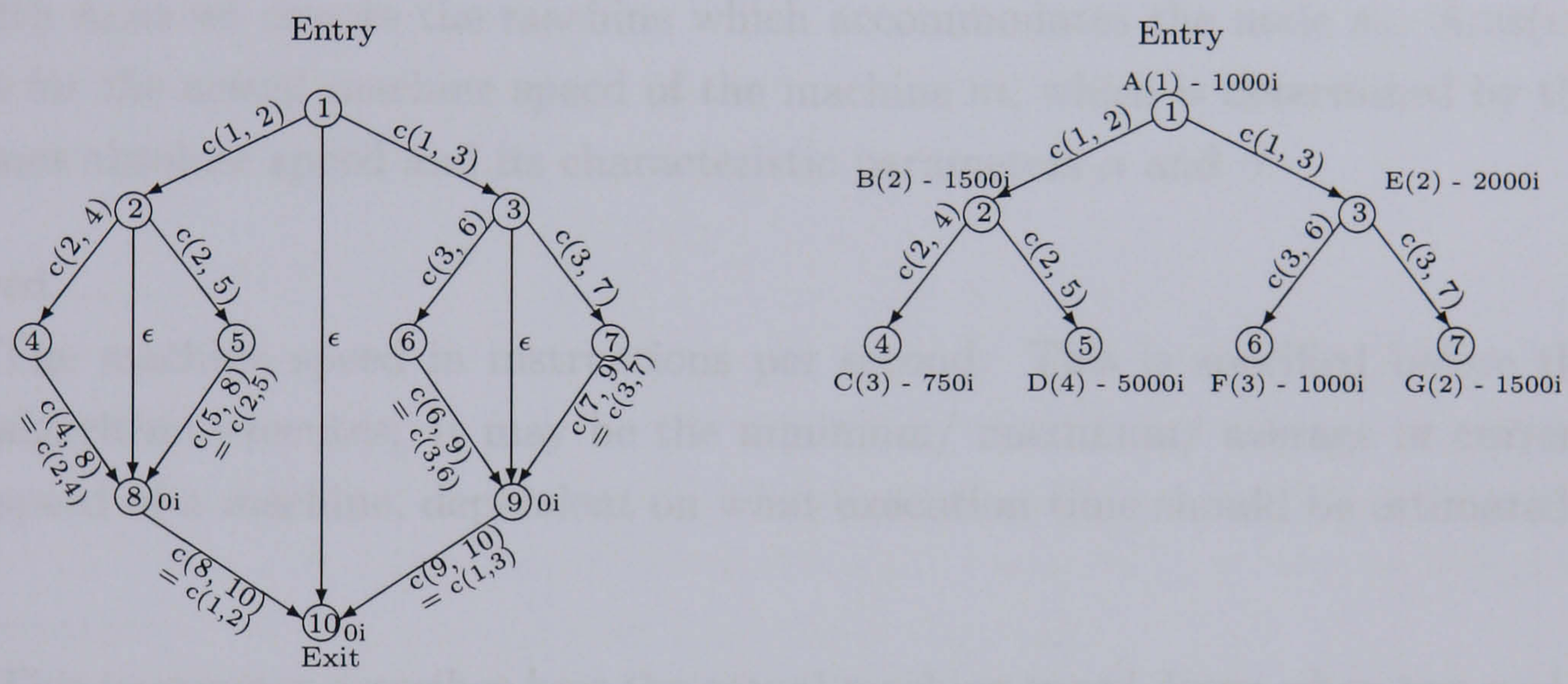


Figure 4.4: simplified DAG (left), tree structure (right)

The DAG left and the tree on the right hand side of Figure 4.4 describe the same application behaviour. Obviously the restriction that a node cannot execute instructions after (one or more) of its children have terminated (n_8, n_9, n_{10} are labelled with 0 instructions) is strong. But this restriction will be eliminated when the algorithm is generalised later on. Note the other important restriction which is necessary to represent a DAG as a tree structure is that the communication cost for the return call is the same as for the invocation (e.g. $c(4,8) = c(2,4)$).

The term node is used when the tree structure is described, where we use vertex for the description of the DAG. Further there is a distinction between the two terms finish execution and terminating. A node finishes execution if it has executed all its instructions and calls its children. Terminate is used to indicate that the focus is

returned to the parent node, as described above.

4.3 Representation of the resources

Resources The NOW consists of several host machines, which are heterogeneous. This means that they can run different operating systems, have different amount of processors and provide different speeds and memory resources. Machines are identified by their name, (which is throughout this chapter a Roman number). The behaviour of a particular machine is simulated using a speed value and two parameters (α and β) which describe the behaviour of the machine when it executes threads in (pseudo) parallel.

With $n_i.m$ we denote the machine which accommodates the node n_i. Ams(m) stands for the actual machine speed of the machine m, which is determined by the machines absolute speed and its characteristic parameters α and β.

m.speed ....

The machine speed in instructions per second. This is specified before the algorithm executes. It may be the minimum/ maximum/ average or current speed of a machine, dependent on what execution time should be estimated.

$\boldsymbol{\alpha}$ .....

This parameter describes how the actual machine speed drops when two nodes collide. A setting of 0 means no speed drop, where a setting of 1 means the $execCount$th ($|m.E|$) part of the speed.
Setting $\alpha \in (0, 1)$ can simulate a multiprocessor machine. A setting of $\alpha = 0.5$ for example means that if there are two nodes executing the machine is running at full speed (simulates a machine with 2 equal processors). Setting $\alpha > 1$ means that for more than one executing service instance the speed drops more rapidly. Obviously the described values are given under the assumption, that the scheduling operating system on the machine does not use processor time itself. Values that describe the real behaviour of a machine are in most cases empirical.

$\boldsymbol{\beta}$ .....

Similar to α it describes the speed drop, but for waiting nodes. A setting of

0 means no speed drop, where a setting of 1 means the *waitCount*th ($|m.W|$) part of the speed.

The actual machine speed of a machine (ams(m)) can be computed by the following formula:

$$p(m) = \begin{cases} 1 & \text{if } |m.E| = 1 \wedge \alpha \geq 1 \\ \alpha * |m.E| & \text{else} \end{cases} \tag{4.1}$$

$$q(m) = \begin{cases} 1 & \text{if } |m.W| = 1 \wedge \beta \geq 1 \\ \beta * |m.W| & \text{else} \end{cases} \tag{4.2}$$

$$d(m) = \begin{cases} p(m) + q(m) & \text{if } p(m) + q(m) > 1 \\ 1 & \text{else} \end{cases} \tag{4.3}$$

$$\text{ams}(m) = \frac{\text{m.speed}}{d(m)} \tag{4.4}$$

$p(m)/q(m)$ are the summands of the denominator which represent the speed drop due to executing/waiting service instances. $d(m)$ is the denominator representing the total speed drop. Note that if the denominator $p(m)+q(m) \leq 1 \Rightarrow$ ams(m) $\equiv$ speed. This means that the *ams(m)* is never faster than m.speed.

The different memory usage of objects (code size and memory allocation etc.) and the time that is used for swapping is not taken into account. To simulate memory effects the machines have to store additionally to the thread load a memory load, which affects the actual machine speed. Although the estimation would benefit from the simulation of a memory load it is not the goal of this thesis to simulate different scheduling algorithms and the memory management of operating systems. The more accurate simulation of heterogeneous machines does improve the accuracy of the time estimation, and is part of the future work.

4.4 Estimating times

This section stepwise evolves the idea of execution time estimation from a sequential to a parallel application. The estimation is necessary to judge whether the application can terminate within a certain time. Performance monitors can use these

times to decide during the run if the remaining parts of an application run short in time and try to reconfigure the distribution or priority of parts during the execution. Furthermore it is possible to store the information of what is causing the speed drop of machines (eg. several threads sharing resources) and optimise the distribution to avoid these collisions. The optimisation algorithm described in Chapter 5 is based on this information.

Some subsection refer to a setting[2] of α and β values. These values are used to specify the effect the execution of parallel parts has on the executing machine when using the algorithm that is introduced in Section 4.4.3.1.

Algorithm dependent data The algorithm keeps track of all executing and waiting nodes. These nodes are stored in the sets E and W. Each node has a start time (n_i.startTime) and an end time (n_i.endTime).

Additionally to detect collisions, each machine m stores its own set of executing and waiting nodes, which we denote by $m.E$ and $m.W$. The actual load of a machine is therefore $|m.E|$ executing and $|m.W|$ waiting threads. Of course this is redundant information and could be retrieved from the sets E and W, but for clarity and efficiency we avoid the search and store them separately.

During the evolution of the algorithm from a simple example we need more data structures. We will describe them when they are used the first time. Though Appendix A shows an overview over all used data structures.

4.4.1 Sequential execution

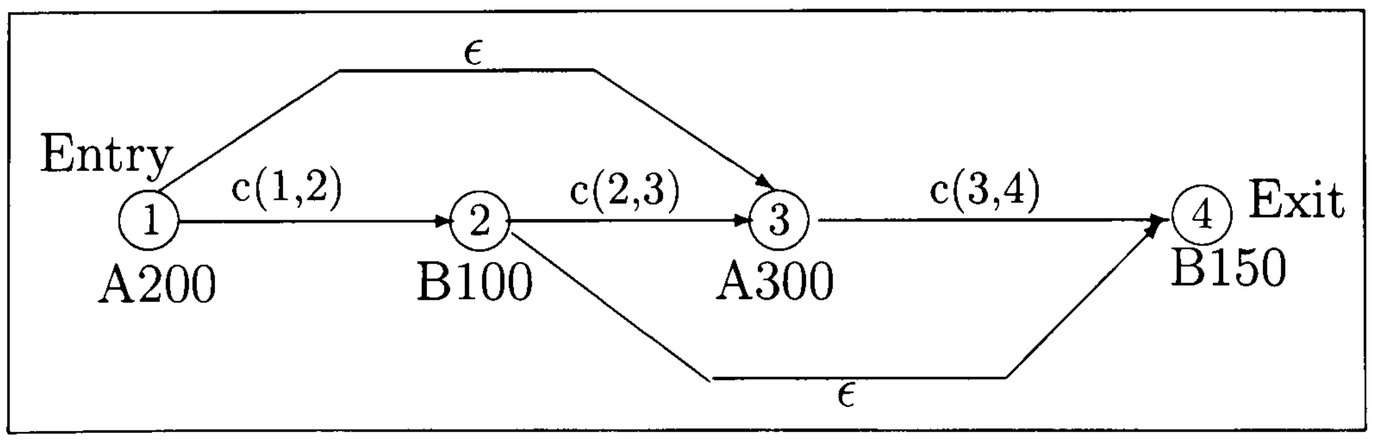


Figure 4.5: DAG for a sequential application

[2] α and β are used to compute the *actual machine speed*. See Section 4.3

Figure 4.5 shows the DAG of a sequential application. Here we have two different objects A and B which the single executing thread uses alternatingly. The ϵ arcs in this DAG are necessary to indicate that the object A in n_1 is the same as in n_3. For the other two vertices it is the same with object B. The ϵ arc will cause n_3, and therefore $n_3.o$ (which is A) to wait for the incoming message (n_2, n_3).

If all nodes are executed serially, the estimation of the time used for execution is fairly simple: the times each node takes to execute is simply added. If N is defined to be the sequence of all vertices and T the set of arcs between these vertices, the total execution time of the application is:

$$SL = \sum_{\forall n \in N} w(n) + \sum_{\forall t \in T} c(t) \tag{4.5}$$

SL ; the total execution time of the application
N ; ..... the sequence of all vertices
w(n) ; ..... the execution time of node n
T ; ..... the arcs between vertices and
c(t) ; ..... is the communication time

$$w(n) = \frac{n.i}{n.m.\text{speed}} \text{ assuming } m.\beta \text{ is } 0 \tag{4.6}$$

4.4.2 Parallel execution without slow down

If nodes can be executed in parallel the computation is more sophisticated. To simplify the algorithm we use the tree structure of the application, which of course is only a very small subset of the possible DAGs. We also specify that the execution of a service instance does not affect the machines speed and memory load where it is executed. This is equivalent to $\alpha = \beta = 0$, which means the machine provides

constant speed. In this very artificial case the time can be computed as follows:

$$\begin{aligned}\text{time}(n_i) = \frac{n_i.i}{n.\text{ams}(n_i.m)} &+ 2 * c(parent(n_i), n_i) \\ &+ \text{MAX}(\{0\} \cup \{\text{time}(c) \mid c \in n_i.\text{children}\})\end{aligned} \tag{4.7}$$

Note: $ams \equiv speed$ when $\alpha = \beta = 0$ as assumed. $n_i.children$ is the set of all children of node n_i. The communication $c(parent(n_i), n_i)$ is counted twice, once for the call and once for the return. For the root node ($parent(root)$ does not exist) we define $c(_, root) = 0$. We define the maximum/ minimum of a set as:

$$MAX(S) \equiv s : s \in S \wedge \forall x, x \in S, \ \wedge s \geq x \tag{4.8}$$

$$MIN(S) \equiv s : s \in S \wedge \forall x, x \in S, \ \wedge s \leq x \tag{4.9}$$

This assumption leads to the question, what happens when α and β are values in $\mathbb{R}^+$. This means both executing and waiting service instances are slowing down the machine while they are running. In this case the equation 4.7 is insufficient, because the actual machine speed ($ams(m)$) is changing during the algorithm according to the load on the particular machine. This side effect demands to adjust the speed at well defined points during the execution of the application. These points are as mentioned before the layers shown in Figure 4.3.

4.4.3 Parallel execution affecting machines

To simulate a speed drop of a machine depending on its load, we set the values $\alpha = 1$ and $\beta = 0$. All executing service instances share the speed in equal parts, and waiting instances do not slow down the machine. This attempt comes closer to the behaviour of a real application than the equations used in preceding subsections. This section will introduce an algorithm to find the estimated start and end times for the tree structure. In Section 4.4.3.1 we describe the estimation for the tree structure, whereby we omit the times that are needed for the calls from one node to another one. The next Section shows the estimation for the example tree shown in Figure 4.4. Before introducing the communication for the algorithm in Section 4.4.3.5 we describe how collisions can be detected and stored in Section 4.4.3.3

and 4.4.3.4.

4.4.3.1 Estimating times for a tree without communication

This section describes the most restricted (and therefore simplest) version of the algorithm. It is assumed, that the application is represented by a tree structure as described in Section 4.2.2. Furthermore no communication cost/time ($c(n_i, n_j) = 0ms$) is taken into account.

For the algorithm it is assumed that each machine is providing a constant speed (m.speed), which is only affected by the number of its waiting or executing nodes. For this number is only changed when a new node is started, or one has terminated, the simulated actual machine speed needs only adjusting at these well defined points. Between these machine condition changes all nodes can execute instructions at the constant actual machine speed. We will describe the algorithm using a combination of structure diagrams and sets.

Additional attributes For this algorithm $n.ei$ stores the currently executed instructions of node n. Furthermore each node stores its `status` which can be *INITIALISED*, *EXECUTING*, *WAITING* or *TERMINATED*. The start and end time of each node is stored in n_i.startTime and n_i.endTime

For each Nassi-Schneiderman diagram [35] a short informal description is given below.

Initialise E with the *root* node, and then start the *root* node. For the estimation of more than one tree all root nodes have to be started before the loop is entered. This allows to simulate the effects of other applications which run at the same time. Then the main algorithm is:

— main

`start(root)`
$E \cup W \neq \emptyset$
`first` $\in E \mid$ `MIN(execTime(`$o \in E$`))` $=$ `execTime(`$first$`)`
`execute(first)`

After initialisation the loop continues until all service instances are executed. The first loop statement assigns to *first* the node, which under current conditions finishes its execution first. This one is executed. If there is more than one node fulfilling the condition, one is taken randomly. The others remain for the next loop iteration.

start(n: **node**) : **void** —

$n.m.W \leftarrow n.m.W \setminus \{n\}$
$n.m.E \leftarrow n.m.E \cup \{n\}$
$W \leftarrow W \setminus \{n\}$
$E \leftarrow E \cup \{n\}$
n`.state` $\leftarrow$ `EXECUTING`

If the node is waiting it is removed from the waiting sets and added to the executing sets. When `start` has been called, the node is set to be executing.

wait(n: **node**) : **void** —

$n.m.E \leftarrow n.m.E \setminus \{n\}$
$n.m.W \leftarrow n.m.W \cup \{n\}$
$E \leftarrow E \setminus \{n\}$
$W \leftarrow W \cup \{n\}$
n.`state` $\leftarrow$ `WAITING`

If the node is executing it is removed from the executing sets and added to the waiting sets.

terminate(n: **node**) : **void** —

$n.m.E \leftarrow n.m.E \setminus \{n\}$
$n.m.W \leftarrow n.m.W \setminus \{n\}$
$E \leftarrow E \setminus \{n\}$
$W \leftarrow W \setminus \{n\}$
n.`state` $\leftarrow$ `TERMINATED`

The node is removed from the executing and waiting sets.

---

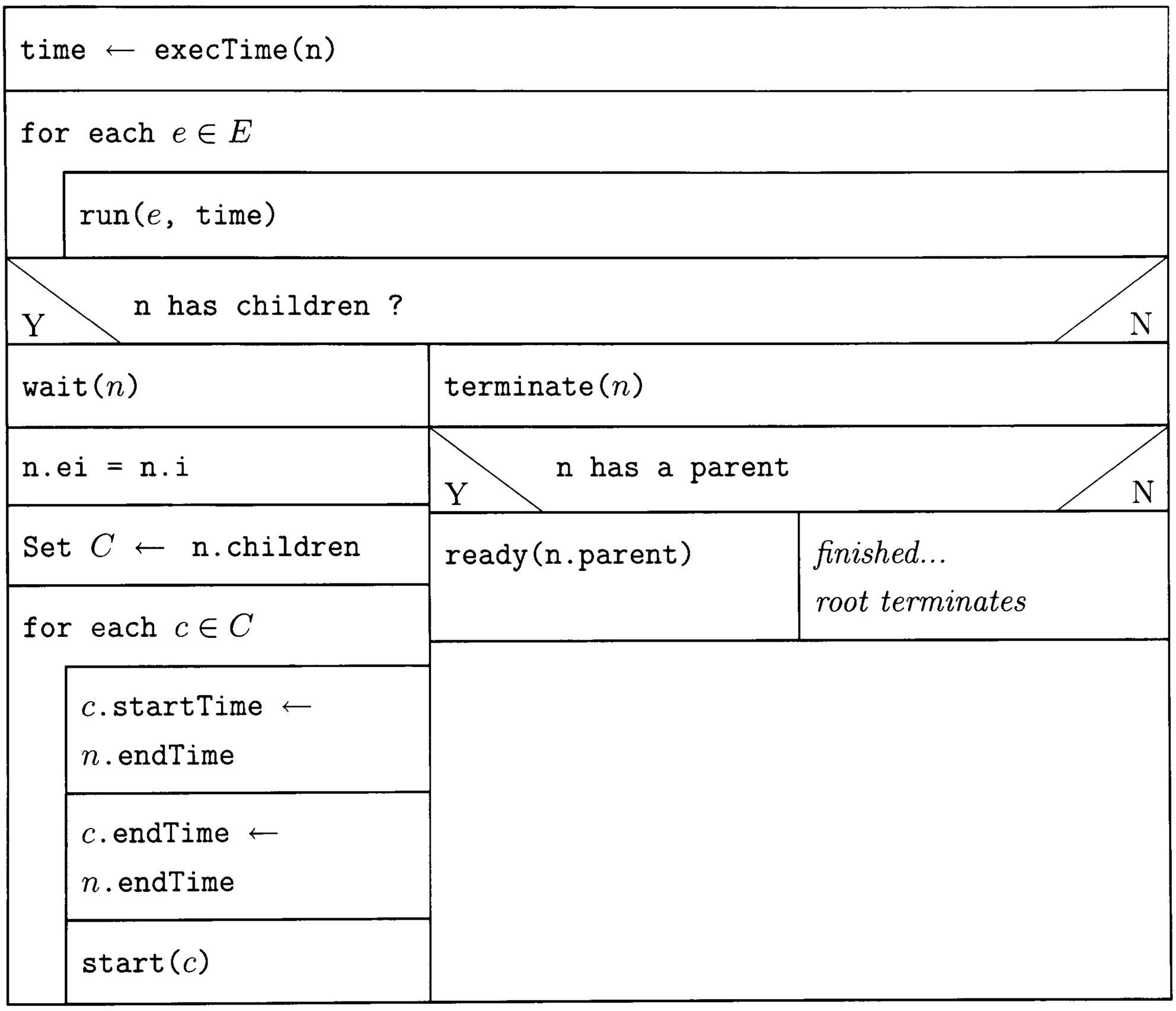
execTime(n : node) : long —

```
return (long) ((n.i - n.ei) / ams(n.m) + 0.5)
```

This method computes the time the passed service instance will take, under current conditions, to finish its execution. The returned time is rounded.

---

execute(n : node) —



All other currently executing objects run for the time the passed object takes for its execution. They run under the current conditions. If the node has children it updates its machine properties, which means changing the runtime conditions. Then the children are started, changing their machine properties and updating the set of executing service instances. If the node is a leaf it terminates; updating the set of executing service instances and informing its parent.

---

run(n: node, time: long) —

`n.endTime += time`
`n.ei += (long) (ams(n.m) * time + 0.5)`

This method updates the end time and the current instruction count of the object. Rounding errors do not affect the algorithm. But the more often conditions change during the execution of a service instance the more inaccurate the result of the end time becomes. If a guarantee is needed that the end time is never estimated to low you should modify *execTime/1* method so it always rounds up (other solutions: measure times in double values and/or use interval arithmetic).

---

ready(n: node) —

Y: $\forall c \in n$.children, c.state = TERMINATED	N
n.endtime $\leftarrow$ MAX($\{c$.endtime $\mid c \in n$.children$\}$)	
terminate(n)	
Y: n has a parent	N
ready(n.parent)	*finished... root terminates*

If all children are ready the machine properties are adjusted, and the node terminates - informing its parent. There is no check necessary if the node has children, because this method is only called from a child.

4.4.3.2 Example for the tree structure

The application is defined by the tree shown in Figure 4.4. Each node represents one object, and therefore the execution time of a node is equivalent to the lifetime

of the object.

For the example we have 4 machines that are identified by Roman numbers. Table 4.1 shows the attributes for the machines. The services list can be ignored for this example. It is used to indicate that not every node can be scheduled for every machine - which is important for the optimisation algorithm discussed later.

Machine	Speed ($\frac{\text{instr}}{\text{s}}$)	Services
I	100	1, 2, 3
II	200	3, 4
III	300	1, 4
IV	200	2, 3

Table 4.1: Machine speeds/ services

Instance	Service	Machine
A	1	III
B	2	IV
C	3	IV
D	4	III
E	2	IV
F	3	IV
G	2	IV

Table 4.2: greedy schedule

Table 4.2 shows a greedy schedule for all nodes. All nodes take the fastest machine, where their corresponding service is provided.

The algorithm results are shown in Figure 4.6. The y axis shows different machines and their scheduled nodes. The x axis shows the time in milliseconds. A filled bar means again executing, whilst a outlined bar means waiting. Note that the calls are omitted in this diagram.

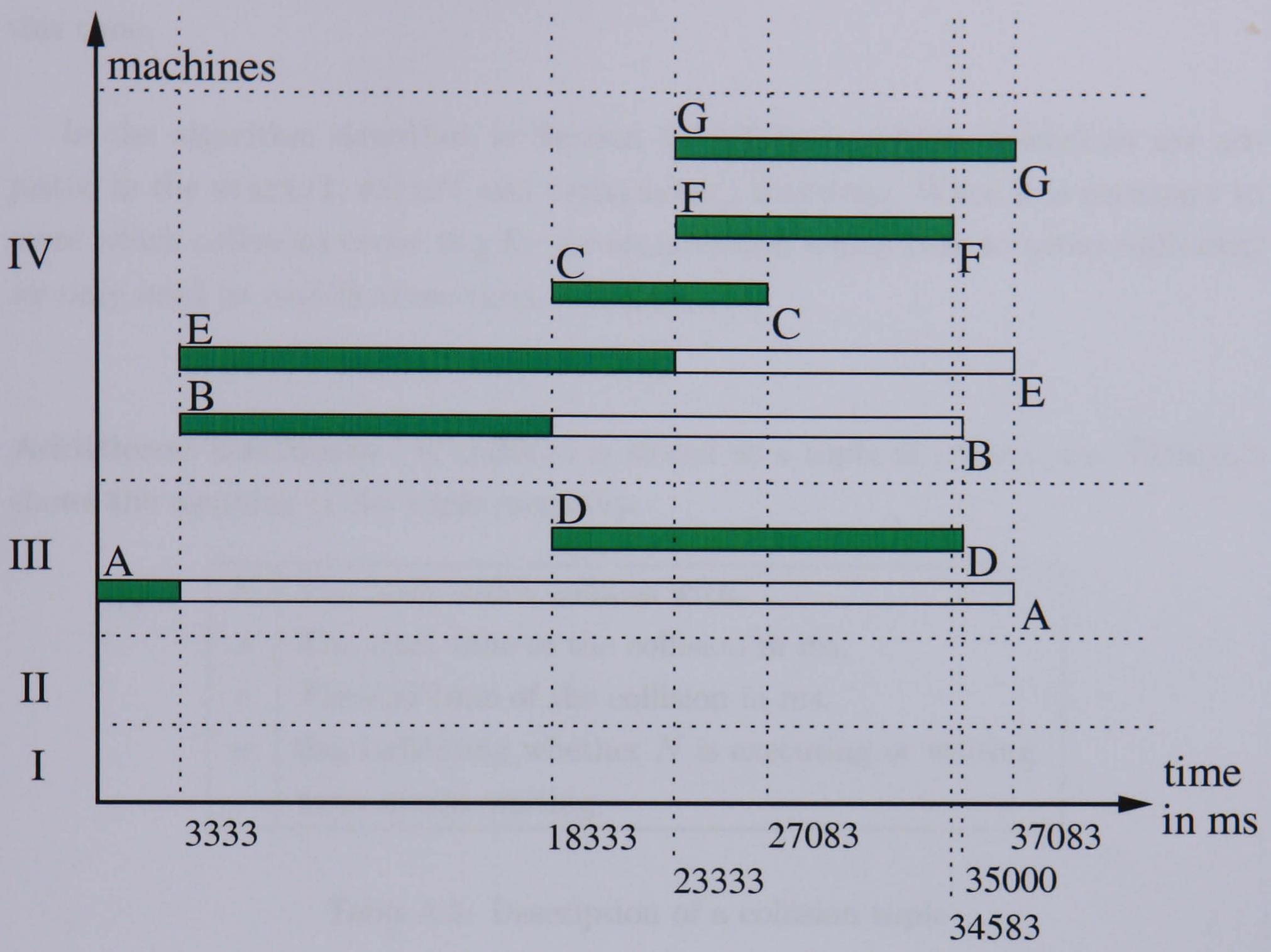


Figure 4.6: Time flow greedy

After A finishes execution B and E are started on machine IV. They both share the machine speed in equal parts. After B, the nodes C and D are started. While D is running at full machine speed, node C has to share the resources with E. At time $= 23333$ E calls its children F and G on machine IV. Now all nodes on machine IV are running at a third of the machine speed. After C terminates, only nodes F and G are sharing their resources. Node G is the last to finish, about 37 seconds after the applications start.

4.4.3.3 Detection of collisions in the tree

Definition of a collision: Two nodes (n,N) collide, when they are executing and/ or waiting on the same machine at the same time. We refer to *wait collision* of n if N was waiting when the collision occurred and *execute collision*, if N was executing. Therefore n has a wait collision with N if and only if N is waiting at

this time.

In the algorithm described in Section 4.4.3.1 the machines conditions are adjusted in the `start/1`, `wait/1` and `terminate/1` functions. When it is necessary to store which collisions occur (e.g for an optimisation which tries to avoid collisions) we only need to modify these three methods.

Additional attributes A *collision* is stored as a tuple of (N, s, e, w). Table 4.3 shows the meaning of the tuple members.

N	The node which collides with.
s	The start time of the collision in ms.
e	The end time of the collision in ms.
w	flag indicating whether N is executing or waiting. `true` means waiting.

Table 4.3: Description of a collision tuple

Collisions are stored in two separate sets within the node structure. Set $n.EC$ stores the collisions that occur during ns execution and $n.WC$ stores those while n was waiting.

Below are the modified structure diagrams of the `start/1`, `wait/1` and `terminate/1` functions.

start(n: **node**) : **void** —

```
for all e ∈ n.m.E ∪ n.m.W
    endCollision(e, n, n.endTime)
n.state ← EXECUTING
n.m.W ← n.m.W \ {n}
W ← W \ {n}
for all e ∈ n.m.E ∪ n.m.W
    startCollision(e, n, n.endTime)
    startCollision(n, e, n.endTime)
n.m.E ← n.m.E ∪ {n}
E ← E ∪ {n}
```

The first loop ends all wait collisions (and all collisions of n) that have started, but not yet ended. The second loop starts execute collisions for all other nodes, and also the appropriate collisions for the node itself.

wait(n: **node**) : **void** —

```
for all e ∈ n.m.E ∪ n.m.W
    endCollision(e, n, n.endTime)
n.state ← WAITING
n.m.E ← n.m.E \ {n}
E ← E \ {n}
for all e ∈ n.m.E ∪ n.m.W
    startCollision(e, n, n.endTime)
    startCollision(n, e, n.endTime)
n.m.W ← n.m.W ∪ {n}
W ← W ∪ {n}
```

As described in `start/1` the first loop ends all execute collisions (and all collisions of n). The second loop starts all wait collisions.

terminate(n: node) : void —

`for all` $e \in n.m.E \cup n.m.W$
`endCollision(`e`,` n`,` n`.endTime)`
$n.m.E \leftarrow n.m.E \setminus \{n\}$
$n.m.W \leftarrow n.m.W \setminus \{n\}$
$E \leftarrow E \setminus \{n\}$
$W \leftarrow W \setminus \{n\}$
n`.state` $\leftarrow$ `TERMINATED`

Ends all collisions.

Additionally we need two more functions, that handle the collision start and end. Because these functions deciding on the nodes state (n.state) and are used often within the `start/1`, `wait/1` and `terminate/1` functions we separated them.

startCollision(n : node, c : node, time : long) —

n.state		
WAITING	**EXECUTING**	**else**
`entrySet` $\leftarrow n.WC$	`entrySet` $\leftarrow n.EC$	`error`
`entrySet` $\leftarrow$ `entrySet` $\cup \{(c, \texttt{time}, _, c.\texttt{state} = \texttt{WAITING})\}$		

When called this method decides on which set to operate depended on the nodes own state. To the chosen set a new collision tuple is added.

endCollision(n : **node**, c : **node**, **time** : **long**) —

n.state		
WAITING	**EXECUTING**	**else**
`entrySet` $\leftarrow n.WC$	`entrySet` $\leftarrow n.EC$	`error`

Y: `n = c`	N
$\forall i \in \{x \in$ `entrySet` $\mid x.e$ `is not set`$\}$	$i \in \{x \in entrySet \mid x.N = c\}$
$i.e \leftarrow$ `time`	i.`e` $\leftarrow$ `time`

Decides on which set to operate depending on the nodes own state. If n equals c it means that the node n itself is causing a configuration change. In this case all pending collisions (those which have not ended yet) must be terminated. If the passed node is different, only the collision which contains this node has to terminate. The **end** attribute of a collision is only set once.

4.4.3.4 Example for collisions

When the modified algorithm is run on the previous greedy example we encounter several collisions. Table 4.4 shows the collisions during the nodes execution and waiting time. The contents of the table is divided into blocks for each node **A** to **G**. These blocks are grouped into execute collisions ($w =$ `false`) and wait collisions ($w =$ `true`). Inside these groups the collided nodes are sorted alphabetically.

during execution				**while waiting**			
Node	start	end	wait	Node	start	end	wait
Collisions for Service instance **A**							
no collisions				**D**	18333	35000	`false`
Collisions for Service instance **B**							
E	3333	18333	`false`	**C**	18333	27083	`false`
				E	18333	23333	`false`
				F	23333	34583	`false`
				G	23333	35000	`false`
				E	23333	35000	`true`
Collisions for Service instance **C**							
E	18333	23333	`false`				
F	23333	27083	`false`				
G	23333	27083	`false`	no collision			
B	18333	27083	`true`				
E	23333	27083	`true`				
Collisions for Service instance **D**							
A	18333	35000	`true`	no collision			
Collisions for Service instance **E**							
B	3333	18333	`false`	**C**	23333	27083	`false`
C	18333	23333	`false`	**F**	23333	34583	`false`
B	18333	23333	`true`	**G**	23333	37083	`false`
				B	23333	35000	`true`
Collisions for Service instance **F**							
C	23333	27083	`false`				
G	23333	34583	`false`	no collision			
B	23333	34583	`true`				
E	23333	34583	`true`				
Collisions for Service instance **G**							
C	23333	27083	`false`				
F	23333	34583	`false`	no collision			
B	23333	35000	`true`				
E	23333	37083	`true`				

Table 4.4: Collisions for the *greedy* example

Obviously there are no collisions while the root node is executing and no collisions for waiting leaf nodes (they are never waiting). Of more interest is the occurance of two collisions with node **E** while **B** is waiting (see *Collisions for Service instance* **B**). The first one is an execute collision (**E** is executing from 18333 until 23333) and the second one a wait collision (**E** waits for **F** and **G** from 23333 until 37083). Note that for this wait collision the end time is only 35000 which is the time when **B** stops waiting for **D**. When comparing Table 4.4 with the Figures 4.4 and 4.6 this case occurs several times.

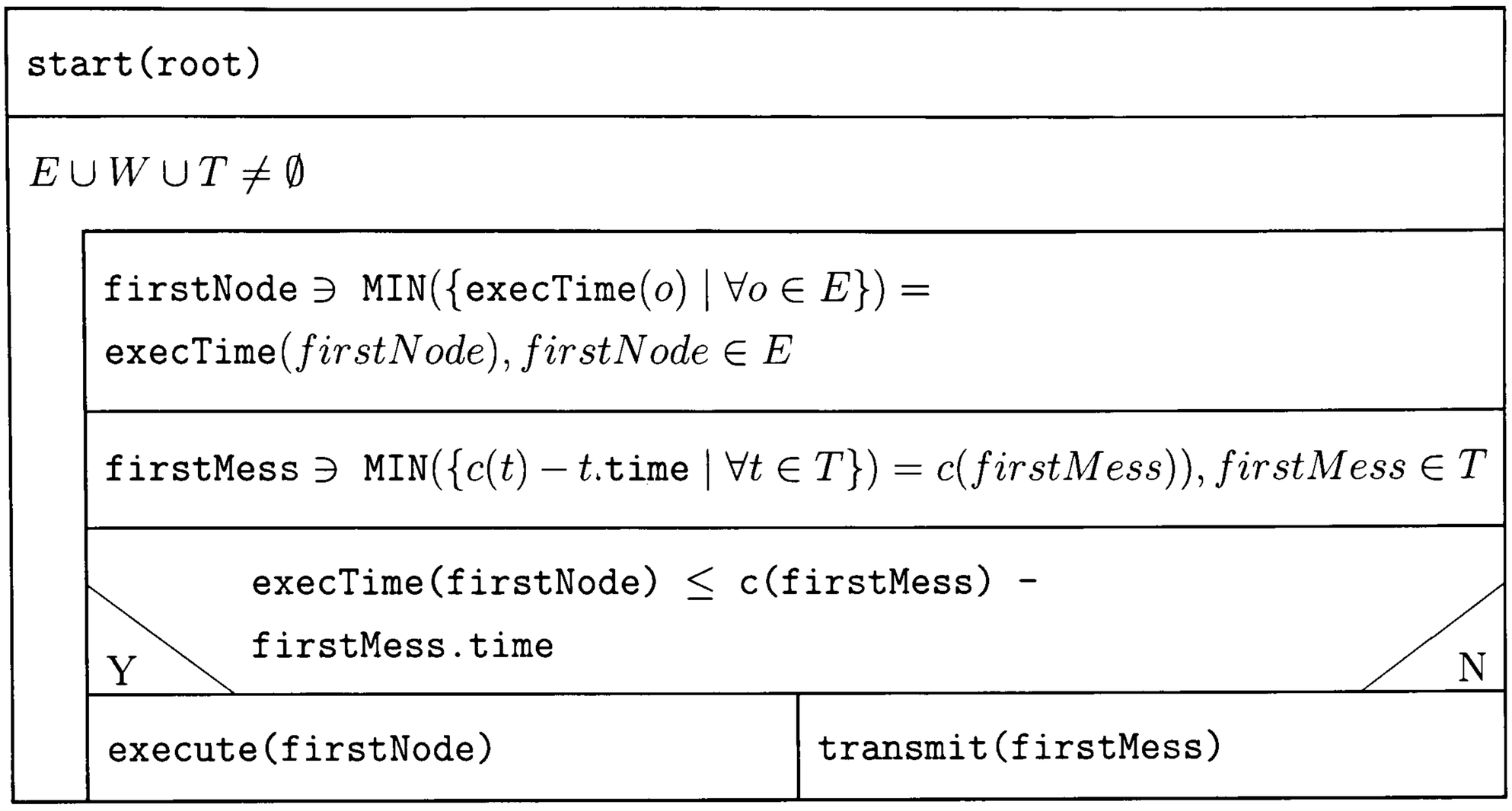
4.4.3.5 Communication in the tree

The only points where communication occurs is when starting or terminating nodes. The point to modify the algorithm described in Section 4.4.3.1 is the `execute/1` and `ready/1` function, this is where nodes are started and terminated.

Additionally the `main` method needs a modification. Because we have to delay the calls, we need a new set T which stores all messages, that are currently processed. The minimum search in `main` needs to be adjusted to find the minimum time, when the first node in E will finish execution and the first message which will reach its addressee. Depending on whether the minimum is found for a node or a message `main` has either to `execute/1` the node or invoke the message.

Additional attributes As mentioned above the set T to store all currently processed messages. To store the progress of a message transmission we store the time that has already been used in the attribute (n_i, n_j).time. The remaining transmission time therefore is $c(n_i, n_j) - (n_i, n_j)$.time.

— main



Find the first node that finish execution under current conditions. Find first message that arrives. If the node finishes before the next message arrives, execute it otherwise transmit the message.

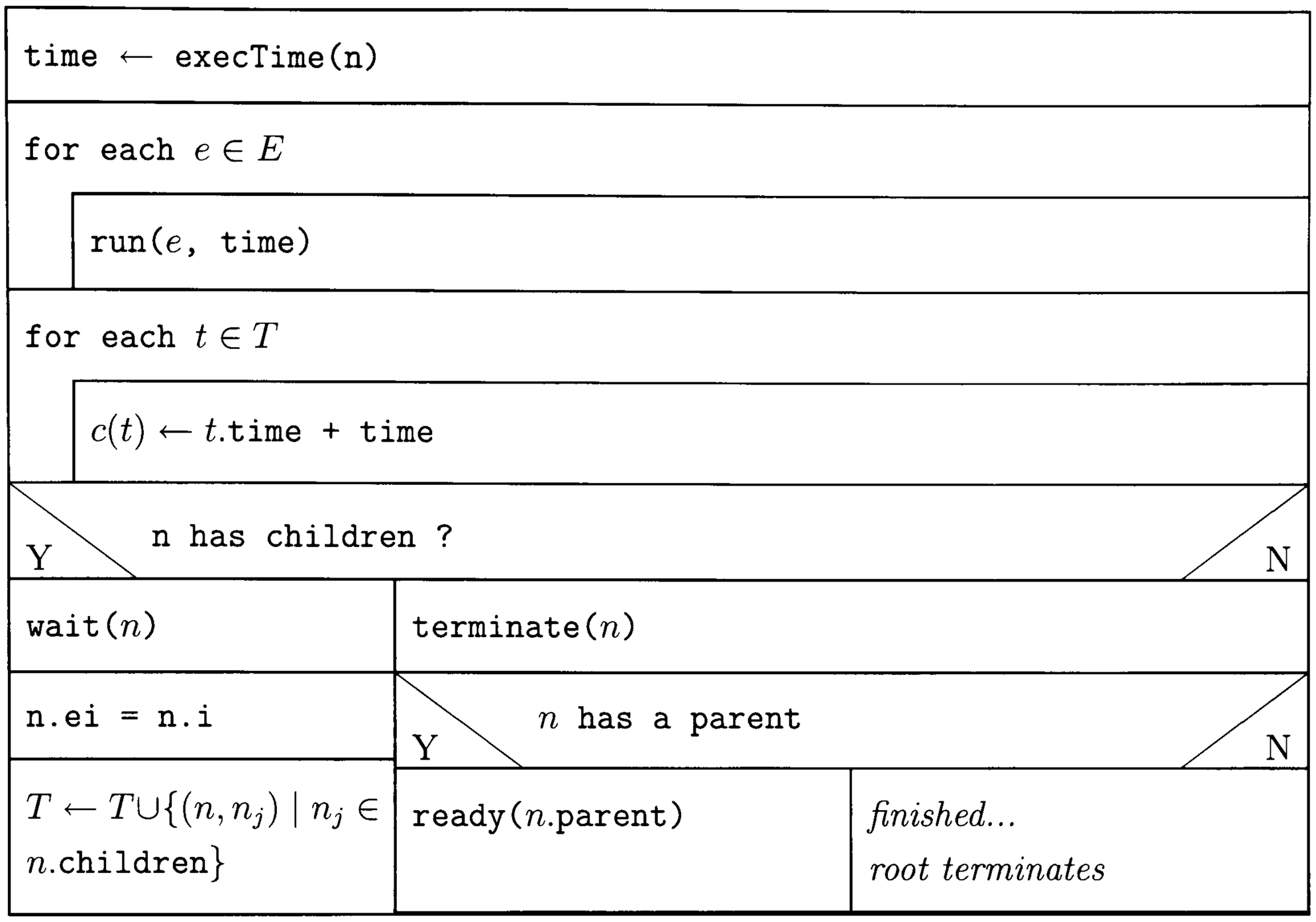
transmit((n_i, n_j) **)** —

```
time ← c(n_i, n_j) − (n_i, n_j).time
for each e ∈ E
    run(e, time)
for each t ∈ T
    c(t) ← t.time + time
T ← T \ {(n_i, n_j)}
n_j.startTime ← n_i.endTime + c(n_i, n_j)
n_j.endTime ← n_j.startTime
start(n_j)
```

All executing nodes can safely run for the time the message transmission took. After that time the conditions are changing, because a new node will be started. The start and end times of the called node are set and the node is started. This is correct for a tree, where each node has only one message calling it.

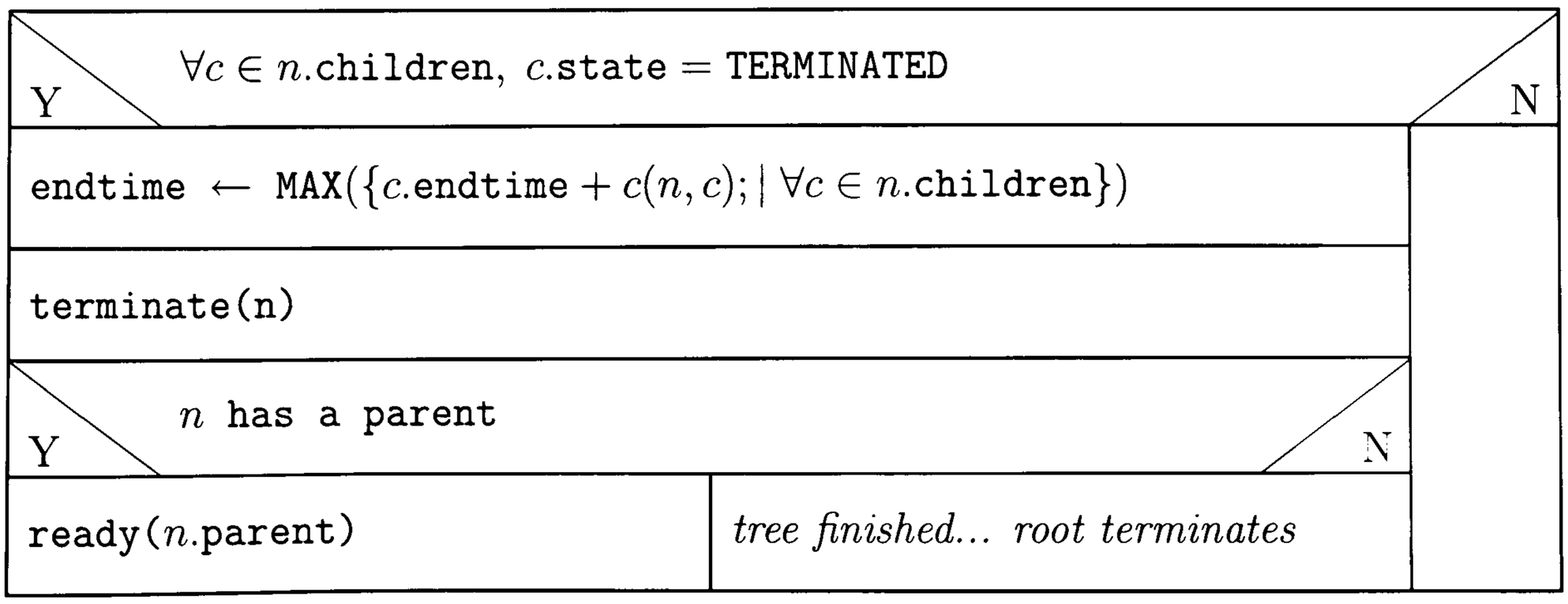
If a message has started transmission and a node is executing we have to take care, that the transmission time is decreased. Therefore we need to modify the `execute/1` method.

execute(n : node) —



The end time of a parent node has to be adjusted according to the communication time, too. In the tree structure we assume that the call to a child takes the same amount of time as the return call to the parent takes.

ready(n: node) —



If all children of a node have terminated, then the node can terminate itself.

The end time of the node is therefore the maximum of the endtime of its children plus their communication cost. The node in this case informs its parent that is has terminated. If n is the root node the application terminates.

4.4.4 Generalisation of the times estimation

This section shows how the existing algorithm can be generalised to take a DAG as an application structure. We only show which methods need to be modified to adjust the behaviour, for all others it is sufficient to replace the passed node by a vertex. Each vertex stores the same information as the nodes do for the tree structure.

To compare the DAG and the tree structure, we need to define the terms parent and child for the DAG:

Parent n_i is parent of n_j when (n_i, n_j) exists. A node in the DAG can have more than one parent.

Child n_i is child of n_j if n_j is parent of n_i.

Ancestor n_i is ancestor of n_j if either n_i is parent of n_j or it exists a n_k such that n_k is parent of n_j and n_i is ancestor of n_k.

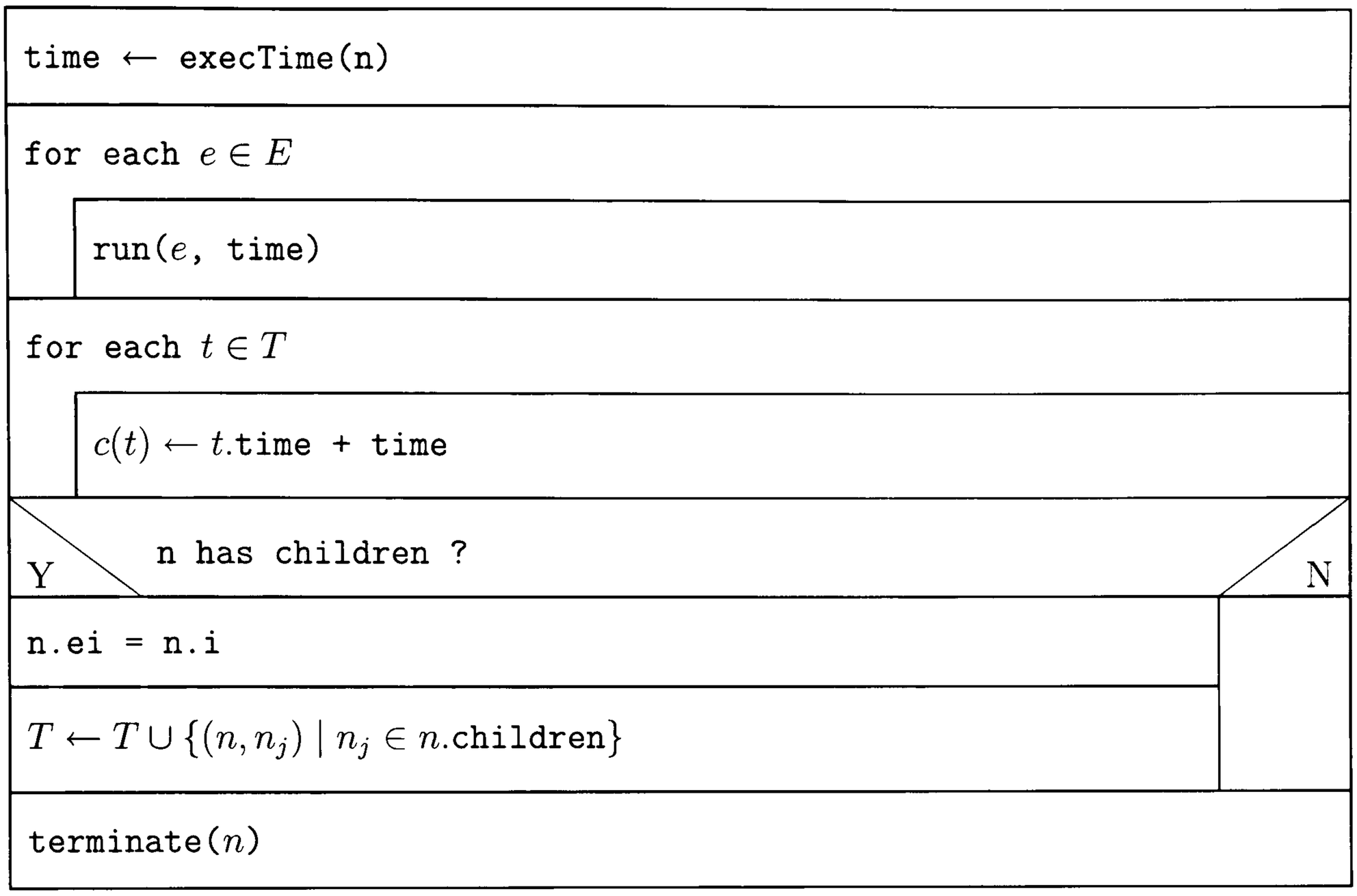
Descendant n_i is descendant of n_j if n_j is ancestor of n_i.

The main differences between the DAG and the tree structure are:

- Control flow does never return from a child to its parent.
- A node can have more than one parent.

As can be seen in Figure 4.4 the return through the tree is realised through the vertices (n_8, n_9, n_{10}). In the algorithm for the tree structure the return is realised through the `ready/1` method. If we convert from the tree structure to the DAG the `ready/1` method has to be removed, because the implicit return is now explicitly signed in the application structure.

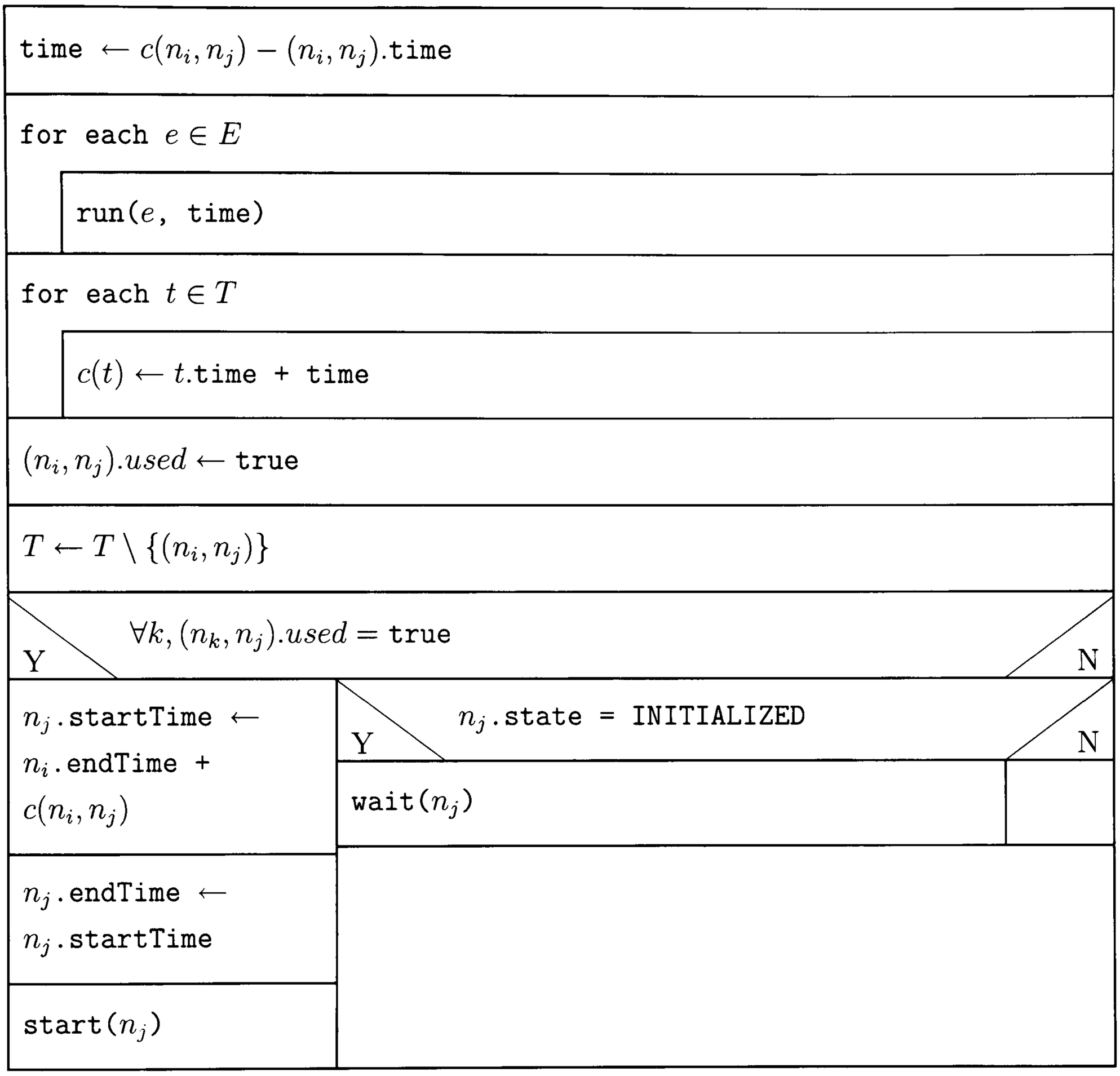
execute(n : node) —



Now the `ready/1` is never called and can therefore be removed for the DAG algorithm. The second point is that a vertex unlike a node can have more than one parent. As mentioned in Section 4.2.1 the vertex must not be started before all messages from its parents have arrived. This can be accomplished by modifying the `transmit/1` function.

Additional attributes For each message (arc) in the DAG it has to be stored if it has already been transmitted. This boolean flag we denote as (n_i, n_j).used. If `true` the message had already been transmitted. We use the notation `.used` instead of (n_i, n_j, used) to denote the extension to the previously explained parts of the algorithm.

transmit((n_i, n_j) **)** —



After an arc has been transmitted its *used* flag is set. The destination vertex is only started if all incoming messages have arrived. Otherwise the destination vertex is set to be waiting after the first message arrival (but only once).

Obviously we are now able to estimate the times for a DAG in the same way as for the tree structure. Even the collision detection introduced in Section 4.4.3.3 does store the collisions of each vertex. The interpretation of these collisions is not as easy as it has been for the tree, but Section 5.5 does describe how this data can be used when defining the object context of a vertex for the optimisation.

4.4.4.1 Example for the DAG algorithm

In Figure 4.3 the qualitative diagram for the threads and their waiting and executing time is shown. For we now have the algorithm to compute this times we can label the layers with times to get a better impression of the concurrency within the application. For this example two machines are used, both have the characteristic values $\alpha = 1$ and $\beta = 0$. Table 4.5 lists the machines speed and the object distribution. To keep the example easy to follow there exist only one thread accessing nodes that belongs to one object.

machine	accommodates	speed
I	A, D	1 $\frac{instr}{ms}$
II	B, C	2 $\frac{instr}{ms}$

Table 4.5: Machines and distribution for DAG example

All non ϵ transitions take 10 ms time before they arrive. Figure 4.7 shows the start and end time of nodes and messages.

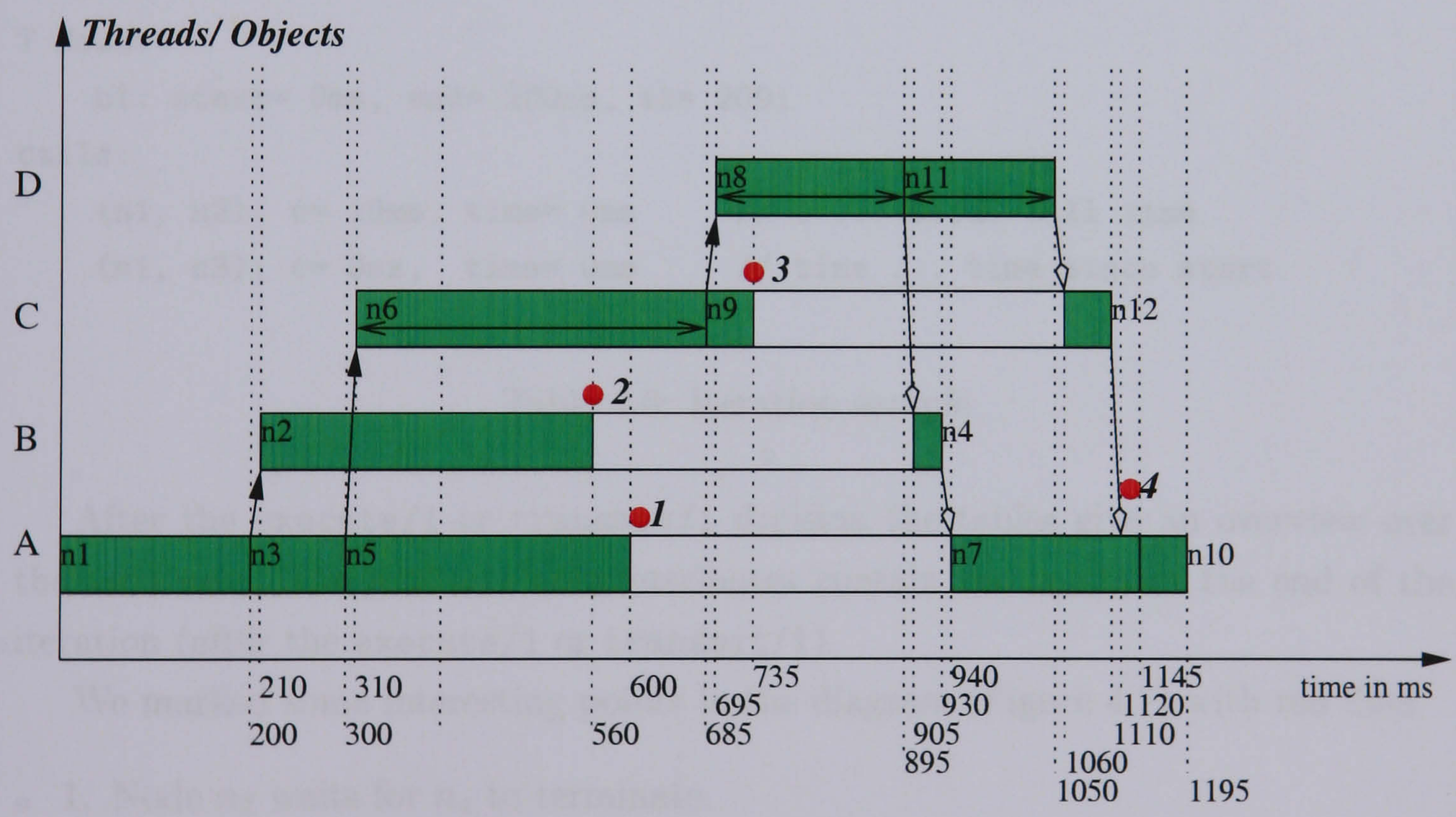


Figure 4.7: Time flow for the DAG example

The data for the times was calculated using the previously described algorithm, but without storing collision data. For each of the 27 iterations (12 nodes, 15

transitions) in the main part of the algorithm appendix A.1 provides a table, which shows the actual machine speed, the remaining times for all executing nodes and currently transmitted messages. In table 4.6 the data for one sample iteration is shown.

```
Iteration 0:
    ams(I)  = 1 i/ms                    // accommodates only n1
    ams(II) = 2 i/ms                    // not used
    first = n1 in 200ms                 // the initial value
    exec(n1):                           // execute n1 - run for 200ms

Sets:                                   // sets after execution
    E    = {}
    T    = {(n1, n2), (n1, n3)}
    I.E  = {}
    II.E = {}

                                        // the listing of nodes
E Nodes:                                // with their properties
W Nodes:
T Nodes:
    n1: start= 0ms, end= 200ms, ei= 200i
calls:
    (n1, n2): c= 10ms, time= 0ms     // c ... total call time
    (n1, n3): c= 0ms,  time= 0ms     // time ... time since start
```

Table 4.6: Iteration sample

After the `execute/1` or `transmit/1` decision the tables give an overview over the used sets. The sets and node attributes contain the values at the end of the iteration (after the `execute/1` or `transmit/1`).

We marked some interesting points in the diagram (Figure 4.7) with red dots.

1. Node n_7 waits for n_4 to terminate.

2. Node n_4 waits for n_8 to terminate.

3. Node n_{12} waits for n_{11} to terminate.

4. Message (n_{12}, n_{10}) arrives before n_7 terminates. Therefore n_{12} starts waiting from 1120ms for n_7 to terminate.

4.5 Summary

This chapter has gradually evolved an algorithm for the estimation of the execution time of a distributed application. First a simple application with sequential execution has been introduced. Then the application was evolved to execute in parallel, first without a slow down of the machine speed and then with a slow down. The slow down of a machine speed depends of the behaviour of a machine, taking into account that each machine runs a different operating system, has not the same number of processors and provides different speeds and memory resources.

Chapter 5

Optimisation of the Execution Time of a Globally Distributed Application

5.1 Introduction

This chapter describes an algorithm to optimises the distribution of objects to the machines in the NOW (explained in Chapter 4). Machines are not able to accommodate all objects. We restrict each object to be of a specific type, the service. Each machine provides a set of services, and can run several instances of those at a time. The algorithm has similarities with Ant Colony Optimisation (ACO)-algorithms [16], though it does not rely on global knowledge.

It is explained why an optimisation is desirable and it sketches the idea of the algorithm. The shown structure diagrams, can guideline an implementation. Several examples and pictures are provided, to explain the algorithm and its behaviour.

Parallel tasks within an distributed application compete. The execution time of those parts is highly dependent on the executing machine and the amount of instructions used to accomplish this task. In Section 5.2 we show that the greedy attempt described in Chapter 4.4.3.2 is not a good solution and how it is possible to find a better schedule. The static scheduling for an application represented by an DAG is proved to be an NP hard problem [26] even in an homogeneous system, where each processor executes only one vertex at a time. Facing this complexity we use a dynamic algorithm which has similarities with the ACO algorithm [16]. As

in the previous section the algorithm is explained first for the tree-structure and then generalised to the DAG. Section 5.3 describes the idea of the optimisation in an informal way, which is then explained in Section 5.4 in more detail.

5.2 Distributing to slower machines

The greedy schedule (schedule shown in Table 4.2) tends to locate all nodes on the same (fastest) machine. Due to this load the machine's speed will break down, and nodes will only have small part of the machine's speed to execute their instructions. This section shows a configuration where collisions are to be avoided, even if it meant to use a much slower machine. Table 5.1 and the flowchart in Figure 5.1 shows this more distributed version of the application.

Instance	Service	Machine
A	1	III
B	2	IV
C	3	II
D	4	III
E	2	I
F	3	II
G	2	IV

Table 5.1: Distribution

In this configuration there are no collisions between executing nodes. The four collisions involving waiting nodes:

1. B waiting collides with G executing.
2. G executing collides with B waiting.
3. A waiting collides with D executing.
4. D executing collides with A waiting.

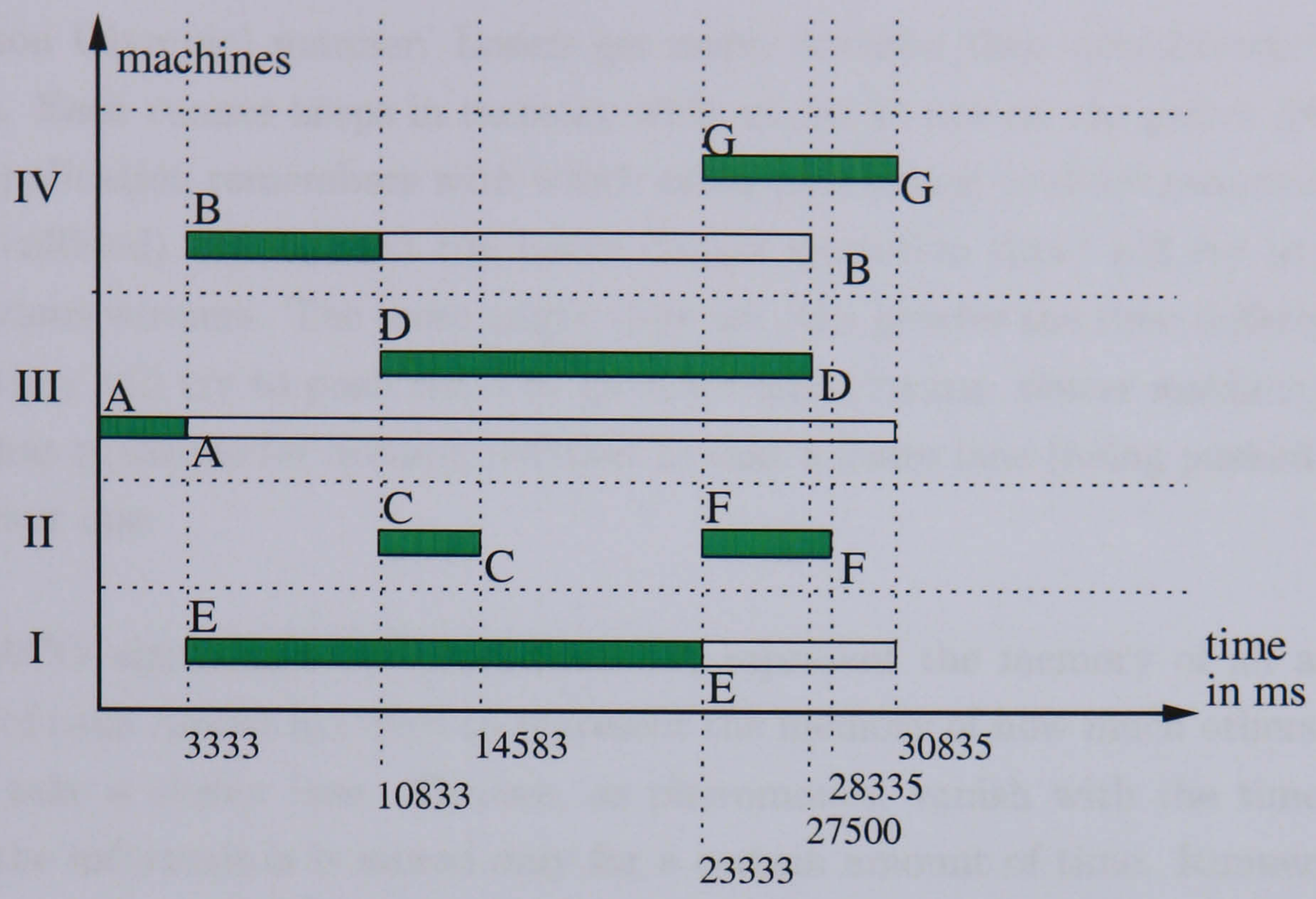


Figure 5.1: Timeflow more distributed

can be ignored if the machines β is 0. In this case all nodes have the full machine speed to execute. When comparing Figure 4.6 and 5.1 the distributed version terminates about 6 seconds earlier than the greedy configuration. Therefore it really makes sense to optimise the distribution. Actually this solution is the best possible distribution when machines provide only the services shown in Table 4.1 and the settings for $\alpha = 1$, $\beta = 0$.

5.3 Idea

First the optimisation for the tree structure described in Section 4.2.2 is discussed. Therefore each node represents one object. This section gives an informal description of the algorithm and is intended to be an easy way to understand the basic idea.

Think of the parallelised application as a relay. The parallel parts are represented by the different competitors, while the serial parts are represented by the runners of one team. Each runner wants his team to win, which means reaching the goal (which is the termination of the last part in the series) first. Even if the runners in our example are not human the idea is to have them behaving in the

same (non Olympic) manner: Losers get angry because they were slower than the winners. Each runner keeps in memory with whom he run on the gravel (each part of the application remembers with which other part it had to share resources - or so to say: collided). In the next run losers (longer execution time) will try to displace the previous winners. The more angry they are (the greater the time difference) the harder they will try to push them to an outer lane (means: slower machine). Every runner has to decide for himself, whether to take a faster lane (being pushed harder) or a slower one.

As ACO algorithms use pheromones to represent the memory of all ants, the bruises of each runner are used to represent the memory of how much others wanted him to take a slower lane. Bruises, as pheromones, vanish with the time, which means the information is stored only for a certain amount of time. Runners in our algorithm are quite unforgiving. If they are now slower than the one that displaced them, they push back (regardless, whether they are colliding or not!). How hard they push is determined by three values:

- The amount of races, that have been run since.
- How heavy the push was.
- The current difference in their (execution) time.

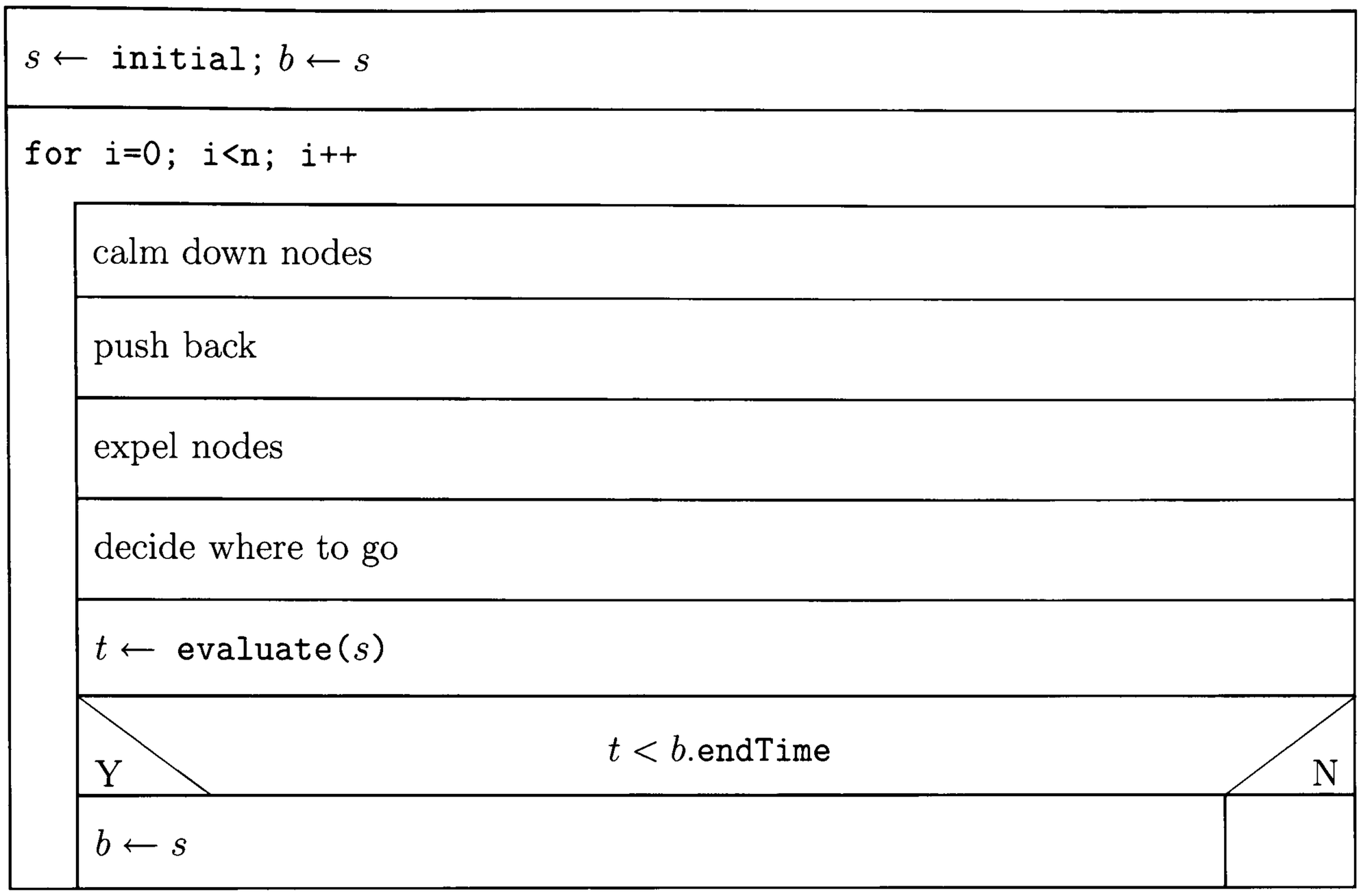
The main difference, between this algorithm and ACO is that there is no global knowledge like pheromones, which affects all competitors.

5.4 Structure

This section gives an overview of the main structure of the optimisation. The particular actions are described in the following sections in more detail.

The algorithm starts with an initial solution s. It runs for a specific amount of n steps, and stores the best found solution in b.

— Overview



With n it is possible to adjust the effort that is made to find a good solution. This algorithm does converge only for specific parameter settings, it does not necessarily mean that a higher n leads to a better result. But the solution found in more steps is guaranteed to be at least as good as solutions found in less steps.

5.4.1 Calm down

Each node n stores in a punishment value $p_{n,m}$ (double), how much they have been pushed away from a machine, for each machine m where it can be executed ($\{m \mid n.\text{service} \in m.S\}$). This value represents the fear of the node to go back to this machine. This memory is decreased using the following function:

$$p_{n,m} \xrightarrow{i \to i+1} (\lambda_0 p_{n,m}) \tag{5.1}$$

λ_0 ..... *factor in* $[0:1]$

λ_0 describes the percentage of the memory that is used for the next iteration i. A setting of $\lambda_0 = 1$ means that the memory is never declining, where a setting of $\lambda_0 = 0$ means there is no memory at all. λ_0 is comparable to the evaporation of pheromones in ant algorithms.

Additionally to the punishment value we store the maximum punishment value that has ever occurred. If $p_{n,m}$ is set it is tested if $p_{max_{n,m}} < p_{n,m}$. In this case $p_{max_{n,m}}$ is assigned to the new higher value. Even the memory of the hardest punishment is allowed to fade with the time. Instead of using the same factor λ for all evaporation factors it is distinguished to see the different behaviours. The hardest punishment value is decreasing with the factor λ_1.

$$p_{max_{n,m}} \xrightarrow{i \to i+1} (\lambda_1 p_{max_{n,m}}) \tag{5.2}$$

λ_1 *must be a value in* $[\lambda_0, 1]$

The third type of memory are the push back lists, which are explained in the next section in more detail. Each of these lists stores a collision instruction count $ci_{n,m,N}$ which represents the maximum number of collision instructions that have occurred between the nodes n and N on machine m. This value is used for the push back, and decreased in each iteration by the factor $\lambda_2 \in [0, 1]$.

$$ci_{n,m,N} \xrightarrow{i \to i+1} (\lambda_2 ci_{n,m,N}) \tag{5.3}$$

For the push back list that is constantly growing a lower limit ci_{min} for $ci_{n,m,N}$ is maintained. If a collision instruction stored in the push back lists is decreased below ci_{min} this entry is removed from the list.

— calm down

```
for each node in s
    for all m ∈ {m | n.service ∈ m.S}
        p_{n,m} ← λ0 p_{n,m}
        pmax_{n,m} ← λ1 pmax_{n,m}
        ci_{n,m,N} ← λ2 ci_{n,m,N}
```

5.4.2 Push back

Each node stores for each machine a list of nodes which had previously tried to displace it. For each node in the list the maximum number of collision instructions ($ci_{n,m,N}$) is maintained. Push back means the attempt to expel all nodes that are stored in the list for machines that are faster than the current location. Additionally the other node has to be accommodated by this machine in the last run. When expelling other nodes from their location the number of collision instructions to determine the weight is used. For a push back this information is not available, because both nodes are not necessarily located on the same machine. To determine the weight of the push back we use the stored maximum number of collision instructions in the push back lists ($ci_{n,m,N}$). The actual push back is performed using the equation described in Section 5.4.3. The reason why the push back was introduced is discussed in Section 5.6.

5.4.3 Expel

Before a new evaluation all competitors try to displace the others to slower machines. How successful the attempt to expel the faster node is, is determined by equation 5.4. Let n_1 try to expel n_2 provided that n_1.endTime $> n_2$.endTime. Then c_1 is the number of instructions that n1 has executed during the collision of n_1 and n_2. If it is a *push back* then the saved maximum number of collision instructions (ci_{n_1,m,n_2}) for node n_2 is used instead of the c_1 value.

$$p_{n_2,n_1.m} \leftarrow p_{n_2,n_1.m} + c_1 \frac{n_1.\text{endTime} - n_2.\text{endTime}}{\text{root.endTime}} \tag{5.4}$$

root.endTime is the endtime of the whole application. The time difference is divided to relativise it to the total execution time. This means if the whole application takes 20000 ms to execute a time difference of 10 ms between two nodes does not weight as much as if the applications execution time was 100 ms.

If the new punishment value $p_{n_2,n_1.m}$ is greater than the stored maximum punishment value $p_{max_{n_2,n_1.m}}$ the new higher value is stored as the maximum:

$$p_{max_{n_2,n_1.m}} \leftarrow MAX(p_{max_{n_2,n_1.m}}, p_{n_2,n_1.m}) \tag{5.5}$$

Furthermore we have to store the collision in the push back list:

$$ci_{n_2,n_1.m,n_1} \leftarrow MAX(ci_{n_2,n_1.m,n_1}, c_1) \tag{5.6}$$

5.4.4 Decide where to go

The most difficult question is, what is the best machine for a node to be executed. Each node tries to find the answer for itself, taking in account its displacement value on this machine ($p_{n,m}$) and the maximum speed of the machine compared with the maximum possible speed.

$$\begin{aligned} \text{maxspeed} &= MAX(m.\text{speed} \in \{m \in M |\ n.\text{service} \in m.S\}) \\ \text{minspeed} &= MIN(m.\text{speed} \in \{m \in M |\ n.\text{service} \in m.S\}) \\ f(m) &= \left(\alpha \frac{m.\text{speed} - \text{minspeed}}{\text{maxspeed} - \text{minspeed}}\right)^{\frac{1}{\beta}} - \left(\gamma \frac{p_{n,m}}{p_{max_{n,m}}}\right)^{\frac{1}{\delta}} \end{aligned} \tag{5.7}$$

---

n ..... *the node that decides*
α ..... *factor in* $[0;1]$
β ..... *exponent in* $(0;\infty)$
γ ..... *factor in* $[0;1]$
δ ..... *exponent in* $(0;\infty)$

α and γ are factors that adjust how greedy the node chooses a fast machine, where γ represents how frightened the node is to go back to a specific machine. These factors only allow a linear adjustment. For exponential behaviour one has to adjust the β and δ parameters. The higher the value is the higher the influence of the summand becomes.
The special case that m.speed = maxspeed = minspeed, which means that there is only one machine speed for the execution of the node n we define the fraction

$$\frac{m.\text{speed} - \text{minspeed}}{\text{maxspeed} - \text{minspeed}} = \frac{0}{0} \equiv 1$$

to be 1. Analogous to this definition we define the second fraction to be 1 if there is no punishment value for the machine ($p_{n,m} = 0$) and therefore no maximum punishment value ($p_{max_{n,m}} = 0$).

$$\frac{p_{n,m}}{p_{max_{n,m}}} = \frac{0}{0} \equiv 1$$

Each node chooses the machine m where $f(m)$ has the highest value.

$$n.m \leftarrow m \ni f(m) = MAX(f(m_i)|n.service \in m_i.services, m_i \in M)$$

Where M is the set of all machines.

5.5 Generalisation of the optimisation

Transferring the optimisation described in Section 5.4 to the DAG means that all vertices will compete with each other. This might lead to unexpected results, when a vertex tries to expel another vertex, which operates on the same object (to say expels itself). These considerations lead to transfer collisions and times from their vertex context to their object context.

$$\text{object.startTime} = MIN(\{n.\text{startTime} \mid n.o = \text{object}\}) \tag{5.8}$$

$$\text{object.endTime} = MAX(\{n.\text{endTime} \mid n.o = \text{object}\}) \tag{5.9}$$

$$\text{object}.EC = \bigcup_{n,\, n.o=\text{object}} \{c \in n.EC \mid c.N.o \neq n.o\} \tag{5.10}$$

$$\text{object}.WC = \bigcup_{n,\, n.o=\text{object}} \{c \in n.WC \mid c.N.o \neq n.o\} \tag{5.11}$$

The set of collisions of an object is the union of all collisions that occur in vertices that operate on this object, without those colliding with other vertices corresponding to the same object. The latter is the case if two vertices execute at the same time on the same object.

We assume that the threads represented by the vertices join at the end of the application again. If threads run into an open end, and therefore the initial object is destroyed before the application finishes we cannot say if the object context represents the application in a decent way.

As mentioned at the beginning, this thesis does not cover the persistent storage of objects and the migration during their lifetimes. If this possibility would exist the vertex context of the collisions would be appropriate - however to measure the time an object migration takes needs to be specified.

Running the optimisation using the object context behaves similar for the DAG as it does for the tree structure.

5.6 To push or not to push – cycles

The push back is introduced in the algorithm to prevent cycles. Figure 5.2 shows how cycles occur for the tree example with reasonable parameter settings. The diagram in Figure 5.3 depicts how the push back changes the period of the cycles.

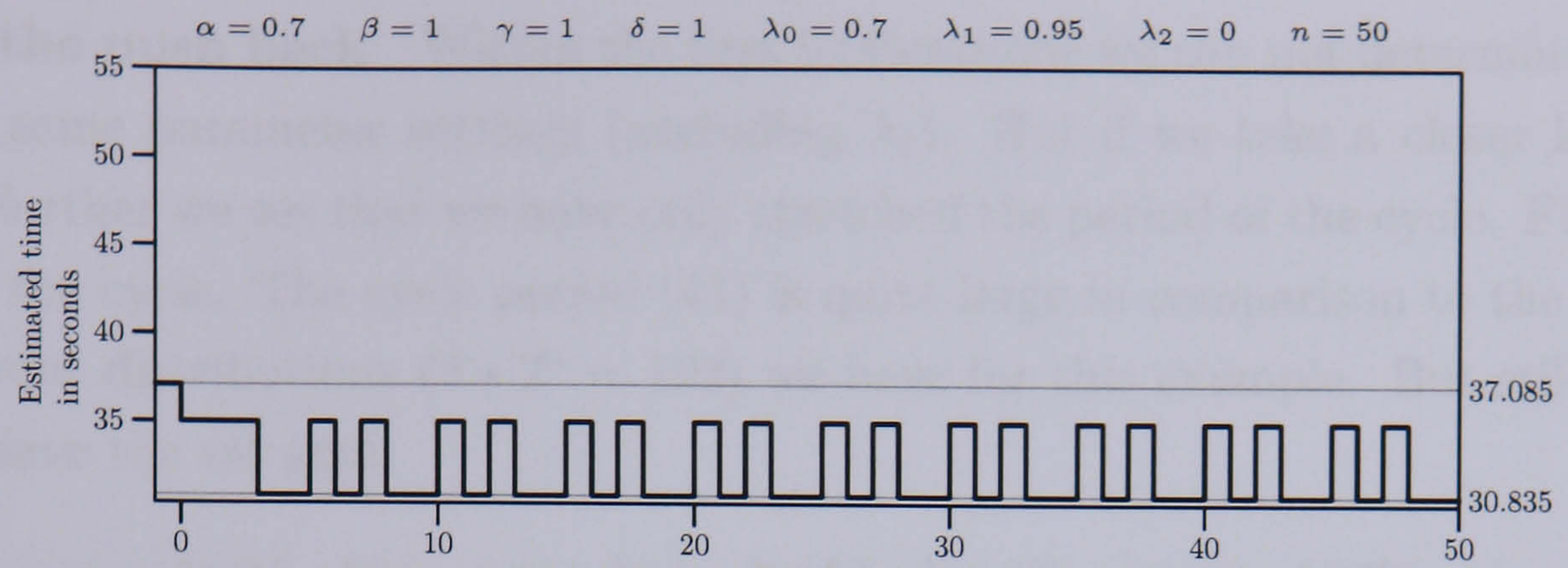


Figure 5.2: Without push back

Without push back The distributions between iteration 3 and 8 cyclic behaviour. The reason for this is, that the fraction $\frac{p_{n,m}}{pmax_{n,m}}$ for the chosen machine in iteration 3 is the same as in iteration 8. For the machines speed does not change we make the same decision we did already in iteration step 3, again. For this cycle will never lead to new solutions we could stop the optimisation process at iteration 8 when we detected the cycle without changing the end result.

In this example this would be a good decision because we have already found the best solution, but unfortunately we can build up cycles that do not contain the best known optimum. The example for the completely greedy behaviour (Figure 5.5) shows the extreme version of the cycle with a period of 1. To avoid these cycles, we introduced the push back.

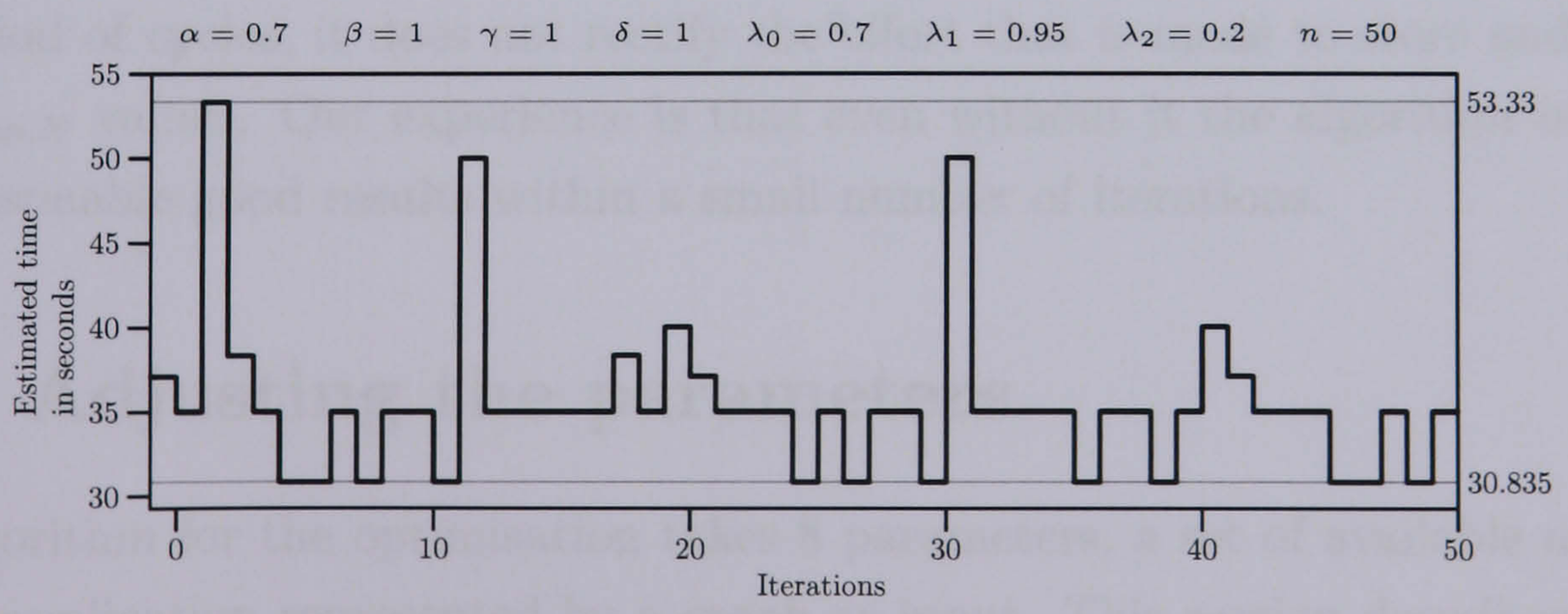


Figure 5.3: Push back with $\lambda_2 = 0.2$ over 50 iterations

Using the push back Within the first 50 iterations we can not determine a cycle for the same parameter settings (excluding λ_2). But if we take a closer look and iterate further we see that we have only stretched the period of the cycle. Figure 5.4 depicts the cycle. The cycle period (41) is quite large in comparison to the number of different distributions ($3 * 2^6 = 192$) we have for this example. But still we did not achieve the set goal.

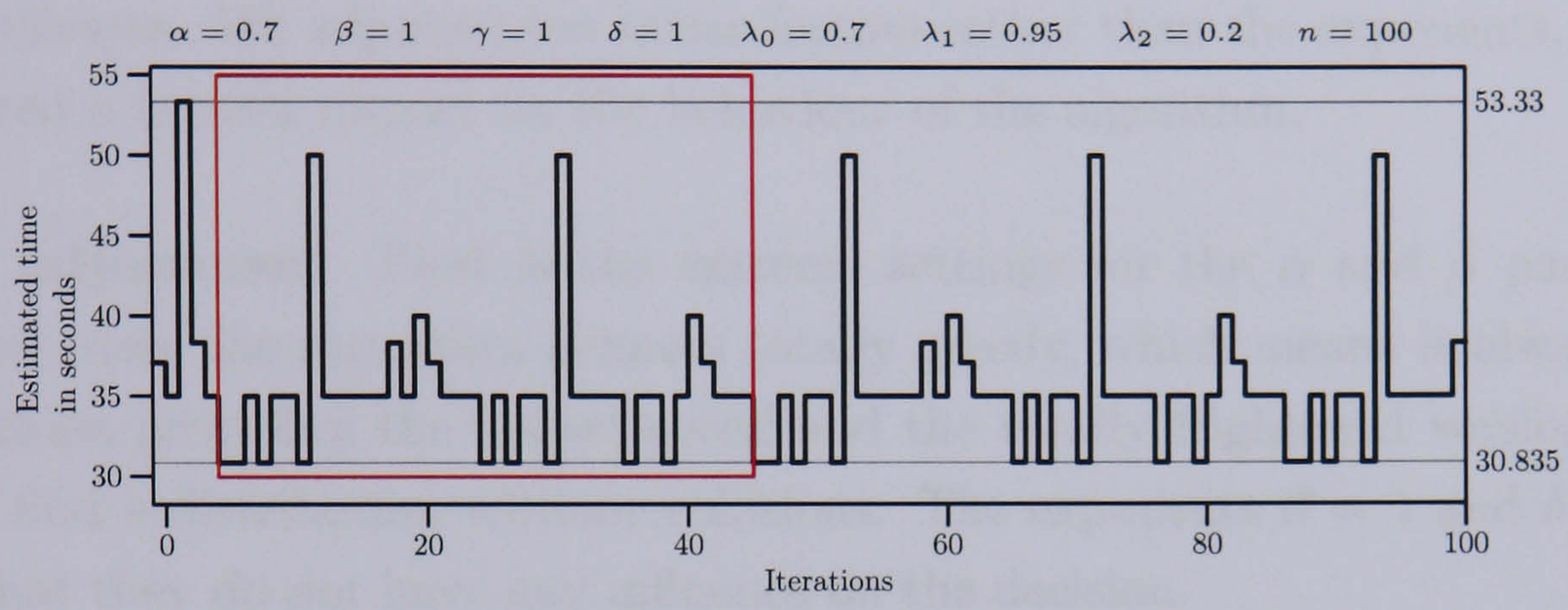


Figure 5.4: Push back with cycle over 100 iterations

Cycles can only occur if the relation $\frac{p_{n,m}}{pmax_{n,m}}$ is the same as for a previously processed equal distribution. Our experience is the period of cycles can be stretched by decreasing the α value - (nodes behaving less greedy, or increase the push back memory (λ_2)). Still cycles are the main problem of this algorithm and it is a challenging task to give the exact conditions that are responsible for cycles.

The use of the push back must still be questioned, because even if it can increase the period of cycles, it does not rectify the effort that is made to store and update the $ci_{n,m,N}$ values. Our experience is that even without it the algorithm is able to find reasonable good results within a small number of iterations.

5.7 Adjusting the parameters

The algorithm for the optimisation takes 8 parameters, a set of available machines and an application represented by a graph as input. This section describes the parameters and their relationship, as well as our experimental settings.

5.7.1 Adjusting the greediness and fear

As described in Section 5.4.4 the α and β value adjusts how greedy the algorithm looks for a fast machine. The values γ and δ specify how frightened an object is to choose a machine, were it was previously expelled from. The α and γ values allow a linear adjustment, which is discussed first. Good results were observed, when a more frightened setting ($\alpha < \gamma$) was chosen, which allowed the usage of slower machines to avoid collisions. We adjusted the linear factors rather than the exponents, because we noticed a heavier impact on the behaviour of the algorithm.

Linear adjustment First is the extreme settings for the α and β parameters described, were the algorithm behaves totally greedy, which means it always takes the machine providing the fastest speed and the totally frightened version, which tries to find a distribution without collisions. The exponents $\beta = 1$ and $\delta = 1$ are set so that they do not have any influence on the decision.

Totally greedy Setting $\alpha = 1$ and $\gamma = 0$ means that the second summand in equation 5.7 is always zero. Therefore the punishment values do not weight in the decision of the new location – its behaviour is totally greedy. All nodes strive to choose the fastest possible machine. Figure 5.6 depicts the estimated times for this case.

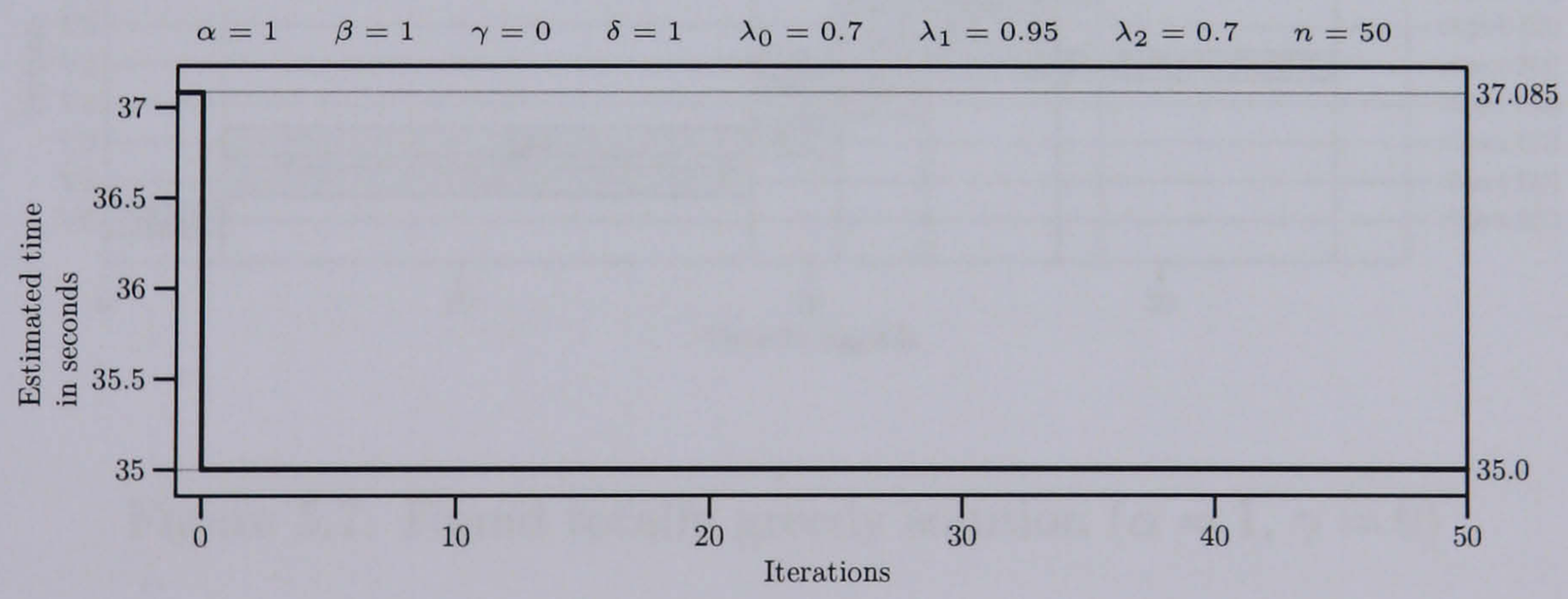


Figure 5.5: Totally greedy behaviour ($\alpha = 1$, $\gamma = 0$)

Because we already start the algorithm with a greedy solution it is interesting to see that the algorithm picks another solution and keeps it. If we only look at the machine speed for object 4, we see that its service is provided by three machines (I, II and IV see Table 4.1). Machine I is the slowest with a speed of $0.1\frac{instr}{ms}$ where II and

IV have the same speed ($0.2\frac{instr}{ms}$). For the greedy solution, it does not matter if the algorithm takes II or IV to execute object 4. In this particular case we have chosen the better one, just because it was checked first. This is the reason why we see two different distributions in Figure 5.5. Figures 5.6 and 5.7 show the distribution and estimated times for the initial greedy solution and the greedy solution chosen by the optimisation.

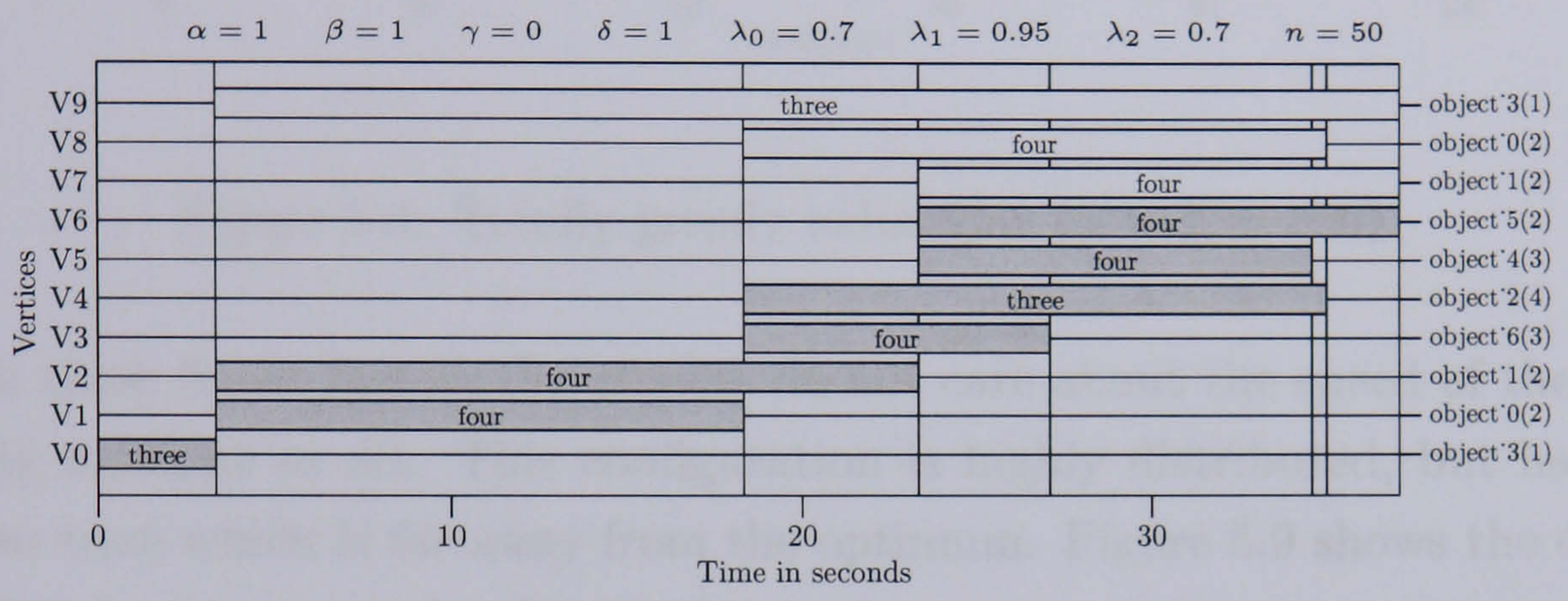


Figure 5.6: Initial greedy solution ($\alpha = 1$, $\gamma = 0$)

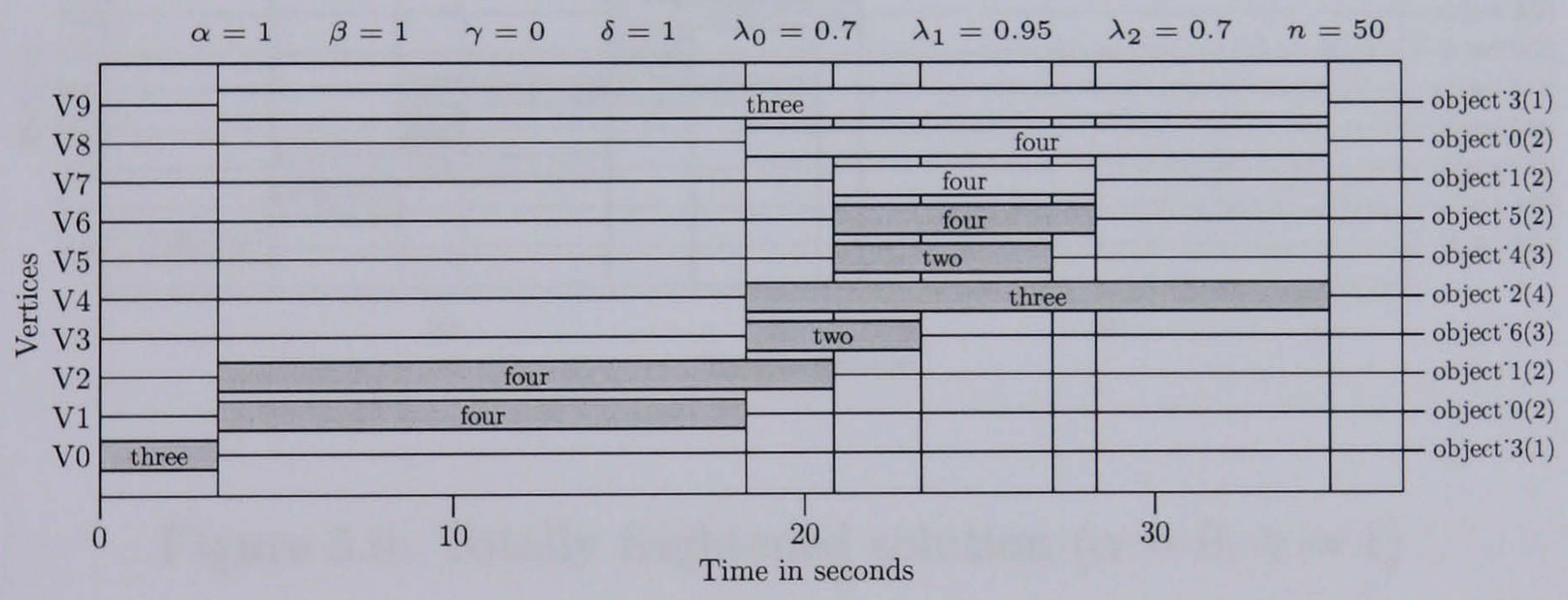


Figure 5.7: Found totally greedy solution ($\alpha = 1$, $\gamma = 0$)

Totally frightened If we have a look at the reverse parameter setting, $\alpha = 0$ and $\gamma = 1$ we experience that the objects try to avoid each other at all cost. This setting leads to the solution, which has the least collisions (the weight of a collision is given in equation 5.4). Figure 5.8 depicts the behaviour for this settings.

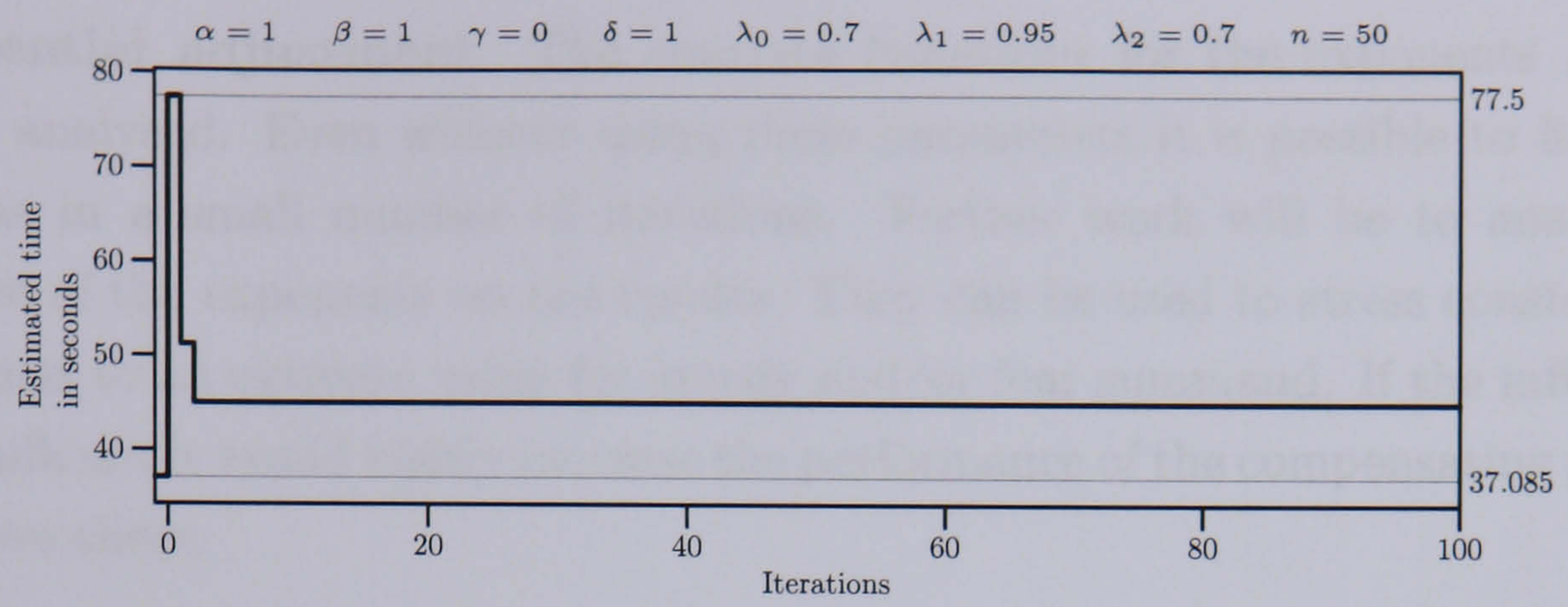


Figure 5.8: Totally greedy behaviour ($\alpha = 1$, $\gamma = 0$)

With these linear factors the objects do not care about the speed of the accommodating machine at all. This configuration is highly distributed, but has also a execution time which is far away from the optimum. Figure 5.9 shows the distribution and time estimation for the third iteration.

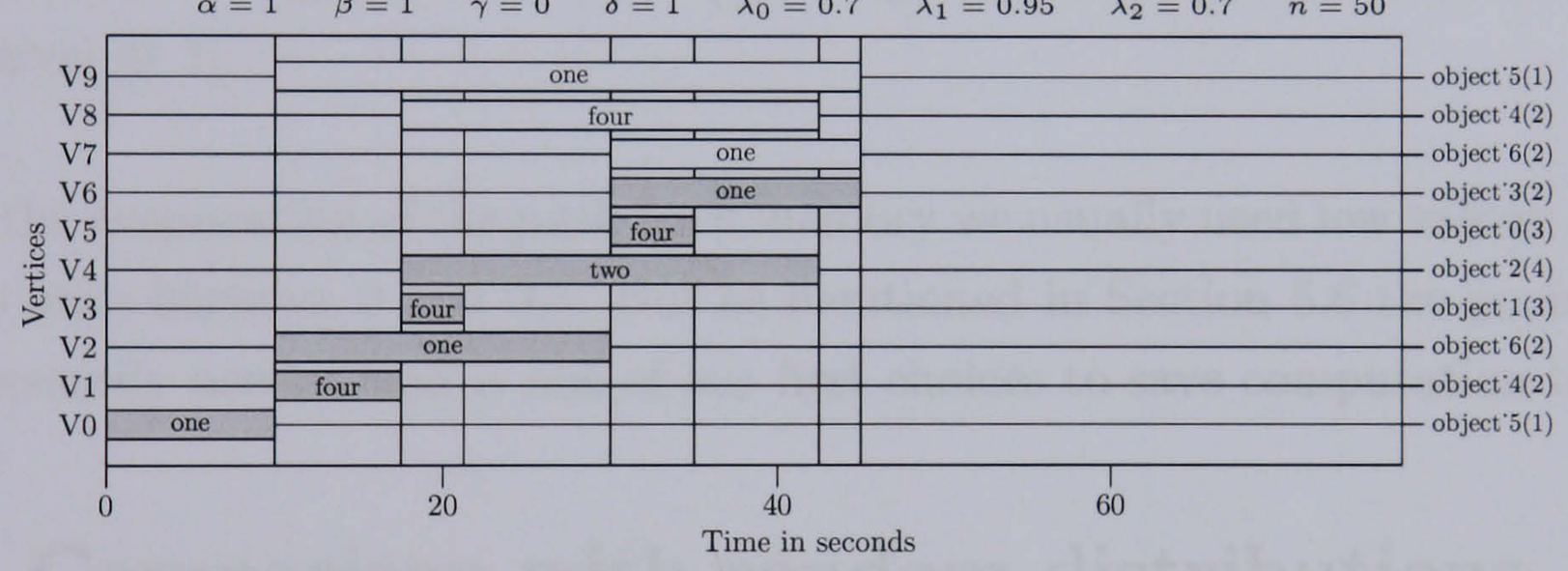


Figure 5.9: Totally frightened solution ($\alpha = 0$, $\gamma = 1$)

Reasonable settings for α and β Neither the greedy nor the frightened attempt leads to satisfying results. Our "best" values for the parameter settings are α values between 0.55 and 0.8, where we set $\gamma = 1$.

The more greedy the algorithm is, the higher is the probability to have short cycles. We experienced, that without the push back and with a $\alpha > 0.85$ and $\gamma = 1$ the period of the cycles often leaves little freedom to find a close to optimal solution. We chose for the examples α values between 0.55 and 0.8, where we set $\gamma = 1$.

Exponential adjustment The concrete behaviour for the exponents β and δ are not analysed. Even without using these parameters it is possible to find good solutions in a small number of iterations. Further work will be to analyse the influence of the exponents on the results. They can be used to stress constellations which lead to an extreme value for greedy and/or fear summand. If the influence is not significant it would highly increase the performance of the compensating function to remove them.

5.7.2 Adjusting the evaporation factors

The choice of the evaporation parameters has a high influence on the behaviour of the algorithm. The best results are experienced, for λ_0 between 0.5 and 0.9. The choice for λ_1 was close below 1 which means that the memory of the total punishment value does fade slowly. The difference between λ_1 and λ_0 specifies how greedy the algorithm becomes over the iterations. λ_1 must be greater than λ_0 because otherwise it cannot be guaranteed to keep the right summand (fear summand) within the interval $[0, 1]$.

For the evaporation of the push back memory we usually used low values. Typical settings were between 0 and 0.5. But as mentioned in Section 5.6 the push back is not necessarily needed and is one of our first choices to save computation time.

5.8 Comparison with random distributions

If we take a look at the different distributions that were evaluated during the iterative optimisation, one can get the impression that a randomly chosen distribution could lead to comparable results. Even if the diagrams shown in this section are not statistically reliable we think that they show that the knowledge of the nodes and the compensating function (equation 5.7) lead to good results.

Figure 5.10 shows the results for a bigger example. The DAG for this example consists of 24 vertices which are corresponding to 7 objects. Each object is available on 4 different machines. This makes a total of 2^{14} possible distributions. The red line in Figure 5.10 shows the estimated times of the described algorithm, where the black line shows the first 100 distributions chosen of a random sequence.

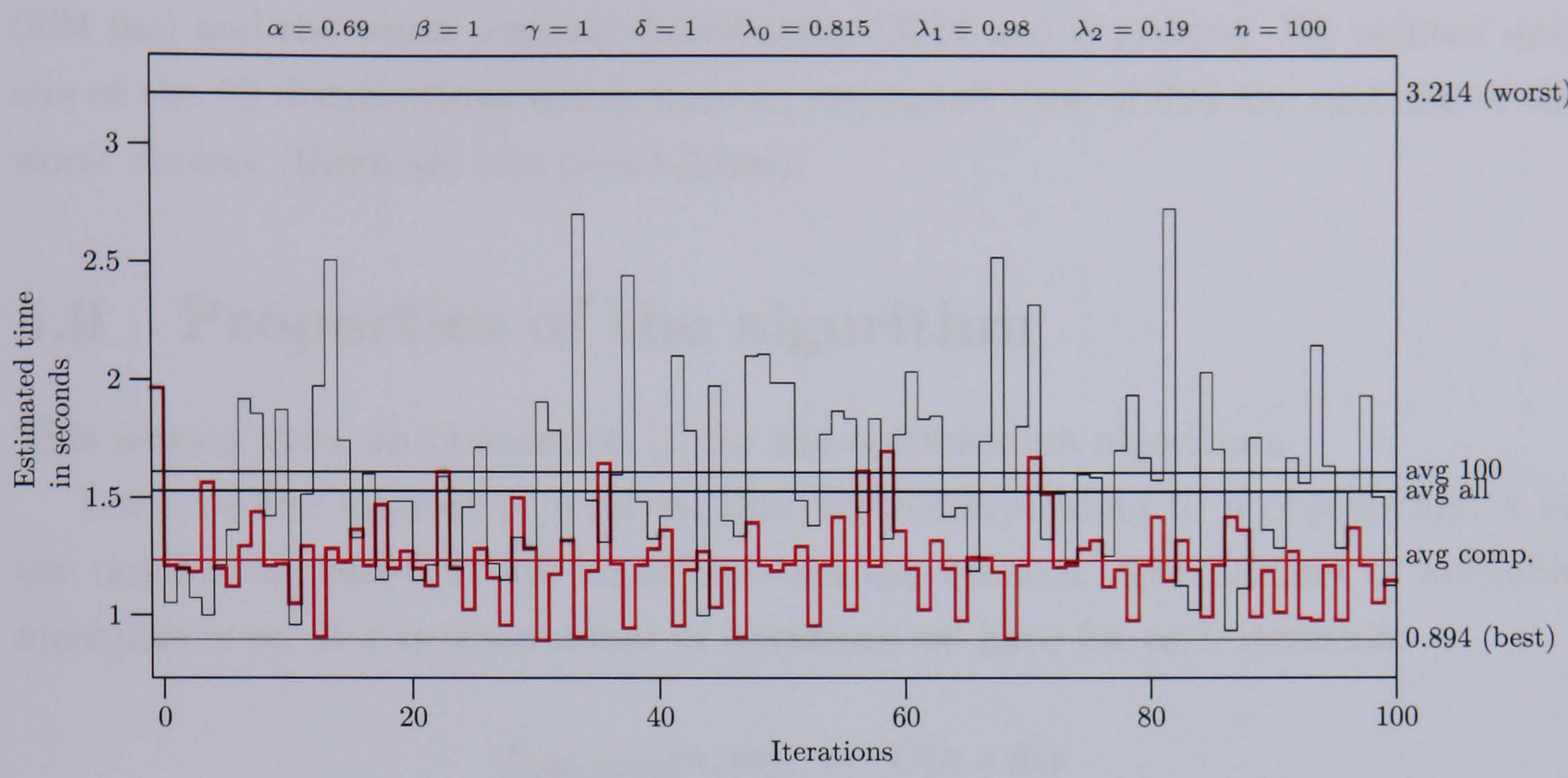


Figure 5.10: Random distributions versus the algorithm choices

We marked the best possible solution, the average time of all possible distributions, the average of the used random sequence and the worst possible distribution. Since 100 random distributions are not at all representative, we have generated a sequence of 1000 distributions and picked the subjective lowest interval out of this sequence. Figure 5.11 shows the comparison between these two sequences.

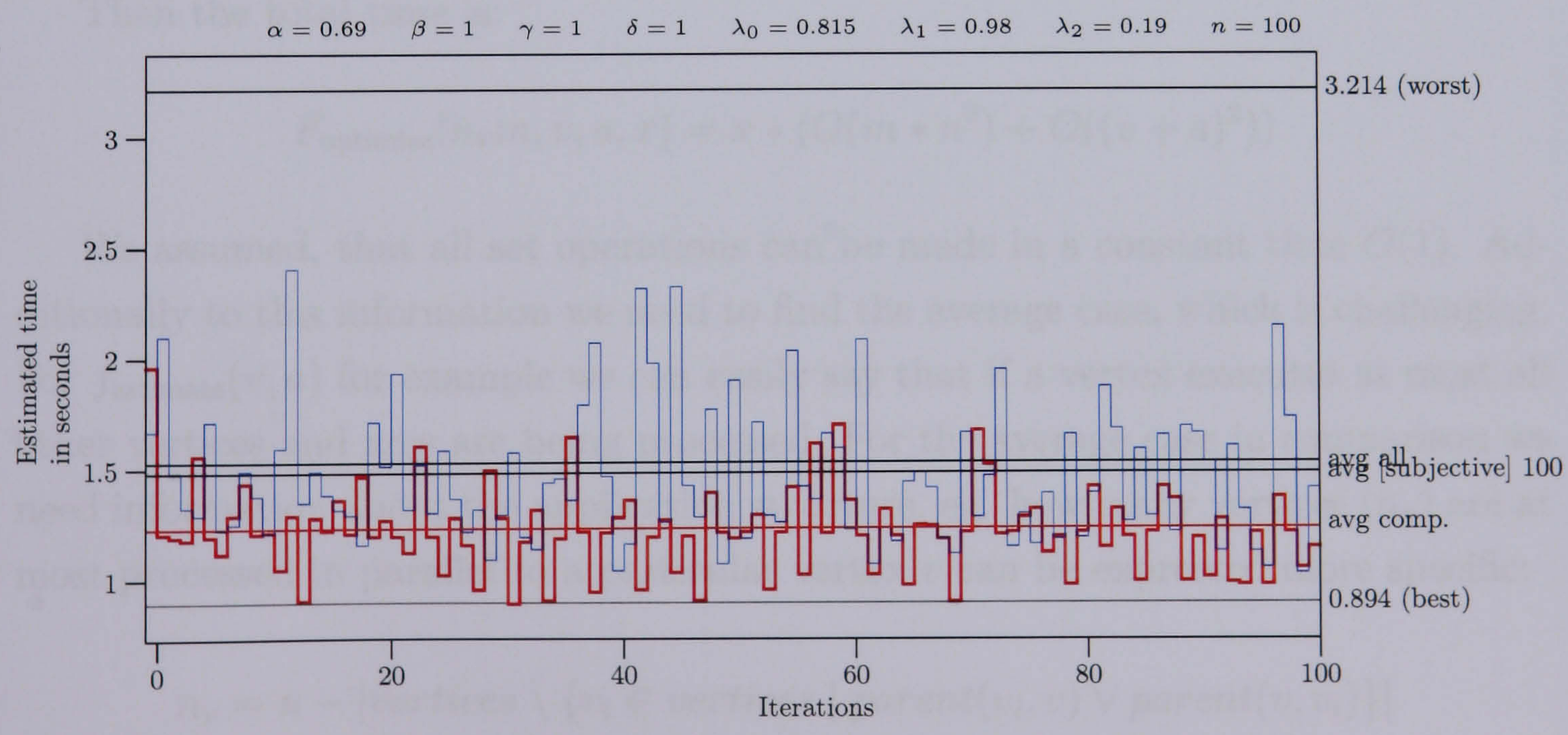


Figure 5.11: Random distributions versus the algorithm choices

In appendix A.2 the estimated execution times for the best found distribution (894 ms) and the worst possible distribution (3214 ms) is printed. We printed only one of the 63 distributions which have an estimated time of 894 ms, and one of the worst choices (there are two possibilities).

5.9 Properties of the algorithm

This section gives an O-notation [3] for the optimisation algorithm.

Let v be the number of vertices, that are corresponding to n objects and a be the number of calls that are made between the vertices. The number of available machines is m. If x is the number of iterations we have for each iteration:

$$\begin{aligned} f_{\text{calm down}}(n,m) &= O(n*m) \\ f_{\text{push back}}(n,m) &= O(n^2*m) \\ f_{\text{expel}}(n) &= O(n^2) \\ f_{\text{decide}}(n,m) &= O(n*m) \\ f_{\text{estimate}}(v,a) &= O((v+a)^2) \\ f_{\text{save solution}}(n) &= O(n) \end{aligned}$$

Then the total time is:

$$F_{\text{optimise}}(n,m,v,a,x) = x*(O(m*n^2)+O((v+a)^2))$$

We assumed, that all set operations can be made in a constant time $O(1)$. Additionally to this information we need to find the average case, which is challenging. For $f_{\text{estimate}}(v,a)$ for example we can easily say that if a vertex executes at most all other vertices and arcs are being processed. For the average case in comparison we need information about the application structure, eg. how many vertices (n_v) are at most processed in parallel to a particular vertex v can be expressed more specific:

$$n_v = n - |vertices \setminus \{v_i \in vertices \mid parent(v_i,v) \vee parent(v,v_i)\}|$$

5.10 Summary

This chapter has described a static optimisation algorithm to schedule application parts. In case every application part will be executed on the fastest possible machine, the overall execution time might slow down enormously, because a vast number of nodes will execute in parallel and as such collide during their execution. The optimisation algorithm described in this chapter minimises the execution time where different nodes collide by changing the distribution of these nodes. The algorithm described in Chapter 4 is used to estimate the execution time for the whole application and the distribution with the minimal execution time is chosen. However the algorithm presented in this chapter will not necessarily find the absolute optimal distribution, since scheduling represented by a DAG is proved to be a NP hard problem [26].

Chapter 6

The Simulation Model

6.1 Introduction

The simulation models a largely distributed application. It is written in Java, requires a JRE version 1.3 or higher to run, and contains the architectures described in Chapter 3. The purpose of this simulation is to compare the results of the three different models with each other.

Within the simulation there exist actually three different distributed systems.

1. the application itself,

2. the performance monitoring within the models

3. the location broker can be located on a different machine as well.

A general overview of the Grid Performance Software is shown in Figure 6.1. A short overview of the main features of the graphical user interface (GUI) is given in Section 6.2, while a complete user guide is given in Appendix B. Furthermore, Section 6.3 describes the implementation of the performance monitoring, and the handling of hardware and performance failures is demonstrated in Section 6.4. Finally, the scalability of the system is discussed in the last Section.

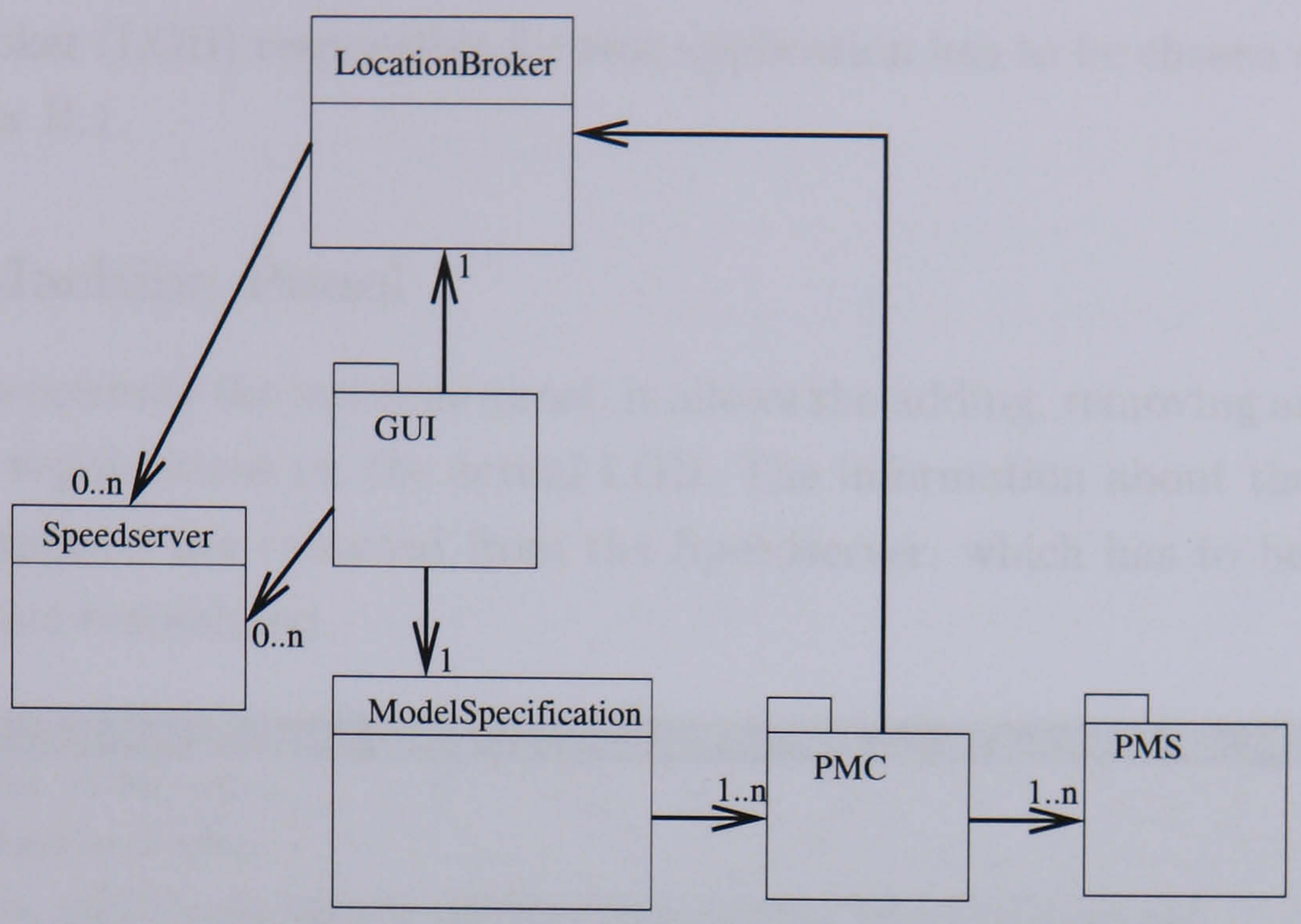


Figure 6.1: General overview of the Grid Performance Software

6.2 The Graphical User Interface

The Graphical User Interface (GUI) is divided into several parts as can be seen in Figure 6.2.

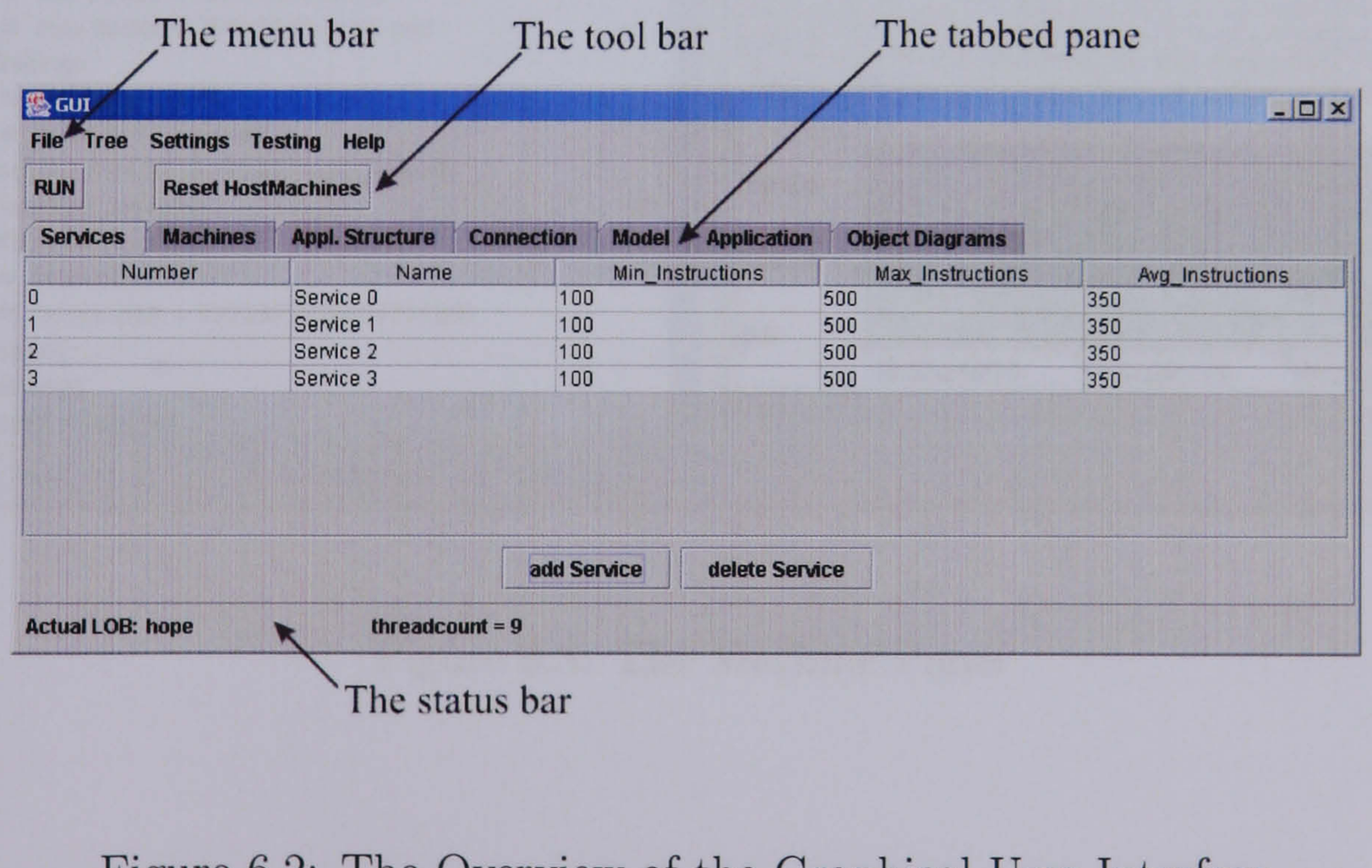


Figure 6.2: The Overview of the Graphical User Interface

Before any settings within the Graphical User Interface (GUI) the location of the

location broker (LOB) responsible for this application has to be chosen as described in Appendix B.1.

6.2.1 Machine Panel

Figure 6.3 represents the machine panel, it allows the adding, removing and changing of machine registrations on the actual LOB. The information about the speed of a machine displayed are collected from the Speedserver, which has to be started on every machine beforehand.

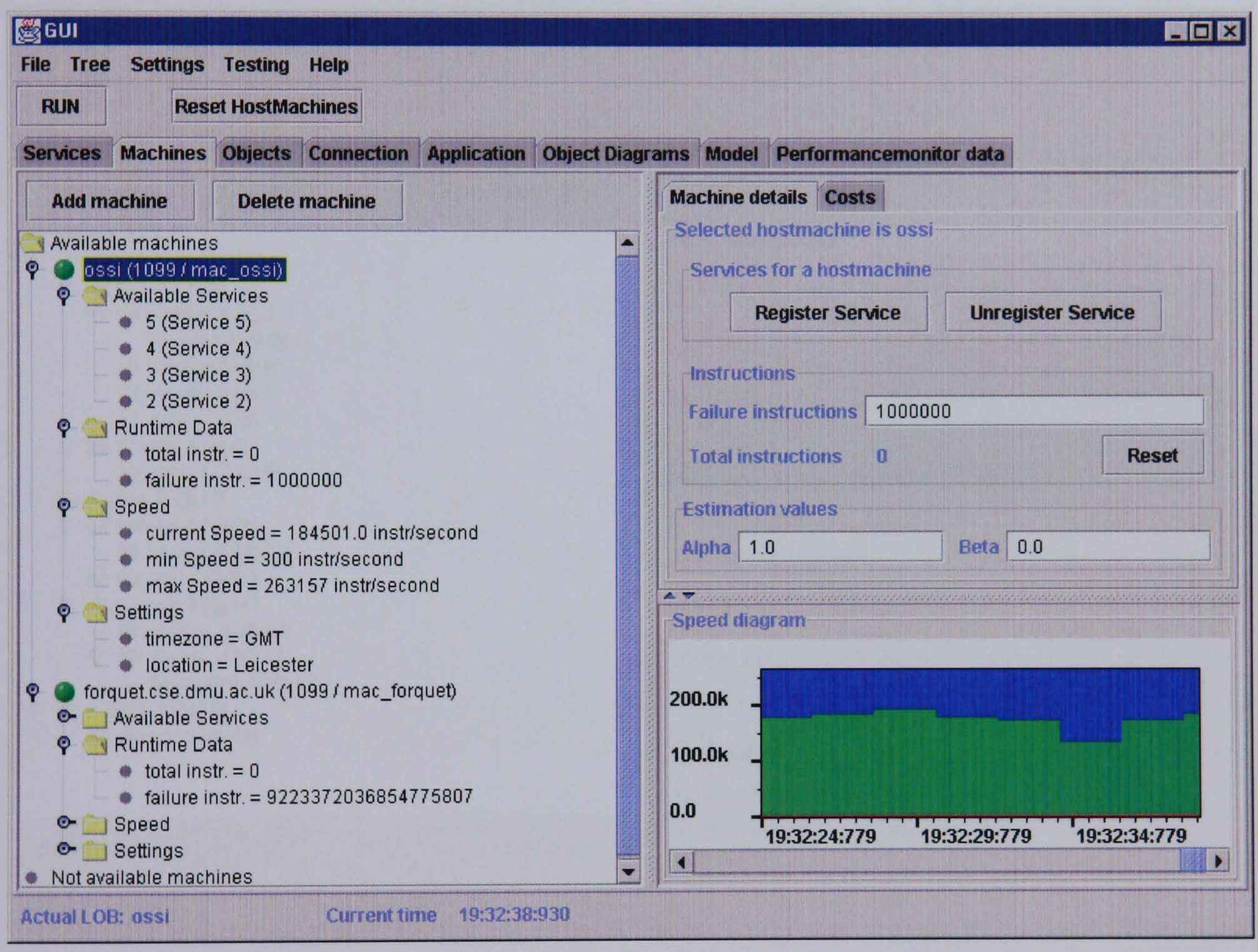


Figure 6.3: The Machine Panel

6.2.2 Application Structure Panel

The application structure panel is used to create a simulated application with all its parameters as can be seen in Figure 6.4. The application structure is shown as a tree. This tree structure can be altered by adding or removing nodes. Nodes in this tree represent an object (service instance) which is part of the application. During runtime of the simulated application the status of each object will change. This status will be identified by a coloured bullet in front of the related node. The different colours and their meaning are explained in Appendix B.1.3.3.

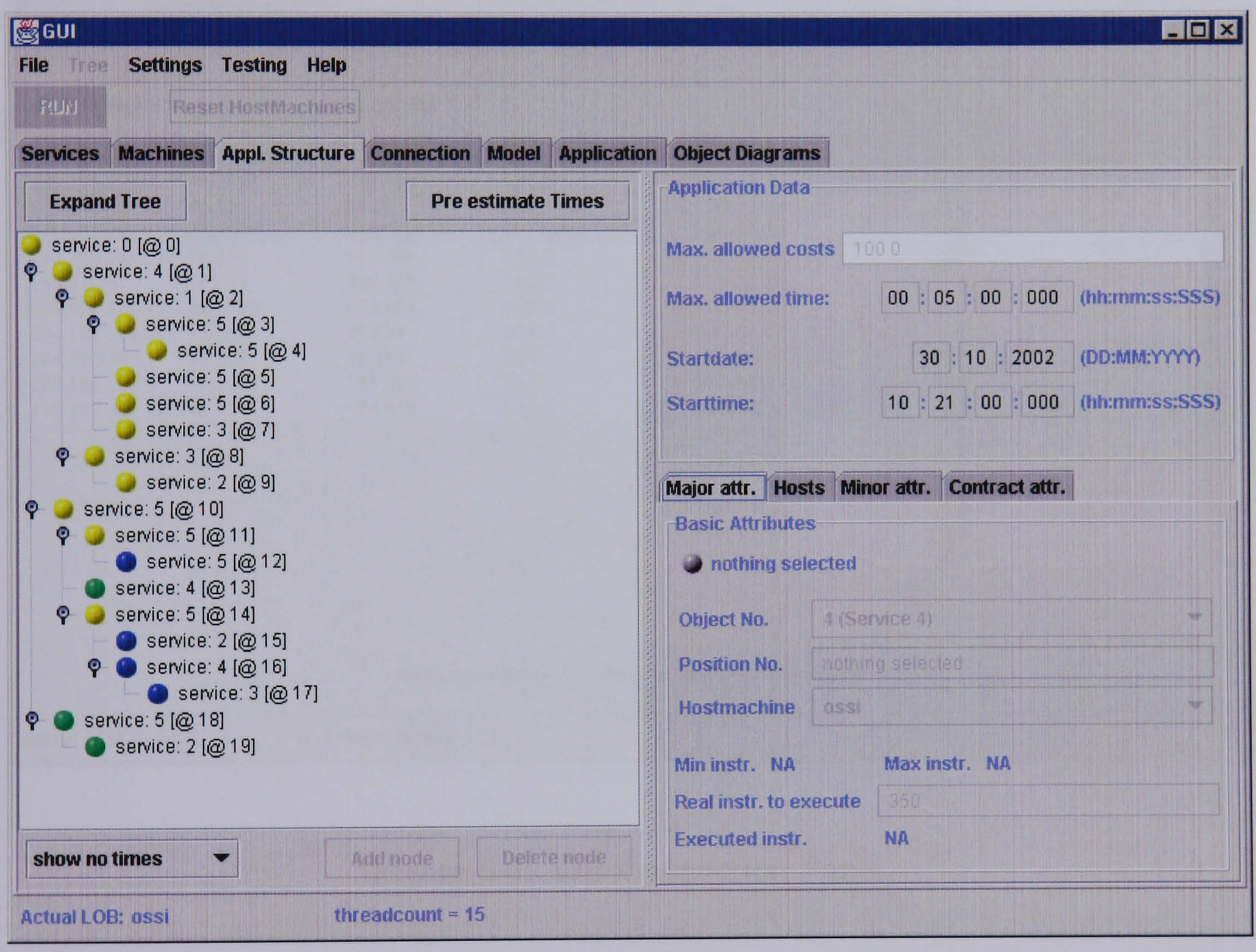


Figure 6.4: The Application Structure Panel

6.2.3 Connection Panel

The connection panel allows to add and remove connections between machines and change the data of these connections on the current LOB. This is to simulate a large global distributed system even when a local area network is used. Connections have a origin and destination address, a min- and max-delay between these connections in milliseconds and an availability in percent.

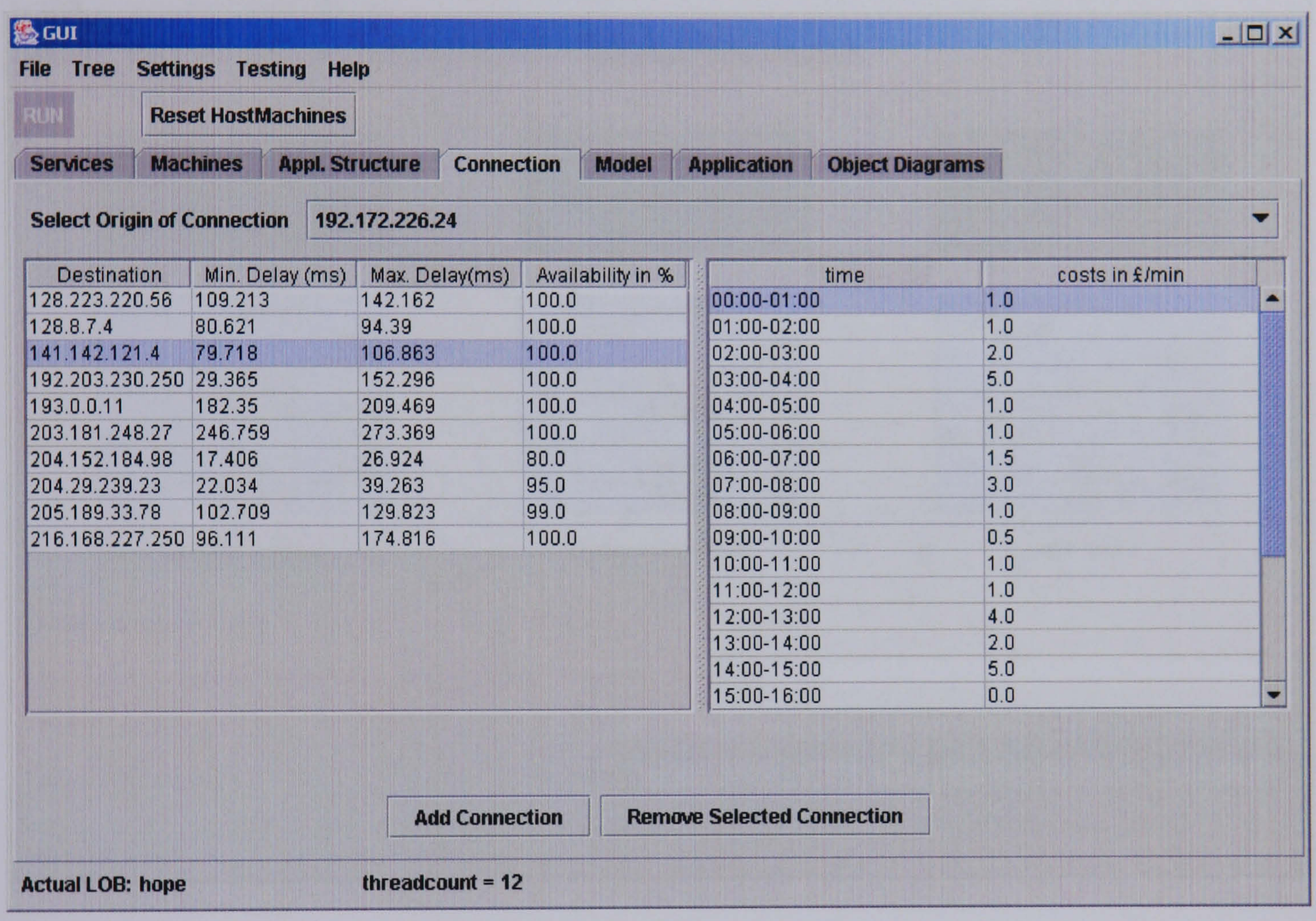


Figure 6.5: The Connection Panel

6.2.4 Model Panel

To execute an application one of the three performance monitoring models (see Figure 6.6) described in Chapter 3 have to be chosen. Each of these models determines the location of the performance monitor.

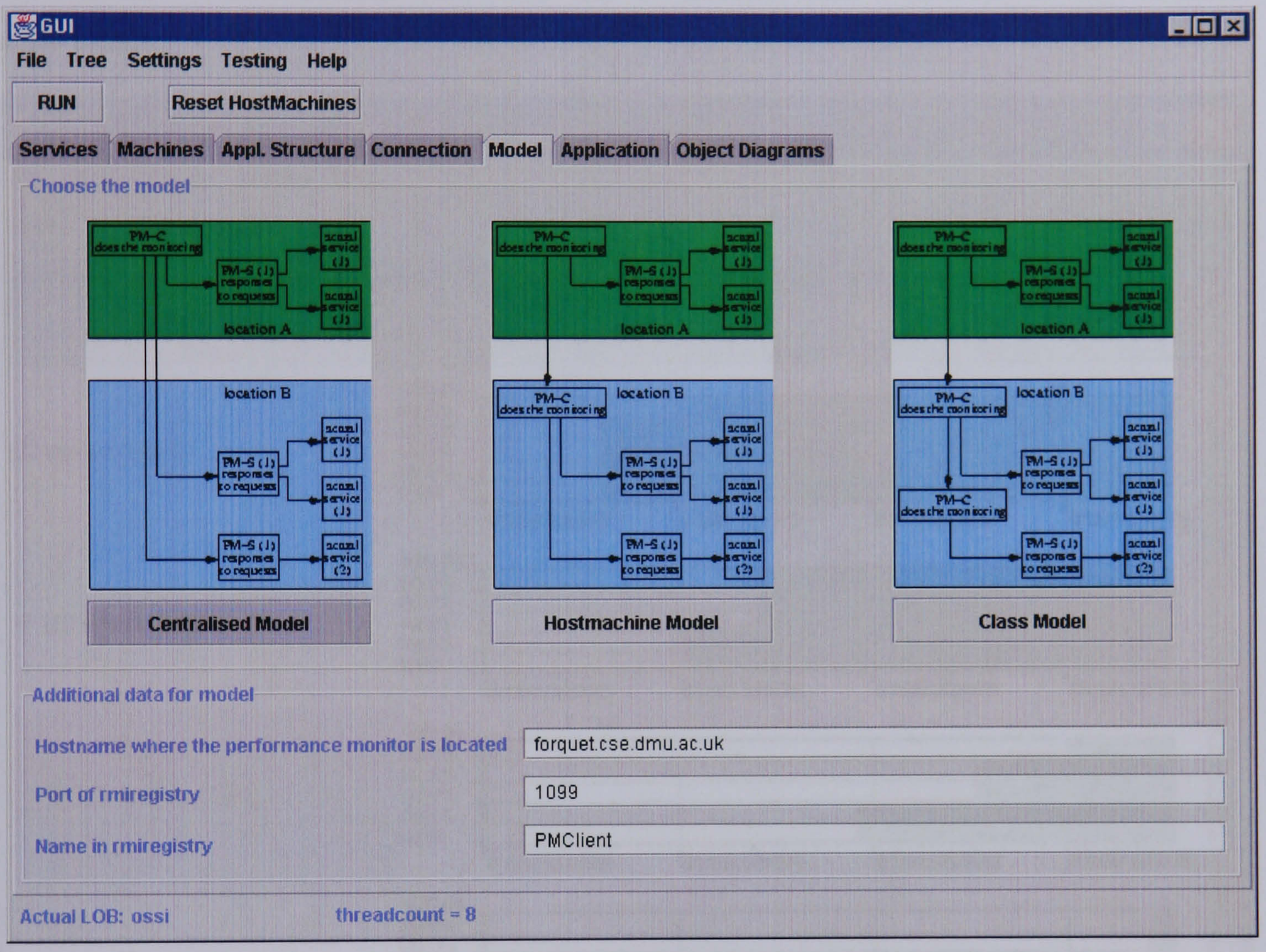


Figure 6.6: The Model Panel

6.2.5 Object Diagrams Panel

The Object Diagrams Panel shown in Figure 6.7 shows information about an executed application. The diagrams show graphically the behaviour during execution of each node. Each diagram has a vertical axis labeled with percentage values for the amount of instructions this node has to execute and a horizontal axis labeled with times.

Figure 6.7: The Object Diagrams Panel

6.3 The Implementation

Every remote call within the simulation is implemented with the help of Java Remote Method Invocation (RMI). RMI is part of the Java 2 platform. This framework enables communication between Java objects on different virtual and/or physical machines. An RMI application has usually two different parts: a client and a server. A typical server application creates objects (called remote objects) and when the client has a reference to the remote object it is able to invoke the methods out of these objects. The server usually registers a remote object in the RMI-registry. The client is then able to get the reference for the remote object out of the RMI-registry via a lookup. RMI uses stub and skeleton classes as proxies to simulate a reference on clients. However RMI does not allow any kind of location transparency which is highly required for a service-based application. This transparency is realised with the architecture of the Performance Monitoring Client and Server as described in Section 6.3.3 and 6.3.4. Further information about RMI can be found in [22].

6.3.1 Simulation of a Large Global Distributed System

The simulation is actually distributed over several machines within the local area network. However it is to simulate a large global distributed system. The additional delay created from the distance between far remote machines are stored in a "Connections" class and added with the help of some real measurements. These measurements are taken from [8] further information are available in Appendix B.3.1. These data always represent a *traceroute* between two machines on different locations on the globe. It represents the different delay at different times of the day as well as the failure rate, when it was not possible to reach the remote location.

6.3.2 The Simulated Application

The original location of every SAObject[1] has been specified by the user interface (compare Section B.1) and additionally with the help of an optimisation algorithm as described in chapter 4.

Pressing the run button in the user interface starts the execution of the simulation. This button is visible on the top left corner in Figure 6.2. As a first instance a class specified for the chosen model is called. This class only prepares the simulation,

[1]Simulated Application Object

starts the required performance monitors and calls the first performance-monitoring client (PMC) responsible for the root class of the simulated application.

The PMC[2] sends a request to the performance monitoring server (PMS)[3] responsible for the root class. This PMS instantiates the first SAObject in the hierarchy. Every SAObject within the simulation is identical. It receives as a parameter, the number of simulated instructions it should execute and which SAObjects it will call as a child, whereby it will call all its children in parallel, only after all instructions have been fully executed. The execution will take place with conventional termination in the way that every parent of an object call is a synchronisation point. That means an object is only returning and terminates itself when every child it has called has been terminated itself. The execution follows an asynchronous communication protocol in the sense of an *publish-subscribe* order.

A SAObject[4] will call its children only indirectly. In the sense that it sends a request, containing the information about the child it wants to call, to the PMS which originally has invoked itself. The PMS then decides if the original distribution demands a remote call or if it is scheduled for execution on the same machine. Depending on which performance monitoring model is currently running it will call the corresponding PMC. In case of the centralised model the whole application is monitored by only one PMC as can be seen in Figure3.1, whereby in the host model there exist for every different machine its own PMC as Figure3.4 presents and in the class model every different service on every machine will have its own PMC (compare Figure3.7).

6.3.3 Performance Monitoring Client

The Performance Monitoring Client (PMC) has to fulfil several tasks. The SAObjects are distributed over a large distributed system and an SAObject has no knowledge about the location of other SAObjects. To realise this transparency between the SAObjects a **middleware** is inserted between the SAObject calls. In this case the middleware is realised within the PMC. One of the responsibilities of a PMC is the knowledge about the SAObjects locations. The PMC has the knowledge about every SAObjects location specified before the start of the execution. However during runtime the situation might change and a different location is to be found. This

[2]More Information about the PMC are available in Section 6.3.3
[3]More Information about the PMS are available in Section 6.3.4
[4]Simulated Application Object

leads to the second purpose of the PMC which is the detection of failures. As soon as a host is not available anymore the PMC starts enquiring for alternative locations of all objects running on this host. The last purpose of the PMC is the actual reconfiguration[5]. Every SAObject failing during runtime need to be restarted on a different location. Furthermore the PMC prevents the start of an SAObject on a machine not functioning in a way it was originally planned.

The PMC consists of four classes as can be seen in Figure 6.8. The class PMClient is responsible for remote access and as such extends UnicastRemoteObject and implements PMClientIntf and HostMachineListener. The interface PMClientIntf consist all the methods needed for remote access and the interface HostMachineListener consist of the notify method for the LocationBroker to inform the PMC of events happening with a host machine the PMC has registered interest in.

Every call to a PMClient object contains an identifier for the application. In case this PMClient has never before dealt with the execution of this application it will create a new instance of the class PMC_Appl and keeps a reference inside a Hashtable. In case this PMClient was already involved in the execution of this application it will find a reference in this Hashtable and is able to call the object PMC_Appl again.

For every call made from this client a new instance of the PMClientLogic is created. In depends of the model and the location of the SAObject to be called, the PMClientLogic will either call directly the PMServer or it will call another PMClient on the remote machine.

A failure can be detected either in the PMClientLogic because of an RemoteException, or because the PMClient got informed via the notify method from the LocationBroker or from other PMClients detecting this failure.

In all three models, when the machine, where the PMC calling the root SAObject is located, fails no detection and reconfiguration will be possible. However this problem can be overcome when the root PMC is either located on the user's machine, because when this machine fails the user will notice and he or she is not interested in the results anymore. Another way to overcome this disadvantage is a replication for the root PMC. This replication needs to keep track of everything happening within the original PMC so that it can take over the work as soon as this machine fails.

[5]see also Section 6.4

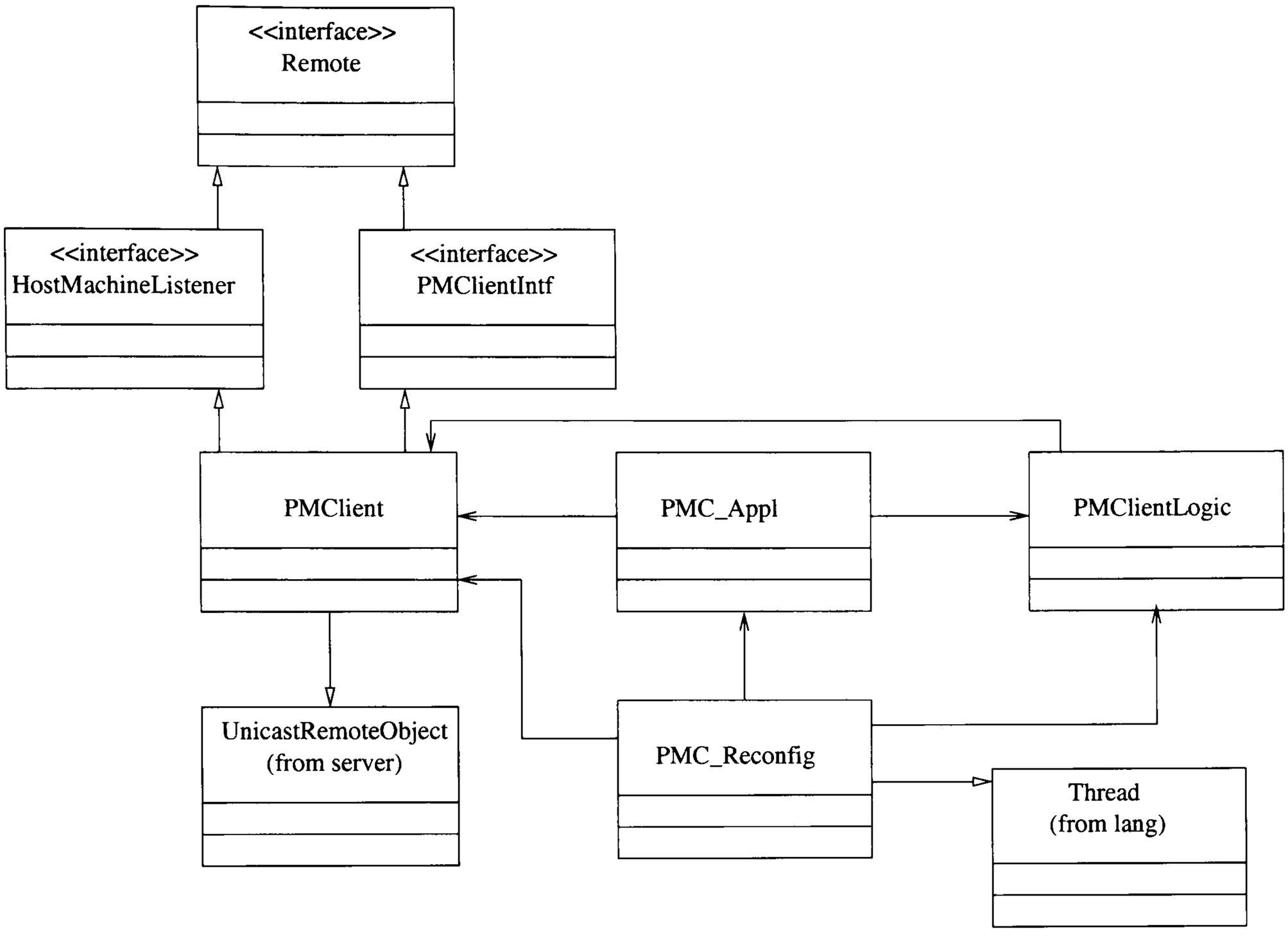


Figure 6.8: *Class Diagram for the Performance Monitoring Client*

6.3.4 Performance Monitoring Server

For every different SAObject class there exist exactly one Performance Monitoring Server (PMS) on every machine. The main function of a PMS is to allow remote access to the SAObjects. Each time a new SAObject is called the PMS creates a new instance of this class.

The PMS consist of three classes as can be seen in Figure 6.9. The class PMServer is responsible for remote access and as such extends UnicastRemoteObject and implements PMServerIntf and HostMachineListener. The interface PMServerIntf consist all the methods needed for remote access and the interface HostMachineListener consist of the notify method for the LocationBroker to inform the PMS in case the HostMachine this machine runs on has created a simulated machine failure. In case of a simulated machine failure every functionality of the PMS or the SAObject is stopped immediately.

When the PMServer receives a request to start an SAObject it would create a new instance of the class PMServerLogic. The PMServerLogic stores all the information needed for this call and creates a new instance of the SAObject class. The SAObject creates for every call to its children a new Thread PMServerCalling. This allows the execution of all children in parallel. The PMServerCalling Object has a reference to the PMServerLogic and as such always calls the PMC from where it was originally called. And the PMC decides if a reconfiguration[6] is necessary and if another PMC or PMS will have to be called.

[6]see also Section 6.4

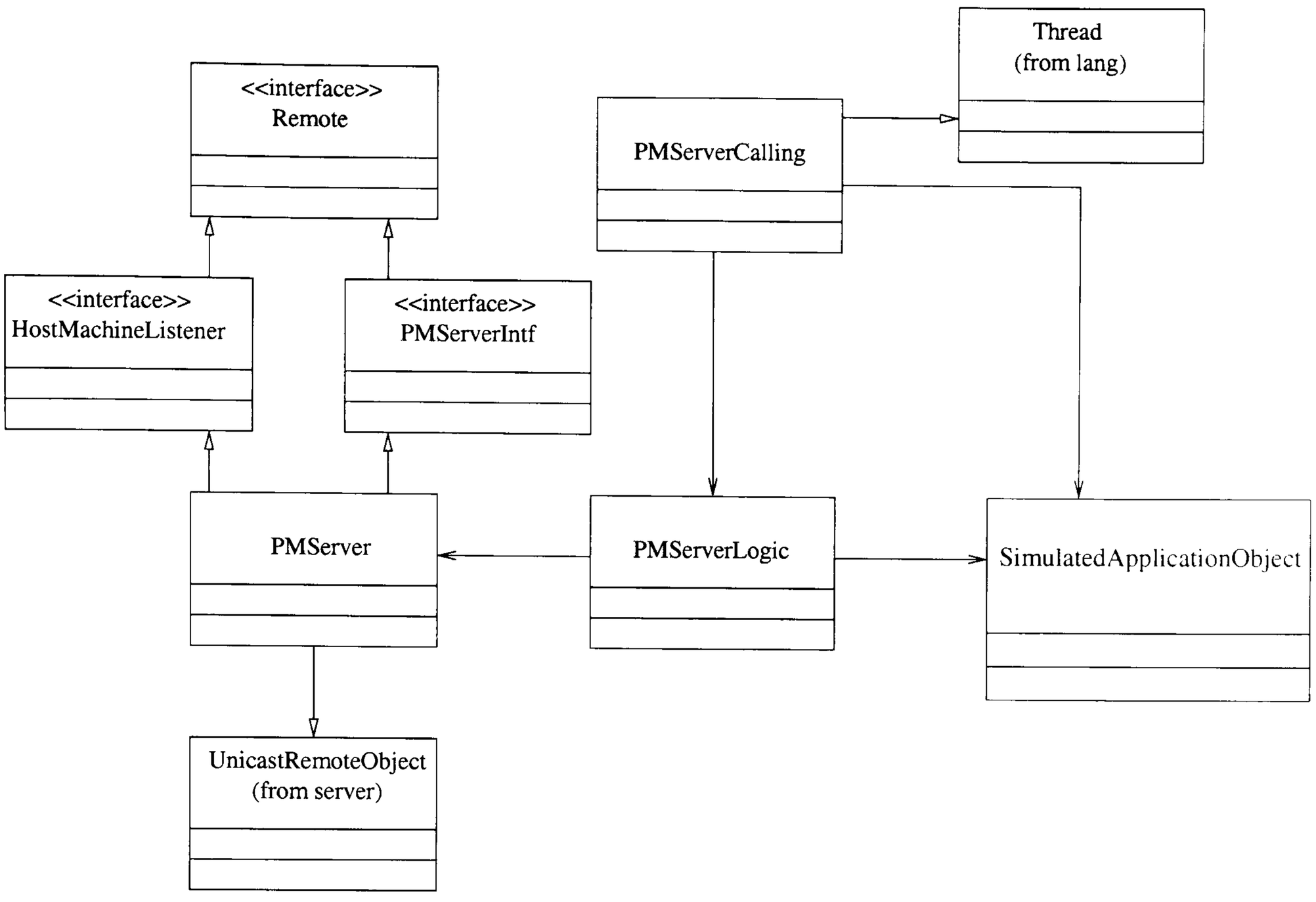


Figure 6.9: *Class Diagram for the Performance Monitoring Server and SimulatedApplicationObject*

6.3.5 Simulated Application Object

The SAObject itself has not much functionality. It is a dummy object which calls the HostMachine Class to execute its instructions and only afterwards it will call each of its children in a new thread. As soon as every child has returned the "results", which has no meaning for this dummy object, it will terminate itself and return the focus back to the PMServerLogic it had been called from.

6.4 Reconfiguration

There exist several reasons why a reconfiguration might be required. It is always possible that during execution an outage of hardware occurs. Whereby this failure can occur either on the network or on the machine where objects are being executed. Reason for reconfiguration are described in Section 6.4.1. Furthermore hardware or even software might be malfunctioning. Even though this is a serious problem this thesis assumes this type of failure will never occur. Finally during execution it might become clear that the contract originally finalised between the user and the service provider is at risk because of a delay within the execution. This delay might occur because of congestion or transmission errors within the network or because of machines being overloaded. Section 6.4.2 introduces an early warning reconfiguration schema to avoid a delay for further SAObjects execution before it becomes a problem for the whole application.

6.4.1 Reconfiguration due to a hardware failure

A hardware failure, either on the machine the object runs on or on the network connecting the machines, results in a failure of every object on this machine.

In case of a network failure the objects are still executing on a remote machine, however the connection is lost and as such there is no use for the rest of the application. When this machine becomes available after a reconfiguration has already been started, the PMC and PMS located on this machine will receive a notify method and are able to stop every still ongoing execution.

The simulation is able to react on real hardware failures. However it is possible to specify in the GUI that a HostMachine will fail after a certain amount of instruction executed on this machine. When these instructions have been executed the HostMachine throws a RemoteException in the same way as the Exception would be thrown when the machine fails in reality.

A hardware failure is detected in four different ways. Either the remote call creates an RMI RemoteException, the location broker detects the failure and sends a message to the corresponding performance monitors, or one of the other performance monitor detects the failure itself and send a message to all other performance monitors it knows of. Every PMC who has knowledge about other PMCs will inform each of them as soon as it has detected a failed machine. In this case every PMC

saves valuable time to start the reconfiguration. Furthermore a hardware failure can be detected by a time-out. Each time before a remote call a specified timer is started, and in the normal case this timer will expire after the remote call has already returned. However when the timer expires and the call has not yet returned it results, the PMC realises that it has to react and starts a reconfiguration.

In case a hardware failure is only simulated, the PMC, which first detects this failure, has to stop every object (the PMC, PMS and every SAObject) on this machine. Furthermore every PMC who has originally started one or more SAObject called from an SAObject on a failed machine need to be stopped. This PMC has to decide if this object will have to be restarted. Not every SAObject will have to be restarted, because when an SAObject has been called from an SAObject on the failed machine, this SAObject will automatically be restarted, when the previous SAObject reached the point in its execution to call this SAObject again. Only this time the performance monitor might choose a different location for all following SAObjects. Since the reconfiguration already has wasted valuable time the rest of the execution will have to speed up its execution time.

For example Figure 6.10 shows an application where object A is located on machine I, objects B, D, F and G are located on machine II, objects C and H are located on machine III and object E is located on machine IV. Let us now assume machine II fails while executing all of its objects. Consequently objects B,C,D,F and G have already started their execution and the results will be lost, and only objects B, F and G will have to be restarted. As a conclusion during runtime object B will call object C and D again. Only this time the performance monitor might choose a different location for object C, since the time for the execution, which has to be repeated, will have to be regained. The difference now is that the machines restarting the execution should run faster in case the overall time left for the execution is already at risk. Therefore these objects will have to speed up their execution. The performance monitor which has the knowledge about every call done from this particular object has to choose the location of the following objects very carefully, to assure that they will reduce their execution time.

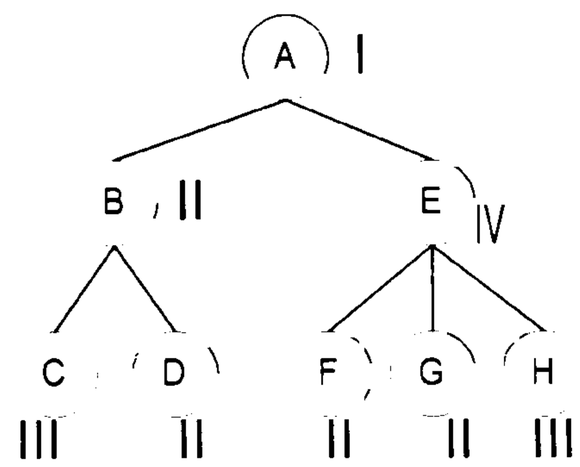


Figure 6.10: *example application*

To decide which SAObject has to be restarted and which SAObject simply has to

stop its execution, a history of every HostMachine is passed between every PMC. This history is simply a vector which stores the host name of a machine. Before every remote call this vector is copied and the new location is added to this vector and passed as an argument to the next PMC or in case of the centralised model to the next PMS.

6.4.2 Early Warning Performance Monitoring

An early detection of a delay within the execution results in a migration of all SAObjects, which have not yet started their execution, and will benefit from a migration. Whereby a delay can be caused when a machine runs low of memory, or simply is overloaded.

As described in Chapter 4 before the start of the execution it is estimated at what time every SAObject will terminate. Since every object has been called from a PMC it will return there after termination. The PMC then has to compare the estimated execution time with the actual time when the object has finished. If an object finishes after the estimated time a delay is detected and the PMC can find an alternative location for objects which haven't started their execution yet as demonstrated in Listing 6.1.

Because the reconfiguration takes time itself, it is advisable to continue executing as originally scheduled and at the same time run an optimisation algorithm as described in Chapter 5. The PMC which detected the delay will start a new Thread to optimise every SAObject which has not started executing, and when the results of the optimisation returns it then has to decide if these SAObjects have already started or even finished their execution. The PMC then informs all other PMC's of its knowledge (if there are any) about the rescheduling. And every SAObject started afterwards takes advantage of this rescheduling.

```
begin
  t_init    // t_init = time when execution starts
  run
  for (i=0; i<totalObjects; i++){
    object^i_term
    get t^i_term  // t^i_term = time when object^i terminates
    t_working = t^i_term − t_init
    if t_working ≤ WCT^i then
      continue // WCT^i = worst case execution time for object^i
    else
      reconfigure all not started objects
  }
end
```

Listing 6.1: The Early Warning Algorithm

6.4.3 Creation of Results

The purpose of this simulation is to compare the results of the centralised model with the host model and the class model[7]. To create these results the same application with the same distribution and the same time constraints is executed for a number of times for each model. All information collected during runtime, will be used to create statistical data for every model. These data allow to compare the different models with each other and are analysed in Chapter 7.

6.5 Scalability

As Colouris [10] states a system is described as scalable if it will remain effective when there is a significant increase in the number of resources and the number of users.

The centralised model as described in Chapter 3 is not fully scalable, because every remote call has to go through one performance monitor. For very large applications the performance monitor becomes a huge bottleneck and as such will be

[7]The three different models are described in Chapter 3

overloaded. However the performance monitoring algorithm in the host and the class model is scalable, since there is no central control anywhere in the system.

As for the current implementation, the performance monitor reconfigures the system directly at the failed object. Every object executed after the failed object in the hierarchy will be terminated, and a new instance will be created from the replacement of the failed object. This reduces the scalability for large applications, however this problem is easily overcome, when persistent storage is implemented. The persistent storage would save all results created during runtime of an object, so that this information is not lost, in case the execution would have to be repeated.

6.6 Summary

This chapter has described the Grid Performance Software designed and developed to simulate a large global distributed system. The user interface is described briefly and further information is provided in Appendix B. In addition this chapter introduced the implementation of the performance monitoring software, this includes the relationship between the performance monitoring client and server and the simulated application objects. Furthermore the reconfiguration of hardware and performance failures were described. Finally we have discussed the aspects of scalability in the system.

Chapter 7

Case Study (e-learning)

7.1 Introduction

This chapter gives an illustrative example of a service which can be used distributed over a large global network. The choice of this example was motivated considering the fact that it has to be clear to a reader unfamiliar with the presented environment. Furthermore it proves the hypothesis introduced in Chapter 3.

As Dorai [15] et al states web based learning is rapidly emerging as an alternative to traditional classroom-based education, whereby a useful course/lecture browsing system would enable cross-referenced access to all the materials pertaining to a course in a synchronised manner. Graves [18] furthermore explains the technology to learn at anyplace-anytime to take some of the friction out of the delivery of academic and student administrative services.

On the other hand such an e-learning course could use additionally some more traditional ways. Assume a university in England would provide online courses, whereby each module could be provided at different locations around the world. A lecturer, e. g. somewhere in Germany, would start the lecture for a duration of thirty minutes to an hour. During this time the students would have to be logged into the system. However they are located all over the world, and enrolled at the university in England. This is also called distant learning via the Internet.

This scenario has the advantage of being completely independent of its location, everything is distributed (see Figure 7.1). And it is even conceivable to support as many as 2000 students logged in at any one time. Figure 7.2 shows that every student is connected to each of the different services, whereby each of them will not

notice where this service is located and not everybody needs to be connected to the same server.

Let us assume a virtual university offers a computer science degree as an e-learning course. The students would electronically enrol at this university. And access the course from wherever they are in the world. The virtual university may subcontract modules within the course from other well established departments or groups. Each of these modules may have a different quality and cost.

7.2 Requirements

The requirements for performance monitoring in the case study of e-learning are defined in the following.

Initial session setup

Before a student can subscribe to a service, the infrastructure needs to be checked to be able to guarantee the minimum quality of service required. In case these requirements are not fulfilled the student needs to be rejected from the beginning.

For example, when a student lives somewhere in a desert and s/he connects to the e-learning course with a 64Kbyte Modem, s/he can not expect a real-time voice connection to the computer at home.

Quality of delivery(transmission)

It is necessary to detect as soon as somebody logs in the system, to guarantee that this person will have the minimum transmission rate negotiated when s/he subscribed to this course. This requires monitoring so that the quality of the transmission will be guaranteed and does not dramatically decrease during connection time.

E. g. a student will not be satisfied when s/he can hear the lecturer but does not understand him or her because the quality of the voice transmission is not good enough.

Availability

Furthermore it needs to be guaranteed, that even when a service fails during runtime there will always be a replacement available somewhere.

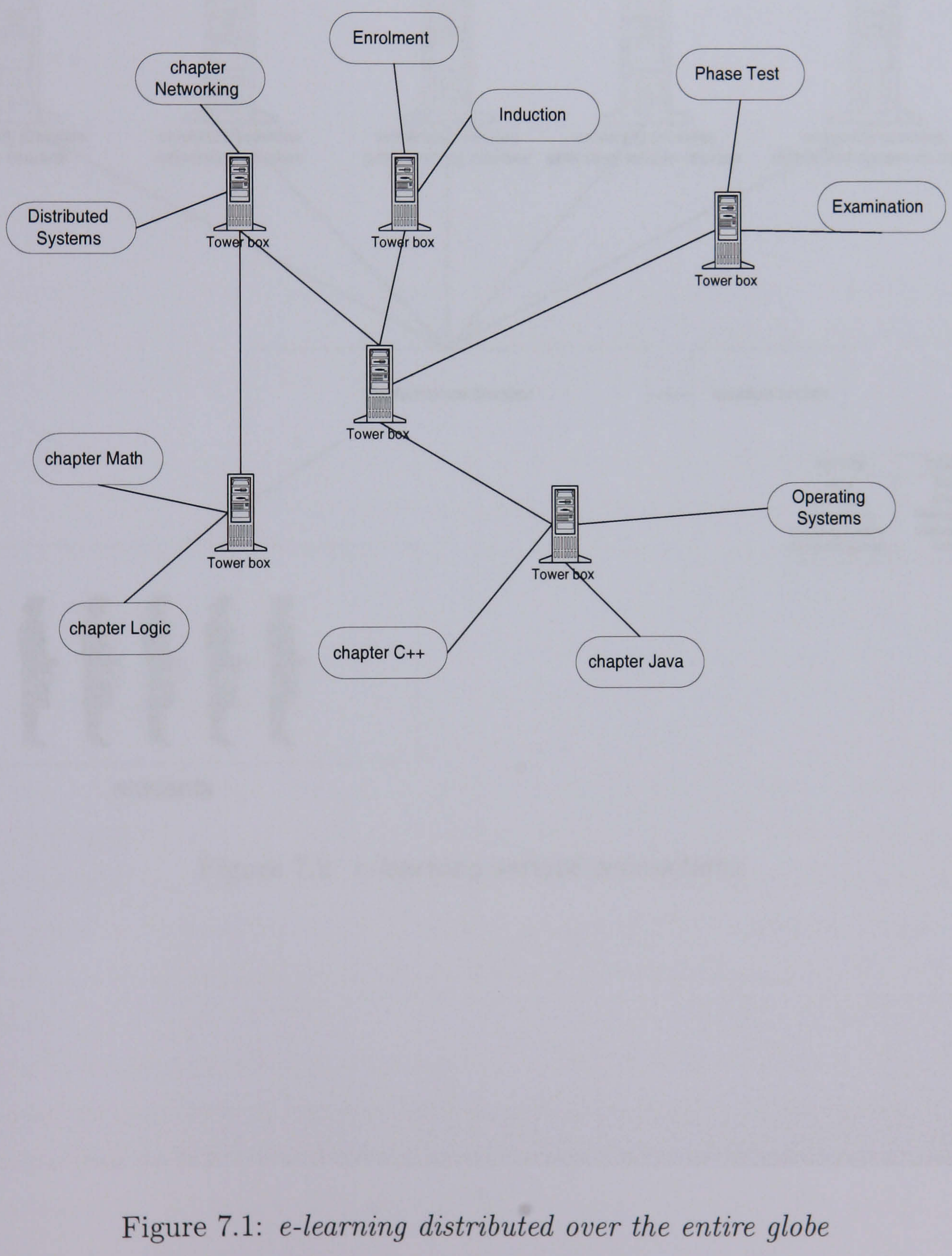


Figure 7.1: *e-learning distributed over the entire globe*

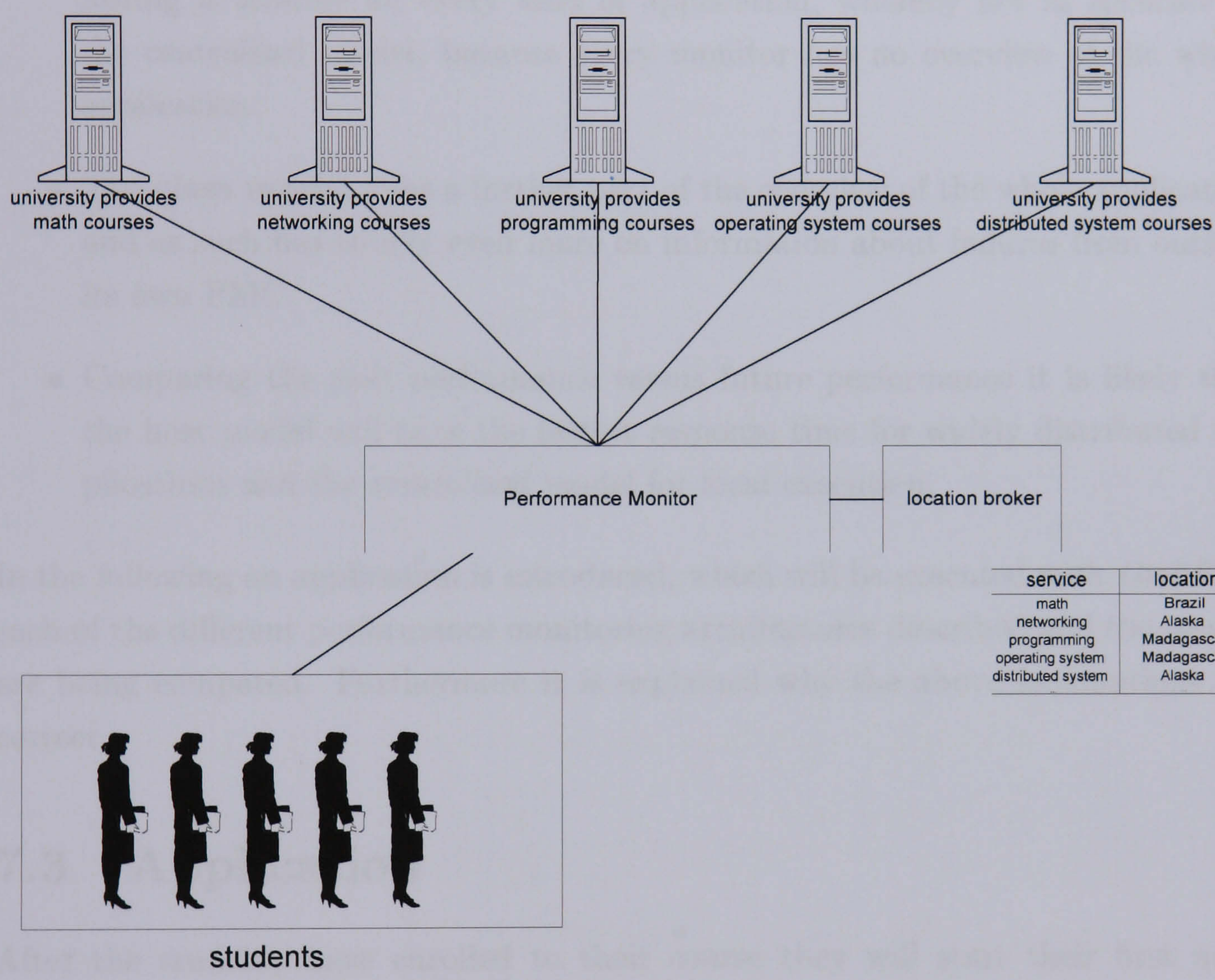


Figure 7.2: *e-learning service connections*

7.2.1 Hypothesis

Earlier it has been predicted that:

- The **centralised model** becomes a big bottleneck for physically widely distributed applications.
- The **host model** should not create any extra traffic and performance monitoring is feasible for every kind of application, whereby not as accurate as the centralised model, because every monitor has no overview of the whole application.
- The **class model** loses a further part of the overview of the whole application and as such has to rely even more on information about failures from outside its own PMC.
- Comparing the past performance versus future performance it is likely that the host model will have the fastest response time for widely distributed applications and the centralised model for local execution.

In the following an application is introduced, which will be executed with *GriPS* for each of the different performance monitoring architectures described and the results are being compared. Furthermore it is explained why the above assumptions are correct.

7.3 Application

After the students have enrolled to their course they will start their first unit. This unit begins with an induction course where the most basic knowledge of the course will be conveyed. Afterwards they will need some in depth knowledge about other subjects such as `math`, `programming languages`, `operating systems`, `networking` and so on. These subjects are available from different service providers.

Services under Grid Performance Software (*GriPS*) are in a tree structure. Figure 7.3 shows the structure of the tree for this e-learning service. This Figure uses the same structure as other applications before: every node in the tree represents one service instance. As explained before an object or in this particular case a service

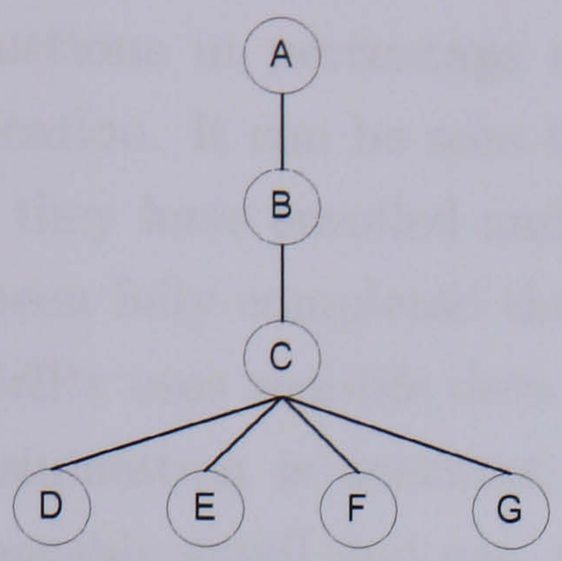


Figure 7.3: *The e-learning example application*

Machine	Services
ossi	0, 1, 2, 3, 4, 5, 6
forquet	0, 1, 2, 3, 4, 5, 6
garfield	3, 4
sienna	2, 3, 6

Table 7.1: Distribution of e-learning services

instance consist of data and allocated memory defined by one class and as such every student executing this application will create his/her own service instance. When one student has to restart the application one service instance terminates and for the next run a new service instance will be created.

The service defined by the service instance are defined in Table 7.2. This table also shows how many fake instructions each of these service instances are suppose to execute before calling their children, if there are any, whereby one fake instruction is defined in Listing B.1. Finally the table shows the distribution for this particular example, i.e. the machine on which each service instance is being executed. Additionally Table 7.1 shows what services are available on which machines. In case one of the machines fails an alternative location is taken from this table.

Service Instance	Service	Instructions	Machine
A	0 Enrolment	250	forquet
B	1 Start year one	350	forquet
C	2 Induction	450	forquet
D	3 Math	3500	garfield
E	4 Programming	5000	ossi
F	5 Operating systems	800	ossi
G	6 Networking	1000	sienna

Table 7.2: E-Learning Scheduling

GriPS produces a time diagram of the first year study as shown in Figure 7.4. This diagram shows for every service instance on the y axis the total amount of

instructions in percentage and on the x axis the time in seconds for the whole application. It can be seen that the students start with their induction course only after they have enrolled and the year has started. Only after the induction course has been fully completed they will start with the different subjects in parallel.

GriPs uses real-life data for the connection between different machines. Since this simulation is executed in a local area network, the real connection time is comparably small and can as such be neglected. GriPs reads the connection data out of a file and uses a replacement file to translate the information for the real machines used. The data used for the connection are described in Appendix B.3.1. Table 7.3 shows the real machine used in this example and the IP address and the location of the machine it is simulating. Figure 7.5 shows the delay in milliseconds added between the different machines. The delays in the diagram are rounded to the nearest millisecond, however GriPs uses the connection data up to microseconds. Some of the connections do not show a difference between the minimum delay and the maximum delay, this is when the difference only appears within microseconds.

real machine name	specification of the machine	Simulated IP address	Location of Simulated machine
forquet	HP model C3000 (PA-8500 CPU), 2GB memory, O/S: HP-UX 11.00	128.8.7.4	College Park, MD, US (University of Maryland)
ossi	Pentium 450Mhz, 256MB memory, Win NT4.0	216.168.227.250	Herndon, VA, US (Verisign)
garfield	Sunblade 150, UltraSparc IIi 650Mhz, 256Mb memory, Solaris 8 (108528-16)	203.181.248.27	Tokyo, Kanto, JP (APAN)
sienna	Sunblade 150, UltraSparc IIi 550Mhz, 128Mb memory, Solaris 8 (108528-16)	193.0.0.11	Amsterdam, North Holland, NL (RIPE)

Table 7.3: Connections used in this example

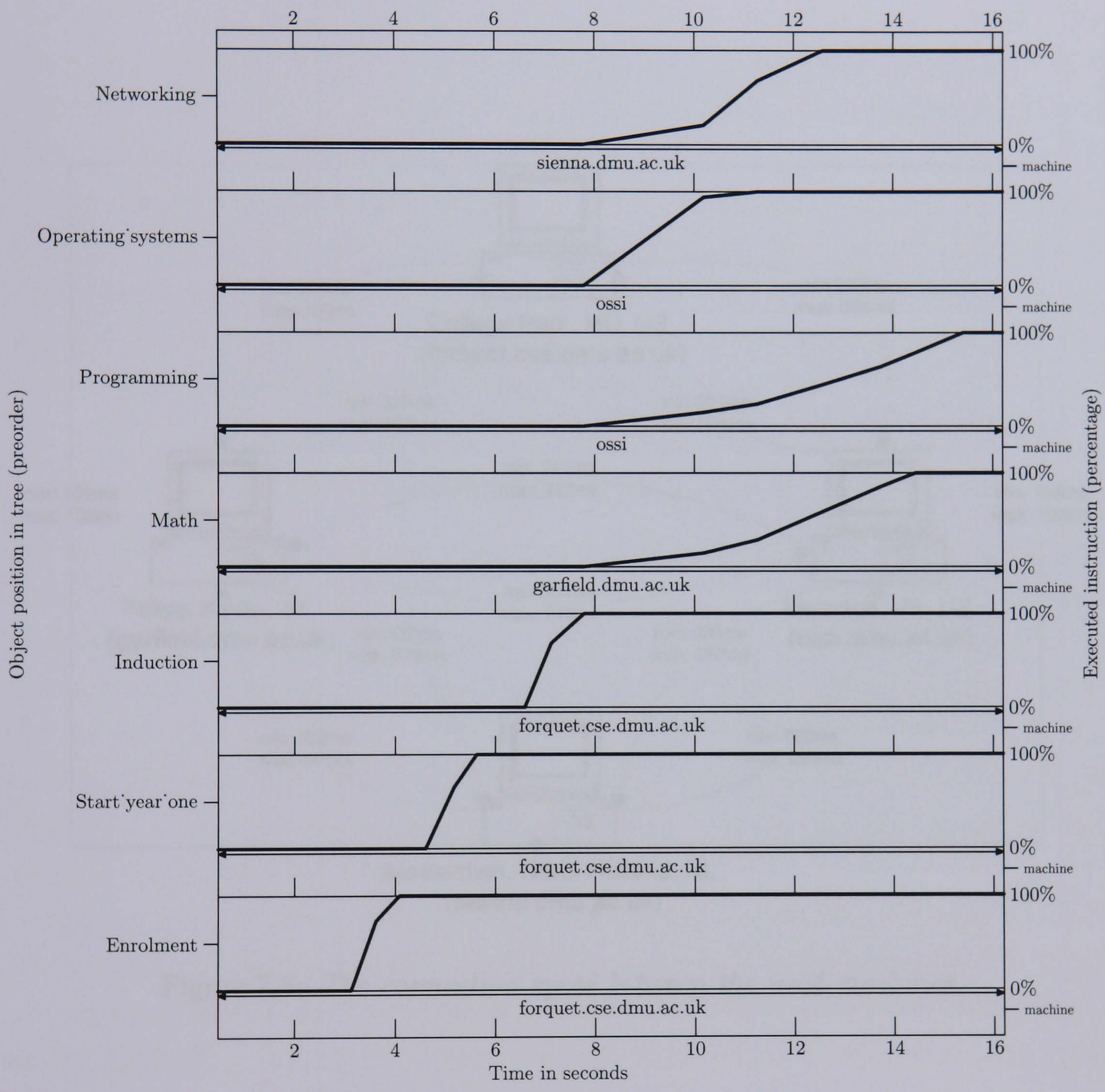


Figure 7.4: *First year study course*

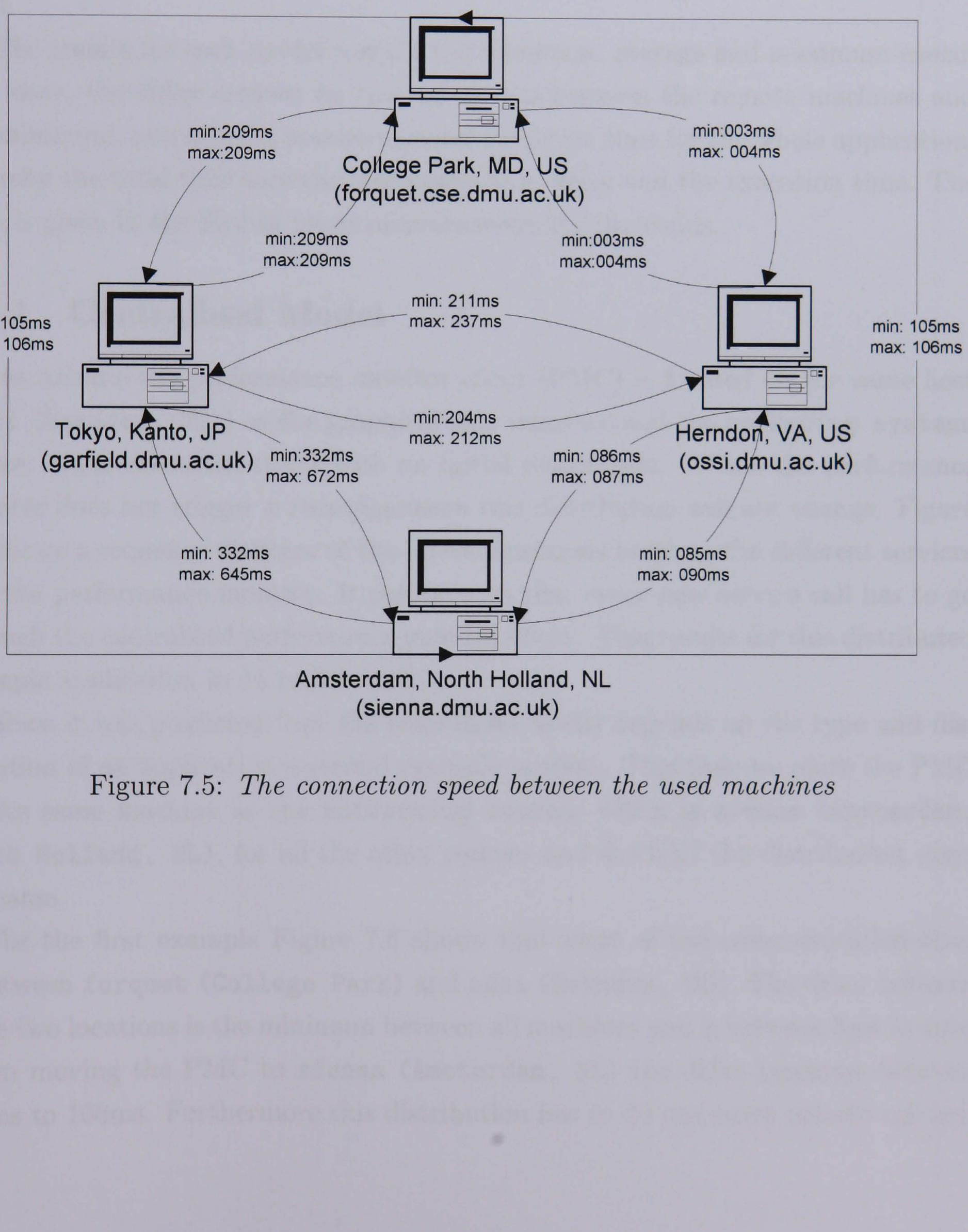


Figure 7.5: *The connection speed between the used machines*

7.4 Analysis

All results presented in this chapter are created by *GriPS* when executing this distribution 100 times. Because of the time involved running the application the execution time for each course is limited to a couple of seconds. In a real e-learning class surely every course would have to execute more instructions, however the behaviour would be the same. First the three models and their results are being presented and at the end the results of all three models are compared in Section 7.4.4.

The results for each model contain the minimum, average and maximum execution time, the delay created by the connection between the remote machines and the minimum, average and maximum total execution time for the whole application, whereby the total time includes, the connection delay and the execution time. The time is given in the format hours:minutes:seconds:miliseconds.

7.4.1 Centralised Model

Let us assume the performance monitor client (PMC) is located on the same host (`ossi (Herndon, US)`) as the graphical user interface and the `operating systems` course. Each execution starts with an initial distribution. When the performance monitor does not trigger a *reconfiguration* this distribution will not change. Figure 7.6 shows a sequence diagram of the interconnections between the different services and the performance monitor. It can be seen that every new service call has to go through the centralised performance monitor client. This results for this distributed example application in 11 remote calls.

Since it was predicted that the centralised model depends on the type and distribution of an application a second example is given. This time we place the PMC on the same machine as the `networking course`, which is `sienna (Amsterdam, North Holland, NL)`, for all the other courses and the GUI the distribution stays the same.

For the first example Figure 7.6 shows that most of the communication time is between `forquet (College Park)` and `ossi (Herndon, US)`. The delay between these two locations is the minimum between all machines and is between 3ms to 4ms. When moving the PMC to `sienna (Amsterdam, NL)` the delay increases between 105ms to 106ms. Furthermore this distribution has to do one extra remote call and

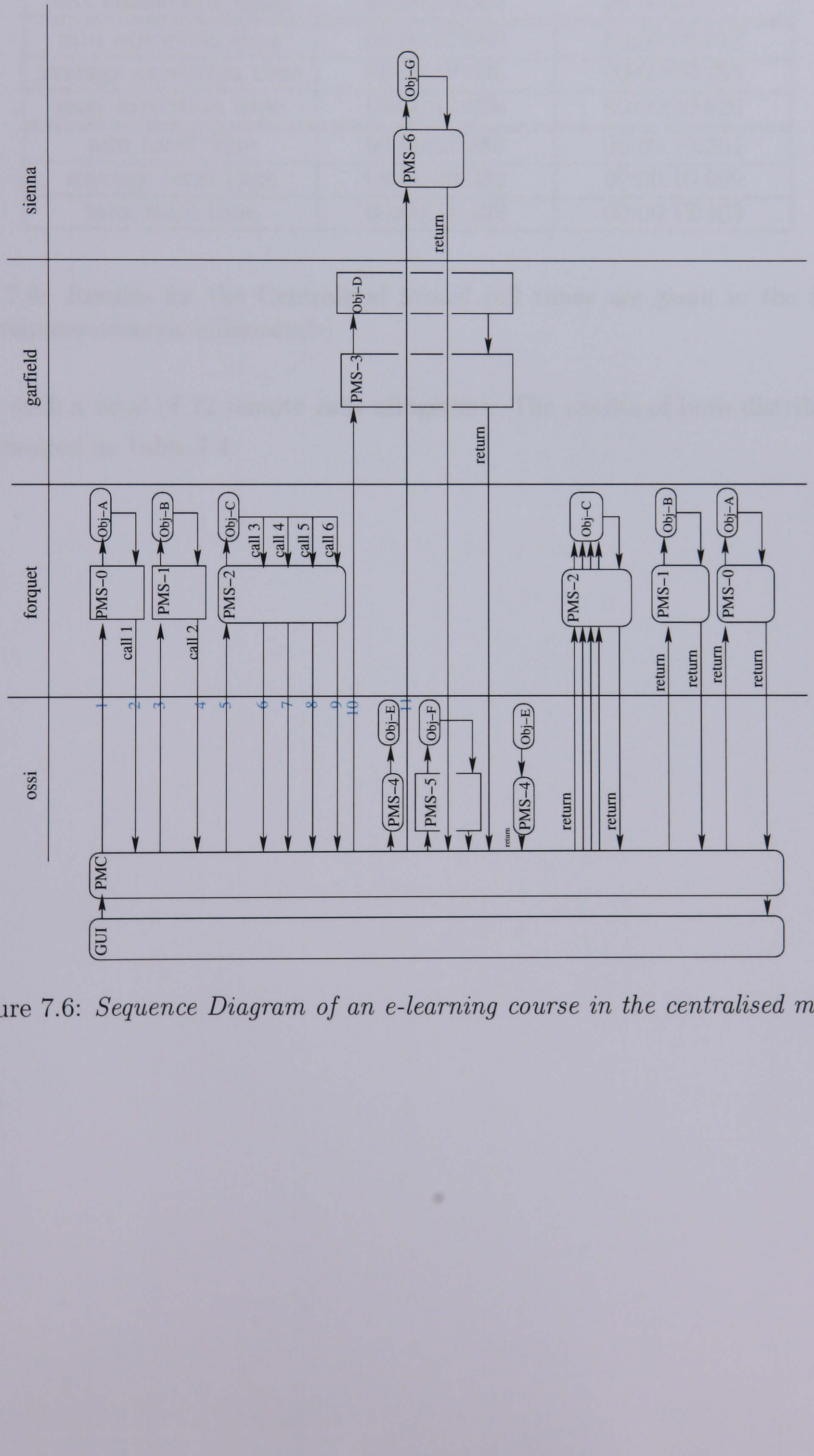


Figure 7.6: *Sequence Diagram of an e-learning course in the centralised model*

	Centralised Model PMC on ossi	Centralised Model PMC on sienna
amount remote calls	11	12
min connection delay (hh:mm:ss:MMM)	00:00:00:328	00:00:01:451
max connection delay	00:00:00:362	00:00:01:777
min execution time	00:00:07:060	00:00:08:813
average execution time	00:00:08:685	00:00:09:295
max execution time	00:00:11:076	00:00:10:626
min total time	00:00:07:388	00:00:10:264
average total time	00:00:09:032	00:00:10:909
max total time	00:00:11:438	00:00:12:403

Table 7.4: Results for the Centralised Model (all times are given in the format hours:minutes:seconds:miliseconds)

has as such a total of 12 remote calls altogether. The results of both distributions are presented in Table 7.4.

7.4.2 Host Model

GriPs executes the host model with exactly the same data and distribution only this time there exist one performance monitoring client for every *host* machine. A sequence diagram of the same application in this model can be seen in Figure 7.7. The host model saves remote calls to the centralised performance monitor and can connect directly to the machine of the next service, which results for this distribution in 4 remote calls all together. Table 7.5 shows the results for this model.

	Host Model
amount remote calls	4
min connection delay	00:00:00:321
max connection delay	00:00:00:323
min execution time	00:00:08:433
average execution time	00:00:09:053
max execution time	00:00:10:375
min total time	00:00:08:754
average total time	00:00:09:375
max total time	00:00:10:698

Table 7.5: Results for the Host Model (all times are given in the format hours:minutes:seconds:miliseconds)

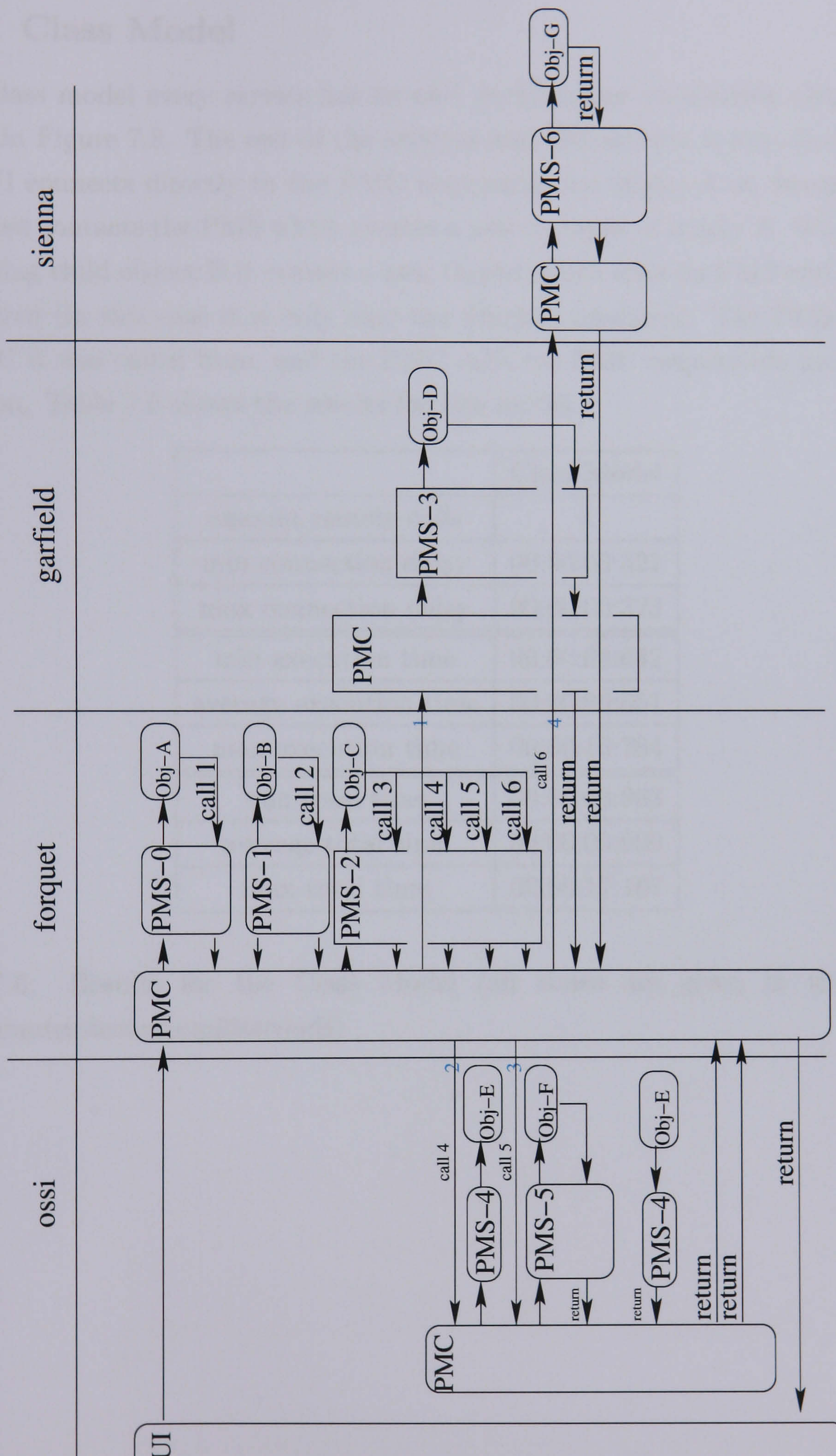


Figure 7.7: *Sequence Diagram of an e-learning course in the host model*

7.4.3 Class Model

In the class model every *service* has its own performance monitoring client as can be seen in Figure 7.8. The rest of the services and distribution is exactly the same. The GUI connects directly to the PMC responsible for object A on `forquet`. The PMC then contacts the PMS which creates a new instance of object A. When object A is calling child object B it creates a new thread which calls its PMS and waits till all children (in this case it is only one) has finished executing. The PMS contacts the PMC it was called from, and the PMC calls the PMC responsible for object B and so on. Table 7.6 shows the results for this model.

	Class Model
amount remote calls	4
min connection delay	00:00:00:321
max connection delay	00:00:00:323
min execution time	00:00:08:642
average execution time	00:00:09:651
max execution time	00:00:16:784
min total time	00:00:08:963
average total time	00:00:09:999
max total time	00:00:17:107

Table 7.6: Results for the Class Model (all times are given in the format hours:minutes:seconds:miliseconds)

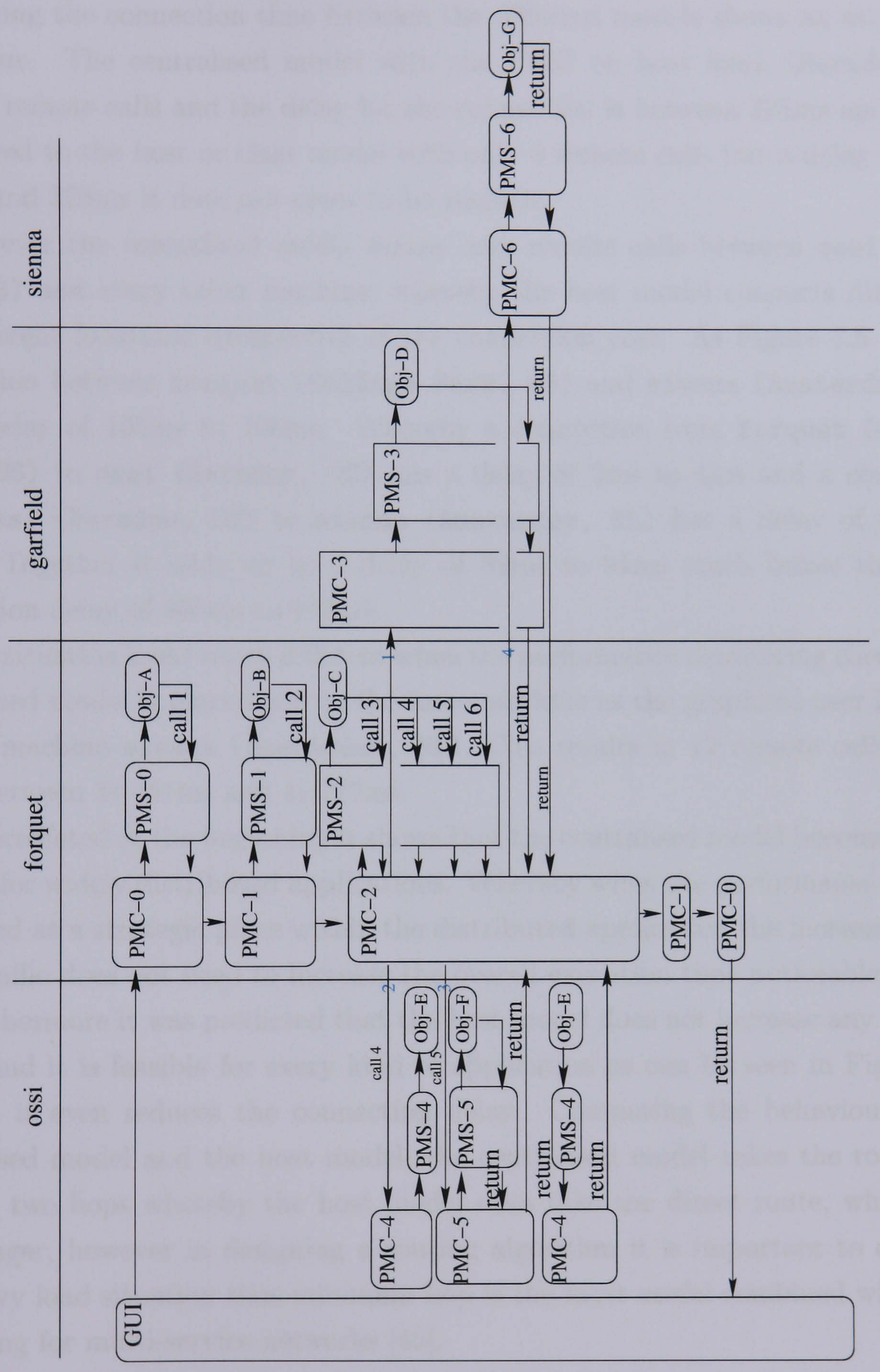


Figure 7.8: *Sequence Diagram of an e-learning course in the class model*

7.4.4 Comparison of the three models without a failure

Comparing the connection time between the different models shows an astonishing behaviour. The centralised model with the PMC on host `ossi (Herndon, US)` does 11 remote calls and the delay for the connection is between 328ms and 362ms. Compared to the host or class model with only 4 remote calls but a delay between 321ms and 323ms it does not seem to be realistic.

However the centralised model makes only remote calls between `ossi (Herndon, US)` and every other machine, whereby the host model connects directly to the different locations irrespective of the connection cost. As Figure 7.5 shows a connection between `forquet (College Park, US)` and `sienna (Amsterdam, NL)` has a delay of 105ms to 106ms. Whereby a connection from `forquet (College Park, US)` to `ossi (Herndon, US)` has a delay of 3ms to 4ms and a connection from `ossi (Herndon, US)` to `sienna (Amsterdam, NL)` has a delay of 86ms to 87ms. Together it adds up to a delay of 89ms to 91ms much below the direct connection delay of 105ms to 106ms.

The situation looks much different when the performance monitoring client in the centralised model is placed not on the same machine as the graphical user interface but on machine `sienna (Amsterdam, NL)`. This results in 12 remote calls with a delay between 1s 451ms and 1s 777ms.

As predicted at the beginning it shows that the centralised model becomes a bottleneck for widely distributed applications. Whereby when the performance monitor is located at a strategic place within the distributed application the increase in network traffic does not need to increase the overall execution time noticeable.

Furthermore it was predicted that the host model does not increase any network traffic and it is feasible for every kind of application as can be seen in Figure 7.9, because it even reduces the connection delay. Comparing the behaviour of the centralised model and the host model, the centralised model takes the route over at least two hops whereby the host model does take the direct route, which does take longer, however in designing a routing algorithm it is important to cater for the heavy load situation thus minimum hop is the most useful combined with Load Balancing for multi-service networks [40].

In case the application runs through without any failure and as such no reconfiguration is needed all three models behave at their optimum. In what follows fault treatments are dealt with.

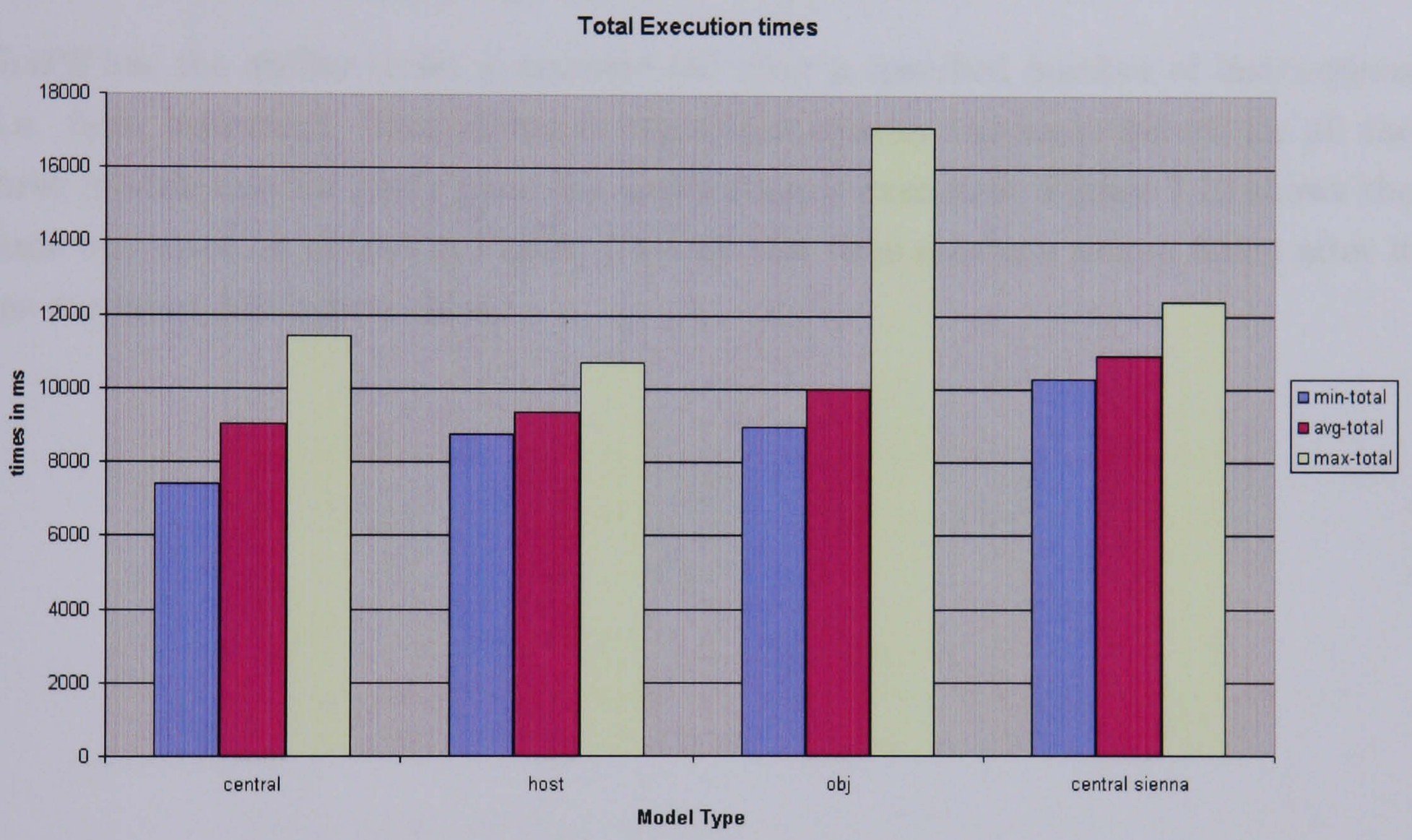


Figure 7.9: *Chart about the total execution time*

7.5 Failure recovery

Three types of failure recovery are considered. First we explain the failure recovery after a time-out. This is clearly not the best solution for a time critical application like this e-learning example, however even this example demonstrates the behaviour clearly. The following subsection describes the early warning recovery of a performance failure. The next subsection then describes a failure recovery for a time-critical application.

7.5.1 Failure recovery after a time-out

GriPS has the ability to let a machine fail after a specified number of instructions (i.e. fault injection). This allows to reproduce exactly the same failure for all the three models and for every time the application is executed. Figure 7.10 shows the same execution as shown in Figure 7.4 only this time machine sienna failed after it has executed 500 instructions.

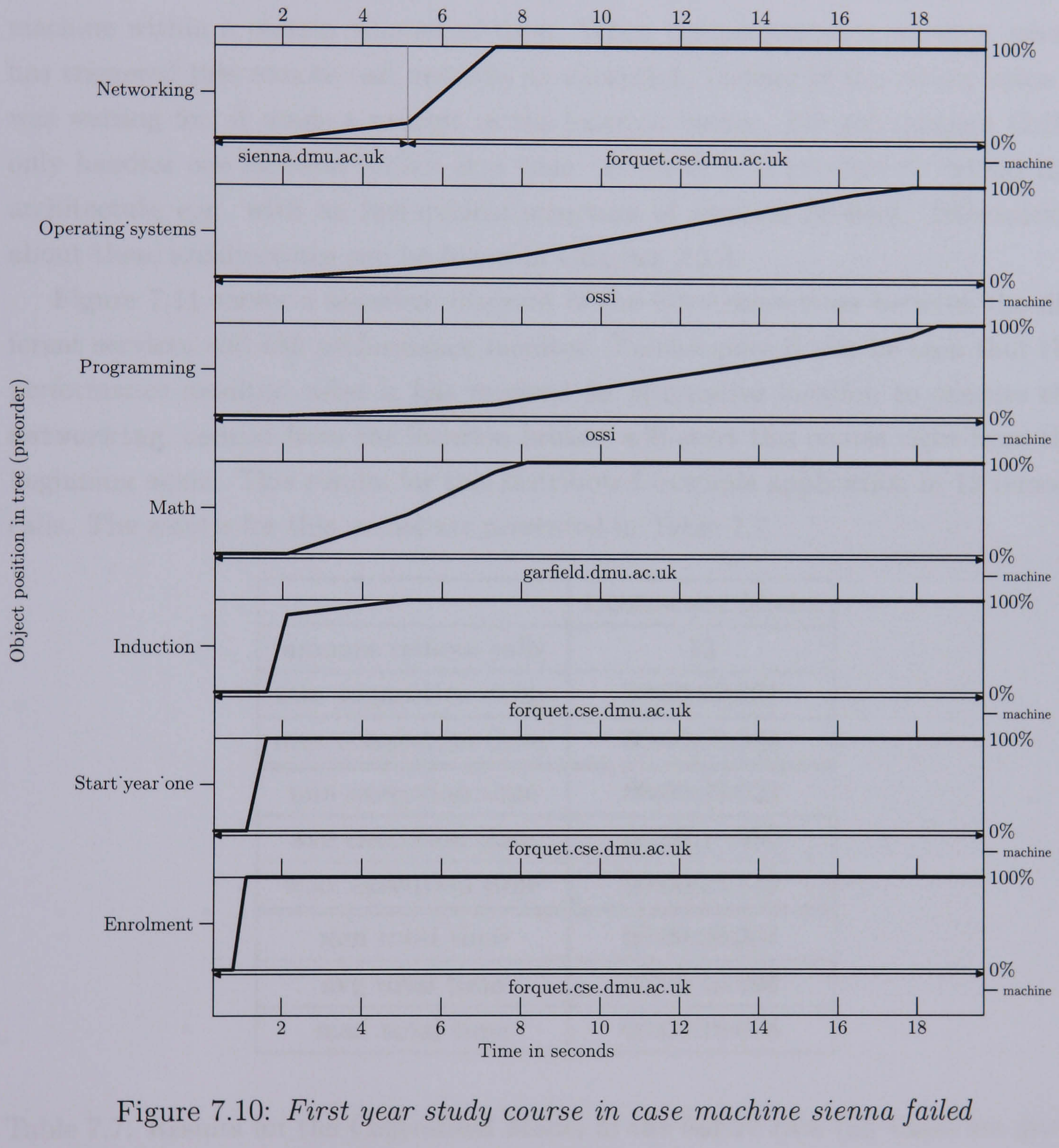


Figure 7.10: *First year study course in case machine sienna failed*

Failure detection and recovery in the centralised model In this example the PMC is located on `ossi (Herndon, US)` as this was the best solution for the centralised model. As soon as the performance monitor detects that machine `sienna` has failed it triggers a reconfiguration. In this case the failure is detected because Java RMI throws an exception as soon as it is not able to connect to the remote machine within a certain amount of time. When the performance monitor, which has triggered this remote call, receives an exception, instead of the return value it was waiting for, it sends a request to the location broker. For the moment GriPs only handles one location broker at a time. However it is possible to extend this architecture e.g. with an hierarchical structure of location brokers. Information about these architectures can be found in Chapter 2.3.3.

Figure 7.11 shows a sequence diagram of the interconnections between the different services and the performance monitor. Furthermore it can be seen that the performance monitor, after it has received an alternative location to execute the `networking course` from the location broker, will start this course right from the beginning again. This results for this distributed example application in 12 remote calls. The results for this model are presented in Table 7.7.

	Centralised Model
amount remote calls	12
min connection delay	00:00:00:331
max connection delay	00:00:00:366
min execution time	00:00:09:023
avg execution time	00:00:10:560
max execution time	00:00:15:712
min total time	00:00:09:354
avg total time	00:00:10:908
max total time	00:00:16:078

Table 7.7: Results for the Centralised Model in the failure case (all times are given in the format hours:minutes:seconds:miliseconds)

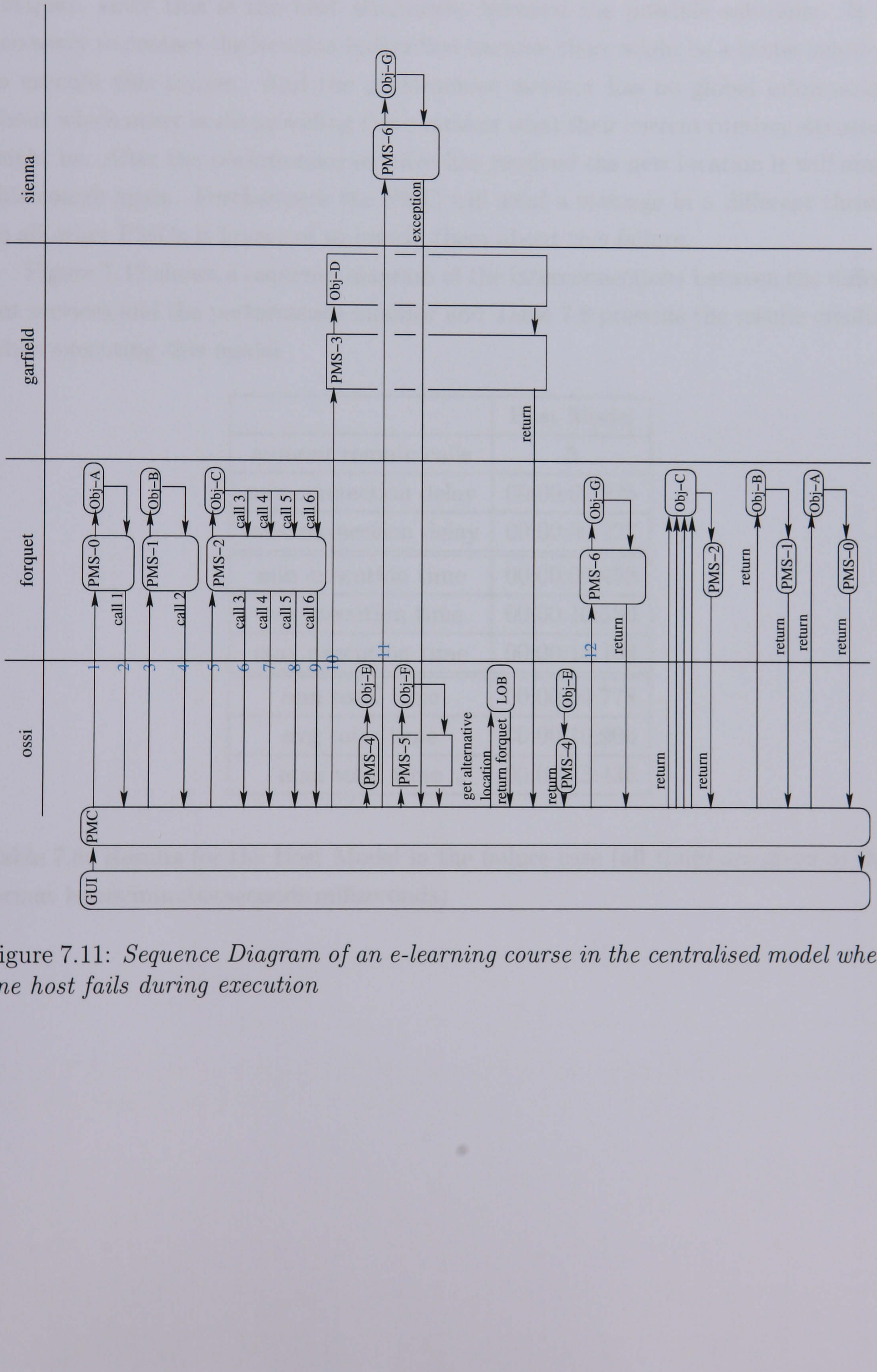


Figure 7.11: *Sequence Diagram of an e-learning course in the centralised model when one host fails during execution*

Failure detection and recovery in the host model In case of the host model the situation is similar. The performance monitor on host `forquet` which has called the failed machine will detect this failure and sends a request to the location broker which is situated on machine `ossi` in this example. The location broker will return `forquet`, since this is the best alternative between the possible solutions. It is necessary to contact the location broker first because there might be a better solution to execute this course. And the performance monitor has no global information about which other hosts providing this course or what their current runtime situation might be. After the performance monitor has received the new location it will start this course again. Furthermore the PMC will send a message in a different thread to all other PMCs it knows of to inform them about this failure.

Figure 7.12 shows a sequence diagram of the interconnections between the different services and the performance monitor and Table 7.8 presents the results created when executing this model.

	Host Model
amount remote calls	5
min connection delay	00:00:00:325
max connection delay	00:00:00:327
min execution time	00:00:09:453
avg execution time	00:00:10:580
max execution time	00:00:13:109
min total time	00:00:09:778
avg total time	00:00:10:906
max total time	00:00:13:436

Table 7.8: Results for the Host Model in the failure case (all times are given in the format hours:minutes:seconds:miliseconds)

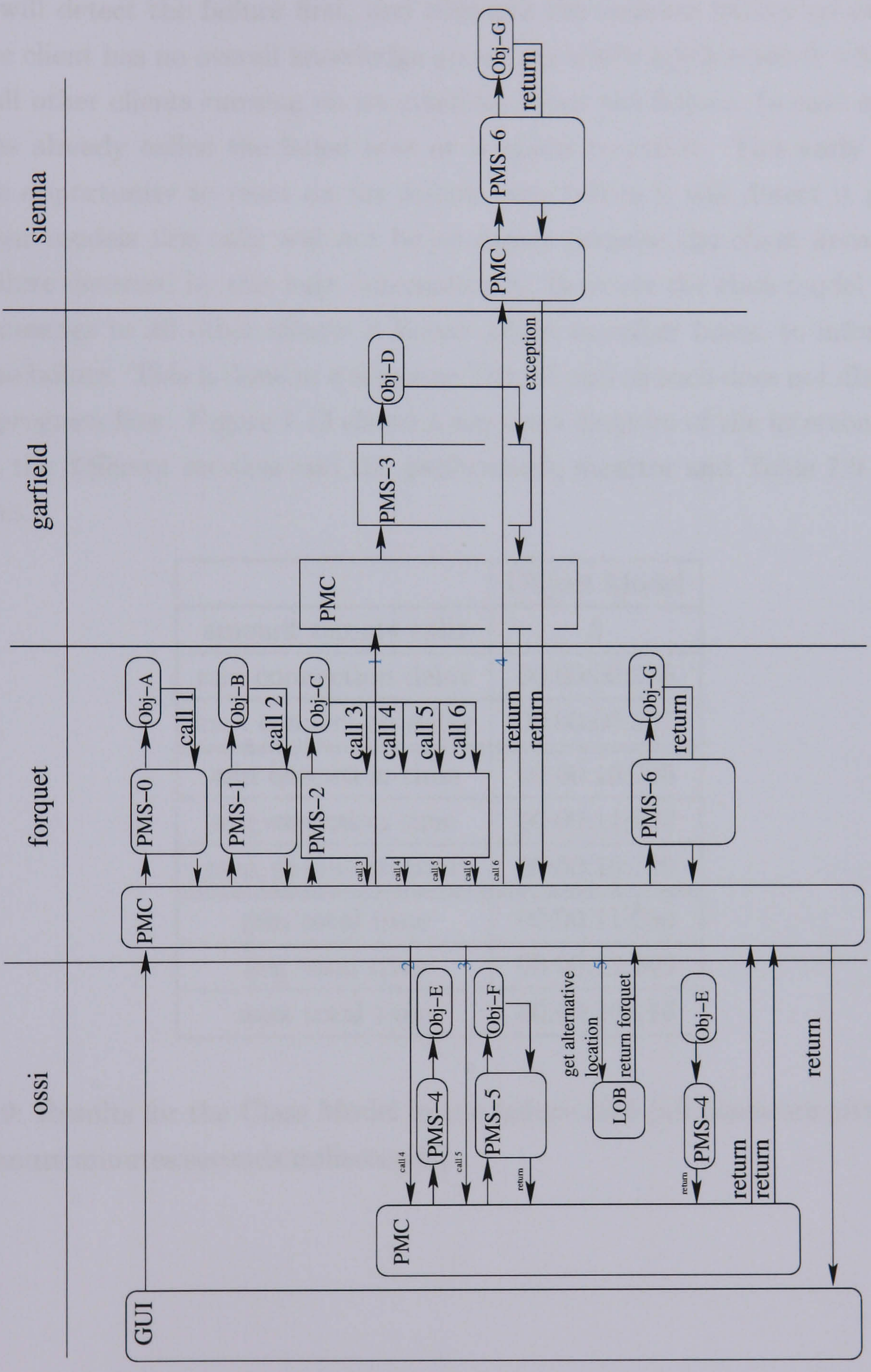


Figure 7.12: *Sequence Diagram of an e-learning course in the host model when one host fails during execution*

Failure detection and recovery in the class model In the class model the PMC-2 will detect the failure first, and contacts the location broker on `ossi`. Because the client has no overall knowledge about the whole application it will have to inform all other clients running on its machine about the failure. In case any other client has already called the failed host or is about to call it. This early message gives the opportunity to react on the failure even before it will detect it itself. In both other models this calls will not be necessary because the client knows about every failure detected by this host automatically. However the class model will also send a message to all other clients it knows about on other hosts, to inform them about the failure. This is done in a separate Thread and as such does not disturb the normal program flow. Figure 7.13 shows a sequence diagram of the interconnections between the different services and the performance monitor and Table 7.9 presents its results.

	Object Model
amount remote calls	5
min connection delay	00:00:00:325
max connection delay	00:00:00:327
min execution time	00:00:10:725
avg execution time	00:00:11:923
max execution time	00:00:19:789
min total time	00:00:11:050
avg total time	00:00:12:249
max total time	00:00:20:116

Table 7.9: Results for the Class Model in the failure case (all times are given in the format hours:minutes:seconds:miliseconds)

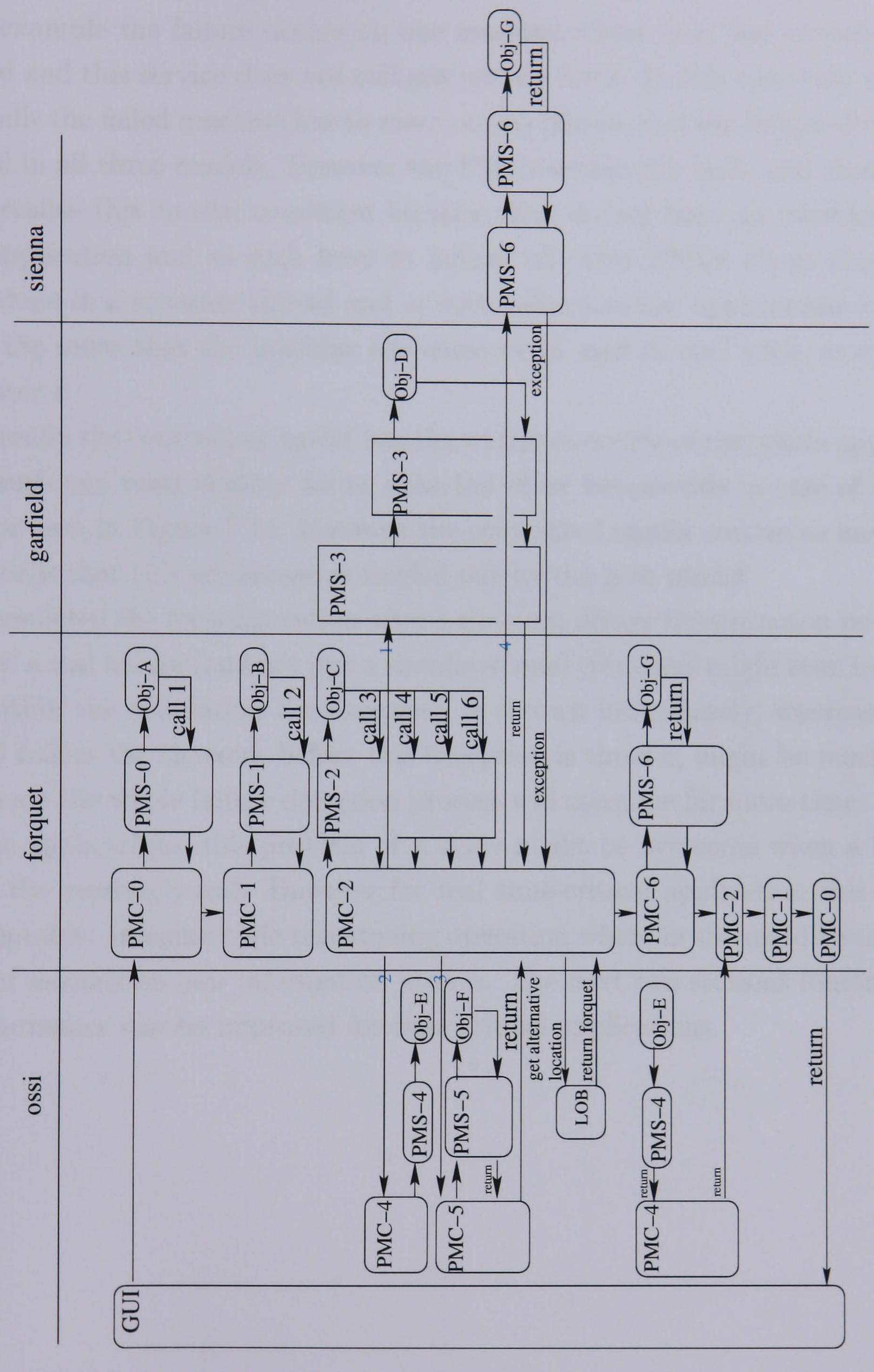


Figure 7.13: *Sequence Diagram of an e-learning course in the class model when one host fails during execution*

7.5.1.1 Comparison of the three models in case of a machine failure

In this example the failure occurs on one machine where only one service is being executed and this service does not call any service itself. In this case only the PMC which calls the failed machine has to react on the failure, and the failure detection is identical in all three models. However the PMCs within the host- and class- model do not realise this special condition because they do not have an overview of the whole application and as such have to inform all other PMCs about this failure. This is done in a separate thread and as such influences the applications execution only in the sense that the machine has some extra load to deal with, as explained in Chapter 4.

Generally the centralised model has the widest overview of the whole application and as such can react slightly faster than the other two models in case of a failure as can be seen in Figure 7.14. However the centralised model creates so much more remote calls that this advantage is singled out by the host model.

As predicted the reconfiguration after a time-out delays the execution noticeably. In case of a real failure (and not just a simulated one) this delay might even be higher. Since within the simulation the exception is thrown immediately, whereas in case of a real failure the timeout, before this exception is thrown, might be much higher and as such the whole failure detection process will consume far more time. For soft real-time applications, this problem of a delay might be overcome when a buffer is used at the receiver's end. However for real time-critical application this delay is not acceptable. Imagine a life threatening operation where in the middle suddenly a couple of seconds no new information arrives. The next two sections illustrate how the performance can be improved for time critical-applications.

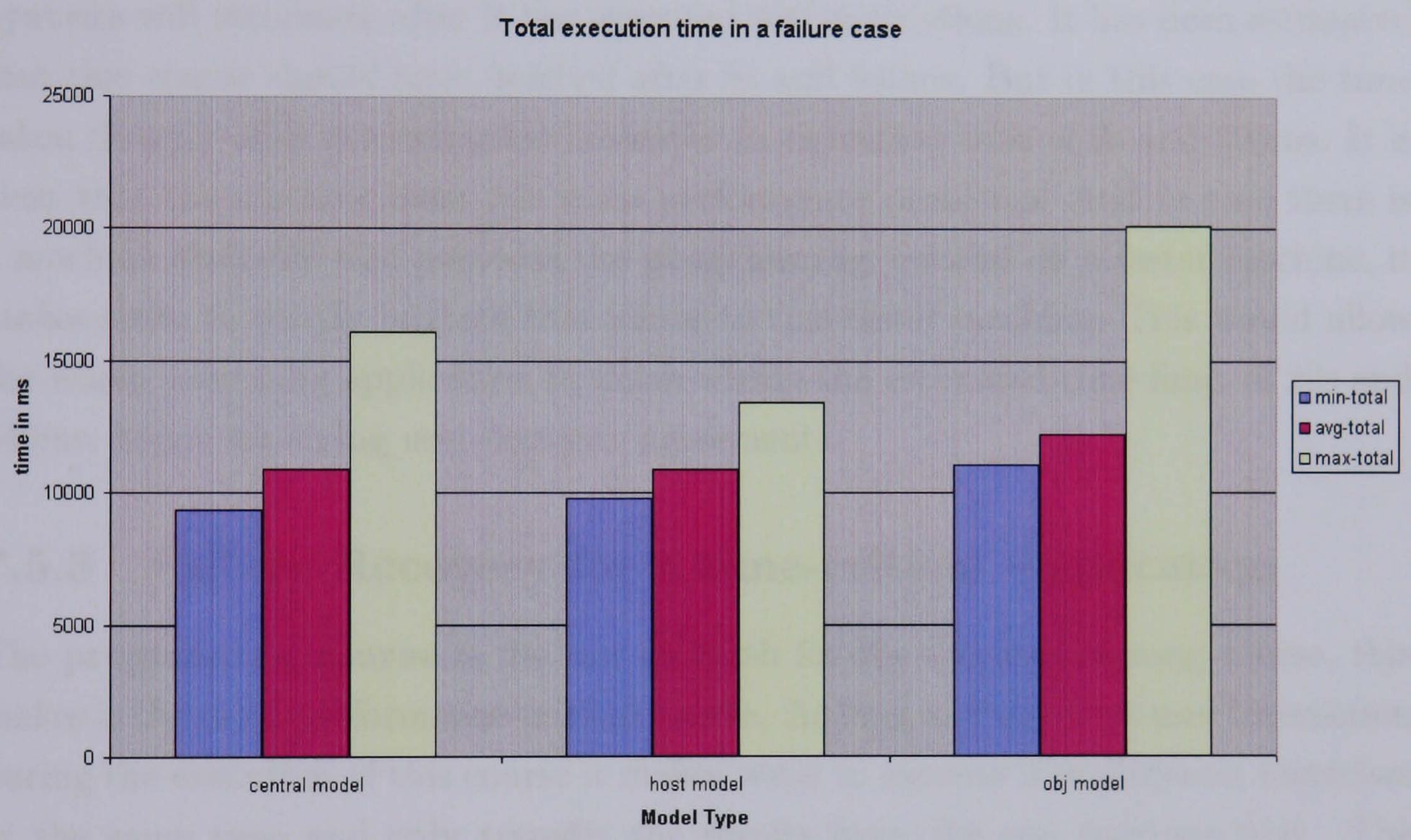


Figure 7.14: *Chart about the total execution time in the failure case*

7.5.2 Early Warning Recovery of a Performance Failure

It is conceivable that one of the machines involved in the execution might run low on memory or is simply overloaded, which causes a delay in the execution of every object on this machine. However the situation might already improve when one or more objects will be migrated to another machine. To detect this kind of situation the execution time of every object is estimated before the execution starts (Chapter 4).

Let us assume that an other user start running some applications on machine `ossi`. This results in the performance breaking down. The course on `operating systems` will terminate after it has executed 800 instructions. It has been estimated that this course should have finished after 8s and 840ms. But in this case the time taken directly after the execution indicates an execution time of 9s and 200ms. It is clear that the machine `ossi` has some performance problems. And in case there is a machine available who provides the `programming course` on a faster machine, it makes sense to simply migrate this course to this faster machine. This would allow the whole e-learning application to finish within the estimated time limit of 22s and 840ms, hence satisfying user-contract agreement.

7.5.3 Failure Recovery for a time-critical Application

The `programming course` is the last to finish for the whole e-learning course, this makes it the most performance-critical course. As long as there is no user interaction during the execution of this course it makes sense to execute it on different machines at the same time and only transfer the results from the one finishing first. The execution of all other replications can be stopped as soon as one finishes successfully. However when choosing the location for the replication it is to bear in mind that every additional execution on a particular host will slow down the execution of every other object running on that host. For the moment this feature is not implemented within *GriPS*.

7.6 Summary

This chapter has introduced a case study on e-learning. One example application has been presented which was executed 100 times for each model on *GriPS*. The

results created during this execution where compared. Like predicted beforehand the centralised model becomes easily a bottleneck for widely distributed applications. Depending on the application and its distribution the centralised model has also the ability to create some usable results, since it has the overview of the whole application. However in general the host model is more likely to create reasonable results independently of the distribution. It keeps a fairly good overview and has the advantage of a direct connection to the next object.

Chapter 8

Conclusions and further research and development

8.1 Vision

A traditional application need to be bought, installed, updated and maintained by the user. On the other hand service-based computing is a new computing paradigm where an application is not sold, but made available for remote usage. It can be executed from all over the world on the machines where it is installed. This thesis has first introduced future distributed environments, where this research is built on. It has been explained how the object oriented programming paradigm helps to build service-based distributed applications. The advantages and disadvantages of local and remote execution have been discussed, with the result that this research concentrates on remote execution. When using objects distributed over a large network it is inevitable to search for these objects before they are executed. Various solutions of how to resolve this **searching** have been discussed, and mechanisms to support the search either with the help of the class names or its functionality are given. However, to allow any successful search in a large distributed system several architectures for location brokers have to be introduced. To have only one centralised location broker would result fairly quickly into a bottleneck, which has to be avoided. This bottleneck can be avoided by using a hierarchical structure of location brokers. After an object has been found it is discussed at what time the different object addresses are bound together. It is possible to bind the addresses either at invocation time or at runtime. The solution used in this research is a combination of both

options. First an initial static distribution is created at invocation time just before the start of the execution, and in any failure case a dynamic reconfiguration and as such a new process of NAB (Network Address Binding) will be initiated during execution. Furthermore to built a service-based application a user, a service provider and the developer will each have a different interest in such an application. These interests have been discussed and resulted in the fact, that before the execution these requirements have to be specified in a contract between the user and the service provider. Additionally a service-based application needs some further development in the area of security, accounting, naming, location management, network and fault configuration as well as performance monitoring. Each of these have been introduced whereby the focus of this research lies on performance monitoring.

8.2 Achievement

8.2.1 Performance Monitoring Client and Server

The performance monitor introduced in this thesis is divided into two parts. The Performance Monitoring Client (PMC) implements the monitoring activity. In contrast the Performance Monitoring Server (PMS) plays a more passive role and responds to requests and instructions issued by the PMC. In all the introduced architectures the PMS is always executed on the same machine where the objects are executed. Indeed every PMS is responsible for only one class and instantiates every object out of this class, whereby different architectures for the location of the PMC are presented. In the centralised model there exist only one PMC for each application, whereby the other two models have even a distributed architecture for the PMC. In one model there exist one PMC on every host used and in the last model there exist for every PMS exactly one PMC. For each of these models the advantages and disadvantages have been discussed. Several algorithms to monitor the availability, reliability, number of remote calls and the network delay created by these remote calls are presented. Whereby one of these algorithms relies on an estimation of the execution time before the actual start of the execution.

8.2.2 Algorithm for Time Estimation

The estimation process of times has been evolved step by step from a sequential towards a parallel executing application. When executing several tasks on the same machine this machine will slow down and as such it is not advisable to use always the fastest possible machine for an object. A machine, which is slower before the execution starts, might be faster during the execution. It is not possible to predict the *exact* run-time behaviour of a machine because a machine might also be used for other application - parts. However this estimation provides the performance monitor with enough information to notice shortfalls and react to compensate.

8.2.3 Algorithm for Optimisation of Execution Time

The static optimisation to schedule an application has been described. This scheduling represented by a DAG is proved to be a NP hard problem [26]. As such the *optimisation* of the object distribution presented in this thesis will not necessarily find the absolute optimal distribution, however the results created by this algorithm are still better than choosing the distribution randomly.

8.2.4 Grid tool for Performance Monitoring

The Grid Performance Software developed to simulate a global distributed application takes as an input an application represented in a tree structure. It can be freely chosen where in the network each part of the simulated application should be executed. Furthermore a simulated machine failure can be injected on one or more of the used machines after it has executed a certain number of instructions. When these instructions have been executed the HostMachine throws a Remote Exception in the same way as the Exception would be thrown when the machine fails in reality. Moreover it was explained how the reconfiguration from such a failure or indeed from a performance point of view has been dealt with.

8.2.5 Evaluation

A case study about e-learning is given as an example. The same e-learning application has been executed 100 times for each model and a comparison of the three different performance monitoring architectures is given. Like predicted the centralised model becomes a bottleneck for widely distributed applications. However,

depending on the application and its distribution even the centralised model produces some acceptable results, since it keeps the overview of the whole application. However the host model can be used more in general since it still is able to keep a fairly good overview and has the advantage of a direct connection to the next object.

8.3 Further Work

There are various aspects in which the work can be extended.

Location Management

As for the location management it has been explained why a centralised management is not appropriate for a large global system. Some ideas have been presented how a hierarchical model can be implemented. However these ideas are clearly not at a stage to be put into practice. Who will be responsible for the naming of a new object? In case the naming is be done by its functionality clearly different people might come up with a different functionality for the same object. Will it be possible for the developer to decide every possible way this object might be used in its lifetime?

Dynamic Reconfiguration and Failure

Furthermore the reconfiguration algorithm can be extended to include failure because of malfunctioning hard- or software. The detection of such a failure is a challenging problem, imagine an object which returns a call even though that it has not terminated its execution, and as such these results are not valid. Or even a machine, which keeps sending not valid messages over the network, and as such congest the network unnecessary.

Response Time

The response time for the performance monitoring can be improved by injecting breakpoints into the code [21] and using persistent storage for partial results.

A mechanism to introduce the measurement of memory load on machines and the speed drop of heavily used connections. Where the usage of connection resources can be realised in the same way we presented for the machines resources.

Billing

How can this "pay as you go" computing be implemented and an appropriate contract between the service provider and the user being accomplished?

Security

Last but not least security will play a major role for any service-based computation. The system has to be protected against attacks like eavesdropping, masquerading, tampering, denial of service, Spam and viruses.

Appendix A

Data structures

E	Set of executing nodes
W	Set of waiting nodes
m	machine
$m.E$	Set of execution nodes on machine m
$m.W$	Set of waiting nodes on machine m
$m.\alpha$	parameter specifying the speed drop
$m.\beta$	parameter specifying the speed drop
m.speed	the maximum speed of the machine
m.services	the set of services that are available on the machine
$ams(m)$	the actual machine speed of m
(n_i, n_j)	message (edge) from node n_i to node n_j
$c(n_i, n_j)$	costs (in ms) of message (n_i, n_j)
(n_i, n_j).time	already transmitted ms
T	Set of currently transmitting messages
n_i	a node in the DAG
$n_i.o$	The related object of the node
$n_i.i$	The total instructions that have to be executed by this node
$n_i.ei$	The instructions that already have been executed
n_i.startTime	The start time of this node in ms
n_i.endTime	The end time of this node in ms
n.state	The state of this node (INITIALISED, EXECUTING, WAITING, TERMINATED)

(N, s, e, w)	a collision tuple. N is the colliding node. s is the start time. e is the end time. w stores if N was waiting or executing.
$n.EC$	Set of collisions that occured while n was executing
$n.WC$	Set of collisions that occured while n was waiting

A.1 Data tables for the time estimation in the DAG

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

Initial values: alpha = 1, beta = 0 for I and II
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
Sets:
  E    = {n1}
  T    = {}
  I.E  = {n1}
  II.E = {}


E Nodes:
  n1:   start= 0ms, end= 0ms, ei= 0i

W Nodes:
T Nodes:
calls:

**************************************************
Iteration 0:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  first = n1 in 200ms
  exec(n1):
```

```
Sets:
  E     = {}
  T     = {(1,2), (1,3)}
  I.E   = {}
  II.E = {}

E Nodes:
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei = 200i
calls:
  (1,2):  c= 10ms, time= 0ms
  (1,3):  c= 0ms, time= 0ms

**************************************************
Iteration 1:
  ams(I)   = 1 i/ms
  ams(II) = 2 i/ms
  first = (1,3) in 0ms
  transmit((1,3)):
Sets:
  E     = {n3}
  T     = {(1,2)}
  I.E   = {n3}
  II.E = {}

E Nodes:
  n3: start = 200ms, end = 200ms, ei = 0i
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei = 200i
calls:
  (1,2):  c= 10ms, time= 0ms

**************************************************
Iteration 2:
```

```
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  first = (1,2) in 10ms
  transmit((1,2)):
Sets:
  E    = {n2, n3}
  T    = {}
  I.E  = {n3}
  II.E = {n2}

E Nodes:
  n3: start= 200ms, end= 210ms, ei= 10i
  n2: start= 210ms, end= 210ms, ei= 0i
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei = 200i
calls:

**********************************************************
Iteration 3:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n3 (ri= 90i) (rt= 90ms)
  n2 (ri= 450i) (rt= 225ms)

  first = n3 in 90ms
  execute(n3):
Sets:
  E    = {n2}
  T    = {(3,5), (3,6)}
  I.E  = {}
  II.E = {n2}

E Nodes:
  n2: start= 210ms, end= 300ms, ei= 180i
W Nodes:
T Nodes:
```

```
  n1: start = 0ms, end = 200ms, ei = 200i
  n3: start= 200ms, end= 300ms, ei= 100i
calls:
  (3,5): c= 0ms t= 0ms
  (3,6): c= 10ms t= 0ms

**************************************************
Iteration 4:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n2 (ri= 450i) (rt= 225ms)
  (3,5) (rt= 0ms)
  (3,6) (rt= 10ms)

  first = (3,5) in 0ms
  transmit((3,5)):
Sets:
  E    = {n2,n5}
  T    = {(3,6)}
  I.E  = {n5}
  II.E = {n2}

E Nodes:
  n2: start= 210ms, end= 300ms, ei= 180i
  n5: start= 300ms, end= 300ms, ei= 0i
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei = 200i
  n3: start= 200ms, end= 300ms, ei= 100i
calls:
  (3,6): c= 10ms t= 0ms

**************************************************
Iteration 5:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n2 (ri= 450i) (rt= 225ms)
```

```
  n5 (ri= 300i) (rt= 300ms)
  (3,6) (rt= 10ms)

  first = (3,6) in 10ms
  transmit((3,6)):
Sets:
  E    = {n2,n5,n6}
  T    = {}
  I.E  = {n5}
  II.E = {n2,n6}

E Nodes:
  n2: start= 210ms, end= 310ms, ei= 200i
  n5: start= 300ms, end= 310ms, ei= 10i
  n6: start= 310ms, end= 310ms, ei= 0i
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei = 200i
  n3: start= 200ms, end= 300ms, ei= 100i
calls:

**************************************************
Iteration 6:
  ams(I)  = 1 i/ms
  ams(II) = 1 i/ms
  n2 (ri= 250i) (rt = 250ms)
  n5 (ri= 290i) (rt = 290ms)
  n6 (ri= 500i) (rt = 500ms)

  first = n2
  execute(n2):
Sets:
  E    = {n5,n6}
  T    = {(2,4)}
  I.E  = {n5}
  II.E = {n6}
```

```
E Nodes:
  n5: start= 300ms, end= 560ms, ei= 260i
  n6: start= 310ms, end= 560ms, ei= 250i
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
calls:
  (2,4) c=0, t=0

**************************************************
Iteration 7:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n5 (ri= 40i)  (rt = 40ms)
  n6 (ri= 250i) (rt = 125ms)
  (2,4) (rt= 0ms)

  first = (2,4) in 0ms
  transmit(2,4)
Sets:
  E    = {n5,n6}
  W    = {n4}
  T    = {}
  I.E  = {n5}
  II.E = {n6}
  II.W = {n4}

E Nodes:
  n5: start= 300ms, end= 560ms, ei= 260i
  n6: start= 310ms, end= 560ms, ei= 250i
W Nodes:
  n4: start= _, end= _, ei=0
T Nodes:
```

```
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
calls:

*****************************************************
Iteration 8:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n5 (ri= 40i)  (rt = 40ms)
  n6 (ri= 250i) (rt = 125ms)

  first = n5 in 40ms
  execute(n5)
Sets:
  E    = {n6}
  W    = {n4}
  T    = {(5,7)}
  I.E  = {}
  II.E = {n6}
  II.W = {n4}

E Nodes:
  n6: start= 310ms, end= 600ms, ei= 330i
W Nodes:
  n4: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
calls:
  (5,7) c= 0ms, t= 0ms

*****************************************************
Iteration 9:
```

```
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n6 (ri= 170i) (rt = 85ms)
  (5,7) (rt= 0ms)

  first = (5, 7)
    transmit(5,7)
Sets:
    E     = {n6}
    W     = {n4, n7}
    T     = {}
    I.E   = {}
    II.E = {n6}
    I.W   = {n7}
    II.W = {n4}

E Nodes:
    n6: start= 310ms, end= 600ms, ei= 330i
W Nodes:
    n4: start= _, end= _, ei=0
    n7: start= _, end= _, ei=0
T Nodes:
    n1: start = 0ms, end = 200ms, ei= 200i
    n3: start= 200ms, end= 300ms, ei= 100i
    n2: start= 210ms, end= 560ms, ei= 450i
    n5: start= 300ms, end= 600ms, ei= 300i
calls:

**************************************************
Iteration 10:
    ams(I)  = 1 i/ms
    ams(II) = 2 i/ms
    n6 (ri= 170i) (rt = 85ms)

    first = n6
    execute(n6)
Sets:
```

```
  E     = {}
  W     = {n4, n7}
  T     = {(6,8), (6,9)}
  I.E   = {}
  II.E  = {}
  I.W   = {n7}
    II.W = {n4}

E Nodes:
W Nodes:
  n4: start= _, end= _, ei=0
  n7: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
calls:
  (6,8) c= 10ms t= 0ms
  (6,9) c= 0ms t= 0ms

**************************************************
Iteration 11:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  (6,8) (rt= 10ms)
  (6,9) (rt= 0ms)

  first = (6,9)
  transmit(6,9)
Sets:
  E     = {n9}
  W     = {n4, n7}
  T     = {(6,8)}
  I.E   = {}
  II.E  = {n9}
```

```
  I.W  = {n7}
  II.W = {n4}

E Nodes:
  n9: start= 685ms, end= 685ms, ei= 0i
W Nodes:
  n4: start= _, end= _, ei=0
  n7: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
calls:
  (6,8) c= 10ms t= 0ms

**************************************************
Iteration 12:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n9 (ri= 100i) (rt= 50ms)
  (6,8) (rt= 10ms)

  first = (6,8) in 10ms
  transmit(6,8)
Sets:
  E    = {n9, n8}
  W    = {n4, n7}
  T    = {}
  I.E  = {n8}
  II.E = {n9}
  I.W  = {n7}
  II.W = {n4}

E Nodes:
  n9: start= 685ms, end= 695ms, ei= 20i
```

```
  n8: start= 695ms, end= 695ms, ei= 0i
W Nodes:
  n4: start= _, end= _, ei=0
  n7: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
calls:

**************************************************
Iteration 13:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n9 (ri= 80i) (rt= 40ms)
  n8 (ri= 200i) (rt= 200ms)

  first = n9 in 40ms
  execute(n9)
Sets:
  E    = {n8}
  W    = {n4, n7}
  T    = {(9,12)}
  I.E  = {n8}
  II.E = {}
  I.W  = {n7}
  II.W = {n4}

E Nodes:
  n8: start= 695ms, end= 735ms, ei= 40i
W Nodes:
  n4: start= _, end= _, ei=0
  n7: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
```

```
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
calls:
  (9,12) c=0, t=0

**************************************************
Iteration 14:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n8 (ri= 160i) (rt= 160ms)
  (9,12) (rt= 0ms)

  first = (9,12)
  transmit(9,12)
Sets:
  E    = {n8}
  W    = {n4, n7, n12}
  T    = {}
  I.E  = {n8}
  II.E = {}
  I.W  = {n7}
  II.W = {n4, n12}

E Nodes:
  n8: start= 695ms, end= 735ms, ei= 40i
W Nodes:
  n4: start= _, end= _, ei=0
  n7: start= _, end= _, ei=0
  n12: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
```

```
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
calls:

**************************************************
Iteration 15:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n8 (ri= 160i) (rt= 160ms)

  first = n8 in 160ms
  execute(n8)
Sets:
  E    = {}
  W    = {n4, n7, n12}
  T    = {(8,11), (8,4)}
  I.E  = {}
  II.E = {}
  I.W  = {n7}
  II.W = {n4, n12}

E Nodes:
W Nodes:
  n4: start= _, end= _, ei=0
  n7: start= _, end= _, ei=0
  n12: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
calls:
  (8, 11) c=0, t=0
  (8, 4) c=10, t=0
```

```
**************************************************
Iteration 16:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  (8,11) (rt= 0ms)
  (8,4) (rt= 10ms)

  first = (8,11) in 0ms
  transmit(8,11)
Sets:
  E    = {n11}
  W    = {n4, n7, n12}
  T    = {(8,4)}
  I.E  = {n11}
  II.E = {}
  I.W  = {n7}
  II.W = {n4, n12}

E Nodes:
  n11: start= 895, end=895, ei= 0i
W Nodes:
  n4: start= _, end= _, ei=0
  n7: start= _, end= _, ei=0
  n12: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
calls:
  (8, 4) c=10,t=0

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

```
Iteration 17:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n11 (ri= 100i) (rt= 100ms)
  (8,4) (rt= 10ms)

  first = (8,4) in 10ms
  transmit(8,4)
Sets:
  E    = {n11, n4}
  W    = {n7, n12}
  T    = {}
  I.E  = {n11}
  II.E = {n4}
  I.W  = {n7}
  II.W = {n12}

E Nodes:
  n11: start= 895, end= 905, ei= 10i
  n4: start=905, end= 905, ei= 0i
W Nodes:
  n7: start= _, end= _, ei=0
  n12: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
calls:

**********************************************************
Iteration 18:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
```

```
  n11 (ri= 90i) (rt= 90ms)
  n4 (ri= 50i) (rt= 25ms)

  first = n4 in 25ms
  execute(n4)
Sets:
  E    = {n11}
  W    = {n7, n12}
  T    = {(4,7)}
  I.E  = {n11}
  II.E = {}
  I.W  = {n7}
  II.W = {n12}

E Nodes:
  n11: start= 895, end= 930, ei= 35i
W Nodes:
  n7: start= _, end= _, ei=0
  n12: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
  n4: start=905, end= 930, ei= 50i
calls:
  (4, 7) c=10ms t=0ms

**********************************************************
Iteration 19:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n11 (ri= 65i) (rt= 65ms)
  (4,7) (rt= 10ms)
```

```
  first = (4,7) in 10ms
  transmit(4,7)
Sets:
  E     = {n11, n7}
  W     = {n12}
  T     = {}
  I.E   = {n11, n7}
  II.E  = {}
  I.W   = {}
  II.W  = {n12}

E Nodes:
  n11: start= 895, end= 940, ei= 45i
  n7: start= 940, end= 940, ei= 0i
W Nodes:
  n12: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
  n4: start=905, end= 930, ei= 50i
calls:

**************************************************
Iteration 20:
  ams(I)   = 0.5 i/ms
  ams(II) = 2 i/ms
  n11 (ri= 55i) (rt= 110ms)
  n7 (ri= 150i) (rt= 300ms)

  first = 110ms
  execute(n11)
```

```
Sets:
  E     = {n7}
  W     = {n12}
  T     = {(11,12)}
  I.E   = {n7}
  II.E  = {}
  I.W   = {}
  II.W  = {n12}

E Nodes:
  n7: start= 940, end= 1050, ei= 55i
W Nodes:
  n12: start= _, end= _, ei=0
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
  n4: start=905, end= 930, ei= 50i
  n11: start= 895, end= 1050, ei= 100i
calls:
  (11,12) c=10, t=0

**************************************************
Iteration 21:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n7 (ri= 95i) (rt= 95ms)
  (11, 12) (rt= 10ms)

  first = (11, 12) in 10ms
  transmit(11,12)
Sets:
  E     = {n7, n12}
```

```
W     = {}
T     = {}
I.E   = {n7}
II.E  = {n12}
I.W   = {}
II.W  = {}

E Nodes:
  n7: start= 940, end= 1060, ei= 65i
  n12: start= 1060, end= 1060, ei= 0i
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
  n4: start=905, end= 930, ei= 50i
  n11: start= 895, end= 1050, ei= 100i
calls:

**************************************************
Iteration 22:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n7 (ri= 85i) (rt= 85ms)
  n12 (ri= 100i) (rt= 50ms)

  first = n12 in 50ms
  execute(n12)
Sets:
  E     = {n7}
  W     = {}
  T     = {(12,10)}
  I.E   = {n7}
```

```
  II.E = {}
  I.W  = {}
  II.W = {}

E Nodes:
  n7: start= 940, end= 1110, ei= 115i
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
  n4: start=905, end= 930, ei= 50i
  n11: start= 895, end= 1050, ei= 100i
  n12: start= 1060, end= 1110, ei= 100i
calls:
  (12, 10) c=10, t=0

**************************************************
Iteration 23:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n7 (ri= 35i) (rt= 35ms)
  (12,10) (rt= 10ms)

  first = (12,10) in 10ms
  transmit(12,10)
Sets:
E     = {n7}
W     = {n10}
T     = {}
I.E   = {n7}
II.E  = {}
I.W   = {n10}
```

```
  II.W = {}

E Nodes:
  n7: start= 940, end= 1120, ei= 125i
W Nodes:
  n10: start= _, end= _, ei= _
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
  n4: start=905, end= 930, ei= 50i
  n11: start= 895, end= 1050, ei= 100i
  n12: start= 1060, end= 1110, ei= 100i
calls:

**************************************************
Iteration 24:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n7 (ri= 25i) (rt= 25ms)

  first = n7 in 25ms
  execute(n7)
Sets:
  E    = {}
  W    = {n10}
  T    = {(7,10)}
  I.E  = {}
  II.E = {}
  I.W  = {n10}
  II.W = {}

E Nodes:
```

```
W Nodes:
  n10: start= _, end= _, ei= _
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
  n4: start=905, end= 930, ei= 50i
  n11: start= 895, end= 1050, ei= 100i
  n12: start= 1060, end= 1110, ei= 100i
  n7: start= 940, end= 1145, ei= 150i
calls:
  (7,10) in 0ms

**************************************************
Iteration 25:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  (7, 10) (rt= 0ms)

  first = (7,10) in 0ms
  transmit(7,10)
Sets:
  E    = {n10}
  W    = {}
  T    = {}
  I.E  = {n10}
  II.E = {}
  I.W  = {}
  II.W = {}

E Nodes:
  n10: start= 1145, end= 1145, ei= 0
W Nodes:
```

```
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
  n5: start= 300ms, end= 600ms, ei= 300i
  n6: start= 310ms, end= 685ms, ei= 500i
  n9: start= 685ms, end= 735ms, ei= 100i
  n8: start= 695ms, end= 895ms, ei= 200i
  n4: start=905, end= 930, ei= 50i
  n11: start= 895, end= 1050, ei= 100i
  n12: start= 1060, end= 1110, ei= 100i
  n7: start= 940, end= 1145, ei= 150i
calls:

**************************************************
Iteration 26:
  ams(I)  = 1 i/ms
  ams(II) = 2 i/ms
  n10 (ri= 50i) (rt= 50ms)

  first = n10 in 50 ms
  execute(n10)
Sets:
  E    = {}
  W    = {}
  T    = {}
  I.E  = {}
  II.E = {}
  I.W  = {}
  II.W = {}

E Nodes:
W Nodes:
T Nodes:
  n1: start = 0ms, end = 200ms, ei= 200i
  n3: start= 200ms, end= 300ms, ei= 100i
  n2: start= 210ms, end= 560ms, ei= 450i
```

```
    n5: start= 300ms, end= 600ms, ei= 300i
    n6: start= 310ms, end= 685ms, ei= 500i
    n9: start= 685ms, end= 735ms, ei= 100i
    n8: start= 695ms, end= 895ms, ei= 200i
    n4: start=905, end= 930, ei= 50i
    n11: start= 895, end= 1050, ei= 100i
    n12: start= 1060, end= 1110, ei= 100i
    n7: start= 940, end= 1145, ei= 150i
    n10: start= 1145, end= 1195, ei= 50
calls:
```

A.2 Distributions for the 24 Vertex DAG

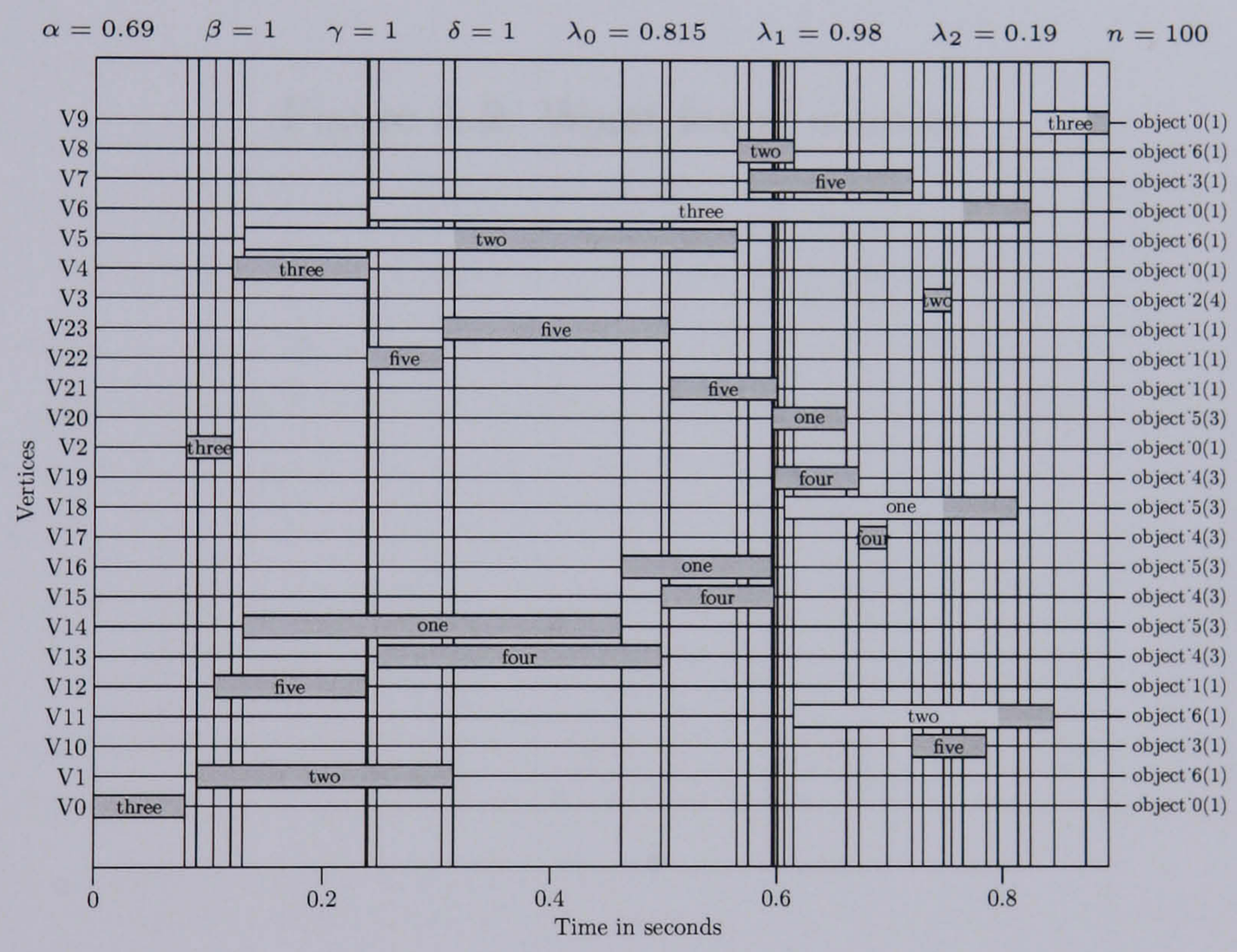


Figure A.1: Best possible distribution

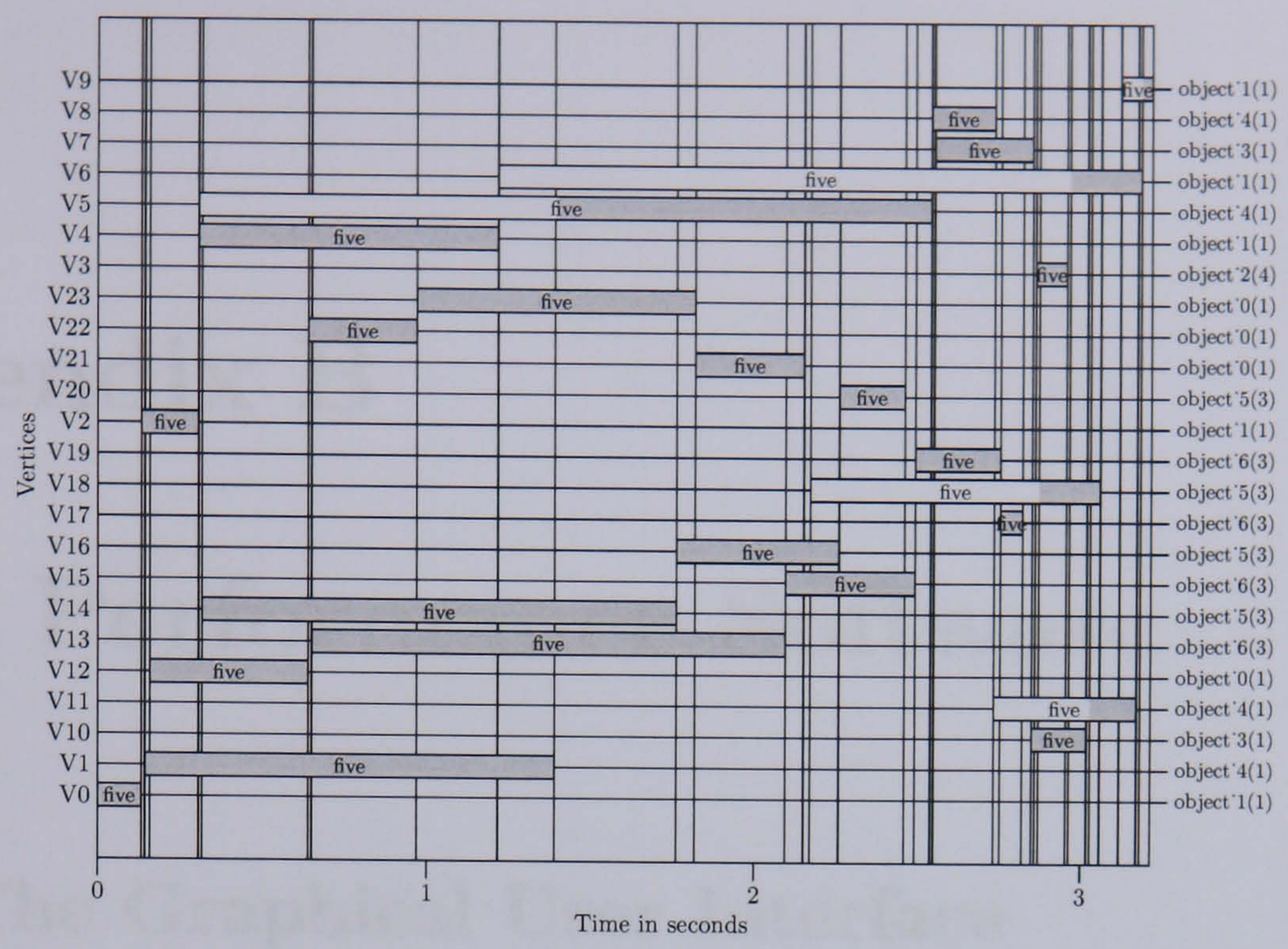


Figure A.2: Worst found solution

Appendix B

Grid Performance Software

B.1 The Graphical User Interface

Figure 6.1 demonstrates the structure of the Grid Performance Software (*GriPS*) used for the Simulation. The Graphical User Interface (GUI) relies on information it receives from SpeedServer and LocationBroker described in the following. Both programs have to be started before using the GUI. All parameters used for the simulation are specified within the GUI and as soon as the simulation is started a class specifically for the model type is instantiated and the corresponding performance monitoring client is called.

The SpeedServer:

The SpeedServer is measuring the speed of the machine where it is running on. Listing (B.2, B.3) shows how the SpeedServer is started from a console window. It is using a small code block that represent a faked instruction and measures the time this code will need to execute. This faked instruction is also used when an object executes on this machine. One faked instruction executes the following code block:

```
long d=0;
for (int i=0; i<100; i++) {
        d += i+ i*i + i*i*i + i*i*i*i;
}
```

Listing B.1: Fake instruction

```
start rmiregistry

start java SpeedServer
```

Listing B.2: Start the SpeedServer from a windows console

```
rmiregistry &

java SpeedServer &
```

Listing B.3: Start the SpeedServer from a Unix console

The LocationBroker:

The LocationBroker is a data structure that provides the simulation and its graphical user interface (GUI) with environmental data such as HostMachines, Connections and Services. A HostMachine provides information about the minimum, average and maximum speed, this machine has reached during its lifetime, additionally the actual speed. Since the LocationBroker keeps checking the speed information of the HostMachine in regular intervals, it is able to notice quickly when a HostMachine is not anymore available. Under normal circumstances (when no failure occurs) the HostMachine will unregister itself from the LocationBroker before it is shut down. In addition the HostMachine class does the actual execution of instructions for every SimulatedApplicationObject (SAObject). This way it is feasible to artificially let one machine fail after it has executed a certain number of instructions.

The LocationBroker also implements the association between ServiceDetails and HostMachine, as well as a mapping from HostMachines to Connections. For convenient storage of a HostMachine object the MachineRegistration class is used. This is necessary, because if the connection to one HostMachine breaks down, there is no chance to get the data needed for a lookup. The MachineRegistration class therefore stores copies of the necessary attributes that are needed, even if the machine is unreachable. On one computer there is exactly one MachineRegistration for one HostMachine, though RMI creates copies of it when returning or passing it as parameter.

Furthermore the LocationBroker provides a feature to inform an object interested in a particular HostMachine if the status of this machine has changed. Whereby the status of a HostMachine can be: available, not available, registered or unregistered. Within the simulation every PMC registers itself at the LocationBroker to listen for an event of every HostMachine it has called. Listing (B.4, B.5) shows how the location broker is started from a console window.

```
start  rmiregistry

start  java sim.LocationBroker
```

Listing B.4: Start the LOB from a windows console

```
rmiregistry &

java sim.LocationBroker &
```

Listing B.5: Start the LOB from a Unix console

The GUI has to connect to a LOB as explained in section B.1.4.1.

B.1.1 The GUI Structure

The GUI is divided into four parts as can be seen in Figure 6.2.

The menu bar

The menu options like loading, saving, exporting, importing of data, creation of applications and changing of settings are accessible. The menu bar is described in detail in Section B.1.2.

The tool bar

Contains two buttons one is used to start a simulated application and the other to reset the total executed instructions of all machines.

The tabbed pane

Contains different panels to create the application and to view the results. These panels can be accessed by pressing on the corresponding tab and are described in detail in Section B.1.3.

The status bar

Displays the actual LOB and the number of threads currently running in this virtual machine.

B.1.2 The menu bar

The menu bar contains five main menus.

File Options to clear, load, save, export and import data, as well as quitting the program can be found here.

Clear Clear or remove the data of services, machines, connections and application or all.

Load Load data of services, machines, connections, application, settings or all.
It is also possible to load only above parts out of a file that contains all data. Load settings, or all will ask you, if you want to use the saved LOB or register the loaded data on the current LOB.

Save Save the data of services, machines and connections of the current LOB, application, settings or all.

Export app.-data The application data is exported into a CSV (**C**omma **S**eparated **V**alues) file. The file contains the used model, the number of remote calls, total number of instructions executed, the minimum and maximum delay of all used connections, the duration of the application and a list of machines with a triplet of executed object, service number and instructions.

Export Obj.-diagram data Exports the data of the *Object Diagrams panel* (s. Section B.1.3.7).
Three files are created during the export. The data file, which contains the pure data of times and percentage of total instructions per object as well as the data when which machine was executing which object.
The second file is a *MetaPost* (.mp)- file to create from the data a simple PS-file. (see [20]).
The last file is a TEX-file to create a document (PDF, DVI, PS, etc) file to view the data. (see [28])
More information how to work with these files is displayed in Appendix B.2.5

Import Connections Importing connections from several files with the additionally option to replace host names or IP-addresses by a replacement string read out of a file. (see Section B.1.3.4, AppendixB.3.1 and B.3.2)

Exit Quitting the program.

Tree Option to create and fill the simulated application tree.

Create Random Tree A random application tree is generated. The current application tree is lost. The number of objects (nodes) and the maximum depth of the tree, as well as the seed for the random generator can be specified in a dialog.
The objects are filled with a randomly chosen service out of all available services. More about the application structure in Section B.1.3.3.

Fill in fastest machines Assign the fastest possible machine to the objects.

Fill in random machines Assign a random machine to the objects. The seed for the random generator can be specified in the *Settings-Dialog* B.1.4.

Reset application Reset all objects of the simulated application to the state *initialised* and their executed instructions count to 0.

Settings Change the default settings and log options.

Change settings Opens the *Settings Dialog.*

Object Diagram logging If selected the data needed to export object diagram data is logged. (If unchecked data export is not available!).

Testing Used only for testing purposes.

Help A small about dialog.

B.1.3 The different panels

There are seven main panels in the GUI that can be selected by pressing on the tab with the name of the panel.

- The *Service Panel* to add, remove and change services
- The *Machine Panel* to add, remove and change machines and their services
- The *Application Structure Panel* to create a simulated application and to set some parameters
- The *Connection Panel* is used to add, remove and change connections
- The *Model Panel* where the performance monitoring model can be chosen
- The *Application Panel* displays some data of the application's results
- The *Object Diagrams Panel* visualises the execution of objects in diagrams.

The different panels are described in the following subsections.

B.1.3.1 The Service Panel

The service panel is used to add, remove or change service registrations on the actual LOB.

A service has a unique service number (when newly created it is one more then the last available number), a name, a minimum, a maximum and average number of instructions to execute.

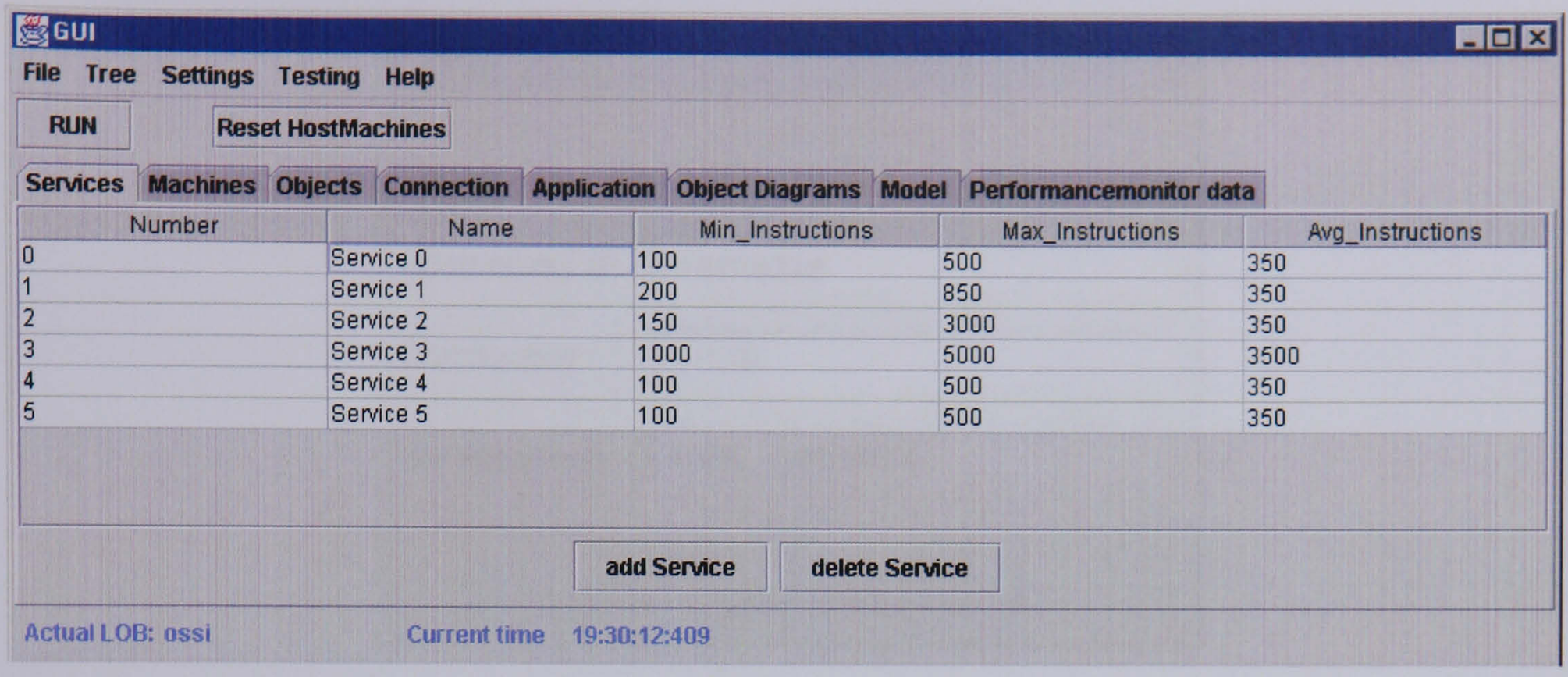


Figure B.1: *The Graphical User Interface Service Panel*

A service can be added by pressing the *add-button*. A default service is created and can be altered by selecting the field in the table and typing the new values. Confirm the new value by pressing *ENTER*.
To delete a service, mark the service in the table and press the *delete-button*. The actual version does not allow to delete multiple services at a time (the service which was selected last is deleted).
The default values of a service can be altered in the Settings-Dialog (see Section B.1.4).

B.1.3.2 The Machine Panel

The machine panel allows the adding, removing and changing of machine registrations on the actual LOB.

Machines and their properties are shown in a tree structure on the left side of the panel. A machine is either in the available or not available branch of the tree. The available branch shows for each machine:

- Which services are provided
- Runtime data such as the total executed instruction count and the failure instruction where the machine will simulate a failure (crash)
- The speed data such as the current, minimum and maximum speed
- Settings like the time zone and location.

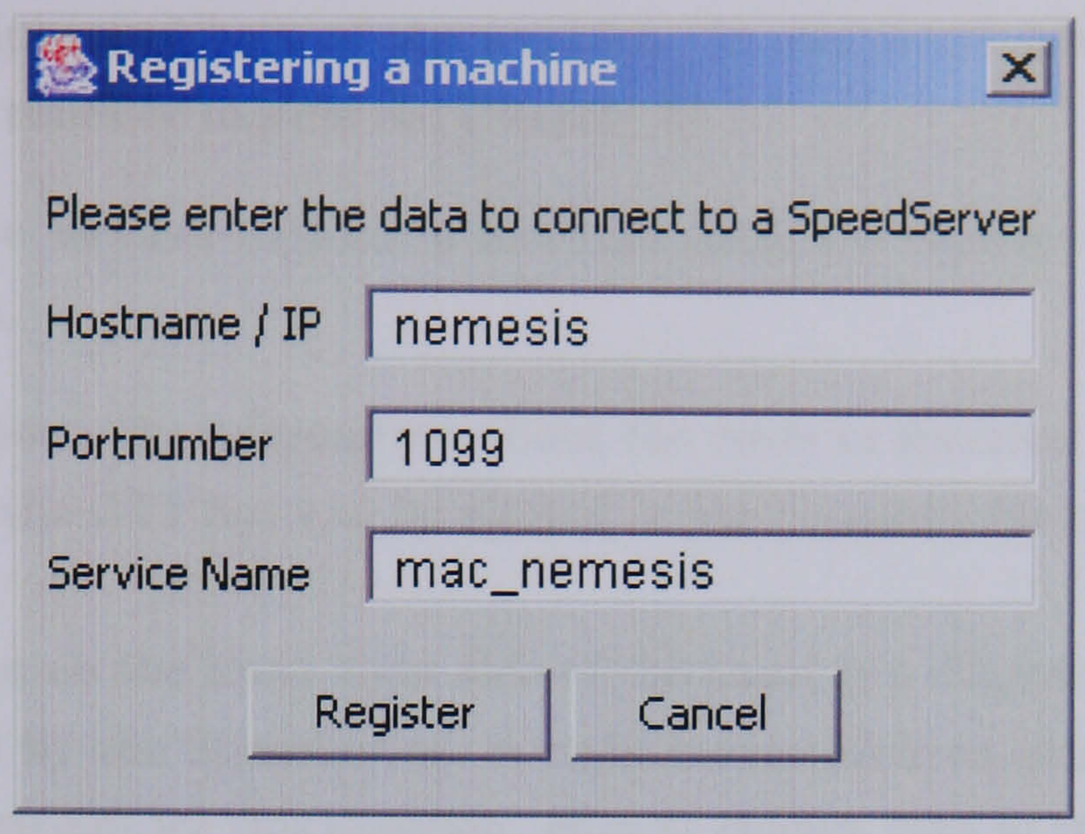


Figure B.2: Adding a machine

Not available machines show only the registered services of this machine.
Whether a machine is available or not is determined by the LOB which is checking registered machines in intervals.

To add a machine one can either press the *add-machine-button*, which opens a dialog (Figure B.2) to enter the data necessary to connect to the remote machine, or use the right mouse-button which is described later.

To connect to a remote machine the machine name or its IP-address, the port number of the rmiregistry and the name how the machine is registered in the rmiregistry are needed.
To remove a machine, select it in the tree and press the *remove-machine-button.*

On the right side of the machine panel two tabs are located which show detailed information on the machine which is selected in the tree. One is *Machine details*, the other one is *Costs.*
The *Machine details* tab allows to:

- register and unregister services
- set the instructions after which the machine will fail
- shows the actual executed instructions
- The *Reset* button resets the actual executed instructions to 0 (Available for all machines).

- Change the alpha and beta of the machine. (α and β are used for estimating the start and end times of objects see Chapter 4).
- Add or remove services to/from a machine using the *register-service* or *unregister service* button.

The *Costs* tab shows the information about the costs in specified times. This is up to now not editable in the GUI but can be altered in the property file of the SpeedServer.

Last but not least on the lower right side of the panel is a diagram showing the actual speed data provided by the SpeedServer. A right mouse click on this diagram allows the user to change the settings for this diagram. How to change data in the dialog is described in Appendix B.2.
Note: Using a right click on the tree pops up a menu, where most of the above actions are available. Additionally one can choose which data is displayed in the machine-tree.

B.1.3.3 The Application Structure Panel

The application structure panel is used to create a simulated application with all its parameters.

This panel as shown in Figure 6.4 is divided in three parts. On the left side there is the application structure shown in a tree. This tree structure can be altered by adding or removing nodes. Nodes in this tree represent an object (service instance) which is part of the application.
At the upper right side are information shown, based on the user – provider contract for the application. The lower right side gives additional information about the selected node.

To add a new node, select the node where you want to add a child-node or sibling to and press the *Add Node* button. This will display a dialog, where you can choose to add the new node as a child or sibling. A sibling can be added above or below the selected node.
To delete a node, select it and press the *Delete Node* button. This will delete this node and all its children.
These options are also available by pressing the right mouse button on the application-tree.

Additionally there is a button to expand the tree and one to pre- estimate the times (as described in Chapter 4) for the application (using the selected speed setting from the Settings dialog). In the combo box below the tree, one can choose which time should be displayed in the tree (No time, pre-estimated time and estimated time).

For every object in this tree the right service instance has to be specified. This is done by selecting the node (which represents an object) and choose the machine and service in the *Major Attributes* tab. Another possibility is to fill in the machines automatically (Further information are available in Section B.1.2.) The number of faked instructions the node executes before calling its children are specified to the average instruction of the service specified in the Service Panel. However this number can be altered manually and it only should remain between the minimum and the maximum instructions specified for this particular service. More information about the Service Panel are in Section B.1.3.1.

The other tabs are of no further interest, because the underlying performance monitor structure is not providing any data for it anymore.

In the upper right part is the general information of the whole application. It is possible to enter the maximum allowed costs and the maximum allowed time that the application is allowed to use. Also the start time/date when the application is simulated to start can be set.

The application structure panel shows during runtime of the simulated application the status of each object. This status will be identified by a coloured bullet in front of the related node. The different colours and their meaning are explained in the following:

- Red bullet: In this object some data such as the service or the machine for this service is not available or not selected.
- Blue bullet: The object is correctly initialised.
- Yellow bullet: The object is started and executing or waiting on its children to terminate.
- Green bullet: The object has terminated.

B.1.3.4 The Connections Panel

This panel allows to add and remove connections between machines and change the data of these connections on the current LOB.

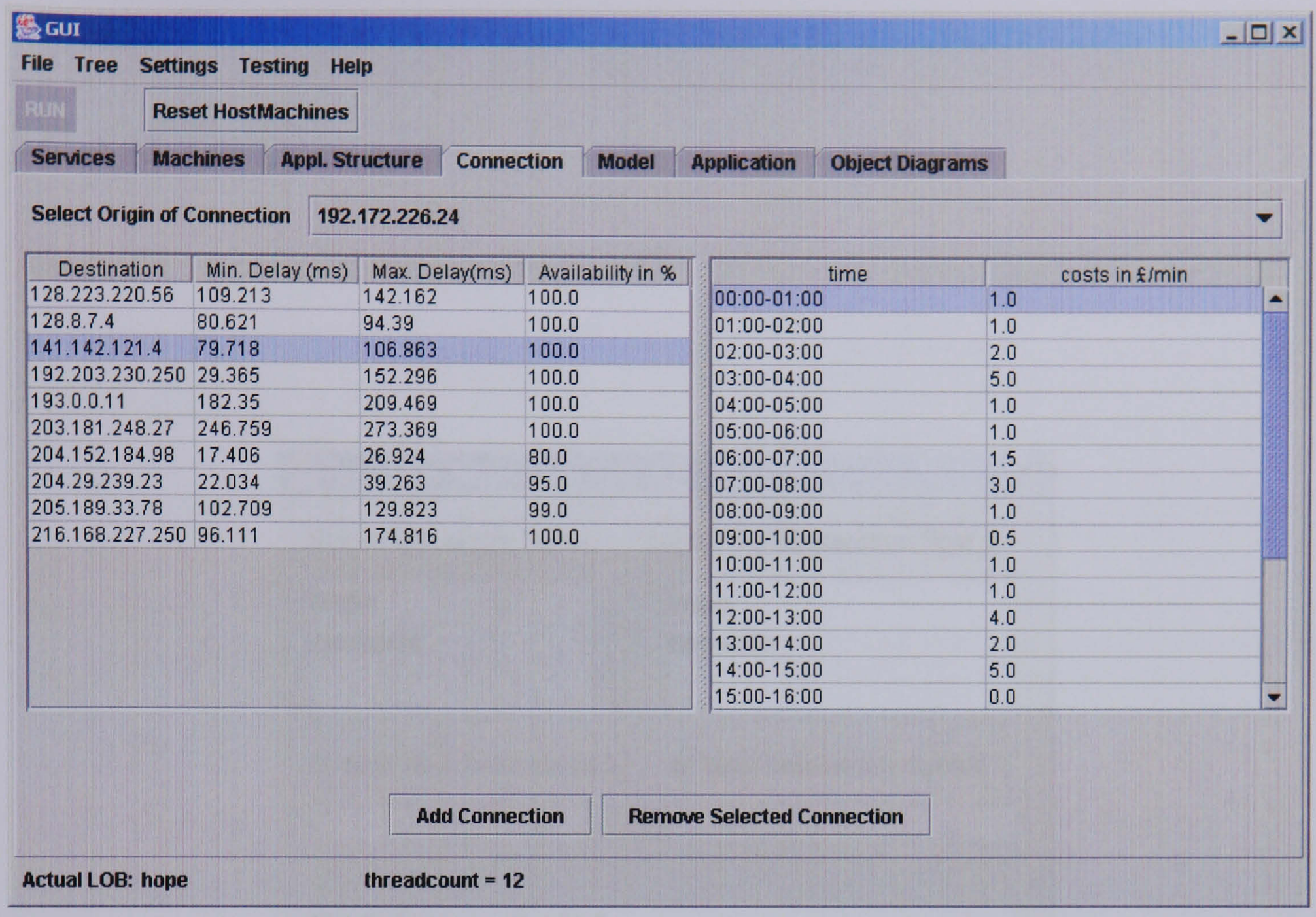


Figure B.3: The Connection Panel

Connections have a origin and destination address, a min- and max-delay between these connections in milliseconds and an availability in percent.

A subset of Connections can be selected by choosing the origin in the combo box at the top of Figure B.3. All connections having the selected origin are shown in the table below (the combo box on the left). Select one connection in the table will show the costs of these connections over the time in the table on the right hand side. To add a connection press the *Add Connection* button.

A dialog pops up as shown in Figure B.4. The source and destination machine of the connection are to be entered. An already registered machines from the lists in the top can be entered or a name can be entered in the text fields below. If something is entered in the text fields, this value will be used. The min- and max-delay of the connection as well as the availability are set to default settings and can be edited.

To remove a connection select one in the left table and press the *Remove Selected Connection* button. Adding and deleting multiple connections is not supported in the current version. There is a possibility to import connections from several files, by using

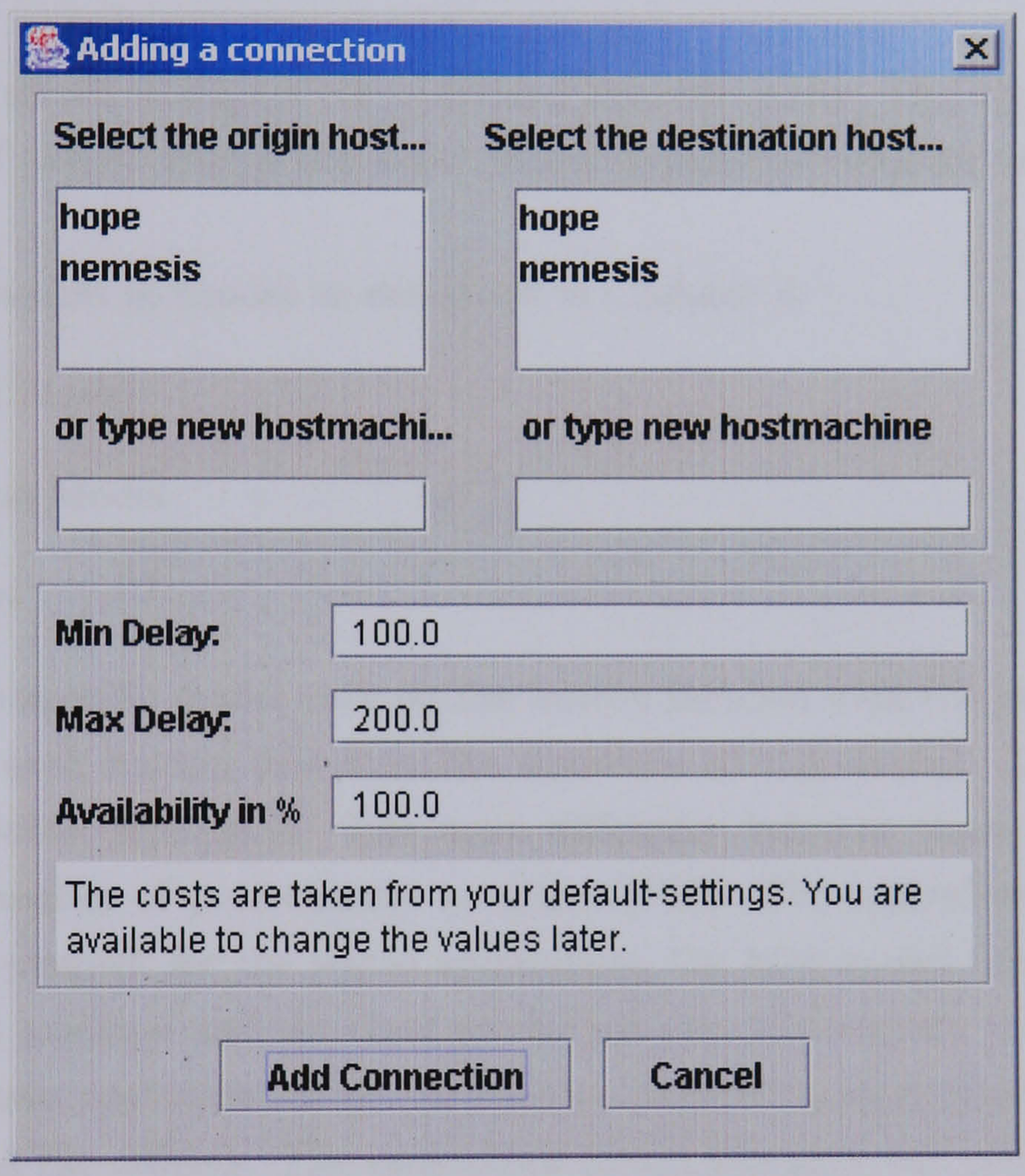


Figure B.4: Adding a connection

the menu *File* and pressing *Import Connections* in there. It is additionally possible to replace names or IP-Addresses of these files by writing a replacement file and specifying it in the import dialog. Examples of how these files are formatted are shown in Appendix B.3.1 and B.3.2.

B.1.3.5 The Model Panel

The model panel allows to specify the performance monitor model used to execute the application.

The panel as shown in Figure 6.6 is divided into two parts. The upper part allows to choose the model, where the in the lower model dependent data for the selected model can be specified.

There are 3 models to choose as described in Chapter 3:

- Centralised Model
- Hostmachine Model
- Class Model

To choose one specific model click on the button labelled with the name of the model. A picture above each button describes the structure of this model. To see the picture more detailed double click on it. The main difference between these models is the location and the number of performance monitors (PM). The centralised model has only one performance monitor for the whole application, the host model has one performance monitor per used machine and the class model has one performance monitor per service instance. In the host and class model the location of performance monitor is specified by the distribution of the objects. The centralised model has only one performance monitor and therefore its location needs to be specified (bottom panel). For the centralised model enter the hostname where the PM is located, the port of the running rmiregistry on this host and the name of the PM with which it is bound in the rmiregistry.

B.1.3.6 The Application Panel

This panel as shown in Figure B.5 gives information about the application.

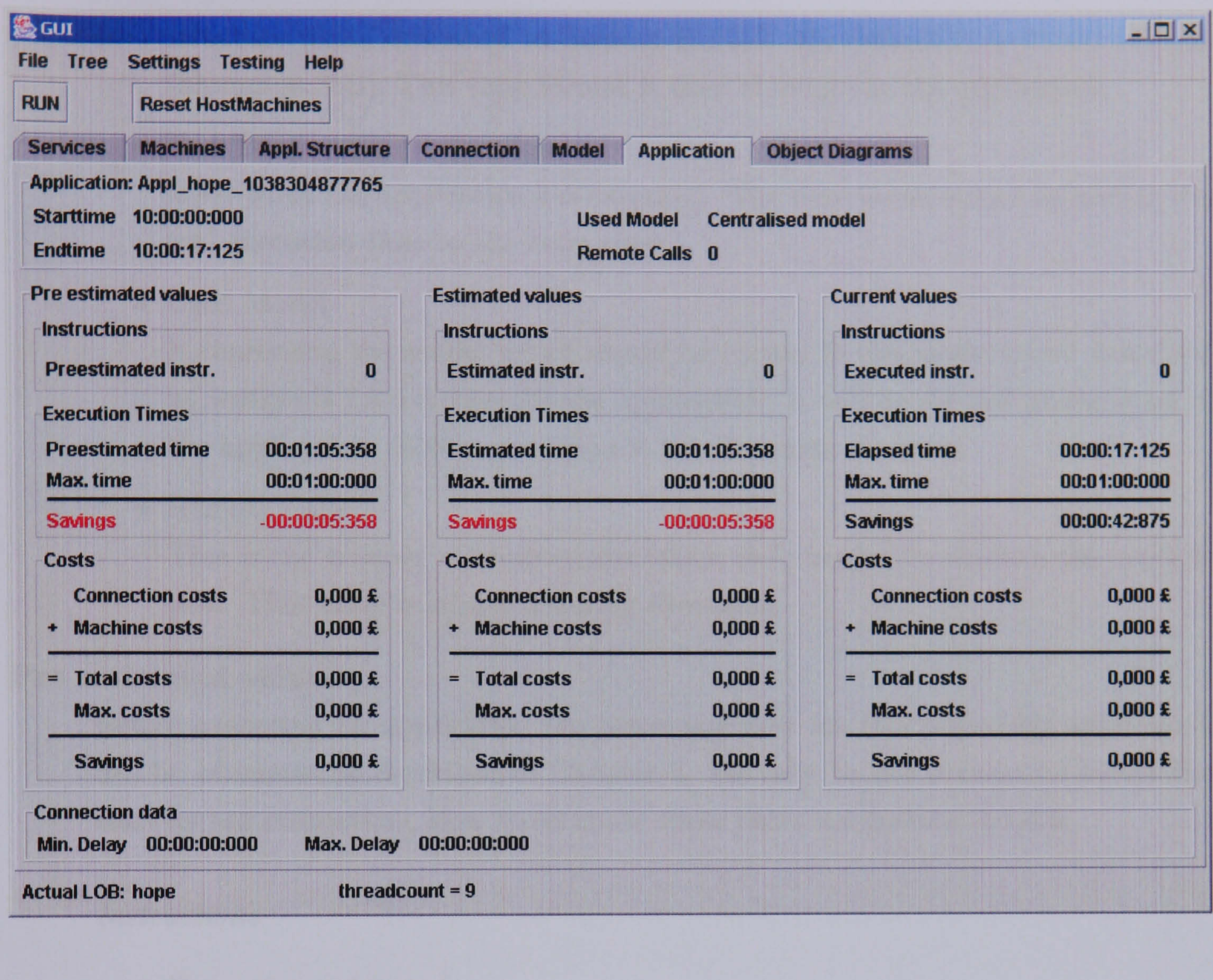


Figure B.5: The Application Panel

It is divided in five major parts. The whole panel is only interesting during execution or afterwards because these values are set or being adjusted only during runtime. Before execution all values are initialised with the default values. The following list describes the five parts of the panel.

Application:

The label of this part shows the name of the application which has the following syntax: Appl_`HOSTNAME`_`STARTTIME`. In detail: `HOSTNAME` is the name of the host machine where the GUI is running and `STARTTIME` is the real start time (milliseconds after 1970) of this particular application. This means that the name for an application changes as soon as you restart the execution because the system time has changed.

Contents:

- Start Time

  Is set at the start of the execution to the value specified in the application-

structure panel described in Section B.1.3.3. By default it is `01:00:00:000` (hh:mm:ss:SSS). This time format is used throughout the application.

- End Time
  Is set when the application has executed. The time is calculated by adding the real execution time to the start time.
- Used Model
  It represents the model which would be chosen in the model panel described in Section B.1.3.5 to execute the application. It will be set just at the start of the application. Before execution it has the value `no data`.
- Remote Calls
  This is the number of remote calls which were needed to execute the application. This value is adjusted during execution.

Pre estimated values

Before executing an application the times and cost for this particular application can be estimated as described in Chapter 4. We refer to this estimation before the start as pre estimation. How to estimate these times see section B.1.3.3.

Instructions

- Pre estimated instr.[1]
  Will show the total number of instructions needed to terminate the application.

Execution times

These values are only set if times were pre-estimated.

- Pre-estimated time
  The estimated time that it will take to execute the application with the current settings.
- Max time
  The time which is allowed to execute the application. This time is specified in the application structure panel.
- Savings
  This time is the difference between the Max time and the Pre-estimated time. A positive number means that the application can be executed in time. A negative number (red digits) means that the execution time will take longer

[1]Not implemented

as it is allowed. The contract can not be accomplished.

Costs

- Connection costs
  Every connection has cost depending on the start time and the duration of the connection usage. The shown value represents the pre-estimated sum of all connection cost.
- Machine costs
  The cost for a machine is time dependent, too. This value represents the sum of all machine cost which will be used to execute the application with the current settings.
- Total cost
  This is the sum of machine and connection cost.
- Max costs
  The cost which is allowed to execute the application. This value is specified in the application structure panel.
- Savings
  This cost is the difference between the Max cost and the Total cost. A positive number means that the application can be executed cheaper than the maximum cost provided by the contract. A negative number (red digits) means that the execution cost will be higher – The contract can not be accomplished.

Estimated values

While executing an application other programs (eg.Performance Monitors) observe the behaviour of this application. For example if a host machine crashes this observer tries to execute parts of the application on another host. This reconfiguration costs time and as a result the execution time has to be estimated again. The values in this part can change during execution. Just after starting an application they are the same as the pre-estimated values. During execution these values will be adjusted in a later version of this software when a part of the application runs faster or slower as expected. At the end of the execution this values should be close to the real values.

Current values

This are the real values (cost, instructions, elapsed time) for the execution and therefore show only information about executed parts. After execution this values represents the real values for the execution.

Connection data

For all connections which can be used there exist statistical data. To add or modify a connection see Section B.1.3.4.

- Min Delay
  This value represents the sum of all min delays of the used connections. This value will be adjusted during the execution. Before the start it has the value `00:00:00:000`.
- Max Delay
  This time value represents the sum of all max delays of the used connections. This value will be adjusted during execution. Before the start it has the value `00:00:00:000`.

The real time value for building up all is between Min Delay and Max Delay.

B.1.3.7 The Object Diagrams Panel

The object diagram panel as shown in Figure 6.7 also shows information about an executed application.

The panel is divided in two parts labeled with `Tree` respectively `Diagram`. It is possible to change these parts from the left side to the other and vice versa by dragging the labels to the other side. The tree part shows the application structure. To fully expand the tree click the **expand** button in the upper left corner of the panel. For each node there exists a diagram in the diagram part. The diagram shows graphically the behaviour during execution. Each diagram has a vertical axis labelled with percentage values and a horizontal axis labelled with times. It can happen that there are too many labels at the axis that it is impossible to read. To change the look of the diagram right click on one. This will pop up a dialog which is described in the Appendix B.2. Changes in this dialog affect all diagrams – It is not possible to change only the look of one particular diagram. The default type of the diagram is `Stairs` and takes the executed instructions dependent on the time into account. This can also be modified in the dialog which was mentioned above. Diagrams can be exported by clicking on the `Export Obj.-diagram data` button in the menu bar (see Section B.1.2).

B.1.4 The Settings Dialog

The settings dialog allows the user to specify default values, timer settings and which LocationBroker is to be used.

The Settings dialog can be opened by selecting *Settings – Change Settings* in the menu bar.

B.1.4.1 Tab LocationBroker

The LocationBroker Tab is shown in Figure B.6 and specifies a LocationBroker.

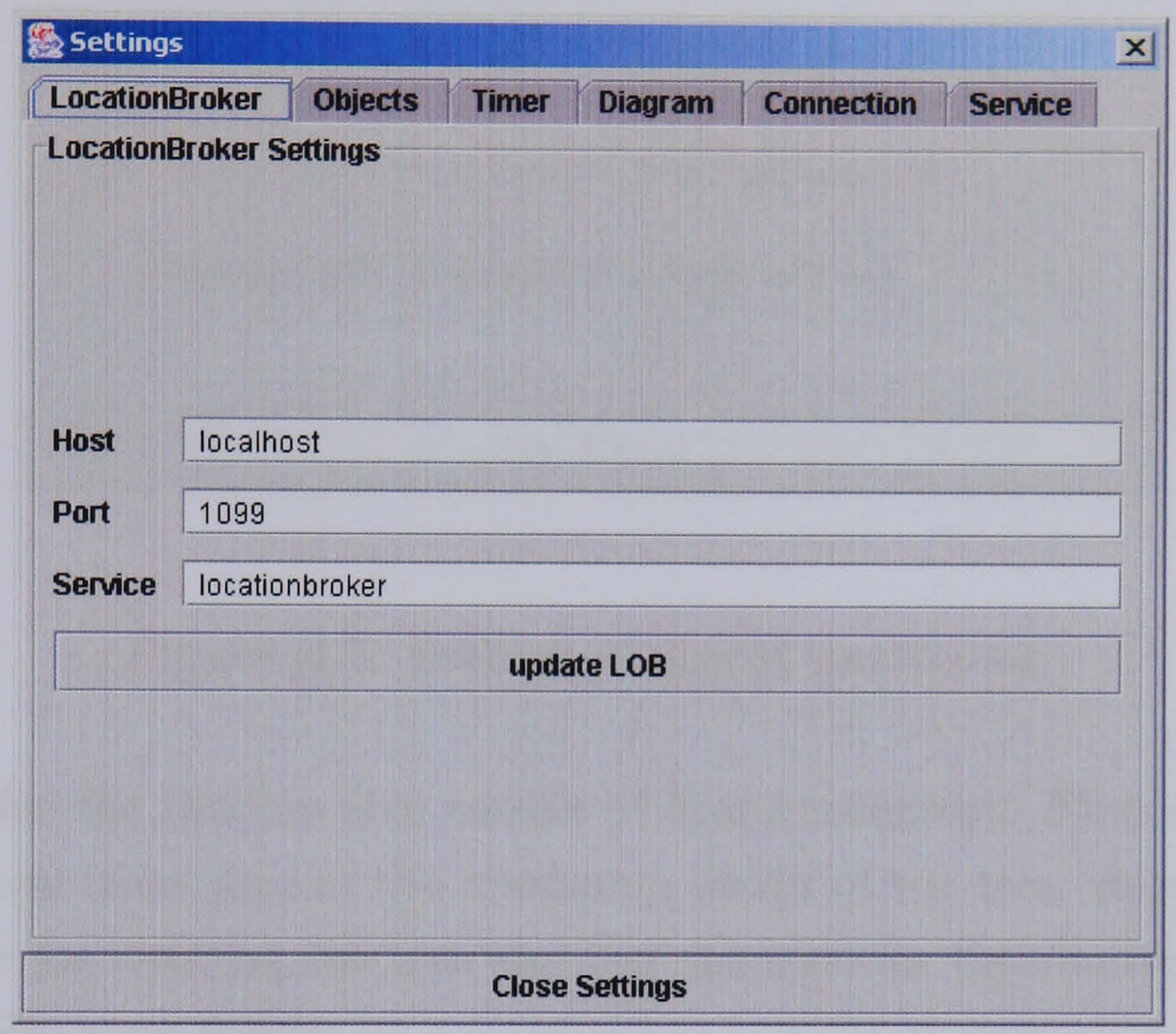


Figure B.6: Setting the LocationBroker

A LocationBroker is specified by the IP-address or name of the machine, a port number of the rmiregistry on the given machine and the name how the LocationBroker is bound in the rmiregistry.
Pressing the *Update LOB* button tries to connect to the LocationBroker.
If the connection was successful the status bar of the main window will show the name of the machine as the actual LocationBroker (Actual LOB).

B.1.4.2 Tab Objects

The objects panel (Figure B.7) specifies the settings for a random tree and the settings for pre-estimating the times of the application.

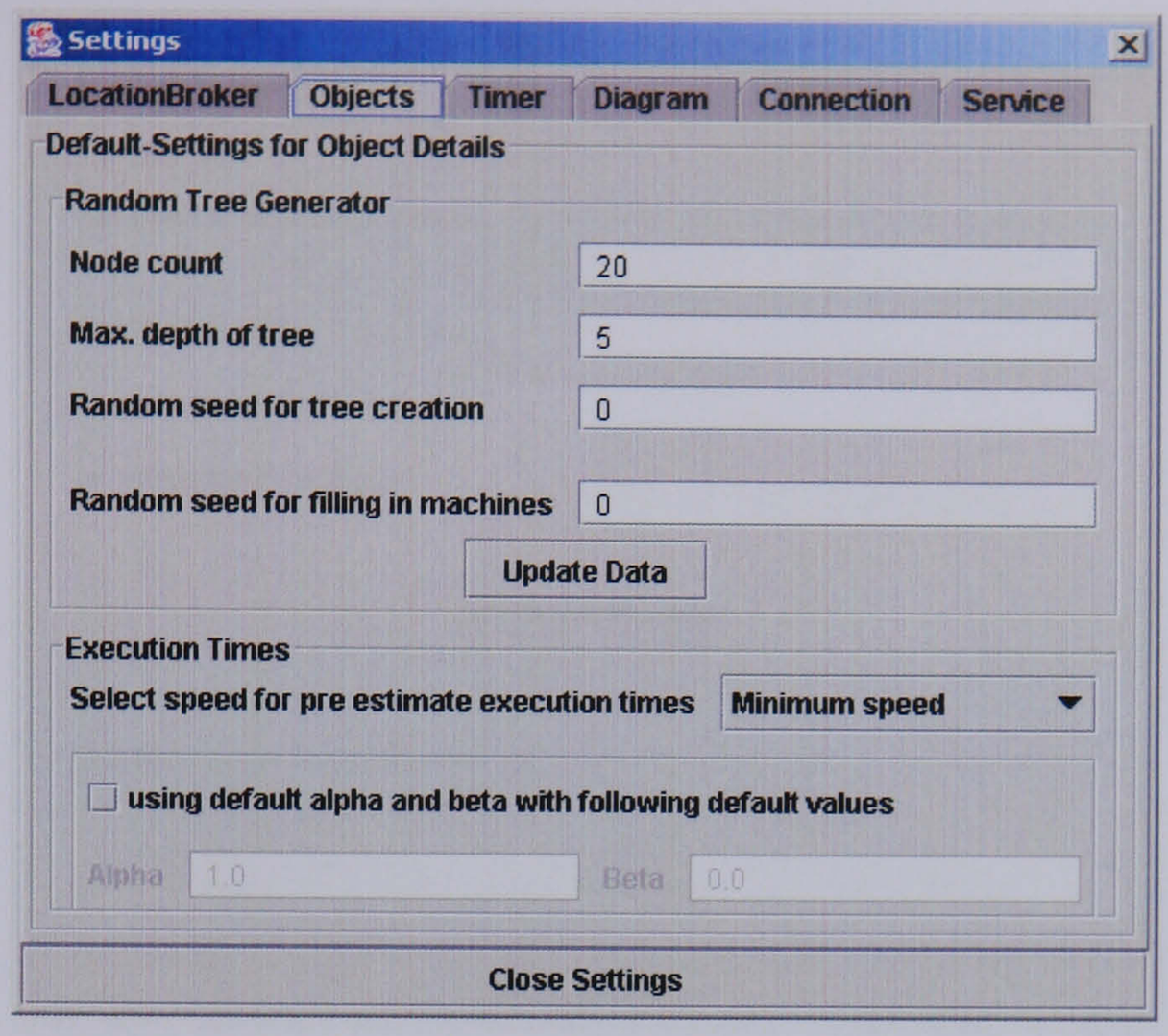


Figure B.7: Setting of object parameter

The settings for the random tree consist of four parameters. First the node count of the randomly chosen tree, second the maximum depth of the tree, third the seed for the random generator for creating the tree and last the seed for the random generator when filling in the machines randomly.
In the pre-estimation part one can select which machine speed should be used, like the minimum, maximum, current, the cached current or a constant speed of 1000 instructions per second. Further the alpha and beta settings off all machines can be overridden by using the default alpha and beta settings, that can be specified here as well.

B.1.4.3 Tab Timer

The timer panel as shown in Figure B.8 is up to now only used for the cache timer.

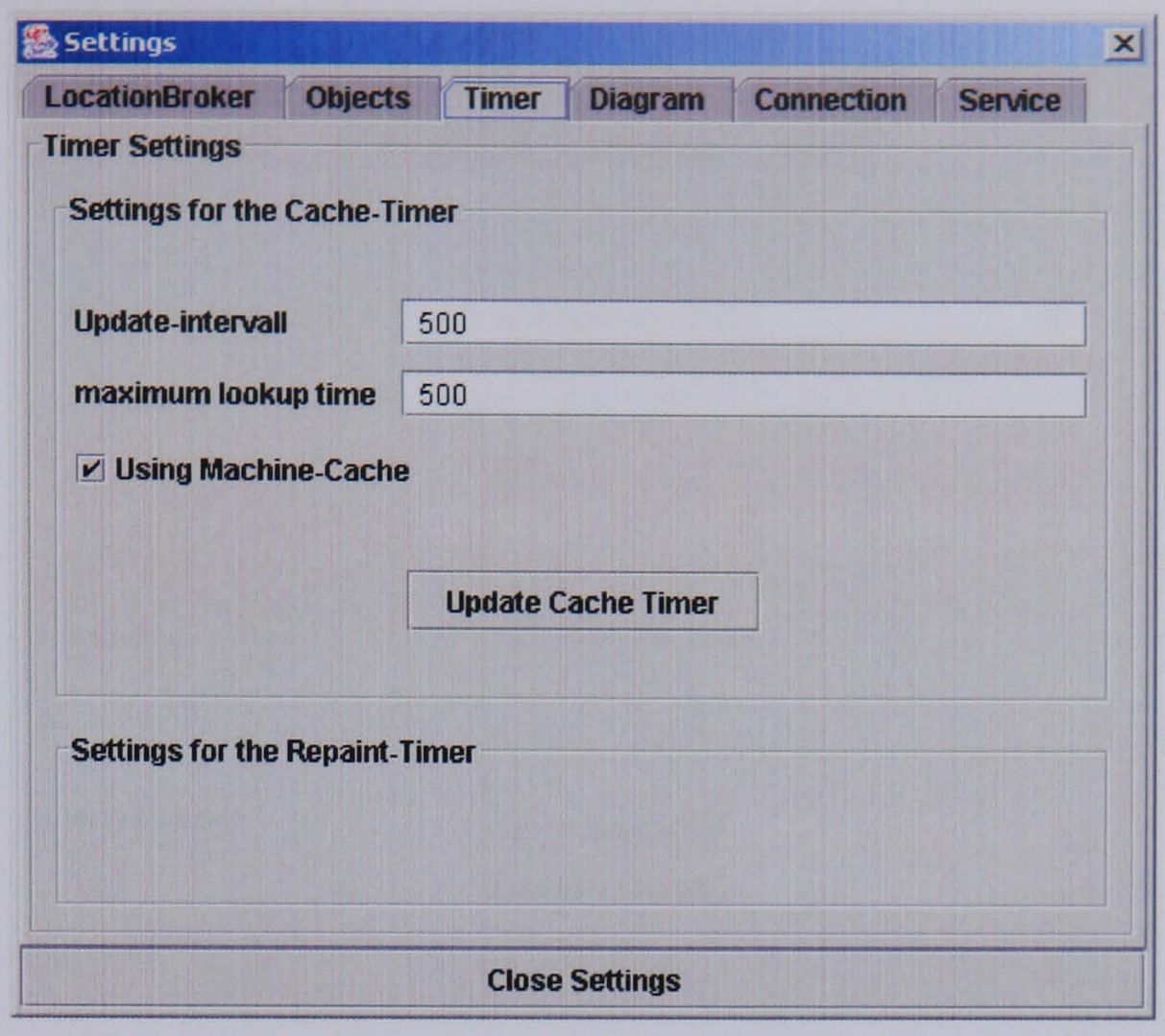


Figure B.8: Setting the timer

The cache timer specifies in which intervals data is cached. The maximum lookup interval specifies the time the caching of one data snapshot is allowed to take. This value should be less than the update interval.
If *Using Machine-Cache* is selected the speed diagrams are active.
There is space reserved for the repainting time of the GUI. But up to now it is not implemented to change this value.

B.1.4.4 Tab Diagram

This panel (Figure B.9) is responsible to set the values for all speed diagrams of the machine panel.

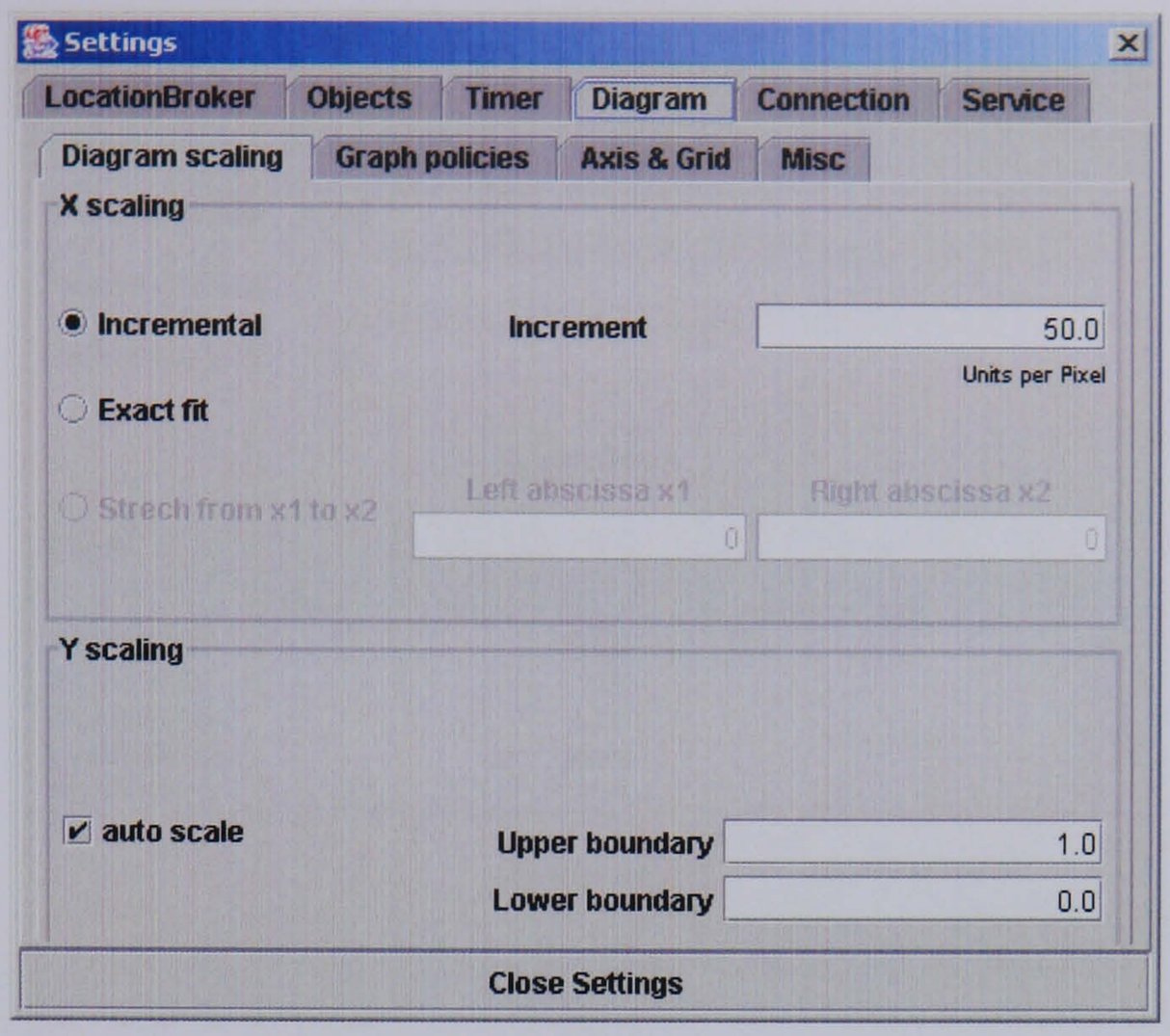


Figure B.9: Setting the speed diagram settings

These values can be changed for only one machine (see Section B.1.3.2). For a description of the dialog see Appendix B.2.

B.1.4.5 Tab Connection

This panel (Figure B.10) allows to set the default values for the creation of connections like min- and max-delay (in milliseconds), the availability (in percent) and the costs in specified times (in pounds per minute).

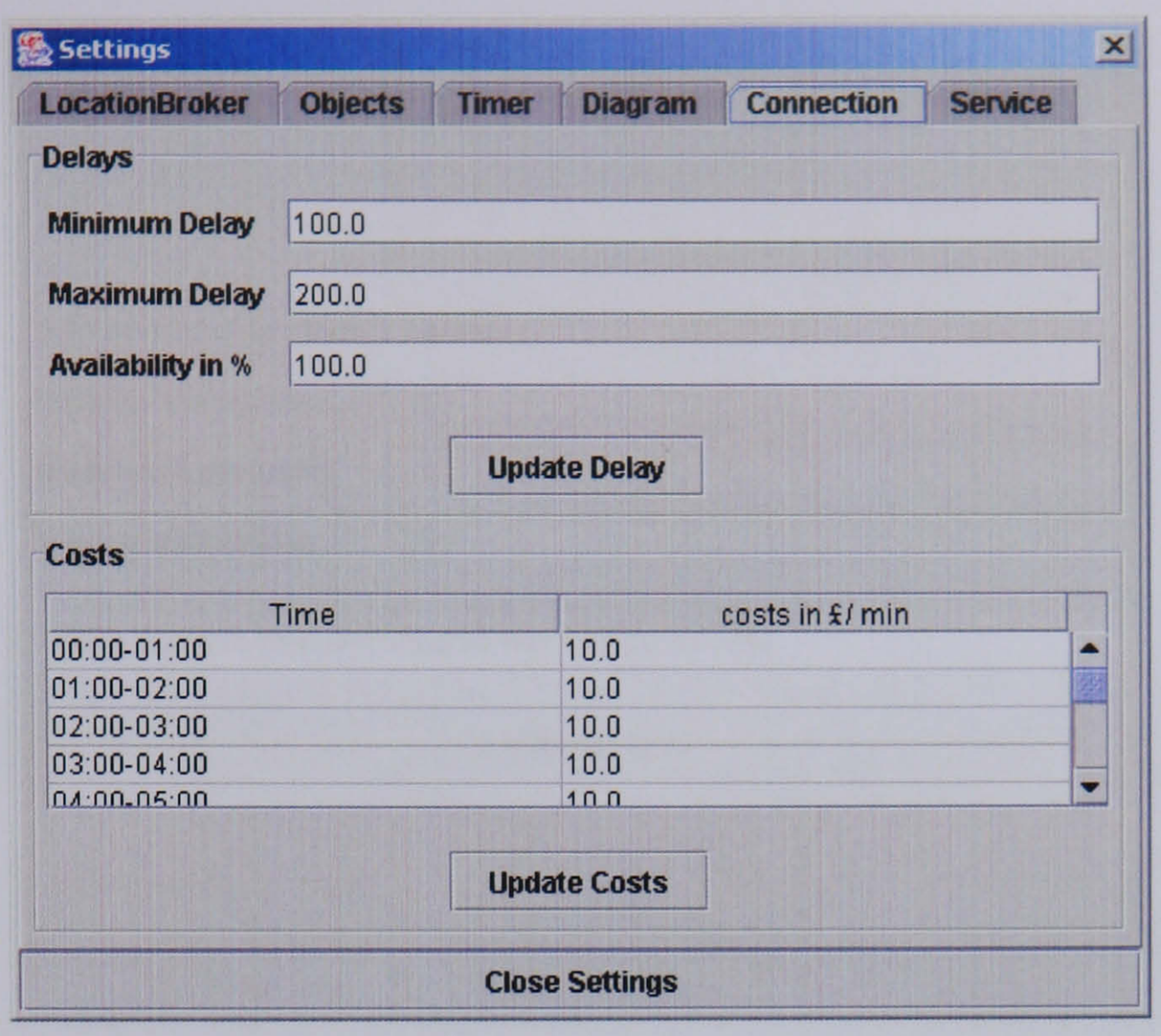


Figure B.10: Setting default values for connections

B.1.4.6 Tab Service

This panel (Figure B.11 allows the user to change the default values that are used when creating a new service.

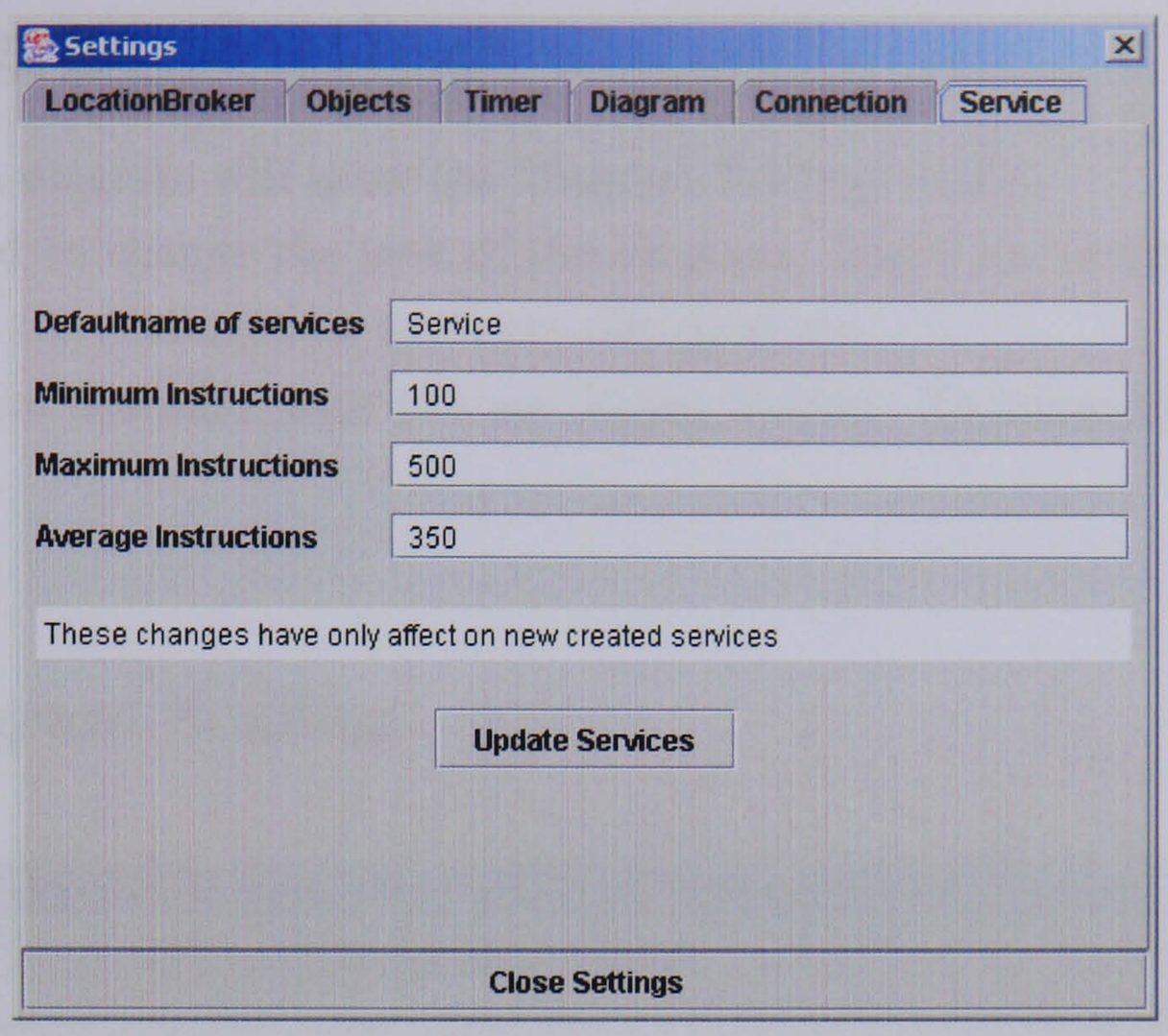


Figure B.11: Setting default values for services

The service name, the minimum, maximum and average instructions can be changed here.

B.2 Diagram Settings

A right click on a diagram will show the Diagram Settings dialog.
This panel is used to change the look of the diagram. There are several options for the axes, the colours, the scaling, how grids are displayed etc..
To use the changed settings, press the *use Values* button. To discard changes press the *cancel* button.

B.2.1 Diagram Scaling

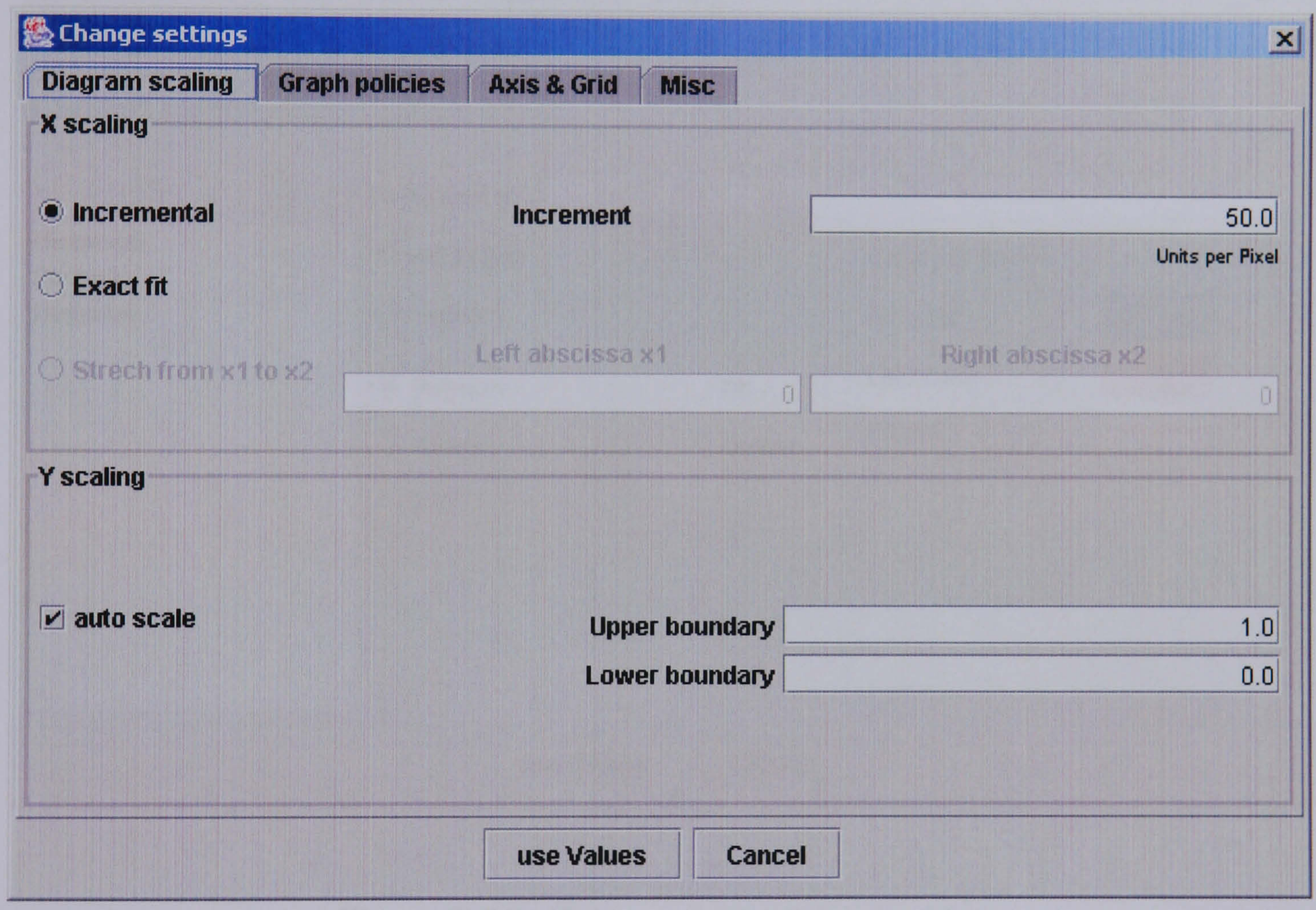


Figure B.12: Scaling values of the diagram

This pane allows to change the scaling of the axis.
The first part is for the x-axis. It can be choosen between the options *incremental* and *exact fit*.
Incremental means, that only the part is shown, which fits into the diagram panel size. How big this part is depends on the *increment* value. A value of 50 Units per pixel means

that each pixel of the diagram in x-orientation represents 50 milliseconds on the x-axis. Exact fit will show the whole data from the minimum x-value to the maximum x-value in the diagram.
The second part is for the y-axis. It can be selected whether to use auto scale, so that all values between the minimum y-value and the maximum y-value are shown, or specify the borders by hand.

B.2.2 Graph Policies

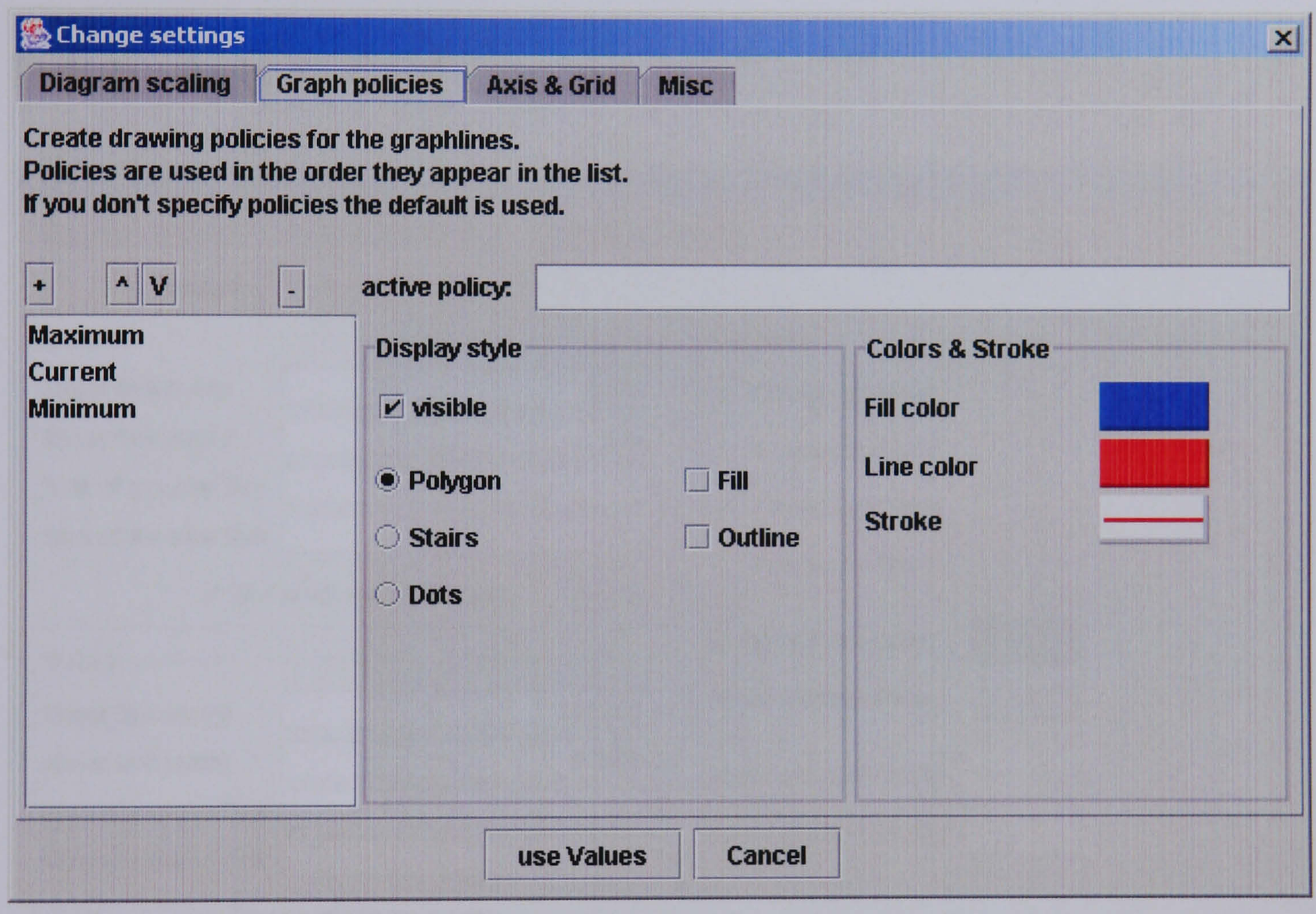


Figure B.13: Policy values of the diagram

Policies are rules how to draw the data of the diagram.
If no policy is specified the default values are used to draw the data.
The policies have an order, how they are used on a data set of the diagram.
A data set has one x-value but can have more then one y-value. In case of the speed diagrams we have three y-values. The first one is the maximum speed, the second the current speed and the third the minimum speed. The object diagrams have only one y-value which is the executed instructions as a percentage of the total instructions.

The first policy is used for the first y-value, the second policy for the second and so on.
A policy has a display style how to draw the data. There it is possible to choose between polygons, stairs and dots (dots is not implemented up to now). Additional attributes are the filling of the stair/polygon and/or outlining it. If a y-value should be shown or not can be toggled by the *visible* check box.
Furthermore the fill- and line-colour and the width of lines can be specified by clicking on the colour or stroke field. This will open a colour chooser where you can choose the new colour, or opens a dialog to enter the width of the stroke.

B.2.3 Axis & Grid

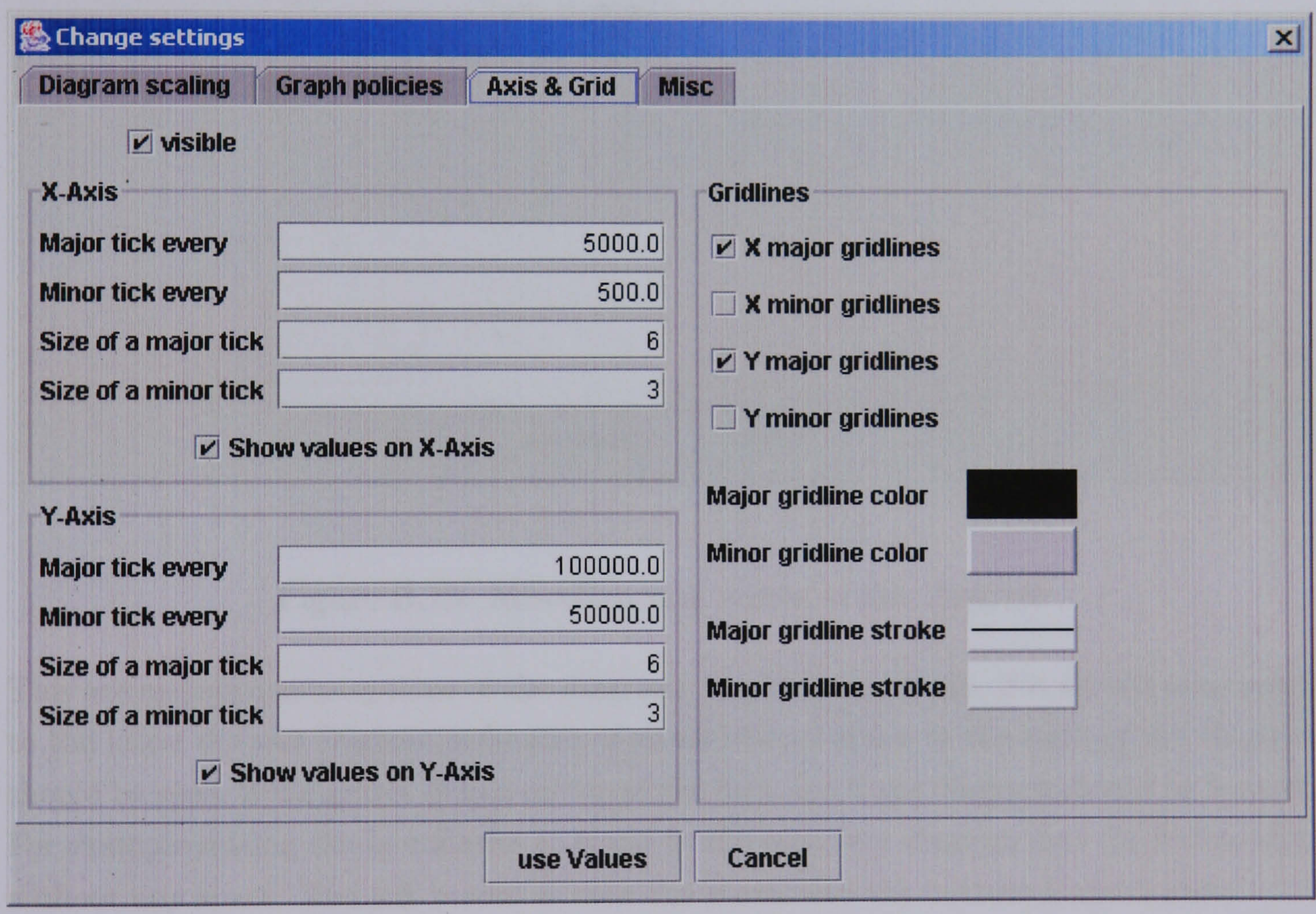


Figure B.14: Axis and grid values of the diagram

Options related to the axes and grid.
The major and minor ticks on the axes, the size of the ticks (in pixel) and if labels are displayed can be specified.
Grid lines can be switched on or off and the colour and line width can be chosen.

B.2.4 Misc

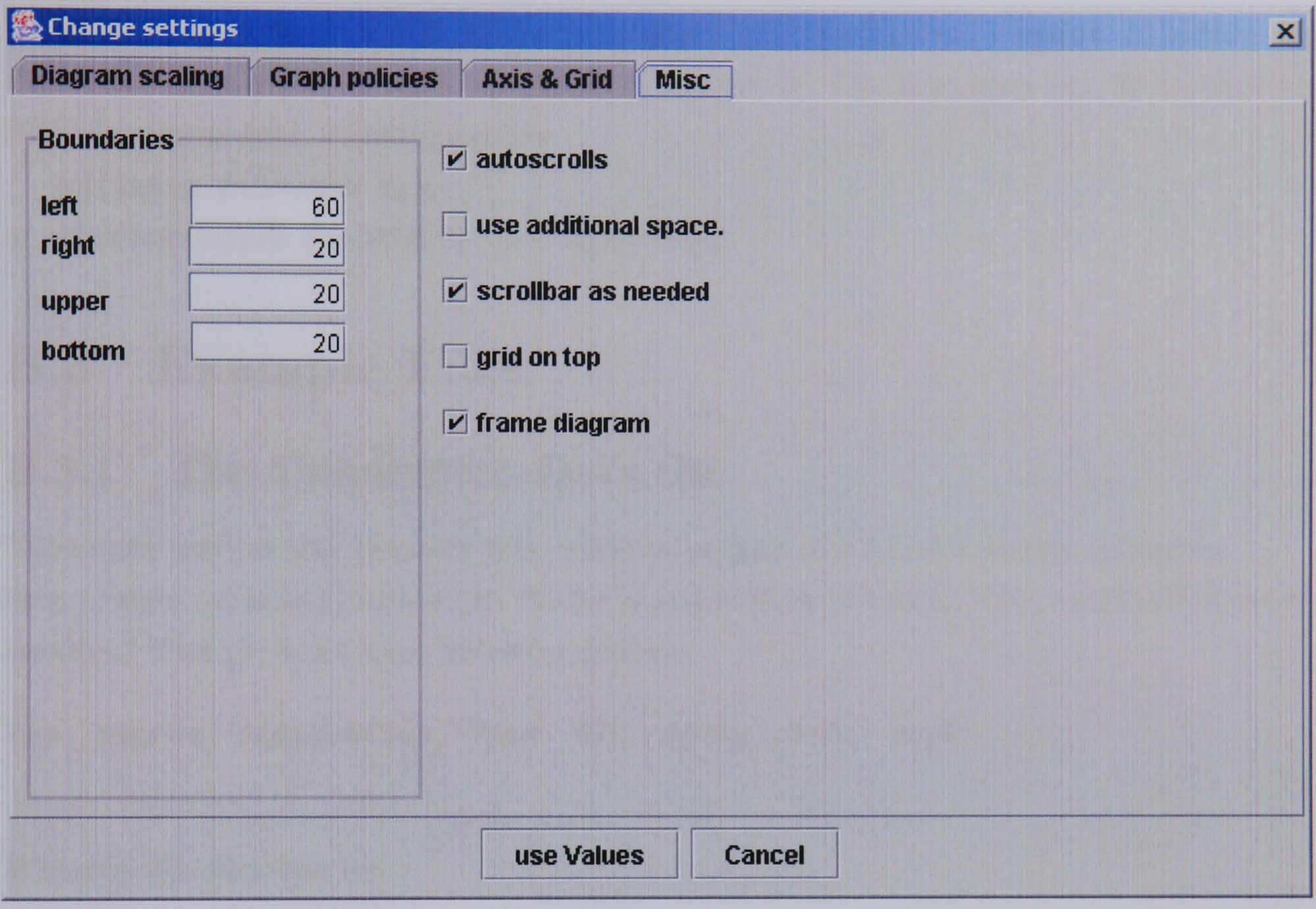


Figure B.15: Miscellaneous values of the diagram

This are miscellaneous options of the diagrams like the boundaries, if it should auto scroll to the latest dataset (highest ordinate), if an additional space to the right of the diagram should be used, if the grid is drawn on top of the data and if the diagram should be framed. For example setting the boundaries to 0 will fit the complete diagram into the visible size, without any space. The left border is then the y-axis and the bottom is the x-axis.
If you want to examine data that was long ago in a speed diagram, switching auto scrolls to off is a good idea.

B.2.5 MetaPost and TEX

When exporting the object diagram three files are created. One datafile storing the times and percentage values of the executed instruction, as well as a table showing when which

object used which machine. The created *MetaPost-* and TEX-file have the same name as the datafile where only the extension *.mp* and *.tex* is added. If your system has a version of TEXinstalled, it is common that *MetaPost* is installed as well. In the first step the *MetaPost* file has to be executed with the command for *MetaPost.* This can be

`mp filename.mp` or `mpost filename.mp`

This will create a PS-file with the name `filename.0`. The next step can be to create a PDF-document with a command like

`pdflatex filename.tex`

or other commands specified by your TEXversion.

B.3 Example Files

B.3.1 The Connection-Data file

"The data used in this research was collected as part of CAIDA's skitter initiative, http://www.caida.org Support for skitter is provided by DARPA, NSF, and CAIDA membership." The file must have following format:

```
Key  Source  Destination  Time  RTT  Count  hop1  hop2  ...
```

Whereby the Key can be:

C Complete The destination and all intermediate hops in the path all replied. The RTT to the destination is valid.

I Incomplete skitter got a reply from the destination, but did not receive a reply from every intermediate hop on the path. The RTT to the destination is valid.

N No reply no reply was received from the destination although a partial path may have been recorded. The RTT has no meaning in this case.

The source and destination are IP addresses or host names.
The time is a UNIX time stamp in milliseconds from 01/01/1970.
The RTT is the round trip time in milliseconds.
Count specifies how many hops where in between and then the hops are following.
For importing the data, only the lines that have a *C* or *I* as key are used and is creating connections between source and destination address with the min- and max- delay based on the RTT.

A short example of a connection data file:

```
# KEY SOURCE          DESTINATION    TIME       RTT     COUNT HOPS ...
reading: ../skitterData/result/a-root.20020315 and look for IP: 203.181.248.27
C     216.168.227.250 203.181.248.27 1016156250 216.874 14    216.168.227.1 ...
C     216.168.227.250 203.181.248.27 1016166239 212.155 14    216.168.227.1 ...
C     216.168.227.250 203.181.248.27 1016176226 210.753 14    216.168.227.1 ...
C     216.168.227.250 203.181.248.27 1016186235 213.538 14    216.168.227.1 ...
C     216.168.227.250 203.181.248.27 1016196240 210.757 14    216.168.227.1 ...
C     216.168.227.250 203.181.248.27 1016206265 236.666 14    216.168.227.1 ...
C     216.168.227.250 203.181.248.27 1016216280 236.928 14    216.168.227.1 ...
C     216.168.227.250 203.181.248.27 1016226293 236.764 14    216.168.227.1 ...
C     216.168.227.250 203.181.248.27 1016236299 236.849 14    216.168.227.1 ...
reading: ../skitterData/result/caida.mae.net.20011231 and look for IP: 203.181.248.27
C     204.29.239.23   203.181.248.27 1009775669 247.176 17    204.29.239.1 ...
C     204.29.239.23   203.181.248.27 1009838024 247.240 17    204.29.239.1 ...
reading: ../skitterData/result/champagne.20020315 and look for IP: 203.181.248.27
C    141.142.121.4    203.181.248.27 1016151311 189.608 9     141.142.121.254 ...
```

B.3.2 The Replacement file

A short example of a replacement file:

```
# ------------------------------------------------------------
# This is a replacement file.
# Format must be:
# original = replacement
# Example:
# 127.0.0.1 = nemesis.dmu.ac.uk
# The Ip-adress 127.0.0.1 will be replaced to nemesis.dmu.ac.uk
#
#  Ottawa, CA - 205.189.33.78
#  College Park, MD, US (University of Maryland) - 128.8.7.4
#  Herndon, VA, US ( Verisign) - 216.168.227.250
#  Tokyo, Kanto, JP (APAN) - 203.181.248.27
#  Amsterdam, North Holland, NL (RIPE) - 193.0.0.11
#  Eugene, OR, USA (University of Oregon) - 128.223.220.56
# ------------------------------------------------------------

128.8.7.4 = forquet.cse.dmu.ac.uk
216.168.227.250 = ossi
128.223.220.56 = liverpool.dmu.ac.uk
205.189.33.78 = buda
203.181.248.27 = garfield.dmu.ac.uk
193.0.0.11 = sienna.dmu.ac.uk
```

Appendix C

Optimization Software

To test the algorithm and their behaviour we have designed and implemented a graphical user interface, which allows to draw graphs quickly. The source code is written in Java and requires the JRE1.4. An initial distribution has to be specified for the designed graph. This distribution can then be optimised using the algorithm described in this paper.

Additionally to an extensive logfile it is possible to export the data gathered during the optimisation run into a MetaPost [20] file, which can be easily included in TEX-documents. It is also possible to export the graph into a MetaPost. Figure C.1 shows the graphical user interface of the test application.

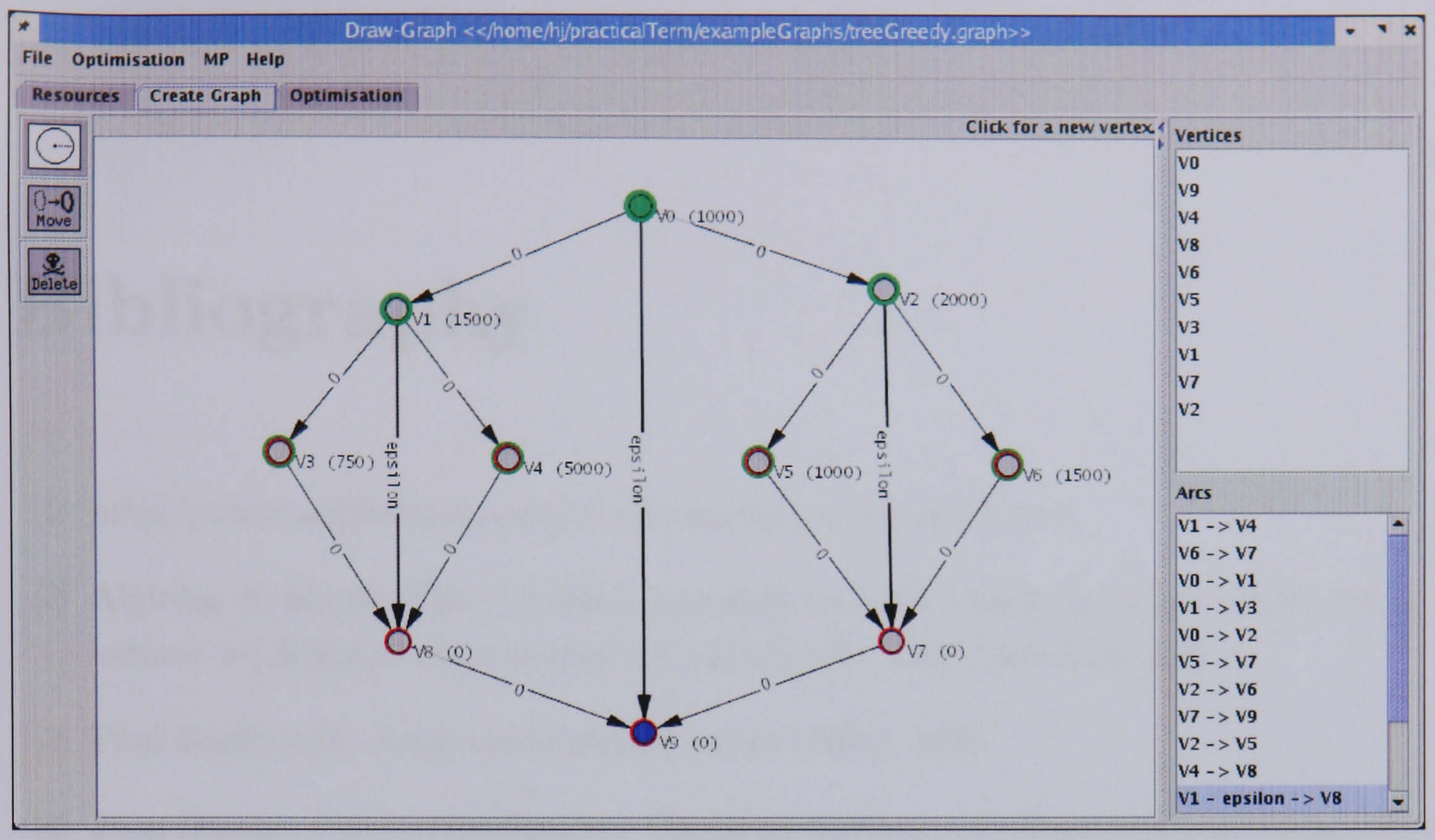


Figure C.1: Screen shot of the test application

Bibliography

[1] http://www.atmformum.com/atmforum/specs/approved.html.

[2] Algirdas Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[3] Paul Bachmann. Analytische zahlentheorie (1894), 1894.

[4] Jean Bacon. *Concurrent Systems Operating Systems, database and Distributed Systems: An Integrated Approach.* Addison Wesley, second edition edition, 1998.

[5] Sung Hoon Baek, Bong Wan Kim, Eui Joung Joung, and Chong Won Park. Reliability and performance of hierarchical RAID with multiple controllers. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 246–254. ACM Press, 2001.

[6] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *Proceedings of the 11th IEEE Real-Time System symposium.* IEEE Press, 1990.

[7] Booch, Jacobson, and Rumbaugh. *Using UML Software Engineering with Objects and Components.* Addison Wesley, updated edition edition, 2000.

[8] http://www.caida.org/tools/measurement/skitter/sample_code/.

[9] D. Coudert and X. Muoz. How graph theory can help communications engineering. citeseer.nj.nec.com/481792.html.

[10] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design.* Addison Wesley, 3rd edition edition, 2001. ISBN 0201-61918-0.

[11] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. Ponder: A language for specifying security and management policies for distributed systems - the language specification. Technical report, Imperial College Research Report DoC 2000/1, October 2000.

[12] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–39, January 2001.

[13] Nicodemos C. Damianou. *A Policy Framework for Management of Distributed Systems.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, Department of Computing, February 2002.

[14] Daniel S. Diamond and Lee L. Selwyn. Considerations for computer utility pricing policies. In *Proceedings of the 1968 23rd ACM national conference*, pages 189–200. ACM Press, 1968.

[15] Chitra Dorai, Parviz Kermani, and Avare Stewart. ELM-N E-Learning media navigator. In *ACM Proceedings of the International Conference on Multimedia*, September 2001.

[16] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[17] Bruce Eckel. *Thinking in Java.* Prentice Hall PTR, 3rd edition edition, 2002. ISBN 0131002872.

[18] William H. Graves. The new challenges of E-Learning. *UBIQUITY ACM IT Magazine and Forum*, 1(43), January 2001.

[19] http://www.etsi.org/.

[20] John D. Hobby. Drawing graphs with metapost. Computing Science Technical Report no. 164, AT&T Bell Laboratories, Murray Hill, New Jersey, 1993.

[21] Christine R. Hofmeister and James M.Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Proceedings of the 13th International Conference on Distributed Computing Systems, IEEE Computer Society Press*, May 1993.

[22] http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html.

[23] http://www.borland.com/jbuilder/.

[24] Mathai Joseph, editor. *Real-Time Systems Specification, Verification and Analysis.* Prentice Hall International (UK) LTD, 1996. ISBN 0-13-455297-0.

[25] A. King and R. Hunt. Protocols and architecture for managing TCP/IP network infrastructures. *Computer Communications*, 23(16):1558–1572, September 2000.

[26] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[27] Leslie Lamport. Synchronizing time servers. Technical report, Systems Research Center Palo Alto, California 94301, June 1987.

[28] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1994.

[29] J. Lehoczky, L. Sha, and Y. Ding. The rate-monotonic scheduling algorithm: exact characterisation and average case behavior. In *Proceedings of the 10th IEEE Real/-Time System Symposium, IEEE Computer Society Press*, 1989.

[30] http://rpmfind.net/linux/rpm2html/mirroring.html.

[31] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[32] Paul McKee. Enabling distributed services. Presented to CDS Club meeting, Topics in Distributed Systems, December 1998.

[33] http://www.ibiblio.org/pub/Linux/LSMTEMPLATE.html.

[34] David L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, 39(10), October 1991.

[35] Ike. Nassi and Ben Shneiderman. Flowchart techniques for structured programming, August 1973.

[36] Neumann. Monitoring and controlling suspicious activity in real-time with IP-Watcher. In *IEEE 11th Annual Computer Security Applications Conference*, 1995.

[37] Jr. Paul K. Harter. Response times in level-structured systems. *ACM Transactions on Computer Systems (TOCS)*, 5(3):232–248, 1987.

[38] Amelia Platt. Final report on CORBA, internal report submitted to mobility campaign group. British Telecom, 1996/97.

[39] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.

[40] Akos L. Redey. *A novel Routing Strategy for Public, Wide Area ATM Networks.* PhD thesis, De Montfort University, 1997.

[41] Klaus Schmaranz. DOLSA - a robust algorithm for massively distributed. dynamic object-lookup services. submitted to J.UCS, 2002.

[42] Jon Siegel. *CORBA 3 Fundamentals and Programming.* John Wiley and Sons, Inc, second edition edition, 2000.

[43] Zaw-Sing Su and Jon Postel. The domain naming convention for internet user applications, August 1982.

[44] Andrew S. Tanenbaum and Maarten von Steen. *Distributed Systems (Principles and Paradigms).* Prentice Hall, 2002. ISBN 0-13-088893-1.

[45] http://www.w3.org/XML.

[46] Hussein. S. M. Zedan. *Distributed Computer Systems*, chapter Reliable systems in occam, pages 132–148. Butterworths, 1990.