PhD THESIS

submitted by

Pushpinder Kaur CHOUHAN

to obtain the degree of

Doctor of Philosophy in Computer Science

at the

École Normale Supérieure de Lyon

Automatic Deployment for Application Service Provider Environments

PhD defense date: *28 September 2006*

Thesis Committee :

President	Mr.	Jean François	MÉHAUT
Reporter	Mr.	Daniel	HAGIMONT
	Mr.	Christian	PEREZ
Examiner	Mr.	John	MORRISON
Director	Mr.	Eddy	CARON
	Mr.	Frédéric	DESPREZ

Contents

1 Introduction **3**

I Context of the study **9**

2 Network Enabled Server Environments **11**
- 2.1 Introduction . . . 11
- 2.2 Other environments . . . 12
- 2.3 Characteristics comparison of NES environments . . . 13
  - 2.3.1 Development . . . 13
  - 2.3.2 Architecture . . . 15
  - 2.3.3 Initialization . . . 20
  - 2.3.4 Task Execution . . . 21
  - 2.3.5 Communication . . . 25
  - 2.3.6 Scheduling and Dynamic Load Balancing . . . 27
  - 2.3.7 Fault Tolerance . . . 29
  - 2.3.8 Data management . . . 30
  - 2.3.9 Security . . . 32
  - 2.3.10 Deployment and Visualization Tools . . . 33
- 2.4 Comparison of Systems . . . 34
- 2.5 Conclusion . . . 36

3 Deadline Scheduling with Priority of tasks on NES **39**
- 3.1 Introduction . . . 39
- 3.2 Related work . . . 40
- 3.3 Scheduling algorithms for NES environments . . . 41
  - 3.3.1 Client-server scheduler with load measurements . . . 41
  - 3.3.2 Client-server scheduler with a forecast correction mechanism . . . 42
  - 3.3.3 Client-server scheduler with a priority mechanism . . . 43
- 3.4 Simulation results . . . 45
- 3.5 Conclusions . . . 46

4 Deployment **49**
4.1 Introduction 49
4.2 Software deployment 50
4.2.1 Automatic Deployment of Applications in a Grid Environment . 51
4.2.2 JXTA Distributed Framework 51
4.2.3 Pegasus 52
4.2.4 Sekitei 53
4.2.5 SmartFrog 55
4.2.6 Software Dock 55
4.3 System deployment 57
4.3.1 Dell OpenManage Deployment Toolkit 58
4.3.2 Kadeploy 58
4.3.3 Warewulf 59
4.4 Conclusion 60

II Systems used to validate the work **63**

5 DIET **65**
5.1 Introduction to DIET 65
5.1.1 DIET design principles 66
5.1.2 Architecture of DIET 66
5.1.3 Deployment of DIET 67
5.1.4 Task execution in DIET 68
5.1.5 DIET distributed scheduling 68
5.1.6 Data management in DIET 69
5.1.7 Fault tolerance in DIET 70
5.2 DIET tools 70
5.2.1 GoDIET 71
5.2.2 LogService 71
5.2.3 VizDIET 72
5.2.4 Fast Agent's System Timer 73
5.3 Conclusion 74

6 GoDIET **75**
6.1 Working of GoDIET 75
6.2 Identification of failure in element launch 77
6.3 XML file as an input 78
6.4 Evaluation of performance and efficacy 80
6.4.1 Evaluation of launch performance 81
6.4.2 Launch problem identification 81
6.4.3 DIET deployment on 585 nodes 84
6.5 Conclusion 84

III Middleware Deployment Planning **85**

7 Heuristic to Structure Hierarchical Scheduler **87**
7.1 Introduction . . . 87
7.1.1 Operating models . . . 88
7.2 Architectural model . . . 89
7.3 Deployment constraints . . . 90
7.4 Deployment construction . . . 91
7.5 Model implementation for DIET . . . 92
7.6 Experimental results . . . 94
7.6.1 Experimental design . . . 94
7.6.2 Performance model validation . . . 95
7.6.3 Deployment selection validation . . . 98
7.7 Conclusion . . . 99

8 Automatic Middleware Deployment Planning on Clusters **101**
8.1 Platform deployment . . . 102
8.1.1 Platform architecture . . . 102
8.1.2 Optimal deployment . . . 104
8.1.3 Deployment construction . . . 107
8.1.4 Request performance modeling . . . 107
8.2 Steady-state throughput modeling . . . 110
8.3 Experimental results . . . 110
8.3.1 Experimental design . . . 111
8.3.2 Model parametrization . . . 111
8.3.3 Throughput model validation . . . 112
8.3.4 Deployment selection validation . . . 114
8.3.5 Validation of model for mix workload . . . 117
8.4 Model forecasts . . . 118
8.5 Conclusion . . . 118

9 Automatic Middleware Deployment Planning for Grid **121**
9.1 Platform Deployment . . . 121
9.2 Heuristic for middleware deployment on heterogeneous resources . . . 122
9.3 Request performance modeling . . . 124
9.4 Steady-state throughput modeling . . . 127
9.5 Experimental Results . . . 128
9.5.1 Experimental Design . . . 128
9.5.2 Model Parametrization . . . 128
9.5.3 Performance model validation on homogeneous platform . . . 130
9.5.4 Heuristic Validation on heterogeneous cluster . . . 132
9.6 Conclusion . . . 133

10 Improve Throughput of a Deployed Hierarchical NES **135**
10.1 Hierarchical deployment model . . . 135
10.2 Throughput calculation of an hierarchical NES . . . 136
10.2.1 Find a bottleneck . . . 137
10.2.2 Remove the bottleneck . . . 138
10.3 Parameter measurement . . . 139
10.4 Simulation results . . . 139
10.4.1 Test-bed . . . 142
10.4.2 Computing a good deployment . . . 143
10.5 Conclusion . . . 144

11 Automatic Deployment Planning Tool **147**
11.1 Introduction . . . 147
11.2 Formulation of performance model for a middleware . . . 148
11.3 Working of ADePT . . . 152
11.4 ADePT as a deployment planner for ADAGE . . . 152
11.5 Conclusion . . . 153

IV Conclusion and Future work **155**

12 Conclusions **157**

A Bibliography **161**

List of Figures

1.1 Some of the possible deployments with 150 nodes . . . 5
1.2 Comparison of requests completed by a centralized DIET scheduler versus a three agent distributed DIET scheduler. . . . 6

2.1 The NeOS architecture . . . 15
2.2 The NetSolve architecture . . . 16
2.3 Architecture of Nimrod-G . . . 16
2.4 Architecture of Ninf . . . 17
2.5 Architecture of PUNCH . . . 18
2.6 Architecture of WebCom-G. (a)General view of WebCom-G. (b) A minimal WebCom installation consists of a Backplane module, a Communication Manager module and number of module stubs for processing, Fault Tolerance, Load Balancing and Security. . . . 19
2.7 NetSolve initialization steps . . . 20
2.8 NetSolve task execution steps . . . 22
2.9 Nimrod: Work procedure . . . 23
2.10 Working of Ninf . . . 23
2.11 Working of PUNCH . . . 24
2.12 Working of WebCom-G . . . 25
2.13 Demand-based Scheduling [54] . . . 28
2.14 Overview of visPerf Monitoring System [62] . . . 34
2.15 Description: WebCom-G Integrated Development Environment . . . 35

3.1 Example for priority scheduling algorithm with fallback mechanism. Task id and execution time is written diagonally in each box. . . . 45
3.2 Priority based tasks are executed without fallback mechanism. . . . 46
3.3 Comparison of tasks executed with and without fallback based on tasks priority. . . . 47
3.4 Comparison of tasks executed with and without fallback based on tasks execution duration. . . . 48

4.1 Components of a workflow generation, mapping and execution system. 53
4.2 Process flow graph for solving CPP [58]. . . . 54
4.3 Sekitei algorithm phases [55]. . . . 54
4.4 Architecture of Software Dock [50]. . . . 56

4.5 Comparison of some deployment tools 57

5.1 Architecture of DIET. 67
5.2 Launch of a DIET platform. 68
5.3 DIET task execution steps. 69
5.4 Interaction of GoDIET LogService, and VizDIET assist users in controlling and understanding DIET platform. 71
5.5 Screen shot of VizDiet. 72
5.6 Overview of the FAST architecture. 73

6.1 Working steps of GoDIET. 76
6.2 An example of GoDIET XML file. 79
6.3 The time for platform launch as a function of the number of servers desired. 82
6.4 Grid'5000 DIET Deployment with 1 MA, 8 LAs, 574 SeDs. 83

7.1 Classification of the operating models. 89
7.2 Star hierarchies with one or two servers for DGEMM 150x150 requests. (a) Real-world platform throughput for different load levels. (b) Comparison of predicted and actual maximum throughput. 95
7.3 Star hierarchies with one or two servers for DGEMM 10x10 requests. (a) Throughput at different load levels. (b) Comparison of predicted and actual maximum throughput. 96
7.4 Star hierarchies with two servers using a Gb/s or a 100 Mb/s network. Workload was DGEMM 10x10. (a) Throughput at different load levels. (b) Comparison of predicted and actual maximum throughput. 96
7.5 Comparison of automatically-generated hierarchy with hierarchies containing twice as many and half as many servers. 97
7.6 Types of platforms compared. 98
7.7 Comparison of automatically-generated hierarchy with intuitive alternative hierarchies. 98

8.1 Platform deployment architecture and execution phases. 102
8.2 Deployment trees of $dMax$ sets 4 and 6. 105
8.3 Example DIET deployments consisting of MAs, LAs, and SeDs (unlabelled circles). 108
8.4 Measured and predicted platform throughput for DGEMM size 10; predictions are shown for several bandwidths. 113
8.5 Measured and predicted platform throughput for DGEMM size 1000; predictions are shown for the serial model with bandwidth 190 Mb/s. . . 114
8.6 Predicted and measured throughput for different CSD trees for DGEMM 200 with 25 available nodes in the Lyon cluster. 115
8.7 Predicted and measured throughput for different CSD trees for DGEMM 200 with 45 available nodes in the Sophia cluster. 116

8.8 Predicted and measured throughput for different CSD trees for DGEMM 310 with 45 available nodes in the Sophia cluster. 116
8.9 Makespan for a group of tasks partitioned to three deployments or sent to a single joint deployment. 118

9.1 Explanation of workload introduced by submitting requests with the increase in the launch of clients on each machine. 129
9.2 Star hierarchies with one or two servers for DGEMM 10x10 requests. (a) Measured throughput for different load levels. (b) Comparison of predicted and measured maximum throughput 130
9.3 Star hierarchies with one or two servers for DGEMM 200x200 requests. (a) Measured throughput for different load levels. (b) Comparison of predicted and measured maximum throughput 131
9.4 Comparison of automatically-generated hierarchy for DGEMM 310*310 with intuitive alternative hierarchies. 132
9.5 Comparison of automatically-generated hierarchy for DGEMM 1000* 1000 with intuitive alternative hierarchy. 133

10.1 Architectural model. 136
10.2 Performance calculation by adding LAs. 140
10.3 Performance calculation by adding SeDs. 141
10.4 High speed network (2.5Gb/s) between INRIA research centers and several other research institutes. 142
10.5 Diagrammatic view of testbed. 142
10.6 Throughput of heterogeneous network as more number of LA are added. 144
10.7 Throughput of heterogeneous network as SeD are converted to LA. . . . 145

11.1 Working of Automatic Deployment Planning Tool (ADePT). 149
11.2 An example of resource description XML file. 150
11.3 An example of application description XML file. 151
11.4 ADePT as a deployment planner for ADAGE. 153

List of Tables

2.1 Table of Comparison 37

3.1 Priority, deadline and computation time of each task. 44

5.1 Example of a plug in scheduler parametric table. 69

6.1 Description of the Grid'5000 clusters used in our experiments. 80
6.2 Problem SeDs as identified by GoDIET / by clients. 82
6.3 Time taken to launch 574 SeDs is 592 secs. 84

8.1 Parameter values for middleware deployment on cluster 112
8.2 A summary of the percentage of optimal achieved by the deployment selected by our model, a star deployment, and a tri-ary tree deployment. 117
8.3 Predictions for the best degree d, number of agents used $|\mathbb{A}|$, and number of servers used $|\mathbb{S}|$ for different DGEMM problem sizes and platform sizes $|\mathbb{V}|$. The platforms are assumed to be larger clusters with the same machine and network characteristics as the Lyon cluster. 119

9.1 A summary of notations used to define platfrom deployment. 122
9.2 A summary of notations used to define performance model. 126
9.3 Parameter values for middleware deployment on Lyon site of Grid'5000 129
9.4 A summary of the percentage of optimal achieved by the deployment selected by our heterogeneous heuristic, optimal degree, and optimal homogeneous model. 131

10.1 Parameter values to calculate the throughput of a deployment. 142

Chapter 1

Introduction

Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and user's quality-of-service requirements. Applications that are submitted to grid are special class of distributed applications that have high computing and resource requirements.

Grid platforms are very promising but are very challenging to use because of their intrinsic heterogeneity in terms of hardware capacities, software environment and even system administrator orientations. To facilitate the use of grid platform, an extension of classical approach Remote Procedure Call (RPC) is used, called "GridRPC". Environments based on this concept are a subset of Network Enabled Server (NES) environments [65]. End-users submit their application to grid platform through NES environments as these environments hide all the complexities of the grid platform from users by providing easy access to the grid resources. Numerous NES environments already exist. We present a survey of the most prevalent NES environments (NeOS [11], NetSolve [9], Nimrod [3], Ninf [70], PUNCH [53], and WebCom [67]) in Chapter 2 .

Basically NES environment have five components. *Clients* provides user interface and submit requests to execute libraries or applications to servers. *Servers* receives requests from clients and execute libraries or applications on their behalf. The *Database* contains the status (dynamic and static information) of the monitored resources. *Monitors* dynamically store and maintains the status of the available computational resources in the database. The *Scheduler* selects a potential server from the list of servers maintained in the database by frequent monitoring and maps client requests to that server.

The main objective of the thesis is to improve the performance of a NES environment so as to use these environments efficiently. Here efficiency means the maximum number of completed requests that can be treated in a time step by these environments. A *completed request* is one for which a response has been returned to the client. We calculate the environments efficiency in terms of throughput (number of completed requests per time unit), because the traditional objective of makespan minimisation is NP-hard in most practical situations [76], [12]. Instead of absolute minimisation of

the execution time, we look at asymptotic optimality by searching an optimal steady state scheduling techniques. In steady-state scheduling techniques [16], startup and shutdown phases are not considered and the precise ordering and allocation of tasks and messages are not required. Instead, the main goal is to characterize the *average* activities and capacities of each resource during each time unit.

To achieve the thesis objective, the main goal is to remove the obstacle that cause the hindrance in the efficiency of the environments. The very first problem which comes into picture is related to the applications scheduling on the selected servers. NES environments are able to find the computing power and the capacity storage necessary for the execution of an application, but it remains to determine a scheduling that offers the greatest possible effectiveness on the servers. NES environment generally use the Minimum Completion Time (MCT) on-line scheduling algorithm where-by all applications are scheduled immediately or refused. This approach can overload interactive servers in high load conditions and does not allow adaptation of the schedule to task dependencies. Thus, we first studied the scheduling techniques that can be adopted for scheduling tasks on NES environment. As a result, we present algorithms for the scheduling of the sequential tasks on a NES environment in Chapter 3. We mainly discuss a deadline scheduling with priority strategy that is more appropriate for multi-client, multi-server scenario.

Even though already some methods and technologies have been added to the basic NES environments to improve the performance like development of good performance forecasting tools, hierarchical arrangements of NES's components, etc. But still there are some important fields that can be improved to increase the efficiency of the NES environments as stated above, by improving the scheduling techniques used at server level to schedule the assigned tasks.

Another important factor that influence the efficiency of the NES environments is the mapping style of the environment's components on the available resources. Generally components of these environments are mapped on the available resources as defined by the user or environment's administrator. We call this process as *deployment*. Their exist some deployment tools, that deploy the NES's components on selected resources. In Chapter 4 we present a survey of some deployment tools.

To show that the efficiency of the environments depends on the arrangements of their components on the available resources (nodes), we did experiments. For experiments, we used an hierarchical Network Enabled Server called DIET (**D**istributed **I**nteractive **E**ngineering **T**oolbox). The DIET scheduler can be hierarchically distributed or used in a centralized fashion. Detailed overview of DIET is provided in Chapter 5. Figure 1.2 shows the experimental results performed with DIET. These experiments were performed using 151 nodes of the Orsay cluster of Grid'5000, a set of distributed computational resources in France. Depending on the available number of nodes, numerous type of deployments are possible. For example, some of the possible deployments with 150 nodes are shown in Figure 1.1. For the experiment we tested two deployments (a and b) shown in Figure 1.1. In the first deployment, one node is dedicated to a centralized scheduler that is used to manage scheduling for the remaining 150 nodes, which are dedicated computational nodes servicing requests. In the sec-

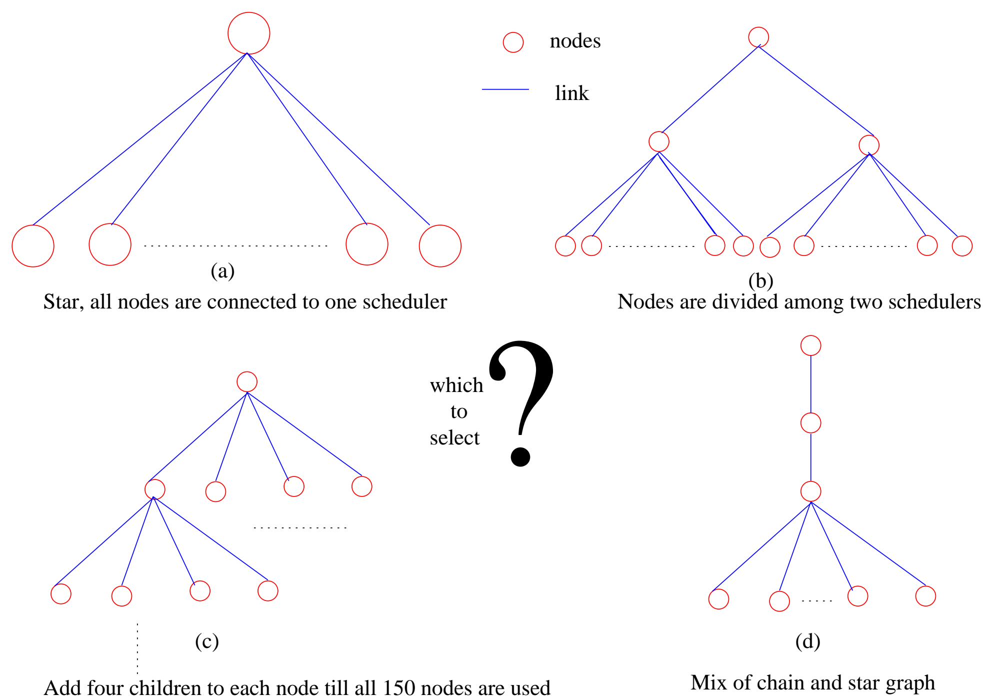


Figure 1.1: Some of the possible deployments with 150 nodes

ond deployment, three nodes are dedicated to scheduling and are used to manage scheduling for the remaining 148 nodes, which are dedicated to servicing computational requests. In this test the centralized scheduler is able to complete 22,867 requests in the allotted time of about 1400 seconds, while the hierarchical scheduler is able to complete 36,307 requests in the same amount of time.

The distributed configuration performed significantly better, despite the fact that two of the computational servers are dedicated to scheduling and are not available to service computational requests. At least for the DIET toolkit, and most likely for other schedulers as well, distributing the task of scheduling can improve performance in large resource environments. However, the optimal arrangement of schedulers is unknown. Should we dedicate four machines to scheduling in the above experiment? Perhaps ten? It is clearly impossible to test all possible arrangements for a given environment.

For the above explained experiments we used DIET deployment tool called GoDIET. And we gave the deployment hierarchy in the form of an XML file with all the required information like, which resource is connected to which, path of the binary files etc. Overview and performance evaluation of GoDIET tool is present in Chapter 6.

Like GoDIET, all existing deployment tools deploy the components on the given resources as defined by the users or NES's administrator, with all the required information. However, the deployment tools does not allow the physicists, chemists, biologists, etc to deploy middleware according to their applications simply on grids: the user who wants to deploy his application must be expertise in the field of selecting

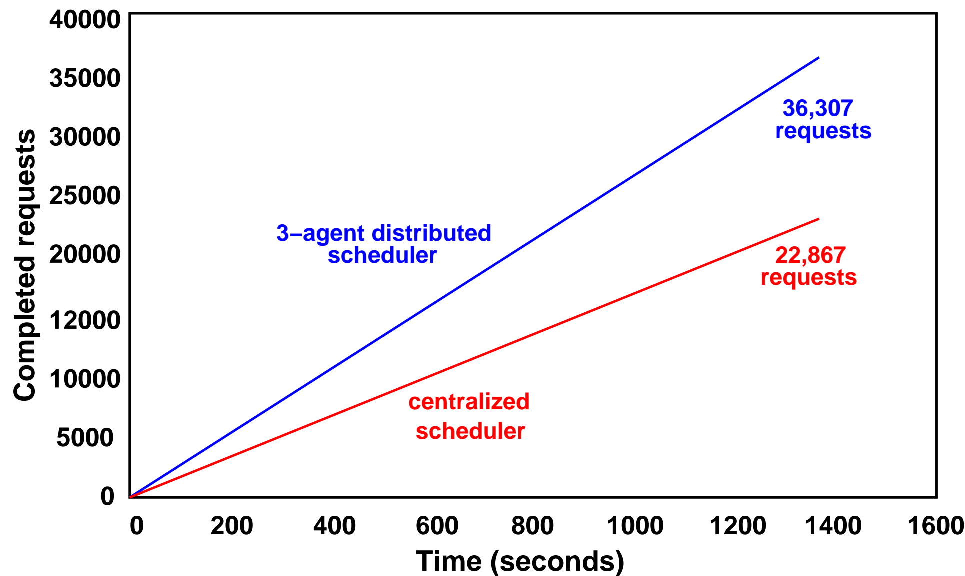


Figure 1.2: Comparison of requests completed by a centralized DIET scheduler versus a three agent distributed DIET scheduler.

the appropriate resources of the grids based on their application type, characteristics of middleware component and available resources. Moreover, the user should select the resources according to the application, should know how to carry out all the file transfers, implementation of the configuration files and the launching of remote tasks for each selected resource, and then finally submit the application to the deployed platform.

Even if, the deployment plan of NES's components is very important factor to influence the throughput of a NES environment, yet neither theoretical nor practical framework exist, to find the best organization of the components. Organizing the nodes in an efficient manner and using the nodes' computation power efficiently is out of the scope of end users. In recent years much work has been focused on the design and implementation of the NES environments but neither a tool nor any algorithms have been developed for an appropriate mapping of environment's components to the resources. Thus a planning is needed to arrange the resources in such a manner that when NES's components are deployed on the arranged resources, maximum number of requests can be processed in a time step by the NES environment. We called this planning as *deployment planning*. Even, environmental designers often note that deployment planning is an important problem but still enough work is not done for efficient and automatic deployment.

To the best of our knowledge, no deployment planning algorithm or model has been given for arranging the NES's components in an efficient way. The questions such as "which resources should be used?", "how many resources should be used?" and "should the fastest and connected resource be used for middleware or as a computational resource?" remained unanswered.

In this thesis we gave the solutions to these questions in Chapters 7, 8, 9, and 10. We also gave an initial idea to develop a tool based on our automatic deployment planning models in Chapter 11.

Part I

Context of the study

Chapter 2

Network Enabled Server Environments

2.1 Introduction

Among the different approaches for porting large scale applications on grids, the client-server computation model is at the same time efficient and simple. Network Enabled Server (NES) environments provide users with an easy access to distributed resources across the Internet. NES environments evolved only a decade ago, when programming paradigms and core middlewares such as ORB, RMI, RPC and web-services, laid a strong foundation for distributed resource connectivity. Some of them make use of a multiple tier structure in which clients, agents, and servers communicate efficiently and the most appropriate resource for a computation request is selected.

The main goal of NES environments is to provide access to the end users (biologists, mathematicians, astrophysicists, etc.) the computational facilities via potential servers to solve their problems. These problems are large (i.e., require numbers of CPU hours) and cannot be solved using a single machine. They also require multiple potential servers during an application lifetime, so as to find the result in appropriate time, or using some specific feature or library. NES environments provide access to high computing power and large storage capacity at low cost.

These environments are popular because they are easy to setup. They can be used in front of a single server or using supercomputers connected through the Internet. They are accessible via various methods (Web browsers, C, C++, Fortran or Java programs, libraries, Problem Solving environments such as Matlab or Scilab, ...).

Theoretically all the NES environments have the same goal, but they are many and differ according to the underlying models they adopt and the features they provide. In this chapter six NES environments are presented in a comparative manner. These environments are NeOS, NetSolve, Ninf, Nimrod, PUNCH, and WebCom. Another NES environment called DIET (Distributed Interactive Engineering Toolbox), which we use for validating our theoretical concepts, is elaborately presented in Chapter 5.

2.2 Other environments

In addition to the NES environments presented in this chapter, other middleware infrastructures are available to execute remote jobs over the grid. Some of them are based on cycle steeling concept unlike NES environments, which are dedicated servers. Some systems that are dedicated but does not provide scheduling system, job management services, queueing mechanisms as provided by NES environments. In this section a small overview of the most common job management systems and most common grid systems is given for the completeness.

Condor[1] is a job management system. Although Condor provide a scheduling system, job management services, queueing mechanism, it is based on cycle steeling concept unlike NES environments, which use servers that are dedicated to some specific applications. Condor [82] is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, which in turn places them into a queue, chooses when and where to run the jobs based following a policy, carefully monitors their progress, and ultimately informs the user upon completion. Condor is the product of the Condor Research Project at the University of Wisconsin-Madison. Most flavors of Unix are supported, as well as Windows XP/2K. Condor also provides a platform for other systems to be built upon it using its core services.

XtremWeb [23] is a project developed at University of Paris-Sud, France. XtremWeb[2] is a software platform designed to serve as a substrate for global computing experiments. XtremWeb uses CPU idle time for task execution. XtremWeb is intended to distribute applications over a set of hosts using cycle stealing scheme and particularly focuses on multi-parameters applications which have to be computed several times with different inputs and/or parameters, each computation being fully independent from each others. Most flavors of Unix are supported, as well as Windows and MacOS-X.

Globus system is a dedicated grid system but does not provide scheduling system, job management services, queueing mechanisms as done by NES environments. Globus [45] is a grid toolkit to build computational grids and grid-based applications. The Globus Toolkit acts as a glue to integrate various resources (such as Desktop PC's, clusters, supercomputers, databases, visualization instruments, etc.) under different domains of interests called Virtual Organizations. *Virtual Organizations* are islands of domains where researchers are working under common interests. This toolkit is being developed by the Globus Alliance [3] and many other collaborators around the world. There are many systems built on top of Globus utilizing the services of Globus like Ninf-G, Nimrod-G, and Condor-G.

JiPANG (Jini-based Portal Augmenting Grids) [4] is a collaborative work with Indi-

[1] http://www.cs.wisc.edu/condor/
[2] http://www.lri.fr/ fedak/XtremWeb/
[3] http://www.globus.org/alliance/
[4] http://ninf.is.titech.ac.jp/jipang/

ana University, Electrotechnical Institute, Tokyo Institute of Technology, and University of Tennessee. JiPANG is a portal system and a toolkit which provides uniform access interface layer to a variety of grid systems, and is built on top of Jini distributed object technology. JiPANG performs uniform higher-level management of the computing services and resources being managed by individual grid systems such as Ninf, NetSolve, Globus, etc. In order to give the user a uniform interface to the grids JiPANG provides a set of simple Java APIs called the JiPANG Toolkits, and furthermore, allows the user to interact with grid systems, again in a uniform way, using the JiPANG browser application.

2.3 Characteristics comparison of NES environments

In this section we compare and contrast six NES environments according to their characteristics under different sections. The sequence of these NES environments is in a alphabetic order NeOS, NetSolve, Nimrod, Ninf, PUNCH, and WebCom. Some characteristics are not present in all NES environments that we are presenting, so the respective sections are not described.

2.3.1 Development

Development of NES environment has been done across the world for many years. These NES environments have been a research development of many research domains and are being developed using different programming technologies.

2.3.1.1 NeOS

Network-Enabled **O**ptimization **S**erver [41] is developed at the Argonne National Laboratory. The first version of NeOS Server was available in September 1995 and the latest version[5] is compatible with Win 9x/XP and all flavors of Unix. NeOS implementation and interfaces are written in Tcl/Tk, Java and Kestrel. NeOS is an environment for solving optimization problems over the Internet.

2.3.1.2 NetSolve

Network-enabled **Solve**r-based [29] system is developed at University of Tennessee, Knoxville. It [6] is available for all popular variants of the UNIX operating system, and part of the system are available for the Microsoft windows platforms. The first version was available in January 1996. The NetSolve system is implemented using the C programming language, with the exception of the thin upper layers of the client API that serve as environment specific interfaces to the NetSolve system. NetSolve is a client-server system that enables users to solve complex problems remotely.

[5] http://www-NeOS.mcs.anl.gov/NeOS/
[6] http://www.cs.utk.edu/netsolve

2.3.1.3 Nimrod

Nimrod [4] is developed at School of Computer Science and Software Engineering Monash University. It came into picture in August 1995. It is implemented in the C language. There are different flavor of Nimrod: Nimrod-G, EnFuzion, Nimrod-O, Active Sheets and Nimrod Portal. EnFuzion [7] and all other flavors of Nimrod [8] are available. Nimrod is a tool for performing parametric studies and for job submissions across loosely coupled workstations.

2.3.1.4 Ninf

Ninf [5] is a collaboration product of AIST, University of Tsukuba, Tokyo Institute of Technology, Real World Computing Partnership, Kyoto University and NTT Software Inc. Ninf [9] projected in the year 1994. Ninf is built using C and C++ languages. There are three flavors of Ninf: Ninf Portal, Ninf-G and Ninf-C. The Ninf Portal is an automatic generation tool for Gird Portals. Ninf-G directly accesses resources on grid managed by the Globus Toolkit. Ninf-C itself does not have any facility to directly access grid, although it can indirectly access via Condor-G. Ninf is a client-server RPC based system for large scale scientific computing.

2.3.1.5 PUNCH

Purdue **U**niversity **N**etwork **C**omputing **H**ubs [54] is developed at Purdue University, USA. PUNCH [10] is compatible with Win 9x/XP and all flavors of Unix. *PUNCH is a demand-based network-computing system that allows users to access and run existing software tools via standard world-wide web browsers. Tools do not have to be written in any particular language, and access to source and/or object code is not required.*

2.3.1.6 WebCom

WebCom [68] is developed at Centre for Unified Computing, Cork, Ireland. WebCom project was started in the year 1996. The grid enabled version of WebCom is WebCom-G, where "G" stands for "Grid". It is available [11] for authorized users. WebCom-G is implemented in Java and is compatible with Win 9x/XP and all flavors of Unix and Linux.

Analysis
The Development of various Network enabled Server systems have been presented. These environments consider a variety of application domains and employ multiple

[7] http://www.axceleon.com/
[8] http://www.csse.monash.edu.au/~davida/nimrod.html/downloads.htm
[9] http://ninf.apgrid.org/packages/welcome.shtml
[10] http://punch.purdue.edu/HubInfo/presentations/1999/iupui.html
[11] http://www.cuc.ucc.ie

programming technologies. Earlier systems employed C and C++, recent years Java is becoming a more popular implementation language.

2.3.2 Architecture

The necessary components for a basic NES environment are Client, Server, Database, Monitors and Scheduler. *Clients* provide user interface and submit requests to execute libraries or applications to servers. *Servers* receive requests from clients and execute libraries or applications on their behalf. The *Database* contains the status (dynamic and static information) of the monitored resources. *Monitors* dynamically store and maintain the status of the available computational resources in the database. The *Scheduler* selects a potential server from a list of servers maintained in the database by frequent monitoring and maps client requests to that server. Some NES systems have merged the functionalities of basic NES components while others have splited the functionalities of NES components, thus giving rise to a different number of components and different naming of components in each NES environment.

2.3.2.1 NeOS

NeOS [11] is an environment for solving optimization problems over the Internet. NeOS have three components: client, server, and solver. NeOS client performs same functionality as of the basic NES client, but NeOS server performs the functionality of database, monitor and scheduler. NeOS solver performs the functionality of basic NES server. Clients submit optimization problems to the NeOS server via e-mail, the World Wide Web, or the NeOS Submission Tool. The server locates the appropriate optimization solver and does all the additional work that is required by solver before and after solving the submitted problem. The solver solves the submitted task for optimization. Connectivity of the NeOS components is shown in Figure 2.1.

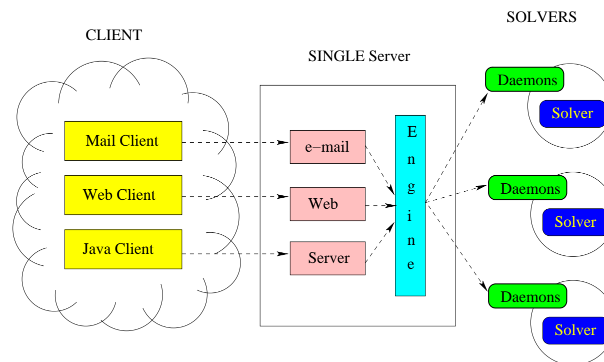


Figure 2.1: The NeOS architecture

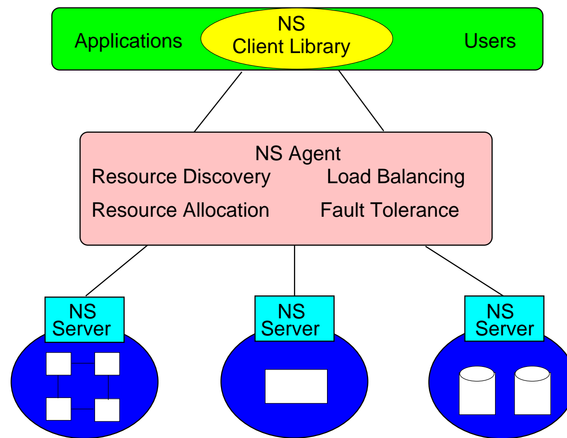


Figure 2.2: The NetSolve architecture

2.3.2.2 NetSolve

NetSolve [29] has three main components: client libraries, agent and server. NetSolve client and server performs the same functionality as of the basic NES client and basic NES server respectively. Where as the functionalities of database, monitor and scheduler is merged in NetSolve agent. Using client libraries, users write applications and submit them to NetSolve for execution. Agent gathers information about the server and, depending on the server capability chooses the appropriate server to execute the client-submitted task. A server is a networked resource that serves up computational hardware and software resources. Figure 2.2 shows how these components are organized.

2.3.2.3 Nimrod

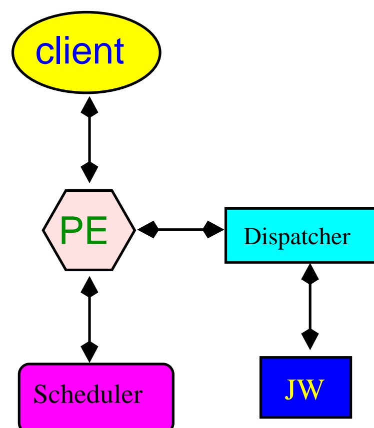


Figure 2.3: Architecture of Nimrod-G

The architecture of Nimrod [21] is shown in Figure 2.3 with its five key components: clients, Parametric Engines (PE), a scheduler, a dispatcher and a Job Wrapper (JW). PE performs the same functionalities as of the basic NES database. Functionalities of monitor is mutually performed by dispatcher and JW. JW also performs tasks execution as basic NES server. Clients act as a user-interface for controlling and supervising an experiment under consideration. They also serve as a monitoring console and list the status of all jobs, which a user can view and control. PE acts as a persistent job control agent and is the central component from where the whole experiment is managed and maintained. It is also called as Task Farming Engine (TFE) in some Nimrod articles. It is responsible for parametrization of the experiment and the creation of jobs, maintenance of the job status, interaction with the clients, the schedule adviser, and the dispatcher. The Nimrod scheduler is responsible for resource discovery, resource selection, and job assignment and the dispatcher primarily initiates the execution of a task on the selected resource as per the scheduler's instruction and periodically updates the status of task's execution to PE. JW interprets a simple script containing instructions for file transfer and subtask execution. It is responsible for staging the application tasks and data; starting execution of the task on the assigned resource and sending results back to the PE via the dispatcher. It basically mediates between the PE and the machine on which the task runs.

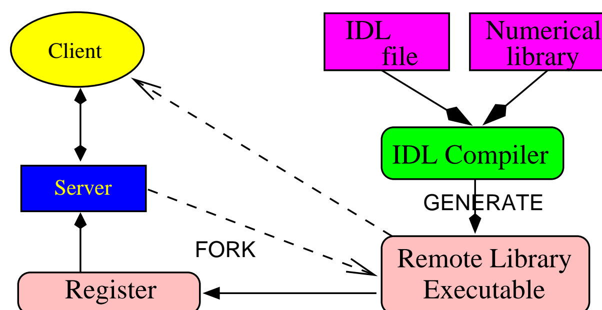


Figure 2.4: Architecture of Ninf

2.3.2.4 Ninf

The basic Ninf [71] system have main two components: a client and a server (performs the functionalities of database manager, scheduler and monitor of basic NES environment). But to help the working of server other three elements are used: Remote Library Executable (RLE), IDL compiler, Register driver. The client facilitates the users by providing an easy to use API. RLE, executes numerical operation. RLE contains network stub which handles the communication between server and clients and marshals arguments. RLEs are implemented as executable programs with stub routine as the main, which is managed by the server process. The IDL compiler compiles interface descriptions and generates 'stub main' for RLE helping to link the executable. The register driver registers remote library executable into the server. The

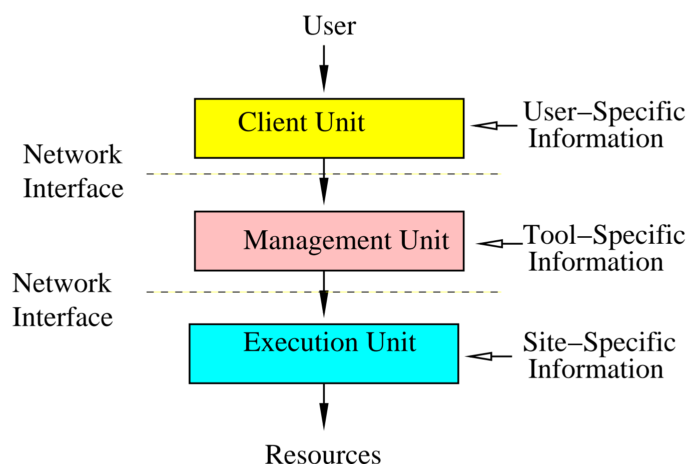


Figure 2.5: Architecture of PUNCH

server provides library interface information and invokes the RLE. Connectivity of the Ninf components is shown in Figure 2.4.

2.3.2.5 PUNCH

The core architecture of PUNCH [54] has three main units: client unit, management unit and execution unit. Management unit performs the working of database management, scheduler and monitor. Execution unit do tasks execution, so work as server. The hierarchical connectivity of the units is shown in Figure 2.5. The client unit acts as channels for command, preferences and directives specified by the user. It provides interface related input and output to the system and is integrated into the local environment, giving transparent access to the network computing system. Management unit acts as a demand-driven scheduling engines for associated software and hardware resources introducing access control policies and match making resources to the requirements. The execution unit provides the management unit with consistent access to the hardware resources.

2.3.2.6 WebCom-G

WebCom-G [72] is modular in design and has a multilayered architecture shown in Figure 2.6(a) with components in each layer performing specific tasks. This design separates the application dependency from the execution platform. The WebCom-G system is vertically integrated to the underlying hardware and co-exists with existing core and user level grid middlewares. Core modules of the WebCom-G architecture (Figure 2.6(b)) are the connection manager module, the load balancing module, the fault tolerance module, the scheduling module, the communication module and the security module which form a layer of WebCom-G. These modules are plugged into the root module called the *BackPlane* module which is responsible for bootstrapping particular instance of WebCom. Each of these modules perform task indepen-

dent of each other to maintain the integrity of WebCom-G system. End users and developers can develop their own modules or extend the existing modules and plug them into WebCom-G system. There exists WebCom-G modules like COM/DCOM modules, .NET modules, CORBA modules, EJB modules, Globus modules (for executing COM/DCOM, .NET, CORBA, EJB and Globus instructions respectively). A layer above these modules consists of utility plug-ins such as the WebCom Information Service (WIS), the Status Analyzer, the Economic Analyzer, grid Administrator, Programming Environments which utilize the underlying core modules. WebCom-G co-exists with other middleware shown in Figure 2.6(a).

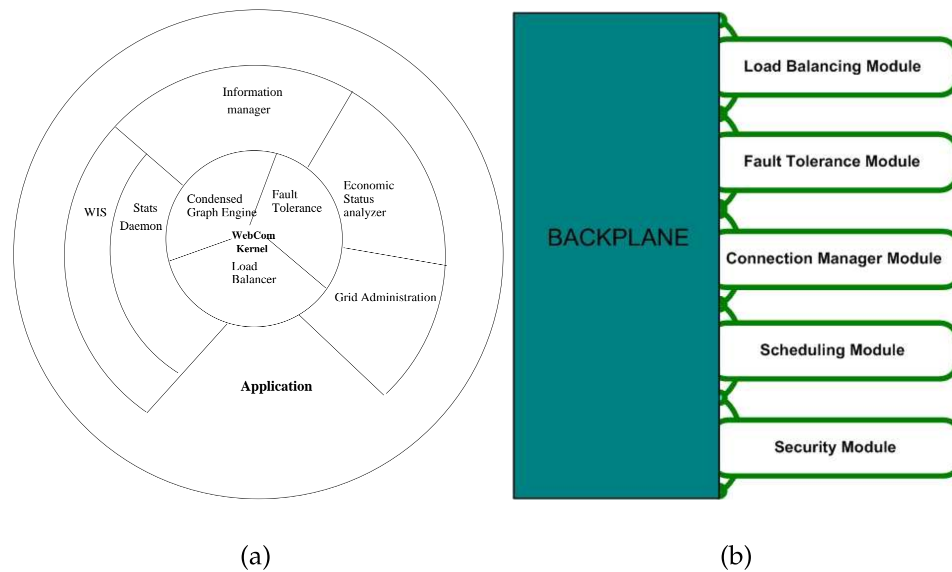


Figure 2.6: Architecture of WebCom-G. (a)General view of WebCom-G. (b) A minimal WebCom installation consists of a Backplane module, a Communication Manager module and number of module stubs for processing, Fault Tolerance, Load Balancing and Security.

Analysis

It can be seen that most of the architectures surveyed employ a two layered client/server model. The exception is WebCom which is hierarchical; to spread the computational load across multiple subnets. This facilitates the exploitation of underlying shared resources. Even this hierarchical structure is dynamically reconfigurable. NeOS is aimed at optimization issues, Nimrod at parametric computations. PUNCH is oriented at WWW browser enabled NES computing and NetSolve is multilayered but does not support a hierarchical structure.

2.3.3 Initialization

As shown in the previous section each NES environment has a different number of components and each component has a different functionality. So the order of component deployment also varies with each NES environment. In the following sections the initialization steps of the surveyed NES environments are outlined.

2.3.3.1 NeOS

As it is a static environment to solve optimization problem, (NeOS servers are initialized by the NeOS administrator) / (users cannot initialize the NeOS server, they can only submit their job for optimization to NeOS server). Thus there are no specific initialization steps that can be mentioned here.

2.3.3.2 NetSolve

First the agent is launched. Then, servers register with it by sending a list of problems that they are able to solve, the speed, workload of the machine on which they are running, and the network's speed (latency and bandwidth) between them and agent. Once this initialization step is performed, a client can call upon the agent to solve a problem. NetSolve [31] initialization steps are shown in Figure 2.7.

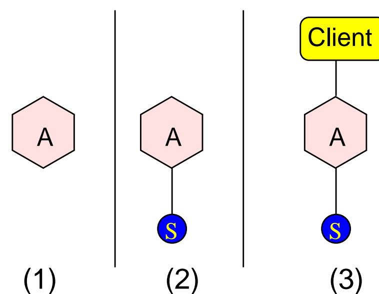


Figure 2.7: NetSolve initialization steps

2.3.3.3 PUNCH

The initial step of PUNCH [54] system is the creation of the input files for the relevant simulation in case of VLSI design and computer architecture followed by user input parameters for the simulation program. Finally the simulation is started via a browser interface.

2.3.3.4 WebCom

A WebCom-G [12] binary distribution file can be downloaded from the official site by the authorized users. Install it on the local machine. Depending on the requirements, one can configure it to act as a master server or a client to one of the WebCom-G portals. Users can launch the WebCom-G IDE, build applications on the fly and launch to run on the local machine or target them to run at the remote machine or they can use the command line execution facility. The initialization takes place within the backplane which checks for modules that are loaded during the startup (that is why backplane is a also referred as bootstrap program). Once the modules are loaded communication is initiated, clients connect and instruction queues are created for each. To each client, work is being distributed by Condensed Graph engine which is scheduled by the scheduler along with the help of load balancer.

Analysis
Components of NES environments can have different functionality and each can be deployed independently. It can be seen that some systems use agents for initializing (NetSolve), others are initialized upon submission of the tasks via their clients or browser (NeOS, Ninf, PUNCH, Nimrod), where WebCom system has to be configured initially but the state can be changed dynamically. Initialization reflect the type of work that each system performs. Some of these systems are used for parametric jobs (Nimrod), some for optimization of their application (NeOS), some for parallel/-workflow computational management of work (WebCom).

2.3.4 Task Execution

The granularity and type of tasks being executed by the NES environment varies. Each of these environments have its own model of task execution. General working of a basic NES environment is described below. The *Monitors* frequently monitor the status of the available resources (server, network, latency, CPU load, etc..) and register the information to the *Database*. *Clients* demand the scheduler to provide them with a suitable computational server for their request to be executed with the help of client APIs (e.g. C, Java, MATLAB, Fortran, Web, e-mail, etc.) or tools constructed using client APIs. The *scheduler* queries the database and selects suitable computing resources based on certain algorithms and return the selection to the client. Clients remotely invoke these libraries or applications on the selected server. The *server* does the computation and returns the resulted data to the client. The following sections shows the task execution mechanism of surveyed NES environments.

2.3.4.1 NeOS

A client sends a job to the NeOS server. The server assigns each job to a solver script that executes application software via requests to a communications daemon assigned

[12] http://portal.webcom-g.org

to the application and solver station. The server can then delegate to the solver machines the tasks of interpreting data and executing software. The NeOS Comms Tool is used to download the user files from the NEOS server to solver, to invoke the solver application, and to return the solver's results to NEOS. The NEOS Server, upon receipt of a job for a particular solver, connects to the machine running the Comms Tool daemon for that solver, uploads the user's data, and requests invocation of the remote solver. The Comms Tool daemon, upon receipt of this message, downloads the user's data and invokes the application software. Intermediate job results can be streamed back over the connection to the server and from there, in turn, streamed back to the user. The Comms Tool daemon sends final results to the NEOS server upon completion of the job, and the server formats and forwards the results to the user. A detailed working of NeOS with the submission flow for three jobs arriving simultaneously is illustrated in [40].

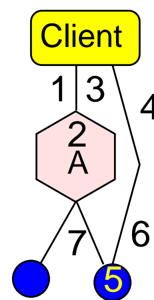


Figure 2.8: NetSolve task execution steps

2.3.4.2 NetSolve

A NetSolve session works as shown in Figure 2.8. (1) A client sends its task to the agent. During a NetSolve deployment, the severs provide all the information regarding its status and the tasks that they can execute to the agent. Then (2) an agent selects the appropriate server for the client request and (3) gives the server a reference to the client. Then agent increases the occupation status of this server by some factor (NetSolve use some mechanism for it). (4) Clients directly connects to the server and provide data for task execution to the server. When (5) tasks is executed (6) result is sent back to the client by the server. (7) the Server conveys its current status to agent after finishing the task, so that agent can remove the factor that the agent added to the server occupation status. Detailed technical explanation of NetSolve component working is available in [19].

2.3.4.3 Nimrod

Working of Nimrod [2] is shown in Figure 2.9. (1) Client sends a request to a PE. (2) PE prepare data for client particular execution. (3) It asks the scheduler to order the re-

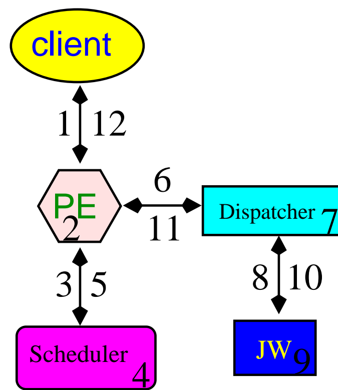


Figure 2.9: Nimrod: Work procedure

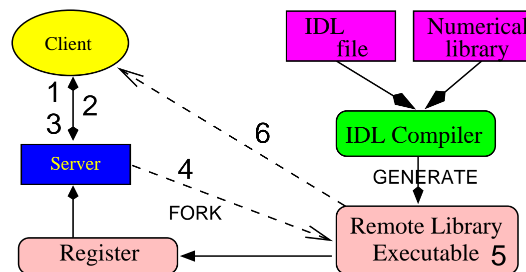


Figure 2.10: Working of Ninf

quest. (4) The scheduler arrange the request according to some scheduling algorithms, and (5) sends the ordered requests back to the PE. (6) The PE forwards the request to the dispatcher, (7) which selects the appropriate Job Wrapper (JW). (8) The JW receives the request from dispatcher. (9) It executes the program for a given set of parameters and (10) sends back the result to dispatcher, which returns the result to PE (11). Reduction of executed data if required is done by the PE. Finally PE gives back the result (12) to the client. Steps (1) and (12) are performed once per experiment, while steps (2) to (11) are run for each distinct parameter set.

2.3.4.4 Ninf

Task execution by Ninf is shown in Figure 2.10. (1) Client sends an interface request to server. (2) Server replies the interface requests. Then (3) client calls the library to invoke the executable. (4) Server searches the Ninf executable associated with the name that the client calls for and (5) upon successful search executes the library and (6) setups a communication link with the client with the help of stubs to return the result. In short, clients ask server to execute his request and server executes the requested libraries and sends back the result.

2.3.4.5 PUNCH

Figure 2.11 shows PUNCH's working. (1) User access PUNCH system via standard WWW browsers. (2) Network Desktop processes and responds to all "non-application-invocation" requests. (3) Application invocation requests are forwarded to an appropriate management unit. (4) Management unit authenticates requests, determines resource requirements and selects execution unit. (5) Management unit sends the request to the execution unit. (6) Execution unit processes requests with the help of available resources and (7) notifies management unit upon completion.

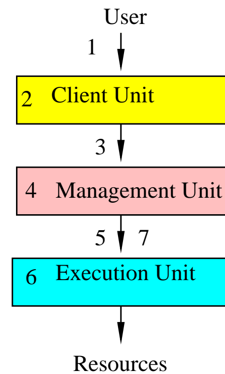


Figure 2.11: Working of PUNCH

2.3.4.6 WebCom

WebCom [66] consists of a master and an arbitrary number of clients. The master initiates the computation. A client may be a master or a java enabled web browser capable of executing atomic instructions. Client connected via web browsers will only be sent atomic instructions (instructions which cannot be further distributed), where as client masters may be sent both atomic and graph instructions.

WebCom deployment consists of number of Abstract Machines (AM). When a computation is initiated one AM acts as the root server and the other AMs act as clients or promotable clients. A promoted client acting as a server accepts connections and executes tasks. Promotion occurs when task, representing Condensed Graph (CG), passed to the client can be partitioned for further distribution. Hence in WebCom environment clients are promoted to masters depending on the type of task it receives [69].

Working of WebCom is shown in Figure 2.12. (1) Computation begins with 2-tier hierarchy. Graph execution begins on the server. (2) Server sends a CG task to a client, causing it to be promoted. (3) The promoted client requests additional clients to be assigned to it by the server. This request might not be serviced. If not the client continues executing the graph on its own. (4) The server issues a redirect directive to a number of its clients. (5) Clients redirected from the server connect as clients to the promoted client. (6) The promoted client directs its new clients to connect to each

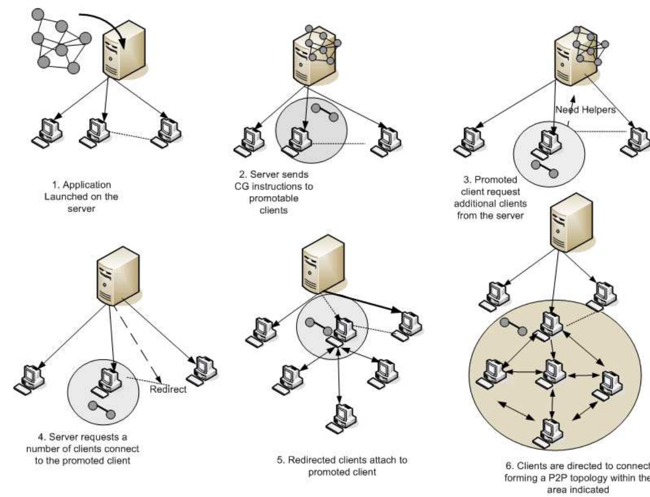


Figure 2.12: Working of WebCom-G

other forming a local peer to peer network. (7) Finally, when the tasks are executed results are sent back to the parent server.

Analysis
The major difference in task execution among NES environments is based on the work done by its components. As in NetSolve the work is done by the server on behalf of its client, but in WebCom the work is done by the clients.

In NetSolve, agents search for potential servers, where as in system like WebCom, servers search for potential clients for faster execution. Even the granularity of the tasks executed is different in most of the surveyed systems. In some systems (NeOS, PUNCH) atomic (single task) tasks have to be submitted where as in some system (WebCom), tasks can be submitted as condensed (task within task) tasks.

2.3.5 Communication

Communication is very important for transferring the task and the tasks' data to the relevant components in the hierarchy of NES components. NES environments use different types of communication technologies. In the following sections we present the respective communication technology of each of surveyed NES environments.

2.3.5.1 NeOS

NeOS communicates with the user through Internet communication using TCP/IP.

2.3.5.2 NetSolve

The Network enabled Solver based system's components use TCP/IPv4 sockets and a NetSolve-specific application layer protocol to communicate with each other [31]. Heterogeneity is supported by implementing a 'handshake' transaction that initiates each communication session, during which computer hosts determine whether they understand the same data format. When an incompatibility is detected, Sun's XDR protocol is used to encode local data to a globally-understood format, allowing each host to decode from this external data representation to its local format. The communication between the NetSolve modules is performed by using the NetSolve protocol (NetSolve-IO). Input/output data are transmitted through a single stream between the NetSolve client module and the server node. The client-agent and client-server communications are performed through the NetSolve Proxy.

2.3.5.3 Nimrod

The interaction between the PE, the scheduler and the dispatcher takes place through network protocol TCP/IP. All communications across the network are done via Remote Procedure Call.

2.3.5.4 Ninf

The communication between a client and the server is achieved by standard TCP/IP socket connection [75]. In an heterogeneous environment, Ninf uses the Sun XDR data format as a default protocol. Clients can also specify call back functions on the client side for various purposes, such as interactive data visualization, I/O of data, etc.

2.3.5.5 PUNCH

PUNCH components communicate through TCP/IP and HTTP protocols. All the interfaces to the PUNCH system is via WWW enabled browsers. These components use access codes and hash functions to authenticate component, passwords and users identity.

2.3.5.6 WebCom

WebCom contains two Communication modules. One that possesses the minimum functionality required for the configuration of traditional grid environment and a more advanced one to bootstrap other modules of the WebCom from a specified code based on the network. This module maintains a separate list of descriptors for servers and clients. The communication module listens for the incoming connections from the client supporting bi-directional communication.

Analysis

NES environments use different types of communication technologies. TCP/IP is used by most of the surveyed NES environments, like NeOS, NetSolve, Ninf and PUNCH. Some also uses third parties to handle communications between their components. Some also use technologies such as RPC and RMI.

2.3.6 Scheduling and Dynamic Load Balancing

Scheduling is a necessity of all existing systems. A good scheduling is the one that can provide fairness in task distribution, have good policy enforcement and balance a system when its fully busy while meeting the deadlines of the given task. NES environments also use scheduling to fully utilize the resources, but each of them has a different scheduling scheme.

When processor load cannot be determined statically, dynamic load balancing techniques can be used for effective speedups. In this section we present the scheduling techniques for each of the NES environments and dynamic load balancing for WebCom.

2.3.6.1 NetSolve

NetSolve use a theoretical model [10] to estimate the performance, given the raw performance and the CPU load. This model gives (p) the estimated performance, as a function of the CPU load (w), the raw performance (P), and the number of processors (n) on the machine:

$$p = \frac{P \times n}{w/100+1}$$

The hypothetical best machine is the one yielding the smallest execution time T for a given problem. It incorporates on-line scheduling heuristics that attempt to optimize service turnaround time. A server has the capability to service simultaneously as many requests as the operating system will allow it to fork processes, but the administer can specify the maximum number of requests it is willing to service at one time; requests for services received once this limit is met will rejected. Similarly, a server can specify its CPU load threshold and, once requests received after this threshold is attained will be refused.

2.3.6.2 Nimrod

Nimrod-G provides a persistent and programmable task-farming engine (TFE) that enables 'plugging' of user-defined schedulers and customized applications or Problem Solving Environments (e.g., ActiveSheets) in place of default components. TFE is a coordination point for processes performing resource trading, scheduling, data and executable staging, remote execution, and result collation. The Nimrod-G system automates the allocation of resources and application scheduling on the grid using economic principles in order to provide some measurable quality-of-service to the end user.

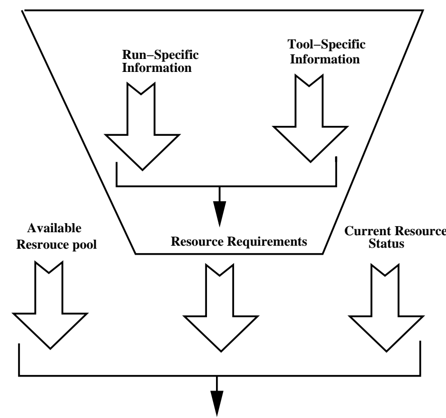


Figure 2.13: Demand-based Scheduling [54]

2.3.6.3 Ninf

Ninf (like NetSolve) makes scheduling decisions based on the dependencies of the input data: any groups of remote calls are analyzed for dependency relationships, and those that may be executed without waiting for another call to finish are immediately sent off to a remote resource.

Although a programmer can define blocks of related function calls, the actual selection of a remote resource and the migration of data is handled by the Ninf Metaserver (like the NetSolve agent). The user provides a problem description to the Metaserver, which tries to match as closely as possible the correct resources to the submitted job, based on a database of resource loads and performance.

2.3.6.4 PUNCH

Client, management and execution units are the three important units in the PUNCH system. Each unit has a specific task to accomplish and the management unit acts as demand-based scheduling engine (Figure 2.13) for respective software and hardware resources. Depending on the requirements of the user requests, the management unit analyzes the requirements and then matches the requirements with related resources. Here the management unit acts as a matchmaker, mapping the requests to the appropriate resources.

2.3.6.5 WebCom

Decisions about when the tasks are to be run once they are ready to executed. The functionality of this module is very trivial because as soon the task are ready they will be soon executed. WebCom is a condensed graph based model providing many scheduling mechanisms (namely critical path analysis, speculative computations, reduced priorities and others).

The Load balancing module [68] is one of the core modules of WebCom which is plugged into the BackPlane. This module decides where the tasks are to be executed.

It works using the Round Robin method to distribute tasks to the clients. When a suitable client is known, this module requests the Backplane to send the instruction to the selected client, if the selected client is not found, it re-schedules the task for a local execution. WebCom-G requires all the variants of load balancing modules to communicate directly with Backplane and depends on the status information of the network provided by the communication module. Load balancing modules developed by the users, may employ different strategies with respect to security, resource access, resource reliability and others.

Analysis
Most of the systems studied have a fixed scheduling scheme, like NetSolve, Ninf and PUNCH. But others like Nimrod and WebCom allows plug-in-scheduling, where users can add scheduling algorithms according to their requirements. All of the systems surveyed except WebCom employs a static load balancing scheme.

2.3.7 Fault Tolerance

The fault tolerance is a significant and complex issue in grid computing systems. Various techniques have been investigated to detect and correct faults in distributed computing systems. Unreliable fault detection is one of the most effective techniques. The availability of albeit unstable resources in non-dedicated environments mandate that all faults in the stages of user computation be handled in a transparent and graceful fashion. Variable stages during the computation exhibits various facets of fault tolerance, and as such they must be handled in a stage-by-stage basis. In the following sections fault tolerance mechanisms and after effect of each component crash for NES environments is stated.

2.3.7.1 NetSolve

The NetSolve system ensures that a user request will be completed unless every single resource capable of servicing the request has failed [30]. When a client sends a request to a NetSolve agent, it receives a sorted list of computational servers to try. When one of theses servers has been successfully contacted, the numerical computation starts. When one of these servers fail during the computation, then another server is contacted and the computation restarts. This whole process is transparent to the user. Each time a computational server malfunction (server unreachable, server stopped, failure during computation, etc.) is detected by a client, this client notifies the failure to the agent. The agent update its tables and takes the necessary measures. If all the servers fail, then the user is notified that the computation can not be performed at that time.

In NetSolve, if agent is crashed, NetSolve system can not be used. NetSolve server crash has same effect as of DIET server's crash.

2.3.7.2 Nimrod

The PE takes the experiment plan as input, described by declarative parametric modeling language (the plan can also be created using the Cluster GUI) and manages the experiment under the direction of schedule adviser. It then informs the dispatcher to map an application task to the selected resource. The PE maintains the state of the whole experiment and ensures that the state is recorded in persistent storage. This allows the experiment to be restarted if the node running Nimrod goes down [20]. PE crash leads to unavailability of Nimrod environment.

2.3.7.3 Ninf

When a server fails to complete a task, the task is simply rescheduled to a new resource. Ninf integrates with the Condor system for checkpointing to enable fault tolerance for computation.

2.3.7.4 PUNCH

PUNCH system's front end have logical accounts one per user on any given management unit. The management unit contains logical accounts for each authorized PUNCH user who has access to the front end. PUNCH can handle single point failures with the help of management unit via scalability. Fault tolerance is achieved through distributing the software resources among appropriate number of management units. If management unit crashes, requests can not be executed by PUNCH.

2.3.7.5 WebCom-G

The fault tolerance module [68] is another core module of WebCom-G which is plugged into the BackPlane. It employs numerous algorithms to check point and re-schedule the task in case of failure or unavailability of a client/machine which is performing the task. Whenever a failure occurs in the execution hierarchy of WebCom, this module kicks-in and reschedules the task to a different client/machine with the help of the Communication module.

Analysis
NetSolve use fault tolerance mechanism based on timers. The web based systems employ a checkpoint and restart mechanism but only WebCom system includes a fault survival mechanism which enables it to re-incorporate results from the failed execution into recovered process.

2.3.8 Data management

Data management is an important component in grid Computing due to different application needs, for example some applications are data intensive and some applications are CPU intensive, further there might be data dependency between the com-

ponents of a system. In any of these cases, data or applications have to be moved to respective location to avoid the overhead of bandwidth.

2.3.8.1 NetSolve

The NetSolve approach provides two services to manage data inside the platform: the Distributed Storage Infrastructure (DSI) and the Request Sequencing Infrastructure (RSI).

DSI used to provide data transfer and storage. RSI used to decrease network traffic amongst client and servers [38]. Data references are managed inside RSI or by file descriptors in the DSI solution. In NetSolve request sequencing approach, the sequence of computations needs to be processed by an unique server. In this case, a client has to have the knowledge of the services provided by a server in order to use this approach.

Data is removed from server DSI depot after request computation, to handle the data persistency. To reduce the data transfer time, DSI depot should be near to the computational servers.

2.3.8.2 Nimrod

Nimrod database is containing routing information that is constructed, accessed, and acted upon by the routing functions. An endpoint association should be stored close to the endpoint so that other entities in the vicinity can quickly locate the endpoint. An endpoint associations should also be stored at authoritative sources so that distant entities can find the endpoint independently of its current location. Node representatives collect maps from component nodes and build detailed maps from this information. Abstract map built by abstraction of service information in detailed map or by configuration of measurement of service information. Map updates distributed to all node representatives and all route agents for the enclosing node. Route agents use maps to generate routes and may request lower-level maps based upon those received. Routes are expressed as sequences of node locators, together with the labels for those node service attributes used in route selection. Routes must include at least the source and destination node locators.

2.3.8.3 Ninf

Ninf-C is based on Condor and thus uses the condor data management techniques. In Condor data placement is handled by the Data Placement Scheduler (DaPs). DaPs intelligently manages and schedules data placement activities/jobs.

2.3.8.4 WebCom

Data management in WebCom is simple and dynamic. The kernel of the WebCom system is Condensed Graph model. The Condensed Graphs unify lazy, imperative and data-driven evaluations of job execution. These evaluations depend on the quality of service requirement; whether job execution has to be completed slowly (lazy

computation) or in a systematic manner (like a workflow) or an on-demand (user has to interfere at this point to give some input or analyze the results) user interactive basis. According to the quality data flow will match the dynamic evaluation techniques. WebCom-G has a dynamic way of switching different evaluation techniques on need basis.

Analysis
Some NES environments that need data management have their own data management models, for eg., NetSolve has DSI. Others use third party data management. Ninf uses Condor data management techniques. Data management in WebCom is performed explicitly by the user in the form of workflow graphs. Some of the systems (NeOS, PUNCH) surveyed have no data management support.

2.3.9 Security

Applications are run within the NES environments (commodity resources and on multiple administrative domains) for the users/clients who are not identified in person. Security is a must for client code that is running in a NES environment and for the owner of the resource on which the code is running. Security with respect to administrator tampering the clients code and client.s trust of running valid code on the providers machine. Security is one of the main concern in distributed systems.

2.3.9.1 NeOS

Security in NeOS is not inbuild. VPN security is used by NeOS for secure connections between the application sent and the server.

2.3.9.2 NetSolve

Security in NetSolve [13] is enabled via Kerberos support. NetSolve servers have two type of installation with Kerberized and non-Kerberized installation. In both the cases clients send request to the server, Non-kerberized installation the server will return a status code indicating the acceptance and Kerberized installation will return an authentication error response to the request. So the clients have to send their Kerberos credentials to the server before they proceed, this is how NetSolve server authenticates its client. The Kerberized server maintains a list of Kerberos principal names to implement access control. Validity of the clients connecting to the server is checked with the access list. There is no encryption or integrity of data stream exchanged between its components.

[13] http://icl.cs.utk.edu/netsolvedev/documents/ug/html/additions.html

2.3.9.3 Nimrod

Nimrod does not have any inbuilt security. Nimrod-G which is globus enabled uses the security provided by Globus (Grid Security Infrastructure).

2.3.9.4 Ninf

Ninf system does not have any built in security mechanisms. Ninf-G uses the security provided by the Globus(Grid Security Infrastructure).

2.3.9.5 PUNCH

There is no implicit security built into the PUNCH system. But the front end to interface with the PUNCH environment gives the required security. As PUNCH environment and their tools can only be accessed WWW enabled browsers. Users are not allowed to change, install, run code on this environment. They are only allowed to use the tools provided by the PUNCH environments to do their experimental research.

2.3.9.6 WebCom

WebCom-G's distributed computing architecture is used to distribute application components for execution. WebCom uses KeyNote-based authorization credentials [44] to assert whether a WebCom server is authorised to schedule and whether a WebCom client is authorised to execute, distributed application components. Secure WebCom provides a meta-language for bringing together the components of a distributed application in such a way that the components need not concern themselves with security issues. WebCom also supports Dynamic administrative coalitions(DAC) [43], where in users holding the administrative powers deligate administrative tasks constrained by some set of rules. This phenomenon is used in the integration of security and workflow in a decentralised administrative architecture to provide fault survival against failures.There is no encryption or integrity of data stream exchanged between its components.

Analysis Only WebCom has a inbuild security system. Where as others use third party security like security provided by VPN.

2.3.10 Deployment and Visualization Tools

It is difficult to see the status of a working production grid system without a customized monitoring system. The status includes many details of the system such as system running information, performance changes, system/software failures, security issues and so forth. Most grid middleware provide a simple monitoring tool for their systems, or provide simple tools to check the status of the system.

2.3.10.1 NetSolve

NetSolve has a visualization tool called VisPerf written in Java. This tool is used to display the status of the resources interacting with the NetSolve system. It is also used to monitor the activity of the NetSolve grid. The overview of VisPref is shown Figure 2.14.

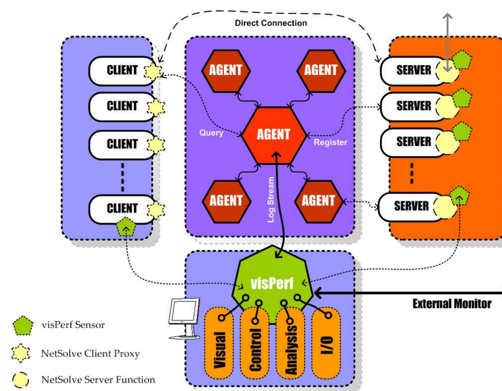


Figure 2.14: Overview of visPerf Monitoring System [62]

2.3.10.2 WebCom

WebCom-G Integrated Development Environment (IDE) is both a deployment and visualization tool. It is a (1) dynamic visualization tool to view the services hosted by the resources on which WebCom is installed, (2)an application development tool and (3) an application launcher and a targeting tool. The *Menu Bar* (Figure 2.15) used for Graph opening, editing, running operations. *Palette* hosts a list of active services, libraries of WebCom-G environment. These can be incorporated by just dragging and dropping on to the canvas and building a graph application. The *Graph Canvas/Drawing Area* is to build your graph applications on the fly. A *Properties Panel* hosts properties pertaining to graphs, nodes, edges present on the canvas.

Analysis
Not all surveyed NES environments have their deployment and /or visualization tools. Only WebCom has both tools. WebCom uses IDE for both deployment and visualization. NetSolve has only a visualization tool. The other systems surveyed have no visualization tools.

2.4 Comparison of Systems

Table 2.1 shows the summary of the environments surveyed and their associated characteristics. DIET has also been added in the table to have a quick view of DIET with

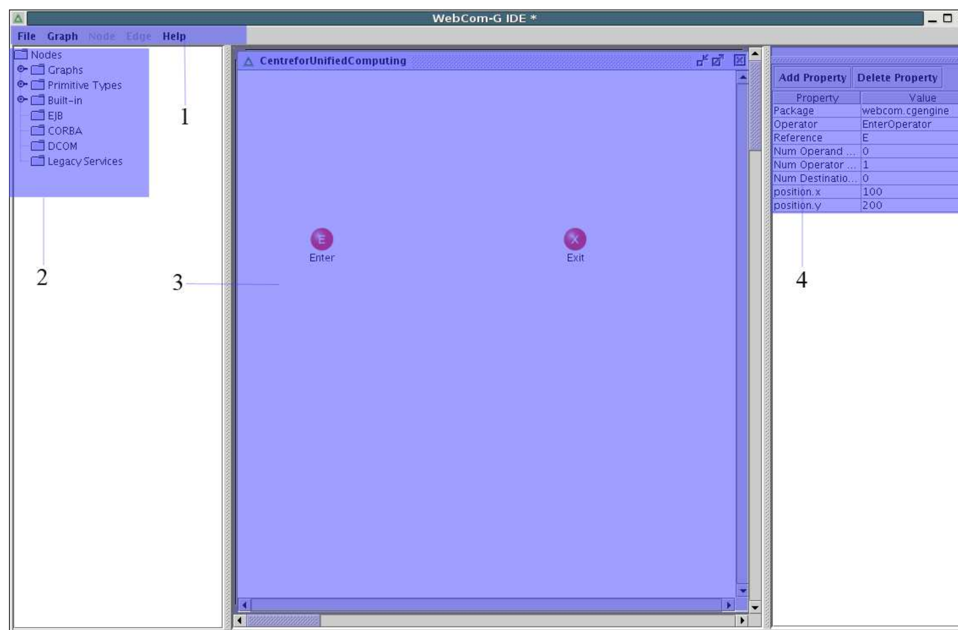


Figure 2.15: Description: WebCom-G Integrated Development Environment

respect to the compared characteristics but detailed explanation of DIET is presented in chapter 5. The table summary can be used to identify the major driving forces behind the development of each system.

Only two of the surveyed environment provides deployment tools to facilitate the work of user for environment deployment.

Three of the environment surveyed are capable of dynamically visualizing task execution. In the other environments, tasks a processed in a "fire and forget" manner.

Support to WebService is available in all environments except one. It seems WebService is the mostly used method to submit the jobs to the environments.

It can be seen that five environments surveyed have scheduling mechanism and one among the left deals with optimization so scheduling is not that important factor for it.

Dynamic load balancing hasn't received any big attention from the environments. Static load balancing is provide by five environments but dynamic load balancing is provided by only one surveyed environment.

Most of the environments use basic timer mechanism for fault tolerance. Checkpointing concept is available only in one among the surveyed environments and thus have in-built fault tolerance. Some of the rest environments use the fault tolerance mechanism provided by the third party like Globus or Condor.

It can be seen, for example, that security has not been a major consideration. Only one of the seven environments surveyed has an integrated security mechanism, however, others do make use of the security features of their third parties component subsystems.

Three environments can potentially exploit the integration of loosely coupled re-

sources. Of these, two require the use of the Globus middleware; the other, in contrast, is capable of acting as stand alone middleware.

The environments that do not have security or fault tolerance important to the user, who is interested in easy deployment and simple task execution. Because the added mechanisms makes the environments heavy and bit complicated to be used by the end users. So according to different needs users can select appropriate NES environments.

2.5 Conclusion

We did a survey of most prevalent NES environment, so as to obtain a depth knowledge of existing NES environments. We have presented six NES environments (NeOS, NetSolve, Nimrod, Ninf, PUNCH, and WebCom) under various headings, enabling an objective comparison to be made. By objectively comparing these systems, an attempt is made to enable potential NES users' to choose an appropriate environment to best suit their needs. In addition to these NES environments we also presented some other systems which are popular grid middleware and grid systems. These systems work in same space as NES environments but are different because some of them are based on cycle steeling concept unlike NES environments, which are dedicated servers. And some other systems which are dedicated does not provide scheduling system, job management services, queueing mechanisms as done by NES environments.

	DIET	NeOS	NetSolve	Nimrod	NINF-G	PUNCH	WebCom-G
nization	LIP,INRIA, ENS Lyon, France	Argonne National Laboratory/MCS	University of Tennessee, Knoxville	Monash University	collaborative work	Purdue University	Centre for Unified Computing
on	DIET v1.1	NeOS v5	NetSolve v2.0	Nimrod-G v.3.0	Ninf-G 4.0.0	PUNCH v1.2	Anyware, WebCom, WebCom-G (v0.1)
pport	Unix/Linux, Mac OSX	Win 9x/XP, Unix/Linux, Mac OS, Solaris	Win XP, Unix/Linux	Win XP, Unix/Linux	Win XP, Unix/Linux, Mac	Win 9x/XP, Unix/Linux, Mac OS, Solaris	Win 9x/XP, Unix/Linux, Mac OS, Solaris
rements	OmniORB, gcc	WWW browser	gcc	gcc	gcc, Java enabled Machine	WWW browser	Java enabled Machine
en in	C, C++	Tcl/Tk, Java, Kestrel	C	C	C, C++	no particular language	Java
uling	Yes	N/A	Yes	Yes	Yes	Yes	Yes
Tolerance (Checkpointing)	No	N/A	No	Yes	Yes, Condor	No	Yes
mic Job Migration	No	N/A	No	No	No	No	Yes
ity	Yes, VPN and CORBA security	N/A	No	No	Yes, Globus	No	Yes, Trust management and Keynote
lization	Yes, VizDIET	No	Yes, VisPerf	No	No	No	Yes, WebCom-IDE
oyment	Yes, GoDIET	No	No	No	No	No	Yes, WebCom-IDE
ervice support	No	Yes	Yes	Yes	Yes	Yes	Yes
t Link	graal.ens-lyon.fr/DIET	www-neos.mcs.anl.gov/neos/	www.cs.utk.edu/ netsolve	www.csse.monash.edu.au/ davida/nimrod.html/ downloads.htm	ninf.apgrid.org/ packages	punch.purdue.edu/	www.cuc.ucc.ie

Chapter 3

Deadline Scheduling with Priority of tasks on Network Enabled Server Environments

In previous chapter, six most common Network Enable Server (NES) environments have been presented, so as to have a good knowledge of the mostly used NES. As mentioned earlier NES endeavor to provide users an easy access to distributed resources by hiding the complicated job distribution and resource allocation strategies from the user. Thus, the two important factors that a NES should efficiently handle are: (a) job distribution and (b) resource allocation. This chapter presents job distribution algorithms on a NES environment. Presented algorithms provide non-preemptive scheduling of sequential tasks on NES. Implementation of non-preemptive scheduling has been done, since at the user level it is not possible to interrupt a running process and block it in order to allow a new process to run.

3.1 Introduction

Currently, NES systems use the Minimum Completion Time (MCT) on-line scheduling algorithm where-by all jobs are scheduled immediately or refused. This approach can overload interactive servers in high load conditions and does not allow adaptation of the schedule to task dependencies. In this chapter we consider an alternative model based on the deadline and priority of the task submitted to NES. The expected response time of each task's execution as a scheduling parameter is called *deadline*. The *priority* for a task indicates how important the tasks is. Higher the task's priority more important is the task. A deadline scheduling with priority strategy is more appropriate for multi-client, multi-server case. We assume that each task, when it arrives in the scheduler, has a given static deadline (possibly given by the client). If a task completes its execution before its chosen relative deadline, it meets the deadline. Otherwise the tasks fails. Importance is first given to the task's priority and then the task is allocated to the server that can meet the task's deadline. This may cause that some already allo-

cated tasks on the server miss their deadline. We augment the benefits of scheduling algorithms with fallback mechanisms and load measurements, which is done with the use of a forecasting tool called FAST [39].

FAST (Fast Agent's System Timer) is a software package allowing client applications to get an accurate forecast of routine needs in terms of completion time, memory space and number of communication, as well as of current system availability. Appropriate tools like, NWS [85] are used to monitor the dynamic availabilities of system resources. FAST is also able to aggregate information in order to forecast the current computation time of given task on a given machine. The goal of FAST is however not to perform task placement, but to acquire the required knowledge to achieve it.

We have presented the simulation results to show that the deadline scheduling with priority along with fallback mechanism can increase the overall number of tasks executed by the NES.

3.2 Related work

While a large number of papers describe priority algorithms for classical operating systems, little research exists around scheduling algorithms using priority and deadline for grid applications.

Scheduling problem that combine tails and deadlines is discussed in [78]. Lower bounds are given for the shop scheduling problems and an algorithm is presented with an improved complexity to solve two parallel machine problem with unit execution time operations. An algorithm in [83] finds minimum-lateness schedules for different classes of DAGs when each task has to be executed in a non-uniform interval. Both papers present interesting theoretical results that can not be directly implemented in a grid platform due to some limitations of models.

A deadline scheduling strategy is given in [80] for multi-client multi-server case on a grid platform. The authors assume that each task (tasks sent to the scheduling agent) receive a deadline from the client. The algorithm presented aims at minimizing deadline misses. It is also augmented with *load correction* and *fallback* mechanisms to improve its performance. The first optimization is done by taking into account load changes (due to previous scheduling decisions) as soon as possible. The fallback mechanisms makes some corrections to the schedule at the server level. If the server finds that a task will not meet the deadline due to prediction errors, the task is re-submitted to the system. Simulation has been provided using the Bricks simulation framework. A background load is simulated and real traces are injected in the model to simulate the extra load of the system. Optimized algorithms are compared to a simple greedy algorithm that does not take deadlines into account. This algorithm is indeed less efficient than the optimized ones. The simulation shows that while the load correction mechanism does not improve the overall execution time significantly, the fallback optimization leads to important reductions of the failure rates without increasing the cost of scheduling. Our work is complementary, as we present the scheduling algorithm that take into account both the deadline and priority of task while scheduling. Even

we have also tested our algorithm with the fallback mechanism.

3.3 Scheduling algorithms for NES environments

The presented scheduling algorithms aim at scheduling sequential tasks on a NES environment. This kind of environment is usually composed of an agent receiving tasks and finding the most efficient server that will be able to execute a given task on behalf of the client. The client submits the request to the portal (an agent) of NES, which is responsible for allocating the resources and scheduling the tasks for execution. Based on different criteria the scheduling of tasks is done. Scheduling algorithms presented below are based on the servers' load, and task's deadline and priority.

3.3.1 Client-server scheduler with load measurements

As we target heterogeneous architectures, each task can have a different execution time on each server. Let T_{aS_i} be the execution time for the task a on server i. This time includes the time to send the data, the time to receive the result of the computation and the execution time:

$$T_{aS_i} = \frac{W_{send}}{P_{send}} + \frac{W_{recv}}{P_{recv}} + \frac{W_{aS_i}}{P_S}$$

where W_{send} is the size of the data transmitted from the client to the server, W_{recv} denotes to the data size transmitted from the server to the client, W_{aS_i} is the number of floating point operations of the task, P_{send} denotes the predicted network throughputs from the client to the server, P_{recv} denotes the predicted network throughputs from the server to the client and P_S is the server performance (in floating point operations per second).

P_{send} should be replaced by the network throughput value measured just before the task. This value is returned by one call to FAST (Section 5.2.4). P_{recv} is estimated using previous measurements. The CPU performance is also dynamic and depends on other tasks running on the target processors. Thus, FAST can be used to provide a forecast of CPU performance, so as to take into account the actual CPU workload.

Algorithm 3.1 gives a straightforward algorithm to get a sorted list of servers that are able to compute the client's task. It assumes that the client takes the first available server, which is most efficient, from the list. However, a loop can be added at the end of the algorithm between the `task_ack` and `task_submit` calls.

For sake of simplicity we define four functions for Algorithm 3.1:

`can_do` This function returns `true` if server S_i have the resource required to compute task T_a. This function takes into account the availability of memory and disk storage, the computational library etc.

Algorithme 3.1 Straightforward algorithm: client-server scheduler with load measurements.

```
1: repeat
2:   for all server S_i do
3:     if can_do(S_i, T_a) then
4:       T_aS_i = W_send/F_Bd + W_recv/P_recv + W_aS_i/F_S_i
5:       List=sort_insert(List,T_aS_i, T_a, S_i)
6:   num_submit=task_ack(List, 2×W_send/F_Bd)
7:   task_submit(List[num_submit])
8: until the end
```

Line 4: $T_{aS_i} = \frac{W_{send}}{F_{Bd}} + \frac{W_{recv}}{P_{recv}} + \frac{W_{aS_i}}{F_{S_i}}$

Line 6: num_submit=task_ack($List$, $\frac{2\times W_{send}}{F_{Bd}}$)

sort_insert This function sorts servers by efficiency. As an input, we have the current $List$ of servers, the time predicted T_{aS_i}, the task name T_a and the server name S_i. Its output is the $List$ of ordered servers.

task_ack This function sends the data and the computation task. To avoid a deadlock due to network problems, the function chooses the next server in the list if the time to send the data is greater than the time given in the second parameter. The output of this function is the server where the data are sent (index number in the array $List$).

task_submit This function performs the remote execution on the server given by task_ack.

3.3.2 Client-server scheduler with a forecast correction mechanism

The previous algorithm assumes that FAST always returns an accurate forecast. In this section, we take into account the gap between the performance prediction and the actual execution time of each task.

The function task_submit is upgraded to return the time of the remote execution. Thus, we can modify the next predictions from the monitoring system as follows to obtain the corrected load values:

$$T_{aS_i} = \frac{T_{aS_i} \times CorrecFAST}{100}$$

where $CorrecFAST$ is an error average between the prediction time and the actual execution time. This value is updated at each execution as follows:

$$CorrecFAST = \frac{nb_exec \times CorrecFAST + \frac{100\times T_r}{T_{aS_i}}}{nb_exec + 1}$$

where T_r is the actual execution time and T_{aS_i} the time predicted.

Algorithme 3.2 Scheduling algorithm with forecast correction mechanism.

1: $CorrecFAST = 100$
2: $nb_exec = 0$
3: **for all** server S_i **do**
4: **if** `can_do`(S_i, T_a) **then**
5: $T_{aS_i} = \frac{W_{send}}{F_{Bd}} + \frac{W_{recv}}{P_{recv}} + \frac{W_{aS_i}}{F_{S_i}}$
6: $T_{aS_i} = \frac{T_{aS_i} \times CorrecFAST}{100}$
7: $List$=`sort_insert`$(List, T_{aS_i}, T_a, S_i)$
8: num_submit=`task_ack`$(List, \frac{2 \times W_{send}}{F_{Bd}})$
9: T_r=`task_submit`$(List[num_submit])$
10: $CorrecFAST = \frac{nb_exec \times CorrecFAST + \frac{100 \times T_r}{T_{aS_i}}}{nb_exec+1}$
11: nb_exec++

3.3.3 Client-server scheduler with a priority mechanism

Until now, for client-server system, either the deadline scheduling or the priority based scheduling are considered. Here we give an algorithm that utilizes both criteria to select a server for the task.

In Algorithm 3.3, tasks have a predefined priority and deadline. A task is allocated on the server task queue according to its priority. If a task can meet its deadline then it is sent to the server for execution. For Algorithm 3.3, we define some new variables. TD_a is the deadline and TP_a is the priority of task T_a. TF_{aS_i} is the changed execution time of task T_a on server S_i after placing it on the server task queue. To simplify the explanation of the algorithm, we define five functions:

Algorithme 3.3 Client-server scheduler with priority mechanism.

1: **repeat**
2: **for all** server S_i **do**
3: **if** `can_do`(S_i, T_a) **then**
4: $T_{aS_i} = \frac{W_{send}}{F_{Bd}} + \frac{W_{recv}}{P_{recv}} + \frac{W_{aS_i}}{F_{S_i}}$
5: **if** $T_{aS_i} < TD_a$ **then**
6: `count_fallback_tasks`$(T_a, T_{aS_i}, TP_a, TD_a)$
7: **if** $TF_{aS_i} < TD_a$ **then**
8: `best_server`$(S_i$,`best_server_name`)
9: `task_submit(best_server_name,task_name)`
10: `Re-submission(task_name)`
11: **until** the end

`can_do` This function returns `true` if server S_i have the resource required to compute task T_a. This function takes into account the availability of memory and disk storage, the computational library etc.

Count_fallback_tasks This function counts the fall-backed tasks. Tasks that cannot meet their deadline after the insertion of the new task are called fall-backed tasks. Task T_a is placed according to its priority TP_a on the server task queue, which may change the execution time of the tasks on the queue.

best_server This function selects the best server among the servers that can execute the task within the deadline time. The best server is selected by comparing the number of fall-backed tasks. Server with less fall-backed tasks is selected for the task execution. If the servers have same number of fall-backed tasks, then the time to compute the task is compared and the server that takes less time is selected.

task_submit This function performs the remote execution on the server given by `task_ack`. The argument of the function is one server, not a list of servers as in Algorithm 3.1.

Re-submission This function submits the fall-backed task to the servers, for recomputing the execution time. If any server can meet the task's deadline then the task is allocated to that server.

			Execution time on server		
Task	Priority	Deadline	S_1	S_2	S_3
1	3	15	3	5	6
2	5	10	5	12	9
3	2	30	11	20	15
4	4	20	10	np	17
5	5	15	12	14	np

Table 3.1: Priority, deadline and computation time of each task.

Figure 3.1 shows an example to explain the behavior of Algorithm 3.3. Lets consider 3 servers with different capacities and 5 tasks. The priority, deadline, and computation time of each task on each server is shown in Table 1. Execution time is the time taken by the dedicated server to compute the task when the server is free. Computation value np denotes that the task cannot be executed on the server, which maybe due to the type of the task, memory requirement etc. A task is allocated to the server while checking the execution time on each server, its priority and deadline. Task T_1 is allocated to server S_1. Task T_2 is also allocated to server S_1. As its priority is higher it shifts the task T_1, so the execution time of T_1 is changed. If task T_3 is placed on server S_1, it will take less execution time but due to its priority the execution time will be changed. So task T_3 is placed on server S_3. Task T_4 is placed on server S_1, but while doing so task T_1 is fall-backed. Re-submission of task T_1 is done and it is allocated to

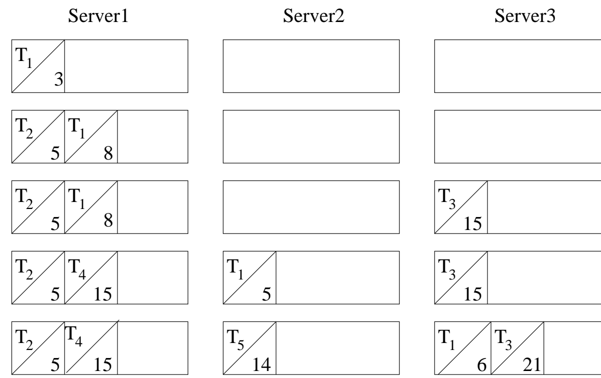


Figure 3.1: Example for priority scheduling algorithm with fallback mechanism. Task id and execution time is written diagonally in each box.

server S_2. Task T_5 is placed on server S_2. As its priority is higher than task T_1, execution time of T_1 changed and the task is then fall-backed. Again re-submission is done and task T_1 is placed on server S_3.

3.4 Simulation results

To simulate the deadline algorithm with priority and the impact of fallback mechanism with this model we use a simulation toolkit called SimGrid [28]. SimGrid provides an excellent framework for setting up a simulation where decisions are taken by a single scheduling process [22, 79].

For experiment, we took 100 servers to execute the submitted tasks. Each task is associated with a priority and deadline. We randomly generate the priority between 1 and 10 and considered tasks deadline to be 5 times of the computation amount needed by the task on dedicated server.

Experiments have been done by fixing the priority of the tasks and varying the priority depending on tasks' size. Figure 3.2 shows when tasks with same priority is submitted the number of executed tasks is less than the tasks executed with random priority. When the number of tasks is less the impact of task priority is negligible. But as the number of tasks increases, tasks' priority plays an important role for increasing the number of tasks executed.

Algorithm 3.3 has been used to check the impact of fallback mechanism on the number of tasks executed under different criteria in Figure 3.3 and Figure 3.4. Figure 3.3(a) shows that the fallback mechanism has no effect if the tasks have the same priority. But in Figure 3.3(b), Figure 3.4(a), and 3.4(b), it can be seen that the fallback mechanism is very useful as the number of submitted tasks (with priority and different sizes) is increased.

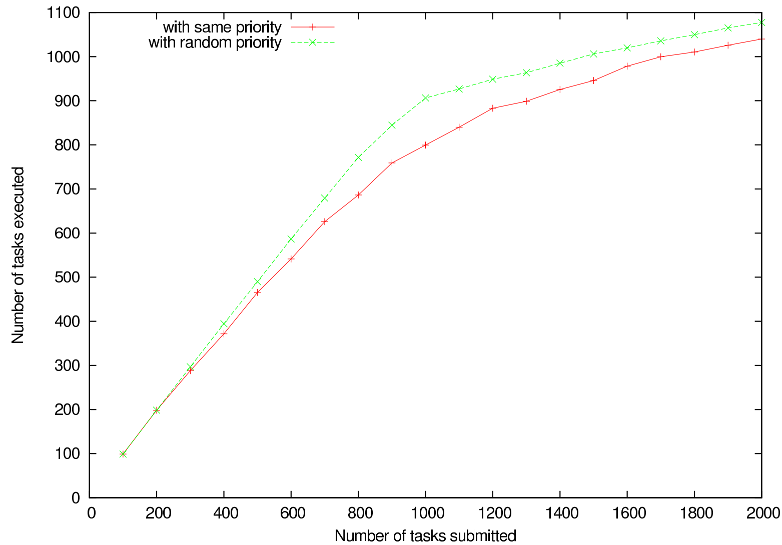


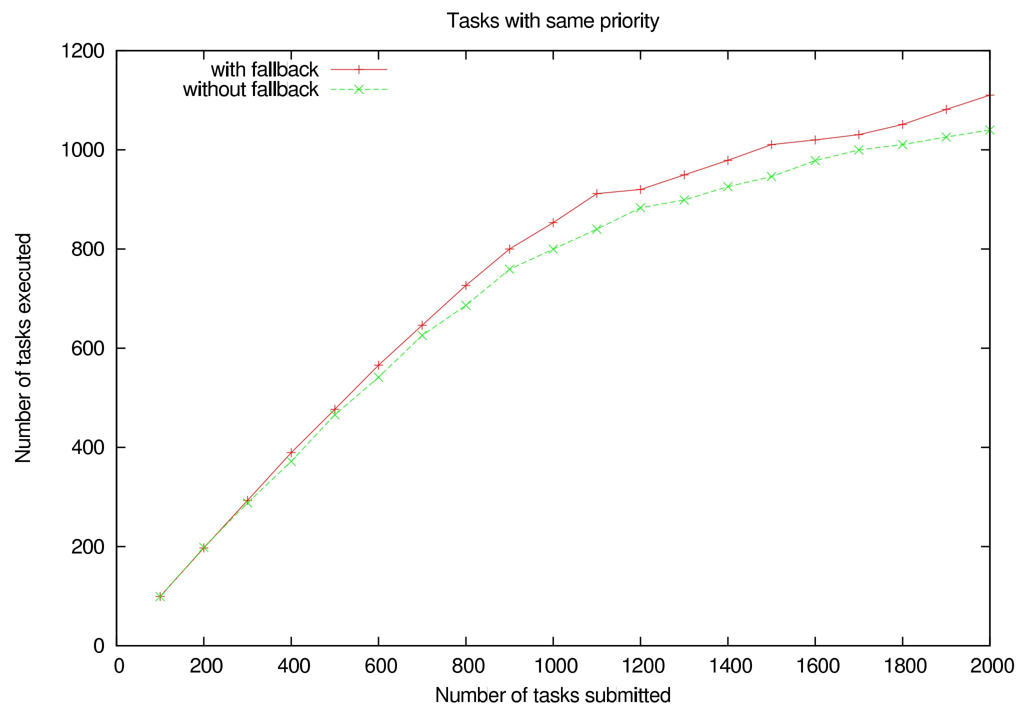
Figure 3.2: Priority based tasks are executed without fallback mechanism.

3.5 Conclusions

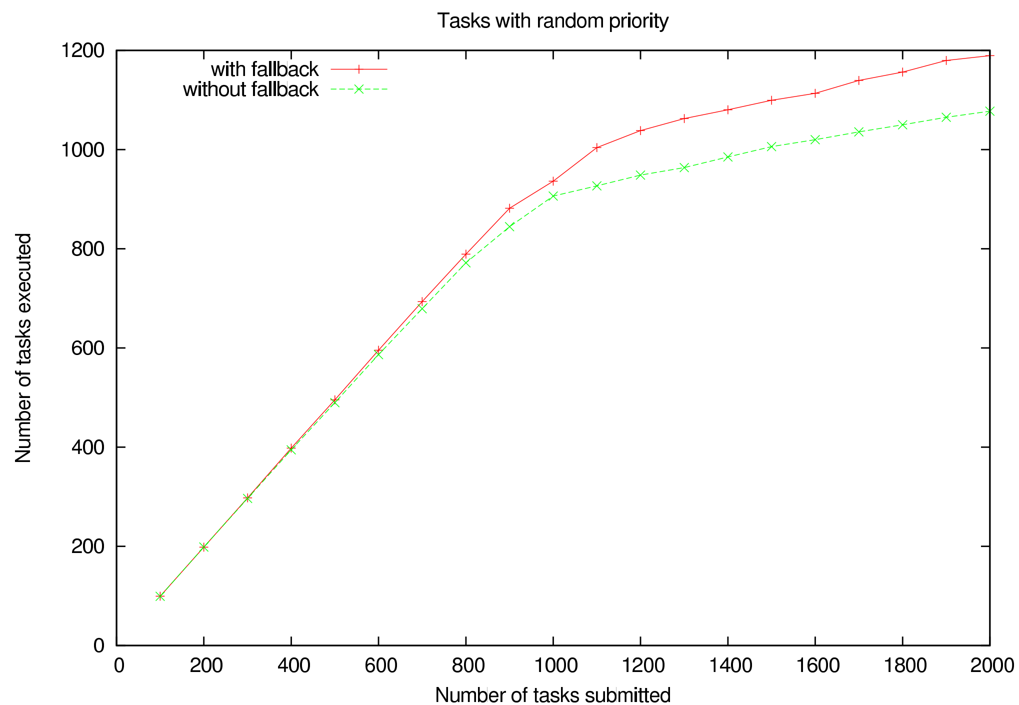
In this chapter we have presented the first step to use NES systems efficiently by providing algorithms for scheduling the sequential tasks on NES systems.

We have shown that the load correction mechanism using FAST [39] and fallback mechanism are efficient to increase the number of executed tasks. We presented an algorithm that considers both priority and deadline of the tasks to select a server. We showed through simulation that the number of tasks that can meet their deadlines can be increased by 1) using task priorities and by 2) using a fallback mechanism to reschedule tasks that were not able to meet their deadline on the selected servers.

Having efficient algorithms to schedule tasks on NES system is an auspicious beginning towards the achievement of best throughput from NES systems. However, resource allocation according to the NES's components is also a very important factor to influence the performance of any NES system. In next Chapter 4 we present the overview of some existing tools that are used for resource allocation.

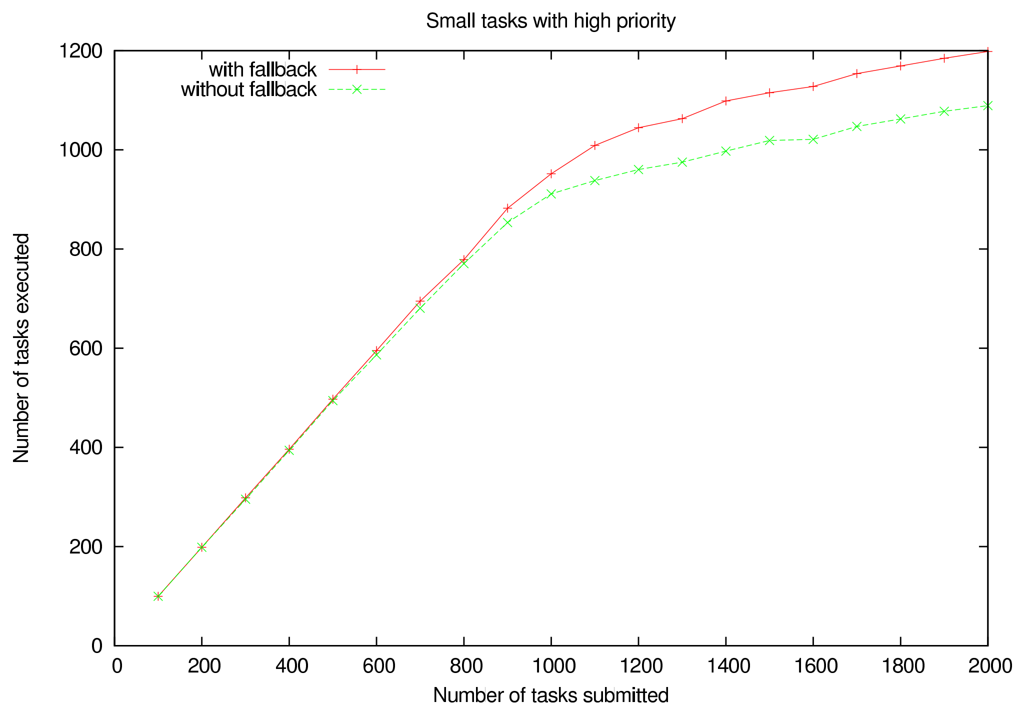


(a) Tasks without priority

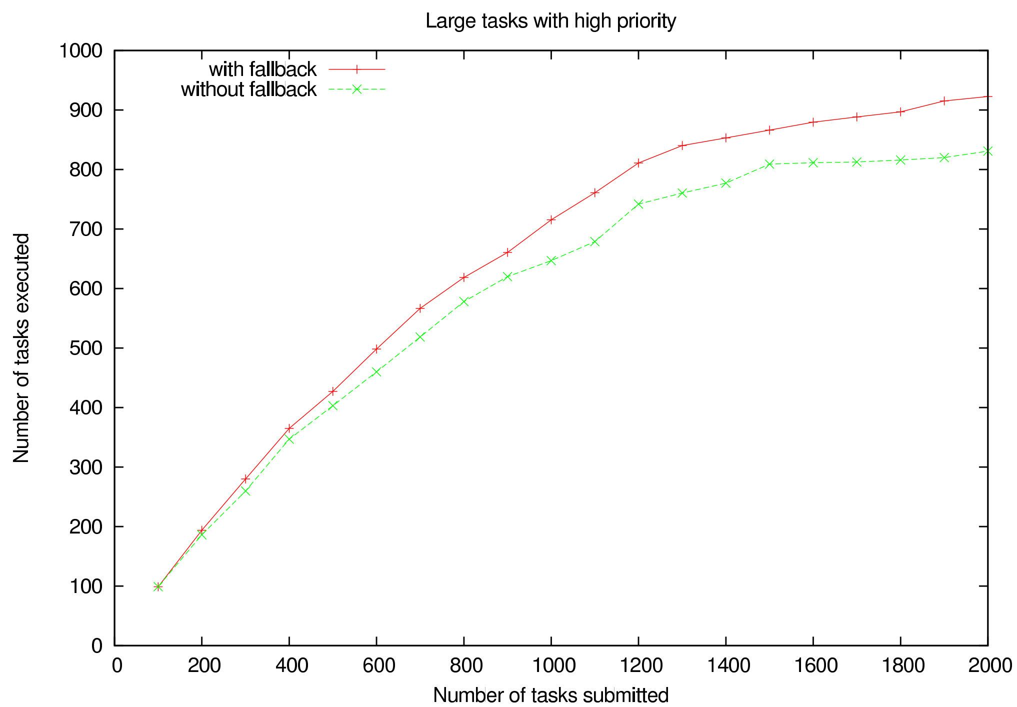


(b) Tasks priority vary between 1-10

Figure 3.3: Comparison of tasks executed with and without fallback based on tasks priority.



(a) Tasks with execution time less than 15 minutes on dedicated servers



(b) Tasks with execution time greater than 4 hours on dedicated server

Figure 3.4: Comparison of tasks executed with and without fallback based on tasks execution duration.

Chapter 4

Deployment

As mentioned earlier the two main factors to use a NES system efficiently are (a) job distribution and (b) resource allocation. In previous Chapter 3 we have presented scheduling algorithms to schedule sequential tasks according to the task's priority and deadline. The algorithms [90] have to be implemented on well deployed NES system on grid resources. But as end users can be biologists, mathematicians, astrophysicists, etc. and for them to select appropriate resource (resource selection) for their application, telling the type and location of the selected resource (resource localization) and mapping their application files (executable, libraries etc) on to the selected resource is very complex task. To help the end users in availing the most from computation grid, an automatic deployment process of applications/ middleware (NES components) on computation grid is required.

Due to the scale of grid platforms, as well as the geographical localization of resources, middleware approaches should be distributed to provide scalability and adaptability. Much work has focused on the design and implementation of distributed middleware. To benefit most from such approaches, an appropriate mapping of middleware components to the distributed resource environment is needed. However, while middleware designers often note that this problem of deployment planning is important, only a few algorithms exist [18, 35, 50, 59, 56] for efficient and automatic deployment.

In this chapter we present some of the existing tools that are used to deploy some specific software/ middleware on set of available resources.

4.1 Introduction

A *deployment* is the mapping of a platform and middleware (applications/software) across many resources. Deployment provides reliability by allowing greater control of the distribution process and increases security as it ensures that only authorized personnel have access to the grid. The tool which deploy the middleware (or application) on to grid or cluster according to deployment plan is referred as *deployment tool*. Input to a deployment tool is the deployment plan which describes which resources should

be used and how they should be connected (directly, hierarchically, etc.) with each other. Deployment is done in phases and order of deployment phases is not rigid. If any problem occurs like selected resource is not reachable, transformation arisen from deployment scheme to the deployment plan etc. then some phases are repeated. But in general the order of deployment phase is as mentioned below.

Initially *Resource discovery* phase, discovers the resources that are compatible with the middleware (application) requirement. *Deployment planning* phase, generates the deployment plan based on middleware (application) description and discovered resources. In *Resource selection* phase, resources are selected according to deployment plan that will host the middleware component (or execution of application). In *Remote files installation* phase , the different parts of each component have to be mapped on some of the selected resources. *Pre-configuration* phase checks the files and other requirements that should be installed and configured before launching the application. *Launch* phase executes the deployment tasks while respecting the precedences described in the deployment plan. *Post-configuration* phase generally launch script(s) for cleaning and killing of processes that were launched.

There exist [7, 18] some deployment tool that takes a user defined deployment plan as input. Selecting a good deployment plan manually according to the middleware (application) and resource description is a complex and very time consuming task. Only expert can do it efficiently in reasonable time. To facilitate the generation of deployment plan a *deployment planning tool* is required. Till now very few deployment planning tools exist and that are also specific for some application (or middleware). The input to the deployment planning tool is a description of the middleware components (or application) and a description of the available resources. Deployment planning tool generates deployment plan by selecting compute nodes and mapping the middleware components (or application processes) onto the selected compute nodes.

In this chapter the two deployment categories: *software deployment* and *system deployment* are presented with some of the existing tools. The comparison of the tools is presented in discussion section.

4.2 Software deployment

Software deployment maps and distributes a collection of software components on a set of resources. Software deployment includes activities such as releasing, configuring, installing, updating, adapting, de-installing, and even de-releasing a software system. The complexity of these tasks is increasing as more sophisticated architectural models, such as systems of systems and coordinated distributed systems, become more common. Many tools have been developed for software deployment based on different approaches. Deployment using mobile agents (eg., Software Dock [50], JADE [18], SmartFrog [48]), or by using AI planning techniques (eg., Pegasus [35, 37], Sekitei [57]) or based on software architecture (eg., JDF). The sequence of these deployment software in this chapter is in a alphabetic order ADAGE, JADE, JDF, Pegasus, Sekitei, SmartFrog, and Software Dock.

4.2.1 Automatic Deployment of Applications in a Grid Environment

Automatic Deployment of Applications in a Grid Environment (ADAGE) [61] is a prototype middleware. It currently deploys only static applications on the resources of a computational grid. Deploying applications can be distributed applications (like CORBA component assembly), parallel applications (like MPICH-G2) or combination of the two applications.

The middleware requires two pieces of information: application description, and resources description. These descriptions are represented as XML documents. Resource description includes compute nodes and their characteristics (operating system, architecture, storage space, memory size, CPU speed and number, etc.) as well as network information (topology, performance characteristics). Using those two pieces of information, ADAGE automatically selects resources which will run the application and maps the application processes onto the selected resources. ADAGE uses either of the two basic scheduling algorithms, round-robin or random. The ADAGE deployment planner selects a job submission method (SSH or Globus2) based on its availability and control parameters. Finally it automatically launches the application processes remotely using the Globus Toolkit (version 2) as a grid access middleware, and initiates the application execution.

Analysis: ADAGE working phases are presented in Figure 4.5. ADAGE implements defined deployment planning given by the user on the resources described in resource description document. For remote installation ADAGE uses `scp` or GridFTP. Adage_deploy launches the processes using the specified remote process creation method (e.g., SSH, Globus, etc.) and configure the application.

4.2.2 JXTA Distributed Framework

JXTA Distributed Framework (JDF) [7] is a framework for automated testing of JXTA-based systems from a single node refereed as control node. JDF provides a generic framework allowing to define custom tests, deploy all the required resources on a distributed testbed and run the tests with various configurations of the JXTA [8] platform.

To use JDF each node should have Java (1.4.x), JXTA 2.2.1 (or before), bourne shell and `ssh` (preferably OpenSSH supporting `ssh` v2) or `rsh`. As JDF is dependent on ssh and bourne shell, it only runs on Unix platforms currently.

JDF requires all information about distribution before hand. Network description file defines the requested JXTA-based network and notion of nodes profile. Node file presents the list of physical nodes and the path of the JVM on each physical node. File transfers and remote control are handled using either `ssh/scp` or `rsh/rcp`. A node (control node) has to be selected to run JDF from. All other physical nodes should be visible from the control node. User submit the JXTA platform that should be tested and the path to the files in the form of an XML file to JDF. JDF is run through a regular shell script which launches a distributed test. This script executes a series of elementary steps: (a) install all the needed files (b) initialize the JXTA network (c) run the specified

test (d) collect the generated log and result files (e) analyze the overall results and remove the intermediate files. Killing of all the remaining JXTA processes can also be done using a kill script.

Analysis: JDF is a deployment tool specific to JXTA. It is a very basic deployment tool, as it has no deployment planning method and the resource selection XML file had to be given by users. JDF uses `scp` to remote file installation and SSH for job launch on deployed platform.

4.2.3 Pegasus

Pegasus [37] is a workflow mapping system that maps an abstract workflow description onto Grid resources. Pegasus uses an AI-based planner (Prodigy Planner [77]) to perform the mapping. Abstract workflow is composed of tasks (application components) and their dependencies (reflecting the data dependencies in the application). Abstract workflow can be created by three different methods. (a) Experienced application developers designs their own executable workflows (workflows already tied to a particular set of resources). (b) By using Chimera, a virtual data system that combines a virtual data catalog with a virtual data language interpreter. (c) By using assistance from intelligent workflow editors such as the Composition Analysis Tool (CAT). The resulting abstract workflow is specified in XML in the form of a DAX (DAG XML description).

Once Pegasus get an abstract workflow, Pegasus [36] consults various Grid information services. Monitoring and Discovery Service (MDS) provides information about the number and type of available resources, static characteristics such as the number of processors and dynamic characteristics such as the amount of available memory. Replica Location Service (RLC) maintains and provides access to mappings between logical names of data items and their target names. Transformation Catalog (TC) determines the location where the computations can be executed. If there are more locations, a location is chosen randomly. Pegasus adds two types of data nodes; transfer nodes and registration nodes. As node name specifies, transfer nodes are used to stage data in or out and registration nodes are used to publish the resulting data products in Grid information services.

Pegasus combine the information from the TC with the MDS information, to make decision about resource assignment. The abstract workflow can only be executed if the input files for the workflow components can be found somewhere in the Grid and are accessible via a data transfer protocol. Pegasus prefers to schedule the computation where the data already exist, otherwise it makes a random choice. Pegasus generates concrete workflow in a form of files which is submitted to DAGMan [46] for execution. Submitted files indicate the operations to be performed on a given remote systems and the order in which the operations need to be performed. DAGMan is responsible for enforcing the dependencies between the jobs defined in the concrete workflow. Jobs can be mapped and executed on a variety of platforms: Condor pools, clusters man-

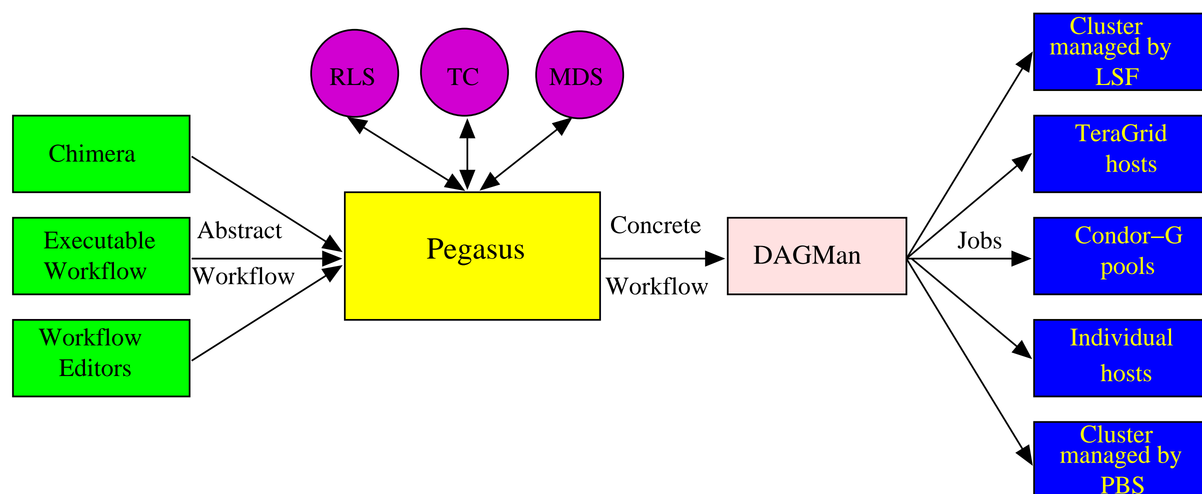


Figure 4.1: Components of a workflow generation, mapping and execution system.

aged by Load Sharing Facility (LSF) [1] or Portable Batch System (PBS) [51], TeraGrid [2] hosts, and individual hosts. Figure 4.1 outline the above explanation.

Analysis: Pegasus uses Prodigy planner to perform the mapping and Grid services like RLS, TC, and MDS for resource selection, remote file installation and preconfiguration. DAGMan performs the launch according to the submitted concrete workflow files.

4.2.4 Sekitei

Sekitei [57] is an AI planner for constrained component deployment in wide area networks. Sekitei is developed to solve the Component Placement Problem (CPP). CPP is concerned with finding a valid component deployment, i.e., a set of components, linkages between them, and a mapping of the resulting DAG onto links and nodes of a wide area network, so that client requirements are satisfied.

The CPP can be viewed as an AI planning problem with resource constraints. The state of the system is described by the availability of interfaces on nodes and placement of components on nodes. This information is described by a set of propositional (boolean) variables. Properties of nodes, links, and interfaces on nodes are described by real-valued resource variables. Operators correspond to placing a component on a node and sending an interface over a link. The CPP goal is translated into a propositional goal of having a component placed on a node. Figure 4.2 describes the process followed to solve the CPP problem. In this process the planner works in conjunction with a component framework. The component framework dictates a component placement problem in XML. This CP problem is translated into a planning problem

[1] http://www.epcc.ed.ac.uk/DIRECT/grid/node45.html
[2] http://www.teragrid.org/

using a compiler. The planner solves the planning problem and gives out a plan. The plan is then decompiled into a deployment plan and fed back onto the component framework. The component framework then uses this deployment plan to deploy the actual components in the network.

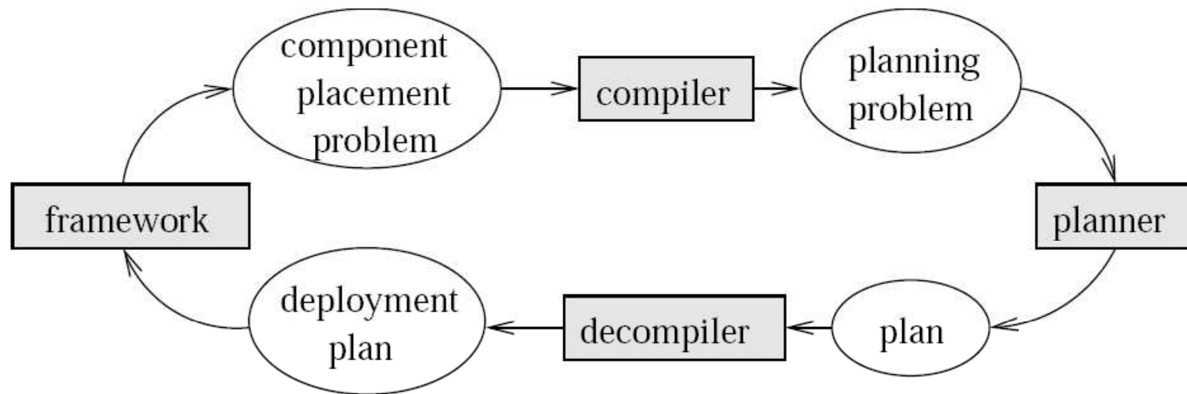


Figure 4.2: Process flow graph for solving CPP [58].

Sekitei [57] limits the search space by combining regression and progression approaches and adding resource checks in layers, so that resource functions are evaluated only for promising operators. Sekitei performs regression (backward search) and progression in the network structure similar to classic planners reasoning about time.

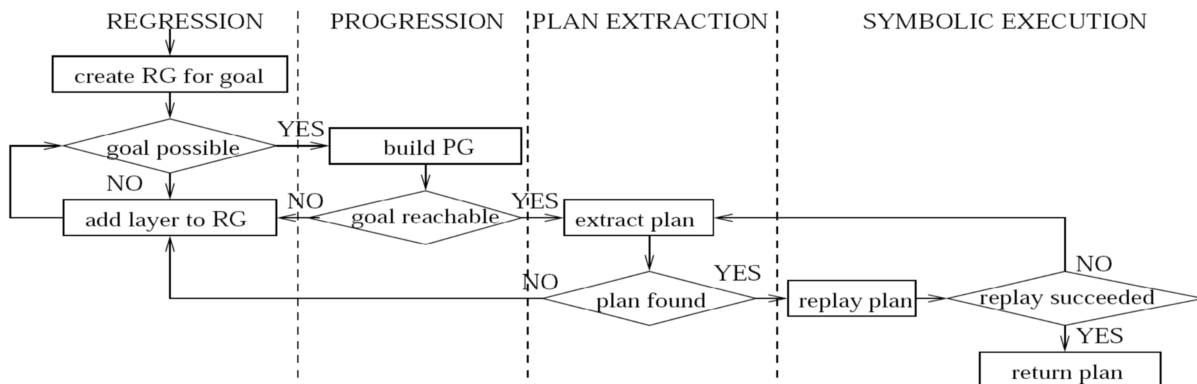


Figure 4.3: Sekitei algorithm phases [55].

The Sekitei algorithm consists of four phases shown in Figure 4.3. Each of the phases solves a relaxed problem. A solution to the relaxed problem is an argument of a new subproblem, which is passed to the next phase of the algorithm. Thus, the regression phase of the algorithm finds a smallest set of possible operators for the original problem with all resource requirements ignored. This set of operators is then used by the progression phase to determine if the goal is reachable given this set of

operators and an aggregated version of resource constraints. If it is not, the algorithm backtracks to the regression phase to obtain a bigger set of possible operators. If the goal is reachable, the progression graph, which contains an aggregated representation of all plans reaching the goal, is passed to the third phase of the algorithm, plan extraction. The plan extraction phase performs a search in the progression graph, and all candidate plans are passed to the last phase of the algorithm for symbolic execution. Success of the fourth stage guarantees that the found plan is correct.

Analysis: Sekitei provides a deployment planner to solve the CPP. Sekitei provides a deployment plan based on CPP and planning problem. Four phases of Sekitei algorithms are repeated till a deployment plan is generated.

4.2.5 SmartFrog

SmartFrog [47] is a framework for describing, deploying, igniting and managing distributed applications. The SmartFrog framework consists of a description language for specifying the configuration of applications, a deployment infrastructure for realizing the application descriptions, a component model to manage the applications through their life cycle, and a set of useful components that populate the framework to support various forms of application behavior.

SmartFrog deployment infrastructure is a Java-based fully distributed network of co-operating daemons that interpret SmartFrog descriptions in order to automatically and correctly instantiate the applications they describe. The SmartFrog deployment infrastructure is a distributed network of components that interprets system descriptions, realizes the systems subcomponents in the correct order, and binds them together. The deployment infrastructure continues to manage the components while the system is running, and is also responsible for the clean, properly sequenced shut-down of the system. If any component fails, the deployment infrastructure can detect this and can be configured to take restorative action, or to shut down the system cleanly. The deployment infrastructure default uses RMI as communication and advertising mechanism, but can use other mechanisms also. Any process can load resources (such as java code)dynamically when needed for the deployment of a SmartFrog application from a central repository such as a web server.

Analysis: SmartFrog is an agent-based application deployment tool. SmartFrog is a very simple and basic tool, as it do file installation on user defined resources. It has no planning for resource discovery and selection for deployment.

4.2.6 Software Dock

The Software Dock framework [50] is an agent-based deployment framework to support the ongoing cooperation and negotiation among software producers themselves and among software producers and software consumers. The architecture of Software Dock is shown in Figure 4.4. The Software Dock employs agents to traverse between software producers and consumers in order to perform software deployment activities by interpreting the descriptions of software systems. The release dock serves

as a release repository for the producer's software systems. The field dock servers as an interface to the consumer site. The field dock standardize the information about consumer organization by creating a common software deployment namespace for accessing consumer-side properties, such as operating system and computing platform.

Initially only an agent responsible for installing the specific software system and the description of the specific software system are loaded onto the consumer site from the originating release dock. Direct communication between agents and docks is provided by standard protocols over the Internet.

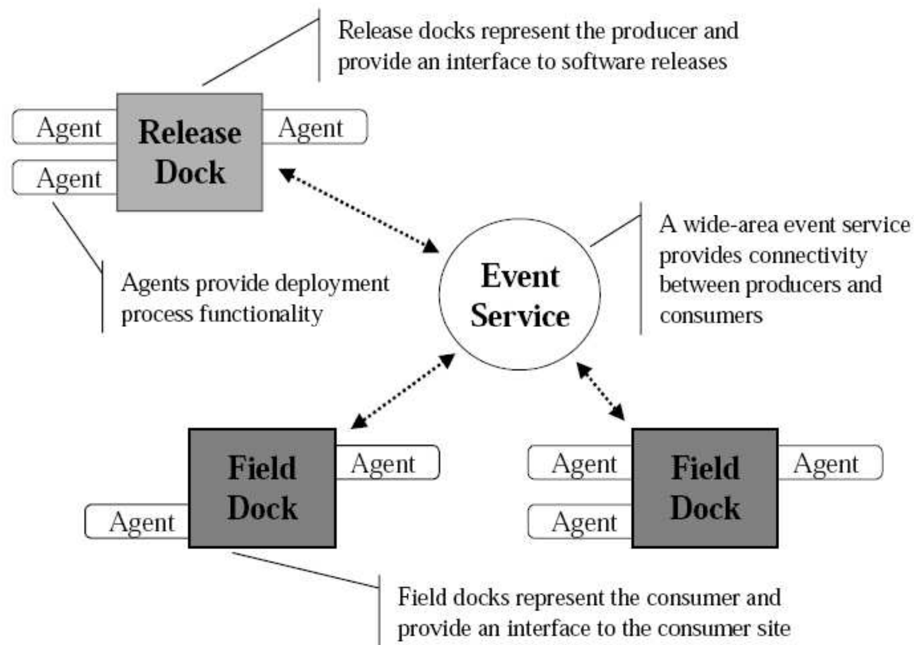


Figure 4.4: Architecture of Software Dock [50].

Each software release is accompanied with generic agents that perform software deployment processes by interpreting the description of the software release. These agents which accompany software releases *dock* themselves at the target consumer site's field dock and perform the respective task (i.e., install or update or reconfigure etc). Similarly, agents associated with deployed software systems can subscribe for the different release events notifications about release-side occurrences. The release dock and the field dock both manage standardized, hierarchical registry of information that records the configuration or the contents of its respective sites and create a common namespace within the framework. Any change in the registry generates an event that agents receive in order to perform subsequent activities.

The wide area event service provides a means of connectivity between software producer and consumer for *push* style capabilities.

Analysis: Software Dock framework performs software deployment activities by interpreting the descriptions of software systems. All resources individually sends the

agents to avail the software installation facility by the agents. Thus, neither resource selection nor planner is used. Mobile agents performs the software installation on the demanded resource.

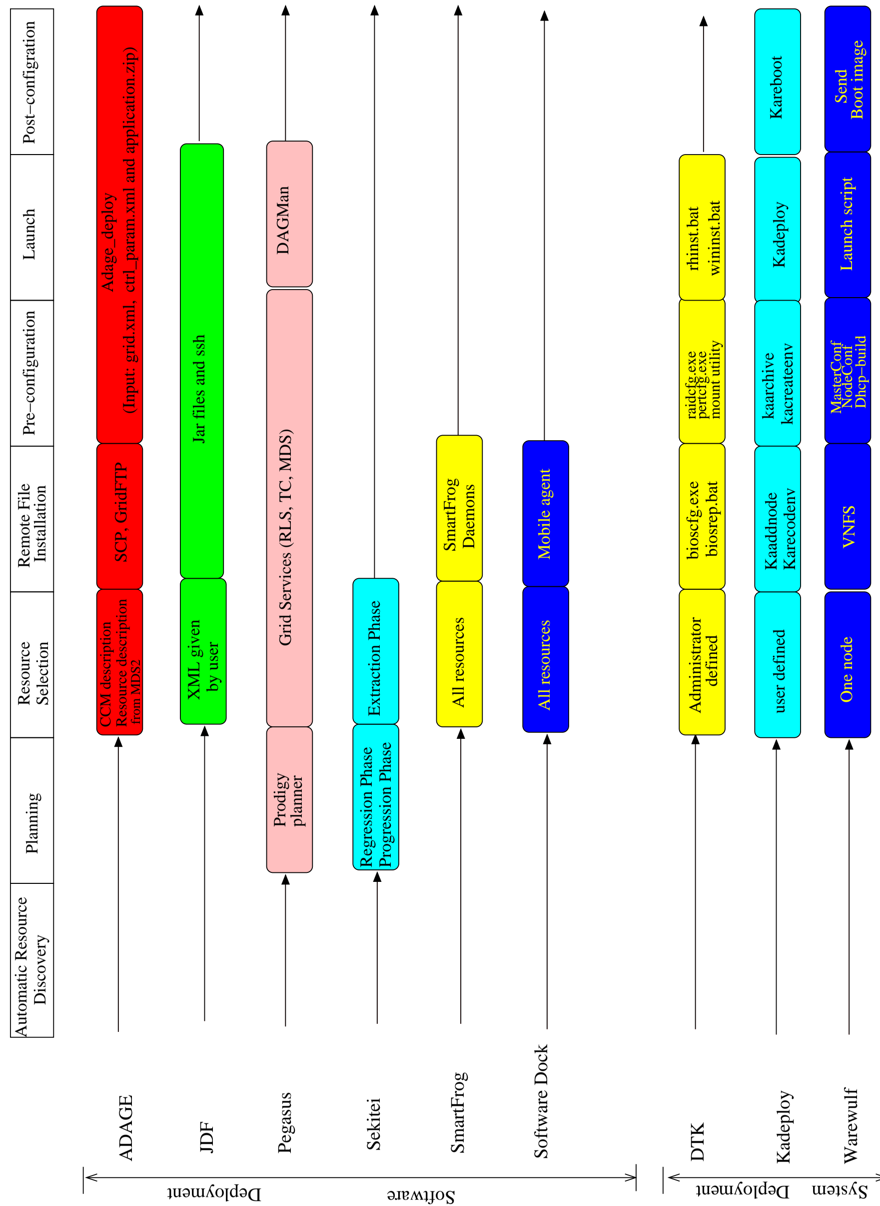


Figure 4.5: Comparison of some deployment tools

4.3 System deployment

System deployment involves two steps, physical and logical. In physical deployment all hardware is assembled (network, CPU, power supply, etc.), whereas logical deploy-

ment is organizing and naming whole cluster nodes as master, slave, etc. As deploying systems can be a time consuming and cumbersome task, tools such as Dell Deployment Toolkit [1], Kadeploy [64] and Warewulf [3] have been developed to facilitate this process.

4.3.1 Dell OpenManage Deployment Toolkit

Dell OpenManage Deployment Toolkit (DTK) performs automated and scripted deployments of Dell PowerEdge servers. DTK comprises a set of MS-DOS utilities, batch scripts, and configuration files.

DTK's utilities can be used as stand-alone tools for configuring individual components such as BIOS, or they can be integrated into scripts for one-to-many mass system deployments. These utilities need at least MS-DOS 6.22 or higher. The DTK's scripts determines the overall execution flow: what executes when, where to find the required files and returns error codes when problems occur. These script are optimized for MS-DOS 7.1 or higher. DTK can now deploy two operating systems, windows (2000 advanced server and 2003 enterprise edition) and Red Hat (Linux 9, Enterprise Linux 2.1 and 3).

DTK deploy operating system in eight steps. (a) *sysinfo.exe* utility acquires system information (configuration and hardware information). (b) Information returned by sysinfo.exe is used by the scripts to perform system specific tasks. (c) BIOS falsh updates and configuration is performed by using the DTK's utilities. *biosflsh.exe* utility updates the system BIOS to a version specified by .hdr file. *bioscfg.exe* utility is used for configuring system BIOS settings and save the settings to a file. (d) Administrator use the saved file with the *biosrep.bat* utility to replicate the model BIOS configuration to multiple systems. (e) The utilities *racadm.exe* and *racconf.exe* are used to configure Dell Remote Assistance Cards (RACs). (f) RAID controllers are configured by using *raidcfg.exe* utility. This utility also creates containers that are used by *pertcfg.exe* utility to create disk partitions that can be used during the OS installation. (g) *mount* utility made available disk partition to MS-DOS deployment tools. (h) *rhinst.bat* and *wininst.bat* scripts are used for scripted OS deployment.

Analysis: DTK is specific for deployments of Dell PowerEdge servers. DTK uses tools to configure each component and scripts for the flow of overall execution. Utility corresponding to each phase of DTK's working is shown in Figure 4.5.

4.3.2 Kadeploy

Kadeploy [63] provides a set of tools for configuration, installation and deployment of an operating system on a set of nodes. Currently it deploys successfully Linux, BSD, Windows, Solaris on x86 and 64 bits computers.

To use Kadeploy, systems should have privileges to execute Kadeploy, nodes' hardware (network interface) must be PXE compliant, this means that the BIOS/EFI should

[3]http://www.warewulf-cluster.org/cgi-bin/trac.cgi

allow to boot from the ethernet card, DHCP server, TFTP server that meets PXE standard and Perl. Kadeploy install the database and uses it to maintain the persistence of the information about the cluster composition and current state.

Kadeploy perform complete deployment in five steps. (a) reboot on the deployment kernel via PXE protocol. (b) pre-installation actions (bench, partitioning and/or file system building if needed etc.) by using preinstallation script (executed before sending the system image). (c) Copy the system environment (system image). (d) post-installation by using post-installation scripts (executed after having sent the system image) and (e) finally, reboot on the freshly installed environment. If the deployment fails on some nodes, these are rebooted on a default environment.

Kadeploy uses different tools to facilitate the deployment. Initially cluster should register current state of its software and hardware composition in database. *Karecordenv* tool is used to register the cluster "software" composition. If any environment is already installed on some partitions of some nodes, it is necessary to register it by giving the appropriate information (image name, image location, kernel path etc.) to the karecordenv tool. *kaaddnode* is used to register the cluster "hardware" composition. The tool registers the information (name and addresses of nodes, disk type and size, partition number and size etc.) in the database. After gathering all information about the cluster nodes, system is ready for deployments.

kaarchive creates an environment image and *kacreateenv* registers the environment image in deployment system and make it available for deployment. Environment is launched by *Kadeploy* which deploy the registered environment on nodes. Finally, cluster node can be rebooted according to requested reboot type using *Kareboot*.

Analysis: Kadeploy is used for configuration, installation and deployment of an operating system on Grid'5000 [4]. Selection of resources for deployment is done by user. Installation and configuration of the files and launch is done some specific Kadeploy tools. Specific tool for each phase is mentioned in Figure 4.5.

4.3.3 Warewulf

Warewulf is a cluster implementation toolkit that manages and distributes Linux to any number of nodes by using a simple master/slave relationship technique.

Warewulf facilitates the process of installing a cluster and long term administration from one master. Warewulf requires at least one network interface for cluster use. This interface will reside on a private physical network that all nodes are connected to and thus automate the distribution of the node file system during node boot. It allows a central administration model for all slave nodes and includes the tools needed to build configuration files, monitor, and control the nodes.

Warewulf has several dependencies [5] (other software that is required on the system for Warewulf to run properly).

Before doing any Warewulf configuration, master node should be fully configured.

[4] https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home

[5] http://www.warewulf-cluster.org/cgi-bin/trac.cgi/wiki/DepInstall

Everything should be working (hardware, software, added correct drivers and configured all network addresses etc.) on master node before slaves (other nodes of the cluster) configuration and installation can be started. The single image is propagated to the nodes via network or CD ROM booting.

Warewulf deploy Linux on cluster in six steps. (a)Install GNU/Linux on the master node (b) Network boot image is created using a standard chroot'able file system referred to as a Virtual Node File System (VNFS) (c) Download and install Warewulf packages. (d) Run Warewulf configuration tools; Masterconf :: configures the master node, Nodeconf :: configures the VNFS, Dhcp-build :: build dhcpd configuration file (e) Start Warewulfd process "/etc/init.d/warewulfd start". (f) Node use etherboot to obtain a boot image via DHCP and TFTP.

Using Warewulf, High Performance Computing (HPC) packages such as LAM-MPI/MPICH, SGE, PVM, etc. can be deployed throughout the cluster. Warewulf is also used for Web clusters (Both HA and load balanced), management of workstations in LAB (similar to the LTSP Project), disk IO clusters, databases, security (active/inline intrusion detection and vulnerability scanning clusters), etc.

Analysis: Warewulf uses master/slave technique for operating system installation on specific nodes. Using the concept of VNFS, the network boot image is created and then from one specific node the image is propagated to other machines. Warewulf uses some tools to configure master node and to build DHCP configuration file.

4.4 Conclusion

No general deployment tool exists, mostly deployment tools deploy specific middleware or application components. For all tools, the information about resources has to be given manually, as none of the deployment tool have automatic resource discovery mechanism. Only two tools have the deployment planning mechanism. Resource selection is mostly user defined or all the discovered resources are selected as to be the part of the platform (that is to be deployed).

System deployment tools have not automatic resource discovery and planning mechanism. The selected resources are defined manually either by user or administrator. Presented system deployment are different from each other in many ways. For example, Kadeploy can deploy many operating systems (Linux, Solaris, Windows) where as Warewulf can deploy only Linux and that also in a master slave fashion. With Warewulf other HPC packages can also be deployed on cluster which is not possible with Kadeploy or DTK. In Warewulf master node provides interactive logins and job queuing to slaves and act as gateway between Internet and cluster network. It provides central management for all nodes. Slave nodes are only available on private cluster network and optimized for computation. Main drawback of Warewulf is a single point failure on master node. Where as Kadeploy and DTK deploy operating system on cluster by copying the boot image on each node of the cluster. Kadeploy deploy operating system n any type of cluster node where as DTK deploy operating system on only Dell server clusters.

From the information presented in this chapter it is clear that a generic deployment tool and a deployment planning tool is needed to transparently avail the power of computational grid and facilitate the end users to easily access computing power of grid.

Part II

Systems used to validate the work

Chapter 5

Distributed Interactive Engineering Toolbox

In previous Part I, we have presented an overview of some existing NES environments, with the scheduling algorithms specific for client-server systems and some deployment tools. In this chapter we presents in detail about a specific grid middleware called **D**istributed **I**nteractive **E**ngineering **T**oolbox (DIET), that we used to verify our deployment planning methods and techniques presented in Part III.

DIET [24] is a toolkit for building applications based on remote computational servers. The goal of DIET is to provide a sufficiently simple interface to mask the distributed infrastructure while providing users an access to greater computational power.

5.1 Introduction to DIET

DIET is developed by GRAAL [1] team at École Normale Supérieure de Lyon. DIET project is started in 1997 and is available for all flavors of UNIX [2] . DIET is implemented using C++ programming language.

DIET consists of a set of elements (agents and servers) that can be used together to build applications using the GridRPC paradigm. DIET has hierarchical arrangement of its components to provide scalability. Communication between components are performed by CORBA. DIET is able to find an appropriate server according to the information given in the client's request (problem to be solved, size of the data involved), the performance of the target platform (server load, available memory, communication performance) and the local availability of data stored during previous computations. Schedulers (agents) are distributed using several collaborating hierarchies connected either statically [33] or dynamically [25].

[1] http://graal.ens-lyon.fr/
[2] http://graal.ens-lyon.fr/DIET

5.1.1 DIET design principles

The aim of DIET is to provide a toolbox that will allow different applications to be ported efficiently over the grid and to allow research teams to validate theoretical results on scheduling or on high performance data management for heterogeneous platforms. DIET design follows below mentioned principles.

Scalability: When the number of requests grows, the agent becomes the bottleneck of the platform. The machine hosting this component (agent) has to be powerful enough but the distribution of the scheduling component is often a better solution. There is of course a trade-off that needs to be found for the number (and location) of schedulers depending on various parameters such as number of clients, frequency of requests, number of servers, performance of the target platform, etc.

Simple improvement: The goal of a toolbox is to provide a software environment that can be easily adapted to match users' needs. Several API have to be carefully designed at the client level, at the server level, and sometimes even at the scheduler level. These API will be used by expert developers to either plug a new application on the grid or to improve the tool for an existing application.

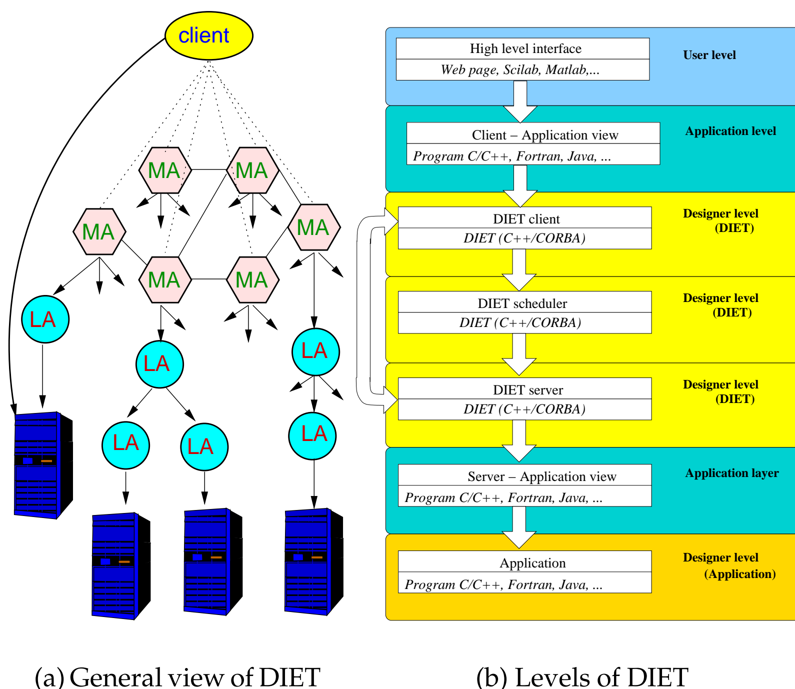
Ease of development: The development of such large environment needs to be done using existing middleware that will ease the design and that will offer good performance at a large scale. DIET uses CORBA as a main low level middleware (OmniORB), LDAP, and several open-source software suites like NWS, OAR, ELAGI, SimGrid, etc.

5.1.2 Architecture of DIET

DIET architecture is an hierarchy of agents to provide greater scalability for scheduling client requests on available servers. The collection of agents uses a broadcast / gather operation to find and select amongst available servers while taking into account locations of any existing data (which could be pre-staged due to previous executions), the load on available servers, and application-specific performance predictions. The main components of DIET [81] are clients, agents and servers, shown in Figure 5.1.

The *client* is the application interface by which users can submit problems to DIET. Users can access DIET via different kinds of client interfaces: web portals, PSEs such as Scilab, or from programs written in C or C++. Many clients can connect to DIET at once and each can request the same or different types of services by specifying the problem to be solved. Clients can use either the synchronous interface where the client must wait for the response, or the asynchronous interface where the client can continue with other work and be notified when the problem has been solved.

The *Master Agent* (MA) serves as the main *portal* to the DIET hierarchy. The MA must be launched before all other agents and servers. All agents and servers are thereafter attached to the MA via a tree at launch-time, though the type of tree is not constrained. All other agents are called *Local Agents* (LA) and can be included in the hierarchy for scalability or to provide a better mapping to the underlying network architecture. *Servers* (SeDs) are attached to each computational resource and either perform



(a) General view of DIET (b) Levels of DIET

Figure 5.1: Architecture of DIET.

the actual computation directly or launch another binary to perform the computation (e.g. in the case of a server on a front-end node of a batch system). Servers also aid in the scheduling process by providing performance predictions to agents during the server selection process. While the top of the tree must be an MA, any number of LAs can be connected in an arbitrary tree and servers can be attached to the MA or to any of the LAs.

5.1.3 Deployment of DIET

The DIET platform is constructed following the hierarchy of agents and SeDs. Initially the naming service should be deployed, so that all the launched elements can easily find each other. To find the element of interest, the searching element needs only to know the port at which the naming service can be found, hostname and a string-based name for the element of interest. As a benefit of this approach, multiple DIET deployments can be launched on the same group of machines without conflict as long as the naming service for each deployment uses a different port and/or a different machine.

Figure 5.2 provides an overview of steps of the DIET's element deployment. After the naming service, the MA is launched; the MA is the root of the DIET hierarchy and thus does not register with any other elements. Then either an LA or a SeD can be deployed as a child of the MA. If an LA is connected to the MA then another LA or SeD

can be connected to this LA. The bottom of the DIET hierarchy consists of SeDs. Once the DIET system is deployed, clients can connect themselves to the MA and submit their task for execution.

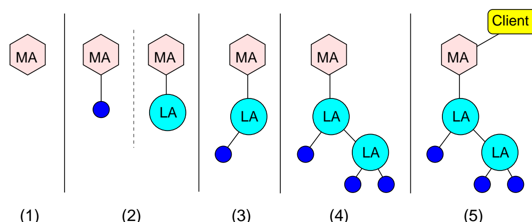


Figure 5.2: Launch of a DIET platform.

5.1.4 Task execution in DIET

Task execution in DIET proceeds in two phases. In the *scheduling phase* the DIET client (1)submits the problem (request) to the MA. Agents maintain a simple list of sub-trees containing a particular service; thus the MA (2) check for all the mandatory information of the task so as (3) broadcasts the request on all sub-trees containing that service. (4) Other agents in the hierarchy perform the same operation, forwarding the request on all eligible sub-trees. When the request reaches the server-level, each server calls their local FAST [39] service to (5) predict the execution time of the request on this server. These predictions are then (6) passed back up the tree where each level in the hierarchy (7) sorts the responses to minimize execution time and (8) passes a subset of responses to the next higher level. Finally, (9,10) the MA returns one or several proposed servers to the client. In the *service phase* the client (11) directly connects the selected server and provides the data for task execution. Once the data is received, a connection between client and server is re-directed. (12) After finishing the task execution, the server re-establishes the connection with the client and the result is sent back.

This description of the agent/server architecture focuses on the common-case usage of DIET. An extension of this architecture is also available [25] that uses JXTA [8] peer-to-peer technology to allow the forwarding of client requests between MAs, and thus the sharing of work between otherwise independent DIET hierarchies.

5.1.5 DIET distributed scheduling

Default DIET uses a Round Robin algorithm, which is based on CPU capacity of the resources to select the appropriate server. In DIET users can use a plug-in scheduler for SeD selection. The DIET plug-in scheduler [81] works in two steps. First an "evaluation table" is generated by each machine. This evaluation table contains the SeD status information, as different parameters and values of these parameters (as shown

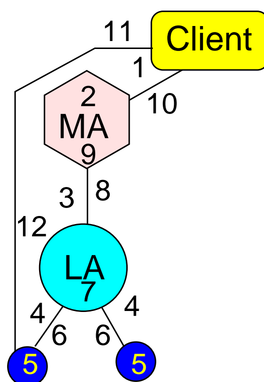


Figure 5.3: DIET task execution steps.

Parameter	Value
Pre_Exe_Time	3.9
$Free_Mem$	5000
CPU_load	12
⋮	⋮

Table 5.1: Example of a plug in scheduler parametric table.

in table 5.1.5). In second step, a selection of the best SeD is done based on the "selection function". This selection function is given by the client and according to this function, a comparison of evaluation tables of different SeDs is done. For example, if one client needs more memory space and has some particular deadline for its request, the selection function given by client will be based upon two parameters, execution time predicted (using FAST [39]) and the free memory space. The SeD that is ranked best based on this selection function will be selected to execute the client request.

5.1.6 Data management in DIET

Data management is provided to allow persistent data to stay within the system for future re-use. This feature avoid unnecessary communication when dependencies exist between different requests.

DIET has two approaches for data management based on Data Tree Manager (DTM) [38] and Juxtaposed Memory (JuxMem) [6] integration with DIET.

DIET components have their own DTM. Initially data for the task is provided to the server. When a task is completed, resulting data can be kept in the environment, thus implementing data persistency. Depending on the clients request, data results can be sent back to the client or kept at the server DTM or both. In the second case, the reference of the data is forwarded to the client. When a DIET server wants to access some data it uses the data reference given by client to request the data from the DTM.

With the JuxMem module, when a task is executed its data is sent to JuxMem and its data reference is sent to the client. The client or server can directly access the JuxMem to pull the required data using the data reference.

DTM is integrated in DIET and so is easy to use, JuxMem is based on P2P, and thus is more reliable.

5.1.7 Fault tolerance in DIET

Fault tolerance in DIET is based on a timer mechanism [26]. In order to take into account servers failure, a time-out is added at the LA level. The value of the timer represents the time during which DIET waits for the first server's answer. If no server responds, the LA replies to its parent that no server is available for the current request. If at least one server answers, a second timer is started. The purpose of this second timer is to fix a compromise between the response time of the scheduler and the aggregation of the answers of different servers. Without this timer, one does not obtain the most effective server but the one which answered most quickly. Both the timers depend on the number of servers. The response time can further be optimized by decreasing the second timer value when the number of server responses increases.

The failure of any agent in the hierarchy could also lead to an infinite wait and the lost of a branch of the scheduler tree. To avoid such a wait, another timer is set, which depends on the depth of the tree.

From the client's point of view, the resource (server) allocated by the MA has to be reserved. Indeed, if two clients send two requests and the chosen server is the same (because this server is available during the FAST [39] prediction step), a conflict can occur. When the second client asks the server for the resolution of its request, the server can be already working on the first client's request. To avoid this problem, a time-stamp mechanism is introduced in the FAST calls, making the resource reservation sequential.

Component crash make different level of effect on the DIET performance. Crash of an LA will lead to the lose of a branch of the DIET hierarchy but requests can be submitted and solved by the other part of the DIET hierarchy. Crash of a server before selecting a server for task execution will not make a worst effect, as best server from the other left servers in the hierarchy can be selected. But a server crash after receiving data from client for request execution will lead to infinite wait for client and if client has some mechanism to know the server's crash then the submitted request has to be submitted again to the DIET hierarchy for execution.

5.2 DIET tools

DIET uses GoDIET [88] for deployment and VizDIET [17] for visualization. DIET uses CORBA for all communication activities, thus DIET can directly benefit from the CORBA naming service and CORBA-based logging service called LogService. The interaction of GoDIET, LogService, and VizDIET is shown in Figure 6.1.

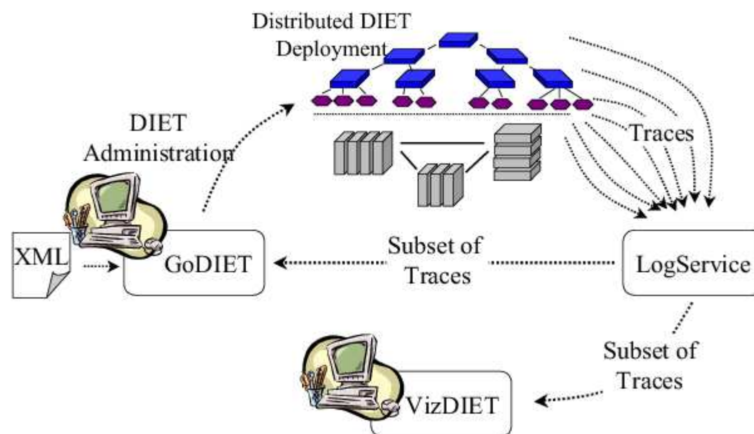


Figure 5.4: Interaction of GoDIET LogService, and VizDIET assist users in controlling and understanding DIET platform.

5.2.1 GoDIET

A deployment of DIET is made using the GoDIET tool [88]. GoDIET provides automatic configuration, staging, execution and management of DIET platform. An XML file describing the hierarchy and requirements is used as an input to GoDIET. For each component to be launched, configuration file is written on local disk including parent agent, naming service location, hostname and/or port endpoint. Configuration file staged remote disk (`scp`) and remote command launched (`ssh`). GoDIET[3] is written in Java.

5.2.2 LogService

LogService is a CORBA-based logging service, that provides interfaces for generation and sending of log messages by distributed components, a centralized service that collects and organizes all log messages, and the ability to connect any number of listening tools to whom LogService will send all or a filtered set of log messages. LogService is robust against failures of both senders of log messages and listeners for log updates. When LogService usage is enabled in DIET, all agents and SeDs send log messages indicating their existence and a special configurable field is used to indicate the name

[3] http://graal.ens-lyon.fr/DIET/godiet.html

of the element's parent. Messages can also be sent to trace requests through the system or to monitor resource performance (e.g. CPU availability on a particular SeD's host or the network bandwidth between an agent and a SeD connected to the agent).

5.2.3 VizDIET

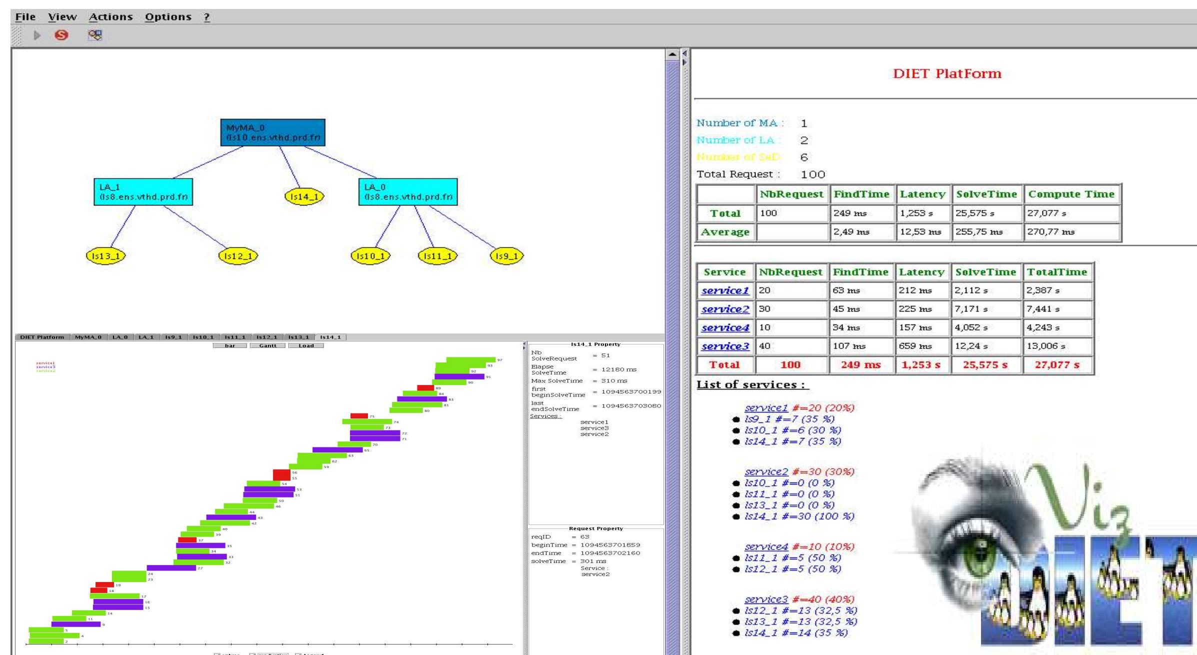


Figure 5.5: Screen shot of VizDiet.

VizDIET [17] is a tool that provides a graphical view of the DIET deployment and detailed statistical analysis of a variety of platform characteristics such as the performance of request scheduling and servicing. VizDIET provides good scalability for DIET platform. VizDIET shows the communication between agents, state of SeD and available services. CPU, memory and network load can also be seen for each node of the DIET platform. To provide real-time analysis and monitoring of a running DIET platform, VizDIET can register as a listener to LogService and thus receives all platform updates as log messages sent via CORBA. Alternatively, to perform visualization and processing post-mortem, VizDIET uses a static log message file that is generated during run-time by LogService and set aside for later analysis. Figure 5.5 presents a screenshot of VizDIET [4].

[4] http://graal.ens-lyon.fr/DIET/vizdiet.html

5.2.4 Fast Agent's System Timer

Fast Agent's System Timer (FAST) [39, 73] is a dynamic performance forecasting tool for grid environments. The goal of FAST is to constitute a simple and consistent Software Development Kit (SDK) for providing client applications with accurate information about task requirements and system performance information, regardless of how theses values are obtained. The library is optimized to reduce its response time, and to allow its use in an interactive environment. FAST is not intended to be a scheduler by itself and provides no scheduling algorithm or facility. It only tries to provide an external scheduler with all information needed to make accurate and dynamic scheduling decisions.

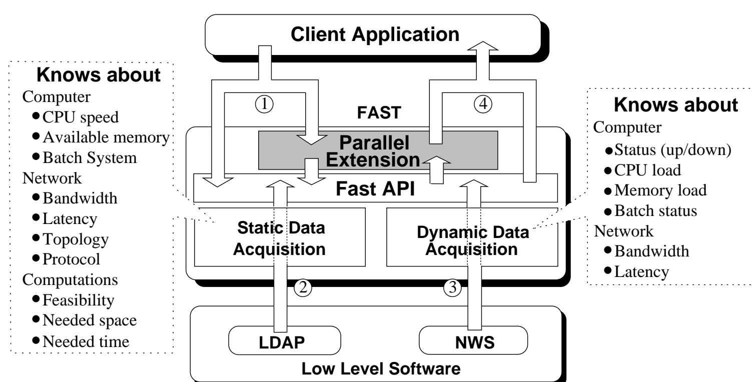


Figure 5.6: Overview of the FAST architecture.

Figure 5.6 gives an overview of FAST's architecture, which is composed of two main modules offering a user API: the *static data acquisition* module and the *dynamic data acquisition* module. The former model forecasts the time and space needs of a sequential computation routine on a given machine for a given set of parameters, while the latter forecasts the behavior of dynamically changing resources, *e.g.,* workload, bandwidth or memory use. FAST relies on low–level software packages. First, LDAP [52] is used to store static data. Then to acquire dynamic data. FAST is essentially based on the NWS (Network Weather Service) [85], a project initiated at the University of California San Diego and now being developed at the University of Santa Barbara. It is a distributed system that periodically monitors and dynamically forecasts performance of various network and computational resources. NWS can monitor several resources including communications links, CPU, disk space, and memory. Moreover, NWS is not only able to obtain measurements, it can also forecast the evolution of the monitored system.

FAST extends NWS as it allows to determine theoretical needs of computation routines in terms of execution time and memory. The current version of FAST only handles regular sequential routines like those of the dense linear algebra library BLAS [42]. However BLAS kernels represent the heart of many applications, especially in numerical simulation. The approach chosen by FAST to forecast execution times of such

routines is an extensive benchmark followed by a polynomial regression. Indeed this kind of routines is often strongly optimized with regard to the platform, either by vendors or by automatic code generation [84]. FAST is also able to forecast the use of parallel routines [27].

5.3 Conclusion

This chapter has presented the overall architecture of DIET, a scalable environment for the deployment on the grid of applications based on the Network Enabled Server paradigm. As NetSolve [29] and Ninf [71], DIET provides an interface to the GridRPC API defined within the Global Grid Forum.

DIET main objective is to improve the scalability of the platform using a distributed set of agents managing a large set of servers available through the network. By being able to modify the number of schedulers, it ables to ensure a level of performance adapted to the characteristics of the platform (number of clients, number and frequency of requests, performance of the target platform). DIET uses VPN (Virtual Private Networks) and CORBA for secure connections between its client and server applications.

Data management and scheduling are also an important part of the performance gain when dependencies exist between requests. The management of the platform is handled by several tools like GoDIET for the automatic deployment of the different components, LogService for the monitoring, and VizDIET for the visualization of the behavior of the DIET's internals.

Next Chapter 6 presents, DIET deployment tool called GoDIET in detail. We have used GoDIET to deploy DIET on resources selected by the deployment planning methods presented in Part III.

Chapter 6

GoDIET: A Deployment Tool for DIET

This chapter presents GoDIET[1] [88], an automated approaches for launching and managing hierarchies of DIET [24] agents and servers across computational grids. Traditionally, users of DIET have either launched agents and servers by hand, or written scripts to manage the launch. These approaches place serious limitations on the diversity, scale, and number of experiments that can be feasibly performed. The launch of DIET on computational grids presents a number of problems shared with many other grid solutions: "how to quickly and reliably launch a large number of components?", "how to interface with a variety of resource environments (ranging from direct execution with `ssh` to launching via a batch system)?", "how to regain control of launched processes to shut them down?", and "how to identify and react to failures in the launch process?". The launch of DIET is further complicated by the fact that agents and servers must coordinate with the rest of the deployment; specifically, the chosen hierarchy is encapsulated in agent and server configuration files and the launcher must ensure that parent agents are fully functional before launching agents and servers that are further down in the hierarchy.

6.1 Working of GoDIET

GoDIET is designed to automate the deployment of DIET platforms and associated services for diverse grid environments. Key goals of GoDIET included portability, the ability to integrate GoDIET in a graphically-based user tool for DIET management, and the ability to communicate in CORBA with LogService [24]. GoDIET is implemented in Java as it satisfies all of these requirements and provides rapid prototyping.

GoDIET automatically generates configuration files for each DIET element while taking into account of user configuration preferences and the hierarchy defined by the user, launches complimentary services (such as a name service and logging services), provides an ordered launch of components based on dependencies defined by the hierarchy, and provides remote cleanup of launched processes when the deployed platform is to be destroyed. Figure 6.1 shows the working step of GoDIET and interaction

[1] http://graal.ens-lyon.fr/DIET/godiet.html

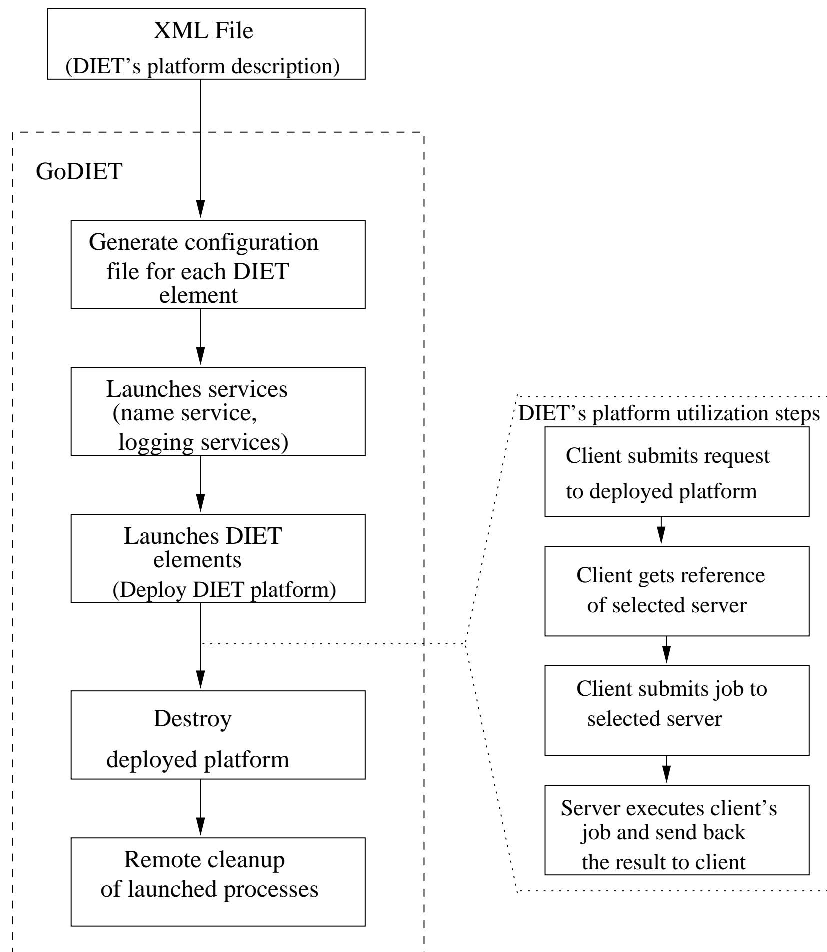


Figure 6.1: Working steps of GoDIET.

steps of a client with DIET deployed platform.

GoDIET basic user interface is a non-graphical console mode and can be used on any machine where Java is available and where the machine has `ssh` access to the target resources used in the deployment. An alternative interface is a graphical console that can be loaded by VizDIET [17] to provide an integrated management and visualization tool. Both the graphical and non-graphical console modes can report a variety of information on the deployment including the run status and, if running, the PID of each component, as well as whether log feedback has been obtained for each component.

`scp` and `ssh` are used to provide secure file transfer and task execution. `ssh` is a tool for remote machine access that has become almost universally available on grid resources in recent years. With a carefully configured `ssh` command, GoDIET can configure environment variables, specify the binary to launch with appropriate command line parameters, and specify different files for the stdout and stderr of the launched process. Additionally, for a successful launch GoDIET can retrieve the PID

of the launched process; this PID can then be used later for shutting down the DIET deployment. In the case of a failure to launch the process, GoDIET can retrieve these messages and provide them to the user. To illustrate the approach used, an example of the type of command used by GoDIET follows.

```
/bin/sh -c ( /bin/echo ✂
    "export PATH=/home/user/local/bin/:$PATH ; ✂
    export LD_LIBRARY_PATH=/home/user/local/lib ; ✂
    export OMNIORB_CONFIG=
/home/user/godiet_s/run_exp01/omniORB4.cfg; ✂
    cd /home/user/godiet_s/run_exp01; ✂
    nohup dietAgent ./MA_0.cfg < /dev/null
> MA_0.out 2> MA_0.err &" ; ✂
    /bin/echo '/bin/echo ${!}' ) ✂
    | /usr/bin/ssh -q user@grid5000.ens-lyon.fr /bin/sh -
```

It is important that each ssh connection can be closed once the launch is complete while leaving the remote process running. If this can not be achieved, the machine on which GoDIET is running may eventually run out of resources (typically sockets) and refuse to open additional connections. In order to enable a scalable launch process, the above command ensures that the ssh connection can be closed after the process is launched. Specifically, in order for this connection to be close-able: (1) the UNIX command nohup is necessary to ensure that when the connection is closed the launched process is not killed as well, (2) the process must be put in the background after launch, and (3) the redirection of all inputs and outputs for the process is required.

DIET provides the features and flexibility to allow a wide variety of deployment configurations, even in difficult network and firewall environments. For example, for platforms without DNS-based name resolution or for machines with both private and public network interfaces, elements can be manually assigned an *endpoint* hostname or IP in their configuration files; when the element registers with the naming service, it specifically requests this endpoint be given as the contact address during name lookups. Similarly, an *endpoint* port can be defined to provide for situations with limited open ports in firewalls. These specialized options are provided to DIET elements at launch time via their configuration files; GoDIET supports these configuration options via more user-intuitive options in the input XML file and then automatically incorporates the appropriate options while generating each element's configuration file. For large deployments, it is key to have a tool like GoDIET to make practical use of these features.

6.2 Identification of failure in element launch

An important aspect of a deploying platform administration tool is the ability to identify and react to errors in the deployment process. When GoDIET identifies an error in the launch of a particular element, it analyzes the elements of the hierarchy that

have not yet been launched and only launches those elements that do not depend on the failed element. For example, if the naming service launch fails, GoDIET will not launch any other elements because all other elements depend on this service. Similarly, if any agent's launch is failed, the bottom subtree hierarchy elements will not be launched. This reactivity saves the user time because time is not wasted trying to launch elements that will fail anyway.

GoDIET can identify errors in the launch process in two ways. First, if errors are reported on standard error during the launch of an element, GoDIET reports these errors to the user and marks the launch of the element as *confused* because GoDIET can not be sure of the status of the element; unlaunched elements are marked with a launch status of *none* and correctly launched elements have a status of *running*. Second, if the user has requested that LogService be launched, GoDIET registers with LogService for traces concerning all running elements. Thus, after the launch of each element GoDIET waits for feedback from LogService concerning the health of the launched element. If LogService does not report that an element has launched successfully, GoDIET marks the log state of that element as *confused*. The verification of launch state is a useful tool for monitoring the health of the deployment because it is always available regardless of whether the user chooses to deploy LogService or not. However, the availability of LogService provides a much stronger verification on launch state because many more types of errors can be caught using this log feedback.

6.3 XML file as an input

As input GoDIET requires an XML file, in which users describe their available compute and storage resources and the desired overlay of agents and servers onto those resources. In short, the GoDIET XML file contains the description of DIET agents and servers and their hierarchy, the description of desired complementary services like LogService, the physical machines to be used, the disk space available on these machines, and the configuration of paths for the location of needed binaries and dynamically loadable libraries. The file format provides a strict separation of the resource description and the deployment configuration description; the resource description portion must be written once for each new grid environment, but can then be re-used for a variety of deployment configurations.

Document Type Definition (DTD) file is used to provide automated enforcement of allowed XML file semantics; an input XML file is verified against the GoDIET DTD using a validating parser.

From Figure 6.2 it can be seen that `diet_configuration` markups surround all of the other sections. `diet_configuration` can optionally contain a `goDiet` section to allow configuration of GoDIET behavior. `diet_configuration` must contain three subsections: the `resources` section, the `diet_services` section and the `diet_hierarchy` section.

In `goDiet` section, four tags can be defined. "debug" controls the verbosity of goDIET output from: 0 (no debugging) -> 2 (very verbose), "saveStdOut" controls

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE diet_configuration SYSTEM "devel/GoDIET-2.0.0/GoDIET.dtd">
<diet_configuration>
  <goDiet debug="1" saveStdOut="no" saveStdErr="no" useUniqueDirs="no"/>
  <resources>
    <scratch dir="/homePath/user/scratch_godiet"/>
    <storage label="g5kBordeauxDisk">
      <scratch dir="/homePath/user/scratch_runtime"/>
      <scp server="frontale.bordeaux.grid5000.fr" login="pkchouhan"/>
    </storage>
    .
    .
    <storage label="g5kOrsayDisk">
      <scratch dir="/homePath/user/scratch_runtime"/>
      <scp server="frontale.orsay.grid5000.fr" login="pkchouhan"/>
    </storage>
    <cluster label="g5kBordo" disk="g5kBordeauxDisk" login="pkchouhan">
      <env path="/homePath/user/demo/bin" LD_LIBRARY_PATH="/homePath/user/demo/lib"/>
      <node label="node-1.bordeaux.grid5000.fr">
        <ssh server="node-1.bordeaux.grid5000.fr"/>
      </node>
                .
                .
      <node label="node-47.bordeaux.grid5000.fr">
           <ssh server="node-47.bordeaux.grid5000.fr"/>
      </node>
    </cluster>
         .
         .
    <cluster label="g5kOrsay" ..>
          .
          .
    </cluster>
  </resources>
  <diet_services>
    <omni_names contact="gdx0005.orsay.grid5000.fr" port="2809">
        <config server="gdx0005.orsay.grid5000.fr" remote_binary="omniNames"/>
    </omni_names>
    <log_central>
        <config server="gdx0007.orsay.grid5000.fr" remote_binary="LogCentral"/>
    </log_central>
    <log_tool>
        <config server="gdx0007.orsay.grid5000.fr" remote_binary="DIETLogTool"/>
    </log_tool>
  </diet_services>
  <diet_hierarchy>
    <master_agent label="MA1">
      <config server="node-5.toulouse.grid5000.fr" remote_binary="dietAgent"/>
      <local_agent label="LA1">
        <config server="node-75.sophia.grid5000.fr" remote_binary="dietAgent"/>
        <SeD label="SeD1">
            <config server="node-65.sophia.grid5000.fr" remote_binary="BLASserver"/>
        </SeD>
           .
           .
        <SeD label="SeD89">
            <config server="node-34.sophia.grid5000.fr" remote_binary="BLASserver"/>
        </SeD>
      </local_agent>
        .
        .
      <local_agent label="LA8">
        <config server="node-30.lyon.grid5000.fr"
                .
                .
      </local_agent>
    </master_agent>
  </diet_hierarchy>
</diet_configuration>
```

Figure 6.2: An example of GoDIET XML file.

whether stdout is redirected to /dev/null or to a file in your remote scratch space called <componentName>.out., "saveStdErr" controls whether stderr is redirected to /dev/null or to a file in your remote scratch space named <componentName>.err., "useUniqueDirs" specifies that a unique subdirectory will be created under the scratch space on each machine for all files relevant to the run. If no, all files are written in the scratch directly.

The `resources` section defines "what machines to use for computation and storage?", "how to access those resources?", and "where to find binaries and libraries on each machine?". It must include at least one "scratch" section, one "storage" section, and one "compute" or one "cluster" section. "compute" tag is used for description of individual machines, "cluster" is to simplify description of large numbers of machines.

The `diet_services` section must contain one "omni_names" section and can optionally include one "log_central" and one "log_tool" section. The `diet_hierarchy` section defines the deployment hierarchy and must include at least one "master_agent" section and one "SeD" section.

6.4 Evaluation of performance and efficacy

Experiments are designed to evaluate the performance and efficacy of GoDIET for the deployment of DIET and associated services at a large scale, the accuracy of GoDIET in identifying errors in the deployment, and the performance of deployments optimized for homogeneous workloads under mixed workload scenarios.

Experiments are done on different sites of Grid'5000, a set of distributed computational resources in France. Table 6.1 provides details of the Grid'5000 sites that we used for our experiments. An abstract example of an XML file used for experiments is shown in Figure 6.2. All tests were performed using the DGEMM application, a simple matrix multiplication provided as part of the Basic Linear Algebra Subprograms (BLAS) package [32].

Site	Cluster	Nodes	Memory	Processor Type
Bordeaux	bordeaux	48	2 GB	dual AMD Opteron 248 2.2GHz
Lille	lille	49	4 GB	dual AMD Opteron 248 2.2GHz
Lyon	lyon	56	2 GB	dual AMD Opteron 246 2.0GHz
Orsay	gdx	216	2 GB	dual AMD Opteron 246 2.0GHz
Rennes	Paraci	64	2 GB	dual Intel Xeon 2.4 GHz
Sophia	azur	103	2 GB	dual AMD Opteron 246 2.0GHz
Toulouse	toulouse	57	2 GB	dual AMD Opteron 248 2.2GHz

Table 6.1: Description of the Grid'5000 clusters used in our experiments.

6.4.1 Evaluation of launch performance

From this experiment we want to test the time required by GoDIET to launch DIET deployments of different sizes. A key activity for GoDIET during launch is to determine when an element has finished launching and registering itself with the naming service; only once these processes are done can GoDIET launch dependent elements. There are two approaches used to define timing of dependent element launch.

Fixed wait: This is the simplest approach, and involves simply sleeping for a fixed period before launching dependent components. Our experiments are performed with a wait of 3 seconds after omniNames and LogCentral, 2 seconds after each agent, and 1 second after each server.

Feedback: This approach uses real-time feedback from LogService to guide the launch process. GoDIET waits for verification that a component has registered with the logging service before launching other components.

For these experiments, the type of DIET hierarchy is fixed and we vary the number of sites (and therefore the number of servers) to be deployed. We obtained access to a subset of the machines at each cluster listed in Table 6.1: 30 nodes at Lyon, 40 nodes at Bordeaux, 40 nodes at Lille, 50 nodes on the Rennes Paraci cluster, 50 nodes at Toulouse, 90 nodes at Sophia, and 140 nodes at Orsay; thus we had 440 nodes in total. We defined our hierarchy to be composed of an MA on the cluster Lyon and an LA on each of the other sites in the deployment; all remaining nodes were allocated as SeDs attached to the LA at their site. To test the performance of GoDIET for different deployment sizes, we first test a deployment using only the smallest site (1 MA, 1 LA, and 28 SeDs total), then a deployment using the three smallest sites (1 MA, 3 LAs, and 102 SeDs total), and so on until we have tested a deployment using all 7 of the sites (1 MA, 7 LAs, and 426 SeDs total). We test the time required for GoDIET to launch each of these deployments using the *fixed wait* approach and using the *feedback* approach. Fig. 6.3 presents the time required to launch these different platforms.

The time for launch is strongly dependent on the number of servers included in the deployment. The cost of the *fixed wait* approach is clearly high relative to the cost of the *feedback* approach in this environment. Grid'5000 benefits from top-level machines and networks and relatively little competition for resources among users; in this environment our selection of fixed wait times of several seconds between elements is likely overgenerous and the *feedback* approach receives notifications very quickly. However, in more difficult network environments these times may be too short. The *feedback* approach is clearly preferable because it automatically controls dependent element launch based on feedback from previous launches and so does not require user intervention to guess the correct amount of wait time.

6.4.2 Launch problem identification

Once a platform has been launched by GoDIET, we want to verify that the platform was indeed correctly launched. We are also interested in "how accurately GoDIET identifies problems in the platform based on launch-time errors and/or LogService

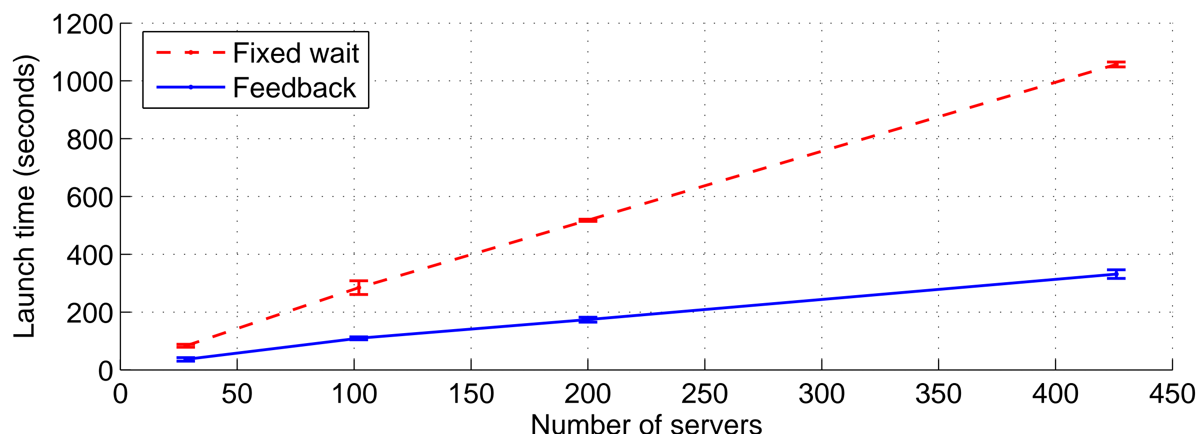


Figure 6.3: The time for platform launch as a function of the number of servers desired.

feedback?". Thus, we did platform verifications on all deployments from Section 6.4.1.

For these experiments we take advantage of one of the DIET scheduling modes that provides round-robin allocation of tasks to SeDs. To test the performance of a deployment with S servers, we run S clients against that deployment. If we have S valid scheduler responses from the agent hierarchy, then we consider that the agent hierarchy is running well. We then verify that each of the S requests was assigned a unique server (as required by our choice of round-robin). If so, and if the servicing phase of all S requests completed successfully, we consider all S servers to be functioning correctly. If less than S unique servers were used to service the S requests, we know some servers or their parent agents were not functioning correctly.

Note that DIET provides best-effort round-robin scheduling and under cases of high rates of arrival of competing client requests, sometimes DIET is not able to provide strict round-robin behavior. Thus, to ensure strict round-robin behavior our testing script only launches a single client at a time; under these conditions, round-robin

Approach	Sites	Rep. 1	Rep. 2	Rep. 3
Fixed wait	1	0/0	0/0	0/0
	3	0/0	0/0	1/40
	5	1/50	0/49	0/39
	7	2/91	1/129	-
Feedback	1	0/0	0/0	0/0
	3	1/1	0/0	0/0
	5	0/0	0/0	0/0
	7	2/30	1/29	-

Table 6.2: Problem SeDs as identified by GoDIET / by clients.

behavior is strictly enforced by DIET.

For each deployment we also identified how many errors GoDIET reported to the user, for comparison against the number of errors seen via the above verification test. In the *fixed wait* approach a server is marked as failed if its launch seemed to have failed, while in the *feedback* approach a server is marked as failed if either its launch failed or log feedback was received for it.

Table 6.2 provides a summary of our analysis of problematic SeDs in the deployments from the previous section. Rep.# refers to number of hierarchy deployment repetition. If values in Rep.# column is represented as x/y then,"x" represents the number of errors for SeDs seen by GoDIET and "y" represents the number of errors found by the client submission verification tests. An "-" means couldn't get the experiment.

The majority of deployments had no errors at all: all agents and SeDs were functioning correctly; this result is promising in terms of the stability and usability of Grid'5000 since such large resource sets typically have some number of unexpected run-time problems. However, when there are problematic SeDs, GoDIET does not effectively estimate the scale of the problem. In fact, when there are problems they are often large with as many as 129 problematic SeDs. In all cases, these large numbers of problems correspond to the size of one of the individual sites in our deployment, so we believe that these errors are site-level errors corresponding to a failed local agent at that site or a transitory problem with the site itself (such as a temporary network partition). In any case, we plan to improve GoDIET to better identify these problems so that DIET administrators and users can be better informed about the platform status.

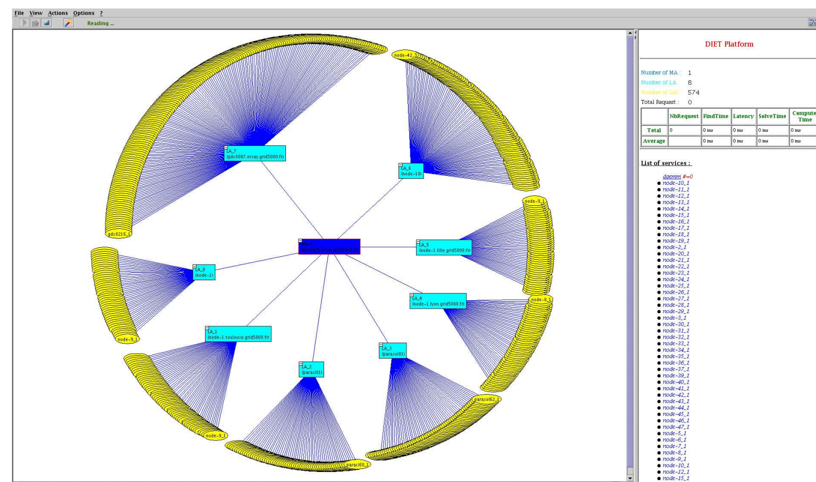


Figure 6.4: Grid'5000 DIET Deployment with 1 MA, 8 LAs, 574 SeDs.

Cluster	SeDs	Launch time (secs)
Bordeaux	44	70
Lille	44	118
Lyon	49	52
Orsay	176	88
Paraci	59	82
Parasol	59	81
Sophia	89	68
Toulouse	54	33

Table 6.3: Time taken to launch 574 SeDs is 592 secs.

6.4.3 DIET deployment on 585 nodes

From the results of different GoDIET launch approaches, we have selected the feedback approach to deploy DIET on a hierarchy of 585 nodes. This experiment was done on 7 sites/8 clusters (Bordeaux, Lille, Lyon, Orsay, Rennes, Sophia, Toulouse) of Grid'5000. OmniNames and log_central were launched at Orsay. The hierarchy is composed of an MA on the cluster Orsay and an LA on each of the 8 clusters. All other reserved nodes were added as servers to the LA on that cluster. GoDIET was located on a machine in the Lyon cluster. The time taken to launch this deployment is *7minutes 25secs*. The time taken to launch SeDs on each cluster is shown in Table 6.3. By comparing the time taken to launch the deployment and the sum of the launch time of SeDs on each site, it can be seen that parallel launching of elements is done by GoDIET on different sites and saved overall launch time.

6.5 Conclusion

We have presented an overview and experimental results of GoDIET [88], an automated approaches for launching and managing hierarchies of DIET [24] agents and servers across computational grids. GoDIET automatically generates configuration files for each DIET component taking into account user configuration preferences and the hierarchy defined by the user, launches complimentary services (such as a name service and logging services), provides an ordered launch of components based on dependencies defined by the hierarchy, and provides remote cleanup of launched processes when the deployed platform is to be destroyed.

Part III

Middleware Deployment Planning

Chapter 7

Heuristic to Structure Hierarchical Scheduler

To efficiently use the computing power of distributed resources, middlewares are developed. However, the method to deploy the middleware elements on distributed resources is mostly defined by user or by administrator. Deployment tools mentioned in Chapter 4 deploy a given platform. None of the tool can select appropriate resources for deployment from a pool of resources. Selection of a resource among a set of resources for a particular middleware element is based on the knowledge about the middleware and the resource. As the throughput of deployed platform depends on the type of application being submitted to the platform. So for end users it becomes very cumbersome task to select a good middleware deployment depending upon the characteristics of the application and power of the resources. An expert may select a good deployment in reasonable time.

In this chapter we presents an heuristic to deploy middleware on homogeneous resources. We call this process as *deployment planning*, a planning needed to arrange the resources in such a manner that when deployment of middleware components will be done, maximum number of requests can be processed by this arrangement (platform). Validation of our deployment planning heuristic is done by implementing the heuristic to structure a hierarchical middleware.

7.1 Introduction

The issue of deployment planning for arbitrary arrangements of distributed schedulers is too broad. We focus on hierarchical arrangements, because a hierarchy is a simple and effective distribution approach and has been chosen by a variety of middleware environments as their primary distribution approach [24, 34, 49, 74]. Before trying to optimize deployment planning on arbitrary, distributed resource sets, we target a subproblem: "what is the optimal hierarchical deployment on a cluster with hundreds to thousands of nodes?". This problem is not as simple as it may sound: one must decide "how many resource should be used in total?", "how many should

be dedicated to scheduling or computation?", and "which hierarchical arrangement of schedulers is more effective i.e., more schedulers near the servers, near the root agent, or a balanced approach?".

In practice, running a throughput test on all deployments to find the optimal deployment is unmanageable; even testing a few samples is time consuming. In this chapter we provides efficient and effective planning for identifying a good deployment that will perform well in homogeneous cluster environments. We consider that a *good deployment* is one that maximizes the steady-state throughput of the system, i.e., the number of requests that can be scheduled, launched, and completed by the servers in a given time unit. Deployment planning heuristic determines that in which hierarchical organization nodes should be arranged so that the goal to maximize steady-state throughput can be achieved for each node. We also provided algorithms to modify the obtained hierarchy to limit the number of resources used to the number of available.

Deployment planning heuristic to deploy NES on homogeneous cluster is based on the constraint to maximize the throughput of agent with respect to its children. We used DIET to validate our deployment heuristic. First we develop detailed performance models for DIET system and validate these models in a real-world environment. We then present real-world experiments demonstrating that the deployments automatically derived by our approach are in practice nearly optimal.

7.1.1 Operating models

The architectural model that we consider is a tree-shaped platform. Processors in the platform can perform three operations in parallel or serial manner. Depending upon the operating way of the processor authors in [14] have mentioned six different architectural models, shown in Figure 7.1, where "*r*" stands for *receive*, "*s*" stands for *send* and "w" stands for *work*, i.e. *compute*. In Figure 7.1, when two squares are placed next to each other horizontally, it means that only one of them can be accomplished at a time, while vertical placement is used to indicate that concurrent operation is possible. "$\|$" (respectively ",") is used to indicate parallel (sequential) order of operations in the models.

$M(r^*\|s^*\|w)$**: Full overlap, multiple-port** - In this first model, a processor node can simultaneously receive data from its parent, perform some (independent) computation, and send data to all of its children. This model is not realistic if the number of children is large.

$M(r\|s\|w)$**: Full overlap, single-port** - In this second model, a processor node can simultaneously receive data from its parent, perform some (independent) computation, and send data to one of its children. At any given time-step, there are at most two communications taking place, one from the parent and/or one to a single child.

$M(r\|s,w)$**: Receive-in-Parallel, single-port** - In this third model, as in the next two, a processor node has one single level of parallelism: it can perform two actions

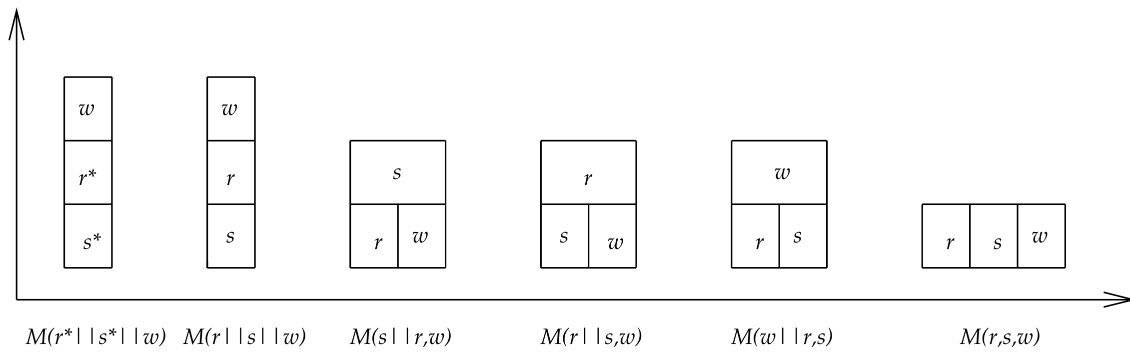


Figure 7.1: Classification of the operating models.

simultaneously. In the $M(r\|s, w)$ model, a processor can simultaneously receive data from its parent and either perform some (independent) computation, or send data to one of its children. The only parallelism inside the node is the possibility to receive from the parent while doing something else (either computing or sending to one child).

$M(s\|r, w)$**: Send-in-Parallel, single-port** - In this fourth model, a processor node can simultaneously send data to one of its children and either perform some (independent) computation, or receive data from its parent. The only parallelism inside the node is the possibility to send to one child while doing something else (either computing or receiving from the parent).

$M(w\|r, s)$**: Work-in-Parallel, single-port** - In this fifth model, a processor node can simultaneously compute and execute a single communication, either sending data to one of its children or receiving data from its parent.

$M(r, s, w)$**: No internal parallelism** - In this sixth and last model, a processor node can only do one thing at a time: either receiving from its parent, or computing, or sending data to one of its children.

We have considered the sixth model “No internal parallelism“ to present our heuristic.

7.2 Architectural model

The target architectural framework is represented by a weighted graph $G = (V, E, w, B)$. Each $P \in V$ represents a computing resource with computing power w measured in MFlop/second. There are one or more client nodes P_c that generate requests for the hierarchy. Each link between two nodes is labelled by the bandwidth value B in Mb/second.

Platform nodes: Available nodes can be divided in two types, agents and servers. $\mathbb{A}$ is a set of agents. Agents are the scheduler nodes that coordinate incoming requests and communicate the requests to the appropriate servers. $\mathbb{S}$ is a set of servers. Servers provide services to clients. Each node $P \in V$ can be assigned to either one server or one agent. When responses are generated by the servers the agents may coordinate the response. Client nodes are extra then the available nodes for the construction of hierarchy. Thus, $|\mathbb{A}| + |\mathbb{S}| = |V|$.

Task processing: The size of the request forwarded by the agent to its children is S_{in} and the size of the response (reply of the request) by each node is S_{out}. The measuring unit of these quantities is Mb/request. The amount of computation needed by $P \in \mathbb{A}$ to process one incoming request is denoted by W_{in} and the amount of computation needed by $P \in \mathbb{A}$ to merge the responses from its children is denoted by W_{out}. In short W_{out} is the time needed for sorting the servers. $W_{X_{Ser}}$ is the amount of computation needed by $P \in \mathbb{S}$ for X_{ser} service. W_{pred} is the amount of computation needed by $P \in \mathbb{S}$ for predicting the computation time of each incoming request.

7.3 Deployment constraints

We consider the steady-state techniques concept [16] for our model because we are interested in maximizing steady-state throughput. In steady-state techniques, the performance of the startup and shutdown phases are not considered. The initial integer formulation is replaced by a continuous or rational formulation. The main idea is to generate the hierarchy when each resource is performing all (sending, receiving and computation) tasks during each time-unit.

The overall throughput of the system depends on the bandwidth of the links, message size of requests, time spent computing by each node on each request, and the computing power of the nodes. Therefore, we have the following constraints:

Computation constraint for agent: As an agent uses its computing power for treating two type of jobs (requests and responses), the throughput of agent according to its computing power can be derived from Equation (7.1).

$$\forall P \in \mathbb{A} : \frac{w}{W^{in} + W^{out}} \tag{7.1}$$

So we re-organize Equation (7.1) to represent the number of requests computed by an agent as $Node_{comp}$ in Equation (7.2). The following equation states that the sub-tree throughput can not be larger than the throughput of the agent.

$$Node_{comp} \leq \frac{w}{W_{in} + W_{out}}, \forall P \in \mathbb{A} \tag{7.2}$$

Communication constraint for agent: Depending on the physical network, bandwidth link is either shared by incoming and outgoing data or bandwidth is fully duplex. Depending on the type of bandwidth utilization either of the two equations represented as Equation (7.3) will be used to calculate the number of requests transmitted

per time-unit along each link.

$$Comm_{req} \leq \frac{B}{S_{in} + S_{out}} || min(\frac{B}{S_{in}}, \frac{B}{S_{out}}), \forall links \tag{7.3}$$

Computation constraint for server: As server also use its computing power in treating two jobs (prediction and real execution), so the number of requests that can be calculated by a $P \in \mathbb{S}$ in a time step is given by Equation (7.4).

$$Server_{comp} \leq \frac{w}{W_{X_{ser}+W_{pred}}}, \forall P \in \mathbb{S} \tag{7.4}$$

Servers have no communication constraint because servers are the leave of the tree and servers have no child to communicate. Servers communication with its parent agent is taken into consideration in agent's communication constraint.

7.4 Deployment construction

For hierarchical deployment construction, we consider that when the response sent by an agent and a server is same in format and size then an agent can have same number of children irrespective of children type (server or agent). But if the format and size of the reply from agents and servers is different then number of servers supported by an agent will be different from the number of agents supported by an agent. For example, if server sends the task's prediction with its status as reply to its parent agent and an agent just sent the list of sorted servers to its parent agent then the number of agents supported by the parent agent will be different from the number of server children supported by it. The maximum nodes that can be connected to agent nodes without making it bottleneck can be calculated from the constraints mentioned in Section 7.3 as shown below.

Number of servers supported by an agent: MSPA (Maximum Servers Per Agent) represents the maximum number of servers that can be supported by a node $P \in \mathbb{A}$.

$$\textbf{MSPA} = \frac{Node_{comp}}{min(Server_{comp}, Comm_{req})} \tag{7.5}$$

Number of agents supported by another agent: The maximum number of agents that can be added to another agent is given as **MAPA** (Maximum Agents Per Agent).

$$\textbf{MAPA} = \frac{Node_{comp_i}}{min(Node_{comp_j}, Comm_{req})}, P_i, P_j \in \mathbb{A} \tag{7.6}$$

We differentiate P_i and P_j as two different nodes even though they are from same set because this equation can be used to calculate the number of intermediate nodes supported by the other nodes by differing the set of P_i and P_j for some NES like DIET.

As the throughput of the platform can be a rational, so the values of MAPA and MSPA are not necessarily integers, they could be rational too. To make these values integers we use either round up or round down. The simple method that we consider

is according to the available number of nodes. If we have enough available nodes we round up, otherwise we round down.

Level of the hierarchy is denoted by l. The total number of nodes required for the hierarchy is **n_req** and can be calculated as follows.

$$\textbf{n_req} = \sum_{k=0}^{l} MAPA^k + MAPA^l \times MSPA \tag{7.7}$$

The hierarchy is constructed using Algorithm 7.1. This algorithm uses Algorithm 7.2 and Algorithm 7.3 to use no more than the number of available nodes.

Algorithme 7.1 Construction Algorithm

1: calculate MSPA using Equation (7.5)
2: calculate MAPA using Equation (7.6)
3: $l = 0$, n_req=0
4: **while** n_req $<$ n **do**
5: $\quad n_req_{low} = n_req$
6: $\quad$ calculate n_req using Equation (7.7)
7: $\quad l++$
8: $l--$, $n_req_{high} = n_req$
9: **if** $(n - n_req_{low}) > (n_req_{high} - n)$ **then**
10: $\quad$ call Make_Hierarchy
11: $\quad$ call Node Removal Algorithm 7.2 with $extra_nodes = n_req_{high} - n$
12: **else**
13: $\quad l--$
14: $\quad$ call Make_Hierarchy
15: $\quad$ call Node Addition Algorithm 7.3 with $extra_nodes = n - n_req_{low}$

Procedure: Make_Hierarchy

1: Add an agent
2: **while** $l > 0$ **do**
3: $\quad$ add MAPA agents to (lowest level) agent without children
4: $\quad l--$
5: add MSPA servers to (lowest level) agent without children

7.5 Model implementation for DIET

We make some assumptions to use DIET for heuristic validation. The MA and LA are considered as having the same performance. We assume that an agent can connect either agents or servers but not both. Root of the tree is always an MA. Thus all clients will submit their request to the DIET hierarchy through one MA. When client requests are sent to the agent hierarchy, DIET is optimized such that large data items

Algorithme 7.2 Node Removal Algorithm

1: **while** ($extra_nodes > 0$) **do**
2: **if** $extra_nodes < MSPA$ **then**
3: remove $extra_nodes$ from the one of the bottom agents. Exit
4: **else if** ($extra_nodes == MSPA \;||\; extra_nodes == MSPA + 1$) **then**
5: remove one bottom agent with all its servers. Exit
6: **else**
7: $remove_agent = extra_nodes$ mod MSPA
8: **if** $remove_agent > (MAPA \times MSPA) + MAPA$ **then**
9: remove $MAPA$ agents with all its servers.
10: add MSPA /* servers to the agent that now have neither agent nor server connected to it*/
11: $left_nodes = extra_nodes - ((MAPA \times MSPA) + MAPA) + MSPA$
12: $extra_nodes = left_nodes$
13: **else**
14: remove $remove_agent$ agents with all its servers
15: $left_nodes = extra_nodes - ((remove_agent \times MSPA) + remove_agent)$
16: $extra_nodes = left_nodes$

Algorithme 7.3 Node Addition Algorithm

1: **while** ($extra_nodes > 1$) **do**
2: **if** all bottom-level agents have maximum number of servers **then**
3: add one node to the bottom agent
4: add parent servers as his servers
5: $extra_nodes - -$
6: **else**
7: **if** $extra_nodes \leq MSPA$ **then**
8: add one agent to the agent having child agents < MAPA
9: add $extra_nodes - 1$ servers to this newly added agent. Exit
10: **else**
11: $add_agent = extra_nodes$ mod MSPA
12: **if** $extra_nodes \geq add_agent + add_agent \times MSPA$ **then**
13: **while** ($add_agent > 0$) **do**
14: add one agent to the agent having child agents < MAPA
15: add MSPA servers to this new added agent
16: $add_agent - -$
17: $extra_nodes = extra_nodes - add_agent + add_agent \times MSPA$
18: **else**
19: $add_agent - -$
20: go to line 13

like matrices are not included in the problem parameter descriptions (only their sizes are included). These large data items are included only in the final request for computation from client to server. Since we are primarily interested in modeling throughput for the agent hierarchy as an example of a hierarchical middleware system, we assume that large data items are already in-place on the server.

7.6 Experimental results

In this section we presents experiments designed to test the ability of deployment heuristic to correctly identify good real-world deployments that are appropriate to a given resource environment and workload. We presents the experimental design, the corresponding parametrization, experiments testing the accuracy of our deployment performance heuristic; the accuracy is key to provide confidence in the deployment algorithms. These algorithms are tested with experiments comparing the performance of the best deployment identified by our deployment algorithms against the performance of other intuitive deployments.

7.6.1 Experimental design

Software: DIET is used for all deployed agents and servers. The deployment arrangements are defined according to experimental goals and will be described in the following sections. Once a deployment has been chosen, GoDIET [88] is used to perform the actual software deployment.

Job types: Since our performance model and deployment approach focus on maximizing steady-state throughput, our experiments focus on testing the maximum sustained throughput provided by different deployments. As an initial study we consider DGEMM, a simple matrix multiplication provided as part of the Basic Linear Algebra Subprograms (BLAS) package [32].

Workload: Measuring the maximum throughput of a system is non-trivial: if too little load is introduced the maximum performance may not be achieved, if too much load is introduced the performance may suffer as well. Thus we use small quantities of steady-state load in the form of a client script that uses a continual loop to launch one request, wait for the response, and then sleep 0.05 seconds. With one client script there is at most a single request in the system. We then introduce load gradually by launching one client script every five seconds; four scripts are launched on each of 35 client machines. A full throughput test thus takes 700 seconds.

Resources: For these experiments we used a 55-node cluster at the École Normale Supérieure in Lyon, France. Each node provided dual AMD Opteron 246 processors @ 2 GHz, a cache size of 1024 KB, and 2 GB of memory. All nodes are connected by both a Gb ethernet and a 100 Mb/s ethernet; unless otherwise noted all communicates were sent over the Gigbit network. As measured with the Network Weather Service [85], available bandwidth on this network is 909.5 Mb/s and latency is 0.08 msec.

Model parametrization: To collect the values needed to calculate MSPA and MAPA, we measure the performance for a benchmark task (X_{ser}) using a small hierarchy of one agent and one server. We assume that the maximum throughput of an agent or a server can be calculated as the inverse of the time required by that element to treat a single request. Thus we measured the time required to treat a request at the server level and at the agent level. At the agent-level the processing time depends on the number of children attached to the agent so we collected benchmarks for a variety of sizes of star-based hierarchies. We then used a linear fit of the results to generate a simple model for agent-level throughput as a function of number of children. To measure data transfer within the hierarchy, we used `tcpdump` to monitor all data transferred between elements and `ethereal` to analyze the data. With these benchmarks we obtained predictions of all the variables that were not known such as $W_{X_{ser}}$, W_{in} and W_{out}. The benefit of this estimation approach is that measurements of the time required to treat a request are fairly easy to collect and each benchmark can be run in a few minutes with only a single client script. However, this approach assumes that there is no overhead to running requests in parallel and that all resources can be perfectly shared. Estimates generated in this fashion will therefore tend to overestimate the throughput of servers and agents.

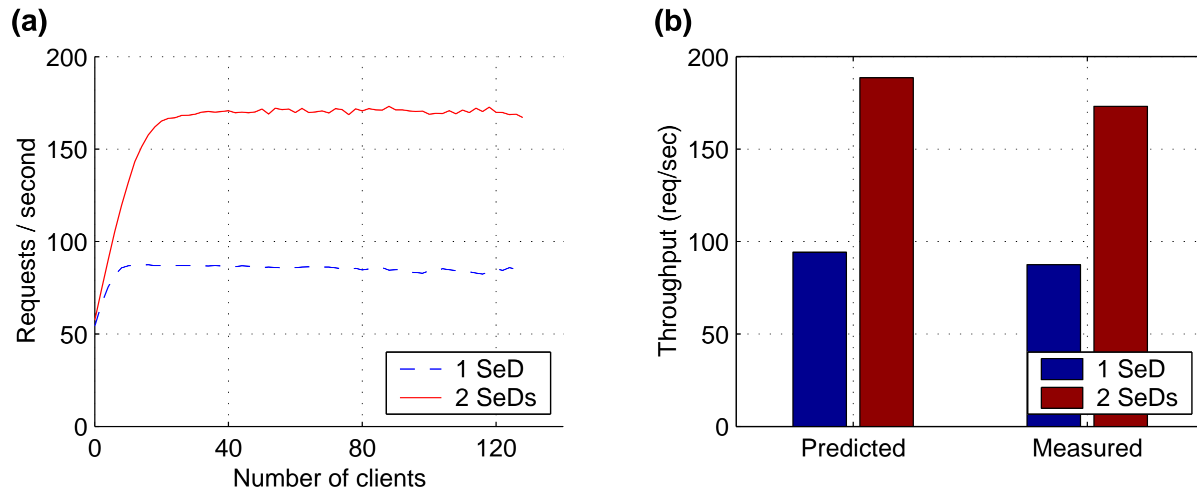


Figure 7.2: Star hierarchies with one or two servers for DGEMM 150x150 requests. (a) Real-world platform throughput for different load levels. (b) Comparison of predicted and actual maximum throughput.

7.6.2 Performance model validation

The usefulness of our deployment approach depends heavily on the ability to predict the maximum steady-state throughput of each element in a deployment. This section presents experiments designed to answer this question.

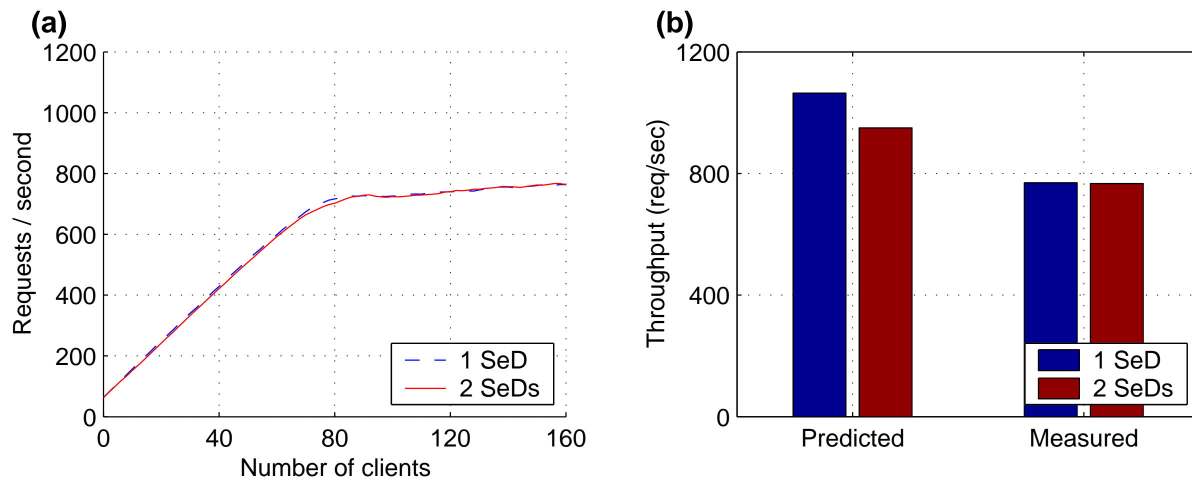


Figure 7.3: Star hierarchies with one or two servers for DGEMM 10x10 requests. (a) Throughput at different load levels. (b) Comparison of predicted and actual maximum throughput.

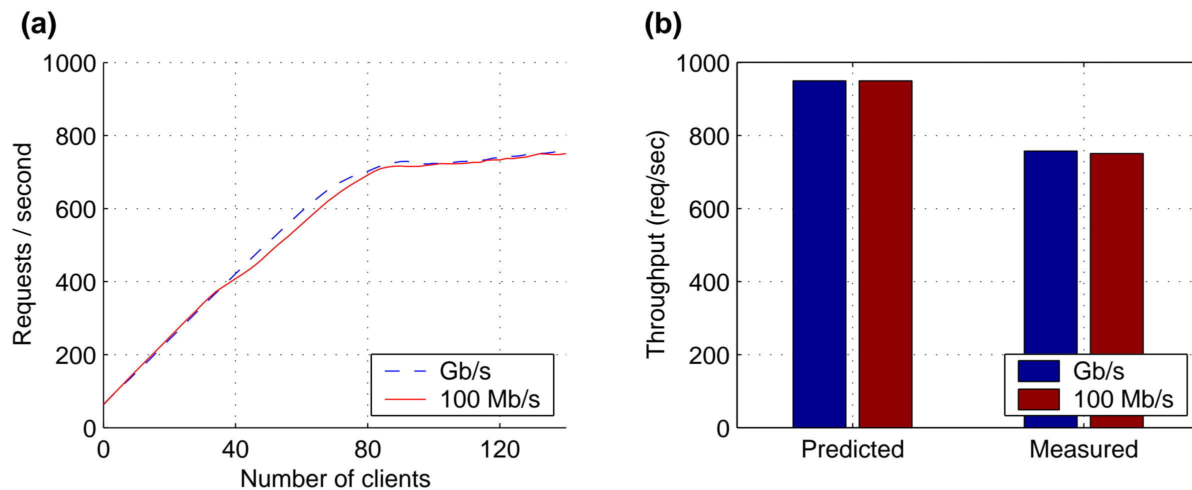


Figure 7.4: Star hierarchies with two servers using a Gb/s or a 100 Mb/s network. Workload was DGEMM 10x10. (a) Throughput at different load levels. (b) Comparison of predicted and actual maximum throughput.

The first test, shown in Figure 7.2, uses a workload of DGEMM150x150 to compare the performance of two hierarchies: an agent with one server versus an agent with two servers. For this scenario, the model predicts that both hierarchies are limited by server performance and therefore, performance will roughly double with the addition of the second server. These predictions are accurate: the model correctly predicts the absolute performance of these deployments and also predicts that the two-server deployment will be the better choice.

Figure 7.3 uses a workload of DGEMM10x10 to compare the performance of the same one and two server hierarchies. The model correctly predicts that both deployments are limited by agent performance and that the addition of the second server will in fact hurt performance. The error in the magnitude of the performance prediction is about 20-30%. We believe that the error in these predictions arises from the fact that some overhead is introduced by trying to run so many requests in parallel on the agent machine. Figure 7.4 compares performance using a Gb/s network versus a 100 Mb/s network; two-server hierarchies are used in both cases with a workload of DGEMM10x10. The model correctly predicts that the network is not the limiting factor on steady-state performance.

In summary, our deployment performance model is able to accurately predict server throughput and can predict the impact of adding servers to a server-limited or agent-limited deployment. The deployment algorithm accuracy could be obtained by adjusting our model parametrization to account for the results of this section. However, adding this adjustment phase to model parametrization greatly complicates the parametrization phase and we prefer to maintain an approach that could be applied rapidly to other software and resource environments.

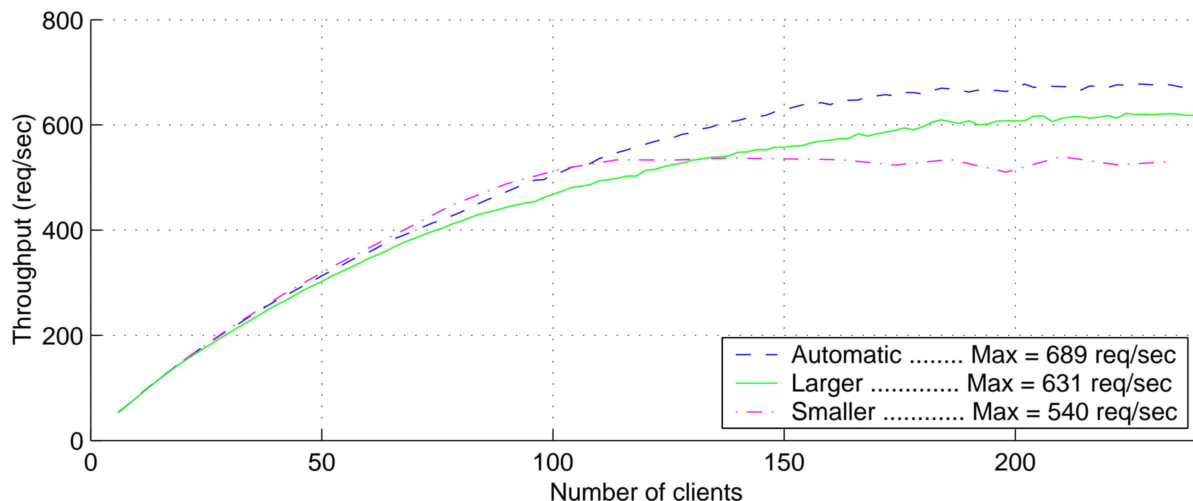


Figure 7.5: Comparison of automatically-generated hierarchy with hierarchies containing twice as many and half as many servers.

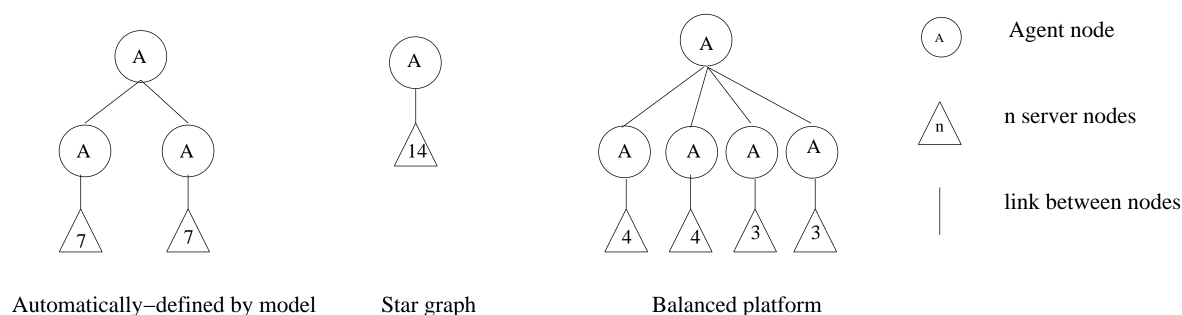


Figure 7.6: Types of platforms compared.

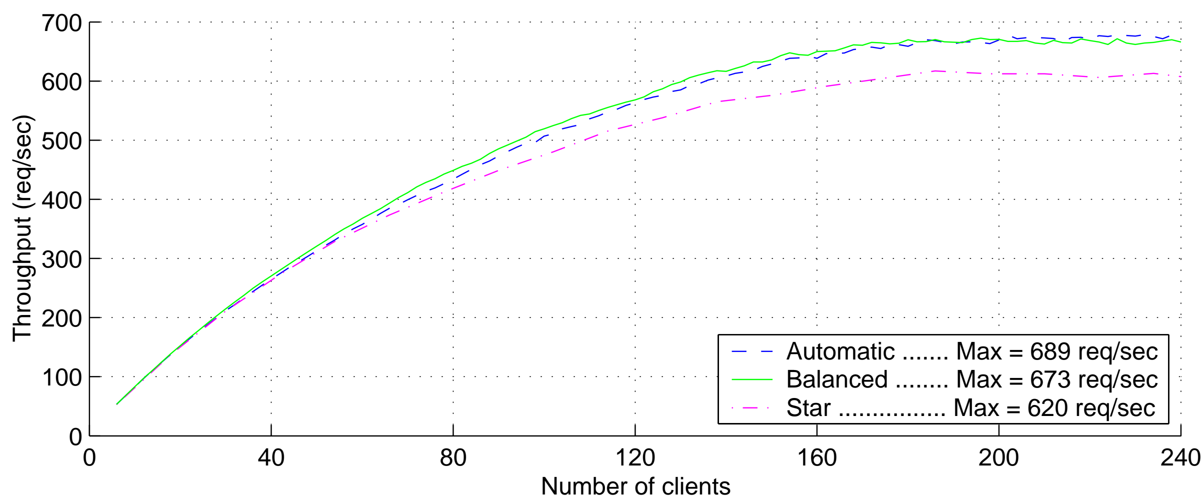


Figure 7.7: Comparison of automatically-generated hierarchy with intuitive alternative hierarchies.

7.6.3 Deployment selection validation

In this section we test the ability of the deployment algorithm to select an appropriate arrangement of agents and servers for deployment. We were not able to find alternative deployment algorithms that could be applied to our scenario as a comparison; in our experience, given the lack of automatic tools users typically define deployments by hand using intuition about the scenario. Thus we compare the model-defined deployment against several alternatives that, in our experience, would be reasonable intuitive choices with users.

We wish to find the deployment on our 50-machine cluster for a workload of DGEMM150x150. Our deployment algorithm predicts that the deployment uses a top-level agent and two middle-level agents where each middle-level agent can support 7 servers. This deployment thus contains 14 servers and 17 machines in total. To check whether the algorithm selected the correct number of servers, Figure 7.5 compares this `Automatic` deployment against deployments with the same two-level hierarchy

of agents but with different numbers of servers. The `Larger` deployment contains twice as many servers (14 on each middle-level agent so 28 servers in total) while the `Smaller` deployment contains roughly half as many servers (4 on each middle-level agent so 8 servers in total). The `Automatic` deployment provides a significantly higher maximum throughput than the others.

Although it is important that the deployment approach can select an appropriate number of resources, it is also important that it select an appropriate hierarchy. We therefore design two alternative deployments that we have found to be typical hierarchical styles for users. To remove the effect of the number of servers, we use exactly 14 servers, the number chosen by our approach, for these deployments. The `Star` deployment uses 14 servers attached directly to the top-level agent. The `Balanced` tries to balance work amongst the agents by using the same branching factor at all levels of the hierarchy; for 14 servers the most balanced hierarchy uses a top-level agent with 4 middle-level agents and 3 or 4 servers attached to each mid-level agent. Figure 7.6 shows these three platforms.

Figure 7.7 compares the performance achieved by these three deployments. The `Automatic` approach performs the best and provides a significant advantage over the `Star` topology. However, the `Balanced` approach performs almost as well as the `Automatic` approach. This result is not surprising: the two hierarchies are in fact fairly similar in structure.

7.7 Conclusion

In this chapter we have presented an heuristic to determine structure for hierarchical scheduler for a homogeneous resource platform. Heuristic determines how many nodes should be used and in what hierarchical organization with the goal of maximizing steady-state throughput.

The main focus of heuristic is to construct an hierarchy, so as to maximize the throughput of each node, where this throughput depend on the number of children a node is connected with, in the hierarchy. Number of children supported by a node depends on the type of children (server or agent). Thus exact calculation of MAPA and MSPA is required for correct structure of hierarchical schedulers. Models for calculation of MSPA and MAPA is given for DIET hierarchical scheduler system. Accordingly the exact number of nodes required to construct the hierarchy is calculated. Algorithms to use only the available number of nodes in hierarchy construction is also given.

This chapter provides the first step for automatic middleware deployment planning on homogeneous cluster. Next chapter provides an optimal algorithm for automatic middleware deployment on homogeneous cluster.

Chapter 8

Automatic Middleware Deployment Planning on Clusters

In previous Chapter 7 we have presented an heuristic for hierarchical middleware deployment for a homogeneous resource. We experimentally validated the heuristic deployments that are implemented under limited condition, for example, an agent can have either server or agents as children but not both. Deployment based on heuristic performed well as compared to other intuitive deployments. However, as the resources are homogeneous, an optimal deployment is possible. Considering the middleware deployment phase constraints' we found an algorithm for optimal middleware deployment on homogeneous resources.

This chapter explains in detail our optimal middleware deployment planning approach. In optimal middleware deployment planning approach we have shown that the optimal arrangement of schedulers is a complete spanning d-ary tree; this result agrees with existing results in load-balancing and routing from the scheduling and networking literature. Even we can automatically derive the optimal theoretical degree d for the tree.

Optimal algorithm consider the throughput of each execution phase of the middleware. Optimal algorithm is based on the throughput achieved by the scheduling phase and servicing phase of the middleware. To test our optimal algorithm, we use DIET middleware. First we develop detailed performance models for DIET system and validate these models in a real-world environment. We then present real-world experiments demonstrating that the deployments automatically derived by our approach are in practice nearly optimal and performs significantly better than other reasonable deployments.

8.1 Platform deployment

8.1.1 Platform architecture

This section defines our target platform architecture; Figure 8.1 provides a useful reference for these definitions.

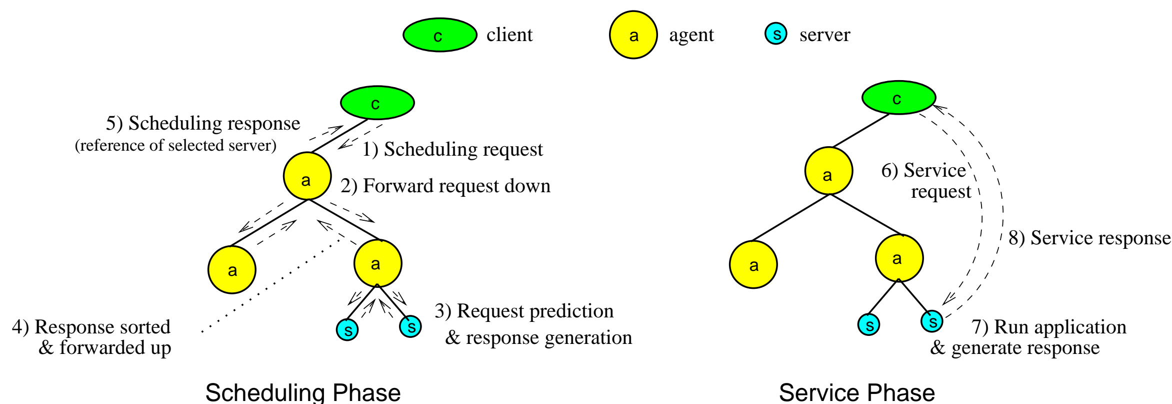


Figure 8.1: Platform deployment architecture and execution phases.

Software system architecture - We consider a service-provider software system composed of three types of elements: a set of client nodes $\mathbb{C}$ that require computations, a set of server nodes $\mathbb{S}$ that is provider of computations, and a set of agent nodes $\mathbb{A}$ that provides coordination of client requests with service offerings via service localization, scheduling, and persistent data management. The arrangement of these elements is shown in Figure 8.1. We consider only hierarchical arrangements of agents composed of a single top-level root agent and any number of agents arranged in a tree below the root agent. Server nodes are leaves of the tree, but may be attached to any agent in the hierarchy, even if that agent also has children that are agents.

Since the use of multiple agents is designed to distribute the cost of services such as scheduling, there is no performance advantage to having an agent with a single child. The only exception to this policy is for the root-level agent with a single server child; this "chain" can not be reduced.

We do not consider clients to be part of the hierarchy nor part of the deployment; this is because at the time of deployment we do not know where clients will be located. Thus a hierarchy can be described as follows. A server $s \in \mathbb{S}$ has exactly one parent that is always an agent $a \in \mathbb{A}$, a root agent $a \in \mathbb{A}$ has one or more child agents and/or servers and no parents. Non-root agents $a \in \mathbb{A}$ have exactly one parent and two or more child agents and/or servers.

Request definition - We consider a system that processes *requests* as follows. A client $c \in \mathbb{C}$ first generates a *scheduling request* that contains information about the service required by the client and meta-information about any input data sets, but does not include the actual input data. The scheduling request is submitted to the

root agent, which checks the scheduling request and forwards it on to its children. Other agents in the hierarchy perform the same operation until the scheduling request reaches the servers. We assume that the scheduling request is forwarded to all servers, though this is a worst case scenario as filtering may be done by the agents based on request type. Servers may or may not make predictions about performance for satisfying the request, depending on the exact system.

Servers that can perform the service then generate a *scheduling response*. The scheduling response is returned upward to the hierarchy and the agents sort and select amongst the various scheduling responses. It is assumed that the time required by an agent to select amongst scheduling responses increases with the number of children it has, but is independent of whether the children are servers or agents. Finally, the root agent forwards the chosen scheduling response (i.e., the selected server) to the client.

The client then generates a *service request* which is very similar to the scheduling request but includes the full input data set, if any is needed. The service request is submitted by the client to the chosen server. The server performs the requested service and generates a *service response*, which is then sent back to the client. A *completed request* is one that has completed both the scheduling and service request phases and for which a response has been returned to the client.

Resource architecture - The target resource architectural framework is represented by a weighted graph $G = (\mathbb{V}, \mathbb{E}, w, B)$. Each vertex v in the set of vertices $\mathbb{V}$ represents a computing resource with computing power w in MFlop/second. Each edge e in the set of edges $\mathbb{E}$ represents a resource link between two resources with edge cost B given by the bandwidth between the two nodes in Mb/second. We do not consider latency in data transfer costs because our model is based on steady-state scheduling techniques [16]. Usually, the latency is paid once for each of the communications that take place. In steady-state scheduling, however, as a flow of messages takes place between two nodes, the latency is paid only one time (when the flow is initially established) for the whole set of messages. Therefore latency will have an insignificant impact on our model and so we do not take it into account.

Deployment assumptions - We consider that at the time of deployment we do not know the client locations or the characteristics of the client resources. Thus clients are not considered in the deployment process and, in particular, we assume that the set of computational resources used by clients is disjoint from $\mathbb{V}$.

A valid deployment thus consists of a mapping of a hierarchical arrangement of agents and servers onto the set of resources $\mathbb{V}$. Any server or agent in a deployment must be connected to at least one other element; thus a deployment can have only connected nodes. A valid deployment will always include at least the root-level agent and one server. Each node $v \in \mathbb{V}$ can be assigned to either exactly one server s, exactly one agent a, or the node can be left idle. Thus if the total number of agents is $|\mathbb{A}|$, the total number of servers is $|\mathbb{S}|$, and the total number of resources is $|\mathbb{V}|$, then $|\mathbb{A}| + |\mathbb{S}| \leq |\mathbb{V}|$.

8.1.2 Optimal deployment

Our objective is to *find an optimal deployment of agents and servers for a set of resources* $\mathbb{V}$. We consider an *optimal deployment* to be a deployment that provides the maximum throughput ρ of completed requests per second. When the maximum throughput can be achieved by multiple distinct deployments, the preferred deployment is the one using the least resources.

As described in Section 8.1.1, we assume that at the time of deployment we do not know the locations of clients or the rate at which they will send requests. Thus it is impossible to generate an optimized, complete schedule. Instead, we seek a deployment that maximizes the *steady-state throughput*, i.e., the main goal is to characterize the *average* activities and capacities of each resource during each time unit.

We define the scheduling request throughput in requests per second, ρ_{sched}, as the rate at which requests are processed by the scheduling phase (see Section 8.1.1). Likewise, we define the service throughput in requests per second, $\rho_{service}$, as the rate at which the servers produce the services required by the clients. The following lemmas lead to a proof of an optimal deployment shape of the platform.

Lemma 1. *The completed request throughput ρ of a deployment is given by the minimum of the scheduling request throughput ρ_{sched} and the service request throughput $\rho_{service}$.*

$$\rho = min(\rho_{sched}, \rho_{service})$$

Proof. A completed request has, by definition, completed both the scheduling request and the service request phases.

Case 1: $\rho_{sched} \geq \rho_{service}$. In this case requests are sent to the servers at least as fast as they can be serviced by the servers, so the overall rate is limited by $\rho_{service}$.

Case 2: $\rho_{sched} < \rho_{service}$. In this case the servers are left idle waiting for requests and new requests are processed by the servers faster than they arrive. The overall throughput is thus limited by ρ_{sched}. ■

The *degree* of an agent is the number of children directly attached to it, regardless of whether the children are servers or agents.

Lemma 2. *The scheduling throughput ρ_{sched} is limited by the throughput of the agent with the highest degree.*

Proof. As described in Section 8.1.1, we assume that the time required by an agent to manage a request increases with the number of children it has. Thus, agent throughput decreases with increasing agent degree and the agent with the highest degree will provide the lowest throughput. Since we assume that scheduling requests are forwarded to all agents and servers, a scheduling request is not finished until all agents have responded. Thus ρ_{sched} is limited by the agent providing the lowest throughput which is the agent with the highest degree. ■

Lemma 3. *The service request throughput $\rho_{service}$ increases as the number of servers included in a deployment increases.*

Proof. The service request throughput is a measure of the rate at which servers in a deployment can generate responses to client service requests. Since agents do not participate in this process, $\rho_{service}$ is independent of the agent hierarchy. The computational power of the servers is used for both (1) generating responses to scheduling queries from the agents and (2) providing computational services for clients. For a given value of ρ_{sched} the work performed by a server for activity (1) is independent of the number of servers. The work performed by each server for activity (2) is thus also independent of the number of servers. Thus the work performed by the servers as a group for activity (2) increases as the number of servers in the deployment increases. ∎

For the rest of this section, we need some precise definitions. A *complete d-ary tree* is a tree for which all internal nodes, except perhaps one, have exactly d children. If n is the depth of the tree, the potential internal node with strictly less than d children is at depth $n-1$ and may have any degree from 1 to $d-1$. A *spanning tree* is a connected, acyclic subgraph containing all the vertices of a graph. We introduce the following definition to aid later discussions.

Definition 1. *A **C**omplete **S**panning **d**-ary tree (CSD tree) is a tree that is both a complete d-ary tree and a spanning tree.*

For deployment, leaves are servers and all other nodes are agents. A degree d of one is useful only for a deployment of a single root agent and a single server. Note that for a set of resources $\mathbb{V}$ and degree d, a large number of CSD trees can be constructed. However, since we consider only homogeneous resources, all such CSD trees are equivalent in that they provide exactly the same performance.

Definition 2. *A $dMax$ set is the set of all trees that can be built using $|\mathbb{V}|$ resources and for which the maximum degree is equal to $dMax$.*

Figure 8.2 shows some examples of trees from the $dMax$ 4 and $dMax$ 6 sets.

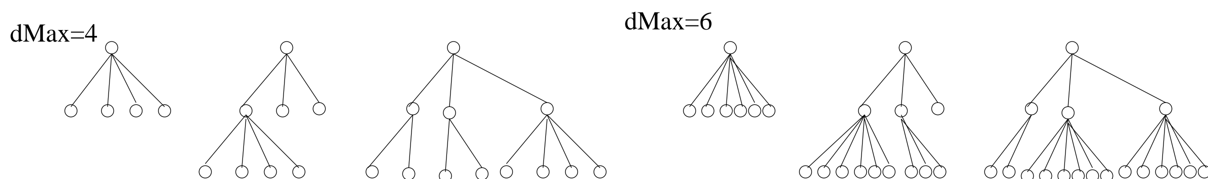


Figure 8.2: Deployment trees of $dMax$ sets 4 and 6.

Theorem 1. *In a $dMax$ set, all $dMax$ CSD trees have optimal throughput.*

Proof. We know by Lemma 1 that the throughput ρ of any tree is limited by either its schedule request throughput ρ_{sched} or its service request throughput $\rho_{service}$. As Lemma 2 states that the scheduling request throughput ρ_{sched} is only limited by the throughput of the agent with the highest degree, all trees in a $dMax$ set have the same

Algorithme 8.1 Algorithm to construct an optimal tree.

```
1:  calculate best_d using Theorem 2
2:  Add the root node
3:  if best_d == 1 then
4:      add one server to root node
5:      Exit
6:  availNodes = |V| − 1
7:  level = 0
8:  while availNodes > 0 do
9:      if ∃ agent at depth level with degree 0 then
10:         toAdd = min(best_d, availNodes)
11:         add toAdd children to the agent
12:         availNodes = availNodes − toAdd
13:     else
14:         level = level + 1
15: if ∃ agent with degree 1 then
16:     remove the child
17: convert all leaf nodes to servers
```

scheduling request throughput ρ_{sched}. Thus, to show that any $dMax$ CSD tree is an optimal solution among the trees in the $dMax$ set, we must prove that the service request throughput $\rho_{service}$ of any CSD tree is at least as large as the $\rho_{service}$ of any other tree in the $dMax$ set.

By Lemma 3, we know that the service request throughput $\rho_{service}$ increases with the number of servers included in a deployment. Given the limited resource set size, $|\mathbb{V}|$, the number of servers (leaf nodes) is largest for deployments with the smallest number of agents (internal nodes). Therefore, to prove the desired result we just have to show that $dMax$ CSD trees have the minimal number of internal nodes among the trees in a $dMax$ set.

Let us consider any optimal tree $\mathcal{T}$ in a $dMax$ set. We have three cases to consider.

1. $\mathcal{T}$ is a CSD tree. As all CSD trees (in the $dMax$ set) have the same number of internal nodes, all CSD trees are optimal. QED.

2. $\mathcal{T}$ is not a CSD tree but all its internal nodes have a degree equal to $dMax$, except possibly one. Then we build a CSD tree having at most as many internal nodes as $\mathcal{T}$. Using case 1, this will prove the result.

Let h be the height of $\mathcal{T}$. As $\mathcal{T}$ is not a CSD tree, seeing our hypothesis on the degree of its internal nodes, $\mathcal{T}$ must have a node at a height $h' < h-1$ which has strictly less than $dMax$ children. Let h' be the smallest height at which there is such a node, let $\mathcal{N}$ be such a node, and let d be the degree of $\mathcal{N}$. Then remove $dMax - d$ children from any internal node $\mathcal{N}'$ at height $h-1$ and add them as children of $\mathcal{N}$. Note that, whatever the value of d, among $\mathcal{N}$ and $\mathcal{N}'$ there is the same number of internal nodes whose degree is strictly less than $dMax$ *before* and *after* this transformation. Then, through

this transformation, we obtained a new tree $\mathcal{T}'$ whose internal nodes all have a degree equal to $dMax$, except possibly one. Also, there is the same number of leaf nodes in $\mathcal{T}$ and $\mathcal{T}'$. Therefore, $\mathcal{T}'$ is also optimal. Furthermore, if $\mathcal{T}'$ is not a CSD tree, it has one less node at level h' whose degree is strictly less than $dMax$; if there are zero nodes at level h' whose degree is strictly less than $dMax$ after the transformation then the value to "h'" is increased. Repeating this transformation recursively we will eventually end up with a CSD tree.

3. $\mathcal{T}$ has at least two internal nodes whose degree is strictly less than $dMax$. Then, let us take two such nodes $\mathcal{N}$ and $\mathcal{N}'$ of degrees d and d'. If $d + d' \leq dMax$, then we remove all the children of $\mathcal{N}'$ and make them children of $\mathcal{N}$. Otherwise, we remove $dMax - d$ children of $\mathcal{N}'$ and make them children of $\mathcal{N}$. In either case, the number of leaf nodes in the tree before and after the transformation is non-decreasing, and the number of internal nodes whose degree is strictly less than $dMax$ is (strictly) decreasing. So, as for the original tree, the new tree is optimal. Repeating this transformation recursively, we will eventually end up with a tree dealt with by case 2. Hence, we can conclude. ■

We select CSD tree because only the $dMax$ CSD tree has minimum height among all optimal trees in the $dMax$ set.

Theorem 2. *A complete spanning d-ary tree with degree $d \in [1, |\mathbb{V}| - 1]$ that maximizes the minimum of the scheduling request and service request throughputs is an optimal deployment.*

Proof. This theorem is fundamentally a corollary of Theorem 1. The optimal degree is not known a priori; it suffices to test all possible degrees $d \in [1, |\mathbb{V}| - 1]$ and to select the degree that provides the maximum completed request throughput. ■

8.1.3 Deployment construction

Once an optimal degree $best_d$ has been calculated using Theorem 2, we can use the algorithm shown in Figure 8.1 to construct the optimal CSD tree.

A few examples will help clarify the results of our deployment planning approach. Let us consider that we have 10 available nodes ($|\mathbb{V}| = 10$). Suppose $best_d = 1$. Algorithm 8.1 will construct the corresponding best platform - a root agent with a single server attached. Now suppose $best_d = 4$. Then Algorithm 8.1 will construct the corresponding best deployment - the root agent with four children, one of which also has four children; the deployment has two agents, seven servers and one unused node because it can only be attached as a chain.

Figure 8.3 shows some possible DIET deployments.

8.1.4 Request performance modeling

In order to apply the model defined in Section 8.1 to DIET, we must have models for the scheduling throughput and the service throughput in DIET. In this section we define performance models to estimate the time required for various phases of request

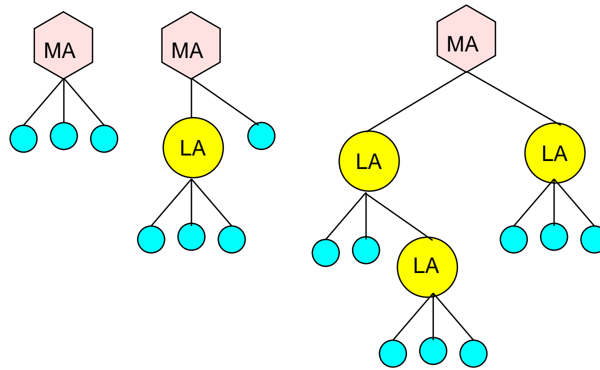


Figure 8.3: Example DIET deployments consisting of MAs, LAs, and SeDs (unlabelled circles).

treatment in DIET. These models will be used in the following section to create the needed throughput models.

We make the following assumptions about DIET for performance modeling. The MA and LA are considered as having the same performance because their activities are almost identical and in practice we observe only negligible differences in their performance.

We assume that the work required for an agent to treat responses from SeD-type children and from agent-type children is the same. DIET allows configuration of the number of responses forwarded by agents; here we assume that only the best server is forwarded to the parent.

When client requests are sent to the agent hierarchy, DIET is optimized such that large data items like matrices are not included in the problem parameter descriptions (only their sizes are included). These large data items are included only in the final request for computation from client to server. As stated earlier, we assume that we do not have a priori knowledge of client locations and request submission patterns. Thus, we assume that needed data is already in place on the servers and we do not consider data transfer times.

The following variables will be of use in our model definitions. S_{req} is the size in Mb of the message forwarded down the agent hierarchy for a scheduling request. This message includes only parameters and not large input data sets. S_{rep} is the size in Mb of the reply to a scheduling request forwarded back up the agent hierarchy. Since we assume that only the best server response is forwarded by agents, the size of the reply does not increase as the response moves up the tree.

W_{req} is the amount of computation in MFlop needed by an agent to process one incoming request. $W_{rep}(d)$ is the amount of computation in MFlop needed by an agent to merge the replies from its d children. W_{pre} is the amount of computation in MFlop needed for a server to predict its own performance for a request. W_{app} is the amount of computation in MFlop needed by a server to complete a service request for *app* service. The provision of this computation is the main goal of the DIET system.

Agent communication model: To treat a request, an agent *receives* the request from its parent, *sends* the request to each of its children, *receives* a reply from each of its children, and *sends* one reply to its parent. By Lemma 2, we are concerned only with the performance of the agent with the highest degree, d. The time in seconds required by an agent for receiving all messages associated with a request from its parent and children is as follows.

$$agent_receive_time = \frac{S_{req} + d \cdot S_{rep}}{B} \tag{8.1}$$

Similarly, the time in seconds required by an agent for sending all messages associated with a request to its children and parent is as follows.

$$agent_send_time = \frac{d \cdot S_{req} + S_{rep}}{B} \tag{8.2}$$

Server communication model: Servers have only one parent and no children, so the time in seconds required by a server for receiving messages associated with a scheduling request is as follows.

$$server_receive_time = \frac{S_{req}}{B} \tag{8.3}$$

The time in seconds required by a server for sending messages associated with a request to its parent is as follows.

$$server_send_time = \frac{S_{rep}}{B} \tag{8.4}$$

Agent computation model: Agents perform two activities involving computation: the processing of incoming requests and the selection of the best server amongst the replies returned from the agent's children.

There are two activities in the treatment of replies: a fixed cost W_{fix} in MFlops and a cost W_{sel} that is the amount of computation in MFlops needed to process the server replies, sort them, and select the best server. Thus the computation associated with the treatment of replies is given

$$W_{rep}(d) = W_{fix} + W_{sel} \cdot d$$

The time in seconds required by the agent for the two activities is given by the following equation.

$$agent_comp_time = \frac{W_{req} + W_{rep}(d)}{w} \tag{8.5}$$

Server computation model: Servers also perform two activities involving computation: performance prediction as part of the scheduling phase and provision of application services as part of the service phase.

Let us consider a deployment with a set of servers $\mathbb{S}$ and the activities involved in completing $|\mathbb{S}|$ requests at the server level. All servers complete $|\mathbb{S}|$ prediction requests

and each server will complete one service request phase, on average. As a whole, the servers as a group require the $(W_{pre} \cdot |\mathbb{S}| + W_{app})/w$ time in seconds to complete the $\mathbb{S}$ requests. We divide by the number of requests $|\mathbb{S}|$ to obtain the average time required per request by the servers as a group.

$$server_comp_time = \frac{W_{pre} + \frac{W_{app}}{|\mathbb{S}|}}{w} \tag{8.6}$$

8.2 Steady-state throughput modeling

In this section we present models for scheduling and service throughput in DIET. We consider two different theoretical models for the capability of a computing resource to do computation and communication in parallel.

Send or receive or compute, single port: In this model, a computing resource has no capability for parallelism: it can either send a message, receive a message, or compute. Only a single port is assumed: messages must be sent serially and received serially. This model may be reasonable for systems with small messages as these messages are often quite CPU intensive. As shown in the following equation, the scheduling throughput in requests per second is then given by the minimum of the throughput provided by the servers for prediction and by the agents for scheduling.

$$\rho = min \left(\frac{1}{\frac{W_{pre}}{w} + \frac{S_{req}}{B} + \frac{S_{rep}}{B}}, \frac{1}{\frac{S_{req}+d \cdot S_{rep}}{B} + \frac{d \cdot S_{req}+S_{rep}}{B} + \frac{W_{req}+W_{rep}(d)}{w}}, \frac{1}{\frac{S_{req}}{B} + \frac{S_{rep}}{B} + \frac{W_{pre}+\frac{W_{app}}{|\mathbb{S}|}}{w}} \right)$$

Send || receive || compute, single port: In this model, it is assumed that a computing resource can send messages, receive messages, and do computation in parallel. We still only assume a single port-level: messages must be sent serially and they must be received serially. Thus, for this model throughput can be calculated as follows.

$$\rho = min \left(\frac{1}{max\left(\frac{W_{pre}}{w}, \frac{S_{req}}{B}, \frac{S_{rep}}{B}\right)}, \frac{1}{max\left(\frac{S_{req}+d \cdot S_{rep}}{B}, \frac{d \cdot S_{req}+S_{rep}}{B}, \frac{W_{req}+W_{rep}(d)}{w}\right)}, \frac{1}{max\left(\frac{S_{req}}{B}, \frac{S_{rep}}{B}, \frac{W_{pre}+\frac{W_{app}}{|\mathbb{S}|}}{w}\right)} \right)$$

8.3 Experimental results

In this section we present experiments designed to test the ability of our deployment model to correctly identify good real-world deployments. Since our performance model and deployment approach focus on maximizing steady-state throughput, our experiments focus on testing the maximum sustained throughput provided by different deployments. The following

section describes the experimental design, Section 8.3.2 describes how we obtained the parameters needed for the model, Section 8.3.3 presents experiments testing the accuracy of our throughput performance models, and Section 8.3.4 presents experiments testing whether the deployment selected by our approach provides good throughput as compared to other reasonable deployments. Finally, Section 8.4 provides some forecasts of good deployments for a range of problem sizes and resource sets.

8.3.1 Experimental design

Software: DIET 2.0 is used for all deployed agents and servers; GoDIET [88] version 2.0.0 is used to perform the actual software deployment.

Job types: In general, at the time of deployment, one can know neither the exact job mix nor the order in which jobs will arrive. Instead, one has to assume a particular job mix, define a deployment, and eventually correct the deployment after launch if it wasn't well-chosen. For these tests, we consider the DGEMM application, a simple matrix multiplication provided as part of the Basic Linear Algebra Subprograms (BLAS) package [32]. For example, when we state that we use DGEMM 100, it signifies that we use matrix multiplication with square matrices of dimensions 100x100. For each specific throughput test we use a single problem size; since we are testing steady-state conditions, the performance obtained should be equivalent to that one would attain for a mix of jobs with the same average execution time.

Workload: Measuring the maximum throughput of a system is non-trivial: if too little load is introduced the maximum performance may not be achieved, if too much load is introduced the performance may suffer as well. A unit of load is introduced via a script that runs a single request at a time in a continual loop. We then introduce load gradually by launching one client script every second. We introduce new clients until the throughput of the platform stops improving; we then let the platform run with no addition of clients for 10 minutes. Results presented are the average throughput during this 10 minute period. This test is hereafter called a *throughput test*.

Resources: The experiments were performed on two similar clusters. The first is a 55-node cluster at the École Normale Supérieure in Lyon, France. Each node includes dual AMD Opteron 246 processors at 2 GHz, a cache size of 1024 KB, and 2 GB of memory. We used GCC 3.3.5 for all compilations and the Linux kernel version was 2.6.8. All nodes are connected by both a Gigabit Ethernet. We measured network bandwidth using the Network Weather Service [85]. Using the default NWS message size of 256 kB we obtain a bandwidth of 909.5 Mb/s; using the message size sent in DIET of 850 bytes we obtain a bandwidth of 20.0 Mb/s.

The second cluster is a 140-node cluster at Sophia in France. The nodes are physically identical to the ones at Lyon but are running the Linux kernel version 2.4.21 and all compilations were done with GCC 3.2.3. The machines at Sophia are linked by 6 different Cisco Gigabit Ethernet switches connected with a 32 Gbps bus.

8.3.2 Model parametrization

Table 8.1 presents the parameter values we use for DIET in the models for ρ_{sched} and $\rho_{service}$. Our goal is to parametrize the model using only easy-to-collect micro-benchmarks. In particular, we seek to use only values that can be measured using a few clients executions. The alternative is to base the model on actual measurements of the maximum throughput of various

system elements; while we have these measurements for DIET, we feel that the experiments required to obtain such measurements are difficult to design and run and their use would prove an obstruction to the application of our model for other systems.

To measure message sizes S_{req} and S_{rep} we deployed a Master Agent (MA) and a single DGEMM server (SeD) on the Lyon cluster and then launched 100 clients serially. We collected all network traffic between the MA and the SeD machines using `tcpdump` and analyzed the traffic to measure message sizes using the Ethereal Network Protocol analyzer[1]. This approach provides a measurement of the entire message size including headers. Using the same MA-SeD deployment, 100 client repetitions, and the statistics collection functionality in DIET [24], we then collected detailed measurements of the time required to process each message at the MA and SeD level. The parameter W_{rep} depends on the number of children attached to an agent. We measured the time required to process responses for a variety of star deployments including an MA and different numbers of SeDs. A linear data fit provided a very accurate model for the time required to process responses versus the degree of the agent with a correlation coefficient of 0.997. We thus use this linear model for the parameter W_{rep}. Finally, we measured the capacity of our test machines in MFlops using a mini-benchmark extracted from Linpack and used this value to convert all measured times to estimates of the MFlops required.

Components	W_{req} (MFlop)	W_{rep} (MFlop)	W_{pre} (MFlop)	S_{rep} (Mb)	S_{req} (Mb)
Agent	3.2×10^{-2}	$1.3\times 10^{-3} + 9.6\times 10^{-4} \cdot d$	-	6.4×10^{-3}	5.3×10^{-3}
SeD	-	-	6.4×10^{-3}	6.4×10^{-5}	5.3×10^{-5}

Table 8.1: Parameter values for middleware deployment on cluster

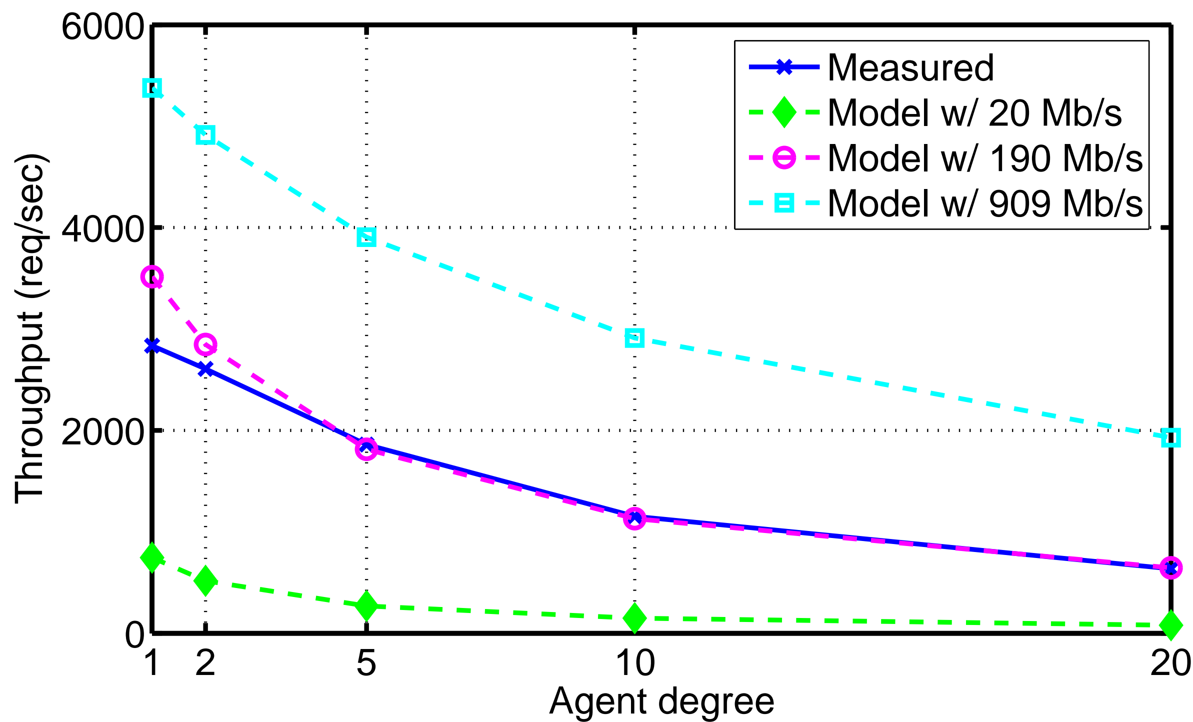
8.3.3 Throughput model validation

This section presents experiments testing the accuracy of the DIET agent and server throughput models presented in Section 8.2.

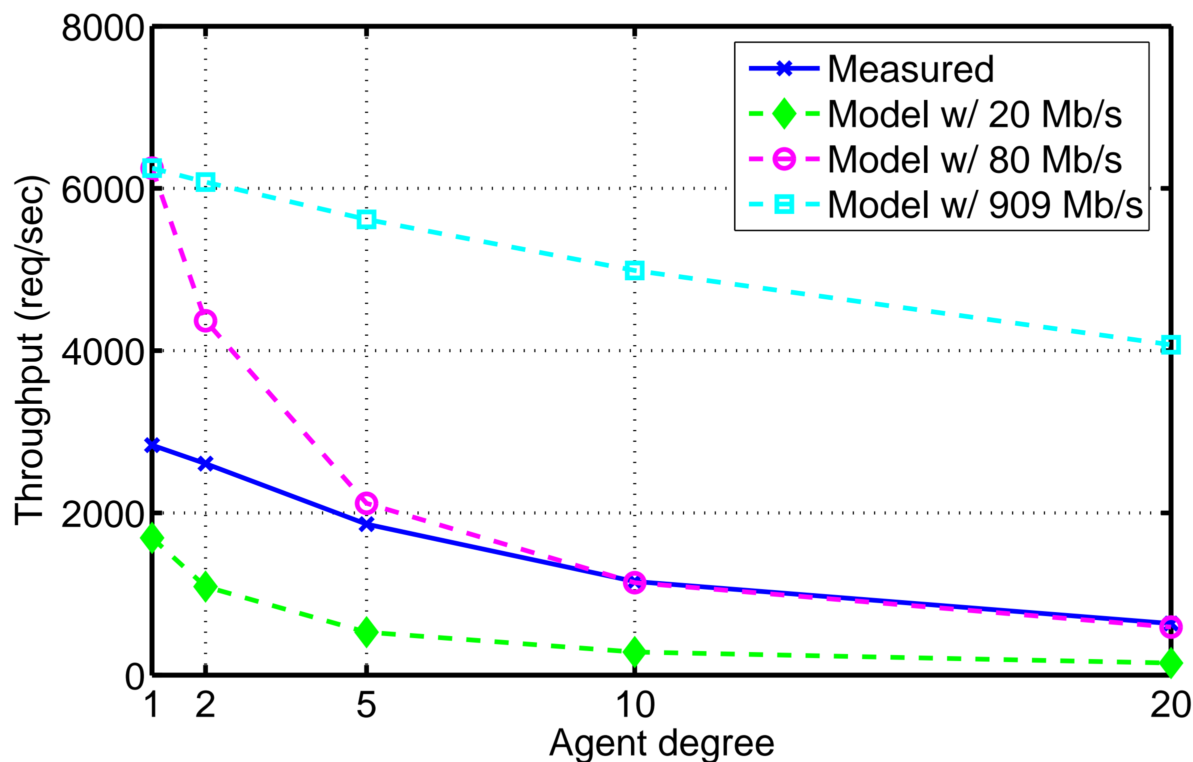
First, we examine the ability of the models to predict *agent throughput* and, in particular, to predict the effect of an agent's degree on its performance. To test agent performance, the test scenario must be clearly agent-limited. Thus we selected a very small problem size of DGEMM 10. To test a given agent degree d, we deployed an MA and attached d SeDs to that MA; we then ran a throughput test as described in Section 8.3.1. The results are presented in Figure 8.4. We verify that these deployments are all agent-limited by noting that the throughput is lower for a degree of two than for a degree of 1 despite the fact that the degree two deployment has twice as many SeDs.

Figures 8.4 (a) and (b) present model predictions for the serial and parallel models, respectively. In each case three predictions are shown using different values for the network bandwidth. The values of 20 Mb/s and 909.5 Mb/s are the values obtained with NWS. Comparison of predicted and measured leads us to believe that these measurements of the network bandwidth are not representative of what DIET actually obtains. This is not surprising given that DIET uses very small messages and network performance for this message size is highly sensitive to the communication layers used. The third bandwidth in each graph is chosen to

[1] http://www.ethereal.com



(a) Serial model



(b) Parallel model

Figure 8.4: Measured and predicted platform throughput for DGEMM size 10; predictions are shown for several bandwidths.

provide a good fit of the measured and predicted values. For the purposes of the rest of this paper we will use the serial model with a bandwidth of 190 Mb/s because it provides a better fit than the parallel model. In the future we plan to investigate other measurement techniques for bandwidth that may better represent the bandwidth achieved when sending many very small messages as is done by DIET.

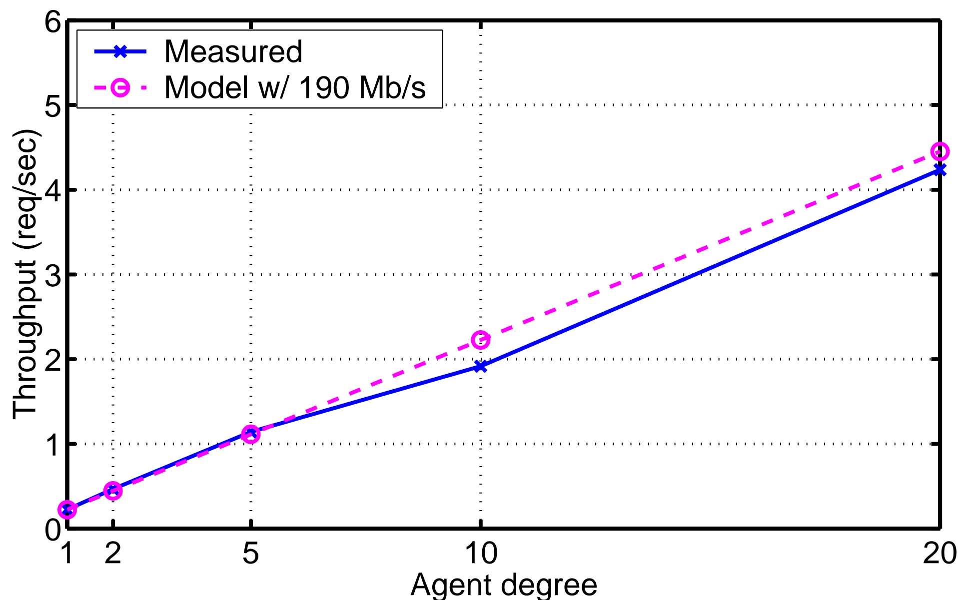


Figure 8.5: Measured and predicted platform throughput for DGEMM size 1000; predictions are shown for the serial model with bandwidth 190 Mb/s.

Next, we test the accuracy of throughput prediction for the *servers*. To test server performance, the test scenario must be clearly SeD-limited. Thus we selected a relatively large problem size of DGEMM 1000. To test whether performance scales as the number of servers increases, we deployed an MA and attached different numbers of SeDs to the MA. The results are presented in Figure 8.5. Only the serial model with a bandwidth of 190 Mb/s is shown; in fact, the results with the parallel model and with different bandwidths are all within 1% of this model since the communication is overwhelmed by the solve phase itself.

8.3.4 Deployment selection validation

In this section we present experiments that test the effectiveness of our deployment approach in selecting a good deployment. For each experiment, we select a cluster, define the total number of resources available, and define a DGEMM problem size. We then apply our deployment algorithms to predict which CSD tree will provide the best throughput and we measure the throughput of this CSD tree in a real-world deployment. We then identify and test a suitable range of other CSD trees including the star, the most popular middleware deployment arrangement.

Figure 8.6 shows the predicted and actual throughput for a DGEMM size of 200 where 25 nodes in the Lyon cluster are available for the deployment. Our model predicts that the best

throughput is provided by CSD trees with degrees of 12, 13 and 14. These trees have the same predicted throughput because they have the same number of SeDs and the throughput is limited by the SeDs. Experiments show that the CSD tree with degree 12 does indeed provide the best throughput. The model prediction overestimates the throughput; we believe that there is some cost associated with having multiple levels in a hierarchy that is not accounted for in our model. However, it is more important that the model correctly predicts the shape of the graph and identifies the best degree than that it correctly predicts absolute throughput.

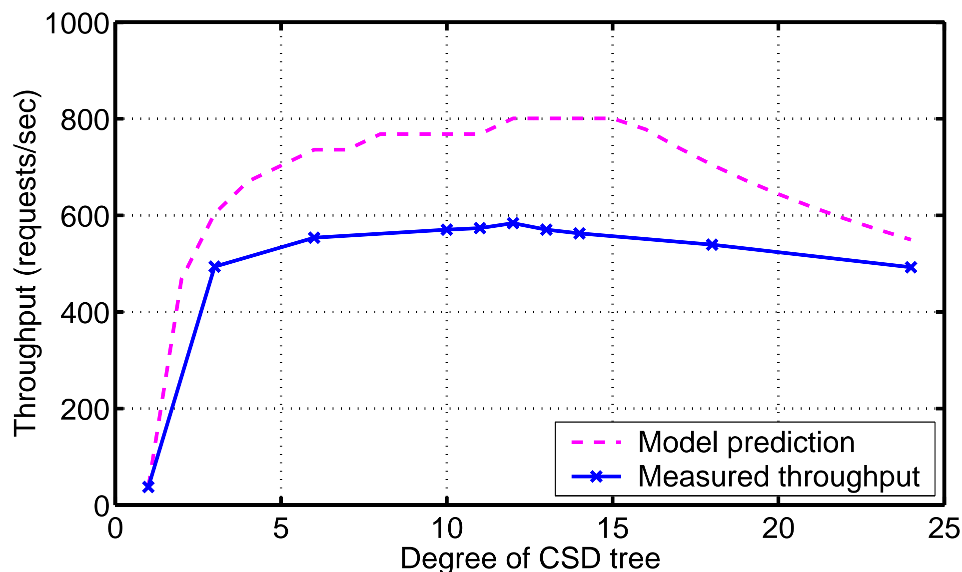


Figure 8.6: Predicted and measured throughput for different CSD trees for DGEMM 200 with 25 available nodes in the Lyon cluster.

For the next experiment, we use the same problem size of 200 but change the number of available nodes to 45 and the cluster to Sophia. We use the same problem size to demonstrate that the best deployment is dependent on the number of resources available, rather than just the type of problem. The results are shown in Figure 8.7. The model predicts that the best deployment will be a degree eight CSD tree while experiments reveal that the best degree is three. The model does however correctly predict the shape of the curve and selects a deployment that achieves a throughput that is 87.1% of the optimal. By comparison, the popular star deployment (degree 44) obtains only 40.0% of the optimal performance.

For the last experiment, we again use a total of 45 nodes from the Sophia cluster but we increase the problem size to 310; we use the same resource set size to show that the best deployment is also dependent on the type of workload expected. The results are shown in Figure 8.8. In this test case, the model predictions are generally much more accurate than in the previous two cases; this is because $\rho_{service}$ is the limiting factor over a greater range of degrees due to the larger problem size used here. Our model predicts that the best deployment is a 22 degree CSD tree while in experimentation the best degree is 15. However, the deployment chosen by our model achieves a throughput that is 98.5% of that achieved by the optimal 15 degree tree. By comparison, the star and tri-ary tree deployments achieve only 73.8% and 74.0% of the optimal throughput.

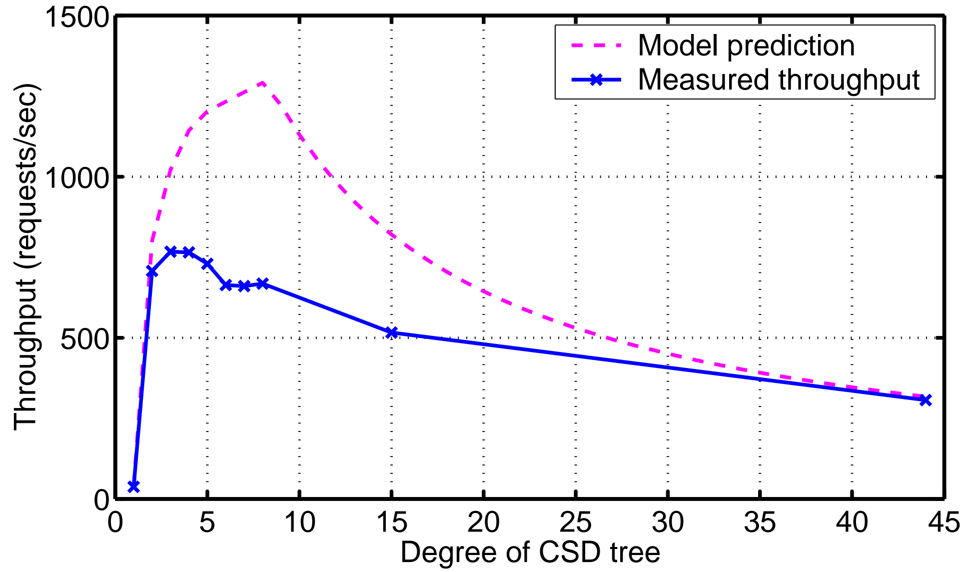


Figure 8.7: Predicted and measured throughput for different CSD trees for DGEMM 200 with 45 available nodes in the Sophia cluster.

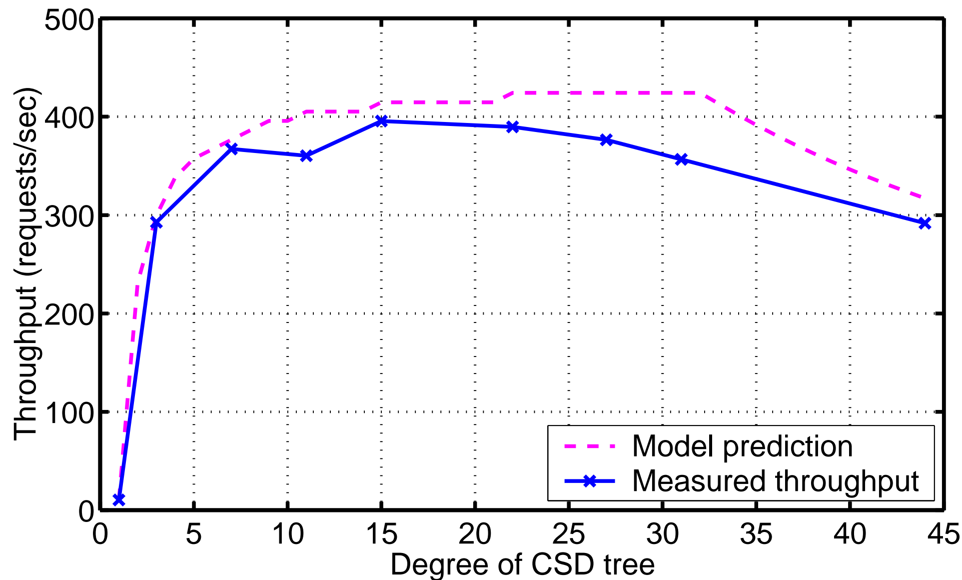


Figure 8.8: Predicted and measured throughput for different CSD trees for DGEMM 310 with 45 available nodes in the Sophia cluster.

Table 8.2 summarizes the results of these three experiments by reporting the percentage of optimal achieved for the tree selected by our model, the star, and the tri-ary tree. The table also includes data for problem size 10, for which an MA with one SeD is correctly predicted to be optimal, and problem size 1000, for which a star deployment is correctly predicted to be optimal. These last two cases represent the usage of the model in clearly SeD-limited or clearly agent-limited conditions.

DGEMM Size	Nodes $\lvert\mathbb{V}\rvert$	Optimal Degree	Selected Degree	Model Performance	Star	Tri-ary
10	21	1	1	100.0%	22.4%	50.5%
100	25	2	2	100.0%	84.4%	84.6%
200	45	3	8	87.1%	40.0%	100.0%
310	45	15	22	98.5%	73.8%	74.0%
1000	21	20	20	100.0%	100.0%	65.3%

Table 8.2: A summary of the percentage of optimal achieved by the deployment selected by our model, a star deployment, and a tri-ary tree deployment.

8.3.5 Validation of model for mix workload

In this section we test whether it is better to deploy one DIET hierarchy on all available nodes and submit all (heterogeneous) requests to the hierarchy, or whether it is better to divide the available nodes in different partitions with one partition per problem size. For example, if we have 100 nodes and four different problems, should we deploy one CSD tree using all 100 nodes to solve the four types of problems, or should we deploy four CSD trees of 25 nodes each and submit only uniform problems to each CSD tree.

For this experiment we use a total of 75 nodes at the Orsay site and DGEMM problem sizes 10, 100, and 1000. First, we divided the available nodes in three sets of 25 nodes each. Then we used our optimal deployment planning algorithm to predict the best value of d to construct a CSD tree with 25 nodes for each problem size. Next, we used the planning algorithm to predict the best value of d for a CSD tree using 75 nodes and each problem size.

For a deployment of size 25 nodes, for sizes 10 and 100 our algorithm predicts a best degree of 2, while for size 1000 the best degree is predicted to be 24. For a deployment of size 75 nodes, degree 2 is again predicted to be optimal for sizes 10 and 100 while degree 74 is predicted to be optimal for size 1000. Next, we tested the makespan of each set of tasks on the three separate deployments, sending only the appropriate problem size to each deployment. Then we tested the makespan of the combined set of tasks including all three problem sizes on the 75 node deployments. The results are shown in Fig. 8.9.

Fig. 8.9 shows that if we deploy three small CSD tree in parallel it take 560.36 seconds to execute 3000 requests but if we deploy one big CSD tree but with degree 74, we can save time, as it took only 466 seconds to execute 3000 requests.

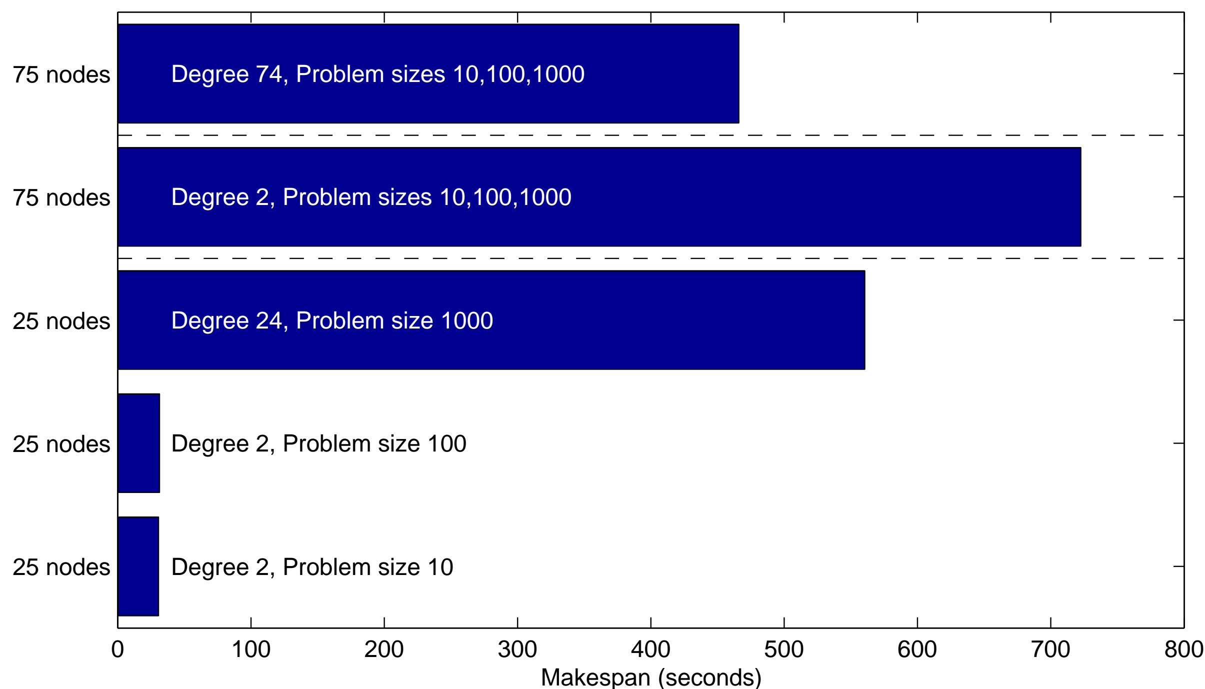


Figure 8.9: Makespan for a group of tasks partitioned to three deployments or sent to a single joint deployment.

8.4 Model forecasts

In the previous section we presented experiments demonstrating that our model is able to automatically identify a deployment that is close to optimal. In this section we use our model to forecast optimal deployments for a variety of scenarios. These forecasts can then be used to guide future deployments at a larger scale than we were able to test in these experiments. Table 8.3 summarizes model results for a variety of problem sizes and a variety of platform sizes for a larger cluster with the characteristics of the Lyon cluster.

8.5 Conclusion

In this chapter we have presented an approach for determining an optimal hierarchical middleware deployment on cluster, homogeneous resources. In optimal middleware deployment planning approach, We have shown that an optimal deployment for hierarchical middleware systems on clusters is provided by a CSD tree. We presented request performance models followed by throughput models for agents and servers in DIET. We presented experiments validating the DIET throughput performance models and demonstrating that our approach can effectively automatically build a tree for deployment which is nearly optimal and which performs significantly better than other reasonable deployments.

Now a days the application size is getting bigger and thus execution of the applications require high computing power and large data storage. It is not always possible to solve the ap-

DGEMM Size / $\lvert\mathbb{V}\rvert$	10			100			500			1000		
	d	$\lvert\mathbb{A}\rvert$	$\lvert\mathbb{S}\rvert$	d	$\lvert\mathbb{A}\rvert$	$\lvert\mathbb{S}\rvert$	d	$\lvert\mathbb{A}\rvert$	$\lvert\mathbb{S}\rvert$	d	$\lvert\mathbb{A}\rvert$	$\lvert\mathbb{S}\rvert$
25	1	1	1	2	11	12	24	1	24	24	1	24
50	1	1	1	2	11	12	49	1	49	49	1	49
100	1	1	1	2	11	12	50	2	98	99	1	99
200	1	1	1	2	11	12	40	5	195	199	1	199
500	1	1	1	2	11	12	15	34	466	125	4	496

Table 8.3: Predictions for the best degree d, number of agents used $|\mathbb{A}|$, and number of servers used $|\mathbb{S}|$ for different DGEMM problem sizes and platform sizes $|\mathbb{V}|$. The platforms are assumed to be larger clusters with the same machine and network characteristics as the Lyon cluster.

plication on one cluster, homogeneous resources. Thus, users access the distributed resources or clusters to execute their applications in reasonable time. Middleware deployment planning presented in this chapter is for homogeneous resources and cannot be applied to heterogeneous resources. However, finding the best deployment among heterogeneous resources is a hard problem since it amounts to find the best broadcast tree on a general graph, which is known to be NP-complete [15]. So next Chapter 9 provides an heuristic for middleware deployment on heterogeneous resources.

Chapter 9

Automatic Middleware Deployment Planning for Grid

In previous chapter, we have presented an optimal middleware deployment planning for homogeneous resources. Homogeneous deployment planning cannot be applied to heterogeneous resources because incase of homogeneous deployment planning we didn't apply any logic to select parent nodes, nodes that will act as scheduler. In case of heterogeneous node we have to calculate the number of children supported by each node, as we did in homogeneous case and also select nodes that will act as parent nodes in the deployment so as to maximize the throughput of the platform. However, finding the best deployment among heterogeneous resources is a hard problem since it amounts to find the best broadcast tree on a general graph, which is known to be NP-complete [15]. As our goal is to find a deployment on grid so in this chapter we present an heuristic for middleware deployment on heterogeneous resources.

Our objective is to generate a best platform from the available nodes so as to fulfill client demand if client demand is at most equal to the maximum throughput that can be achieved by the use of available nodes in a time step. Throughput is the maximum number of requests that can be completed (real execution is completed) in a time step.

9.1 Platform Deployment

Our target platform architecture is shown in Figure 8.1. The software system architecture and client request execution is done in same manner as defined in Section 8.1.1. In this section we define the resource architecture and some assumptions considered for the middleware deployment on heterogeneous resources.

Resource architecture - The target resource architectural framework is represented by a weighted graph $G = (\mathbb{V}, \mathbb{E}, w, B)$. Each vertex i in the set of vertices $\mathbb{V}$ represents a computing resource with computing power w_i in MFlop/second for resource i. For the heuristic we have assumed that the bandwidth link between the nodes is homogeneous. That means, each edge e in the set of edges $\mathbb{E}$ represents a resource link between resources with edge cost B measured in Mb/second. We do not consider latency in data transfer costs because our model is based on steady-state scheduling techniques [16]. Usually, the latency is paid once for each of the communications that take place. In steady-state scheduling, however, as a flow of messages takes place between two nodes, the latency is paid only one time (when the flow is initially

established) for the whole set of messages. Therefore latency will have an insignificant impact on our model and so we do not take it into account.

Deployment assumptions - A valid deployment consists of a mapping of a hierarchical arrangement of agents and servers onto the set of resources $\mathbb{V}$. Any server or agent in a deployment must be connected to at least one other element; thus a deployment can have only connected nodes. A valid deployment will always include at least the root-level agent and one server. Each node $v \in \mathbb{V}$ can be assigned to either exactly one server s, exactly one agent a or the node can be left idle. Thus if the total number of agents is $|\mathbb{A}|$, the total number of servers is $|\mathbb{S}|$, and the total number of resources is $|\mathbb{V}|$, then
$|\mathbb{A}| + |\mathbb{S}| \leq |\mathbb{V}|$.

Note that since the use of multiple agents is designed to distribute the cost of services such as scheduling, there is no performance advantage to have an agent with a single child. Thus, any chain can and should be reduced by moving the leaf child of the chain into the position of the first agent in the chain. The only exception to this policy is for the root-level agent with a single server child; this "chain" can not be reduced.

We consider that at the time of deployment we do not know the client locations or the characteristics of the client resources. Thus clients are not considered in the deployment process and, in particular, we assume that the set of resources used by clients is disjoint from $\mathbb{V}$. For the simplicity defined variables are listed in Table 9.1.

Variable	Representation
$\mathbb{V}$	set of computing resources
i	computing resoruce
w_i	computing power of resource i
$\mathbb{E}$	set of edges
e	resoruce link between two resoruces
$\mathbb{S}$	set of servers
s	server
$\mathbb{A}$	set of agents
a	agent

Table 9.1: A summary of notations used to define platfrom deployment.

9.2 Heuristic for middleware deployment on heterogeneous resources

Our objective is to *find a deployment that provides the maximum throughput ρ of completed requests per second*. A *completed request* is one that has completed both the scheduling and service request phases and for which a response has been returned to the client. When the maximum throughput can be achieved by multiple distinct deployments, the preferred deployment is the one using the least resources.

We define the scheduling request throughput of node i in requests per second, ρ_{sched_i}, as the rate at which requests are processed by the scheduling phase (see Section 8.1.1). Likewise, we define the service throughput of node i in requests per second, $\rho_{service_i}$, as the rate at which the servers produce the services required by the clients.

Lemma 4. *The completed request throughput ρ of a deployment is given by the minimum of the scheduling request throughput ρ_{sched_i} among all resources i and the service request throughput $\rho_{service}$ of the deployment.*

$$\rho = min(\forall v_i(\rho_{sched_i}), \rho_{service})$$

From Lemma 4 it is clear that throughput of the deployment platform is dependent on the scheduling throughput of each node and servicing throughput of the deployment. But as scheduling throughput and servicing throughput depends on the computing power of the node. The placement of the nodes in the hierarchy depends indirectly on the computing power of the nodes. As explained in Section 9.1 all the resources are of different computing power. So selection of the resources for mapping of appropriate middleware element is very crucial for heterogeneous resources even for homogeneous resources it was not easy as shown in experimental results in Chapter 1.

For the sake of simplicity we have defined some procedures for the middleware deployment heuristic.

- `calc_sch_pow` is a function to calculate the scheduling power of each node according to the computing power of the node and number of its children.
- `calc_hier_ser_pow` is a function to calculate the servicing power provided by the hierarchy when load is equally divided among the servers of the hierarchy.
- `sort_nodes` is a function to sort the available nodes according to there scheduling power calculated by function `calc_sch_pow` with maximum number of children.
- `empty_array` is a function to remove all the node ids from the agent and server arrays.
- `add_servers_list` is a function to add id of the nodes that will be servers in the hierarchy.
- `calc_min_val` is a function that calculates the minimum throughput possible among scheduling throughput, servicing throughput of the constructing hierarchy and the client volume.
- `count_child` is a function that counts the number of children that a node can support without decreasing scheduling power below the demanded throughput, while calculating the children.
- `shift_nodes` is a function to shift up the node id in the server array if any server is converted as an agent.
- `plot_hierarchy` is a function to fill the adjacency matrix. Adjacency matrix is filled according to the number of children that each agent (from agent array) can support.
- `write_xml` is a function to generate an XML file according to the adjacency matrix, that is given as an input to GoDIET to deploy the hierarchical platform.

9.3 Request performance modeling

In order to apply the heuristic presented in Section 9.2 to DIET, we must have models for the scheduling throughput and the service throughput in DIET. In this section we define performance models to estimate the time required for various phases of request treatment in DIET. These models will be used in the following section to create the needed throughput models.

We make the following assumptions about DIET for performance modeling. The MA and LA are considered as having the same performance because their activities are almost identical and in practice we observe only negligible differences in their performance.

We assume that the work required for an agent to treat responses from SeD-type children and from agent-type children is the same. DIET allows configuration of the number of responses forwarded by agents; we have configured DIET such that only the reference of best server is forwarded to the parent.

When client requests are sent to the agent hierarchy, DIET is optimized such that large data items like matrices are not included in the problem parameter descriptions (only their sizes are included). These large data items are included only in the final request for computation from client to server. As stated earlier, we assume that we do not have a priori knowledge of client locations and request submission patterns. Thus, we assume that needed data is already in place on the servers and we do not consider data transfer times.

The following variables will be of use in our model definitions. S_{req} is the size in Mb of the message forwarded down the agent hierarchy for a scheduling request. This message includes only parameters and not large input data sets. S_{rep} is the size in Mb of the reply to a scheduling request forwarded back up the agent hierarchy. Since we assume that only the best server response is forwarded by agents, the size of the reply does not increase as the response moves up the tree.

w_i is the computing power in MFlop/second for resource i. W_{req_i} is the amount of computation in MFlop needed by an agent i to process one incoming request. $W_{rep_i}(d_i)$ is the amount of computation in MFlop needed by an agent i to merge the replies from d_i children. W_{pre_i} is the amount of computation in MFlop needed for a server i to predict its own performance for a request. W_{app_i} is the amount of computation in MFlop needed by a server i to complete a service request for *app* service. The provision of this computation is the main goal of the DIET system. For the simplicity defined notations are listed in Table 9.2.

Agent communication model: To treat a request, an agent *receives* the request from its parent, *sends* the request to each of its children, *receives* a reply from each of its children, and *sends* one reply to its parent. The time in seconds required by an agent i for receiving all messages associated with a request from its parent and d_i children is as follows:

$$agent_receive_time = \frac{S_{req} + d_i \cdot S_{rep}}{B} \tag{9.1}$$

Similarly, the time in seconds required by an agent for sending all messages associated with a request to its d_i children and parent is as follows:

$$agent_send_time = \frac{d_i \cdot S_{req} + S_{rep}}{B} \tag{9.2}$$

Server communication model: Servers have only one parent and no children, so the time in seconds required by a server for receiving messages associated with a scheduling request is

Algorithme 9.1 Heuristic: To find the best hierarchy

```
1: for (i=0;i<nodes;i++) do
2:    child[i]=nodes;
3:    sch_pow[i]=calc_sch_pow(child[i],input_values[i]);
4: sort_nodes(sch_pow, sorted_nodes);
5: vir_max_sch_pow = calc_sch_pow(1,input_values[sorted_nodes[0]]);
6: acount=0; ncount=0;scount=0;
7: agents[acount++]=sorted_nodes[ncount++];
8: servers[scount++]=sorted_nodes[ncount++];
9: d=1; nb_sed=d; node_used=nb_sed+1;
10: vir_max_sch_pow=calc_sch_pow(d,input_values[agents[0]]);
11: vir_max_ser_pow=calc_hier_ser_pow(input_values[],nb_sed,servers[]);
12: if (vir_max_sch_pow < min_ser_cv) then
13:    select_d=1; nb_sed=d;nodes_hierarchy=2;
14: else
15:    throughput_diff = | vir_max_sch_pow - min_ser_cv | ;
16:    while (diff>throughput_diff) do
17:       diff=throughput_diff;
18:       d++; nb_sed=d; node_used=nb_sed+1;
19:       empty_arrays(agents,servers);
20:       acount=0; ncount=0; scount=0; count=1;
21:       agents[acount]=sorted_nodes[ncount]; ncount++; acount++;
22:       vir_max_sch_pow=calc_sch_pow(d,input_values[agents[0]]);
23:       add_servers_list(sorted_nodes[],servers[],d,scount);
24:       vir_max_ser_pow=calc_hier_ser_pow(input_values[],nb_sed,servers[]);
25:       throughput_diff=calc_min_val(vir_max_ser_pow,client_volume,vir_max_sch_pow);
26:       count_child(vir_max_sch_pow,nodes,add_child[],input_values[]);
27:       while ((node_used<nodes)&&(vir_max_sch_pow>vir_max_ser_pow)) do
28:          child_added=0;
29:          if (add_child[sorted_nodes[ncount]]>1) then
30:             nb_sed−−; ncount++; agents[acount]=servers[0]; acount++;
31:             scount=shift_nodes(scount,servers); scount−−;
32:             servers[scount]=sorted_nodes[d+count];
33:             count++; scount++; nb_sed++; node_used++; child_added++;
34:             while (child_added<add_child[sorted_nodes[ncount]]) do
35:                vir_max_ser_pow=calc_hier_ser_pow(input_values[],nb_sed,servers[]);
36:                if ((vir_max_ser_pow<client_volume)&&(node_used<
                   nodes)&&(vir_max_ser_pow<vir_max_sch_pow)) then
37:                   servers[scount]=sorted_nodes[d+count]; scount++;
38:                   count++; nb_sed++; child_added++; node_used++;
39:                   vir_max_ser_pow=calc_hier_ser_pow(input_values[],nb_sed,servers[]);
40:                else
41:                   child_added=add_child[sorted_nodes[ncount]];
42:          else
43:             node_used=nodes;
44:          throughput_diff=calc_min_val(vir_max_ser_pow,client_volume,vir_max_sch_pow);
45:       nodes_hierarchy=nb_sed+acount;
46:       if (vir_max_sch_pow<vir_max_ser_pow) then
47:          if (diff<throughput_diff) then
48:             selected_d=d-1;
49:          else
50:             selected_d=d;
51:             diff=throughput_diff;
52:       if (d==nodes-1) then
53:          diff=throughput_diff;
54: plot_hierarchy(hierarchy,nodes,add_child,sorted_nodes,nodes_hierarchy,selected_d);
55: write_xml(nodes_hierarchy,hierarchy,sed_binary_name,agent_binary_name,xml_file_name);
```

Notation	Representation
ρ	throughput of the platfrom
ρ_{sched_i}	scheduling request throughput of resource i
$\rho_{service}$	service request throughput
S_{req}	size of incomming request
S_{rep}	size of the reply
W_{pre_i}	amount of computation of resoruce i to merge
	the replies from d_i children
W_{sel_i}	amount of computation of resoruce i needed
	to process the server reply
W_{fix_i}	fixed cost to process the server reply
W_{req_i}	amount of computation need by resoruce i
	to process one incoming request
W_{app_i}	amount of computation needed by a server i
	to complete a service request for *app* service
d_i	children supported by resource i
B	Bandwidth link between resoruces
N	Number of requests completed by $\mathbb{S}$ in a time step
C	Constant to denote time

Table 9.2: A summary of notations used to define performance model.

as follows:

$$server_receive_time = \frac{S_{req}}{B} \tag{9.3}$$

The time in seconds required by a server for sending messages associated with a request to its parent is as follows:

$$server_send_time = \frac{S_{rep}}{B} \tag{9.4}$$

Agent computation model: Agents perform two activities involving computation: the processing of incoming requests and the selection of the best server amongst the replies returned by the agent's children.

There are two activities in the treatment of replies: a fixed cost W_{fix_i} in MFlops and a cost W_{sel_i} that is the amount of computation in MFlops needed to process the server replies, sort them, and select the best server by agent i. Thus the computation associated with the treatment of replies is given by

$$W_{rep_i}(d_i) = W_{fix_i} + W_{sel_i} \cdot d_i$$

The time in seconds required by the agent for the two activities is given by the following equation.

$$agent_comp_time = \frac{W_{req_i} + W_{rep_i}(d_i)}{w_i} \tag{9.5}$$

Server computation model: Servers also perform two activities involving computation: performance prediction as part of the scheduling phase and provision of application services as part of the service phase.

We suppose that a deployment with a set of servers $\mathbb{S}$ completes N requests in a given time step. Then each server i will complete N_i requests, a fraction of N such that:

$$\sum_{i=1}^{N} N_i = N \tag{9.6}$$

On average each server i do prediction of N requests and complete N_i service request in a time step. Lets say, the servers as a group require C seconds to complete the N requests, then

$$C = \frac{W_{pre_i} \cdot N + W_{app_i} \cdot N_i}{w_i} \tag{9.7}$$

From Equation 9.7, we can calculate the requests completed by each server i as:

$$N_i = \frac{C \cdot w_i - W_{pre_i} \cdot N}{W_{app_i}} \tag{9.8}$$

From Equation 9.6 and Equation 9.8, we get time taken by the servers to process N requests as

$$C = N \cdot \frac{1 + \sum_{i=1}^{N} \frac{W_{pre_i}}{W_{app_i}}}{\sum_{i=1}^{N} \frac{w_i}{W_{app_i}}} \tag{9.9}$$

so, time taken by the servers to process one request is

$$server_comp_time = \frac{1 + \sum_{i=1}^{N} \frac{W_{pre_i}}{W_{app_i}}}{\sum_{i=1}^{N} \frac{w_i}{W_{app_i}}} \tag{9.10}$$

9.4 Steady-state throughput modeling

In this section we present models for scheduling and service throughput in DIET. In this section we presents $M(r, s, w)$, no internal parallelism model 7.1.1 for the capability of a computing resource to do computation and communication. In this model, a computing resource has no capability for parallelism: it can either send a message, receive a message, or compute. Only a single port is assumed: messages must be sent serially and received serially. This model may be reasonable for systems with small messages as these messages are often quite CPU intensive.

Servicing throughput of server i is:

$$\rho_{service_i} = \frac{N_i}{C + \frac{S_{req}+S_{rep}}{B} \cdot N} \tag{9.11}$$

So according to Equations 9.6 and 9.11, servicing throughput of platform is:

$$\rho_{service} = \sum_{i=1}^{N} \frac{N_i}{C + \frac{S_{req}+S_{rep}}{B} \cdot N} = \frac{1}{\frac{1+\sum_{i=1}^{N} \frac{W_{pre_i}}{W_{app_i}}}{\sum_{i=1}^{N} \frac{w_i}{W_{app_i}}} + \frac{S_{req}+S_{rep}}{B}} \tag{9.12}$$

The scheduling throughput ρ_{sched} in requests per second is given by the minimum of the throughput provided by the servers for prediction and by the agents for scheduling.

According to Lemma 1, the completed request throughput ρ of a deployment is given by the minimum of the scheduling request throughput ρ_{sched} and the service request throughput $\rho_{service}$, so the following equation presents the throughput of the platform,

$$\rho = min\left(\frac{1}{\frac{W_{pre_i}}{w_i}+\frac{S_{req}}{B}+\frac{S_{rep}}{B}}, \frac{1}{\frac{W_{req_i}+W_{rep_i}(d_i)}{w_i}+\frac{S_{req}+d_i\cdot S_{rep}}{B}+\frac{d_i\cdot S_{req}+S_{rep}}{B}}, \frac{1}{\frac{S_{req}}{B}+\frac{S_{rep}}{B}+\frac{1+\sum_{i=1}^{N}\frac{W_{pre_i}}{W_{app_i}}}{\sum_{i=1}^{N}\frac{w_i}{W_{app_i}}}}\right) \forall v_i \quad (9.13)$$

9.5 Experimental Results

In this section we present experiments designed to test the ability of our deployment model to correctly identify good real-world deployments. Since our performance model and deployment approach focus on maximizing steady-state throughput, our experiments focus on testing the maximum sustained throughput provided by different deployments.

9.5.1 Experimental Design

Software: DIET 2.0 is used for all deployed agents and servers; GoDIET [88] version 2.1.0 is used to perform the actual software deployment.

Job types: In general, at the time of deployment, one can know neither the exact job mix nor the order in which jobs will arrive. Instead, one has to assume a particular job mix, define a deployment, and eventually correct the deployment after launch if it wasn't well-chosen. For these tests, we consider the DGEMM application, a simple matrix multiplication provided as part of the Basic Linear Algebra Subprograms (BLAS) package [32]. For example, when we state that we use DGEMM 100, it signifies that we use matrix multiplication with square matrices of dimensions 100x100. For each specific throughput test we use a single problem size; since we are testing steady-state conditions, the performance obtained should be equivalent to that one would attain for a mix of jobs with the same average execution time.

Workload: Measuring the maximum throughput of a system is non-trivial: if too little load is introduced the maximum performance may not be achieved, if too much load is introduced the performance may suffer as well. A unit of load is introduced via a script that runs a single request at a time in a continual loop. We then introduce load gradually by launching one client script every second. We introduce new clients until the throughput of the platform stops improving; we then let the platform run with no addition of clients for 10 minutes. Figure 9.1 shows the manner in which requests are submitted to the deployment by the clients launched on machines.

9.5.2 Model Parametrization

Table 9.3 presents the parameter values we use for DIET in the model. Our goal is to parametrize the model using only easy-to-collect micro-benchmarks. In particular, we seek to use only val-

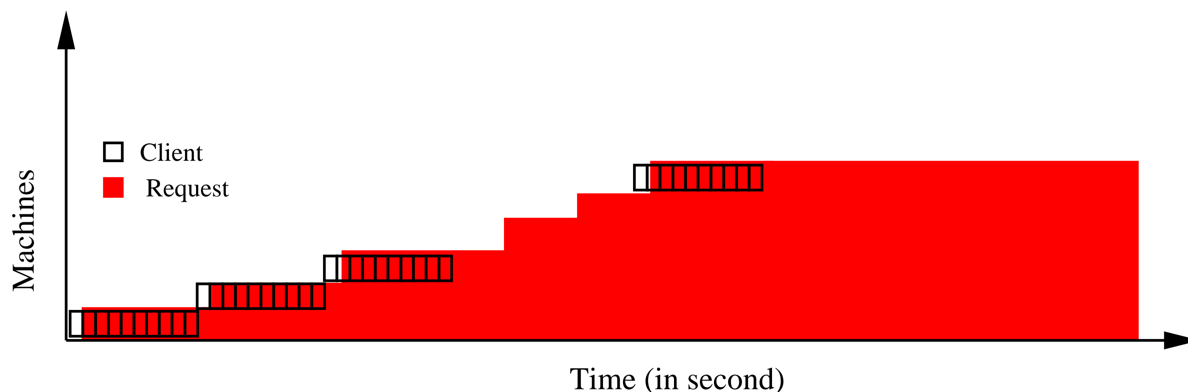


Figure 9.1: Explanation of workload introduced by submitting requests with the increase in the launch of clients on each machine.

ues that can be measured using a few clients executions. The alternative is to base the model on actual measurements of the maximum throughput of various system elements; while we have these measurements for DIET, we feel that the experiments required to obtain such measurements are difficult to design and run and their use would prove an obstruction to the application of our model for other systems.

To measure message sizes S_{req} and S_{rep} we deployed a Master Agent (MA) and a single DGEMM server (SeD) on the Lyon cluster and then launched 100 clients serially from Lyon cluster. We collected all network traffic between the MA and the SeD machines using `tcpdump` and analyzed the traffic to measure message sizes using the Ethereal Network Protocol analyzer[1]. This approach provides a measurement of the entire message size including headers. Using the same MA-SeD deployment, 100 client repetitions, and the statistics collection functionality in DIET, we then collected detailed measurements of the time required to process each message at the MA and SeD level. The parameter W_{rep} depends on the number of children attached to an agent. We measured the time required to process responses for a variety of star deployments including an MA and different numbers of SeDs. A linear data fit provided a very accurate model for the time required to process responses versus the degree of the agent with a correlation coefficient of 0.97. We thus use this linear model for the parameter W_{rep}. Finally, we measured the capacity of our test machines in MFlops using a mini-benchmark extracted from Linpack and used this value to convert all measured times to estimates of the MFlops required.

DIET elements	W_{req} (MFlop)	W_{rep} (MFlop)	W_{pre} (MFlop)	S_{rep} (Mb)	S_{req} (Mb)
Agent	1.7×10^{-1}	$4.0\times10^{-3} + 5.4\times10^{-3}\cdot d$	-	5.4×10^{-3}	5.3×10^{-3}
SeD	-	-	6.4×10^{-3}	6.4×10^{-5}	5.3×10^{-5}

Table 9.3: Parameter values for middleware deployment on Lyon site of Grid'5000

[1]http://www.ethereal.com

9.5.3 Performance model validation on homogeneous platform

The usefulness of our deployment heuristic depends heavily on the performance model of the middleware. This section presents experiments designed to show the correctness of the performance model presented in previous section. These experiments are performed on Lyon site.

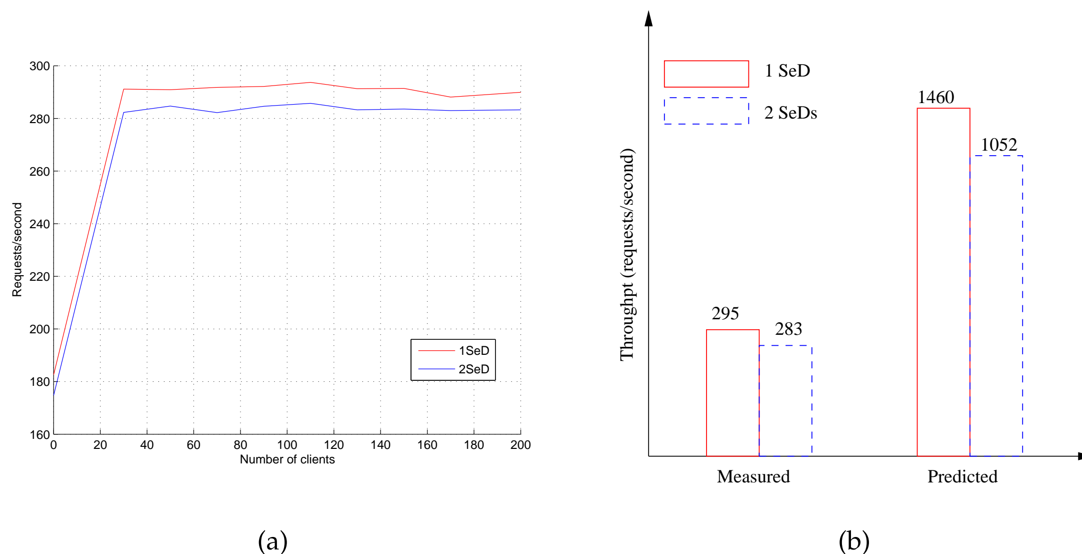


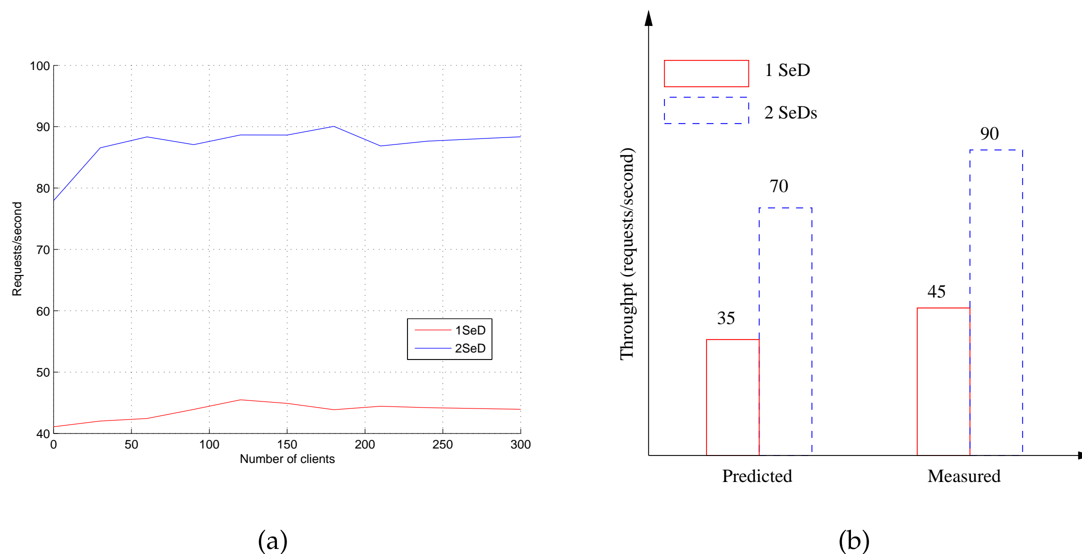
Figure 9.2: Star hierarchies with one or two servers for DGEMM 10x10 requests. (a) Measured throughput for different load levels. (b) Comparison of predicted and measured maximum throughput

Experimental results shown in Figure 9.2 uses a workload of DGEMM 10x10 to compare the performance of two hierarchies: an agent with one server versus an agent with two servers. The model correctly predicts that both deployments are limited by agent performance and that the addition of the second server will in fact hurt performance. Important is the correct prediction by our model to judge the effect of adding servers than to correctly predict the throughput of the platform.

Experimental results shown in Figure 9.3, uses a workload of DGEMM 200x200 to compare the performance of the same one and two server hierarchies. For this scenario, the model predicts that both hierarchies are limited by server performance and therefore, performance will roughly double with the addition of the second server. The model correctly predicts that the two-server deployment will be the better choice.

In summary, our deployment performance model is able to accurately predict the impact of adding servers to a server-limited or agent-limited deployment.

To verify our heuristic we compared the predicted deployment given by the heuristic with the experimental results presented in Chapter 8. Table 9.4 presents the comparison by reporting the percentage of optimal achieved by the deployments selected by different means.



(a) (b)

Figure 9.3: Star hierarchies with one or two servers for DGEMM 200x200 requests. (a) Measured throughput for different load levels. (b) Comparison of predicted and measured maximum throughput

DGEMM Size	Nodes $\|\mathbb{V}\|$	Optimal Degree	Homogeneous Degree	Heuristic Degree	Heuristic Performance
10	21	1	1	1	100.0%
100	25	2	2	2	100.0%
200	45	3	9	10	-
310	45	15	22	33	89.0%
1000	21	20	20	20	100.0%

Table 9.4: A summary of the percentage of optimal achieved by the deployment selected by our heterogeneous heuristic, optimal degree, and optimal homogeneous model.

9.5.4 Heuristic Validation on heterogeneous cluster

To validate our heuristic we did experiments using two sites, Lyon and Orsay of Grid'5000, a set of distributed computational resources in France. We used 200 nodes of Orsay for the deployment of middleware elements and 30 nodes of Lyon for submitting the requests to the deployed platform.

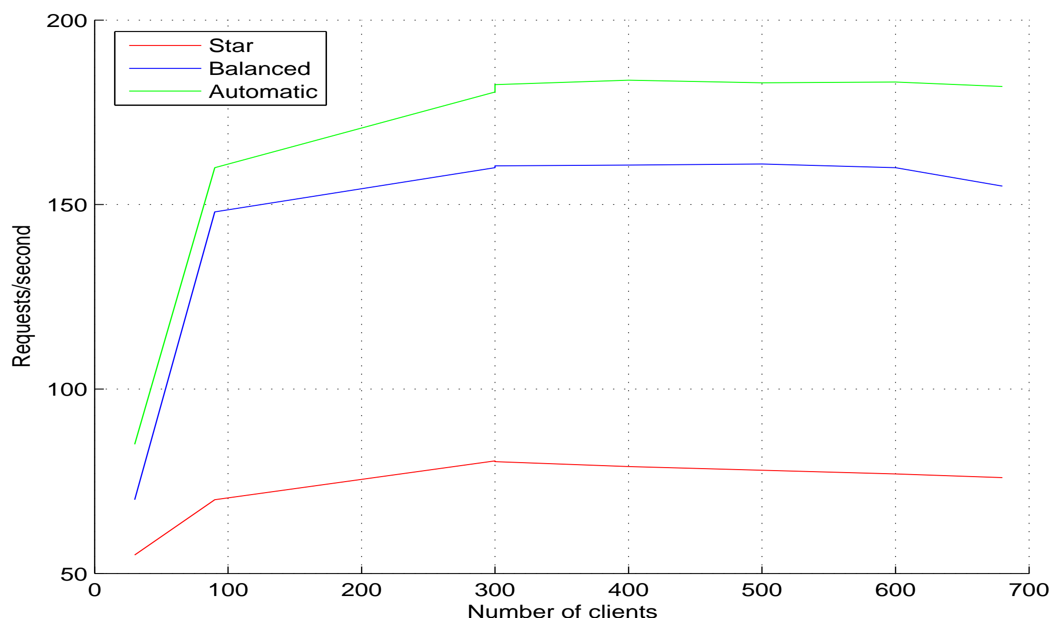


Figure 9.4: Comparison of automatically-generated hierarchy for DGEMM 310*310 with intuitive alternative hierarchies.

To convert the homogeneous cluster into heterogeneous cluster we changed the workload of the reserved nodes by launching different size of matrix multiplication as the background program on some of the nodes. Different size of matrix utilized the computing power in varying manner thus the available computing power of the nodes varies. We used the measured times of Lyon nodes (Section 9.5.2) for Orsay nodes because the machine configuration of two sites is exactly the same. After launching the matrix program in background on machines we used Linpack mini-benchmark to measure the capacity of the nodes in MFlops and used this value to convert all measured times of Lyon nodes to estimates of the MFlops of each node.

We compared two different deployments with the automatically generated deployment by our heuristic. First deployment is the simple star type, where one node act as agent and all rest nodes are directly connected to the agent node and act as servers. In second deployment we deployed a balanced graph, one top agent connected to 14 agents and each agent connected to 14 servers except one that could had only 3 servers.

Clients submitted DGEMM problems of two different sizes. First we tested the deployments with DGEMM 310*310. Heuristic generated deployment used only 156 nodes and deployment is organized as: top agent connected with 9 agents and each agent again connected to 9 agents.

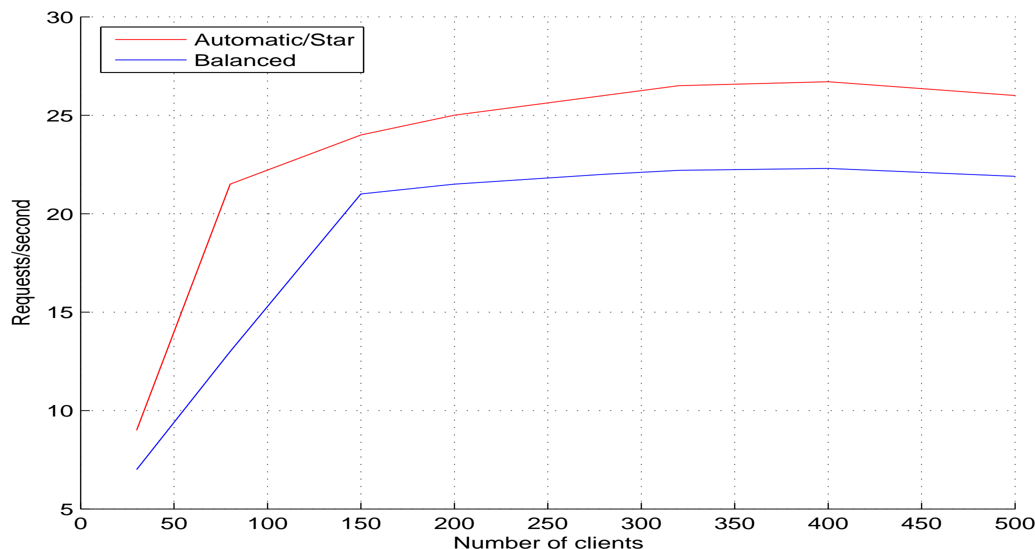


Figure 9.5: Comparison of automatically-generated hierarchy for DGEMM 1000* 1000 with intuitive alternative hierarchy.

Two agents are connected with 9 SeDs, 6 agents are connected with 7 SeDs and one with 5 SeDs. Automatically generated deployment performed better than the two comparing deployments. Results of the experimental results are shown in Figure 9.4.

Second experiment is done with DGEMM1000*1000. Heuristic generated a star deployment for this problem size. Results in Figure 9.5 shows that star performed better than the second compared deployment.

9.6 Conclusion

We present a deployment heuristic that predicts the maximum throughput that can be achieved by the use of available nodes. Heuristic predicts good deployment for both homogeneous ad heterogeneous resources. Comparison is done to test the heuristic for homogeneous resources and heuristic performed up to 90% as compared to the homogeneous optimal algorithm presented in Chapter 8. To validate the heuristic experiments are done on site of Grid'5000. Experiments have shown that automatically generated deployment by the heuristic performs better than some intuitive deployments.

Chapter 10

Improve Throughput of a Deployed Hierarchical NES

It is not always evident to deploy the middleware platform from scratch according to the specifications of new jobs. Sometime it is useful to modify the existing hierarchy with less cost and time then to deploy a new platform according to new requests. This chapter presents a model to analyze existing hierarchical deployments. An mathematical model is used to analyze existing hierarchical deployment. It identify a bottleneck node, and remove the bottleneck by adding resources in the appropriate area of the hierarchy. The solution is iterative and heuristic in nature. We validated our model by applying it to improve an user defined deployment platform's for DIET middleware.

10.1 Hierarchical deployment model

We model a collection of heterogeneous resources (a processor, a cluster, etc.) and the communication links between them as nodes and edges of an undirected hierarchical graph (tree-shaped). Each node is a computing resource capable of computing and communicating with its neighbors at same/different rates. We assume that one specific node, referred as client, initially generates requests. The client sends the requests to its neighbor node, which is the head of the hierarchy. This node check whether the request is right (means having all the parameters that a request should have), and if so, then the request is flooded to the neighboring nodes down in the hierarchy. These nodes forward the requests down the hierarchy till request reaches to the connected servers. These servers send reply packets to their parent nodes. These packets contain the status (memory available, number of resources available, performance prediction, ...) of the server. Parent node compares the reply packet sent to it, by each of its connected server and selects the best server among them. Now the reply packet of the selected server is sent by the node to its parent. Finally the reply packet is received by the head node. Best server (or list of available servers, ranked in order of availability) among the selected servers is being informed by the head node to the client. The client contacts the selected server. Then client sends the input data to the server. Finally the server executes the function on behalf of the client and returns the results.

The target architectural framework is represented by a weighted graph $G = (V, E, w, B)$. Each $P_i \in V$ represents a computing resource of computing power w_i, meaning that node P_i

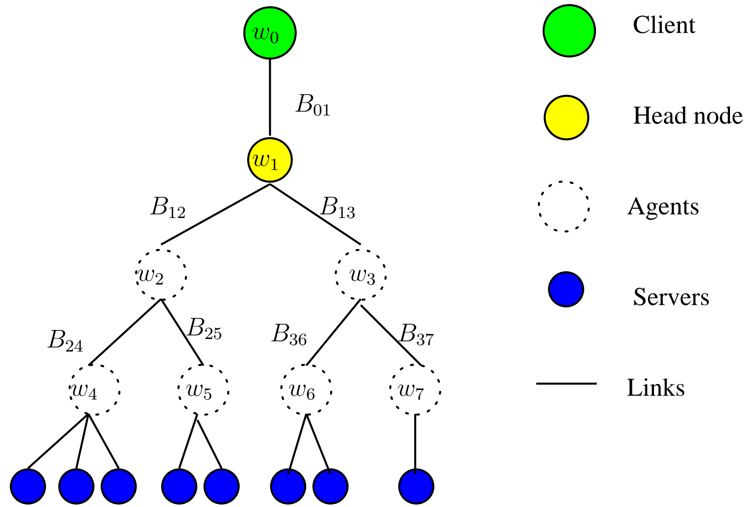


Figure 10.1: Architectural model.

execute w_i MFlop/second (so bigger the w_i, the faster the computing resource P_i). There is a client node, i.e. a node P_c, which generates the requests that are passed to the following nodes[1]. Each link $P_i \rightarrow P_j$ is labelled by the bandwidth value $B_{i,j}$ which represents the size of data sent per second between P_i and P_j. Measuring unit of bandwidth of link is Mb/second.

The size of the request generated by the client is $S_i^{(in)}$ and the size of the request's reply created by each node is $S_i^{(out)}$. The measuring unit of these quantities is thus in Mb/request. The amount of computation needed by P_i to process one incoming request is denoted by $W_i^{(in)}$ and the amount of computation needed by P_i to merge the reply requests of its children is denoted by $W_i^{(out)}$. We denote by $W_i^{(\text{DGEMM})}$ the amount of computation needed by P_i to process a generic problem (for example, BLAS [32] matrix multiplication called DGEMM).

The current architectural model does not consider data management, we focus on data in place applications (due to security problem, or parameter programming). Nevertheless, data management could be easily added like a new level of the modeling. The application target and the missing evaluate tools to estimate the data movement explain why we does not consider this aspect. In the same paradigm we consider the result that are very small can be neglected.

10.2 Throughput calculation of an hierarchical NES

Our objective is to compare the maximum number of requests answered per second by a specific type of architecture so that best architecture can be selected. $\alpha_i^{(in)}$ denotes the number of incoming request (request coming from client) processed by P_i during one time-unit. Note that this number is not necessarily an integer and may be a rational. In a similar way, $\alpha_i^{(out)}$ is the number of outgoing requests (selection of the best server based on the reply packets)

[1] We use only one Client node for sake of simplicity but modeling many different clients with different problem types can be done easily.

computed during one time-unit by the node P_i. Servers are connected to the local agents at the last level of the graph. Therefore, $\alpha_i^{(\text{DGEMM})}$ denote the number of problem solved by the node P_i if P_i is a server.

10.2.1 Find a bottleneck

Number of requests replied in a time step depend on bandwidth of the link, size of the request, fraction of request being computed by a processor in a time step and the computing power of the processor. Therefore, we have the following constraints:

Computation constraint for agent: $\forall P_i : \dfrac{\alpha_i^{(in)} \times W_i^{(in)} + \alpha_i^{(out)} \times W_i^{(out)}}{w_i} \leq 1$

Note that, it is necessary for each incoming request, that there should be a corresponding reply, and that each request is broadcasted along the whole hierarchy. Thus, there is no need to make a distinction between the $\alpha_i^{(in)}$ and the $\alpha_i^{(out)}$, that all are equal to the maximum throughput of the platform ρ. The previous equation can thus be simplified in the following equation:

$$\forall P_i : \rho \times \frac{W_i^{(in)} + W_i^{(out)}}{w_i} \leq 1 \tag{10.1}$$

Communication constraint for agent: ρ request for computations and ρ replies to these requests are transmitted per time-unit along each link $P_i \to P_j$. Therefore, we have:

$$\forall P_i \to P_j : \rho \times \frac{S_i^{(in)} + S_j^{(out)}}{B_{i,j}} \leq 1 \tag{10.2}$$

Server's computation constraint: Each server P_i process $\alpha_i^{(\text{DGEMM})}$ problem per time-unit. Therefore, we have

$$\forall P_i \text{ s.a } P_i \text{ is a server} : \frac{\alpha_i^{(\text{DGEMM})} \times W_i^{(\text{DGEMM})}}{w_i} \leq 1$$

All these values are linked to ρ by the equation $\rho = \sum_{P_i \text{ s.a } P_i \text{ is a server}} \alpha_i^{(\text{DGEMM})}$. Therefore, we have

$$\rho \leq \sum_{P_i \text{ s.a } P_i \text{ is a server}} \frac{w_i}{W_i^{(\text{DGEMM})}} \tag{10.3}$$

No internal parallelism: In this model, the computation and other operation performed by the node is done sequentially, so the summation of all operations performed by an agent should be less than the time step. Therefore, for all P_i, we have:

$$\underbrace{\rho \left(\frac{S_{parent(i)}^{(in)}}{B_{parent(i),i}} + \frac{S_i^{(out)}}{B_{parent(i),i}} \right)}_{\text{Communications with the parent}} + \underbrace{\rho \left(\sum_{P_i \to P_j} \frac{S_i^{(in)} + S_j^{(out)}}{B_{i,j}} \right)}_{\text{Communications with the slaves}} + \underbrace{\rho \left(\frac{W_i^{(in)} + W_i^{(out)}}{w_i} \right)}_{\text{Local computations}} \leq 1. \tag{10.4}$$

It is noteworthy that if on P_i, computations can be performed in parallel with communications, the previous constraints should be changed as:

$$\rho \underbrace{\left(\frac{S_{parent(i)}^{(in)}}{B_{parent(i),i}} + \frac{S_i^{(out)}}{B_{parent(i),i}} \right)}_{\text{Communications with the parent}} + \rho \underbrace{\left(\sum_{P_i \to P_j} \frac{S_i^{(in)} + S_j^{(out)}}{B_{i,j}} \right)}_{\text{Communications with the slaves}} \leq 1. \tag{10.5}$$

Theorem 3. *The maximum number of requests that can be processed by the platform in steady state is obtained from constraints* (10.1), (10.2), (10.3), (10.4) *and* (10.5) *and is represented by the following formula:*

$$\rho = \min \left(\frac{w_i}{W_i^{(in)} + W_i^{(out)}}, \frac{B_{i,j}}{S_i^{(in)} + S_j^{(out)}}, \sum_{P_i \text{ s.a } P_i \text{ is a server}} \frac{w_i}{W_i^{(\mathrm{DGEMM})}}, \frac{1}{\frac{S_{parent(i)}^{(in)}}{B_{parent(i),i}} + \frac{S_i^{(out)}}{B_{parent(i),i}} + \sum_{P_i \to P_j} \frac{S_i^{(in)} + S_j^{(out)}}{B_{i,j}} + \frac{W_i^{(in)} + W_i^{(out)}}{w_i}} \right) \tag{10.6}$$

Theorem 4. *When maximizing the throughput, at least one of the constraints* (10.1), (10.2), (10.3), (10.4) *and* (10.5) *is tight. This constraint represent the bottleneck of the platform.*

10.2.2 Remove the bottleneck

Even when neglecting the servers constraints, finding the best topology is a hard problem since it amounts to find the best broadcast tree on a general graph, which is known to be NP-complete [15]. Note that even when neglecting the request mechanism, as soon as you take in account the communications of the problem's data, the problem of finding the best deployment becomes NP-complete too [13].

Even in real life, the topology of the underlying platform is particular and enforce some parts of the deployment. Therefore, we propose to improve the throughput of a given deployment by breaking its bottleneck. Using the previous theorems, we can find the bottlenecks and get rid of them by adding a supporter agent to the parent of a loaded agent so as to divide the load of that particular agent. We add new node according to the greedy Algorithm 10.1.

Algorithme 10.1 Algorithm to add LA.

1: **while** (number of available nodes > 0) **do**
2: Calculate the throughput ρ of structure.
3: Find a node whose constraint is tight and that can be split
4: **if** no such node exist **then**
5: The deployment cannot be improved. Exit
6: Split the load by adding new node to its parent
7: Decrease the number of available nodes

In Algorithm 10.1, line 3 checks whether it is possible to divide the load of a node or not. There may be many reasons for this condition to be false, for example, a node P_i having only one child cannot divide its load.

10.3 Parameter measurement

We used Distributed Interactive Engineering Toolbox (DIET) [24] to estimate the values of the different parameters. Name of DIET's elements Master Agent (MA), Local Agent (LA), and servers (SeDs) will be referred in the explanation of the experiments. Experiments are done on a homogeneous cluster, composed of 16 processors (bi-PIII 1.4Ghz). To observe the effect on the computation time of each component, to process one request, we did different experiments by varying the number of LAs and SeDs with one MA and one client. We focus on linear approximation, as this approximation gives good result for our modeling.

The effect of adding LAs is shown in Figure 10.2, in the form of time taken to execute one request by each component. The time taken to compute a request by MA, increases with the addition of the LAs. MA take approximately 0.01 seconds for an incoming request and the time to compute an outgoing request is very very small, varies between 0.0001 to 0.00018 seconds. To compute an incoming and outgoing request by an LA is approximately between 0.0066 to 0.0076 and 0.0001125 to 0.000116 seconds respectively. The time taken by SeD for computation is very less effected by the increase in LAs.

In Figure 10.3, we have shown the effect of number of servers on the components computing capacity. For an incoming and outgoing request, computation time taken by MA, LA and SeD increases linearly. For incoming request MA and LA take approximately 0.009 to 0.01 and 0.008 to 0.022 seconds respectively. In case of outgoing request variation is very small for MA it is between 0.0001 to 0.0002 seconds and for LA 0.0002 to 0.00035 seconds. SeD computation time ranges between 0.00018 to 0.00019 seconds.

The time needed by each component to reply is so small that, to estimate which fraction should be incurred to computations and which fraction should be incurred to communications is very difficult. Therefore, we have considered it to be only computations (and therefore $S_i^{(out)}$ =0) as a very small amount of data is exchanged here.

From these experimental results, it is observed that increasing the number of LAs accordingly increases the overall time needed to process one request on a MA or a LA, the fraction of time spent incurred to computations is almost constant. The behavior is similar when varying the number of SeDs.

From these measurements, we estimated the time taken by the components to communicate and compute the request. $S_i^{(in)}$ is then calculated by summing the communication time taken by each component and dividing it by the bandwidth of the local link. Similarly, the value of $W_i^{(in)}$ and $W_i^{(out)}$ is calculated by dividing the computation time of incoming request and the outgoing request by the processing power. Parameter values are summarized in the Table 10.1.

10.4 Simulation results

To validate our model, we considered a real heterogeneous network that we consider is already deployed according to the DIET middleware elements. We did simulation using a sim-

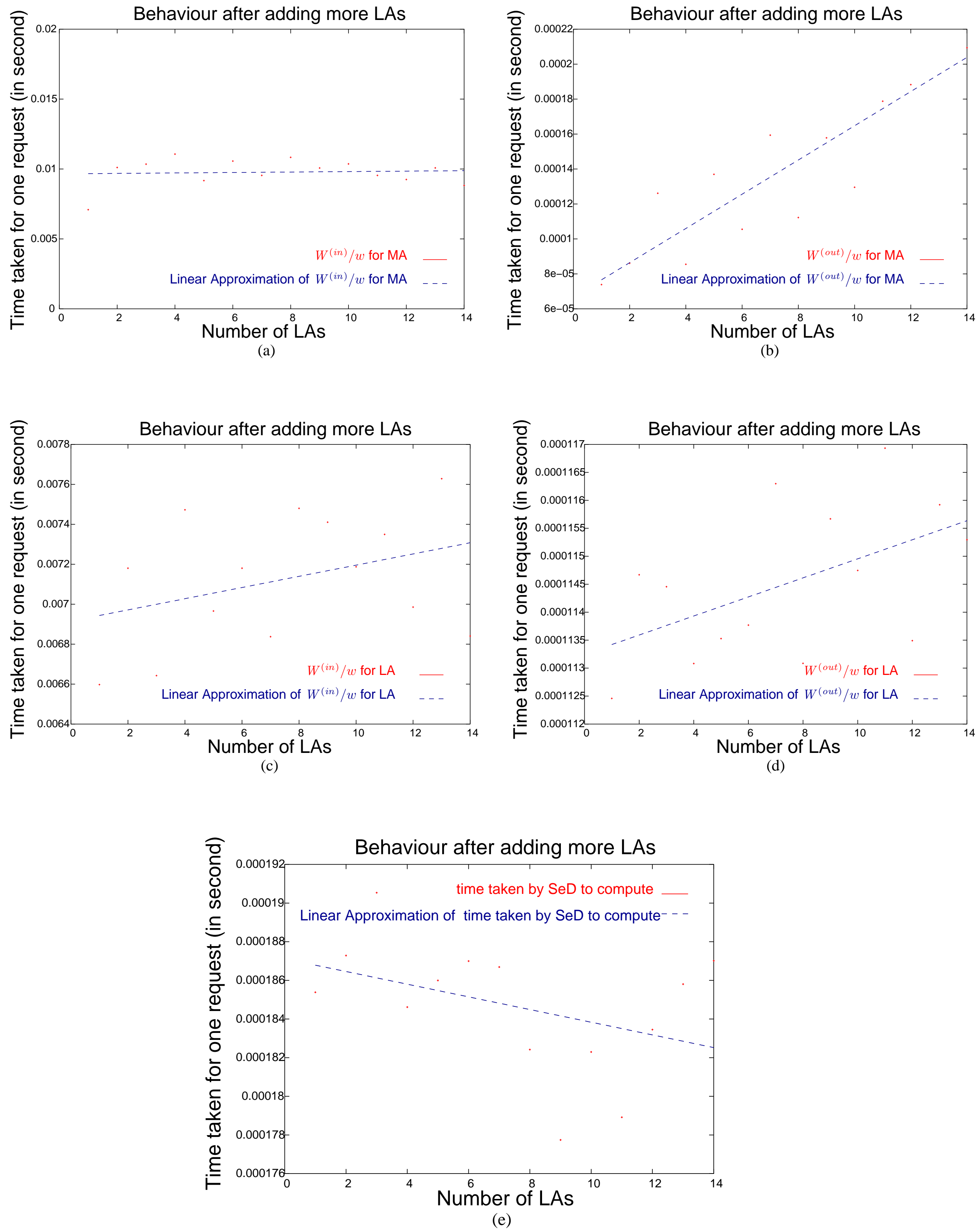


Figure 10.2: Performance calculation by adding LAs.

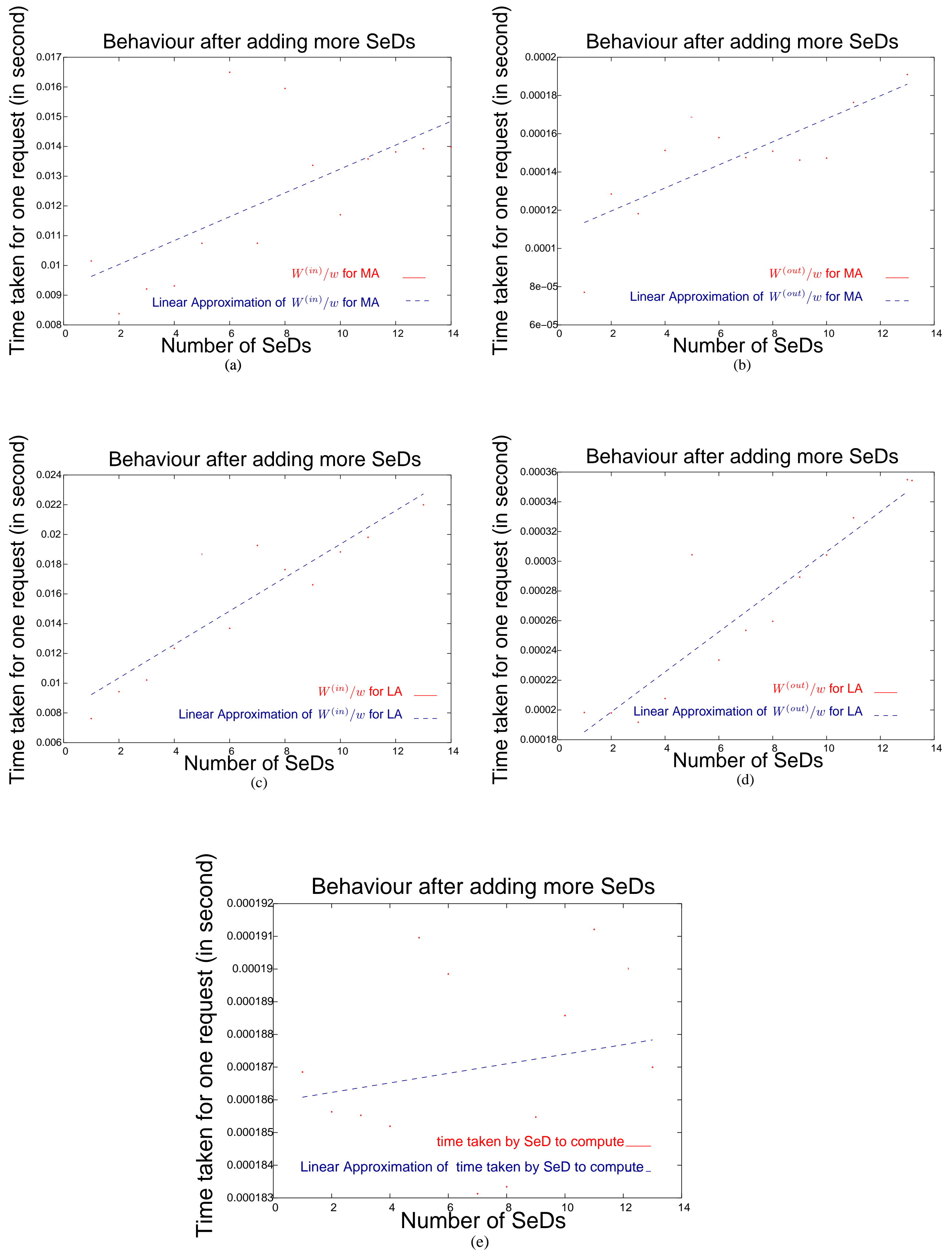


Figure 10.3: Performance calculation by adding SeDs.

Components	$S_i^{(in)}$	$W_i^{(in)}$	$W_i^{(out)}$
Client	0.339	0.014	0
MA	0.010	0.159	0.78 e-3
LA	0.012	0.079	0.19 e-3

Table 10.1: Parameter values to calculate the throughput of a deployment.

ple modeling of a real heterogeneous network shown in Figure 10.4. Diagrammatic view of heterogeneous network as DIET deployed platform is shown in Figure 10.5.

10.4.1 Test-bed

Distributed networks are all heterogeneous, i.e., every node has its own computing power (may be different from other nodes) and bandwidth link between two nodes are also mostly different. In the heterogeneous network, we have one client (`veloce`) and one MA at *Rocquencourt*. This MA is connected to two LAs: one at *Rennes* and another at *Grenoble*. 40 servers (`paraski` cluster) are connected to LA at Rennes. LA at Grenoble is connected with two LAs, LA at Grenoble has 200 servers (`icluster` cluster) and LA at *Sophia* has 14 servers (`galere` cluster). The power of the different nodes and the bandwidth of the different link between the nodes are depicted on Figure 10.5.

Figure 10.4: High speed network (2.5Gb/s) between INRIA research centers and several other research institutes.

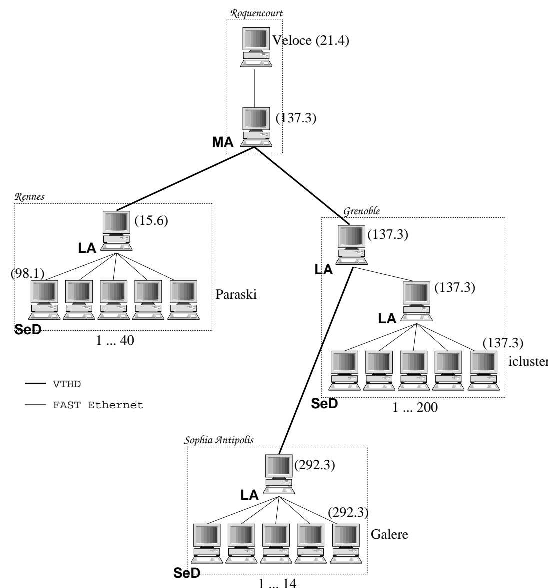


Figure 10.5: Diagrammatic view of testbed.

On this heterogeneous platform, the w_i's range from 15.6 MFlop/s to 292 MFlop/s and the bandwidth $B_{i,j}$ range from 10Mb/s to 2.5Gb/s.

10.4.2 Computing a good deployment

Real heterogeneous network (shown in Figure 10.5) have one client node, one MA, four LAs and 255 servers. Client sends request to the deployed platform through MA. 255 servers are used to do the real execution of the request using Equation 10.3, means tasks execution is balanced among all servers of the platform. The overall throughput of heterogeneous network is calculated by using the formula mentioned in section 10.2.1. The performance of the original deployment (the natural one which is depicted in Figure 10.5) is rather low since it enable to process at most only 4 request per second. But the throughput of the network can be improved by breaking the bottleneck. To improve the throughput of the network, we used the Algorithm 10.1. Requirement of nodes to break the bottleneck is fulfilled by either of two ways. By considering that we have enough available extra nodes that we can use to improve the throughput of the hierarchy. Or by using servers as required nodes to improve the throughput of the hierarchy. Adding new nodes to remove the bottleneck is not very feasible as its not always possible that extra nodes are available to improve the throughput of deployed platform. But it is advantages to use new nodes then replacing servers because by replacing the servers as agents, we limit (decrease) the overall performance of servers and thus the throughput of the platform.

10.4.2.1 Addition of new nodes

Bottleneck of the deployed platform shown in Figure 10.5, is located at `icluster` (see Figure 10.6). `icluster` has 200 servers so, broadcasting all the request and gathering the answer is very time-consuming. Seven more nodes have to be added, so that it will not be the bottleneck of the platform anymore. The eighth node has to be added at the `Rennes`'s cluster. The gateway is very slow and, even if the number of servers is not as important as on the `icluster`, it has become an issue. Nine more nodes are then added on the `icluster` and then two again on Rennes. At this point, we have added total of 18 nodes and the tight equation is Equation (10.3), which means that all servers are working at full speed and that there is no hope of improving the throughput of the platform anymore. Platform with 260 nodes were giving throughput of 4 requests/second, which is improved to 35 requests/second buy adding only 18 nodes.

10.4.2.2 Rearrangement of platform nodes

As stated earlier, first bottleneck of the deployed platform shown in Figure 10.5, is located at `icluster` (see Figure 10.6). From 200 servers, seven servers are converted as LAs to share the load of the LA at icluster, so that it will not be the bottleneck of the platform anymore. Next bottleneck occurred at `Rennes`'s cluster, as the number of servers (40) at Rennes's cluster is more than then number of servers attached to other LAs in the platform. So server at the Rennes's cluster is added as eighth LA to share the load of parent LA and remove the bottleneck from Rennes's cluster. Nine more servers are converted as LAs at `icluster` and then one server is converted to LA at `Rennes`. At this point, total 17 servers are converted to LAs and the tight equation is Equation (10.3). The throughput of deployed platform with 260 nodes is improved from 4requests/second to 33 requests by rearranging the nodes.

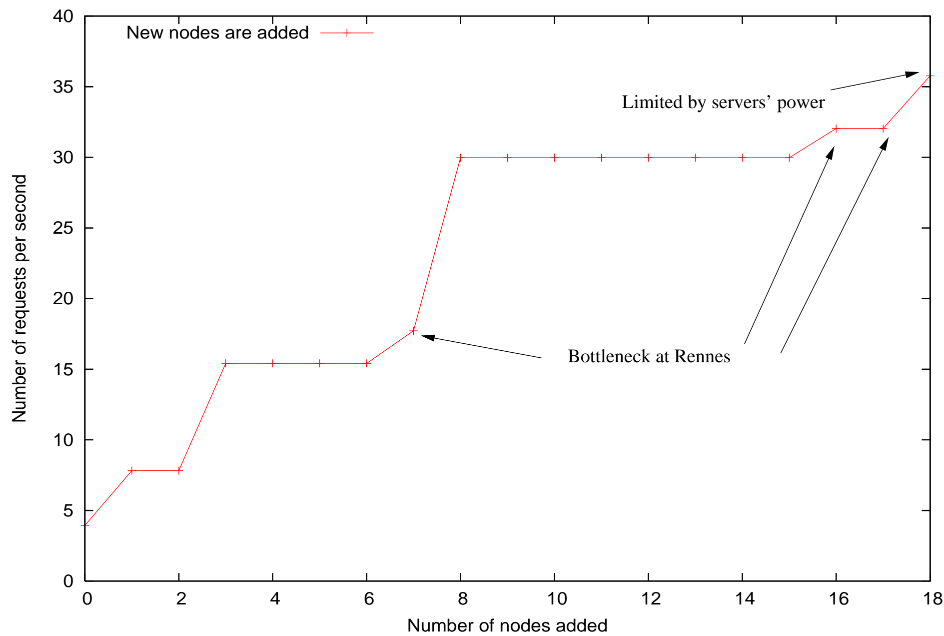


Figure 10.6: Throughput of heterogeneous network as more number of LA are added.

10.5 Conclusion

An efficient deployment for NES environment is very important. We have used theoretical steady-state scheduling framework to propose a new model that enables to find bottlenecks in the organization of a hierarchical NES such as the DIET middleware. This model enables to improve the overall performance of NES by breaking bottlenecks and therefore to perform automatic deployment or redeployment. This model can analyze the effects on performance if different changes are done in the hierarchy's configuration such as number of agents or servers in the hierarchy, problem size, bandwidth, nodes' computing power, parameter values etc.

In Chapters 7, 8, 9, 10, we have presented deployment planning for middleware on distributed resources and technique to improve the existing deployment. Using these deployment planning techniques and launching tool an automatic middleware deployment can be developed. In next Chapter 11, we presents initial steps of an Automatic Deployment Planning Tool (ADePT).

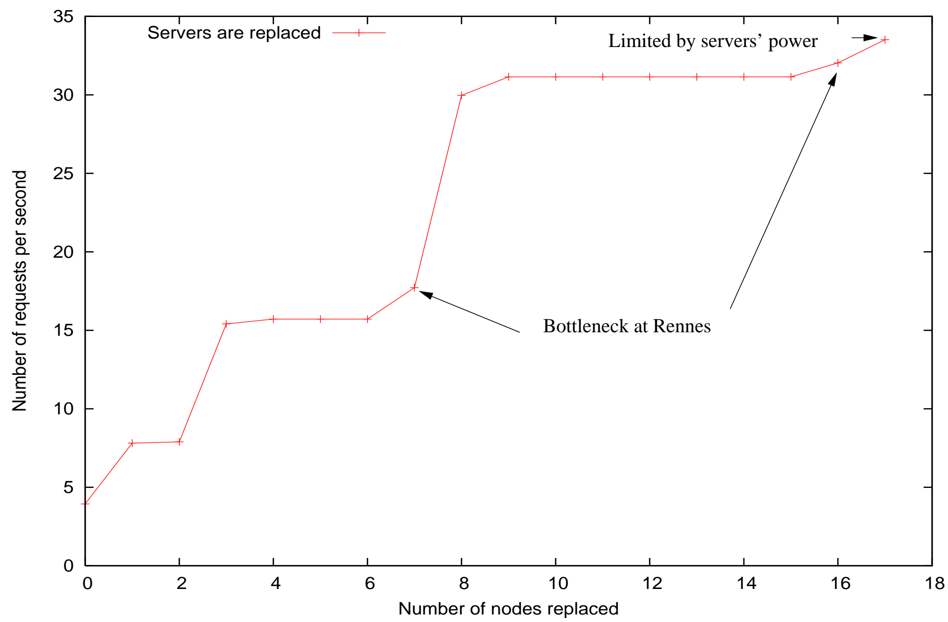


Figure 10.7: Throughput of heterogeneous network as SeD are converted to LA.

Chapter 11

Automatic Deployment Planning Tool

In previous Chapters 8, 9, 10, we have presented theoretical concepts to generate best deployment plan for a middleware. To present our work in a simple form as a tool for end users, in this chapter we present the initial steps to build a tool for automatic deployment planning based on our theoretical concepts.

11.1 Introduction

For simplicity in this chapter we named the deployment planning tool for which initial building steps will be described, as Automatic Deployment Planning Tool (ADePT). ADePT is a deployment planning tool that generates the deployment plan based on resource description, application description, and middleware description (in the form of performance model). Figure 11.1 gives an outline of ADePT working, its input and output. ADePT provides an optimal solution for hierarchical middleware environments on cluster and implement heuristics for hierarchical middleware deployment on grid. ADePT can also determine the bottleneck in the deployed hierarchy and thus helps to know whether the performance of a given deployment can be improved or not.

Resource description provides the information like computing power, storage capacity, etc. about the resources. Resource information can be provided as an XML file to the ADePT. Resource description should define all compute hosts (resources) that a user wants to use. Example of a resource description XML file is shown in Figure 11.2. “Scratch” tag specify a local pathname that deployment tool can use as scratch space (e.g., temp storage of config files). User must have write access to the mentioned directory. “Storage” tag define all storage space that will be needed to run the application. “Bandwidth_link” tag mention the link between two resources. We have considered that link between resources is homogeneous so only one tag is enough. But as the improvement will be done in the ADePT, the bandwidth links between resources can also be submitted as a separate XML file. If resources are homogeneous then only “cluster” tag is enough to simplify the description of large numbers of resources. To define all heterogeneous resources, “compute” tags should be used for individual resources. XML file must include at least one “scratch" tag, one “storage" tag, one “bandwidth_link” tag and one “compute" or one “cluster" tag.

Application description provides the information about the application that will be submitted to the deployed plan, like size of the scheduling request, size of the response request,

amount of computation need for the prediction, execution etc. of the application on the dedicated resource. An example of application description XML file is shown in Figure 11.3.

Deployed plan is the deployed hierarchy of a NES's environment whose performance has to be improved. For an example, the "diet_hierarchy" tag of Figure 6.2, is the deployed plan for a hierarchical middleware DIET. This tag as an XML file should be submitted to ADePT if the DIET hierarchy has to be improved. Generated output by the ADePT is also similar to this XML file.

Performance model is a method to define the functionality of each NES's component according to the NES's working phases. Performance model defines the method to calculate the throughput of each component. Formulation of performance model according to the NES environment is explained in detail in Section 11.2.

Techniques that ADePT use to select the deployment plan is based on the theoretical results presented in Chapters 8, 9, and 10. Theoretical results of Chapter 8 has shown that a complete d-ary spanning tree provides an optimal deployment for hierarchical middleware environments on homogeneous clusters. Chapter 9 has presented an heuristic to deploy hierarchical middleware on grid, by selecting efficient resources from grid according to the functional requirements of the middleware elements. A mathematics model is presented in Chapter 10 that can analyze the deployed platform and find a bottleneck node. Using Algorithm 10.1 the bottleneck can be removed by dividing the load of the node, which is the cause of bottleneck. Load of loaded node is divided by adding a node to its parent and divide the number of its children nodes among the loaded node and newly added node.

11.2 Formulation of performance model for a middleware

ADePT takes performance model of a middleware as an input. Performance models are based on middleware description (functionality of middleware's components in the form of middleware working phases, operating model 7.1.1 etc). In this section an idea is given for the generation of the performance model for a middleware.

Description of working phases of the middleware

Generally NES environments processes *requests* in two phases. First user submit an application, called *scheduling request* to root node. Root node is the entrance node of the deployment, to which all users submit their application. Root node checks the scheduling request and forwards it down in the hierarchy. Other parent node in the hierarchy perform the same operation until the scheduling request reaches the servers. We assume that the scheduling request is forwarded to all servers, though this is a worst case scenario as filtering may be done by the parents based on scheduling request type. Servers may or may not make predictions about performance for satisfying the request, depending on the exact system.

Servers that can perform the service then generate a *scheduling response*. The scheduling response is forwarded back up the hierarchy and the parents sort and select amongst the various scheduling responses. Finally, the root node forwards the chosen scheduling response (i.e., the selected server) to the user.

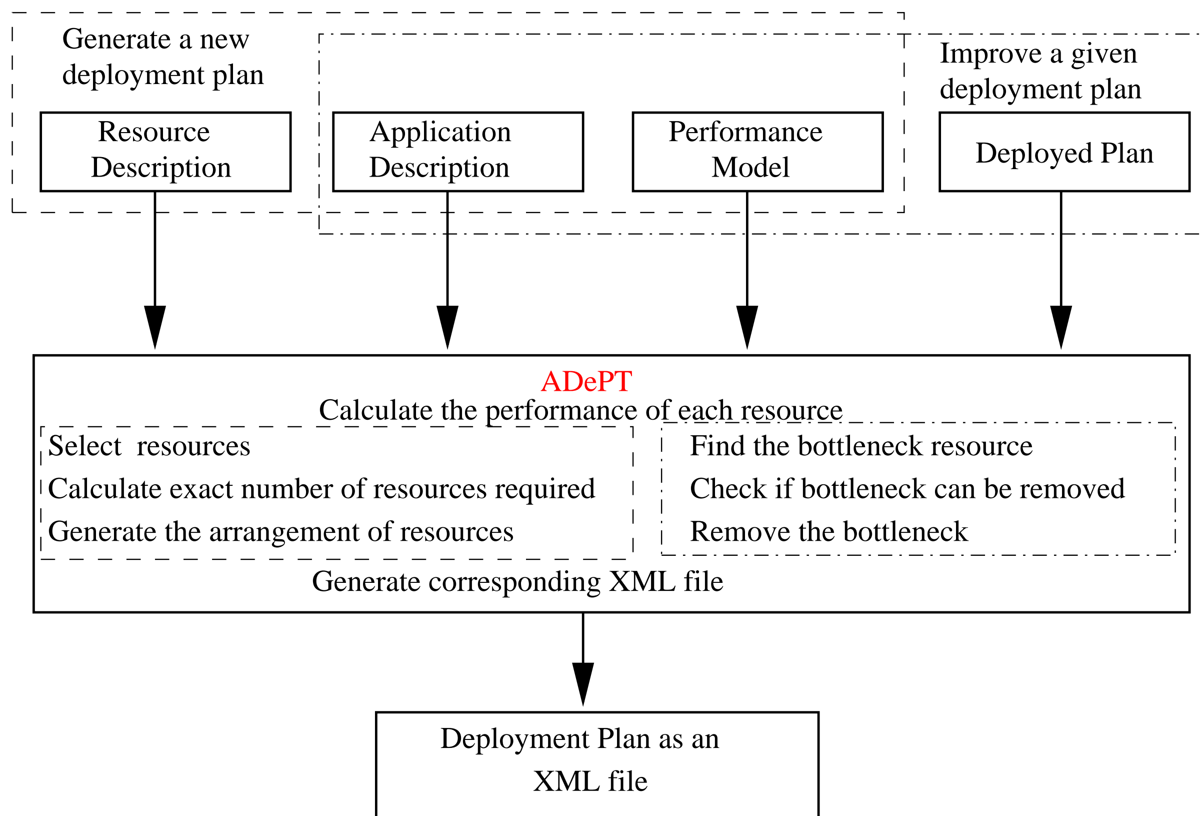


Figure 11.1: Working of Automatic Deployment Planning Tool (ADePT).

User then generates a *service request* which is very similar to the scheduling request but includes the full input data set, if any is needed. The service request is submitted by the user to the chosen server. The server performs the requested service and generates a *service response*, which is then sent back to the user.

Description of performance model

Depending upon the middleware's working phases, the performance models can be defined. We define performance models to estimate the time required for the treatment of requests in various phases by each component of the middleware. According to the middleware's component functionality, performance model can be calculated depending upon the different working phases in which a middleware component has taken part. Below a general idea to generate the performance model for each middleware component is given.

Parent node's communication model: To treat a request, a node *receives* the request from its parent, *sends* the request to each of its children, *receives* a reply from each of its children, and *sends* one reply to its parent. Thus, time required by a parent node is associated with the size of requests and responses received from its parent and children , bandwidth link and number of children.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE resource_description SYSTEM "../resource_descript.dtd">
<resource_description>
    <scratch dir="/homePath/user/scratch_direct"/>
    <storage label="g5kBordeauxDisk">
      <scratch dir="/homePath/user/scratch_runtime"/>
      <scp server="frontale.bordeaux.grid5000.fr" login="pkchouhan"/>
    </storage>
    .
    .
    <storage label="g5kOrsayDisk">
      <scratch dir="/homePath/user/scratch_runtime"/>
      <scp server="frontale.orsay.grid5000.fr" login="pkchouhan"/>
    </storage>
    <bandwidth_link B="190"/>
    <compute label="node1" disk="disk1">
      <ssh server="node1.site.fr"  login="pkchouhan"/>
      <env path="/homePath/user/demo/bin"  LD_LIBRARY_PATH="/homePath/user/
           demo/lib"/>
    </compute>
                .
                .
                .
                .

    <cluster label="g5kBordo" disk="g5kBordeauxDisk" login="pkchouhan"
         total_nodes="50">
      <env path="/homePath/user/demo/bin" LD_LIBRARY_PATH="/homePath/user/
           demo/lib"/>
      <node label="node-1.bordeaux.grid5000.fr">
      <ssh server="node-1.bordeaux.grid5000.fr"/>
      </node>
                .
                .
      <node label="node-47.bordeaux.grid5000.fr">
           <ssh server="node-47.bordeaux.grid5000.fr"/>
      </node>
    </cluster>
          .
          .
    <cluster label="g5kOrsay" ..>
           .
           .
    </cluster>

  </resource_description>
```

Figure 11.2: An example of resource description XML file.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE app_description SYSTEM "../application_descript.dtd">
<app_description>

    <compute label="node1" disk="disk1" login="pkchouhan">
       <wapp="0.0046"/>
       <wreq="0.0064"/>
       <wfix="0.001"/>
       <wsel="0.0009"/>
       <sres="0.0064"/>
       <srep="0.0053"/>
    </compute>
    <compute label="node2" disk="disk2" login="pkchouhan">
       <wapp="0.0044"/>
       <wreq="0.0054"/>
       <wfix="0.0009"/>
       <wsel="0.0008"/>
       <sres="0.0064"/>
       <srep="0.0053"/>
    </compute>
          .
          .
    <compute label="nodeX"  ..>
          .
          .
    </compute>
    <cluster label="g5kBordo" disk="g5kBordeauxDisk" login="pkchouhan"
         total_nodes="50">
       <wapp="0.0046"/>
       <wreq="0.0064"/>
       <wfix="0.001"/>
       <wsel="0.0009"/>
       <sres="0.0064"/>
       <srep="0.0053"/>
    </cluster>
          .
          .
    <cluster label="g5kOrsay" ..>
           .
           .
    </cluster>
  </app_description>
```

Figure 11.3: An example of application description XML file.

Server node communication model: Servers have only one parent and no children, so the time required by a server for receiving request and sending replies is associated with the size of request and/or response and bandwidth link.

Parent node's computation model: As parent node perform two activities involving computation: the processing of scheduling requests and the selection of the best server amongst the replies returned from its children. So parent computation model is based on the amount of computation need for these two activities, parent's computing power and number of its children.

Server node's computation model: Servers also perform two activities involving computation: performance prediction as part of the scheduling phase and provision of application services as part of the service phase. So server computation model depends on the amount of computation needed for these two activities and server's computing power.

As an example, the performance models for DIET can be seen in Part III. Using the performance model, the throughput of any system can be calculated with different operating models (mentioned in 7.1.1).

11.3 Working of ADePT

To generate a deployment plan for a middleware deployment, ADePT needs performance models 11.2 for middleware's components, application description (what is the execution time of the application on dedicated servers, application parameters, etc.) and resource description (computing power, bandwidth link between nodes, etc.) as input. First, throughput of each resource is calculated according to the given performance model and then exact number of resources are selected, that are needed to generate the deployment. Based on the type of resources the arrangement of resources is defined. Then ADePT generates an XML file describing the middleware deployment plan.

To improve a given deployment for a middleware, ADePT needs performance model for middleware components, and an XML file representing of the deployment plan. First, throughput of each resource is calculated and then the load of resource with minimum throughput is shared either by adding a new resource in hierarchy or by rearranging the hierarchies resources. The process of removing the bottleneck is repeated till all the servers in the hierarchy will perform at their full speed. Then the corresponding hierarchy is translated as an XML file.

Generated XML file is submitted to the deployment tool to deploy the middleware. Deployment tool deploys the hierarchy according to the deployment plan presented as XML file.

11.4 ADePT as a deployment planner for ADAGE

In Chapter 4 we have surveyed some deployment tools. Most of these tools use user defined deployment plan for deployment. Only two tools (Sekitei and Pegasus) uses AI based planner for the generation of deployment plan. The planner used by these tools are specific for the application to be deployed.

Among surveyed tools, ADAGE seems very interesting tool for the deployment of grid applications. Even though ADAGE contains a deployment model that specifies how a particular

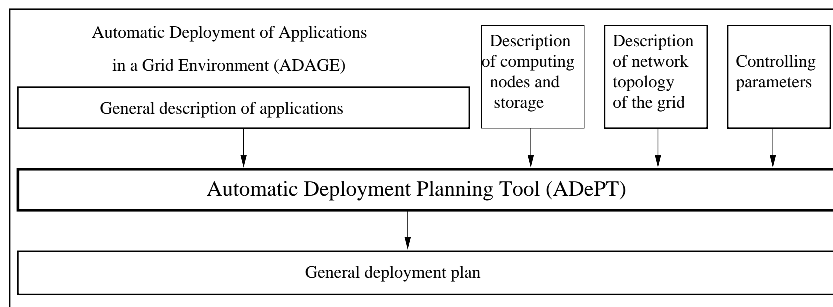


Figure 11.4: ADePT as a deployment planner for ADAGE.

component can be installed, configured and launched on a machine. However ADAGE has no intelligent deployment planning algorithm. Currently, there are two very basic planners implemented: round-robin and random. ADAGE team note a strong need of intelligent deployment planning algorithms [60]. As such, ADAGE is a framework where to plug planner. The two provided planner are just toy planner to validate the proposition.

Figure 11.4 gives an outline how ADePT can be merged in ADAGE. The base of this figure is taken from Figure 12.2 of [60]. ADAGE team thought that deployment planner is difficult to conceive. So to minimize the number of implementations of planners for each planning algorithm they introduced the concept of general description. Thus each box in Figure 11.4 represents the general layout of the information. "Generic application description", represents an application which is converted to general form of application description by a simple converter. The latter is given as input to the deployment planner, here it is ADePT. "Description of computing nodes and storage" provides the information regarding the computing power, storage capacity of grid resources. "Description of network topology of the grids" provides the description of network topology as high level of abstraction, and thus easy to exploit by the planner to place the components of the applications. As the "Deployed plan" in Figure 11.1, submit a plan to which components of the middleware can be launched, so we assume that a deployment plan that has to be improved can be submitted to ADePT through "Description of network topology" by the ADAGE with some specific tag that defines that given description is an existing deployment. Or it may be better that ADAGE add a new input format that can accept a deployed plan and can be used for improvement of the given deployment or redeployment. As "Performance model" provides a indirect control on the performance of deployment plan by controlling the throughput calculation model in ADePT, similarly we can assume that required information by ADePT can be transferred to it by "Controlling parameters" of ADAGE with some modification, if required. "General deployment plan" in ADAGE is represented by XML and ADePT also provides its deployment as an XML file.

11.5 Conclusion

In this chapter we have given our point of view to generate an automatic deployment planning tool by implementing our theoretical results.

ADePT is a complementary for the deployment tools as it generates the deployment plan automatically. ADePT saves time of end users, as it generates best deployment, which executes

more user's submitted applications in a time step. Even tool is not dependent on any particular middleware, applications, and resources thus can be used to generate a deployment plan for any middleware based on any type of application using different resources. As most of the deployment tool takes XML file as an input its advantageous that ADePT generates deployment plan in the form of XML file.

We also gave an outline that mention, how ADePT can be merged in ADAGE. The ideas are very initial but we feel that it is a strong step, which will be very helpful to merge our theoretical work to be used in other grid tools.

Part IV

Conclusion and Future work

Chapter 12

Conclusions

The work presented in this thesis concerns about the efficient utilization of the resources provided by the grid platform through NES environments. As we saw in the first introductory chapter of this thesis, grid platforms are very promising but are very challenging to use because of their intrinsic heterogeneity in terms of hardware capacities, software environment and even system administrator orientations. Thus end-users uses grid resources to execute their applications through NES environments, as these environments hides all the complexities of the grid platform from users by providing easy access to the grid resources.

We did a survey of most prevalent NES environment, so as to obtain a depth knowledge of existing NES environments. In Chapter 2, we have presented six NES environments (NeOS, NetSolve, Nimrod, Ninf, PUNCH, and WebCom) under various headings, enabling an objective comparison to be made. By objectively comparing these systems, an attempt is made to enable potential NES users' to choose an appropriate environment to best suit their needs. In addition to these NES environments we also presented some other systems which are popular grid middleware and grid systems. These systems work in same space as NES environments but are different because some of them are based on cycle steeling concept unlike NES environments, which are dedicated servers. And some other systems which are dedicated does not provide scheduling system, job management services, queueing mechanisms as done by NES environments.

After obtaining the knowledge of NES environments we analyzed the two main factors that affects the efficient utilization of the NES environments. Even if we assume that NES environment selects the best server for the execution of the user submitted requests, then also these factors effect the performance of NES environment. These two factors are:

- scheduling technique used to schedule tasks on selected server
- deployment planning of NES's components on the distributed resources.

Scheduling in client-server scenario

To meet the goal of the thesis, i.e., to use NES environments efficiently, the very first problem with which we are confronted relates to applications scheduling. It is possible to find the computing power and the capacity storage necessary for a task by using the available performance forecasting tools like NWS [85], FAST [39], etc. However even if once the server is identified,

that is best suited to solve the task, still it remains to determine a scheduling that offers the greatest possible effectiveness for the execution of tasks on the selected servers.

NES environment generally use the Minimum Completion Time (MCT) on-line scheduling algorithm where-by all applications are scheduled immediately or refused. This approach can overload interactive servers in high load conditions and does not allow adaptation of the schedule to task dependencies. Thus, we first studied the scheduling techniques that can be adopted for scheduling tasks on NES environment. As a result, we presented algorithms for the scheduling of the sequential tasks on a NES environment in Chapter 3. We mainly discussed a deadline scheduling with priority strategy that is more appropriate for multi-client, multi-server scenario. Experimentally we proved that the deadline scheduling with priority along with fallback mechanism can increase the overall number of tasks executed by the NES.

Automatic deployment planning

Once we have good scheduling algorithms to schedule the tasks on servers, the next important factor that influence the efficiency of the NES environments is the deployment planning to map the environment's components on the available resources. Generally components of these environments are mapped on the available resources as defined by the user or environment's administrator. Mapping of the components on to the available resources is called "deployment". Their exist some deployment tools, that deploy the NES's components on selected resources. In Chapter 4 we have presented a survey of some deployment tools (ADAGE, JADE, JDF, Pegasus, Sekitei, SmartFrog, and Software Dock). However, the deployment tools does not allow the non-expert users to deploy middleware according to their applications simply on grids because the input of these tool is a detailed deployment plan that is given by the users. Deployment plan should mention all the pre-requisites required to use the grid resources efficiently.

As NES is composed of different components and each component have specific function to perform. Appropriate resource from a pool of resources should be selected to deploy according to the component's functionality. As explained in the introductory chapter of this thesis, that depending on number of available resources many deployments are possible. So good deployment is one that can execute maximum number of users' submitted applications in a time step.

Thus, to remove the second obstacle in the efficient use of NES environment, it is important to generate a automatic deployment planning algorithms and heuristics. To generate a deployment planning for a NES environment, we divided the deployment planning in three parts. First, we tried to find an optimal deployment of middleware on cluster (homogeneous resources). We thought of deployment on cluster because even to find a good deployment is time consuming on homogeneous resources. We have proved this by an experiment in Chapter 1. Secondly, we tried to find a deployment on heterogeneous resources. Because grid have heterogeneous resources even if we consider a cluster of cluster, it may be possible that two clusters' resources can be different. Finally, we thought to improve the existing deployment, because it is not always evident to deploy the middleware platform from scratch according to the specifications of new submitting applications. Sometime it is useful to modify the existing hierarchy with less cost and time then to deploy a new platform according to new requests.

Optimal deployment planning on cluster

Finding a good deployment for homogeneous resources is not as simple as it may sound: one must decide “how many resource should be used in total?”, “how many should be dedicated to scheduling or computation?”, and “which hierarchical arrangement of schedulers is more effective i.e., more schedulers near the servers, near the root node, or a balanced approach?”.

Initially we have presented an heuristic in Chapter 7, for hierarchical middleware deployment for a homogeneous resource. But the heuristic deployments are implemented under limited condition, for example, a parent node can have either servers or other parent nodes as children but not both. But later we found that if we consider the middleware working phases, we can obtain an algorithm for optimal middleware deployment on homogeneous resources.

In Chapter 8, we have shown that the optimal deployment on cluster is a **C**omplete **S**panning **d**-ary (CSD) tree complete spanning d-ary tree; this result conforms with results from the scheduling literature. More importantly, we present an approach for determining the optimal degree d for the tree. A CSD tree is a tree that is both a complete d-ary tree and a spanning tree.

Automatic deployment planning on grid

After finding an optimal solution for the first step of the automatic deployment planning, we move onto second step. However, finding the best deployment among heterogeneous resources is a hard problem since it amounts to find the best broadcast tree on a general graph, which is known to be NP-complete [15].

So we presented a deployment heuristic that predicts the maximum throughput that can be achieved by the use of available nodes in Chapter 9. The main focus of heuristic is to construct an hierarchy, so as to maximize the throughput of each node, where this throughput depends on the number of children a node is connected with, in the hierarchy. The given heuristic provides a deployment that can meet the user requests demand, if user demand is at most equal to the maximum throughput.

Improvement of existing deployment

Finally, we gave a mathematical model that can analyze an existing deployment and can improve the performance of the deployment by finding and then removing the bottlenecks in Chapter 10. This is an heuristic approach for improving deployments of NES environments in heterogeneous grid environments. The heuristic is used to improve the throughput of a deployment that has been defined by other means. The approach is iterative; in each iteration, mathematical models are used to analyze the existing deployment which identify the primary bottleneck, and remove the bottleneck by adding resources in the appropriate area of the hierarchy. Using this model we can evaluate a virtual deployment before a real deployment, provide a decision builder tool (i.e., designed to compare different hierarchies or add new resources) and take into account the hierarchies’ scalability.

Validation of deployment planning

Deployment planning algorithms and heuristic presented in part III were validated by implementing them to deploy a hierarchical middleware DIET (presented in Chapter 5) on different sites of Grid’5000, a set of distributed computational resources in France. To deploy DIET for

experiments we used DIET deployment tool called GoDIET, presented in Chapter 6. Presented experiments are designed to test the ability of our deployment planning models, to correctly identify good real-world deployments. To implement our deployment planning, we have also presented performance models for DIET middleware. Since our performance model and deployment approach focus on maximizing steady-state throughput, our experiments focus on testing the maximum sustained throughput provided by different deployments. We have presented experiments testing the accuracy of our throughput performance models, and tested whether the deployment selected by our approach provides good throughput as compared to other reasonable deployments.

A tool based on our automatic deployment planning models has been outlined in Chapter 11.

Future Work

This work has led to opening up the further possibilities of improving the performance of NES environments and their easy use. Here we are presenting some of them.

In the near future one of the principal objective is the implementation of theoretical deployment planning techniques as Automatic deployment Planning Tool (ADePT).

It will be interesting to validate our theoretical concept of deployment planning by experimenting with other hierarchical middleware like WebCom [67]. We would also like to implement deployment planning for *arbitrary arrangements* of distributed resources like P2P etc. As most of the NES have their own deployment tool, it will also be interesting to use deployment tools as *plug-in* to ADePT. Because for the experiments we did DIET deployment manually according to the generated deployment plan. We manually submit the generated hierarchy.xml to GoDIET and launch GoDIET to deploy the hierarchy. Thus if we use the deployment tools as plug-in then according to the generated deployment plan for any NES environment, corresponding deployment tool can be applied.

For the deployment plan generation we always give a set of available resources as an input. But as the grid resources dynamically get connected or disconnected, so generated deployment plan may not perform as well as it was estimated, because some selected resources may not be available at the time of user's request submission. So it will be interesting if deployment planning tool can discover the resources at the time of deployment plan generation. Due to the use of *resource discovery* concept, new resources can be known dynamically and in that case it will also be interesting to develop *redeployment* approaches that can dynamically adapt the deployment to workload levels after the initial deployment. We also envision enabling interactive deployment and reconfiguration whereby users can dynamically reconfigure portions of the their deployment. Even resource discovery and redeployment will also provides the fault tolerance to the deployed hierarchy.

Another future work may include a tool for user's enjoyment. It is true that to generate a deployment plan it is advantageous to use a deployment planning tool like ADePT, but it may be enjoyable for users to use a *visual deployment tool*. Visual tool will allow users to build a graphical model of their desired deployment and export this model to a deployment tool for launch.

Appendix A

Bibliography

[1] « Simplifying System Deployment using the Dell OpenManage Deployment Toolkit », October 2004. Dell Power Solutions.

[2] D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, R. Sutherst et N. White. « The Nimrod Computational Workbench: A Case Study in Desktop Metacomputing ». Dans *The 20th Australasian Computer Science Conference* (Macquarie University, Sydney, Australia, February 1997).

[3] D. Abramson, R. Sosic, J.Giddy et B. Hall. « Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations ». The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[4] D. Abramson, R. Sosic, J.Giddy et B. Hall. « Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations ». The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[5] K. Aida, A. Takefusa, H. Ogawa, O. Tatebe, H. Nakada, H. Takagi, Y. Tanaka, S. Matsuoka, M. Sato, S. Sekiguchi et U. Nagashima. « Ninf Project ». APAN Conference 2000, Aug. 2000.

[6] G. Antoniu, L. Bouge et M. Jan. « JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service ». Dans *Kluwer Journal of Supercomputing* (2004).

[7] G. Antoniu, L. Bougé, M. Jan et S. Monnet. « Large-scale Deployment in P2P Experiments Using the JXTA Distributed Framework ». Dans *In Euro-Par 2004: Parallel Processing* (Pisa, Italy, Aug. 2004), Numéro 3149 of Lect. Notes in Comp. Science, pp. 1038–1047.

[8] G. Antoniu, M. Jan et D. A. Noblet. « Enabling the P2P JXTA Platform for High-Performance Networking Grid Infrastructures ». Dans *Proc. of the first Intl. Conf. on High Performance Computing and Communications (HPCC '05)* (Sorrento, Italy, September 2005), Numéro 3726 dans Lect. Notes in Comp. Science, Springer-Verlag, pp. 429–440.

[9] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi et S. Vadhiyar. « Users' Guide to NetSolve V1.4 ». UTK Computer Science Dept. Technical Report CS-01-467, 2001.

[10] D. C. Arnold, H. Casanova et J. Dongarra. « Innovations of the NetSolve Grid Computing System ». *Concurrency and Computation: Practice and Experience* **14**, numéro 13-15 (2002), 1457–1479.

[11] A. S. Artelys, E. D. Dolan, J. P. Goux, R. Fourer et T. S. Munson. « Kestrel: An Interface from Modeling Systems to the NEOS Server ». Rapport technique, 2003.

[12] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela et M. Protasi. *Complexity and Approximation – Combinatorial optimization problems and their approximability properties*. Springer, 1999.

[13] C. Banino, O. Beaumont, A. Legrand et Y. Robert. « Scheduling strategies for master-slave tasking on heterogeneous processor grids ». Dans *PARA'02: International Conference on Applied Parallel Computing* (2002), LNCS 2367, Springer Verlag.

[14] O. Beaumont, L. Carter, J. Ferrante, A. Legrand et Y. Robert. « Bandwidth-centric allocation of independent tasks on heterogeneous platforms ». Dans *International Parallel and Distributed Processing Symposium IPDPS'2002* (2002), IEEE Computer Society Press.

[15] O. Beaumont, A. Legrand, L. Marchal et Y. Robert. « Optimizing the steady-state throughput of Broadcasts on heterogeneous platforms ». Rapport technique 2003-34, LIP, June 2003.

[16] O. Beaumont, A. Legrand, L. Marchal et Y. Robert. « Steady-state scheduling on heterogeneous clusters: why and how? ». Dans *6th Workshop on Advances in Parallel and Distributed Computational Models* (2004).

[17] R. Bolze, E. Caron et F. Desprez. « A Monitoring and Visualization Tool and Its Application for a Network Enabled Server Platform ». Dans *Parallel and Distributed Computing Workshop of ICCSA 2006* (Glasgow, UK., 8-11 May 2006), LNCS, éditeur.

[18] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. Palma, V. Quéma et J. B. Stefani. « Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters ». Dans *24th IEEE Symposium on Reliable Distributed Systems (SRDS)* (Orlando, Florida, USA, octobre 2005).

[19] M. Brzezniak, T. Makiela et N. Meyer. « Integration of NetSolve with Globus-Based Grids ». Dans *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 449–455.

[20] R. Buyya. « Economic-based Distributed Resource Management and Scheduling for Grid Computing ». *CoRR* **cs.DC/0204048** (2002).

[21] R. Buyya, D. Abramson et J. Giddy. « Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid ». Dans *HPC Asia 2000* (Beijing, China, May 2000), pp. 283–289.

[22] Y. Caniou et E. Jeannot. « Efficient Scheduling Heuristics for GridRPC Systems ». Dans *IEEE QoS and Dynamic System workshop (QDS) of International Conference on Parallel and Distributed Systems (ICPADS)* (New-Port Beach California, USA, juillet 2004), pp. 621–630.

[23] F. Cappello, S. Djilali, G. Fedak, T. Hérault, F. Magniette, V. Néri et O. Lodygensky. « Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid ». *Future Generation Comp. Syst.* **21**, numéro 3 (2005), 417–437.

[24] E. Caron et F. Desprez. « DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid ». *International Journal of High Performance Computing Applications* (2006). To appear.

[25] E. Caron, F. Desprez, F. Petit et C. Tedeschi. « Resource Localization Using Peer-To-Peer Technology for Network Enabled Servers ». Research report 2004-55, LIP, décembre 2004.

[26] E. Caron, F. Desprez, F. Petit et V. Villain. « A Hierarchical Resource Reservation Algorithm for Network Enabled Servers ». Dans *IPDPS'03. The 17th International Parallel and Distributed Processing Symposium* (Nice - France, avril 2003).

[27] E. Caron et F. Suter. « Parallel Extension of a Dynamic Performance Forecasting Tool ». Dans *Proceedings of the International Symposium on Parallel and Distributed Computing* (Iasi, Romania, Jul 2002), pp. 80–93.

[28] H. Casanova. « SIMGRID: A toolkit for the simulation of application scheduling. ». Dans *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'01)* (May 2001), IEEE Computer Society.

[29] H. Casanova et J. Dongarra. « NetSolve: a network server for solving computational science problems ». Dans *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)* (1996), p. 40.

[30] H. Casanova et J. Dongarra. « NetSolve's Network Enabled Server: Examples and Applications ». Rapport technique UT-CS-97-392, 1997.

[31] H. Casanova et J. Dongarra. « NetSolve version 1.2: Design and Implementation ». LAPACK Working Note 140, Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, November 1998. UT-CS-98-406, Nov 1998.

[32] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt et R. V. de Geijn. « Parallel Implementation of BLAS: General Techniques for Level 3 BLAS ». Technical Report CS-TR-95-40, University of Texas, Austin, Oct. 1995.

[33] S. Dahan, J. M. Nicod et L. Philippe. « Scalability in a GRID Server Discovery Mechanism ». Dans *10th IEEE Int. Workshop on Future Trends of Distributed Computing Systems* (Suzhou, China, mai 2004), IEEE Press, pp. 46–51.

[34] S. Dandamudi et S. Ayachi. « Performance of Hierarchical Processor Scheduling in Shared-Memory Multiprocessor Systems ». *IEEE Trans. on Computers* **48**, numéro 11 (1999), 1202–1213.

[35] E. Deelman, J. Blythe, Y. Gil et C. Kesselman. « Pegasus: Planning for Execution in Grids ». GriPhyN technical report 2002-20, 2005.

[36] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. H. Su, K. Vahi et M. Livny. « Pegasus: Mapping Scientific Workflows onto the Grid ». Dans *European Across Grids Conference* (2004), pp. 11–20.

[37] E. Deelman, G. Singh, M. H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta et K. Vahi. « Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems ». *Scientific Programming Journal* (2006). To appear.

[38] B. Del-Fabbro, D. Laiymani et L. Philippe. « Data Management in Grid Applications Providers ». Dans *Procs of the 1st IEEE Int. Conf. on Distributed Frameworks for Multimedia Applications, DFMA'2005* (Besançon, France, February 2005), pp. 315–322.

[39] F. Desprez, M. Quinson et F. Suter. « Dynamic Performance Forecasting for Network Enabled Servers in an heterogeneous Environment ». Dans *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)* (June 25-28 2001), CSREA Press.

[40] E. Dolan, P. Hovland, J. More', B. Norris et B. Smith. « Remote Access to Mathematical Software ». Rapport technique ANL/MCS-P889-0601, June 2001.

[41] E. D. Dolan, R. Fourer, J. J. Moré et T. S. Munson. « Optimization on the NEOS Server ». volume 35/6.

[42] J. Dongarra, J. Du Croz, S. Hammarling et R. Hanson. « An Extended Set of Fortran Basic Linear Algebra Subroutines ». *ACMTMS* **14**, numéro 1 (1988), 1–17.

[43] S. N. Foley, B. P. Mulcahy et T. B. Quillinan. « Dynamic Administrative Coalitions with WebCom DAC ». Dans *WeB2004 The Third Workshop on e-Business* (Washington D.C., USA, December 2004).

[44] S. N. Foley, T. B. Quillinan, J. P. Morrison, D. A. Power et J. J. Kennedy. « Exploiting KeyNote in WebCom: Architecture Neutral Glue for Trust Management ». Dans *Proceedings of the Nordic Workshop on Secure IT Systems Encouraging Co-operation* (Reykjavik University, Reykjavik, Iceland, octobre 2000).

[45] I. Foster et C. Kesselman. « Globus: A Metacomputing Infrastructure Toolkit ». *The International Journal of Supercomputer Applications and High Performance Computing* **11**, numéro 2 (Summer 1997), 115–128.

[46] J. Frey. « Condor DAGMan: Handling Inter-Job Dependencies. », 2002. http://www.cs.wisc.edu/condor/dagman/.

[47] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray et P. Toft. « SmartFrog: Configuration and Automatic Ignition of Distributed Applications », May 29 2003. HP Labs, Bristol, UK.

[48] P. Goldsack et P. Toft. « SmartFrog: a Framework for Configuration ». Dans *Large Scale System Configuration Workshop* (2001), National e-Science Centre UK. http://www.hpl.hp.com/research/smartfrog/.

[49] A. Halderen, B. Overeinder et P. Sloot. « Hierarchical Resource Management in the Polder Metacomputing Initiative ». *Parallel Computing* **24** (1998), 1807–1825.

[50] R. S. Hall, D. Heimbigner et A. L. Wolf. « A Cooperative Approach to Support Software Deployment Using the Software Dock ». Dans *Proceedings of the 21st International Conference on Software Engineering* (mai 1999), ACM Press, pp. 174–183.

[51] R. Henderson et D. Tweten. « Portable Batch System: External reference specification ». Rapport technique, NASA, Ames Research Center, 1996.

[52] T. Howes, M. Smith et G. Good. *Understanding and deploying LDAP directory services*. Macmillian Technical Publishing, 1999. ISBN: 1-57870-070-1.

[53] N. H. Kapadia, J. Fortes et C. E. Brodley. « Application Performance Modelling in a Computational Grid Environment ». Proceedings of 8th IEE International Symposium on High Performance Distributed Computing (HPDCS)(1999).

[54] N. H. Kapadia et J. A. B. Fortes. « PUNCH: An architecture for Web-enabled wide-area network-computing ». *Cluster Computing* **2**, numéro 2 (1999), 153–164.

[55] T. Kichkaylo. « Planning with Arbitrary Monotonic Resource Functions », mai 09 2003.

[56] T. Kichkaylo, A. Ivan et V. Karamcheti. « Constrained component deployment in wide area networks using AI planning techniques ». Dans *International Parallel and Distributed Processing Symposium* (Apr. 2003).

[57] T. Kichkaylo, A. Ivan et V. Karamcheti. « Sekitei: An AI planner for Constrained Component Deployment in Wide-Area Networks ». Technical report 2002-851, 2004.

[58] T. Kichkaylo, A. A. Ivan et V. Karamcheti. « Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques ». Dans *17th International Parallel and Distributed Processing Symposium (IPDPS-2003)* (Los Alamitos, CA, avril 22–26 2003), IEEE Computer Society, pp. 3–3.

[59] T. Kichkaylo et V. Karamcheti. « Optimal resource aware deployment planning for component based distributed applications ». Dans *The 13th High Performance Distributed Computing* (June 2004).

[60] S. Lacour. *Contribution à l'automatisation du déploiement d' applications sur des grilles de calcul*. Phd thesis, L'Université de Rennes 1, 2005.

[61] S. Lacour, C. Pérez et T. Priol. « Deploying CORBA Components on a Computational Grid: General Principles and Early Experiments Using the Globus Toolkit ». Dans *Proceedings of the 2nd International Working Conference on Component Deployment (CD 2004)* (Edinburgh, Scotland, UK, May 2004), W. Emmerich et A. L. Wolf, éditeurs, Numéro 3083 dans Lect. Notes in Comp. Science, Springer-Verlag, pp. 35–49. Held in conjunction with the 26th International Conference on Software Engineering (ICSE 2004).

[62] D. Lee, J. Dongarra et R. S. Ramakrishna. « visPerf: Monitoring Tool for Grid Computing ». Dans *International Conference on Computational Science* (2003), pp. 233–243.

[63] C. Martin et O. Richard. « Parallel launcher for cluster of PC ». http://www-id.imag.fr/Logiciels/kadeploy/index.html.

[64] C. Martin et O. Richard. « Parallel Launcher for Cluster of PC ». Dans *Parallel Computing, Proceedings of the International Conference* (Sep. 2001).

[65] S. Matsuoka, H. Nakada, M. Sato et S. Sekiguchi. « Design Issues of Network Enabled Server Systems for the Grid », 2000. Advanced Programming Models Working Group Whitepaper, Global Grid Forum.

[66] J. P. Morrison, J. J. Kennedy et D. A. Power. « WebCom: A Volunteer-Based Metacomputer ». The Journal of Supercomputing, Volume 18(1): 47-61, January 2001.

[67] J. P. Morrison, J. J. Kennedy et D. A. Power. « WebCom: A Web-Based Distributed Computation Platform ». Proceedings of Distributed computing on the Web, Rostock, Germany, June 21 - 23, 1999.

[68] J. P. Morrison, J. J. Kennedy et D. A. Power. « WebCom: A Web-Based Distributed Computation Platform ». Proceedings of Distributed computing on the Web, Rostock, Germany, June 21 - 23, 1999.

[69] J. P. Morrison et D. A. Power. « Master Promotion & Client Redirection in the WebCom System ». Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2000), Las Vagas, Nevada, June 26 - 29, 2000.

[70] H. Nakada, M. Sato et S. Sekiguchi. « Design and Implementations of Ninf: towards a Global Computing Infrastructure ». *Future Generation Computing Systems* **15**, numéro 5-6 (1999), 649–658.

[71] H. Nakada, H. Takagi, S. Matsuoka, U. Nagashima, M. Sato et S. Sekiguchi. « Utilizing the Metaserver Architecture in the Ninf Global Computing System ». Dans *HPCN Europe 1998: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking* (London, UK, 1998), Springer-Verlag, pp. 607–616.

[72] D. A. Power, A. Patil, S. John et J. P. Morrison. « WebCom-G ». Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2003), Las Vegas, Nevada, June 23-26, 2003.

[73] M. Quinson. « Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment ». Dans *International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02), in conjunction with IPDPS'02* (April 15-19 2002).

[74] J. Santoso, G. van Albada, B. Nazief et P. Sloot. « Simulation of Hierarchical Job Management for Meta-Computing Systems ». *International Journal of Foundations of Computer Science* **12**, numéro 5 (2001), 629–643.

[75] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima et H. Takagi. « Ninf: Network based Information Library for Global World-Wide Computing Infrastructure ». Dans *High-Performance Computing and Networking (HPCN'97 Europe)*, B. Hertzberger et P. Sloot, éditeurs, volume 1225 des *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, Vienna, avril 1997.

[76] B. A. Shirazi, K. M. Kavi et A. R. Hurson, éditeurs. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[77] G. Singh, E. Deelman, G. Mehta, K. Vahi, M. H. Su, G. B. Berriman, J. Good, J. C. Jacob, D. S. Katz, A. Lazzarini, K. Blackburn et S. Koranda. « The Pegasus portal: web based grid computing ». Dans *SAC* (2005), pp. 680–686.

[78] F. Sourd et W. Nuijten. « Scheduling with Tails and Dead-lines ». *Journal of Scheduling* **4** (2001), 105–121.

[79] H. O. Soushi. « On Dynamic Resource Management Mechanism using Control Theoretic Approach for Wide-Area Grid Computing ».

[80] A. Takefusa, H. Casanova, S. Matsuoka et F. Berman. « A Study of Deadline Scheduling for Client-Server Systems on the Computational Grid ». Dans *the 10th IEEE Symposium on High Performance and Distributed Computing (HPDC'01)* (San Francisco, California., août 2001).

[81] G. Team. « DIET User's Manual ». http://graal.ens-lyon.fr/DIET.

[82] D. Thain, T. Tannenbaum et M. Livny. « Condor and the Grid ». Dans *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox et T. Hey, éditeurs. John Wiley & Sons Inc., December 2002.

[83] J. Verriet. « Scheduling UET, UCT DAGS with Release Dates and Deadlines ». Rapport technique, Utrecht University, Department of Computer Science, 1995.

[84] R. Whaley et J. Dongarra. « Automatically Tuned Linear Algebra Software (ATLAS) ». Dans *Proceedings of IEEE SC'98* (1998). Best Paper award.

[85] R. Wolski, N. Spring et J. Hayes. « The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing ». *The Journal of Future Generation Computing Systems* **15**, numéro 5-6 (1999), 757–768.

International Journal Articles

[86] P. K. Chouhan, H. Dail, E. Caron et F. Vivien. « Automatic Middleware Deployment Planning on Clusters ». *International Journal of High Performance Computing Applications* (2007). To appear.

[87] P. K. Chouhan, A. Patil, J. P. Morrison, F. Desprez et E. Caron. « Comparison and Survey of Network Enables Server Environments ». *Special Issue on Grid Technology with the Journal of Supercomputing* (2006). To Submit.

Conference Articles

[88] E. Caron, P. K. Chouhan et H. Dail. « GoDIET: A Deployment Tool for Distributed Middleware on Grid 5000 ». Dans *EXPEGRID workshop at HPDC2006* (Paris, June 2006).

[89] E. Caron, P. K. Chouhan, H. Dail et F. Vivien. « How should you Structure your Hierarchical Scheduler? ». Dans *IEEE International Conference HPDC 2006* (Paris, France, June 2006). Poster.

[90] E. Caron, P. K. Chouhan et F. Desprez. « Deadline Scheduling with Priority for Client-Server Systems on the Grid ». Dans *Grid Computing 2004. IEEE International Conference On Grid Computing. Super Computing 2004* (Pittsburgh, Pennsylvania, oct 2004), R. Buyya, éditeur. Short Paper.

[91] E. Caron, P. K. Chouhan et A. Legrand. « Automatic Deployment for Hierarchical Network Enabled Server ». Dans *The 13th Heterogeneous Computing Workshop (HCW 2004)* (Santa Fe. New Mexico, avril 2004).

Research Reports

[92] E. Caron, P. K. Chouhan et H. Dail. « Automatic Middleware Deployment Planning on Clusters ». Rapport technique 2005-26, Laboratoire de l'Informatique du Parallélisme (LIP), mai 2005. Also available as INRIA Research Report RR-5573.

[93] E. Caron, P. K. Chouhan et H. Dail. « GoDIET: A Deployment Tool for Distributed Middleware on Grid 5000 ». Rapport technique RR-5886, Institut National de Recherche en Informatique et en Automatique (INRIA), 2006. Also available as LIP Research Report 2006-17.

[94] E. Caron, P. K. Chouhan, H. Dail et F. Vivien. « Automatic Middleware Deployment Planning on Clusters ». Research report 2005-50, Laboratoire de l'Informatique du Parallélisme (LIP), octobre 2005. Revised version of LIP Research Report 2005-26.

[95] E. Caron, P.-K. Chouhan et F. Desprez. « Deadline scheduling with Priority for client-server systems ». Research report RR-5335, INRIA, octobre 2004. Also available as LIP Research Report 2004-33.

[96] E. Caron, P.-K. Chouhan et A. Legrand. « Automatic Deployment for Hierarchical Network Enabled Server ». Research report 2003-51, Laboratoire de l'Informatique du Parallélisme (LIP), novembre 2003. Also available as INRIA Research Report RR-5146.