

Winter 2002

# A Policy-Based Resource Brokering Environment for Computational Grids

Ahmed Hamdan Al-Theneyan  
*Old Dominion University*

Follow this and additional works at: [https://digitalcommons.odu.edu/computerscience\\_etds](https://digitalcommons.odu.edu/computerscience_etds)

 Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

Al-Theneyan, Ahmed H.. "A Policy-Based Resource Brokering Environment for Computational Grids" (2002). Doctor of Philosophy (PhD), dissertation, Computer Science, Old Dominion University, DOI: 10.25777/8zqc-7x67  
[https://digitalcommons.odu.edu/computerscience\\_etds/70](https://digitalcommons.odu.edu/computerscience_etds/70)

This Dissertation is brought to you for free and open access by the Computer Science at ODU Digital Commons. It has been accepted for inclusion in Computer Science Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact [digitalcommons@odu.edu](mailto:digitalcommons@odu.edu).

# **A POLICY-BASED RESOURCE BROKERING ENVIRONMENT FOR COMPUTATIONAL GRIDS**

by

Ahmed Hamdan Al-Theneyan  
M.Sc. Computer Science, August 1998, Old Dominion University  
B.Sc. Computer and Information Sciences, July 1995, King Saud University, Saudi  
Arabia

A Dissertation Submitted to the Faculty of  
Old Dominion University in Partial Fulfillment of the  
Requirement for the Degree of

**DOCTOR OF PHILOSOPHY**

**COMPUTER SCIENCE**

**OLD DOMINION UNIVERSITY**  
December 2002

Approved by:

\_\_\_\_\_  
Mohammed Zubair (Director)

\_\_\_\_\_  
Piyush Mehrotra (Director)

\_\_\_\_\_  
Hussein Abdel-Wahab (Member)

\_\_\_\_\_  
Chester Grosch (Member)

\_\_\_\_\_  
Kurt Maly (Member)

\_\_\_\_\_  
Ravi Mukkamala (Member)

## ABSTRACT

### A POLICY-BASED RESOURCE BROKERING ENVIRONMENT FOR COMPUTATIONAL GRIDS

Ahmed Hamdan Al-Theneyan

Old Dominion University, 2002

Co-Directors: Dr. Mohammed Zubair

Dr. Piyush Mehrotra

With the advances in networking infrastructure in general, and the Internet in particular, we can build grid environments that allow users to utilize a diverse set of distributed and heterogeneous resources. Since the focus of such environments is the efficient usage of the underlying resources, a critical component is the resource brokering environment that mediates the discovery, access and usage of these resources. With the consumer's constraints, provider's rules, distributed heterogeneous resources and the large number of scheduling choices, the resource brokering environment needs to decide where to place the user's jobs and when to start their execution in a way that yields the best performance for the user and the best utilization for the resource provider.

As brokering and scheduling are very complicated tasks, most current resource brokering environments are either specific to a particular grid environment or have limited features. This makes them unsuitable for large applications with heterogeneous requirements. In addition, most of these resource brokering environments lack flexibility. Policies at the resource-, application-, and system-levels cannot be specified and enforced to provide commitment to the guaranteed level of allocation that can help in attracting grid users and contribute to establishing credibility for existing grid environments.

In this thesis, we propose and prototype a flexible and extensible *Policy-based Resource Brokering Environment (PROBE)* that can be utilized by various grid systems. In designing PROBE, we follow a policy-based approach that provides PROBE with the intelligence to not only match the user's request with the right set of resources, but also to assure the guaranteed level of the allocation. PROBE looks at the task allocation as a

Service Level Agreement (SLA) that needs to be enforced between the resource provider and the resource consumer. The policy-based framework is useful in a typical grid environment where resources, most of the time, are not dedicated. In implementing PROBE, we have utilized a layered architecture and façade design patterns. These along with the well-defined API, make the framework independent of any architecture and allow for the incorporation of different types of scheduling algorithms, applications and platform adaptors as the underlying environment requires. We have utilized XML as a base for all the specification needs. This provides a flexible mechanism to specify the heterogeneous resources and user's requests along with their allocation constraints. We have developed XML-based specifications by which high-level internal structures of resources, jobs and policies can be specified. This provides interoperability in which a grid system can utilize PROBE to discover and use resources controlled by other grid systems.

We have implemented a prototype of PROBE to demonstrate its feasibility. We also describe a testbed environment and the evaluation experiments that we have conducted to demonstrate the usefulness and effectiveness of our approach.



Copyright © 2002  
by  
Ahmed Hamdan Al-Theneyan. All rights reserved.

**DECLARATION**

I declare that this thesis is my own work and contains no materials that have been submitted in any form for another degree or diploma at any university or other tertiary institution. Information that has been derived from the published and unpublished work of others has been referenced.

---

Ahmed H. Al-Theneyan

October 28, 2002

To my parents

*Hamdan and Aljawharah*

my sisters

*Asma, Amal, Amany, Sarah, Alanood and Norah*

and my brothers

*Ibraheem, Mohammed and Abdulmohsen*

## ACKNOWLEDGMENTS

First and foremost, I would like to express my thanks and gratitude to Allah, the Most Gracious, the Most Merciful whom granted me the ability and willing to start and complete this thesis. Indeed, all thanks go to Allah for giving me the motivation, determination, patience and paving my path to achieve my goals. I pray to his greatness to inspire me the right path to his content and to enable me to continuo the work started in this thesis to the benefits of my country.

I would like to express my sincere appreciation and gratitude to my advisor, Prof. Mohammed Zubair and co-advisor, Prof. Piyush Mehrotra whom I have the privilege to work with. I am very grateful for their time, efforts and understanding throughout my graduate career and during the completion of this thesis. I would also like to thank them for pushing me in the right direction when I was wasting time or missing the point.

To my committee, Profs. Hussein Abdel-Wahab, Chester Grosch, Kurt Maly and Ravi Mukkamal, thank you for your useful advice, constructive criticism and suggestions on improving the thesis. I would also like to thank Prof. Irwin Levinstein for being the chair of my committee.

Far too many people to mention individually have assisted in so many ways during my work at ODU. They all have my sincere gratitude. In particular, I would like to thank Aymen Abdelhamid, Saad Al-Sayary, Hesham Anan, Prof. Waleed Farag, Mohammed Kholief, Prof. Xiaoming Liu, Prof. Emad Mohammed, Prof. Shunichi Toida and Ashraf Wadaa. Also, I would like to thank the faculty and colleagues at the Computer Science department of Old Dominion University for their support and encouragement. Special thanks go to Phyllis Wood and Ida Brown for their kindness and helpful nature in handling office matters. I also want to acknowledge the help of Suzana Meservey and Nancy Bollinger who proofread the thesis.

My special gratitude also goes to Abdulhadi Al-Abdulhadi, Abdullah Al-Baddah, Abdulazeez Al-Bader, Tawfeeq Al-Bakri, Prof. Ahmed Al-Fahaid, Abdullah Al-Mansoori, Sultan Al-Saadi, Abdulazeez Al-Swaileem, Youssef Al-Thabiti, Khaled Al-Qadi, Youssef Al-Omran, Prof. Alaaeldin Aly, Ashraf Basha, Mohammed Battishah,

Prof. Aymen Eldeib, Ashraf Elswify, Prof. Elsayed Hemayed, Mahdi Rahoui, Prof. Salah Serghini, Alaaeldin Sleem, Prof. Ahmed Taha, Prof. Sameh Yamany and Hamad Al-Zoman. All have been good friends and supportive brothers in the United States. I am thankful to for their supports and encouragement over the years. Special thanks also go to my friends at Muslim Students Association (MSA) and Saudi Students Association (SSA) of Old Dominion University with whom I practiced my activism during my staying at ODU. They made me feel Norfolk like home for me.

Many thanks also go to other relatives and friends whose encouragement and support have remained constant in spite of the distance: Saud Al-Theneyan, Fahad Al-Theneyan, Naser Al-Theneyan, Abdulmohsen Al-Qabbany, Khaled Al-Dossary, Fahad Al-Fadhel, Abdulazeez Al-Howaidy, Prof. Mohammed Al-Jlayl, Saad Al-Maliki, Khaled Al-Mujjayesh, Ahmed Al-Nasser, Ahmed Al-Omran, Abdullah Al-Selaimi, Adeeb Al-Swaeery and Mohammed Al-Qahtani. I also thank numerous other relatives and friends whose names did not appear here, I appreciate their supports in the spirit of continuous friendship and brotherhood.

And last, but not least, no words in the existing contemporary dictionaries will be enough to use to appreciate the supportive will of my dearest parents and their prayer, love, sacrifice and endless support in all my endeavors. They have never relented in their prayers, jointly supported by my beloved sisters and brothers. To them, I owe this dissertation.

## TABLE OF CONTENTS

	Page
LIST OF TABLES .....	xv
LIST OF FIGURES .....	xvii
 Chapter	
I. INTRODUCTION .....	1
1.1 Background .....	4
1.2 Resource Brokering Environment: Functionality and Characteristics.....	6
1.2.1 Functionalities.....	6
1.2.2 Characteristics.....	8
1.3 Objective .....	10
1.4 Approach.....	11
1.5 Focus and Contribution.....	13
1.6 Organization of the thesis .....	15
II. RELATED WORK .....	16
2.1 Batch Queuing Systems .....	16
2.1.1 NQS.....	17
2.1.2 PBS .....	17
2.1.3 DQS.....	18
2.1.4 LSF.....	18
2.1.5 Load Leveler .....	18
2.2 Grid Systems.....	19
2.2.1 NetSolve.....	19
2.2.2 Ninf .....	21
2.2.3 Globus.....	22
2.2.4 Legion .....	24

Chapter	Page
2.2.5 DISCWorld .....	25
2.2.6 Sun Grid Engine.....	26
2.3 Brokering Systems.....	26
2.3.1 Condor.....	26
2.3.2 AppLeS .....	27
2.3.3 Nimrod .....	28
2.3.4 EZ-Grid.....	28
2.4 Integrated systems.....	29
2.4.1 Gateway .....	29
2.4.2 UNICORE.....	29
2.5 Other related systems.....	29
2.5.1 RCS .....	30
2.5.2 SNIPE .....	30
2.5.3 PARDIS .....	30
2.6 Arcade.....	30
2.6.1 Overview.....	31
2.6.2 Architecture.....	31
2.6.3 Application Specification.....	32
2.7 Related Technologies.....	35
2.7.1 CORBA.....	36
2.7.2 DCOM.....	36
2.7.3 RMI.....	37
2.7.4 Jini.....	37
2.7.5 Jiro.....	39
2.7.6 J2EE .....	40
2.7.7 JXTA.....	41
2.8 Conclusion .....	41
<b>III. PROBE: A POLICY-BASED RESOURCE BROKERING ENVIRONMENT</b>	
<b>FOR COMPUTATIONAL GRIDS .....</b>	<b>44</b>

Chapter	Page
3.1 Overview .....	44
3.2 Design Goals .....	44
3.3 Architecture.....	45
3.3.1 Client Interface Module .....	45
3.3.2 Resource Broker.....	47
3.3.3 Policy Enforcement Manager .....	47
3.3.4 Resource Repository .....	48
3.3.5 Resource Monitor.....	49
3.3.6 Job Repository .....	49
3.3.7 Job Monitor.....	49
3.3.8 Resource Daemon .....	50
3.4 Scenarios .....	50
3.5 Meeting Design Goals.....	52
3.5.1 Platform Independence .....	52
3.5.2 Modularity.....	52
3.5.3 Scalability .....	54
3.5.4 Site Autonomy .....	55
3.5.5 Interoperability.....	56
3.6 Functionalities.....	59
3.6.1 Resource Brokering .....	59
3.6.2 QoS Brokering .....	62
3.6.3 Monitoring .....	64
3.7 Summary .....	66
<b>IV. RESOURCE BROKER: A DETAILED ARCHITECTURAL VIEW.....</b>	<b>68</b>
4.1 Overview .....	68
4.2 Architecture.....	68
4.3 Resource Daemon: Detailed Architecture .....	71
4.4 Design Pattern .....	73
4.5 Flexible Job Language (FJL) .....	76



Chapter	Page
4.6 Job State Transition Diagram.....	77
4.7 Resource Types.....	78
4.7.1 Resource Specification Language.....	79
4.8 Issues.....	84
4.8.1 Rescheduling.....	84
4.8.2 Allocation Assurance.....	85
4.9 Summary.....	86
<b>V. POLICY-BASED FRAMEWORK FOR RESOURCE BROKERING.....</b>	<b>87</b>
5.1 Overview.....	87
5.2 Philosophy.....	87
5.3 Design Goals.....	89
5.4 Architecture.....	90
5.5 Caching.....	93
5.6 Policy Specification Language.....	94
5.6.1 Syntax.....	95
5.6.2 XML representation of PSL.....	96
5.6.3 Examples.....	97
5.7 Policy Parsing.....	98
5.8 Policy Optimization.....	99
5.9 Actions.....	100
5.10 Summary.....	102
<b>VI. IMPLEMENTATION.....</b>	<b>103</b>
6.1 Environment.....	103
6.2 Enhancing Jini for Use Across Non-Multicastable Networks.....	105
6.2.1 Global Tunneling Lookup Service (GTLS).....	107
6.2.2 Tunneling Service (TS).....	108
6.2.3 Jini Modifications.....	109
6.2.4 A scalable alternative for super grids.....	110

Chapter	Page
6.2.5 Experimental Results .....	113
6.2.6 Future Enhancements.....	115
6.3 Client Interfaces .....	116
6.3.1 Command-line Interface .....	116
6.3.2 Visual Interface.....	117
6.4 Package Design.....	120
6.4.1 Package <i>probe</i> .....	120
6.4.2 Package <i>probe.common</i> .....	120
6.4.3 Package <i>probe.core</i> .....	121
6.4.4 Package <i>probe.repository</i> .....	122
6.4.5 Package <i>probe.algorithms</i> .....	123
6.4.6 Package <i>probe.util</i> .....	127
6.4.7 Package <i>probe.resources</i> .....	128
6.4.8 Package <i>probe.jobs</i> .....	128
6.4.9 Package <i>probe.daemons</i> .....	130
6.4.10 Package <i>probe.policy</i> .....	133
6.4.11 Package <i>probe.client</i> .....	134
6.5 Summary .....	135
VII. EVALUATION AND EXPERIMENTAL RESULTS .....	136
7.1 Overview.....	136
7.2 Experimental Testbed .....	137
7.3 Test Applications .....	141
7.3.1 Single Job.....	141
7.3.2 Co-Allocation Job .....	142
7.3.3 Parametric Job.....	142
7.3.4 Pathfinder – Sample DAG application .....	143
7.4 Experiments .....	146
7.4.1 Qualitative Experiments.....	146
7.4.2 Quantitative Experiments.....	157

Chapter	Page
7.5 Conclusion .....	166
VIII. CONCLUSIONS AND FUTURE WORK .....	168
8.1 Conclusions.....	168
8.2 Future Work .....	169
8.3 PROBE Extensions .....	170
8.3.1 Predictor .....	171
8.3.2 Fault Handler .....	172
8.3.3 Event Handler .....	174
8.4 Enhancing Jini to support Scalability .....	175
8.4.1 Overview .....	175
8.4.2 Proposed Solution .....	176
8.4.3 Scenario.....	178
REFERENCES .....	180
APPENDIX A Experiment Results .....	197
A.1.Overhead of broadcasting/delivery for the <i>Collaboration</i> approach .....	197
A.2.Overhead of broadcasting/delivery for the <i>Hierarchal Tunneling</i> approach..	197
A.3.Overhead of XML Parsing.....	198
A.4.Performance of Resource Matching.....	199
A.5.Performance of SLA Monitoring. ....	199
A.6.Memory usage.....	200
A.7.Overall Overhead of Brokering. ....	203
APPENDIX B List of Acronyms and Terms.....	204
APPENDIX C Glossary .....	207
APPENDIX D Extended Bibliography.....	212
VITA.....	216

## LIST OF TABLES

Table	Page
1. COMPARISON OF THE RELATED WORKS.....	42
2. ALLOCATION VISION FOR DIFFERENT TYPES OF GRID RESOURCES.....	79
3. FURTHER SPECIFICATIONS ABOUT GLOBUS DOMAIN .....	138
4. FURTHER SPECIFICATIONS ABOUT PROBE I GRID.....	139
5. FURTHER SPECIFICATIONS ABOUT PROBE II GRID .....	140
6. VERSION NUMBERS OF THE SOFTWARE PACKAGES USED IN THE EXPERIMENTS .....	140
7. SUMMARY OF THE QUALITATIVE EXPERIMENTS.....	155
8. OVERHEAD OF BROADCASTING/DELIVERY FOR THE COLLABORATION APPROACH .....	197
9. OVERHEAD OF BROADCASTING FOR THE HIERARCHAL TUNNELING APPROACH .....	197
10. OVERHEAD OF DELIVERY FOR THE HIERARCHAL TUNNELING APPROACH .....	198
11. PARSING TIME FOR DIFFERENT XML DOCUMENTS.....	198
12. PERFORMANCE OF RESOURCE MATCHING UNDER DIFFERENT DATA RETRIEVAL APPROACHES .....	199
13. PERFORMANCE OF SLA MONITORING UNDER DIFFERENT DATA RETRIEVAL APPROACHES .....	199
14. MEMORY USAGE FOR SMALL GRID WHEN NO SLAS ARE APPLIED .....	200
15. MEMORY USAGE FOR MEDIUM GRID WHEN NO SLAS ARE APPLIED ....	200
16. MEMORY USAGE FOR LARGE GRID WHEN NO SLAS ARE APPLIED .....	201
17. MEMORY USAGE FOR SMALL GRID WITH AN AVERAGE OF 5 SLAS PER RESOURCE .....	201
18. MEMORY USAGE FOR MEDIUM GRID WITH AN AVERAGE OF 5 SLAS PER RESOURCE .....	202

Table	Page
19. MEMORY USAGE FOR LARGE GRID WITH AN AVERAGE OF 5 SLAS PER RESOURCE .....	202
20. COMPLETION TIME OF A 100000 ms JOB UNDER DIFFERENT EXECUTION ENVIRONMENTS .....	203
21. BROKERING OVERHEAD FOR DIFFERENT JOB SIZES UNDER THE PROBE/GLOBUS EXECUTION ENVIRONMENT .....	203

## LIST OF FIGURES

Figure	Page
1. A Typical Grid Environment .....	5
2. A Typical Batch Queuing System.....	17
3. The NetSolve System.....	19
4. The Ninf System .....	21
5. The Globus Resource Management Architecture .....	23
6. The Legion Resource Management Infrastructure .....	25
7. The Arcade system architecture.....	32
8. Snapshots of the visual specification in Arcade .....	35
9. Sequence of steps required to use Jini Technology .....	38
10. Architecture of the Jiro Technology .....	40
11. PROBE Architecture.....	46
12. Using Jini in PROBE .....	54
13. Different approaches in applying the layered architecture in the repository objects..	57
14. Different grid environments interoperate via PROBE.....	59
15. Brokering Scenarios.....	61
16. Brokering cycle.....	63
17. Schema to specify disseminating options .....	65
18. Overall Architecture of the <i>Resource Broker</i> .....	69
19. An overall event diagram for interaction between the different components of the <i>Resource Broker</i> .....	70
20. PROBE Resource Daemon .....	72
21. Different platform adaptors for the resource daemon.....	73
22. Partial Class Diagram that illustrates the use of the Façade Design Pattern in PROBE's brokering infrastructure.....	74
23. Flexible Job Language (FJL). .....	75
24. Example FJL script representing a sample DAG application. ....	76
25. A Job State Transition Diagram in the Resource Broker.....	77

Figure	Page
26. Class diagram of the resource types.....	78
27. A schema for specifying resources .....	82
28. An example script of a resource using the resource specification language.....	83
29. Using the Resource Parser to write and retrieve resources information to/from the Resource Repository .....	84
30. PROBE's vision of the allocation process .....	88
31. Overall Architecture of the Policy Enforcement Manager .....	91
32. Architecture of the Local Policy Enforcer .....	93
33. Schema for the Policy Scripting Language.....	97
34. Example PSL script describing a resource policy.....	97
35. Example PSL script describing a client policy .....	98
36. Action Flow .....	101
37. Example of a dynamic replaceable parameter .....	101
38. Different non-multicastable subnets connected by the Tunneling Service (TS) .....	107
39. New format of the outgoing request message.....	110
40. Hierarchal Tunneling Approach .....	111
41. Class diagram shows the implementing the Tree Algorithm.....	112
42. Overhead of the Collaboration approach .....	114
43. Overhead of the Hierarchal Tunneling approach.....	115
44. Command line interface of PROBE.....	117
45. Snapshots of the resource-related screens.....	118
46. Snapshots of the request-related screens.....	119
47. Class diagram of Repository Adaptors .....	123
48. Class diagram of Scheduling Algorithms .....	124
49. Pseudo-algorithm for the Static EA-CPM .....	125
50. Class diagram of Queuing Algorithms .....	126
51. PROBE PlugInHelper Utility.....	127
52. PROBE ResourceDaemonHelper Utility .....	127
53. Class diagram of Application Types.....	129

Figure	Page
54. Class diagram of Resource Daemons .....	131
55. Class diagram of Action Infrastructure.....	134
56. PCG Test Bed Environment.....	137
57. FJL script representing a sample single application .....	141
58. FJL script representing a Co-Allocation application .....	142
59. FJL script representing a Parametric application.....	143
60. The Pathfinder System .....	144
61. FJL script representing the Pathfinder application .....	145
62. Basic PROBE with different plug-ins.....	147
63. Steps involved in successful execution of a Single Job.....	150
64. Scenario of the waiting/rescheduling experiment.....	152
65. Scenario of the SLA monitoring experiment.....	154
66. Parsing time for different XML document .....	159
67. Performance of Resource Matching under different data retrieval approaches.....	160
68. Performance of SLA Monitoring under different data retrieval approaches .....	161
69. Memory usage for different kinds of grids where no SLAs are applied.....	162
70. Memory usage for different kinds of grids with an average of five SLAs per resource .....	163
71. Completion time of a 100 seconds job under different execution environments .....	164
72. Brokering overhead of a 100 seconds job under different execution environments.	165
73. Brokering overhead for different job sizes under the PROBE/Globus execution environment .....	166
74. Architecture of Extended PROBE .....	171
75. The Prediction Process.....	172
76. MS Outlook recurrence window .....	174
77. Implementing the <i>Load</i> class .....	176
78. Implementing the Load-balancing algorithm.....	177
79. Scenario of the load balancing process .....	178



# CHAPTER I

## INTRODUCTION

The increasing availability of inexpensive, high-speed computational resources is making it feasible for engineers and scientists to address large-size simulations and computational problems. Multidisciplinary Design Optimization (MDO) methods, for example, are being explored at NASA Langley Research Center (LaRC) for the design and optimization of aerospace vehicles [87]. Very often, these problems require heterogeneous computational resources that are distributed geographically. For example, simulating the airflow around an airplane may require a Computational Fluid Dynamics (CFD) code to be run on a supercomputer, whereas a workstation may be sufficient for the control code.

Due to current advances in networking infrastructure, specifically in the Internet, many groups, both research and commercial, are attempting to build grid environments that allow users to utilize distributed heterogeneous resources to solve their problems [28],[45],[110]. A key component in these grid environments is the resource brokering environment, since management of the shared resources is central to building an efficient grid system. In such environments, the broker's primary role is to efficiently schedule resources based on the user's requirements and the constraints placed by the resource providers. That is, given a set of application requirements and the capabilities and status of the resources under its control, the resource brokering environment acts as a matchmaker, choosing the right set of resources for the job. This may include co-allocation, in which multiple resources need to be simultaneously allocated to complete a job, and advanced reservations, wherein resources may need to be reserved for use at a future time to satisfy some real-time constraints.

A grid environment is generally dynamic in nature since the sets of resources comprising the system are quite varied and are always changing. The resource brokering

---

The journal model for this dissertation is the IEEE/ACM Transactions on Networking.

environment should be able to handle a diverse set of resources, ranging from computational resources to data resources, including data from real-time instruments. These resources may lie in different administrative domains, each with its own set of policies and rules for access and usage. The resource brokering environment needs to be flexible enough to accommodate policies for both the provider and the consumer, and the rights of both need to be respected. In addition, the resource brokering environment has to be scalable, not only from the point of view of the number of resources it is handling, but also with respect to the number of clients wishing to use its services. The resource brokering environment also should be able to handle a variety of client interfaces, ranging from interactive queries to batch applications.

Several research groups are implementing resource brokering environments for grid systems [3],[15],[18],[25],[29],[42],[59],[89],[115]. Most of these resource brokering environments are either specific to a grid system or have limited features that make them unsuitable for large applications with heterogeneous requirements. For example, resources are assumed to be dedicated and their load is assumed to be predictable; tasks<sup>1</sup> are assumed to be profiled where resource usage can be estimated in advance. Such restrictions discourage resource providers and resource consumers from using the underlying grid. In addition, the issue of fairness to users who are looking for the satisfaction of the job's requirements during the lifetime of the allocation, has not been addressed by most of these brokering efforts.

The focus of our work is to design and implement a general-purpose, modular and integrated *Policy-based Resource Brokering Environment (PROBE)* with well-defined Application Programming Interfaces (APIs) that can easily be utilized in various grid environments to develop brokering tools. PROBE has all the critical features that are necessary to support large-scale applications with varying requirements. We divide PROBE into a set of extensible and replaceable modules that define the basic services and capabilities necessary for a distributed resource brokering environment. The

---

<sup>1</sup> We use the terms request, application, job and task interchangeably to refer to the user's application, or one of its sub-modules, being created to satisfy the user's request.

flexibility and the ease of replacement of these modules make future users' requirements easier to satisfy. Moreover, scalability and high availability can be achieved by allowing modules to be replicated across distributed resources.

The main module of PROBE is a *Resource Broker* that can support a variety of underlying scheduling heuristics. The design of the *Resource Broker* is based on the façade design pattern and uses XML as the underlying specification language. Facade objects are introduced to provide single and simplified interface to more general facilities of a subsystem. This approach provides support for plug-and-play of any scheduling algorithm or application problem the user might provide. PROBE also adopts a policy-based approach for resource brokering. The *Policy Enforcement Manager* is the module that is in charge of enforcing policies and providing allocation assurance. Both the client and the resource provider can identify their policies. When requested, the *Policy Enforcement Manager* finds the appropriate resource(s) that can match the client request and then returns the set to the *Resource Broker*, which in turn creates a schedule and starts the allocation. PROBE goes far beyond the normal matching/allocation process of a typical resource brokering environment to assure the guaranteed level of allocation. It does so by introducing the concept of Service Level Agreements (SLAs) and policy enforcement. In contrast to other resource brokering environments, PROBE looks at the allocation process as an SLA between the client and the resource provider that needs to be enforced.

In implementing PROBE, we leverage off existing technologies where possible. For example, we use Java for implementing the modules allowing us to build a platform-independent system. Similarly, we use XML to describe resources, user's requests and their policies, since it provides a flexible mechanism to specify the heterogeneous resources along with their allocation constraints. Sun's Jini technology [14],[73], provides the lookup and discovery protocols necessary to keep track of a dynamic set of services. However, our experience with Jini has revealed some problems in using the technology for resource management, such as the lack of security and the inability to use across networks that do not support multicasting. To address this limitation, we enhanced Jini

with a tunneling service that propagates Jini's multicast messages across such networks [9].

Finally, we have evaluated to show that it delivers what it promises in terms of functionalities, characteristics and performance. We describe an experimental testbed that we use to carry out our experimental results. We show how we can integrate PROBE with different plug-ins such as different application types, scheduling algorithms, queuing algorithms, and platform adaptors. For example, we have integrated PROBE with Globus and Sun Grid Engine, the most popular and widely accepted systems in the grid community. We also implement some static and dynamic scheduling algorithms for Directed Acyclic Graph (DAG) applications based on the classic Critical Path Method (CPM). This provides a testbed for our experiments to evaluate PROBE with respect to its ease of use and deployment. We utilize a range of job types ranging from sample jobs to real test-case application, Pathfinder, an aircraft Multidisciplinary Design Optimization (MDO) problem [87]. We utilize these job types to conduct a number of experiments with different requirements to evaluate the performance of our framework. These experiments demonstrate the effectiveness of our technique and the applicability of PROBE as a general-purpose resource brokering environment.

In this chapter, we first give a brief overview of grid environments before focusing on a resource brokering environment, a major component of such environments that mediates the discovery, access and usage of these resources. Then, we describe a high-level approach of a general-purpose policy-based resource brokering environment in terms of its functionality and desirable characteristics.

## **1.1 Background**

A grid environment is one that combines geographically distributed resources into a virtual metacomputer in support of large-scale problems. This virtual metacomputer can be used to access powerful computational resources that are not available at one particular site, to aggregate computational resources superior to the ones offered by a single site, and to exploit the power of parallelism [3],[22],[46].

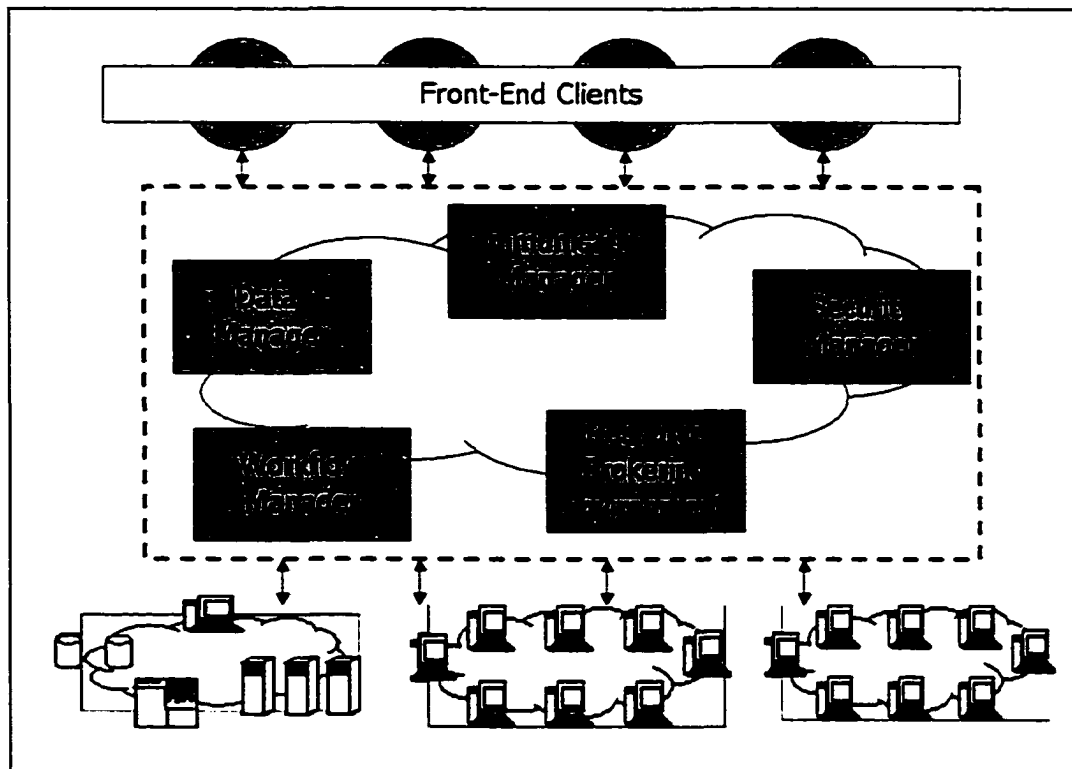


Fig. 1. A Typical Grid Environment

As shown in Fig. 1, a typical grid environment is usually comprised of a three-tier architecture. The first tier provides the user interface. The second tier, also called the middle tier, consists of a set of cooperating management modules that interface the first tier with the back-end resources. The second tier typically includes the *Communication Manager*, which acts like a mediator between the different components providing the basic communication infrastructure for the system; the *Security Manager*, which controls access to the system; the *Workflow Manager*, which manages the overall automation of the users' processes; the *Data Manager*, which handles access to shared data in the system; and the *Resource Brokering Environment*, which manages the distributed heterogeneous collection of shared resources of the system. The third tier consists of the distributed collection of shared resources that execute the users' applications. In general,

a lightweight daemon resides on each resource, providing a gateway to that resource. Most of the existing grid systems, as we explain in chapter II, employ this architecture with slight variations in the middle tier functionalities based on the scope and objectives of the system.

## **1.2 Resource Brokering Environment: Functionality and Characteristics**

For the efficient management of the shared resources, we must have a resource brokering environment that provides easy access to and utilization of the resources in a secure, scalable and robust manner. The resource brokering environment is mainly tasked with monitoring, brokering and providing an interface to the diverse, heterogeneous resources of the environment.

We focus here on the resource brokering environment component of grid environments, describing desired functionalities and characteristics. Later in this chapter, we present an overall view of architecture for a general-purpose resource brokering environment in terms of these functionalities and characteristics.

### **1.2.1 Functionalities**

The main functionalities of a resource brokering environment are monitoring, brokering, and prediction. Resource monitoring is an active area of research [61],[67]. The resource brokering environment has to keep track of the current status of the available resources. Each resource generally has some static characteristics, e.g., the speed of the CPU on a compute engine, along with some dynamic attributes, e.g., the load on a machine. The resource brokering environment should keep track of not only the static, but also the dynamic information.

Resource brokering is one of the most challenging issues in building a grid environment [117]. For the efficient use of distributed shared resources, the resource brokering environment has to support brokering in various ways:

1. *Resource allocation.* The resource brokering environment is responsible for allocating resources to various tasks of an application. This can be done in several ways: *Client-Controlled Allocation* is when the client specifies the resource to the resource brokering environment; *Broker-Controlled Allocation* is when the resource brokering environment decides for the client based on some client-specified constraints. In either case, the resources may be allocated *statically* [4], i.e., before the start of execution, or *dynamically* [5], where the allocation may change during execution due to resource failure, poor performance, optimization, etc.
2. *Co-allocation.* For some applications, the resource brokering environment needs to allocate multiple resources, ensuring that a set of resources is available for use simultaneously.
3. *Advanced reservation.* Some mission-critical applications, such as real-time applications, require resources to be available at a certain time. For these applications, advanced reservation is important, because it ensures that a resource is available for use at the required time [44]. Advance reservation is generally required to guarantee co-allocation of resource.
4. *Rescheduling.* Sometimes, due to resource failure, job failure, poor performance, load imbalance, optimization issues, etc., the resource brokering environment has to adjust the current schedule. This might include process migration, where the resource brokering environment needs to save the execution state of the process (variables, stack, and the point of execution).
5. *Job monitoring.* The resource brokering environment has to keep track of all the jobs that occupy the managed resources. Sometimes, due to poor performance, resource failure or fairness issues, a job has to be stopped, resumed, cancelled or migrated to another resource.

Also, for efficient scheduling of resources, it is more useful for the resource brokering environment to use an estimate of the performance in the near future rather than current performance. Based on some historical performance information, the resource brokering environment should be able to predict the performance each resource is going to deliver

at the time of the allocation [119],[121]. This could result in more efficient scheduling of the resources.

In a typical grid environment, the resource brokering environment works in conjunction with the *Security Manager* to authenticate and authorize all the resource requests using the credentials provided within the request. However, the resource brokering environment does not ensure the integrity of the credentials. This is assumed to be part of the *Security Manager* design. A detailed discussion of security is beyond the scope of this thesis.

### 1.2.2 Characteristics

We outline here the desirable characteristics of a resource brokering environment. We use these characteristics later as base requirements in designing PROBE.

- **Resource Heterogeneity:** Grid environments can include a variety of resources, each with different architectures, different operating systems, different configurations, different vendors and different software availability. The resource brokering environment needs to be flexible enough to accommodate all types of resources and manage them efficiently.
- **Modularity:** The resource brokering environment has to be flexible enough to handle the dynamic behavior of the resources and the unpredictable needs of the clients. Over time, new functionalities may need to be added and the existing ones modified or removed. The resource brokering environment's components should be modular so that they can be extended, modified or replaced without interfering with other parts of the system.
- **Interoperability:** The resource brokering environment should have an open, rich Application Programming Interface (API) and should use some public standards allowing grid systems to interoperate. Recently, there has been some effort to provide interoperability among existing grids. For example, the Grid Interoperability Project (GRIP) [52] is a research project that investigates the



interoperability of Globus and UNICORE. An interoperability layer has been developed to map the two grids.

- **Scalability:** The number of resources, clients and required functionality can grow without any limitations; the performance of the resource brokering environment should scale without excessive degradation. The resource brokering environment's architecture should be scalable enough to handle the dynamic behavior of the resources. A service that may prove to be a bottleneck must be replicated, a hierarchy of services can then be constructed and the load can be balanced among the replicated components using any of the available load-balancing techniques [32],[34],[108].
- **Platform independence:** The resource brokering environment should be platform-independent so that it can function on a variety of platforms (e.g., Linux, NT, or Solaris) without any modifications.
- **Fault tolerance:** In a mission-critical system such as the resource brokering environment, which requires high availability, fault-tolerance is a very critical issue [113]. A failure in one of its components should not affect the resource brokering environment in general. The resource brokering environment should also be able to keep track of all the available resources and be aware of the failures as soon as they occur. A detailed discussion of the fault tolerance issue is outside the scope of this thesis.
- **Support for site autonomy:** A grid environment consists of a distributed collection of shared resources, generally controlled by different administrative domains. Administrators in such domains want to make sure that their systems are safe, secure and available to their priority users. Each may have their own set of rules and policies. The resource brokering environment needs to be flexible enough to accommodate these policies.
- **Heterogeneous Client Interfaces:** One of the main characteristics of a resource brokering environment is to support a diverse set of client interfaces in which the client can interact with the system efficiently. Examples include interactive mode, both command-line and visual, that are easy to use and set the user free from

coding. Batch mode is another way, though it may require some programming effort. This can be done by providing an interface to an existing programming language such as Java, C, FORTRAN, etc. or by providing some user-friendly scripting mechanism for the use of the client.

### 1.3 Objective

Efficient resource brokering is one of the most important features a typical grid environment must have. For the efficient use of distributed shared resources, the resource brokering environment has to support brokering in various ways which might include allocation, co-allocation, dynamic scheduling, support of varying scheduling heuristics and job monitoring. In such an environment where resources are most often not dedicated, it is very important to assure the client that the quality of the allocation is guaranteed even after the allocation is made. Both resource providers and resource consumers want to specify their policies, and the rights of both need to be respected.

In building grid systems, and resource brokering environments in particular, different approaches can be applied. For example, a resource brokering environment could store resource information using a replicated network directory service such as the Lightweight Directory Access Protocol (LDAP) [62] or RDBMS, which enables complex queries to span and aggregate many resources. Similarly, scheduling algorithms can vary from one system to another. It is not known which approaches are best. Therefore, it is important to give the grid systems the flexibility to adopt different approaches as their environments require.

Advancements in networking infrastructure have fueled a growing interest in developing grid environments that allow users to utilize distributed heterogeneous resources to solve their problems. We have examined several systems [3],[15],[18],[25],[29],[42],[59],[89],[115], most of which are either specific to a grid system or have limited features that make them unsuitable for large applications with heterogeneous requirements. For example, some resource brokering environments, such as system-centric ones [80], allow only resources to specify their policies; others,

application-centric ones [18], allow only clients to specify their policies. Moreover, the underlying assumptions made while developing these environments make interoperability with other grid systems an issue. For example, sometimes resources are assumed to be of homogenous types, dedicated and their load predictable or tasks are assumed to be profiled where resource usage can be estimated in advance. Such restrictions discourage resource providers and resource consumers from using the underlying grid. In addition, fairness is one of the issues that has not been addressed by most of these brokering efforts where the user is looking for the assurance that its job's requirements are going to be satisfied during the lifetime of the allocation.

The main objective of this work *is to build a general-purpose policy-based resource brokering infrastructure that can easily accommodate different types of grid requirements*. With this goal in mind, we have designed and implemented PROBE, a general-purpose, modular, heterogeneous, distributed Policy-based ResOurce Brokering Environment that can be utilized by various grid environments.

In the following subsection, we give an overview of the approach that we have chosen to implement PROBE. In chapter III, we describe in detail the architecture of PROBE.

## 1.4 Approach

As mentioned earlier in this chapter, the resource brokering environment is one of the major components of a typical grid environment. The principal purposes of a resource brokering environment is to keep track of the distributed resources that comprise the execution environment and to provide information about these resources to the client upon request. Earlier, we described a desired set of functionalities and characteristics for a resource brokering environment. Based on these, we have designed and implemented a Policy-based ResOurce Brokering Environment (PROBE), as shown in Fig. 11. PROBE is a modular and fully-integrated resource brokering environment framework with well-defined APIs flexible enough to be utilized on various grid environments. As we explain

in Chapter II, no existing resource brokering environment provides all these functionalities nor has all these characteristics.

PROBE has been divided into a set of extensible and replaceable modules, where each module implements a specific function. These modules interact with each other to achieve the overall functionality of PROBE. The *Client Interface Module* provides an interface to interact with different clients, including other PROBE deployments. The *Resource Repository* maintains up-to-date information and historical performance information about all the available resources. The *Resource Broker* is the core component of PROBE that allocates resources based on client's requirements. The *Policy Enforcement Manager* works with the *Resource Broker* in finding resources and is responsible for enforcing policies. The *Resource Monitor* keeps track of the current status of the resources and updates the *Resource Repository* periodically. The *Job Monitor* monitors the execution of the jobs that occupy the managed resources while the *Job Repository* keeps information about all the currently running jobs. The PROBE infrastructure has been implemented using Jini technology that provides a plug-and-play networking environment [14]. A detailed discussion about these modules and the approaches that we have followed in implementing them will be given in chapter III.

PROBE adopts a policy-based approach for resource brokering in which both the clients and the resource providers can identify their policies. In order to provide a common understanding about allocation quality and responsibilities, PROBE uses a Service Level Agreement (SLA), which can be viewed as a contract between the resource provider and the resource consumer. PROBE goes far beyond allocating resources to provide allocation assurance by enforcing SLAs and assuring that the appropriate actions will be taken in case of violating the agreements. By committing to provide the guaranteed level of allocation, PROBE provides one means of attracting grid users and contributes to establishing credibility to existing grid environments. The policy framework is explained in great detail in chapter V.

We end this subsection by describing a typical scenario that illustrates how PROBE handles a client's request. Consider a situation in which a client sends a job consisting of sub-tasks, their dependencies and constraints through one of PROBE's client APIs. The

*Client Interface Module* on receiving the problem description creates a *Job* object and passes the request to the *Resource Broker*. The *Resource Broker* consults with the *Policy Enforcement Manager*, which then tries to find the appropriate matched resource(s) and returns the set to the *Resource Broker*. Given this set of resources, the *Resource Broker* constructs a schedule based on the underlying scheduling algorithm, the user's job and the provided sub-set of resources. As each sub-task in the job gets allocated onto the designated resources, a Service Level Agreement (SLA) is established between the client and the resource provider based on the client's terms; the *Policy Enforcement Manager* is notified to start monitoring that SLA and the *Job Monitor* is informed so that it can keep track of the job. After the successful completion of the last sub-task, the *Resource Broker* terminates the schedule.

### 1.5 Focus and Contribution

To support the idea that a general-purpose policy-based resource brokering environment can add a significant value to grid environments, we have made several novel research contributions during the work of this thesis. The main contributions are:

- Methodology and prototype implementation of a general-purpose policy-based resource brokering infrastructure that can be easily utilized by various grid systems. In building grid systems, and brokering environments in particular, different approaches can be applied. It is not known which approaches are the best. The layered approach, along with the façade design patterns and the well-defined APIs give grid systems the required flexibility to adopt different approaches.
- An interoperable brokering infrastructure that acts as a mediator in which a grid system can use to discover and use resources controlled by other grid systems. The Global Grid Forum (GGF) [47] is the main forum that is developing interoperable standards for the grid. PROBE provides a rich, open API and a set of specifications based on the public standards proposed by the Global Grid Forum and standard tools such as XML. The script specifications

of resources, jobs and their associated policies are based on XML. Using XML allows us to leverage off existing freely available XML parsers and editors to develop our tools. Also, such an XML-based specification presents the potential of inter-framework portability. With its open architecture, rich interfaces and the use of XML as the underlying specification language, PROBE can be viewed as an interoperability layer that maps existing grids.

- Policy-based resource brokering framework that goes far beyond the typical matching/allocation process to provide allocation assurance. The policy-based framework allows both the resource consumers and the resource providers to specify their policies and goes further in assuring that the level of the allocation is guaranteed even after the allocation is made. For each allocated task, PROBE creates an SLA and continues to monitor that SLA assuring that the appropriate action(s) (if any) are taken in case of violations.

Such an assurance is very useful in a typical grid environment where resources, most of the time, are not dedicated. The policy-based approach provides one means of attracting grid users and contributes to establishing credibility to existing grid environments by committing to provide the guaranteed level of allocation with the right action (compensation, credit, etc.) if such guarantees are not met. This helps in encouraging high performance users to use grid systems as they make a commitment to provide the guaranteed level of allocation.

- Enhancements to the Jini infrastructure that enable the technology to function in a scalable manner across non-multicastable networks. Jini [14],[73] is a distributed computing technology introduced by Sun Microsystems that can be used to build a flexible network of resources and services to be shared by a group of clients. However, Jini relies on multicasting in its internal protocols. This creates difficulties when the technology is deployed across networks that do not support multicasting. To address this limitation, we enhanced Jini with a tunneling service that propagates Jini multicast messages across such

networks. We also provide another alternative for super grids that relies on building a hierarchy of these services.

In summary, all the above contributions provide an available methodology and prototype implementation of a resource brokering environment that can be easily utilized by various grid environments. This thesis presents the design and development of PROBE and demonstrates the effectiveness and the applicability of PROBE as a general-purpose resource brokering environment.

## **1.6 Organization of the thesis**

The rest of this thesis is organized as follows. In Chapter II, we review several related systems, focusing on their resource brokering components. Chapter III describes the approach that we have followed in designing our resource brokering infrastructure and gives detail about how we met our design goals. In Chapter IV, we focus more on the *Resource Broker*, the heart of our brokering infrastructure. A policy-based framework for resource brokering is presented in Chapter V. Detail about the implementation of PROBE is given in Chapter VI. Chapter VII then describes the experimental testbed and the evaluation experiments that we carried out. Finally, the thesis is concluded and the future work described in Chapter VIII.

## **CHAPTER II**

### **RELATED WORK**

The problem of managing a distributed heterogeneous collection of shared resources has been an active area of research. As a result, several groups [3],[6],[15],[18],[25],[29],[42],[54],[59],[89],[115], both commercial and educational, have been working in this area. This work can be classified into four categories: batch queuing systems that are intended for local heterogeneous systems and have minimal brokering functionalities; grid systems that map well in wide area networks and offer applications a number of services including security, resource management, and communication; brokering systems that focus mainly on brokering and can be used in conjunction with other grid systems; and integrated systems that aim to provide end-to-end systems for utilizing distributed heterogeneous resources.

Generally, the resource brokering environment is a part of a larger system. In this chapter, we look at several systems and focus on issues pertinent to resource brokering. We use the desired functionalities and characteristics identified in the previous chapter as a base from which to compare and contrast the systems described in this chapter. In each system, some of the described functionalities and characteristics are either missing, partially missing, or handled by other components of the system. Exploring these systems help us understanding why PROBE is a better alternative.

#### **2.1 Batch Queuing Systems**

In a typical batch queuing system, as shown in Fig. 2, the user submits his/her job to a queuing agent, which in turn places the job onto the sufficiently un-loaded resource. Once the job has been executed, the result is returned to the user.

Batch queuing systems are intended for use with locally-distributed homogeneous environments. They don't map well in wide area distributed heterogeneous environments where heterogeneity and administrative boundaries complicate the task of the system. In



most systems, the main focus is a single resource in a single domain and possibly multiple resources in a single domain.

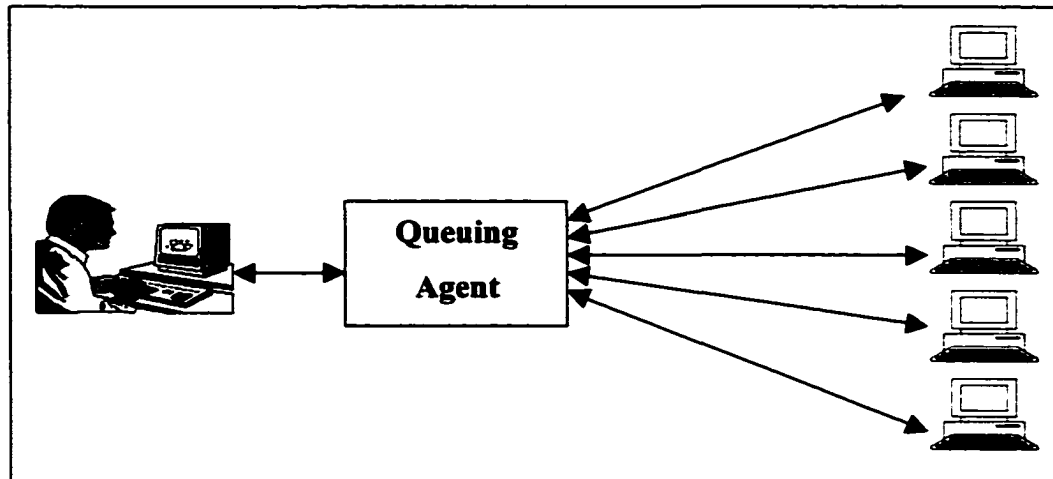


Fig. 2. A Typical Batch Queuing System

### 2.1.1 NQS

Network Queuing System (NQS) [76] is a UNIX-based batch queuing system. In this system, a request is defined as a shell script that contains the shell commands to be executed when the job runs. Standard output and error can be returned to the user. NQS has no support for parallelism. An enhanced version, the Generic NQS (GNQS) [60], an open source batch processing system for UNIX operating systems.

### 2.1.2 PBS

The Portable Batch System (PBS) [16] is a batch queuing system developed at the Numerical Aerodynamic Simulation Complex at NASA. PBS provides some features that allow the placement policy to be configured according to the site's needs and the provisioning of the allocated jobs. A batch scheduling language is also supported.

To simplify the common tasks of submitting jobs and jobs provisioning, a web-based interface, PBSWeb [86], has been developed.

### **2.1.3 DQS**

The Distributed Queuing System DQS [51] is a batch queuing system developed by the Super Computations Research Institute at Florida State University. It achieves some fault tolerance where jobs allocated on failed resources can be restarted. Like other batch queuing systems, users can submit their jobs using shell scripts. Also, parallelism is supported via the PVM system [114].

### **2.1.4 LSF**

Load Sharing Facility (LSF) [98], developed by Platform Computing Corporation, is one of the most popular commercial batch queuing systems. Unlike other batch queuing systems, LSF provides distributed load sharing and batch processing to heterogeneous resources. It also has some built-in fault tolerance where another host can be elected as the master in case of a master queuing agent failure. LSF also supports check-pointing and process migration for some platforms. LSF may be run via the command line or through a graphical user interface (GUI).

### **2.1.5 Load Leveler**

Load Leveler [68] by IBM is a batch queuing system that controls user access and balances the workload across the resources. Users who wish to submit a program for execution must create a Load Leveler script and submit it for execution. This script contains information about the job and about the nodes on which the user wants the job to run.

The Extensible Argonne Scheduling sYstem (EASY) is a scheduling system, which provides a better scheduling mechanism through which jobs can be selected to run. EASY was incorporated into Load Leveler to produce EASY-LL [109].

## 2.2 Grid Systems

### 2.2.1 NetSolve

NetSolve [25],[26] is a research project at the University of Tennessee and the Oak Ridge National Laboratory that allows users to solve complex scientific problems remotely. As demonstrated in Fig. 3, NetSolve has a three-tiered architecture in which the client sends requests to the *NetSolve Agent*, which in turn chooses the best resource according to the size and nature of the problem and other resource and network parameters. The client then directly uses the *Computational Server* on that resource to do the actual computation.

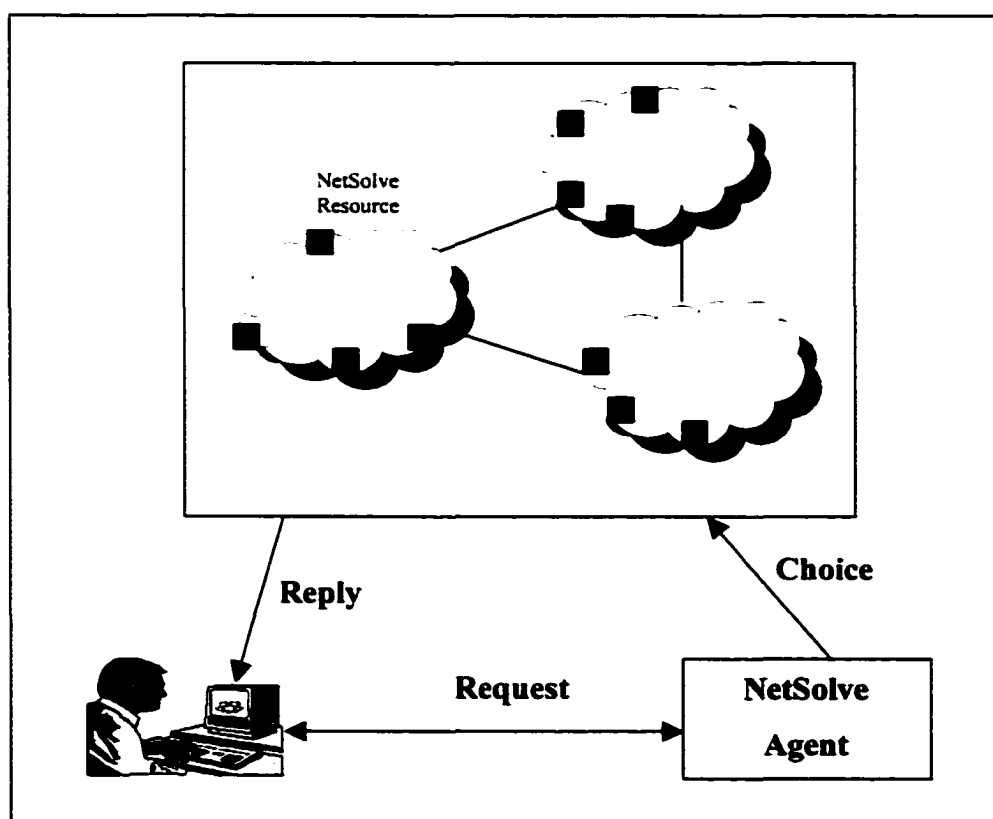


Fig. 3. The NetSolve System

One of the major components of the system is the *NetSolve Agent* that acts like a resource brokering environment managing the set of resources registered with the NetSolve system. The system can have more than one *NetSolve Agent*, each having its own view of the system. *NetSolve Agents* communicate as needed to maintain a consistent view of the system. The *NetSolve Agent* does some load balancing in order to use the available computational resources as efficiently as possible.

Every computational resource runs a *Computational Server* that has access to pre-installed libraries on that host. Clients cannot plug-in their codes, as they need to have them as NetSolve libraries registered with one of the available NetSolve *Computational Servers* [24]. When a *Computational Server* is initiated, it has the option to register only with one *NetSolve Agent* or for that *NetSolve Agent* to announce its presence.

NetSolve supports fault tolerance in which a failure of a resource can be detected at any time and subsequently reported to the *NetSolve Agent*, which keeps track of the status on all the resources. NetSolve provides the user with a diverse set of client interfaces, including an interactive mode (Matlab, shell) and a programming mode (C, FORTRAN, Java, and Matlab) that allow the user to use NetSolve efficiently [13].

NetSolve has integrated numerous systems (either in part or in whole) to help in achieving its functionality. These systems include Ninf [103], Legion [54], Globus [42], Condor [80], Internet Backplane Protocol (IBP) [63] and the Network Weather Service (NWS) [120].

One of the main problems with NetSolve is that a single NetSolve system cannot scale up to large networks. This problem becomes more of a challenge with the growth of NetSolve *Computational Servers* and their clients. Another difficulty is that NetSolve does not allow the client to export its code into the server. For a client to plug his code into NetSolve, he needs to have a library registered within one of the available computational servers. At its current stage of development, NetSolve does not have any security model. Brokering is also partially supported where all the *NetSolve Agent* does is allocate resources to tasks.

### 2.2.2 Ninf

Ninf [103],[107] (Network based Information Library for High Performance Computing) is a research project at the Electrotechnical Laboratory in Japan. This grid system allows users to access computational resources distributed across a wide area network with an easy-to-use interface. It is based on a three-tier RPC-based scheme, where libraries are installed and registered in the hosts and clients can build their applications by calling the predefined libraries with the Ninf Remote Procedure Call (RPC).

As shown in Fig. 4, the *MetaServer* is the resource brokering environment that maintains global information about all the resources available in the system. It uses the Lightweight Directory Access Protocol (LDAP) technology [62] and helps in achieving load balancing and location transparency. The *MetaServer* chooses the best resource with respect to the computational ability and the current load status. Ninf provides the client with diverse set of programming interfaces that allow the user to interact with the system efficiently. These interfaces include C, FORTRAN, Java, and Lisp.

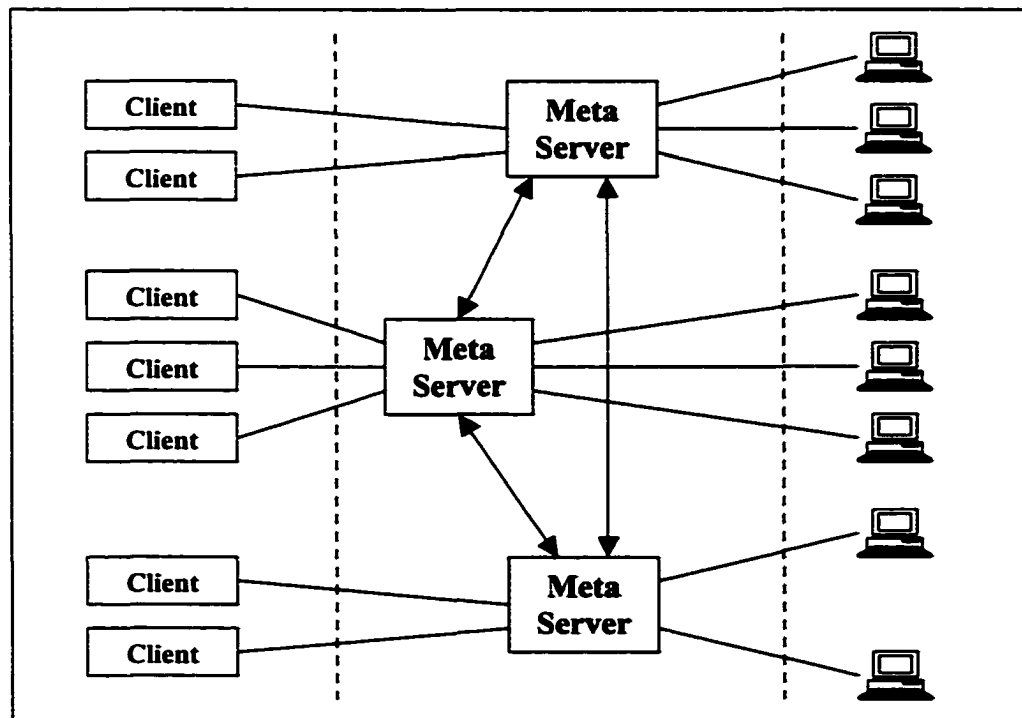


Fig. 4. The Ninf System

One of the drawbacks of Ninf is that it does not allow the client to export his/her code into the server. For a client to plug-in his code to the Ninf, he needs to have a Ninf library registered within one of the available Ninf servers. Then, anyone can use the library simply by utilizing the Ninf RPC. Another drawback to Ninf is that it has been designed for numerical applications; this results in the data types in the IDL (Interface Definition Language) being limited. Moreover, fault tolerance and security are not yet supported.

Ninf and NetSolve are very similar to each other in their design, motivation and drawbacks. Both are targeted to numerical applications. There is a rough correspondence between the *Ninf MetaServer* and *NetSolve Agent* and the *Ninf Server* and the *NetSolve Computational Server*. The development teams for both are currently collaborating to make the two systems interoperate and to standardize the basic protocols [89].

### 2.2.3 Globus

The Globus grid system [40],[42], at Argonne National Laboratory and the University of Southern California, provides the basic software infrastructure for computations that use geographically distributed computational and information resources. A central element of the Globus system is the Globus metacomputing toolkit that defines the basic services and capabilities necessary to construct a computational grid. The toolkit comprises of a set of components that implement basic services for resource management, security, communication and information infrastructure [41].

The main focus of the resource management infrastructure in Globus is to provide a uniform and scalable mechanism for naming and locating computational resources [35]. As shown in Fig. 5, Globus uses a layered architecture for resource management. The Metacomputing Directory Service (MDS) is the service that provides information about the current availability and capability of resources. It uses the data representation and an application programming interface (API) based on the Lightweight Directory Access Protocol (LDAP) [62]. Clients describe their resource requirements through a Resource Specification Language (RSL), which in turn is used to exchange information about

resource requirements between components. Resource brokers then translate RSL into more concrete resource requirements (Ground RSL). The Dynamically-Updated Request Online Co-allocator (DUROC) provides a co-allocation service where it splits request into constitutive components, submits each component to the appropriate resource manager and manipulates the resulting set of requests as a whole [43]. The Globus Resource Allocation Manager (GRAM) provides a uniform interface to a range of local management tools such as NQE [33], LSF [98], Load Leveler [68], PBS [16] and Condor [80]. Each GRAM is responsible for a particular set of local resources. It processes the RSL requests for resources, allocates the required resources, and manages and monitors the active jobs. It also periodically updates the MDS with information about the current availability and capability of resources.

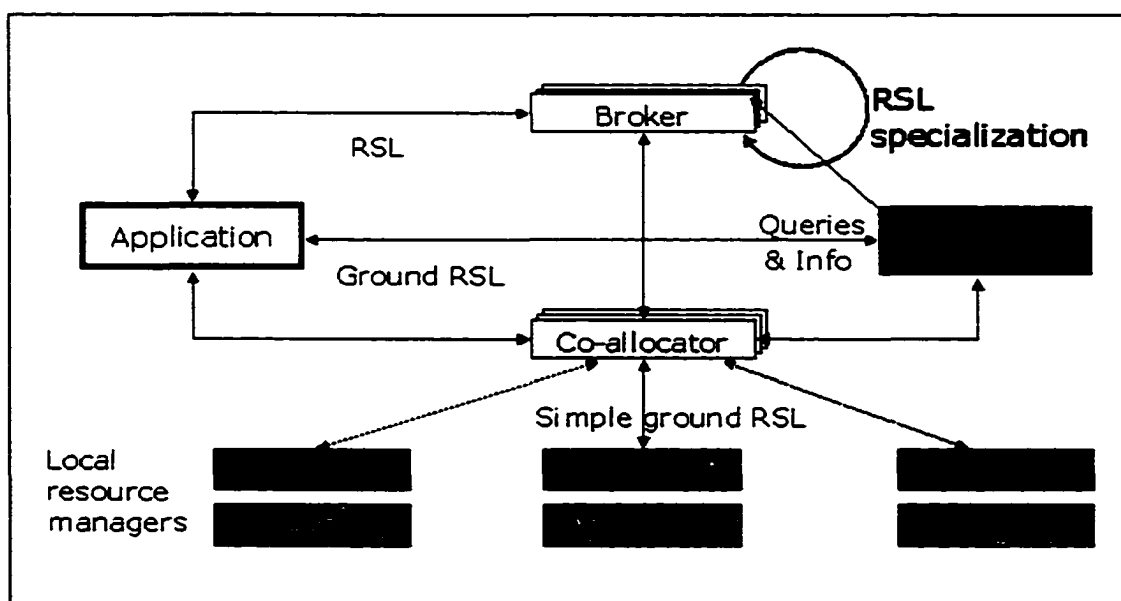


Fig. 5. The Globus Resource Management Architecture

Unlike NetSolve and Ninf, Globus allows clients to plug-in their codes and run applications written in multiple languages. The *HeartBeat Monitor* provides the ability to detect the failure of resources in the environment.

Scheduling is partially supported, as the main focus is to provide interfaces to other underlying resource brokering environments and to support site autonomy and security.

Porting to the Windows platform is still an issue, as Globus has only Windows support on the client side.

Globus has been successfully implemented and deployed on a large testbed named GUSTO comprising 15 sites, 330 computers and 3600 processors [40].

#### 2.2.4 Legion

Legion [54],[55],[56], at the University of Virginia, is an object-based metasystem that allows users to access a large collection of heterogeneous resources unified into a single coherent system. It has been built on top of Mentat [53], an object-oriented parallel processing system.

Each component of the system is an object, an active process that responds to calls from other objects. Every object is defined and managed by its class object that creates new instances, activates/deactivates the object, and provides information about the object to the client. Legion has three kinds of objects: core objects that are essential to the system (such as classes, hosts, vaults, contexts and binding agents); service objects that are useful for improving the system (such as cache objects and file objects); and user objects that allow users to provide their own classes either as executables or to enhance the system.

Each resource is represented by a Legion object. Two kinds of resources are supported: *Hosts* (computational resources) and *Vaults* (storage resources). The resource management infrastructure has three major parts: *Collection* (information database), *Scheduler* and *Enactor* (schedule implementer) [29]. As we can see in Fig. 6, *Collection* collects information about the resources. The *Scheduler* queries the *Collection* to find the desired resource, maps from object to resource and then passes the information to the *Enactor*. The *Enactor* carries out the reservation, confirms it with the *Scheduler* and places objects on the host. It then monitors object execution and notifies the *Scheduler* when rescheduling is needed.



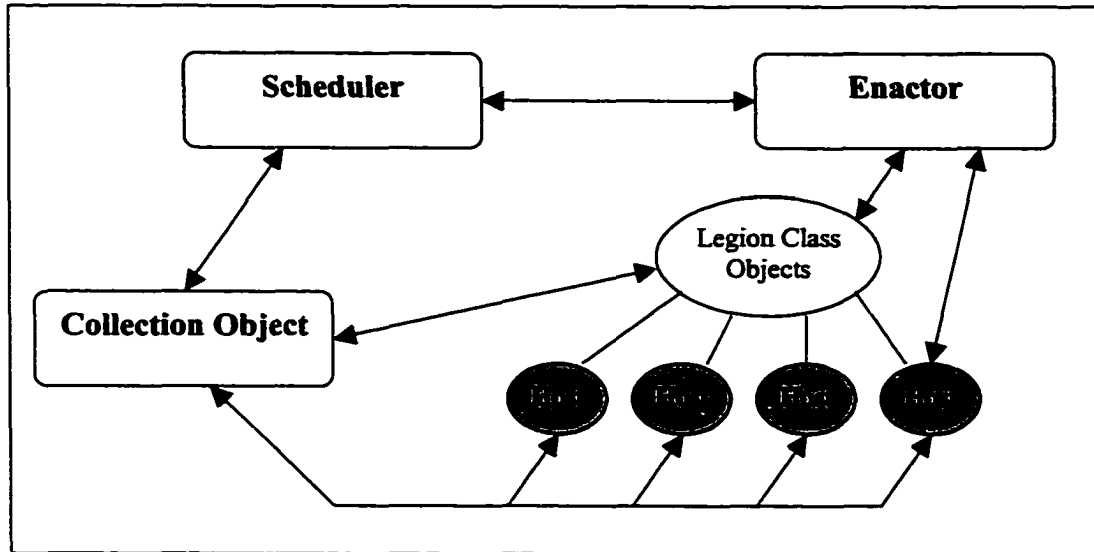


Fig. 6. The Legion Resource Management Infrastructure

Globus and Legion share common objectives and some design features. There is a rough correspondence between Globus's *DUROC* and Legion's *Scheduler*; Globus's *Information Services* and Legion's *Collection*; and Globus's *GRAM* and Legion's *Host objects*. Both allow clients to plug-in their codes and run applications written in multiple languages. The major difference is that Legion relies on an object-oriented programming model and presents a whole-cloth approach, while Globus relies on a set-of-services approach. The whole-cloth approach adds some complexity to Legion where, unlike Globus, Legion cannot be used in part and is very complicated to set up and use. In addition, portability is still an issue and scheduling is only partially supported as resources cannot be co-allocated.

### 2.2.5 DISCWorld

Distributed Information Systems Control World (DISCWorld) [59] is a service-oriented grid system being developed at the University of Adelaide. When a user submits a request to the system, it gets decomposed into services. Scheduling is supported, data and services may be moved to the host at which the least cost is found. Due to some

scheduling constraints, the services that the user can request are limited to those defined and written for the DISCWorld system; users can't submit their binaries. Moreover, all nodes have to be aware of each other in order to make intelligent scheduling decisions with respect to moving the data and the services. This might result in wastage of the bandwidth due to the huge amount of information being exchanged.

### **2.2.6 Sun Grid Engine**

Formally known as CODINE, Sun Grid Engine [115] is the new name of Sun Microsystems' distributed resource management tool for computational grids. Sun Grid Engine accepts jobs submitted by users and schedules them for execution on appropriate resources based on the specified resource management policies. Policies are determined by the particular needs of the organization. As of its current status, Sun Grid Engine does not have a security model.

Grid Engine is an open source community effort which is sponsored by Sun Microsystems and compatible with the Sun Grid Engine. Its main objective is to extend Sun's Grid Engine.

## **2.3 Brokering Systems**

### **2.3.1 Condor**

Condor [15],[80], at the University of Wisconsin-Madison, is a high-throughput computing system that runs on a cluster of workstations to harness wasted CPU cycles. The main goal of Condor is to use workstations that would otherwise be idle without disturbing other use.

It has a classified advertisement (*classad*) matchmaking framework to manage the system's variety of resources [101]. Condor entities, both provider and consumer, advertise their characteristics and their requirements in these *classads*. A specific matchmaking service (*matchmaker*) matches the *classads* and informs the matching

entities to establish contact. A ranking mechanism, based on the application constraints, is used to select the best resource when multiple resources satisfy the request.

The *classad* has been designed to match only a single resource, making the job of the resource brokering module very difficult when dealing with jobs that require multiple resources. Condor DAGMan is a module that has been introduced recently to allow users to specify dependencies between jobs so that Condor can manage them automatically. DAGMan submits jobs to Condor in an order represented by a Directed Acyclic Graph (DAG). The disadvantage of this approach is that it does not give the system an overall view of the entire DAG.

Another drawback of Condor is that it does not map well onto wide area environments, where issues such as site autonomy and heterogeneity complicate the job of the resource brokering environment. Such systems can be used within a wide area grid environment such as Globus and Legion where mediators between the systems need to be implemented. Currently, the Globus GRAM interface to Condor enables Globus users to submit jobs to Condor pools. The development teams of both systems are working together on integrating the two systems. In addition, to allow checkpointing and to perform remote system calls, code must be linked with Condor libraries [80].

### 2.3.2 AppLeS

The AppLeS system [18],[112], at the University of California in San Diego, is a system that provides tools for efficient scheduling of distributed supercomputing applications. The AppLeS approach is application-centric where everything is evaluated in terms of its impact on the application. A recent effort within the AppLeS project is the development of AppLeS templates. Built based on the expertise gained while developing AppLeS agents, these templates are stand-alone classes that can be re-used to automatically schedule applications of similar structure.

The Network Weather Service (NWS) [119],[120],[122], at the University of California in San Diego and the University of Tennessee, is a distributed resource performance forecasting Service for computational grids. Its goal is to provide accurate

forecasts of dynamically changing performance characteristics from a distributed set of resources. NWS takes periodic measurements of the resource and uses numerical models to dynamically generate forecasts of future performance levels. AppLeS uses NWS as back-end probing system to monitor the varying performance of resources used by its applications.

### **2.3.3 Nimrod**

Nimrod [3], at Griffith University in Australia, is a system for managing the execution of parameterized simulations on distributed workstations. It incorporates a distributed scheduling component that manages the scheduling of individual parametric experiments onto a set of idle resources in a local area network. Nimrod/O [79] is an extension of Nimrod that employs a number of different optimization algorithms. The work is continued in Nimrod/G that runs on top of Globus [1],[2].

### **2.3.4 EZ-Grid**

EZ-Grid [30], at the University of Houston, is a high-level job submission interface. It has been layered on top of the Globus metacomputing toolkit using its services whenever possible. EZ-Grid has a policy engine that provides authorization and cost-based accounting on top of Globus.

Currently, EZ-Grid has no concrete scheduling model. Researchers are working to define a good scheduling algorithm and to interface EZ-Grid with other systems such as IBM Load Leveler [68], PBS [16], Sun Grid Engine [115] and NWS [120]. The focus is on achieving efficient job execution in a grid environment in the presence of deadline and budget constraints.

## **2.4 Integrated systems**

Numerous projects are focusing on building a seamless and secure environment that allows resources to be accessed over the WWW so as to provide ease of access and to eliminate software distribution. Such systems rely on some of the existing grid systems as their back-end infrastructure. These include Gateway and UNICORE.

### **2.4.1 Gateway**

Gateway [10] is a system that provides seamless and secure access to remote resources through a web-based user interface. It has been layered on top of the Globus metacomputing toolkit where it can play the role of the job broker. It uses the Globus MDS to identify resources, GRAM to allocate resources, and GASS for high-performance data transfer [58].

### **2.4.2 UNICORE**

UNICOR (UNiform Interface to COmputing RESources) [102] is a system that provides seamless, intuitive and secure access to computing resources distributed across networks. As of now, UNICORE has no brokering model. The user selects a resource based on the availability at the job preparation time. The work is being continued in UNICORE Plus [6] and GRIP [111] to provide interoperability between Globus and UNICORE.

## **2.5 Other related systems**

Related systems are being developed at several other places. In this section, we briefly summarize some of efforts. Appendix D contains references to additional examples not covered in this chapter.

### **2.5.1 RCS**

RCS [12], at the Institute of Scientific Computing in Switzerland, is a single-user homogeneous system that provides an easy-to-use mechanism for using computational resources remotely. Numerical libraries are installed in the distributed hosts, which the user can access remotely.

### **2.5.2 SNIPE**

SNIPE [39], at the University of Tennessee & Oak Ridge Laboratory, is a system whose aim is to provide a reliable, secure, fault tolerant environment for distributed computing applications and data stores across the global Internet. It relies on the Resource Cataloging and Distribution System (RCDS) [85] and the Parallel Virtual Machine (PVM) [114]. SNIPE uses RCDS as a framework for replication of resource registries and globally accessible state. It uses facilities provided by PVM for message passing, task management and resource management.

### **2.5.3 PARDIS**

PARDIS [74],[75], is a system developed at Indiana University that provides support for building PARallel DIStributed applications. It employs the key idea of CORBA [93], as it has an Interface Definition Language (IDL) compiler, communication library and object repository database. PARDIS can exist as a communication subsystem in grid environments.

## **2.6 Arcade**

In this subsection, we describe Arcade, the grid system that inspired the need for this effort.

### 2.6.1 Overview

Arcade [31] is a web-based integrated grid environment that is being built to provide support for a team of discipline experts to collaboratively design, execute, and monitor multidisciplinary applications on a distributed heterogeneous network of workstations and parallel machines. This framework is suitable for applications that, in general, consist of multiple heterogeneous modules interacting with each other to solve an overall design problem, such as the multidisciplinary design optimization of an aircraft.

### 2.6.2 Architecture

As shown in Fig. 7, Arcade is based on a three-tier architecture. The first tier is a web-based, lightweight client, which provides the user interface to the whole system. It consists of applets that allow users to design an application, monitor and allocate resources, and execute, monitor and steer the application in a collaborative manner. It also has interfaces that allow the system administrator to manage the system, including resource registration and user management and authentication. Most of the logic of the system is contained in the Java-based middle tier. Among other modules, the middle tier consists of the *User Interface Manager* that provides logic to process the user input and coordinate among the other components; the *Execution Manager* that manages the overall execution of the application; the *Data Manager* that manages the shared data; the *Resource Manager* that manages the distributed heterogeneous resources of the system; and the *Security Manager* that controls access to the system. The third tier consists of the distributed resources that are used to actually execute the user modules and application codes. A lightweight *Resource Controller* executes on each resource providing a gateway to the resource.

The user generally does not need to be aware of the three-tier architecture and interacts directly with the middle tier only. For example, during the application specification phase, the user employs the visual and script applets to specify the application. During the execution phase, the middle tier (specifically the *Execution*

*Manager*) manages the overall execution, including the necessary communication and data staging.

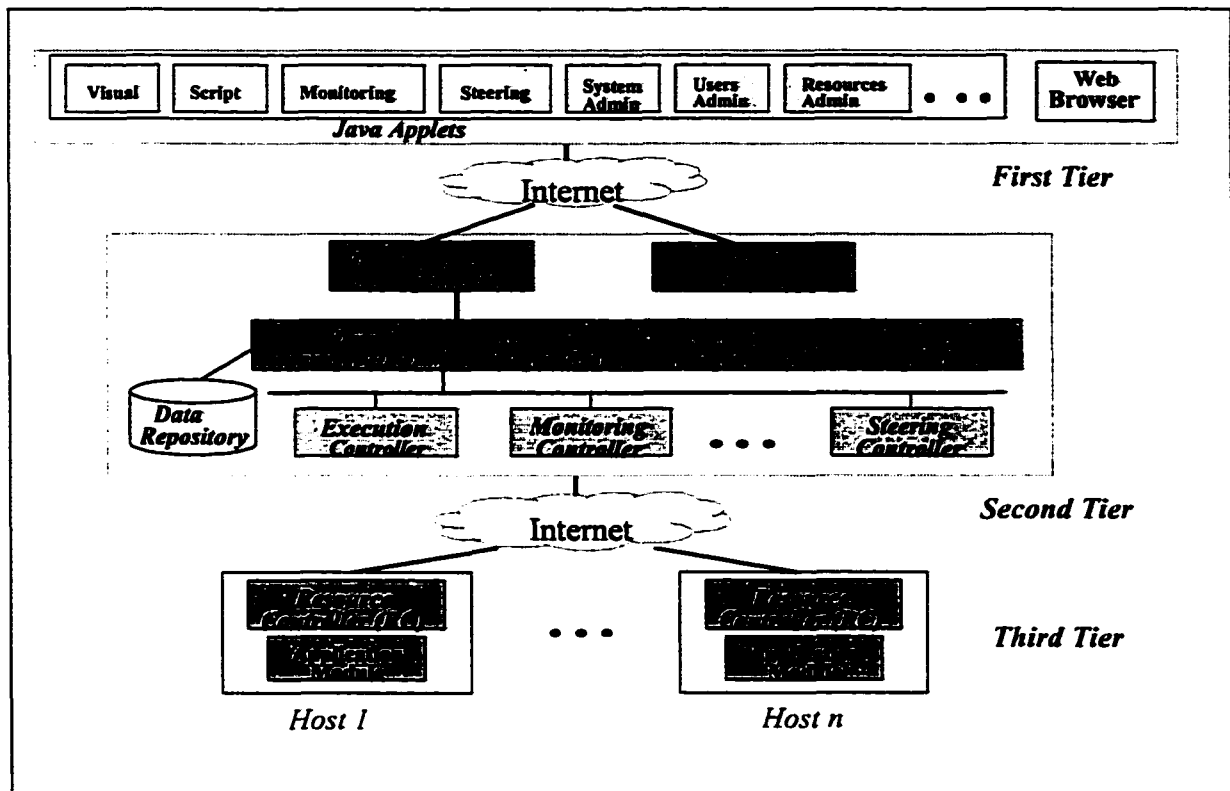


Fig. 7. The Arcade system architecture

Arcade is still in its early stages. The *Security Manager*, *Data Manager* and *Resource Manager* are not yet implemented; nevertheless, some of their functionalities are embedded in the *Communication Manager*. More information about the architecture of Arcade can be obtained from [31].

### 2.6.3 Application Specification

In the Arcade framework, a distributed application consists of a collection of heterogeneous modules (application codes from different disciplines). Arcade targets applications in which these modules are very coarse grained. A typical distributed application requires these modules to be executed in some order and possibly on different



machines. For certain problems, a set of modules may need to be executed iteratively, for example, until a desired optimization criterion is reached.

In Arcade, each application is internally represented as a Java *Project* object. This is the central object in the Arcade framework. All the information related to the application, both static and dynamic, is stored within this object. The *Project* object is a complex object that is shared by all the processes of the middle tier and supports methods that are used by these processes. When the user requests the execution of an application, the web-based client interface (the first tier) passes the corresponding *Project* object to the middle tier's *Execution Manager*, which handles the overall execution of the application.

To be able to support a wide variety of distributed applications, Arcade supports different types of modules. All these modules have a common set of properties and, hence, are derived from a general *Module* object. Some common attributes of the *Module* object are *Module Name*, *Module Directory* and *Input/Output Names*. The following types of modules derive from the general *Module*:

- ***Normal Module***: This is the basic module in the Arcade framework and is used to represent the executable parts in the applications. A *Normal Module* is identified by its executable code, command line arguments, resource requirements, and input/output file requirements.
- ***Loop Modules***: These modules allow a set of “*internal*” modules to be iteratively executed. There are two kinds of looping modules: the *For Module* for a predetermined number of iterations and the *While Module*, where the iteration condition is tested at the beginning of the loop. These modules have an associated *Project* object, which represents the set of internal modules.
- ***If Module***: This module provides a mechanism for testing the value of a condition. The truth-value of the condition determines whether the modules in the *then-block* or the optional *else-block* (each represented by a *Project* object) will be executed.
- ***Hierarchical Module***: This is an abstract *Module* representing a sub-graph, i.e., a recursively defined collection of modules.

In the current prototype, there are two ways to specify distributed applications: visually or by XML script [8]. The web/browser-based visual interface, as shown in Fig. 8, is designed to be intuitive to use. The visual interface has been designed to allow users to drag and drop modules providing the information required for each module. The dependencies between modules can be specified graphically. The system supports control dependencies using hierarchical modules to specify the bodies of loops and the *then* and *else* blocks of conditionals. Such an approach shows just the data dependencies at each level, hiding the control structure in the hierarchy. The visual representation is, thus, clean with no cluttering of control and data dependencies. However, this approach does not provide an overall view of the application in a single window, forcing users to look through multiple windows. Arcade is currently experimenting with other views. Once specified, the same visual representation of the application can also be used for visual monitoring during execution.

The script specification of Arcade is based on XML. Using XML allows Arcade to leverage off existing freely available XML parsers and editors in order to develop its tools. Also, such an XML-based script presents the potential of inter-framework portability. Thus, if a piece of the overall application needs to be executed by another framework, we could translate that portion of the XML specification into the framework specific representation.

In addition, there is a one-to-one correspondence between the visual- and script-based interfaces, allowing users to go back and forth between the two. Thus, some users will specify the application visually and then use the script representation to make changes. On the other hand, some users may be more comfortable writing the XML script using an offline editor and then using the visual representation for execution. To support this possibility, we have developed translators that translate a script-based specification to a visual-based specification, and vice-versa. These translators are integrated with Arcade tools and are transparent to users.

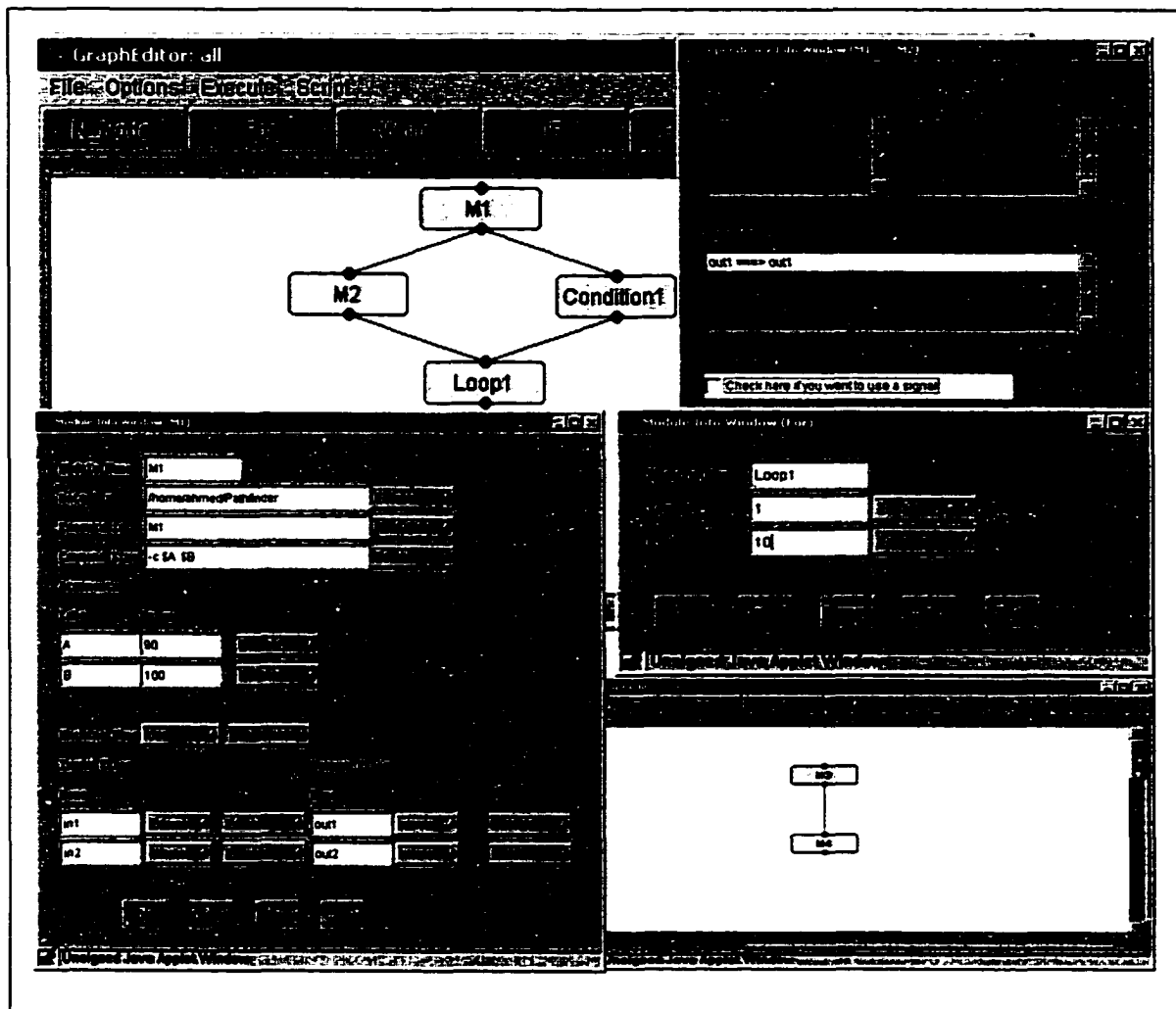


Fig. 8. Snapshots of the visual specification in Arcade

## 2.7 Related Technologies

There are a number of commercial technologies on the market that can be used to build the basic infrastructure of distributed systems in general, and resource brokering environments in particular. In this section, we provide a brief summary of some of these technologies and contrast them with one another.

### **2.7.1 CORBA**

The Common Object Request Broker Architecture (CORBA) [93] from the Object Management Group (OMG) is a standard for the development and deployment of applications in distributed, heterogeneous environments. CORBA automates many common network programming tasks such as object registration, location, and activation; framing and error-handling; and parameter marshalling and demarshalling.

CORBA relies on a protocol called the Internet Inter-ORB Protocol (IIOP), which allows object references to be passed across networks. The Object Request Broker (ORB) is the middleware that establishes the client-server relationships between objects. Each server object has an interface and exposes a set of methods. Using an ORB, a client can transparently invoke a method on a server object that can be on the same machine or across a network.

CORBA supports multiple languages and provides legacy integration capabilities that other distributed computing technologies do not address. Thus, it is more suitable where legacy support is needed.

### **2.7.2 DCOM**

Distributed Component Object Modeling (DCOM) [36], is a distributed object model developed by Microsoft that supports remote objects via a protocol called the Object Remote Procedure Call (ORPC). Unlike CORBA, a DCOM server can support multiple interfaces, each representing different behaviors of the server. A DCOM client interacts with the DCOM server by acquiring a reference to one of the DCOM server's interfaces and invoking methods through that reference. The major disadvantage of DCOM is that clients need access to the DCOM runtime, which in most circumstances is available only on Windows platforms.

### 2.7.3 RMI

Java Remote Method Invocation (RMI) [71], from Sun Microsystems, provides a simple and direct model for distributed computation with Java objects. These objects can be new Java objects or can be simple Java wrappers around other applications.

RMI provides the mechanism by which the client and the server communicate and pass information back and forth. An RMI server creates some remote objects, makes references to them accessible via an RMI Registry service, and waits for clients to invoke methods on these remote objects. An RMI client gets a remote reference to one or more remote objects in the server and then invokes methods on them.

RMI relies on a protocol called Java Remote Method Protocol (JRMP) that supports mobile code, making it possible to transport both object state and object implementation across networks. Therefore, the client does not need to have previous knowledge of the service and does not need to use a complex API to figure out how to use new services. CORBA and DCOM do not support such a feature; instead, they allow object references to be passed across networks, while the implementation and execution of those objects remain in the server. Built on top of Java, RMI brings the power of Java safety and portability to distributed computing.

RMI over IIOP [116] is a standard which has been recently introduced by Sun Microsystems and International Business Machine (IBM). It allows Java clients to access CORBA objects as if they were RMI Java objects.

### 2.7.4 Jini

Jini [14],[73] is a connection technology introduced by Sun Microsystems that can be used to build a flexible network of resources and services to be shared by a group of clients. It is based on the idea of federating groups of clients and the resources required by those clients. Built on top of Java and RMI, Jini provides simple mechanisms for resources to join together in a federation with no human intervention and then provide their services to the clients on the network. Jini provides the necessary protocols for services to register themselves with lookup services and for clients to then discover these

services. Additional features make the system resilient to failures such as removal of resources, network outages, etc. The whole technology can be segmented into three categories: infrastructure, programming model and services.

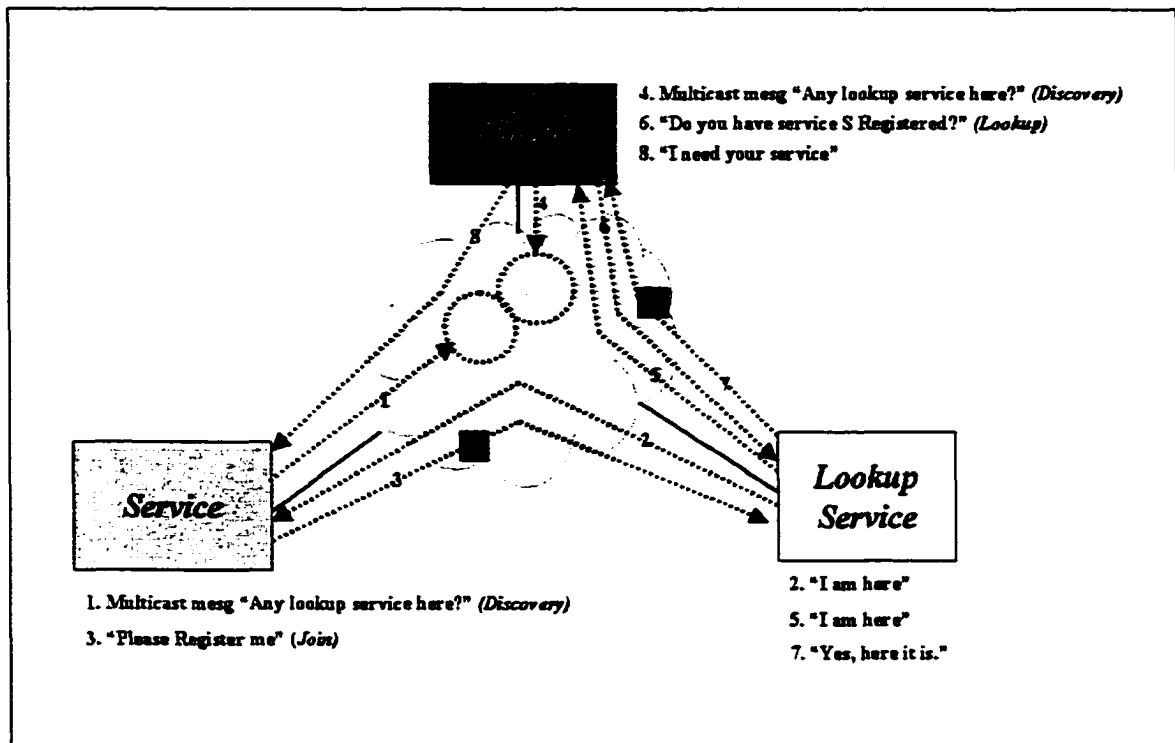


Fig. 9. Sequence of steps required to use Jini Technology

The infrastructure includes lookup services that serve as a repository of services and uses RMI, which defines the mechanism of communication between the members. The programming model includes interfaces such as discovery, lookup, leasing, remote events and transactions which ease the task of building distributed systems [14]. A service is a central concept within Jini. It is essentially an entity that can be used by a person, program or another service to perform a required task. The runtime infrastructure supports the *discovery* and *join* protocol that enables services to discover and register with lookup services. *Discovery* is the process by which a service locates lookup services on the network and obtains references to them. *Join* is the process by which a resource registers the services it offers with lookup services. In particular, the resource may post,

with the lookup service, objects representing the services they provide, including any code required to use the services. On the other hand, clients use the same protocol to locate and contact services. The discovery protocol is used to locate lookup services. Once an appropriate lookup service has been found, the client can query it to find the reference to the service that it requires. A client may then download the posted object and utilize it to directly use the service. Fig. 9 shows a simplified version of the sequence of steps that take place for a service to discover and join a lookup service and for a client to use the lookup service to locate and interact with the service that it is seeking.

### 2.7.5 Jiro

The Jiro technology [72] is a pure Java technology-based implementation of the Federated Management Architecture (FMA) specification that provides developers with the infrastructure required to build distributed resource management solutions. As shown in Fig. 10, the Jiro technology leverages the functionality of both RMI and Jini. It leverages both the remote communication protocol and the distributed garbage collection from RMI. It also relies heavily on Jini where it leverages the dynamic extensible network behavior, the lookup service, the lookup *discovery/join* protocol and some of the Jini programming model.

By providing the infrastructure, Jiro allows the developer to focus more on the features. Jiro provides a set of Jini services that provide functionality common to many management solutions. These services, referred to as Base Management Services, include: *Lookup Service* that provides a mechanism allowing all Jiro services available in the management domain to be registered and located; *Transaction Service* that provides a light-weight two-phase commit service; *Event Service* that provides an event delivery mechanism allowing publishers to post events and subscribers to receive them; *Logging Service* that supports sophisticated log messages that can be used to log any information that requires reliable persistent record; *Scheduling Service* that enables the automation of task execution based on a performance schedule; and *Security Service* that extends the

Java Authorization and Authentication Service (JAAS) to support remotely supplied login modules.

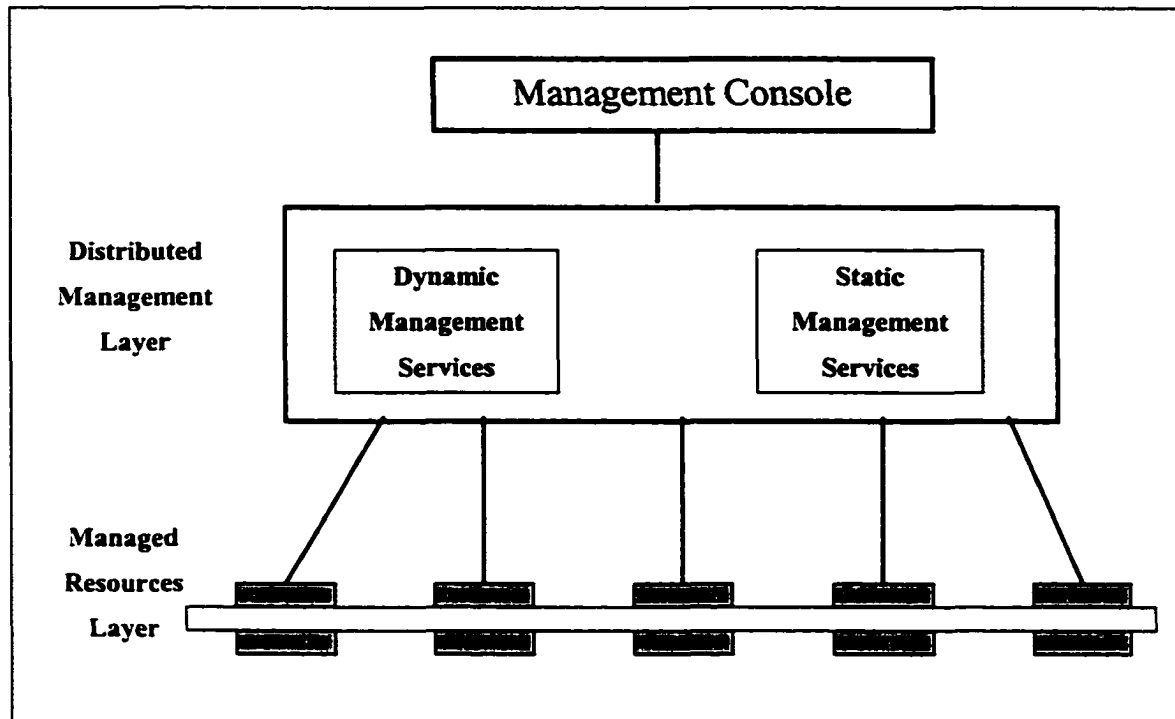


Fig. 10. Architecture of the Jiro Technology

### 2.7.6 J2EE

Java 2 Enterprise Edition (J2EE) [69] is a specification by Sun Microsystems that defines the standards to build multi-tiered distributed enterprise applications. Enterprise JavaBeans (EJB) technology is the basis of J2EE that provides the infrastructure for handling the business logic in a distributed computing environment.

EJB relies heavily on the Java Naming and Directory Interface (JNDI) for clients to lookup and locate distributed services. Unlike RMI, JNDI does not allow the injection of client-side proxy into the client virtual machine. Moreover, JNDI does not provide an effective approach for keeping track of distributed services on a dynamic basis.



Recently, there have been some efforts to integrate Jini and EJB. This work focuses in providing a Jini-EJB bridge where instead of JNDI the Jini Lookup Service is used to locate the Enterprise Java Beans.

### **2.7.7 JXTA**

Peer-to-peer (P2P) computing [94] is an evolving distributed methodology where each participant can be both a client and service. JXTA [99] is an open research project by Sun that provides a P2P-based infrastructure for distributed computing applications. The beauty of the JXTA specification is that it is independent of the transport protocol as implementation can be done over TCP/IP, HTTP, etc. Nevertheless, security and efficient message passing is still a big concern with this technology. Recently, there has been an interest in building P2P-based grid environments.

## **2.8 Conclusion**

This chapter presents some of the related work. Of course, this is not a comprehensive list of all the research that has been done in this area, but we believe that it covers the major efforts.

Most of these resource brokering environments presented in this chapter are either specific to a grid system or have limited features that make them unsuitable for large applications with heterogeneous requirements. For example, resources are assumed to be dedicated, of homogenous type, and their load is assumed to be predictable; tasks are assumed to be profiled where resource usage can be estimated in advance; and so on. Such restrictions discourage resource providers and resource consumers from using the underlying grid. In addition, the issue of interoperability has not been addressed by most current resource brokering environments. Recently, there have been some efforts in addressing this issue. For example, Grid Interoperability Project (GRIP) [52] is a research project that investigates on the interoperability of Globus and UNICORE. An interoperability layer has been developed to map the two grids. Similarly, the

development teams in both NetSolve and Ninf are collaborating to make the two systems interoperate and to standardize their basic protocols [89].

TABLE 1  
COMPARISON OF THE RELATED WORKS

Category	System Name	Resource Type	Environment	Client Policy	Resource Policy	Features
<b>Batch Queuing Systems</b>	NQS	Homogeneous	LAN	No	No	Very simple, No parallelism
	PBS	Homogeneous	LAN	Yes	No	Popular, Web-based interface
	DQS	Homogeneous	LAN	No	No	Fault tolerance, Parallelism
	LSF	Heterogeneous	LAN	No	No	Fault tolerance, Load balancing
	Load Leveler	Homogeneous	LAN	Yes	No	Load balancing
<b>Grid Systems</b>	NetSolve	Homogeneous	LAN	No	No	Load Balancing, Fault tolerance, Not scalable, Minimal brokering
	Ninf	Homogeneous	LAN	No	No	Limited IDL
	Globus	Heterogeneous	WAN	Yes	No	Commonly used, No brokering
	Legion	Heterogeneous	WAN	No	No	Whole-cloth design, very complicated
	DISCWorld	Heterogeneous	WAN	Yes	No	Restricted brokering
	Sun Grid Engine	Homogeneous	WAN	No	Yes*	*System policy, no brokering
<b>Brokering Systems</b>	Condor	Heterogeneous	LAN	Yes	Yes	System-centric

Table 1, concluded

	AppLeS	Heterogeneous	LAN	Yes	No	Application-centric
	Nimrod	Heterogeneous	LAN	Yes	No	Focuses on parametric applications
	EZ-Grid	Heterogeneous	WAN	Yes	No	No brokering model yet
<b>Integrated Systems</b>	UNICORE	Heterogeneous	WAN	Yes	No	Layered on top of Globus, no brokering,
	Gateway	Heterogeneous	WAN	Yes	No	Layered on top of Globus, no brokering,

Table 1 shows a comparison of the efforts that have been discussed throughout this chapter. The type of resources the systems support can be either *Homogeneous* or *Heterogeneous*. Environments where the system maps very well can be either a tightly coupled Local Area Network (LAN) or a loosely coupled Wide Area Network (WAN). *Client Policy* and *Resource Policy* columns specify whether or not the systems allow such policies to be specified. The *features* column notes any feature that has not been covered by other columns. In the following chapter, we present the architecture of PROBE, a general-purpose policy-based brokering infrastructure, which can handle these deficiencies.

We ended this chapter by critically reviewing some of the existing distributed computing technologies [14],[36],[69],[71],[72],[93],[94] that can be used to build the infrastructure of PROBE. The attractive features that Jini has and the degree of modularity it provides, make it the most appropriate candidate for building the infrastructure of PROBE. As Jini is layered on top of Java RMI, it can support mobile code, making it possible to transport not only object state but also object implementation across networks. This feature can help us in applying the plug-and-play feature that PROBE supports in an effective manner.

## CHAPTER III

### PROBE: A POLICY-BASED RESOURCE BROKERING ENVIRONMENT FOR COMPUTATIONAL GRIDS

#### 3.1 Overview

The work described in this thesis is motivated by the lack of a general-purpose distributed heterogeneous resource brokering middleware facility in support of grid environments. The main objective is to design and implement a prototype of a policy-based resource brokering infrastructure in support of grid systems. In chapter I, we discussed the high-level approach of PROBE a general-purpose Policy-based distributed ResOurce Brokering Environment for computational grid that can be easily utilized by various grid systems.

In this chapter, we discuss the design and development of PROBE in greater detail. In section 3.2, we present our key design goals. Section 3.3 presents the architecture of PROBE and describes the various modules in the system, while section 3.4 describes some typical scenarios that illustrate the interactions among these modules. We discuss in detail how we met our design goals in section 3.5. Finally, section 3.6 focuses on the main functionalities the system provides.

#### 3.2 Design Goals

The design of PROBE is driven by the following goals:

- *Platform Independence*: PROBE must function on many platforms.
- *Modularity*: the design of PROBE has to be flexible enough to handle the dynamic behavior of the managed resources and the unpredictable future needs of the clients.

- *Scalability*: as the number of resources and clients continues to grow, PROBE should maintain service without fundamental change in the application's architecture or major degradation of the performance.
- *Site Autonomy*: PROBE is targeted to distributed heterogeneous systems in which resources, most likely, are distributed across different administrative domains. PROBE should give administrators the flexibility to specify their usage policies and the right of both resource providers and resource consumers should be respected.
- *Interoperability*: because of the diverse grid implementation, PROBE should support interoperability allowing existing grid system to discover, access and utilize resources controlled by other grid systems.

In section 3.5, we discuss, in more detail, these design goals and how we achieve them.

### 3.3 Architecture

PROBE employs a layered three-tier architecture. The work within PROBE has been divided into a set of flexible and extensible modules, each implementing an individual function. These modules are loosely coupled and have been implemented using the Jini infrastructure. The PROBE architecture along with the interactions between the different modules is illustrated in Fig. 11. The main components in the architecture include *Client Interface Module*, *Resource Broker*, *Policy Enforcement Manager*, *Resource Repository*, *Resource Monitor*, *Job Repository* and *Job Monitor*. In the following sections, we provide description of these modules.

#### 3.3.1 Client Interface Module

The *Client Interface Module* provides an interface to handle all the client interactions with the brokering system. The client can be a user application, some other component of

the grid system, e.g., *Workflow Manager*, or another grid system. It also provides an interface to other instances of PROBE and helps in achieving consistency across different PROBEs managing the resources in the system.

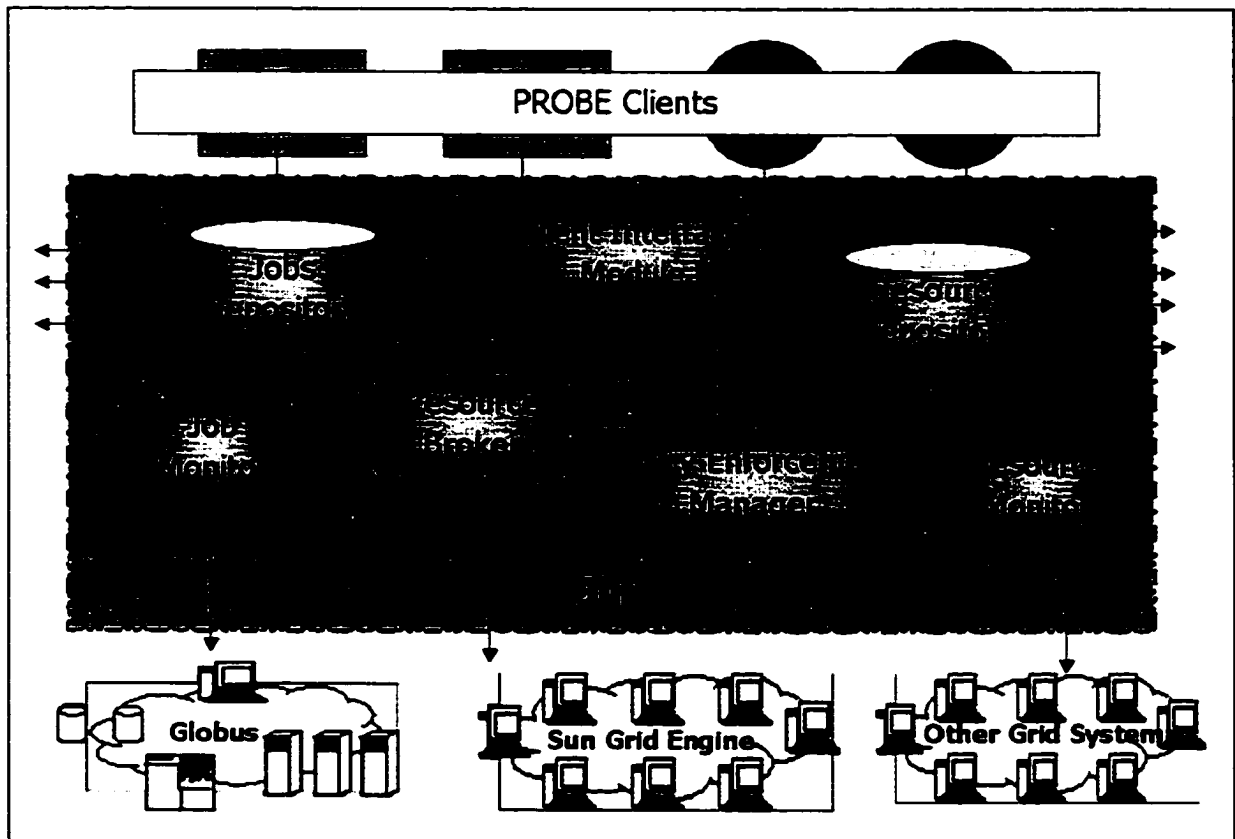


Fig. 11. PROBE Architecture

As we explain later in 4.5, clients express their requests using XML. The *Client Interface Module* parses XML requests, checks their validity and creates the corresponding job/resource objects that can be manipulated by the different components of the system. The *Client Interface Module* could be installed in the client's machine or in distributed places accessible to the clients.

### 3.3.2 Resource Broker

This is the core component of PROBE that allocates resources based on the client requirements. The *Resource Broker* gets the problem description along with the resource requirements from the *Client Interface Module*, consults with the *Policy Enforcement Manager* to find the matched resources, creates a schedule based on the underlying scheduling algorithm, and then allocates the required resources.

The *Resource Broker* maintains an internal queue of jobs currently in the system including those that have not been scheduled yet and those that failed and need to be rescheduled. A queuing algorithm selects the next job to schedule.

The design of the *Resource Broker* follows a layered façade design pattern and uses XML as the underlying specification language. This makes the *Resource Broker* flexible and generic enough not only to handle the different kinds of user applications but also to handle the different kinds of scheduling techniques that can be utilized. This approach makes algorithms and application types look like black boxes allowing the users to plug in their scheduling and queuing algorithms as needed. Resource brokering is briefly discussed in section 3.6.1. More detail about the *Resource Broker* module is given in chapter IV.

### 3.3.3 Policy Enforcement Manager

The *Policy Enforcement Manager* is the component that is in charge of enforcing the policies. In contrast to other resource brokering environments, both the client and the resource provider can identify their policies. When requested, the *Policy Enforcement Manager* finds the appropriate resource(s) that can match the client request and returns the set to the *Resource Broker*.

What distinguishes PROBE from other resource brokering environments is that it goes far beyond the typical matching/allocation process to guarantee the provided level of allocation by providing the means of policies and Service Level Agreements (SLAs) and ensuring that the appropriate actions are taken in case of violation of the allocation terms.

PROBE looks at the allocation process as an SLA between the client and the resource provider.

At the time of a job's allocation, the *Policy Enforcement Manager* is notified so that it can create an SLA based on the provided policy. The *Policy Enforcement Manager* keeps monitoring this SLA during the life-time of the allocation and takes appropriate action (as specified in the policy) when a violation occurs.

The policy framework has been divided into a set of flexible and extensible components and uses a layered façade design pattern where plug-ins can be added and future needs can be incorporated. An XML-based Policy Scripting Language (PSL) has been introduced to handle the requirements of both the resource provider and the resource consumer.

To achieve high levels of scalability and performance, the *Policy Enforcement Manager* caches the minimal set of policy related information that it needs for resource matching and SLA monitoring. Also, to optimize the performance, we have introduced several techniques where we can avoid multiple and unnecessary parsing and optimize policies locally at their associated resources. In section 3.6.2, we briefly discuss our policy-based approach in handling resource brokering. The design of the policy framework is explained in greater detail in chapter V.

### 3.3.4 Resource Repository

The *Resource Repository* maintains up-to-date information about all the available resources in the system. To support prediction, the *Resource Repository* keeps some historical performance information about the resources. For the sake of scalability and high availability, we can have distributed *Resource Repositories* with each having its own set of resources. Of course, these *Resource Repositories* need to interact with each other to maintain consistency.

As we explain later in this chapter, we have adopted a layered approach in designing the repositories internal to PROBE. This makes the design independent of the underlying protocol. A protocol layer has been introduced that acts as an intermediate layer between



the protocol and the repository objects. It adapts the requests received from the repository object to the appropriate protocol format and adapts the responses from the protocol dependent objects to the internal format of PROBE.

### 3.3.5 Resource Monitor

The *Resource Monitor* keeps track of the current status of the resources. It updates the *Resource Repository* and the *Policy Enforcement Manager* frequently with up-to-date information about the resources.

The *Resource Monitor* supports different approaches for monitoring the status of the resources. This includes the *Push Mode* approach where the daemon that resides on the resource sends the required information to the *Resource Monitor*; and the *Pull Mode* approach where the *Resource Monitor* sends a request to the daemon asking about the current status of the resource. More on resource monitoring is given in section 3.6.3.1.

### 3.3.6 Job Repository

The *Job Repository* keeps information about all the currently running jobs that occupy resources. We have applied the same layered approach being applied in the *Resource Repository* to make the design independent of any underlying protocol.

### 3.3.7 Job Monitor

The *Job Monitor* keeps an eye on the jobs that occupy the managed resources along with their progress. It provides an interface to interact with some external components, e.g., *Workflow Manager*, and provides information about the current jobs that are occupying the resources. In case of job failure, the *Job Monitor* informs the *Resource Broker* to re-schedule the failed job.

### 3.3.8 Resource Daemon

PROBE requires that a daemon to be started on each resource under the control of their administrative domains. This daemon, implemented as a Jini service, acts as a gateway between PROBE and the managed resource. It handles the collection of statistical data about the resource and keeps track of the allocated jobs within the resource. It can also be used as an integration base to interact with other grid systems. Detailed design of the *Resource Daemon* is given in chapter IV.

## 3.4 Scenarios

In this subsection, we present high-level scenarios that can occur within PROBE. They are provided to describe the functionality of PROBE's modules and their interactions.

When PROBE is installed in an environment, the first thing that happens is that daemons are started on all the resources under the control of their administrative domains. These daemons act as gateways between PROBE and the managed resources. Each daemon registers with the *Resource Monitor*, providing the policies the resource provider wants to enforce on the resource. The *Resource Monitor* then notifies the *Policy Enforcement Manager* to keep track of the associated policies while matching the resource with the user's requirements. Based on the data probing approach (pull or push, as described in section 3.6.3.1), the *Resource Monitor* updates both the *Resource Repository* and the *Policy Enforcement Manager* periodically with up-to-date information about the resource.

Using any of the client APIs that PROBE supports, a client sends a problem description to the *Client Interface Module*. This request can be either an information-retrieval request, in which the client needs to get up-to-date information about resources, or a task-brokering request, in which the client has a problem and is looking for specific kinds of resources to solve that problem.

In the course of information retrieval, the following sequence of operations takes place:

- The *Client Interface Module* queries the *Resource Repository* or the *Job Repository* based on the constraints specified by the client.
- If the information is available within the repositories, the *Client Interface Module* will send it back to the client.
- Otherwise, the request is propagated to other PROBES by the *Client Interface Module*.

In the course of task brokering, the following sequence of operations takes place:

- The *Client Interface Module* passes the request to the *Resource Broker*.
- A unique job identifier is created and passed back to the *Client Interface Module* where the request can be tracked.
- The *Resource Broker* then consults with the *Policy Enforcement Manager*, which tries to find the appropriate resource(s) that can match the client request and returns the set to the *Resource Broker*.
- The *Resource Broker* then, based on the underlying scheduling algorithm, the user's job and the provided sub-set of resources, constructs a schedule and starts implementing it.
- At the time of the allocation:
  - A Service Level Agreement (SLA) is established between the client and the resource provider based on the client's terms.
  - The *Policy Enforcement Manager* is notified to start monitoring that SLA.
  - The *Job Monitor* is informed so that it keeps track of the job.
- A job can failed, be cancelled or complete successfully. In case of a successful finish:
  - The *Job Monitor* informs the *Resource Broker* where the schedule can be modified and then terminated.
  - The *Policy Enforcement Manager* is notified for each sub-task's finish to terminate the associated SLA.
- In case of job failure:

- The *Job Monitor* informs the *Resource Broker* where the schedule can be modified and the failed job can then be re-scheduled.
- Associated SLAs are cancelled and new ones will be created based on the new schedule.

### **3.5 Meeting Design Goals**

The design of PROBE is driven by several key goals. These goals include platform independence, modularity, scalability, site autonomy and interoperability. These goals have certain implications for the design of PROBE and the approaches that we have chosen in order to implement the prototype of the system. In this subsection, we identify the key design goals of PROBE.

#### **3.5.1 Platform Independence**

The underlying technology we use in implementing PROBE is Java. Besides being simple, safe, object-oriented, robust, and tightly integrated with the World Wide Web technologies, Java is a portable and platform-independent language enabling the resulting prototype implementation to run on any operating system platform with an implementation of the Java Virtual Machine (JVM). The JVM acts like a virtual computer making it possible to run programs written in Java on any machine, once they have been translated into bytecode.

PROBE is entirely written in Java and uses technologies (Jini/RMI) written in Java. Since the Java programming language is platform independent, PROBE can be considered to be platform independent that can run on heterogeneous systems.

#### **3.5.2 Modularity**

We have divided PROBE into several flexible modules, where each module implements an individual function. These modules define the basic services and capabilities required

to construct a distributed resource brokering environment. Some of these modules are broken down into sub-modules as we explain later in chapter IV and chapter V. Dividing into modules provides flexibility and ease of replacement making it easier to satisfy users' requirements in the future. Also, scalability and high availability can be achieved by replicating modules. Because the architecture of PROBE is so flexible, its different modules can be coallocated in one process or fully distributed across a number of machines.

Built on top of Java, as explained in chapter II, the Jini connection technology can be used to build a plug-and-play network of resources. Its attractive features and the degree of modularity it provides, make it appropriate for building the infrastructure of PROBE. As shown in Fig. 12, each module of PROBE has been implemented as a Jini service and thus has to register with a Module Lookup Service (MLS) dedicated for maintaining the list of modules in the environment. Modules could be in the same host, distributed across hosts in the same subnets or distributed across different subnets. Also, each daemon, representing a resource, has been implemented as a Jini service and thus has to register with a Resource Lookup Service (RLS) dedicated for maintaining the list of resources in the environment. The MLS and the RLS have also been implemented as Jini Lookup Services. Modules, daemons and their corresponding lookup services can be replicated and distributed across networks as the underlying grid environment continues to grow.

A service, representing either a module or resource, uses the discovery and join protocol to discover and register with its corresponding lookup services. It posts, with the lookup service, a service proxy, which is an object representing the services it provides. Services use the same protocol to locate and contact each other.

One issue with Jini is that it cannot be used efficiently across networks that do not support multicasting. To address this limitation, we enhanced Jini with a tunneling service that propagates Jini multicast messages across such networks [9]. In chapter VI, we describe this enhancement in more detail.

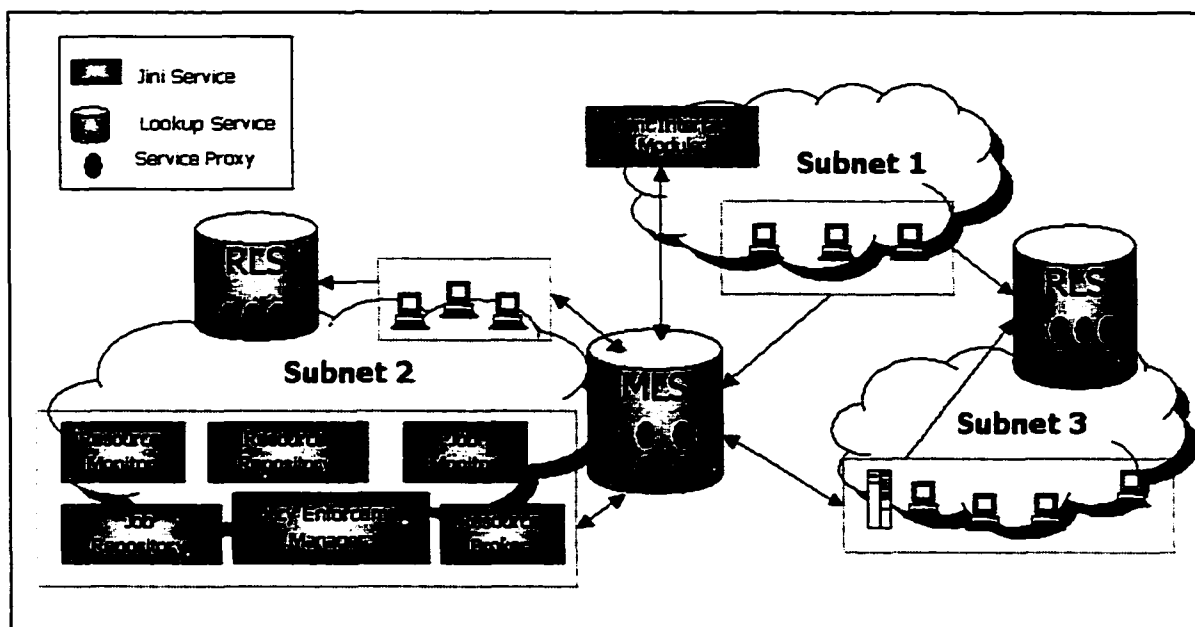


Fig. 12. Using Jini in PROBE

### 3.5.3 Scalability

A critical factor for a distributed resource brokering environment is its ability to grow, to some extent, with the number of resources, clients and the required capabilities. The resource brokering environment is expected to handle very high loads as the underlying environment continues to grow. The busiest resource brokering environment may even have hundreds of thousands of concurrent requests. To deal with this type of load, the resource brokering environment needs to have an extremely scalable architecture.

Scalability is one of the biggest challenges in building a distributed resource brokering environment, and it is becoming more of a challenge with the growth of resources and their clients. Most of the existing early scalability architectures achieve only limited scalability at the cost of excessive hardware requirements and network traffic.

Built on top of Jini and based on our modular architecture, we have designed PROBE in a way that it can be capable of scaling with the environment without resource problems or performance bottleneck. Given the flexible nature of Jini, PROBE's modules

(especially the heavily loaded ones) can be distributed across different processes on different machines to achieve high scalability.

One of the issues with replicating modules is keeping track of the various replicas. A replica should be added and subtracted with no harm. The Jini Lookup Service along with its protocols allows us to easily discover all the modules providing a specific service. In 8.4, we describe a proposed further enhancement to Jini, which allows it to support scalability for distributed applications.

The degree of flexibility that PROBE has, along with the Jini's enhancements make it easy to set up highly scalable distributed brokering architecture to meet the needs of typical grid environment. Through our scalable architecture, PROBE can process a large number of concurrent client requests and manage a large number of distributed heterogeneous resources.

#### **3.5.4 Site Autonomy**

As we mention in chapter I, a grid environment has a distributed collection of shared resources controlled by different administrative domains. Administrators in such domains want to make sure that their systems are safe, secure and available to their priority users.

Administrators control the daemons that run on behalf of their resources and specify their usage policies. For example, a site might insist that a resource cannot be accessed if the load is greater than 50%, the free physical memory is less than 512 MB, or not between 8 am and 5 pm.

PROBE's policy-based resource brokering approach, allows both the provider and the consumer to specify their policies and assure that the rights of both the owner and the consumer are respected. Using this approach, not only each administrative domain, but also each resource owner can identify their own policies.

We have noticed the urgency of having a flexible language that provides the necessary power to express the diverse kinds of rules that both resource providers and consumers can have. We have designed and implemented a very flexible XML-based Policy Scripting Language (PSL), which can be used for this purpose. A detailed

explanation of the policy framework and the Policy Scripting Language is given in chapter V.

### 3.5.5 Interoperability

One of the major issues behind this research is to build a general-purpose, stand-alone resource brokering environment that can be easily used in various grid environments and at the same time can be smoothly layered on top of various grid systems. Our modular approach, open architecture, rich interfaces, layered approach and the use of XML for resource, job and policy specifications, allow us to build an interoperable framework that grid systems can interoperate with. We achieved interoperability at different levels:

- **Layered Approach.** In building grid systems, brokering environments in particular, different approaches can be applied. It is not known which approaches are best. PROBE adopts a layered architecture for the internal repositories (both resources and jobs), brokering infrastructure, resource daemon and policy framework. The main objective is to make the targeted module independent of the underlying protocol. A protocol layer has been introduced which acts as an intermediate layer between the underlying protocol and the module object. This layer is considered to be a part of the module object. It adapts the requests received from the module object to the appropriate protocol format and adapts the responses from the protocol dependent objects to the internal format of PROBE. This layered architecture gives grid systems the flexibility to adopt different approaches as their environments require and makes the framework independent of any architecture.

Fig. 13 illustrates the use of the layered approach in implementing the repositories internal to PROBE. Later on in this chapter we explain the usage of this approach within the PROBE resource daemon. Chapter IV explains the usage within the *Resource Broker* and the resource daemon, and chapter V explains the usage within the *Policy Enforcement Manager*.



However, the misuse of the layering feature might result in some overhead in the performance, mainly when communication is involved.

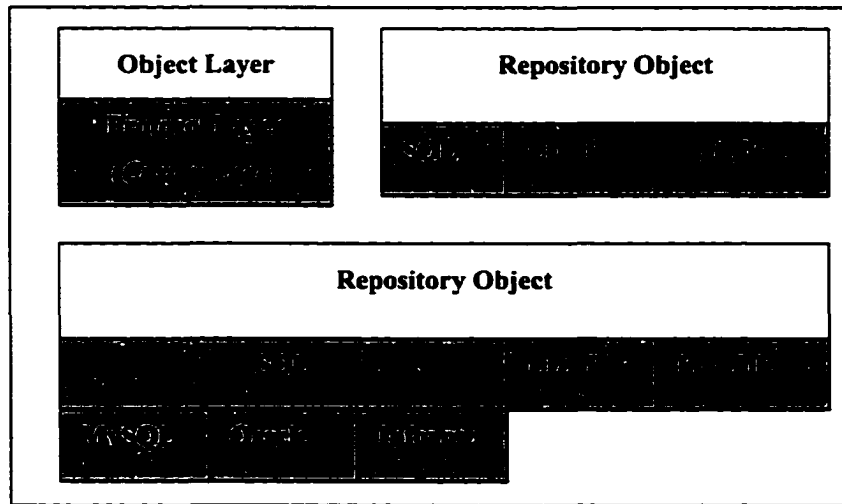


Fig. 13. Different approaches in applying the layered architecture in the repository objects

- **Open APIs.** The Global Grid Forum (GGF) [47] is a community-initiated forum of individual researchers and practitioners working on distributed computing, or grid technologies. The focus is to generate the best practice documents, protocols, and API specifications to enable interoperability between existing grids.

In designing PROBE, we follow the protocols and APIs suggested by the Grid Forum. For example, we follow the resource specification defined by the Grid Information Service Group [78], extend it and express it using XML (as explained in section 4.7.1). Also, we have studied most of the existing grid environments; mainly the most widely accepted ones such as Globus and Sin Grid Engine. PROBE provides a rich, open API and a set of specifications based on public standards proposed by the Global Grid Forum and standard tools such as XML. For example, the *Client Interface Module* has been built so as to provide rich and flexible interface to other grid environments. It also provides an interface to other

resource brokering environments. The other resource brokering environments can belong to the same grid system or may be part of another system, e.g., Globus or NetSolve. The *Client Interface Module* also provides an API that can be used by the resource brokering environment's clients. A client can be a user application, some other component of the system, e.g., *Workflow Manager*, or may be another system, e.g., Globus and NetSolve.

- **Flexible specification languages.** The extensible Markup Language (XML) [38] is a specification for creating structured documents and data. The beauty of XML is that it isolates the content format of the source from the content format of the target making it possible to take data from any source and deliver it to any target. XML is evolving and quickly becoming a standard way to identify and describe data because it has proved easy to use and deploy. This standard has been recommended by the World Wide Web Consortium (W3C) and can be used as a common meta-language that enables data to be transformed from one structure to another.

We provide a set of script specifications for resources, jobs and their associated policies based on XML. Sections 4.7.1, 4.5, and 5.6 respectively describe these specifications in detail. Using XML allows us to leverage off existing freely available XML parsers and editors to develop our tools. It makes the development of our tools easier by using the existing freely available XML parsers and editors. Also, such an XML-based script presents the potential of inter-framework portability.

Some grid systems such as NetSolve, Ninf and Condor cannot map well onto wide area environments where site autonomy and heterogeneity complicate their task. For such systems, an interoperability layer needs to be developed such that those systems can be integrated with wide area grid environments such as Globus and Legion. Moreover, there is an increasing trend towards integrating existing grid systems together to form super grid environments. With its interoperability, heterogeneity, flexibility, scalability, rich-

context and easy-to-extend modules, PROBE can be that interoperation layer that can integrate a variety of grid environments as illustrated in Fig. 14.

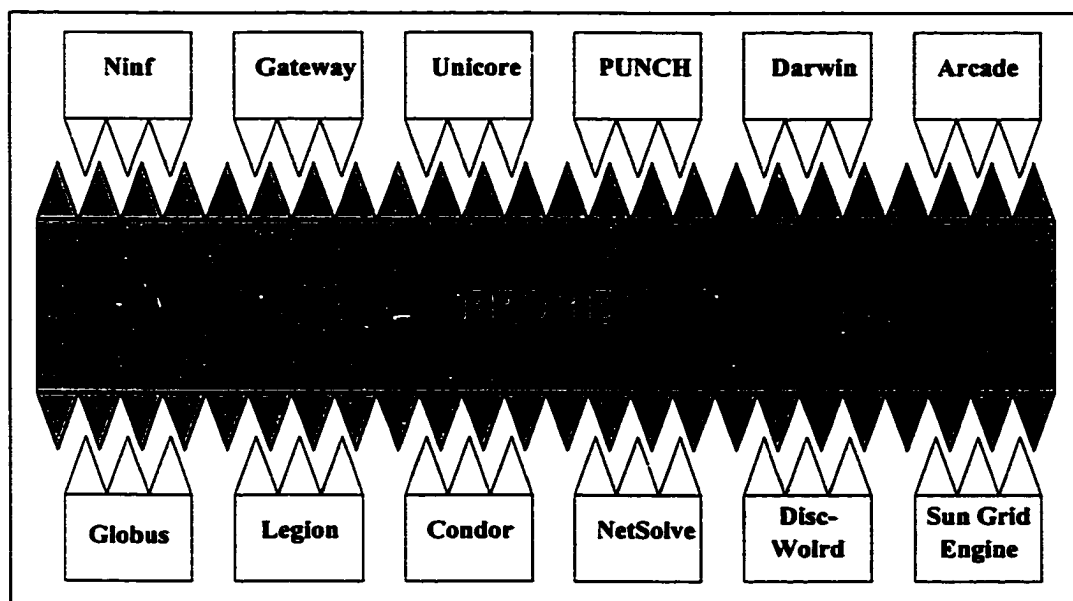


Fig. 14. Different grid environments interoperate via PROBE

### 3.6 Functionalities

In this subsection, we describe the main functionalities the PROBE system provides.

#### 3.6.1 Resource Brokering

Task scheduling is one of the most critical issues in building a heterogeneous distributed resource brokering environment and is known to be an NP-Complete problem [37]. Many heuristics have been developed to generate near-optimal schedules [84],[123],[124]. Scheduling is said to be static when the resource on which the job is going to be allocated is assigned before execution [4]. Dynamic scheduling is performed at run time as a means of maximizing resource utilization, job throughput, or other metrics depending on the

scheduling policy [5],[57]. Static scheduling is easy to implement, and is more widely used [23].

Scheduling of user's required tasks is a very challenging issue in building a resource brokering environment and as a result most of the available resource brokering environments implement only minimal scheduling capabilities [17],[27],[66],[112]. Most of the existing efforts suffer from limitations such as:

- resources are dedicated;
- resources are of homogeneous types;
- resources do not fail;
- resource load is predictable;
- task is profiled and its resource usage is known in advance;
- task can be allocated on any resource; etc.

PROBE provides efficient brokering of resources. The *Resource Broker* module is the one in charge of this task. As we detail in chapter IV, the design allows the plug-and-play of any scheduling algorithm and application problem the user might provide. As we illustrate in Fig. 15, the *Client Interface Module* receives a problem description from a client including a task that needs to be scheduled and allocated. The *Client Interface Module* then passes the information to the *Resource Broker*, where a unique job identifier is created and passed back to the *Client Interface Module* so that the request can be tracked. PROBE takes placement restrictions into account while scheduling tasks. The *Resource Broker* then consults with the *Policy Enforcement Manager* and based on the underlying scheduling algorithm, the user's job and the provided sub-set of matched resources, constructs a schedule and starts implementing it. Based on the client's choice, the *Resource Broker* can allocate the targeted resource(s). The allocation decision can take several approaches:

- *Client-Controlled Allocation*, in which the client specifies the resource statically to the resource brokering environment. For example: "run my aircraft design application on tango.cs.odu.edu".
- *Broker-Controlled Allocation*, in which the resources are chosen by the *Resource Broker* based on some constraints specified by the client. For example: "run my

*biomedical problem (Bio) for 6000 combinations on nodes each with at least 1000 MHZ CPU speed”.*

- *Dynamic allocation:* In this case, the allocation decision may change during execution due to resource failure, poor performance, load imbalance, etc.

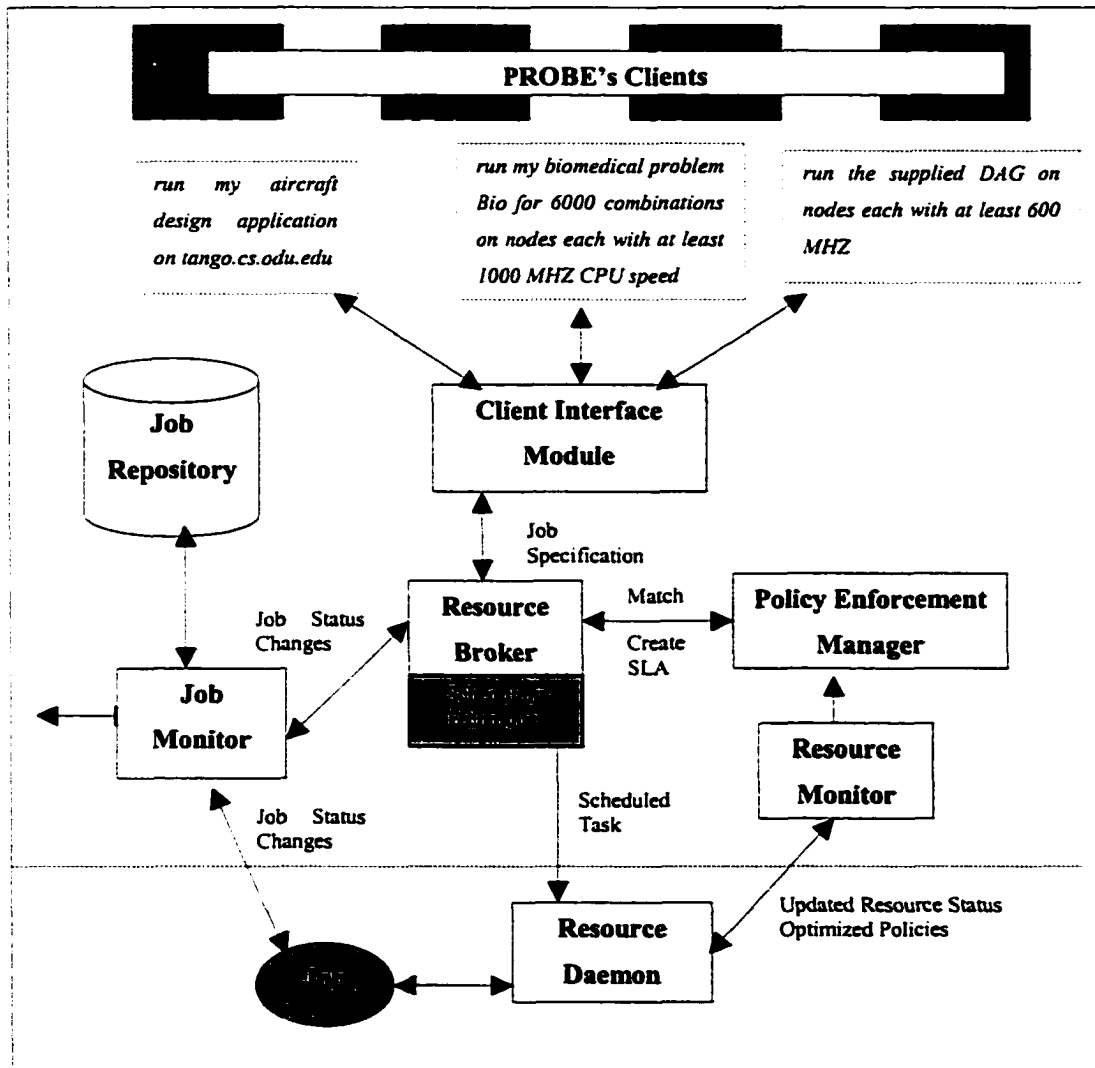


Fig. 15. Brokering Scenarios

Sometimes, the task requires co-allocation where a set of resources needs to be available for use simultaneously. The current implementation of PROBE supports a plugin for this kind of application.

After the schedule is created, the *Resource Broker* implements it. The job is dispatched to the resource once it is ready. The *Resource Broker* hands the scheduled task to the daemon that runs on the resource. Authentication and data staging will be done at this phase by other components of the systems, e.g., *Data Manager* and *Security Manager*. If successful, the daemon spawns a process to monitor the job execution. At this time, the *Resource Broker* informs the *Job Monitor* to monitor the execution of the job. When the job finishes successfully, the *Resource Broker* terminates the schedule. A detailed design of the *Resource Broker* is given in chapter IV.

### 3.6.2 QoS Brokering

In a typical grid environment where resources, most of the time, are not dedicated, it is very important to assure the client that the QoS of the allocation is ensured even after the allocation is made. One of the main issues behind this effort is to provide a QoS policy framework that makes it easy for both the resource provider and the resource consumer to define their policies.

On the other hand, policy-based frameworks are increasingly being used within the network community as means of guaranteeing a given level of the provided Quality of Service (QoS). In such frameworks, a Service Level Agreement (SLA) is defined as a formal negotiated agreement of service levels between two parties, the service provider and the service consumer. An SLA can comprise one or more policies in which a policy can be seen as a set of conditions and actions that need to be taken when those conditions are met.

PROBE employs a policy-based approach for resource brokering that attempts not only to match the user's request with the right set of resources, but also to assure the guaranteed level of the allocation. The *Policy Enforcement Manager* is the module that is in charge of enforcing the policies, in which both the clients and the resource providers can identify their policies. When requested, the *Policy Enforcement Manager* finds the appropriate resource(s) that can match the client request and gives them to the *Resource Broker*. Unlike other resource brokering environments, PROBE goes far beyond

matching/allocating resources to provide allocation assurance by introducing the concept of Service Level Agreements (SLAs) and assuring that the appropriate action will be taken in case of violations of the agreements.

The brokering process requires interaction between different modules of the system. In order to simplify the process, Fig. 16 defines the different stages that need to be considered while handling user's requests. These stages are:

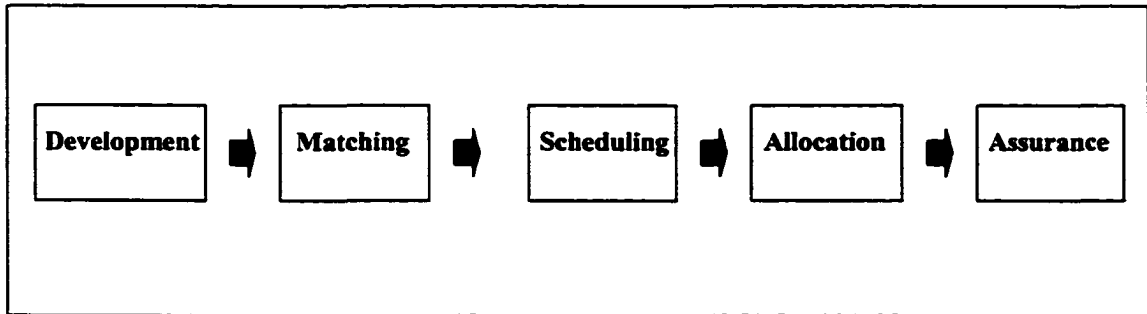


Fig. 16. Brokering cycle

- *Development*: the stage where the client specifies its requirements. PROBE works in conjunction with other components of the underlying grid system. The client could hand its requirements to the *Workflow Manager*, which in turn creates the appropriate request and hands it to PROBE.
- *Matching*: this is where the system matches the client's requirements with the applicable set of resources.
- *Scheduling*: a schedule is created based on the underlying scheduling algorithm and using the matched set of resources.
- *Allocation*: the resulting schedule is implemented and a Service Level Agreement is created for each resource allocation.
- *Assurance*: SLAs are monitored to assure that the allocation terms are not violated. Appropriate action(s) (if specified) will be taken in case of a violation.

A detailed discussion of the policy-based framework is given in chapter V.

### 3.6.3 Monitoring

Monitoring is the process of obtaining, collecting and presenting the information required by an *observer* about the observed system [61],[67]. It is one of the critical issues in building a distributed computing environment in general and a distributed resource brokering environment in particular where distribution and issues such as site autonomy and resource heterogeneity complicate the task of monitoring.

PROBE has three observers; *Resource Monitor* that monitors the status in the managed resources; *Job Monitor* that monitors the jobs occupying them; and the *SLA Monitoring Agent*, part of the *Policy Enforcement Manager* infrastructure, that monitors the SLAs being created for the allocated jobs and their associated policies. In this subsection, we describe those observers and the approaches that we have chosen in order to implement them.

#### 3.6.3.1 Resource Monitoring

As we mentioned earlier in this chapter, PROBE's design employs that a daemon resides on each resource to provide a gateway to the resource. It collects statistical data about the resource and keeps track of the allocated jobs. For example, in UNIX environment, the resource daemon opens a pipe to read from a program that gets this information such as *top*, *ps*, *who*, and *w*.

The *Resource Repository* holds the up-to-date information about the status of the resources. The *Resource Monitor* is the component that monitors the underlying resources and keeps the *Resource Repository* up-to-date. For this, PROBE supports two approaches. The first one is the *Push Mode* approach where the daemon that resides on the resource sends the required information to the *Resource Monitor* either periodically or based on some specific events (event-driven mechanism). The second one is the *Pull Mode* approach where the *Resource Monitor* sends a request to the resource daemon asking about the current status of the resource. This mode can also be performed either



periodically or on-demand (event-driven mechanism). The event-driven mechanism has some advantages over the periodic one since it does not fill up the network with massive traffic and also provides more accurate results. However, it may have poor performance since it does not rely on cache information in the *Resource Repository*. The *Resource Monitor* and the resource daemon provide an API where information about resources can be obtained using all these modes. The user can either chose *Pull*, *Push* or a hybrid approach that combines both.

In section 4.7, we present a schema that can be used to describe resources. That schema relies on the DTD given on Fig. 17 in order to specify the disseminating options.

```
<!--Disseminating.dtd-->
<!ELEMENT Disseminating (Push?,Pull?)>
<!ELEMENT Push (Periodic?,EventBased?)>
<!ELEMENT Pull (Periodic?,EventBased?)>
<!ELEMENT Periodic EMPTY>
<!ATTLIST Periodic Interval CDATA>
<!ELEMENT EventBased EMPTY>
```

Fig. 17. Schema to specify disseminating options

### 3.6.3.2 Jobs Monitoring

The *Job Monitor* monitors the execution of the currently running jobs on the resources of the system. It provides an API to interact with some internal components, e.g., *Resource Broker*, and also external components, e.g., *Workflow Manager*.

The *Job Monitor* provides an API to manipulate the currently running jobs. In some situations, e.g., poor performance or failure, a job may have to be stopped, resumed, cancelled or migrated to another resource. The API provides support such tasks. In case of resource failure, the *Job Monitor* will inform the *Resource Broker* so that it can re-schedule all the failed jobs.

The *Execution Monitor*, part of the PROBE resource daemon, keeps track of the allocated jobs within the resource and updates the *Job Monitor* about their status changes.

This part has been implemented using the distributed event notification mechanism in Jini.

The current implementation of the *Job Repository*, which keeps information about all the currently running jobs, has been done using MySQL. As we explain in chapter IV, a user's request, including the job description, is represented using XML. We store the XML specification of the jobs in the object-relational form and use the *Request Parser* to write/retrieve jobs information to/from the *Jobs Repository*.

Also, to make it easier for the user to track the job and make sure that it has been executed correctly, the standard output and the standard error are redirected to *\$ID.out* and *\$ID.err* respectively, where *\$ID* represents the unique identification being assigned to the job.

### 3.6.3.3 SLA Monitoring

The *SLA Monitoring Agent*, part of the *Policy Enforcement Manager* infrastructure, is the place where the allocation is assured. Once the job is allocated, an SLA is created with the user's policy. The *SLA Monitoring Agent* keeps monitoring the associated policies and takes the appropriate action (if any) in case of violations. For example, a credit could be issued to the user.

The *SLA Monitoring Agent* provides an API to interact with some internal components, e.g., *Resource Broker*, and also external components, e.g., *Workflow Manager*, where SLAs can be manipulated. Based on changes in the job's status, an SLA might be stopped or terminated. More detail about SLA monitoring is given in chapter V.

## 3.7 Summary

In this chapter, we have described the overall architecture of PROBE, a Policy-based Resource Brokering Environment, in great detail. We have discussed the various approaches that we have chosen to implement the prototype along with the related issues. As we explain in chapter VI, the implementation of PROBE focuses on providing

prototype modules, as shown in Fig. 11. Given our modular approach, rich APIs and the interoperable architecture, more functionality can easily be added in the future.

## CHAPTER IV

### RESOURCE BROKER: A DETAILED ARCHITECTURAL VIEW

#### 4.1 Overview

One of the major tasks of a resource brokering environment is to provide an efficient brokering of resources. Given the application's constraints, provider's rules, distributed heterogeneous resources and the large numbers of scheduling choices, the resource brokering environment has to decide where to place the user's jobs and when to start their execution in a way that yields the best performance to the user and the best utilization to the resource provider [65].

As we have stated earlier in chapter III, the *Resource Broker* module is the module that is in charge of the brokering tasks within PROBE. The *Resource Broker* needs to be flexible and generic enough not only to handle the different kinds of user tasks but also to handle the different kinds of scheduling techniques the system is going to incorporate.

In this chapter we present the design and implementation of the flexible, extensible and generic brokerage infrastructure for computational grids following a layered approach and façade design pattern and using XML as the underlying specification language.

#### 4.2 Architecture

We have designed and implemented a resource brokering infrastructure for computational grids that can be easily utilized by various grid systems [7]. As illustrated in Fig. 18, we have divided the *Resource Broker* into two flexible agents, where each agent implements an individual function. These agents define the basic services and capabilities required to construct a distributed resource brokering system. Dividing into agents provides

flexibility and ease of replacement making it easier to satisfy users' requirements in the future. Also, scalability and high availability can be achieved by replicating those agents.

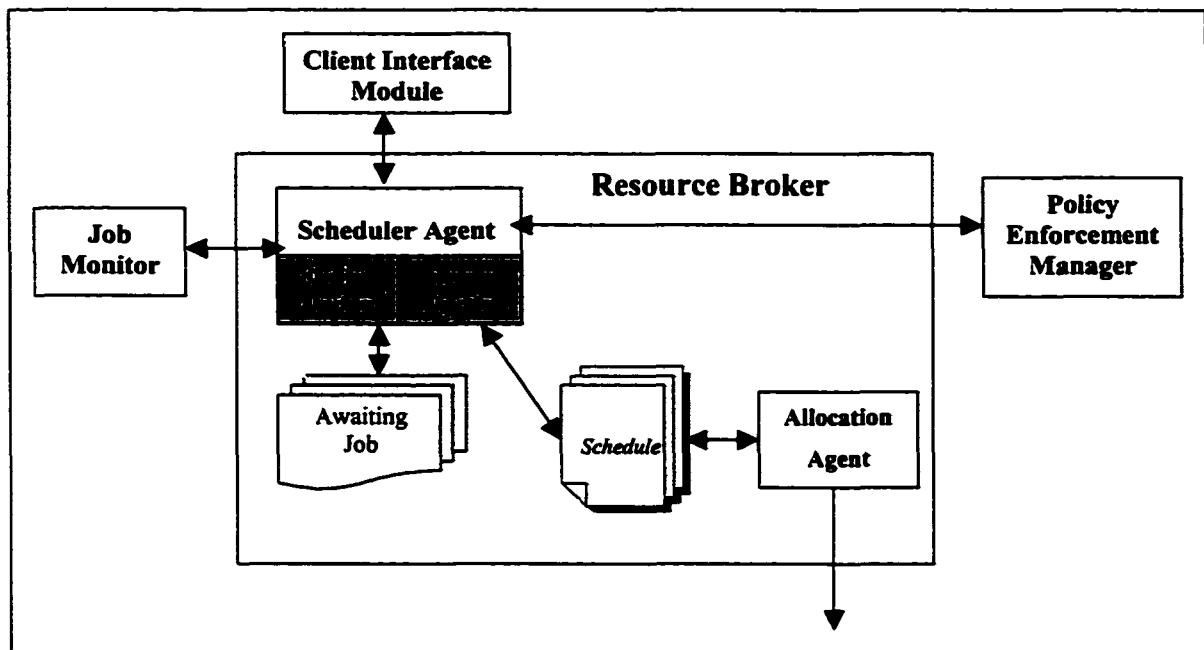


Fig. 18. Overall Architecture of the *Resource Broker*

1. **Scheduler Agent.** This is the heart of our *Resource Broker* and the first point of contact for the user's job. Based on the underlying scheduling algorithm, the user's job and the matched sub-set of resources provided by the *Policy Enforcement Manager*, the *Scheduler Agent* is going to construct a near optimal active schedule object and pass it to the *Allocation Agent* where it is going to be implemented.

The schedule is an active object that has an order and placement of tasks that need to be allocated. The *Scheduler Agent* creates the schedule based on the application type and the underlying scheduling algorithm. The schedule then gets manipulated by the different components of the *Resource Broker* as necessary.

A unique job ID is assigned for each job at the time of creating the schedule by the *Scheduler Agent*. In case of aggregated jobs, unique job IDs are assigned for the job and all its sub-tasks. This makes it easy to track jobs.

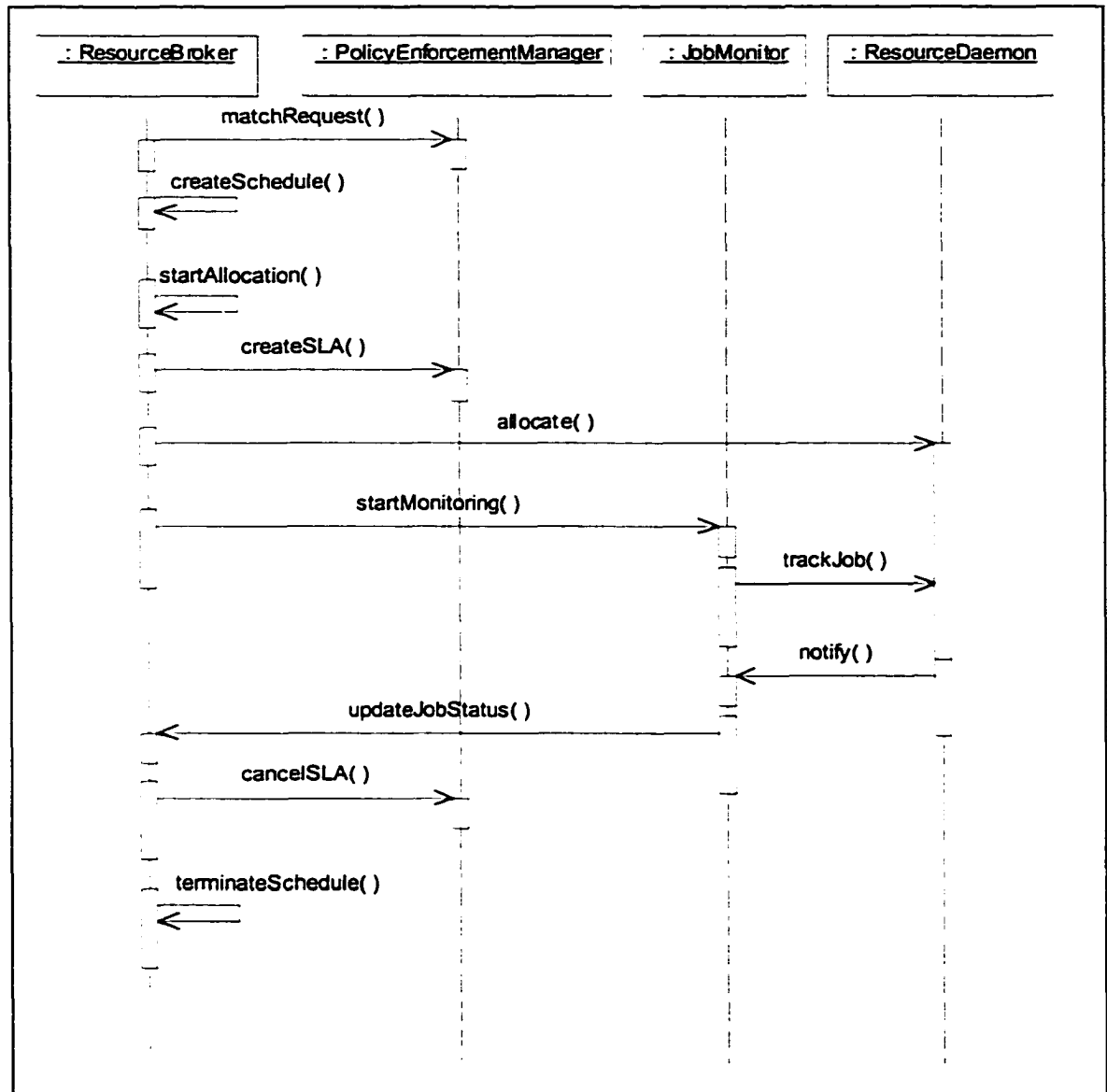


Fig. 19. An overall event diagram for interaction between the different components of the *Resource Broker*

The *Scheduler Agent* maintains an internal queue of jobs currently in the system and that have not been scheduled yet including those that failed and need to be rescheduled. The *Scheduler Agent* uses a queuing algorithm to select the next job to schedule. The approach we follow allows the users to plug in their

scheduling and queuing algorithms as needed. The design approach makes these algorithms look like black boxes to the *Scheduler Agent*.

2. **Allocation Agent.** The *Allocation Agent* is responsible for implementing the created schedule, i.e., launching the tasks on the designated resources. The *Allocation Agent* notifies both the *Policy Enforcement Manager* so that it creates an SLA based on the provided policy and keeps on monitoring that SLA during the life-time of the allocation; and the *Job Monitor* which in turn keeps on monitoring the allocated job. The *Job Monitor* then updates the *Scheduler Agent* as necessary about the significant changes in the job status (*Finished*, *Failed*, *Stopped*, etc). The *Scheduler Agent* in such a case might need to cancel the associated SLAs and re-schedule some of the associated tasks based on the underlying scheduling and queuing algorithms.

We follow a layered approach and façade pattern approach in designing and implementing these modules. In section 4.4, we explain this approach in greater detail.

### 4.3 Resource Daemon: Detailed Architecture

PROBE requires a daemon to be started on each resource under the control of their administrative domains. This daemon, implemented as a Jini service, acts as gateways between PROBE and the managed resource. It also can be used as an integration base to interact with other grid systems. As illustrated in Fig. 20, the work within the daemon has been divided into five components:

1. **Core Daemon:** implements the infrastructure necessary for the daemon to be a Jini service and for managing the interactions among the other components.
2. **Data Collector:** handles the collection of statistical data about the resource and passes it to the *Local Policy Enforcer* for optimization and local policy parsing before handing it to the *Resource Monitor*.

3. **Execution Monitor:** keeps track of the allocated jobs within the resource and updates the *Job Monitor* about their status. This part has been implemented using the distributed event notification mechanism in Jini.

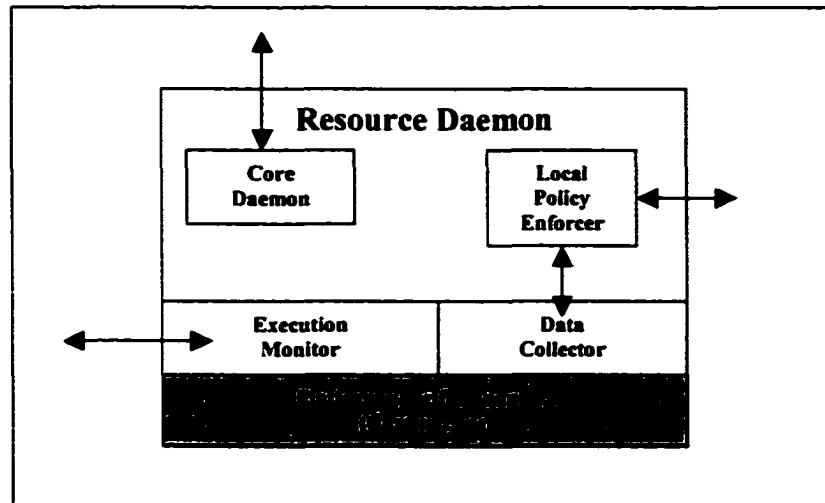


Fig. 20. PROBE Resource Daemon

4. **Local Policy Enforcer:** a resource can have two kinds of policies: allocation policies that define how the resource can be utilized, and internal policies that are meant for internal use within the resource such as setting a warning level to avoid an allocation violation. The *Local Policy Enforcer* keeps track of the policies associated with the resource along with the local policies. It also does some optimization of the associated policies before updating the *Policy Enforcement Manager*. Details about this component is given in chapter V.
5. **Platform Specific Adaptor:** maps the data collection and job execution/monitoring requests to the specific platform (such as Globus, Sun Grid Engine, UNIX, Linux, NT, etc). For example, in a UNIX-based resource daemon, the data collector may open a pipe to some of the existing UNIX utilities such as *top*, *ps*, *uname* and *vmstat* so that it can read the current statistics. Fig. 21 illustrates some of the platform adaptors of the current prototype implementation of PROBE.



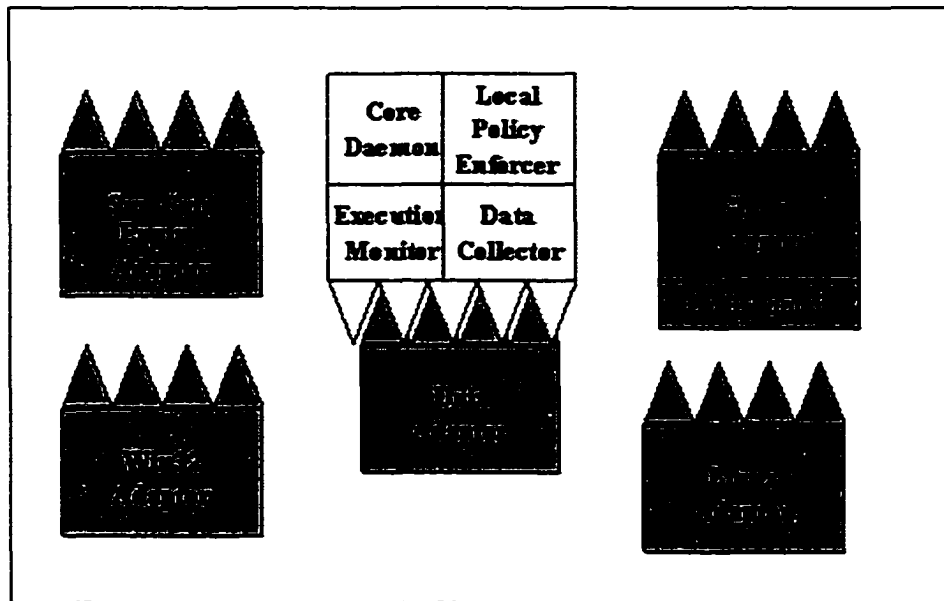


Fig. 21. Different platform adaptors for the resource daemon

#### 4.4 Design Pattern

Design patterns are simple and elegant solutions to specific problems in object-oriented software design [106]. They represent solutions that have worked out for experienced Object Oriented designers in the past. The Facade design pattern is the one that wrap a complex set of classes into a much simpler interface. A facade object is introduced to provide a single, simplified interface to more general facilities of a subsystem.

We have noticed the need of decoupling the *Resource Broker* from any specific queuing algorithm, scheduling algorithm and the type of jobs that it is going to deal with. One way to address this issue is to use a facade object that defines a higher-level interface and makes the subsystem easier to use. As shown in Fig. 22, we follow the facade design pattern for the objects being used by the *Resource Broker*. This shields the *Resource Broker* from any particulars of the users' queuing algorithms, scheduling algorithms and jobs. The *Resource Broker* sees them as black boxes. To simplify the figure, we have

shown only some of the correspondence classes and hidden the signatures of the operations

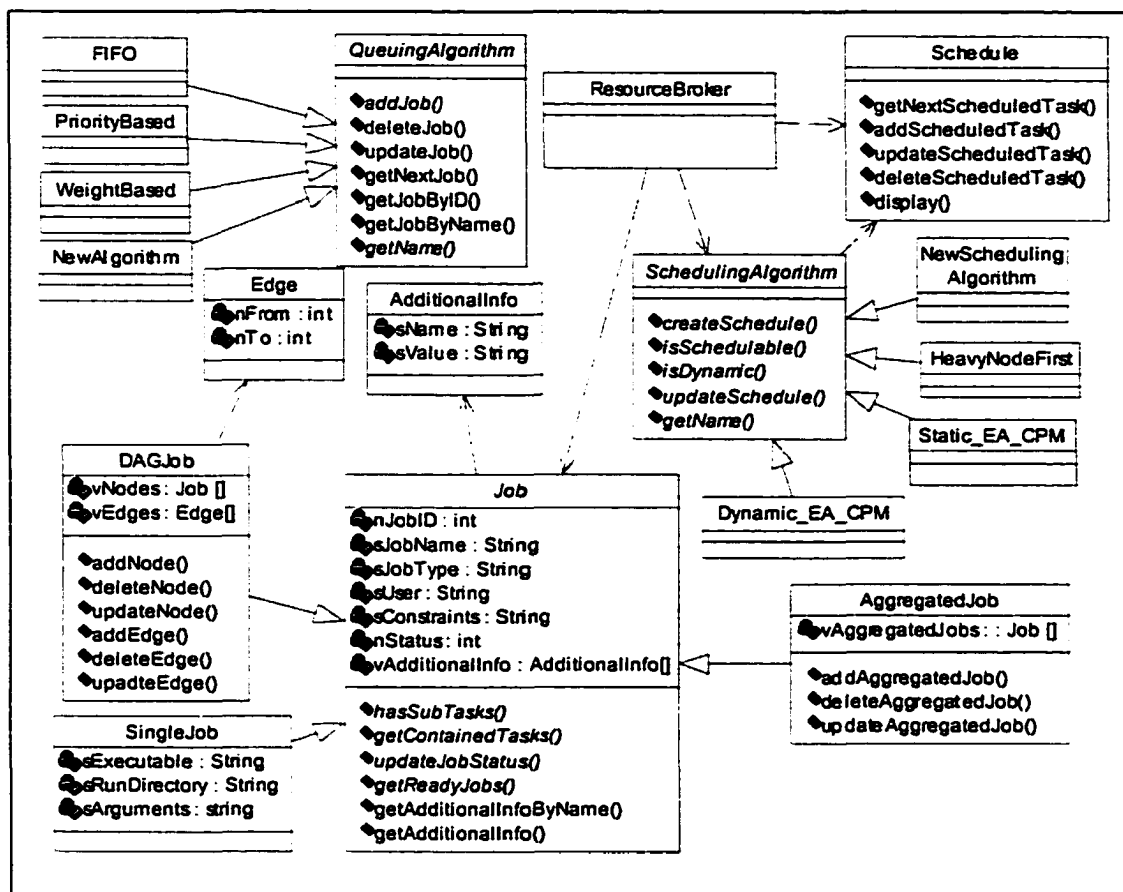


Fig. 22. Partial Class Diagram that illustrates the use of the Façade Design Pattern in PROBE's brokering infrastructure

An example of the use of the façade approach is the job types. *Job* is an abstract class and needs to be implemented by the job type. The *Resource Broker* and the *Scheduling Algorithm* have a unified interface to a set of Job Types. This makes the design independent of any job type. Initially, we support *Single*, *Aggregated* and *Direct Acyclic Graph (DAG)* jobs. A *Single Job* is the basic job type in our framework that represents the executable portion of an application. An *Aggregated Job* is where a group of tasks are combined to form a unified job such as: *CoAllocation Job* that requires that a set of

resources are available for use simultaneously; and *Parametric Job* where the same program is repeatedly executed with different initial conditions as a means of exploring the behavior of a complicated system. A *DAG Job* represents an application program that consists of a collection of heterogeneous modules (application codes from different disciplines). A typical distributed application requires these modules to be executed in some order and possibly on different machines.

Adding new job types to the system does not require modification to the code nor its recompilation. One needs to create a class inheriting Job and implement the abstract methods. The same approach is used for the scheduling algorithm and the queuing algorithm. This gives PROBE the flexibility to plug and play any one of them based on the requirements of the overall system. In chapter VII, we demonstrate this approach in greater detail.

```

<!--Request.dtd-->
<!ENTITY % JobType "Single|Aggregated|DAG">
<!ENTITY % aggregationType "CoAllocation|Parametric">
<!ENTITY % CoAllocationTiming "SameTime|DifferentTime">
<!ENTITY % PolicyDTD SYSTEM "Policy.dtd">
%PolicyDTD;
<!ELEMENT Request ((%JobType;))>
<!ELEMENT Single (Policy?,AdditionalInfo*)>
<!ATTLIST Single
    Name          CDATA #IMPLIED
    Executable     CDATA #IMPLIED
    RunDirectory   CDATA #IMPLIED
    Arguments      CDATA #IMPLIED>
<!ELEMENT Aggregated (Single+,Rule?,AdditionalInfo*)>
<!ATTLIST Aggregated
    Name          CDATA #IMPLIED
    Type           (%aggregationType;) #IMPLIED
    Timing         (%CoAllocationTiming;) #IMPLIED>
<!ELEMENT DAG ((%JobType;)+,Dependency*,Rule?,AdditionalInfo*)>
<!ATTLIST DAG
    Name          CDATA #IMPLIED>
<!ELEMENT Dependency EMPTY>
<!ATTLIST Dependency
    From          CDATA #IMPLIED
    To            CDATA #IMPLIED>

```

Fig. 23. Flexible Job Language (FJL).

#### 4.5 Flexible Job Language (FJL)

The underlying language used to specify the user's request is based on XML. This allows our system to interact with external systems and exchange jobs information. We have designed a Flexible Job Language (FJL) that can be used to express the user's request. FJL can be extended to satisfy complicated user's requirements in the future. Fig. 23 illustrates the schema that specifies how the request can be generated. This schema relies on the Policy Scripting Language (PSL) in which the user can specify the associated policy. PSL is explained, in detail, in section 5.6. An example FJL script representing a sample DAG application is given in Fig. 24.

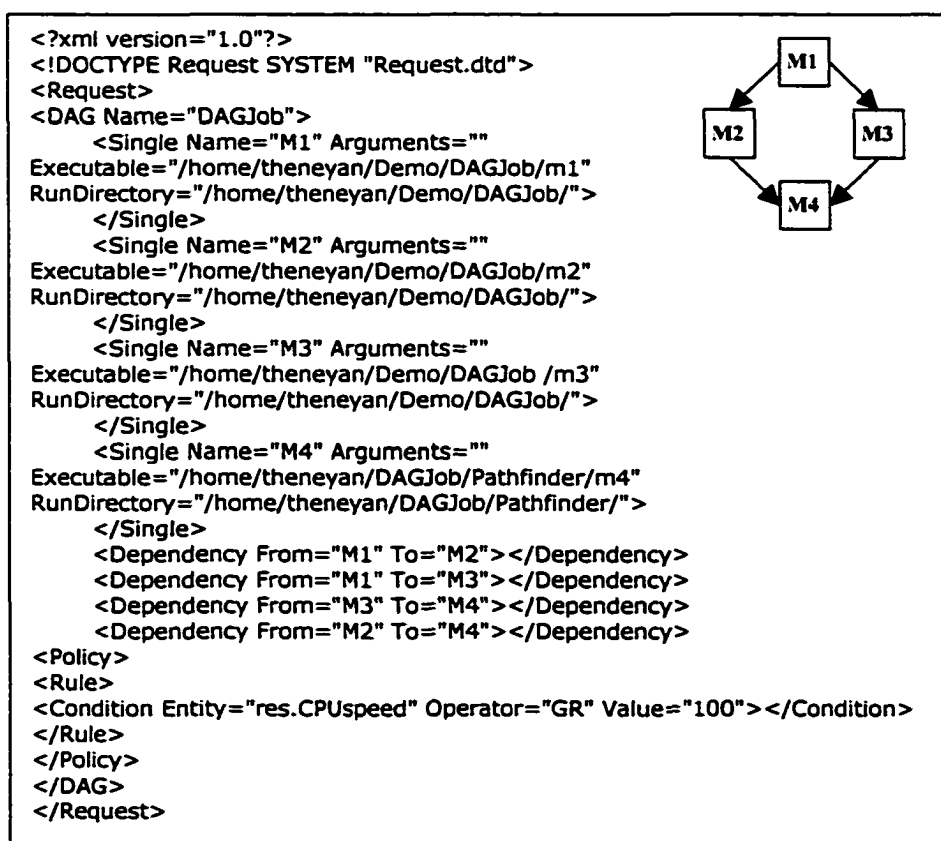


Fig. 24. Example FJL script representing a sample DAG application.

#### 4.6 Job State Transition Diagram

When a job is submitted to the *Resource Broker*, it passes through a series of states till it completes successfully, is cancelled or fails. The Job Monitor API allows the job to be monitored. It allows even sub-tasks of an aggregated application (such as DAG, Co-Allocation, Parameterized, etc) to be monitored. Fig. 25 shows the possible states a job can pass through when submitted to the *Resource Broker*. Job states are described below:

- **Waiting:** A job is in a waiting state when it is submitted to the PROBE system and is waiting for resource allocation. This could happen when the job is failed, stopped and then resumed, or can't be scheduled at the current time.
- **Scheduled:** A job transitions to this state when it is scheduled but not yet allocated.
- **Running:** A job transitions to this state when the resource gets allocated and the execution starts.
- **Stopped:** A job transitions to this state when the user stops the request. The user can stop the request at any time. A stopped job can be resumed.

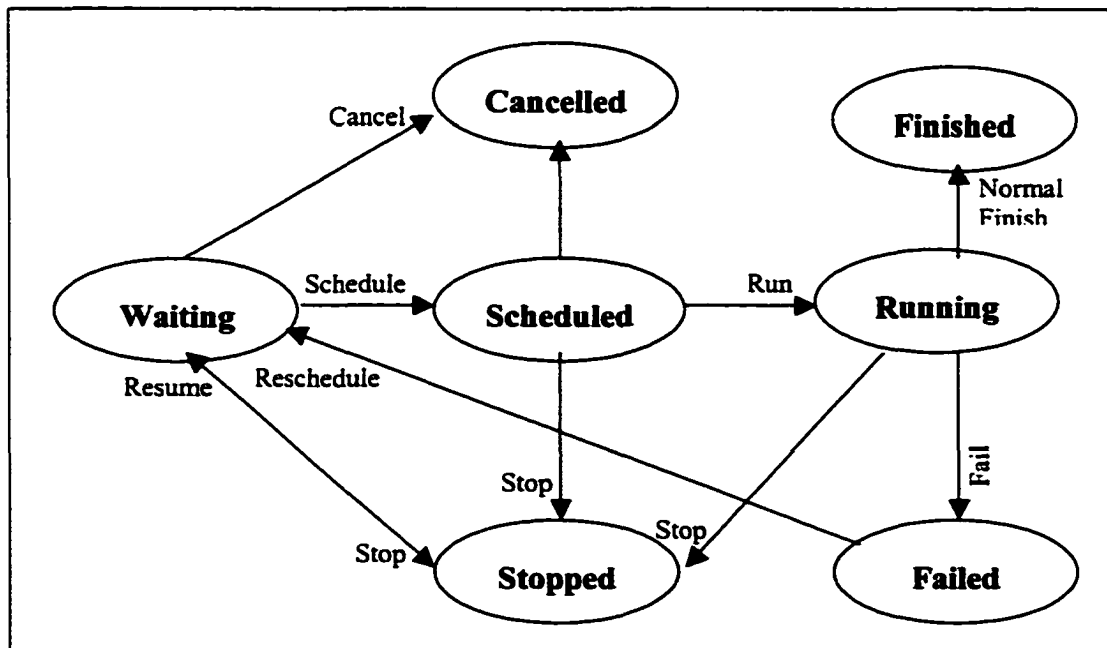


Fig. 25. A Job State Transition Diagram in the Resource Broker

- **Cancelled:** A job transitions to this state when the user cancels the request. The user can cancel the request at any time.
- **Failed:** A job can fail due to several reasons. It could be due to process failure, server crash, networking failure, etc.
- **Finished:** A job transitions to this state when it normally finishes its execution.

#### 4.7 Resource Types

A *resource* denotes any entity that is meant to be shared in a grid environment. It could be *computational*, *network*, *software*, *data* or *storage*. The current prototype implementation of PROBE focuses on computational grids. However, the design of PROBE allows the *Resource* object that represents the managed resource, as shown in Fig. 26, to look like a black box for the different components of PROBE.

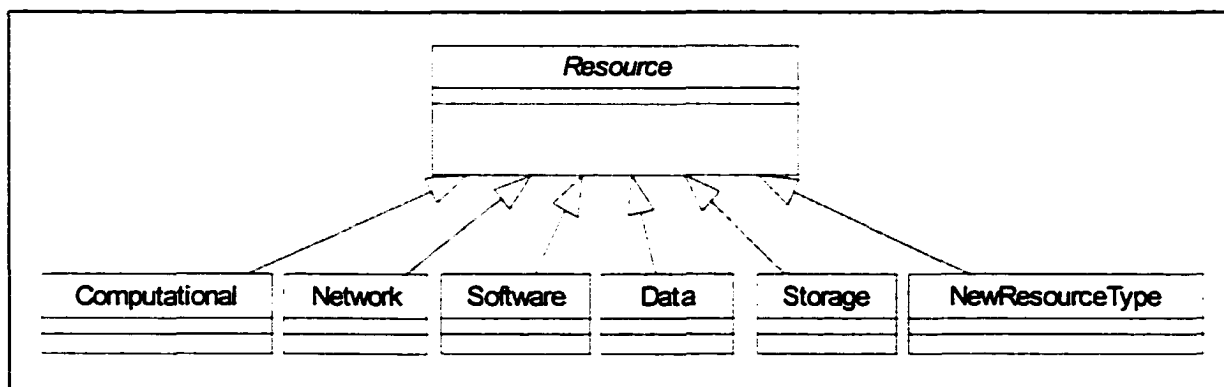


Fig. 26. Class diagram of the resource types

A new resource type can be easily added by extending the *Resource* abstract class. PROBE APIs are flexible enough to handle such resource heterogeneity. The vision of allocation varies from one resource type to another as illustrated in Table 2.

TABLE 2  
ALLOCATION VISION FOR DIFFERENT TYPES OF GRID RESOURCES

Resource Type	Allocation Philosophy	Client-side Example
Computational	Executes a request.	<i>run my aircraft design optimization problem in a set of machines each with at least 1 GHz CPU speed and 256 MB of free physical memory.</i>
Software	Obtains a license or uses software. PROBE could be viewed as license manager in such case.	<i>give me a license to use the CFD package.</i>
Data	Obtains the right to use access for a data source. Resource denotes the data being stored or retrieved.	<i>retrieve the data that satisfies my query.</i>
Storage	Stores/retrieves data into/from storage server. Resource denotes the place where the data get stored. This includes physical storages, digital libraries, databases, etc.	<i>store my data in storage with at least 10 GB of free space and a rotation speed of at least 7200 rpm.</i>
Network	Offers a network service.	<i>assign me a link where bandwidth <math>\geq 1</math> Mbps and availability <math>&gt; 90\%</math></i>

#### 4.7.1 Resource Specification Language

We need a flexible language that provides the necessary richness to express the diverse kinds of heterogeneous resources managed by the system along with their allocation constraints.

The Grid Information Service (GIS) working group [48] of the Global Grid Forum (GGF) [47] focuses on services that either provide or consume information on the Grid. They have proposed a simple set of objects that can be used to describe computational

resources in the Grid [78]. We follow the specification defined by the Grid Information Service Group, extend it and express it using XML. A resource can be described using the Document Type Definition (DTD) shown in Fig. 27. The different entities are described as the following:

- **Resource:** describes the main entity that contains information about the resources. This information is either given by the vendor or internal to the brokering environment.
  - **CanonicalSystemName:** a string indicating the architecture-manufacturer-operatingSystem, e.g., sparc-sun-solaris2.8.
  - **Manufacturer:** the manufacturer of the computational resource, e.g., Sun Microsystems.
  - **Model:** the model of the computational resource, e.g., sun4u.
  - **SerialNumber:** the serial number of the computational resource.
  - **MachineHardwareName:** the machine hardware name as given out by the vendor.
  - **HostID:** the host id number as given by the vendor.
  - **Type:** the type of the compute resource. This includes one or more out of the following list:
    - **Workstation:** a stand-alone workstation.
    - **PC:** a personal computer.
    - **SIMD:** a Single Instruction stream, Multiple Data stream machine.
    - **MIMD:** a Multiple Instructions stream, Multiple Data stream machine.
    - **SM:** a computational resource using shared memory between multiple nodes.
    - **DM:** a computational resource using distributed memory between multiple nodes.
  - **ResourceID:** the resource id number as given by the brokering environment.
  - **IPAddress:** the IP address of the resource.



- **OperatingSystem:** It contains information about the resources operating system.
  - **Name:** the name of the resource Operating System, e.g., Red Hat Linux.
  - **Version:** version of the Operating System.
  - **Release:** The release version of the Operating System, e.g., 7.2.
  - **Type:** The type of operating system, e.g. POSIX, BSD, etc.
- **Memory:** It contains both highly dynamic and relatively static information about the resources memory.
  - **PhysicalMemorySize:** The total size of the main memory in KB.
  - **FreePhysicalMemory:** The free main memory in KB.
  - **PhysicalMemoryAccessTime** : the average access Time of the main memory in ms.
  - **VirtualMemorySize:** the virtual memory size in KB.
  - **FreeVirtualMemory:** the free virtual memory in KB.
  - **TotalSwapSpace:** the total swap space in KB.
  - **FreeSwapSpace:** the free total swap space in KB.
  - **PageFaultRate:** the page fault rate in term pages/second.
- **Cache:** It contains cache information for the resource.
  - **TotalDataCache:** the total data cache size in K.
  - **TotalInstructionCache:** the total instruction cache size in K.
- **Benchmark:** It contains benchmark information for the resource.
  - **SPECint92:** SPECint92 rating of the machine.
  - **SPECfloat92:** SPECfloat92 rating of the machine.
  - **lapack100:** LAPACK rating of machine for solving a matrix of 100.
  - **lapack500:** LAPACK rating of machine for solving a matrix of 500.
  - **lapack1000:** LAPACK rating of machine for solving a matrix of 1000.
  - **mflops:** Stores MFlop rating of the machine.
- **CPU:** It contains both highly dynamic and relatively static information about the resources processor(s) as well as current load information.
  - **CpuType:** type of computer processor (Pentium, Sparc, RS6000, MIPS, etc.).

```

<!-- Resource.dtd -->
<!ENTITY % PolicyDTD SYSTEM "Policy.dtd">
%PolicyDTD;
<!ENTITY % DisseminatingDTD SYSTEM "Disseminating.dtd">
% DisseminatingDTD;
<!ENTITY % AvailabilityStatus "Available|NoneAvailable">
<!ELEMENT Resource (OperatingSystem, Memory, Cache, Benchmark,CPU,SystemDynamicInfo,Policy?,
Disseminating?)>
<!-- ATTLIST Resource
CanonicalSystemName CDATA #IMPLIED
Manufacturer CDATA #IMPLIED
Model CDATA #IMPLIED
SerialNumber CDATA #IMPLIED
MachineHardwareName CDATA #IMPLIED
HostID CDATA #IMPLIED
Type CDATA #IMPLIED
ResourceID CDATA #IMPLIED
IPaddress CDATA #IMPLIED
-->
<!-- ELEMENT OperatingSystem EMPTY -->
<!-- ATTLIST OperatingSystem
Name CDATA #IMPLIED
Version CDATA #IMPLIED
Release CDATA #IMPLIED
Type CDATA #IMPLIED
-->
<!-- ELEMENT Memory EMPTY -->
<!-- ATTLIST Memory
PhysicalMemorySize CDATA #IMPLIED
FreePhysicalMemory CDATA #IMPLIED
PhysicalMemoryAccessTime CDATA #IMPLIED
VirtualMemorySize CDATA #IMPLIED
FreeVirtualMemory CDATA #IMPLIED
TotalSwapSpace CDATA #IMPLIED
FreeSwapSpace CDATA #IMPLIED
PageFaultRate CDATA #IMPLIED
-->
<!-- ELEMENT Cache EMPTY -->
<!-- ATTLIST Cache
TotalDataCache CDATA #IMPLIED
TotalInstructionCache CDATA #IMPLIED
-->
<!-- ELEMENT Benchmark EMPTY -->
<!-- ATTLIST Benchmark
SPECint92 CDATA #IMPLIED
SPECfloat92 CDATA #IMPLIED
lapack100 CDATA #IMPLIED
lapack500 CDATA #IMPLIED
lapack1000 CDATA #IMPLIED
mflops CDATA #IMPLIED
-->
<!-- ELEMENT CPU EMPTY -->
<!-- ATTLIST CPU
cpuType CDATA #IMPLIED
fpuType CDATA #IMPLIED
Count CDATA #IMPLIED
Speed CDATA #IMPLIED
Load1 CDATA #IMPLIED
Load5 CDATA #IMPLIED
Load15 CDATA #IMPLIED
LoadModified CDATA #IMPLIED
-->
<!-- ELEMENT SystemDynamicInfo EMPTY -->
<!-- ATTLIST SystemDynamicInfo
heartbeat CDATA #IMPLIED
bootTime CDATA #IMPLIED
numberOfInteractiveUsers CDATA #IMPLIED
Status (%AvailabilityStatus);>

```

**Fig. 27. A schema for specifying resources**

- fpuType : type of floating point processor.
- Count: number of CPU's in the compute resource.
- Speed: clock rate of the CPU's in MHz.
- Load1: the load average in the last minute.
- Load5: the load average in the last five minutes.
- Load15: the load average in the last fifteen minutes.
- LoadModified: the time at which the load averages was last modified.
- SystemDynamicInfo
  - Heartbeat: the last time the resource was known to be alive.
  - BootTime: the last time the resource was known to be rebooted.
  - NumberOfInteractiveUsers: The number of the interactive users.
  - Status: The current availability status of the resource.
- Policy: It contains information about the usage policy as described in the Policy Specification Language. A detailed explanation about how a policy can be specified is given in chapter V.

```
<?xml version="1.0"?>
<!DOCTYPE Resource SYSTEM "Resource.dtd">
<Resource CanonicalSystemName="sparc-sun-solaris2.8" Manufacturer=" Sun Microsystems"
Model="sun4u" SerialNumber="11-22-33" MachineHardwareName="" HostID="12345"
Type="Workstation" ResourceID="2" IPaddress="128.82.7.107">
<OperatingSystem Name="Solaris" Version="" Release ="2.8" Type=""/>
<Memory PhysicalMemorySize="512000" FreePhysicalMemory="24000"
PhysicalMemoryAccessTime="" VirtualMemorySize="" FreeVirtualMemory=""
TotalSwapSpace="20000" FreeSwapSpace="15000" PageFaultRate=""/>
<Cache TotalDataCache="" TotalInstructionCache="" />
<Benchmark SPECint92="" SPECfloat92="" lapack100="" lapack500="" lapack1000=""
mflops=""/>
<CPU cpuType="Sparc" fpuType="" Count="1" Speed="750" Load1="" Load5="" Load15=""
LoadModified="1019999999"/>
<SystemDynamicInfo heartbeat="1019999999" bootTime="1010000000"
numberOfInteractiveUsers="5" Status="Available"/>
<Disseminating>
<Push> <Periodic Interval="60"> </Push>
</Disseminating>
</Resource>
```

Fig. 28. An example script of a resource using the resource specification language

- **Dissemination:** When a resource registers with PROBE, its status needs to be updated regularly based on the disseminating option. This entity describes the dissemination option being used in monitoring the resource. More about resource monitoring is given in 3.6.3.1.

An example script representing a Solaris workstation is given in. Fig. 28. Also, as illustrated in Fig. 29, the current implementation of PROBE uses MySQL in the underlying implementation of the repository. We store the XML specification of the resources in the object-relational form and use the *Resource Parser* to write and retrieve resource information to/from the *Resource Repository*.

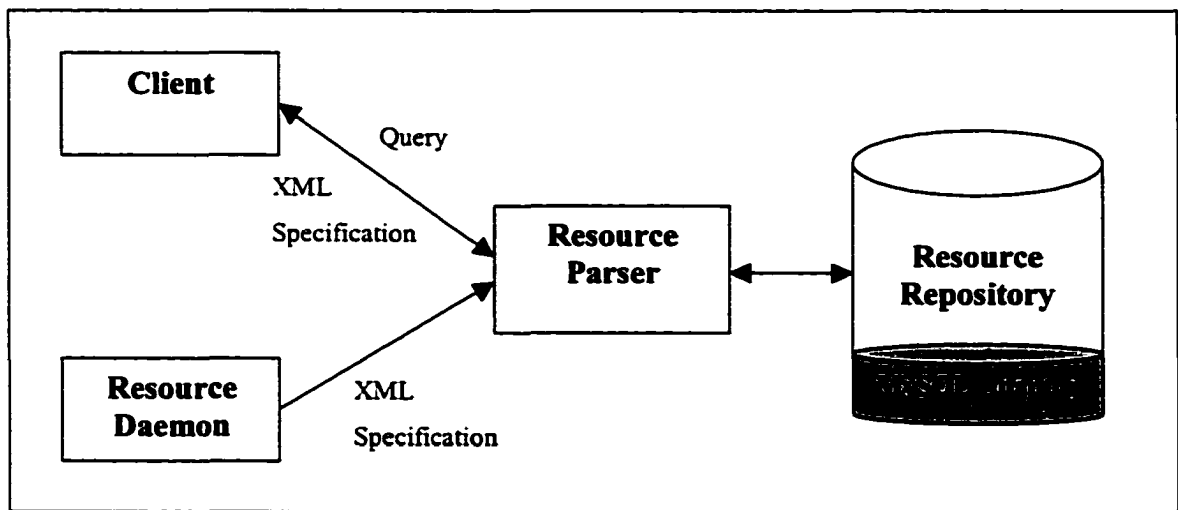


Fig. 29. Using the Resource Parser to write and retrieve resources information to/from the Resource Repository

## 4.8 Issues

### 4.8.1 Rescheduling

Rescheduling is one of the important issues that has not received enough attention from most existing resource brokering efforts. PROBE supports rescheduling in various ways. Jobs that are aborted due to resource or job failure are kept in an internal queue within

*Scheduler Agent* that in turns uses an underlying queuing algorithm to select the next job to schedule. Sometimes, due to poor performance, load imbalance, optimization issues, etc, the resource brokering environment has to adjust the current schedule. PROBE supports such a dynamic scheduling in which the current schedule can be re-examined and the job executions reordered.

However, our rescheduling approach does not support process migration since it requires process persistence where the resource brokering environment needs to save the execution state of the process (variables, stack, and possibly even the point of execution). Condor [80] migrates the whole process through checkpoints. However, to allow checkpointing, object code of the application must be re-linked with the Condor augmented system library. This adds more limitation on types of process that can be migrated. For example multi-process jobs cannot be migrated and inter-process communications such as pipes, semaphores and shared memory are not allowed [81].

#### **4.8.2 Allocation Assurance**

In a typical grid system, resources are designed to work as stand-alone units rather than being dedicated to the system. Management and control of such a system is tedious and challenging issue. Allocation assurance is another issue that has not been addressed by most current resource brokering efforts. An allocation needs to satisfy the job's requirements during the lifetime of the allocation. The performance of the client's allocated task should not suffer after the allocation is made. For example, let us say that the client asks for a resource where Free Physical Memory has to remain greater than 256 MB, then suddenly another allocated task or a local user's task competes in using the resource which results in affecting the level of allocation the client has requested.

Most existing efforts focus on resolving this issue by making some assumptions that might restrict the usage of the underlying grid system. For example, in [105], all resources are assumed to be dedicated and their loads are predictable, and tasks are assumed to be profiled where resource usage can be estimated in advance. We believe

such restrictions do not encourage either the resource provider or the resource consumer to use the underlying grid.

Our brokering infrastructure is flexible enough where the user can plug-in any kind of scheduling algorithms that can help in resolving fairness issues before they occur. In the following chapter, we describe how the policy-based framework helps in improving fairness and providing some confidence to the user to use the underlying grid environment by assuring the guaranteed level of allocation.

#### **4.9 Summary**

In this chapter, we focused on the design and implementation of the *Resource Broker*, the core component of PROBE that accepts clients' tasks and schedules them accordingly. We described a flexible and extensible XML-based schema that clients can use to describe their application problems.

We showed how the design of our brokering infrastructure is flexible and how the layered façade design approach makes it easy to plug-in application types and scheduling techniques. However, allocation assurance is one of the major issues that most existing resource brokering efforts ignore. In the next chapter, we focus more on a policy-based framework that helps in resolving this issue.

## CHAPTER V

### POLICY-BASED FRAMEWORK FOR RESOURCE BROKERING

#### 5.1 Overview

A typical grid environment has a distributed heterogeneous collection of shared resources controlled by different administrative domains. In general, the resource provider wants to control the utilization of its resources. This can be done via a resource-specific policy. In the same manner, the resource consumer wants to specify its application requirements. The rights of both the provider and the consumer need to be respected.

Some resource brokering environments are system-centric, allowing only resource providers to specify their policies; others are application-centric, allowing only clients to specify their policies. Moreover, allocation assurance is one of the major issues, which has not been addressed by most current resource brokering efforts. PROBE's approach allows both clients and resource providers to specify their policies. The *Policy Enforcement Manager* enforces these policies. In particular, the selection of the resources takes into account both the client's requirements and the resource constraints.

In this chapter, we focus on our policy framework. We begin this chapter by explaining our philosophy and outlining the design goals. Then, we describe in detail the architecture that we chose in order to implement our policy framework, the different approaches and their tradeoffs.

#### 5.2 Philosophy

The network community has been utilizing policy-based frameworks in order to guarantee a given level of Quality of Service (QoS) [20],[100],[118]. In such frameworks, a Service Level Agreement (SLA) is defined as a formal negotiated agreement between two parties, the service provider and the service consumer. Each SLA

comprises one or more policies. A policy can be seen as a set of conditions and actions that need to be taken when those conditions are met.

One result of enabling SLA on grid systems is that it provides one means of attracting grid users and contributes to establishing credibility to existing grid environments. It does so by committing to provide the guaranteed level of allocation with the right action (compensation, credit, configuration, etc) if such guarantees are not met or are approaching violation. This will help in encouraging high performance users to use grid systems as they make a commitment to provide the guaranteed level of allocation.

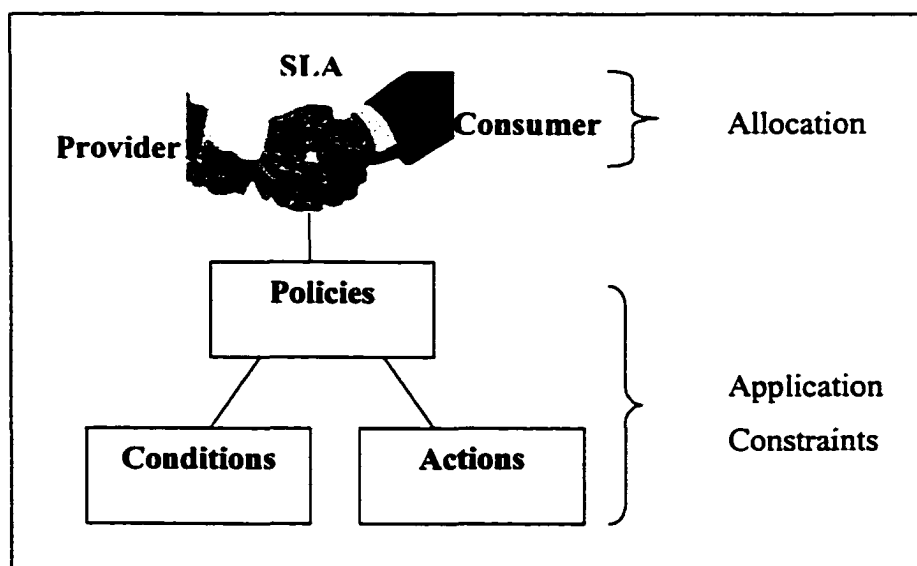


Fig. 30. PROBE's vision of the allocation process

As illustrated in Fig. 30, PROBE looks at the allocation process as a Service Level Agreement (SLA) between the resource consumer and the resource provider. PROBE goes far beyond the typical matching/allocation process to provide allocation assurance by providing the means of policies and SLAs and ensuring that the appropriate action will be taken in case of a violation.

In order to provide a common understanding about allocation quality and responsibilities, PROBE creates a Service Level Agreement (SLA) that can be viewed as a contract between the resource provider and the resource consumer. At the time of a



job's allocation, the *Resource Broker* notifies the *Policy Enforcement Manager* so that it can create an SLA based on the client's terms. The *Policy Enforcement Manager* keeps monitoring this SLA during the life-time of the allocation and takes appropriate action(s) (as specified in the policy) when a violation occurs. The *Policy Enforcement Manager* interacts with the *Resource Monitor* to get up-to-date information, such as the status of the resources, and the policy related information. The API is flexible enough to let the *Policy Enforcement Manager* talk to an external source of information such as Globus's MDS. External alert systems could also be notified when a violation occurs.

Resource providers could also specify some local policies internal to their resources to ensure that the appropriate action will be taken before a violation occurs.

### 5.3 Design Goals

The key design goals of our policy framework are:

- *Flexible architecture*: It must be flexible and general so that it can incorporate existing brokering requirements as well as evolve to meet future needs. To address this goal, we have divided our policy framework into a set of flexible and extensible components and used a layered approach and façade design pattern where future needs can be incorporated. The architecture of the framework is given in section 5.4.
- *Scalability*: As the underlying grid environment continues to grow, the *Policy Enforcement Manager* is expected to handle massive number of clients, resources and their associated SLAs. The architecture has to be scalable to handle this issue. Modularity, distribution and caching, as we explain later, help us build a scalable policy-based framework that can process a large number of concurrent client requests and manage large number of distributed heterogeneous resources. We achieve distribution at different levels. The *Policy Enforcement Manager*, as part of PROBE, can be replicated and distributed; the components of the *Policy Enforcement Manager* can be replicated and distributed; and policy parsing is distributed across resources where each resource has its own local policy enforcer.

- *Efficient matching*: The *Policy Enforcement Manager* caches all the minimal policy related information that it needs for resource matching and SLA monitoring. For efficient retrieval of the cached data, we index the data using a *HashMap* where objects can be retrieved using an  $O(1)$  algorithm. Also, we apply some optimization techniques where policies are parsed and optimized locally at their associated resources, unnecessary parsing is avoided, and unavailable resources are excluded from the matching process. This minimizes the effort needed by the *Policy Enforcement Manager*. We explain caching in 5.5 and optimization techniques in 5.8.
- *Powerful Specification Language*: We need a very flexible and extensible language that can handle the requirements of both the resource provider and the resource consumer. To address this issue, we have designed a flexible and extensible Policy Scripting Language (PSL) using XML. PSL is described in 5.6.

As we explain in this chapter, our design is driven by these goals.

## 5.4 Architecture

As shown in Fig. 31, we have divided the *Policy Enforcement Manager* into seven components, where each component implements an individual function. These components interact with each other to achieve the overall functionality of the *Policy Enforcement Manager*. Below, we give an outline of these components:

- **Policy Keeper**: the main component that maintains the internal cache of the *Policy Enforcement Manager*. It provides an interface where objects in the cache can be put into, removed or retrieved from the cache very effectively. The data is indexed using a *HashMap* for efficient retrieval.
- **Policy Parser**: the parsing engine. Both the *Policy Matcher* and the *SLA Monitoring Agent* use this component to evaluate the policies at hand. The *Expression Builder* and *External Evaluator* provide more flexibility and extensibility to the parsing engine where plug-ins can be easily added.

- **Expression Builder:** This module builds expressions based on existing entities, external entities or previously defined expressions. For example, Memory Utilization could be defined as  $((\text{PhysicalMem} - \text{FreePhysicalMem}) / \text{PhysicalMem}) * 100$ .

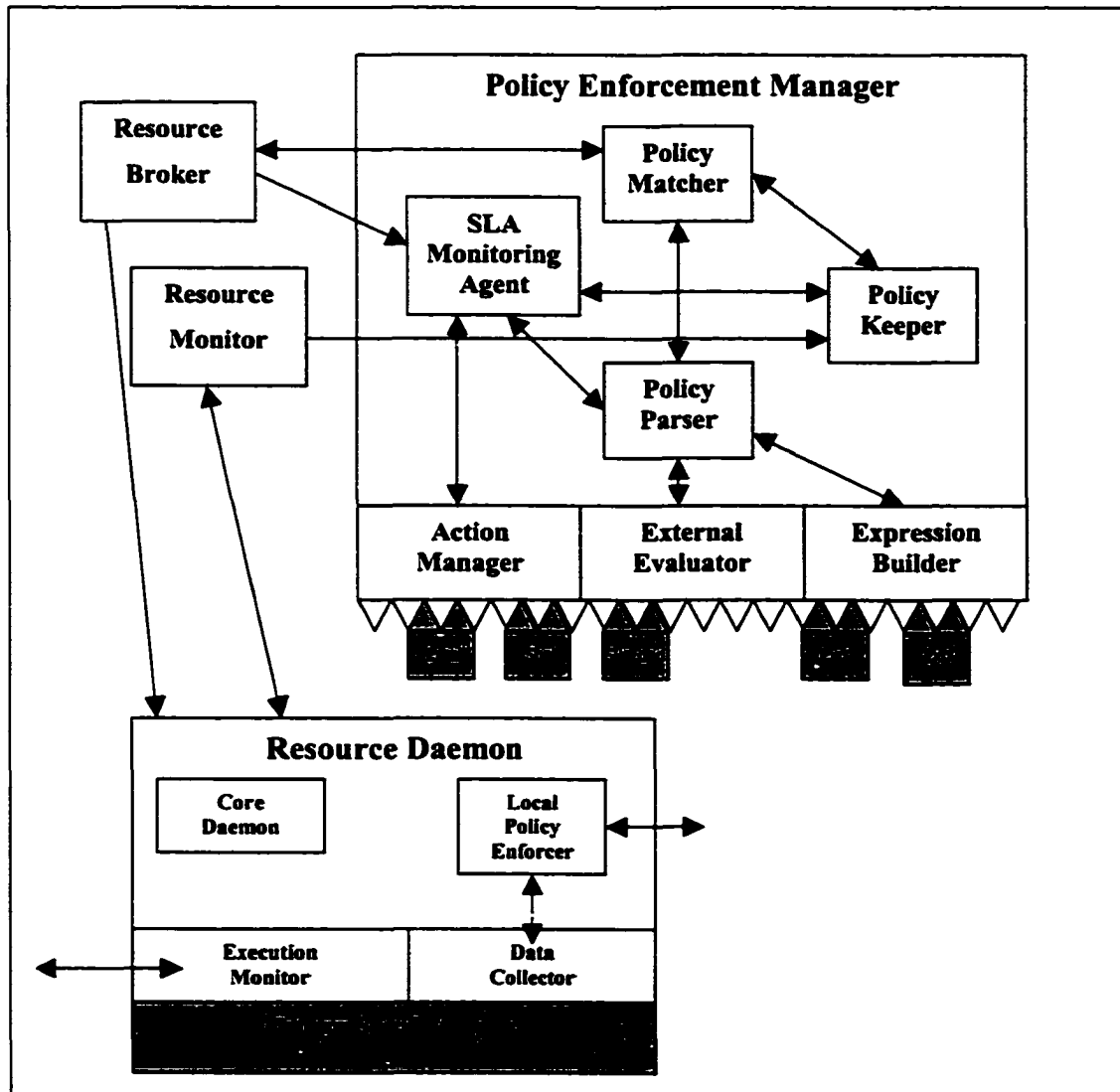


Fig. 31. Overall Architecture of the Policy Enforcement Manager

- **External Evaluator:** This module evaluates the external entities. Entities like time, environmental variables, PROBE variables (system load, etc), system-

specific variables, etc can be easily evaluated if their plug-ins are available. The system could have a dynamic variable that the user could manipulate to affect the brokering process.

- **Policy Matcher:** This module matches the client's policy and the resources' policies. A subset of matched resources is constructed and passed to the *Resource Broker* that then constructs the appropriate schedule and starts the allocation process.
- **SLA Monitoring Agent:** This module is responsible for assuring the allocation. Once the job is allocated, an SLA is created with the client's policy. The *SLA Monitoring Agent* continues monitoring the associated policies and takes the appropriate action (if any) in case of violations.
- **Action Manager:** A policy might have action(s) associated with it that need to be triggered in case of a violation. An action can be anything that the associated *Action Processor* can handle. For example, we could have a *Logging Action Processor* whose only function is to log a message that a specific SLA has been violated. Another possible handler could trigger an event to some external system (e.g., accounting) that then takes the appropriate action (e.g., crediting the client's account).

Our policy framework is distributed. Within each resource daemon, we have a *Local Policy Enforcer* that manages the policies internal to the resource, and optimizes the SLA's policies prior to updating the *Policy Enforcement Manager*. As shown in Fig. 32, we have divided the *Local Policy Enforcer* into five components. Those components are: *Policy Monitor*, *Policy Parser*, *Expression Builder*, *External Handler* and *Action Manager*. Except for the *Policy Monitor*, the other components are identical to that of the *Policy Enforcement Manager*.

A resource might have two kinds of policies: allocation policies that define how the resource can be used, and internal policies that are meant for internal use within the resource. The *Policy Monitor* monitors and optimizes the local and internal policies. In case of a violation, the *Policy Monitor* triggers the associated action(s), if any.

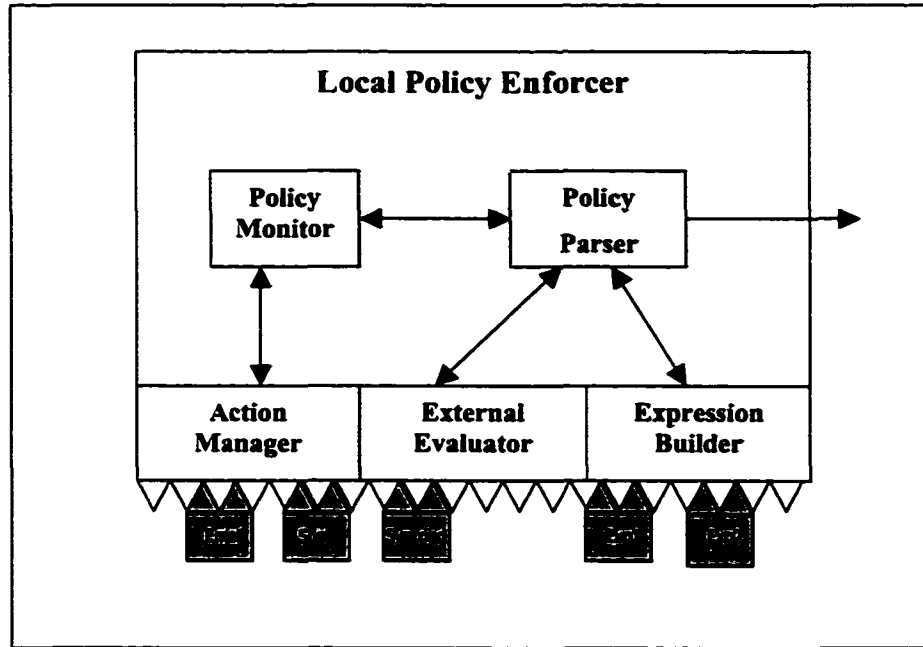


Fig. 32. Architecture of the Local Policy Enforcer

## 5.5 Caching

As the underlying grid environment continues to grow, the *Policy Enforcement Manager* is expected to handle a large numbers of clients, resources and their associated SLAs. To achieve a high level of scalability and performance, the *Policy Enforcement Manager* caches a minimal set of policy related information that it needs for resource matching and SLA monitoring.

The *Policy Keeper* is the component that maintains the cached information about all the SLAs available in the system and their associated policies and actions. Caching helps in achieving near real-time performance while matching resources or monitoring their associated SLAs. Internal cache reduces the cost of loading the data from the *Resource Repository* for each request. For efficient retrieval of the cached data we use a *HashMap*, a very fast data structure where indexed objects can be retrieved using an  $O(1)$  algorithm.

In order to tackle consistency issues, the *Resource Monitor* feeds the *Policy Keeper* with the up-to-date status of the resources making sure that the internal cache is consistent with the system information. To address concurrency, we apply some form of synchronization for both read and write operations. Java does not have a ready solution where concurrency can be handled efficiently. It provides object synchronization where the Java runtime ensures that only one thread can access the synchronized object at a time. This is not efficient since it allows one read operation at a time. We have implemented some wrapper applications where read and write lock can be handled properly.

One drawback of caching, in general, is that one has to pay the price of expensive use of memory. However, the cost is very small compared to the gained performance. In 7.4.2, we analyze the performance of caching.

Another issue is how to recover the cached data when a failure happens and the component restarts. Different recovery mechanisms could be applied. For example, data could be serialized to permanent storage or reloaded from the *Resource Repository*. In the current prototype, we store the information in the underlying *Resource Repository*. When the *Policy Enforcement Manager* is restarted, the state is able to be restored. In 8.3, we describe an extension of PROBE where we propose a new module that handles failures and recovery issues. A detailed discussion of failure/recovery issues is outside the scope of this thesis.

## 5.6 Policy Specification Language

The *Policy Enforcement Manager* needs a flexible and extensible language that can handle the requirements of both the resource provider and the resource consumer. To address this issue, we have designed a Policy Specification Language (PSL) using XML. In this subsection, we discuss PSL in more detail. We begin by explaining the syntax of PSL, and then we present its XML representations. We conclude the section by presenting some examples that demonstrate the use of PSL.

### 5.6.1 Syntax

A policy is a set of conditions and associated actions that are triggered when these conditions are met. The policy script should have the flexibility to express both the conditions and the actions.

We look at the condition as an expression built based on *basic conditions* (entity, comparison operator, threshold value) and *logical operators* (AND, OR, NOT). The policy script is the one that determines how the policy can be evaluated based on the scripting language that we describe in this section. Initially, we support the following items in the policy script:

- **Basic Condition.** This represents the condition that needs to be evaluated either at the time of matching the resource or monitoring the associated SLAs. A basic condition is in the following form:

*[Basic Entity] [Comparison operator] [Threshold Value]*

A Basic Entity can be:

- Resource related entity such as Load, CPU speed, Free Physical Memory, etc. Resource related entity takes the prefix of “*res*”.
- Job related entity such as user, priority, etc, that takes the prefix of “*job*”.
- External entity that needs to be evaluated by the *External Evaluator* such as time, some sort of environmental variable, etc. External entity takes the prefix of “*ext*”.
- Expression that needs to be calculated with the help of the *Expression Builder*. Expression takes the prefix of “*exp*”

Comparison operators are:

- Less than
- Less than or equal.
- Greater than.

- Greater than or equal.
- Equal.
- Not equal.

The threshold value is a constant or another basic entity. Applicable value types are: *Float*, *String* and *Error*.

- **Logical operators.** Logical operators (such as AND, OR, NOT) are supported for aggregating conditions.

Actions are triggered when some policy conditions are met. The policy script supports actions where one or more action(s) can be specified in case of violations. Each action has a type specifying its *Action Processor* and a set of parameters (name-value pairs) specifying the behavior of the *Action Processor* when the action is triggered. A detailed explanation of actions and how they are being handled is given later in this chapter.

## 5.6.2 XML representation of PSL

We have noticed the need of having a flexible language that provides the necessary richness to express the diverse kinds of policies that both resource providers and consumers can have. We have designed and implemented a very flexible XML-based Policy Scripting Language (PSL), which can be used for specifying the policies of both providers and consumers. Fig. 33 shows the schema for specifying the policies and restrictions.

In order to overcome the overhead of XML parsing and to minimize the memory requirement, we parse the policy script once, extract the necessary information and convert the condition into the infix notation where the *Policy Parser* can easily parse it.



```

<!--Policy.dtd-->
<!ENTITY % operator "AND|OR|NOT">
<!ENTITY % comparison "EQ|NEQ|GR|GREQ|LS|LSEQ">
<!ELEMENT Policy (Rule,Action*)>
<!ELEMENT Rule ((Condition)|(%operator;))>
<!ELEMENT AND ((Condition)|(%operator;))*>
<!ELEMENT OR ((Condition)|(%operator;))*>
<!ELEMENT NOT ((Condition)|(%operator;))*>
<!ELEMENT Condition EMPTY>
<!--ATTLIST Condition
      Entity CDATA
      Operator (%comparison;)
      Value CDATA >
<!--ELEMENT Action (AdditionalInfo*)>
<!--ATTLIST Action
      Type          CDATA #IMPLIED>
<!--ELEMENT AdditionalInfo EMPTY>
<!--ATTLIST AdditionalInfo
      Name          CDATA #IMPLIED
      Value         CDATA #IMPLIED

```

Fig. 33. Schema for the Policy Scripting Language

### 5.6.3 Examples

In this subsection, we present some examples that demonstrate the use of our Policy Scripting Language to express policies for both the client and the resource.

Fig. 34 illustrates an example of a resource policy script that could be part of a resource specification. In this policy, the resource provider wants the resource to be allocated only when the Free Physical Memory is less than 128 MB or the Free Swap Space is less than 10 GB.

```

<?xml version="1.0"?>
<!DOCTYPE Resource SYSTEM "Resource.dtd">
<Policy>
<Rule>
<OR>
<Condition Entity="res.FreePhysicalMem" Operator="LS" Value="128000"></Condition>
<Condition Entity="res.FreeSwapSpace" Operator="LS" Value="10000000"></Condition>
</OR>
</Rule>
</Policy>

```

Fig. 34. Example PSL script describing a resource policy

On the other hand, in Fig. 35 we present a sample client's request where the client is looking for a resource with an available physical memory that is greater than 512 MB and the Free Swap Space is greater than 20 GB. The client wants an e-mail to be sent to the given e-mail address in case of a violation.

```
<?xml version="1.0"?>
<!DOCTYPE Request SYSTEM "Request.dtd">
<Request>
  <Single Name="Initialization" Arguments="" Executable="/home/theneyan/Demo/App1/initial.csh" RunDirectory="/home/theneyan/Demo/App1">
    <Policy>
      <Rule>
        <AND>
          <Condition Entity="res.FreePhysicalMem" Operator="GR" Value="512000"></Condition>
          <Condition Entity="res.FreeSwapSpace" Operator="GR" Value="20000000"></Condition>
        </AND>
      </Rule>
      <Action Type="Email">
        <AdditionalInfo Name="To" Value="theneyan@cs.odu.edu"></AdditionalInfo>
        <AdditionalInfo Name="Subject" Value="Violation"></AdditionalInfo>
        <AdditionalInfo Name="Body" Value="Your Policy has been violated"></AdditionalInfo>
      </Action>
    </Policy>
  </Single>
</Request>
```

Fig. 35. Example PSL script describing a client policy

## 5.7 Policy Parsing

Internally, the *Policy Enforcement Manager* caches the minimal set of information that allows it to answer all kinds of questions that arise while parsing policies. Basically, these questions reveal the values of resource related entities (*FreePhysicalMem*, *CPUload*, etc), job related entities (*user*, *priority*, etc), expressions (*resource utilization*, etc) or external entities (*time*, *environmental variables*, etc). Policies are parsed and optimized locally at their associated resources as we explain in the next section. Also, the XML representation is parsed once and a string representing the condition portion in the infix notation is saved in the internal cache. This minimizes the effort needed by the *Policy Enforcement Manager*.

The *Policy Parser* is the parsing engine that is used by both the *Policy Matcher* and the *SLA Monitoring Agent* to evaluate the policies at hand. Upon request, the *Policy*

*Parser* parses the given policy. It relies on the cached information, *Expression Builder* and *External Evaluator* to evaluate the entities included in the policy script.

## 5.8 Policy Optimization

One of the main goals of our policy framework is to effectively parse policies, mainly at the time of monitoring SLAs that is expected to happen regularly during the lifetime of the allocation.

To optimize the performance of the *Policy Enforcement Manager*, the *Local Policy Enforcer* at each resource optimizes the associated policies and returns the optimized policy scripts along with the resource statistics when updating the resource status. Optimizations are done at several levels:

- **Logical operators are short-circuited.** A short-circuit operator does not evaluate its second operand if the evaluation of its first operand alone would determine the result. C++ and Java use short-circuit evaluation for the *Boolean* operators *AND* and *OR*. The parsing engine supports short-circuit for logical operators. For the *AND* operator, if either operand is *false*, the operator returns *false*, thus the parsing engine stops if the first operand is evaluated to be *false* and the second operand is not evaluated. For the *OR* operator, if either operand is *true*, the operator returns *true*, thus the parsing engine stops if the first operand is evaluated to be *true* and the second operand is not evaluated. Examples are given below:
  - **AND example:** (*false*) AND ( (*Free Physical Memory* < 512000) OR ( *Free Swap Space* < 10000000) ). The parsing engine returns *false* before parsing the second operand.
  - **OR Example:** (*true*) OR ( (*Free Physical Memory* > 512000) OR ( *Free Swap Space* > 10000000) ). The parsing engine returns *true* before parsing the second operand.
- **Avoid Multiple Parsing.** Our policy framework avoids parsing entities that have been already parsed at the resource level. Instead of parsing the same entities

multiple times, it can be done once and the parsed value embedded in the updated script. Let us say we have a resource policy as the following:

$(\text{user}=\text{"theneyan"}) \text{ AND } ( \text{Free Physical Memory} > 512000) \text{ OR } ( \text{Free Swap Space} > 10000000) )$

Let us say that the Free Physical Memory was 518000 kilobytes (KB) and the Free Swap Space was 15000000 KB. The *Local Policy Enforcer* could come up with:

$(\text{user}=\text{"theneyan"}) \text{ AND } (\text{true})$

When the *Policy Enforcement Manager* needs to match that resource, it uses the optimized script, so that it does not need to evaluate the same entities again.

- **Excluding non-available resources.** Using the above optimization techniques, a resource whose policy evaluated to be *false* is excluded from the matching process since there will be no point for the *Policy Matcher* to consider the resource at its current status since it is not going to match with any request.

## 5.9 Actions

As stated before, actions are associated with policy conditions and are triggered when the conditions are met. When the guaranteed level of allocation is not met, the appropriate action(s) need to be taken as specified in the policy script.

When an action is created, it gets assigned an action type specifying its *Action Processor* and a set of parameters. Each parameter is a name-value pair specifying the behavior of the *Action Processor* when the action is triggered. When a policy is violated, the *SLA Monitoring Agent* notifies the *Action Manager* so that it can trigger the corresponding action as illustrated in Fig. 36.

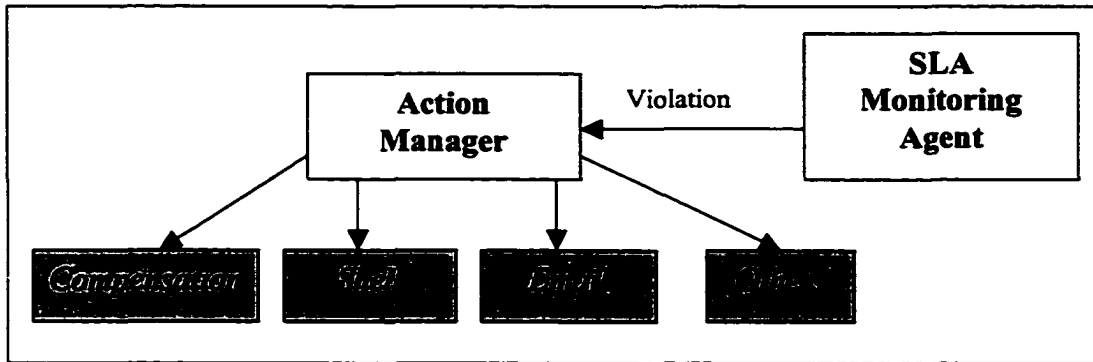


Fig. 36. Action Flow

Depending on the terms of the SLA, the violation in the guaranteed level of allocation may result in variety of actions; this may include *Compensation* where a credit could be issued to the client, *Shell* where a designated shell script can be executed and predefined arguments can be passed or *Email* where a detailed email regarding the violation can be sent via email. Each action is handled by what we call an *Action Processor*.

An *Action Processor* can handle many actions of different action types. Initially, we support the *Email* and the *Shell* action processors. New *Action Processors* can be easily added as needed. The *Action Manager* caches references to all the existing *Action Processors* and has an API where the new ones could be added on the fly.

```

<?xml version="1.0"?>
<!DOCTYPE Policy SYSTEM "Policy.dtd">
<Policy>
<Rule>
<AND>
<Condition Entity="res.FreePhysicalMem" Operator="GR" Value="512000"></Condition>
<Condition Entity="res.FreeSwapSpace" Operator="GR" Value="20000000"></Condition>
</AND>
</Rule>
<Action Type="Compensation">
<AdditionalInfo Name="Customer" Value="$job.user"></AdditionalInfo>
<AdditionalInfo Name="Credit" Value="10"></AdditionalInfo>
<AdditionalInfo Name="FreePhysicalMem" Value="$res.FreePhysicalMem"></AdditionalInfo>
<AdditionalInfo Name="FreeSwapSpace" Value="$res.FreeSwapSpace "></AdditionalInfo>
</Action>
</Policy>
  
```

Fig. 37. Example of a dynamic replaceable parameter

The system also supports dynamic parameters whose values can change on the fly. A dynamically replaceable parameter could be any basic entity preceded by the dollar sign “\$”. This gives the client the necessary power to track down violations as they occur. Fig. 37 illustrates an example of policy with some dynamic replaceable parameters.

### **5.10 Summary**

In this chapter, we explained our policy framework in greater detail. The policy-based approach provides one means of attracting grid users and contributes to establishing credibility to existing grid environments by committing to provide the guaranteed level of allocation with the right action (compensation, credit, etc) if such guarantees are not met. We believe that such a policy-based framework can help in encouraging high performance users to use grid systems as it makes a commitment to assure the guaranteed level of allocation.

## CHAPTER VI

### IMPLEMENTATION

This chapter focuses on the implementation of the current prototype of PROBE. We describe the tools and environments that we have used to implement the current prototype. The PROBE infrastructure has been implemented using Jini technology. One issue with Jini is that it cannot be used across networks that do not support multicasting. We detail an enhancement for Jini in order to enable it across networks that do not support multicasting. We also describe a variety of client interfaces and helper utilities that we have developed to demonstrate the use of PROBE. We end this chapter by focusing on overview of the whole package.

#### 6.1 Environment

The current prototype implementation of PROBE is based on the following:

- **Programming Language: Java**

The underlying technology we use in implementing PROBE is Java [50]. Besides being simple, safe, object-oriented, robust, and tightly integrated with the World Wide Web technologies, Java is a portable and platform-independent language enabling the resulting prototype implementation to run on any operating system platform with an implementation of the Java Virtual Machine (JVM). The current prototype uses Java 2 SDK, Standard Edition, version 1.4.1 that can be obtained from: <http://java.sun.com/j2se/1.4/>.

- **Distributed Computing Technology: Jini**

The distributed nature of Jini allows us to create very scalable systems that inherit all of the intrinsic benefits that Jini has to offer. The major advantages that Jini

has over other distributed computing technologies are the semantics and mechanisms that help with dealing with network and hardware failures and permit the silent addition and removal of resources with their services on a network. Also, as Jini is layered on top of Java RMI, it can support mobile code, making it possible to transport not only object state but also object implementation across networks. This feature helps us in applying the plug-and-play feature of PROBE in an effective manner. Jini technology is explained in chapter II in great detail. The current prototype uses Jini reference implementation version 1.1 that can be obtained from: <http://www.sun.com/software/jini/>.

- **Repository Infrastructure: MySQL**

The current repository adaptors that we have implemented for both the *Resource Repository* and the *Job Repository* are RDBMS-based ones. We store the XML specification of the resources and job's information in the object-relational form where the data can be easily updated, queried and reformatted as needed using SQL. The relational model has several advantages since it enables complex queries to span and aggregate many resources. It also leverages sophisticated and scalable database technologies.

MySQL [88] is the most widely used open source database management system. It is light-weight and considered to be one of the fastest, most stable and most secure databases ever developed. In short, MySQL is very fast, secure, reliable, and easy to use. The repositories adaptors that we have used through the current implementation of PROBE use MySQL version 3.23 as their background infrastructure. MySQL can be obtained from: <http://www.mysql.com/>.

- **XML Parser: JAXP**

We use the Java API for XML Processing (JAXP) [70] to parse all the XML documents. We have implemented several user-friendly parsers to parse resources, requests and their associated policies. The API is flexible enough to be



utilized by external applications. The current implementation uses the JAXP 1.2 reference implementation. It can be obtained from: <http://java.sun.com/xml/jaxp/>.

- **XML Editor: xmloperator**

The xmloperator is an open source, free software that can be used to edit XML documents. It is written in Java where it can run on any machine that supports the Java platform. PROBE supports a Graphical User Interface (GUT) where clients can submit requests and monitor and view current state of resources and requests. We have integrated the GUI of PROBE with the xmloperator XML editor release 1.11. The xmloperator can be obtained from: <http://www.xmloperator.net/>.

## **6.2 Enhancing Jini for Use Across Non-Multicastable Networks**

Jini's internal protocols rely on multicasting for discovering and joining lookup services. This becomes an issue when deploying Jini across non-multicastable networks. Some routers on the Internet do not support routing of multicast packets for a variety of reasons. Also, some organizations are not willing to open their firewalls to multicast so as to avoid security problems. Similarly, a local area network divided into subnets may disable multicast traffic across the subnets to avoid unnecessary traffic that may result in performance degradation. This blocking of multicast traffic across subnets prohibits the use of Jini in such an environment.

One method for working around this problem is to use a tunneling mechanism where the multicast traffic is encapsulated in a unicast packet and is then transferred through unicast routers and non-collaborative firewalls. This method has been used in several projects. For example, *MRoutd* has been used to achieve tunneling in the *Mbone* [104]. However, there are many problems in the approach taken by the *MRoutd* implementation, such as the lack of platform independence, wastage of available bandwidth due to the transfer of a large amount of control information and the fact that it forwards all the multicast traffic interfaces. Other projects, such as *mTunnel* [95],[96] and *liveGate* [83], were designed to overcome some of these problems, however there are several reasons

for building our own tunneling mechanism and not using some of those existing ones. Having decided to use Jini, we would like to take advantage of the open source code of Jini and embed our mechanism within Jini. Using a pure Jini approach allows us to leverage the capabilities of Jini while activating tunneling in the background without the aid of any member of the federation. Also, unlike other tunneling mechanisms, in our environment, we do not need to tunnel some of the control information such as the multicast address group and port to which the message is supposed to be delivered. This is because in the context of Jini, our needs are very specific: we need to tunnel only the multicast request and announcement messages that have predetermined multicast endpoints. Providing the right proxies, as explained in the next section, can easily satisfy these requirements.

To solve this problem, we have enhanced Jini in order to support systems like PROBE that need to work with resources in different domains. In particular, we have introduced a lightweight service called the Tunneling Service (TS) for tunneling multicast messages across subnets. Our approach, as illustrated in Fig. 38, involves establishing a tunneling service end point, TS, at each subnet. Each TS provides a window between its subnet and the rest of the world. The TSs are implemented as Jini services and thus have to register with a known Global Tunneling Lookup Service (GTLS) dedicated for maintaining the list of TSs in the environment. The GTLS is implemented as a lookup service that can be started at any subnet of the federation. TSs will collaborate with each other in order to tunnel all the multicast messages across subnets that do not support multicast..

Given such an architecture, the scenario is as follows. Each TS establishes the appropriate multicast endpoints and listen for incoming multicast requests and announcements from within its subnet and will then tunnel the messages out to all the other TSs. Also, each TS is going to listen for incoming tunneled multicast requests and announcements from other subnets and will multicast them locally. Any connection that needs to be setup between the clients, services and the lookup services directly uses the unicast protocol even if it has to cross subnet boundaries. The TSs are not involved in this phase of the interaction.

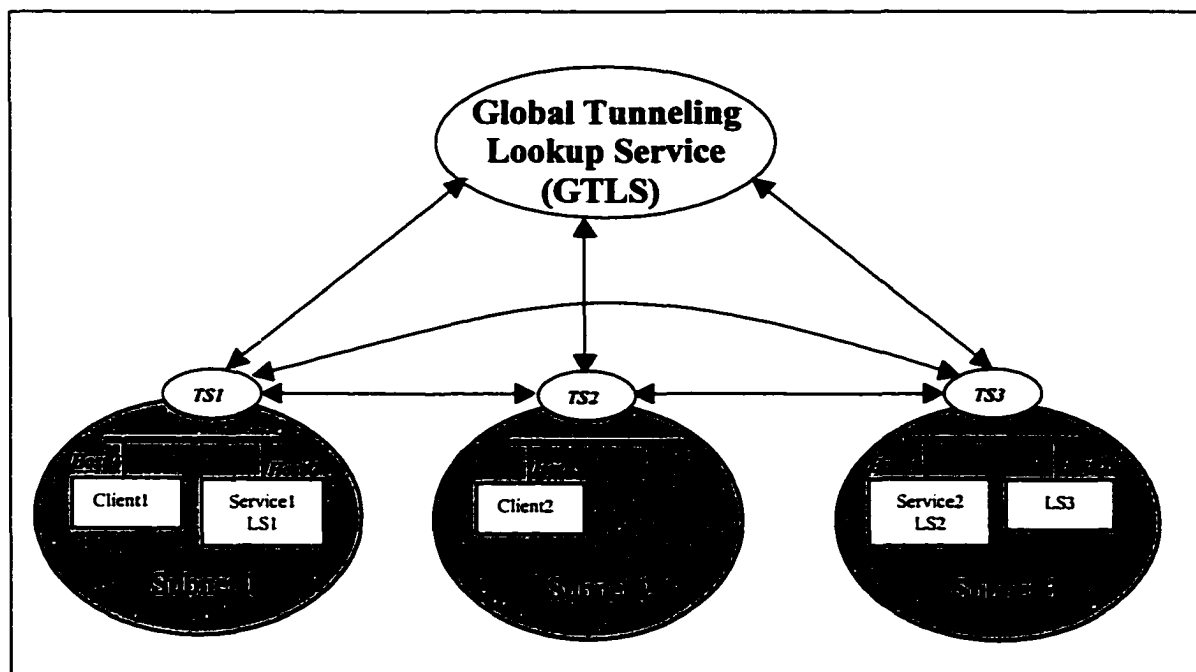


Fig. 38. Different non-multicastable subnets connected by the Tunneling Service (TS)

The underlying aim of our implementation is to make enhancements to Jini that are compatible with the Jini functionality. Thus, we would like the tunneling service to be active in the background without making any changes as far as possible to the behavior of the clients, services and the lookup services. Also, we would like the implementation to work without any modification even if the underlying network supports multicasting and the tunneling service is not required. In the next few subsections we describe the implementation of the Global Tunneling Lookup Service and the Tunneling Service.

### 6.2.1 Global Tunneling Lookup Service (GTLS)

In order for the system to work properly, each of the TSs needs to know about all the other TSs in the environment. Thus, we need a central repository that keeps track of all the currently active TSs. Jini provides the functionality required for just such a repository. Hence, we implemented this repository as a lookup service called the Global

Tunneling Lookup Service. Using the distributed events interface of Jini, every TS can be notified when a new TS joins or leaves the system. In our implementation, since each TS relies on the unicast discovery protocol in all its interactions with the GTLS, it needs to know the IP address and the port where the GTLS is running.

### 6.2.2 Tunneling Service (TS)

The Tunneling Service is the central concept in our solution. This service can be patched into the runtime infrastructure of Jini as a new service just like any other standalone service. A TS has to be started on each subnet that is taking part in the larger system. The system administrator can do this. On the other hand, if suitably modified, the first Jini client, service or LS to start in a subnet could check to see if a TS is already running in the subnet. If not, it can start one. The tunneling service consists of four major parts: the *core tunneling* subsystem which is published at the GTLS as a proxy; the *listener* which keeps track of local multicast requests and announcements and uses other TSs' proxies for tunneling messages; the *notifier* which keeps track of all the other active TSs; and the *wrapper* which implements the infrastructure necessary for the TS to be a Jini service.

**The Core Tunneling Subsystem:** The core tunneling subsystem is the proxy to the service that is posted with the GTLS by the wrapper. The TSs need to contact the GTLS and download each other's proxies in order to achieve tunneling amongst them. The proxy consists of two methods: one for the incoming tunneled request messages and the other for the incoming tunneled announcement messages. Incoming tunneled requests from other TSs are multicast across the local subnet so that the local LSs can respond appropriately. Similarly, incoming tunneled announcements from other TSs are multicast for the discovering entities in the local subnet.

**The Listener:** This is the part of the service that is in charge of catching the necessary multicast traffic, the multicast requests and the multicast announcements from within the local subnet. It listens for incoming multicast requests from any discovering entity in its

subnet, at the same multicast request endpoint as any other LS (224.0.1.85/4160). Similarly, it listens for incoming multicast announcements from any LS in its subnet at the same multicast announcement endpoint as any other discovering entity (224.0.1.84/4160). When it receives a request or announcement message, it tunnels it to all the other TSs using their references and proxies that it holds.

**The Notifier:** This part has been implemented using one of the most useful mechanisms of Jini, the distributed event notification mechanism. When a TS starts up, it sends an inquiry to the GTLS about all the currently registered TSs. Then it uses the remote events model supported by Jini to request that the GTLS notify it whenever a new TS registers or leaves the environment.

**The Wrapper:** The wrapper is the main segment of the service. It publishes the TS's proxy in the GTLS and renews its lease as and when necessary. Also, it launches the assistant subsystems, the Listener and the Notifier, and keeps track of them. If more functionality is needed, such as the encryption of the data for security reasons or the detection of unnecessary TSs, this can be added as subsystems of the wrapper.

### 6.2.3 Jini Modifications

We would have preferred to implement our system without making any modifications to Jini. However, to overcome some of the obstacles of tunneling, we have had to modify the format of the outgoing request messages. Note that only the message formats need to be modified, the behavior of the rest of Jini remains intact and does not need to be changed.

The problem deals with the host address of the sender in a tunneled request message. When responding to a request message from a discovering entity, an LS uses the port number included in the message. However, it obtains the IP address of the sender by inquiring for the source of the multicast message. This works well within a subnet where the multicast message is originating from the discovering entity itself. However, in the

case of a tunneled request, the IP address is going to represent the TS's host and not the host of the original discovering entity. To overcome this problem, we have added the IP address of the host of the sending entity in the header of the request message, as shown at the top of Fig. 39. We don't need to add it in the announcement message since it already contains the host IP address.

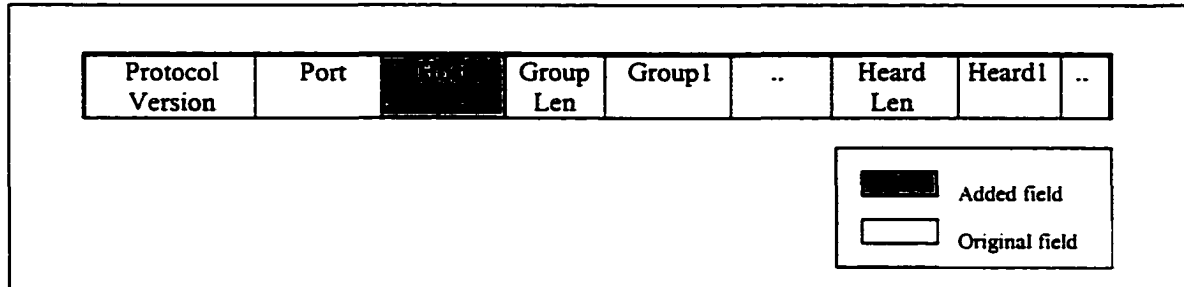


Fig. 39. New format of the outgoing request message

The mechanisms described above have been implemented using the Java Development Kit (JDK) 1.4 and the current Jini reference implementation 1.1 with the modifications that we have described in the previous subsection.

#### 6.2.4 A scalable alternative for super grids

Scalability is one of the main issues when applying the above mentioned solution, the *Collaboration approach*, to super grids that connect resources at massive numbers of loosely coupled subnets where multicasting is not enabled. Each TS has to know about and interact with all other TSs in the system. Scalability becomes an issue and TS becomes a bottleneck as the number of broadcasted messages or TSs to broadcast to increase. We achieve better scalability by building a hierarchy of federations as shown in Fig. 40.

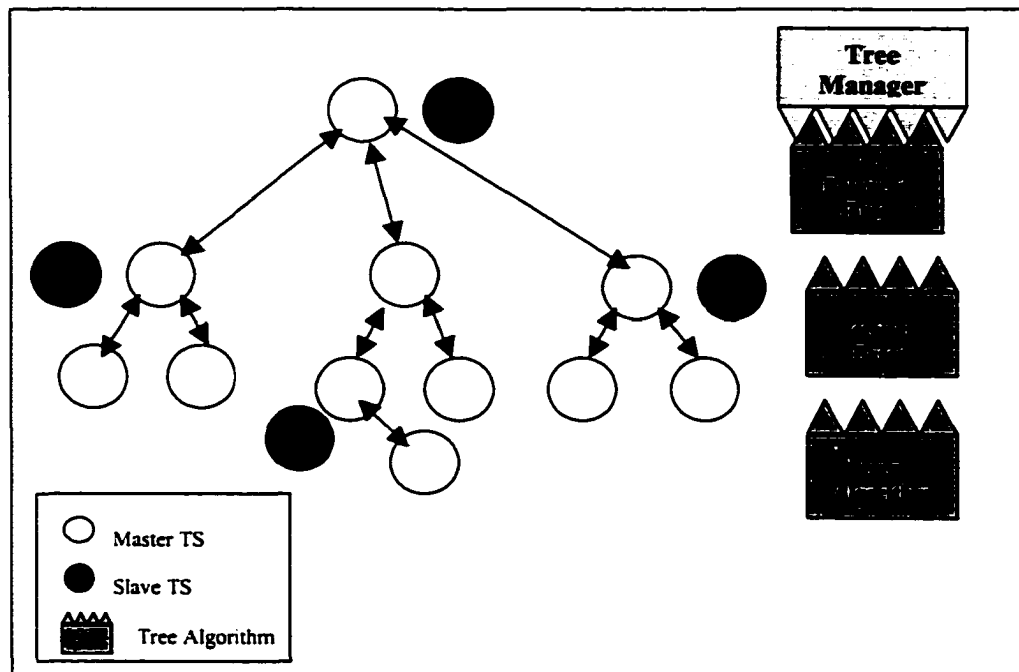


Fig. 40. Hierarchical Tunneling Approach

Instead of using the Jini Lookup Service to keep track of the distributed TSs, we introduce *Hierarchical Tunneling Manager*, where we can build a hierarchy of TSs. In this scenario, TS registers with a centralized *Tree Manager* that organizes the registered TSs in a tree based on a given *Tree Algorithm*. A TS node can be either a *root node*, an *intermediate node* or a *leaf node*. A TS does not need to keep track of all the TSs in the system, instead it keeps track of its parent and children, if any. New TSs are assigned a position based on the underlying tree algorithm. As shown in Fig. 41, *TreeAlgorithm* is an abstract class that needs to be implemented by the underlying tree algorithm. The user can plug-in any algorithm as his environment requires. Also, for critical subnets, Master/Slave approach could be applied to ensure high availability. A slave TS is started where necessary and keeps track of the master TS. It uses the remote events model supported by Jini to be notified whenever the status of the master TS changes. The slave TS takes over when the master one dies.

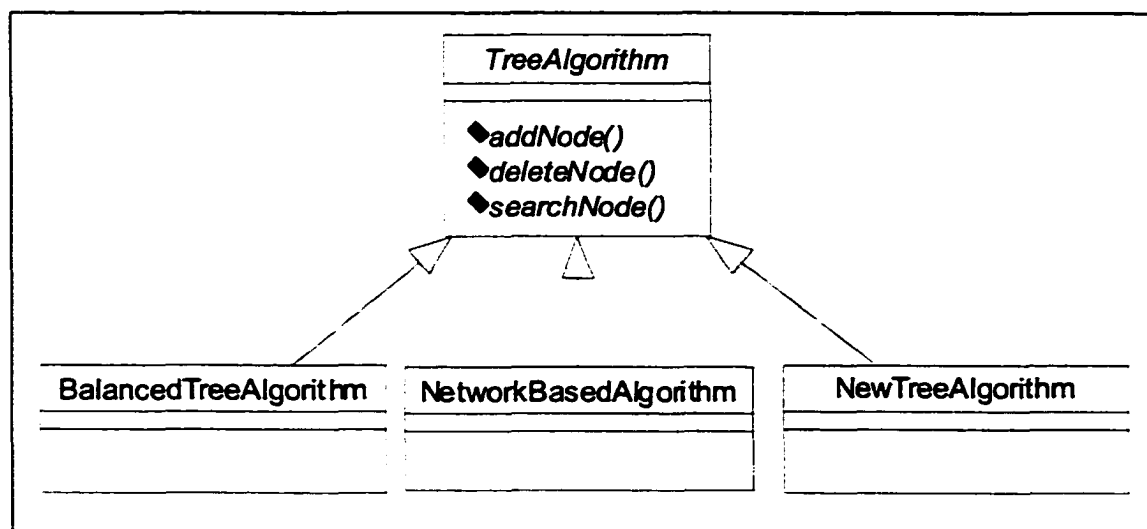


Fig. 41. Class diagram shows the implementing the Tree Algorithm

Using this alternative approach, each TS listens for incoming multicast requests and announcements in its subnet and broadcasts it as we describe below:

- A root node sends it to its children (if any).
- An intermediate node sends it to its parent and children (if any).
- A leaf node sends it to its parent.

For a broadcast traffic (via other TSs), we have the following scenarios:

- A root node multicasts it locally and sends it to its children except the one that it has received from.
- An intermediate node multicasts it locally in its subnet and sends it to its parent and children (if any) except the one that it has received from.
- A leaf node multicasts it locally in its subnet.

The *Hierarchal Tunneling* approach has some advantages over the *Collaboration* approach. In this scheme, the TSs are lighter in weight since they do not have to keep track of all the currently active TSs. This approach also gives the ability to perform several tunneling tasks concurrently. Additional functionality can be easily incorporated,



which cannot be done if we use the Jini Lookup Service. However, the root node is a potential communication bottleneck since all messages have to go through it. On the other hand, the *Collaboration* approach is a purely Jini approach which leverages off Jini technology in using the mechanism for storing proxies in the GTLS and also the event notification interface for keeping track of active TSs.

We implement the alternate approach and compare both schemes. In the following section, we show the experimental results that we have performed.

### 6.2.5 Experimental Results

We have conducted a number of experiments with different requirements to test both approaches. In this subsection, we discuss in details each of these experiments and present the performance data.

To measure the scalability of the two approaches, we apply different alternatives and measure the overhead of each one with respect to the following factors:

- Number of participant Tunneling Services (TSs).
- Overhead of broadcasting defined as the time that it takes for a TS to broadcast a tunneled message. This indicates whether or not a TS becomes a bottleneck.
- Overhead of delivery defined as the time that it takes for a broadcasted tunneled message to reach the entire participant TSs. This shows how the overall performance gets affected.

All the experiments were conducted using our experimental testbed, described in section 7.2. The machines where TSs run are connected via 100 Mbps Ethernet and thus communication cost between the machines are relatively small. Detailed observations are given in appendix A. All times are based on at least six measurements.

In order to simulate a large number of subnets that do not have multicasting enabled, we have implemented another version of the TS, called *Dummy TS*, where all it does is to listen to incoming broadcast traffic and discard the received packets. The *Dummy TS* does not listen for incoming multicast traffic. This enables us to start several TSs on the same subnet as if they were in different subnets.

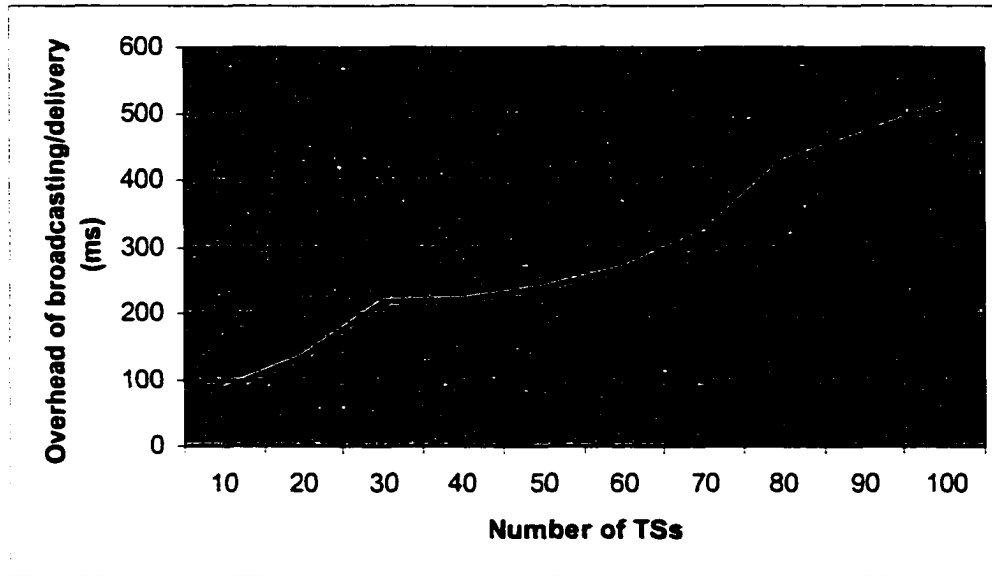


Fig. 42. Overhead of the Collaboration approach

Fig. 42 illustrates the overhead of the *Collaboration* approach. Our results show that as the number of the TSs increases, each TS becomes a bottleneck and the *Collaboration* approach scales poorly. Also, delivery time suffers with such increment. The data material of the figure are given in appendix A.1.

Fig. 43 illustrates the overhead of the *Hierarchal Tunneling* approach. The underlying *Tree Algorithm* that we use in the experiment is a *Balanced Tree* algorithm, which assigns TSs in a regular basis. We did our measurements for the *Root* node. We expect the broadcasting time to be similar for an intermediate node and less for a leaf node. However, the delivery time is expected to be a little bit higher for a leaf node.

Unlike the *Collaboration* approach, the *Hierarchal Tunneling* approach is more scalable as the performance of a TS and the overall performance did not get affected with the increased number of participant TSs. As the number of TSs approaches 100, the *Hierarchal Tunneling* approach gains a factor of 5.71 for the broadcasting time and 3.02 for the delivery time over the *Collaboration* approach

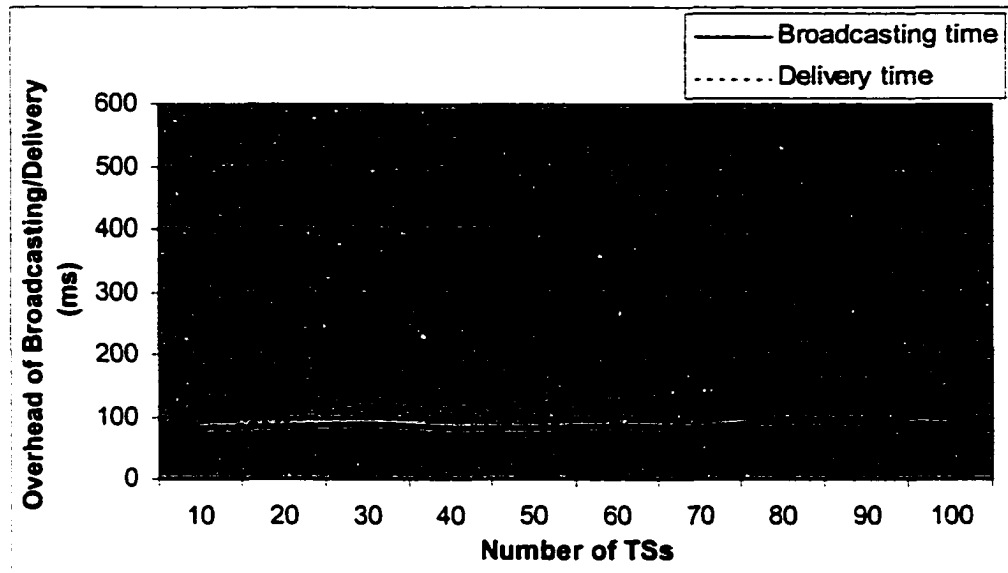


Fig. 43. Overhead of the Hierarchal Tunneling approach

### 6.2.6 Future Enhancements

There are some other issues that we have not addressed and in particular new features can be added to the system. For example, tunneled data can be encrypted when transported across subnets, so we can make sure that only the intended TSs can read it [95]. In addition, tunneling can be done on demand, i.e., we can have a TS only where needed. Thus, the first Jini client, service or LS that starts in the subnet can start the TS dynamically. Sometimes, we might have more than one TS running on the same network and not be aware of each other. Mechanisms like the ones used by mTunnel [95],[96] can be added in order to detect unnecessary TSs. A TS might send a multicastable test message periodically to a specific group address and port and wait for a response. TSs within the same network, if any, exchange messages to identify the redundant TSs. We will be examining our design and adding features as necessary in the near future.

### 6.3 Client Interfaces

One of the main characteristics of PROBE is to support a diverse set of client interfaces in which the client can interact with the system efficiently. Besides having our open, rich APIs, we support the visual and the command-line interfaces to illustrate the use of PROBE. These interfaces allow clients to interact with PROBE, giving them the ability to submit requests and to monitor and view the current state of resources and requests. Both interfaces are easy to use and set the client free from coding. Batch mode is another way, which may require some programming effort. This can be done by providing an interface to an existing programming language such as Java, C, FORTRAN, etc. or by providing some user-friendly scripting mechanism for the use of the client. We are planning to support the batch mode in the near future. Below, we describe the supported interfaces.

#### 6.3.1 Command-line Interface

The user-level prompt consists of “PROBE” followed by the angle bracket (>):

```
PROBE>
```

Fig. 44 provides a list of available commands that can be obtained using the To list the “help” command.

```

xterm
PROBE> help
PROBE usage:
search [-xml XMLfile] criteria : searches the Resource Repository based on the given criteria.
  If the -xml option is given, it will generate the result in the given XML file
monitor resource_name interval : monitors the status of a resource in a regular bases.
submit XMLfile : submits a request.
check requestID : checks the current status of an already submitted request.
stop requestID : stops an already submitted request.
resume requestID : resumes an already stopped request.
cancel requestID : cancels a submitted request.
get_output requestID: retrieves the output of an already submitted request.
h/help/H/Help : displays the usage message.
q/quit/Q/Quit : exits the program.
PROBE> submit Test.xml
Your request id is : 1
PROBE> check 1
The status of request: 1 is RUNNING
PROBE>

```

Fig. 44. Command line interface of PROBE

### 6.3.2 Visual Interface

The Graphical User Interface (GUI) addresses usability concerns in order to ensure that a novice user can quickly and easily learn to interact with PROBE. The GUI consists of the following components: menus, request editor, monitoring windows, error messages, and help features. Menus consist of a heading describing the options it provides and one or more sublevels which contain the available commands. The following is a list of menus and sub-menus of the main application:

- Resources
  - Search
  - Monitor
  - Exit
- Requests
  - New
  - Check
  - Retrieve Output
  - Stop

- Resume
  - Cancel
- Help
  - Help Topics
  - About PROBE

Fig. 45 and Fig. 46 show some snapshots of the visual interface of PROBE.

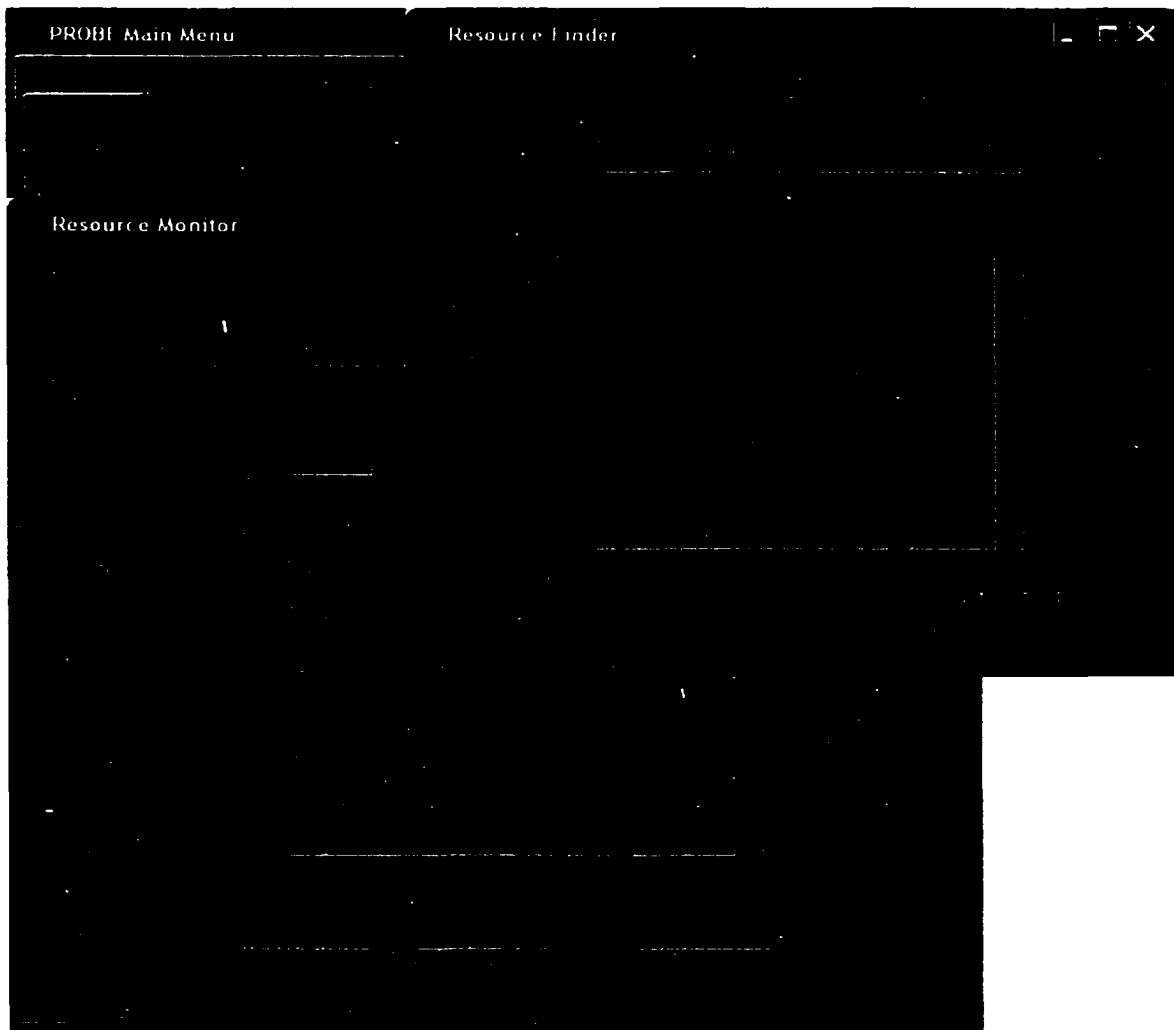


Fig. 45. Snapshots of the resource-related screens

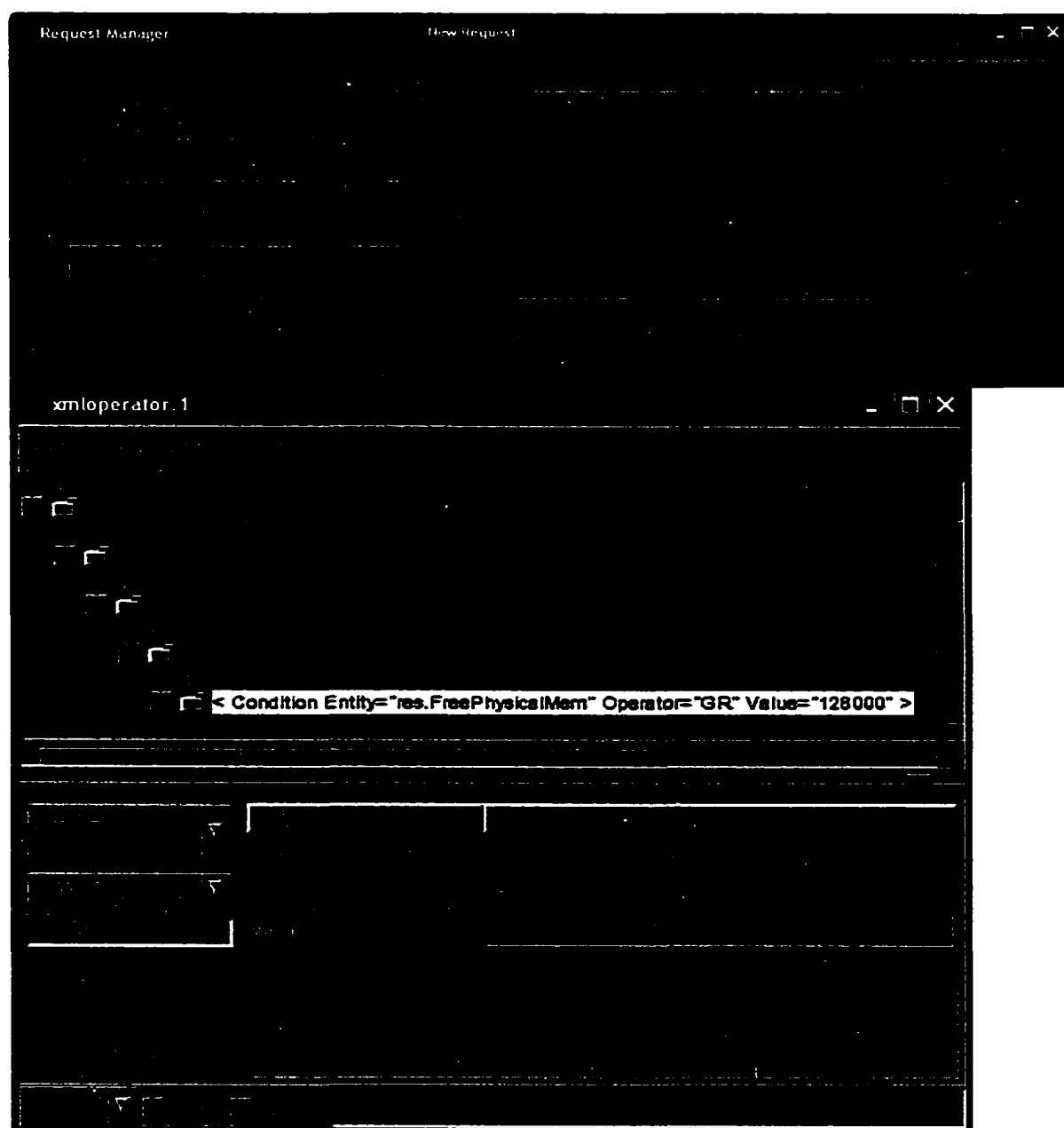


Fig. 46. Snapshots of the request-related screens

## 6.4 Package Design

The implementation of PROBE is structured into several Java packages. This includes: *probe*, *probe.common*, *probe.core*, *probe.repository*, *probe.resources*, *probe.jobs*, *probe.daemons*, *probe.policy*, *probe.algorithms*, *probe.util* and *probe.client*. The following subsections give an overview of these packages.

### 6.4.1 Package *probe*

This is the main package, which contains the package hierarchy of all the classes necessary for the PROBE implementation.

### 6.4.2 Package *probe.common*

This package contains classes that are used across all the packages. This includes:

- Common data types.
  - Constants.java: a holder class for global constants such as job status.
  - Parameter.java: name and value pairs.
- XML Parsers.
  - ResourceParser.java: acts as a translator providing a one-to-one mapping between the *Resource* object and its XML specification. It provides a convenient API for creating, manipulating, and checking the validity of a resource specification.
  - RequestParser.java: provides a convenient API for creating, manipulating, and checking the validity of a request specification.
- Plug-in Injector that provides mechanism for adding a plug-in on the fly. This class inherits the *ClassLoader* abstract class provided by Java where it can dynamically loads classes into RAM and then makes it easy to transfer them over networks.
- Event notification wrappers that provide convenient classes that help in handling Jini distributed events.



- Data locking wrapper that provides convenient classes where concurrency is handled efficiently. It supports an easy to use interface where read and write lock can be obtained and then released.

#### 6.4.3 Package *probe.core*

This package provides classes and interfaces that are fundamental to the design of the PROBE framework. Each module is represented by an interface, a wrapper, and sub-modules, if any. The wrapper publishes the module's proxy in the Module Lookup Service (MLS), gets the references to other modules and renews the lease as and when necessary. Below, we list the modules along with their corresponding classes:

- ClientInterfaceModule
  - ClientInterfaceModule.java
  - ClientInterfaceModuleService.java
- ResourceRepository
  - ResourceRepository.java
  - ResourceRepositoryService.java
- JobRepository
  - JobRepository.java
  - JobRepositoryService.java
- ResourceBroker
  - ResourceBroker.java
  - ResourceBrokerService.java
  - SchedulingAgent.java
  - Schedule.java
  - ScheduledTask.java
  - AllocationAgent.java
  - ReScheduler.java
- PolicyEnforcementManager
  - PolicyEnforcementManager.java

- PolicyEnforcementManagerService.java
- PolicyKeeper.java
- PolicyMatcher.java
- PolicyParser.java
- SLAMonitoringAgent.java
- JobMonitor
  - JobMonitor.java
  - JobMonitorService.java
  - JobEvent.java
- ResourceMonitor
  - ResourceMonitor.java
  - ResourceMonitorService.java

#### 6.4.4 Package *probe.repository*

The design of the repositories internal to the PROBE system is independent of the underlying protocol. A protocol layer has been introduced which acts as an intermediate layer between the protocol and the repository objects. It adapts the requests received from the repository object to the appropriate protocol format and adapts the responses from the protocol dependent objects to the internal format of PROBE. This package contains the classes of the various plug-in repository adaptors that the system possesses.

As shown in Fig. 47, this protocol layer is presented as a *RepositoryAdaptor* abstract class that needs to be implemented by the underlying protocol. It supports a set of abstract methods where jobs and resources can be manipulated. We have implemented an SQL-based repository adaptor and tested it using MySQL.

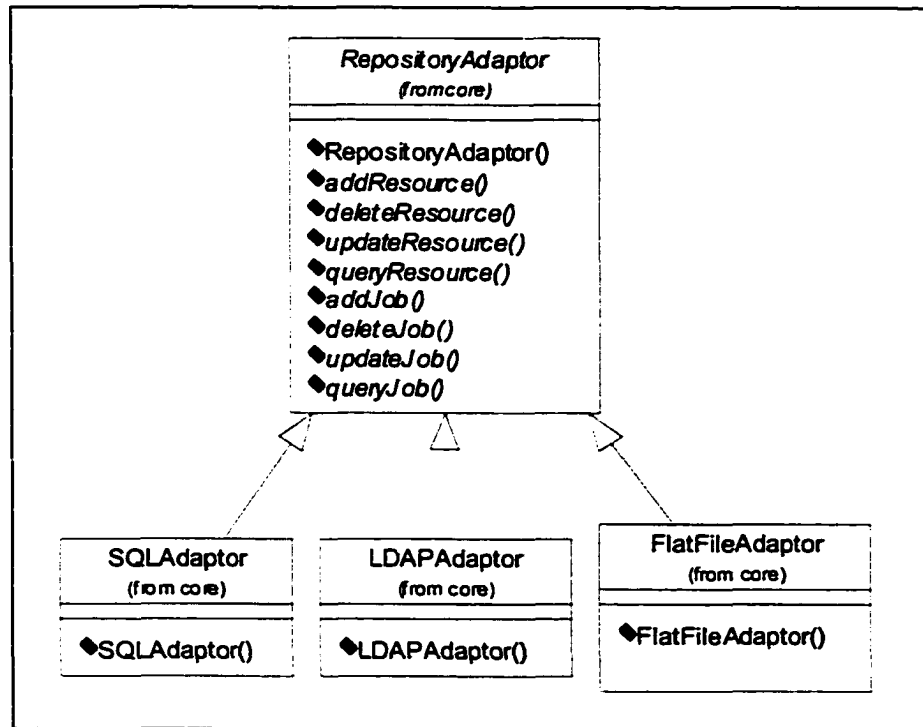


Fig. 47. Class diagram of Repository Adaptors

#### 6.4.5 Package *probe.algorithms*

This package contains the package hierarchy and the classes of the various plug-in scheduling and queuing algorithms the system supports. This includes:

- Package *probe.algorithms.scheduling*
  - `SchedulingAlgorithm.java`
  - `SimpleAlgorithm.java`
  - `Static_EACPM.java`
- Package *probe.algorithms.queuing*
  - `QueuingAlgorithm.java`
  - `FCFSQueuingAlgorithm.java`

## Scheduling Algorithms

As shown in Fig. 48, *SchedulingAlgorithm* is an abstract class and needs to be implemented by the provided algorithm. The *Resource Broker* has a unified interface to a set of scheduling algorithms making the design independent of any scheduling algorithm. *Italic methods represent the abstract methods that need to be implemented by the added scheduling algorithm. These methods are:*

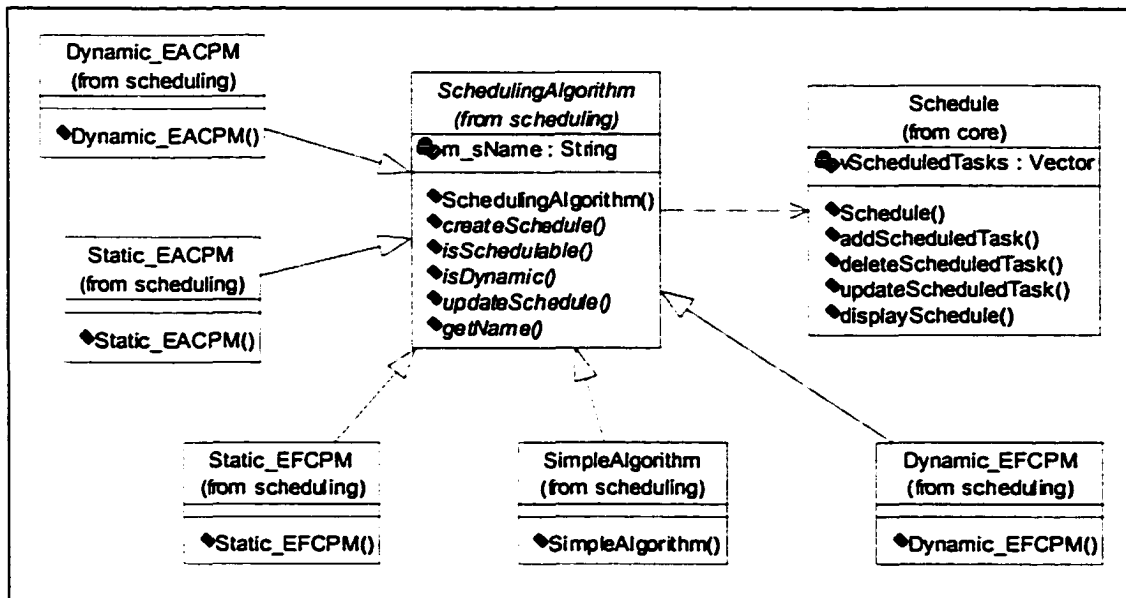


Fig. 48. Class diagram of Scheduling Algorithms

- *createSchedule*, to create the corresponding *Schedule*. This is an active object that has the order and placement of tasks that need to be allocated. The *Schedule* provides an API where scheduled tasks can be manipulated. This is very useful in rescheduling.
- *IsSchedulable*, to test whether or not the given problem can be scheduled. Sometimes, the schedule might require some additional information. If this information is missing, the scheduling algorithm can't proceed.
- *isDynamic*, to denote whether or not this is a dynamic scheduling algorithm.
- *updateSchedule*, for dynamic scheduling. In some cases, the allocation decision may need to be changed during execution. The Resource Broker calls this method

whenever the status of the job or one of its subtasks is changed. If the scheduling algorithm supports dynamic scheduling, then this method has to be implemented.

- *getName*, to get the name of the scheduling algorithm.

We have added some scheduling algorithms by inheriting the *SchedulingAlgorithm* abstract class and implementing its abstract methods. For example, we have developed a static algorithm for Directed Acyclic Graph (DAG) based on the Critical Path Method (CPM) [82] that yields assignment of high priority tasks. This algorithm, referred to as Static EA-CPM and its Pseudo-algorithm is shown in Fig. 49. In this algorithm, each node is associated with two numbers:

- *Weight*, representing the amount of computation it requires.
- *Path Weight* defined as:
  - the *Weight* if it is a leaf node,
  - and for a non-leaf node, its own *Weight* plus the largest *Weight* of its children.

1. *Initialization. Separates tasks into ready and waiting tasks and sets all the available resources to idle.*
2. *Assigns ready tasks to idle resources.*
  - 2.1. *Sorts ready tasks by their path weights*
  - 2.2. *Sorts resources by their CPU speeds in descending order.*
  - 2.3. *Assigns ready tasks until either no more available resources or no more ready tasks.*
3. *Once task is done, update its resource status to be idle and update its dependencies.*
  - 3.1. *If there are some ready tasks, then go to step 2.*
  - 3.2. *If both ready and waiting lists are empty, the stop.*

Fig. 49. Pseudo-algorithm for the Static EA-CPM

## Queuing Algorithms

For jobs that can be satisfied, the *Resource Broker* maintains an internal queue where such jobs are going to be held in a queue and based on a given queuing algorithm, the *Resource Broker* is going to select one job at a time and re-schedule it.

As shown in Fig. 50, *QueuingAlgorithm* is an abstract class and needs to be implemented by the provided queuing algorithm. The *Scheduler Agent* has a unified interface to a set of queuing algorithms. This makes the design independent of any queuing algorithm. It supports a set of methods where waiting jobs can be manipulated. The *addJob* and *getName* are the abstract method that the added queuing algorithm needs to implement. Other methods are common among all queuing algorithms. However, they can be overridden, if necessary.

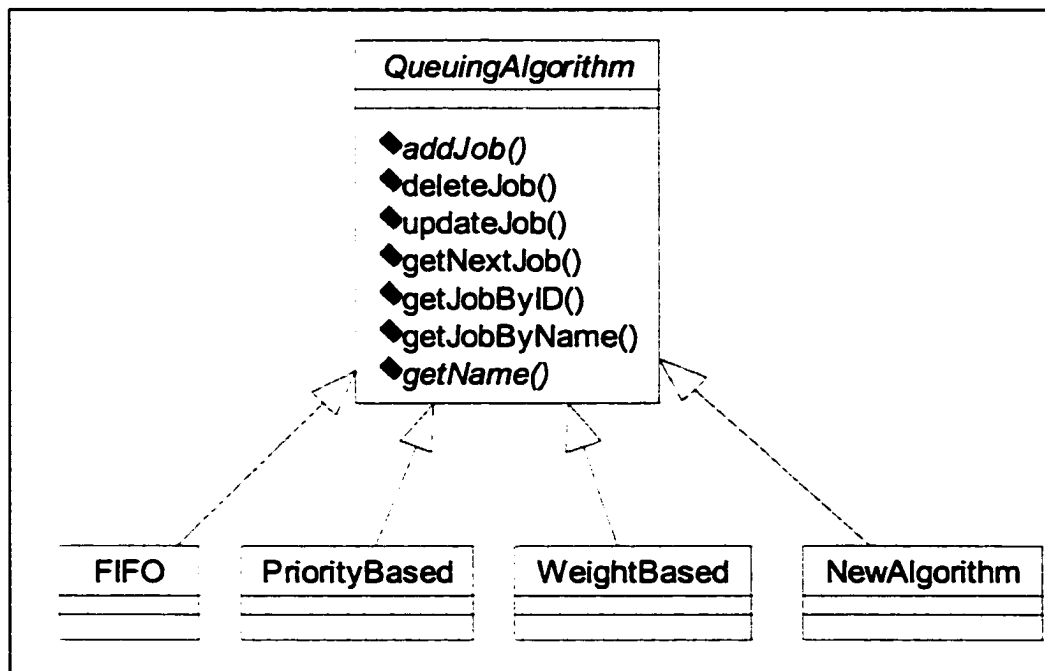
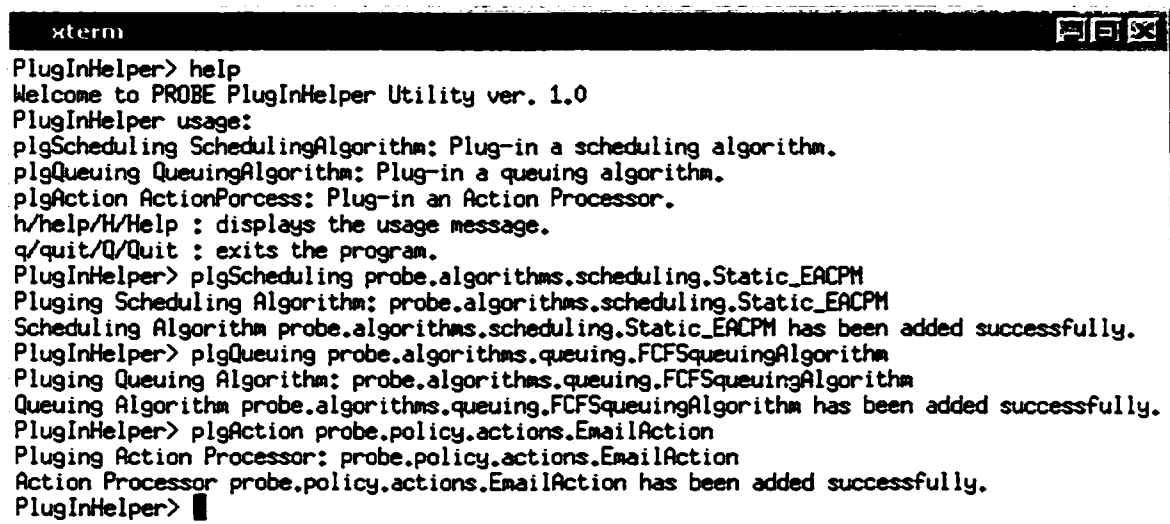


Fig. 50. Class diagram of Queuing Algorithms

We have added a First-In First-Out (FIFO) queuing algorithm by inheriting the *QueuingAlgorithm* and implementing the abstract methods. In this algorithm, the awaiting jobs are put into an ordered list and the first one is selected for re-scheduling.

### 6.4.6 Package *probe.util*

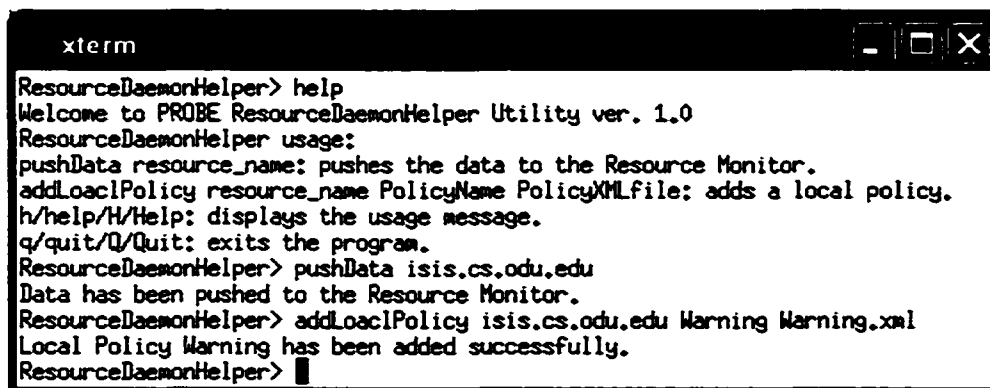
This package provides some supporting utilities that can be used in conjunction with PROBE. The current implementation supports a *Plug-in Helper* utility where various plug-ins can be added on the fly, and a *Resource Daemon Helper* utility where the resource provider can interact with resources local to their domain. Fig. 51 and Fig. 52 show some snapshots that demonstrate the use of the *Plug-in Helper* and *Resource Daemon Helper* utilities.



```

xterm
PlugInHelper> help
Welcome to PROBE PlugInHelper Utility ver. 1.0
PlugInHelper usage:
plgScheduling SchedulingAlgorithm: Plug-in a scheduling algorithm.
plgQueuing QueuingAlgorithm: Plug-in a queuing algorithm.
plgAction ActionProcess: Plug-in an Action Processor.
h/help/H/Help : displays the usage message.
q/quit/Q/Quit : exits the program.
PlugInHelper> plgScheduling probe.algorithms.scheduling.Static_EACPM
Plugging Scheduling Algorithm: probe.algorithms.scheduling.Static_EACPM
Scheduling Algorithm probe.algorithms.scheduling.Static_EACPM has been added successfully.
PlugInHelper> plgQueuing probe.algorithms.queuing.FCFSqueuingAlgorithm
Plugging Queuing Algorithm: probe.algorithms.queuing.FCFSqueuingAlgorithm
Queuing Algorithm probe.algorithms.queuing.FCFSqueuingAlgorithm has been added successfully.
PlugInHelper> plgAction probe.policy.actions.EmailAction
Plugging Action Processor: probe.policy.actions.EmailAction
Action Processor probe.policy.actions.EmailAction has been added successfully.
PlugInHelper> █
  
```

Fig. 51. PROBE PlugInHelper Utility



```

xterm
ResourceDaemonHelper> help
Welcome to PROBE ResourceDaemonHelper Utility ver. 1.0
ResourceDaemonHelper usage:
pushData resource_name: pushes the data to the Resource Monitor.
addLocalPolicy resource_name PolicyName PolicyXMLfile: adds a local policy.
h/help/H/Help: displays the usage message.
q/quit/Q/Quit: exits the program.
ResourceDaemonHelper> pushData isis.cs.odu.edu
Data has been pushed to the Resource Monitor.
ResourceDaemonHelper> addLocalPolicy isis.cs.odu.edu Warning Warning.xml
Local Policy Warning has been added successfully.
ResourceDaemonHelper> █
  
```

Fig. 52. PROBE ResourceDaemonHelper Utility

#### 6.4.7 Package *probe.resources*

This package contains a list of all the resource types supported by the system and their associated classes. This includes the abstract class *Resource* that needs to be extended by all the future resource classes as illustrated in Fig. 26. Currently, PROBE has only the *ComputationalResource* class since the focus of the current prototype is on computational grids.

#### 6.4.8 Package *probe.jobs*

This package contains all the classes that are associated with the supported job types. This includes the abstract class *Job*, as shown in Fig. 53, that needs to be extended by all the future job classes. For simplification, we do not show data members and methods, e.g., sets and gets, that do not add much to the model.

The *Resource Broker* and the *Scheduling Algorithm* have a unified interface to a set of Job Types. This makes the design independent of any job type. Italic methods represent the abstract methods that need to be implemented by the added job type. These methods are:

- *hasSubTasks*, to indicate whether or not the application has some subtasks.
- *getContainedTaks*, to get the contained sub-tasks, if any.
- *updateJobStatus*, to update the status of the job or one of its contained sub-tasks.
- *getReadyJobs*, to obtain a list of the tasks that are ready either at scheduling time or allocation.

Each job has a unique ID, name, type, user, constraints and status. The *vAdditionalInfo* container gives the flexibility where additional fields can be added. This is very helpful for scheduling algorithms that require some additional information to be given in advance prior to creating the schedule. For example, a scheduling algorithm might require some profiling to be done and that the job's execution time be known in advance. The API is flexible so that such additional information can be easily manipulated.



We have incorporated several types of application by inheriting the *Job* abstract class and implementing its abstract methods. These application types are:

- *SingleJob*, attributes specific to single application have been introduced.
- *AggregatedJob*, for this kind of application we have added a couple of methods where the aggregated sub-tasks can be easily manipulated. These methods are:

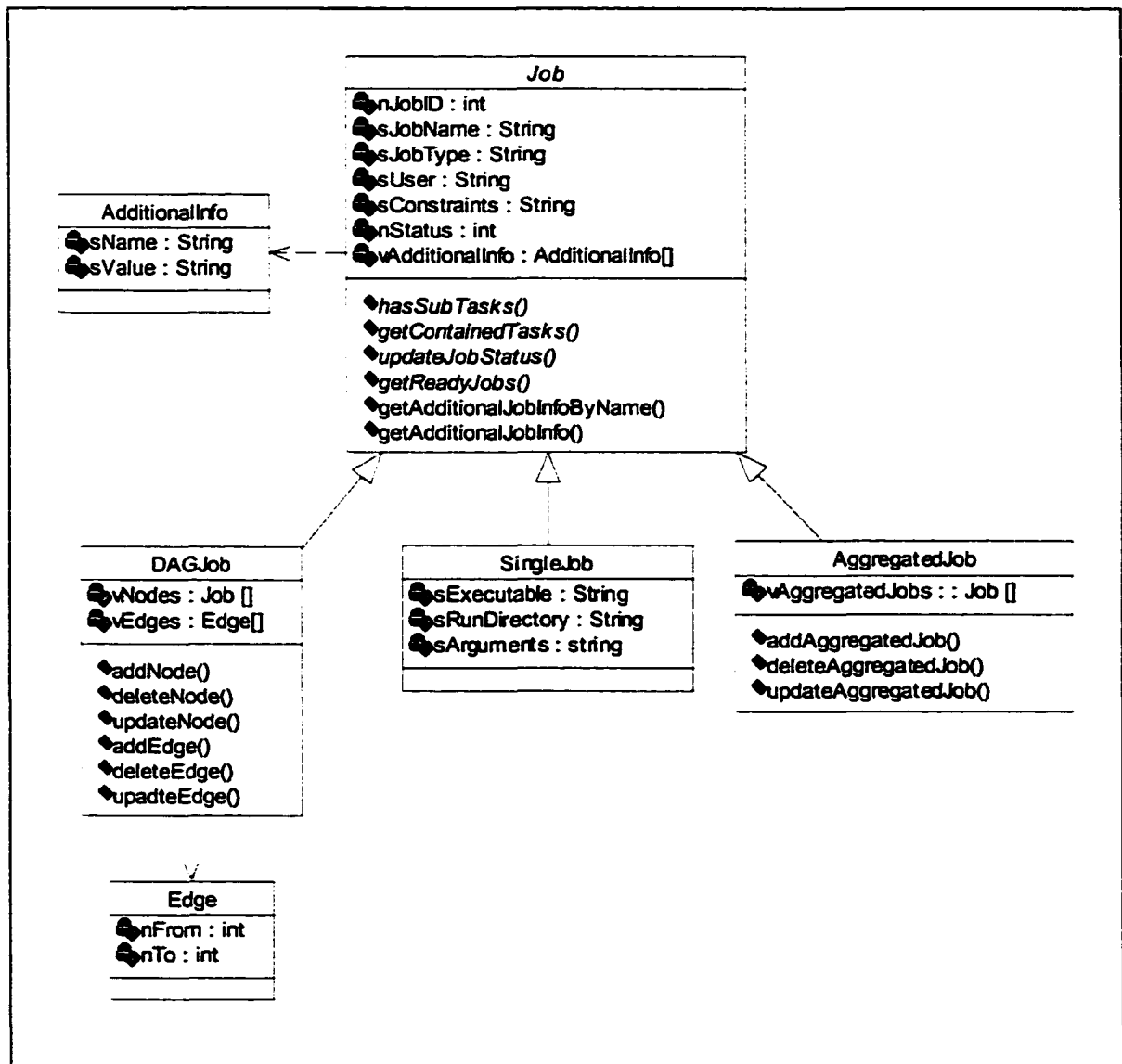


Fig. 53. Class diagram of Application Types

- *addAggregatedJob*, where a sub-task can be added to the application.
- *deleteAggregatedJob*, where an existing sub-task can be removed from the application.
- *updateAggregatedJob*, where the information of a sub-task can be changed.
- *DAGJob*, we have added the following methods where contained sub-tasks and their dependencies can be easily maintained:
  - *addNode*, to add a new node to the DAG.
  - *deleteNode*, to delete an existing node along with its dependencies from the DAG.
  - *updateNode*, to update the information of an existing node.
  - *addEdge*, to add a dependency between two existing nodes.
  - *deleteEdge*, to delete an existing dependency.
  - *updateEdge*, to update an existing dependency.

The abstract methods are implemented so as to fit the added job type. For example, *hasSubTasks* returns true in case of *DAG* and *Aggregated* job types if the application has at least one sub-task.

#### 6.4.9 Package *probe.daemons*

This package contains all the classes and interfaces that are necessary to implement a resource daemon.

- *Daemon.java*
- *DaemonService.java*
- *ProtocolAdaptor.java*
- *PolicyMonitor.java*
- *JobThread.java*
- *TimerThread.java*
- Platform Adaptors
  - *UnixAdaptor.java*

- LinuxAdaptor.java
- Win32Adaptor.java
- Other Grids Adaptors
  - GlobusAdaptor.java
  - GlobusJob.java
  - SGEAdaptor.java

The resource daemon is the component that acts as a gateway between PROBE and the managed resource. It can also be used as an integration base to interact with other grid systems. As shown in Fig. 54, a protocol layer has been introduced which maps the data collection and job execution/monitoring requests to the specific platform.

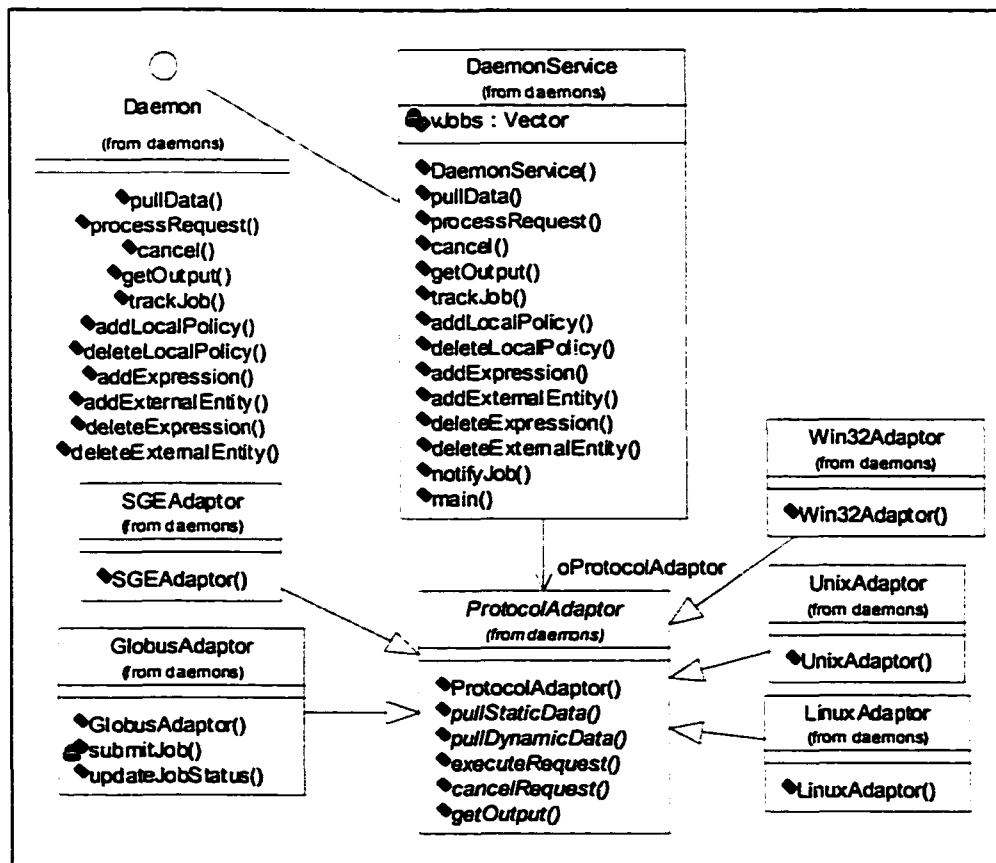


Fig. 54. Class diagram of Resource Daemons

The protocol layer is presented as a *ProtocolAdaptor* abstract class that needs to be implemented by the underlying daemon adaptor. It supports a set of abstract methods where statistics about the resource can be collected and tasks allocated and then monitored. These methods are:

- *pullStaticData*, to collect static data about the resource. This mainly is called whenever the resource daemon starts up.
- *pullDynamicData*, to collect dynamic data about the resource. this method is going to be called based on the dissemination option (pull/push, periodic/on-demand).
- *executeRequest*, to handle the allocated task.
- *cancelRequest*, to cancel an already submitted task.
- *getOutput*, to retrieve the output of an already submitted task.

### **Integration with various platforms**

We have developed several adaptors for different platforms as well as different grid environments. These platforms are *Unix*, *Win32* and *Linux*. The *Data Collector* relies on the existing utilities that the platform supports.

### **Integration with various grid systems**

Globus and Sun Grid Engine are the most popular grid systems that have wide acceptance in the grid community. We described our efforts in integrating with these grid environments.

We have integrated PROBE with Globus 2.0 using the Java Commodity Grid (CoG) Kit 0.9.13 [77]. PROBE acts as a client for the Globus GRAM and generates RSL on the fly for each job being submitted to a resource managed by Globus. We have used the following packages:

- **RSL**: to manipulate the translated RSL request and check its validity.
- **GRAM**: to create, submit and monitor jobs with the RSL being created by the RSL package.

- **MDS:** to query and collect data about the status of the resources being managed by Globus.
- **GSI:** to enable secure access to the resources.

We have also integrated with Sun Grid Engine 5.3 via the easy-to-use command line interface. Grid Engine is an open source community effort sponsored by Sun Microsystems and compatible with the Sun Grid Engine. Its main objective is to extend Sun's Grid Engine. We are planning to have a pure JNI adaptor that allows PROBE to interact effectively with Sun Grid Engine 5.3 in the near future.

#### **6.4.10 Package *probe.policy***

This package contains the package hierarchy and the classes that are necessary to implement the policy framework.

- SLA.java
- Action.java
- Expression.java
- ExternalEntity.java
- CachedResource.java
- Package *probe.policy.actions*
  - ActionProcessor.java
  - EmailAction.java
  - ShellAction.java

#### **Action Processors**

An action processor is the component that handles specific kinds of actions when policy terms are not met. When an action is created, it gets assigned an action type specifying its action processor and a set of parameters. Each parameter is a name and value pair, specifying the behavior the action processor has to take when the action is triggered. As shown in Fig. 55, our infrastructure eases the plug-and-play for action processors.

*ActionProcessor* is an abstract class that has some abstract methods that need to be implemented by the underlying action processor. These methods are:

- *getName*, to get the name of the action processor.
- *takeAction*, to take the supplied action.

We designed and implemented some action processors by inheriting the *ActionProcessor* abstract class and implementing its abstract methods. The current implementation of PROBE supports *Shell* where a designated shell script is executed and predefined arguments can be passed, and *Email* where a detailed email regarding the violation is sent via email.

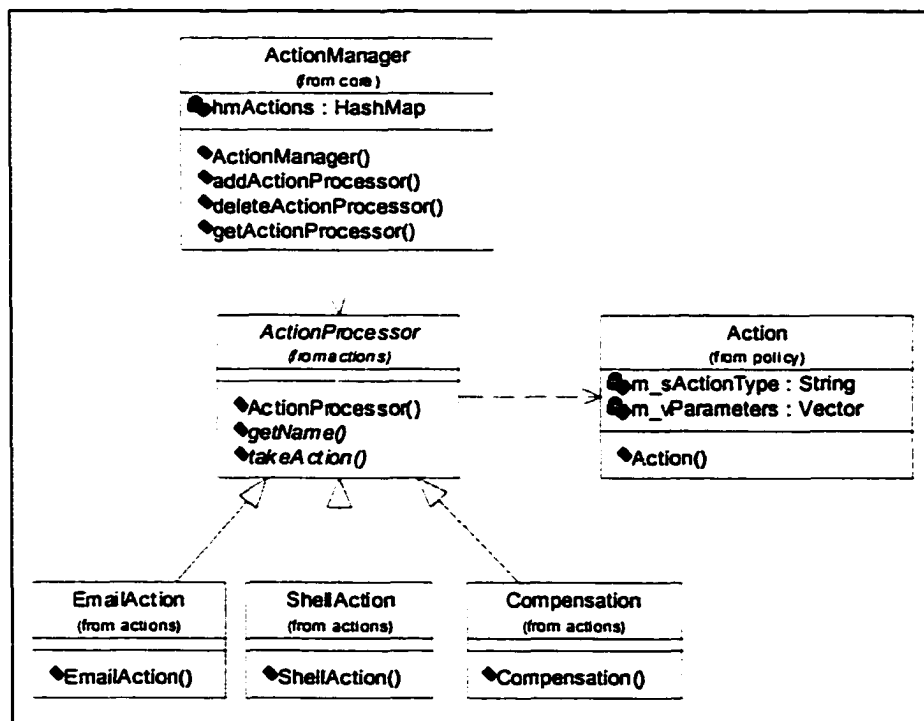


Fig. 55. Class diagram of Action Infrastructure

#### 6.4.11 Package *probe.client*

This package provides the classes necessary to interface with PROBE. The current implementation of PROBE supports visual and command-line interfaces where the client

can interact with the system. This package includes the package hierarchy and the classes necessary to implement those interfaces. It has two packages:

- Package *probe.client.cml*
  - InteractiveAPI.java
- Package *probe.client.gui*
  - RMclient.java
  - MainInterface.java
  - SearchInterface.java
  - MonitorInterface.java
  - MonitorThread.java
  - NewRequestInterface.java
  - RequestManager.java
  - About.java

## 6.5 Summary

In this chapter, we have described the current prototype implementation of PROBE. We have presented the tools and environments that we have used to implement the current prototype. We also described an enhancement for Jini in order to enable it across networks that do not support multicasting. We presented two approaches to resolve this issue, the *Collaboration* approach, which is a pure Jini solution that relies on the Jini's Lookup Service; and the *Hierarchal Tunneling* approach that relies on building a hierarchy of Tunneling Services (TSs). As the number of participant TSs continues to grow, the *Hierarchal Tunneling* approach is more scalable since it gives the ability of performing several tunneling tasks concurrently. Our experiment shows that as the number of TSs approaches 100, the broadcasting time is 5.71 times faster than that of the *Collaboration* approach and the delivery time is 3.02 times faster. Finally, we presented how the implementation of the PROBE prototype is structured into functional modules and packages using class diagrams and package overviews.

## CHAPTER VII

### EVALUATION AND EXPERIMENTAL RESULTS

In this chapter, we describe the methodologies that we use to evaluate the effectiveness of PROBE as a general-purpose policy-based resource brokering environment for computational grids. We describe the experimental testbed that we use to carry out our experimental results. We also present the results obtained when the PROBE framework is applied in the context of different experiments. These results demonstrate the effectiveness of our approach.

#### 7.1 Overview

Globus [42] and Sun Grid Engine [115] are the most popular and widely accepted grid systems in the grid community. Besides having our own grid environments managed by PROBE, we have layered PROBE on top of those two grid systems as shown in Fig. 56.

We start up with the basic skeleton of PROBE and add various plug-ins. For example, we add support for different kinds of applications such as *Single*, *DAG* and *Aggregated*; implement a scheduling algorithm based on the classic Critical Path Method (CPM) to schedule DAG into heterogeneous resources [82]; implement a First-In First-Out (FIFO) queuing algorithms; etc. The flexible design of the system made it easy for us to incorporate these plug-ins.

To conduct our test cases, we use different kinds of applications. For example, we use Pathfinder, an aircraft preliminary Multidisciplinary Design Optimization application that demonstrates the methodology for multidisciplinary communications and couplings between several engineering disciplines. We also use some simulated problems representing other application types.

The evaluation shows the flexibility and effectiveness of PROBE as a general policy-based resource brokering environment that can be utilized by various grid systems.



## 7.2 Experimental Testbed

Most systems usually evaluate their work in a tightly coupled network (LAN) to avoid issues such as heterogeneity and site autonomy that complicate the task of the resource brokering environment.

As PROBE targets a loosely coupled network environment, we believe that our test should prove that our system could map well in such an environment. We have chosen to evaluate PROBE in a loosely coupled network (the Internet) with heterogeneous resources. As shown in Fig. 56, our testbed environment, called PROBE Computational Grid (PCG) testbed, is made up of the four loosely coupled administrative domains.

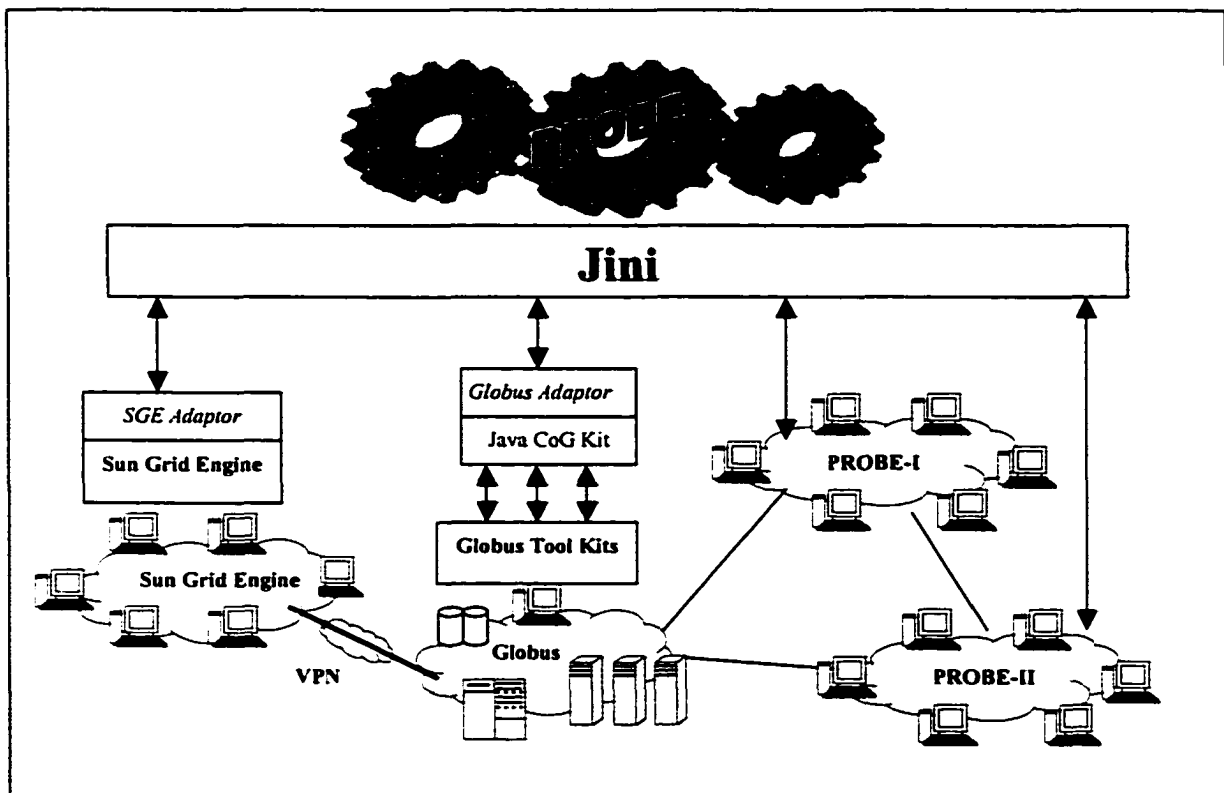


Fig. 56. PCG Test Bed Environment

In the first domain, we have installed Globus Tool Kits 2.0 on a 733 MHz PIII PC with Redhat 7.2 Linux. This administrative domain has seven PCs. Some of the resources have a resource policy stating that the resource cannot be used when its free physical

memory drops below 64 MB. A *Globus adaptor* has been installed where PROBE can interoperate with Globus. Table 3 gives further specifications about this grid.

TABLE 3  
FURTHER SPECIFICATIONS ABOUT GLOBUS DOMAIN

Host name	Manufacturer	Resource Type	CPU	OS	Memory
globus	Dell	PIII/Dimension L866r	900	Redhat 7.2 Linux	192
hesham	Dell	PIV/ Dimension 4300s	1600	Windows XP	256
imran	Dell	PIV/Dimension 4500s	1800	Windows XP	256
neptune	Gateway	PIII/GP7-450	733	Redhat 7.2 Linux	128
sanhour	Dell	PIV/Dimension 4500s	1800	Windows XP	256
riyadh	Dell	PIV/Dimension 4500s	1800	Windows XP	256

The second administrative domain belongs to a private organization called Trendium Incorporation. In this administrative domain, we have installed Sun Grid Engine 5.3 on Sun ULTRAstation-10 workstation with Solaris 2.8. The administrative domain has 32 Sun ULTRAstation-10 workstations; each has Solaris 2.8, CPU of 440 MHz and memory of 512. This administrative domain has a system wide policy that restricts resources from being accessed between 9 AM and 5 PM. This administrative domain is accessible via a Cisco Virtual Private Network (VPN) server. The VPN client is established at the first domain. A resource daemon with a *Sun Grid Engine (SGE) adaptor* has been installed in which PROBE can interoperate with the Sun Grid Engine.

In the third administrative domain, we have installed PROBE on Sun ULTRAstation-10 workstation with Solaris 2.8. This administrative domain has 15 Sun workstations with Solaris 2.8 and 3 PCs with Windows 2000. Some of the resources have a resource policy stating that the resource cannot be used when the number of interactive users exceeds 5. Table 4 gives further specifications about this grid.

TABLE 4  
FURTHER SPECIFICATIONS ABOUT PROBE I GRID

Host name	Manufacturer	Resource Type	CPU	OS	Memory
brain	Sun	ULTRAstation-10	333	Solaris 2.8	128
cash	Sun	Sun-Blade-1000	750	Solaris 2.8	1024
cheeta	Sun	ULTRAstation-10	300	Solaris 2.8	128
dilbert	Sun	Sun-Ultra-250	400 (dual processors)	Solaris 2.8	2084
dot	Sun	ULTRAstation-10	300	Solaris 2.8	128
escher	Dell	PIV/Dimension 4300s	1600	Windows 2000	128
egbert	Sun	Sun-Blade-1000	333	Solaris 2.8	128
grenada	Sun	ULTRAstation-10	300	Solaris 2.8	64
hutch	Sun	Sun-Blade-1000	750 (dual processors)	Solaris 2.8	1024
isis	Sun	Sun-Blade-1000	750	Solaris 2.8	1024
labpc4	Dell	PIII/Dimension L866r	864	Windows 2000	265
labpc43	Dell	PIII/Dimension L1000R	1000	Windows 2000	128
o2	Sun	Sun-Blade-1000	333	Solaris 2.8	128
pitfall	Sun	ULTRAstation-10	333	Solaris 2.8	128
puma	Sun	ULTRAstation-10	300	Solaris 2.8	128
tabby	Sun	ULTRAstation-10	333	Solaris 2.8	128
tango	Sun	Sun-Blade-1000	750 (dual processors)	Solaris 2.8	1024
yakko	Sun	ULTRAstation-10	333	Solaris 2.8	128

The fourth administrative domain has 4 Sun workstations with Solaris 2.8 and 3 PCs with Windows 2000. Some of the resources have a resource policy stating that the resource cannot be used when its load exceeds 50%. Table 5 gives further information

about this grid. Resources in this domain are managed by PROBE running in the third administrative domain.

TABLE 5  
FURTHER SPECIFICATIONS ABOUT PROBE II GRID

Host name	Manufacturer	Resource Type	CPU	OS	Memory
res-audio	Sun	ULTRAstation-10	333	Solaris 2.8	128
res-client1	Sun	ULTRAstation-10	333	Solaris 2.8	128
res-iri	Sun	Sun-Ultra-30	296	Solaris 2.8	128
res-nt7	Dell	PIV/Dimension 4500	1800	Windows 2000	128
res-nt9	Dell	PIV/Dimension 4500	1800	Windows 2000	128
res-nt10	Dell	PIV/Dimension 4500	1800	Windows 2000	128
res-video	Sun	Sun-Ultra-30	296	Solaris 2.8	128

The version numbers and release dates of the software packages used in the experiments are shown in Table 6.

TABLE 6  
VERSION NUMBERS OF THE SOFTWARE PACKAGES USED IN THE  
EXPERIMENTS

Package	Release
Java	1.4.1
Jini	1.1
MySQL	3.23
JAXP	1.2
xmloperator	1.11
Globus	2.0
Java Commodity Grid Kit	0.9.13
Sun Grid Engine	5.3

### 7.3 Test Applications

In order to evaluate the effectiveness of PROBE, we need to test it under different scenarios. We use different kinds of test applications. Some of which are real and some are simulated. In this section, we describe the test applications that we use to conduct our experiments along with their allocation constraints.

#### 7.3.1 Single Job

We use different kinds of single jobs in our evaluation. In this subsection, we describe a sample single job that represents a weather-modeling application. This application requires a resource where the CPU load is less than 60% and the available scratch space is greater than 20 GB. An FJL script representing the sample single job is given in Fig. 57.

```
<?xml version="1.0"?>
<!DOCTYPE Request SYSTEM "Request.dtd">
<Request>
  <Single Name="WeatherModeling" Arguments=""
  Executable="/home/theneyan/Demo/App1/weather.csh"
  RunDirectory="/home/theneyan/Demo/App1">
    <Policy>
      <Rule>
        <AND>
          <Condition Entity="res.ResourceLoad" Operator="LS" Value="60"></Condition>
          <Condition Entity="res.FreeSwapSpace" Operator="GR" Value="20000000"></Condition>
        </AND>
      </Rule>
      <Action Type="Shell">
        <AdditionalInfo Name="ScriptName" Value="compensation.sh"></AdditionalInfo>
        <AdditionalInfo Name="AccountNumber" Value="11-22-33"></AdditionalInfo>
        <AdditionalInfo Name="Credit" Value="5"></AdditionalInfo>
      </Action>
    </Policy>
  </Single>
</Request>
```

Fig. 57. FJL script representing a sample single application

### 7.3.2 Co-Allocation Job

For this kind of application, we developed a simulated client-server application representing one client and two servers. This kind of application requires that a set of resources be available for use simultaneously. We specify that each resource needs to have at least 128 MB of available physical memory. An FJL script representing the sample single job application is given in Fig. 58.

```
<?xml version="1.0"?>
<!DOCTYPE Request SYSTEM "Request.dtd">
<Request>
<Aggregated Name="CoAllocationTest" Type="CoAllocation">
  <Single Name="client" Arguments=""
    Executable="/home/theneyan/Demo/App1/client.csh"
    RunDirectory="/home/theneyan/Demo/App1"></Single>
    <Single Name="server1" Arguments=""
      Executable="/home/theneyan/Demo/App1/server1.csh"
      RunDirectory="/home/theneyan/Demo/App1"></Single>
      <Single Name="server2" Arguments=""
        Executable="/home/theneyan/Demo/App1/server2.csh"
        RunDirectory="/home/theneyan/Demo/App1"></Single>
    <Policy>
      <Rule>
        <Condition Entity="res.FreePhysicalMem" Operator="GR" Value="128000"></Condition>
      </Rule>
    </Policy>
  </Aggregated>
</Request>
```

Fig. 58. FJL script representing a Co-Allocation application

### 7.3.3 Parametric Job

We have a Computational Fluid Dynamics (CFD) simulation for polishing equipment. In this application, a simulation program *Polish(x,y)* is repeatedly executed with different initial conditions as a means of exploring the behavior of the polishing equipment. We need to run the application for three different combinations of  $x$  and  $y$ . We specify that each resource needs to have at least 128 MB of available physical memory. An FJL script representing the application is given in Fig. 59.

```

<?xml version="1.0"?>
<!DOCTYPE Request SYSTEM "Request.dtd">
<Request>
<Aggregated Name="Polishing" Type="Parametric">
  <Single Name="Simulation1" Arguments="x=1 y=1"
Executable="/home/theneyan/Demo/App1/simulation.csh"
RunDirectory="/home/theneyan/Demo/App1"></Single>
  <Single Name="Simulation2" Arguments="x=1 y=2"
Executable="/home/theneyan/Demo/App1/simulation.csh"
RunDirectory="/home/theneyan/Demo/App1"></Single>
  <Single Name="Simulation3" Arguments="x=2 y=2"
Executable="/home/theneyan/Demo/App1/simulation.csh"
RunDirectory="/home/theneyan/Demo/App1"></Single>
</Aggregated>
<Policy>
<Rule>
<Condition Entity="res.FreePhysicalMem" Operator="GR" Value="128000">
</Condition>
</Rule>
</Policy>
</Request>

```

Fig. 59. FJL script representing a Parametric application

#### 7.3.4 Pathfinder – Sample DAG application

The Multidisciplinary Design Optimization Branch (MDOB) at NASA Langley Research Center (LaRC) is conducting basic research in Multidisciplinary Design Optimization (MDO) methods and tools for the design and optimization of aerospace vehicles throughout their flight envelope. Their main objective is to increase design confidence and cut development time [87].

Pathfinder is an aircraft preliminary MDO system that demonstrates the methodology for multidisciplinary communications and couplings between several engineering disciplines. It has been developed jointly by the NASA/LaRC and Lockheed Martin Engineering and Science Services. The current version consists of the disciplines of aerodynamics and structures coupled aero-elastically. As shown in Fig. 60, these disciplines, represented by multiple heterogeneous modules, interact with each other to solve the overall design problem. Typically these modules consist of various Fortran and C programs and have been developed as separate codes. These modules have been integrated through the use of scripts that make the process of specifying and optimizing

the overall design of such application a long and tedious process often taking several weeks. A solution is reached when the design variables are no longer changing or a satisfactory feasible design is obtained.

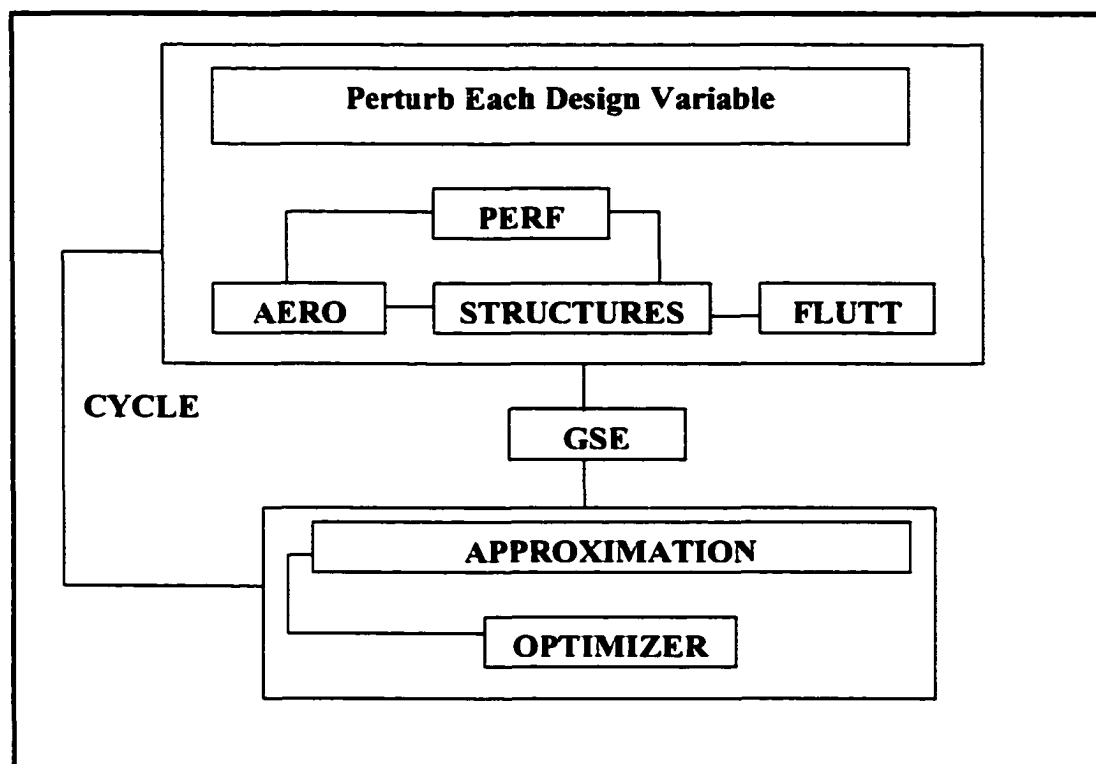


Fig. 60. The Pathfinder System

The Pathfinder system has an outer analysis cycle and an inner system optimization cycle. A solution is reached when the design variables are no longer changing or a satisfactory feasible design is obtained. In our experiments, we focus only on one sweep of the Pathfinder system. We took out the scripts that integrate the Pathfinder's modules, and instead used FJL to integrate them and then used this application as one of the driving forces for our prototype. The FJL script representing the Pathfinder application is given in Fig. 61.



```

<?xml version="1.0"?>
<!DOCTYPE Request SYSTEM "Request.dtd">
<Request>
<DAG Name="Pathfinder">
  <Single Name="PERF" Arguments=""
Executable="/home/theneyan/Demo/Pathfinder/PERF.csh"
RunDirectory="/home/theneyan/Demo/Pathfinder/">
    </Single>
  <Single Name="AERO" Arguments=""
Executable="/home/theneyan/Demo/Pathfinder/AERO.csh"
RunDirectory="/home/theneyan/Demo/Pathfinder/">
    </Single>
  <Single Name="STRUCTURES" Arguments=""
Executable="/home/theneyan/Demo/Pathfinder/STRUCTURES.csh"
RunDirectory="/home/theneyan/Demo/Pathfinder/">
    </Single>
  <Single Name="FLUTTER" Arguments=""
Executable="/home/theneyan/Demo/Pathfinder/FLUTTER.csh"
RunDirectory="/home/theneyan/Demo/Pathfinder/">
    </Single>
  <Single Name="GSE" Arguments=""
Executable="/home/theneyan/Demo/Pathfinder/GSE.csh"
RunDirectory="/home/theneyan/Demo/Pathfinder/">
    </Single>
  <Single Name="APPROXIMATION" Arguments=""
Executable="/home/theneyan/Demo/Pathfinder/APPROXIMATION.csh"
RunDirectory="/home/theneyan/Demo/Pathfinder/">
    </Single>
  <Single Name="OPTIMIZER" Arguments=""
Executable="/home/theneyan/Demo/Pathfinder/OPTIMIZER.csh"
RunDirectory="/home/theneyan/Demo/Pathfinder/">
    </Single>
    <Dependency From="PERF" To="AERO"></Dependency>
    <Dependency From="PERF" To="STRUCTURES"></Dependency>
    <Dependency From="PERF" To="FLUTTER"></Dependency>
    <Dependency From="AERO" To="GSE"></Dependency>
    <Dependency From="STRUCTURES" To="GSE"></Dependency>
    <Dependency From="FLUTTER" To="GSE"></Dependency>
    <Dependency From="GSE" To="APPROXIMATION"></Dependency>
    <Dependency From="APPROXIMATION" To="OPTIMIZER"></Dependency>
  <Policy>
  <Rule>
  <AND>
    <Condition Entity="res.CPUspeed" Operator="GR" Value="700"></Condition>
    <Condition Entity="res.FreePhysicalMem" Operator="GR" Value="128000"></Condition>
  </AND>
  </Rule>
  </Policy>
</DAG>
</Request>

```

Fig. 61. FJL script representing the Pathfinder application

## 7.4 Experiments

The overall objective of the evaluation is to evaluate the effectiveness of PROBE as a general-purpose policy-based resource brokering environment for computational grids. The evaluation of our work has been divided into two parts: *Qualitative Evaluation*, in which we test whether or not the system delivers its promise; and *Quantitative Evaluation*, in which we evaluate how effectively the system delivers its promise. We conduct a number of experiments with different requirements to test the effectiveness of our framework. In this subsection, we discuss these experiments to demonstrate the effectiveness of PROBE. All these experiments were conducted using the PCG testbed.

### 7.4.1 Qualitative Experiments

In qualitative evaluation, we investigated whether or not the PROBE prototype delivers what it promises in terms of functionalities and characteristics.

We tested PROBE within multiple administrative domains. In our experiments, each domain, and in fact resources within each domain, specify their own set of rules and policies. Different policies are assigned at resources in the PCG testbed as the following:

- *First domain*, some of the resources have a resource policy stating that the resource cannot be used when its free physical memory goes below 64 MB.
- *Second domain*, we specify a system wide policy that all resources are not to be accessed between 9 AM and 5 PM.
- *Third domain*, some of the resources have a resource policy stating that the resource cannot be used when the number of users exceeds 5.
- *Fourth domain*, some of the resources have a resource policy stating that the resource cannot be used when its load exceeds 50%.

We submitted different kinds of applications with varying allocation constraints. PROBE was flexible enough to accommodate and adopt these policies. The rights of both the resource provider and the resource consumer were respected. The XML-based Policy Scripting Language (PSL) was flexible enough to provide the necessary richness for both resource providers and consumers to express the diverse kinds of policies.

Also, we tested the ability of PROBE to run in a heterogeneous environment consisting of different hardware and software platforms (e.g., Linux, MS Windows, Solaris) without any modifications. No problems were encountered. On the other hand, the PCG testbed has various kinds of resources, each with different architectures, different operating systems, different configurations and different vendors. We make sure that PROBE can accommodate all types of resources and manage them efficiently.

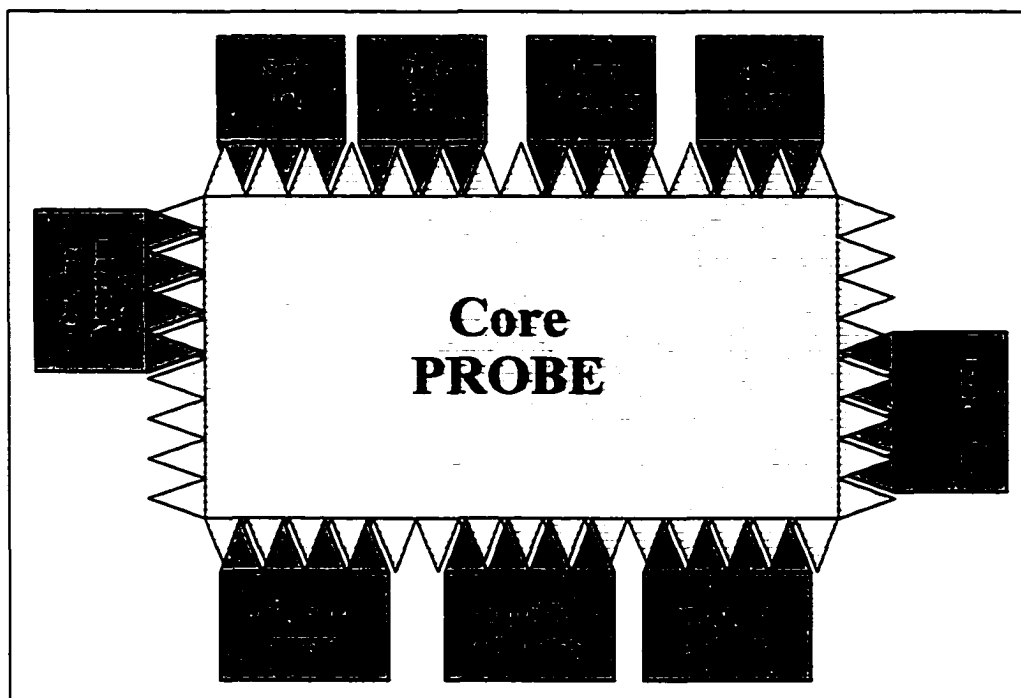


Fig. 62. Basic PROBE with different plug-ins

As a general-purpose resource brokering environment framework, we made sure that PROBE can be easily integrated with existing grid environments and incorporate different grid requirements. As shown in Fig. 62, we started with the basic skeleton of PROBE that has only the core components. Then, we integrated PROBE with different plug-ins. In chapter VI, we have provided the details on how to incorporate different plug-ins. As a sample below, we give a sequence of steps that is necessary to plug-in the static EA-CPM scheduling algorithm that we have explained earlier in 6.4.5:

- Inherit the *SchedulingAlgorithm* abstract class and implement its abstract methods as follow:
  - *createSchedule*, we follow the Pseudo-algorithm shown in Fig. 49.
  - *isSchedulable*, this algorithm requires some additional information such as the node weight and the path weight. We test whether or not the supplied job has this additional information. If not, this method returns *false*.
  - *isDynamic*, since this is a static algorithm, this method returns *false*.
  - *updateSchedule*, this method is meant for dynamic scheduling. Since this algorithm is static, we ignore this method.
  - *getName*, we return the name of the scheduling algorithm, “Static Early Assignment CPM”.
- Compile the Java file.
- Now that we have the class file, we can plug-in this algorithm using two different approaches:
  - *On the fly*: the *Plug-in Helper* utility allows the injection of different plug-ins on the fly. It loads the provided classes into RAM and then makes it easy to transfer them over networks.
  - *At start-up time*: the *Resource Broker* provides the facility where scheduling and queuing algorithms can be provided via the command-line. It relies on the *Plug-in Injector* class where classes can be dynamically loaded.

#### 7.4.1.1 Brokering

In testing the functionalities of our brokering infrastructure, we touch upon different aspects of brokering such as:

- Submitting different kinds of applications.
- Applying different scheduling techniques.
- Applying different queuing techniques.
- Rescheduling.

- Job Monitoring.

In this subsection, we describe typical brokering scenarios that occur within PROBE. We discuss different cases and show how PROBE handles them.

### Experiment 1: Successful Finish

Using the command-line client's interface of PROBE, we submitted two FJL-based problem descriptions: one for a Single job as illustrated in Fig. 57, and the other for a job of type DAG representing the Pathfinder problem as illustrated in Fig. 60. We also specified some application constraints. For the single job, we were looking for a resource with a CPU load of less than 40%. For the Pathfinder problem, we were looking for seven resources each of at least 700 MHz and 128 MB of Memory.

Each problem got forwarded to the *Client Interface Module*, which created the *Job* object and then passed the request to the *Resource Broker*. Below, we summarize our observations for these two problems:

- **Single Job:** A unique job identification is created and passed back to the *Client Interface Module* for tracking purposes. The *Scheduler Agent* then consults with the *Policy Enforcement Manager*. After enforcing all the policies and application requirements, one resource (res-client1/PROBE II) is made available for the job and passed to the *Scheduler Agent*, which then creates the *Schedule*. Next, the Allocation Agent allocates the job. When the job finishes successfully, the *Schedule* is terminated by the *Scheduler Agent*. Fig. 63 illustrates the sequence of operations involved in this experiment.
- **DAG Job:** After generating the unique IDs for the job and all its sub-tasks, the *Scheduler Agent* consults with the *Policy Enforcement Manager*. Out of the 23 resources that are available in the system, the *Policy Enforcement Manager* selects a subset of 4 appropriate resources (cash, hutch, isis and tango/ PROBE I) and notifies the *Scheduler Agent* of the selection. The selection is based on system policies, resource policies and the application constraints

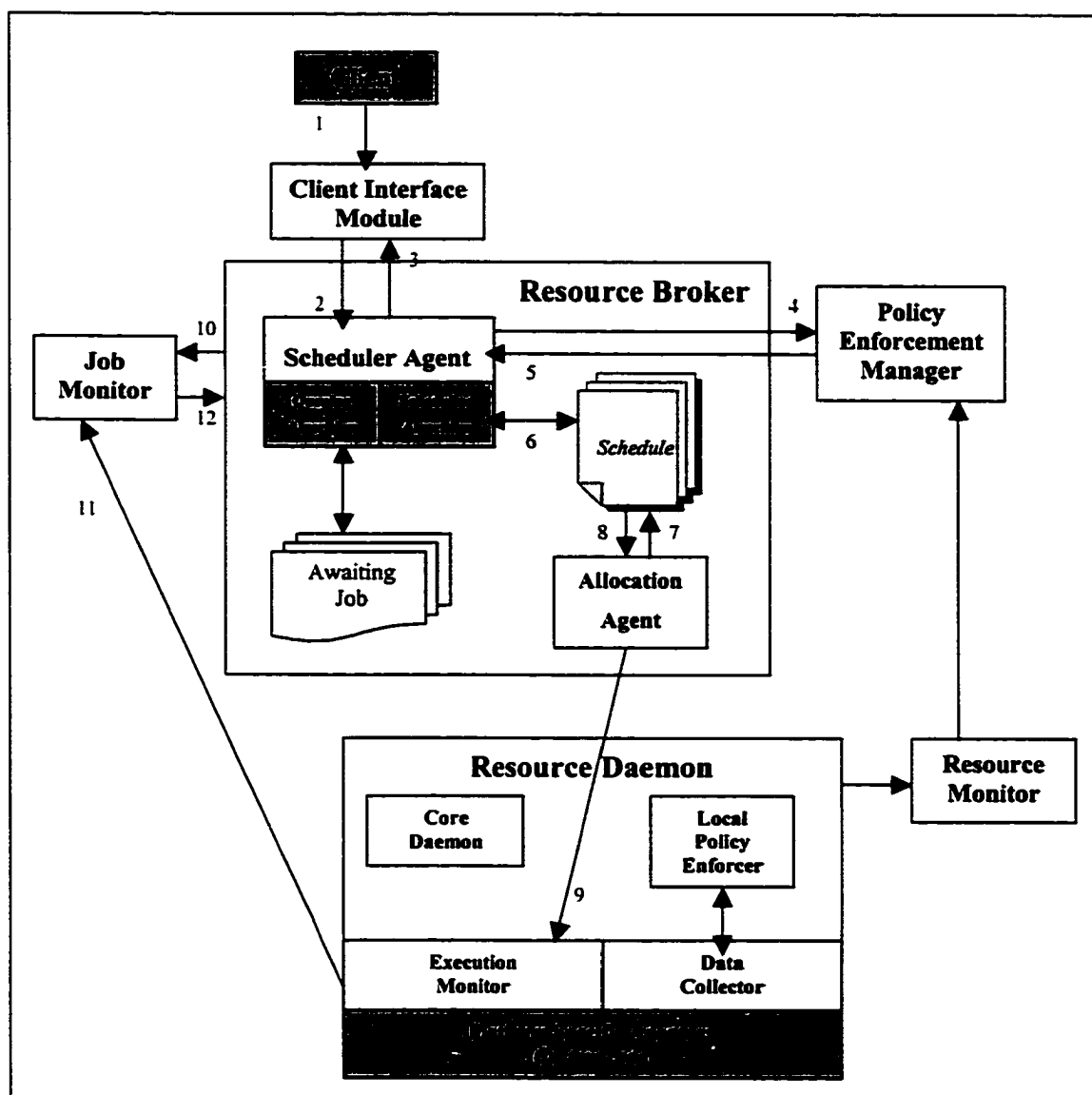


Fig. 63. Steps involved in successful execution of a Single Job

Given this set of 4 resources, the *Scheduler Agent* constructs the appropriate schedule. The underlying algorithm used for scheduling the DAG is a static CPM-based, which first assigns high priority tasks to the required resources. As each sub-task in the DAG gets allocated onto the designated resources, the *Job Monitor* is informed so that it can keep track of the job. After the successful completion of the OPTIMIZER sub-task, the schedule is terminated by the *Scheduler Agent*.

### **Experiment 2: Schedule cannot be created**

We set the underlying scheduling algorithm to be the static early assignment CPM-based for DAG application that we described earlier in chapter VI. As we explain, this algorithm requires that both the node weight and the path weight are known in advance. We submit a *DAG* application that does not satisfy these requirements. The *Resource Broker* denies the request after the underlying scheduling algorithm *isSchedulable* method returns *false*.

### **Experiment 3: Waiting/Rescheduling**

The *Resource Broker* maintains an internal queue of jobs currently in the system and that have not been scheduled yet including those that failed and needs to be rescheduled. We plug-in a First-In First-Out (FIFO) queuing algorithm that we described in the last chapter.

We submitted some simple *Single* tasks where no resource is available to accept that job. This is done by manipulating policies and running some consumers on the candidate resources. The *Resource Broker* holds the jobs in the awaiting queue. From time to time, it uses the FIFO queuing algorithm to select the next job to schedule. Once we terminate the consumers, the candidate matches become available and the *Resource Broker* starts rescheduling the awaiting jobs on a FIFO bases. Fig. 64 illustrates this experiment.

### **Experiment 4: Failure/Rescheduling**

We submitted a simple *Single* task, and it took its normal path throughout the *Resource Broker* till it gets allocated. We ran an *interrupter job* that caused our job to fail. Once the job failed, the *Job Monitor* was notified which in turn notified the *Resource Broker* about the change of the job's status. The *Resource Broker* put the job in the awaiting queue. When the queuing algorithm selected the job again, the *Resource Broker* rescheduled it. It then got allocated and finished successfully.

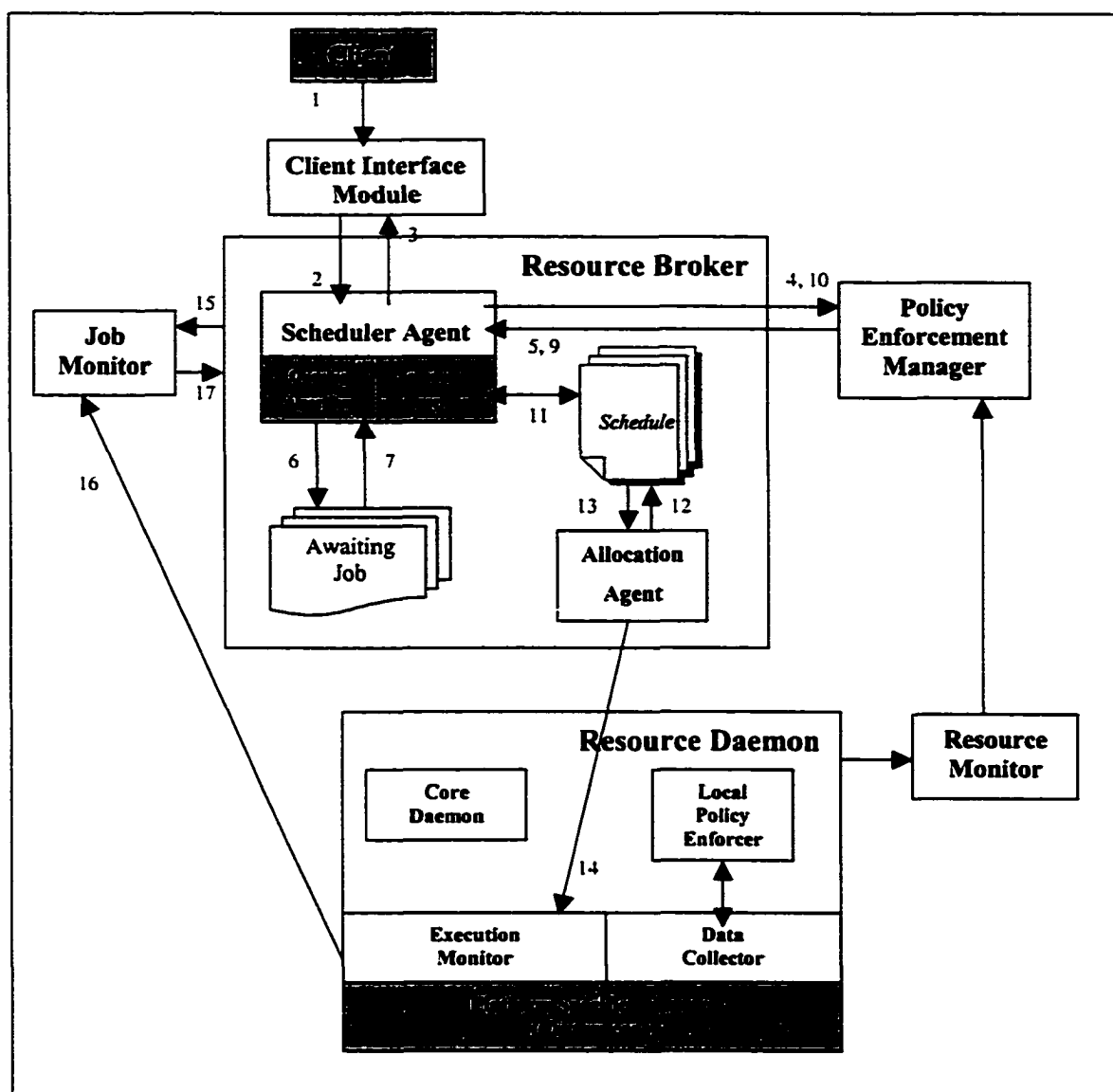


Fig. 64. Scenario of the waiting/rescheduling experiment

### Experiment 5: Job Monitoring

We tested different scenarios of job monitoring such as: canceling, stopping, resuming or retrieving the standard output/error. PROBE was able to handle these issues in an effective manner.



#### 7.4.1.2 Policy Framework

The two major functions that the *Policy Enforcement Manager* is tasked with are resource matching and assurance. In the previous subsection, we demonstrated the resource matching function via the different resource brokering experiments. In this subsection, we describe some policy-based experiments that we conducted to test our policy framework. These experiments include:

- SLA monitoring.
- SLA violation.
- Local policing.

##### Experiment 1: SLA monitoring

We submitted a simple Single job, which has one policy where no action is being specified in case of violations. Once the job is allocated, a Service Level Agreement (SLA) is established between the client and the resource provider based on the client's terms. The dissemination option in the allocated resources is *periodic Push* where the resource daemon is asked to update the *Resource Monitor* about the status of the resource every 30 seconds. Every time the *Resource Monitor* gets notified about the status of the resource, it notifies the *Policy Enforcement Manager* where the associated SLA is monitored as described below:

- associated SLA is fetched from the *Policy Keeper*.
- within that SLA, associated policy is parsed with the help of the *Policy Parser* and violation, if any, is detected.

Once the job is terminated successfully, the *Policy Enforcement Manager* is notified to terminate the associated SLA. Fig. 65 illustrates this experiment.

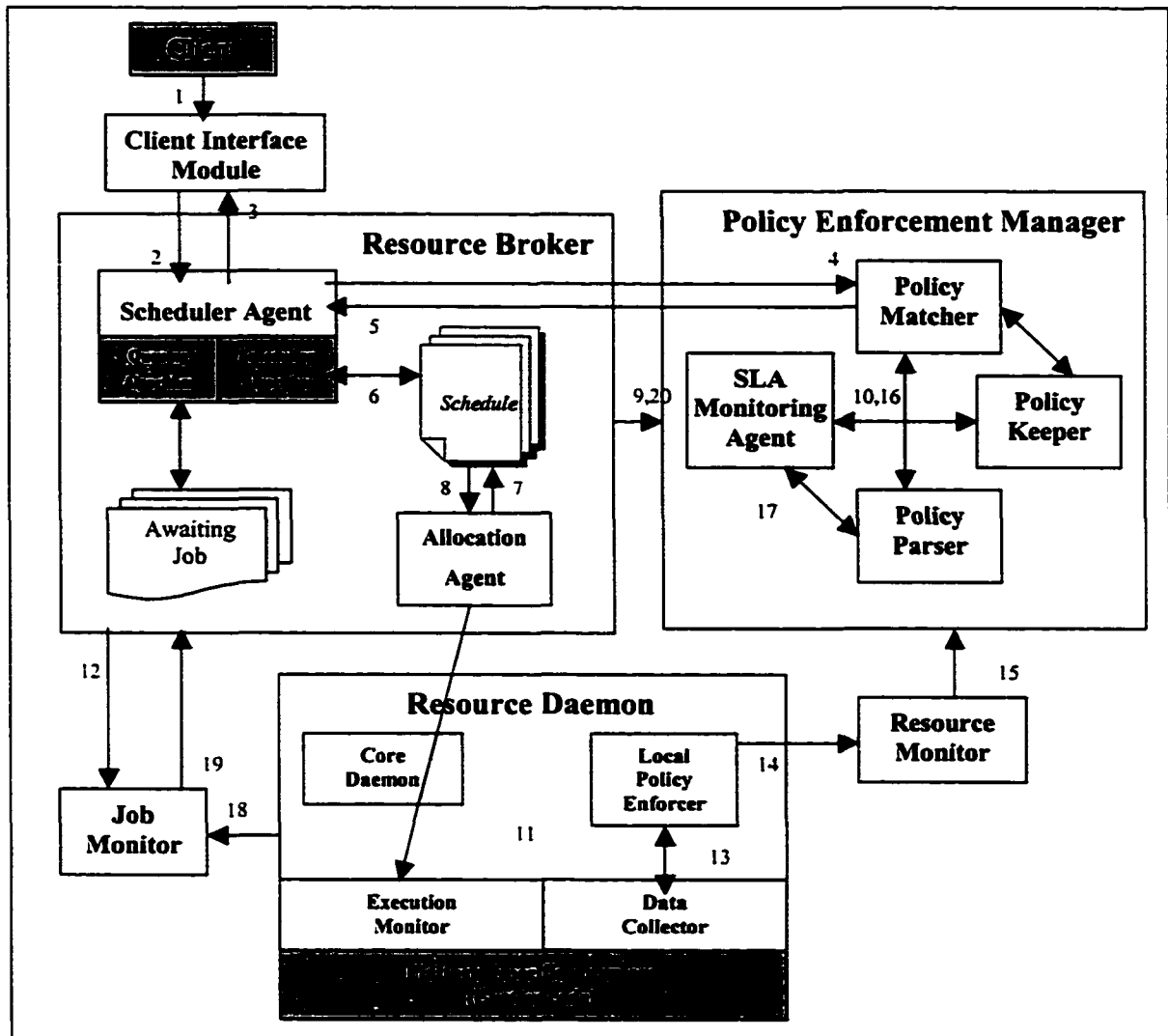


Fig. 65. Scenario of the SLA monitoring experiment

### Experiment 2: SLA violation

We ran the same scenario described in the previous experiment. This time, we specified a policy so that the available physical memory is at least 128 MB. We specify two actions to be taken in case of violation of allocation terms: one is to run a shell script, and the other is to send an e-mail to a pre-specified e-mail address. Once the job is allocated, we run a competing application on the assigned resource so that the guaranteed level of allocation is violated. Once the *Policy Enforcement Manager* was notified about the

status of the resource, the associated policies were evaluated which resulted in a violation. The two actions were then triggered.

### Experiment 3: Local Policing

Resource providers can specify some local policies internal to their resources to ensure that the appropriate action is taken before a violation occurs. These local policies can be specified on the fly using the *Resource Daemon Helper* utility as explained in 6.4.6. The API of the resource daemon is flexible enough to handle this issue.

We repeat the same scenario described above. Once the job is allocated, we specify a local policy so that when the free physical memory approaches 140 MB (warning level) a shell script should be triggered so that it kills some of the local jobs until the free physical memory reaches 160 MB (safe level). Again, we ran a competing application on the assigned resource in a manner so that the free physical memory drops below 140 MB. Once the data collection is due, the local policy is evaluated locally at the resource daemon, which results in a violation. The action associated with the local policy was triggered which resulted in the competing application being terminated.

TABLE 7  
SUMMARY OF THE QUALITATIVE EXPERIMENTS

Category	Experiment	Result
Ease of Deployment	Application types	Different application types have been implemented including Single, DAG and Aggregated.
	Scheduling algorithms	Two scheduling algorithms have been implemented. This includes: simple scheduling algorithm in which resources are assigned in First-Come-First-Serve (FCFS) bases; and <i>static EA-CPM</i> that yields assignment of high priority tasks.

Table 7, continued

	Queuing algorithms	A First-In First-Out (FIFO) queuing algorithm has been implemented.
	Repository adaptor	We developed an SQL repository adaptor using MySQL as background infrastructure.
	Daemon adaptors	Daemons for different platforms (Unix, Linux and Windows) and different grid systems (Globus and SGE) have been implemented.
	Action processors	<i>Shell</i> and <i>Email</i> action processors are supported.
Heterogeneity	System	System modules can run in heterogeneous environment in terms of different software and hardware platforms.
	Resources	PROBE can manage resources of heterogeneous types.
Site Autonomy		PROBE can handle different policies being applied at different sites and to resources within the same site.
Brokering	Scheduling technique	We plug-in the FIFS and the static EA-CPM scheduling algorithms via the command-line and on the fly using the <i>Plug-in Helper</i> utility. Scheduling has been tested using different kinds of applications.
	Queuing technique	We plug-in the FIFO queuing algorithm via the command-line and on the fly using the <i>Plug-in Helper</i> utility. Rescheduling of failed jobs and those that cannot be scheduled has been tested using this queuing algorithm.
	Schedule cannot be created	Jobs that cannot be scheduled due to missing information required by the scheduling algorithm are rejected.

Table 7, concluded

	Job cannot be scheduled	Jobs that cannot be scheduled are kept in the awaiting queue and then rescheduled.
	Failed jobs	Failed jobs are rescheduled.
	Stop	Job can be stopped at any time.
	Resume	Stopped jobs can be resumed.
	Cancel	Job can be cancelled at any time.
	Retrieve Output	Standard output and errors can be retrieved.
Allocation Assurance	SLA monitoring	SLAs are monitored in near real-time.
	SLA violation	Violations are captured as soon as they occur.
	Actions	Specified actions are triggered when violation occurs.
	Local policing	Local policies can be added on the fly and then evaluated in the appropriate way.

#### 7.4.2 Quantitative Experiments

The objective of the quantitative evaluation is to evaluate how effectively the prototype implementation of PROBE delivers the promise. In order to evaluate the effectiveness of PROBE, its performance needed to be evaluated under different scenarios. The brokering time is dominated by the following factors:

- *Parsing*, the time it takes the *Client Interface Module* to parse the supplied FJL-based request and construct the *Job* object.
- *Matching*, the time it takes the *Policy Enforcement Manager* to match resource(s) for the supplied request.
- *Scheduling*, the time it takes the *Resource Broker* to construct the appropriate schedule based on the underlying scheduling algorithm.
- *Allocation*, the time it takes the *Resource Broker* to implement the constructed schedule and allocate the associated tasks.

- *Communication*, the overhead of communication among the involved components.

We define the overall overhead of brokering that PROBE adds as:

$$\text{Brokering} = \text{Parsing} + \text{Matching} + \text{Scheduling} + \text{Allocation} + \text{Communication}$$

However, *Scheduling* time can vary from one algorithm to another and from one application to another. Similarly, *Allocation* time can vary based on the number of sub-tasks that need to be allocated and their allocated resources. In our brokering experiments, we use a very simple scheduling algorithm that we have implemented that assigns resources on a First-Come-First-Serve (FCFS) bases. Also, we run all the modules in the same machine so that the overhead of communication is relatively small. Thus, our quantitative evaluation consists of the following measurements:

- **Cost of XML parsing:** a major factor when PROBE interoperates with other grid systems is to efficiently parse the exchanged resource and request specifications. PROBE has some parsing tools where such specifications can be handled. In this experiment, we measure the cost of the parsing tools for both requests and resources.
- **Performance of resource matching and SLA monitoring:** to achieve high level of scalability and performance, PROBE caches some policy related information that it needs for resource matching and SLA monitoring. Internal cache reduces the cost of loading the data from the *Resource Repository* for each request. In this experiment, we measure the performance gained by caching for both resource matching and SLA monitoring. One drawback of caching, in general, is that one has to pay the price of the expensive use of memory. We measure the memory usage for the cached data as the underlying grid grows.
- **Overall overhead of brokering:** one of the main objectives behind this effort is to build an interoperable brokering infrastructure that acts as a mediator where a grid system can use PROBE to discover and use resources controlled by other

grid systems. We layer PROBE on top of Globus and Sun Grid Engine, the most-widely accepted grid systems in the grid community. We then measure the overhead due to these systems.

We conducted a number of experiments with different requirements to evaluate the performance of our framework. All these experiments were performed on our experimental testbed. We make sure that allocated tasks are not interrupted by other users. All the times are based on at least five measurements. Below, we discuss in detail each of these experiments. The numerical data for all the experiments is given in Appendix A.

#### 7.4.2.1 XML parsing

PROBE provides two XML parsing utilities, the *ResourceParser* and the *RequestParser*. These utilities provide convenient APIs for creating, manipulating, and checking the validity of a resource and request specifications respectively.

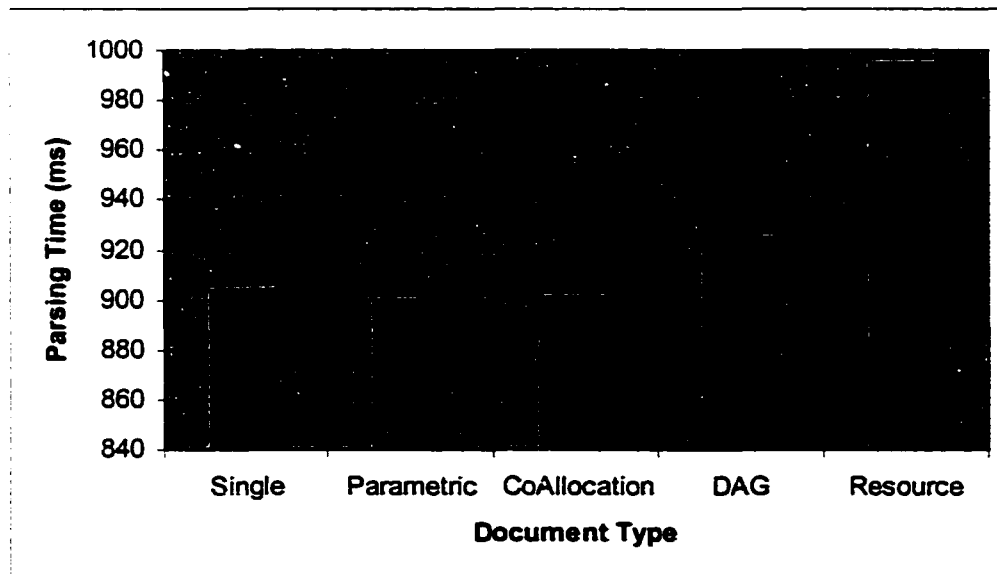


Fig. 66. Parsing time for different XML document

To measure the performance of our XML parsing utilities, we ran different experiments in which we parsed different kinds of documents that we have proposed. We

ran our experiments on a Sun workstation with 750 MHz processor, 1024 MB of RAM and Solaris 2.8. We used Sun's JDK 1.4 and Sun's JAXP XML parser 1.2. We parsed using SAX 2.0 with validation turned on. We measured the time it takes to parse each XML document. The result of the experiment is shown in Fig. 66. The raw data is given in (Appendix A – Table 11).

#### 7.4.2.2 Performance of resource matching and SLA monitoring

In this experiment, we measured the performance gained by using caching versus loading the data from the *Resource Repository* for both request matching and SLA monitoring. We applied different data retrieval approaches including:

- *Caching*, where the *Policy Enforcement Manager* relies on the data that it internally caches.
- *Local Resource Repository*, where the *Policy Enforcement Manager* consults with a MySQL-based *Resource Repository* installed on the same machine.
- *Remote Resource Repository*, where the *Policy Enforcement Manager* consults with a MySQL-based *Resource Repository* installed on a remote machine connected via fast Ethernet (100 Mbps).

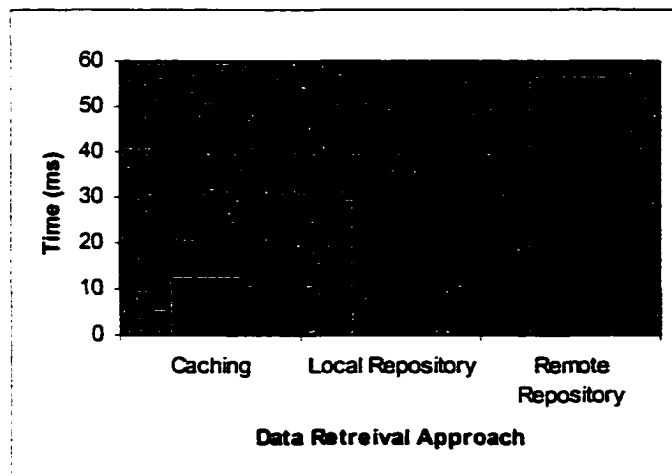


Fig. 67. Performance of Resource Matching under different data retrieval approaches



We performed our experiments on a Sun workstation with 750 MHz processor, 1024 MB of RAM and Solaris 2.8. We submitted the Single job described earlier in this chapter and discovered that the matching process, on average, takes 11.8 ms when values were cached, 34.8 ms when we rely on a local *Resource Repository* and 55.6 ms when we rely on remote *Resource Repository*. This implies that caching provides a factor of 2.95 performance gain compared to the local *Resource Repository* and 4.71 compared to the remote *Resource Repository*. Fig. 67 illustrates the results that we obtained from this experiment. The raw data of the figure is given in (Appendix A, Table 12).

We also performed similar experiments to measure the performance of the SLA monitoring process. As shown in Fig. 68, we found that on average, it takes 2.4 ms to monitor an SLA when values were cached, 24.2 ms when we rely on a local *Resource Repository* and 41.4 ms when we rely on remote *Resource Repository*. Thus, caching provides a factor of 10.08 performance gain compared to the local *Resource Repository* and 17.25 compared to the remote *Resource Repository*. The data material of the figure is enclosed in (Appendix A, Table 13).

The runtime performance when policy-related information is not cached leads to poor and unacceptable resource matching and SLA monitoring times that make the system almost unusable for large grids.

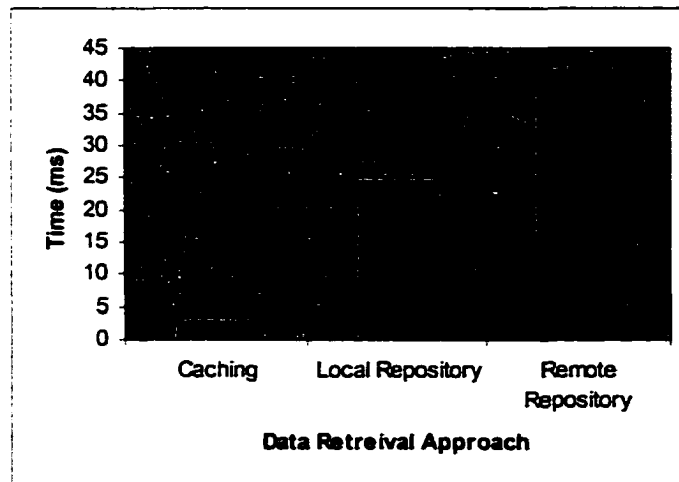


Fig. 68. Performance of SLA Monitoring under different data retrieval approaches

One drawback of caching, in general, is the price of memory usage. We analyze the memory usage for different kinds of grids. We divided our experiments into 3 kinds of grids:

- *Small Grid*: this grid is typical of small organizations where resources range from 10 to 90.
- *Medium Grid*: this grid is typical of many administrative domains where resources range from 1,000 to 9,000.
- *Large Grid*: this grid has a massive number of organizations, possibly on different continents, where resources range from 10,000 to 90,000.

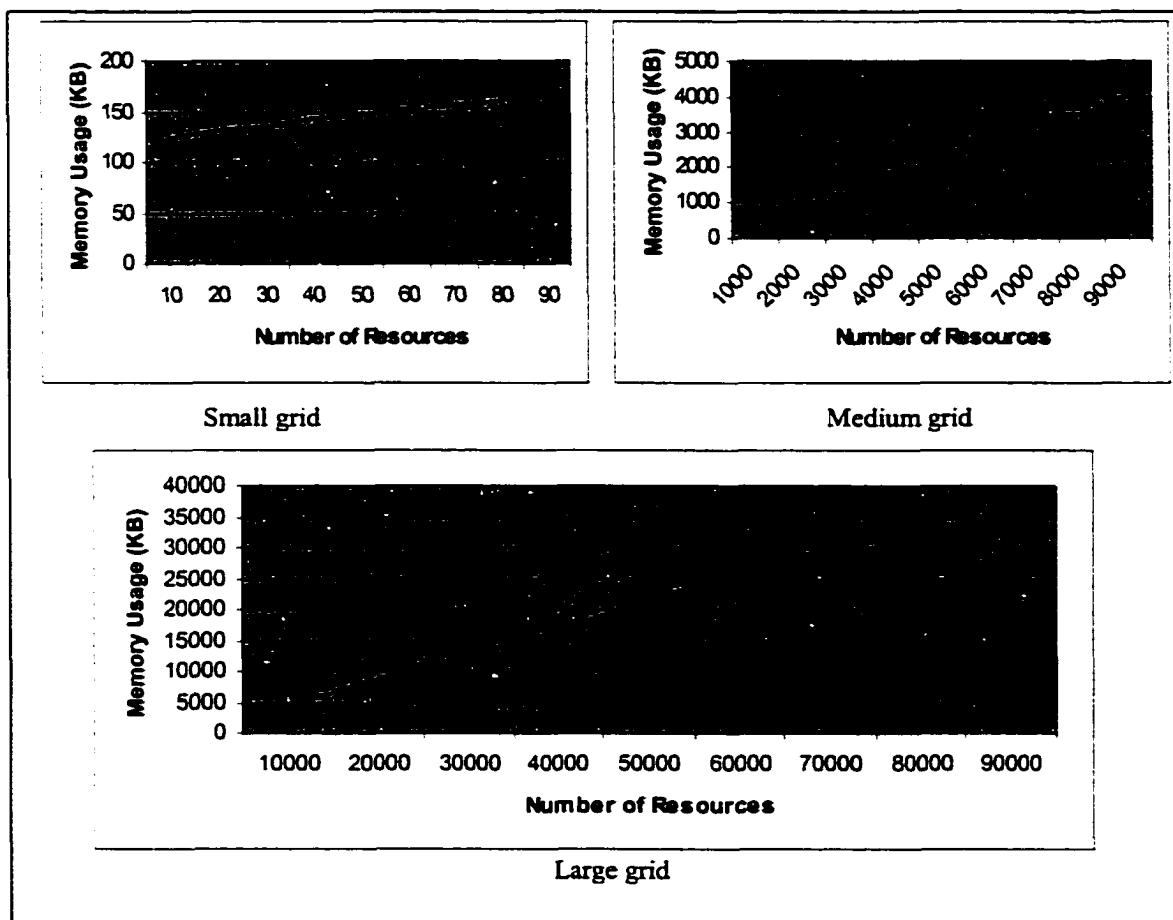


Fig. 69. Memory usage for different kinds of grids where no SLAs are applied

The *Policy Enforcement Manager* caches two kinds of data: resource status, associated policies and client's SLAs. In order to simulate the massive number of resources that *Medium* and *Large* grids require, we developed a simulation application where we can simulate such huge numbers of resources. We apply average sizes for names and policy scripts. For example, we used 15 characters as the average length of the resource name and 20 characters as the average length of the policy string for both resource and SLAs.

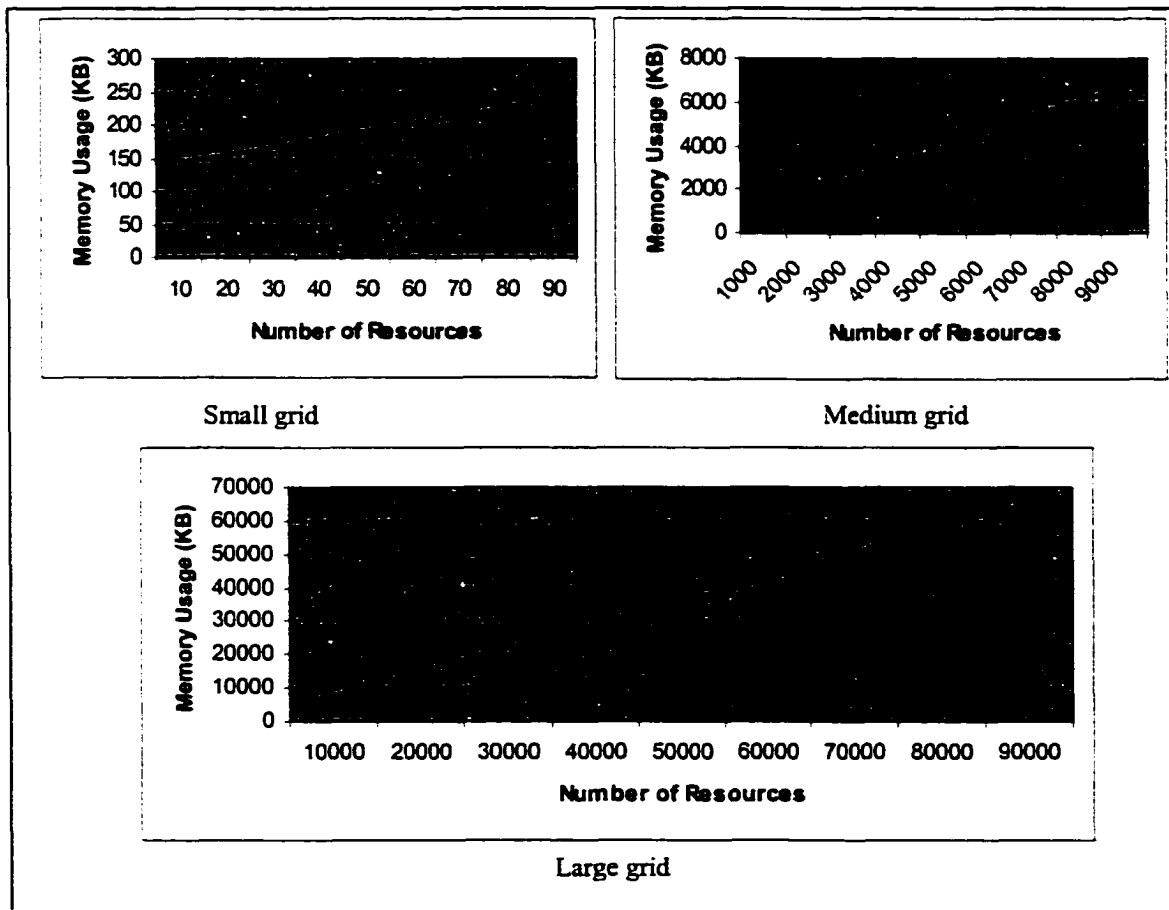


Fig. 70. Memory usage for different kinds of grids with an average of five SLAs per resource

We ran two different simulations. In the first one, we monitored resources where no SLA is applied. The result of the experiment is shown in Fig. 69. On the average, the

memory usage increases by almost 0.4 kilobyte (KB) for each added resource. The data material of the figure is enclosed in (Appendix A, Tables 14-16).

In the second simulation, we applied five SLAs for each resource assuming that all resources are occupied and each one has five allocated tasks. The result of this experiment is shown in Fig. 70. On the average, the memory usage increases by almost 0.7 KB for each added resource. The data material of the figure is shown in (Appendix A, Tables 17-19).

### 7.4.2.3 Overall overhead of Brokering

In order to measure the overall overhead of brokering and to avoid other factors that might affect our measurements, we ran all the brokering experiments that we describe in this section on the same machine. We designed a *Single* job representing a shell script that sleeps for 100 seconds (100,000 ms) from this experiment. We submitted the problem to different execution environments including: Globus, Sun Grid Engine (SGE) and PROBE layered on top of each one of these systems. The times quoted in this experiment are the total elapsed time from when the client submits the request to PROBE until the job is finished and its schedule gets terminated. As all the components run on the same machine, the overhead of communication is relatively small.

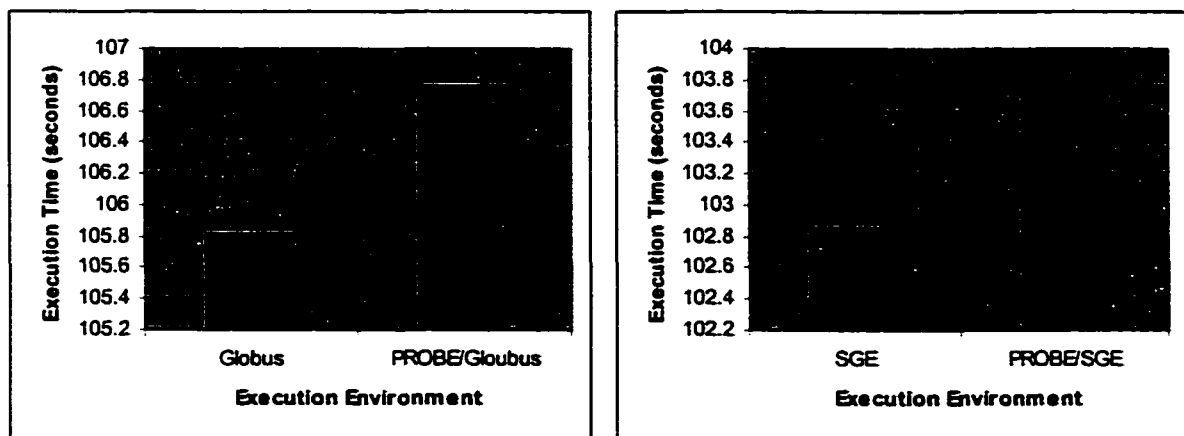


Fig. 71. Completion time of a 100 seconds job under different execution environments

Fig. 71 shows the completion time, in seconds, under different execution environments. The runtime overhead in the SGE environment is approximately 2776 ms. The overhead of brokering is increased with the Globus environment, 5809 ms, because of the need to parse the Resource Specification Language (RSL) request and authenticate that request, while in the case of the SGE, the submitted request is directly executed.

However, when we repeated the same experiments using PROBE, it added 943 ms in the case of the Globus system and 997 ms in the case of the Sun Grid Engine. The raw data of the figure is given in (Appendix A, Table 20)

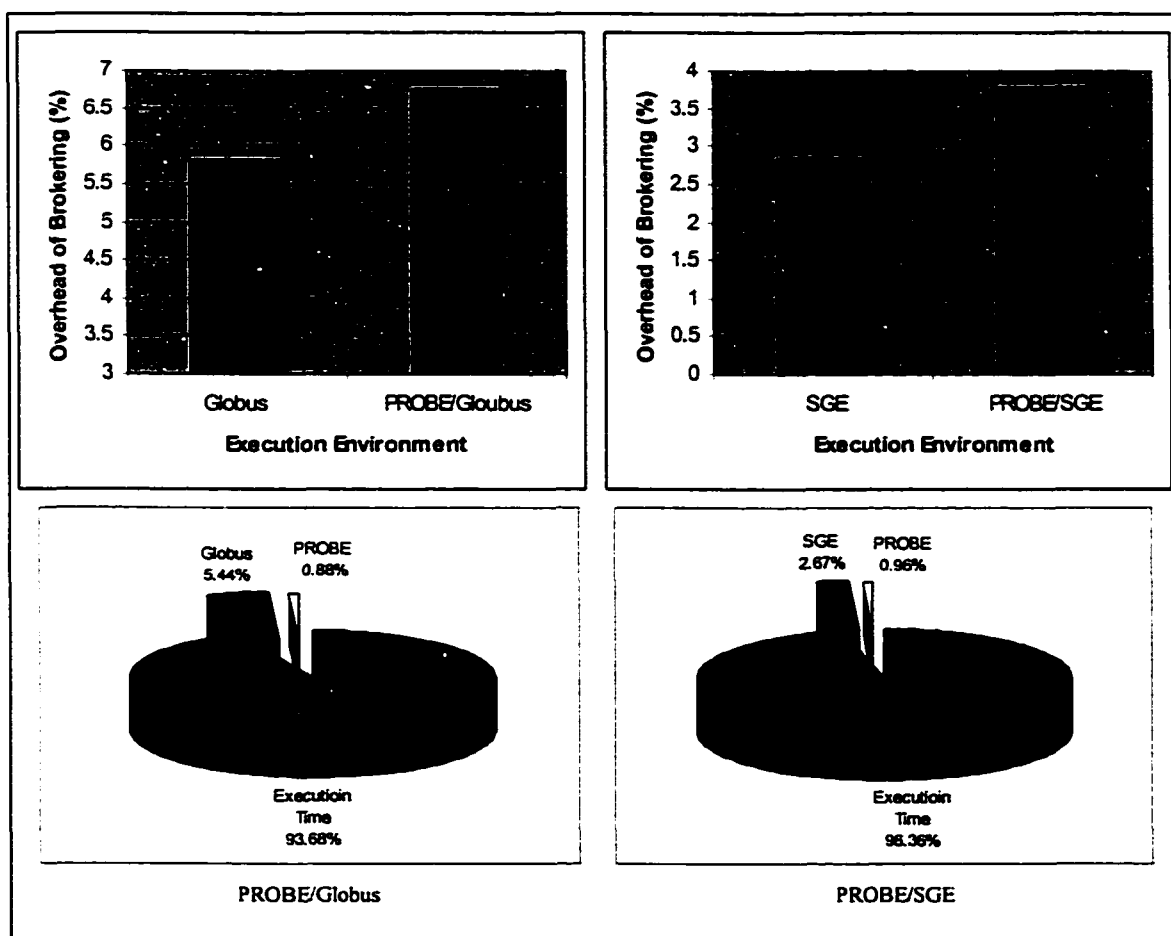


Fig. 72. Brokering overhead of a 100 seconds job under different execution environments

We also performed a brokering experiment with different job sizes in the PROBE/Globus execution environment. As shown in Fig. 73, as the size of the job

increases, the overhead of the brokering decreases exponentially. Since PROBE is targeted to large applications executing on grid systems, the overhead of brokering should be acceptable. The raw data of the figure is given in (Appendix A, Table 21).

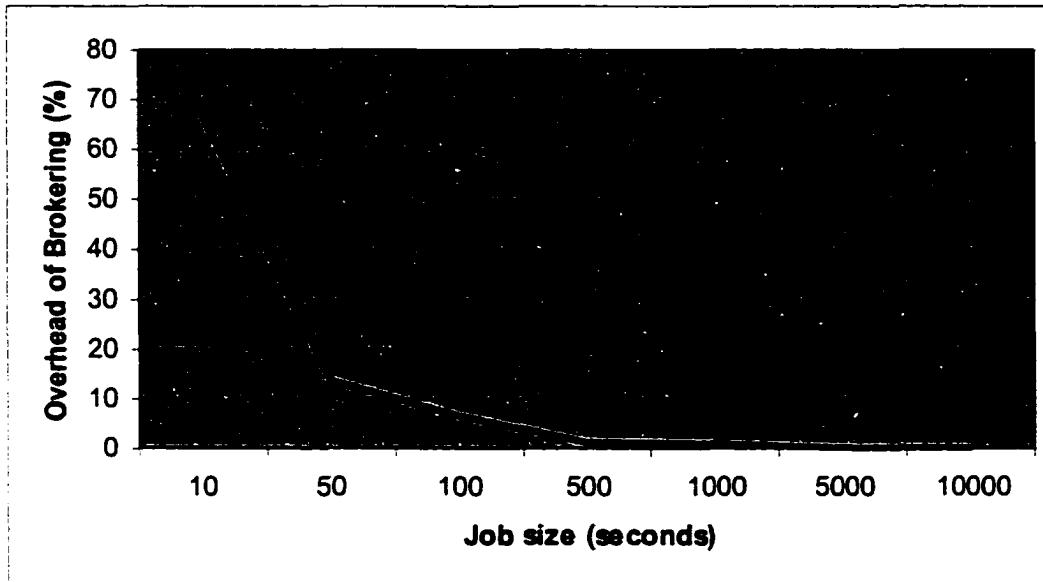


Fig. 73. Brokering overhead for different job sizes under the PROBE/Globus execution environment

## 7.5 Conclusion

In this chapter, we have presented the evaluation of the PROBE prototype implementation. We described the PROBE Computational Grid (PCG) experimental testbed and presented the results that we have obtained when the PROBE framework is applied in the context of different scenarios.

We divided our evaluation into two parts: *qualitative evaluation*, in which we demonstrated that the system delivers what it promises in terms of functionalities and characteristics; and *quantitative evaluation*, in which we tested the performance of the system.

In the qualitative evaluation, PROBE has been tested, running in a heterogeneous environment in terms of software and hardware where we did not encounter any problem.

We started with the basic skeleton of PROBE. Then, we showed how we can apply different plug-ins. We tested PROBE within multiple administrative domains where not only each administrative domain but also each resource owner identifies its own policies. We tested the different functionalities of our brokering infrastructure and the policy framework.

We have analyzed the performance of the PROBE prototype implementation. A major benefit of caching is to decrease the matching time and the SLA monitoring time. We have compared the performance of different data retrieval approaches. Results show that caching adds significant performance improvement. Our experiments show that the average matching time using caching is 3 times faster than that of one not using caching for resource matching and 10 times faster for the SLA monitoring. Thus, caching does decrease response time and improves the overall performance. Examples of small, medium and large grids have been presented. From these examples, it is easy to see that on the average the memory usage increases by almost 0.4 KB for each added resource and by 0.7 KB for each added resource when five SLAs, on the average, are assigned.

Interoperability is one of the main objectives behind this effort. Our brokering infrastructure can act as a mediator where a grid system can use to discover and use resources controlled by other grid systems. We layer our system on top of Globus and Sun Grid Engine, the most widely accepted grid systems. We found that the overhead added by our system is relatively small compared to the functionality it provides.

These experimental results demonstrate the effectiveness of our technique and the applicability of PROBE as a general-purpose resource brokering environment.

## **CHAPTER VIII**

### **CONCLUSIONS AND FUTURE WORK**

In this chapter, we draw conclusions from the work presented in this thesis and offer some suggestions for further improvements and extensions.

#### **8.1 Conclusions**

Computational grids are evolving and are becoming a basic infrastructure for the future of high performance and distributed computing. A critical component in such an environment is the resource brokering environment that mediates and controls the access and use of the underlying resources. Issues such as distribution, site autonomy and resource heterogeneity complicate the task of the resource brokering environment. Several research groups are implementing resource brokering environments for grid systems. Based on our review, we conclude that these systems are either specific to a particular grid environment or have limited features that make them unsuitable for large applications with heterogeneous requirements, and make interoperability with other grid systems big concern. In addition, the issue of allocation assurance to users who are looking for the satisfaction of the job's requirements during the lifetime of the allocation, has not been addressed by most of these brokering efforts.

The work presented in this thesis focuses on the problem of providing a general-purpose brokerage infrastructure for computational grids that is flexible enough to be utilized on various grid systems. Several contributions towards the resolution of this problem have been made. In this thesis, we have discussed the analysis, design, implementation and evaluation of PROBE, a framework of a policy-based resource brokering infrastructure for computational grids that addresses this problem.

PROBE enables grid systems to evolve and expand. It has well-defined APIs that can be utilized by grid environments to develop their brokering tools. The layered approach, façade design pattern and the APIs give grid systems the flexibility to adopt different approaches as their environments require.



Similarly, the policy-based approach provides one means of attracting grid users and contributes to establishing credibility for existing grid environments by committing to provide the guaranteed level of allocation with the right action (compensation, credit, etc) if such guarantees are not met.

We have described a testbed for our experiments to evaluate PROBE with respect to ease of use, deployment and performance. Interoperability is one of the main objectives of this effort. PROBE can act as a mediator where a grid system can use it to discover and use resources controlled by other grid systems. We layer PROBE on top of Globus and Sun Grid Engine, the most widely accepted grid systems. We observed that the overhead PROBE adds is relatively small compared to the functionality it provides.

However, the problem of having a generic brokering infrastructure is by no means completely solved. The remainder of this chapter presents some future directions for research in this area.

## 8.2 Future Work

There are several areas of research that can be further explored. One of the recent research directions is to apply economic principles to resource brokering. In computational economy, grid users want to minimize the “cost” of their computation whereas resource owners want to maximize their “profit”. This has been an active area of research recently. Buyya [21] has proposed an economic-based model for the grid. Others such as Java Market [11] and Popcorn [91] have models that are limited to specific environments. The *Resource Broker* can be extended to adopt economic-based scheduling policies via the *Policy Enforcement Manager*.

Also, for efficient scheduling of resources, it is more useful for PROBE to use an estimate of the performance in the near future rather than current performance. Based on historical performance information, PROBE should be able to predict the performance each resource is going to deliver at the time of the allocation. This could result in a more efficient scheduling of the resources. Thus, another direction for future research is to extend the model of PROBE given in this thesis to handle predictions. As we describe in

section 8.3, a new module, called *Predictor*, can be introduced for that purpose. The *Predictor* is going to keep historical performance information and predict future performance. Work plan to interface with the Network Weather Service (NWS) [119], a Distributed Resource Performance Forecasting Service for computational grids by the University of Tennessee.

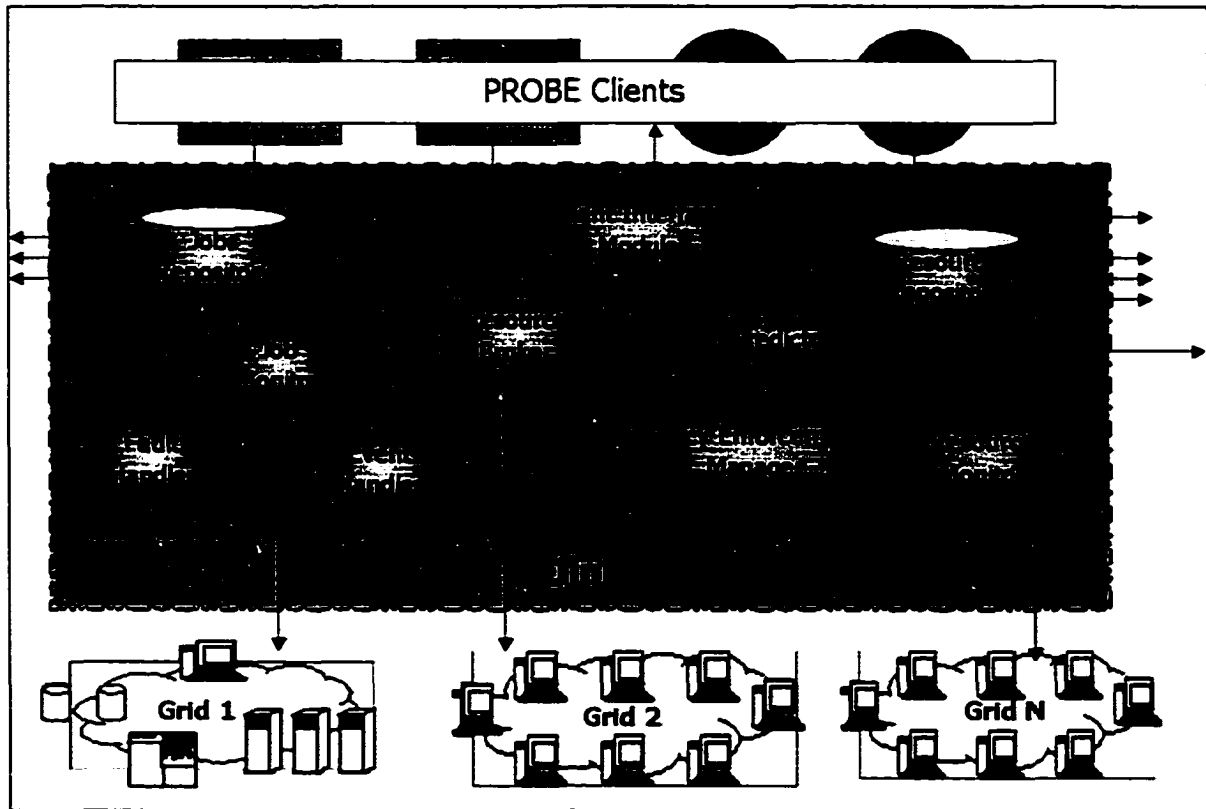
Peer-to-peer (P2P) computing is an evolving approach to distributed computing where each participant can be both a client and server. In the past few years, several P2P systems have been widely used, especially Napster [90] and Gnutella [49]. Recently, there has been an interest towards building P2P-based grid environments. Both P2P and grid technologies focus on the flexible and innovative use of heterogeneous resources distributed across networks. As a result, many of the challenges and standards are closely related. Recently, the Global Grid Forum (GGF) joined forces with the P2PWG [97] to combine efforts. A Peer-to-Peer area is being formed within the Global Grid Forum. Also, as we explained in chapter II, JXTA [99] is an open research project by Sun Microsystems that provides a P2P-base infrastructure for distributed computing applications. JXTA is independent of the transport protocol where implementation can be done over TCP/IP, HTTP, etc. We believe that JXTA is going to play an important role in building infrastructure for P2P grids. However, security and efficient message passing is still a big concern. We are currently investigating on having a P2P version of PROBE.

### 8.3 PROBE Extensions

As we have stated earlier in chapter II, our modular approach, the well-defined APIs and the layered architecture make it easy to extend the system to handle future needs. In this subsection, we describe a proposed extension to our brokering infrastructure. We propose three additional modules:

- *Predictor*: predicts future performance of resources based on historical performance information that is provided by the *Resource Repository*. It also provides a gateway to other prediction tools.

- **Fault Handler:** handles heartbeat monitoring of the underlying resources as well as the PROBE's modules and achieves fault-tolerance.
- **Event Handler:** handles scheduled brokering events within the system.



**Fig. 74. Architecture of Extended PROBE**

Fig. 74 illustrates the architecture of the extended PROBE. Both *Fault Handler* and *Event Handler* are expected to interact with all the other components and shown in multidirectional arrows. Below, we give a brief description about these proposed modules.

### 8.3.1 Predictor

Archived performance data can be used to predict the behavior of the resource in terms of the performance that it is going to deliver in the future. The *Predictor* module is going to

summarize this historical data and based on the underlying prediction technique, the *Predictor* can forecast what the resource is going to deliver in the near future.

When the *Policy Enforcement Manager* tries to find the appropriate resource(s) that can match the client's request, it would rely on the summarized data being generated by the *Predictor* so that it can match the best resource(s). Prediction is going to help in minimizing SLA violations and thus reduce the resulting penalties a resource provider has to pay in case of violations. Fig. 75 illustrates the prediction process.

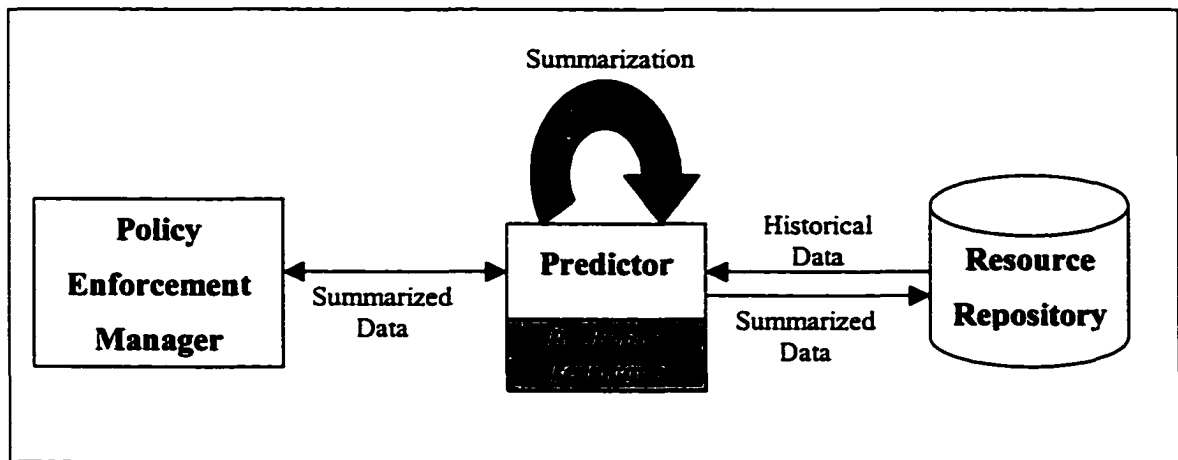


Fig. 75. The Prediction Process

We are going to investigate in standard statistical prediction techniques such as means, medians and autoregressive for use in prediction. We also are going to study existing prediction tools [92],[120] and see how we can interface them with PROBE.

### 8.3.2 Fault Handler

PROBE needs to be fault tolerant with respect to the failure of its internal components. The side effects caused by the failure of any component should be as low as possible minimizing the drop in the performance of the system. On the other hand, a failure can happen at any time due to a hardware, software or network problem such that the resource becomes unavailable. PROBE has to keep track of all the available resources

and be aware of the failures as soon as they occur. The *Fault Handler* module is the one that is going to handle fault tolerance issues.

The *Fault Handler* module should provide a simple mechanism for monitoring the status of the distributed set of resources and modules of the system and handle faults as they occur. Each service, acting on behalf of a resource or a module, generates a periodic “I am alive” message. It also provides an API where the *Fault Handler* can register to receive such a message. This API could be implemented on top of Jini’s event notification programming model.

The *Fault Handler* expects to receive the periodic frequent “I am alive” message from the modules/resources that it tracks. However, there is no guarantee of receiving the message as it may be lost, delayed or a failure may have occurred. Thus, if the message is not received and the *Fault Handler* times out, it will examine that resource/module and based on some set of useful failure-mode assumptions, it will determine whether or not the resource/module has failed. Investigation needs to be done on efficient failure-mode assumptions and how to handle them.

In case of a failure, the *Fault Handler* will inform the components that have an interest in such failure. Also, it will keep track of the number of failures. In case of a module failure, the *Fault Handler* will keep trying to restart the module a prespecified number of times before assuming its failure.

Modules can detect failures when they try to contact each other or when they try to contact the resources. The *Fault Handler* should provide an API where clients can report faults of other services. It should also provide an API where they can get information about other’s faults.

Since the *Fault Handler* module may become a bottleneck, it may be replicated. As we have stated before, the modular approach gives us high availability by allowing multiple instances of the *Fault Handler* as well as other modules to be instantiated on distributed hosts. *Fault Handlers* could keep a watch on each other as well as watching their fellow modules/resources. When a service, representing a module or a resource, is started, it has the option to notify one or more *Fault Handlers*, based on its importance.

*Fault Handlers* will continuously exchange information about the states of services in order to maintain a consistent view of the system.

Checkpointing, process migration and recovery procedures will be needed for fault tolerance. Application-specific fault recovery mechanisms can be built on top of the *Fault Handler* or other modules of the system as required. For example, a component that caches information such as the *Policy Enforcement Manager* could provide some recovery procedures to restore the cached data in case of a failure.

### 8.3.3 Event Handler

Some grid users might prefer to schedule tasks on a regular basis or based on some conditions. System administrators on the other hand might need to schedule some administrative tasks internal to the system. We propose a new module, *Event Handler*, to handle such events. Events could be:

- Periodic: *run my simulation every Sunday.*
- Conditional: *after 5 pm, use scheduling algorithm B.*

The *Event Handler* has been inspired by the task manager that Microsoft Outlook supports. Recurrence options could take similar form to the ones the Microsoft Outlook supports as shown in Fig. 76.

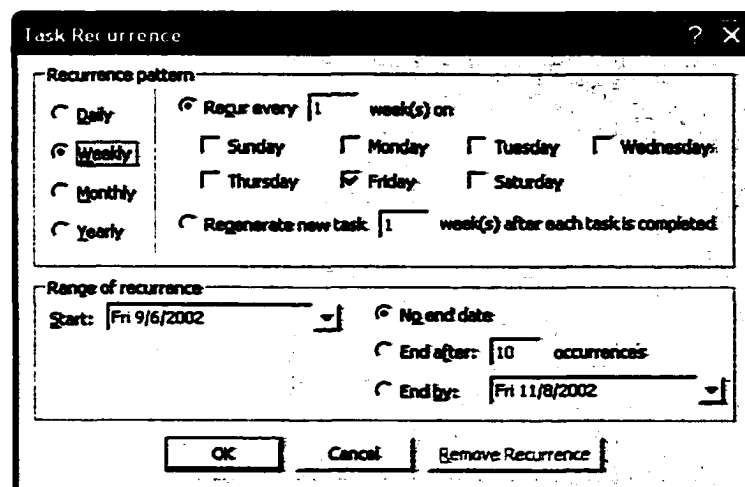


Fig. 76. MS Outlook recurrence window

The *Event Handler* needs to have a flexible API where events can be manipulated. An XML-based specification will be provided where events can be specified and exchanged with other systems.

## 8.4 Enhancing Jini to support Scalability

### 8.4.1 Overview

Most of the existing distributed computing technologies such as CORBA, DCOM, EJB, RMI and Jini provide the infrastructure necessary to build a scalable distributed system. Although they all give the flexibility of replicating and distributing the different components of the system, none of these technologies has gone further by providing some scalability features such as keeping track of the replicated component and achieving load balancing among them.

On the other hand, most existing distributed computing environments want to take advantage of the growing network infrastructure. To achieve higher scalability, a service is replicated and a load-balancing agent is added to keep track of the replicated services. When requested, the load-balancing agent provides the client with the most appropriate service based on a given load-balancing technique.

Each distributed technology has some version of a *Naming Service*, e.g., *Lookup Service* in Jini, which serves as a repository of services available in the underlying distributed system. We feel that the *Naming Service* is the appropriate place to embed scalability logic where the *Naming Service* can act as a load-balancing agent keeping track of the distributed replicated services and when requested provide the client with the appropriate service.

In section 6.2, we described an enhancement that we have done to enable Jini across networks that do not support multicasting. In this subsection, we describe a proposed enhancement for Jini that allows it to provide an embedded scalability solution to distributed applications. The multicasting enhancement described earlier along with the scalability enhancement will allow Jini to scale up to the level of the Internet.

### 8.4.2 Proposed Solution

Since Jini has the advantage of the open source, we would like to make use of this feature and embed the scalability enhancement within Jini. The main objective is to make enhancements to Jini that are compatible with its functionality. Thus, we would like the scalability enhancements to be active in the background without making any changes, as far as possible, to the behavior of the clients and services.

We propose to have a scalability feature embedded within the Lookup Service (LS). The LS is going to keep track of the replicated services, within its domain, along with their loads. Based on the underlying discovery protocol and the load-balancing algorithm, the different LS are going to consult with each other and provide the client with the appropriate service instance.

As illustrated in Fig. 77, each service is going to inherit an abstract *Load* class and implement the *computeLoad* abstract method where its load can be computed frequently based on a given work-load algorithm. This can vary from one module to another. For example, the number of connected clients, cost of parsing of the XML-based request can be critical factors in measuring the load of the *Client Interface Module*. Load will be measured in the scale of 0-100 units, where 0 means an idle module and 100 means a fully-occupied one.

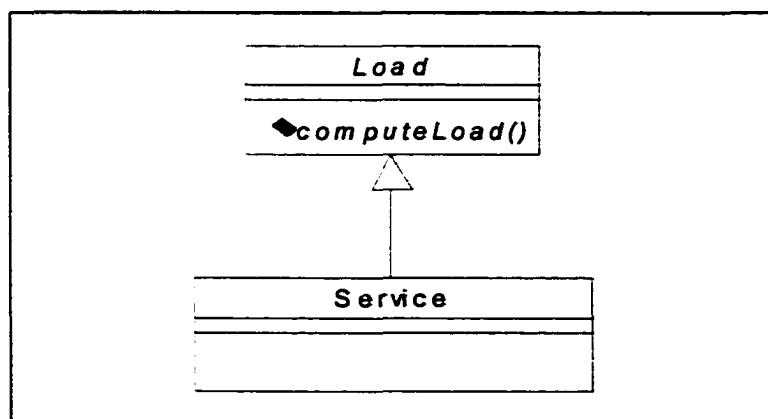


Fig. 77. Implementing the *Load* class



Requests need to be distributed over the replicated services to avoid scalability bottleneck. To ensure that the load is balanced among the replicated services, a load-balancing algorithm is used to decide which service should be given a particular unit of work. Common algorithms include server-load, round robin, random, weight-based, network-response time and user-specific algorithm [32].

In order to decouple the LS from the underlying load-balancing algorithm, a facade object will be introduced to shield the LS from the load-balancing algorithm. As shown in Fig. 78, *LoadBalancingAlgorithm* is an abstract class and needs to be implemented by the provided load-balancing algorithm. It should support a set of methods where services can be manipulated. These methods are:

- *addService*, where a service can be added to the list of services.
- *deleteService*, where an existing service can be removed from the list of services.
- *getNextService*, where the next available service can be retrieved.

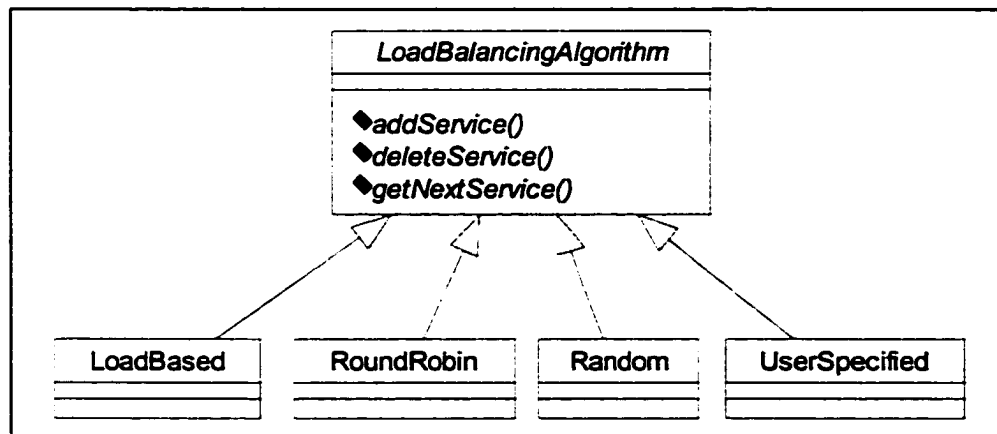


Fig. 78. Implementing the Load-balancing algorithm

The LS will have a unified interface to a set of load-balancing algorithms. This makes the design independent of any load-balancing algorithm. Initially, we plan to support the following load balancing algorithms:

- **Load-based**, where the algorithm favorites certain services based on the load.

- **Round Robin**, where the algorithm cycles through the list of services in order.
- **Random**, where the algorithm chooses the next service randomly.

### 8.4.3 Scenario

In this subsection, we describe a typical scenario that will occur when we apply this load-balancing enhancement in the context of the Jini infrastructure. The sequence of operations are illustrated in Fig. 79.

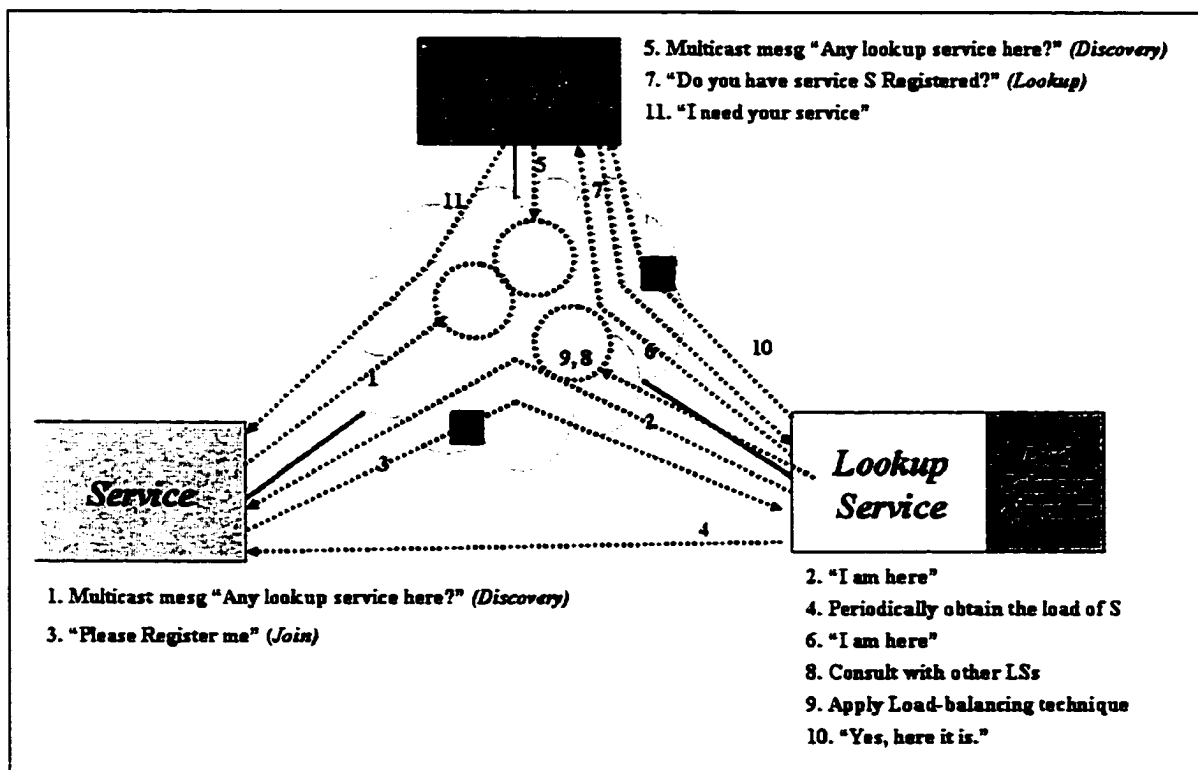


Fig. 79. Scenario of the load balancing process

- When a service starts, it registers with the Lookup Service (LS) offering service S. Multiple instances may be started offering the same service. This could be in the same LS or across different LSs.

- Each service computes its load frequently based on a given work-load algorithm. The LS keeps track not only of the services but also their loads. LS periodically probes services by calling the *computeLoad* method in order to get the up-to-date load.
- Client sends a message to the LS(s) requesting service S. Based on the discovery protocol and load-balancing algorithm in use, the LSs respond by consulting with each other and the LS that holds the appropriate service (least load in case of load-based algorithm) will respond.
- Thereafter, the client communicates directly with the service.

## REFERENCES

- [1] D. Abramson, J. Giddy, I. Foster, and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 520-528, Cancun, Mexico, May 2000.
- [2] D. Abramson, R. Buyya and J. Giddy, "Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid", *The HPC-Asia 2000*, pp. 283-289, Beijing, China, May 2000.
- [3] D. Abramson, R. Sasic, J. Giddy and B. Hall, "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", *Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing*, pp. 112-121, Virginia, August 1995.
- [4] I. Ahmad, "Resource Management of Parallel and Distributed Systems with Static Scheduling: Challenges, Solutions and New Problems", *Concurrency: Practice and Experience*, vol. 7, no. 5, pp. 339-348, August 1995.
- [5] I. Ahmad, "Resource Management of Parallel and Distributed Systems: Dynamic Scheduling", *Concurrency: Practice and Experience*, vol. 7, no. 7, pp. 587-590, October 1995.
- [6] J. Almond and D. Snelling, "UNICORE: Uniform access to supercomputing as an element of electronic commerce", *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 539-548, October 1999, NH-Elsevier.

- [7] A. Al-Theneyan, P. Mehrotra and M. Zubair, "A Resource Brokering Infrastructure for Computational Grids", *To appear, Proceedings of the 9th International Conference on High Performance Computing (HiPC 2002)*, Bangalore, India, December 2002.
- [8] A. Al-Theneyan, A. Jakatdar, P. Mehrotra and M. Zubair, "XML-Based Visual Specification of Multidisciplinary Applications", *The First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2001)*, Brisbane, Australia, May 2001. *Published in Future Generation Computer Systems based on best papers from CCGrid2001*, vol. 18, no. 4, pp. 539-548, March 2002, NH-Elsevier.
- [9] A. Al-Theneyan, P. Mehrotra and M. Zubair, "Enhancing Jini for Use Across Non-Multicastable Networks", *Proceedings of the First Saudi Technical Conference and Exhibition*, vol. II, pp. 18-23, Riyadh, Saudi Arabia, November 2000. Also in *Operating Systems Review*, ACM SIGOPS, vol. 35, no. 2, pp. 21-30, April 2001.
- [10] E. Akarsu, G. Fox, T. Haupt, A. Kalinichenko, K. Kim, Sheethaalnath and C. Youn, "Using Gateway System to Provide a Desktop Access to High Performance Computational Resources", *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, California, August 1999.
- [11] Y. Amir, B. Awerbuch, R. Borgstrom, "A Cost-Benefit Framework for Online Management of a Metacomputing System", *The International Journal for Decision Support Systems*, Elsevier Science, vol. 28, no. 1-2, pp. 155-164, April 2000.
- [12] P. Arbenz, W. Gander, and M. Oettli, "The Remote Computation System", *Parallel Computing*, vol. 23, pp. 1421-1428, 1997.

- [13] D. Arnold, S. Blackford, J. Dongarra, V. Eijkhout and T. Xu, "Seamless Access to Adaptive Solver Algorithms", *Proceedings of the 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, Switzerland, August 2000.
- [14] K. Arnold, B. Osullivan, R. W. Scheifler, J. Waldo and A. Wollrath, "The Jini Specification", *Addison-Wesley*, ISBN: 0201616343, 1999.
- [15] J. Basney and M. Livny, "Managing Network Resources in Condor", *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*. Pittsburgh, Pennsylvania, August 2000.
- [16] A. Bayucan, R. L. Henderson, T. Proett, D. Tweten, and B. Kelly. Portable Batch System External Reference Specification. NAS Scientific Computing Branch, NASA Ames Research Center, California, June 1996.
- [17] F. Berman and R. Wolski, "Scheduling from the Perspective of the Application", In *Proceedings of Symposium on High Performance Distributed Computing*, pp. 100-111, 1996.
- [18] F. Berman and R. Wolski, "The AppLeS Project: A Status Report", *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, May 1997.
- [19] F. Berman, R. Wolski, S. Figueira, J. Schopf and G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks", *Proceedings of Supercomputing*, 1996.
- [20] D. C. Blight and T. Hamada, "Policy-Based Networking Architecture for QoS Interworking in IP Management", *Proceedings of Integrated Network Management*, pp. 811-826, Boston, Massachusetts, May 1999.

- [21] R. Buyya, "Economic-based Distributed Resource Management and Scheduling for Grid Computing", *Ph.D. Thesis*, School of Computer Science and Software Engineering, Monash University, Melbourne, Australia, April 2002.
- [22] R. Buyya, "High Performance Cluster Computing", *Prentice Hall*, 1999.
- [23] T. Casavant and J. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", *IEEE Transactions on Software Engineering*, vol. SE-14, no. 2, pp. 141-154, 1988.
- [24] H. Casanova, J. Dongarra and D. Doolin, "Java Access to Numerical Libraries", *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1279-1291, 1997.
- [25] H. Casanova and J. Dongarra, "NetSolve: A Network Server for Solving Computational Science Problems", *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 3, pp. 212-223, 1997.
- [26] H. Casanova and J. Dongarra, "NetSolve: A Network Enabled Server, Examples and Users", *Proceedings of Heterogeneous Computing Workshop*, pp. 19-28, Orlando, Florida, 1998.
- [27] H. Casanova, M. Kim, J. Plank and J. Dongarra, "Adaptive Scheduling for Task Farming with Grid Middleware", *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 13, no. 3, pp. 231-240, 1999.
- [28] C. Catlett and L. Smarr, "Metacomputing", *Communications of the ACM*, vol. 35, no. 6, pp. 44-52, 1992.

- [29] S. Chapin, J. Karpovich and A. Grimshaw, "The Legion Resource Management System", *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99), in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS '99)*, pp. 162-178, San Juan, Puerto Rico, April 1999.
- [30] B. Chapman, B. Sundaram, K. Thyagaraja, "EZ-Grid: Integrated Resource Brokerage for Computational Grid", *Technical Report*, Department of Computer Science, University of Houston, Houston, Texas.
- [31] Z. Chen, K. Maly, P. Mehrotra and M. Zubair, "ARCADE: A Web-Java Based Framework for Distributed Computing", *Proceedings of the WebNet 99*, October 1999.
- [32] T. Chou and J. Abraham, "Load Balancing in Distributed Systems", *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 401-412, July 1982.
- [33] Cray Research Incorporation, "Introducing NQE", *IN-2153 2.0*, Craysoft Publications, May 1995.
- [34] G. Coulouris, J. Dollimore and T. Kindberg, "Distributed Systems: Concepts and Design", Second Edition, *Addison-Wesley*, 1996.
- [35] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith and S. Tuecke, "A Resource Management Architecture for Metacomputing Systems", *Processings of the 4th Workshop on Job Scheduling Strategies for Parallel*, pp. 62-82, Springer-Verlag LNCS 1459, 1998.
- [36] Distributed Component Object Model (DCOM) web site. [Online]. Available: <http://www.microsoft.com/com/tech/dcom.asp>.



- [37] H. El-Rewini, H. Ali, and T. Lewis, "Task Scheduling in Multiprocessing Systems", *IEEE Computer*, vol. 28, no. 12, pp. 27-37, December 1995.
- [38] Extensible Markup Language (XML) specification web site. [Online]. Available: <http://www.xml.org>.
- [39] G. Fagg, K. Moore and J. Dongarra, "Scalable Networked Information Processing Environment (SNIPE)", *International Journal on Future Generation Computer Systems*, Elsevier Publ., vol. 15, No 5-6, pp. 595-605, 1999.
- [40] I. Foster and C. Kesselman, "The Grid: Blueprint for a Future Computing Infrastructure", *Morgan Kaufmann Publishers*, USA, 1999.
- [41] I. Foster and C. Kesselman, "The Globus Project: A Status Report", *Proceedings of the IPPS/SPDP '98 Heterogeneous Computing Workshop*, pp. 4-18, 1998.
- [42] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", *International Journal of Supercomputer Applications*, vol. 11, no. 2, pp. 115-128, 1997.
- [43] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt and A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation", *International Workshop on Quality of Service*, 1999.
- [44] I. Foster, A. Roy and V. Sander, "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation", *Proceedings of the 8th International Workshop on Quality of Service*, pp. 181-188, June 2000.

- [45] N. Furmento, S. Newhouse, and J. Darlington, "Building Computational Communities from Federated Resources", *Proceedings of the 7th International European Conference on Parallel Processing*, Manchester, UK, Springer-Verlag, Lecture Notes in Computer Science, vol. 2150, pp. 855-863, 2001.
- [46] W. Gentzsch, "Special Issue on Metacomputing: From Workstation Clusters to Internet Computing", *Future Generation Computer Systems*, no. 15, North Holland, 1999.
- [47] Global Grid Forum web site. [Online]. Available: <http://www.gridforum.org>.
- [48] Global Grid Forum. Information Service Area Group. [Online]. Available: <http://www-unix.mcs.anl.gov/gridforum/gis/>.
- [49] Gnutella web site. [Online]. Available: <http://www.gnutella.com>.
- [50] J. Gosling, B. Joy, and G. Steele, "The Java Language Specification", The Java Series. Addison Wesley Longman, 1996. ISBN 0-201-63451-1.
- [51] T. Green. "DQS User Interface Preliminary Design Document". *Technical Reference*. Supercomputer Computations Research Institute, Florida State University. [Online]. Available: [http://www.scri.fsu.edu/~pasko/dqs\\_user\\_guide/dqs\\_user\\_guide.html](http://www.scri.fsu.edu/~pasko/dqs_user_guide/dqs_user_guide.html).
- [52] Grid Interoperability Project (GRIP). [Online]. Available: <http://www.grid-interoperability.org/>.
- [53] A. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat", *IEEE Computer*, vol. 26, no. 5, pp. 39-51, 1993.

- [54] A. Grimshaw, A. Ferrari, F. Knabe and M. Humphrey, "Legion: An Operating System for Wide-Area Computing", *IEEE Computer*, vol. 32, no. 5, pp. 29-37, 1999.
- [55] A. Grimsaw and W. Wulf, "Legion: A View From 50,000 Feet", *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, August 1996.
- [56] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds, "A Synopsis of the Legion Project", *UVa CS Technical Report CS-94-20*, Department of Computer Science, University of Virginia, June 1994.
- [57] B. Hamidzadeh, D. J. Lilja, and Y. Atif, "Dynamic scheduling techniques for heterogeneous computing systems", *Concurrency: Practice and Experience*, vol. 7, no. 7, pp. 633-652, October 1995.
- [58] T. Haupt, E. Akarsu, G. Fox, C. Youn, "The Gateway system: uniform web access to remote resources", *Concurrency: Practice and Experience*, 2000, vol. 12, number 8, pp. 629-642, 2000.
- [59] K. Hawick, H. James, A. Silis, D. Grove, K. Kerry, J. Mathew, P. Coddington, C. Patten, J. Hercus, and F. Vaughan, "DISCWorld: An Environment for Service-Based Metacomputing", *Future Generation Computing Systems (FGCS)*, vol. 15, pp. 623-635, 1999.
- [60] S. Herbert. "Generic NQS - Free Batch Processing For UNIX". *Academic Computing Services*. The University Of Sheffield. UK. [Online]. Available: <http://www.shef.ac.uk/uni/projects/nqs/Product/Generic-NQS/v3.4x/>.

- [61] Y. Hoffner, "The Management of Monitoring in Object-Based Distributed Systems", *Proceedings of the Third International Symposium on Integrated Network Management*, San Francisco, April 1993.
- [62] T. Howes and M. Smith, "LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol", *Macmillan Technical Publishing*, ISBN 1-57870-000-0, 1997.
- [63] Internet Backplane Protocol homepage. [Online]. Available: <http://www.cs.utk.edu/~plank/IBP>. Last visit April 2002.
- [64] H. James and K. Hawick, "Resource Descriptions for Job Scheduling in DISCWorld", *Proceedings of Integrated Data Environments (IDEA5) Workshop*, Australia, 1998.
- [65] H. James, K. Hawick, and P. Coddington, "Scheduling Independent Tasks on Metacomputers", *Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems*, pp. 156-162, August 1999.
- [66] H. James, "Scheduling in Metacomputing Systems", *Ph.D. Thesis*, Department of Computer Science, University of Adelaide, Adelaide, Australia, July 1999
- [67] J. Joyce, K. Slind, and B. Unger, "Monitoring Distributed Systems", *ACM Transactions on Computer Systems*, vol. 5, no. 2, pp. 121-150, May 1987.
- [68] International Business Machines Corporation. "Workload Management with LoadLeveler". [Online]. Available: <http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246038.html?Open>.

- [69] Java 2 Platform, Enterprise Edition (J2EE). [Online]. Available: <http://java.sun.com/j2ee/>.
- [70] Java API for XML processing (JAXP). [Online]. Available: <http://java.sun.com/xml/jaxp>.
- [71] Java Remote Method Invocation (RMI). [Online]. Available: <http://java.sun.com/products/jdk/rmi/>.
- [72] Jiro Technology Web Page. [Online]. Available: <http://www.jiro.org>.
- [73] W. Keith, "Core Jini", *Prentice Hall*, ISBN 013014469X, 1999.
- [74] K. Keahey and D. Gannon, "PARDIS: CORBA-based Architecture for Application-Level PARallel DIStributed Computation", *Proceedings of Supercomputing '97*, November 1997.
- [75] K. Keahey and D. Gannon, "PARDIS: A Parallel Approach to CORBA", *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, August 1997.
- [76] B. Kingsbury. "The Network Queueing System". [Online]. Available: <http://power.curtin.edu.au/mirrors/nqs/Manuals/Papers/MNQS/MNQS0001/MNQS0001.txt>.
- [77] G. Laszewski, I. Foster, J. Gawor, P. Lane, "A Java Commodity Grid Kit", *Concurrency and Computation: Practice and Experience*, vol. 13, Issues 8-9, pp. 643-662, 2001.

- [78] G. Laszewski and S. Fitzgerald. "Representing Compute Resources for the Grid". *Global Grid Forum, Information Service Area Group*. Technical Report. GWD-GIS-005. [Online]. Available: <http://www-unix.mcs.anl.gov/gridforum/gis/old/papers/resources.pdf>.
- [79] A. Lewis, A. and T. Peachy, "Nimrod/O: A Tool for Automatic Design Optimization", *Proceedings of the 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000)*, Hong Kong, December 2000.
- [80] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations", *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104-111, June 1988.
- [81] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System", *University of Wisconsin-Madison, Computer Sciences Technical Report*, no. 1346, April 1997.
- [82] G. Liu, "Two Approaches to Critical Path Scheduling for a Hetrogeneous Environment", *M.S.Thesis*, Department of Computer Science, Old Dominion University, Norfolk, VA, USA, October 1998.
- [83] Live Networks Inc.. *The "liveGate" Multicast Tunneling Server*. [Online]. Available: <http://www.lvn.com/liveGate/>.
- [84] V. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems", *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1384-1397, November 1988.
- [85] K. Moore, S. Browne, J. Cox, and J. Gettler, "The Resource Cataloging and Distribution System", *Technical Report UT-CS-97-346, Computer Science Department, University of Tennessee*, December 1996.

- [86] G. Ma and P. Lu, "PBSWeb: A Web-based Interface to the Portable Batch System", *Proceedings of the 12th International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pp. 24-30, Las Vegas, Nevada, November 2000.
- [87] Multidisciplinary Optimization Branch (MDOB) at NASA Langley Research Center. [Online]. Available: <http://fmad-www.larc.nasa.gov/mdob/MDOB/index.html>.
- [88] MySQL web site. [Online]. Available: <http://www.mysql.com>.
- [89] H. Nakada, M. Sato and S. Sekiguchi, "Design and Implementations of Ninf: Towards a Global Computing Infrastructure", *Future Generation Computing Systems, Metacomputing Issue*, vol. 15, Issues 5-6, pp. 649-658, 1999.
- [90] Napster web site. [Online]. Available: <http://www.napster.com>.
- [91] N. Nisan, S. London, O. Regev, N. Camiel, "Globally Distributed Computation over the Internet - The POPCORN project", *Proceedings of the 18th International Conference on Distributed Computing Systems*, pp. 592-601, Amsterdam, The Netherlands, May 1998.
- [92] R. Nudd, D.J. Kerbyson, E. Papaefstathiou, J.S. Harper, S.C. Perry, and D.V. Wilcox, "PACE: A Toolset for the Performance Prediction of Parallel & Distributed Systems", *International Journal of High Performance Computing Applications*, Special Issue on Performance Engineered Systems, vol. 14, no. 4, pp. 228-251, Fall 2000.
- [93] OMG's CORBA web site. [Online]. Available: <http://www.corba.org>.

- [94] A. Oram, "Peer-to-Peer: Harnessing the Power of Disruptive Technologies", *O'Reilly Press*, USA, 2001.
- [95] P. Parnes, K. Synnes, and D. Schefstrom, "Lightweight Application Level Multicast Tunneling using mTunnel", *Journal of Computer Communication*, vol. 21, pp. 1295-1301, 1998.
- [96] P. Parnes, K. Synnes, and D. Schefstrom, "mTunnel: A Multicast Tunneling System With A User Based Quality-Of-Service Model", *Proceedings of the European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, 1997.
- [97] Peer-to-Peer Working Group (P2PWG) web site. [Online]. Available: <http://www.p2pwg.org>.
- [98] Platform Computing Corporation. "Load Sharing Facility". [Online]. Available: <http://www.platform.com/products/wm/lst/index.asp>.
- [99] Project JXTA web site. [Online]. Available: <http://www.jxta.org/>.
- [100] R. Rajan, D. Verma, S. Kamat, E. Felstaine, and S. Herzog, "A policy framework for integrated and differentiated services in the Internet", *IEEE Network*, vol. 13, no. 5, pp. 36-41, September/October 1999.
- [101] R. Raman, M. Livny and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing", *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pp. 28-31, Chicago, Illinois, July 1998.



- [102] M. Romberg, "The UNICORE Architecture: Seamless Access to Distributed Resources", *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing HPDC-8*, IEEE Computer Society, Los Alamitos, CA, pp. 287-293, August 1999.
- [103] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima and H. Takagi, "Ninf: A Network based Information Library for a Global World-Wide Computing Infrastructure", *HPCN'97 (LNCS-1225)*, pp. 491-502, 1997.
- [104] K. Savetz, N. Randall and Y. Lepage, "MBONE: Multicasting Tomorrow's Internet", *New Riders Publishing*, ISBN 1-56205-397-3, 1996.
- [105] U. Schwiegelshohn and R. Yahyapour. "Fairness in Parallel Job Scheduling", *Journal of Scheduling*, vol. 3, no. 5, pp. 297-320, John Wiley, 2000.
- [106] D. Schmidt, M. Stal, H. Rohnert and, F. Buschmann, "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects", *Wiley & Sons*, ISBN 0-471-60695-2, 2000.
- [107] S. Sekiguchi, M. Sato, H. Nakada and U. Nagashima, "Ninf: Network based information library for globally high performance computing", *Proceedings of Parallel Object-Oriented Methods and Applications (POOMA)*, pp. 39-48, February 1996.
- [108] B. Shirazi, A. Husson, and K. Kavi, "Scheduling and Load Balancing in Parallel and Distributed Systems, chapter Introduction to Scheduling and Load Balancing. IEEE Computer Society Press, Los Alamitos, CA, 1995. ISBN 0-8186-6587-4.

- [109] J. Skovira, W. Chan, and H. Zhou, "The EASY - LoadLeveler API Project", *proceedings of the IPPS workshop on Job Scheduling Strategies in Parallel Processing*, pp. 41-47, March, 1996.
- [110] S. Smallen, W. Cirne, J. Frey, F. Berman, R. Wolski, M. Su, C. Kesselman, S. Young and M. Ellisman, "Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience", *Proceedings of the 9th Heterogenous Computing Workshop (HCW 2000@IPDPS)*, pp. 241-252, Cancun, Mexico, May 2000.
- [111] D. Snelling, S. Berghe, G. Laszewski, P. Wieder, J. MacLaren, J. Brooke, D. Nicole and H. Hoppe. "A Unicore Globus Interoperability Layer". *Draft of a Use Case Study for GPA WG*. [Online]. Available: [http://www-unix.gridforum.org/mail\\_archive/gpa-wg/doc00000.doc](http://www-unix.gridforum.org/mail_archive/gpa-wg/doc00000.doc).
- [112] N. Spring and R. Wolski, "Application Level Scheduling of Gene Sequence Comparison on Metacomputers", *Proceedings of the 12th ACM International Conference on Supercomputing*, pp. 141-184, Melbourne, Australia, July 1998.
- [113] P. Stelling, I. Foster, C. Kesselman, C. Lee and G. von Laszewski, "A Fault Detection Service for Wide Area Distributed Computations", *Proceedings of the 7th IEEE Symposium on High Performance Distributed Computing*, pp. 268-278, 1998.
- [114] V. Sunderam, J. Dongarra, A. Geist, and R. Manchek, "The PVM concurrent computing system: Evolution, experiences, and trends", *Parallel Computing*, vol. 20, no. 4, pp. 531-547, April 1994.
- [115] Sun Microsystems. Sun Grid Engine Software. [Online]. Available: <http://www.sun.com/software/gridware/>.

- [116] Sun Microsystems. "RMI over IIOP". [Online]. Available: <http://www.java.sun.com/products/rmi-iiop>.
- [117] S. Vadhiyar and J. Dongarra, "A Metascheduler For The Grid", *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pp. 343-351, Edinburgh, Scotland, July 2002.
- [118] D. Verma, M. Beigi, and R. Jennings, "Policy Based SLA Management in Enterprise Networks", *Proceedings of Policy Workshop 2001*, pp. 29-31, Springer-Verlag, January 2001.
- [119] R. Wolski, "Dynamically Forecasting Network Performance Using the Network Weather Service", *Journal of Cluster Computing*, vol. 1, no. 1, pp. 119-132, 1998.
- [120] R. Wolski, N. T. Spring and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing", *The Journal of Future Generation Computing Systems*, vol.15, no. 5-6, pp. 757-768, October 1999.
- [121] R. Wolski, N. Spring and J. Hayes, "Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid", *Journal of Cluster Computing*, vol. 3, no. 4, pp. 293-301, 2000.
- [122] R. Wolski, N. Spring and C. Peterson, "Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service", *Proceedings of the ACM/IEEE Supercomputing Conference*, San Jose, CA, November 1997.

- [123] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks in an Unbounded no. of Processors", *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951-967, September 1994.
  
- [124] H. Zhou, "Scheduling DAGs on a bounded number of processors - A new approach", *Proceedings of International Conference on Parallel and Distributed Processing, Techniques and Applications*, vol. 2, pp. 823-834, August 1996.

## APPENDIX A

### Experiment Results

#### A.1. Overhead of broadcasting/delivery for the *Collaboration* approach

TABLE 8  
OVERHEAD OF BROADCASTING/DELIVERY FOR THE COLLABORATION  
APPROACH

Number of TSs	Observation 1	Observation 2	Observation 3	Observation 4	Observation 5	Observation 6	Average
10	85	72	93	92	88	87	86.16667
20	147	126	130	119	142	148	135.3333
30	184	210	235	245	244	190	218
40	224	208	231	200	243	225	221.8333
50	235	216	248	220	277	237	238.8333
60	274	252	271	268	291	261	269.5
70	365	289	290	355	343	285	321.1667
80	549	433	389	385	426	377	426.5
90	532	512	415	412	444	502	469.5
100	485	639	481	495	491	492	513.8333

#### A.2. Overhead of broadcasting/delivery for the *Hierarchal Tunneling* approach.

TABLE 9  
OVERHEAD OF BROADCASTING FOR THE HIERARCHAL TUNNELING  
APPROACH

Number of TSs	Observation 1	Observation 2	Observation 3	Observation 4	Observation 5	Observation 6	Average
10	88	83	81	68	84	90	82.33333
20	71	66	92	100	90	88	84.5
30	86	83	75	100	94	103	90.16667
40	86	80	76	105	75	72	82.33333
50	67	89	84	90	93	81	84
60	72	110	76	78	91	78	84.16667
70	79	63	81	78	109	84	82.33333
80	95	121	84	86	92	84	93.66667
90	77	96	86	115	92	93	93.16667
100	87	80	109	97	72	94	89.83333

**TABLE 10**  
**OVERHEAD OF DELIVERY FOR THE HIERARCHAL TUNNELING APPROACH**

<b>Number of TSs</b>	<b>Observation 1</b>	<b>Observation 2</b>	<b>Observation 3</b>	<b>Observation 4</b>	<b>Observation 5</b>	<b>Observation 6</b>	<b>Average</b>
10	88	84	81	68	84	91	82.66667
20	86	75	102	122	102	92	96.5
30	108	102	109	141	121	128	118.1667
40	124	112	103	139	109	101	114.6667
50	127	134	114	127	134	115	125.1667
60	112	150	114	122	148	111	126.1667
70	134	126	126	127	177	136	137.6667
80	157	177	172	145	155	154	160
90	140	166	165	191	166	146	162.3333
100	170	165	181	178	149	175	169.6667

### **A.3. Overhead of XML Parsing**

**TABLE 11**  
**PARSING TIME FOR DIFFERENT XML DOCUMENTS**

<b>XML Document</b>	<b>Observation 1</b>	<b>Observation 2</b>	<b>Observation 3</b>	<b>Observation 4</b>	<b>Observation 5</b>	<b>Average</b>
Single	902	910	906	893	908	903.8
Parametric	912	892	904	897	889	898.8
CoAllocation	910	898	890	903	902	900.6
DAG	930	921	915	923	933	924.4
Resource	991	979	1004	998	1000	994.4

**A.4. Performance of Resource Matching.**

TABLE 12  
PERFORMANCE OF RESOURCE MATCHING UNDER DIFFERENT DATA  
RETRIEVAL APPROACHES

Methodology	Observation 1	Observation 2	Observation 3	Observation 4	Observation 5	Average
Caching	12	11	14	12	10	11.8
Local Repository	34	33	39	37	31	34.8
Remote Repository	56	54	60	59	49	55.6

**A.5. Performance of SLA Monitoring.**

TABLE 13  
PERFORMANCE OF SLA MONITORING UNDER DIFFERENT DATA RETRIEVAL  
APPROACHES

Methodology	Observation 1	Observation 2	Observation 3	Observation 4	Observation 5	Average
Caching	3	2	2	2	3	2.4
Local Repository	26	22	21	24	28	24.2
Remote Repository	40	41	43	38	45	41.4

**A.6. Memory usage**

**TABLE 14**  
**MEMORY USAGE FOR SMALL GRID WHEN NO SLAS ARE APPLIED**

<b>Number of Resources</b>	<b>Memory Usage</b>
10	124
20	132
30	136
40	144
50	148
60	152
70	156
80	160
90	164

**TABLE 15**  
**MEMORY USAGE FOR MEDIUM GRID WHEN NO SLAS ARE APPLIED**

<b>Number of Resources</b>	<b>Memory Usage</b>
1000	804
2000	1388
3000	1388
4000	1892
5000	2516
6000	2968
7000	3476
8000	3480
9000	4036



**TABLE 16**  
**MEMORY USAGE FOR LARGE GRID WHEN NO SLAS ARE APPLIED**

<b>Number of Resources</b>	<b>Memory Usage</b>
10000	4368
20000	8692
30000	12520
40000	16300
50000	21512
60000	25804
70000	29904
80000	32316
90000	36408

**TABLE 17**  
**MEMORY USAGE FOR SMALL GRID WITH AN AVERAGE OF 5 SLAS PER  
 RESOURCE**

<b>Number of Resources</b>	<b>Memory Usage</b>
10	144
20	156
30	168
40	180
50	192
60	204
70	212
80	228
90	242

TABLE 18  
MEMORY USAGE FOR MEDIUM GRID WITH AN AVERAGE OF 5 SLAS PER  
RESOURCE

Number of Resources	Memory Usage
1000	1024
2000	2032
3000	2572
4000	3288
5000	3820
6000	4420
7000	5396
8000	6076
9000	6756

TABLE 19  
MEMORY USAGE FOR LARGE GRID WITH AN AVERAGE OF 5 SLAS PER  
RESOURCE

Number of Resources	Memory Usage
10000	7424
20000	14660
30000	21744
40000	29100
50000	36024
60000	42852
70000	50660
80000	57872
90000	65084

**A.7. Overall Overhead of Brokering.**

**TABLE 20**  
**COMPLETION TIME OF A 100000 ms JOB UNDER DIFFERENT EXECUTION**  
**ENVIRONMENTS**

<b>Grid System</b>	<b>Observation 1</b>	<b>Observation 2</b>	<b>Observation 3</b>	<b>Observation 4</b>	<b>Observation 5</b>	<b>Average</b>
Globus	105844	106004	105466	106020	105710	105808.8
PROBE/Globus	106721	107128	106321	106939	106649	106751.6
SGE	103002	102943	102781	102558	102594	102775.6
PROBE/SGE	104424	103773	103593	103614	103461	103773

**TABLE 21**  
**BROKERING OVERHEAD FOR DIFFERENT JOB SIZES UNDER THE**  
**PROBE/GLOBUS EXECUTION ENVIRONMENT**

<b>Job Size</b>	<b>Observation 1</b>	<b>Observation 2</b>	<b>Observation 3</b>	<b>Observation 4</b>	<b>Observation 5</b>	<b>Execution Time</b>	<b>Brokering Percentage</b>
10	16538	16746	16901	16744	16667	16719.2	67.192
50	56642	56892	56721	56893	57014	56832.4	13.6648
100	106721	107128	106321	106939	106649	106751.6	6.7516
500	506412	506764	506827	506721	506926	506730	1.346
1000	1006728	1006886	1007032	1006753	1006544	1006788.6	0.67886
5000	5006876	5006784	5006511	5006778	5006835	5006756.8	0.135136
10000	10006631	10006743	10006741	10007004	10006906	10006805	0.06805

## APPENDIX B

### List of Acronyms and Terms

API (Application Programming Interface)  
AppLeS (Application Level Scheduling)  
CFD (Computational Fluid Dynamics)  
ClassAds (Classified Advertisement Language)  
CM (Communication Manager)  
CORBA (Common Object Request Broker Architecture)  
DAG (Directed Acyclic Graph)  
DCOM (Distributed Component Object Model)  
DISCWorld (Distributed Information Systems Control World)  
DM (Data Manager)  
DQS (Distributed Queuing System)  
DTD (Document Type Definition)  
EJB (Enterprise JavaBeans)  
EM (Execution Manager)  
FJL (Flexible Job Language)  
FMA (Federated Management Architecture)  
GGF (Global Grid Forum)  
GIS (Grid Information Service)  
GNQS (Generic Network Queuing System)  
GRAM (Globus Resource Allocation Manager)  
GRIP (Grid Interoperability Project)  
GTLS (Global Tunneling Lookup Service)  
GUI (Graphical User Interface)  
IBP (Internet Backplane Protocol)  
IIOP (Internet Inter-ORB Protocol)  
J2EE (Java 2 Enterprise Edition)

JAAS (Java Authorization and Authentication Service)  
JAXP (Java API for XML Processing)  
JDK (Java Development Kit)  
JNDI (Java Naming and Directory Interface)  
JRMP (Java Remote Method Protocol)  
JVM (Java Virtual Machine)  
LAN (Local Area Network)  
LDAP (Lightweight Directory Access Protocol)  
LS (Lookup Service)  
LSF (Load Share Facility)  
MDO (Multidisciplinary Design Optimization)  
MDS (Metacomputing Directory Service)  
MLS (Module Lookup Service)  
NQS (Network Queuing System)  
NWS (Network Weather Service)  
OMG (Object Management Group)  
ORB (Object Request Broker)  
ORPC (Object Remote Procedure Call)  
P2P (Peer-to-peer computing)  
P2PWG (Peer-to-Peer Working Group)  
PBS (Portable Batch System)  
PCG (PROBE Computational Grid)  
PSL (Policy Scripting Language)  
PROBE (Policy-based Resource Brokering Environment)  
PVM (Parallel Virtual Machine)  
QoS (Quality Of Service)  
RCDS (Resource Cataloging and Distribution System)  
RLS (Resource Lookup Service)  
RM (Resource Manager)  
RMI (Remote Method Invocation)

**RPC (Remote Procedure Call)**  
**RSL (Resource Specification Language)**  
**SLA (Service Level Agreement)**  
**SDK (Software Development Kit)**  
**SGE (Sun Grid Engine)**  
**SM (Security Manager)**  
**TS (Tunneling Service)**  
**UNICOR (UNiform Interface to COmputing RESources)**  
**VPN (Virtual Private Network)**  
**W3C (World Wide Web Consortium)**  
**WAN (Wide Area Network)**  
**XML (eXtensible Markup Language)**

## APPENDIX C

### Glossary

**Action**

Actions are the result of some met policy conditions.

**Cluster Computing**

Many computational resources connected together by a local area network and can be viewed as a unified resource.

**Co-Allocation**

This is the kind of job that requires that a set of resources is available for use simultaneously.

**CORBA**

The Common Object Request Broker Architecture (CORBA) is a distributed computing standard from the Object Management Group (OMG) for the development and deployment of applications in distributed, heterogeneous environments.

**DAG**

A Direct Acyclic Graph (DAG) represents an application program that consists of a collection of heterogeneous modules (application codes from different disciplines) with acyclic dependencies among the modules.

**DCOM**

The Distributed Component Object Modeling (DCOM) is a distributed object model developed by Microsoft for the development of distributed applications.

**Federated Management Architecture (FMA)**

A specification from Sun Microsystems for heterogeneous storage resources and storage network management.

**Grid**

An environment that combines geographically distributed heterogeneous resources in independent administrative domains into a virtual metacomputer in support of large-size problems.

**Heterogeneous**

An architecture in which the elements are of different types.

**Homogeneous**

An architecture in which each element is of the same type.

**Jini**

A connection technology introduced by Sun Microsystems that can be used to build a flexible network of resources and services to be shared by a group of clients. It is based on the idea of federating groups of clients and the resources required by those clients.

**Jiro**

A pure Java technology-based implementation of the Federated Management Architecture (FMA) specification that provides developers with the infrastructure required to build distributed resource management solutions.

**Job**

We use this term usually to refer to the application, or one of its sub-modules, being created to satisfy the user's request.



**Load balancing**

The degree by which the work is distributed equally among the available replicated components in a typical distributed environment.

**Metacomputing.**

An approach in which more than the local resources are used to solve a large-scale computational problem.

**Middleware.**

A layer between the application and the operating system that provides seamless services to the high-level application.

**Parametric Application**

An application where the same program is repeatedly executed with different initial conditions as a means of exploring the behavior of a complicated system across a parameter space.

**Policy**

A set of conditions and actions that need to be taken when those conditions are met.

**Policy-based Resource Brokering Environment (PROBE)**

A general-purpose, stand-alone, heterogeneous, distributed Policy-based Resource Brokering Environment that can be easily used by various grid environments.

**Profiling**

The measuring of the performance and resource requirements of an application.

**Resource**

In a typical grid environment, a resource denotes any entity that is meant to be shared. It could be computational, storage, software, network, etc.

**Resource Brokering Environment**

A middleware software application that mediates the discovery, access and usage of distributed resources, often heterogeneous, in a grid environment.

**RMI**

Java Remote Method Invocation (RMI) is a distributed computing technology by Sun Microsystems that provides a simple and direct model for distributed computation with Java objects.

**Service Level Agreement (SLA)**

A formal negotiated agreement between two parties, the service provider and the service consumer. This agreement provides a common understanding about quality of the service and responsibilities of both parties.

**Scalability**

It is the degree by which a system or component continues to grow and maintain service without fundamental change in the application's architecture or major degradation of the performance.

**Scheduling**

Order and placement of tasks into a set of resources.

**Task**

Same as Job.

**Workflow Manager**

A component within a typical grid environment that automates the business process of the user.

**XML**

The extensible Markup Language (XML) is a specification for creating structured documents and data.

## APPENDIX D

### Extended Bibliography

This appendix contains some references for efforts not covered in chapter II.

- [1] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, "Charlotte: Metacomputing on the Web", *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, pp.181-188, Dijon, France, September 1996.
- [2] A. Baratloo, P. Dasgupta, and Z. Kedem, "Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms" *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, pp. 122-129, 1995.
- [3] D. Becker, T. Sterling, D. Savarese, J. Dorband, U. Ranawake and C. Packer, "Beowulf: A Parallel Workstation for Scientific Computation", *Proceedings of the 1995 International Conference on Parallel Processing*, pp. 11-14, Oconomowoc, Wisconsin, August 1995.
- [4] P. Chandra, Y. Chu, A. Fisher, J. Gao, C. Kosak, T. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, "Darwin: Customizable Resource Management for Value-Added Network Services", *IEEE Network*, Number 1, vol. 15, January 2001.
- [5] K. Chandy, A. Chelian, B. Dimitrov, Z. Dobes, J. Garnett, J. Kiniry, H. Le, J. Mandelson, M. Richardson, A. Rifkin, E. Schooler, P. Sivilotti, W. Tanaka, and L. Weisman, "A New Approach To Collaborative Distributed Computing", *newsletter of the Center for Research on Parallel Computation*, 1996.

- [6] K. Chandy, A. Rifkin, P. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman, "A World-Wide Distributed System Using Java and the Internet", *Proceedings of the High Performance Distributed Computing (HPDC '96)*, pp. 11-18, Syracuse, New York, March 1996.
- [7] H. El-Rewini and T. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Machines", *Journal of Parallel and Distributed Computing*, vol. 9, Number 2, pp. 138-153, 1990.
- [8] G. Fox, W. Furmanski, M. Chen, C. Rebbi, J. Cowie, "WebWork: Integrated Programming Environment Tools for National and Grand Challenges", *NPAC Technical Report SCCS-715*, Syracuse University, June 1995.
- [9] R. Freund, T. Kidd, D. Hensgen, L. Moore, "SmartNet: A Scheduling Framework for Heterogeneous Computing," *Proceedings of 2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 514-521, Beijing, China, June 1996.
- [10] J. Gehring and A. Streit, "Robust Resource Management for Metacomputers", *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC 2000)*, pp. 105-112, Pittsburgh, Pennsylvania, 2000.
- [11] J. Gehring and A. Reinefeld, "MARS - A Framework for Minimizing Job Execution Time in a Metacomputing Environment", *Future Generation Computer Systems (FGCS)*, vol. 12, Number 1, pp. 87-99, 1996.
- [12] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors", *Journal of Parallel and Distributed Computing*, vol. 16, pp. 276-291, 1992.

- [13] D. Hensgen, T. Kidd, D. John, M. Schnaidt, H. Siegel, T. Braun, J. Kim, S. Ali, C. Irvine, T. Levin, V. Prasanna, P. Bhat, R. Freund, and M. Gherrity, "An Overview of the Management System for Heterogeneous Networks (MSHN)", *8th Workshop on Heterogeneous Computing Systems (HCW '99)*, pp. 184-198, San Juan, Puerto Rico, April 1999.
- [14] N. Kapadia and J. Fortes, "PUNCH: An Architecture for Web-Enabled Wide-Area Network-Computing", *Cluster Computing: The Journal of Networks, Software Tools and Applications*, vol. 2, Number 2, pp. 153-164, Baltzer Science Publishers, The Netherlands, September 1999.
- [15] D. Lifka, M. Henderson and K. Rayl, "Users Guide to the Argonne Sp Scheduling System", *Mathematics and Computer Science Division Technical Memorandum Number ANL/MCS-TM-201*, Argonne National Laboratory, May 1995
- [16] MAUI Scheduler web site. [Online]. Aviliable: <http://supercluster.org/maui/>.
- [17] M. Migliardi and V. Sunderam, "The Harness Metacomputing Framework", *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.
- [18] MPI web site. [Online]. Aviliable: <http://www-unix.mcs.anl.gov/mpi/>.
- [19] M. Neary, A. Phipps, S. Richman and P. Cappello, "Javelin 2.0: Java-Based Parallel Computing on the Internet", *Proceedings of European Parallel Computing Conference (Euro-Par 2000)*, pp. 1231-1238, Germany, 2000.

- [20] B. Neuman and S. Roa, "The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems", *Concurrency: Practice and Experience*, vol. 6, Number 4, pp. 339-355, June 1994.
- [21] J. Pruyne and M. Livny, "Parallel Processing on Dynamic Resources with CARMI", *Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '95)*, pp. 259-278, April 1995.
- [22] PVM web site. [Online]. Available: <http://www.epm.ornl.gov/pvm/>.
- [23] F. Ramme and K. Kremer, "Scheduling a Metacomputer by an Implicit Voting System", *Proceedings of the 3rd IEEE International Symposium on High-Performance Distributed Computing*, San Francisco, pp 106-113, 1994.
- [24] J. Weissman, "Scheduling Multi-Component Applications in Heterogeneous Wide-area Networks", *Heterogeneous Computing Workshop, International Parallel and Distributed Processing Symposium IPDPS*, pp. 209-215, May 2000.
- [25] J. Weissman, "Prophet: Automated Scheduling of SPMD Programs in Workstation Networks", *Concurrency: Practice and Experience*, vol. 11, Number 6, pp. 301-321, November 1999.

## **VITA**

Ahmed Hamdan AL-Theneyan was born in Riyadh, Saudi Arabia, on March 9, 1974. He received his Bachelor of Science in Computer and Information Sciences from King Saud University, Riyadh, Saudi Arabia in 1995. He worked then as Lecturer for the Department of Computer Science at Institute of Public Administration (IPA) from 1995 to 1996. On 1996, Ahmed was awarded a scholarship from the Royal Embassy of Saudi Arabia and IPA for a Master's degree program in Computer Science. In the Fall of 1996, Ahmed joined Old Dominion University and received his Master of Science degree in Computer Science in 1998. In August 1998 and after having finished his Master's degree, Ahmed was offered a teaching assistantship and was accepted in the Ph.D. program of the same department. Ahmed was then awarded a fellowship grant from NASA Langley Research Center in 1999. He worked as a Research Assistant in the Institute for Computer Applications in Science and Engineering (ICASE) at NASA Langley Research Center from 1999 to 2001. During his years at NASA and ODU, Ahmed has been involved in a wide range of network and distributed computing activities where he explored the applicability of Jini, CORBA and other distributed computing technologies for distributed middleware applications. Ahmed is currently working as a senior member of technical staff at Trendium Incorporation. His current research focuses on Distributed Computing, Resource Brokering, Computer Networks, Web Applications and Object-Oriented Design.

**Permanent Address:** Department of Computer Science  
Old Dominion University  
Norfolk, VA 23529-0162