Hierarchical Scheduling in Grid Systems

By
Khaldoon Al-Zoubi

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario
March 2006

© Copyright
2006, Khaldoon Al-Zoubi

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

Hierarchical Scheduling in Grid Systems

Submitted by

Khaldoon Al-Zoubi

In partial fulfillment of
the requirements for the degree of
Master of Computer Science

Douglas Howe, Director

---

late Sivarama Dandamudi, Thesis Supervisor

Jean-Pierre Corriveau, Thesis Co-Supervisor

Carleton University
2006

ABSTRACT

This research is mainly focused on the first two stages of the Grid scheduling, namely: *Resource discovery* and *resource selection* stages. We propose a self-discovery method for the *resource discovery* stage. In addition, we propose an adaptive child scheduling method for the *resource selection* stage. We also propose three rescheduling algorithms in the resource selection stage: (1) the Butterfly algorithm in order to reschedule jobs when better resources become available, (2) the Fallback algorithm in order to reschedule jobs that had their resources taken away from the Grid before the actual resource allocation and (3) the Load-Balance algorithm in order to balance load among resources. With the purpose of increasing system scalability and flexibility, the proposed hierarchal scheduling approach is combined with the Peer-to-Peer (P2P) systems approach in one hybrid system.

A Grid model (consisting of 2400 nodes) is built to test the proposed ideas through simulation over a number of different workloads and scenarios. We compare the performance of the proposed hierarchal systems against the P2P systems approach according to three metrics: (1) the total response time, (2) the average waiting time, and (3) the average execution time.

ACKNOWLEDGMENT

First of all, I thank ALLAH, the creator, for giving me the ability and the will to accomplish anything I want to do.

I am always in debt for my parents, my uncle, Abraham, my wife, Sahar, my son, Tariq, my sisters and my brothers for their infinite support in anything I want to do, particularly in this research.

I would like to strongly thank my thesis supervisor, the late Professor Sivarama Dandamudi, for his inputs and discussions during this research. I was truly lucky to have the late professor Dandamudi as my supervisor in this research.

I cannot thank Professor Jean-Pierre Corriveau enough for becoming my acting supervisor after the unexpected and sad death of late Professor Dandamudi. I am always grateful for his help in reviewing this thesis and assuring its completion as originally planned.

I would like to thank Professor Gabriel Wainer for granting me the permission to use the DEVS CD++ tools, which made it possible to build the simulation model. I also thank him and his students for their help with the tools.

CONTENT

ABSTRACT .... **III**

ACKNOWLEDGMENT .... **IV**

LIST OF TABLES .... **VII**

LIST OF FIGURES .... **VIII**

LIST OF ACRONYMS .... **X**

CHAPTER 1: INTRODUCTION **1**

1.1 INTRODUCTION 1
1.2 WHY GRID COMPUTING? 7
1.3 GRID ARCHITECTURE 13
1.3.1 Centralized Architecture *14*
1.3.2 Distributed Architecture *15*
1.3.3 Hierarchal Architecture *18*
1.4 THESIS CONTRIBUTIONS 19
1.5 THESIS ORIGANIZATION 20

CHAPTER 2: BACKGROUND **22**

2.1 INTRODUCTION 22
2.2 ADAPTIVE HIERARCHICAL SCHEDULING (AHS) POLICY 23
2.3 GRID SCHEDULING STAGES 24
2.3.1 First Stage: Resource Discovery *27*
2.3.1.1 Information gathering about static resources 30
2.3.1.2 Job requirements and matching 37
2.3.2 Second Stage: Resources Selection and Job Scheduling *40*
2.3.2.1 Dynamic information gathering 41
2.3.2.2 Rescheduling 42
2.3.2.3 Metascheduling 42
2.3.3 Third Stage: Job Execution *42*
2.3.3.1 Preemption and Checkpointing 42
2.3.3.2 Cycle harvesting 43

CHAPTER 3: HIERARCHICAL SCHEDULING IN GRID SYSTEMS **44**

3.1 INTRODUCTION 44
3.2 HIERARCHAL GRID SCHEDULING STAGES 47
3.2.1 First Stage: Resource Discovery *52*
3.2.1.1 First Method: Layering Resources 55
3.2.1.2 Second Method: Self Discovery 60
3.2.2 Second Stage: Job Scheduling and Resources Selection *71*
3.2.3 Dynamic algorithms *79*
3.2.3.1 Butterfly rescheduling algorithm 80
3.2.3.2 Fallback rescheduling algorithm 84
3.2.3.3 Load-Balance Algorithm 86
3.3 HYBRID MODEL: HIERARCHAL AND P2P GRID SCHEDULING 88

CHAPTER 4: GRID MODEL BASIC COMPONENTS **94**

4.1 INTRODUCTION 94
4.2 DEVS CD++ TOOLS 96
4.3 COMMUNICATION MODEL COMPONENTS 98

4.3.1 Communication Atomic models 101
4.3.1.1 Cable Model 101
4.3.1.2 Router Model 103
4.3.1.3 Timer Model 104
4.3.1.4 Node Model 104
4.3.2 Communication Structured Models 104
4.3.2.1 Top Model (Backbones Layer) 104
4.3.2.2 Backbone Model (Nets Layer) 105
4.3.2.3 Net Model (Networks Layer) 107
4.3.2.4 Network Model (Hosts Layer) 108
4.4 NODE MODEL DISCUSSIONS 109

CHAPTER 5: GRID SIMULATION MODEL, EXPERIMENTS, RESULTS AND DISCUSSIONS 112

5.1 INTRODUCTION 112
5.2 COMMUNICATION MODEL 115
5.3 NODE MODEL 121
5.4 SYSTEM MODEL 122
5.4.1 Peer-to-Peer (P2P) System 122
5.4.2 Hierarchal (one Grid tree) System 123
5.4.3 Hybrid (several Grid trees) System 124
5.5 GRID JOBS 124
5.6 RESOURCES 126
5.7 WORKLOADS 127
5.8 PERFORMANCE METRICS 128
5.9 SIMULATION EXPERIMENTS 131
5.9.1 First Experiment: Continuous Job Submissions 134
5.9.1.1 Results and Discussions 135
5.9.2 Second Experiment: Job Submissions with Different Stochastic Rates 144
5.9.2.1 Results and Discussions 145
5.9.3 Third Experiment: Job Submissions with The Same Stochastic Rate 151
5.9.3.1 Results and Discussions 153
5.9.4 Fourth Experiment: Algorithms Impact on System Performance 158
5.9.4.1 Results and Discussions 158
5.9.5 Fifth Experiment: Resource Change Impact on System Behaviour 162
5.9.5.1 Results and Discussions 162

CONCLUSIONS AND FUTURE WORK 167

REFERENCES 173

APPENDIX A – RESULT TABLES 183

A.1 FIRST EXPERIMENT RESULTS SET 183
A.2 SECOND EXPERIMENT RESULTS SET 187
A.3 THIRD EXPERIMENT RESULTS SET 190
A.4 FOURTH EXPERIMENT RESULTS SET 191
A.5 FIFTH EXPERIMENT RESULTS SET 192

LIST OF TABLES

Table 1: Samples of Transferred Data Time for the Shown Path in Figure 33 120
Table 2: Simulated Computational Servers 126
Table 3: Workloads Used in Simulation (Servers = 520, Workstations = 1880) 128
Table 4: Results Set in Experiment 1 186
Table 5: Results Set in Experiment 2 189
Table 6: Results Set in Experiment 3 190
Table 7: Results Set in Experiment 4 191
Table 8: Results Set in Experiment 5 192

LIST OF FIGURES

Figure 1: Accessing the Virtual Computing Power of the Grid 13
Figure 2: Job Scheduling in Centralized Grid System 15
Figure 3: Job Scheduling in Distributed Grid System 16
Figure 4: An Example of Joining a New Peer in the P2P systems 17
Figure 5: An Example of Hierarchical Scheduling 24
Figure 6: Grid Scheduling Stages and Capabilities. 26
Figure 7: Resources Sharing in Virtual Organizations (VOs). 29
Figure 8: VOs Indexes Structure in MDS Systems. 29
Figure 9: Hierarchical Grid Scheduling Stages and Capabilities – See also Figure 6. 46
Figure 10: Transferring User’s job to Allocated Resources. 49
Figure 11: Resources Discovery Stage Overview 52
Figure 12: Submitting Actual Jobs as Batches 54
Figure 13: Illustration of Advertisement Messages. 54
Figure 14: Logical Channels for GSs with Different Static Resources 57
Figure 15: Internal Illustration of Resources and Jobs 58
Figure 16: Logical Channels in the Self-discovery method. 67
Figure 17: Example of Two Channels for a Parent and two of its Children 70
Figure 18: Example of Fixing Broken Channel by Scheduler from Figure 17 70
Figure 19: Example of Broken and Fixed Channel 71
Figure 20: An Example of Mapping RFCs to Logical Channels 76
Figure 21: Space/Time Sharing in AHS 77
Figure 22: Illustration of the Butterfly Rescheduling Algorithm 83
Figure 23: Rescheduling Fallback Algorithm. 85
Figure 24: An Example of Load Balance among Channels 87
Figure 25: Hybrid Model – Hierarchical and P2P Grid Scheduling 89
Figure 26: IP Addresses Hierarchal in the Internet Model 100
Figure 27: Top Model Overview 105
Figure 28: Backbone Model Overview 106
Figure 29: Net Model Overview 108
Figure 30: Network Model Overview 109
Figure 31: Valid IP Addresses in the Communication Model 116
Figure 32: An Example of the Communication Model 117
Figure 33: An Example of a Communication Path. 121
Figure 34: Average Waiting Times for 28.9MB Size Jobs in Experiment 1 136
Figure 35: Average Waiting Times for 312.7MB Size Jobs in Experiment 1 137
Figure 36: Average Waiting Times for 1GB Size Jobs in Experiment 1 137
Figure 37: Average Waiting Times for 10GB Size Jobs in Experiment 1 138
Figure 38: Average Waiting Times for 100GB Size Jobs in Experiment 1 138
Figure 39: Total Response Times for 28.9MB Size Jobs in Experiment 1 139
Figure 40: Total Response Times for 312.7MB Size Jobs in Experiment 1 139
Figure 41: Total Response Times for 1GB Size Jobs in Experiment 1 140
Figure 42: Total Response Times for 10GB Size Jobs in Experiment 1 140
Figure 43: Total Response Times for 100GB Size Jobs in Experiment 1 141
Figure 44: Average Execution Times for 28.9MB Size Jobs in Experiment 1 141
Figure 45: Average Execution Times for 312.7MB Size Jobs in Experiment 1 142
Figure 46: Average Execution Times for 1GB Size Jobs in Experiment 1 142

Figure 47: Average Execution Times for 10GB Size Jobs in Experiment 1 143
Figure 48: Average Execution Times for 100GB Size Jobs in Experiment 1 143
Figure 49: Average Waiting Times for 28.9MB Size Jobs in Experiment 2............................. 146
Figure 50: Average Waiting Times for 312.7MB Size Jobs in Experiment 2........................... 147
Figure 51: Average Waiting Times for 1GB Size Jobs in Experiment 2.................................. 147
Figure 52: Average Waiting Times for 10GB Size Jobs in Experiment 2................................ 148
Figure 53: Average Waiting Times for 100GB Size Jobs in Experiment 2.............................. 148
Figure 54: Average Execution Times for 28.9MB Size Jobs in Experiment 2 149
Figure 55: Average Execution Times for 312.7MB Size Jobs in Experiment 2 149
Figure 56: Average Execution Times for 1GB Size Jobs in Experiment 2 150
Figure 57: Average Execution Times for 10GB Size Jobs in Experiment 2 150
Figure 58: Average Execution Times for 100GB Size Jobs in Experiment 2 151
Figure 59: Average Waiting Times for 1 Hour Arrival Rate in Experiment 3 154
Figure 60: Average Waiting Times for 1 Day Arrival Rate in Experiment 3............................. 154
Figure 61: Average Waiting Times for 1 Week Arrival Rate in Experiment 3........................... 155
Figure 62: Average Waiting Times over Different Arrival Rates in Experiment 3..................... 155
Figure 63: Average Execution Times for 1 Hour Arrival Rate in Experiment 3 156
Figure 64: Average Execution Times for 1 Day Arrival Rate in Experiment 3.......................... 156
Figure 65: Average Execution Times for 1 Week Arrival Rate in Experiment 3 157
Figure 66: Average Execution Times over Different Arrival Rates in Experiment 3 157
Figure 67: Average Waiting Times When Disabling Algorithms (Exp1 Scenario).................... 159
Figure 68: Average Waiting Times When Disabling Algorithms (Exp2 Scenario).................... 160
Figure 69: Total Response Times When Disabling Algorithms (Exp1 Scenario) 160
Figure 70: Average Execution Times When Disabling Algorithms (Exp1 Scenario) 161
Figure 71: Average Execution Times When Disabling Algorithms (Exp2 Scenario) 161
Figure 72: Number of Saved Jobs by the Fallback Algorithm (Exp1 & Exp2 Scenarios) 163
Figure 73: Average Waiting Times (Exp1 Scenario) in Experiment 5 164
Figure 74: Average Waiting Times (Exp2 Scenario) in Experiment 5 164
Figure 75: Total Response Times (Exp1 Scenario) in Experiment 5 165
Figure 76: Average Execution Times (Exp1 Scenario) in Experiment 5 165
Figure 77: Average Execution Times (Exp2 Scenario) in Experiment 5 166

LIST OF ACRONYMS

AHS	Adaptive Hierarchal Scheduling
AET	Average Execution Time
AWT	Average Waiting Time
DEVS	Discrete EVent Simulation
EF	External Function
FTP	File Transfer Protocol
GS	Grid Scheduler
GSS	Grid System Scheduler
GIS	Grid Information System
IF	Internal Function
IP	Internet Protocol
IPv4	IP version 4
ISP	Internet Service Provider
JD	Job description
JST	Jobs Status Table
JMR	Job Minimum Requirements
LGS	Leaf Grid Scheduler

LS	Local Scheduler
NWS	Network Weather Service
OF	Output Function
P2P	Peer-to-Peer
PGSS	Peer Grid System Scheduler
RFC	Request For Computation
RFJM	Request For Job Matching
RD	Resource Discovery
SS	System Scheduler
TA	Time Advance
TRT	Total Response Time
TTL	Time To Live
WAN	Wide Area Network

CHAPTER 1: INTRODUCTION

1.1 INTRODUCTION

The current state of computing is equivalent in some respects to the state of the electricity circa 1910s [18]. At that time, electrical power was produced by generators for specific individuals or organizations needs such as oil companies and paper mills. Such electric power was not only limited but also underutilized, since each organization had to build and operate new generators in order to make electrical power available for its needs. The shortage of electrical power at that time, for example, prevented many electrical devices from being manufactured or even researched. In fact, the real influence of electricity in our lives stems from the creation of the electric power Grid that provided (via sharing generators) reliable power at low cost for both individuals and industries. Analogically, the term "Computation Grid" was adopted from the electricity Grid to *amplify computational power* via *sharing* computational resources since both Grids are similar, to some point, in their infrastructure and purpose: obviously, the electrical Grid was also used to increase electrical power via *sharing* electrical resources.

Grid computing cannot afford to ignore many lessons learned from the history of the electrical power Grid. Some of these significant lessons are the reality of business and politics, and the complexity of managing system's resources [18]. Oil companies, for instance, had to maintain backup generators in case some of their electrical generators failed during oil drilling. Ultimately, this led to the need for sharing electrical power with other organizations in order to maximize generator use via sharing, hence increasing the availability of electrical power. In addition to the business needs, electrical Grids developers had to adjust to new regulations that were driven by political forces and simultaneously rise to difficult technical challenges due to the complex nature of the Grid system. For example, a failing power plant causes its region's load to be 'dumped' onto a nearby plant, and if this plant becomes overloaded and shuts off, both regions 'fall' on another nearby power plant, and so on. Thus it is quite possible for one faulty generator to cause a major blackout in the entire electrical Grid as exemplified by the largest power blackout in the world's history: on August 14, 2003 eight U.S. states and the Canadian province of Ontario were affected, leaving up to 50 million people with no electricity for several days.

Reference to "the Grid" started in the mid-1990s [18, 19] to denote infrastructure for both scientific and commercial distributed computing communities and has been gaining popularity ever since. The "Grid" is a parallel and distributed system that enables large collections of geographically distributed heterogeneous systems that usually span over several organizations to share a variety of resources dynamically (at runtime) depending on their availability, capability, user's requirements and any other predefined rules set by local systems and resources owners. The type of sharing in the Grid gives the

impression of a powerful self-managing *virtual* computer [8]. For example, a user of the Grid may enter a command from his/her workstation to run a type of simulation that in fact requires a fast supercomputer on the Grid. Now, from the user's point of view, the workstation is the one that runs the application and report results, not the Grid system. However, in reality, the Grid has to:

- Hunt for the required resources among a massive number of diverse resources and match them to the user requirements knowing that allocated resources may actually be residing on the other side of the globe,
- Authenticate the user's identity and confirm the user's authorization for using the allocated resources. (Security is now probably the most important issue in Grid systems),
- Schedule the user's application to be mapped onto (i.e. use) allocated resources along with other jobs that possibly require access to the same resources. (Grid scheduling is probably the second most important issue in the Grid systems, and is the main topic of this research). For example, a Grid system may need to migrate a job to different resources because either the allocated resources become overloaded (e.g. heavy local load) or simply disappear from the Grid (e.g. resources owner decides to take them off the Grid), and finally
- Report results to the user (e.g. the user receives a notification text message on his/her cellular phone).

The fundamental principle behind Grid computing is that a number of *coordinated and shared* resources (e.g. computers) can solve problems faster and better than what one resource (e.g. computer) can do. These computers can be anywhere – they simply have

to be connected together. Therefore, the Internet can be an ideal choice to link thousands or millions of computers since it already exists and connects the whole world – if a node's IP address is known, it can then receive data from another node.

Grid systems have the potential to offer users access to advanced computing machines and other resources, enabling a solution for massively complex tasks beyond the capabilities of a single machine or local network. For example, the gravitational wave detector network [20, 38] collects a TeraByte of data each day. This data must be analyzed using different algorithms to detect astronomical phenomena such as black holes. The usual analysis of data indicating a black hole cannot, for example, reveal its location. Now suppose that three Intel TFLOPS supercomputers are needed to analyze the 100GB of raw data within an hour in order to enable astrophysicists to determine the event's location and view it via their telescopes before its visual signal fads. (And the time to download the data onto these 3 machines is non-negligible). The Intel TFLOPS is currently the world's most powerful supercomputer: it uses more than 9000 Pentium Pro processors to achieve a processing rate of one teraflop [14], with 594 Gbytes of RAM, and two independent 1 Tbyte disk systems. And it is not easily accessible. Alternatively, a Grid system may be used, with the additional virtue of being able to broadcast results quickly around the planet.

In order to understand the responsibilities of such a computational Grid in a simple way, we suggest answering the following questions from a user prospective:

1) How do I run my job? This is the issue of *Job submission*, which can be addressed in a simple command form.

2) How can I verify my job will run the way I want (e.g. are there suitable resources)? This is the problem of *Job requirements* (and any other imposed constraints) onto a set of resources. (We will see it is similar to looking up an entry from a database table.)

3) How does the system discover resources for my job? Answer: It discovers resources via gathering information about system's resources that have to be matched to job's minimum requirements (i.e., *resources discovery* step).

4) Where does the system execute my job? Answer: It selects best resources based on its knowledge from the discovered resources set (i.e. *resources selection* step).

5) Can the system guarantee my job executes on the selected resources? Answer: No. *Scheduling is tentative*. Note that selected resources are not guaranteed to live in the system more than the jobs requests themselves. However, the system may provide reliable services via, for example, *advance reservations* (as discussed in the next chapter).

6) What will happen to my job if its selected resources disappear? Answer: It migrates to different resources (i.e. issues of *migration* and *re-scheduling*).

7) Why do resources disappear? Answer: Resources may disappear in the system because they are owned by local systems, and therefore are controlled by them.

8) What do I gain in using a Grid system? Answer: Such a system, with great computational power, can solve problems that were considered unsolvable, since a Grid system offers more computational power than the most powerful present computers. Furthermore, it is simply cheaper to *share* expensive resources rather than purchasing them.

9) How does the system manage my job in the presence of other users? Answer: It does it via *Scheduling* (e.g. it balances *workload*s among resources).

10) Where does the system get its computational power? Answer: It gets its power by *sharing* huge sets of resources via the cooperation of participating companies. Simply, the more computers work together on solving a problem, the faster the corresponding Grid. Its power comes from exploiting the *underutilized* resources in organizations. The key observation is that resources are typically busy only about 5% of the time (more on this later).

11) How can I know my job's status and output? Answer: Grid systems allow a user to monitor his/her job (i.e. *monitoring*). Such systems also produce jobs output once they are executed or during their execution (i.e. *termination*).

12) What is the difference between a Grid and *cluster* systems? Answer: A cluster is a group of individual, stand-alone computers that work together and are viewed as a single computing system. The individual computers that make up the cluster communicate with each other via high-speed connections such as a Gigabit Ethernet. Conversely, a Grid consists of multiple systems (e.g. clusters) that work together and span over large geographical regions while maintaining their distinct identities. Most importantly, a cluster system, like an "end system" in a Grid, is owned and controlled by a single entity. In summary, clusters are a form of resources that are usually supported in Grid systems.

13) It is too expensive to build a system to connect companies together? Answer: No. They can be connected via the Internet – we repeat: if a computer's IP address is known, it can receive data.

14) But communication in the Internet is unreliable? Could a Grid be reliable? Answer: Yes. *Fault tolerance* [2] is a major research area in the use of Grids. Achieving reliability is key.

15) How will this system scale? The answer is non-trivial and lies in the notions of hierarchical ad distributed processing.

1.2 WHY GRID COMPUTING?

Why do we need Grids? To answer this question, consider for instance two students. Each one of them has to purchase ten books for the next school's term. Suppose, each book costs about a $100 – each student has to pay about $1000 to buy all books. Now, these two students decide to create a system where each one of them only buys five books (i.e. spends only $500); they *share* the ten books between each other. Of course, a well-coordinated sharing policy is needed to get the full advantage of the system resources (i.e. 10 books). Most probably, for this to work, the two students must not need the same book on the same day or afternoon. Therefore, the system can perform well most of the time.

Because each student actually owns only five books, one student may not lend his/her book to the other student if he/she needs it – the *sharing* problem. To overcome some of these problems, the two students decide to enlarge their system by bringing other students (e.g. say 50) from their school and other schools. Moreover, the students decide to allow different type of resources into the system (e.g. calculators).

Now, the students have probably hundreds of books, calculators, articles, old exams…etc. By building this system:

- None of the students probably needs to purchase expensive school things (i.e. resources) until they graduate,
- Each student still own their things and, for instance, a student doesn't have to give her book up if she needs it,
- Most resources in the system are put into use most of the time (i.e. *boost resources utilization*),
- Multiple resources may be used by one student at one time (i.e. in *parallel*). For instance, a student may need various articles and books for his current research,
- Many students may cooperate – *sharing* ideas among wider audience,
- Each student has an access to a huge collection of resources that he/she may not even dream of (e.g. old exams for the last five years),

Of course, some students may pull out of the system (with their resources) if they are not gaining what they expect from the system. Probably, it is not the fault of the system if it fails; most probably it is the fault of the students who didn't *coordinate* well among themselves to make the created system a success.

Assuming a Grid system among 50 students, the following capabilities are required in order to perform well. The system has to:

- Authenticate student identity. Of course, not anyone is allowed to use the system. Furthermore, a resource owner may only permit (i.e. *authorize*) a group of other students to use his/her donated resource. For example, a student

may put an expensive calculator into the system and want only her friends (e.g. five students) to use this calculator because she doesn't trust anybody else with her calculator. Note that *authentication* and *authorization* fall in what we call system *security*.

- Keep track of what resources it has at all times. For example, when a student adds a book (i.e. resource) to the system; he sends an email (i.e. *advertisement*) with a description of the new book to the student who is responsible of *managing these resources*. In addition, if a student decides to pull out of the system, his resources need to be removed from the resources list. Thus each request to the Grid must trust this Grid will *discover the resources* that *match* the request at hand.

- *Schedule* a (student's) request to use specific resources (e.g. books). For example, once a student *selects* the needed resources, he sends an email to the student who is responsible for managing requests to use the system resources (i.e. the *system scheduler*). Of course the system scheduler must *re-schedule* (i.e. *migrate*) a student's request to use a different copy of a book (i.e. resource), if

    - the owner of that book decides to keep his book and takes it off the system (i.e. *resource disappearing*) or

    - Too many students want to use the same book (i.e. *overloaded* resource) and another copy of that book becomes available.

- Grant (students) access to system resources. For example, students may choose a free delivery system to transfer their goods across schools and departments.

Each student may wait a day or two to receive a book if he is receiving it from a different school. This kind of *delay* is a cost that must be paid when a system is *distributed among geographical areas*.

The example above gives some issues with the use of a Grid when a group of individuals/organization (e.g. virtual organizations [18, 19]) comes together to *share* a large collection of resources). The system becomes very powerful if it offers well-designed capabilities such as security (i.e. authentication and authorization), resources advertisement and discovery, scheduling, balancing workload, job migration and an efficient reaction to resources disappearing of the system.

Benefits of Grids can be extensive. They include:

1) Expanding computing power by the use of underutilized resources

Grids are motivated by the multitude of unused computers on evenings, weekends, and daytime hours that can provide significant computational power. Most machines in a typical organization are busy less than 5% of the time (i.e. 1 hour and 12 minutes a day) [8]. Therefore, these resources are unused for about 95% of the time (i.e. 22 hours and 48 minutes a day). Grids unleash the hidden computing power that is not being used, giving companies a huge expansion in computing power. Several studies reports that utilization can be increased without affecting production drastically [18].

Now, suppose that machines on which a set of applications normally runs most of the time are experiencing a heavy unusual load. All or some of these applications may be executable on unused machines either in the same or in a different organization via the

Grid. The following conditions must be met to enable the Grid to transfer these applications to some idle machines:

- The minimum requirements for the applications [38] must be specified. For example: "application X only runs on INTEL architecture and must finish within three hours".
- The target idle machines must meet the jobs minimum requirements (JMR) and any other imposed conditions by users (e.g. deadlines).

Of course, not all jobs are suitable for Grid computing. For example, a batch job that spends a lot of time processing input files and generating output files makes a good candidate for Grid computing. However, it doesn't make sense to submit a small program that takes a few seconds to run on a typical PC to the Grid because of the Grid's computing overhead (such as scheduling and job transferring delays).

2) Improved productivity and collaboration among organizations

Collaboration brings (in a dynamic and geographically distributed manner) huge resources to form one powerful computing system which is called the "Grid" to *amplify productivity*, allowing widely isolated departments and businesses to create virtual organizations [19] in order to *share* data and resources.

Sharing resources among a number of organizations reduces cost in a spectacular way. An organization may 'borrow' resources (e.g. databases, supercomputers, etc.) from another organization "on the fly". Of course, borrowing is cheaper than buying.

It is likely that the *future* way for companies and government agencies to support research is by providing access to available resources. But how many companies are

willing to donate an Intel TFLOPS or a Cray T3E supercomputer to a university? In comparison, how many companies could be willing to give only access to such machines when they are idle? For a student in a university, there is no difference between evenings, weekends or business hours. However, there is a big difference in a company! Most machines are probably idle on weekends in most companies. Large amounts of research money are spent on buying resources, but with the advent of the ”Grid”, providing access to expensive resources is probably more efficient for everyone. Furthermore, it is much easier to convince the “right people” to gain access to a resource than to expect it to be donated.

Sharing in the Grid starts with data (i.e. “data Grid”) in the shape of files or databases [8]. A “data Grid” can increase data capacities by spanning over many systems and have greater capabilities than on any single system. Of course, sharing is not limited to data, but also expands to other resources, such as software, tools, computers, licenses, and others.

3) Solving complex problems

A “Grid” creates a *virtual* powerful machine that is capable of solving complex problems that were previously unsolvable. Supercomputers rely on thousands of well coordinated processors to solve complex problems that were unsolvable by one computer with one processor.

Suppose now we place three Intel TFLOPS supercomputer to work together in a cluster. They will then be faster and better than one Intel TFLOPS if they are well coordinated. Suppose further that we group three such clusters together. They will then

be more powerful than one cluster, again, if they are well coordinated. Should these clusters be geographically dispersed, they could be thought of as a Grid.

In summary, the Grid is a virtual supercomputer that can span over the entire world and address problems currently unsolvable (Figure 1).

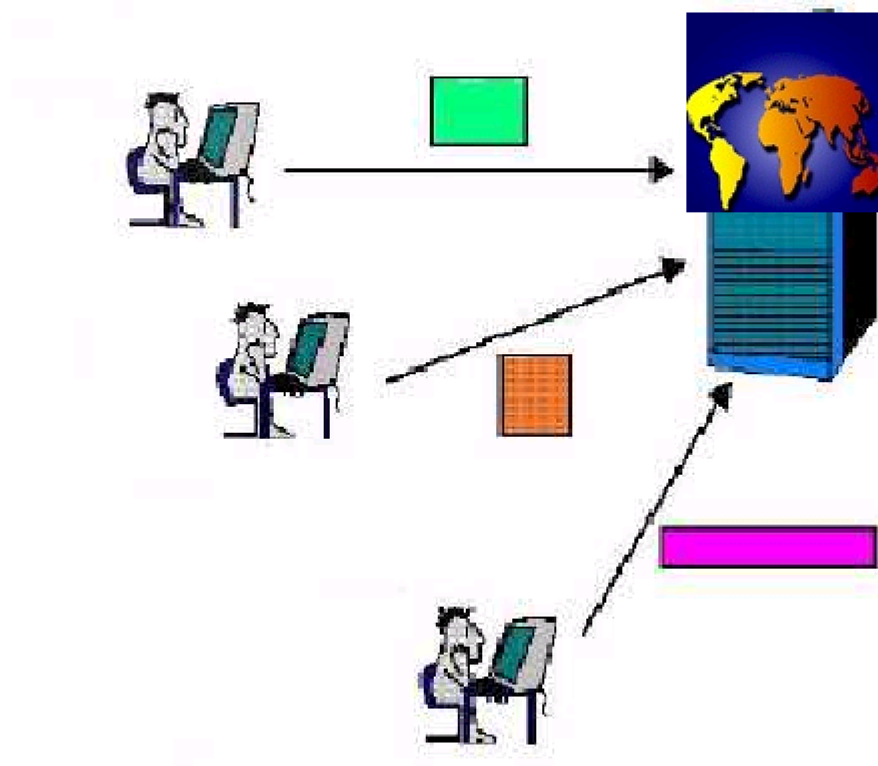


Figure 1: Accessing the Virtual Computing Power of the Grid

1.3 GRID ARCHITECTURE

The system architecture plays a vital role in the overall system performance during the scheduling stages, which are discussed in the next chapter, since the adopted

architecture *dictates* the scheduling schemes in the Grid system. Of course the system architecture should always *scale* while allowing Grid scheduler(s) to:

- Discover resources (for submitted jobs) from huge resources collections,
- Select best resources with respect to a user conditions (i.e. economic, performance, etc.),
- Retrieve jobs from their host machines and submit them to resources in order to execute,
- Collect information about system resources,
- Migrate (i.e. reschedule) jobs to different resources at any scheduling stage, and
- Report results to users.

In this section, we take a brief look at three architecture types: centralized, distributed and hierarchal architectures.

1.3.1 Centralized Architecture

In the centralized scheduling systems, all jobs are submitted to a single Grid scheduler, that does not belong to a specific local system and is responsible for assigning resources to all jobs across geographically distributed domains. Thus, the central Grid scheduler is responsible for tracking the status of every job in the system and must maintain up-to-date information about every resource in the system, as shown in Figure 2. This approach can be found in many current Grid systems [26] such as Condor [13, 42], PBS [33, 36], Maui [24, 30], and LSF [28, 29].

Obviously, when the number of jobs increases, the worse it gets in making scheduling decisions regardless of the way resources are structured. Note that some Grid

Information Systems, like the Monitoring and Discovery System (MDS) [31], structure resources in a hierarchical way to achieve scalability. However, if those hierarchal resources are still accessed by a single scheduler, the overall system performance will still get worse and worse as the number of jobs increases in the system.

Finally, such an architecture introduces one point of failure, as in the case of any centralized system – the whole system comes down when the central scheduler fails to operate.

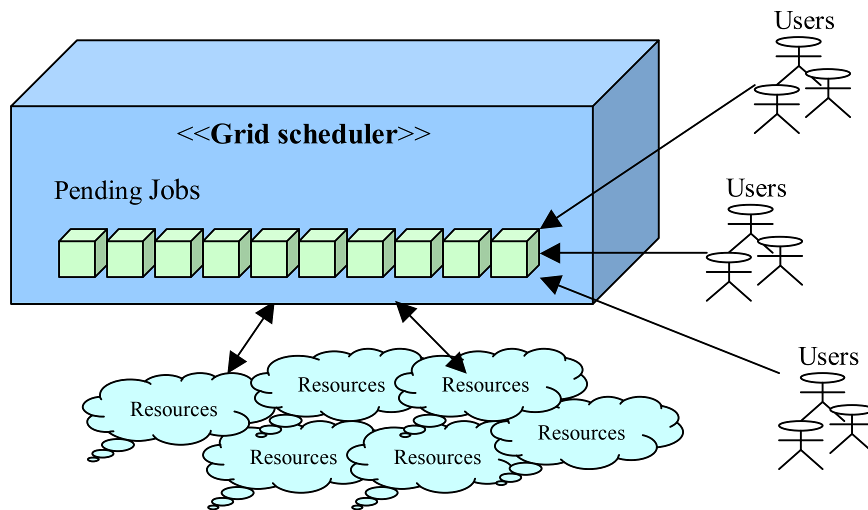


Figure 2: Job Scheduling in Centralized Grid System

1.3.2 Distributed Architecture

In this architecture, several Grid schedulers (GSs) are distributed and coordinate among each other to access system resources as shown in Figure 3.

In general, Grid schedulers (GSs) in distributed systems gather information about their local resources. A user submits a request to the Grid via a GS. In turn, that GS submits the request to its local resources or to remote resources via other schedulers. Of course, this architecture is an improvement over the centralized one with respect to fault tolerance and performance, since scheduling is performed by several distributed Grid schedulers (GSs). On the other hand, it introduces new *non-trivial* issues, such as how those Grid schedulers locate other schedulers in the system.

A peer-to-peer (P2P) system [4, 10, 32, chapters 24-27 in [34], 44] is an example of distributed systems. We found that there is a strong tendency among researchers to use the Peer-to-Peer (P2P) approach in order to replace the centralized architectures in the Grid systems. The P2P systems are currently used to share data among several peers across the network. Gnutella [4] and Napster [32] are examples of such file sharing systems.

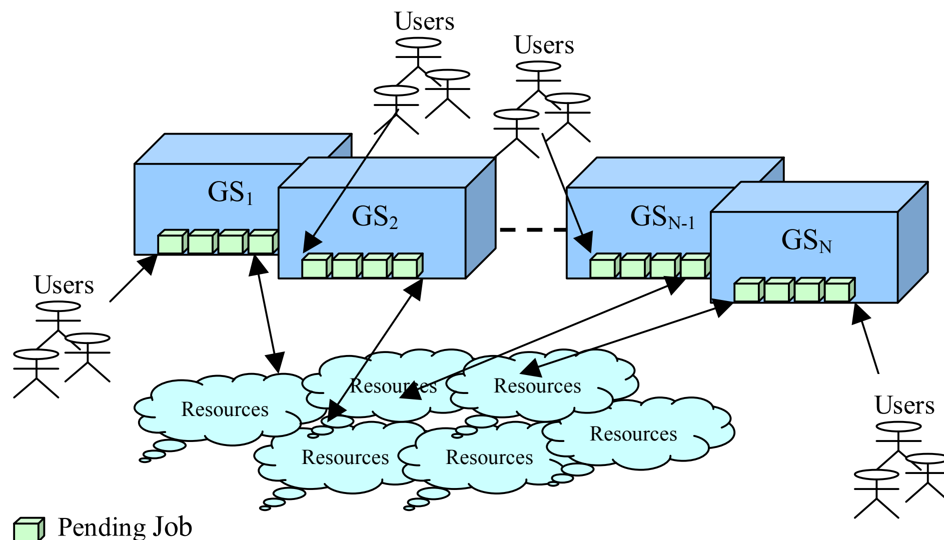


Figure 3: Job Scheduling in Distributed Grid System

In P2P systems, nodes are called *peers*, if a peer is connected directly to another peer; then both of them are considered *neighbors* (i.e. neighbors are not necessarily geographically close to each other). When a peer wants to join the system, it requests a set of neighbors via contacting one or more *contact services*, as shown in Figure 4.

A user's workstation submits a request to the system via one of its peers. The peer in turns forwards the request to its neighbors, and the neighbors forward the request to their neighbors, and so on. Once the request is received by a peer that can satisfy the request, it contacts the initial requesting workstation to offer its service. Note that the source workstation often receives multiple responses. In this case, the workstation ignores the unwanted responses.

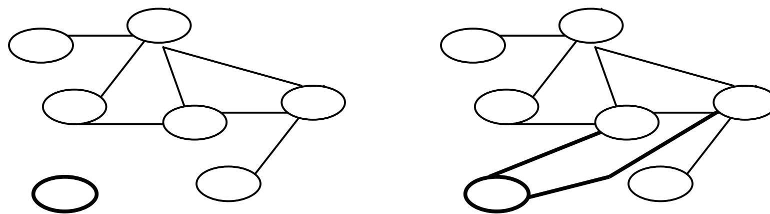


Figure 4: An Example of Joining a New Peer in the P2P systems

Many papers address Grid scheduling with the assumption that resources will be discovered via P2P systems. Thus, in this case, scheduling turns out to be *one* issue: when will a peer *accept* a request from a neighbor or *refuse* that request and forwards it to its neighbors. For example, [44] proposes a Grid scheduling architecture that combines local and Grid schedulers in one unit with two queues: Local queues and Grid

queues. The algorithm in [44] is summarized as follows (i.e. Note that [44] doesn't address how Grid schedulers locate their peers and leaves it up to the P2P approach):

```
     // AWT is the Approximate Wait Time for a job to stay
in
     // the local scheduler queue before it is executed.
     // GTH is a Grid scheduler threshold
     Place a received job in the Grid queue;
     If AWT < GTH then
          Move job from the Grid queue to the Local queue;
     Else
          Job is migrated to a peer Grid scheduler;
```

1.3.3 Hierarchal Architecture

As stated earlier, many researchers have proposed replacing the centralized architecture approach with the P2P systems approach. Few researchers seem to instead advocate a hierarchical approach. For example, [48] states that "*Although it seems obvious that a Grid scheduler may consist of more than a single scheduling layer, many details of an appropriate scheduling architecture have not yet been established*". Moreover, some other researches have ruled out the hierarchal architecture approach

altogether. For example, [44] states that "*A hierarchy where Grid schedulers are organized into a tree and jobs flow up and down the tree is an interesting approach, but we do not expect it to scale as well as a P2P approach*".

In fact, our research not only shows that a hierarchal architecture scales well, but also that it offers a substantial performance improvement over the P2P approach. Moreover, our proposed hierarchal architecture does not need to replace the P2P based Grid systems, but in fact can be combined with the P2P approach in one Hybrid system (see next section).

1.4 THESIS CONTRIBUTIONS

The main contributions of this research are summarized below.

- Propose a *hierarchal architecture* that can be combined with the well-known P2P approach so that it provides better system scalability, efficient resources management, etc. The proposed hierarchal architecture also reduces the average job waiting time, increases system parallelism and deals with user imposed soft conditions (i.e. conditions that a user likes to have, but is willing to continue without them until they become available).
- Propose a *self-discovery* method to be used in the *resource discovery* stage. This method produces a set of logical channels to be used as paths by jobs (in the next scheduling stage) to get to their physical resources. We have observed that many studies jump over the *resource discovery* stage into the second scheduling stage by assuming that all jobs can execute anywhere in the Grid, or simply assuming

that resources will be discovered using the P2P approach. However, as it will be shown, those scheduling stages have to be dealt with in sequence.

- Propose an *adaptive child scheduling* method to be used in the *resource selection* stage. This method uses a self-scheduling scheme by exploring the parent-child relationship. Three rescheduling algorithms are also proposed for this stage:
    - The *Butterfly* algorithm to reschedule jobs when better resources become available,
    - The *Fallback* algorithm to reschedule jobs that had their resources taken away from the Grid before the actual resource allocation and
    - The *Load-Balance* algorithm to balance load among resources.
- Provide a Grid simulation model that can be expanded, in theory, to any desired number of nodes (i.e. we stopped at 2400 nodes). This model not only shows that the proposed schemes are implementable in real life, but also provided a reasonable number of nodes for the simulation of a Grid system. We believe that 75 nodes was the maximum used in other related work on Grid simulations [51].

Ultimately, to obtain end-to-end Grid scheduling, the job execution stage has to be addressed. We did not tackle this third scheduling stage.

1.5 THESIS ORIGANIZATION

The rest of the thesis is organized as follows. *Chapter 2* provides a brief background of the hierarchal scheduling in parallel and cluster systems and it discusses

the three Grid scheduling stages: resource discovery, resource selection and job execution. *Chapter 3* presents the proposed ideas for applying the hierarchical scheduling schemes to manage resources in the Grid systems. It also serves as the specifications for the Grid model in the next chapters. In that chapter, we discuss the proposed schemes for the first two stages of the Grid scheduling: *resource discovery* and *resource selection*. In addition, it introduces three rescheduling algorithms: the butterfly, fallback and load-balance. It also proposes a hybrid system to combine the proposed hierarchal schemes with the well-known peer-to-peer (P2P) approach. *Chapter 4* gives a brief introduction to the Discrete Event Simulation (DEVS) CD++ tools. It also discusses the basic elements that construct the Grid model used to run a series of experiments through simulation. *Chapter 5* discusses the Grid simulation model, its assumptions and the results obtained over several experiments (Detailed result tables are given in *Appendix-A*). Finally, *conclusions* are provided.

CHAPTER 2: BACKGROUND

2.1 INTRODUCTION

This chapter provides a background on the Adaptive Hierarchical Scheduling (AHS) policy and the three stages of the Grids scheduling: resource discovery, resource selection and jobs execution.

The Adaptive Hierarchical Scheduling (AHS) policy [14] has been considered for several systems: Shared memory, distributed memory (multicomputers) and cluster systems. In this chapter, we give a brief introduction to the AHS method. (Please refer to [14] for more details).

Once more, Grid scheduling is performed in three stages. First, *resource discovery*, which produces an initial list of matched resources. Second, resources are selected (i.e. *resource selection* stage) from the list obtained in the first stage based on different conditions and constraints. Third, the Grid execute jobs on the selected resources (i.e. *jobs execution* stage).

2.2 ADAPTIVE HIERARCHICAL SCHEDULING (AHS) POLICY

The AHS method has the system (e.g. cluster) schedulers prearranged in a tree of schedulers. The core of the AHS policy deals with job assignment, affinity-scheduling, and self-scheduling [1, 14]. For self-scheduling, the parent-child relationship between the nodes in the tree is key. Specifically, when a non-root node can handle more work, it initiates self-scheduling by sending a request for computation (RFC) message to its parent node requesting more computation. If the parent node doesn't have work to pass to that child at the time of the RFC, it in turn generates its own RFC and sends it to its parent on the next level of the tree. This process is recursively followed until either the RFC reaches the root system scheduler or a node with unassigned computation is encountered along the path. In the case the RFC reaches the root system scheduler, the latter backlogs the request if there are no jobs waiting to be scheduled. However, if a scheduler with unassigned computation is found, the root scheduler applies the space-sharing policy to the waiting jobs as discussed next.

Job and Task Assignment: The algorithm first determines the ideal number of jobs that can be moved down one level from the parent node to the child node as a single assignment by applying a space-sharing policy to divide waiting jobs among children's partitions. For example, the (root) system scheduler (SS), in Figure 5 has 4 waiting jobs. Suppose that upon receiving an RFC message from S1, it applies the space sharing policy and passes 2 jobs to S1. In this case, S1 will apply the space sharing policy on the received two jobs and pass (say one job) to S2. Now, suppose that S3 sends an RFC

message to S1 to request more work to do. In this case, S1 can pass the other job to S3 without generating an RFC to the SS. Furthermore, S1 or S3 can now apply the time sharing policy on the job it has by chopping it into a number of tasks and distributing it among its children in a time-sharing fashion.

Affinity-Scheduling: Once the ideal number of computations to be transferred is determined, the policy selects the best jobs to be passed to a child. For example, jobs have to have their data in a partition before passing it to that partition.

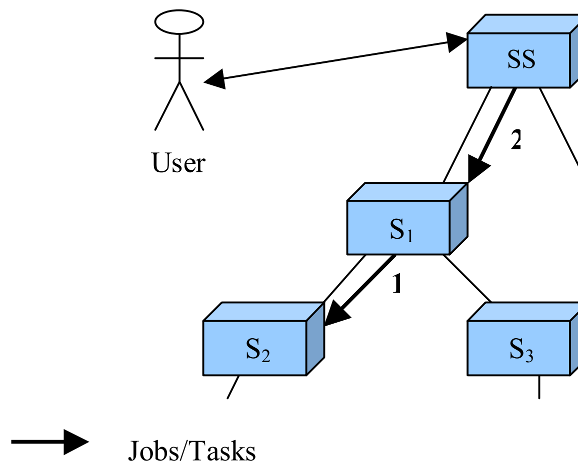


Figure 5: An Example of Hierarchical Scheduling

2.3 GRID SCHEDULING STAGES

Grid systems engage in the management of vast collections of heterogeneous resources that are distributed over vast geographical regions, dynamically available, not controlled nor owned by the Grid system in which the information about the local

systems (that actually own the resources) is often outdated. In this environment, the scheduler (broker) becomes one of the most significant components of the Grid system, since it has the responsibility of discovering resources that meet the user's submitted job requirements, selecting best resources, scheduling jobs on the selected resources, migrating jobs to different resources (e.g. due to workload), executing user's job (i.e. mapping jobs to resources) and finally reporting results to the user.

Scheduling computation (i.e. jobs) in the Grid environment is a very challenging task. Grid characteristics must be taken into account to be able to perform efficient scheduling. Grid schedulers (or brokers) must make scheduling decisions in an environment where they have:

- No control over the resources since they don't own them.
- Distributed resources.
- Dynamic existence of the resources. Resources may be added or removed from the system at any time.
- Up-to-date information collection. Detailed and up-to-date information gathering about resources is essential in order to make the best possible job/resource matching,
- Heterogeneous resources. Jobs must be matched to resources so that they get computed on them as requested. For example, a job may only run on an INTEL architecture and the Linux operating system.
- Distributed Data. For example, a job may need its data to be fetched from remote systems.

- Local management security policies. For example jobs inputs/outputs have to pass through local systems firewalls.
- Network related issues (e.g. connection unreliability, bandwidth, etc.).
- *Tentative* scheduling until the allocation of actual resources: target resources may be taken off from the Grid before a job actually uses them.

It is important to draw the line between Grid schedulers and local schedulers. The primary difference is that local schedulers own and control local resources – they are well informed about their local resources (which is completely the opposite in the case of Grid schedulers). As a result, Grid schedulers should gather information from local schedulers as much as the latter are willing to give up. Note that the lack of resources ownership and control is at the root of most (if not all) problems in Grid scheduling.

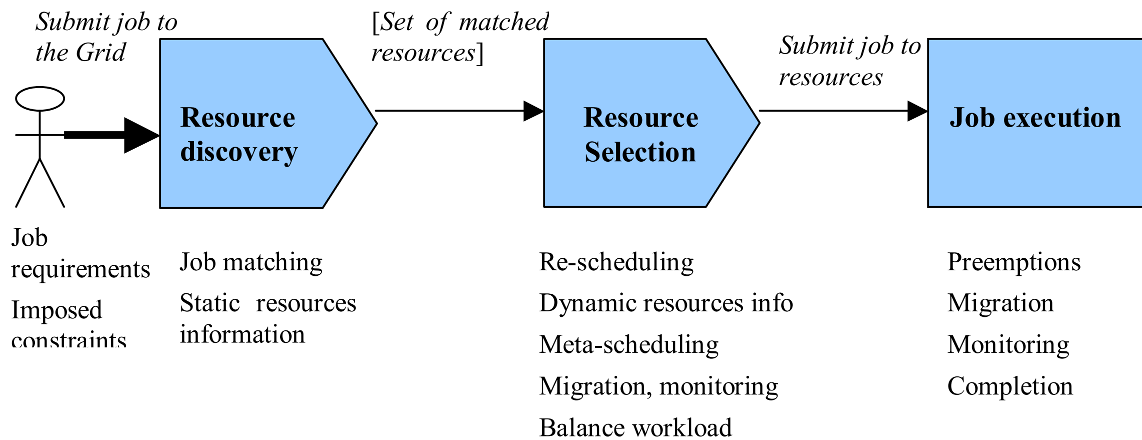


Figure 6: Grid Scheduling Stages and Capabilities.

The three stages (and related issues) of Grid scheduling are summarized in Figure 6. In the rest of this section, we examine Grid scheduling and related issues in more details.

2.3.1 First Stage: Resource Discovery

The first step to schedule a job in the Grid is to find resources for it. The user must be authorized to access the discovered resources and the resources must meet the submitted job minimum requirement (JMR). Of course, Grid schedulers can never carry out this step without gathering the sufficient information to enable them to match the jobs to the suitable resources.

Currently, Grid schedulers may gather information from two main sources: The Grid information system (GIS) and the local schedulers (i.e. the schedulers that actually own and control the resources). Grid schedulers also need to know the job requirements or any other imposed constraints by the user in order to match those jobs to suitable resources.

At the beginning of *resource discovery*, Grid schedulers start with an empty set of resources for a newly submitted job, and end up with a set of *relevant resources* for the new arrived job. In other words, *Relevant resources* are the ones authorized for user access and meeting the minimum job requirements (JMR)). The best resources, based on available information, are then selected in *the next stage* of Grid scheduling, *resources selection* – (see Figure 6).

Resource discovery is a complex task in Grid systems because of resources distribution, dynamic resources availability (where resources come and go dynamically without any centralized coordination), and also due to the concept of *virtual organizations* (VOs) [18,

19] – different groups with different users. The term "*virtual organizations*" was used in [18, 19] to describe a group of individuals or/and organizations share resources dynamically according to certain sharing rules. This adds complexity to resource discovery, since resources are not 'smoothly' partitioned in Grid systems, and different set of resources or/and users may belong to different VOs. Figure 7 shows an example of two overlapping VOs: some of the resources and users belong to both of them, but the remaining of resources and users belong to only one of them. Consequently, users are only authorized to access resources contained in their VOs. Of course, users must first register in a VO to be able to access resources in that VO, and VOs should not only keep track of registered users IDs and passwords but also other information such as billing, access duration, and so on.

In MDS systems [31] (Figure 8), for example, resources of a virtual organization (VO) register in one place that is called "*Index*". An *Index* of a VO can register with another *Index*; hence resources registrations are structured in a hierarchical manner. In this case, as shown in Figure 8, the scheduler places a search query to *discover* resources Once it receives a set of filtered resources, it places another query to *select* resources.

Figure 7: Resources Sharing in Virtual Organizations (VOs).

Figure 8: VOs Indexes Structure in MDS Systems

2.3.1.1 Information gathering about static resources

The term "static resources" is defined in this thesis as the resources that are needed for jobs to be able to execute until completion. Schedulers *match* collected static resources to jobs requirements in order to determine if such a job is executable on those static resources. Static resources usually need manual intervening in order to be changed (e.g., operating-systems upgrade). Note that we will refer to "static resources" hereafter in this thesis as simply "resources".

Therefore, Grid schedulers can never achieve their goal of discovering resources in order to be matched to a user's application without first gathering the necessary information about advertised resources in the Grid. Lots of hypothetical schedulers presume that 100 percent of the needed information has been collected and is correct [43]. This is a great oversimplification in a Grid environment. In fact, information services [9] are a critical part of Grid scheduling, providing fundamental mechanisms for resources discovery and monitoring, thus enabling Grid schedulers to do a better a job, since the more a scheduler is informed about its environment the smarter it gets in making valuable decisions that can play a major role in the overall system performance. For example, imagine that a Grid scheduler queues a job to run on a Linux machine to find out later that the job only runs on a newer version of Linux than the installed one on that machine. The scheduler, in this case, made a dreadful decision because it didn't know what Linux version is currently installed on that machine. The job minimum requirements are also important to enable Grid schedulers to do better matching. We will discuss job requirements in the next section.

Now, a method must be developed to gather information dynamically in order to keep Grid scheduler(s) up to date as much as possible. This is not a trivial task in Grid systems (or any other dynamic environment) where resources may appear or disappear dynamically at run time. For example, in the mobile IP systems [37], a laptop computer can be connected to any network and can still get access to the Internet. An event must be triggered to inform the network router of the existence of that laptop (so that it can get access to the Internet). This can be done in two forms: First, the laptop may broadcast a message (i.e. called advertisement message) on the network to say simply "I am here" to notify the network router, enabling the router to do what it has to do to ensure of routing packets to the subject laptop. Alternatively, 'somebody' (e.g. network router) has to keep sending a special message similar to the ICMP message (i.e. called solicitation message) to discover if anybody just got attached to its network, allowing, for example, the above laptop to discover it.

The dynamic connecting approach of the mobile IP can be applied to the Grid systems. A Grid scheduler has to receive some kind of message (i.e. advertisement message) from somebody to inform it about new resources or changes in the current resources. Furthermore, a Grid scheduler may also send a solicitation message to check for example the status of certain resources. Note that some resources (once they are reserved to be used) may require schedulers to keep sending periodic confirmation messages to validate the reservation: if such message is not received by resources within certain time, the reservation will then be cancelled. These types of messages may be extended to be used as a solicitation messages by the Grid schedulers, since they have to reach end-systems anyway.

Grid Information System (GIS)

Grids schedulers may gather information [43] about resources from a Grid Information System (GIS) which in turns gets its information from the actual resources. The Monitoring and Discovery System (MDS) [31] is an example of GIS systems. The European Data Grid Relational Grid Monitoring Architecture (R-GMA) and Hawkeye, part of the Condor project [13], are other examples of GISs [60]. In the four sets of experiments that were conducted in [60] to assess the performance of service components of MDS2, R-GMA and Hawkeye GISs under different conditions, they found, due to high loads, a strong advantage to caching or pre-fetching the data, in addition to the need to have primary components at well-connected sites.

Furthermore, it is highly important to agree on a protocol (i.e. language) to describe resources. However, this is an area of on-going research and there is debate on how to structure a resource description and what *schema* to use (e.g. XML) [43].

The ClassAds Language

As described above, information must be collected to enable Grid schedulers to match applications to resources and a technique must be defined to represent resources and applications. As an example, the classAds (classified advertisement) language [40, 42] is used in Condor [13] matching. It is generic language that can be used in resource discovery and matching for dynamic heterogeneous systems like the Grid where resources ownership and control is distributed, with the possibility of resources disappearance/appearance. The following shows an example of a resource advertisement in ClassAds language. See [40, 42] for more details.

```
[
Type = "Machine";
Activity = "Idle";
Disk = 80.0G;
Memory = 256M;
LoadAvg = 0.045;
MIPS = 104;
Arch = "INTEL";
OpSys == “LINUX”;
KFlops = 21893;
Name = "nice";
Requirements = Type == "Job" && LoadAvg < 0.3
]
```

The following shows an example of a job submission in ClassAds language. This job description (JD) defines the submitted job requirements – we will revisit the job requirements subject later in this section.

```
[
Type = "Job";
CompletionDate = undefined;
Cmd = "run";
```

```
Requirements = Arch == "INTEL" && OpSys == "LINUX" && Memory
> 20
Rank = (Memory > 32) * ((Memory * 100) + (IsDedicated *
10000) + MIPS)
]
```

Local scheduler information

The more information Grid schedulers know about local resources, the smarter they become. Therefore, the Grid schedulers should gather information from local systems as much as they are willing to give. Local schedulers are the best source of reliable information about their resources, since they control and own those resources – nobody can use local resources without going through those local schedulers.

The following shows some of the attributes that can be gathered from local schedulers, enabling them to make smarter decisions. These attributes are mainly based on [28, 48] where [28] offers objective re-assessment (e.g. advanced reservation vs. guaranteed completion time) of the communication attributes that are listed in [48].

Advanced reservation

Advance reservation [48] defines the timeframe (start and end times) to limit a set of resources to a specific user or users. Some systems like Maui [24], a default time length is created if reservation timeframe is not specified. This feature can be very helpful when scheduling a critical job that has to finish before a deadline. In the black hole example that was described earlier, three Intel TFLOPS supercomputers were needed to analyze the 100GB of raw data within one hour to enable astrophysicists to determine the blast location in space and view it via their telescopes before its visual

signal fades. In this case, not only reservation may be needed, but also guaranteed completion.

Guaranteed Completion Time

Local schedulers still reserve the right to decide when a job will start and end within a given timeframe. However, they guarantee that jobs will be completed before the given deadline. For instance, backfilling based scheduling (i.e. a new job is allowed to overtake earlier queued jobs if its early execution will not delay others) can predict the maximum completion time but not the starting time of the job's execution.

Resources allocation on demand

Multistage jobs are possibly better suited with different resources for different stages. For example, a job requires high bandwidth for database transfer stage, but it doesn't at a subsequent stage. Therefore, that job may reserve a high bandwidth for only the database transfer stage, but not for other execution stages.

Accordingly, local systems that support offering potential resources *on demand* for the Grid are taken into account by the Grid schedulers when scheduling multistage jobs.

Resources cost

Grid schedulers can use this information to evaluate different resources and select the best-suited offer for a job. For example, assume a company provides two sets of clusters to the Grid: Set A and Set B. Also assume a job can execute on both sets of clusters and requires input data files that only exist on set A. Now, the cost is high to schedule that job on set B because a database must be transferred to it. Cost is usually

used as a general term that may indicate performance, charging for the use of resources, and so on.

Job workflow

Grid jobs are often complex and contain various stages that depend on each other. A local scheduler may take into account dependencies between allocations if they are provided by the higher-level scheduling instance.

In multistage jobs or in a complex job execution, a job stage may not start execution because of its dependency on another required resource such as another job's output, another stage execution output of the same multistage job and so on. These dependencies are called the *workflow* of a job. For example, a job may not start unless a database is transferred to the subject job's location.

The workflows are very complex in the Grid environment and therefore workflow management [15] is required for workflows with hefty number of tasks and completion time (e.g. days). Workflow management in GriPhyN [15, 22], which is outlined below, is an example of these systems. It must:

- Locate which components in the job generate the desired data outputs.
- Create components set based on their execution order.
- Locate the physical files of each component in the components set. This creates a set of components physical locations (CPL).
- Identify the components output locations (COL) based on the computational requirements of the CPL set to execute them in line with the discovered resources.

- Determine the physical locations of the input data files set and select locations are more appropriate given the COL set.
- Augment the workflow description to include jobs set to move components (CPL) and input data files to the appropriate target locations (COL).

Cancellation Policy

Some local management systems define rules that must be met by the Grid schedulers to keep resources allocation valid (i.e. before actual execution of a job). For example, local schedulers may require a confirmation message for each job or jobs that are scheduled to use the local resources, and if not received within some time, allocated resources are then cancelled. The number of such cancellations measures the level of local resources reliability.

2.3.1.2 Job requirements and matching

The job descriptions (JDs) describe the jobs minimum requirements to be able to execute on certain resources, hence they enable schedulers to discover resources for those jobs via matching them to advertised resources. The syntax of JDs can be expressed in a language similar to ClassAds that we described briefly earlier in this section. Thus, the job minimum requirements (JMRs) are related to the type of gathered information about resources. JDs should then describe jobs from these angles. Job requirements can be divided, as in [38], to three categories: computation requirements, data requirements, and network requirements.

Computation requirements

Grid schedulers should be informed (via JDs) of the computational needs of a job in order to execute that job until completion. The importance of job requirements in matching the job to the best possible resources by schedulers dictates the need for jobs to be able to sufficiently describe their requirements.

Computational resources go beyond the type of resources such as architectures, memory, disk space, operating systems, etc. to other important factors such as the required time for a job to start and to complete, execution cost (e.g. price per hour), resources reliability, machines performance and failure rate, etc. All of those factors should be considered by schedulers to make a good choice in matching and selecting resources for a job.

As described earlier, a Grid environment presents a challenge in matching jobs to large collection of heterogeneous resources – there are too many dissimilar types of computing architectures, operating systems or even versions of operating systems, software installations and environments, such as shells and compilers. A given application may be compatible with only restricted set of computational resources. For example, if a job is limited to an INTEL architecture and a specific version of Linux operating system, then Grid schedulers have no choice but to schedule it on a machine that meet these requirements, regardless of the number of idle machines that don't meet that job minimum requirements.

Data requirements

Data is essential in computing a job and cannot be ignored. Consider, for example, a job that requires its files to be retrieved from remote location. The time required to transfer those files must be taken into consideration when scheduling that job on selected resources.

Network requirements

Data intensive computation applications dedicate the biggest fraction of their execution time to transferring data. Such applications require minimum assurance of network bandwidth or communications reliability and speed between network nodes. For this reason, those applications count on guaranteed reservations of network bandwidth to achieve a satisfactory degree of Quality-of-Service (QoS).

Data-intensive applications can be very difficult to support in Grid systems [18], since they require high bandwidth data rate all the way between resources (e.g. remote data repository) and application's home (e.g. user's workstation) whose bandwidth tends to be smaller than local disks bandwidth. As stated earlier, the main goal of building Grids is to increase resources utilizations. Therefore, we can never expect a machine on the Grid to stay idle while waiting for huge data to be transferred to it – machines can execute other jobs while waiting for the data to be transferred and cached locally.

Networks usually impose constraints on foreign jobs communications. For example, firewalls can stop communication between any two machines, and system administrators can shut down services if they misinterpret remote accesses.

2.3.2 Second Stage: Resources Selection and Job Scheduling

In the previous stage (resource discovery), the Grid schedulers gather static information about resources in order to be able to match them to the JMRs. For example, if a job requires a UNIX machine, the Grid must then allocate a UNIX machine to be able to execute that job.

Now, in this stage, Grid schedulers have to select resources from the obtained list in the resource discovery stage so that they schedule matched job on them (in order to be executed later). The *resources selection* is based on users imposed constraints (e.g. time, cost etc.) and more collected dynamic information (e.g. load, prices etc.). For example, a user may require that her job to be completed with one hour or less at a price of $100 per hour.

In this stage, Grid schedulers meet new challenges due to the dynamic nature and characteristics of the Grids, such as:

- Mapping jobs to best resources based on their knowledge via gathering dynamic information about resources (e.g. workload) and user's imposed constraints (e.g. execute a job with a certain time).
- Rescheduling a job on different resources because of reasons such as disappearing targeted resources from the system, overloaded resources, better resources become available, and so on.

Grid schedulers have to gather dynamic information about resources to be able to carry their objectives at this stage to comply with users imposed constraints and to increase parallelism in the system so that jobs are efficiently scheduled together. The

kind of dynamic information, such as resources accessibility and system workload, etc., can be provided by different services, like for example NWS (Network Weather Service) [58, 59].

Note that predications using mathematical algorithms may also be used to improve resources selection at this stage; [49] is an example of such algorithms.

2.3.2.1 Dynamic information gathering

The Network Weather Service (NWS) can be used to forecast the performance in metacomputing environments (i.e. gathering dynamic information). [59] discusses how the NWS is designed, and how it predicts the performance of a metacomputing system. [59] also focuses on the NWS ability to predict the system's "weather" via TCP/IP end-to-end latency and throughput.

Grid schedulers may also collect cost information to be able to survive in the reality of the business world. For example, a user requires using expensive ASIC tools in the Grid, but for not more than a $100 per hour. For the Grid to be able to meet this request, it has to then collect additional information like prices. GRACE (Grid Architecture for Computational Economy) [7] is an example of systems that gather this type of information for Grids. It even enables customers (i.e. users) and vendors (resources owners) to negotiate the cost of resources according to the expected starting time, the expected ending time, or the required storage, etc.

2.3.2.2 Rescheduling

Rescheduling is modifying initial matching in response to dynamic system changes like resources disappearance or other jobs load in the system.

2.3.2.3 Metascheduling

Metascheduling [16, 43] is the harmonization of scheduling of several applications running on the same Grid simultaneously. The system takes into account both the needs of an application (i.e. requirements and constraints) and the overall performance of the system.

Systems usually use databases to store information about all applications in the system such as the status of applications, the predicted execution costs, and so on.

2.3.3 Third Stage: Job Execution

This stage occurs once a job is transferred to the selected resources and starts executing.

2.3.3.1 Preemption and Checkpointing

Preemption allows jobs to be interrupted (i.e. blocked) while running, to be restarted later. Some local management systems may permit temporary preemption of a job by a Grid scheduler (i.e. higher-level scheduler). Jobs can then be resumed without a loss of work because of *checkpointing* (i.e. checkpoint file is created).

Condor [5] uses *preemption* in a number of ways in order to apply the policies supplied by both users and local management systems, for example:

- When an owner reclaims her computer after an absence (e.g. by touch of the mouse),
- When a computer is busy with another local job or any other reason based on the owner's policies, or
- When another computer becomes available to execute the Grid job.

2.3.3.2 Cycle harvesting

Cycle harvesting is a way to get the most out of unused computers by repeatedly using workstations that otherwise would be idle. For example, if a computer is configured to use this method, it starts executing Grids jobs as soon as it becomes idle for a period of time. This can be useful in an organization with too many unused computers for a long period of time such as weekends, nights or even lunch. The workstation owner can regain his machine once he starts using it again, maybe with a touch of the mouse, the same way as screensaver works. PBS [33] systems support this type mechanism.

CHAPTER 3: HIERARCHICAL SCHEDULING IN GRID SYSTEMS

3.1 INTRODUCTION

This chapter proposes ideas for applying hierarchical scheduling scheme to manage resources in Grid systems. It also serves as the specifications for our Grid model used in the next chapters. In this chapter, we discuss the schemes we propose for the first two stages of the Grid scheduling: At the end of this chapter, we combine our proposed hierarchical architecture with the well-known technique of Peer-to-Peer (P2P) systems in a single hybrid system. Note that, in this thesis, a Grid scheduler (GS) can be referred to as node, scheduler, child or parent. The Leaf Grid scheduler (LGS) is the Grid scheduler that resides directly on top of resources and is connected with them via local scheduler(s). The root of the Grid tree is called either the system scheduler (SS) or the Grid system scheduler (GSS).

Resource discovery, in the general method, must produce a set of matched resources to a user's job. A Grid system must collect detailed dynamic information (e.g. system load) about the available resources to make good scheduling decisions. Of

course, the more information a scheduler knows about the system (e.g. load), the better decisions it makes.

In our hierarchical approach, the resource discovery stage may be performed using one of two methods: layering resources (first method) or self-discovery (second method). We have implemented the second method because it is more efficient and easier to implement. In fact, we have come up with the self-discovery method during the implementation of the model.

In the Layering resources method (ie the first method), Grid schedulers (GSs) receive resources advertisements from their children; they update their tables and pass these advertisements (about static resources) to their parents in the hierarchy. Actually, this method turns out to be non-trivial to implement, since a child may advertise different set of resources. This difficulty was our motivation to come up with the second method.

In the Self-discovery method, the GSS initiates resource discovery for one or more jobs at the same time by passing their JMRs as one block to all children partitions (i.e. children subtrees). At the end of this method, all schedulers will have none or a number of job bags for the jobs that are executable on their partition, where each distinctive bag is associated with one logical channel (i.e. please see section 3.2.1 for more details). Thus the main difference between the two methods is that schedulers in the first method *collect* all advertised resources in their children partitions. Conversely, in the *Self-discovery* method, schedulers are only concerned about the jobs that are executable on their children partitions, hence, canceling out irrelevant information about resources.

Therefore, at the end of the *resource discovery* phase (see Figure 9) a set of logical channels are produced that every job must take to reach its final physical resources.

Consequently, the resource discovery stage affects directly the way jobs are scheduled on selected resources. Note that Leaf Grid schedulers (LGSs) may receive resources advertisements from anywhere as soon as they reflect the resources below them. For example, LGSs may receive advertisement from a Grid Information system (GIS) or from local schedulers (i.e. note that LGSs may be combined in one unit with local schedulers).

The *resource selection* (i.e. job scheduling on selected channel) stage (see Figure 9) focuses mostly on scheduling multiple jobs simultaneously, re-scheduling because of workload or disappearing/reappearing of resources. Note that moving jobs from parent to child is according to the *adaptive hierarchy scheduling* (AHS) method, which in turns uses the collected dynamic information to schedule a job on a specific child's partition. Three rescheduling algorithms are also used in this stage: Butterfly, Fallback and Load-Balancing algorithms.

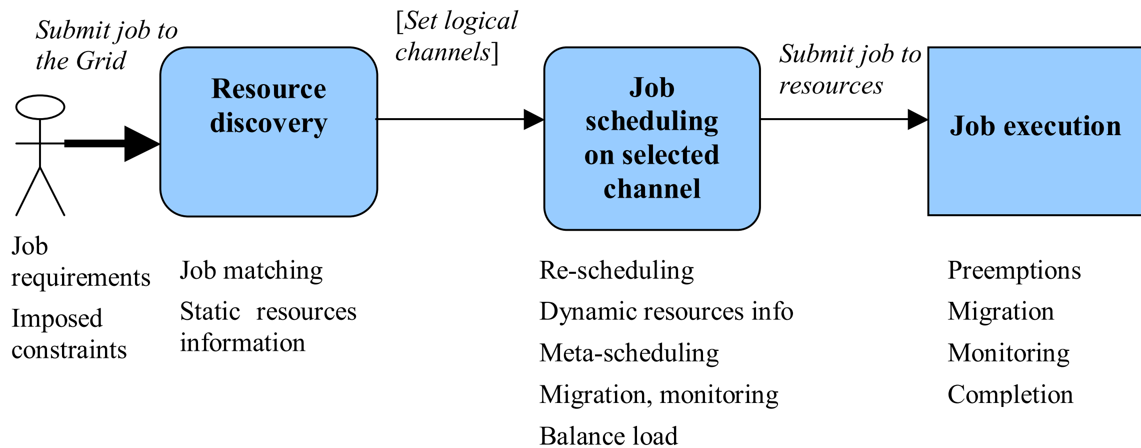


Figure 9: Hierarchical Grid Scheduling Stages and Capabilities – See also Figure 6.

In this thesis, we don't address the *execution stage* (see Figure 9). However, we assume that once a leaf GS is ready to map a Grid job to resources, it contacts the user's workstation to retrieve the physical job. Leaf GSs map a Grid job in the same way a typical Grid scheduler would do in the *execution stage* of the general method. For simplicity, we assume that a job executes until completion. However, we are still interested in the location of the resources that execute a job (see the Butterfly algorithm in this chapter).

3.2 HIERARCHAL GRID SCHEDULING STAGES

Grid schedulers are structured in a tree form that we call *Grid tree*. Users submit their jobs to the Grid via the Grid system scheduler (GSS), which is the root node of the Grid tree. Theoretically, a user can submit the whole job as a batch, with its description (JD), to the Grid system. Obviously, transferring entire jobs in an environment like the Grid is very expensive as explained below. Thus, we assume that a user only submits a job description (JD) to the Grid System Scheduler (GSS) where the job description includes the job minimum requirements (JMR) and any other imposed constraints by the user. The GSS then adds more information to the job description (e.g. sequence number), discovers resources for it and schedules it according to the AHS method on one of its children. The system allocates (i.e. grants actual access to) the needed resources for the job when it reaches a leaf Grid scheduler (i.e. a node on top of resources). The LGS connects directly with the user's workstation and serves as a middleware between the

user workstation and the allocated resources as shown in Figure 10 (note that LGSs may be combined with local schedulers in one unit).

Transferring entire jobs when actual resources are allocated (or about to be allocated) can have a lot of benefits:.

First, the nature of the Grid allows it to handle huge jobs.

Second, the user may interact with resources directly during the job execution stage giving her the impression of running the job on her workstation. In this way, the user may monitor the job progress more closely and may even decide to abort the job before it completes. We will re-visit this point when we discuss the *butterfly* algorithm.

Third, some jobs can't be executed as a batch. For example, a user wants to build software project which is stored under a configuration management tools (e.g. ClearCase) on his workstation. Of course, the user would like to submit just the JD and tell the Grid where to find the makefile and the source code. Although syntax for job description to the Grid system is out of our research scope, nevertheless, we would expect something similar to the following (see also chapter 2 for classAds [42] examples):

```
Type = "Job", Size = 20M, Dir = "/usr/project", Cmd =
"make", Cmd = "run proj.exe", JMR= (Arch = "INTEL" or
"AMD", OS = "Linux")
```

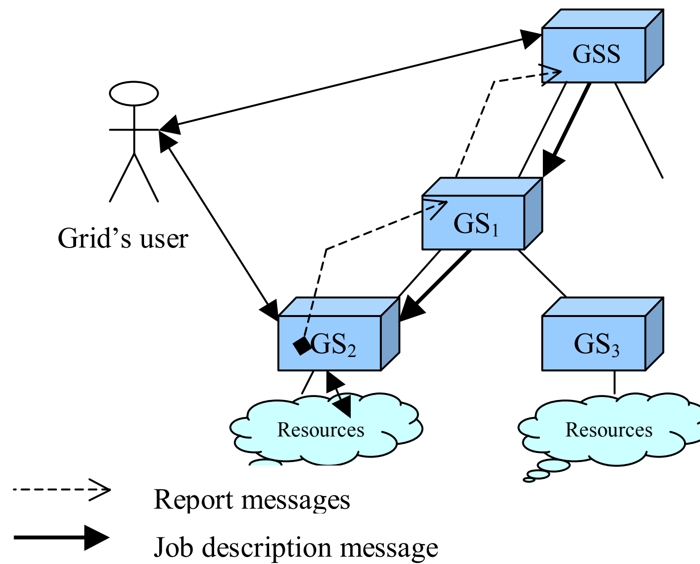


Figure 10: Transferring User's job to Allocated Resources.

The Leaf Grid scheduler (LGS) that transfers the necessary preparation (e.g. source code) to execute the job serves as an agent between the user workstation and the allocated resources for the job. The main drawback for this scheme is a *security* issue since the user workstation runs directly on the top of the resources via an LGS. Security is a fundamental issue in Grid systems but is out of the scope of this thesis. (Please refer to [55] for security related issues in the Grid environment). In essence, when expensive resources are shared in the Grid systems, their owner must make sure that these resources are only used while well-protected.)

In this thesis, we refer to job description (JD) as the submitted job minimum requirement (JMR) and any other imposed conditions by the user. Although we sometime view a JD as a job when it is transmitted from one GS to another, we still only

mean a job description (JD). Conversely, we always call the transfer of data from user's workstation to a LGS a job or application.

In summary, Grid schedulers (GSs) operations, in our approach, are:

- Resource discovery (produces set of logical channels – see Figure 11). We discuss two methods (but we've only implemented the second method):
    - In the first method (layering resources),
        - Schedulers collect resources advertisements about static resources from their children.
        - Job matching uses JMRs and the available information about static resources.
        - The Resource discovery phase is repeated for a job at every level in the Grid tree.
    - In the second method (Self-discovery),
        - Information about resources is only collected and matched by leaf Grid schedulers (LGSs).
        - Each scheduler inserts matched jobs (only Ids) into bags and passes those bags to their parents, hence, a scheduler knows matched jobs on a partition by looking into those bags.
        - The Resource discovery phase is performed one time for a job.

- Schedule jobs on selected resources (on their children's channels) according to the AHS method, users imposed constraints and any collected dynamic information (e.g. system load).
- Monitor scheduled jobs on their partitions. They keep information about the jobs that they've already scheduled on their children's partitions (called jobs status tables (JST)). By doing so:
    - It becomes easier for GSs to make rescheduling decisions.
    - Easier for GSs to recover from failures by duplicating vital information. For example, suppose a GS fails and one of its children then inherits its place to become the new parent as described in [2]. The new parent then merges its JST (i.e. because used to be child) with its new children's JSTs (i.e. its former siblings) to recover the dead parent's JST.
- Balance load among its children's channels. See the *Load balance* algorithm.
- Reschedule jobs because of resources disappearance. See the *fallback* rescheduling algorithm.
- Reschedule a job on a preferred resource, which was unavailable at the time of that job scheduling. See the *butterfly* rescheduling algorithm.

Of course it is unrealistic to jump into job scheduling phase (on selected resources) without first going through the resource discovery stage. We could have assumed that all jobs are executable anywhere in the Grid in order to exclude the resource discovery stage from our work. Indeed, many publications make this oversimplification (e.g. see [17]). We chose to address the resource discovery stage because as it is going to be obvious

through out this chapter that resource discovery has a direct influence on the design and implementation of the job scheduling and resource selection stage.

For the rest of the chapter, we first add more discussions on the *resource discovery* stage in our approach. We then discuss the adaptive hierarchical scheduling (AHS) method, which is responsible for *pushing* Grid's jobs into children partitions. We then illustrate three rescheduling dynamic algorithms.

Finally, please recall that we assume that jobs run to completion when they are executed, since job execution stage is out of this thesis scope.

3.2.1 First Stage: Resource Discovery

The purpose of this stage is to produce a set of *logical channels* to be used as paths by jobs, in the next stage, in order to get to their physical resources (see Figure 9). Logical channels serve as a map for jobs to know how to reach resources that can meet their requirements.

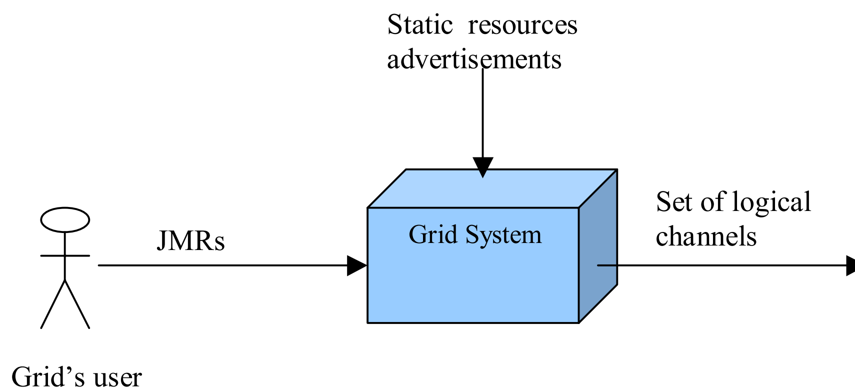


Figure 11: Resources Discovery Stage Overview

Logical channels are generated based on both gathered static resources advertisements by the system (via LGSs) and the JMR(s) for subject job(s) as shown in Figure 11. Note that the job description (JD) of a job contains both the JMR and any other imposed conditions by the user (e.g. cost).

JMRs can be submitted explicitly to the GSS as part of the job description (JD) or submitted implicitly by submitting the entire job as a batch to a queue that is associated with a set of resources, which in turns submits it to the GSS, as shown in Figure 12. Either way, the JMR for a Grid job is still submitted to the GSS to discover resources for it.

Grid systems also need to gather static information from resources in order to be able to perform job matching. By static information we mean the information that is needed from resources to be able to determine that a specific Grid job is computable on those resources. These types of resources usually need intervening to change them such as architecture, operating systems and so on. Schedulers can gather them either from a General information system (GIS) or directly from local schedulers. Today, Grid schedulers and local schedulers are combined in one unit [44].

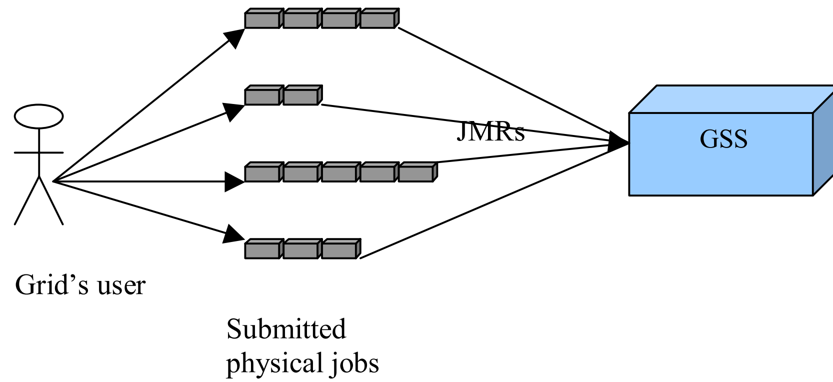


Figure 12: Submitting Actual Jobs as Batches

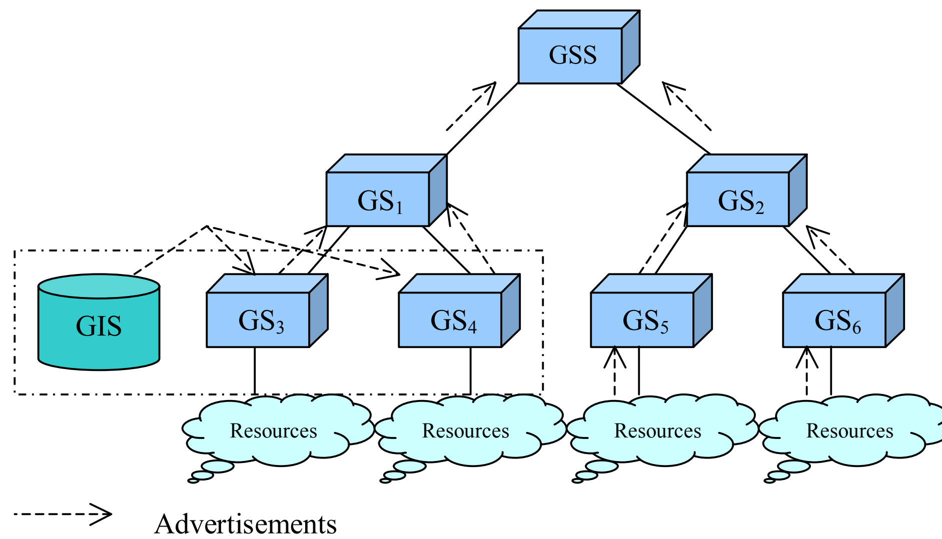


Figure 13: Illustration of Advertisement Messages

In the following two sections, we discuss two methods of resource discovery in the Grid trees: *Layering resources* and *Self-discovery*. The main difference between the two methods is on how to produce *logical channels*, which jobs must take in order to reach physical resources.

3.2.1.1 First Method: Layering Resources

In general, information gathering can be via probing resources (i.e. through agents) by sending some type of solicitations or by getting advertisements from those resources. For example, as shown in Figure 13, GS_3 and GS_4 receive advertisements directly from a GIS system whereas GS_5 and GS_6 receive them from local schedulers (or resources agents). Now, when a GS receives an advertisement message, as shown in Figure 13, it

- Updates its resources and
- Passes the advertisement to its parent in the hierarchy tree.

Grid schedulers (GSs) manage resources on their partitions in tables that we call recourse managers (RM), which are built in hierarchical structure. This way, each GS manages only resources on its partition. Thus, it becomes easier to match resources to a job minimum requirement (JMR) since a GS only worries about resources that exist on its partition.

The Recourse manager (RM) of a GS is a database table about resources of each of its children. It doesn't mean that a parent's RM has to keep a detailed resources information about each of its children (i.e. it may only keep a reduced resources set (RRS) of its children resources). For example, consider the case when a child's RM contains an INTEL architecture with Linux operating systems resources. In this case, the parent's RM may only store "INTEL architecture". Therefore, when a job requires an INTEL architecture, the JD is passed to all children to verify if this job is still computable on the

parent's subtree. However, if a JD doesn't match a child partition, it means this job can't use that child partition (i.e. it can't be scheduled on this specific partition).

Recourse managers (RMs) are essential in the scheduling process since GSs must always verify that a job will compute on a child's partition before it schedules the job on that child's partition. The fact that a job may not be computable on certain children partitions because of lack of resources, adds more complexity to the Grid scheduling process. Furthermore, children may advertise different sets of resources because they receive distinct resources from grandchildren,. For instance, Figure 14 shows a Grid tree of four levels of schedulers. Now suppose that GS_7 and GS_8 advertise different resources. This prevents GS_3 from advertising one type of resources, since jobs that can execute on GS_7's partition not necessarily can compute on GS_8's partition and vice versa. Therefore, GS_3 should advertise two types of resources. Suppose further that GS_4 advertises the same resources as GS_8. In this case, GS_1 advertises two types of resources since it stores one type of resources for both GS_4 and GS_8 (i.e. received via GS_3) and the other type for GS_7's resources (i.e. received via GS_3).

Figure 14 shows the multiple paths, which we call *logical channels*, that jobs must take in order to reach resources that can execute them. Schedulers, in the next stage, must push jobs into channels that will be able to guide jobs to their matched physical resources.

A scheduler rearranges the data structures of resources, if needed, according to received advertisements from its children. For example, advertisements may cause a scheduler to divide a set of resources into two or more, or to combine resources from different children into one set of resources.

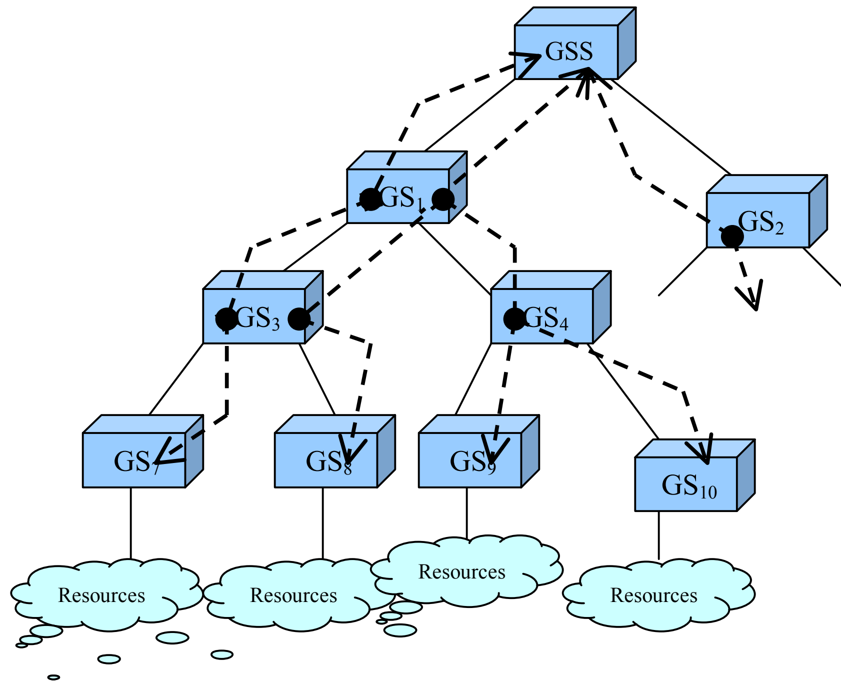
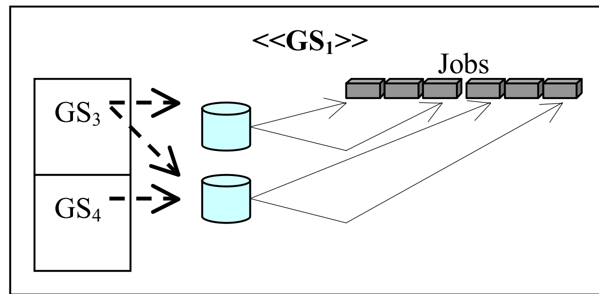
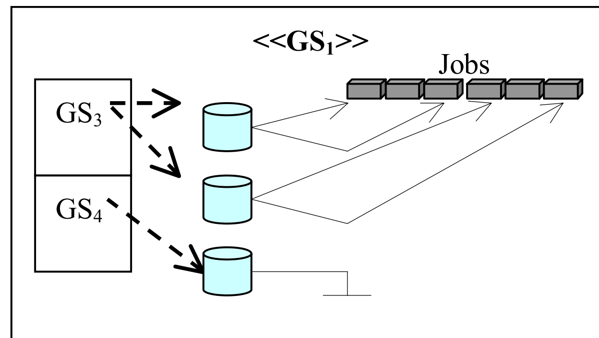


Figure 14: Logical Channels for GSs with Different Static Resources

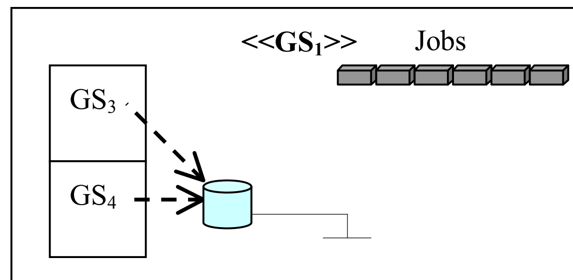
Consider Figure 15, as an example, which shows an inside look of the GS_1 scheduler of Figure 14. The first subfigure (in Figure 15) shows that GS_1 has two children GS_3 and GS_4. GS_3 advertises two types of resources where one of them is similar to the advertised one by GS_4 (i.e. similar resources serve as backup resources to each other since they share same channel). Now, queued jobs in GS_1 are actually waiting to use one of three logical channels.



- GS_1 internal illustration from Figure 14
- GS_3 advertises two distinct sets of resources
- GS_4 advertises one set of resources and similar to one of GS_3's types



- GS_4 advertises different set of resources
- No jobs are executable on GS_4 after resources change
- GS_4 previous jobs didn't lose their resources, since they still executable on GS_3



- GS_3 advertises different set of resources, but similar to GS_4.
- All jobs lost their resources and must be re-scheduled.

Figure 15: Internal Illustration of Resources and Jobs

The second subfigure (in Figure 15) shows that GS_4 changes its resources (i.e. broken channel) but none of the queued jobs (that previously matched GS_4's resources) need to be rescheduled because they can still use the other channel via GS_3 resources. In third subfigure (in Figure 15), GS_3 advertises the same resources as current GS_4 resources (i.e. one output channel) causing all of the queued jobs in GS_1 to be rescheduled. For example in this method:

- A child transfers all of its resources (i.e. ftp a file) to its parent when that child's resources change.
- A child also has to inform its parent of the number of resources sets it advertises.
- A job points (e.g. C++ pointer) to its primary matched resources and to one or more *backup* resources. For example, a job may match GS4's resources and both GS3's resources in the first subfigure of Figure 15. Thus, one of these resources can be the primary one and one or more can be backup resources. Note that a job may have backup resources on every scheduler on its path. Suppose that a job comes back to a scheduler because of resources disappearance, the scheduler can then make a quick decision (using its jobs status tables (JSTs)) on whether the returned job is still executable on any of its children. Of course, if a job returned to a scheduler doesn't have a backup on that scheduler, the scheduler then sends it back to its parent without spending the time of trying to discover resources for it on any of its children's partitions.

Of course, the above points are design choices that can vary from one strategy to another.

Now, the layering-resources method that we just described in this section structures resources in a hierarchal fashion. As a result, each GS in the Grid tree has to collect resources and match jobs to those resources. In attempting to *implement* this method, we learned that it was non-trivial to implement due to several reasons, such as:

- Resources sets from different sites do not exactly match each other easily in order to share one channel.
- Every time a resource set is changed, schedulers perform too much work to rearrange resources sets and pending jobs on those resources, as shown in Figure 15.
- Resource discovery is performed for jobs at every level of the hierarchy.
- It is difficult to detect new appeared resources.

The above points mainly led us to come up with the self-discovery method which is discussed in the next section.

3.2.1.2 Second Method: Self Discovery

The Grid system, in the *self-discovery* method, *omits* irrelevant dissimilarities between resources of different sites. The principle behind this method is resources are equivalent to each other if they match the same set of jobs. In other words, if the same jobs set is executable on different resources, the Grid system will then consider those different resources as alike, since their differences do not affect the jobs computability at that time. For example, one site advertises INTEL architectures and another site

advertises AMD architectures. Now, suppose the Grid has ten jobs that they can be executed on either INTEL or AMD platforms. In this case, the Grid system considers the INTEL architecture as *equivalent* to the AMD architecture for those ten jobs since they can be executed on either platform. However, suppose another set of jobs only requires AMD architecture to execute. The Grid system, in this case, considers the INTEL architecture as *nonequivalent* to the AMD for the latter set of jobs.

Leaf Grid schedulers (LGSs), in the self-discovery method, are the only schedulers that collect information about local resources via GIS(s) or local schedulers as described in the previous section. In fact, having LGSs as the only Grid schedulers to collect and store information about local resources is a major benefit of this method since:

(1) Information will be more likely up-to-date (i.e. note that in our model, we assume that LGSs are combined in one unit with local schedulers, since it is usually the case [44]), and

(2) Scalability is improved in the overall system (i.e. information is distributed across the Grid).

LGSs start resource discovery by sending the Request for Job Matching (RFJM) message to their parents, which in turn forward the RFJM message to the grandparents, and so on until the RFJM message is received by the GSS, enabling it to initiate the resource discovery stage to all of its raw jobs (i.e. jobs that have not been through resources discovery). However, if the GSS has no raw jobs, it will then backlog the RFJM message until receiving new jobs. Now, the GSS starts the resource discovery stage by:

- Broadcasting a special message to all of its children to destroy all channels in the system,
- Passing all raw jobs to all of its children as one block. The children in turn pass the raw jobs as one block to the grandchildren, and so on until they reach the LGSs at the bottom of the Grid tree.

Note that

- LGSs save all requests that they receive from their parents regardless if they have matched or not, enabling LGSs to perform rematching, if needed, due to resources change, and
- Intermediate schedulers (GS) always pass one RFJM message to their parents on behalf of their children and suppress other RFJMs preventing the GSS from initiating any unnecessary resource discovery.
- An LGS sends an RFJM message in any of the following two cases: At starts up, or its *Bag* becomes empty (i.e. all of the jobs in its bag are completed and out of the system).
- A logical channel is created for every unique jobs bag.

When a scheduler receives a *job bag* from one of its children, it checks if it has similar bag (i.e. bag with the same jobs). Now, if the received bag is equivalent to another existing-channel's bag, the scheduler then:

- Creates a new branch for that channel and binds it with the child's channel (e.g. informs the child with the channel's port number).
- Re-calculates the channel processing power based on the new created branch.
- Updates parent, if any, with the new processing power of this channel.

However, if the received bag is unique, the scheduler then:

- Creates a new channel with a new port number (i.e. one branch) and binds it with the child's channel (e.g. informs the child with the channel's port number).
- Initializes the channel processing power based on the received info from child.
- Updates parent, if any, with a copy of the new bag and the channel's port number.

The following logic defines a scheduler steps (i.e. the above points) upon receiving a bag from a child. Note that, in our case, the channel's processing power is the number of CPUs residing under that channel, since we are assuming that our computational resources are parallel computers (please see chapter 5 for more details about the Grid simulation model).

```
If new bag = an existing channel bag then
  {
  Create new branch with child's received info
  Increment channel processing power by the new branch processing power
```

```
  Increment total processing power of scheduler by the new branch processing power
  Inform child with this channel number
  Update parent with the new processing power of this channel
  }
Else
  {
  Create new channel with new number
  Create new branch with child's received info
  Initialize the new channel processing power with the new branch processing power
  If this channel is the first created channel in this scheduler
    {
    Initialize the scheduler processing power with the new branch processing power
    }
  Else
    {
    Increment total processing power of scheduler by the new branch processing power
    }
  Inform child with this channel number
  If scheduler has parent
    {
    Send bag with other info such as channel's number and processing power
```

```
    }
  }
```

Note that, in our case, the GSS informs all LGSs in the Grid tree to delete a saved request (i.e. no need to perform rematching), once the job starts executing, since we do not address the execution stage in this thesis. However, in reality, the GSS should update LGSs when a job is completed, since a job may migrate during execution.

Consider, for example, the GSS, in Figure 16, initiates resource discovery phase (upon receiving one or more RFJM messages from children) for six jobs: J_1, J_2, J_3, J_4, J_5, and J_6. The GSS then passes these jobs as one block to all of its children; the GSS's children in turn forward those jobs again as one block to the grandchildren, and so on until those jobs reach the LGSs at the bottom of the Grid tree. Now, LGSs start matching those jobs once they arrive, if a job matches an LGS's resources; it inserts it in its *bag* (i.e. note that LGSs still keep matched/unmatched requests, enabling them to perform job rematching, if resources happen to change). Now suppose that all six jobs match all resources in GS_7, GS_8, and GS_9. But, only J_1, J_2, and J_3 match resources at GS_{10}. Therefore, as shown in Figure 16, at the end of the resource discovery stage for those six jobs, the system ends up with two channels: jobs J_4, J_5, and J_6 can use only one channel where J_1, J_2, and J_3 can use both channels.

Now, assume that jobs J_1 and J_2 are already queued at GS_{10} in Figure 16. Suppose further that resources at GS_{10} are changed (e.g. operating system upgrade). In this case there are four possibilities:

1. All matched jobs (J_1, J_2 and J_3) still match and all mismatched jobs (J_4, J_5 and J_6) still mismatch to GS_{10}'s resources (i.e. no change to GS_{10}'s bag). Thus, the resources change in this case is *irrelevant* and nothing is done about it.
2. Job J_3 (i.e. not queued at GS_4 yet) doesn't match anymore. In this case, GS_{10} builds a new *bag* for the matched jobs and forwards it to GS_4, which in turns create new channel with GS_{10} and forwards the bag to GS_1. Of course, there is no need for GS_{10} to reschedule any job because both J_1 and J_2 are still computable on its partition.
3. One or more of the already mismatched jobs (J_4, J_5 and J_6) match new resources. GS_{10} then reacts in similar way to the previous case. Of course, quick detection for appearing new relevant resources can be proven powerful in the Grid environment, since new appearing resource may relieve another overloaded similar resource, by migrating jobs to the new one.
4. One (or both) of jobs J_1 or J_2 (i.e. already queued at GS_4) doesn't match anymore. Now the job (J_1, J_2 or both) that lost its resources needs to be rescheduled and a new *bag* needs to be created as was described previously.

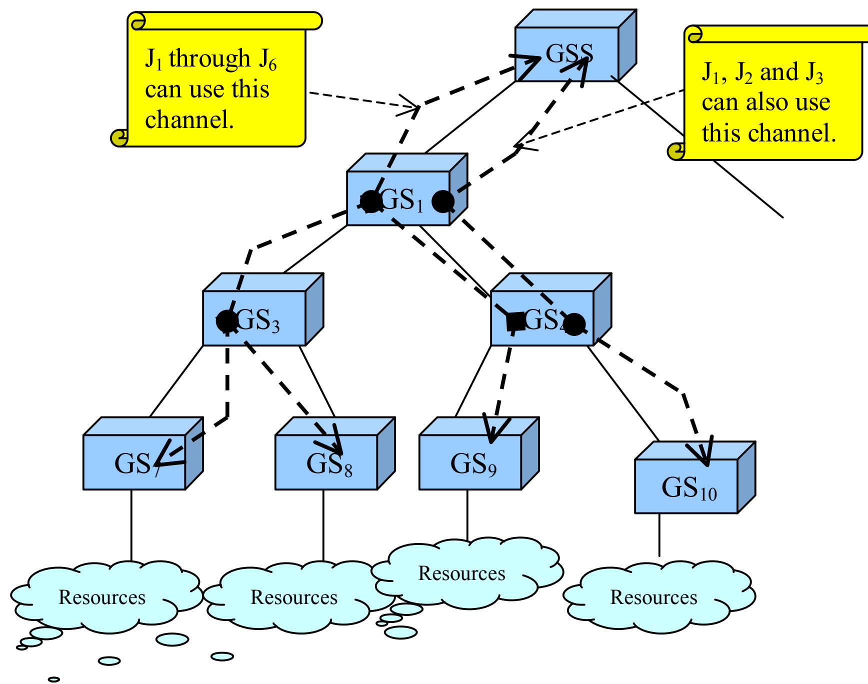


Figure 16: Logical Channels in the Self-discovery method

In our implementation, we store a *bag* in a stream of bytes. Suppose GS_{10} in Figure 16 successfully matches J_1, J_2 and J_3 to its local resources out of J_1 through J_6 jobs set. In this case, GS_{10}'s bag is stored as the following:

J_1	J_2	J_3	J_4	J_5	J_6
1	1	1	0	0	0

Now assume that J_1, J_2 and J_5 are completed. GS_{10} will then receive from its parent an update messages (that we call *holes*) to indicate that J_1, J_2 and J_5 are out of the system. Upon receiving those update messages, GS_{10}:

- Unsets the first two bytes (i.e. make them *holes*).

- Send an RFJM message to its parent, if matched jobs in its bag drop below the threshold value.
- Deletes J_1, J_2 and J_5 requests.
- Slides bag backward by making the first byte in the bag corresponds to J_3. At this point, the first six bytes in the GS_{10}'s bag will then be as the following (i.e. X corresponds to a *hole*):

J_3	J_4	X	J_6	X	X
1	0	0	0	0	0

Suppose further that the GSS has three jobs (J_7, J_8 and J_9) that have not been through resource discovery phase when it receives an RFJM message from one of its children. The GSS then initiates the resource discovery for those jobs by passing them to all of its children. Note that in our case, we chose to store all job requests at LGSs to enable them to perform re-matching when it has resources change and to prevent them from sending unnecessary RFJM messages. However, if LGSs do not store job requests locally, the GSS will then initiate resource discovery for all jobs from J_3 through J_9 without J_5 – design issue of space versus communications. In our case, the GSS only needs to transmit J_7, J_8 and J_9, since JMRs are also stored in LGSs. Now, assume that all of those jobs match resources at GS_{10}; the bag will then be as the following (of course, a bag shrinks or grows by bytes as necessary).

J_3	J_4	X	J_6	J_7	J_8	J_9
1	0	0	0	1	1	1

Each scheduler associates a bag, a port, a parent port and branches with each channel. Each channel's branch stores both the child's IP address and the child's port number. Parent and child schedulers can communicate with each other (via IP addresses) by indicating the parent/child port number.

Figure 17 shows an example of two channels between a parent and two of its children. Now suppose that the right channel in Figure 17 is broken because of a resource change. Suppose further that the LGS (of the broken channel) sends similar bag to the left channel's bag. In this case, the parent will fix the broken channel by connecting the broken right channel to the left one, as shown in Figure 18. However, the parent, that fixed the broken channel, still needs to send a message to its parent in order to cancel the broken channel. Otherwise, the grandparent will not realize that one of its channels is broken. If that parent is the only one uses the subject channel at the grandparent, then the grandparent deletes the channel. However, if the channel at the grandparent is used by more than one child, then the grandparent just removes that the parent reference to the subject channel, as shown in Figure 19. Of course, at all times, children have to update their parents of any changed information in a channel, particularly, the channel's processing power.

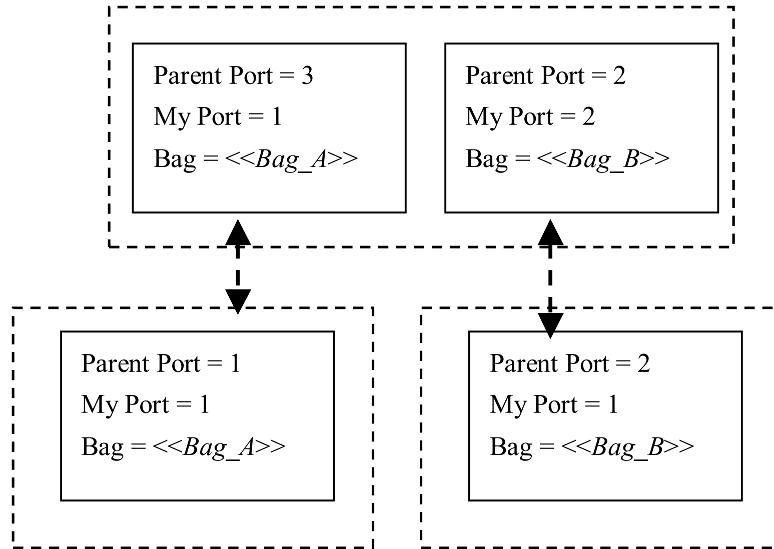


Figure 17: Example of Two Channels for a Parent and two of its Children

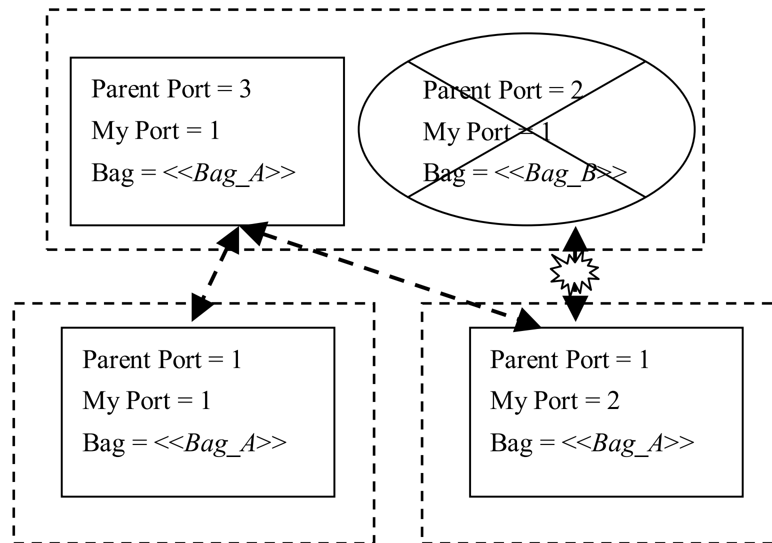


Figure 18: Example of Fixing Broken Channel by Scheduler from Figure 17

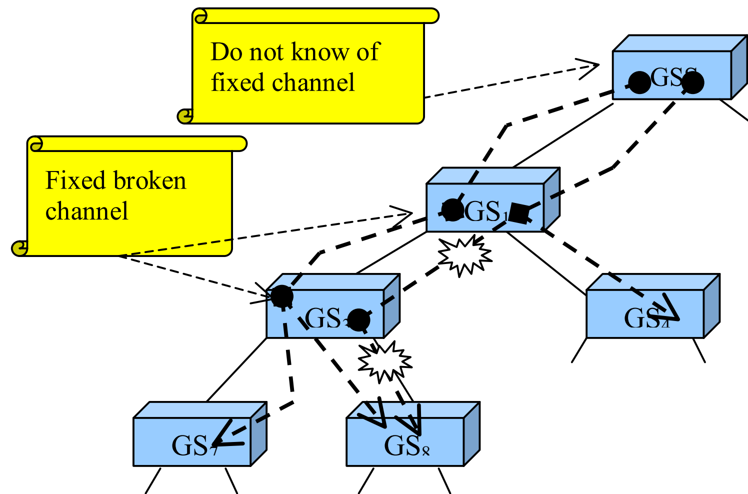


Figure 19: Example of Broken and Fixed Channel

3.2.2 Second Stage: Job Scheduling and Resources Selection

When a job description (JD) is submitted to the GSS, it is queued in a special queue until it goes through the resource discovery phase (i.e. in our case, the *self-discovery* method), as described in the previous section. The resource discovery phase constructs logical channels for all submitted jobs in order to get to physical resources that are able to execute them.

In this stage, a scheduler pushes jobs into a child's channel upon receiving a Request for Computation (RFC) message on that channel, hence, it is a self-scheduling method. Therefore, this stage starts when LGSs start transmitting RFCs to their parents

(i.e. note that RFC message, in this stage, works quite like the RFJM message in the previous stage).

In our case, we assume that LGSs are combined with local schedulers (LSs) in one unit. Thus an LGS sends an RFC message to its parent: (1) once it completes a job and (2) its bag still contains more matched jobs (i.e. jobs are removed from bags once they are completed). Note that the AHS method in this stage is based on the AHS method for parallel and cluster systems presented in [14].

In general, Grid scheduling, in this stage, uses *self-scheduling* by exploring the parent-child relationship. When a non-root GS wants some work to do, it initiates *self-scheduling* by sending a request for computation (RFC) message to its parent (via a channel) requesting computation from it. If the parent GS doesn't have computations that can be pushed on that child's channel at the time of receiving the RFC, it in turn generates its own RFC and sends it to its parent on the next level of the Grid tree. This process is recursively followed until the RFC reaches the Grid system scheduler (GSS) or a GS with computations is encountered along the path. Note that intermediate schedulers send one RFC message per channel to their parent, but still mark all channels' branches that have received RFCs on. For example, in Figure 16, assume that GS_3 receives two RFC messages from its children: One from GS_7's branch and the other from GS_8's branch. In this case GS_3 passes one RFC message to GS_1, if GS_3 has no jobs to pass to its children. However, if GS_4 receives two RFC messages from its children: One from GS_9 and the other from GS_{10}. In this case GS_4 passes (if it cannot satisfy either RFC) both RFCs to GS_4 since those RFCs have come from two different channels. In other words, schedulers always receive one RFC message from similar resources.

Schedulers use space-sharing policy to distribute computations among channels, upon receiving an RFC message on a channel from a child, as follows:

$$B_{share} = \lceil B_{rate} \times N \rceil$$

<u>Where</u>: B_{share} is branch share of all jobs within scheduler's subtree. B_{rate} is the branch transfer rate. N is the number of jobs within a scheduler subtree. B_{rate} is calculated as follows:

$$B_{rate} = \frac{B_{pwr}}{\sum_{i=1}^{M} (C_{pwr})_i}$$

<u>Where</u>: B_{pwr} is the branch processing power. C_{pwr} is a channel processing power (i.e. total processing power for all of its branches). M is the number of channels for a scheduler.

The processing power, in our case, is the number of CPUs that resides beneath a channel. For example if channel A with one branch is connected to a parallel computer with (305 nodes, 4 CPUs per node) and channel B with two branches is connected to two parallel computers with (64 nodes, 2 CPUs per node for each computer). In this case, channel-A processing power is 1220, and channel-B processing power is 256. Now suppose there 10 jobs and an RFC is received on one of channel-B's branches. The scheduler will then push one job into that branch from channel-B.

Now, once a scheduler determines the number of jobs that will be pushed into a channel's branch, it builds a list of those jobs as one block and pushes them into that

branch. Of course, a scheduler only pushes into a channel the jobs that are already inserted in that channel's bag (i.e. jobs that match resources below). Otherwise, an RFC message is forwarded on that channel to the parent. A scheduler performs the following steps to collect the jobs in order to be pushed into a channel's branch (i.e. note that a scheduler quits once the allowed number of jobs is collected):

1. Invokes the "butterfly" algorithm (discussed later in this section). This algorithm enables the scheduler to re-schedule jobs on an appeared better resource.
2. Collects jobs from the unassigned (i.e. unpushed) jobs.
3. Invokes the "load-balance" algorithm (discussed later in this section). This algorithm enables the scheduler to re-schedule jobs to balance load among resources.

Note that the AHS method for parallel and cluster systems [14] (outlined in chapter 2), only considers the unassigned computation when scheduling jobs into a child. It doesn't also build logical channels, since jobs do not go through resource discovery, as in the case of the Grid. The AHS method in [14] can be viewed as a tree with one channel (i.e. all jobs match all resources).

We expect, in reality, schedulers to collect more dynamic information to make better scheduling decisions. This information can be performance related (e.g. load) or economic related (e.g. prices), as described in chapter 2. The NWS [59] is an example of systems that gather performance related information, and the GRACE [7] is an example of systems that gather economic related information. Note that the difference between collected static information and dynamic collected information is that, on the one hand,

the static information must exist to enable a job to execute. On the other hand, dynamic information allows the selection of the best resource among a set of resources able to execute the target job. For example, two resources may be able to execute a job, based on the static information. However, one of those resources will be selected based on the dynamic information.

User-imposed constraints are also taken into account when selecting resources (i.e. channels). For example, a user may set a deadline, price range, etc. Those constants can be divided into two categories: soft and hard conditions. Soft conditions are the ones that user is hoping to have, but can continue without them until they become available (or if the ever become available). For instance, a customer, prefers to pay $100 per hour to use a resource, but is willing to pay $150 per hour for a while. On the other hand, hard conditions are the ones that user is not willing to give up. For example, a user requests that her job to be executed within one hour or forget about it.

Consider Figure 20 as an example, which shows a Grid tree with four hierarchy levels where all sites have equivalent processing power. Suppose the resource discovery phase has already produced the channels that are shown in Figure 20, and suppose further that each of the LGSs: GS_7, GS_8, GS_9, and GS_{10} transmits an RFC message to its parent. Now presume that the GSS has three jobs that can push into the two channels. It will then push one job into the first channel (satisfying RFC_1), and pushes two jobs into the second channel (satisfying RFC_2). Now, assume once the GS_1 receives the two jobs, it pushes both of them to GS_4's channel. In this case, GS_1 realizes that RFC_2 was not satisfied, since no jobs were pushed into GS_3's second channel, GS_1 will then reissue RFC_2 message on behalf of GS_3's second channel.

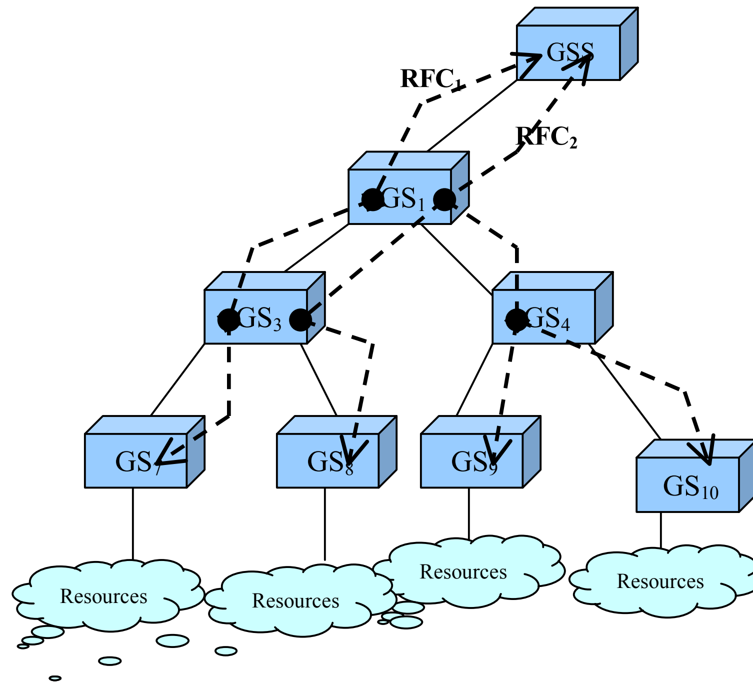


Figure 20: An Example of Mapping RFCs to Logical Channels

Now, the leaf GS is responsible for bringing the physical job into the system and will serve as a middleware (i.e. user agent) between the allocated resources and the user workstation. In this way, the user's application will run on the top of the allocated resources giving the impression of running the application on his/her workstation. For example, a start-up company cannot afford an expensive development tool and assume those tools are part of a larger firm's Grid resources. Therefore, this start-up company may rent those needed tools (say for two hours a day). By running the start-up company application directly on the top of the tools, the Start-up Company will then benefit of interacting directly with tools for two hours without interruption. Once it finishes using the allocated resources, it aborts the application. Reservation of resources is important

feature of Grid systems. Maui [24] Grid scheduling is an example that uses the reservation concept to maintain allocated resources.

GSs may also break a Grid job into sub-jobs in order to run it on multiple sites simultaneously. In our approach, a GS can theoretically distribute sub-jobs among its children's channels and then collect the results from them. Therefore, that GS can serve as an agent between resources and user's workstation. Breaking Grid jobs into sub-jobs is out of the scope of this thesis ([21] presents a proposed architecture to break Grid jobs into sub-jobs). For example, assume a job requires using four INTEL TFLOPS in order to complete within (say one hour). Suppose now that the Grid is able to discover these computers whereas each machine is located at different location. Thus, the Grid should be able to submit that job in parallel to those computers.

Now, as soon as the jobs reach leaf GS (LGS), the actual job is brought to the system by that LGS to be divided into sub jobs to share resources. Figure 21 shows an example of a Grid job sharing two clusters where the two clusters.

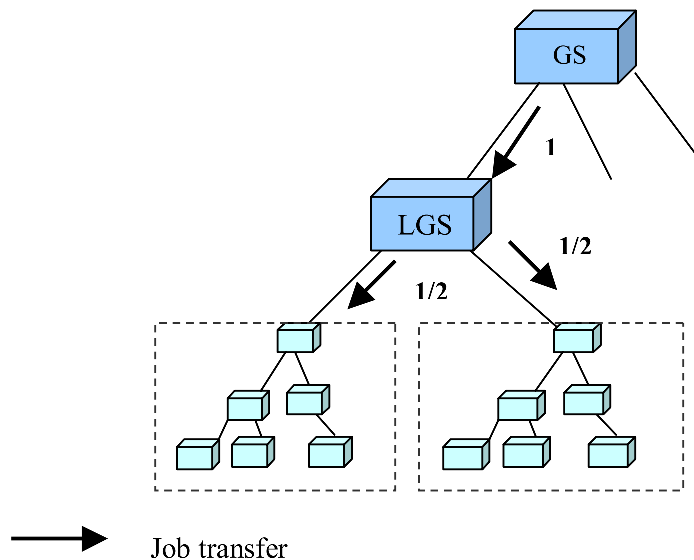


Figure 21: Space/Time Sharing in AHS

It becomes difficult, in the general scheme shown in Figure 6, to make a job running on multiple resources (e.g. say 2 clusters) together to achieve parallelism. Furthermore, we don't see how running a job in parallel can be achieved in the P2P systems, since forwarding requests is the only relationship between a peer and its neighbors.

One of the most attractive capabilities of the Grid systems is the ability to use the massive underutilized computing power. Thus, a Grid system should aim to use computing resources in parallel to get the full advantage of its power (i.e. simply, 10 computers can work faster and better than one computer can do). Such computing power is the driving force behind the Grid's popularity in the scientific communities such as the biomedical field, physics modeling, and many others. Nowadays, the approach in writing such huge applications is to be dividable into sub-jobs where each sub-job is executed on different resources (e.g. computer) in parallel without the need for these sub-jobs to communicate with each other [8]. Of course, executing one job on different machines requires an algorithm that can split that job into a number of independent sub-jobs, since contention occurs if those sub-jobs are not truly independent. For example, a sub-job needs another sub-job outputs before it can even start. Note that there are no current practical tools that can divide an arbitrary job to be run in parallel, and the art of automatic transformation into parallelism of an arbitrary job is in its infancy stage [8]. Therefore, we should realize that not all Grid jobs could currently be divided into sub-jobs to be run in parallel in the Grid systems. However, as stated earlier, new intensive

computation applications are currently being written to take the full advantage of Grid parallelism in the future [8].

3.2.3 Dynamic algorithms

Grid systems are extremely dynamic environments, hence, various unforeseen events can occur during scheduling stages that will force Grid systems to accordingly react in order to adapt to the new circumstances. As a result, we propose three dynamic algorithms to take some of these unexpected circumstances (during the resource selection stage) into account:

- *Butterfly Re-scheduling* algorithm to handle rescheduling jobs when better resources become available. Of course, a “better” resource is decided based on a predefined metric(s). In our case, we use the shorter geographical distance between the user and the selected resources as the preference metric.
- *Fallback Re-scheduling* algorithm to handle rescheduling jobs that their resources had taken away from the Grid before the actual resource allocation (since Grid scheduling is tentative until the actual allocation of the resources).
- *The Dynamic load balancing* algorithm to balance load among resources.

GSs store information about jobs that currently scheduled in their partitions in Jobs Status Tables (JST). Each entry in a JST table contains information a bout a job such as the Job description (JD) and its state. A job is always in one of the following states:

- *Raw*: Once a job is accepted by the GSS, it goes into the raw state until it goes through the resource discovery stage.
- *Tentative*: (i.e. tentative scheduling) a job may lose its resources, because their owner can take Grid's resources at any time. It is not expensive to reschedule a job in this state. In our case, we only perform jobs rescheduling in this state.
- *Ready*: Once a job reaches the LGS (i.e. the scheduler on the top of the needed resources), it goes into the ready state. At this state, the LGS connects to the user workstation and starts transferring the necessary data from the user's workstation to the Grid and serves as middleware (i.e. user agent) between resources and user's workstation.
- *Blocked*: When a job is preempted from using the allocated resources. For example, some jobs only run when resources are idle. In our case, we assume that jobs never get preempted. As stated earlier, the execution stage is out of the scope of this thesis.
- *Running*: When a job is actually using the allocated resources.

3.2.3.1 Butterfly rescheduling algorithm

Interacting directly with the resources can be helpful when, for example, a user rent a resource's access (e.g. ASIC tool) for few hours. Of course, in this case he wants a full uninterrupted access for the entire purchased period. Grids give companies the power to actually search for needed resources with the best conditions without, for instance, dealing with salesmen, signing business contracts, and so on. Suppose a company needs

to lease a verification ASIC tools to verify its new chip before moving the chip into the silicon stage. The company subsequently has to:

- *Discover* a supplier (maybe taking weeks) who is willing to grant access to the required resource,
- Convince management of the need of required resources (e.g. countless presentations),
- Deal with the supplier's salesmen (i.e. signing contracts), and if they are lucky, they can access the needed resource from their site.

With the advent of the Grids systems, the above company can search the Grid, *in the same way we search the Internet*, by submitting the JMR (i.e. the verification ASIC tools in this case) along with its conditions (say $300 per hour or less). Now, if the Grid system discovers the required resource of the customers (satisfying its conditions, then the company will have this resource "on the fly" via the Grid. However, if this company can't find the needed resources for $300 per hour (i.e. imposed conditions); it can keep increasing the price until it finds the needed resources for the best price in the Grid. Furthermore, it is quite possible to discover a resource constrained by user conditions (e.g. ASIC tools ≤ $300 per hour), but that is busy for the time being. Then, resolution depends on the user conditions: a user may decide to wait for the busy resource or use a more expensive resource and switch to other cheaper resources only when they become available.

The proposed *butterfly re-scheduling* algorithm is for rescheduling a job on the closest (i.e. geographically) resource with respect to the user location once it becomes

available if it had been busy when the user's job was scheduled. Interestingly, our proposed algorithm may keep jumping (like a *butterfly*) among resources until it settles on the closest resources. For example, a user in Ottawa Canada submits a job to the Grid. Suppose the Grid discovers three resources: in Ottawa, Toronto and India. Now, suppose the job gets scheduled on the resources in India because both resources in Ottawa and Toronto are busy. The Grid then migrates the job to the closer resource once it becomes available (say in Toronto), and then eventually to Ottawa, if resources in Ottawa become available.

The principle behind this scheme is to reschedule an already-scheduled job to "better" resources (with respect to predefined metrics) when they become available because those preferred resources had been busy when the job arrived to the Grid. Therefore, this algorithm can be extended to any *soft conditions* imposed by the user.

Our choice is to apply this algorithm to geographical distances, because executing jobs on nearby resources reduces communication costs that can be very expensive, particularly, in the Grids environment.

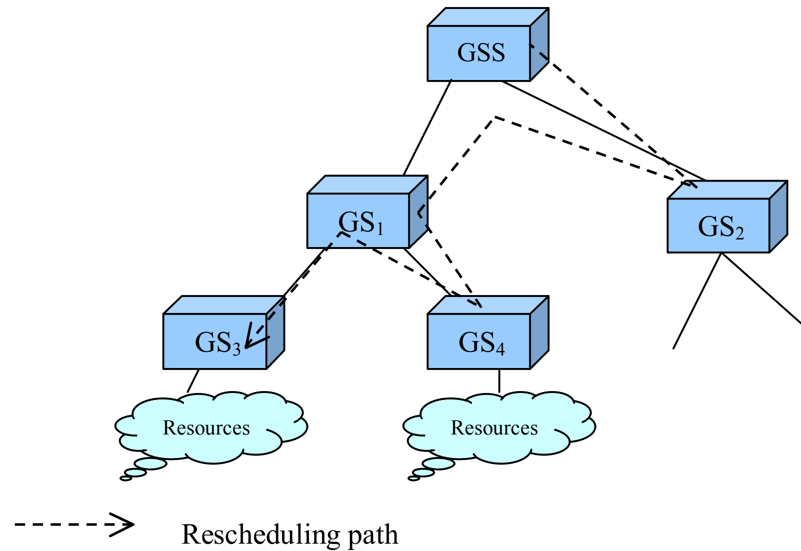


Figure 22: Illustration of the Butterfly Rescheduling Algorithm

The algorithm determines the closer resources to the user from the IP addresses. As will be discussed in the next chapter, the router prefix of an IP address is an indication of its geographical location. For example, suppose a job is submitted to the Grid from workstation with IP address of `132.4.4.5` as shown in Figure 22. Now, assume that IP addresses of GS_1, GS_2, GS_3 and GS_4 are `132.4.1.1`, `133.6.6.7`, `132.4.4.6` and `132.4.5.5` respectively. The GSS realizes that GS_1 is closer to the user than GS_2, since both IP addresses of GS_1 and the user's workstation have the same router prefix (`132.4`) of their IP addresses. Suppose the GSS pushes the job to GS_2 because GS_1 is busy at this time. In this case, the GSS marks this job as a "butterfly" job (i.e. scheduled to use unpreferred resources). Assume GS_1 sends an RFC to the GSS after that specific job is pushed to GS_2, causing the GSS to send a cancellation message to the GS_2 for that job (i.e. maybe also send another replacement job to GS_2) and reschedule the subject job on GS_1. GS_1 recognizes that GS_3 is closer to the user because they share the same router prefix (`132.4.4`). Therefore, if the job is scheduled on GS_4 because of GS_3

is being busy; the job is rescheduled on GS_3 (if it becomes available). Of course, not always a scheduler can tell the closer child to the user based on IP address. For example, if the user's workstation IP address was `135.4.4.5,` it becomes difficult to the GSS to tell which one is closer, GS_1 or GS_2 – probably, in this case, it doesn't matter that much because the user's workstation is too far from the resources anyway.

3.2.3.2 Fallback rescheduling algorithm

The fallback algorithm is intended to reschedule jobs that become incomputable because of losing their required resources. When an LGS detects resources change, it performs rematching for all of its saved requests (i.e. exist in Grid tree). Now, if an LGS ends up with the same jobs bag, this resource change is then irrelevant. However, if it produces a different bag, it will then pass it to its parent. Schedulers handle received bags in this stage as previously illustrated in the resource discovery stage. In addition, Schedulers mainly have to carry out the following (i.e. note that schedulers always inform parents of all of these activities):

- Remove any jobs that become incomputable,
- Update other channels processing power, if needed, and
- Delete any broken channels.

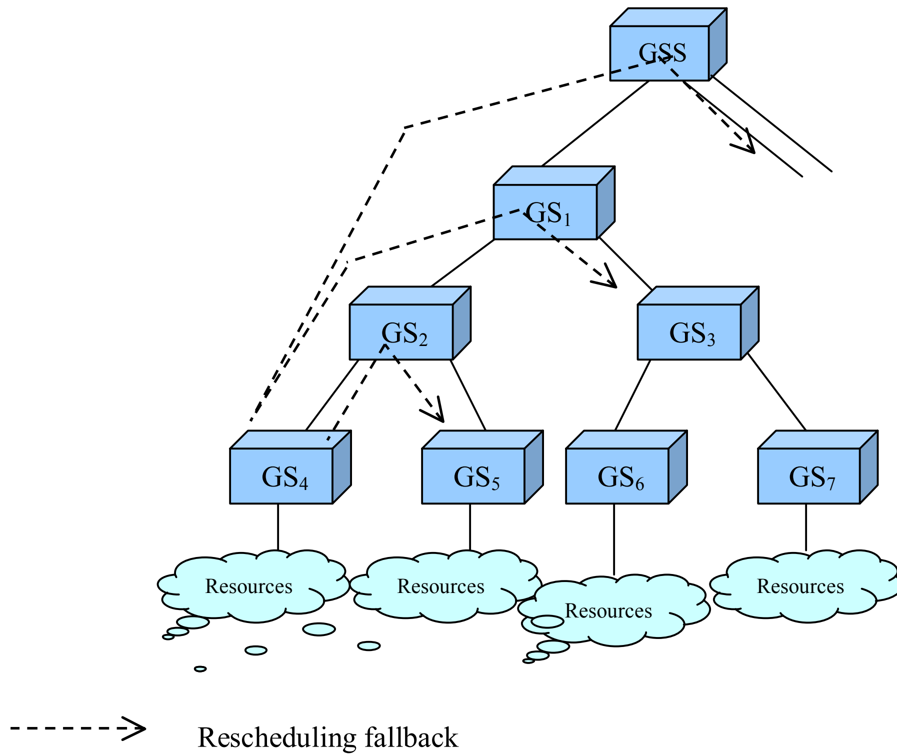


Figure 23: Rescheduling Fallback Algorithm

Suppose, for instance, a job at GS_4 in Figure 23 becomes incomputable because its required resources just disappeared from GS_4's partition. In this case, GS_4 cancels the incomputable job and informs GS_2 (i.e. its parent). GS_2 then checks if this returned job is still computable on GS_5, if yes then this job is still computable on GS_2. Otherwise, GS_2 cancels this job and informs GS_1. This process can go on until the cancelled job reaches the GSS. If the GSS cannot reschedule the subject job on a child's partition, the cancelled job thus becomes incomputable on the entire Grid and user must be notified in this case.

3.2.3.3 Load-Balance Algorithm

As stated earlier, when a scheduler receives an RFC message on a child's channel, it determines the number of jobs to be pushed into that child's channel by considering all jobs within its subtree. Once the number of jobs is determined, schedulers collect those jobs via the following steps:

1. Gathering jobs from already assigned jobs that are preferred to be rescheduled on the new available resources (i.e. butterfly algorithm),
2. Gathering jobs from unassigned jobs (that have not been pushed to children's channels yet), and
3. Gathering jobs from already assigned jobs to balance load among children's channels.

Schedulers are always responsible for balancing all assigned jobs among their children's channels within their subtrees, since a parent may cause a child's subtree to be imbalanced. Of course, parents do not know how assigned jobs are distributed within their children's subtrees. Consider, for example, the jobs distribution in Figure 24. Suppose GS_1 receives an RFC message from GS_4 and determines that 5 jobs can be pushed into GS_4's channel. Assume further that no jobs are preferred to be rescheduled on GS_4's partition. Now GS_1 has to reschedule 2 of the already assigned jobs in GS_3's partition. In this case, if GS_1 reschedules the two jobs that are waiting in GS_8, it would ruin up the balance of GS_3's subtree. Well, since GS_3 is the only scheduler is responsible

for balancing its subtree. GS_3 then balances its subtree once it receives an RFC message from one of its children.

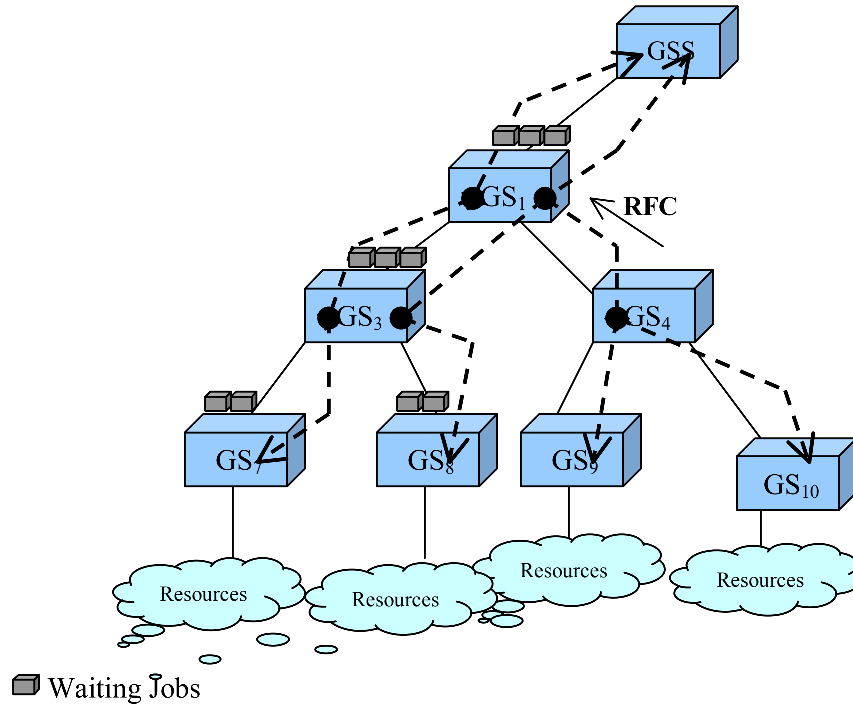


Figure 24: An Example of Load Balance among Channels

Note that schedulers may reschedule a job from a "better" resource in order to balance load among resources. In our approach, a scheduler marks that job as a "butterfly" job in order to be rescheduled later, if the opportunity presents itself. In other words, we prefer a job to execute as soon as possible rather than waiting for a "better" resource, since this "better" resource may not be available soon or simply disappear altogether.

3.3 HYBRID MODEL: HIERARCHAL AND P2P GRID SCHEDULING

The Hierarchal system (one Grid tree) has major drawbacks:

- All requests are submitted to the GSS, which may become overwhelmed with too many requests,
- It is difficult to bring too many organizations to agree on things such as constructing the Grid tree, controlling GSS policies, dealing with new joined organizations, and
- It is difficult to convince companies to replace their Peer-to-Peer (P2P) based Grid systems.

The Hybrid system, consisting of several Grid trees that act also as peers to each other (as shown in Figure 25), does not only overcome the above drawbacks, but also provides organizations more efficient ways to manage their own resources, such as stamping foreign requests with low priority, isolating their resources swiftly from the entire Grid without pumping out their pending requests of using their resources, etc.

Note that the P2P system can be viewed as a hybrid system where each Grid tree has only one scheduler, and the hierarchal system can be viewed as a hybrid system with one peer. In our case, we assume that (1) requests are always forwarded to neighbors as one block (i.e. a request dies when its hop count reaches 0), (2) foreign and home requests are queued in the same fashion.

The principle behind P2P systems, as described in chapter 1, is that once a user submits a *request* to a peer node, it in turns, passes the *request* to all or some of its neighbors if it cannot process the *request* locally. The peer node that chooses to process the received *request*, it contacts the *request*'s source directly or via the intermediate peer nodes. However, in our hybrid model, the whole *Grid tree* becomes a single peer system where a Grid's GSS is also an acting peer on behalf of its *Grid tree* with neighboring Grids. We will refer to this GSS as a PGSS (peer GSS).

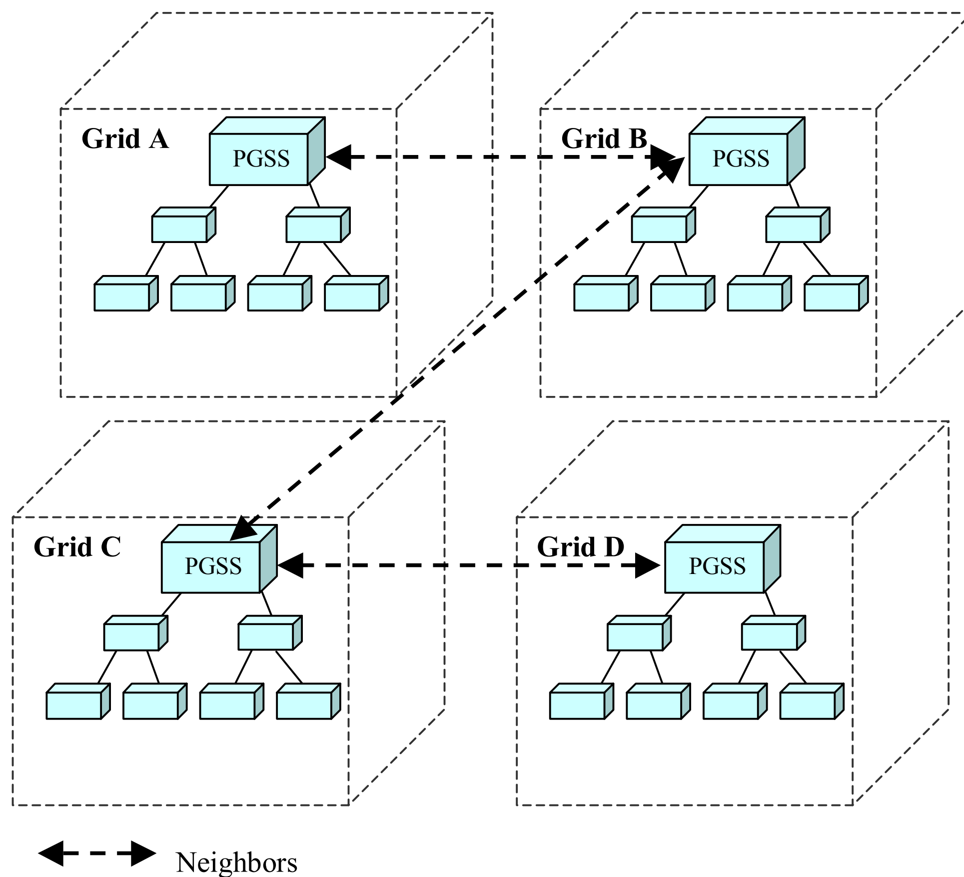


Figure 25: Hybrid Model – Hierarchical and P2P Grid Scheduling

Consider, as an example, the four Grid trees shown in Figure 25 where ($Grid_A$ and $Grid_B$), ($Grid_C$ and $Grid_B$) and ($Grid_D$ and $Grid_C$) are neighbors respectively. Suppose a user submits his job on $Grid_A$, the job will then be sent to the GSS of $Grid_A$. Now, if the GSS in $Grid_A$ processes the job locally, it will then go through the techniques that we've been discussing so far in this chapter. However, if the GSS in $Grid_A$ cannot handle the submitted request for any reason such as hefty load or lack of resources, it will then act as a peer on behalf of that submitted request. In this case, the GSS in $Grid_A$ sends the job to its neighbor in $Grid_B$. Once the request is received by the GSS in $Grid_B$, it may choose to process it locally or to forward it to $Grid_C$, which again may forward it to $Grid_D$.

Note that the following three policies are extended from the traditional P2P scheduling to the hybrid system:

First, support for multiple *requests* to be transferred to a PGSS together as a single assignment. For example, the sender PGSS groups a number of *requests* in single *file* and FTPs it to its neighboring PGSSs. A neighboring PGSS may accept some of those requests and then forwards that file to its neighbors.

Second, contrary to the traditional P2P scheduling, the *request* files should be forwarded by a PGSS at all times to its neighbors regardless whether some of those requests get accepted by a receiving PGSS or not. By doing so, jobs are allowed to be scheduled on multiple *Grid trees*, hence *Grid trees* serves as *backups* to each other. Thus, once a foreign job is accepted by a PGSS, it schedules it on its Grid without telling the source PGSS about it. The neighboring PGSS that accepts the job contacts the sources PGSS when the job is about to run. By doing so, we gain the following:

- Preventing the source PGSS from getting overwhelmed with many responses at the same time. Thus, since jobs most likely will be ready to run on different *Grid trees* at different times, the source PGSS doesn't get too many responses at the same time.
- Scheduling on Grids is *tentative* by nature; consequently a job request may die on a foreign *Grid tree*. For example, suppose that a PGSS accepts a job from its neighbor at certain time since its tree status allows it to do so. Assume now that after some time that PGSS decides to kill all foreign requests because, for example, it becomes busy with local requests within its Grid tree. In this case, a job may be scheduled and killed on a foreign *Grid tree* without promising the source PGSS to execute that job.
- The source PGSS may change its mind and decides to run jobs (that already have been transferred to neighbors) locally due to some dynamic changes on its tree. Of course, we expect PGSSs to not treat jobs differently. Thus, a PGSS should schedule requests on its tree regardless of whether they have been transferred to neighbors or not. In this case, if a source PGSS receives a response from other peers to execute a job, the source PGSS may simply choose to not respond, if that specific job is doing satisfactory progress on the local *Grid tree*. However, we should remember that when a source PGSS receives a response from another PGSS, it means that subject job is about to run. In this case, the source PGSS should be careful when it declines the remote PGSS's help.

- Grids can serve as *backups* to each other: a PGSS may transfer a number of jobs to neighboring peers in order to schedule those jobs on multiple *Grid trees* at the same time. In this case, a job is executed on the first *Grid tree* that is able to do so. For example, suppose that a PGSS pushes a request to one of its children and at the same time sends it to its neighborhood. If it receives a response from a PGSS, it knows that job is about to be executed on that remote PGSS. On the other hand, if it doesn't receive a response from any of its peers, the job is still progressing on its current Grid tree, eventually getting a chance to be executed.

In traditional P2P scheduling, a peer node doesn't forward a request to its neighbors if it accepts the request, since it is literally the one that will carry out the service. Thus, it makes no sense to forward the request since the actual request is being served. In our *hybrid system*, a receiving PGSS forwards a request (or requests) at all times, because the request is accepted by a Grid tree (and not by a node), hence an accepted job is only scheduled tentatively as in the case of any scheduled job on the Grid systems.

Of course, we have to stop *request* files at some point of being forwarded by PGSSs. In this case, many techniques from networking can be adopted to do so. For example, a time to live (TTL) value can be assigned to every *request* file. Now, every PGSS receives a *request* file, it decrements the TTL value by one until it becomes zero. Certainly, the source PGSS controls how far a request file travels away. For example, the source PGSS can set the TTL value to one to prevent it from going beyond its neighbors to avoid for example receiving too many responses.

The hybrid model combines advantages of both system types. Some of these advantages are:

- It reduces the number of contact services that are needed for peers' registrations, since a peer is in this case an entire system.
- It becomes quicker for a job to discover resources, since jumping among systems is not like jumping among nodes.
- It enables Grid trees to use other systems as *backup* to balance loads, since jobs can always migrate to other Grids, as described earlier.
- It takes the advantage of already existed peers to be configured as GSSs. For example, if a company decides to the use hierarchal scheduling scheme, it can then use one of its peers as a GSS.

CHAPTER 4: GRID MODEL BASIC COMPONENTS

4.1 INTRODUCTION

The Grid model presented in this chapter is described using two models: the communication and the node models.

This chapter first presents the basic components that make up the communication model, which in some respects resembles the Internet. In addition, it provides an overview of the node model, which is the implementation of the proposed Grid schemes discussed in chapter 3. Note that the Grid simulation model, along with its assumptions, will be presented in the next chapter.

Mathematical analysis or modeling is generally used to study systems or phenomena that are difficult to work with in the real world, such as oil spills in oceans, earthquakes, forest fires and so on. We chose to study our Grid schemes through modeling and simulation so that they can be observed in a controlled environment.

One of the challenges that we've faced is how to illustrate and analyze the proposed scheduling ideas of the previous chapter, since Grids are large systems with resources that span over multiple organizations and, as in our case, communicate with each other via the Internet.

From our viewpoint, we have to build a communication model that resembles the Internet where nodes of the model communicate with each other using their IP addresses. Of course, we don't claim that we are modeling the Internet. However, we believe that our communication model provides what is necessary to simulate the proposed Grid schemes at the same time as nodes communicate through the Internet. The Discrete Event Simulation (DEVS) CD++ is used to build our communication model. This chapter provides a brief overview about the CD++ tools. Please see [56] for more details about the DEVS family tools.

In our communication model, several nodes are connected to each other in Internet style and communicate with each other using their IP addresses. By having a communication model, it becomes possible to run a series of experiments for our proposed scheduling schemes. Consequently, all external factors fall under the control of our model.

Of course, once the communication model is built, we can then configure the Grid system into the desired experimental system (e.g. P2P system or Grid tree(s)). For example, we can configure nodes of IP addresses of `132.2.3.8` and `132.2.3.4` to be children of the node with IP address `132.2.3.10`. Note that a node in the communication model is simply a computer with an IP address.

The Node model is the implementation of the Grid schemes as were presented in the previous chapter. The Node model differentiates between nodes as workstations, Grid System Scheduler (GSSs), Grid Schedulers (GSs), Local Schedulers (LSs) and Leaf Grid Schedulers (LGSs). The node recognizes its type from the way it is configured. For example, if a node discovers itself that it doesn't have a parent but does have children, it then knows that it should behave as the GSS node.

4.2 DEVS CD++ TOOLS

There are two types of models in the DEVS CD++ tools: *Atomic* models and *structured* (or *coupled*) models. Each model communicates with other models via input and output ports.

Structured models are a construction of a number of connected (i.e. via ports) atomic models and/or other structured models. Structured models are written in the CD++ script language and must define the following:

- The sub-components that make up the structured model,
- The input ports to enable the model to receive data from external models,
- The output ports to enable the model to send data to external models,
- The internal connections (IC) that define connections among internal components,
- The external input connections (EIC) that define the model's input-ports connections from external models,

- The external output connections (EOC) that define the model's output-ports connections to external models.

Atomic (i.e. primitive) models are simply a component that cannot be broken into elements. They are written as C++ classes and are compiled with the tools code. Programmers write the atomic model code according to *formal specifications* in the following wrapper functions:

- Function `initFunction`: It is used to initialize the atomic model.
- Function `externalFunction`: It is used to process external events when they are received on one of the model's input ports.
- Function `internalFunction`: It is used to process an internal event in the atomic model (e.g. timer expiration).
- Function `outputFunction`: It is used to send an event (e.g. message) on one of its output ports. This function is usually triggered based on an internal event. For example an output is scheduled after a certain time.

In summary, the atomic model must define the following:

- The input ports, if any, to enable the model to receive data from other models,
- The output ports, if any, to enable the model to send data to other models,
- The model States. CD++ tools keep every model in two states: Active and passive.

- The internal function (IF) that defines the model behavior according to internal events,
- The external function (EF) that defines the model behavior according to external events (i.e. receiving data on an input port),
- The output function (OF) that defines the allowed states in which to send a message on an output port,
- The time advance (TA) that defines an event that should be triggered after time is advanced. Although, we sometimes refer to the TA as delay, it is different from the conventional meaning of “delay”. In other words, the model doesn’t sleep when it is delayed, but it actually schedules an event after some time. For example, components A, B and C schedule events for in one hour, one minute and one second respectively. The CD++, in this case, advances time to one second from now to trigger component C’s event. Afterward, it advances time 59 seconds in order to trigger component B’s event, and then advances time 59 minutes to trigger component A’s event.

4.3 COMMUNICATION MODEL COMPONENTS

The Internet, not surprisingly, is a hierarchal system of nodes, networks, nets and backbones. An IP (Internet Protocol) [46, 47] address is a distinctive identifier for a node or host connection on a network. An IPv4 address is a 32 bit binary number usually represented as 4 decimal values, each representing 8 bits, in the range of 0 to 255 separated by decimal points as shown in the following example.

Example (see also Figure 26):

```
IP address = 132.179.220.200

132      .179      .220      .200

10000100.10110011.11011100.11001000
```

Every IP address is made of two parts, one identifying the routing prefix (i.e. the network address) and one identifying the host. The subnet mask decides which bits belong to the network address and which belong to the host address. Now when a packet arrives at a router, it checks if it contains an entry for that specific routing prefix (i.e. router address) in its routing table. If so, it forwards the packet to the found router. Otherwise, it forwards the packet to its default router, which is usually a higher level router in the hierarchy (i.e. a country router serves as a default router for a province router). It is possible, for a packet to reach the highest routers level in the Internet (e.g. an international backbone router). Once a packet reaches a local network router (i.e. Ethernet), it is then broadcasted to every node on the network. Now, it is obvious that the number of the levels in the routers hierarchy depends on the number of bits used for the routing prefix (i.e. subnetting masks).

In our case, we assume that the least 8 bits of an IP address represent the node, and the other 24 bits represent the routing prefix in which each group of 8 bits represent a level in the routing hierarchy, as shown in Figure 26.

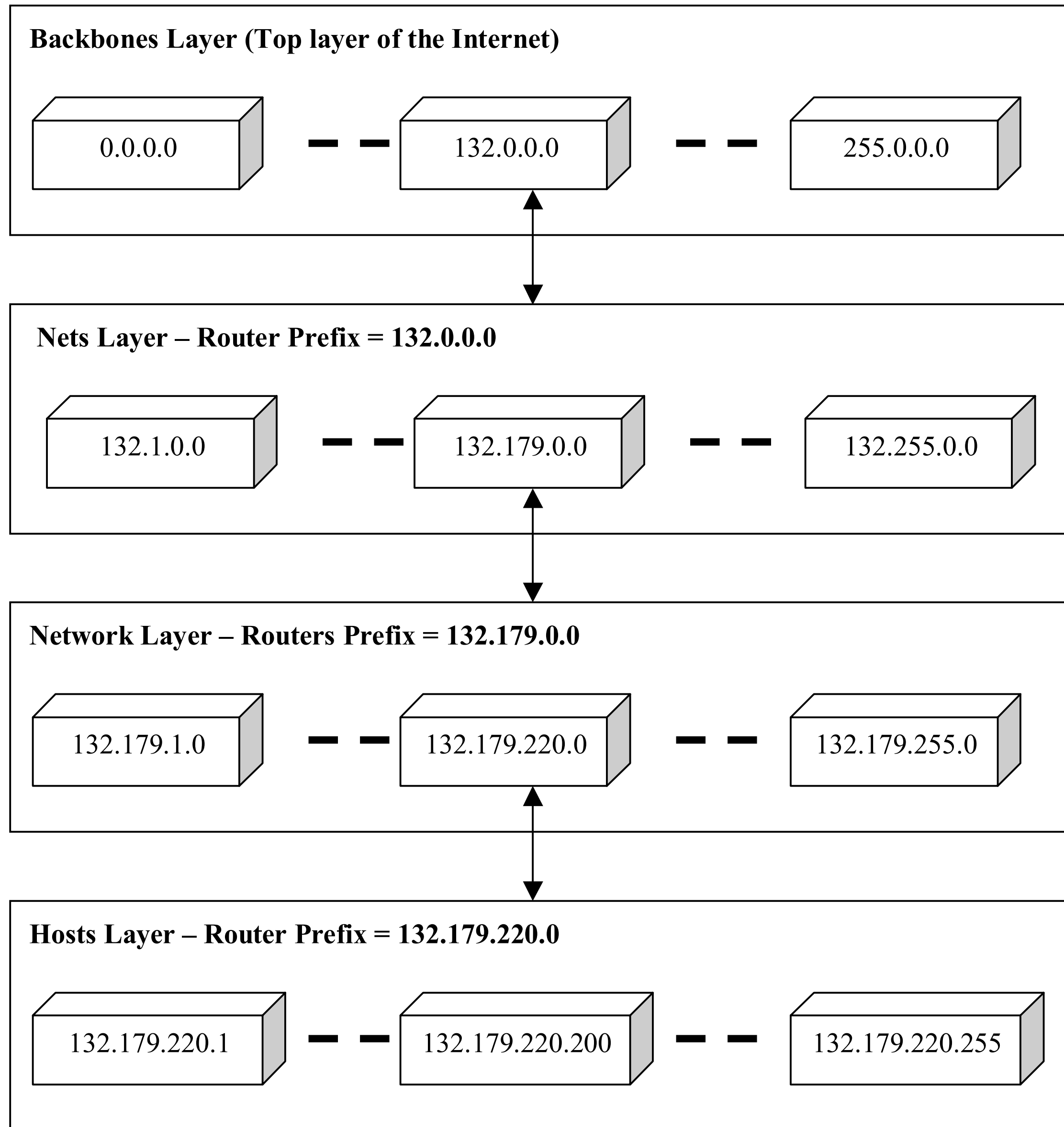


Figure 26: IP Addresses Hierarchal in the Internet Model

The Communication model defines four atomic models: Router, cable, node and timer. It also includes the following four structured models (in which all components are connected via a *Cable* model with configured distance and transmission rate):

- The *Top* model, which is made up of a number of *backbone* models.

- The *backbone* model, which is made up of a number of *Net* models and the *backbone router*.
- The *Net* model, which is made of a number of *Network* models and the *Net router*.
- The *Network* model, which is made of a number of *nodes* and the *Network router*.

4.3.1 Communication Atomic models

4.3.1.1 Cable Model

The purpose of this component is to account for the time to transfer data and for routers processing time, since it is difficult to determine the number of routers a packet goes through in the Internet.

The cable component has two ports: one is used for input and the other for output. The cable component calculates the time that a specific data should take to be transferred through it. For example, data *A*, *B* and *C* could require one hour, one minute and one second to be transferred through a cable respectively. The cable, in this case, buffers data in ascending order (i.e. shortest at front) and schedules the first data in the buffer to be released according to its calculated time. In our example, data *C* will be at the front of the queue, and will be scheduled for release in one second from now. After data *C* is released, data *B* moves to the front of the queue and will be scheduled for release in 59

seconds, since it was simultaneously waiting for release with data *C*. Finally, data *A* will be released 59 minutes after the release of data *B*.

The Cable component uses distance (in Kilometers), transmission rate (in Mbit/s) and the TCP window size (in Kilobytes) in order to calculate the required time for a data to be transferred. Distance and transmission rates are configured by the user, but the TCP window size is set as constant in the code. The distance is used to calculate the propagation time (t_{prop}). The transmission rate is used to calculate the transmission time (t_{trans}). The TCP window size [46] is used to control the flow of data between two nodes.

Based on [45], for transmitting frames in the stop-and-wait flow control scheme, the cable component calculates the required time (rounded to the nearest milliseconds) to transfer data, as follows:

$$T = nT_W$$

where n is the number of transmitted frames (or TCP windows in our case) and T_w is the time to send one frame (or TCP window in our case). T is the total time to send the data through the cable. T_w is calculated as follows (ignoring acknowledgment time and processing time at both ends):

$$T_W = t_{prop} + t_{trans}$$

Where t_{prop} is the propagation time of the cable and t_{trans} is the time for the transmitter to send out all of the bits in a window. Both t_{prop} and t_{trans} are calculated as the following:

$$t_{prop} = \frac{d}{V}$$

Where d is the distance in kilometers and V is the speed of the light, 3 x 10^8 m/sec.

$$t_{trans} = \frac{L}{R}$$

where L is the length in bits of the frame (window, in our case) and R is the transmission rate of the cable.

Consider for example that a cable is connected in a wide-area network (WAN) with length of 1000 km and transmission rate of 100 Mbps. Thus, the propagation time is 1000 x 1000 / 3 x 10^8 = 3.33 ms. Now, assume the window size is set to 424 bits. As a result, the window transmission time is 424 / (100 x 10^6) = 4.24 µsec.

The cable components play major role in the communication model, since they control the time to transfer data from a node to another.

4.3.1.2 Router Model

The *router* component's purpose is to route packets to their destination (please see Figure 26). When a router receives a packet, it checks if the destination address of that packet matches its IP address. (Of course routers have to mask the necessary bits of an IP address to perform this checking). If *yes*, it routes that packet to its destination node or to a lower router that can forward it toward its final destination. However, if it doesn't know how to route that packet, it sends it to its default router (i.e. a higher level router).

The router component has two input ports and two output ports. One of the input ports is used to receive packets from higher level router and the other from lower level

router. In the same way, the two output ports consist of one to send packets to the default router and the other to forward packets to a lower router (or to destination nodes).

4.3.1.3 Timer Model

The Timer component is used by nodes to schedule events. For example, a node may schedule an event to be triggered after 5 minutes. Note that cables do not hold Timer's messages since Timers are usually in the same machine. However, in our case, we use one timer for all nodes for simplicity reasons. The Timer model is actually a special node with "`0.0.0.1`" IP address.

4.3.1.4 Node Model

The *node* model is any computer attached to the communication model and that can be reached by any other node via its IP address.

However, this model also contains the actual Grid schemes implementation. We shall revisit this model later in this chapter.

4.3.2 Communication Structured Models

4.3.2.1 Top Model (Backbones Layer)

Figure 27 shows an overview of the communication top model, which can contain up to 255 backbones. In summary, the top model is described as follows:

- The input ports of the backbone models are connected to the backbones router internal output port. This connection allows the backbones router to route packets to a lower router.
- The output ports of the backbone models are connected to the backbones router internal input port. This connection allows the backbones router to receive packets from lower router. In this case, the backbones router is the default router for that lower router.

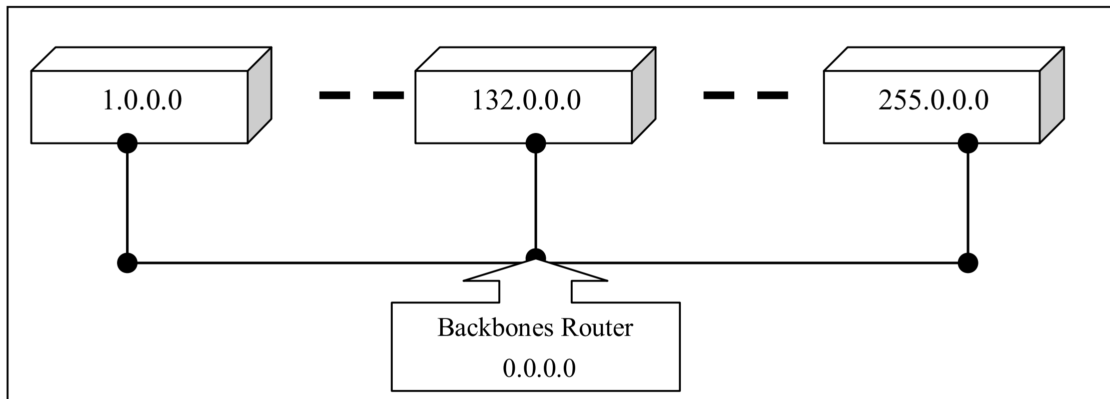


Figure 27: Top Model Overview

4.3.2.2 Backbone Model (Nets Layer)

Figure 28 shows an overview of the Backbone model, which can contain up to 255 Nets. The backbone model is described as follows:

- The input ports of the net models are connected to the net's router's internal output port. This connection allows the net's router to route packets to a lower router.
- The output ports of the net models are connected to the net's router internal input port. This connection allows the net's router to receive packets from lower router. In this case, the net's router is the default router for that lower router.
- The input port of the backbone model is connected to the net's router external input port. This connection allows the net's router to receive routed packets to it from a higher router.
- The output port of the backbone model is connected to the net's router external output port. This connection allows the net's router to send packets to the higher (default) router.

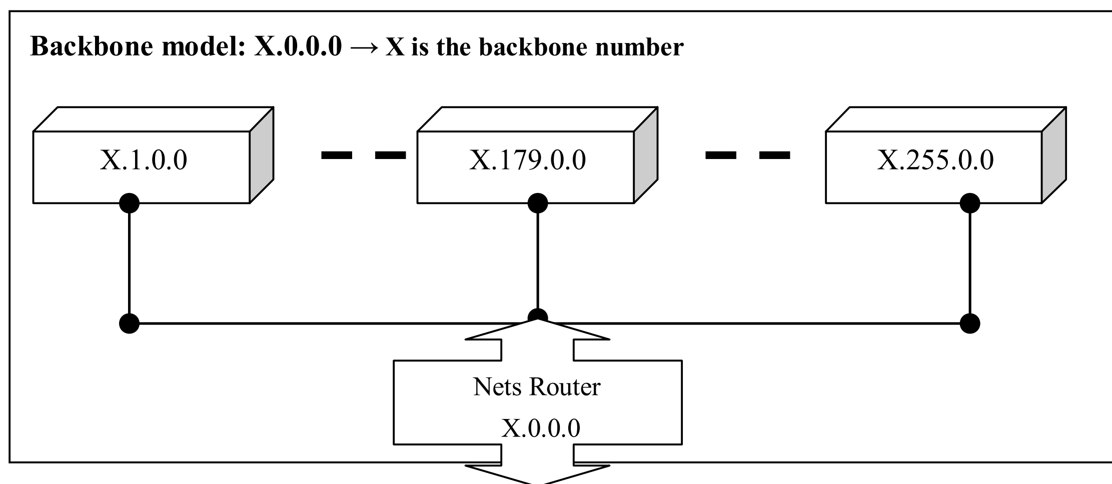


Figure 28: Backbone Model Overview

4.3.2.3 Net Model (Networks Layer)

Figure 29 shows an overview of the Net, which can contain up to 255 networks. The Net model is described as follows:

- The input ports of the network models are connected to the network's router internal output port. This connection allows the network's router to route packets to a lower router.
- The output ports of the network models are connected to the network's router internal input port. This connection allows the network's router to receive packets from lower router. In this case, the network's router is the default router for that lower router.
- The input port of the net model is connected to the network's router external input port. This connection allows the network's router to receive routed packets to it from higher router.
- The output port of the net model is connected to the network's router external output port. This connection allows the network's router to send packets to the higher (default) router.

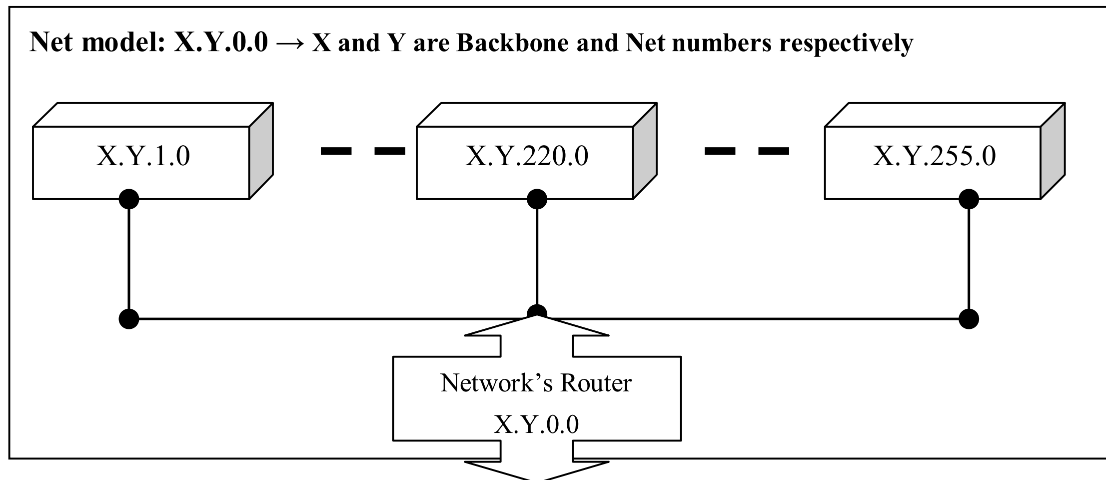


Figure 29: Net Model Overview

4.3.2.4 Network Model (Hosts Layer)

Figure 30 shows an overview of the Network model, which can contain up to 255 nodes. The network model is described as follows.

- The input ports of the nodes are connected to the nodes' router's internal output port. This connection allows the nodes' router to send packets to nodes.
- The output ports of the nodes are connected to the nodes' router internal input port. This connection allows the nodes' router to forward packets on nodes behalf.
- The input port of the network model is connected to the nodes' router external input port. This connection allows the nodes' router to receive routed packets to it from higher router.

- The output port of the network model is connected to the nodes' router external output port. This connection allows the nodes' router to send packets to the higher (default) router.

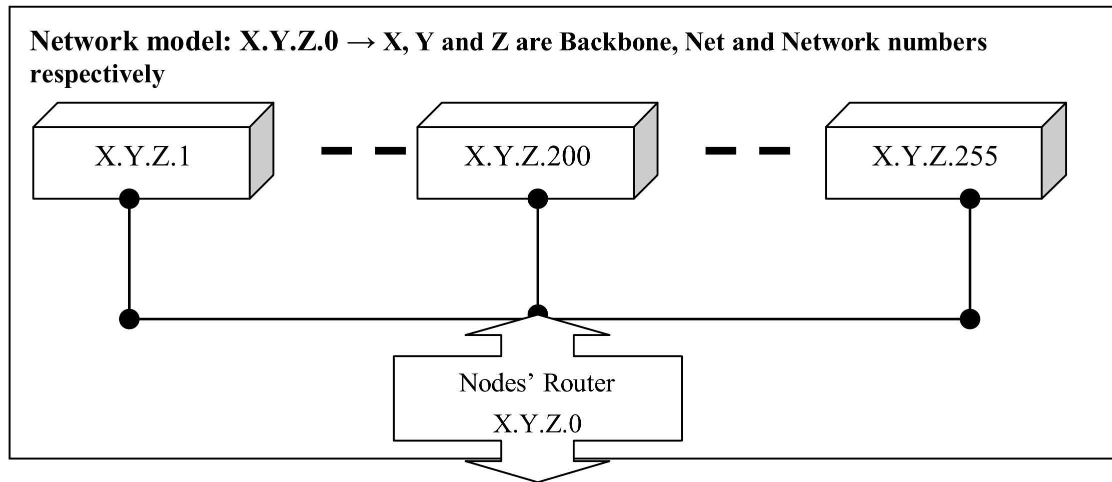


Figure 30: Network Model Overview

4.4 NODE MODEL DISCUSSIONS

All of the Grid model implementation is in the "Node" component since a node can be configured to be any Grid component, such as system scheduler, local scheduler or workstation.

The node component can be configured to operate as any of the following Grid node types:

- Workstation: This node is responsible for submitting request to the Grid in order to execute Grid job. Every workstation submits its request to the Grid via a "Grid entry" node. Peers, GSSs, and PGSSs nodes are used as Grid entries in P2P

systems, Hierarchal and Hybrid systems respectively. Peers, in case of P2P systems, broadcast their IP addresses to all nodes on their networks to enable them to submit requests to the Grid. GSSs or PGSSs broadcast their IP addresses to their trees, and once this information arrives at the bottom of a Grid tree, it is broadcasted to local networks. Therefore, a network must have at least one Grid scheduler to be allowed to have workstations, and thus to become part of the Grid system. In our implementation, we limit workstations to accept service from only one server. Thus, a workstation ignores any received service offer from another server, if it is already being served.

- Peer: This node is specific to P2P systems and it is assumed to be both local and act as a Grid scheduler. It also serves as a Grid entry to workstations on its network. Once a peer receives a JD (job description) request, it checks if it can meet that request from workstation with respect to some factors like deadlines. If yes, it contacts that workstation and times (100 milliseconds in our case) for the workstation response. Otherwise, it decrements the hop count in the workstation's request and forwards it to its neighbors.
- Grid system scheduler (GSS): This node is a Grid scheduler that has children, but no parent. It operates as already described in the previous chapter.
- Peer Grid system scheduler (PGSS): This node is a GSS node with neighbors. It is operates as described in the previous chapter.
- Grid scheduler (GS): This node is a scheduler with a parent and children. It operates as described in the previous chapter.

- Local scheduler (LS)/ Leaf scheduler (LGS): This node is a scheduler with a parent but no children. It advertises local resources. Note that we assume that LS is a combination of both Grid and local schedulers. It operates as described in the previous chapter.

CHAPTER 5: GRID SIMULATION MODEL, EXPERIMENTS, RESULTS AND DISCUSSIONS

5.1 INTRODUCTION

The Grid simulation model is broken into three submodels: The Communication, Node and System models.

The communication model consists of 2400 nodes that are spanned over four backbones, four nets and ten networks (i.e. sites) where each network contains 15 nodes that can be used by the Grid. As stated in the previous chapter, the communication model depends on the DEVS CD++ simulator in order to simulate the behavior of the components and their communication. Note that a node in the communication model is simply a computer with an IP address.

The node model contains the implementation of the proposed schemes in chapter 3. It is independent of the CD++ tools (i.e. the node model is separated from the communication model). In other words, the Grid code can be moved to real machines that communicate via other communication means (e.g. TCP/IP). A node in the Node model can be configured to operate as peer, local scheduler (LS), Grid scheduler (GS),

leaf Grid scheduler (LGS), Grid system scheduler (GSS), peer GSS (PGSS) or workstation. The node configuration determines the system model type.

The system model is the configured system type under experiment. The three possible types: Peer-to-Peer (P2P), Hierarchal (one Grid tree), and Hybrid (several Grid trees that acting as peers to each other) systems. As a result of our research, we chose to compare our proposed hierarchal system against the P2P system, since the P2P system, to our knowledge, is currently the only system in the Grid computing area being studied and proposed by many researchers [4, 10, 32, chapters 24-27 in [34], 44] to replace the centralized approach for Grid systems. The P2P system is a distributed system, which is originally used to transfer files, and which offers resource discovery and balances load among resources via its distributed nature. Thus, our choice to compare the proposed schemes against the P2P system is justifiable.

Note that we have observed that researchers usually compare their proposed schemes against a centralized system (perhaps because Grid computing is still a new area). But it is clear that the centralized approach is impractical in the Grid environment for many obvious reasons. Therefore, in our case, we did not run any of the workloads on the centralized systems, since it is unrealistic to have one scheduler to handle all requests.

In this chapter, we study three Grid system types (i.e. P2P, hierarchal with one grid tree and hybrid with multiple grid trees) over a number of different scenarios. In the *first three experiments*, we compare the performance of the P2P system to the Hierarchal (one Grid tree), Hybrid (4 Grid trees) and Hybrid (16 Grid trees) systems. In the *fourth experiment*, we study the degree of contribution of the proposed rescheduling algorithms

in the overall system performance. In the *fifth experiment*, we study the Fallback algorithm by repeating some cases of first and second experiments with the possibility of resources change during scheduling.

In the Grid simulation model, every local network (i.e. site) advertises 2 to 6 computational resources where those resources (i.e. servers) are parallel computers with different processing power and operating systems. The types of those resources are selected at random at runtime. The rest of the nodes in a site operate as workstations that submit requests with random requirements (i.e. in our case, operating systems) to the Grid. However, a workstation is constrained to submit one job at a time to the Grid and to accept service offer from only one resource. Note that the Grid model contains 520 computational resources (i.e. servers) and 1880 workstations (i.e. ratio 1:3.6). Therefore, no more than 1880 requests may exist simultaneously in the Grid, since each workstation can submit one request at a time.

We use three performance metrics to compare the three system types: *Total response time*, *average waiting time* and *average execution time*. *Total response time* is the time taken from submitting the first job request until the completion of the last submitted job. For example, a workload has 1000 jobs. Now, suppose that the first request was submitted to the Grid at 5:00 PM and the last job was completed at 10:00 PM, the total response time will then be 5 hours. *Average waiting time* is the time from submitting a job request to the Grid until the start of the actual job transferring to the selected resources. *Average execution time* is the time taken from submitting the actual job to the Grid until the job's output is received by the submitter workstation. These values are averaged over 20 different simulation runs.

5.2 COMMUNICATION MODEL

The communication model that is used by nodes to communicate with each other, consists of 2400 nodes that are spanned across four backbones. Each backbone (600 nodes) consists of four Nets, where each Net consists of 10 networks where every network consists of 15 nodes. Therefore, there are (4 backbones) (4 Nets) (10 networks) (15 nodes) which adds up to 2400 nodes in total in the communication model. Figure 32 shows an example of interconnections for two backbones, two nets and two networks (i.e. sites) in the model.

The communication model depends on the CD++ simulator (see chapter 4) to simulate all communication aspects among all nodes. Note that we've separated the communication model from the implementation schemes to be able to move the Grid code to real machines, if needed. In that case, Grid messages would simply be sent through real communication channels (e.g. TCP/IP) instead of the simulated communication model (since it depends on the CD++ tools). For example, the interface functions of the simulated communication model can wrap the necessary TCP interface functions, enabling the Grid messages to be transmitted via TCP/IP.

The communication model uses (in its fields) numbers 131 through 134 to designate a backbone, numbers 1 through 4 to designate a net, number 1 through 10 to designate a network and numbers 1 through 15 to designate a node, as shown below in Figure 31. For example, IP address 131.4.3.2 is valid, but 135.2.2.2 is invalid (i.e. node that doesn't exist in the model).

Network Numbers	Net Numbers	Network Numbers	Node Numbers
131 … 134	1 … 4	1 … 10	1 … 15

Figure 31: Valid IP Addresses in the Communication Model

Since the communication model represents the physical communication among nodes, it is static through out the simulation. In other words, our communication model will always consist of 2400 nodes. Note that the model can expand easily to any number of nodes we want. For example, we can add 600 nodes to the model by simply adding one more backbone, which is trivial to do. However, we repeat, here we used precisely 2400 nodes.

The communication model presents numerous challenges that we have to address to bring it as close as possible to emulating actual communication over the Internet (which is excessively hard since the Internet is very large unpredictable public network).

Some of challenges that we tried to address in the communication model are:

- Internet load: Our communication packets are not the only packets using the Internet backbones. Therefore, we should introduce some realistic load factor in order to account for external usage.
- Routers processing time: Internet routers process a multitude of packets (not limited to those relevant to our experiments). Furthermore, it is impossible to determine the number of routers a packet goes through in the Internet,

particularly, if we note that packets may use different paths when they travel between two points in the Internet. However, in our communication model, packets always use the same path between two points, as shown in Figure 33, which illustrates the path that a packet would take if it is transmitted from node 132.1.1.1 to node 131.1.1.1.

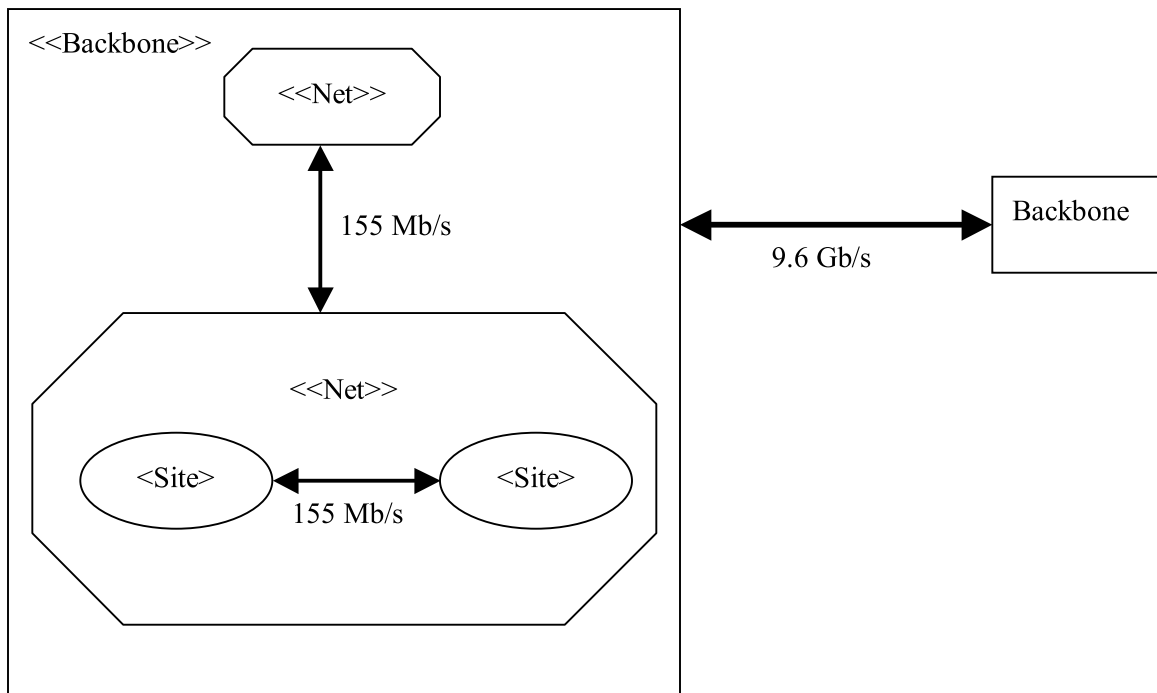


Figure 32: An Example of the Communication Model

The studies in [11, 12 and 35], which are based on actual statistics from Internet Service Providers (ISP), show that Internet links are extremely underutilized. The studies show that the traffic among backbones has an average utilization (in the two directions) in the range of 5.4% to 10.3%. The study in [35] also shows that utilization is typically in the range of 3-5% for traffic on a corporate T3 (45 Mb/s) link. However, we

discovered that introducing a 10% load (on the communication model capacity, to account for external traffic on the Internet) did not affect significantly the time to transfer data between any two nodes in the model. Yet, we've still assumed a 10% external load in the model.

The cable models, as discussed in the previous chapter, connect the communication model components together. Cables calculate the time that requires transferring data based on:

- Distance (configured by the user) is used to calculate the cable's propagation time.
- Transmission time (configured by the user) is used to calculate the time that takes the transmitter to send out all data bits.
- TCP window size to control data flow [46, 47], since TCP segments cannot be transmitted by the sender node until it receives the required acknowledgment for previously transmitted segments.
- Data size, which is specific to the transferred data.

The study in [39] has set the TCP window size to 25 megabytes instead of the typical current size of 64 kilobytes to transfer radiological imaging over large networks because of the assumption that future TCP window size is predicted to be one gigabyte. In our case, we chose to keep the TCP window size at 64 kilobytes. Of course, TCP window size influences the time required to transfer data. For example, transferring 9 gigabytes between two backbones, as shown in Figure 33, takes around 34 minutes and 15 seconds

if the TCP window size is set to 25 megabytes. However, it takes around 51 minutes and 24 seconds when the TCP window size is set to our choice of 64 kilobytes.

Components in the communication model are connected with cables. Based on studies in [3, 11, 12, 27, 35, 54], we use the following (see also Figure 32):

- Backbones are connected with each other with two cables. Each cable is 9.6 Gb/s, 1000 km (e.g. AboveNet OC-192 link). Thus, data travels 2000 km to travel from a backbone to another.
- Nets are connected with each other using two cables. Each cable is 155 Mb/s, 50 km (e.g. OC3 link). Thus, data travels 100 km from a net to another within the same backbone or from a net router to its backbone router.
- Networks are connected with each other using two cables. Each cable is 155 Mb/s, 50 km (e.g. OC3 link). Thus, data travels 100 km from a network to another within the same net or from a network router to its net router.
- Nodes are attached to a network with 100 Mbytes/sec, similar to the assumption in [1].

Most of the Grid dialogue messages are in the range of few bytes. Therefore, they usually take few milliseconds to be transmitted between two nodes in different backbones. For example, it takes an RFC message around 6 milliseconds to travel 2200 km from node 132.1.1.1 to node 131.1.1.1 in another backbone, as shown in Figure 33.

However, it takes 10-gigabytes of data 57 minutes, 12 seconds and 12 milliseconds to travel the similar path.

Now, Figure 33 shows an example of the path that data takes when it is transmitted from node 132.1.1.1 to node 131.1.1.1 in another backbone. In this example, the data travels through 7 routers and 6 cables (i.e. four cables of 50 km, 155Mb/sec, and two cables of 1000 km, 9.6 Gb/s). Thus data travels 2200 km in total, which is more than three times the distance between Toronto and New York City. Note that Table 1 shows examples of the required time to transfer different data sizes in the communication model from node 132.1.1.1 to node 131.1.1.1.

All of the messages in the communication model are embedded in IP packet. In our case, the IP packet header consists of the source IP address, the destination IP address and the data length.

Data Size (Gigabytes)	Time in (hours : minutes : seconds : milliseconds)
1	00:05:42:700
10	00:57:07:012
20	01:54:14:030
50	04:45:35:078
100	09:31:10:156

Table 1: Samples of Transferred Data Time for the Shown Path in Figure 33

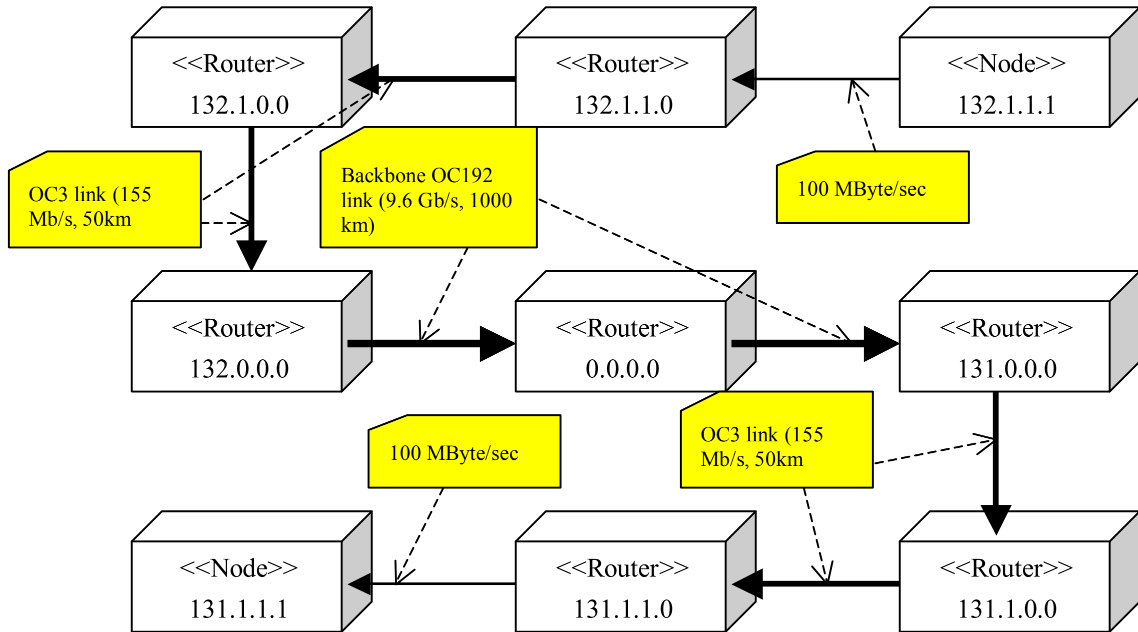


Figure 33: An Example of a Communication Path

5.3 NODE MODEL

A node, in the communication model, is simply a computer with an IP address. Yet, a node also contains the implementation of the proposed schemes for the Grid systems in this thesis. Recall that a node, in our implementation, can operate as a peer, or a local scheduler (LS), or a Grid scheduler (GS), or a leaf Grid scheduler (LGS), or a Grid system scheduler (GSS), or a peer GSS (PGSS) or a workstation. The node configuration determines the system type between peer-to-peer (P2P), hierarchal (i.e. one Grid tree) or hybrid (i.e. multiple Grid trees serve as peers to each other).

5.4 SYSTEM MODEL

We compare the performance of the 3 abovementioned types of systems according to a number of metrics, over different scenarios. Note that the peer-to-peer (P2P) system is a distributed system and the only system, to our knowledge, that is currently well thought-out by researchers [4, 10, 32, chapters 24-27 in [34], 44] to replace the centralized systems. Consequently, we repeat, it is a reasonable choice to be compared against the proposed hierarchal systems of this research.

5.4.1 Peer-to-Peer (P2P) System

Peers broadcast their IP addresses on their local networks, enabling workstations to discover them. Each workstation uses one peer from its site as its Grid entry (i.e. submits all of its requests to the Grid through that entry).

Once a job request is received at the Grid entry, it checks if it satisfies that job request's requirements. If it does, it then contacts the workstation to offer its service and times for the workstation response (fixed at 100 ms, in our case). Workstations may accept a peer's service by responding to it or may refuse a peer's service by simply not responding to it. If a peer cannot accept a workstation's request, it decrements the hop count in the JD message and forwards that message to its neighbors.

We arbitrarily set the hop count to 100, which is a large number, to allow job requests to go through all peers in the system at least once. We also configured three to six neighbors for each peer. Those neighbors are geographically close. However, this is not

the case in reality. Yet we made this choice to improve performance in the P2P system. Here, a peer accepts job requests under the following conditions:

- A request matches local resources (in our case, operating systems, as discussed later in this chapter), and
- Resources can complete the job within a deadline. (We assume the deadline is three times the estimated usage of resources by that job.)

Note that we've observed, in our simulation, that some jobs do not get the chance to run in the P2P systems. Therefore, in our model, if a workstation doesn't get a service offer within 2 minutes, it resubmits the job request. Note, also, that we assume in our model that peers and local schedulers are combined in one unit [44].

5.4.2 Hierarchal (one Grid tree) System

The Grid System Scheduler (GSS) broadcasts its IP address to the local networks via its children, enabling workstations to discover it, and use it as their Grid entry. Thus, all workstations submit their requests to the GSS.

As stated earlier, the Grid model is constructed of 4 backbones. Each backbone contains 4 Nets where each Net contains 10 sites. Of course, a site consists of a number of resources (i.e. servers) and workstations.

Now, the Grid tree is constructed by connecting the GSS to 4 children where each child holds one backbone. Each backbone-scheduler has 4 children where each child holds one Net. Each Net-scheduler has two children where each child holds 5 sites.

5.4.3 Hybrid (several Grid trees) System

The Peer Grid System Scheduler (PGSS) broadcasts its IP address to the local networks via its children, enabling workstations within its Grid tree to discover it, and use it as their Grid entry. Thus, all workstations within a Grid tree submit their requests to their tree PGSS.

We use two Hybrid systems in our simulation: 4 Grid trees (Hybrid-4T) and 16 Grid trees (Hybrid-16T). In the Hybrid-4T system, the Grid tree of the hierarchal system (see previous subsection) is broken into 4 Grid trees where each backbone has one Grid tree. In the Hybrid-16T (16 trees), each Net has one Grid tree.

5.5 GRID JOBS

Physical jobs are simulated in the way they are transferred and executed. The communication model simulates transferring data by calculating the required time that is needed to transmit a job via a link (see cable component discussion in the previous chapter). Also, a node component that is configured as a resource simulates the time that is needed to execute a job. This time is taken as the total of: a) the time it takes a job to be transferred to the allocated resource, b) the time it takes the job to use the resource and c) the time it takes the output to be transferred back to the workstation (see workloads discussion later in this chapter).

Of course, if we use real jobs, we then need to use real machines and replace the simulated communication model with real communication networks with hundreds or

thousands of machines, which is unreasonable. It is also impractical to construct a Grid with few machines in a laboratory, since Grid systems are constructed of various machines over large geographical areas.

Although our research does not consider the execution stage, we still have to simulate a job execution on a server (i.e. computational resource). And, we are still interested in where a job gets executed.

As stated earlier, the communication model simulates the required time to transfer data via a link. We arbitrarily assume a job produces output data five times the size of the original input job's size, based on [44]. Also, note that a server (i.e. a computational resource) in our simulation, does not execute more than one job at a time.

Gaussian distribution is regularly used to simulate the required time to run a job on a server (i.e. as in the case in [6, 17, 23, 25, 44, 50, 51, 52, 53, 54]) with respect to the input job size and server's processing power. Researchers in [44] assume that job sizes are correlated to the amount of work performed by each job, because their original collected traces do not record how much input data is used and how much output data is generated. They use Gaussian distribution where they set the input data size mean to μ `= b * CPUs * wall time in seconds`, where `b = 100`, `wall time` is the amount of wall clock run time, and `CPUs` is the number of CPUs in a parallel computer. (Similarly, we assume that our computational resources are parallel computers, as described in next section). However, in our case, we calculate the execution time based on the input data size and the number of CPUs in the local resources.

5.6 RESOURCES

We assume all resources are parallel machines made of a number of interconnected nodes, with a number of CPUS within each node, as in the case of most related work such as [17, 25, 44]. We chose the following parallel computers as our computational resources as shown in Table 2 (which is originally based on real machines as indicated in [17, 23, 25, 44]).

Server	Number of Nodes	CPUs per node
1	184	16
2	305	4
3	144	8
4	1024	4
5	64	2
6	512	4
7	128	2

Table 2: Simulated Computational Servers

We've duplicated those servers (i.e. computational resources) in every network (i.e. sites) in our model in the range of 2 to 6 servers per site. Note that the servers are picked up at random from Table 2. Servers also select operating systems at random (between UNIX, LINUX and Windows) for those computers. Of course, jobs cannot run on any machine unless local resources match their requirements (i.e. in our case the required operating system).

We also assume 0.1 local loads for all computational resources at all time as this is the typical case in most studies such as [51] and [23].

Note that selecting resources, jobs, etc. at random makes it unsuitable to compare system performance based on only one run. Therefore, we repeat, all of our collected results are based on the average of 20 runs. Also, we have noticed that results did not vary much from a run to another for the same workloads, perhaps because of the large number of resources, workstations, sites etc. in the simulated model. Finally, recall that the simulation model consists of 520 servers versus 1880 workstations – a ratio of 1:3.6.

5.7 WORKLOADS

Unfortunately it was difficult to find real traces for the Grid computing, since it is still a new area. However we were able to base our workloads (listed in Table 3 below) on real traces for parallel machines from [17, 23, 25, 44, 53, 57] and Grid simulation for physics and engineering problems from [3, 41].

We use a Poisson distribution to generate input data sizes for submitted jobs to the Grid where the Poisson mean is set to the desired average input size from Table 3 (that was originally based on real traces from [3, 17, 23, 25, 41, 44, 53, 57]). In this way, jobs are generated with different sizes, but with the desired average input size, which is the typical case in reality.

Workload	Number of Jobs	Average Input Size (in MB)	Servers to Jobs Ratio
1	10,000	28.9	1:19
2	10,000	312.7	1:19
3	10,000	1000	1:19
4	10,000	10,000	1:19
5	10,000	100,000	1:19
6	3000	28.9	1:6

Workload	Number of Jobs	Average Input Size (in MB)	Servers to Jobs Ratio
7	3000	312.7	1:6
8	3000	1000	1:6
9	3000	10,000	1:6
10	3000	100,000	1:6
11	1560	28.9	3:1
12	1560	312.7	3:1
13	1560	1000	3:1
14	1560	10,000	3:1
15	1560	100,000	3:1
16	1040	28.9	2:1
17	1040	312.7	2:1
18	1040	1000	2:1
19	1040	10,000	2:1
20	1040	100,000	2:1
21	520	28.9	1:1
22	520	312.7	1:1
23	520	1000	1:1
24	520	10,000	1:1
25	520	100,000	1:1

Table 3: Workloads Used in Simulation (Servers = 520, Workstations = 1880)

5.8 PERFORMANCE METRICS

We use three performance metrics to compare the systems under experiment: *Total response time*, *average waiting time* and *average execution time*.

The Total response time (TRT) is the time elapsed from submitting the first job request until the completion of all jobs in a workload. For example, suppose that the first request was submitted to the Grid at 5:00 PM and the last job of a workload was completed at

10:00 PM, the total response time will then be 5 hours. The purpose of this metric is to show the degree of *parallelism* in the Grid, since we view the Grid as a huge virtual machine. For that reason, the TRT metric is only measured in the experiments that have the workstations constantly submitting jobs to the Grid (i.e. one job after another). Clearly, the TRT metric is meaningless for experimental scenarios that assume people arrive at different stochastic rates to submit jobs to the Grid (e.g. experiment 2 in section 5.9), since all/some workstations are inactive (i.e. waiting for somebody to use them) for period(s) of times. Therefore, those waiting periods must be excluded in order to obtain the TRT correctly. However, one can argue that we can subtract the waiting time for each workstation by simply measuring the time from submitting the request until that workstation receives the output of the subject job. Afterwards, we simply could add all jobs response times to obtain the total response time. This is what we thought before we ran the experiments in section 5.9! But this forgets that Grid systems are parallel systems, hence, all activities are carried out in parallel (i.e. scheduling, workstations/resources idle times are happening simultaneously). By adding the response times for all jobs, we will obtain the total of all jobs waiting and execution times. (Other metrics are discussed next in this section). In other words, we will obtain the total time to complete all jobs *in sequence*, which defeats the main purpose of this metric, which is to measure the degree of *parallelism* in studied systems. Nevertheless, if we assume, for the sake of argument, that we can magically exclude all workstations parallel waiting times, we will then end up with the scenario that workstations submit their jobs as one job after another. The Total Response Time (TRT) is calculated as follows:

$$TRT = (LJC - FRS)$$

where: LJC (Last Job Completion Time) is the time recorded when receiving the output (i.e. at workstation) of the last completed job in the workload, and FRS (First Request Submission) is the time at which the first request by a workstation was transmitted.

The waiting time (WT) for a job is the duration from submitting the job's request to the Grid until the start of the actual job transferring to selected resources. For example, if a workstation submits a request to the Grid at 5:00 PM, and gets a service offer from a resource at 6:00 PM, the waiting time for that job will be 1 hour. The purpose of this metric is to measure the scheduling time it takes a job until a resource is allocated for it. Thus, this metric measures the time it takes a request to go through the first two stages of the grid scheduling stages (i.e. resource discovery and resources selection stages); those stages being the main focus of this thesis. The Average waiting time (AWT) is calculated as follows:

$$AWT = \frac{1}{N}\sum_{j=1}^{N}(SJTT - RT)_j$$

where: N (Jobs count) is the number of jobs in a workload, SJTT (Start Job Transferring Time) is the time when a workstation receives a service offer from a resource and starts transferring the physical job, and RT (Request Time) is the time when a workstation submits that job request to the Grid.

The execution time (ET) is the time from submitting the actual job to the Grid until the job's output is received at the submitter workstation. For example, assume a workstation receives a service offer from a resource at 5:00 PM. Next suppose that this workstation receives the output of the job at 6:00 PM. Then the execution time will be 1 hour. Although all systems, in our model, function the same way when a request is

mapped to a resource, we still need this metric to measure the location that a job was executed at. The average execution time (AET) is calculated as follows:

$$AET = \frac{1}{N}\sum_{j=1}^{N}(JCT - SJTT)_j$$

where: N (Jobs count) is the number of jobs in a workload, JCT (Job Completion Time) is the time when the job's output is received at the workstation, and SJTT (Start Job Transferring Time) is the time when a workstation receives a service offer from a resource and starts transferring the physical job.

Note that since this thesis is mainly concerned with the first two scheduling stages (i.e. the spent time in these two stages is basically the waiting time for that request), we choose to break the *average response tim*e (i.e. the time from submitting a request to the Grid until the workstation receives the output) into the two metrics of the average waiting and execution times.

5.9 SIMULATION EXPERIMENTS

In this section we study three types of system over a number of different experiments. The goals of those experiments are summarized below (please see experiments discussions in next subsections for more details):

- To study the performance of the proposed hierarchal scheduling schemes by comparing it to the well-known P2P systems over different scenarios (experiment 1, 2 and 3). The main differences between the first three experiments are summarized as follows:

- In experiment 1: All workstations keep continuously submitting jobs to the Grid. This scenario represents the case where a set of jobs are executed in one batch. Of course, this scenario is not very realistic, but it enables us to study systems performance in the worst case scenario. Indeed, it cannot get worse than having every workstation continuously pumping jobs into the system.
- In experiment 2: Each workstation behaves differently from other workstation. In this experiment, jobs are assumed to arrive at a different stochastic rate (i.e. Poisson distribution). Therefore, we don't know where requests come from, when requests arrive at the Grid, or when the next request is submitted from a workstation. This scenario is somehow more realistic. Therefore, it can be viewed as the average case scenario.
- In experiment 3: All workstations submit requests to the Grid at the same rate. In this experiment, we can study the impact of jobs arrival rate on the systems under experiment by stepping through different rates. Clearly, this scenario is unrealistic, but it can help us understand how the different system types react to different jobs arrival rate.

- To study the contribution of the rescheduling algorithms (i.e. butterfly and load-balance algorithms) in the overall the performance of the hierarchal scheduling scheme (experiment 4). We rerun some workloads with scenarios of experiments 1 and 2 while having our proposed algorithms disabled.
- To study the performance of the proposed fall-back rescheduling algorithm (experiment 5). We re-run some workloads with scenarios of experiments 1 and 2

scenarios, with the assumption that resources may change in a stochastic fashion. We also record the number of saved jobs due to losing resources.

Now, regardless of the configured system or experiment, the following assumptions apply:

1. The number of used resources is still the same. Thus, the computational power of the Grid is the same for all systems. In our case, 520 servers versus 1880 workstations.
2. A job is submitted by one workstation and executed by one server. Therefore, no more than 1880 requests may exist simultaneously in the Grid, since we have 1880 workstations at all times. Note that in the case of the P2P system the same request can be resubmitted more than once if the submitter workstation does not receive a service offer from a server within a specified time (i.e. in our case, 2 minutes).
3. A workstation submits a request to the Grid via its Grid entry. It picks a job size based on a Poisson distribution, a required operating system at random for that request and submits it to its Grid entry (i.e. the Grid scheduler where that workstation submits requests to the Grid through it).
4. Given the focus on resource discovery and selection:
   4.1. The execution stage is alike for all systems. In fact, once a job's request is submitted to a resource, it invokes the same code. In the execution stage, a resource connects with a workstation and downloads the simulated physical job in a similar way regardless of the system under assessment. However, we are still interested in the location of where jobs get executed.

4.2. Physical jobs are simulated in the way they are transferred and executed.

5. All results are obtained over averaging 20 different runs.

5.9.1 First Experiment: Continuous Job Submissions

The purpose of this experiment is to study the system behavior when jobs are submitted one after another. This scenario is possible when an organization, for instance, execute thousands of jobs in one batch as one job after another.

In this experiment, we can measure the degree of *parallelism* in the Grid from the time it takes the Grid to compute all jobs in a workload, since Grid systems are viewed as a giant virtual parallel machine.

Now, note the following assumptions:

- The number of workstations in a site determines the number of jobs that will be submitted from that site. For example, if site A has 3 workstations and site B has 6 workstations. Most likely, site B will submit twice the requests that are submitted from site A. Therefore, the workload under this experiment is already distributed among sites.
- All workstations are active simultaneously all the time. In other words, a workstation submits a request to execute a job and then submits another request to execute a second job immediately after it receives the first job output.
- All requests are submitted from all locations with the same probability. For example, job number 5 from a workload may come to the Grid from any site.

5.9.1.1 Results and Discussions

Note that the detailed result sets are presented in Table 4 in Appendix-A.

The average waiting time (AWT) showed a substantial improvement against the P2P system regardless of the number of used Grid trees or workload as shown in Figure 34, Figure 35, Figure 36, Figure 37 and Figure 38. Interestingly, the AWT starts declining slowly when the Hybrid system contains too many Grid trees. Perhaps, this is because it gets closer and closer to the P2P system as a result of continuously increasing the trees in the system. Of course, if we keep breaking the Grid tree into small subtrees until each subtree has only one scheduler, we will end up with a P2P system.

We were expecting average waiting times in the hierarchal systems to be in the range of milliseconds when there are only 520 jobs in a workload regardless of the average jobs size, since (1) the scheduled requests are small and independent of the average jobs size in the subject workload and (2) there are 520 servers in the Grid model at all times, thus, each server should get a request (i.e. job) immediately. However, the simulation proved us wrong. Actually, we had to trace the model log files in order to figure out the reason. It turned out that one or more requests were not mapped to resources because there were no available resources that matched their requirements. For example, at one of the 100 GB jobs-size runs, 518 requests got mapped to resources in milliseconds (as expected). However the two requests left did not match the two available resources and had to wait until other resources (that matched their requirements) became available. Of course, since a 100GB job takes a long time to execute that will considerably affect the overall average waiting time.

The total response time (TRT) also showed a significant improvement against the P2P system regardless of the number of used Grid trees or workload, as shown in Figure 39, Figure 40, Figure 41, Figure 42 and Figure 43. This metric shows that the proposed hierarchal systems takes more advantage of parallelism than the P2P system.

The average execution time (AET) is almost the same for small jobs (28.9MB and 312.7MB) as shown in Figure 44 and Figure 45, but starts to differ significantly when job-size increases beyond 1GB, as shown in Figure 46, Figure 47 and Figure 48, which makes sense, since the model is built with high performance links. Surprisingly, the P2P system did not perform well against the hierarchal systems at 1GB size, since it doesn't take a long time to transfer a 1GB job across a high-performance link.

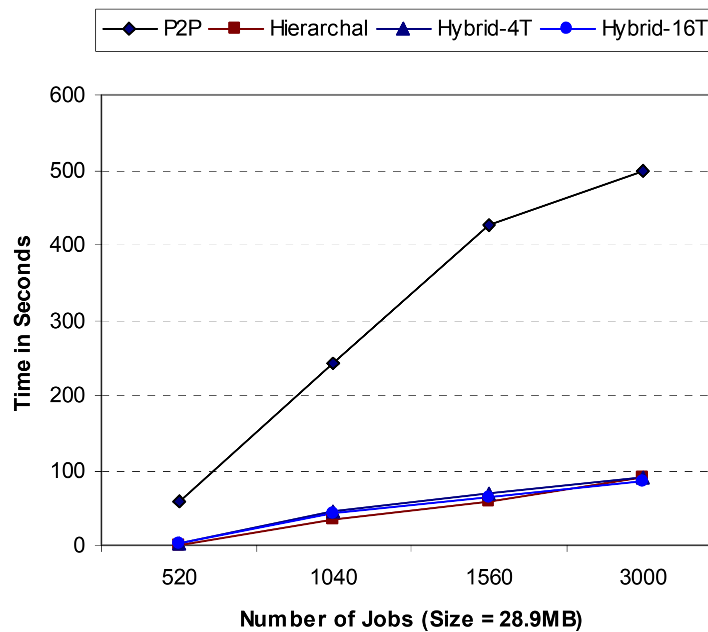


Figure 34: Average Waiting Times for 28.9MB Size Jobs in Experiment 1

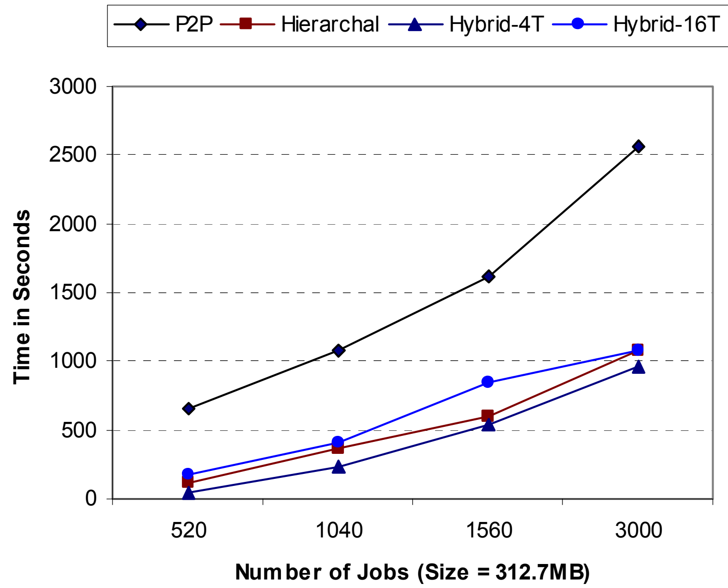


Figure 35: Average Waiting Times for 312.7MB Size Jobs in Experiment 1

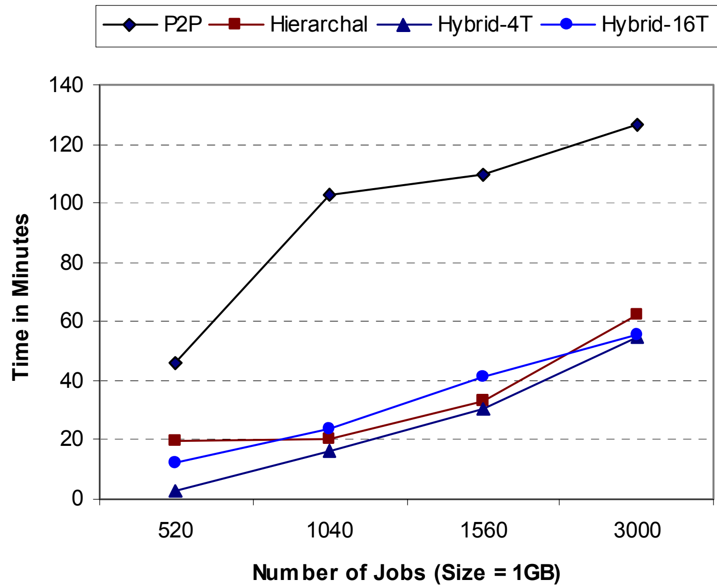


Figure 36: Average Waiting Times for 1GB Size Jobs in Experiment 1

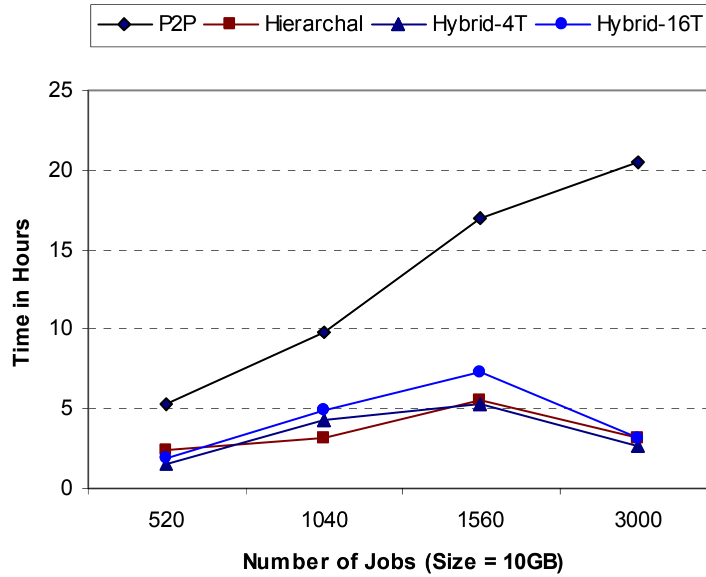


Figure 37: Average Waiting Times for 10GB Size Jobs in Experiment 1

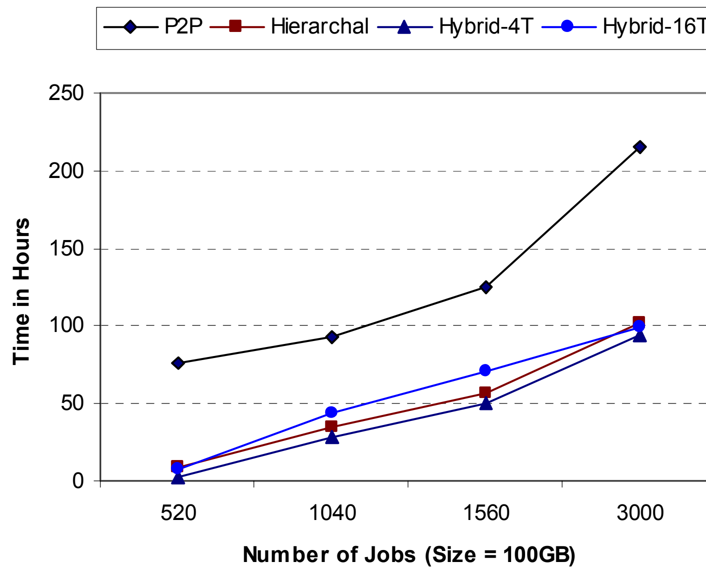


Figure 38: Average Waiting Times for 100GB Size Jobs in Experiment 1

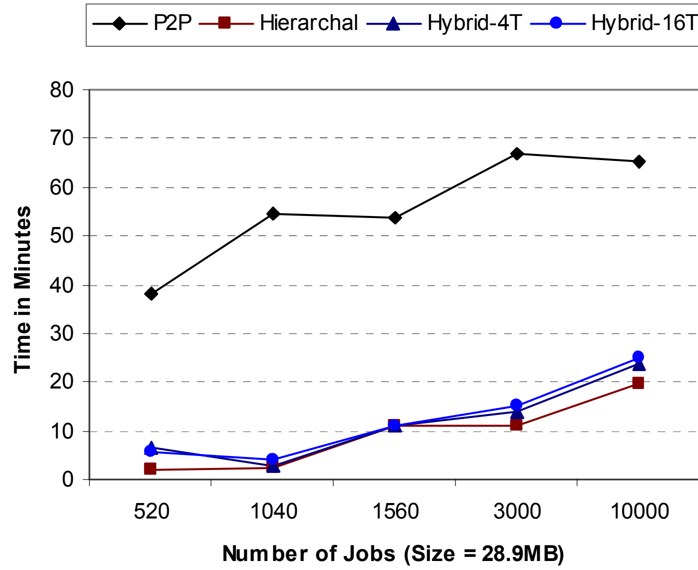


Figure 39: Total Response Times for 28.9MB Size Jobs in Experiment 1

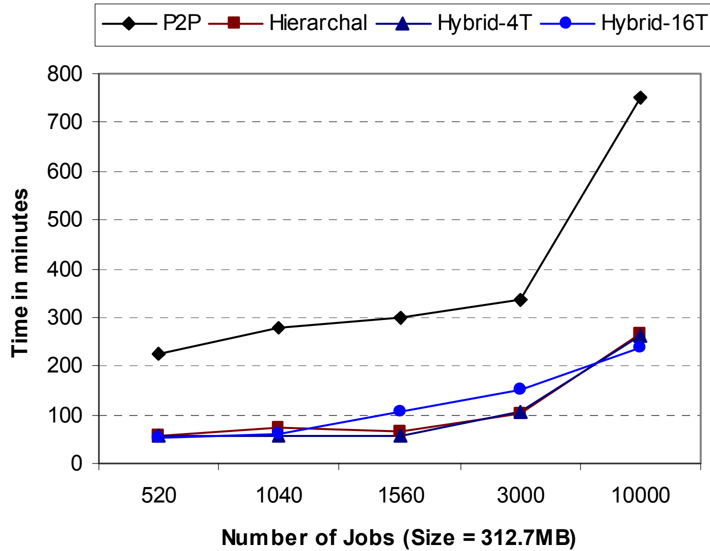


Figure 40: Total Response Times for 312.7MB Size Jobs in Experiment 1

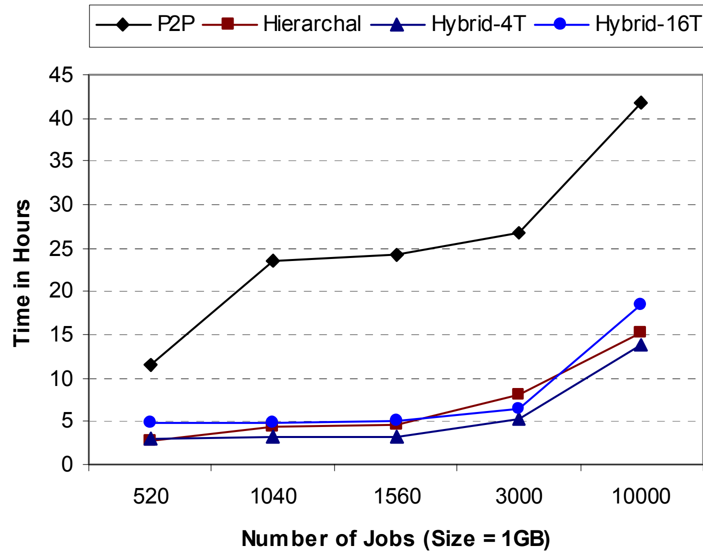


Figure 41: Total Response Times for 1GB Size Jobs in Experiment 1

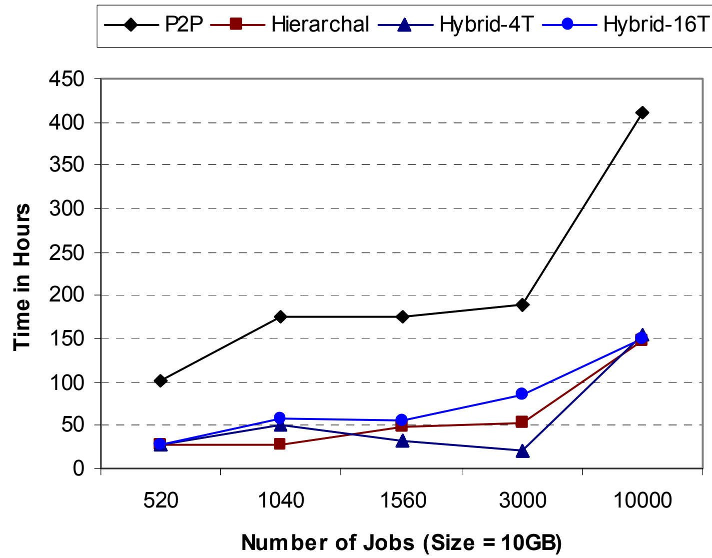


Figure 42: Total Response Times for 10GB Size Jobs in Experiment 1

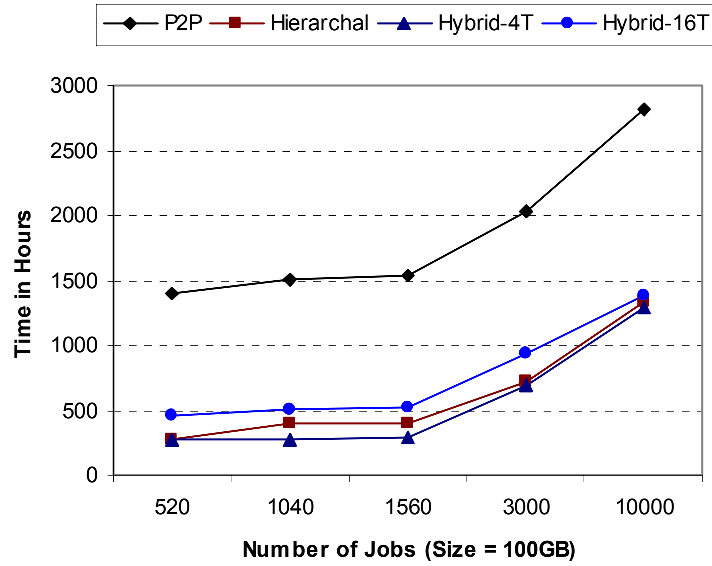


Figure 43: Total Response Times for 100GB Size Jobs in Experiment 1

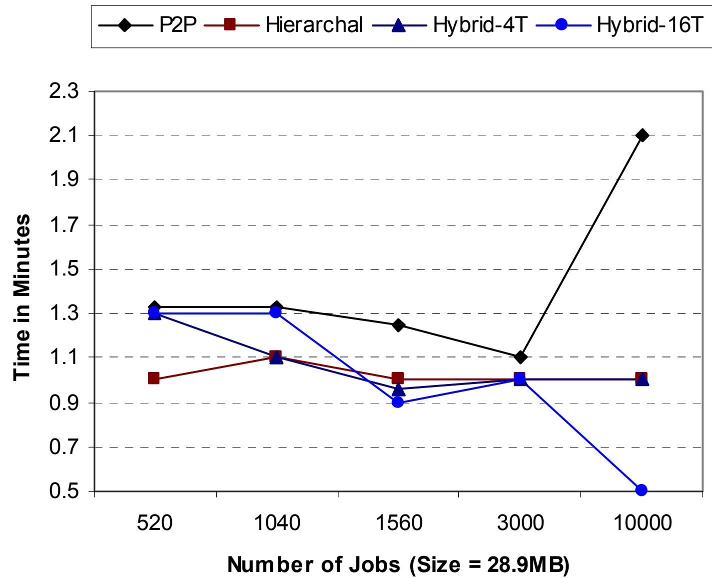


Figure 44: Average Execution Times for 28.9MB Size Jobs in Experiment 1

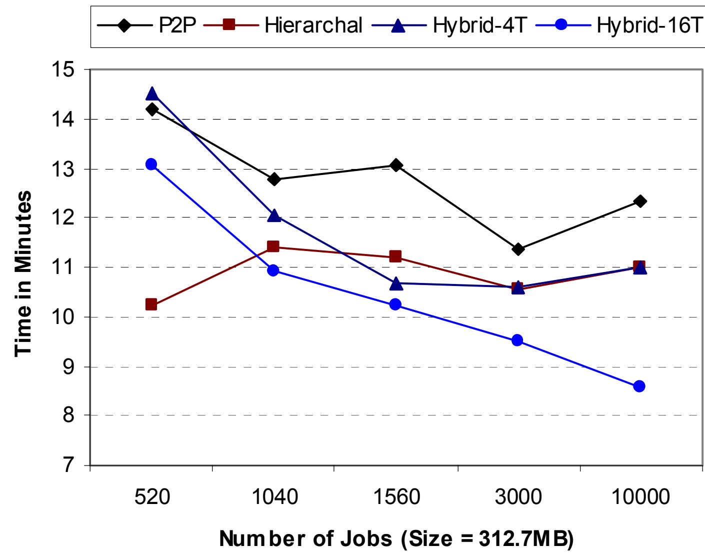


Figure 45: Average Execution Times for 312.7MB Size Jobs in Experiment 1

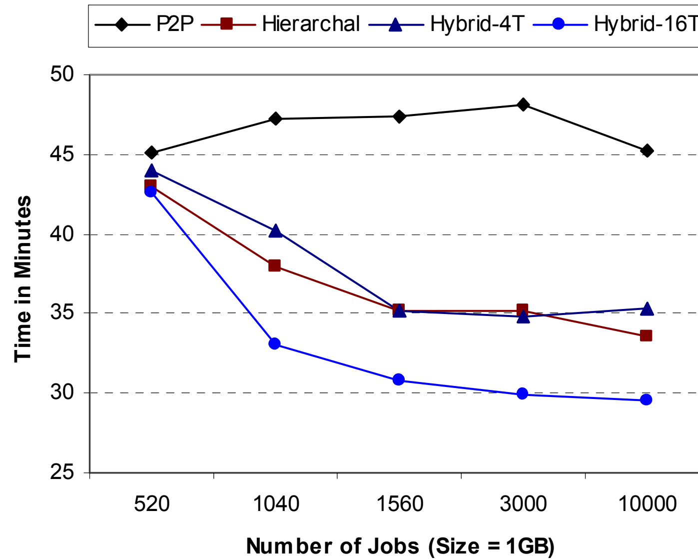


Figure 46: Average Execution Times for 1GB Size Jobs in Experiment 1

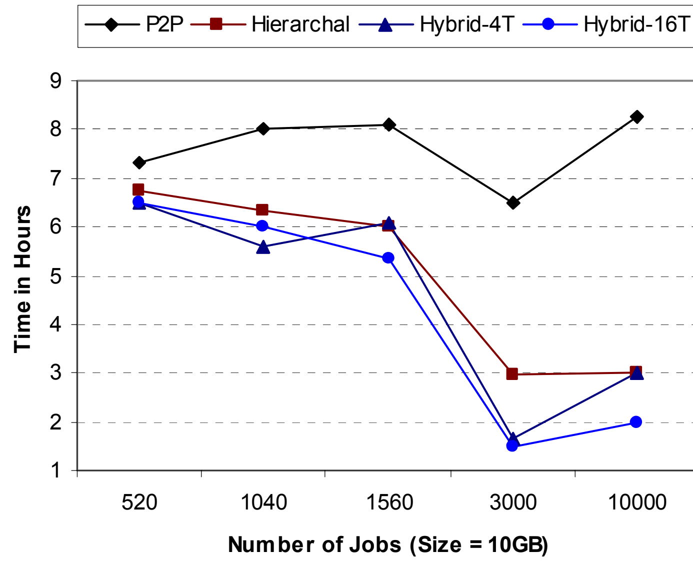


Figure 47: Average Execution Times for 10GB Size Jobs in Experiment 1

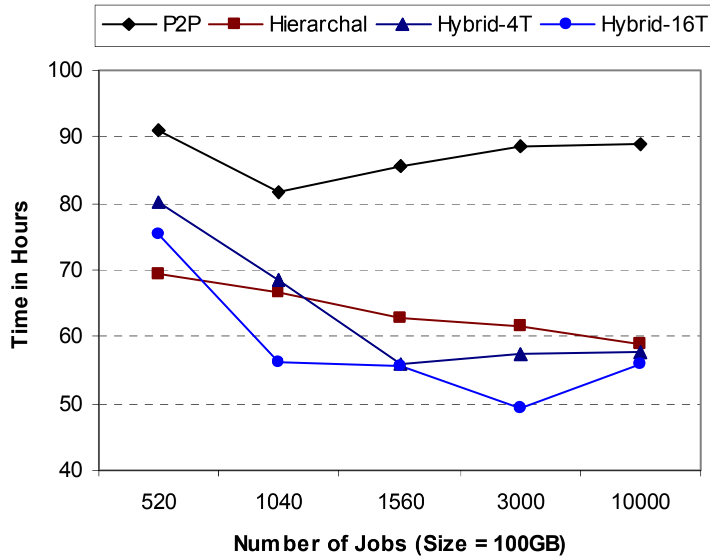


Figure 48: Average Execution Times for 100GB Size Jobs in Experiment 1

5.9.2 Second Experiment: Job Submissions with Different Stochastic Rates

The purpose of this experiment is to study the system behavior when jobs requests arrive to the Grid with different stochastic rates. For example, a workstation submits jobs to the Grid (if it is inactive) within 10 minutes submission rate (i.e. stochastic rate), but another one submits jobs within 5 hours rate. Obviously, this scenario is more realistic scenario than the presented ones in experiment 1 and 3.

Now, note the following assumptions:

- The number of workstations in a site does not determine the number of jobs that will be submitted from that site. For example, if site A has 3 workstations and site B has 6 workstations it is not assumed that site B is going to submit twice as many requests as site A. Also, it is possible that all requests will be submitted from site A.
- All workstations are not active simultaneously. In other words, when a workstation submits a request to execute a job, it waits for a period of time (i.e. each workstation operates using different stochastic rate) before submitting another request to execute a different job once it receives the first job's output. This case is typical when a workstation is used by people who arrive at different times to use that workstation.
- All requests are submitted from all locations with the same probability.
- Workstations operate using different stochastic submission rates; each workstation selects, at random, one of the following Poisson interval mean: 10 minutes, 30 minutes, 1 hour, 5 hours, 1 day or 1 week.

5.9.2.1 Results and Discussions

Note that the detailed result sets are presented in Table 5 in Appendix-A. Note also that we excluded the total response time (TRT) metric in comparing systems in this experiment because of the workstations waiting times (please see the full explanation in section 5.8).

As in the previous experiment, the average waiting time (AWT) showed a substantial improvement against the P2P system regardless of the number of used Grid trees or the workload, as shown in Figure 49, Figure 50, Figure 51, Figure 52 and Figure 53. The AWT of the hierarchal systems was less than 500ms for workloads with average jobs-size less than 1GB. In other words, requests were mapped to resources once they arrived to the Grid. Of course, because jobs arrived at a stochastic rate, the system would then have the time to distribute them among available resources. In fact, the proposed hierarchal schemes put scheduling-jobs in the hand of the Grid schedulers. However, in the P2P scheme, scheduling jobs is in the hand of jobs themselves – jobs continue searching for resources by themselves (i.e. schedulers only forward those jobs to neighbors). As a result, for example, Grid schedulers in P2P systems cannot break Grid jobs into subjobs so that they are executed in parallel among different sites because scheduling jobs is made blindly by those schedulers. In contrast, scheduling jobs *is not* made blindly in the proposed hierarchal schemes. Hence, jobs may, at least in theory, be broken and distributed among multiple children's partitions.

The average execution time (AET) was very close among all systems for very small average job-size workloads (at 28.9MB). However, the hierarchal schemes showed

considerable improvement against the P2P system for large and medium jobs-size workloads (more than 312.7 MB) beyond 520 jobs (because the model has 520 servers at all times – a job per resource), as shown in Figure 54, Figure 55, Figure 56, Figure 57 and Figure 58. This actually makes sense for several reasons, such as:

(1) The simulation model is built with high-performance links. Thus, the only difference among all systems in the execution stage is where jobs get executed.

(2) The P2P system is manually configured to have neighbors geographically close to each other (i.e. note that this is not necessarily true in real life). Thus, jobs grab the first close available resources.

(3) Jobs randomly arrive at the system with different arrival rates. Thus, in the P2P systems, it is more likely that jobs have close available resources.

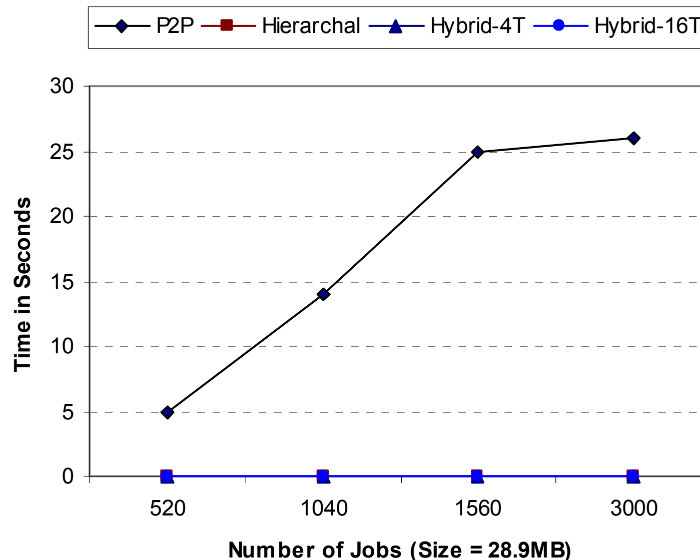


Figure 49: Average Waiting Times for 28.9MB Size Jobs in Experiment 2

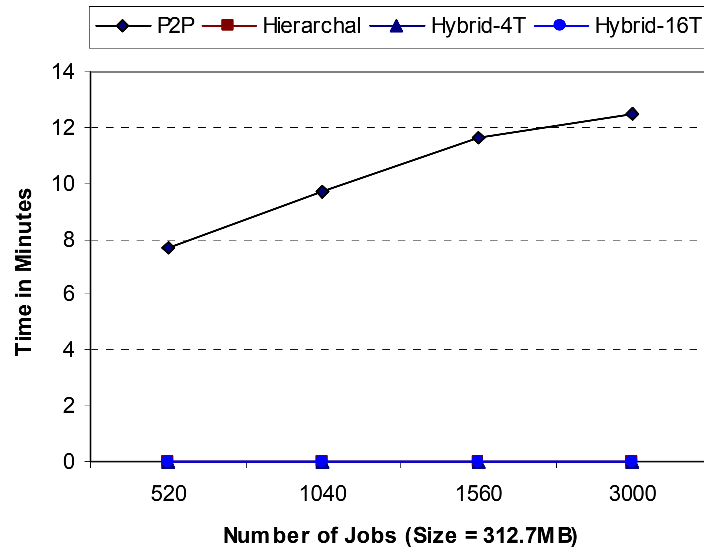


Figure 50: Average Waiting Times for 312.7MB Size Jobs in Experiment 2

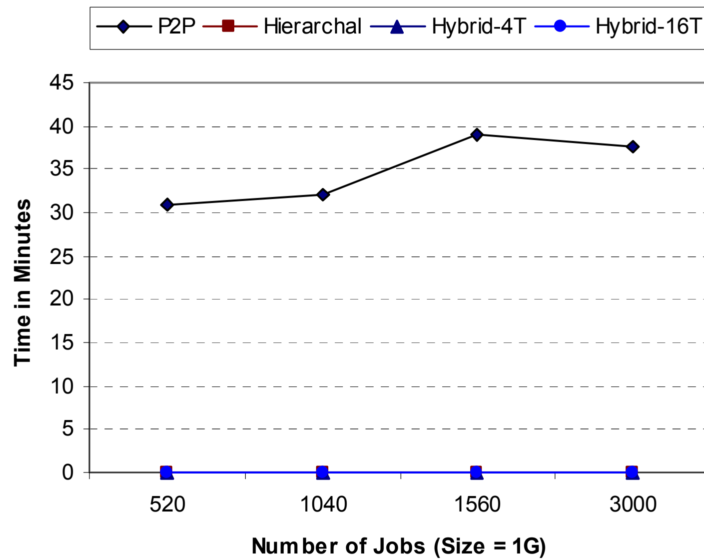


Figure 51: Average Waiting Times for 1GB Size Jobs in Experiment 2

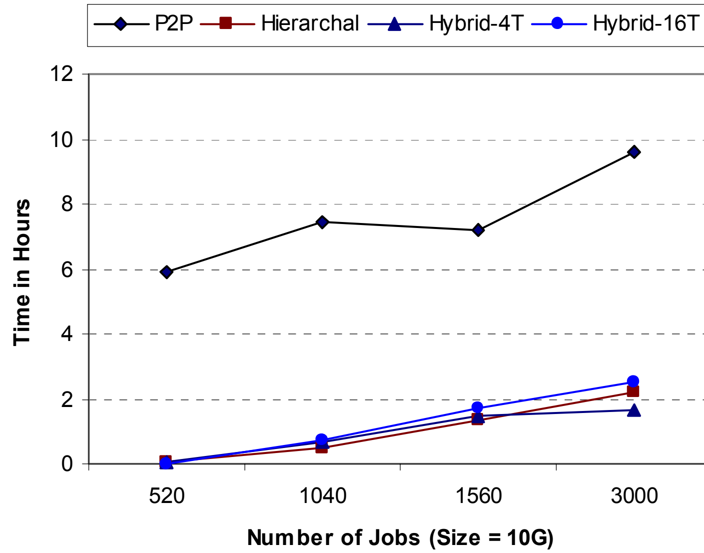


Figure 52: Average Waiting Times for 10GB Size Jobs in Experiment 2

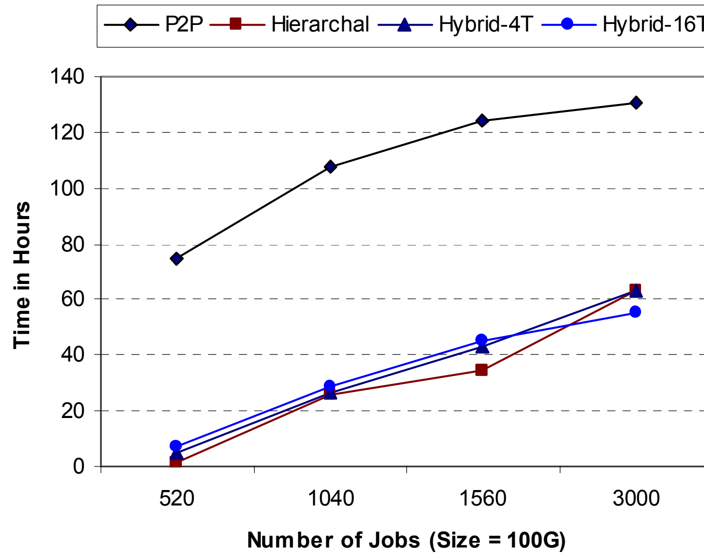


Figure 53: Average Waiting Times for 100GB Size Jobs in Experiment 2

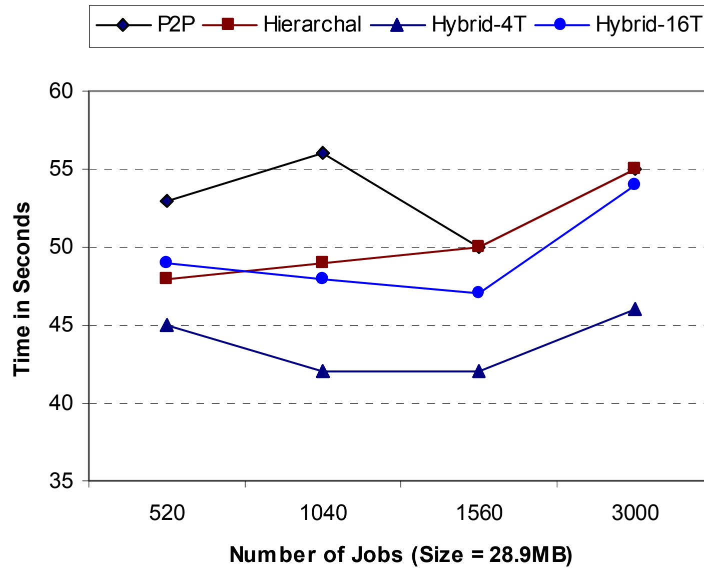


Figure 54: Average Execution Times for 28.9MB Size Jobs in Experiment 2

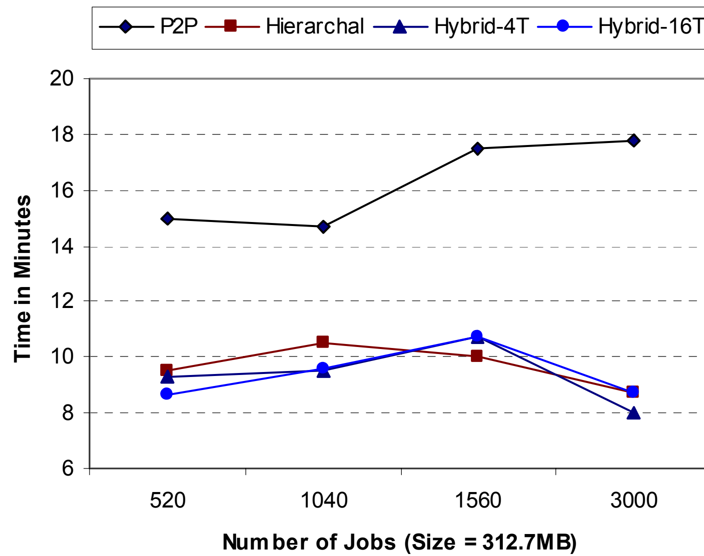


Figure 55: Average Execution Times for 312.7MB Size Jobs in Experiment 2

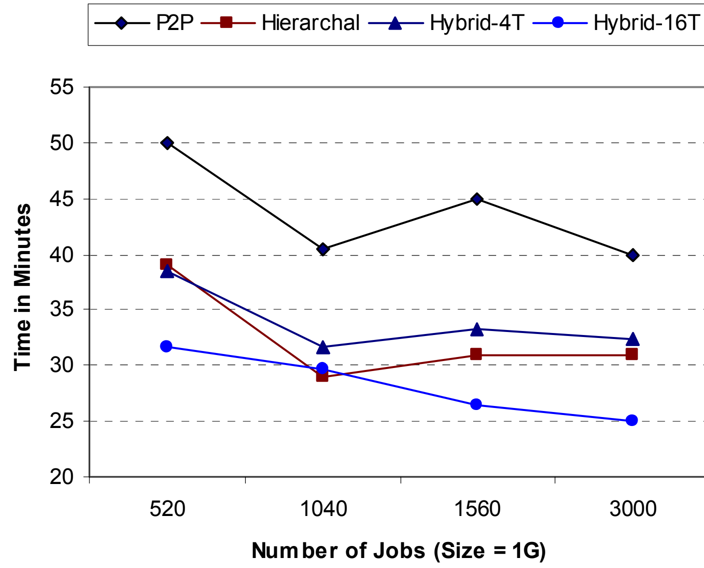


Figure 56: Average Execution Times for 1GB Size Jobs in Experiment 2

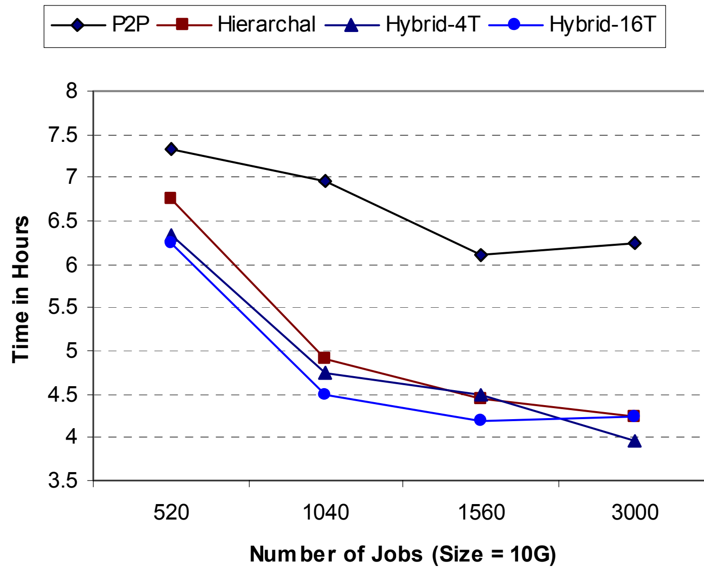


Figure 57: Average Execution Times for 10GB Size Jobs in Experiment 2

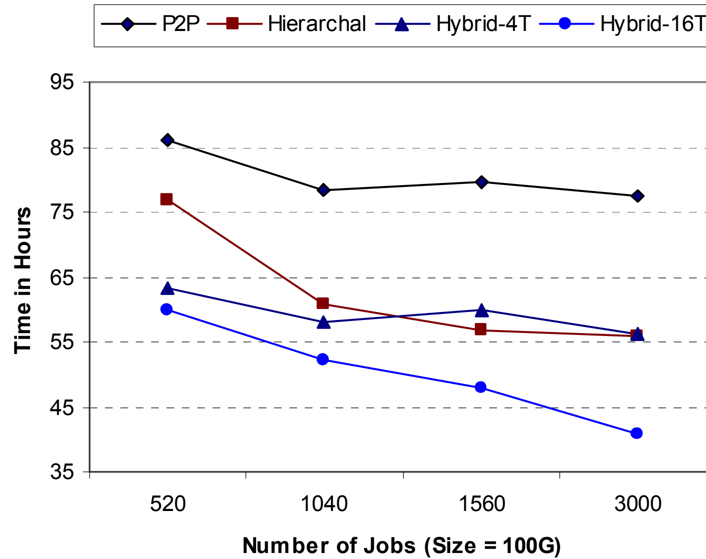


Figure 58: Average Execution Times for 100GB Size Jobs in Experiment 2

5.9.3 Third Experiment: Job Submissions with The Same Stochastic Rate

The purpose of this experiment is to study the system behavior when jobs requests arrive to the Grid within the same stochastic rate (i.e. all workstation submits jobs according to a Poisson process). We assume all users arrive to workstations at the same rate (e.g. Poisson's arrival mean = 1 hour). A user's arrival to a workstation determines when that workstation submits a request to the Grid. In other words, a workstation stays inactive until a user submits a request to the Grid from that workstation.

In this experiment, we only consider the P2P and hierarchal (i.e. one grid tree) systems because it serves the purpose of the experiment and saves very much time in

collecting results via simulation (i.e. note that we had to wait sometimes one week to get results for one simulation run for the P2P system).

Note the following assumptions:

- The number of workstations in a site determines the number of jobs that will be submitted from that site. For example, if site A has 3 workstations and site B has 6 workstations then, most likely, site B will submit twice the requests that are submitted from site A. Therefore, the load under experiment is already distributed among sites.
- All workstations are not active simultaneously. In other words, when a workstation submits a request to execute a job, it waits for a period of time before submitting another request to execute a different job after it receives the first job's output.
- A job is submitted by one workstation and executed by one server. However, in the case of a P2P system, the same request can be resubmitted more than once if the submitter workstation does not receive a service offer from a server within a specified time (i.e. in our case, 2 minutes).
- All requests are submitted from all locations with the same probability.
- All workstations have the same stochastic submission rate. For example, a Poisson submission rate mean is set to 10 minutes for all workstations. Note that results are collected for several values of Poisson interval mean: 1 hour, 1 day and 1 week. Furthermore, the average job size is 10 GB for all runs.

5.9.3.1 Results and Discussions

The detailed result sets are presented in Table 6 in Appendix-A. Also, we excluded the total response time (TRT) metric in comparing systems in this experiment because of the workstations waiting times (please see the full explanation in section 5.8).

As in the previous experiments, the average waiting time (AWT) showed a substantial improvement against the P2P system regardless of the number of used Grid trees or workload as shown in Figure 59, Figure 60 and Figure 61. The gap between AWT of the hierarchal system and the P2P system gets larger when increasing the arrival rate of the jobs, as shown in Figure 62. This indicates that the hierarchal system uses available resources more efficiently than the P2P system. This observation enhances the point that we've raised in previous experiments discussion that the hierarchal schemes put scheduling-jobs in the hand of schedulers, but the P2P system puts scheduling-jobs in the hand of jobs themselves. Of course, the smarter the schedulers get the better system performance we get.

The average execution time (AET) showed that the P2P system starts closing the gap with the hierarchal system when increasing the job-arrival rate until they meet, as shown in Figure 63, Figure 64, Figure 65 and Figure 66. This actually makes sense for several reasons, such as:

(1) The simulation model is built with high-performance links. Thus, the only difference among all systems in the execution stage is where jobs get executed.

(2) The P2P system is manually configured to have neighbors geographically close to each other (i.e. note that this is not necessary true in real life). Thus, jobs grab

first close available resources. Of course, the higher the rate, the more likely jobs will find nearby available resources.

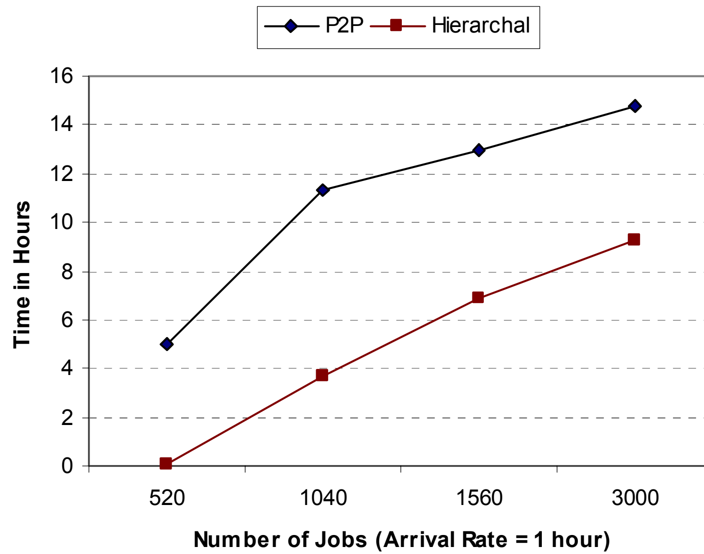


Figure 59: Average Waiting Times for 1 Hour Arrival Rate in Experiment 3

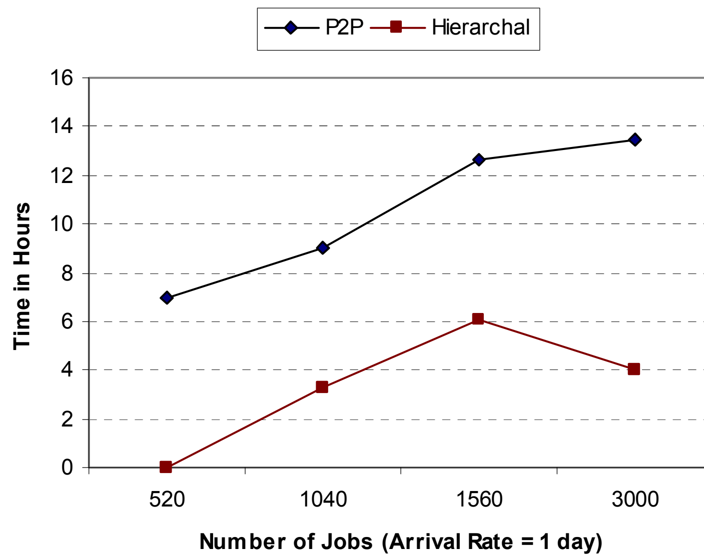


Figure 60: Average Waiting Times for 1 Day Arrival Rate in Experiment 3

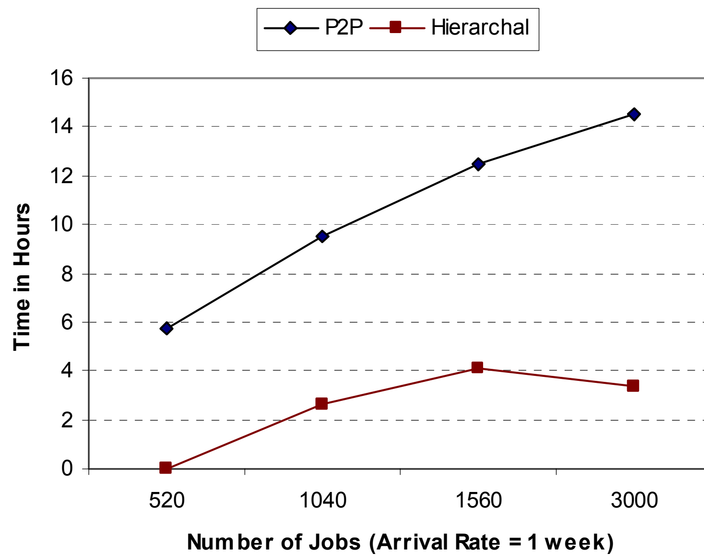


Figure 61: Average Waiting Times for 1 Week Arrival Rate in Experiment 3

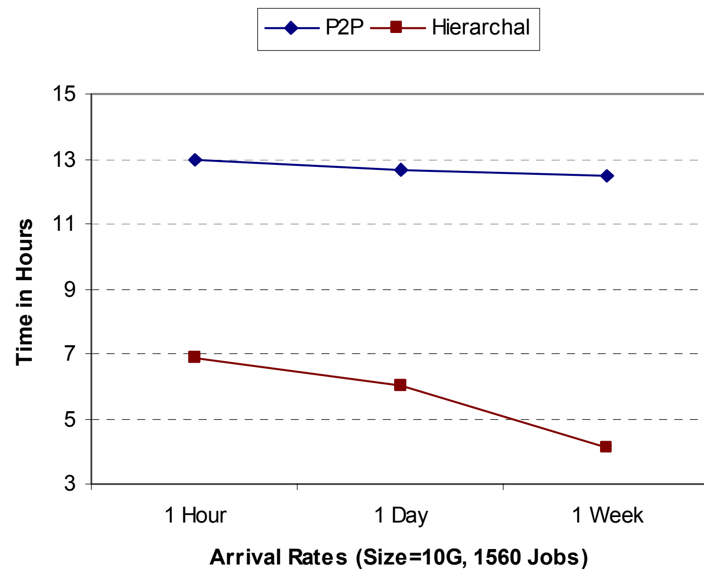


Figure 62: Average Waiting Times over Different Arrival Rates in Experiment 3

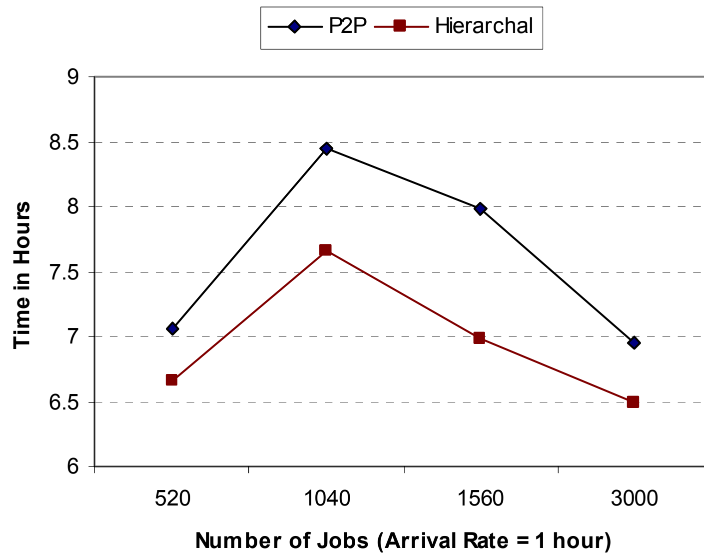


Figure 63: Average Execution Times for 1 Hour Arrival Rate in Experiment 3

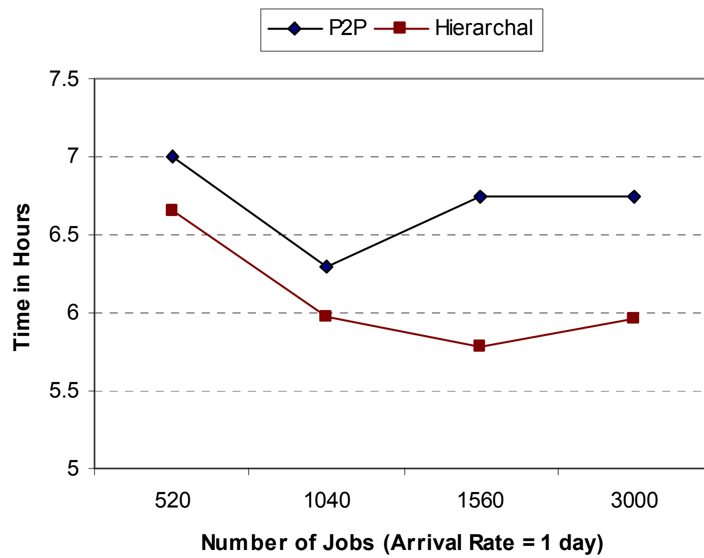


Figure 64: Average Execution Times for 1 Day Arrival Rate in Experiment 3

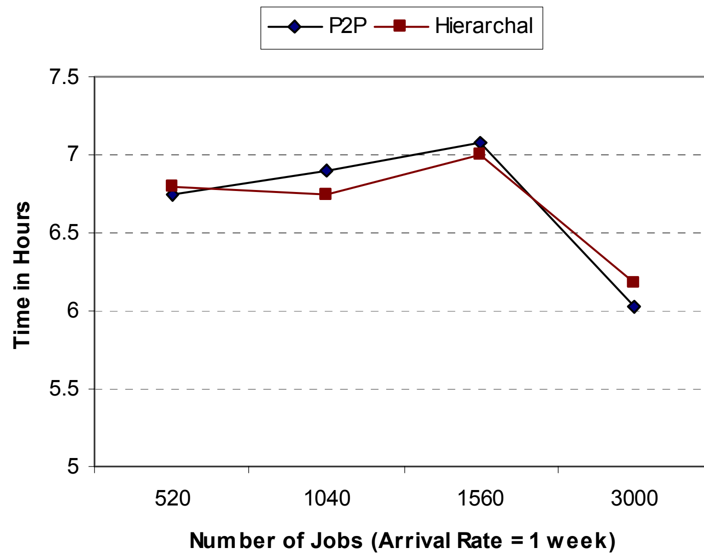


Figure 65: Average Execution Times for 1 Week Arrival Rate in Experiment 3

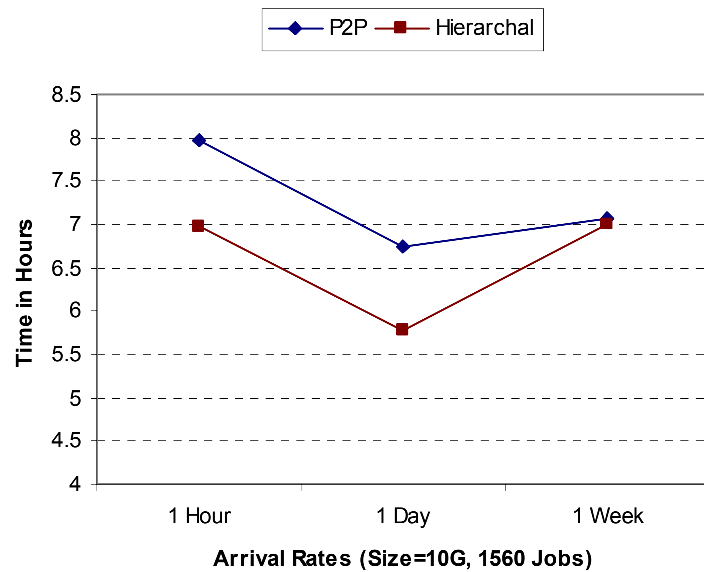


Figure 66: Average Execution Times over Different Arrival Rates in Experiment 3

5.9.4 Fourth Experiment: Algorithms Impact on System Performance

The purpose of this experiment is to study the contributions of the rescheduling algorithms (i.e. butterfly and load-balance algorithms) in the overall performance of the hierarchal scheduling. We chose to consider both of those algorithms together because they share the same purpose in rescheduling jobs from a child's partition (i.e. subtree) to another "better" partition. Therefore, by only enabling one algorithm, the principle of that algorithm will subsequently be defeated since it is designed to work with the other algorithm.

In this experiment, we repeat workloads for 100GB-average-size jobs using experiments 1 and 2 scenarios, while disabling the above two algorithms. The experiment is only carried out for the hierarchal system (i.e., one Grid tree) because that's the model it is designed for. Also, this saves considerable time in collecting results via simulation (e.g. 10 hours to complete one run).

5.9.4.1 Results and Discussions

The detailed result sets are presented in Table 7 in Appendix-A.

The load-balance and the butterfly algorithms proved their significant role on the average waiting time (AWT) when the number of jobs in workloads was increased, regardless of the used scenario, as shown in Figure 67 and Figure 68. This observation was also confirmed in studying the total response time (TRT), as shown in Figure 69. In fact, the AWT and TRT increased significantly (in a straight line) with increasing the number of jobs in workloads when our proposed algorithms are disabled. This behavior

actually makes sense, since the more pending jobs in the system; the more rescheduling is performed by those algorithms. Therefore, both load-balance and butterfly algorithms not only cut the overall system average waiting time, but also increase the degree of parallelism in the system.

The average execution time (AET) is either increased (i.e. first experiment scenario – more pressure on the system) or almost maintained at the same level (i.e. second experiment scenario – less pressure on the system) when our algorithms are disabled. However, interestingly, when enabling those algorithms, the AET decreases sharply in a straight line when increasing the number of jobs in the workloads, regardless of the scenario used. Of course, as mentioned earlier, the more pending jobs in the system; the more rescheduling is performed by those algorithms.

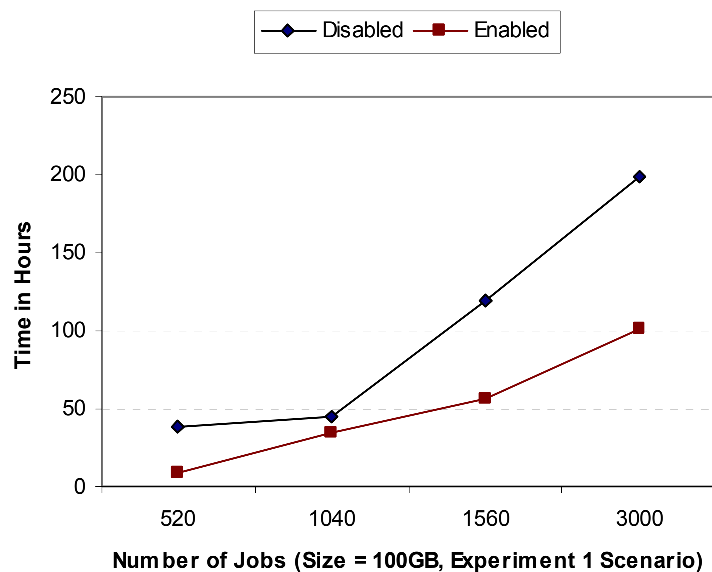


Figure 67: Average Waiting Times When Disabling Algorithms (Exp1 Scenario)

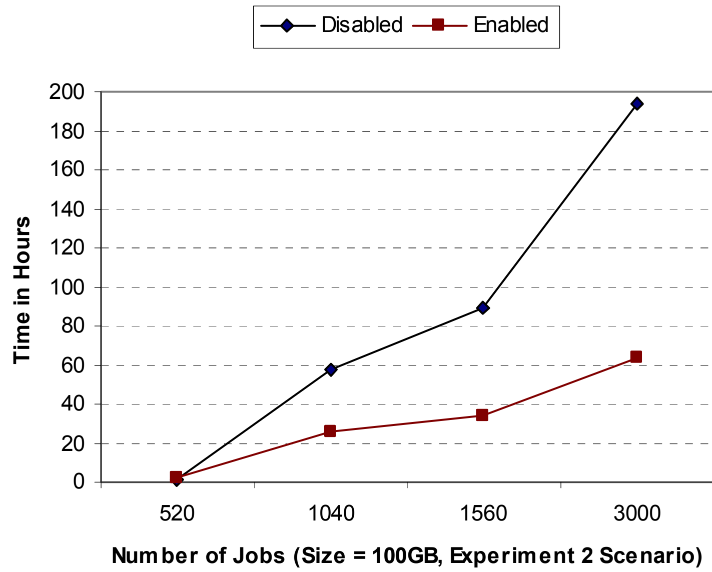


Figure 68: Average Waiting Times When Disabling Algorithms (Exp2 Scenario)

Figure 69: Total Response Times When Disabling Algorithms (Exp1 Scenario)

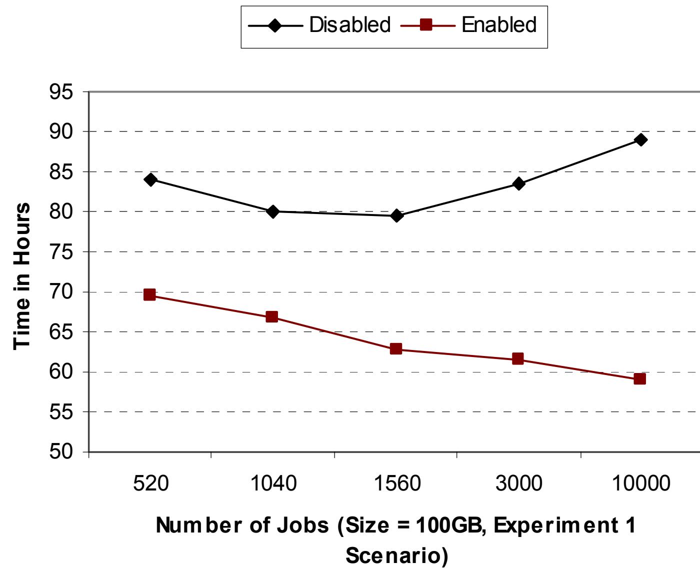


Figure 70: Average Execution Times When Disabling Algorithms (Exp1 Scenario)

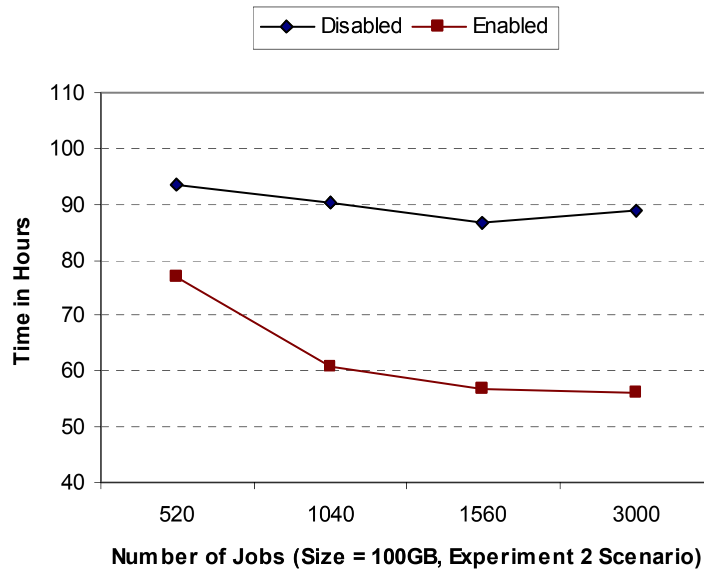


Figure 71: Average Execution Times When Disabling Algorithms (Exp2 Scenario)

5.9.5 Fifth Experiment: Resource Change Impact on System Behaviour

The purpose of this experiment is to study the performance of the proposed hierarchal scheduling when system resources change during scheduling. This experiment also aims to investigate the proposed fall-back algorithm by counting the number of saved jobs (i.e. due to resources change).

In this experiment, we use one of the following Poisson stochastic rates: one day, three days, one week, one month, three months and six months. At simulation start-up, a parallel computer selects its advertised operating system, and also its stochastic changing rate. Now, when a parallel computer changes its advertised operating system, it then reselects an operating system (perhaps the same one) and a changing rate (perhaps different from the old one). For example, a computer selects a 6 months changing rate and reselects a 3 months changing rate when it changes its resources.

Note that a saved job by the fall-back algorithm is counted only once. Therefore, if a job is saved more than once, it will be counted only once.

In this experiment, we repeat workloads for 10GB-size jobs using experiments 1 and 2 scenarios, while enabling resources change behavior. The experiment is only performed for the hierarchal system (for the same reasons as in the previous experiment).

5.9.5.1 Results and Discussions

The detailed result sets are presented in Table 8 in Appendix-A.

The fallback algorithm not only maintained the overall system performance when there is a possibility of resource changes, but also was able to save many jobs from not

being able to complete due to losing their selected resources, as shown in Figure 73, Figure 74, Figure 75, Figure 76 and Figure 77. Actually, this makes sense, since the fallback algorithm performs its rescheduling while resources are busy executing other jobs anyway – a job is already waiting in the system even if it falls on other resources.

The more jobs in the system also the more saved jobs, as shown in Figure 72. Of course, more jobs are saved in the experiment-1 scenario than in the experiment-2 scenario because the Grid is put under more pressure in experiment-1. Furthermore, the number of saved jobs is proportional to the number of jobs for the selected workload.

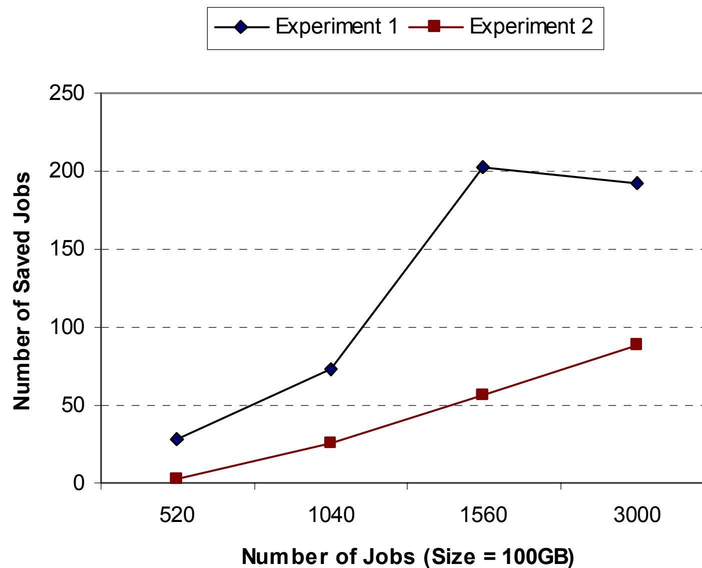


Figure 72: Number of Saved Jobs by the Fallback Algorithm (Exp1 & Exp2 Scenarios)

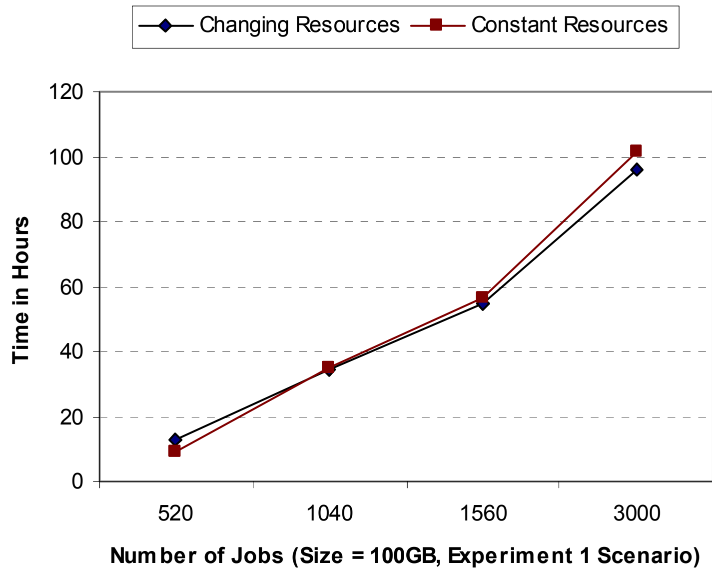


Figure 73: Average Waiting Times (Exp1 Scenario) in Experiment 5

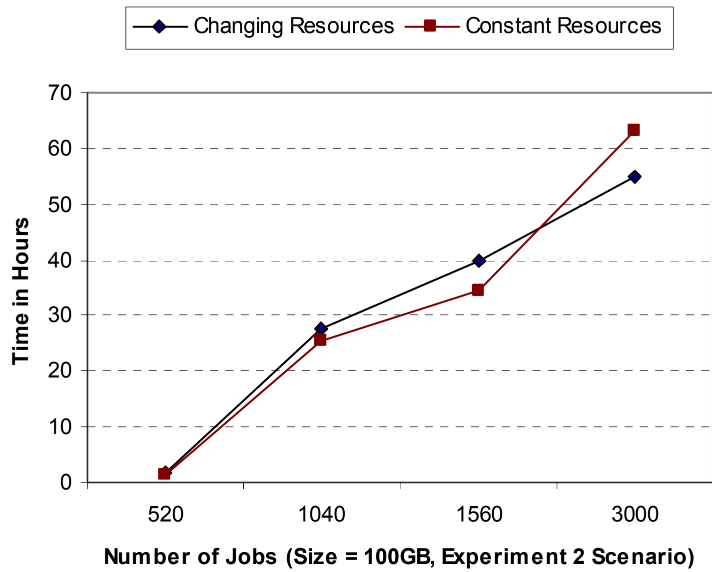


Figure 74: Average Waiting Times (Exp2 Scenario) in Experiment 5

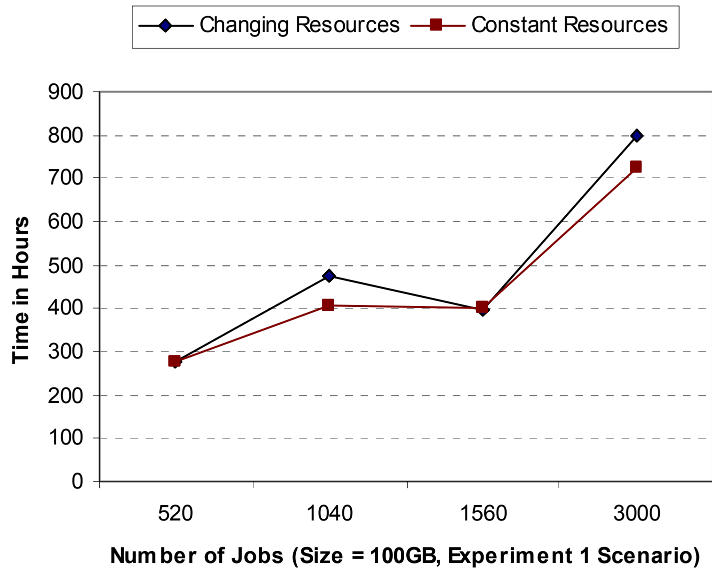


Figure 75: Total Response Times (Exp1 Scenario) in Experiment 5

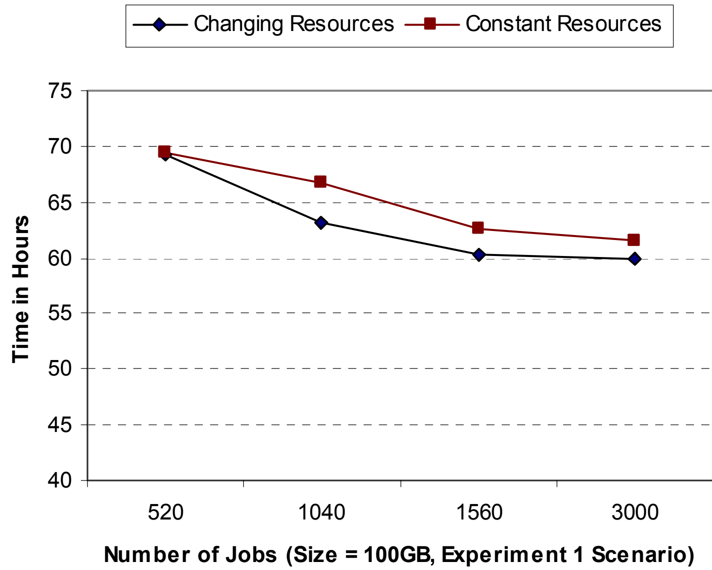


Figure 76: Average Execution Times (Exp1 Scenario) in Experiment 5

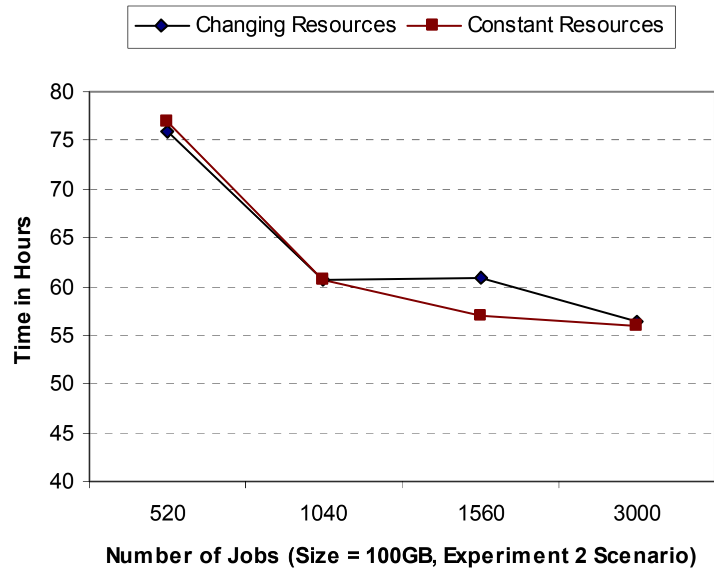


Figure 77: Average Execution Times (Exp2 Scenario) in Experiment 5

CONCLUSIONS AND FUTURE WORK

The hierarchal approach has not only shown substantial improvement over the P2P-based Grid system, in the experiments reported in this dissertation, but also has demonstrated its capability to be successfully combined with a hybrid system. The hybrid approach, as discussed previously, increases system scalability and provides better resources management than the single Grid tree or the P2P approaches.

Both the average waiting time (AWT) and the total response time (TRT) metrics showed a large improvement of the hierarchal approach against the P2P system regardless of the number of used Grid trees, workload or scenario. The average execution time (AET) metric also showed a significant improvement over the P2P system particularly for medium and large-size jobs. However, in some cases, the AET showed almost identical results for all systems for small-size jobs. This actually makes sense because we've used high-performance communication links and manually configured neighbors in the P2P-based systems to be geographically close to each other. Furthermore, we were only interested in where a job got executed, hence jobs were handled the same way when they reached physical resources, regardless of the system under experiment.

The three proposed rescheduling algorithms showed to contribute significantly to the overall performance of the proposed hierarchical system. The fallback algorithm saved not only a number of jobs from not being able to execute because of resources change, but also maintained almost the same system performance when resources are constant. The number of saved jobs was proportional to the number of jobs in the selected workload and the rates at which resources change. In addition, both the butterfly and load-balance algorithms saved the system from performing poorly when increasing the number of jobs in workloads. Obviously, the more jobs exist simultaneously in the system, the more rescheduling is needed to balance load among resources and/or transfer jobs to better available resources.

We would like to emphasize that many studies in the Grid scheduling literature, jump over the resource discovery stage directly into the second scheduling stage (i.e. resource selection) by assuming that all jobs can execute anywhere in the Grid, or simply assuming that resources will be discovered using the P2P approach. However, as we've shown in this thesis, those stages have to be dealt with in a sequence because of their dependency on each other. In the P2P-based Grid systems, a scheduler queues a request locally (if that request is accepted), contacts the user's workstation and communicates with the workstation one to one. On the other hand, it forwards rejected requests to it neighbors. Therefore, all of the P2P-based studies come down to one scheduling question: under what circumstances does a scheduler (i.e. peer) accept a request or instead forward that request to its neighbors (see section 1.3.2)? However, in the hybrid system, PGSSs may still forward accepted requests to neighboring Grid trees (i.e. in our case, a PGSS always forward accepted requests to its neighbors so that Grid trees serve as

backups to each other). This shows one of the distinctions between the hybrid and the P2P systems: a peer in a hybrid system is actually an entire system on its own (i.e. Grid tree), but a peer in the P2P system is essentially a node (i.e. one scheduler). In practice, we expect a specific organization to construct its own Grid tree(s) on top of their resources in order to gain more control over its own resources. Therefore, a request, in the hybrid approach, jumps from a system to another (or from an organization to another) rather than from a node to another, as with the P2P systems. This observation may provide some explanation about the fact that many jobs in our workloads didn't get the chance to execute when the P2P-based system was used. To overcome this problem, we adopted a rule that workstations resubmit their requests to the P2P-based Grid system if they don't receive a service offer within a configured number of minutes (e.g. 2 minutes).

Furthermore, the P2P approach puts the burden of discovering resources on the jobs. Schedulers "blindly" forward requests to their neighbors with the hope that those jobs will find appropriate resources. However, as was shown in this thesis, the hierarchal approach gives schedulers more "say" in discovering resources for jobs and in distributing them among resources. This is not a trivial issue if we want to gain the full benefits of the Grid systems. For example, a Grid system should aim to use computing resources in parallel to get the full advantage of its power (i.e. simply, 10 computers can work faster and better than one computer can do). Therefore, Grid schedulers, in the future, need to break Grid jobs into subjobs and to execute them in parallel on multiple resources. Currently, we don't see how peers in the P2P-based Grid systems can carry out this task. However, in theory, any Grid scheduler in a Grid tree may break a job into subjobs and execute that job in parallel among its children partitions (see Figure 21).

Job scheduling using one central scheduler (that is responsible for discovering resources, selecting resources and executing jobs) is a very common approach and still exists today in many commercial and public-domain systems [26] such as Condor [13, 42], PBS [33, 36], Maui [24, 30], and LSF [28, 29]. Conversely, the peer-to-peer (P2P) approach which is currently used for file sharing systems has been proposed by researchers in order to replace the centralized architectures in the Grid systems. Of course, this is because the P2P approach is a distributed system; hence, it can be a more practical approach than the centralized one for many obvious reasons. However, to our knowledge, the P2P approach has not been used for the Grid systems yet.

One of purposes of this thesis is to “start” the work of putting multiple layers of schedulers (e.g. a Grid tree, in our case) on top of resources where jobs will eventually run. The idea of building layers of schedulers has been mentioned in literature as the way to go in the future for Grid scheduling. However, we couldn’t find any published scheme to structure Grid schedulers in layers. Note that [34] (probably the only book devoted entirely to the open research questions of Grid resource management) advocates the need of putting “multiple layers of schedulers” (in the first page of its preface). There are still many open research questions in Grid resource management [34]. In this dissertation, we proposed a hierarchy structure where schedulers are placed in layers on top of resources where jobs will eventually run.

Clearly, we did not intend to tackle all of the Grid open research questions. Instead, this research has mainly focused on the first two scheduling stages (i.e. resource discovery and resource selection) of Grid systems. There is definitely a need for future research work into the jobs execution stage, as well as in security and fault-tolerance.

Moreover, many issues can be explored in this last stage to improve the overall performance and reliability of the hierarchal Grid systems. Completion deadlines, resources reservations, jobs preemptions are examples of such issues.

Because we were mainly concerned with the two first scheduling stages (i.e. waiting time for a request), we chose to break the *average response tim*e (i.e. the time from submitting a request to the Grid until the workstation receives the output) into the average *waiting* and *execution* times. However, in the future, once the execution stage is addressed, the *average response time* will be definitely measured as one metric.

Finally, without a doubt, security appears to us as the most important issue to currently tackle in Grid systems or any other system. Fundamentally, the Grid has to authenticate a user identity and authorize that user to access resources. Of course, the process of authentication and authorization can become very complicated, since users are usually authorized to access only a set of resources (i.e. not all resources in the Grid). In our case, we can treat the process of authentication and authorization as part of the job requirement to use resources. Therefore, if a user is not authorized to use a specific resource, the request will subsequently not match that resource. In future work, we will need to come up with a method to accumulate those user Ids in the Grid trees. Do we store them in LGSs where matching is performed between resources and requests? Or in GSSs where requests are submitted by users? Or in intermediate GSs enabling them to control access to their partitions? Considerable experimentation will be required to answer such questions.

Fault-tolerance is also a major issue in Grid computing. As a system increases both in size and complexity, the possibility of a component failure also increases.

However, fault-tolerance has so far received little in Grid computing literature. The work in [2] provides an excellent start to incorporate fault-tolerance capability to our hierarchal architecture in this thesis. The work in [2] proposes a fault-tolerance layer, which is embedded in Grid schedulers underneath the resource management layer (i.e. this layer is actually the scheduling component which is this thesis topic). We believe this thesis completes the work in [2] by providing the resource management layer that handles Grid scheduling. The fault-tolerance layer consists of facilities that (1) monitor the status of the resources, services and application, (2) detect and report failures, and (3) analyze and recover from failures. Furthermore, the work in [2] structures Grid schedulers in a tree (similar to Grid trees in this thesis). For example, once a scheduler dies, one of its children takes its place and becomes the parent. Parents and children continuously send "I am alive" message to each other and if this message is not received within a specified time from a scheduler, it is assumed dead. We envision using the proposed fault-tolerance layer in the Grid schedulers and repeat our simulation experiments with the possibility of scheduler failures. We will also need to address failure tolerance for the hybrid system, since work in [2] only assumes one single tree.

REFERENCES

[1] J. Abawajy and S. Dandamudi. Scheduling Parallel Jobs with CPU and I/O Resource Requirements in Cluster Computing Systems. Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on Pages: 336 – 343.

[2] J. Abawajy and S. Dandamudi. Fault-Tolerant Grid Resource management infrastructure. 2003. Neural, Parallel & Scientific Computations archive, Volume 12, Pages: 289 – 306, Issue 3 (2004).

[3] Bill Allcock, Ian Foster, Veronika Nefedova, Ann Chervenak, Ewa Deelman, Carl Kesselman, Jason Lee, Alex Sim, Arie Shoshani, Bob Drach and Dean Williams. High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies. ACM/IEEE SC 2001 Conference (SC'01)

[4] E. Adar and B. A. Huberman. Free riding on Gnutella. First Monday, 5, 2000. Also available from http://www.firstmonday.dk/issues/issue5_10/adar/.

[5] Alain Roy and Miron Livny. Condor and Preemptive Resume Scheduling. Chapter 9 in [34] book.

[6] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance evaluation model for scheduling in a global computing system. The International Journal of High Performance Computing Applications, vol. 14, No. 3, pp. 268-279, 2000.

[7] R. Buyya, D. Abramson, J. Giddy: An Economy Driven Resource Management Architecture for Global Computational Power Grids. International Conference on Parallel and Distributed Processing Techniques and Applications (2000).

[8] Viktors Berstis. Fundamentals of Grid Computing. Available from http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf.

[9] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing, August 2001. Also available from http://www.globus.org/research/papers/MDS-HPDC.pdf.

[10] Cao, J., Kwong, O.M.K., Wang, X. and Cai, W. (2005). A peer-to-peer approach to task scheduling in computation Grid. Int. J. Grid and Utility Computing, Vol. 1, No. 1, pp.13–21.

[11] K.G. Coffman and Andrew Odlyzko. The Size and Growth Rate of the Internet. http://www.firstmonday.org/issues/issue3_10/coffman/

[12] K.G. Coffman and Andrew Odlyzko. Internet growth: Is there a “Moore’s Law” for data traffic?. http://www.dtc.umn.edu/~odlyzko/doc/internet.moore.pdf.

[13] Condor Project. http://www.cs.wisc.edu/condor/.

[14] S. Dandamudi. Hierarchical scheduling in Parallel and Cluster systems. Kluwer Academic Publishers, 2003.

[15] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. Workflow Management in GriPhyN. Chapter 7 in [34] book.

[16] Holly Dail, Otto Sievert, Fran Berman, Henri Casanova, Asim YarKhan, Sathish Vadhiyar, Jack Dongarra, Chuang Liu, Lingyun Yang, Dave Angulo, and Ian Foster. Scheduling in the Grid Application Development Software Project. Chapter 6 in [34] book.

[17] Carsten Ernemann, Volker Hamscher, Uwe Schwiegelshohn, Ramin Yahyapour. On Advantages of Grid Computing for Parallel Job Scheduling. http://wwwcs.uni-paderborn.de/pc2/papers/files/392.pdf

[18] Ian Foster and Carl Kesselman, editors. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 2004.

[19] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Available from http://www.globus.org/research/papers/anatomy.pdf..

[20] W. Gibbs. Ripples in spacetime. Scientific American, April 2002.

[21] F. Giacomini and F. Prelz. Definition of architecture, technical plan and evaluation criteria for scheduling, resource management, security and job description. Technical Report DataGrid-01-D1.4-0127-1 0, European DataGrid Project, 2001. Available from http://server11.infn.it/workload-Grid/docs/DataGrid-01-D1.4-0127-1_0.doc

[22] GriPhyN: The Grid Physics Network. http://www.griphyn.org.

[23] S. Hotovy. Analysis of the early workload on the Cornell Theory Center IBM SP2. Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. Pages: 272 – 273.

[24] David B. Jackson. Grid Scheduling with Maui/Silver. Chapter 11 in [34] book.

[25] J. Kaufman, Toby J. Lehman, and J. Thomas. Grid computing made simple. Available from http://www.aip.org/tip/INPHFA/vol-9/iss-4/p31.pdf.

[26] Krzysztof Kurowski, Jarek Nabrzyski, Ariel Oleksiak, and Jan W˛eglarz. Multicriteria Aspects of Grid Resource Management. Chapter 18 in [34] book.

[27] Rajesh Kalmadyand and Brian Tierney. A Comparison of GSIFTP and RFIO on a WAN. Available from http://edg-wp2.web.cern.ch/edg-wp2/docs/GridFTP-rfio-report.pdf.

[28] Ian Lumb and Chris Smith. Scheduling Attributes and Platform LSF. Chapter 12 in [34] book.

[29] LSF. http://www.platform.com/services/support/docs/LSFDoc51.asp.

[30] Maui. http://www.supercluster.org/maui.

[31] GT Information Services: Monitoring & Discovery System (MDS). http://www-unix.globus.org/toolkit/mds/

[32] Napster. http://www.napster.com.

[33] Bill Nitzberg, Jennifer M. Schopf, and James Patton Jones. PBS Pro: Grid Computing and Scheduling Attributes. Chapter 13 in [34] book.

[34] J. Nabrzyski, J. Schopf and J. Weglarz. Grid resource management: state of the art and future trends, Kluwer Academic Publishers, 2004.

[35] A. Odlyzko. Internet traffic growth: Sources and implications. http://www.dtc.umn.edu/~odlyzko/doc/itcom.internet.growth.pdf

[36] PBS: The Portable Batch System. http://www.openpbs.org/.

[37] Charles E. Perkins. Mobile IP: Design Principles and Practices, Addison-Wesley, 1997.

[38] Michael Russell, Gabrielle Allen, Tom Goodale, Jarek Nabrzyski, and Ed Seidel. Application Requirements for Resource Brokering in a Grid Environment. Chapter 3 in [34] book.

[39] Marisa Ruffolo and Mark S. Daskin. Design of a Large Network for Radiological Image Data Extended Abstract. http://www.kellogg.northwestern.edu/msom2005/papers/Daskin.pdf.

[40] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In Proceedings of the Seventh IEEE International Symposium
on High-Performance Distributed Computing, 1998.

[41] Long Term Technology Review of the Science & Engineering Base. FULL REPORT - April 2000. http://www.rcuk.ac.uk/lttr/finalversion/LTTR_ContentsPage.htm.

[42] Rajesh Raman, Marvin Solomon, Miron Livny, and Alain Roy. The ClassAds Language. Chapter 17 in [34] book.

[43] Jennifer M. Schopf. Ten Actions When Grid Scheduling. Chapter 2 in [34] book.

[44] H. Shan, W. Smith, L. Oliker and R. Biswas. Job Scheduling in a Heterogeneous Grid Environment. http://www-library.lbl.gov/docs/LBNL/549/06/PDF/LBNL-54906.pdf

[45] William Stallings. Data and Computer Communications, fifth edition. Prentice Hall, 1997.

[46] Richard Stevens. TCP/IP Illustrated, Volume 1. Addison Wesley, 1994.

[47] Richard Stevens. TCP/IP Illustrated, Volume 2. Addison Wesley, 1994.

[48] Uwe Schwiegelshohn and Ramin Yahyapour. Attributes for Communication between Grid Scheduling Instances. Chapter 4 in [34] book.

[49] Jennifer M. Schopf and Lingyun Yang. Using PredictedVariance for Conservative Scheduling on Shared Resources. Chapter 15 in [34] book.

[50] A. Takefusa Bricks: A Performance Evaluation System for Scheduling Algorithms on the Grids. JSPS Workshop on Applied Information Technology for Science (JWAITS 2001).

[51] A. Takefusa, H. Casanova, S. Matsuoka, F. Berman. A Study of Deadline Scheduling for Client-Server Systems on the Computational Grid. 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10 '01) P. 406.

[52] A. Takefusa, S. Matsuoka. Performance Issues in Client-Server Global Computing. International Workshop on Global and Cluster Computing (WGCC'2000).

[53] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a Performance Evaluation System for Global Computing Scheduling Algorithms. In

Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC), pages 97–104, August 1999.

[54] A. Takefusa, O. Tatebe, S. Matsuoka, and Y. Morita. Performance Analysis of Scheduling and Replication Algorithms on Grid Datafarm Architecture for High-Energy Physics Applications. Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), pp. 34-43. 2003.

[55] Mary R. Thompson and Keith R. Jackson. Security Issues of Grid Resource Management. Chapter 5 in [34] book.

[56] Gabriel Wainer. Published papers about the DEVS tools. Available from http://www.sce.carleton.ca/faculty/wainer.html.

[57] Parallel Workloads Archive. Available from http://www.cs.huji.ac.il/labs/parallel/workload/.

[58] Rich Wolski, Lawrence J. Miller, Graziano Obertelli, and Martin Swany. Performance Information Services for Computational Grids. Chapter 14 in [34] book.

[59] Wolski, R. Dynamically Forecasting Network Performance Using the Network Weather Service. Cluster Computing (1998)

[60] Xuehai Zhang, Jeffrey Freschl, and Jennifer M. Schopf. A performance study of monitoring and information services for distributed systems. In Proceedings of the IEEE Twelfth International Symposium on High-Performance Distributed Computing (HPDC-12), 2003.

APPENDIX A – RESULT TABLES

A.1 FIRST EXPERIMENT RESULTS SET

Average Jobs Size	System Type	Number of Jobs	Average Waiting Time (HH:MM:SS)	Average Execution Time (HH:MM:SS)	Total Response Time (HH:MM:SS)
28.9 MB	P2P	520	00:00:59	00:01:17	00:38:02
		1040	00:04:03	00:01:16	00:54:42
		1560	00:07:06	00:01:13	00:53:45
		3000	00:08:31	00:01:04	01:07:02
		10000	00:04:06	00:02:05	01:05:11
	Hierarchal (1 Grid tree)	520	50ms	00:01:01	00:01:54
		1040	00:00:36	00:01:09	00:02:45
		1560	00:00:59	00:01:02	00:11:00
		3000	00:01:31	00:01:03	00:10:57
		10000	00:01:17	00:01:00	00:19:40
	Hybrid (4 Grid trees)	520	00:00:02	00:01:15	00:06:34
		1040	00:00:45	00:01:10	00:03:00
		1560	00:01:10	00:00:58	00:11:15
		3000	00:01:40	00:01:02	00:14:30
		10000	00:01:51	00:01:00	00:24:30
	Hybrid (16	520	00:00:03	00:01:17	00:05:40

Average Jobs Size	System Type	Number of Jobs	Average Waiting Time (HH:MM:SS)	Average Execution Time (HH:MM:SS)	Total Response Time (HH:MM:SS)
	Grid trees)	1040	00:40:43	00:01:18	00:04:11
		1560	01:15:00	00:00:55	00:11:30
		3000	01:35:00	00:01:01	00:15:41
		10000	00:01:43	00:00:48	00:25:15
312.7 MB	P2P	520	00:10:52	00:12:12	03:44:50
		1040	00:18:33	00:12:46	04:38:40
		1560	00:27:50	00:13:04	04:59:39
		3000	00:34:59	00:11:21	05:38:16
		10000	00:35:59	00:12:21	12:31:55
	Hierarchal (1 Grid tree)	520	00:02:33	00:10:13	00:56:15
		1040	00:06:00	00:11:25	01:13:58
		1560	00:10:39	00:11:13	01:07:42
		3000	00:18:40	00:10:34	01:44:04
		10000	00:18:45	00:11:00	04:25:07
	Hybrid (4 Grid trees)	520	00:00:41	00:16:49	00:56:56
		1040	00:04:38	00:12:04	00:58:33
		1560	00:09:12	00:10:40	00:58:37
		3000	00:16:56	00:10:37	01:45:02
		10000	00:17:22	00:10:59	04:22:57
	Hybrid (16 Grid trees)	520	00:03:04	00:13:04	00:54:56
		1040	00:06:55	00:10:52	01:01:06
		1560	00:14:19	00:10:15	01:48:43
		3000	00:18:29	00:09:28	02:30:07
		10000	00:18:15	00:08:37	03:57:45
1 GB	P2P	520	00:40:01	00:44:07	11:35:14
		1040	01:42:40	00:46:14	23:34:53
		1560	01:23:18	00:47:18	24:16:33
		3000	02:06:44	00:48:11	22:48:48
		10000	02:15:30	00:45:15	41:45:25
	Hierarchal (1 Grid tree)	520	00:19:28	00:43:58	02:51:52
		1040	00:20:12	00:38:58	04:22:49
		1560	00:33:18	00:36:12	04:38:28

Average Jobs Size	System Type	Number of Jobs	Average Waiting Time (HH:MM:SS)	Average Execution Time (HH:MM:SS)	Total Response Time (HH:MM:SS)
		3000	01:02:17	00:36:12	08:10:19
		10000	01:14:55	00:34:35	15:14:44
	Hybrid (4 Grid trees)	520	00:19:28	00:43:58	02:51:52
		1040	00:20:12	00:38:58	04:22:49
		1560	00:33:18	00:36:12	04:38:28
		3000	01:02:17	00:36:12	08:10:19
		10000	01:14:55	00:34:35	15:14:44
	Hybrid (16 Grid trees)	520	00:02:47	00:43:59	02:57:48
		1040	00:16:22	00:40:17	03:09:54
		1560	00:30:40	00:35:12	03:16:04
		3000	00:54:32	00:34:46	05:22:03
		10000	01:08:17	00:35:15	13:50:57
10 GB	P2P	520	05:25:28	07:30:32	101:38:36
		1040	09:45:07	07:54:40	175:14:56
		1560	16:53:23	08:07:24	176:54:27
		3000	17:26:23	06:32:11	190:44:49
		10000	18:13:11	08:15:25	410:30:30
	Hierarchal (1 Grid tree)	520	02:22:43	06:46:04	27:53:24
		1040	03:10:45	06:22:13	28:02:44
		1560	05:33:00	06:04:34	48:23:54
		3000	09:13:50	05:56:49	53:35:54
		10000	08:45:45	03:07:26	146:42:00
	Hybrid (4 Grid trees)	520	01:35:18	06:27:38	27:29:55
		1040	04:22:52	05:37:17	51:43:18
		1560	05:22:39	06:07:35	33:01:31
		3000	02:40:38	01:41:10	20:24:40
		10000	03:10:16	03:01:27	155:44:46
	Hybrid (16 Grid trees)	520	01:53:30	06:34:41	27:30:13
		1040	04:55:11	05:56:54	57:33:15
		1560	07:16:33	05:20:17	55:01:34
		3000	03:08:39	01:33:19	85:29:41
		10000	03:15:51	2:04:40	150:08:11

Average Jobs Size	System Type	Number of Jobs	Average Waiting Time (HH:MM:SS)	Average Execution Time (HH:MM:SS)	Total Response Time (HH:MM:SS)
100 GB	P2P	520	75:49:20	90:58:56	1403:03:27
		1040	92:19:51	81:36:34	1503:08:27
		1560	125:01:00	85:25:49	1538:53:24
		3000	215:29:40	88:33:15	2036:30:25
		10000	221:17:45	89:09:30	2818:11:11
	Hierarchal (1 Grid tree)	520	09:13:29	69:23:21	275:37:39
		1040	35:06:13	66:48:48	407:07:50
		1560	56:31:55	62:42:03	401:09:46
		3000	101:21:54	61:29:54	725:21:37
		10000	99:55:56	59:00:53	1332:30:35
	Hybrid (4 Grid trees)	520	02:23:05	80:20:05	275:37:39
		1040	28:24:10	68:37:08	276:58:56
		1560	49:38:07	56:02:57	295:34:23
		3000	94:33:02	57:23:46	696:11:00
		10000	94:43:05	57:36:38	1295:56:24
	Hybrid (16 Grid trees)	520	07:21:54	75:21:54	459:25:59
		1040	43:25:48	56:14:17	513:24:44
		1560	71:00:13	55:36:47	519:37:09
		3000	98:54:55	49:18:19	937:01:22
		10000	99:30:40	56:05:10	1390:30:25

Table 4: Results Set in Experiment 1

A.2 SECOND EXPERIMENT RESULTS SET

Average Jobs Size	System Type	Number of Jobs	Average Waiting Time (HH:MM:SS)	Average Execution Time (HH:MM:SS)
28.9 MB	P2P	520	00:00:05	00:00:53
		1040	00:00:14	00:00:56
		1560	00:00:25	00:00:50
		3000	00:00:26	00:00:55
	Hierarchal (1 Grid tree)	520	71ms	00:00:53
		1040	100ms	00:00:55
		1560	113ms	00:00:50
		3000	120ms	00:00:54
	Hybrid (4 Grid trees)	520	56ms	00:00:39
		1040	142ms	00:00:42
		1560	201ms	00:00:40
		3000	149ms	00:00:50
	Hybrid (16 Grid trees)	520	115ms	00:00:48
		1040	111ms	00:00:48
		1560	201ms	00:00:48
		3000	401ms	00:00:51
312.7 MB	P2P	520	00:07:42	00:14:55
		1040	00:09:40	00:14:40
		1560	00:011:37	00:17:26
		3000	00:12:30	00:17:45
	Hierarchal (1 Grid tree)	520	201ms	00:09:35
		1040	191ms	00:10:33
		1560	250ms	00:10:44
		3000	345ms	00:08:44
	Hybrid (4 Grid trees)	520	175ms	00:09:22
		1040	331ms	00:09:33
		1560	341ms	00:10:48

Average Jobs Size	System Type	Number of Jobs	Average Waiting Time (HH:MM:SS)	Average Execution Time (HH:MM:SS)
		3000	389ms	00:07:58
	Hybrid (16 Grid trees)	520	337ms	00:08:39
		1040	254ms	00:09:37
		1560	345ms	00:10:46
		3000	401ms	00:08:44
1 GB	P2P	520	00:30:58	00:49:54
		1040	00:32:10	00:40:29
		1560	00:38:54	00:45:31
		3000	00:37:24	00:39:56
	Hierarchal (1 Grid tree)	520	99ms	00:38:55
		1040	200ms	00:29:11
		1560	149ms	00:30:57
		3000	209ms	00:31:15
	Hybrid (4 Grid trees)	520	156ms	00:38:32
		1040	301ms	00:31:42
		1560	435ms	00:33:17
		3000	167ms	00:32:18
	Hybrid (16 Grid trees)	520	489ms	00:31:48
		1040	423ms	00:29:44
		1560	469ms	00:26:29
		3000	154ms	00:25:05
10 GB	P2P	520	05:54:04	07:20:25
		1040	07:26:24	06:52:49
		1560	07:13:06	06:10:48
		3000	09:38:02	06:15:51
	Hierarchal (1 Grid tree)	520	00:02:55	06:47:38
		1040	00:29:54	04:55:11
		1560	01:22:40	04:28:55
		3000	02:14:26	04:17:22
	Hybrid (4 Grid trees)	520	00:03:05	06:22:11
		1040	00:39:45	04:53:25
		1560	01:33:35	04:29:17

Average Jobs Size	System Type	Number of Jobs	Average Waiting Time (HH:MM:SS)	Average Execution Time (HH:MM:SS)
		3000	01:39:11	03:58:05
	Hybrid (16 Grid trees)	520	00:01:29	06:25:11
		1040	00:45:45	04:28:22
		1560	01:46:49	04:18:15
		3000	02:31:29	04:15:32
100 GB	P2P	520	74:45:55	86:05:39
		1040	107:18:07	78:15:02
		1560	124:28:49	79:37:31
		3000	130:43:04	77:27:59
	Hierarchal (1 Grid tree)	520	01:37:07	75:57:48
		1040	25:34:59	60:45:56
		1560	34:20:16	56:56:22
		3000	63:09:38	56:01:59
	Hybrid (4 Grid trees)	520	04:45:00	63:15:00
		1040	26:25:15	58:11:00
		1560	42:57:19	59:49:03
		3000	63:07:32	56:21:00
	Hybrid (16 Grid trees)	520	06:55:18	60:00:11
		1040	28:43:56	52:12:09
		1560	44:59:54	48:04:09
		3000	55:33:50	41:07:24

Table 5: Results Set in Experiment 2

A.3 THIRD EXPERIMENT RESULTS SET

Job's Arrival Rate	System	Number of Jobs	Average Waiting Time (HH:MM:SS)	Average Execution Time (HH:MM:SS)
1 hour (size = 10 GB)	P2P	520	04:55:44	07:05:44
		1040	11:17:06	08:25:13
		1560	13:00:39	07:56:15
		3000	14:46:12	06:55:44
	Hierarchal (1 Grid tree)	520	00:04:43	06:39:50
		1040	03:44:54	07:37:04
		1560	06:51:51	06:57:33
		3000	09:21:10	06:30:26
1 day (size = 10 GB)	P2P	520	06:51:16	07:00:27
		1040	09:06:55	06:17:42
		1560	12:41:30	06:44:59
		3000	13:26:42	06:43:31
	Hierarchal (1 Grid tree)	520	00:02:07	06:49:39
		1040	03:20:27	05:57:58
		1560	06:05:51	05:48:38
		3000	04:06:59	05:56:40
1 Week (size = 10 GB	P2P	520	05:43:07	06:45:47
		1040	09:33:57	06:52:17
		1560	12:29:25	07:06:02
		3000	14:30:15	06:03:07
	Hierarchal (1 Grid tree)	520	00:01:36	06:46:25
		1040	02:39:10	06:47:04
		1560	05:10:14	07:01:36
		3000	03:26:35	06:12:22

Table 6: Results Set in Experiment 3

A.4 FOURTH EXPERIMENT RESULTS SET

Used Scenario (Job Size = 100GB)	Enabled/Disabled Algorithms (Hierarchal System)	Number of Jobs	Average Waiting Time HH:MM:SS	Average Execution Time HH:MM:SS	Total Response Time HH:MM:SS
Experiment 1	Enabled	520	09:13:29	69:23:21	275:37:39
		1040	35:06:13	66:48:48	407:07:50
		1560	56:31:55	62:42:03	401:09:46
		3000	101:21:54	61:29:54	725:21:37
		10000	99:55:56	59:00:53	1332:30:35
	Disabled	520	38:35:11	83:55:56	314:03:13
		1040	44:59:31	80:05:28	511:23:33
		1560	119:40:01	79:28:10	958:32:10
		3000	198:55:05	83:27:02	1595:18:48
		10000	199:51:25	88:59:35	2116:17:19
Experiment 2	Enabled	520	01:37:07	75:57:48	277:25:42
		1040	25:34:59	60:45:56	278:05:03
		1560	34:20:16	56:56:22	233:10:07
		3000	63:09:38	56:01:59	552:49:46
	Disabled	520	01:38:11	93:38:14	278:20:43
		1040	47:19:52	90:18:36	549:31:54
		1560	69:02:26	86:48:16	549:30:54
		3000	193:51:24	88:56:32	1127:12:06

Table 7: Results Set in Experiment 4

A.5 FIFTH EXPERIMENT RESULTS SET

Used Scenario (Job Size = 100GB)	Resources Behavior (Hierarchal System)	Num of Jobs	Average Waiting Time HH:MM:SS	Average Execution Time HH:MM:SS	Total Response Time HH:MM:SS	Saved Jobs (%)
Exp1	Changing Resources	520	13:13:59	69:13:14	276:16:15	5.38%
		1040	34:31:32	63:12:18	477:03:22	7.02%
		1560	55:05:45	60:19:36	399:06:56	13.21%
		3000	96:54:30	60:01:05	800:06:58	6.40%
	Constant Resources	520	09:13:29	69:23:21	275:37:39	NA
		1040	35:06:13	66:48:48	407:07:50	NA
		1560	56:31:55	62:42:03	401:09:46	NA
		3000	101:21:54	61:29:54	725:21:37	NA
Exp2	Changing Resources	520	01:42:04	76:05:15	278:46:04	0.38%
		1040	27:42:42	60:47:23	304:54:01	2.5%
		1560	39:57:23	56:57:59	333:25:21	3.65%
		3000	54:53:32	56:32:38	131:26:10	2.93%
	Constant Resources	520	01:37:07	75:57:48	277:25:42	NA
		1040	25:34:59	60:45:56	278:05:03	NA
		1560	34:20:16	56:56:22	233:10:07	NA
		3000	63:09:38	56:01:59	552:49:46	NA

Table 8: Results Set in Experiment 5