Using Swarm Intelligence for Distributed Job Scheduling on the Grid

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Azin Moallem

©Azin Moallem, 03/2009. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

With the rapid growth of data and computational needs, distributed systems and computational Grids are gaining more and more attention. Grids are playing an important and growing role in today networks. The huge amount of computations a Grid can fulfill in a specific time cannot be done by the best super computers. However, Grid performance can still be improved by making sure all the resources available in the Grid are utilized by a good load balancing algorithm. The purpose of such algorithms is to make sure all nodes are equally involved in Grid computations. This research proposes two new distributed swarm intelligence inspired load balancing algorithms. One is based on ant colony optimization and is called AntZ, the other one is based on particle swarm optimization and is called ParticleZ. Distributed load balancing does not incorporate a single point of failure in the system. In the AntZ algorithm, an ant is invoked in response to submitting a job to the Grid and this ant surfs the network to find the best resource to deliver the job to. In the ParticleZ algorithm, each node plays a role as a particle and moves toward other particles by sharing its workload among them. We will be simulating our proposed approaches using a Grid simulation toolkit (GridSim) dedicated to Grid simulations. The performance of the algorithms will be evaluated using several performance criteria (e.g. makespan and load balancing level). A comparison of our proposed approaches with a classical approach called State Broadcast Algorithm and two random approaches will also be provided. Experimental results show the proposed algorithms (AntZ and ParticleZ) can perform very well in a Grid environment. In particular, the use of particle swarm optimization, which has not been addressed in the literature, can yield better performance results in many scenarios than the ant colony approach.

Acknowledgements

First and foremost I would like to express my deepest gratitude to my supervisor, Dr. Simone A. Ludwig, whose expertise, understanding and advice were a great help by leading me to this stage in my graduate studies.

I would also like to thank Dr. Ian McQuillan, Dr. Michael Horsch and Dr. Chris Zhang for being my committee members and for their suggestions on this research that guided me towards being a better researcher and helped me to improve the quality of this work.

I am grateful to all the faculty and staff at the University of Saskatchewan who provided such a nice atmosphere for the students to pursue their studies.

Finally, I want to send my deepest gratitude to my lovely parents whose support was always there for me through all difficulties. They helped me to get to this stage in my life. I have no means to compensate what they have given me.

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents iv

List of Tables vi

List of Figures vii

1 Introduction 1
- 1.1 Introduction to the Grid . . . 1
  - 1.1.1 History of the Grid . . . 3
  - 1.1.2 Grid Architecture . . . 4
- 1.2 Scheduling Jobs in Computational Grids (Resource Management) . . . 9
- 1.3 Load Balancing Problem . . . 11
- 1.4 Swarm Intelligence Techniques for Load Balancing . . . 15
  - 1.4.1 Social Insect Systems - Ant Colony . . . 15
  - 1.4.2 Particle Swarm Optimization . . . 16
- 1.5 Problem Statement . . . 17
- 1.6 Thesis Organization . . . 17

2 Related Work 18
- 2.1 Overview . . . 18
- 2.2 Centralized Load Balancing Approaches . . . 18
  - 2.2.1 Classical Approaches . . . 18
  - 2.2.2 Agent-Based Approaches . . . 19
  - 2.2.3 Artificial Life Approaches . . . 21
- 2.3 Decentralized Load Balancing Approaches . . . 22
  - 2.3.1 Classical Approaches . . . 22
  - 2.3.2 Ant Colony Approaches . . . 23
- 2.4 Job Migration . . . 25
- 2.5 Summary . . . 26

3 Proposed Load Balancing Approaches 27
- 3.1 Overview . . . 27
- 3.2 Contributions and Benefits . . . 27
- 3.3 Ant Colony Optimization . . . 30
- 3.4 Particle Swarm Optimization . . . 33
- 3.5 Ant Colony Load Balancing: AntZ . . . 36
- 3.6 Particle Swarm Optimization: ParticleZ . . . 41
- 3.7 Summary . . . 45

4 Experimental Design and Implementation **46**
4.1 Overview . . . 46
4.2 Prototype and Initial Results . . . 46
4.3 GridSim Toolkit . . . 48
4.3.1 GridSim Architecture . . . 52
4.4 General Design . . . 54
4.5 AntZ Design and Implementation . . . 56
4.6 ParticleZ Design and Implementation . . . 58
4.7 Summary . . . 61

5 Experimental Results **62**
5.1 Overview . . . 62
5.2 System Model . . . 62
5.3 Application Model . . . 63
5.4 Performance Evaluation Criteria . . . 64
5.4.1 Makespan . . . 64
5.4.2 Load . . . 64
5.4.3 Standard Deviation . . . 64
5.4.4 Load Balancing Level of the System . . . 65
5.5 Comparison Against Classical Approaches . . . 65
5.6 Experimental Results . . . 66
5.6.1 AntZ Parametric Measurement Effects . . . 73
5.6.2 ParticleZ Parametric Measurement Effects . . . 75
5.7 Summary . . . 77

6 Conclusion and Future Work **78**
6.1 Conclusion . . . 78
6.2 Future Work . . . 81

References **84**

A Source code **89**

List of Tables

3.1 Load table information in nodes 40

5.1 Grid resource characteristics 67
5.2 Scheduling parameters and their values 67
5.3 Gridlet Characteristics 67
5.4 Average load balancing level of the system for different algorithms 70
5.5 Predicting execution time based on number of jobs 71

List of Figures

1.1 Proposed service architecture and service level at LCG Tiers [1] 3
1.2 Evolution of the Grid technology 4
1.3 Virtual organizations 5
1.4 A generic view of the Grid 6
1.5 Globus toolkit modules 8
1.6 Categorization of load balancing algorithms 12
1.7 Centralized load balancing model 12
1.8 Decentralized load balancing model 13
1.9 Hierarchical load balancing model 14

3.1 A. Ants in a pheromone trail between nest and food; B. an obstacle interrupts the trail; C. ants find two paths to go around the obstacle; D. a new pheromone trail is formed along the shorter path [2]. 31
3.2 The flow chart of an ant behavior capable of clustering objects 33
3.3 Overview of the AntZ system 37
3.4 Different phases of the AntZ algorithm 38
3.5 Overview of the ParticleZ system 42
3.6 Different phases of the ParticleZ algorithm 43

4.1 UML class diagram of the prototype design 47
4.2 A modular architecture for GridSim platform and components 50
4.3 A flow diagram in GridSim based simulations [3] 51
4.4 The UML diagram of the GridSim package design [3] 53
4.5 UML class diagram of the design 55
4.6 UML class diagram for the Ant Colony scheduling 57
4.7 UML sequence diagram for the Ant Colony scheduling 58
4.8 UML class diagram for the Particle Swarm scheduling 59
4.9 UML sequence diagram for the Particle Swarm scheduling 60
4.10 UML sequence diagram for the Particle Swarm scheduling 60

5.1 Comparing the makespan of different approaches 68
5.2 Simulation time related to each algorithm in milliseconds 69
5.3 Communication overhead related to each algorithm 69
5.4 Effect of the increase in number of jobs on performance of the algorithms . 71
5.5 Effect of the increase in job length on performance of the algorithms 72
5.6 Effect of increasing number of resources on execution time 73
5.7 Effect of single and random injection points on the performance of the algorithms 74
5.8 Effect of the change in wandering steps on AntZ makespan 75
5.9 Effect of the change in wandering steps on AntZ communication number . . 75
5.10 Effect of decay rate on AntZ makespan 76
5.11 Effect of link number on ParticleZ makespan 76
5.12 Effect of link number on ParticleZ communication number 77

Chapter 1

Introduction

1.1 Introduction to the Grid

The term "the Grid" was coined in the mid-1990s to denote a (then) proposed distributed computing infrastructure for advanced science and engineering. Much progress has since been made on the construction of such an infrastructure and on its extension and application to commercial computing problems. Early definitions for the Grid go back to 1998, when Carl Kesselman and Ian Foster defined the Grid as follows [4]:

"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities."

Putting it in simple words, Grid computing aims to connect geographically distributed computers allowing their computational power and storage capabilities to be shared.

With the rapid growth of data and computational needs, distributed systems and Grids are gaining more attention to solve the problem of large-scale computing [5]. There are several options for establishing distributed systems, and Grid Systems [4] are one of the common ones for distributed applications [5].

Various Grid application scenarios have been explored in both academia and industry. We present a brief description of some of these applications, however, a more detailed description of each can be found in [6].

- **Distributed Aircraft Engine Diagnostics**. The U.K. Distributed Aircraft Maintenance Environment (DAME) project is using Grid technologies for the challenging and important problem of computer-based fault diagnosis. The problem can be considered an inherently distributed problem because of the huge amount of data sources and stakeholders involved.

- **NEES Grid Earthquake Engineering Collaboratory**. The U.S. Network for Earthquake Engineering Simulation (NEES) is a project which enables remote access to the specialized equipment used to study the behavior of different structures (example: bridge columns) when subjected to the forces of an earthquake.

- **World Wide Telescope**. Advances in digital astronomy enable the systematic survey and the collection of vast amounts of data from telescopes gathered all over the world.

- **Biomedical Informatics Research Network**. The goal of this U.S. project is to bring together biomedical imaging data to be used for research and, hence, to improve clinical cases.

- **Virtual Screening on Desktop Computers**. A drug discovery application in which, an intra-Grid composed of desktop PCs is used for virtual screening of drug candidates.

- **Infrastructure for Multiplayer Games**. Butterfly.net is a service provider for the multiplayer videogaming industry. It uses Grid technologies to deliver scalable services to game developers.

Another recent example of an application of the Grid in real world systems is the application of a large Grid system for the Large Hadron Collider (LHC) at CERN[1]. This scientific experiment intends to answer questions about the Big Bang and the building blocks of our world by simulating collisions between protons on a small scale. This Grid has a three-tier achitecture. The first layer is located at CERN and is considered the origin of the data. The second layer is composed of eleven data centers in Europe, North America and Asia. Third-tier data centers are located world wide in 250 universities in which the analysis of the received data takes place[2]. Figure 1.1, shows LCG (LHC Computing Grid) tiers architecture from a service level view. More information about this project can be found in the technical design report in [1].

The next section reviews the history of the Grid from its emergence until now and its future trends.

[1]The European Organization for Nuclear Research
[2]http://www.irdanesh.com/1387/06/25/grid-cern-lhc/

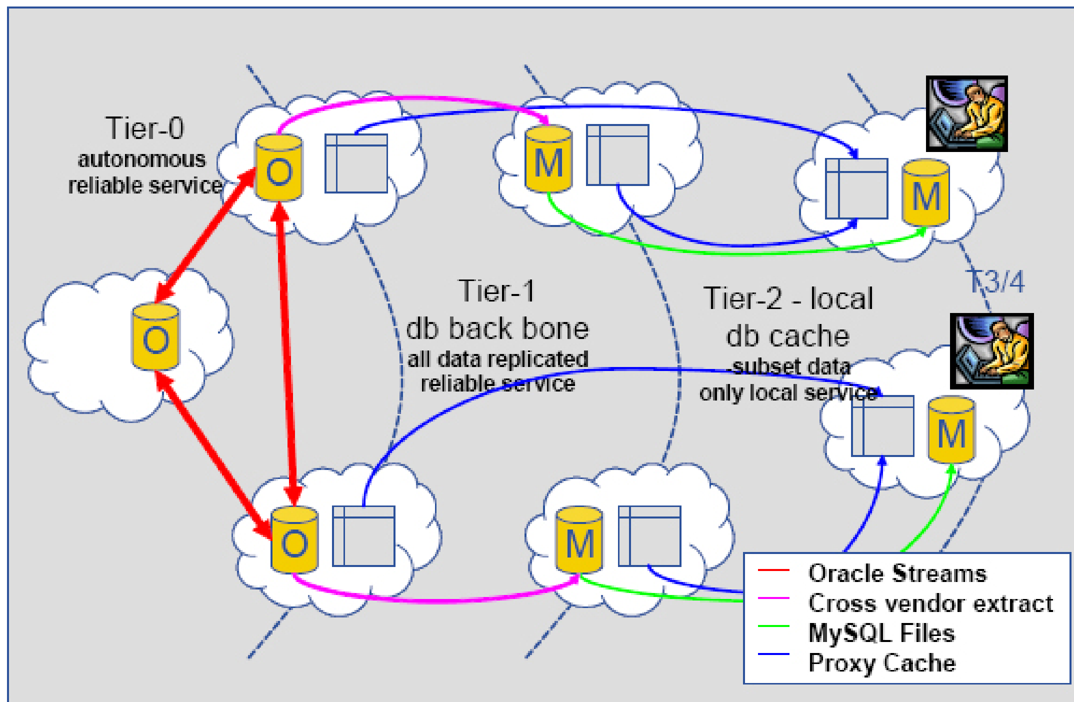


Figure 1.1: Proposed service architecture and service level at LCG Tiers [1]

1.1.1 History of the Grid

The transition from metacomputing (computing of computing; organization of large computer networks before emergence of the Grid) to Grid computing took place in the mid-1990s with the introduction of middleware designed to function as a wide-area infrastructure to support diverse online processing and data-intensive applications. Systems such as the Storage Resource Broker [7], Globus Toolkit [8], Condor [9][10], and Legion [11][12] were developed primarily for scientific applications.

The evolution of the Grid technology is well shown in Figure 1.2 [6]. As illustrated in the figure, early experiments for the Grid worked with custom tools or specialized middleware that focused on message-oriented communication between computing nodes. By 1998, the open source Globus Toolkit (GT2) [8] had emerged as a standard software infrastructure for Grid computing. As the interest in Grids continued to grow, and in particular as industrial interest emerged, the importance of true standardization increased. The Global Grid Forum, established in 1998 as an international community and standards organization,

worked out to be the natural place for such standards to be developed, and indeed multiple standardization activities are currently under way. In particular, 2002 saw the emergence of the Open Grid Services Architecture (OGSA), which is a community standard with multiple implementations including the OGSA-based Globus Toolkit 3.0, released in 2003. The next section provides introductory discussions about the architecture of the Grid followed by more details on OGSA.

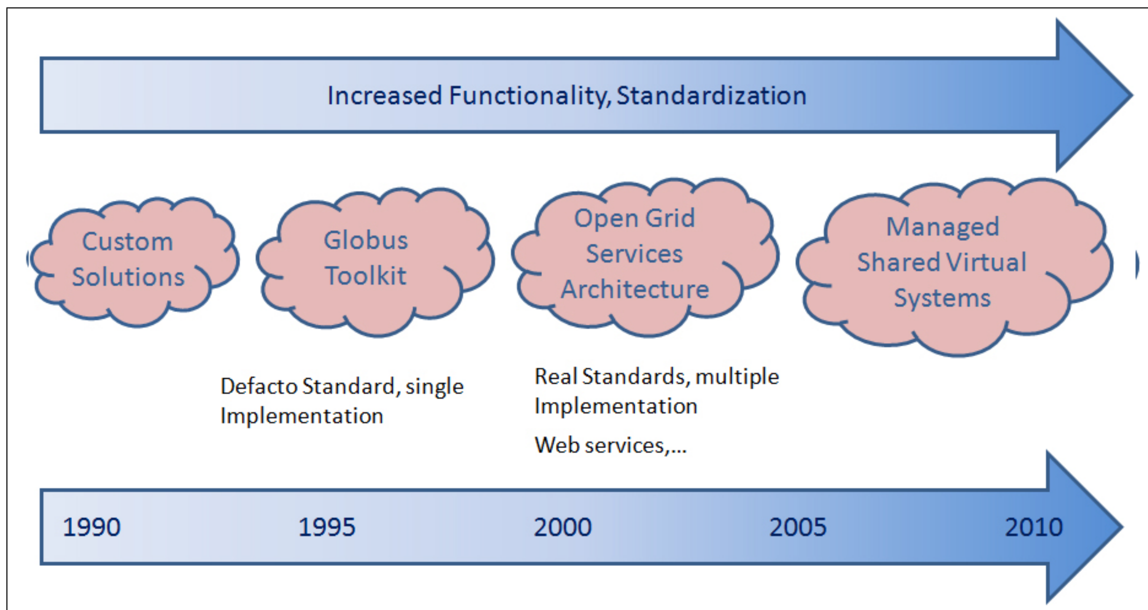


Figure 1.2: Evolution of the Grid technology

1.1.2 Grid Architecture

The main concern underlying the Grid is *coordinated resource sharing and problem solving in dynamic, multi-institutional, virtual organizations*. This sharing, of course, should be controlled by both resource providers and consumers by defining clearly what is shared and what are the conditions under which sharing occurs. A set of individuals and institutions defined by this sharing rules form Virtual Organizations (VOs). Thus, an actual organization can be part of one or more VOs by sharing some of its resources [13]. Figure 1.3 shows three actual organizations with both computational and data resources to share, and two virtual organizations (VO-A and VO-B) each of which can have access to a subset of resources in each of the organizations.

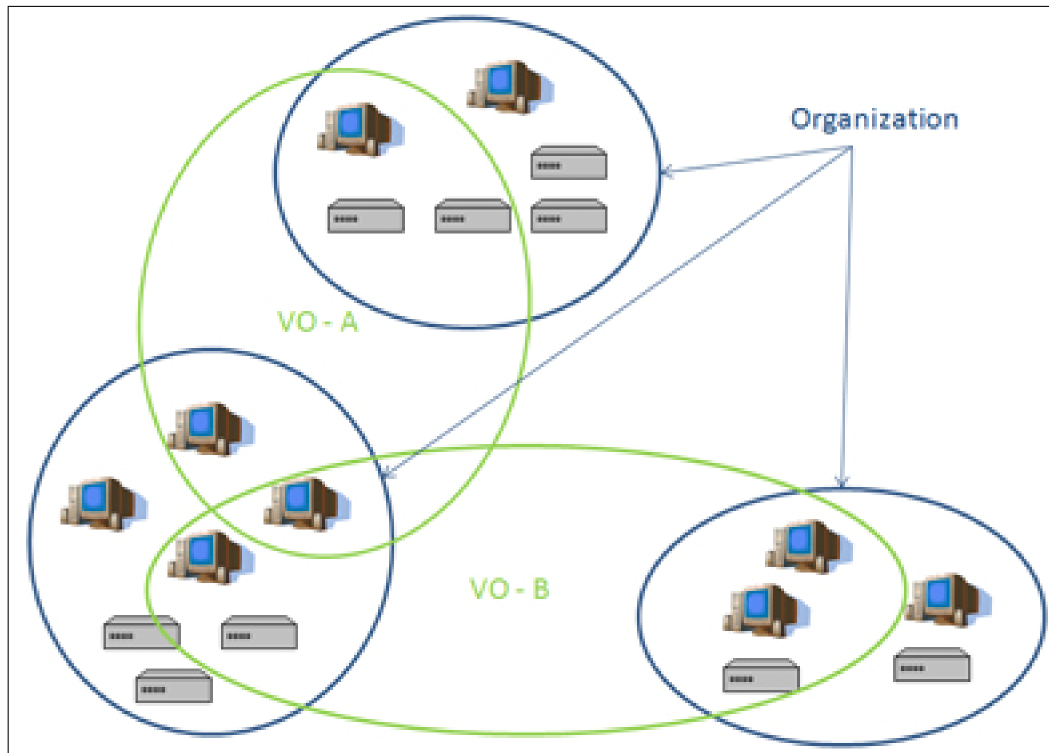


Figure 1.3: Virtual organizations

Historically, the architecture of a Grid was often described in terms of *layers*, each providing a specific function. Figure 1.4 depicts a layered architecture of a typical Grid.[1]

The **Network Layer** provides the connectivity between the resources in the Grid.

The **Resource Layer** contains all the resources that are part of the Grid, such as computers, storage systems, clusters and specialized resources such as sensors.

The **Middleware Layer** provides the tools so that the lower layers can participate in a unified Grid environment.

The **Application and Serviceware Layer** includes all applications that use the resources of the Grid to fulfill their mission. It is also called the Serviceware Layer because it includes all common services that represent mostly application-specific management functions such as billing, time logging, and others.

With the emergence of new requirements and web services, the need for standardizing a service oriented architecture arose. OGSA, appeared to address key concerns in Grid

[1]http://www.sei.cmu.edu/isis/guide/engineering/architectures.htm

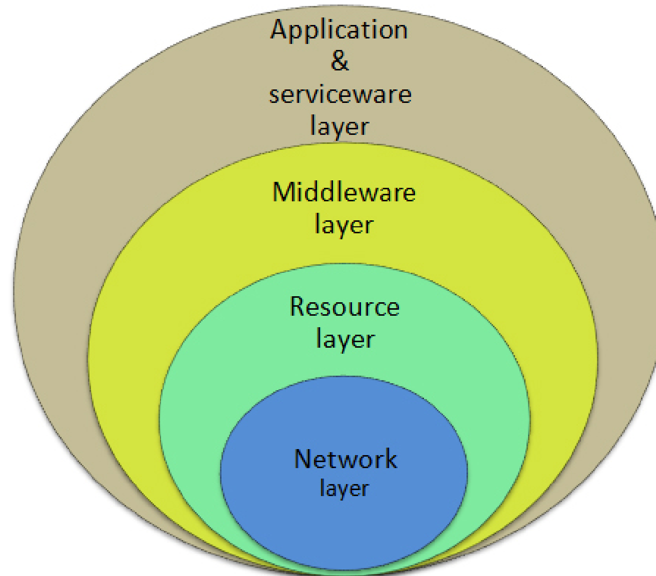


Figure 1.4: A generic view of the Grid

systems by defining a set of capabilities and behaviours. OGSA, is the specification for standards-based Grid computing and is centred on stateful web services. OGSA deals with the middleware layer depicted in Figure 1.4 in a service-oriented architecture. OGSA addresses issues about services and their interfaces, the individual and collective state of resources belonging to these services, and the interaction between these services [14]. An OGSA Grid can be described in terms of the following capabilities [14]:

- **Infrastructure services.** OGSA architecture wants to insure that the web service infrastructure follows specific guidelines such as standards defined by WSDL (Web Service Description Language), its naming policies, security and so forth.
- **Execution Management services.** OGSA-EMS are concerned with the problems of instantiation, management and completion of the units of work. Examples of units of work may include either OGSA applications or legacy (non-OGSA) applications (a database server, a servlet running in a Java application server container, etc.). EMS services can be divided into three classes:
  - **Resources** that model processing, storage, executables, resource management and provisioning.

  - **Job management** is concerned about handling jobs.
  - **Resource selection services** that collectively decide where to execute a unit of work.

- **Data services.** Data services are related to these OGSA services concerned with the management, access to and update of data resources, along with the transfer of data between resources. These are collectively called data services.

- **Resource Management services.** Resource management performs several forms of management on resources in a Grid. In an OGSA Grid there are three types of management related to resources:
  - Management of the physical and logical resources themselves (e.g., rebooting a host, or setting VLANs on a network switch).
  - Management of the OGSA Grid resources exposed through service interfaces (e.g., resource reservation, job submission and monitoring).
  - Management of the OGSA Grid infrastructure, exposed through its management interfaces (e.g., monitoring a registry service).

- **Security services.** OGSA security services facilitate the enforcement of the security-related policy within a (virtual) organization. In general, the purpose of the enforcement of security policy is to ensure that the higher-level business objectives can be met.

- **Self-management services.** In a self-managing environment, system components, including hardware components such as computers, networks and storage devices, and software components such as operating systems and business applications, are self-configuring, self-healing and self-optimizing.

- **Information services.** An information service needs to support a variety of Quality of Service (QoS) requirements for reliability, security, and performance.

One thing worth mentioning is that the entire set of OGSA capabilities which are introduced here does not have to be present in a Grid environment, only a subset of these capabilities may suffice. For more information about the standards and conventions that should be followed in an OGSA environment the reader is referred to [14].

Open Grid Services Infrastructure (OGSI) is the concrete specification of the OGSA infrastructure. It is the middleware for Grid services. OGSI defines how to build a Grid service and it defines the mechanisms for creating, managing, and exchanging information for Grid services [15].

Using the standards above the Globus toolkit is developed by the Globus Alliance[1]. The Globus toolkit is the most common toolkit and it is an implementation of the OGSA framework described earlier. The Globus toolkit includes software modules for Security, Data management, Execution management, Information Services, Fault detection, etc.

Figure 1.5, shows the Globus toolkit modules and a brief description about each module is provided [8].

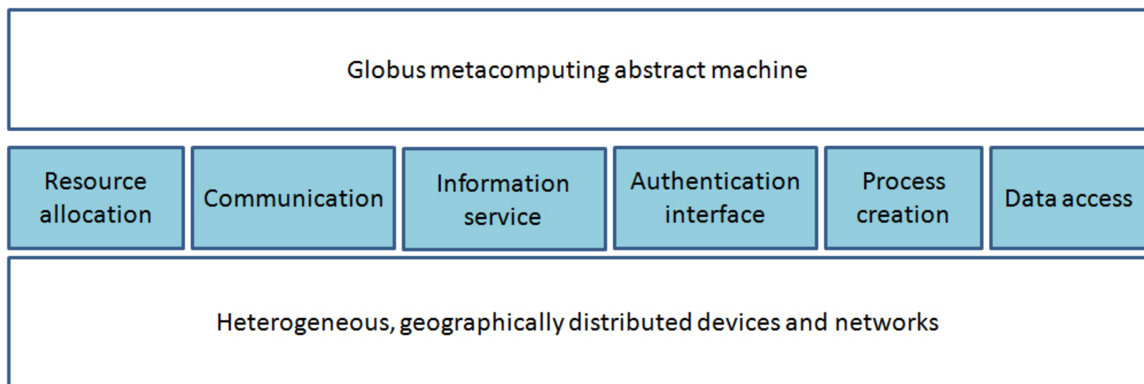


Figure 1.5: Globus toolkit modules

- **Resource location and allocation.** This component has the responsibility to express application resource requirements, and to identify resources that meet these requirements and schedule resources. Resource allocation involves scheduling the resources and performing any initialization required for subsequent process creation, data access, etc.

- **Communications.** This component provides basic communication mechanisms. These mechanisms must permit the efficient implementation of a wide range of communication methods including message passing and remote procedure call (RPC).

- **Unified resource information service.** This component provides the toolkit

[1]http://www.globus.org/alliance/

with the ability to obtain real-time information about metasystem structure and status. The mechanism must allow components to send as well as receive information. Support for scoping and access control is also required.

- **Authentication interface.** This component provides basic authentication mechanisms that can be used to validate the identity of both users and resources. These mechanisms will be used for other security services such as authorization and data security that need to know the identity of parties involved in an operation.

- **Process creation.** This component initiates computation on a resource when it has been located and allocated. The responsibilities of this component can be stated as follows: setting up executables, creating an execution environment, starting an executable, passing arguments, integrating the new process within the overall computation and managing termination and shutdown.

- **Data access.** This component is responsible for providing high-speed remote access to persistent storage such as files. Some data resources such as databases may be accessed via distributed database technology or the Common Object Request Broker Architecture (CORBA). The Globus data access module addresses the problem of achieving high performance when accessing parallel file systems and network-enabled I/O devices such as the High Performance Storage System (HPSS).

Having described all components in a Grid environment we focus now on the resource allocation component and provide further details.

1.2 Scheduling Jobs in Computational Grids (Resource Management)

The resource management system is the central component of a Grid system. Its basic responsibilities are to accept requests from users, match user requests to available resources for which the user has permission to use and schedule the matched resources [16]. Workload and resource management are two essential functions provided at the service level of the Grid software infrastructure [17]. To be able to fully benefit from such Grid systems, resource management and scheduling are key Grid services, where issues of task allocation

and load balancing represent a common challenge for most Grids [18]. In a computational Grid, at a given time, the task is to allocate the user defined jobs efficiently both by meeting the deadlines and making use of all the available resources [19]. In a Grid system, resources are added and removed dynamically. Different types of applications with different resource requirements are being executed. Resource owners set their own resource usage policies and costs. This necessitates a need for extra decision making policies between resource users and resource providers to meet the quality of service constraints [16].

Grid systems are classified into two categories: **compute** and **data** Grids. In compute Grids the main resource that is being managed by the resource management system is compute cycles (i.e. processors), while in data Grids the focus is to manage data distributed over geographical locations. The architecture and the services provided by the resource management system are affected by the type of Grid system it is deployed in. Resources which are to be managed could be hardware (computation cycle, network bandwidth and data stores) or software resources (applications) [16].

In traditional computing systems, resource management is a well-studied problem. Resource managers such as batch schedulers, workflow engines, and operating systems exist for many computing environments. These resource management systems are designed to work under the assumption that they have complete control of a resource and thus can implement the mechanisms and policies needed for the effective use of that resource. Unfortunately, this assumption does not apply to the Grid. When dealing with the Grid we must develop methods for managing Grid resources across separately administered domains, with the resource heterogeneity, loss of absolute control, and inevitable differences in policy that is the result of heterogeneity. The underlying Grid resource set is typically heterogeneous [6].

The term "load balancing" refers to the technique that tries to distribute work load between several computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, throughput, or response. The load balancing mechanism aims to equally spread the load on each computing node, maximizing their utilization and minimizing the total task execution time. In order to achieve these goals, the load balancing mechanism should be "fair" in distributing the load across the computing nodes; by being fair we mean that the difference between the "heaviest-loaded" node and the "lightest-loaded" node should be minimized [20].

1.3 Load Balancing Problem

Load balancing has always been an issue since the emergence of distributed systems. In a distributed system there might be scenarios in which a task waits for a service at the queue of one resource, while at the same time another resource which is capable of serving the task is idle. The purpose of a load balancing algorithm is to prevent these scenarios as much as possible [21].

For parallel applications, load balancing attempts to distribute the computation load across multiple processors or machines as evenly as possible with the objective to improve performance. Generally, a load balancing scheme consists of three phases: *information collection, decision making and data migration*. During the *information collection* phase, the load balancer gathers the information of the distribution of workload and the state of computing environment and detects whether there is a load imbalance. The *decision making* phase focuses on calculating an optimal data distribution, while the *data migration* phase transfers the excess amount of workload from one overloaded processor to another underloaded one [22].

Load balancing algorithms can be classified into sub categories from various perspectives. From one view point, they can be divided into *static*, *dynamic* or *adaptive* algorithms. In static algorithms, the decisions related to balancing the load are made at compile time. This means these decision are made when resource requirements are estimated [23]. On the other hand, a load balancer with dynamic load balancing allocates/reallocates resources at runtime and uses the system-state information to make its decisions. Adaptive load balancing algorithms are a special class of dynamic algorithms. They adapt their activities by dynamically changing their parameters, or even their policies, to suit the changing system state [24].

From another point of view, methods used in load balancing can be divided into three classes, i.e., *centralized, distributed (decentralized) and hierarchical* [16] as shown in Figure 1.6.

In a centralized approach, all jobs are submitted to a single scheduler. This single scheduler is responsible for scheduling the jobs on the available resources. Since all the scheduling information is available at once, the scheduling decisions are optimal but this approach is not very scalable in a Grid system [16]. As the size of the Grid increases,

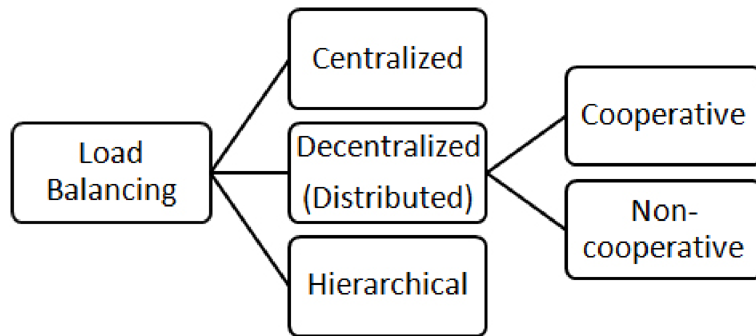


Figure 1.6: Categorization of load balancing algorithms

keeping all the information about the state of all the resources would be a bottleneck. Therefore scalability is an issue in centralized approaches in addition to bringing a single point of failure to the system. Figure 1.7, shows a system with a central load balancing architecture.

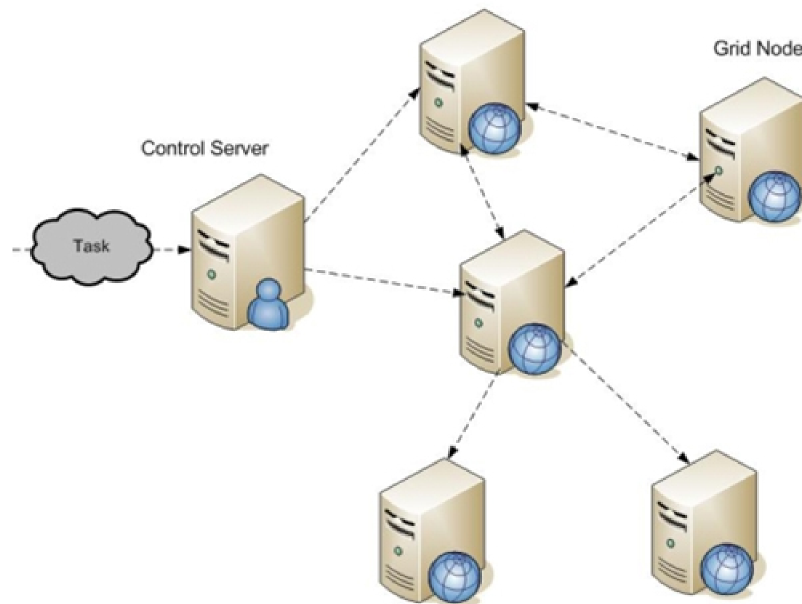


Figure 1.7: Centralized load balancing model

In a decentralized model there is no central scheduler and scheduling is done by the resource requestors and owners independently. This approach is scalable, being distributed

in nature, and suits Grid systems. But individual schedulers should cooperate with each other in scheduling decisions and the schedule generated may not be the optimal schedule. A decentralized load balancing system architecture is shown in Figure 1.8. This category of load balancing is perfect for peer-to-peer architectures and dynamic environments. Based on whether or not schedulers cooperate with each other, decentralized approaches can be further classified as cooperative or non-cooperative [16].

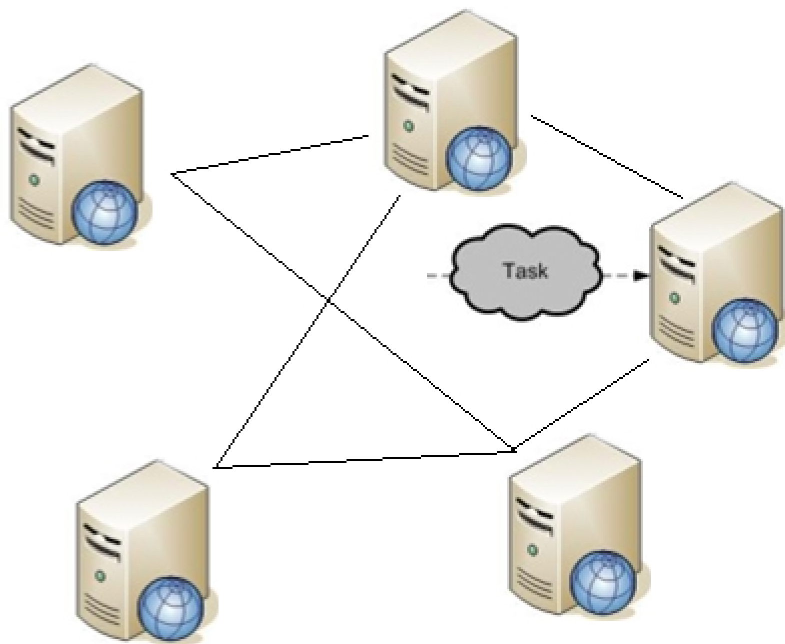


Figure 1.8: Decentralized load balancing model

In a hierarchical model shown in Figure 1.9, the schedulers are organized in a hierarchy. High level resource entities are scheduled at higher levels and lower level smaller sub-entities are scheduled at lower levels of the scheduler hierarchy. This model is a combination of the above two models [16].

Load balancing algorithms can be further classified as *System-Level* or *Application-Level*. Application-level load balancing focuses on minimizing the makespan of a parallel application. Here, makespan is defined as the completion time of all the jobs being sent to the Grid. System-level load balancing, also known as distributed scheduling, aims to maximize process throughput or the overall utilization rate of the machines.

Each of these classes has its advantages and disadvantages depending on a number of

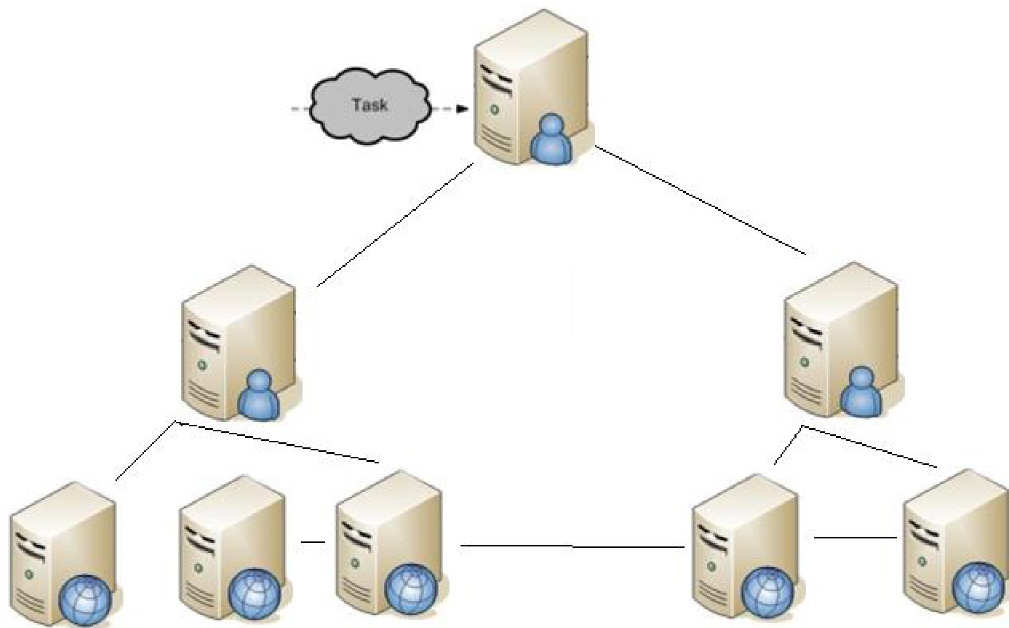


Figure 1.9: Hierarchical load balancing model

factors, e.g., the size of a system, dynamic behavior, etc. [25]. However, all centralized approaches have certain disadvantages:

1. A central scheduler (load balancer) needs current knowledge about the entire state of the system at each point in time. This makes it scale badly with the growth in the size of the system.

2. Failure of the scheduler results in failure of the whole system, while in a distributed approach only some of the work will be lost.

3. Distributed schedulers are much more dynamic and flexible to changes than centralized approaches, because they do not need the state of the system at each step to do their job.

There has been a great effort in recent years in developing distributed load balancing algorithms, while trying to minimize all the communication needs resulting from the distributed nature. In this research, we have focused on designing distributed load balancing algorithms with the inspiration from swarm intelligence contexts. The next section

describes the advantages of swarm intelligence techniques for problems such as load balancing and others.

1.4 Swarm Intelligence Techniques for Load Balancing

As artificial life and swarm intelligence techniques are increasingly being used for solving optimization problems, they have proven themselves as a good candidate in this area. This can be inferred by recent research in the area. Related research on distributed load balancing is reviewed in Chapter 2.

The notion of complex collective behaviour emerging from the behaviour of many relatively simple units, and the interactions between them, is fundamental to the field of artificial life. The understanding of such systems offers new ideas in creating artificial systems which are controlled by such emergent collective behaviour; in particular, the exploitation of this concept might lead to completely new approaches for the management of distributed systems, such as load balancing in Grids [26].

As artificial life techniques have proved to be useful in optimization problems they are a good candidate for load balancing where we aim to minimize the load difference between the *heaviest* and *lightest* node. The benefit of these techniques stems from their capability in searching large search spaces, which arise in many combinatorial optimization problems, very efficiently [27]. Job scheduling is known to be NP-complete when we want to solve it on a single processor, therefore the use of heuristics and involving distribution is necessary in order to cope in practice with its difficulty [19].

1.4.1 Social Insect Systems - Ant Colony

Among swarm intelligence techniques, "social insect systems" are good candidates in many ways. Social insect systems are complex adaptive systems that are able to self-organize within a set of constraints [28]. A social insect colony functions as an integrated unit that is capable of the following [29]:

- Ability to process a large amount of information in a distributed manner.
- Make decisions about how to allocate individuals for various tasks.
- Coordinate the activities of tens or thousands of workers.

- Exhibits flexibility and robustness in response challenges.

Among all their characteristics discussed, two of their aspects are of particular interest. First, they are robust. They function smoothly even though the colony may be continuously growing, or may suffer a sudden traumatic reduction in their numbers because of an accident, predation, experimental manipulation, or may spontaneously split into two distinct colonies of half the size [30]. They routinely cope with gross and minor disturbances of habitat and with seasonal variations in food supply [31]. Second, they are tiny insects with no or very small memory and computational ability, yet they are surviving in our complex real world because of their huge number and adaptability to their environment. Using the idea of their robustness in the real world provides us with novel ideas to use in artificial life. For example, it gives us the ability to deal with the dynamic topology of todays networks as nodes may come and go arbitrarily; and being simple provides us with the efficiency we need in dealing with large scale systems. The application of swarm intelligence to network problems arises when a group of autonomous programs (agents) are working together. This is referred to as *Ant Colony Optimization* (ACO) or multi-agent systems. Each individual or program or autonomous module can be represented as an agent and these multi-agents can be used for network applications such as finding the shortest path, routing, load balancing, management, etc [32].

1.4.2 Particle Swarm Optimization

Another artificial life technique which performs well in optimization problems is Particle Swarm Optimization (PSO). PSO is a stochastic search method that was developed by Kennedy and Eberhart in 1995 [33]. The algorithm is an Evolutionary Algorithm (EA) that imitates the sociological behavior of a flock of birds or school of fishes. In bird flocking, the population benefits from sharing each individuals information and discoveries or past experience with the whole population. Each individual (called particle) in the population (called swarm) will "fly" over the search space to search for the global optimum [34]. PSO is easily implemented as it uses numerical encoding. The convergence speed of PSO relies on different parameter settings. A more detailed analysis about the convergence of PSO can be found in [35] and [36]. We will be using the idea of particle swarm optimization to design a new algorithm for distributed Grid job scheduling.

1.5 Problem Statement

This research investigates, proposes, implements and compares two new approaches for distributed load balancing inspired by Ant-Colony and Particle Swarm Optimization[1]. There are several objectives a good load balancer should address such as fairness, robustness and distribution; a detailed description of each is provided later. We are addressing these requirements with this research. In the Ant-Colony approach each job submitted to the Grid invokes an ant and the ant searches through the network to find the best node to deliver the job to. Ants leave information related to the nodes they have seen as a pheromone in each node which helps other ants to find lighter resources more easily. In the particle swarm approach, each node in the network is considered to be a particle and it tries to optimize its load locally by sending or receiving jobs to and from its neighbours. This process being done locally for each node, results in a move toward the global optima in the whole network.

1.6 Thesis Organization

The rest of this research is organized as follows: Chapter 2 is dedicated to related work similar to this research. We are classifying load balancing algorithms into two categories, centralized and decentralized, and are reviewing some relevant works in each category. The contributions of this research and the benefits it brings to the field of Grid load balancing are discussed in Chapter 3. It is followed by an introduction to Ant Colony Optimization and Particle Swarm Optimization, before the proposed approaches are described in details. Chapter 4 is dedicated to experimental design. A detailed overview of the GridSim toolkit is provided and it is followed by design documentations of the algorithms. Chapter 5 focuses on experimental results. Performance criteria and environmental settings are introduced in this chapter; a thorough comparison between the performance of the algorithms and some other classical approaches is also provided. Finally, Chapter 6 is dedicated to conclusions and future work.

[1] This research is published in the Proceedings of the 24th Annual ACM Symposium on Applied Computing, March 2009

Chapter 2

Related Work

2.1 Overview

The research in the area of load balancing is diverse and it has been an issue in networks since their emergence. There is more research on algorithms done in the area of centralized load balancing than decentralized load balancing. In this chapter we review some research to see how load balancing algorithms have evolved and get familiar with their diversity. There are two sections in this chapter. In the first section we have a look at centralized load balancing approaches which include classical approaches, artificial life inspired and agent-based techniques. In the second section we investigate the literature of decentralized load balancing approaches with the focus on the ant colony load balancing approach.

2.2 Centralized Load Balancing Approaches

In this section we first review some classical approaches in the area of centralized load balancing. As the classical approaches did not satisfy all the requirements of large computational needs, new approaches evolved. We investigate two common approaches in the area of load balancing related to this research. They are agent-based and artificial life approaches.

2.2.1 Classical Approaches

Classical approaches in centralized load balancing have been around since the emergence of the networks. We will look at them briefly.

The **Random** approach is the simplest load balancing approach which assigns tasks to resources in a random fashion regardless of the task properties or resource abilities.

In the **Round-Robin** scheme the tasks are assigned to resources on a rotating basis. Obviously, the characteristics of tasks or resources is not an issue while scheduling.

MET (Minimum Execution Time) assigns each task to the resource which performs it in the least amount of execution time regardless of whether this resource is available or not at that time [38].

MCT (Minimum Completion Time) assigns each task to the resource which obtains the earliest completion time for that task. This causes some tasks to be assigned to resources that do not have minimum execution time. Regarding complexity, if we have m number of machines it takes *O(m)* time to map a given task to resources [38].

The **Min-Min** method finds the execution time of each task on each resource available, then it chooses the smallest such completion time of the task-resource assignment. It updates the completion times after such assignment and repeats the scenario until all tasks are assigned. If we have m number of machines and s number of tasks this heuristic takes $O(s^2m)$ time to complete [38].

Max-Min is very similar to Min-Min, except that it assigns a task with the maximum expected completion time to the corresponding resource. So, it takes $O(s^2m)$ time as well [38].

Suffrage is based on the idea that a task should be assigned to a certain host and if it does not go to that host, it will "suffer" the most; meaning that the task should be scheduled with more priority. For each task, its suffrage value is defined as the difference between its best MCT and its second-best MCT and tasks with high suffrage value take precedence [39].

2.2.2 Agent-Based Approaches

Cao et al. in [17], have addressed Grid load balancing issues using a combination of intelligent agents and multi-agent approaches. At the global Grid level, each agent is a high-level representative of a Grid resource and acts as a service provider of high performance computing power. Agents are organized into a logical hierarchy by different role assignments. There are three roles in the system: Broker, Coordinator and Agent. They cooperate with each other to discover available Grid resources for tasks using a peer-to-peer mechanism for service advertisement and discovery. The hierarchical model can help when issues of scalability arises. When the number of agents increases, the hierarchy can help in processing

many activities in a local domain and does not have to rely on some central agents. Still their architecture of agents incorporates a central agent which coordinates the hierarchy at the highest level.

In [40], an agent-based load balancing algorithm is proposed and is applied to drug discovery and design. Its architecture is hybrid and the algorithm performs well in meeting QoS (Quality of Service) and utilizing idle computational resources dynamically. However, as there is a global information repository which maintains the global information of all the resources in the Grid the same problem as all centralized approaches exist. It results in a single point of failure which leads to critical problems in case the central part fails.

Another agent-based load balancing model is introduced in [41]. This is a credit-based load balancing model. It works according to two policies: *selection policy* and *location policy*. In the selection phase it decides which task should be migrated because of overloaded machines and in the location phase it is decided where it should be sent to. This mechanism not only works for load balancing in clusters and networks but can also be applied in balancing agents with different properties in a multi-agent system. In their approach each agent has a credit and the decision upon which an agent will be migrated or will remain untouched is dependent on its credit. Each agent's credit changes in accordance with the behaviour of the agent system and its interactions. There is a central host which is a decision maker about whether there is a need for an agent to migrate and also it is the commander for selection and location policies.

Cao et al. [42], proposes an agent-based load balancing approach in which an agent-based Grid management infrastructure is coupled with a performance-driven task scheduler that has been developed for local Grid load balancing. This work addresses the problem of load balancing in a global Grid environment. A genetic algorithm-based scheduler has been developed for fine grained load balancing at the local level (such as a multiprocessor or a cluster of workstations). This is then coupled with an agent-based mechanism that is applied to balance the load at a higher level (Grid level). Agents cooperate with each other to balance workload in the global Grid environment using service advertisement and discovery mechanisms [42]. In this research, the scalability is an issue of great importance as a genetic algorithm approach is being used in the local level load balancing part and this may result in a bottleneck for the system.

2.2.3 Artificial Life Approaches

Subrata et al. [18], used Genetic Algorithms [43] and Tabu search [44] for performing centralized load balancing simulations. Both of theses techniques are evolutionary search techniques to find solutions to optimization and search problems. The techniques used are inspired from evolutionary biology and have behaviours such as inheritance, mutation, selection, and crossover. They propose a centralized scheduler in which a typical assignment of tasks to the resources is considered as a solution, and Genetic Algorithms and Tabu search are used to search in the search space and improve the solution. They have proved that the two techniques work better compared to some classical algorithms like *Min-min*, *Max-min* and *Suffrage* in terms of time makespan. The makespan is the difference between the time the first job is sent to the Grid until the last job comes out of the Grid. Each of these three algorithms select a job from a set of tasks, calculate its completion time on each existing processor and assign it to a resource iteratively. Each algorithm differs in the way they choose a task from the set. The Min-min algorithm chooses a task with the minimum completion time in the set. In Max-min, the task with the maximum completion time is chosen first. In suffrage, as introduced before, a metric is defined as suffrage and is used to choose the task. In their implementation a metric is defined as the difference between the first minimum completion time, and second best minimum completion time and the task with the highest suffrage is chosen.

Literature using particle swarm optimization for load balancing is less rich compared to other approaches such as Ant-Colony load balancing.

One application of particle swarm optimization, which is a subset of evolutionary algorithms, in job scheduling is provided in [45], where a fuzzy based particle swarm optimization approach is proposed. They create a fuzzy membership matrix representation of the job scheduling problem out of the existing jobs and resources. Each element in the matrix defines the degree of membership of the specific job to a specific resource. By using PSO, the fitness of such a matrix is improved. The representations of the position and velocity of the particles in the conventional PSO is extended from the real vectors to fuzzy matrices using the membership matrix. The position matrix indicates a fuzzy potential scheduling solution. As the approach is a central approach and does not take the arrival of new jobs in a peer-to-peer like architecture into account it needs further investigations

for such environments.

Another use of particle swarm optimization in Grid task scheduling is investigated in [46]. The mechanism is that they produce a task resource assignment graph out of each task scheduling scheme, and therefore, the problem can be considered as a graph optimal selection problem. Then, a particle swarm algorithm is applied to find the optimal solution in this graph. The longest path of the task-resource assignment graph is considered as the fitness value and it encodes every task-resource assignment as a particle. However, this approach needs the information about the available resources and tasks, and does not address the fact being exposed to a dynamic environment.

Salman et al. [47], have tried to solve the task assignment problem with particle swarm optimization, where they try to find the best mapping between tasks and resources. Each mapping of tasks to resources is considered as a particle. These particles fly over the search space to find the global solution. They compare their approach with a genetic algorithm solution over a number of randomly generated mapping problem instances, and show that PSO can perform better than GA in most test cases.

2.3 Decentralized Load Balancing Approaches

Research in the area of distributed load balancing is diverse. Many researchers have used Ant colony for routing and load balancing. In this section, we provide an overview of some of the work in this area.

2.3.1 Classical Approaches

There are several classical approaches in the area of load balancing which have been around since the emergence of networks.

In **sender-initiated algorithms**, load distributing activity is initiated by an overloaded node (sender) trying to send a task to an underloaded node (receiver) [24].

In **receiver-initiated algorithms**, load distributing activity is initiated from an underloaded node (receiver), which tries to get a task from an overloaded node (sender) [24].

A **stable symmetrically initiated adaptive algorithm** uses the information gathered during polling (instead of discarding it, as the previous algorithms do) to classify

the nodes in the system as sender/overloaded, receiver/underloaded, or OK (nodes having manageable load). The information about the state of the nodes is maintained at each node by a data structure composed of a senders list, a receivers list, and an OK list. These lists are maintained using an efficient scheme and list-manipulative actions, such as moving a node from one list to another or determining to which list a node belongs. These actions impose a small and constant overhead, irrespective of the number of nodes in the system. Consequently, this algorithm scales well to large distributed systems [24].

In the **State Broadcast Algorithm (SBA)** the information policy is based on status broadcast messages. Whenever the state of a node changes, because of the arrival or departure of a task, the node broadcasts a status message that describes its new state. This information policy enables each node to hold its own updated copy of the system state vector (SSV) and guarantees that all the copies are identical.

While the information policy of the previous algorithm is based on broadcast messages, the information policy of the **Poll when Idle Algorithm (PID)** is based on polling. The node starts to poll a subset of the system nodes whenever it enters an idle state.

2.3.2 Ant Colony Approaches

Ant colony optimization has been widely used in both routing and load balancing [48]. As we described earlier in Chapter 1, Ant Colony Optimization (ACO) is considered a subset of social insect system approaches. The main idea underlying this approach is the ability of ants to carry and deposit pheromones (information trails) along their way. This information can later be used by other ants passing the same way.

One approach which is very similar to the ant colony algorithm we propose in this thesis is Messor [49]. Montresor et al. have used an ant colony approach to develop a framework called Anthill which provides an environment for designing and implementing Peer-to-Peer systems. They have developed Messor which is a distributed load balancing application based on Anthill and they have performed simulations to show how well Messor works. In the algorithm, they propose ants can be in one of the two following states: *Search-Max* or *Search-Min*. In the Search-Max state the ants try to find an overloaded node in the network and in the Search-Min state they search for underloaded nodes. Finally, they switch jobs between overloaded and underloaded nodes and hence achieve the load balancing. However, they have not addressed the problem of topology changes in the

network and do not provide evidence to show how good their approach is in comparison to other distributed load balancing approaches.

In [50], a very similar approach to Messor is provided. In this work an agent-based self-organization is proposed to perform complementary load balancing for batch jobs with no explicit execution deadlines. In particular, an ant-like self-organizing mechanism is introduced and is shown to be able to yield good results in achieving overall Grid load balancing through a collection of very simple local interactions. Ant like agents move through the network to find the most overloaded and underloaded nodes but the difference to previous research is they only search $2m + 1$ steps before making the decision and try balancing the load after finding this information. Different performance optimization strategies are carried out. However, they do not compare their results with other distributed load balancing strategies.

Salehi et al. [51], have done similar research to [49] and [50] with some small modifications. They present an ecosystem of intelligent, autonomous and cooperative ants. The ants in this environment can reproduce offspring when they realize that the system is drastically unbalanced. They may also commit suicide when they find equilibrium in the environment. They wander m steps instead of $2m + 1$ and they balance k overloaded node and k underloaded nodes instead of one at a time. A new concept called *Ant level load balancing* is presented for improving the performance of the mechanism. When the ants meet each other at the same node they exchange the information they carry with them and continue on their way.

Sim et al. [52] [48], present a Multiple Ant Colony Optimization (MACO) for load balancing circuit-switched networks. In MACO more than one colony of ants are used to search for optimal paths and each colony of ants deposits a different type of pheromone represented by a different colour. MACO optimizes the performance of a congested network by routing calls via several alternative paths to prevent possible congestion along an optimal path.

Another related and similar research to the ant colony approach we propose in this thesis is done by Al-Dahoud et al. [32]. In their research each node sends a coloured colony throughout the network; this approach helps in preventing ants of the same nest from following the same route and hence enforcing them to be distributed all over the nodes in the network. However, their experimental results are confined to a small number

of nodes and all the jobs have the same properties.

Heusee et al. [53], have used multi-agent systems which have some similarity to ants to solve the problem of routing and load balancing in dynamic communication networks. They have proposed two kinds of routing agents depending on when the distance vector update occurs. The update can be performed while agents are finding their way to their destination (forward routing) or when they backtrack their way back to their source (backward routing).

Other similar research which benefits from the Ant colony's power mostly focus on load balancing in routing problems [31] and [48]. In [48], the research provides a survey of four different routing algorithms: ABC, AntNet, ASGA. ABC is an Ant-Based Control system. They have simulated a network with a typical distribution of calls between nodes; nodes with an excess of traffic can become congested and cause calls to be lost. Using the ants concept, the ants move randomly between nodes, selecting a path at each intermediate node based on the distribution of simulated pheromones at each node. As they move, they deposit simulated pheromones as a function of their distance from their source node, and the congestion encountered on their way [31]. In AntNet, they have applied ideas of the ant colony paradigm to solve the routing problem in datagram networks. Ants collect information about the congestion status of the followed paths and leave this information locally in the nodes. On the way back from the destination to the source, the local visiting table of each visited nodes are modified accordingly [54]. ASGA integrates ant colony systems with genetic algorithms. Each agent in the ASGA system encodes the sensitivity to link and sensitivity to pheromone parameters. Each agent in the population has to solve the problem using an ant system and each agent has a fitness according to the solution found [55].

2.4 Job Migration

Some researchers have considered job migration (migration of partly executed jobs) in their load balancing algorithms. However, job migration is not very beneficial in practice and some research work have tried to investigate this ([56] [57] [58]). It involves collecting all system states (e.g. virtual Memory image, process control blocks, unread I/O buffer, data pointers, timers etc.) of the job which is large and complex. Studies have shown that [18]:

- Job migration is often difficult to achieve in practice.
- The operation is generally expensive in most systems.
- There are no significant benefits of such a mechanism over those offered by non-migratory counterparts.
- There are very rare cases in which job migration can provide slight improvements. These conditions usually have high variability in *both* job service demands and the workload generation process [56].

According to these, we are not concerned with job migration for our proposed approaches.

As most of the classical approaches are based on centralized load balancing and this category are mostly used in many standard toolkits like Globus; there are efforts to develop robust decentralized approaches to benefit from their advantages. The review in the related literature reveals that there are not as many decentralized approaches as there are centralized ones. On the other hand, existing decentralized approaches which are mostly based on Ant Colonies are not accompanied with various performance measures to state how they perform in different scenarios and situations. Still, decentralized approaches in the Grid infrastructure are fewer in number than approaches designed for networks and peer-to-peer systems. In this research, we introduce two new load balancing algorithms based on Ant Colony and particle swarm optimization. The Ant Colony approach is similar to some approaches we reviewed in this section, while the particle swarm is a completely new approach. We will measure their performance under different scenarios to have a good understanding of their responsiveness.

2.5 Summary

In this chapter we investigated different research areas of load balancing and provided details about several research work which addressed the problem in both areas of centralized and distributed load balancing. In the next chapter, we propose two new approaches inspired by artificial life techniques for distributed job scheduling for the Grid.

Chapter 3

Proposed Load Balancing Approaches

3.1 Overview

In this chapter, the characteristics and requirements of a good load balancing algorithm are investigated. Furthermore, the contributions the research makes in the area is given in Section 3.2. The proposed approaches using ant colony and particle swarm optimization are introduced in Section 3.2 and 3.3 respectively, including detailed descriptions and pseudo-code listed in Sections 3.4 and 3.5.

3.2 Contributions and Benefits

In the previous two chapters we have discussed what load balancing algorithm is, outlined the different categories of load balancing algorithms, and reviewed some related work done in the area of load balancing and job scheduling on the Grid. Putting these all together reveals some issues and requirements which a load balancing algorithm should address. A list of these requirements is provided here:

- **Optimum resource utilization.** A load balancing algorithm should optimize the utilization of resources whether they are resources in the Grid such as computational or data resources, time or cost related to these resources, etc. As the Grid environment leaves us with a dynamic search space this optimality is inevitably a partial optimal solution which improves the performance.

- **Fairness.** When a load balancing algorithm is fair, it means that the difference between the heaviest loaded node and lightest loaded node in the network is minimized keeping in mind that the search space is dynamic. The load is defined by the number of jobs assigned to each resource relative to its computational power.

- **Flexibility.** It means that as the topology of the network or the Grid changes, the algorithm should be flexible enough to adhere to the changes in the network.

- **Robustness.** Robustness refers to the fact that when failures in the system occur the algorithm should have a way to deal with the failure and be able to cope with the situation, not to break down because of the failure; on the contrary, the algorithm should be able to manage it.

- **Distribution.** Distribution for managing resources and running the load balancing algorithm has the benefit of leaving out the single point of failure which centralized approaches are affected by.

- **Simplicity.** By simplicity we try to point out both the size of single software units which are being transferred among resources in the Grid, and also the overhead that these units bring to resources in order to make load balancing decisions. The size of software units are important as the take up bandwidth when want to transfer themselves between resources. As there units are being executed in Grid nodes there is a preference to keep needed computations as simple as possible.

As we described earlier in Chapter 1 (Section 1.4), artificial life techniques have shown their usefulness in many optimization problems as well as in job scheduling as it can be encoded as an optimization problem. As in nature, these systems have evolved to adopt and work under many conditions and changes, both in the environment and in their population they show good flexibility and robustness in many circumstances which provides us inspirations for computer algorithms. On the other hand they are distributed and simple in nature which can be considered a perfect solution without the single point of failure problem and the overhead for the network.

This research introduces two new algorithms in the area of swarm intelligence techniques for load balancing. One of the algorithms is taking its inspiration from social insect systems and more specifically Ant Colony systems. The other algorithm is based on particle swarm optimization. Ant Colony optimization has been used for load balancing and routing purposes in networks and also in Grid resource scheduling. In this research, we are suggesting a new approach for applying Ant colony optimization to the problem of load balancing. In the previous approaches, ants act independently from jobs being submitted

while in our approach there is a close binding between jobs and load balancing ants. On the other hand, particle swarm has not been used for distributed load balancing in the Grid before and we are proposing a new way to use it in the Grid. A list of the benefits and expansions which this research has brought to the area of Grid load balancing is summarized below:

1. Well-known research works in the area of Grid load balancing, even those with inspiration from artificial life techniques like genetic algorithms and Tabu search, have been suggested using centralized approaches which, as we mentioned before, have many drawbacks. Literature using swarm intelligence techniques for distributed load balancing is less rich and has not been around for a long time.

2. Although a variety of ant colony inspired approaches have been used for distributed load balancing, there is no comparison of this approach with any other distributed artificial life technique. In this research, we compare the performance of the ant colony approach with another artificial life inspired technique, particle swarm. We will be performing measurements and comparisons between the two algorithms to find which can be more effective and under which conditions. We will also compare the performance of the algorithms in comparison to two other classical techniques.

3. Particle swarm optimization has been used to address the problem of centralized load balancing [45, 46, 47], but it has never been used for distributed load balancing in a dynamic environment such as the Grid. In this research we will see that this approach can perform very well in this regard.

4. Most of the research and experimental results, especially in the area of distributed load balancing and ant colony, have used their own developed infrastructure to simulate the performance of their approaches, thus the question remains how well they will perform in a real world environment. We have used a reliable simulation platform, **GridSim**, which provides us with reliable results and takes a step further to do the evaluations in a more realistic environment. A detailed description about the **GridSim** framework will be provided later.

The rest of this chapter is organized as follows: the next two sections will provide information about Ant colony optimization and particle swarm optimization. Then, we

propose our new algorithms, AntZ and ParticleZ, which are based on the two approaches introduced.

3.3 Ant Colony Optimization

In the early 1990s, Ant Colony Optimization (ACO) [59, 60, 61] was introduced by Dorigo and colleagues as a novel nature-inspired meta-heuristic for solving hard Combinatorial Optimization (CO) problems [62]. ACO takes its inspiration from the foraging behavior of real ants. Ants use a signalling communication system based on the deposition of pheromone over the path they follow, marking their trail. Pheromone is a hormone produced by ants that establishes a sort of indirect communication among them. Basically, an individual ant moves at random, but when it finds a pheromone trail, there is a high probability that this ant will decide to follow this trail.

Individual ants have a relatively basic and unsophisticated behaviour. They have a very limited memory and exhibit individual behaviour that appears to have a large random characteristic. Acting as a collective, however, ants manage to perform a variety of complicated tasks with great reliability and consistency [26]. One of the well-known and classical examples of the ants being able to do complicated tasks is finding the shortest path between a nest and a food source. An example of the construction of a pheromone trail while searching for a shorter path is shown in Figure 3.1, which was first presented in [63]. In Figure 3.1A, there is a path between the food and the nest established by the ants. In Figure 3.1B, an obstacle is inserted in the path. Thus, the ants spread to both sides of the obstacle, since there is no clear trail to follow (Figure 3.1C). As the ants go around the obstacle and find the previous pheromone trail, a new pheromone trail will be formed around the obstacle. This trail will be stronger in the shortest path than in the longest path, as shown in Figure 3.1D, as the shorter path receives a higher amount of pheromone in a time unit [2]. Although, all ants are moving at approximately the same speed and deposit a pheromone trail at approximately the same rate, it is the fact that it takes longer to contour the obstacle on their longer side than on their shorter side which makes the pheromone trail accumulate faster on the shorter side. It is the ant's preference to follow higher pheromone trail levels which makes this accumulation even faster on the shorter [64].

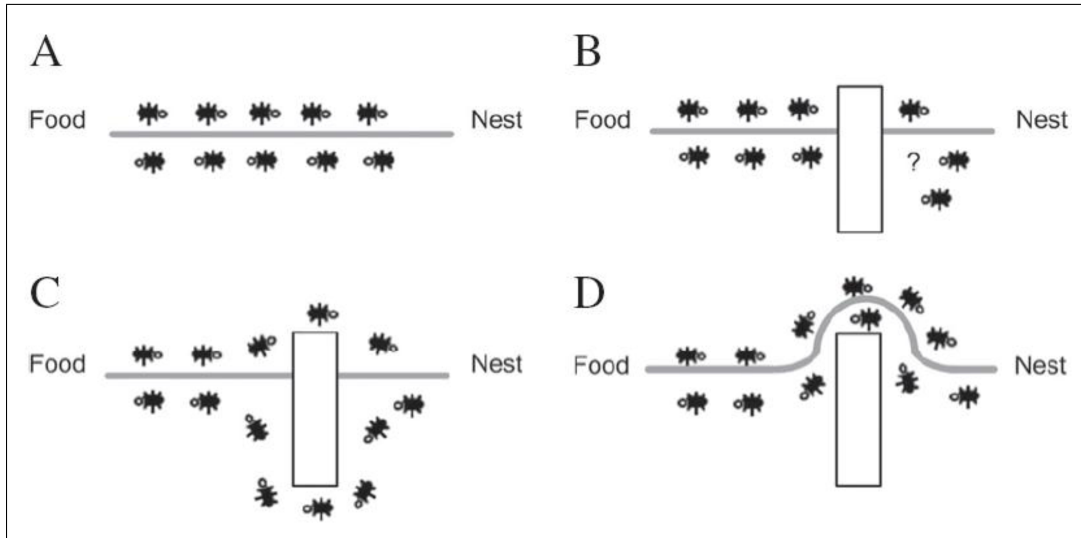


Figure 3.1: A. Ants in a pheromone trail between nest and food; B. an obstacle interrupts the trail; C. ants find two paths to go around the obstacle; D. a new pheromone trail is formed along the shorter path [2].

As we have seen in the example, a single ant has no global knowledge about the task it is performing; yet by indirect communication skills, they tend to be able to do tasks which seem intelligent. The ant's actions are based on local decisions and are seemingly unpredictable. The intelligent behavior naturally emerges as a consequence of the self-organization and indirect communication between the ants. This is usually called *Emergent Behavior* or *Emergent Intelligence*.

Depending on the species, ants may lay pheromone trails when travelling from the nest to the food, or from the food to the nest, or when travelling in either direction. They also follow these trails with a trustworthiness, which, among other variables, is a function of the trail strength. Ants leave pheromones as they walk by stopping briefly and touching their gaster on the ground, which carries the pheromone leaving gland. The strength of the trail they lay is a function of the rate at which they make deposits, and the amount per deposit. Since pheromones evaporate, the strength of the trail when it is encountered by another ant is a function of the original strength, and the time since the trail was laid. Most trails consist of several trails from many different ants, which may have been laid at different times; it is the composite trail strength which is sensed by the ants.

By now we have explained that, indirect communication between the ants via pheromone

trails enable them to find the shortest paths between their nest and food sources. This characteristic of real ant colonies is exploited in artificial ant colonies, in order to solve combinatorial optimization problems [65, 66, 67].

What makes the ant colony approach especially interesting for the distributed load balancing problem is its distributed nature. Other artificial life techniques like genetic algorithms, tabu search or particle swarm optimization, though being quite powerful for optimization problems, they have one drawback that make them better for centralized environments; the solutions should be compared with each other by evaluating their usefulness (i.e. fitness) which prevents a completely distributed approach. Although distributed versions of them have been introduced recently, this distribution is achieved by adding some extra steps to the classical algorithms, which have an overhead as well (for example it is called migration in GA) [68]. On the other hand, ants begin to move toward the optimized solution by communicating indirectly through the environment with other local ants, and this communication is even biased through iterations.

Besides the ability of indirect communication via leaving pheromone on their paths, ants are capable of other complex behaviours without having any intelligence incorporated in them. One of the behaviours is the ability of ants to cluster objects (like dead corpses) in their nests. At the first glance, they may seem to be directed by a leader to cluster objects, but Figure 3.2 shows a very simple behavior for an ant which enables it to cluster objects without any intelligence. This figure shows a flow chart of an ant which moves around randomly until it encounters an object; if the ant has been carrying an object, it will drop the object, otherwise the ant picks it up and continues on its way [49]. As can be seen, each ant, by following this very simple behaviour, seems to be cooperating and piling dead corpses in the nest. In load balancing we are using the same pattern of behaviour only in a reverse way. The ants want to distribute jobs as many as possible rather than piling them.

Taking the idea of leaving trails to guide other ants and the idea to cluster objects we have enough information about an ant's behaviour. We introduce our proposed algorithm (AntZ) in Section 3.4 based on this information.

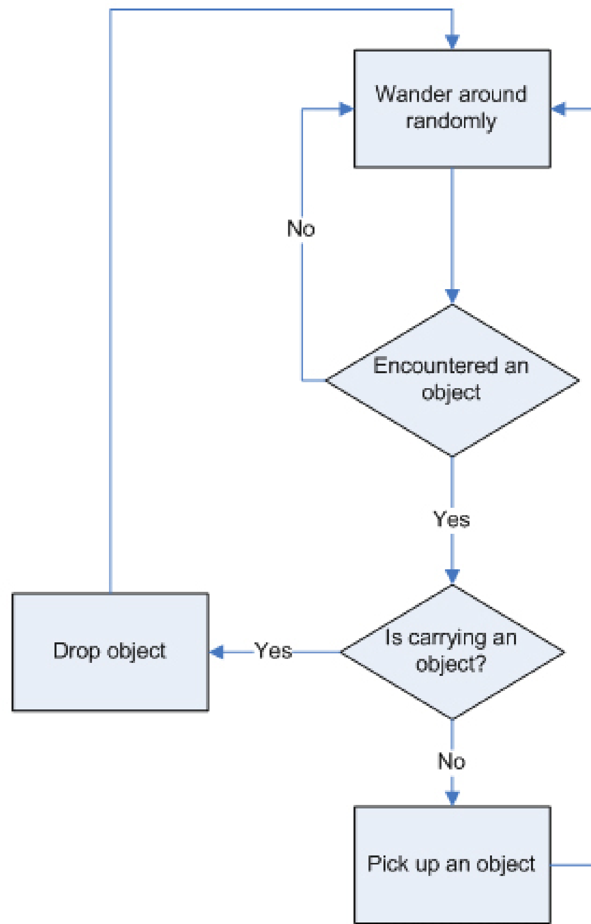


Figure 3.2: The flow chart of an ant behavior capable of clustering objects

3.4 Particle Swarm Optimization

Particle swarm optimization (PSO) has roots in two methodologies. It relates to artificial life in general, and in particular ,to bird flocking, fish schooling, and swarming theory. It is also related to evolutionary computation, and has ties to both genetic algorithms (GA) and evolutionary programming. The system is initialized with a population of random solutions and searches for the optimum solution by updating itself through generations. However, unlike GA, PSO (in its standard form) has no evolutionary operators such as crossover and mutation. In PSO, the potential solutions, called particles, fly through the problem space by following the current optimum particles [33]. Relationships, similarities and differences between PSO and GA are briefly reviewed in [69].

In a PSO system, multiple candidate solutions can coexist and collaborate simultaneously. Each solution candidate, called a *particle*, flies in the problem search space (similar to the search process for food of a bird swarm) looking for the optimal position to land. A particle, as time passes through its quest, adjusts its position according to its own *experience*, as well as according to the experience of neighbouring particles [46]. There are two main characteristics for each particle in a PSO algorithm: its position which defines where the particle lies relative to other solutions in the search space; and its velocity which defines the direction and how fast the particle should move to improve its fitness. As in any evolutionary algorithm, the fitness of a particle is a number representing how close that particle is to the optimum point compared to other particles in the search space.

One of the advantages of the particle swarm optimization technique over other social behavior inspired techniques is its implementation simplicity. As there are very few parameters to adjust in a particle swarm optimization approach, it is simpler than other evolutionary techniques. Yet, as it is a new approach, it has not yet been widely used for dynamic Grid job scheduling.

A simple particle swarm optimization pseudo-code can be seen in Algorithm 3.4.1. The first step is the initialization step. Particles are created and their positions and velocity vectors are assigned randomly. After that, until a final criterion is met, the algorithm runs by calculating the fitness value for each particle. Each particle has a history of its best fitness value found so far and it will be updated when the particle finds a position which is better than all the positions it has been in given its history. We call this value *pBest*. On the other, hand the algorithm keeps track of the best particle and its fitness among all the particles in the search space which is the global optimum so far and we refer to it as *gBest*. At the end of each iteration, both the local best and the global best solutions are updated and used in the next iteration.

In Algorithm 3.4.1 it is shown that both the *pBest* and *gBest* get updated. Equations 3.1 and 3.2 state what is done to update the best local and best global solutions mathematically. On the other hand, $\hat{\mathbf{g}}$ is the current optimal solution with fitness $f(\hat{\mathbf{g}})$. The current position of the particle is denotaed by x_i and $\hat{\mathbf{x}}_i$ is the representative for the best position of the particle so far in the ith iteration.

The last step in the algorithm is to update each particle's position and velocity given all the information we have collected so far.

Equation 3.3 is used to calculate the particle's new velocity according to the three terms which constitute the equation. The first term is the effect of the particle's previous velocity on its current velocity. The second term controls the effect of the particle's current distance from the best position the particle has been in during its lifetime; as this distance increases the velocity also increases to guide the particle toward the best position it has been in its history. The third term controls the particle to move toward the best particle in the swarm and as it gets farther from it, the effect of this term will be more. Then, the particle flies toward a new position according to Equation 3.4 [46].

$$\text{If} f(\mathbf{x}_i) < f(\hat{\mathbf{x}}_i),\ \hat{\mathbf{x}}_i \leftarrow \mathbf{x}_i \tag{3.1}$$

$$f(\mathbf{x}_i) < f(\hat{\mathbf{g}}),\ \hat{\mathbf{g}} \leftarrow \mathbf{x}_i. \tag{3.2}$$

Algorithm 3.4.1: PARTICLESWARMOPTIMIZATION()

$globalBestFitness \leftarrow 0$

for $eachParticle \leftarrow 1$ **to** n

$\{ initializeParticle$

while $notConverged$

- **for** $particle \leftarrow 1$ **to** n
  - $fitness \leftarrow calculateFitnessValue$
  - **if** $fitness > BestFitnessInHistory$
    - **then** $pBest \leftarrow updateBestFitnessInHistory$
  - **if** $fitness > globalBestFitness$
    - **then** $gBest \leftarrow updateGlobalBestParticle$
  - $velocity \leftarrow updateParticleVelocities(pBest, gBest)$
  - $position \leftarrow updateParticlePositions(velocity)$

Two factors characterize the particle's status in the search space: its position and its velocity. The m-dimension position for the *ith* particle in the *kth* iteration can be denoted as $x_i(k) = (x_{i1}(k), x_{i2}(k), ..., x_{im}(k))$. Similarly, the velocity (i.e., distance change) is also

an m-dimensional vector, for the ith particle in the kth iteration and can be described as $v_i(k) = (v_{i1}(k), v_{i2}(k), ..., v_{im}(k))$. The particle updating mechanism for a particle can be formulated as in Equation 3.3 and 3.4:

$$v_{id}^{k+1} = w \times v_{id}^{k} + c_1 \times rand_1 \times [pbest - x_{id}^{k}] + c_2 \times rand_2 \times [gbest - x_{id}^{k}] \tag{3.3}$$

$$x_{id}^{k+1} = x_{id}^{k} + v_{id}^{k+1} \tag{3.4}$$

The term v_{id}^{k}, called the velocity for particle i in the kth iteration, represents the distance to be travelled by this particle from its current position, x_{id}^{k} represents the particle position in the kth iteration, $pbest$ represents its best previous position (i.e. its experience), and $gbest$ represents the best position among all particles in the population. Further, $rand_1$ and $rand_2$ are two random functions with a range [0,1], having similar or different distributions. Also, c_1 and c_2 are positive constant parameters called acceleration coefficients (which control the maximum step size of the particle). The inertia weight w, is a user specified parameter that controls together with c_1 and c_2, the impact of previous historical values of particle velocities on its current velocity. A larger inertia weight pressures towards global exploration (searching new area), while a smaller inertia weight pressures toward fine-tuning the current search area. Suitable selection of the inertia weight and acceleration coefficients can provide a balance between the global and the local search. The random values involved, prevent the optimization from being caught in local optimal. A detailed analysis on the effect of parameter selection on the convergence of PSO is provided in [70].

3.5 Ant Colony Load Balancing: AntZ

In this section, a new load balancing algorithm which is developed based on the concepts of ant colony optimization is described. This algorithm (AntZ) is developed by merging the idea of how ants cluster objects with their ability to leave trails on their paths so that it can be a guide for other ants passing their way. We are using the inspiration of how ants are able to cluster objects trying to use an inverse version and use it to spread the jobs in the Grid. Figure 3.3, shows an overview of a network being used by ants. In the figure, circles denote the resources in the Grid and ants are carrying job information moving from one node to another to deliver the jobs.

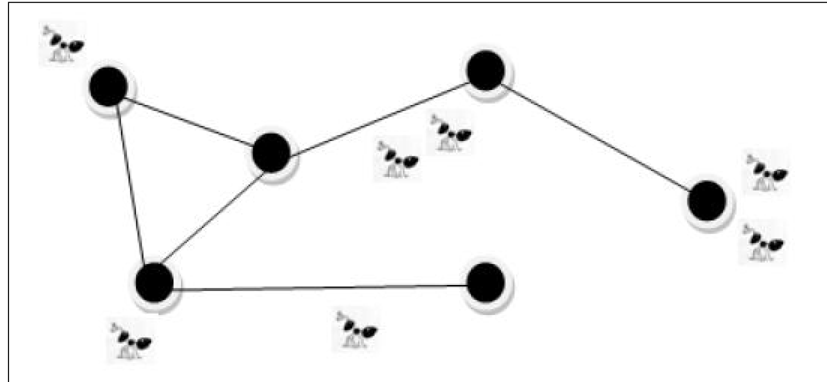


Figure 3.3: Overview of the AntZ system

Figure 3.4, shows the sequential events which occur in the system when a job is submitted to the Grid.

A description of each phase follows:

1. **Job Submission**: A user submits a job to its local resource node. The jobs each user submits are independent of each other.

2. **Ant Invocation**: An ant is created and invoked in response to the user's request. The ant is initialized with the job supposed to be scheduled on the Grid.

3. **Ant Search**: The ant starts searching the Grid to deliver the job to the best suitable node (lightest loaded node) by taking one step a time. Each step consists of leaving one resource and moving to another resource in the Grid. The number of steps each ant takes can either be fixed for all the ants or different for each one.

4. **Pheromone Laying**: The ants carry the load information of the visited nodes along with themselves. As the ant is moving from one node in the network to another it builds up statistical information about the load of the nodes it has visited in resources.

5. **Decision Making**: The ants decide which resource to choose for their next step either by looking at the load table information of nodes or they choose a node randomly by the probability of a mutation factor.

6. **Job Execution**: Finally, the ant delivers the job to a resource and dies. Once the job is completed the answer will be sent back to the original resource.

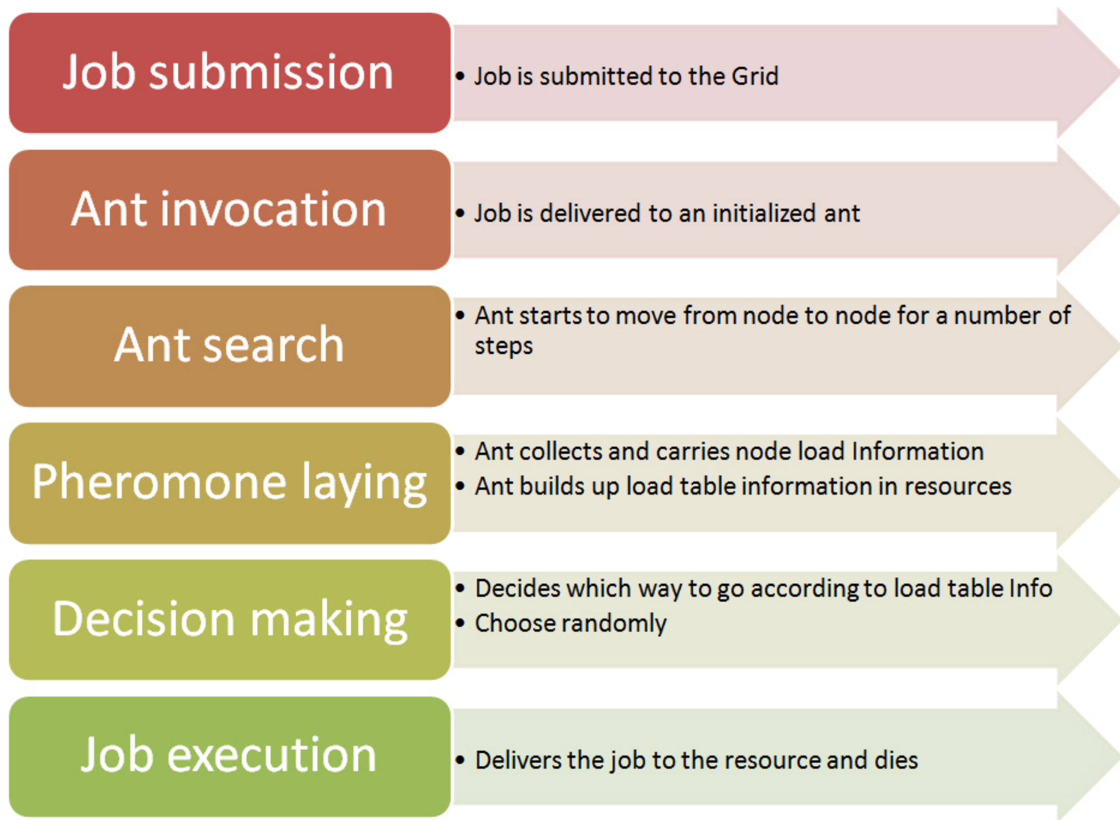


Figure 3.4: Different phases of the AntZ algorithm

A pseudo-code of the AntZ approach is provided in Algorithm 3.5.1. AntZ is a distributed algorithm and each ant can be considered as an agent working independently. The pseudo-code addresses the main functions that an ant performs during its life cycle. Collectively, all the ants show the desired behaviour by following these steps. More details about this algorithm can be found in the Appendix, which provides the complete source code of the implementation.

As shown in the pseudo-code, when a job is submitted to a local node in the Grid an ant is initialized and starts working. In each iteration, the ant collects the load information of the node it is visiting (*getNodeLoadInformation()*) and adds it in its history. The ant also updates the load information table in the visiting nodes (*localLoadTable.update()*). This update simply is a table entry update with new information about load status of resources.

When moving to the next node the ant has two choices. One choice is to move to a random node with a probability of mutation rate, *mutRate*. The other choice is to use the load table information in the node to choose where to go. The mutation rate decreases

with a *DecayRate* factor as time passes so that the ant will be more dependent to load information than to random choice. This iterative process is repeated until the finishing criteria is met which is a predefined number of steps. Finally, the ant delivers its job to the node and finishes its task.

Ants build up a table in each node, shown in Table 3.1. This table acts like a pheromone an ant leaves while it is moving and guides other ants to choose better paths rather than wandering randomly in the network. Entries of each local table are the nodes that ants have visited on their way to deliver their jobs together with their load information.

Algorithm 3.5.1: ANTZALGORITHM($MutRate, MaxSteps, DecayRate$)

```
step ← 1
initialize()
while step < MaxSteps
  do {
    currentLoad ← getNodeLoadInformation()
    AntHistory.add(currentLoad)
    localLoadTable.update()
    if random() < MutRate
      then nextNode = RandomlyChosenStep()

      else nextNode = chooseNextStep()
    MutRate ← MutRate − DecayRate
    step ← step + 1
    moveTo(nextNode)
  }
deliverJobToNode()
```

Reading the information in the load table in each node and choosing a direction, which is represented as the *chooseNextStep()* procedure in Algorithm 3.5.1, the ant uses a simple policy. It chooses the lightest loaded node in the table. The corresponding pseudo-code is provided in Algorithm 3.5.2. As shown in the algorithm, the ant chooses the lighter node in the table and in case of a tie, the ant chooses one with an equal probability.

NodeIp	Load
192. 168. 35. 25	0.8
...	...
...	...

Table 3.1: Load table information in nodes

Algorithm 3.5.2: CHOOSENEXTSTEP()

```
bestNode ← currentNode
bestLoad ← currentLoad
for entry ← 1 to n
  if entry.load < bestLoad
    then bestNode ← entry.node

    else if entry.load = bestLoad
      if random.next < probability
        then bestNode ← entry.node
```

As the number of jobs submitted to the network increases, the ants can take up a huge amount of bandwidth of the network, so moving ants should be as simple and small-sized as possible. To do this, instead of carrying the job while the ant is searching for a "light" node, it can simply carry the source node information to which the job was delivered and a unique job id of the source node. Thus, whenever an ant reaches its destination the job can be downloaded from the source as needed.

The algorithm has some parameters which can be set according to the specific scheduling requirements (i.e. size of the network, job specifications, etc.). The effect of these parameters and their values on the performance of the algorithms are investigated. One of the parameters is *MaxSteps* which defines how many steps an ant should be moving around until it delivers the assigned job to a node in the Grid. If the ant wanders too much before delivering its job, it will cause an increase in the execution time of each job and hence decrease the performance. On the other hand, if the ant gives up too quickly

without moving around then the pheromone (load table information) which it leaves behind decreases, which in turn decreases the performance of the algorithm. Furthermore, the ant might not have enough time to encounter a good and light node. Thus, all these parameters should be set carefully.

Another parameter which influences the performance of the AntZ algorithm is *MutRate*. As the ants are moving and they are using the load table information to decide which way to go, they sometimes randomly choose an arbitrary node in the Grid to move towards it. The probability of choosing their way randomly is controlled by *MutRate*. *MutRate* decreases with a decay rate (*DecayRate*), while the ant is alive and is searching. This parameter (*DecayRate*) can also have an effect on the performance of the AntZ algorithm.

3.6 Particle Swarm Optimization: ParticleZ

Using the idea of particle swarm optimization described in Section 3.3, which proposes a new approach for scheduling jobs in the Grid. In the ParticleZ algorithm, all the nodes in the Grid are considered as a flock or group of swarms (of bees) and each node in the Grid is a particle in this flock. Figure 3.5, simply shows a symbolic representation of a Grid running the ParticleZ algorithm. In the figure, each bee is in position of each resource in the Grid, the black lines between bees are a representation of links between resources in the Grid and the honey they are carrying is the representation for jobs submitted. When a bee has a lot of honey to carry, it will share it with one of its neighbours who has less honey.

As we described in Section 3.3, each particle in the particle swarm optimization is defined by two characteristics: its *position* and its *velocity*. Following the analogy from the particle swarm optimization perspective, the position of each node in the flock can be determined by its load. This definition helps as we are actually searching in the load search-space and we are trying to minimize the load, so each node in this search space takes a position according to its load. The velocity of each particle in its position can be defined by the load difference the node has compared to its other neighbour nodes. As the particles are trying to balance the load, they can move toward each other by the changes they make to their position (i.e. load), this change in each particle's position can be achieved by exchanging jobs between them. The larger their difference is, the faster

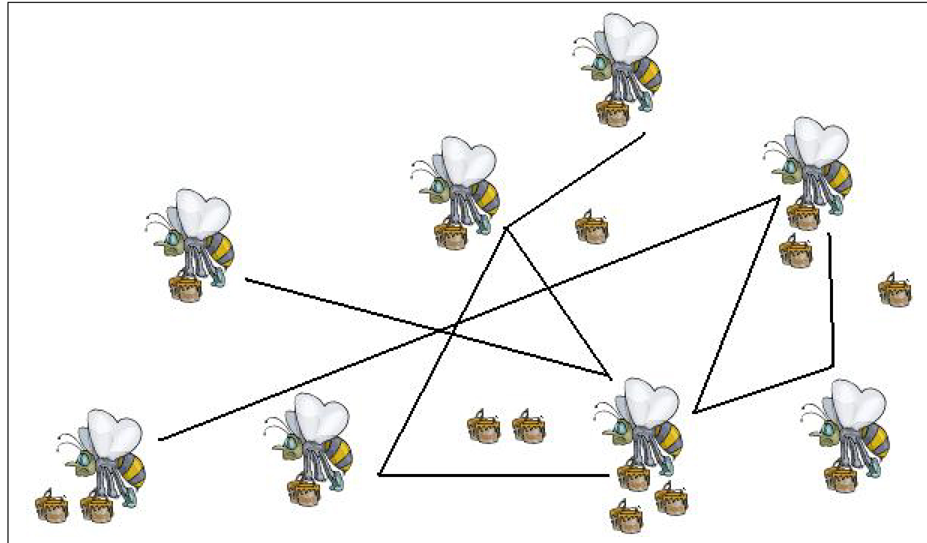


Figure 3.5: Overview of the ParticleZ system

they will move toward each other, with a larger velocity.

Figure 3.6 shows the different phases of the ParticleZ algorithm. A description of each phase follows:

1. **Job Submission**. A user submits a job to its local resource node. The jobs each user submits is supposed to be independent of each other.

2. **Queueing**. As jobs are submitted to the nodes in the network they go in a local queue list of jobs in each node waiting for their turn to be executed.

3. **Node communication**. Nodes communicate with each other about their load information to find a better (lighter) candidate to execute their workload. Actually, resources are trying to move toward their best neighbours (particles) by submitting jobs assigned to them (only those jobs waiting in the queue list to be executed) to other lower loaded neighbour nodes.

4. **Job Exchange**. Nodes submit some of their jobs to their best found neighbours. The amount of workload being submitted is being controlled by a threshold defined by the load difference. Another factor effecting the amount of load exchange is the difference between the lightest neighbour and the second lightest neighbour as we do not want to burden too much load on the lightest neighbour, that it exceeds the

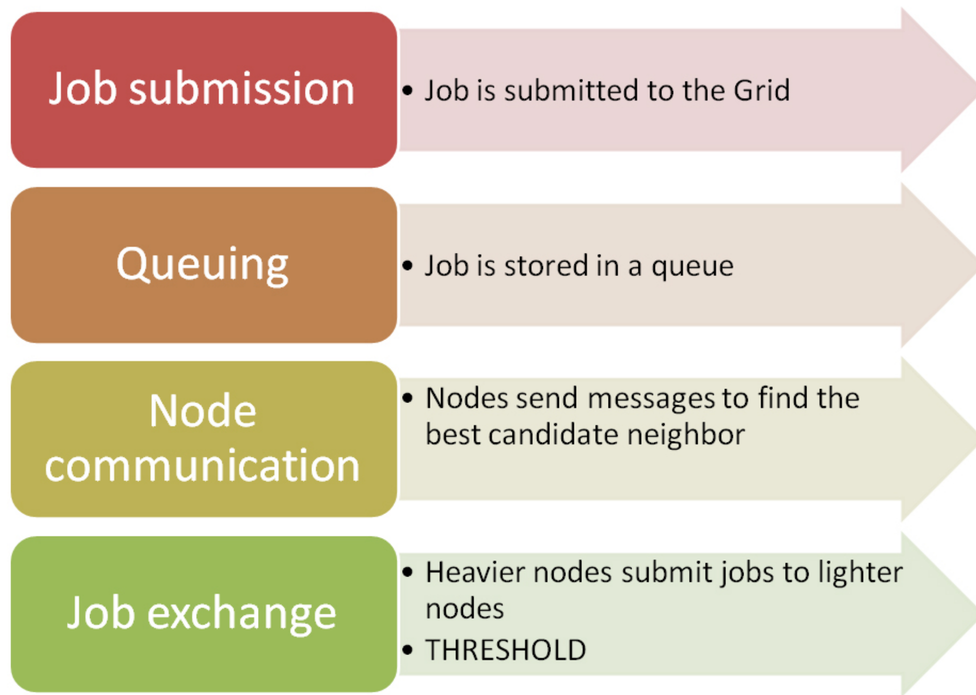


Figure 3.6: Different phases of the ParticleZ algorithm

second lightest neighbour load.

Taking into account that all nodes are exchanging their loads in parallel, and the dynamic nature of the environment, the network will reach to a partial global optima quickly. Thus, each node will submit some jobs to one of its neighbours, which has the minimum load among all. If all its neighbours are busier than the node itself, no job is submitted by the current node.

The pseudo-code describing this scenario can be seen in Algorithm 3.6.1. This is the pseudo-code of each individual particle (resource), which runs the ParticleZ algorithm. As can be seen in the pseudo-code, if there are any jobs in the queue waiting to be executed the node tries to submit them to a lighter node in its neighbourhood, and hence spread the load fairly among resources.

In exchanging load from a heavier loaded node to a lighter loaded node, attention must be paid not to burden the lighter node, so that it exceeds the load of the second lightest node among neighbours.

If this happens we are not only distributing the load fairly but we are creating a load imbalance. To achieve this, we define a *THRESHOLD* variable which tells how much

load exchange should happen between nodes. It is calculated by subtracting *lightestLoad* from *secondLightestLoad* among neighbours and the load exchange happens as long as the *velocity* is greater than the *THRESHOLD* value.

There are some issues related to the PSO which are necessary to be addressed. In the algorithm we propose, the particle tries to move toward its best local neighbour only, while in the classical PSO algorithm particles keep track of their best global solutions so far. The reason we have not included the history of each particle is that we are dealing with a dynamic environment in which the problem being solved is changing all the time as users are submitting jobs unpredictably; thus, the global best solution that the particle has seen might not be valid in this dynamic environment.

Algorithm 3.6.1: PARTICLEZALGORITHM()

$sourceLoad \leftarrow currentNodeLoad()$
while $running$
  do if $jobsQueue.size > 0$
  then
$$\begin{cases} lightestLoad \leftarrow chooseBestNeighbour(sourceLoad) \\ secondLightestLoad \leftarrow chooseSecondLightestNeighbour(sourceLoad) \\ velocity \leftarrow sourceLoad - lightestLoad \\ THRESHOLD \leftarrow lightestLoad - secondLightestLoad \\ \textbf{while } velocity > THRESHOLD \\ \quad \textbf{do} \\ \quad \begin{cases} submitJobs(velocity, destLoad) \\ sourceLoad \leftarrow currentNodeLoad() \\ velocity \leftarrow sourceLoad - lightestLoad \end{cases} \end{cases}$$

The equation for updating the velocity of each particle which was introduced in Section 3.3 takes the following form in our design of ParticleZ:

$$v_{id}^{k+1} = gbest - x_{id}^{k} \tag{3.5}$$

As mentioned earlier, we are dealing with an environment which is changing dynamically (i.e. the search space is changing), so the use of the past experience of each particle is

not useful; therefore, we assign zero to c_1 in order to omit the effect of the past history of the particle at the optimum point. Also again, because of the dynamicity of the problem the previous velocity should not effect our current decision, therefore we assign a value of zero for w as well. On the other hand, we want to use neighbour particles to decide which one is better to share the work load with; we have used a value of one for c_2.

In Equation 3.6, the formula for updating a particle's position is shown which is the same as the one we introduced in Section 3.2. As mentioned, the position of a particle (x_{id}) is its load value and it changes while the resource submits jobs to its neighbours.

$$x_{id}^{k+1} = x_{id}^{k} + v_{id}^{k+1} \tag{3.6}$$

3.7 Summary

In this chapter, we reviewed the requirements of a good load balancing algorithm and listed the characteristics of this research based on related work. The optimization approaches used to develop the algorithms were introduced and the AntZ and ParticleZ algorithms were described in details. Regarding the characteristics introduced in section 3.2, the algorithms are distributed and flexible in response to changes in the Grid. They are simple as the size of an ant or a communication message in ParticleZ is very small. More on this issue will be provided while we show the experimental results in Chapter 5. The next chapter is dedicated to the design and implementation of the approaches introduced.

Chapter 4

Experimental Design and Implementation

4.1 Overview

In this chapter, we provide the design and implementation of the proposed approaches. In the first step, we implemented the algorithms simply without any simulation framework, to be able to test the feasibility of the proposed approaches. As the primary results were satisfactory, we implemented the algorithms considering real world parameters with a Grid simulation toolkit. **GridSim**, was chosen as the simulation toolkitis explored and discussed later in this chapter. We will continue by providing specific design and implementation details about AntZ and ParticleZ.

4.2 Prototype and Initial Results

In order to validate the feasibility of the proposed algorithms we first tried a simple version of each of them in a simulation framework without any well-known toolkits. We used Java as the language to simulate and run our experiments, and we simplified the simulation in some aspects (e.g. having only one Processing Element (PE) for each resource). A class diagram of the primary design is provided in Figure 4.1[1].

As shown in the class diagram, each node has an attribute that defines its CPU speed and each job is characterized by its length which is the number of instructions it contains. By these two characteristics we will be able to know how long it will take for each job to run on a resource. Each node has a *loadTable* attribute which contains the information each ant leaves in visited nodes and also the *connectedNodes* attribute is accommodating the neighbours of each resource.

Although the nature of the system is distributed, we did not want to run a thread for

[1]The UML diagrams in this chapter are drawn using http://www.gliffy.com/

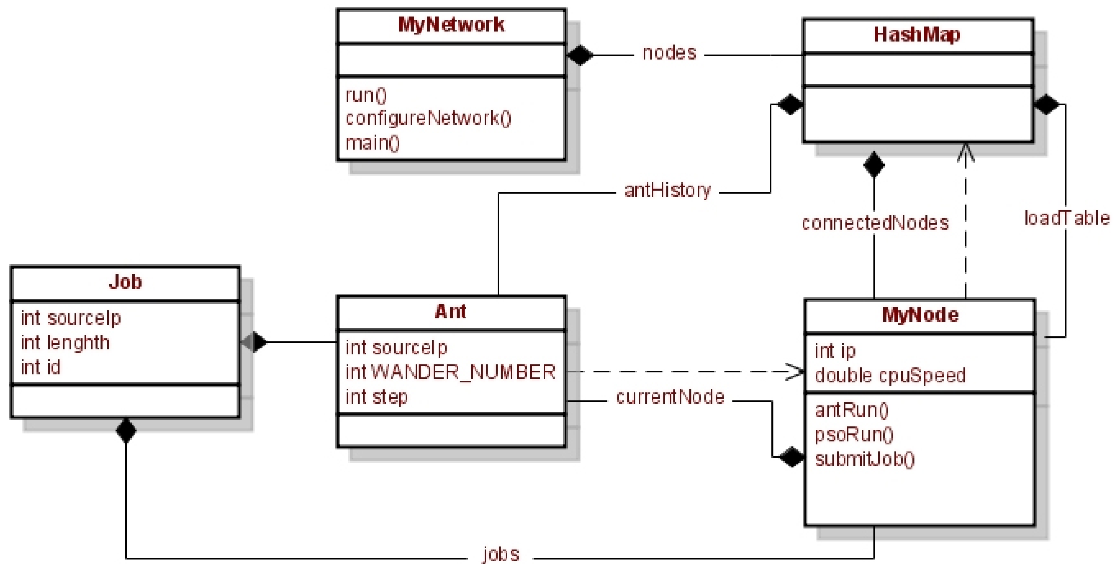


Figure 4.1: UML class diagram of the prototype design

each resource in the network, as it will have a huge burden on the system due to the large number of resources. We tried controlling resources by defining time stamps in the system to control the execution manually. Each node (resource) in the Grid will be provided with one time stamp and part of the jobs will be executed within the time provided. These time stamps are given to resources in a sequential order. The type of load balancing algorithm used can be defined in the class **MyNetwork**, accordingly, either *antRun()* or *psoRun()* is called from each resource and the simulation starts and continues by manually calling them at each time stamp. Obviously, Class **Ant** is only used for the AntZ algorithm. Each Ant moves from node to node leaving information about visited nodes in the *loadTables*. The procedure is exactly as we have described earlier. The preliminary experimental results which were driven from this first design showed satisfactory results about the effectiveness of both algorithms. Thus, in the next step we tried to simulate and run the algorithms in a more realistic environment. Therefore, we chose the **GridSim** toolkit which is a Java-based toolkit for Grid simulations.

4.3 GridSim Toolkit

One of the current and complete frameworks for simulating Grid-related algorithms or applications is the **GridSim** toolkit. In this section we are going to introduce GridSim and explore details of its architecture and the benefits of using it. In the next section, we will describe the design of our proposed strategies and how it fits into the GridSim toolkit architecture.

The GridSim toolkit is a java-based discrete-event Grid simulation toolkit. The toolkit supports modelling and simulation of heterogeneous Grid resources (time-shared and space-shared), users and application models. It also provides primitives for the creation of application tasks, mapping of tasks to resources, and their management [3].

A time-shared policy refers to a scheduling policy that shares time between running application tasks in a resource. An example of a time-shared policy is the Round Robin scheduling algorithm. In this scheduling scheme, a specific unit of time, called time slice or quantum, is defined. All executing processes are kept in a circular queue. The scheduler goes around this queue, allocating the CPU to each process for a time interval of one quantum. New processes are added to the tail of the queue. When a process is still running at the end of a quantum, the CPU is preempted and the process is added to the tail of the queue. If the process finishes before the end of the quantum, the process itself releases the CPU voluntarily. A disadvantage for this kind of scheduling is that every time a process is allocated to the CPU, a context switch occurs, which adds overhead to the process execution time. In [71, 70] the cost of context switching is analyzed in more detail.

On the other hand, a space-shared policy shares space (i.e. cpu space) between application tasks, so at each time only one application can run on one processing element. Examples of this scheduling policy can be First Come First Served, Shortest Job First, etc.

The GridSim toolkit supports the modelling and simulation of a wide range of heterogeneous resources, such as single or multiprocessor, shared and distributed memory machines like PCs, workstations, SMPs (Symmetric Multiprocessing), and clusters with different capabilities and configurations. It can also be used for the modelling and simulation of application scheduling on various classes of parallel and distributed computing systems such as clusters, Grids, and P2P networks [3].

There are some reasons why we chose the GridSim toolkit to simulate and evaluate our

scheduling algorithms are [3]:

- It allows modelling of heterogeneous types of resources.
- Resource capability can be defined in the form of MIPS (Million Instructions Per Second) and SPEC (Standard Performance Evaluation Corporation) benchmark.
- Application tasks can be heterogeneous and they can be CPU or I/O intensive.
- There is no limit on the number of application jobs that can be submitted to a resource.
- Network speed between resources can be specified.
- It supports simulation of both static and dynamic schedulers.
- Statistics of all or selected operations can be recorded. These statistics can then be further analyzed using GridSim statistics analysis methods.

A multi-layer architecture for the development of the GridSim platform and its applications is shown in Figure 4.2.

A brief description of each of the layers is given [3]:

- The first layer is concerned with what GridSim is based on with its scalable Java interface and the runtime machinery, called JVM (Java Virtual Machine), whose implementation is available for single and multiprocessor systems. The cJVM is a Java Virtual Machine (JVM) that provides a single system image of a traditional JVM while executing on a cluster [72]. SMP, refers to symmetric multiprocessing. It involves a multiprocessor computer-architecture where two or more identical processors can connect to a single shared main memory[1].
- The second layer is composed of a basic discrete-event infrastructure which is built using the interfaces provided by the first layer. One of the popular discrete-event infrastructure implementations available in the Java language is **SimJava** [73]. This infrastructure builds the basis of the GridSim toolkit. SimJava consists of many entities each of which are running in their own application thread. These entities

[1]http://en.wikipedia.org/wiki/Symmetric_multiprocessing

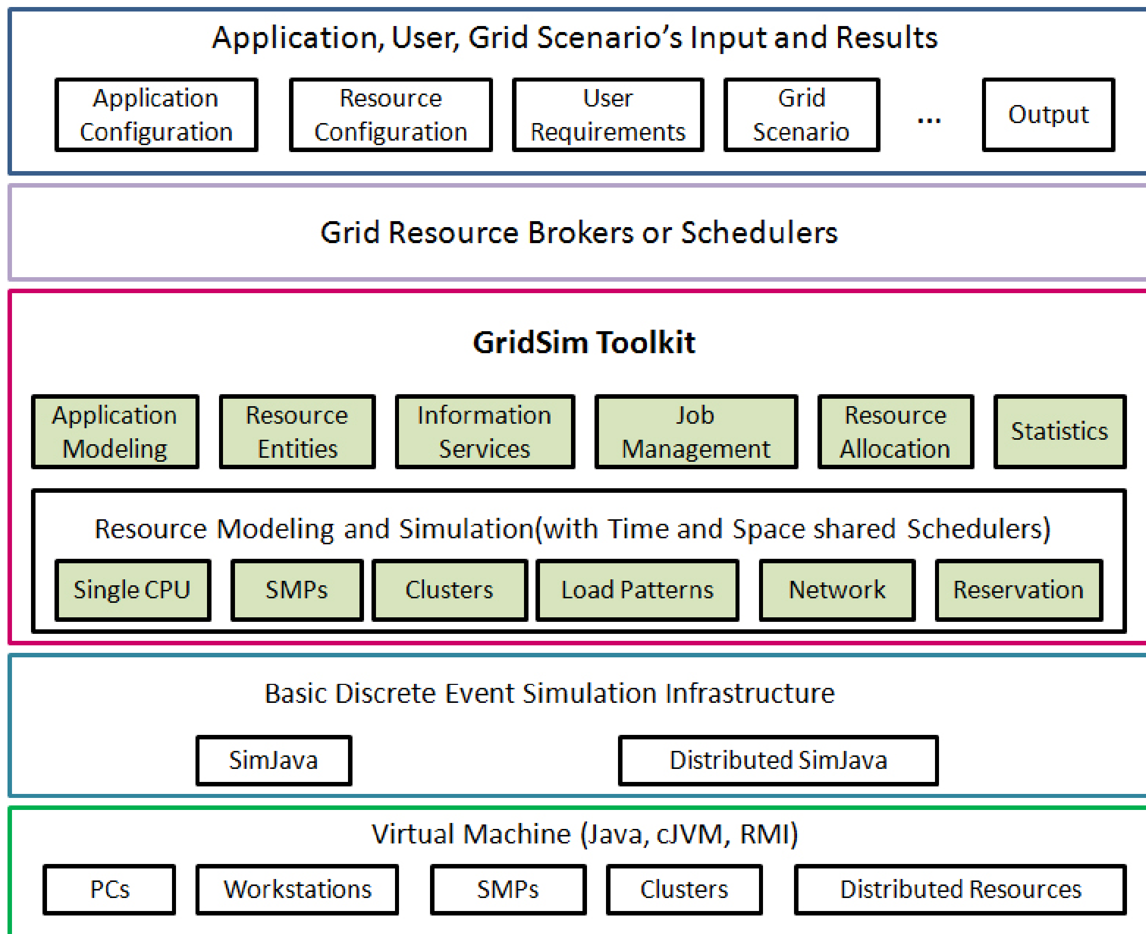


Figure 4.2: A modular architecture for GridSim platform and components

communicate with each other in an event-driven environment by sending and receiving messages to/from each other. There is a central system class which controls all the threads and advances the simulation time.

- The third layer is concerned with modelling and simulation of core Grid entities such as resources, information services, application tasks and so on. The GridSim toolkit focuses on this layer that simulates system entities using the discrete-event services offered by the lower-level infrastructure.

- The fourth layer is concerned with the simulation of resource aggregators called Grid resource brokers or schedulers. This layer will be our focus as we will implement our scheduling algorithms in this layer.

- The final layer focuses on application and resource modelling with different characteristics using the services provided by the two lower-level layers. It can be used for evaluating scheduling and resource management policies, heuristics and algorithms.

A typical Grid-based simulation contains entities which play the role of users, brokers, resources, information services, statistics, and network based I/O, as shown in Figure 4.3 [3]. Each of the entities in the figure have specifications in the GridSim environment. We briefly describe each entity and its responsibilities [3].

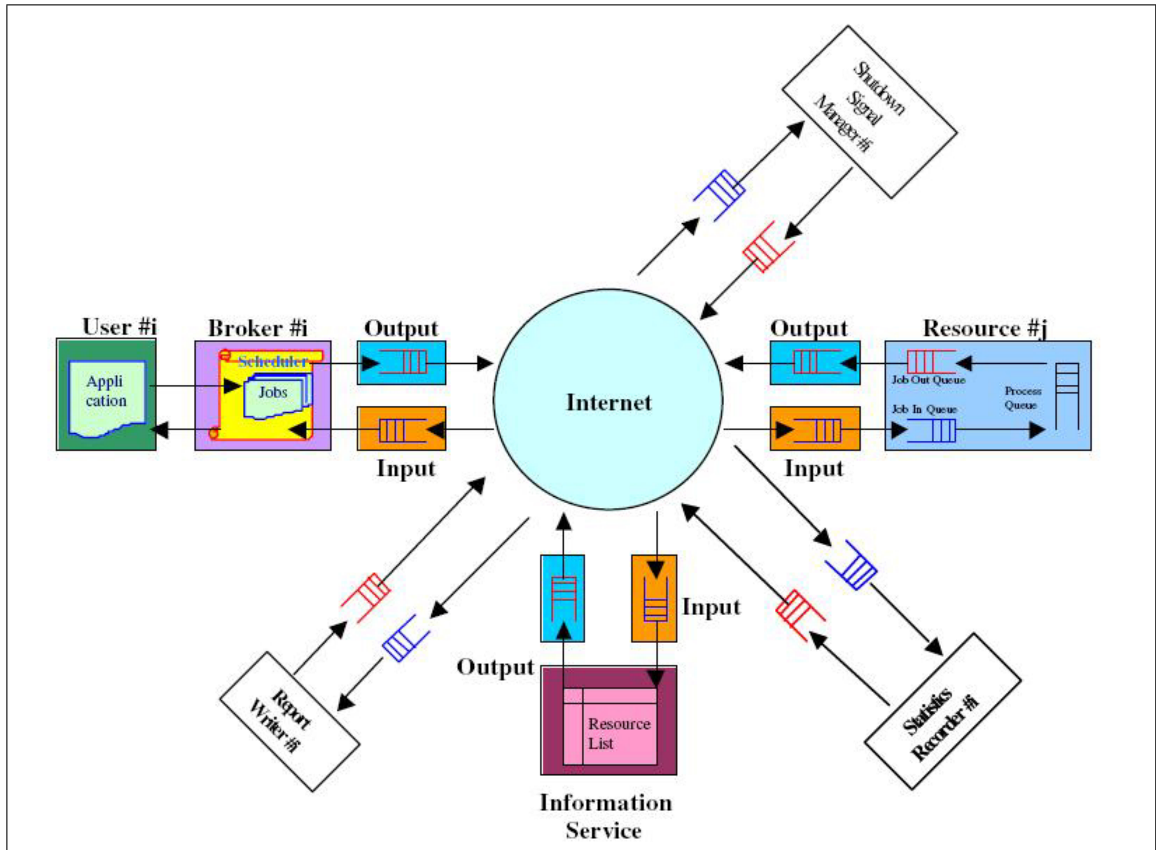


Figure 4.3: A flow diagram in GridSim based simulations [3]

User. Each instance of the User entity represents a Grid user.

Broker. Each user is connected to an instance of the Broker entity. Every job a user sends to the Grid is first submitted to its broker and the broker then decides how to schedule the parametric tasks according to the user's scheduling policy. Before scheduling the tasks, the broker can dynamically retrieve a list of available resources from the global directory entity.

Resource. Each instance of the Resource entity represents a Grid resource. Each resource has some special characteristics which differentiates it from other resources as follows:

- number of processors
- cost of processing
- speed of processing
- internal process scheduling policy, e.g. time-shared, space-shared, etc.
- local load factor
- time zone.

The resource speed and the job execution time can be defined in terms of the ratings of standard benchmarks such as MIPS and SPEC.

Grid information service. Provides services to enable resource registration while keeping track of a list of resources available in the Grid. The brokers can query this entity for resource contact, configuration, characteristics and status information.

Input and output. The exchange of information between the GridSim entities happens via their Input and Output entities. Every networked GridSim entity has I/O channels or ports, which are used for establishing a link between the entity and its own Input and Output entities. Note that the GridSim entity and its Input and Output entities are threaded entities, i.e. they have their own execution thread within the *body()* method that handles events.

4.3.1 GridSim Architecture

Figure 4.4 shows a UML view of the GridSim package design. A detailed description of the role of each of the entities can be found in the GridSim documentation [3]. Each class in the figure has three parts: attributes, methods and internal classes. Modifiers *public*, *private* and *protected* are indicated with "+", "-" and "#" respectively.

In order to simulate an application scheduling algorithm using GridSim, we have followed the steps below which are also suggested by the GridSim team [3].

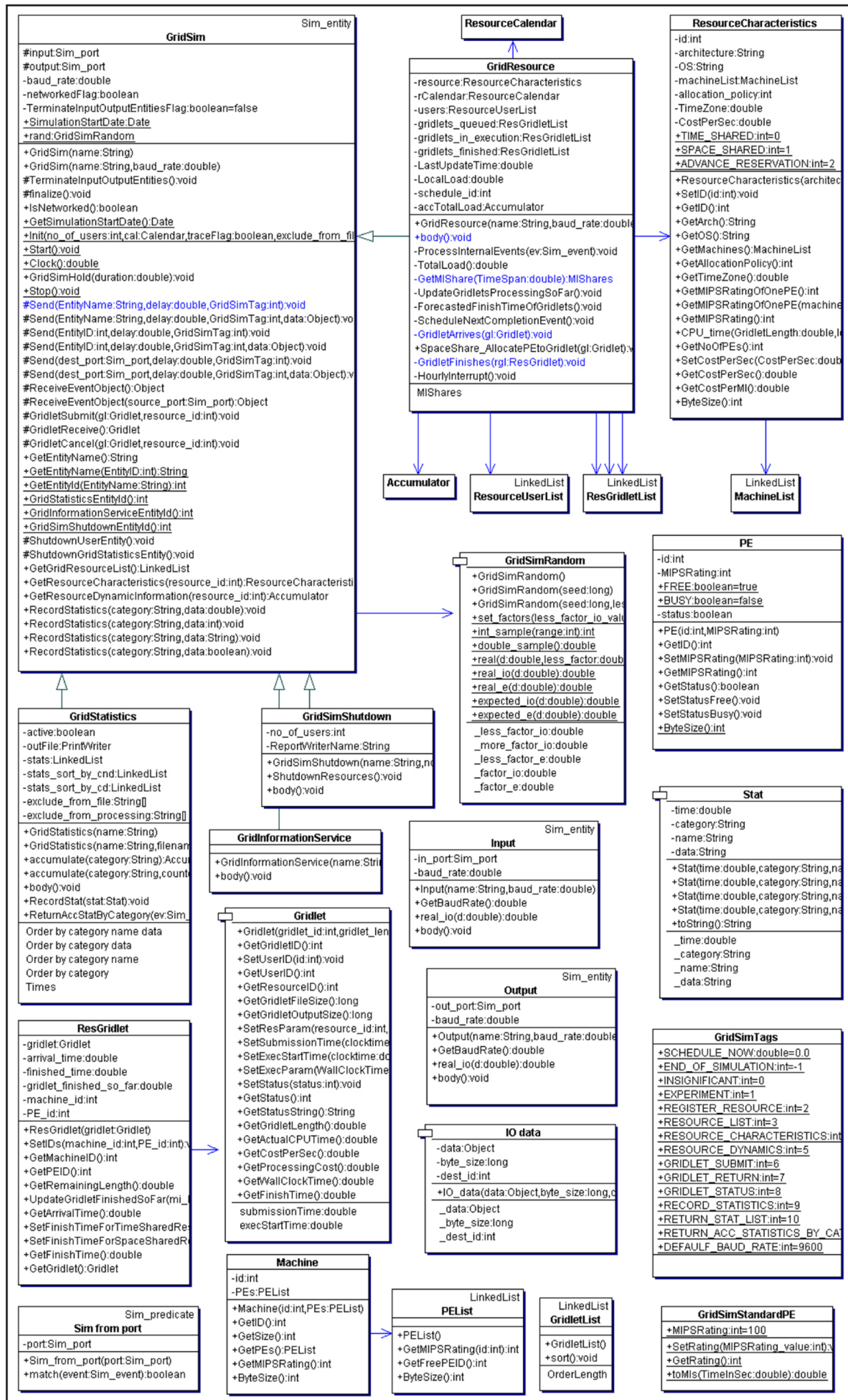


Figure 4.4: The UML diagram of the GridSim package design [3]

- Create Grid resources with different capabilities and configurations (i.e single or multiprocessor, with time/space-shared resource manager, connection links and their speed, etc.).

- Create users with different characteristics and requirements. Each user can submit jobs (Gridlets) with different characteristics and at different intervals. They can also have quality of service requirements.

- Create a GridSim user entity that creates and interacts with the resource broker scheduling entity to coordinate the execution experiment. It can also directly interact with the Grid information service entity and resource entities for acquiring Grid information and submitting or receiving processed Gridlets. However, the implementation of a separate resource broker entity is encouraged.

- Implement a resource broker entity that performs application scheduling on Grid resources. To do this, based on cost for example, access the Grid information service, and then inquire the resource capabilities including cost. Depending on the processing requirements, develop a schedule for assigning Gridlets to resources and coordinate the execution.

4.4 General Design

In order to implement the AntZ and ParticleZ algorithms following the guidelines the GridSim team proposes, we override the *AllocPolicy* class. In the design specifications of the GridSim each resource has an allocation policy attached to it. This extended class provides us with the main functionality needed to implement in our scheduling algorithms. Figure 4.5, depicts the UML class diagram of the design. Although there are many details for each class shown, we have omitted many classes from the actual class diagram to keep it simple by showing the most important ones in order to focus on the main concepts.

As can be seen in the figure, both classes which implement our scheduling algorithms are inheriting the **AllocPolicy** class. Developers must implement the body of the methods in AllocPolicy themselves according to their scheduling policy. On the other hand there is a class called **MyGridSimulator** which extends Class **GridSim** in the GridSim toolkit. This class creates all the resources and submits jobs to the Grid. Each **GridResource**

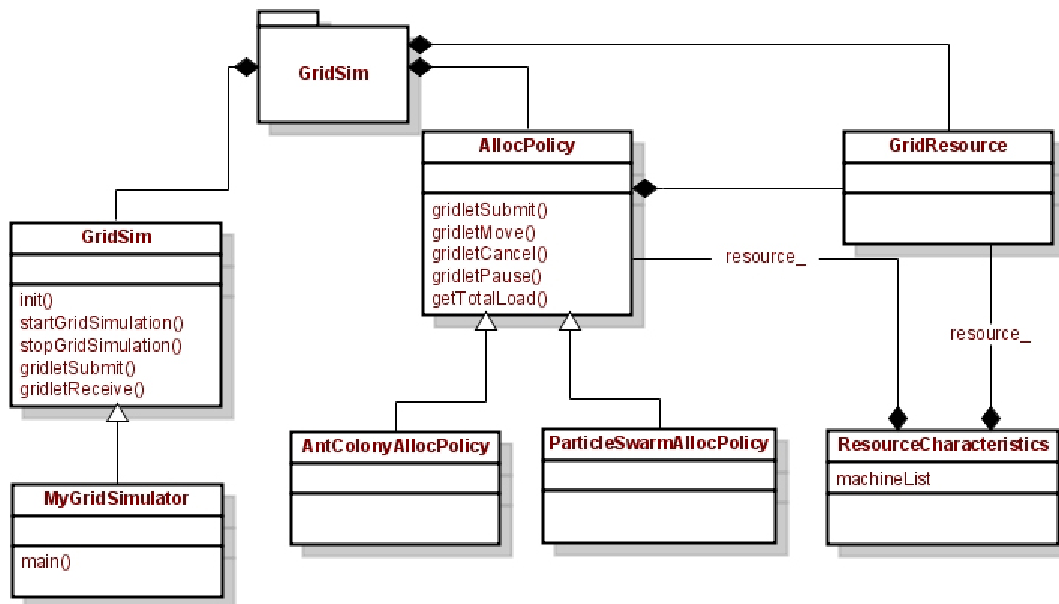


Figure 4.5: UML class diagram of the design

has an allocation policy which in our case can be either of the classes for the Ant colony or particle swarm policy. The characteristics of each Grid resource such as its processing elements etc. are provided in the **ResourceCharacteristics** class.

Each Grid resource is being created and initialized with a specific scheduling algorithm. Jobs are being sent to the Grid and they are delivered to their destination resources according to the scheduling algorithm defined for the system. Resources can be created using different attributes according to the simulation needs. The process of creating a Grid resource is as follows[1]:

1. Create PE (Processing Element) objects with a typical MIPS or SPEC rating.
2. Assemble created PEs together to create a machine.
3. Group one or more objects of the machine to form a Grid resource. A resource having a single machine with one or more PEs (Processing Elements) can be managed as a time-shared system using a round-robin scheduling algorithm. A resource with multiple machines is treated as a distributed memory cluster and can be managed as

[1] http://www.gridbus.org/gridsim/doc/api/

a space-shared system using the FCFS (First Come First Serve) scheduling policy or its variants.

4.5 AntZ Design and Implementation

In this section, we describe in detail how the ant colony scheduling is designed to work. As we described earlier, each scheduling algorithm can be implemented in GridSim by creating a new resource broker. For scheduling jobs on resources there are two issues involved. One problem is how to choose a resource among all the resources available in the Grid and the other problem is related to how to schedule assigned jobs to one resource on the CPU. By extending the **AllocPolicy** class, we have incorporated the ant colony scheduling, which tries to select best resources to deliver the job to, with the scheduling needed to coordinate tasks in one resource together within one class. Class diagram of **AntColonyAllocPolicy** can be seen in Figure 4.6. Some details are omitted from the class diagram to focus on more important attributes and methods. As can be seen in the figure, the **AntColonyAllocPolicy** class is inherited from **AllocPolicy** in GridSim. To manage the jobs which are assigned to one resource it uses a Round Robin scheduling algorithm; thus, whenever a job is submitted to a node it uses a Round Robin scheduling policy to execute them inside the node. In order to do this, it contains a list of executing jobs (*gridletInExecList_*). According to the Round Robin policy, jobs in this list get an equal time stamp to execute in a node. There is also a *loadTable* which the entries are filled by visiting ants and it acts as the pheromone the ants leave. Class **Ant** is an inner class of **AntColonyAllocPolicy** and uses its functionalities while the ants are moving from node to node. It has a small memory to carry a history of visited nodes and also the gridlet it is scheduling.

Figure 4.7, shows a sequence diagram which depicts how the ant colony scheduling works. A step by step description of the scenario is as follows:

1. **MyGridSimulator**, is simulating the jobs which are sent to the Grid. When MyGridSimulator sends a job to a **GridResource**, the resource delivers the received job to its scheduling policy to handle the incoming request accordingly.

2. In response to receiving a gridlet, **AntColonyAllocPolicy** creates a new **Ant** object and sends it out to explore the Grid and find a lightly loaded resource to deliver the

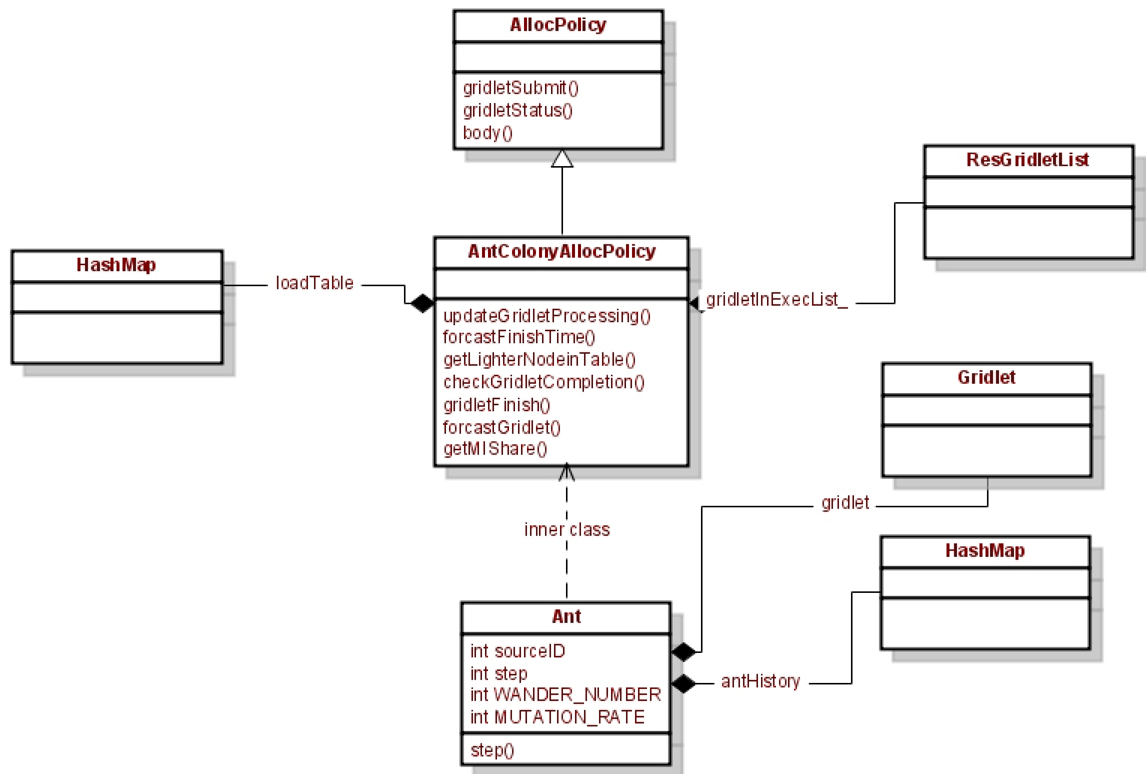


Figure 4.6: UML class diagram for the Ant Colony scheduling

job.

3. The ant takes steps as described earlier by collecting a history of visiting nodes (addHistory) and decides which step to take next by either reading the *loadTable* information in visiting nodes or by mutating and going to a random node. This probability that the ant may move randomly will prevent it from getting caught in a local minima. To choose a random node the ant needs to send a request to **GridInformationService** as this entity contains the information about the resources in the Grid.

4. When the ant chooses its next resource to move to, the gridlet and the ant will move to the destination node. The gridlet is submitted to the destination node to be scheduled again.

5. If this is the last step the ant has taken, the job will get executed and the result will be sent to **MyGridSimulator**. Otherwise the whole process is repeated again.

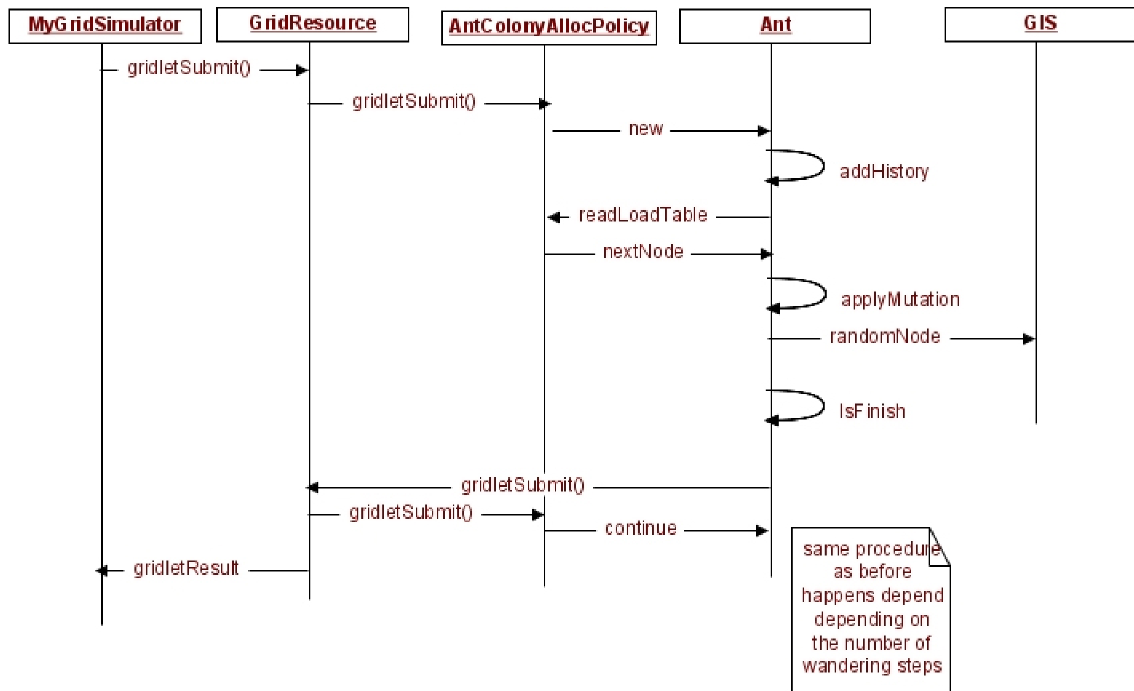


Figure 4.7: UML sequence diagram for the Ant Colony scheduling

As we said earlier the scheduling policy inside each node is implemented as a Time shared policy and more specifically the Round Robin policy.

4.6 ParticleZ Design and Implementation

In this section we describe in detail how the particle swarm scheduling is designed to work. As we described earlier each scheduling algorithm can be implemented in GridSim by creating a new resource broker. By extending the **AllocPolicy** class, again we have incorporated the particle swarm scheduling, with the scheduling needed to coordinate tasks in one resource within one class. The class diagram of **ParticleSwarmAllocPolicy** can be seen in Figure 4.8. Some details are omitted from the class diagram to focus on more important attributes and methods.

As can be seen in the figure, the **ParticleSwarmAllocPolicy** class is an extension of **AllocPolicy** in GridSim. It has a space shared FCFS (First Come First Served) scheduling algorithm inside, meaning that whenever a job is submitted to a node it is scheduled with

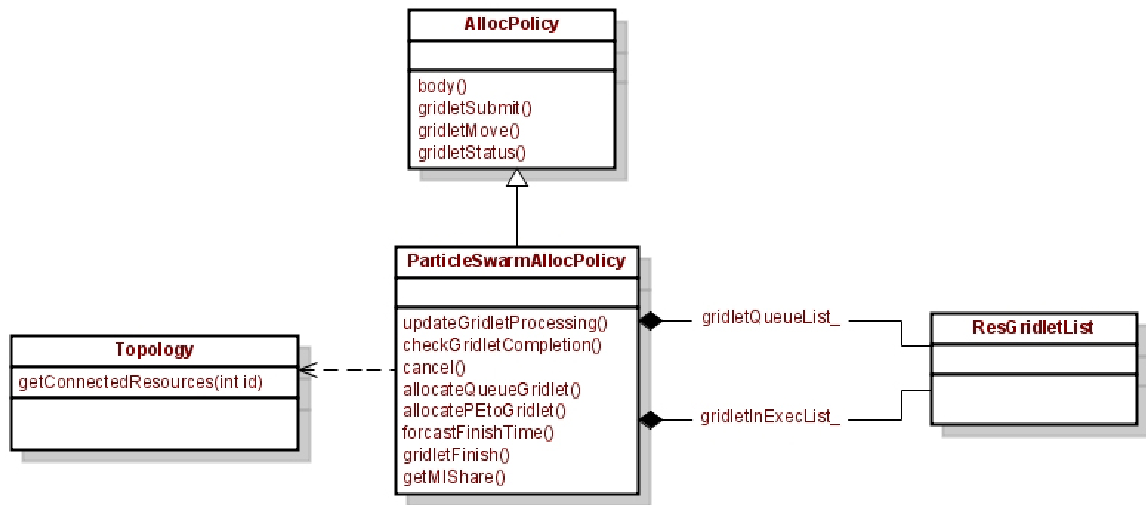


Figure 4.8: UML class diagram for the Particle Swarm scheduling

an FCFS scheduling policy inside the node. As FCFS is a space-shared scheduling policy when there are more than one PEs available in the machine more than one gridlet can be executed at the same time. **ParticleSwarmAllocPolicy**, has a list of jobs being executed (*gridletInExecList_*) and also a list of jobs which are waiting in the queue to find a free PE to get executed (*gridletQueueList_*). The jobs waiting in the queue are actually the jobs which can be delivered to other resources while the scheduling algorithm is running to balance the load.

As mentioned earlier, particle swarm optimization works based on best neighbouring particles. The information related to the topology of the network in the simulation is kept in class **Topology**. By querying this class, we can find the neighbours of one resource. In order to make a random connection graph for the specific number of resources we have, we first create a *Minimum Spanning Tree* with all the resources; then, we randomly add some links to the tree to generate the final topology of the Grid. Thus, we can have control on the number of links and the topology of our Grid in different simulations. Thus, when a resource tries to find its neighbours it sends a message to the **Topology** class to retrieve a list of its connected resources.

Figure 4.9, shows the sequence diagram of a typical run of the ParticleZ algorithm. This sequence diagram shows the case when a Gridlet is sent to a node in the Grid and a PE is immediately allocated to it, hence, the gridlet can right away run on the node it is

sent to.

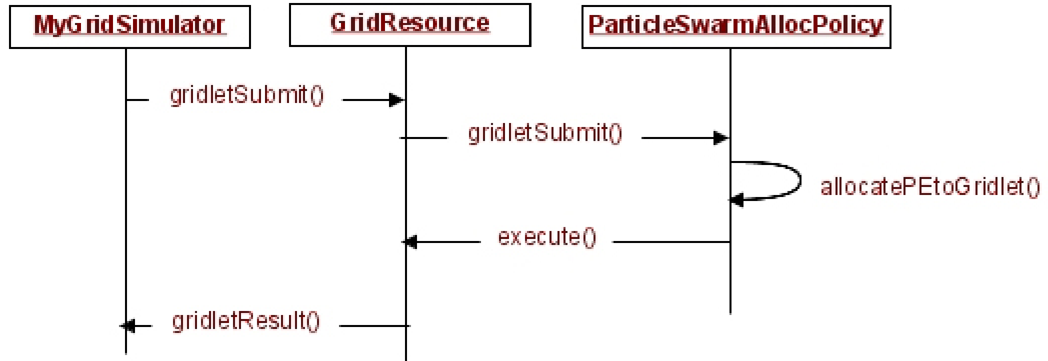


Figure 4.9: UML sequence diagram for the Particle Swarm scheduling

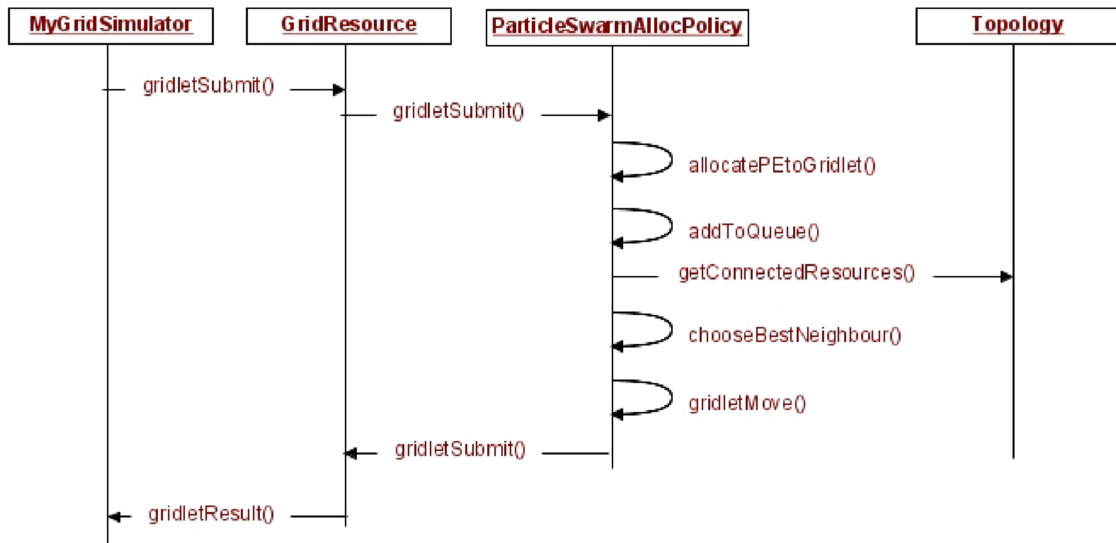


Figure 4.10: UML sequence diagram for the Particle Swarm scheduling

Figure 4.10, shows another scenario in which there is no free PE to be allocated to the gridlet sent in the first step. In these cases the gridlet will be added to a queue. On the other hand, each resource is exchanging information with its neighbours to find the best and lightest loaded neighbour; once found, the resource cancels the gridlet and removes it from the queue and moves it to another node for execution by resubmitting the gridlet. Once the gridlet is executed the results will be sent back to the sender, which in this case

is the **MyGridSimulator** class.

4.7 Summary

In this chapter, we provided detailed specifications of GridSim and described how the algorithms were developed using the provided GridSim architecture. The next chapter focuses on running some simulations in different scenarios to evaluate the performance of the proposed algorithms. Furthermore, we investigate the effect of different settings for different parameters on the performance of the algorithms.

Chapter 5

Experimental Results

5.1 Overview

This chapter is dedicated to experimental results and measurements which are a key factor in evaluating each simulation. In the first step, we provide information on how the environment setting is chosen and how the Grid is constructed. We describe the application model next, which discusses the characteristics of the jobs which are sent to the Grid and are used to run the experiments. The performance evaluation criteria which are used to evaluate the performance of the algorithms are introduced consequently. We also provide details about two classical algorithms (Random and State Broadcast Algorithm) which have been used to evaluate and compare the performance of our algorithms with. Finally, experimental results and diagrams are provided with a thorough analysis about each of them.

5.2 System Model

For our experimental purposes we assume that the Grid consists of a set of resources connected via different communication networks with different speeds. In general, each resource may contain multiple number of computing nodes (machines), and each computing node (machine) may have single or multiple Processing Elements (PEs). The computational power or the speed of each processor is defined by the number of Cycles Per Unit Time (CPUT). It is actually the GridSim framework's ability that provides us with the definition of the computational power of PEs in CPUT.

Generally, each resource may consist of one or several machines and each machine by itself can have one or multiple processing elements. Processors in each computing node can be heterogeneous, thus, they may have different processing power. In our simulations,

without loss of generality and to emphasize on the basic ideas of the algorithms, we assume each resource consists of one machine and each machine is equipped with one or several processors (the variations of this random number for experiments will be provided later). The processors in the same or different computing nodes have different processing power.

A computing node in the Grid may also have a local user (or multiple local users) that uses the node for other computations (that is, the node is not a dedicated node). As such, at any one time, a computing node may have background workload associated with it, which will affect the completion time of the Grid jobs assigned. The GridSim provides us with the ability to define the background workload according to historical and statistical information for each node. As such, each resource has a background load associated which is taken from the average load that the resource has experienced at similar times (such as working days or weekends).

5.3 Application Model

For our application model, we assume that tasks which are submitted to the Grid (or the application which is being run) consists of a set of independent tasks with no required order of execution. The tasks are of different computational sizes, meaning each task requires a different computation time and data transmission time for completion. They can also have different input and output size requirements.

The length of each task is presented in Millions of Instructions (MI). Tasks can be classified into one of two categories: data intensive and computationally intensive tasks. In this research, we are concerned with computationally intensive tasks as they are more common in todays real life applications (like some of bio informatics problems, etc) and the waste of computational power of resources is more costly than their memory.

Some researchers have considered job migration (migration of partly executed jobs) in their load balancing algorithms. However, as discussed in Chapter 2, job migration is far from trivial in practice. Thus, in this research, we do not consider migration of partly executed jobs.

5.4 Performance Evaluation Criteria

In this section we define our performance evaluation criteria which are used to evaluate the performance of our algorithms. The criteria include makespan, load, standard deviation and load balancing level. A description of each will follow.

5.4.1 Makespan

One of the most common measures in evaluating the performance of an scheduling algorithm is measuring the makespan. The makespan is the "total application execution time". The total application execution time is measured from the time the first job is sent to the Grid, until the last job comes out of the Grid. As we generate gridlets and topologies randomly, although every simulation yields roughly the same result, each single simulation is different from another one; thus, we have used an average makespan in order to simulate realistic conditions. We have used an average of ten runs in order to take care of the small variations of the results of each run.

5.4.2 Load

For each resource in the Grid, the load related to that resource is dependent on the number of jobs which are assigned to the node by the Grid scheduler and the power of its processing elements. Equation 5.1, shows how the *GridLoad* is calculated.

$$GridLoad = \frac{AssignedJobNumber}{\sum_{i=1}^{MaxNumberofPE} powerofPE(i)} \tag{5.1}$$

The total *load* can be calculated using the Equation 5.2. For the experiments, our aim is to minimize this value. According to this equation when *GridLoad* increases it results in an increase in *load* and a decrease in *GridLoad* decreases *load.* The *load* is a value between 0 and 1, where 0 is not busy and 1 represents being busy.

$$load = 1 - \frac{1}{GridLoad} \tag{5.2}$$

5.4.3 Standard Deviation

One of the aims of a load balancing algorithm is to minimize the variations in workloads on all machines. Regarding this, standard deviation in workload is often taken as the

performance measure of a load balancing algorithm. The smaller the standard deviation, the better the load balancing scheme is. By looking at the changes in the standard deviation of the workload with respect to time, it is easier to visualize the effect of load balancing upon the time of the system [41]. Equation 5.3, shows the standard deviation of the load in the system.

$$d = \sqrt{\frac{\sum_{i=1}^{n}(\overline{load} - load_i)^2}{n}} \tag{5.3}$$

In the equation, $\overline{load}$ is the average load of the system and $load_i$ is the load of the *ith* resource at each point in time.

5.4.4 Load Balancing Level of the System

We define the *load balancing level* of the system to be a measure of how good a load balancing algorithm is. The *load balancing level* of the system is defined in Equation 5.4. The most effective load balancing is achieved when d equals to zero and the *load balancing level* equals to 100%.

$$LoadBalancingLevel = (1 - d) * 100\% \tag{5.4}$$

5.5 Comparison Against Classical Approaches

We have implemented two common classical approaches (Random and State Broadcast Algorithm) in order to evaluate the performance of our algorithms and discuss their benefits over classical ones.

The **Random** approach is a simple scheduling algorithm in which the jobs being sent to the Grid are assigned randomly to different resources. Obviously this approach does not make a very good scheduling algorithm but it has some benefits. It does not have any decision making overhead on the system on the other hand it gives a good benchmark to see how our proposed algorithms improve the performance of scheduling compared to a plain random assignment.

The other approach we use to evaluate the performance of our algorithms is the **State Broadcast Algorithm (SBA)**. This algorithm is common in networks whose communication system consists of a broadcast medium. As described in Chapter 2, the algorithm is

based on broadcast messages between resources. Whenever the state of a node changes, due to the arrival or departure of a task, the node broadcasts a status message that describes its new state. This information policy enables each node to hold its own updated copy of the system state vector (SSV) and guarantees that all the copies are identical.When a job is sent to a resource at the time of scheduling, the resource searches through its own state vector to find the best resource available to deliver the job at that particular time. SBA is a good benchmark to evaluate the performance of our algorithms as it resembles central approaches in which the status of the whole Grid is known at the time of scheduling although being a distributed approach. SBA performs like central approaches, which by nature always outperform distributed ones [25], however, it has its disadvantages which will be described later.

5.6 Experimental Results

In order to evaluate the performance of our algorithms we investigate a set of experiments to measure the criteria we introduced in the previous section and also investigate the effect of different values for the parameters of each algorithm. As described earlier, ParticleZ is implemented with a space shared FCFS policy inside the resources and the AntZ is accompanied with a time shared Round-robin policy to schedule the jobs when they are received by a resource. In all the experiments we have compared our algorithms with both the Random and the SBA approach in order to have an understanding of how well they perform.

The characteristics of the resources we have used as Grid resources are shown in Table 5.1. There is one machine for each Grid resource and each machine has a random number of PEs ranging between 1 and 5. Each PE has a different processing power. Without loss of generality, we set the local load factor for resources to be zero; this does not affect the performance measure of the algorithms. Setting it to zero helps us analyze the effect and behaviour of the algorithms better.

For the first set of experiments, we compare the makespan of the different algorithms. Different values for the different parameters in each of the algorithms and system parameters are shown in Table 5.2.

As said earlier, the gridlets which are sent to the Grid are supposed to be independent

Number of machines per resource	1
Number of PEs per machine	1 - 5
PE ratings	10 or 50 MIPS
Bandwidth	1000 or 5000 B/S

Table 5.1: Grid resource characteristics

Number of resources	100
Number of gridlets	1000
ParticleZ link number	149
AntZ wander number	4
AntZ mutation rate	0.5
AntZ decay rate	0.2

Table 5.2: Scheduling parameters and their values

of each other. The characteristics of the gridlets sent to the Grid to compare the makespan of different algorithms are shown in Table 5.3.

Length	0 - 50000 MI
File size	100 + (10% to 40%)
Output size	250 + (10% to 50%)

Table 5.3: Gridlet Characteristics

Figure 5.1 shows a comparison between the makespan of different algorithms with parameter specifications described earlier. As the experimental results show SBA is performing best amongst all. This is expected as the SBA is keeping track of the state of all the resources at each point in time which makes it able to make more optimal decisions at each point in time. After SBA, ParticleZ has the smallest makespan. Comparing ParticleZ and AntZ with each other, ParticleZ performs better than AntZ by a factor of 1.72; also, ParticleZ performs better than Random-SpaceShared by a factor of 3.42, and AntZ

performs better than Random-TimeShared by a factor of 1.83.

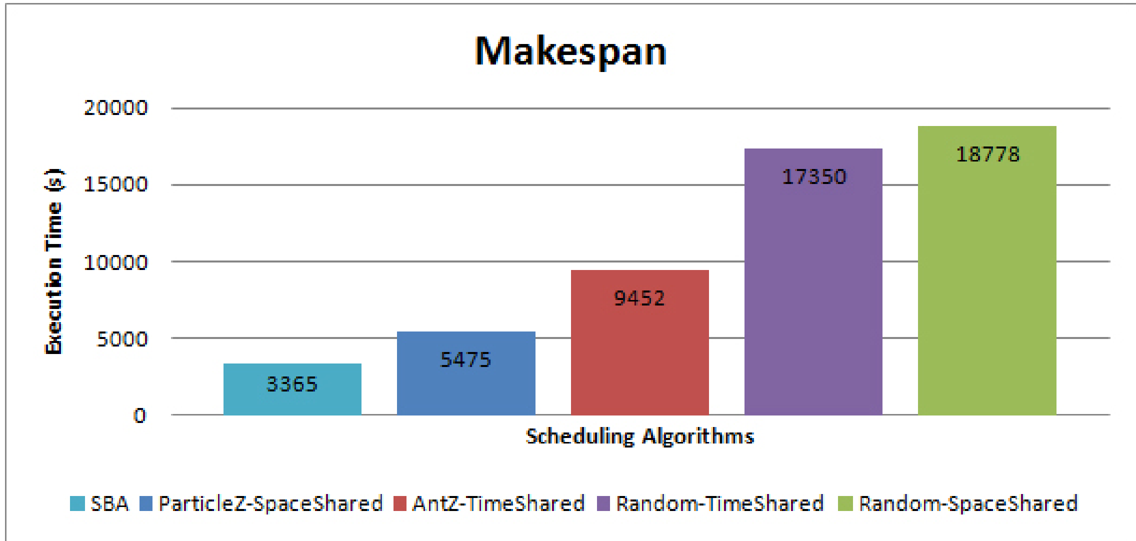


Figure 5.1: Comparing the makespan of different approaches

One important but hidden drawback that SBA suggests is related to the overall effort or overall cpu cycles and the time it takes for the Grid to execute it. As there is a copy of the system state vector in all machines, in order to schedule each task, each machine is using some time and cpu cycles to search the state vector individually. This causes a lot of cpu cycles to be wasted but as we are running a parallel platform this disadvantage can not be seen. In order to highlight its effect see Figure 5.2.

The time shown in this figure shows only the simulation time of each algorithm. As the simulation is being done on a single machine the effect of parallelism is discarded and as can be seen SBA, although having a very low makespan actually takes longer to run and it is because all these wasted seconds can not be seen in the previous figure because of parallelism. This effect is even worse as the number of resources grow in the Grid. Note that this figure shows only the simulation time and does not count for different job lengths and etc.

Another drawback related to SBA is the number of communications it takes. Figure 5.3 shows the number of extra communications of each algorithm to achieve the load balancing. For ParticleZ, each communication message a node sends to its neighbours to acquire their load status and its response; also, each job exchange between two resources is considered as a communication. For AntZ, each ant taking a step while searching for the best node

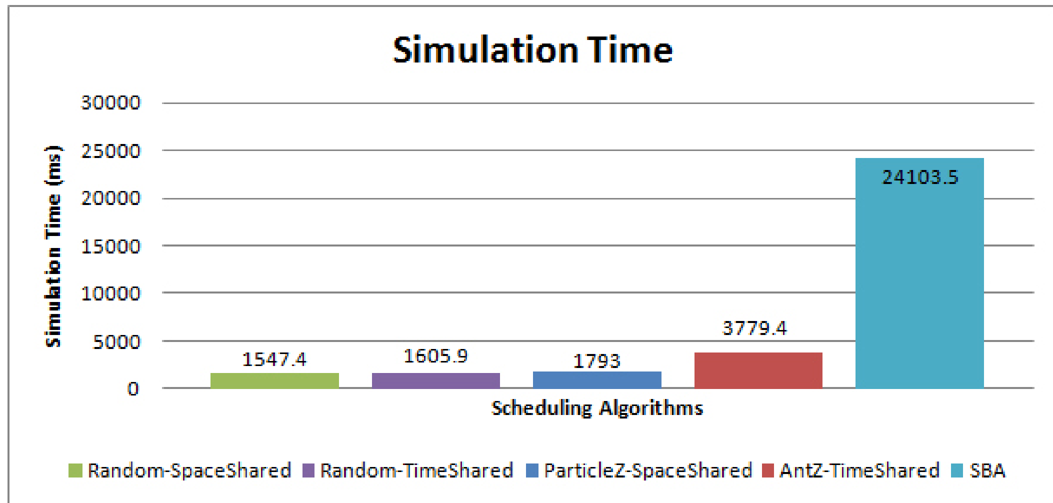


Figure 5.2: Simulation time related to each algorithm in milliseconds

to deliver the job to, is considered as a communication. Finally for SBA, each broadcast message a resource sends to other resources is considered as communication overhead.

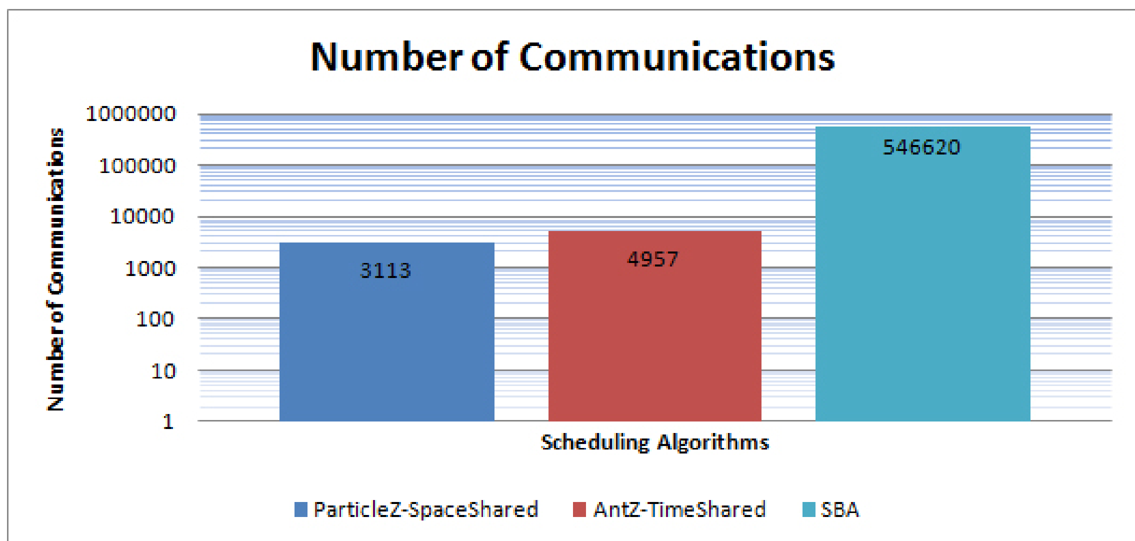


Figure 5.3: Communication overhead related to each algorithm

The numbers shown in the figure are the average of ten runs with the same parameter setting as described earlier. As shown in the figure, AntZ has a higher number of communication overhead compared to ParticleZ. Obviously, the other two random approaches have no communication overhead at all, thus, they are not shown in the figure. SBA has

the highest number of communications by a factor of around 1300. This huge number of communications can be a bottleneck for the network and in scenarios with congested networks the probability of messages being lost increases.

In the next experiment we investigate how fair each of the algorithms is. Table 5.4 shows the load balancing level of the system described earlier in Equation 5.4 along with their standard deviation from several runs. The closer the value approaches 100%, the better the load balancing level of the algorithms is. It means that the load is spread more fairly among all the resources. According to the experimental results both ParticleZ and SBA have the best load balancing levels. AntZ along with the other random approaches rank third in spreading the load uniformly among resources.

Algorithm	**Average Load Balancing Level**	**Standard Deviation**
ParticleZ-SpaceShared	81%	2.1%
SBA	80%	0.48%
Random-SpaceShared	67%	1.3%
AntZ-TimeShared	65%	0.7%
Random-TimeShared	62%	0.97%

Table 5.4: Average load balancing level of the system for different algorithms

In the next set of experiments we investigate the effect of increasing the number of jobs on the performance of the algorithms. Thus, we keep a fixed number of resources and run the experiments while we increase the number of jobs being sent to the Grid. The specifications and parameter settings of the algorithms and the system are listed in Tables 5.1 to 5.3.

As can be seen in Figure 5.4, all the algorithms show a linear growth in response to the increasing number of jobs. However, SBA along with the proposed approaches show a much smoother growth compared to the random approaches. Among them ParticleZ and SBA are quite close to each other. Table 5.5 shows each algorithm with its prediction trend line for the 100 node Grid. As can be seen, ParticleZ and SBA have the smallest slope among all other approaches.

In Figure 5.5, we investigate the effect of increasing the length of jobs on the perfor-

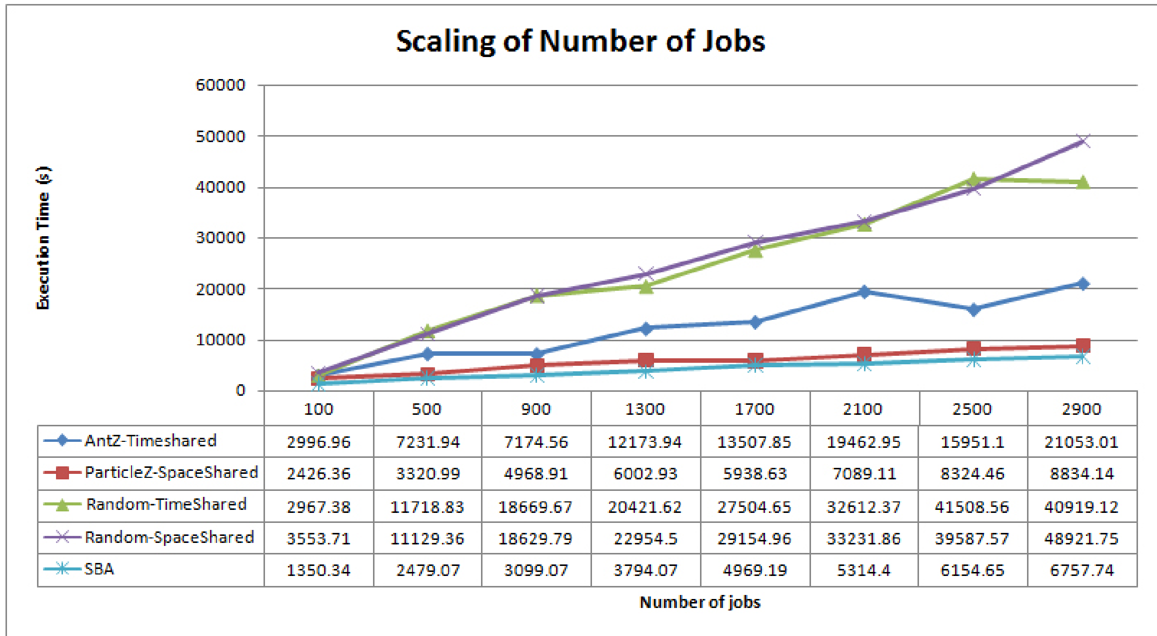


Figure 5.4: Effect of the increase in number of jobs on performance of the algorithms

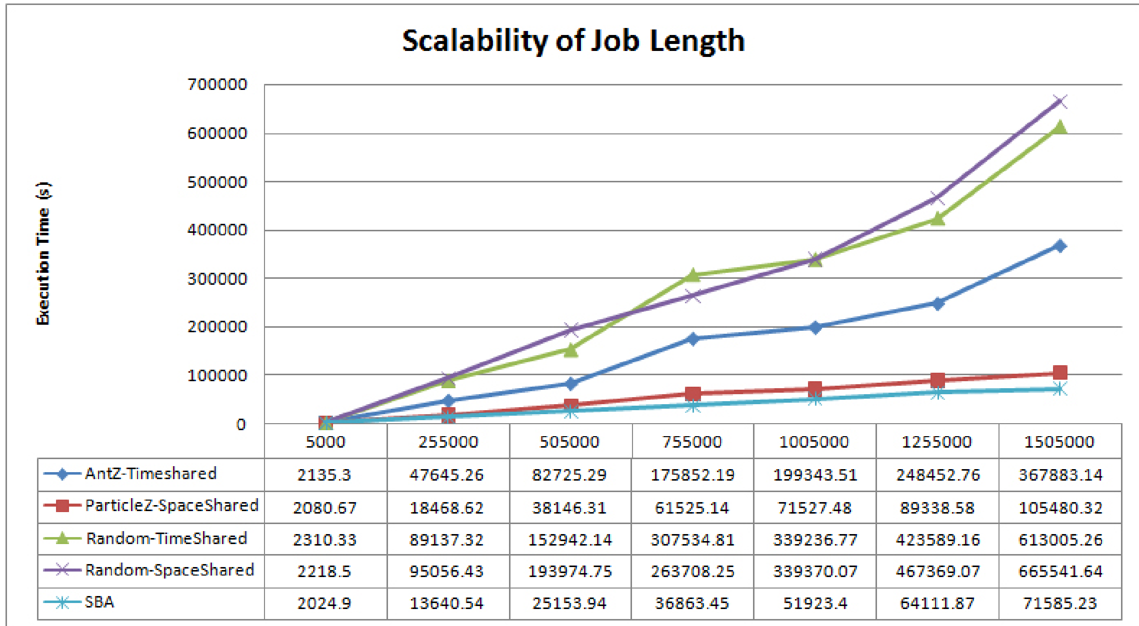
mance of the algorithms. Length of the jobs is defined in Millions of Instructions (MIs) in GridSim.

Algorithm	**Prediction trend line**
SBA	762.5 * Number of Jobs + 808.5 [s]
ParticleZ-SpaceShared	906.7 * Number of Jobs + 1782 [s]
AntZ-TimeShared	2478 * Number of Jobs + 1291 [s]
Random-TimeShared	5518 * Number of Jobs - 291.2 [s]
Random-SpaceShared	6069 * Number of Jobs - 1419 [s]

Table 5.5: Predicting execution time based on number of jobs

Parameter settings to run this experiment are the same as described in Table 5.1 to 5.3. We increase the length of the gridlets by adding 250,000 MIs at each step and investigate its effect on the makespan. The numbers at the bottom of the diagram show the execution time for each algorithm. As can be seen the growth is linear for all the approaches and the results show the best performance is acheived by both the ParticleZ and SBA algorithm.

AntZ ranks third and the other two random approaches as expected do not respond well to the larger lengths of gridlets but for small gridlet lengths they can perform comparably to others.



	5000	255000	505000	755000	1005000	1255000	1505000
AntZ-Timeshared	2135.3	47645.26	82725.29	175852.19	199343.51	248452.76	367883.14
ParticleZ-SpaceShared	2080.67	18468.62	38146.31	61525.14	71527.48	89338.58	105480.32
Random-TimeShared	2310.33	89137.32	152942.14	307534.81	339236.77	423589.16	613005.26
Random-SpaceShared	2218.5	95056.43	193974.75	263708.25	339370.07	467369.07	665541.64
SBA	2024.9	13640.54	25153.94	36863.45	51923.4	64111.87	71585.23

Figure 5.5: Effect of the increase in job length on performance of the algorithms

Figure 5.6, shows how increasing the number of resources, while having the same number of jobs being sent to the Grid, improves the performance of the Grid in terms of execution time. In this experiment, 3000 jobs are sent to the Grid with varying number of resources, and as can be seen increasing the number of resources has a decreasing exponential effect on the execution time. ParticleZ and SBA are performing better when we have a small number of resources (50) and a large number of jobs compared to the number of resources (3000). As the number of resources increases the performance, the difference between the algorithms drops.

One of the very interesting performance questions which arises in a distributed algorithm like AntZ and ParticleZ is: how the algorithms respond if all the jobs are injected from a single point in the Grid. From the AntZ's perspective it will take longer to build the load table information and from the ParticleZ's perspective it will have a negative effect as the jobs will need more time to be spread fairly. By incorporating some randomness in

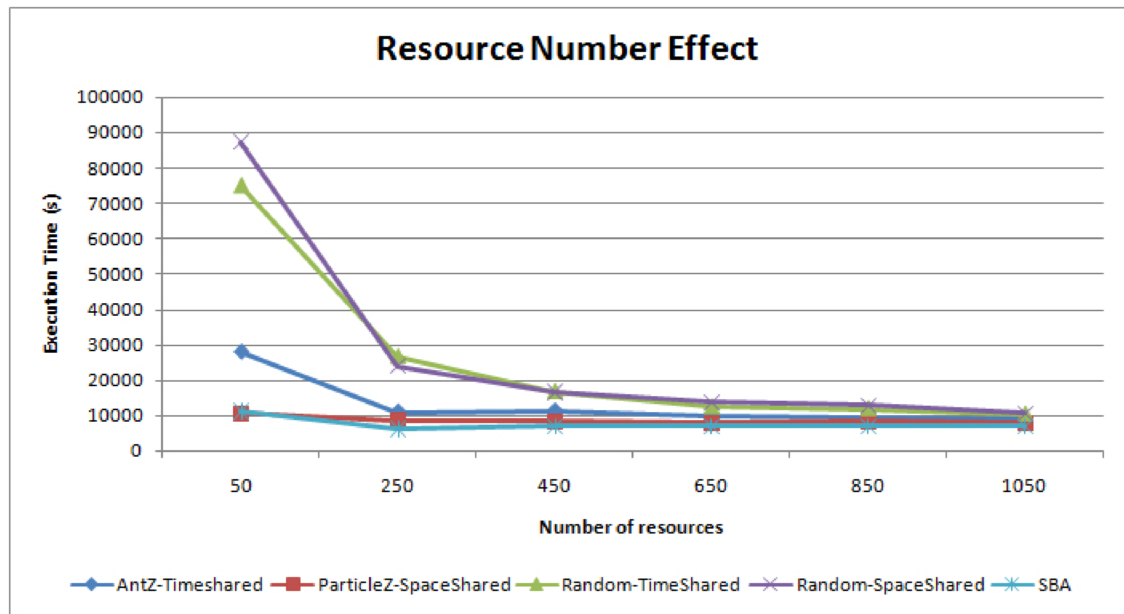


Figure 5.6: Effect of increasing number of resources on execution time

the nodes being chosen during the decision making precess, this bad effect can be reduced in both algorithms. We have investigated this effect to see how much it will slow down or have a negative effect on the performance of the algorithms.

The random approaches obviously will perform very poorly if we send all jobs to one node. Figure 5.7 shows AntZ copes better than ParticleZ in response to sending all the jobs to one node in the Grid. The reason lies in the mutation factor which is incorporated inside AntZ. With the mutation, an ant moves randomly from time to time which helps in building up the load tables more quickly to overcome the negative effect. It can be inferred from the figure, that ParticleZ's performance decreases by a factor of 2.4 for a one hundred node network with gridlets of a length between 0 and 50000. On the other hand, AntZ's performance decreases by a factor of 1.36 in the same scenario setting.

5.6.1 AntZ Parametric Measurement Effects

Now that we have a good understanding of how well the algorithms work in comparison and in different kinds of parameter settings, we investigate algorithm specific performance measures and their effect on the algorithms in the next set of experiments.

First, we investigate the effect of wandering steps on the performance of the AntZ

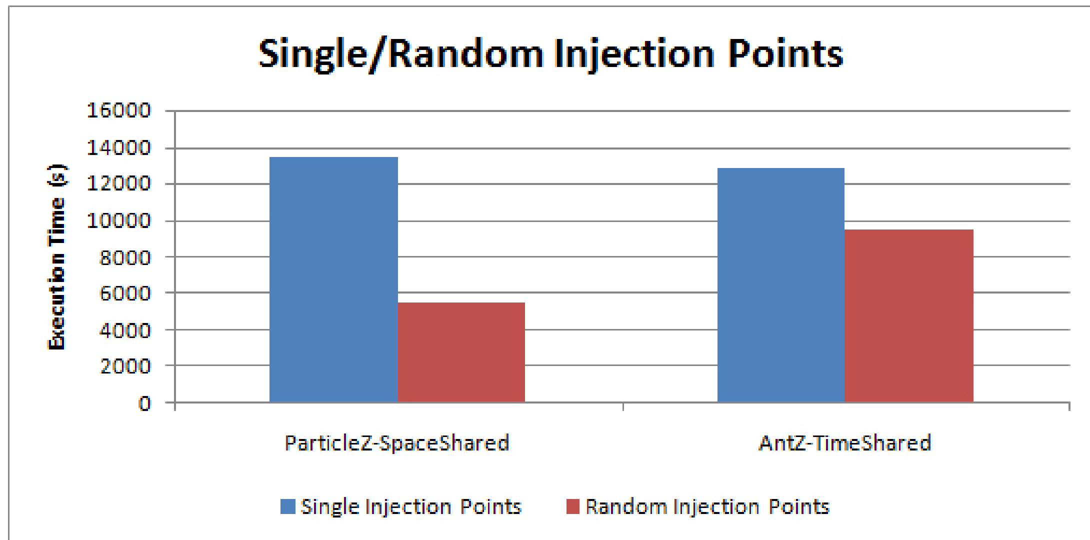


Figure 5.7: Effect of single and random injection points on the performance of the algorithms

algorithm. We have a one hundred node Grid with one thousand jobs being sent to the Grid. Figure 5.8 shows that as we increase the number of steps an ant wanders until it delivers the job to its destination, the makespan of the algorithm improves, but this increase is larger at the beginning but later on the rate drops to a great extent.

After about 5 or 6 steps the increase in wandering steps does not seem to have an effect on the performance of the algorithm. The reason behind this phenomenon is that although increasing the number of wandering steps seems to have a positive effect on the performance of the algorithms as tables will be updated more frequently and ants have more time to decide which way to go; but on the other hand, it increases the delay before the jobs are being delivered to resources and this delay has a negative effect on the performance.

Figure 5.9, shows how increasing the number of wandering steps can effect the communication overhead which are introduced to the system. The figure shows that while we increase the wandering steps, the communication overhead also increases linearly. In another experiment we measure how different values of decay rate can effect the performance of the AntZ algorithm. As you remember, while the ant is moving we decrease its mutation rate by a factor; this factor is called the decay rate. By doing this experiment we can find out what the best decay rate value for a set of specific attributes of a Grid and its jobs is. The results are shown in Figure 5.10. For the set of attributes we have, 0.2 is the best

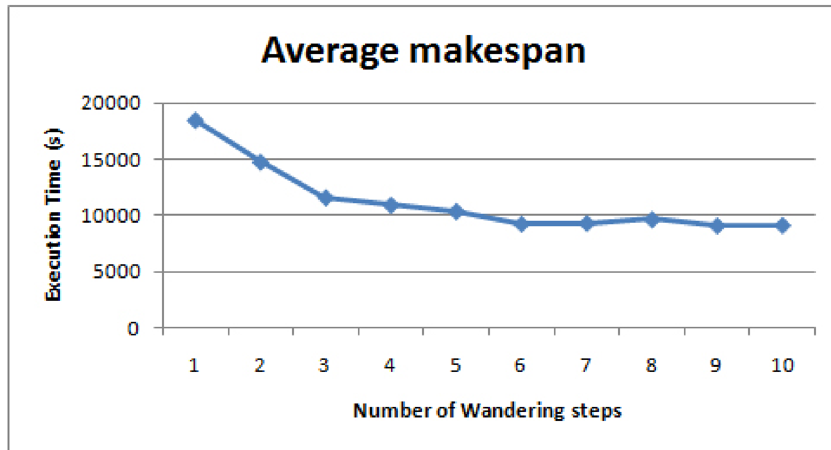


Figure 5.8: Effect of the change in wandering steps on AntZ makespan

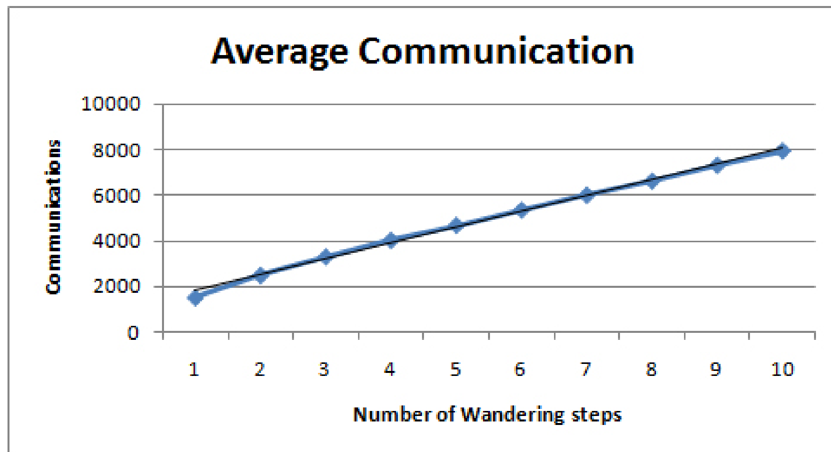


Figure 5.9: Effect of the change in wandering steps on AntZ communication number

decay rate while the mutation rate is set to be 0.5 for this experiment.

5.6.2 ParticleZ Parametric Measurement Effects

In the next set of experiments we will measure the effect of different ParticleZ parameter settings on the performance of this algorithm. One of the parameters which can affect the performance of ParticleZ is the number of links that connect resources together. As each particle (resource) communicates with its neighbours to find the lightest node, the number of neighbours can effect the performance of the algorithm.

Figure 5.11 shows the effect of increasing the number of links and the connectivity of

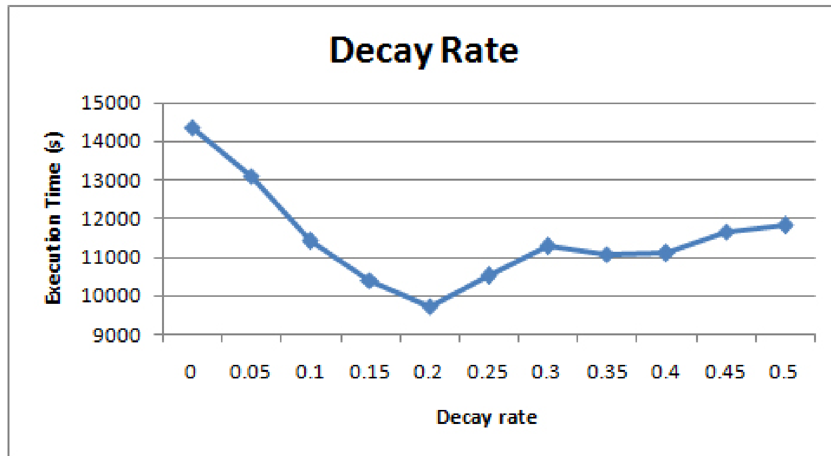


Figure 5.10: Effect of decay rate on AntZ makespan

the resources on ParticleZ's makespan. Although it is better to communicate with more resources before exchanging jobs, however, it is not always good as communicating with more resources has an extra time overhead which prevents a significant improvement in the performance of the system.

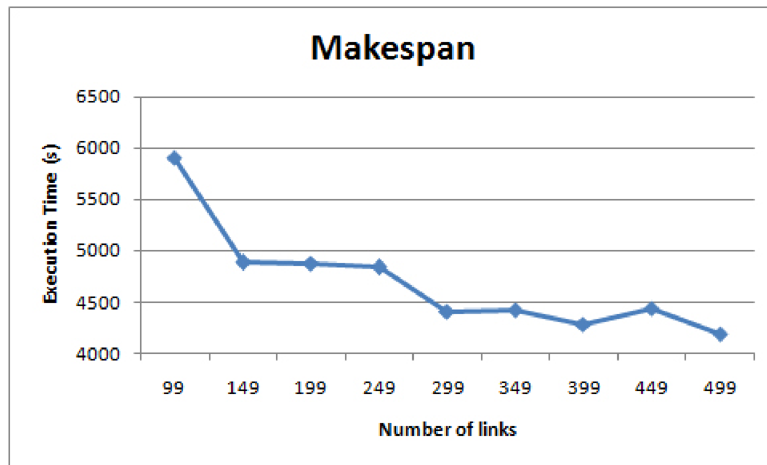


Figure 5.11: Effect of link number on ParticleZ makespan

Figure 5.12 shows the effect of increasing the number of links on the communication overhead of the ParticleZ algorithm. As can be seen in the figure, it has a linear growth with an increasing number of links.

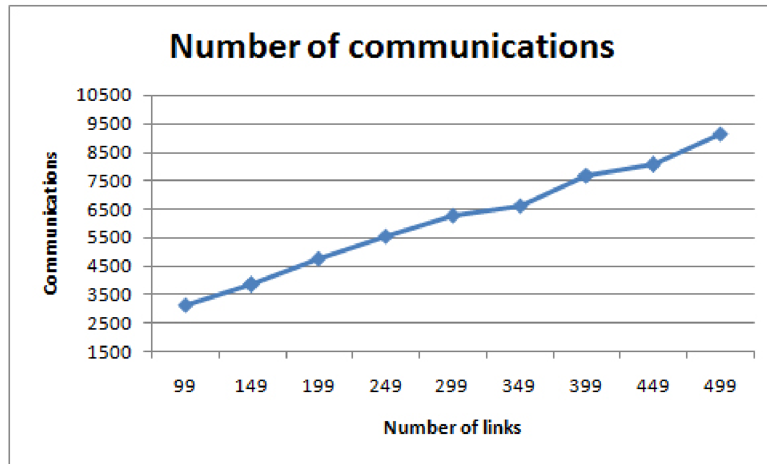


Figure 5.12: Effect of link number on ParticleZ communication number

5.7 Summary

In this chapter, we first introduced our system and application models and discussed several performance evaluation criteria which can be used to evaluate the performance of our algorithms. Then, we presented several experimental results comparing the performance of different algorithms in different scenarios and we also investigated several parameter settings and their effect on the performance of each of the algorithms. The simulation results have shown the power of these algorithms in distributed job scheduling. The next chapter summarizes the experimental results with a deep analysis followed by an outlook at the future work related to this research.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this research we have investigated the use of swarm intelligence techniques in designing distributed Grid job scheduling algorithms. Specifically, we have taken inspiration from social insect systems and sociological behaviour of birds and school of fishes in designing two distributed algorithms. We discussed several characteristics which a good load balancing algorithm should possess. The approaches introduced in this research fulfill all those characteristics such as fairness, robustness, flexibility, distribution and simplicity. The algorithms proposed can also be applied in similar load balancing environments such as Peer-to-Peer systems.

The algorithms introduced have some common characteristics. Both algorithms are distributed in nature as they have taken inspiration from real world insects and animals which are inherently distributed. Jobs are supposed to be independent of each other in both designs.

AntZ, which has taken its inspiration from Ant colony optimization creates and dispatches an ant in response to any gridlet submission to the Grid. The ants move in the Grid and leave some information about visited nodes in the resources while they are moving. This information acts like pheromone and guides other ants to get to lighter regions of the Grid.

The ParticleZ algorithm takes its inspiration from particle swarm optimization. Each resource in the Grid acts as a particle in the flock of particles. It has two characteristics associated with it: its load which defines the position of the particle among all particles and its load difference with its best neighbour which controls how fast the particle moves toward its neighbours. Moving toward a neighbour happens by submitting some of the jobs the particle owns to its best neighbour.

The SBA approach is a classical load balancing algorithm for distributed broadcast systems. In this approach each resource keeps an updated state vector of the load level of all resources in the Grid and uses this vector to send the jobs submitted to the light resources in the Grid. The state vectors get updated by broadcasting a message in the Grid each time a resource's load changes.

We have simulated our proposed algorithms using GridSim. We have also evaluated and compared the performance of our algorithms with other classical approaches (i.e. SBA and Random). The simulations have shown the algorithms proposed can perform well for scheduling jobs in a Grid network where jobs are being submitted from different sources.

Analyzing the results show that SBA has the smallest makespan among all and ParticleZ performs better than AntZ in this regard. Although SBA has the smallest makespan among all the approaches, comparing its simulation time with others reveals that there are many computational activities going on in parallel in all machines to execute SBA, which although it does not effect the overall makespan, it increases the computational complexity for the overall Grid and therefore makes SBA the worst approach among all in this regard.

Comparing the number of communications each algorithm is concerned with while executing, SBA shows a huge number of communications compared to the other two approaches. ParticleZ involves the smallest number of communications among all.

ParticleZ along with SBA win the competition among all other approaches regarding the "fairness" measure, as they have the highest load balancing level amongst all other approaches.

Looking at the scalability of the algorithms, all approaches show a linear behaviour in response to an increasing number of jobs. ParticleZ and SBA have the smallest slope and are very close to each other; AntZ ranks third after them.

Regarding an increase in the lengths of the jobs, all approaches show a linear behaviour; however, ParticleZ along with SBA are best among all. Furthermore, an increase in the number of resources decreases makespan in an exponential manner.

Having discussed several important results in Chapter 5, ParticleZ proves to perform slightly better than AntZ in many regards. On the other hand, looking at the results show ParticleZ has the advantages of SBA leaving its disadvantages aside. However, there is one drawback associated with ParticleZ. When jobs being sent to the Grid are focused on one or a small number of resources and are not spread throughout the Grid, ParticleZ's

performance drops lower than AntZ's performance. The reason is the mutation factor incorporated inside AntZ which makes it better to deal with such situations.

We also investigated several algorithm-related parametric effects for both the AntZ and ParticleZ algorithms. We investigated the effect of different wandering steps on execution time and communication overhead of the AntZ algorithm. Results show as we increase the number of wandering steps the performance of the AntZ improves but there is a limit to this improvement after which the performance stays the same although the number of wandering steps is increased. We also studied the effect of different decay rates on the performance of the AntZ and found the best decay rate in our simulation environment.

For the ParticleZ algorithm, we investigated the effect of different link numbers on both the execution time and communication overhead of the algorithms. The communication overhead grows linearly by increasing number of links while the makespan decreases.

The advantages of our proposed algorithms can be categorized as follows: 1) Looking at the simulation results, ParticleZ shows good performance results and optimum resource utilization. 2) The algorithms have proved to be "fair" compared to a random and SBA approach. ParticleZ has a load balancing level of 81%, SBA has a load balancing level of 80%, AntZ achieves a load balancing level of 65% and the random approaches have a load balancing level near 65%. 3) Both ParticleZ and AntZ are flexible approaches in dealing with the changes which happen in the Grid. 4) Both proposed approaches are distributed in nature. As the algorithms have taken inspiration from sociological systems being distributed is an inherent part and we used this ability in designing the approaches. 5) Both algorithms are very simple which is a benefit for a distributed system. In the AntZ approach, the ants which have to move among resources to find the best resource to deliver the job to, are very small in size and perform small computations in each resource. The ParticleZ has also simple computations as it only sends small messages and has to choose the lightest resource amongst all neighbour resources. 6) Looking at the scalability of the algorithms they show linear growth in response to both an increase in the number of jobs and an increase in the length of jobs.

To summarize, this research compared two different approaches (Ant Colony and particle swarm inspired algorithms) for developing load balancing algorithms and shows the benefit of swarm intelligence techniques in the distributed Grid job scheduling domain. On the other hand, it shows, although particle swarm has not been used widely in designing

distributed load balancing algorithms, it performs quite well and it even outperforms the ant colony approach in many scenarios. One of the important characteristics of the designed algorithms compared to central approaches is their responsiveness to scalability of the Grid. In centralized approaches, an increase in the number of resources in the Grid can always be a problem as the information of all the resources has to be kept and known at all time but our distributed approaches work quite well with a large number of resources and gridlets. The shortcoming of scalability was seen by running SBA simulations with a large number of resources and examining the simulation time.

In conclusion, we can say classical approaches like Random and SBA although suitable for small sized networks are not efficient for large Grids. On the other hand, in their current state, the algorithms do not address the problem of dynamic resource failure in the Grid. A mechanism should be in place that prevents gridlet loss while any resource in the Grid shuts down. Another issue which is worth considering is the special scenario when all resources in the Grid are too busy to take on new jobs. The question arises what should be done with new gridlets being submitted to the Grid. Another issue worth considering is that although we have simulated the algorithms within a simulation framework similar to a real world scenario; it may still need some small modification, for example, we have not considered issues related to security in this research. One of the steps which can be taken toward adding security is limiting ants from performing different actions in different resources.

6.2 Future Work

In this section we explore future steps and enhancements which can be done to enhance this research.

One of the important issues in large-scale Grids and peer-to-peer systems is resource failures and the robustness of the system. As the size of the Grids are continually increasing the probability of resource failures will also increase. As such, developing fault tolerant algorithms which are able to deal with these failures are gaining more and more attention. Failures which happen in a Grid environment can be divided into two categories. In one category a resource may shutdown manually, thus, it can send a notice message or perform some additional steps before shutting down. In another scenario, the resources may fail

suddenly without any notice. Thus, we need to incorporate a mechanism to deal with both categories of resource failures in our system without affecting jobs submitted by users.

Regarding the first class of failures, one step which is common for both algorithms is related to the jobs which are assigned and are being executed inside resources. We do not need to worry about successfully completed gridlets as they are already sent back to the users. For the rest of the jobs which are not yet completed, when the resource fails a failure message should be sent back to the users making them aware that the gridlet has not completed its execution, thus, it needs to be rescheduled once again to the Grid.

The proposed approach to be taken to address the failure issues follows: In the AntZ algorithm when a resource fails, as the information of that resource exists in load tables, there may be ants heading to that resource to deliver their jobs to. Thus, when the ant finds a resource has failed the following three steps should be performed:

- Send a notification to the owner of the job about the failure.
- The resource Id should be removed from the load table and the ant history.
- The user re-invokes the ant to continue finding a resource to submit its job to.

The ParticleZ algorithm can deal with failures more easily. At the time a resource wants to share its workload with other resources, it simply sends a message and queries about its available neighbours, therefore, whenever a resource breaks down it is simply eliminated from this process automatically. Thus, in case of ParticleZ, a message sent to the user about the uncompleted gridlets will suffice. Yet, when a resource fails without further notice the situation is more complex. One valid approach is the following.

When a job is sent to the Grid by a user, the worst case execution time will be estimated for that job. This predicted time represents the worst case in which the user must have received the results of its job submission. Then, an event will be scheduled for the predicted time. At this specific time, the user will check whether the job result was returned; if the job result has already come back successfully, no further actions will be taken; otherwise the job will be submitted again and the whole process repeats.

Another important issue is paying attention that the nodes may not be dedicated nodes in the Grid, and each may have their own background workload. Thus, the local load of each resource should be incorporated in the load calculation formula which effects the decision making of the algorithms accordingly.

Another issue which would be interesting to address is the security problems and authentication required for the messages and jobs to be sent and received.

In this research we have simulated of the proposed algorithms with a simulation platform developed for the Grid, and the results proved to be promising. The next step would be to apply the algorithms in a real world Grid or incorporate the algorithm in existing Grid applications such as the Sun Grid Engine or Globus toolkit.

References

[1] J. Knobloch and L. Robertson. Lhc computing grid. Technical report, CERN-LHCC, 2005.

[2] S. L. Heitor and M. Perretto. Reconstruction of phylogenetic trees using the ant colony optimization paradigm. In *WOB*, pages 49–56, 2004.

[3] R. Buyya and M. M. Murshed. Gridsim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14:1175–1220, 2002.

[4] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1998.

[5] K. Q. Yan, S. C. Wang, C. P. Chang, and J. S. Lin. A hybrid load balancing policy underlying grid computing environment. *Computer Standards and Interfaces*, 29(2):161–173, 2007.

[6] I. Foster and C. Kesselman. The grid in a nutshell. pages 3–13, 2004.

[7] Ch. Baru, R. Moore, A. Rajasekar, and M. Wan. The sdsc storage resource broker. In *Proceedings of CASCON*, 1998.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[9] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[10] M. J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. *8th International Conference on Distributed Computing Systems*, pages 104–111, Jun 1988.

[11] A. S. Grimshaw, Wm. A. Wulf, and CORPORATE The Legion Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[12] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. *Computer*, 32(5):29–37, 1999.

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15:2001, 2001.

[14] I. Foster et.al. The open grid services architecture, version 1.5, 2006.

[15] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, D. Snelling, and Vanderbilt. Open grid services infrastructure (ogsi).

[16] Ch. Kandagatla. Survey and taxonomy of grid resource management systems. University of Texas, Austin.

[17] J. Cao, D. P. Spooner, S. A. Jarvis, and G. R. Nudd. Grid load balancing using intelligent agents. *Future Generation Computer Systems*, 21(1):135–149, 2005.

[18] R. Subrata, A. Y. Zomaya, and B. Landfeldt. Artificial life techniques for load balancing in computational grids. *Journal of Computer and System Sciences*, 73(8):1176–1190, 2007.

[19] C. Grosan, A. Abraham, and B. Helvik. Multiobjective evolutionary algorithms for scheduling jobs on computational grids. ADIS International Conference, Applied Computing, Salamanca, Spain, Nuno Guimares and Pedro Isaias (Eds.), 2007.

[20] S. Salleh and A.Y. Zomaya. *Scheduling in Parallel Computing Systems:Fuzzy and Annealing Techniques*. The Springer International Series in Engineering and Computer science, 1999.

[21] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. *SIGMETRICS Performance Evaluation Review*, 11(1):47–55, 1981.

[22] Y. Li and Z. Lan. *A Survey of Load Balancing in Grid Computing*. Springer Berlin / Heidelberg, 2005.

[23] B. Yagoubi and Y. Slimani. Dynamic load balancing strategy for grid computing. *Proceedings of World Academy of Science, Engineering and Technology*, 2006.

[24] N. G. Shivaratri, Ph. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[25] W. Zhu, Ch. Sun, and C. Shieh. Comparing the performance differences between centralized load balancing methods. *IEEE International Conference on Systems, Man, and Cybernetics*, 3:1830–1835 vol.3, Oct 1996.

[26] R. Schoonderwoerd, O. E. Holland, J. L. Bruten, and L. J. M. Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive Behavior*, (2):169–207, 1996.

[27] R. Subrata and A. Y. Zomaya. A comparison of three artificial life techniques for reporting cell planning in mobile computing. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):142–153, Feb 2003.

[28] E. Bonabeau. Social insect colonies as complex adaptive systems. *Ecosystems*, 1:437 – 443, 1998.

[29] EO. Wilson and B. Hoelldobler. Dense heterarchies and mass communication as the basis of organization in ant colonies. Trends in Ecology and Evolution, 1988.

[30] N. R. Franks. Army ants: A collective intelligence. *American Scientist*, pages 139–145, March 1989.

[31] R. Schoonderwoerd, O. Holland, and J. Bruten. Ant-like agents for load balancing in telecommunications networks. In *AGENTS '97: Proceedings of the first international conference on Autonomous agents*, pages 209–216, New York, NY, USA, 1997. ACM.

[32] A. Al-Dahoud and M. Belal. Multiple ant colonies for load balancing in distributed systems. Proceedings of The first International Conference on ICT and Accessibility, 2007.

[33] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. *Proceedings of the Sixth International Symposium on Micro Machine and Human Science, MHS '95.*, pages 39–43, Oct 1995.

[34] D. S. Liu, K.C. Tan, and W. K. Ho. A distributed co-evolutionary particle swarm optimization algorithm. *IEEE Congress on Evolutionary Computation, CEC 2007.*, pages 3831–3838, Sept. 2007.

[35] J. Kennedy and R. C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[36] Y. L. Zheng, L. H. Ma, L. Y. Zhang, and J. X. Qian. On the convergence analysis and parameter selection in particle swarm optimization. *International Conference on Machine Learning and Cybernetics*, 2003.

[37] A. Moallem and S. A. Ludwig. Using artificial life techniques for distributed grid job scheduling. *Proceedings of the 24th Annual ACM Symposium on Applied Computing*, 2009.

[38] K. Etminani and M. Naghibzadeh. A min-min max-min selective algorihtm for grid task scheduling. *ICI 2007. 3rd IEEE/IFIP International Conference in Central Asia on Internet*, pages 1–7, Sept. 2007.

[39] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, School of Computing, Queens University, 2006.

[40] S. Chen, W. Zhang, F. Ma, J. Shen, and M. Li. A novel agent-based load balancing algorithm for grid computing. In *GCC Workshops*, pages 156–163, 2004.

[41] K. P. Chow and Y. K. Kwok. On load balancing for distributed multiagent computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(8):787–801, Aug 2002.

[42] J. Cao, D. P. Spooner, S. A. Jarvis, S. Saini, and G. R. Nudd. Agent-based grid load balancing using performance-driven task scheduling. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 49.2, Washington, DC, USA, 2003. IEEE Computer Society.

[43] H. Muhlenbein. Evolutionary algorithms: Theory and applications. In *Local Search in Combinatorial Optimization*. Wiley, 1993.

[44] F. Glover. Tabu search part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[45] A. Abraham, H. Liu, W. Zhang, and TG. Chang. Scheduling jobs on computational grids using fuzzy particle swarm algorithm. In *Proceedings of 10th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 500–507, 2006.

[46] T. Chen, B. Zhang, X. Hao, and Y. Dai. Task scheduling in grid based on particle swarm optimization. In *ISPDC '06: Proceedings of the Fifth International Symposium on Parallel and Distributed Computing*, pages 238–245, Washington, DC, USA, 2006. IEEE Computer Society.

[47] A. Salman, I. Ahmad, and S. Al-Madani. Particle swarm optimization for task assignment problem. *Microprocessors and Microsystems*, 26:363–371, November 2002.

[48] M. S. Kwang and H. W. Sun. Ant colony optimization for routing and load-balancing: survey and new directions. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 33(5):560–572, 2003.

[49] A. Montresor, H. Meling, and . Babaolu. *Messor: Load-Balancing through a Swarm of Autonomous Agents*. Springer Berlin / Heidelberg, 2003.

[50] J. Cao. Self-organizing agents for grid load balancing. *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 388–395, Nov. 2004.

[51] M. Amini Salehi and H. Deldari. Grid load balancing using an echo system of intelligent ants. In *PDCN'06: Proceedings of the 24th IASTED international conference on Parallel and distributed computing and networks*, pages 47–52, Anaheim, CA, USA, 2006. ACTA Press.

[52] K.M. Sim and W. H. Sun. *Multiple Ant Colony Optimization for Load Balancing.* Springer Berlin / Heidelberg, 2003.

[53] M. Heusse, S. Guerin, D. Snyers, and P. Kuntz. A new distributed and adaptive approach to routing and load balancing in dynamic communication networks.

[54] G. Di Caro and M. Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[55] T. White and B. Pagurek. Asga: Improving the ant system by integration with genetic algorithms. In *University of Wisconsin*, pages 610–617. Morgan Kaufmann, 1998.

[56] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. *SIGMETRICS Performance Evaluation Review*, 16(1):63–72, 1988.

[57] W. Zhu, P. Socko, and B. Kiepuszewski. Migration impact on load balancing—an experience on amoeba. *SIGOPS Operating Systems Review*, 31(1):43–53, 1997.

[58] Ch. Li, Ch. Ding, and K. Shen. Quantifying the cost of context switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 2, New York, NY, USA, 2007. ACM.

[59] M. Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, Politecnico di Milano, Italy, 1992.

[60] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy. Technical report, 1991.

[61] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26:29–41, 1996.

[62] M. Dorigo and Ch. Blum. Ant colony optimization theory: a survey. *Theoretical Computer Science*, 344(2-3):243–278, 2005.

[63] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of ECAL91 - European Conference on Artificial Life*, 1991.

[64] M. Dorigo, M. Birattari, and T. Stu?tzle. Ant colony optimization artificial ants as a computational intelligence technique. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

[65] J. L. Deneubourg, S. Aron, S. Goss, , and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3(2):159–168, 1990.

[66] J. E. Bell and P. R. Mcmullen. Ant colony optimization techniques for the vehicle routing problem. *Adv. Eng. Inf.*, 18(1):41–48, 2004.

[67] M. Dorigo and L. M. Gambardella. Ant colonies for the travelling salesman problem, 1996.

[68] W. Yi, Q. Liu, and Y. He. Dynamic distributed genetic algorithms. *Proceedings of the 2000 Congress on Evolutionary Computation*, 2:1132–1136 vol.2, 2000.

[69] J. Kennedy and R. Eberhart. Particle swarm optimization. *Proceedings of IEEE International Conference on Neural Networks*, 4:1942–1948 vol.4, Nov/Dec 1995.

[70] I. C. Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information Processing Letters*, 85(6):317–325, 2003.

[71] R. Fromm and N. Treuhaft. Revisiting the cache interference costs of context switching.

[72] Y. Aridor, M. Factor, and A. Teperman. cjvm: A single system image of a jvm on a cluster. In *Proceedings of the International Conference on Parallel Processing*, pages 4–11, 1999.

[73] F. W. Howell and R. McNab. Simjava: a discrete event simulation package for java with applications in computer systems modelling. In *Proceedings of the First International Conference on Web-based Modelling and Simulation*, San Diego, CA, 1998. The Society for Computer Simulation.

Appendix A

Source code

```
package allocpolicy.antZ;

import eduni.simjava.Sim_event;
import eduni.simjava.Sim_port;
import eduni.simjava.Sim_system;
import grid.StatisticalAnalysis;
import grid.Topology;
import gridsim.*;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Random;


public class AntColonyAllocPolicy extends AllocPolicy {

    private ResGridletList gridletInExecList_; // storing exec Gridlets
    private MIShares share_; // a temp variable
    private double lastUpdateTime_; // a timer to denote the last update time
    private AntPool antPool;
    private HashMap<Integer, Double> loadTable = new HashMap<Integer, Double>();
    private Sim_port output;
    private boolean log = false;

    public AntColonyAllocPolicy(String resName, String entityName,
                                Sim_port output) throws Exception {
        super(resName, entityName);

        this.gridletInExecList_ = new ResGridletList();
        this.share_ = new MIShares();
        this.lastUpdateTime_ = 0.0;
        this.gridletInExecList_ = new ResGridletList();
        this.antPool = AntPool.getInstance();
        this.loadTable = new HashMap<Integer, Double>();
        this.output = output;
    }

    public void body() {

        double time1, time2 = GridSim.clock();

        // a loop that is looking for internal events only
        Sim_event ev = new Sim_event();
        while (Sim_system.running()) {

            time1 = time2;
```

```
            time2 = GridSim.clock();

            if ((log) && (time2 - time1 > 200))
                try {
                    StatisticalAnalysis.getAntColony_log().
                            writeChars(String.valueOf(calculateTotalLoad
                                    (gridletInExecList_.size())) + "\n");
                } catch (IOException e) {
                    e.printStackTrace();
                }


            super.sim_get_next(ev);
            if (ev.get_tag() == GridSimTags.END_OF_SIMULATION ||
                    super.isEndSimulation()) {
                gridletInExecList_.clear();
                break;
            }

            // Internal Event if the event source is this entity
            if (ev.get_src() == super.myId_) {
                internalEvent();
            }

            // CHECK for ANY INTERNAL EVENTS WAITING TO BE PROCESSED
            while (super.sim_waiting() > 0) {
                // wait for event and ignore since it is likely to be related to
                // internal event scheduled to update Gridlets processing
                super.sim_get_next(ev);
                // System.out.println(super.get_name()
                // + ".body(): ignore internal events : " + ev.get_tag());
            }
        }
    }

    public void gridletCancel(int gridletId, int userId) {

    }

    public void gridletMove(int gridletId, int userId, int destId, boolean ack) {

    }

    public void gridletPause(int gridletId, int userId, boolean ack) {

    }

    public void gridletResume(int gridletId, int userId, boolean ack) {

    }

    public int gridletStatus(int gridletId, int userId) {
        ResGridlet rgl;
```

```
    // Find in EXEC List first
    int found = super.findGridlet(gridletInExecList_, gridletId, userId);
    if (found >= 0) {
        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletInExecList_.get(found);
        return rgl.getGridletStatus();
    }
    // if not found in all lists
    return -1;
}

public void gridletSubmit(Gridlet gl, boolean ack) {

    Ant ant = antPool.getPool().get(gl.getGridletID());
    if (ant == null) {
        StatisticalAnalysis.Communications++;
        ant = new Ant(gl);
        antPool.getPool().put(gl.getGridletID(), ant);
    }
    if (ant.isFinish()) {

        // ResGridlet rgl = ant.getGridlet();
        updateGridletProcessing();

        // reset number of PE since at the moment, it is not supported
        if (gl.getNumPE() > 1) {
            String userName = GridSim.getEntityName(gl.getUserID());
            System.out.println();
            System.out.println(super.get_name() + ".gridletSubmit(): "
                    + " Gridlet #" + gl.getGridletID() + " from "
                    + userName + " user requires " + gl.getNumPE()
                    + " PEs.");
            System.out.println("--> Process this Gridlet to 1 PE only.");
            System.out.println();

            // also adjusted the length because the number of
            // PEs are reduced
            int numPE = gl.getNumPE();
            double len = gl.getGridletLength();
            gl.setGridletLength(len * numPE);
            gl.setNumPE(1);
        }

        // adds a Gridlet to the in execution list
        ResGridlet rgl = new ResGridlet(gl);
        rgl.setGridletStatus(Gridlet.INEXEC); // set the Gridlet status to exec
        gridletInExecList_.add(rgl); // add into the execution list

        // sends back an ack if required
        if (ack) {
            super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, true, gl
                    .getGridletID(), gl.getUserID());
        }
        forecastGridlet();
```

```
            antPool.getPool().remove(gl.getGridletID());
        } else {
            try {

                //System.out.print("Load for resource ID " + resId_ + " is "
                //      + getLoad());
                //System.out.print("\n");
                ant.step(calculateTotalLoad(gridletInExecList_.size()));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    ////////////////////// Private Methods ////////////////////////
    protected double calculateTotalLoad(int size) {
        int totalRating = 0;
        PEList peList = (resource_.getMachineList().getMachine(0)).getPEList();
        for (int i = 0; i < peList.size(); i++) {
            totalRating += ((PE) peList.get(i)).getMIPSRating();
        }

        totalRating = totalRating / 10;
        // Devide by the lowest PE rate in the Grid.
        // Here we have 10 and 50 so we divide by 10
        double val = (size + 1.0) / totalRating;
        int numGridletPerPE = (int) Math.ceil(val);

        // load is between [0.0, 1.0] where 1.0 is busy and 0.0 is not busy
        double localLoad = resCalendar_.getCurrentLoad();
        double load = 1.0 - ((1 - localLoad) / numGridletPerPE);
        if (load < 0.0) {
            load = 0.0;
        }

        return load;
    }

    private void updateGridletProcessing() {
        // Identify MI share for the duration (from last event time)
        double time = GridSim.clock();
        double timeSpan = time - lastUpdateTime_;

        // if current time is the same or less than the last update time,
        // then ignore
        if (timeSpan <= 0.0) {
            return;
        }

        // Update Current Time as the Last Update
        lastUpdateTime_ = time;

        // update the GridResource load
        int size = gridletInExecList_.size();
```

```
        double load = super.calculateTotalLoad(size);
        super.addTotalLoad(load); // add the current resource load

        // if no Gridlets in execution then ignore the rest
        if (size == 0) {
            return;
        }

        // gets MI Share for all Gridlets
        MIShares shares = getMIShare(timeSpan, size);
        ResGridlet obj;

        // a loop that allocates MI share for each Gridlet accordingly
        // In this algorithm, Gridlets at the front of the list
        // (range = 0 until MIShares.maxCount-1) will be given max MI value
        // For example, 2 PEs and 3 Gridlets. PE #0 processes Gridlet #0
        // PE #1 processes Gridlet #1 and Gridlet #2
        int i = 0; // a counter
        Iterator iter = gridletInExecList_.iterator();
        while (iter.hasNext()) {
            obj = (ResGridlet) iter.next();

            // Updates the Gridlet length that is currently being executed
            if (i < shares.maxCount) {
                obj.updateGridletFinishedSoFar(shares.max);
            } else {
                obj.updateGridletFinishedSoFar(shares.min);
            }

            i++; // increments i
        }
    }

    private MIShares getMIShare(double timeSpan, int size) {
        // 1 - localLoad_ = available MI share percentage
        double localLoad = super.resCalendar_.getCurrentLoad();
        double TotalMIperPE = super.resource_.getMIPSRatingOfOnePE() * timeSpan
                * (1 - localLoad);

        // This allocpolicy.TimeShared is not Round Robin where each PE for 1
        // Gridlet only.
        // a PE can have more than one Gridlet executing.
        // minimum number of Gridlets that each PE runs.
        int glDIVpe = size / super.totalPE_;

        // number of PEs that run one extra Gridlet
        int glMODpe = size % super.totalPE_;

        // If num Gridlets in execution > total PEs in a GridResource,
        // then divide MIShare by the following constraint:
        // - obj.max = MIShare of a PE executing n Gridlets
        // - obj.min = MIShare of a PE executing n+1 Gridlets
        // - obj.maxCount = a threshold number of Gridlets will be assigned to
        // max MI value.
```

```
    //
    // In this algorithm, Gridlets at the front of the list
    // (range = 0 until maxCount-1) will be given max MI value
    if (glDIVpe > 0) {
        // this is for PEs that run one extra Gridlet
        share_.min = TotalMIperPE / (glDIVpe + 1);
        share_.max = TotalMIperPE / glDIVpe;
        share_.maxCount = (super.totalPE_ - glMODpe) * glDIVpe;
    }

    // num Gridlet in Exec < total PEs, meaning it is a
    // full PE share: i.e a PE is dedicated to execute a single Gridlet
    else {
        share_.max = TotalMIperPE;
        share_.min = TotalMIperPE;
        share_.maxCount = size; // number of Gridlet
    }

    return share_;
}

private double forecastFinishTime(double availableRating, double length) {
    double finishTime = (length / availableRating);

    // This is as a safeguard since the finish time can be extremely
    // small close to 0.0, such as 4.5474735088646414E-14. Hence causing
    // some Gridlets never to be finished and consequently hang the program
    if (finishTime < 1.0) {
        finishTime = 1.0;
    }
    return finishTime;
}

private int getLighterNodeinHistory() {

    int bestIp = resId_;
    double bestLoad = calculateTotalLoad(gridletInExecList_.size());
    ArrayList<Integer> equalLoads = new ArrayList<Integer>();

    Iterator iterator = loadTable.keySet().iterator();
    while (iterator.hasNext()) {
        Integer id = (Integer) iterator.next();
        Double load = loadTable.get(id);

        if (load < bestLoad) {
            bestIp = id;
            equalLoads.add(id);
        } else if (load == bestLoad) {
            equalLoads.add(id);
        }
    }
    Random random = new Random();
    if (equalLoads.size() > 0)
        return equalLoads.get(random.nextInt(equalLoads.size()));
```

```
    else
        return bestIp;
}

private void printLoadTable() throws IOException {
    Iterator iterator = loadTable.keySet().iterator();

    System.out.print("ID         Load");
    System.out.print("\n");
    System.out.print("--------------");
    System.out.print("\n");

    while (iterator.hasNext()) {
        Integer key = (Integer) iterator.next();
        System.out.print(key + "    ");
        System.out.print(String.valueOf(loadTable.get(key)));
        System.out.print("\n");
    }
}

private void internalEvent() {

    // this is a constraint that prevents an infinite loop
    // Compare between 2 floating point numbers. This might be incorrect
    // for some hardware platform.
    if (lastUpdateTime_ == GridSim.clock()) {
        return;
    }

    // update Gridlets in execution up to this point in time
    updateGridletProcessing();

    // schedule next event
    forecastGridlet();
}

private void checkGridletCompletion() {
    ResGridlet rgl;

    // a loop that determine the smallest finish time of a Gridlet
    // Don't use an iterator since it causes an exception because if
    // a Gridlet is finished, gridletFinish() will remove it from the
    // list.
    int i = 0;
    while (i < gridletInExecList_.size()) {
        rgl = (ResGridlet) gridletInExecList_.get(i);

        // if a Gridlet has finished, then remove it from the list
        if (rgl.getRemainingGridletLength() <= 0.0) {
            gridletFinish(rgl, Gridlet.SUCCESS);
            continue; // not increment i coz the list size also decreases
        }
        i++;
    }
```

```
}

/**
 * Updates the Gridlet's properties, such as status once a Gridlet is
 * considered finished.
 *
 * @param rgl
 * @param status
 */
private void gridletFinish(ResGridlet rgl, int status) {
    // NOTE: the order is important! Set the status first then finalize
    // due to timing issues in ResGridlet class.
    rgl.setGridletStatus(status);
    rgl.finalizeGridlet();

    // sends back the Gridlet with no delay
    Gridlet gl = rgl.getGridlet();
    super.sendFinishGridlet(gl);

    // remove this Gridlet in the execution
    gridletInExecList_.remove(rgl);
}

private void forecastGridlet() {
    // if no Gridlets available in exec list, then exit this method
    if (gridletInExecList_.size() == 0) {
        return;
    }

    // checks whether Gridlets have finished or not. If yes, then remove
    // them since they will effect the MIShare calculation.
    checkGridletCompletion();

    // Identify MIPS share for all Gridlets for 1 second, considering
    // current Gridlets + No of PEs.
    MIShares share = getMIShare(1.0, gridletInExecList_.size());

    ResGridlet rgl;
    int i = 0;
    double time;
    double rating;
    double smallestTime = 0.0;

    // For each Gridlet, determines their finish time
    Iterator iter = gridletInExecList_.iterator();
    while (iter.hasNext()) {
        rgl = (ResGridlet) iter.next();

        // If a Gridlet locates before the max count then it will be given
        // the max. MIPS rating
        if (i < share.maxCount) {
            rating = share.max;
        } else { // otherwise, it will be given the min. MIPS Rating
            rating = share.min;
```

```
        }

        time = forecastFinishTime(rating, rgl.getRemainingGridletLength());

        int roundUpTime = (int) (time + 1); // rounding up
        rgl.setFinishTime(roundUpTime);

        // get the smallest time of all Gridlets
        if (i == 0 || smallestTime > time) {
            smallestTime = time;
        }

        i++;
    }

    // sends to itself as an internal event
    super.sendInternalEvent(smallestTime);
}

////////////////////////// Inner Classes //////////////////////////

public class Ant {

    private Gridlet gridlet;
    private int sourceID;
    private HashMap<Integer, AntHistory> antHistory;
    private int WANDER_NUMBER = 4;
    private int step;
    private boolean isfinish = false;
    private double MUTATION_RATE = 0.5;

    public Ant(Gridlet gridlet) {

        this.gridlet = gridlet;
        this.sourceID = gridlet.getResourceID();
        this.antHistory = new HashMap<Integer, AntHistory>();
        //try {
        //System.out.print("Ant ID " + gridlet.getGridletID()
        //        + " initiated in " + sourceID + "  ");
        //System.out.print("\n");
        //printLoadTable();
        //} catch (IOException e) {
        //   e.printStackTrace();
        //}
    }

    public boolean isFinish() {
        return isfinish;
    }

    public void step(double load) throws IOException {

        int newSourceId;
```

```
        //System.out.print("Ant ID " + gridlet.getGridletID() + " took step "
        //      + step);
        step++;
        AntHistory newNode = new AntHistory(sourceID, load);
        antHistory.put(sourceID, newNode);

        Iterator iterator = antHistory.keySet().iterator();
        while (iterator.hasNext()) {
            AntHistory temp = antHistory.get(iterator.next());
            loadTable.put(temp.getIp(), temp.getLoad());
        }

        // TODO: experiment a huristic for choosing nodes
        // TODO: experiment a random value for continuing searching
        //System.out.print(" from " + sourceID);

        Random random = new Random();
        if (random.nextDouble() < MUTATION_RATE) {
            newSourceId = Topology.gerRandomNodeId();
        } else
            newSourceId = getLighterNodeinHistory();
        if (MUTATION_RATE > 0)
            MUTATION_RATE = MUTATION_RATE - 0.2;
        //System.out.print(" to " + newSourceId);
        //System.out.print("\n");

        if (step == WANDER_NUMBER)
            isfinish = true;

        // TODO : how to set the Cost and is it ever used ?!
        if (sourceID != newSourceId) {
            StatisticalAnalysis.Communications++;
            sourceID = newSourceId;
            gridlet.setResourceParameter(sourceID, gridlet.getCostPerSec
                    (sourceID));
            // TODO : what to do with ack
            // TODO : check if the source and dest is the same do nothign bad
            sim_schedule(output, 0, GridSimTags.GRIDLET_SUBMIT_ACK,
                    new IO_data(gridlet, gridlet.getGridletFileSize(),
                            sourceID, 0));
        } else {
            gridletSubmit(gridlet, true);
        }
    }

    public int getsourceID() {
        return sourceID;
    }

    public Gridlet getGridlet() {
        return gridlet;
    }
}
```

```
    /**
     * Gridlets MI share in Time Shared Mode
     */
    private class MIShares {
        /**
         * maximum amount of MI share Gridlets can get
         */
        public double max;

        /**
         * minimum amount of MI share Gridlets can get when it is executed on a
         * PE that runs one extra Gridlet
         */
        public double min;

        /**
         * Total number of Gridlets that get Max share
         */
        public int maxCount;

        /**
         * Default constructor that initializes all attributes to 0
         */
        public MIShares() {
            max = 0.0;
            min = 0.0;
            maxCount = 0;
        }
    }

}

package allocpolicy.antZ;

/**
 * Created by IntelliJ IDEA.
 * User: Azin
 * Date: 5-Nov-2007
 * Time: 4:55:24 AM
 * To change this template use File | Settings | File Templates.
 */
public class AntHistory {
    private int ip;
    private double load;

    public AntHistory(int ip, double load) {
        this.ip = ip;
        this.load = load;
    }


    public int getIp() {
        return ip;
    }
```

```
    public double getLoad() {
        return load;
    }
}

package allocpolicy.antZ;

import java.util.ArrayList;
import java.util.HashMap;

public class AntHistoryList {

    private static AntHistoryList instance;
    private HashMap<Integer, ArrayList> nodeAntHistory;

    private AntHistoryList() {
        nodeAntHistory = new HashMap<Integer, ArrayList>();
    }

    public static AntHistoryList getInstance() {
        if (instance == null)
            instance = new AntHistoryList();
        return instance;
    }

    public void add(int ID, double cost) {
        ArrayList<AntHistory> list = nodeAntHistory.get(new Integer(ID));
        if (list == null) {
            list = new ArrayList<AntHistory>();
        }
        list.add(new AntHistory(ID, cost));
        nodeAntHistory.put(ID, list);


    }

/*    public int getLighterNodeinHistory(int ID, double cost) {


        double bestLoad = cost;
        ArrayList<Integer> equalLoads = new ArrayList<Integer>();

        ArrayList arrayList = nodeAntHistory.get(new Integer(ID));
        for (int i = 0; i < arrayList.size(); i++) {

            AntHistory antHistory = (AntHistory) arrayList.get(i);
            if ((antHistory.getLoad() < bestLoad)) {
                equalLoads.add(antHistory.getIp());
            } else if (antHistory.getLoad() == bestLoad) {
                equalLoads.add(antHistory.getIp());

            }
        }
```

```
        Random random = new Random();
        return equalLoads.get(random.nextInt(equalLoads.size()));
    } */


}

package allocpolicy.antZ;

import java.util.HashMap;

import allocpolicy.antZ.AntColonyAllocPolicy.Ant;

public class AntPool {

 private HashMap<Integer, Ant> gridletID_antID;
 private static AntPool instance;

 public static AntPool getInstance(){
 if(instance == null)
instance = new AntPool();
 return instance;
 }

 private AntPool(){
 gridletID_antID = new  HashMap<Integer, Ant>();
 }

 public HashMap<Integer, Ant> getPool(){
 return gridletID_antID;
 }


}

package allocpolicy.particleZ;

import java.util.HashMap;

public class AllocPolicyList {
private HashMap<Integer, ParticleSwarmAllocPolicy> allocPolicies;
private static AllocPolicyList allocPolicyList;

public AllocPolicyList(){
allocPolicies = new HashMap<Integer, ParticleSwarmAllocPolicy>();
}

public static AllocPolicyList getInstance(){
if(allocPolicyList == null)
allocPolicyList = new AllocPolicyList();
return allocPolicyList;
}

public  void addallocPolicy(ParticleSwarmAllocPolicy allocPolicy){
```

```
allocPolicies.put(allocPolicy.getresId(), allocPolicy);
}

    public  ParticleSwarmAllocPolicy getAllocPolicy(int id){
return allocPolicies.get(id);
}

}

package allocpolicy.particleZ;

import java.util.ArrayList;
import java.util.Iterator;
import java.io.IOException;

import eduni.simjava.Sim_event;
import eduni.simjava.Sim_system;
import grid.Topology;
import grid.StatisticalAnalysis;
import gridsim.AllocPolicy;
import gridsim.GridSim;
import gridsim.GridSimTags;
import gridsim.Gridlet;
import gridsim.Machine;
import gridsim.MachineList;
import gridsim.PE;
import gridsim.PEList;
import gridsim.ResGridlet;
import gridsim.ResGridletList;

public class ParticleSwarmAllocPolicy extends AllocPolicy {

    private ResGridletList gridletQueueList_; // Queue list
    private ResGridletList gridletInExecList_; // Execution list
    private double lastUpdateTime_; // the last time Gridlets updated
    private int[] machineRating_; // list of machine ratings available
    private boolean log = false;


    public ParticleSwarmAllocPolicy(String resName, String entityName)
            throws Exception {
        super(resName, entityName);
        // initialises local data structure
        this.gridletInExecList_ = new ResGridletList();
        this.gridletQueueList_ = new ResGridletList();
        this.lastUpdateTime_ = 0.0;
        this.machineRating_ = null;
    }

    public void body() {

        double time1, time2 = GridSim.clock();

        // Gets the PE's rating for each Machine in the list.
```

```
// Assumed one Machine has same PE rating.
MachineList list = super.resource_.getMachineList();
int size = list.size();
machineRating_ = new int[size];
for (int i = 0; i < size; i++) {
    machineRating_[i] = super.resource_.getMIPSRatingOfOnePE(i, 0);
}

// a loop that is looking for internal events only
Sim_event ev = new Sim_event();
while (Sim_system.running()) {

    time1 = time2;
    time2 = GridSim.clock();

    if ((log) && (time2 - time1 > 200))
        try {
            StatisticalAnalysis.get_particle_log().
                    writeChars(String.valueOf(calculateTotalLoad(
                            gridletInExecList_.size() +
                                    gridletQueueList_.size())) + "\n");
        } catch (IOException e) {
            e.printStackTrace();
        }
    super.sim_get_next(ev);

    // if the simulation finishes then exit the loop
    if (ev.get_tag() == GridSimTags.END_OF_SIMULATION
            || super.isEndSimulation()) {
        break;
    }
    // Internal Event if the event source is this entity
    if (ev.get_src() == super.myId_ && gridletInExecList_.size() > 0) {
        updateGridletProcessing();   // update Gridlets
        checkGridletCompletion();    // check for finished Gridlets
    }

    if (gridletQueueList_.size() > 0) {
        double load = calculateTotalLoad(gridletInExecList_.size()
                + gridletQueueList_.size());
        int destID = resId_;
        double delta = Double.MAX_VALUE;
        AllocPolicyList allocPolicyList = AllocPolicyList.getInstance();

        ArrayList<Integer> neighbours = Topology.
                getConnectedResources(resId_);
        for (int i = 0; i < neighbours.size(); i++) {
            int neighbourId = neighbours.get(i);
            ParticleSwarmAllocPolicy allocPolicy = allocPolicyList.
                    getAllocPolicy(neighbourId);
            double destLoad = allocPolicy.calculateTotalLoad();
            StatisticalAnalysis.Communications++;
            StatisticalAnalysis.Communications++;
            if (destLoad < load) {
```

```
                        load = destLoad;
                        destID = neighbours.get(i);
                    }
                }
                for (int i = 0; i < neighbours.size(); i++) {
                    int neighbourId = neighbours.get(i);
                    ParticleSwarmAllocPolicy allocPolicy = allocPolicyList.
                            getAllocPolicy(neighbourId);
                    double destLoad = allocPolicy.calculateTotalLoad();
                    // If in the meanwhile of these processes there is appearing a
                    // node which by decrease in its load it is not the second best
                    // any more we quit this round of body()
                    if (load - destLoad < 0) {
                        delta = 0.2;
                        break;
                    } else if ((load - destLoad < delta)) {
                        delta = load - destLoad;
                    }
                }
//                double THRESHOLD = delta;
                if ((destID != resId_)) {
                    while ((gridletQueueList_.size() > 0) &&
                            (calculateTotalLoad(gridletInExecList_.size() +
                                    gridletQueueList_.size()) - load > delta)) {
                        ResGridlet obj = (ResGridlet) gridletQueueList_.get(0);
                        gridletMove(obj.getGridletID(), obj.getUserID(),
                                destID, true);
                    }
                }
            }
            allocateQueueGridlet();

            // CHECK for ANY INTERNAL EVENTS WAITING TO BE PROCESSED
            while (super.sim_waiting() > 0) {
                // wait for event and ignore since it is likely to be related to
                // internal event scheduled to update Gridlets processing
                super.sim_get_next(ev);
                // System.out.println(super.resName_+
                // ".allocpolicy.SBA.body(): ignore internal events");
            }

        }
    }

    public void gridletCancel(int gridletId, int userId) {
    }

    public void gridletMove(int gridletId, int userId, int destId, boolean ack) {

        // cancels the Gridlet
        ResGridlet rgl = cancel(gridletId, userId);

        // if the Gridlet is not found
        if (rgl == null) {
```

```
        System.out.println(super.resName_
                + ".allocpolicy.SBA.gridletMove(): Cannot find "
                + "Gridlet #" + gridletId + " for User #" + userId);

        if (ack) { // sends back an ack if required
            super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, false, gridletId,
                    userId);
        }
        return;
    }

    // if the Gridlet has finished beforehand
    if (rgl.getGridletStatus() == Gridlet.SUCCESS) {
        System.out
                .println(super.resName_
                        + ".allocpolicy.SBA.gridletMove(): " +
                        "Cannot move Gridlet #"
                        + gridletId + " for User #" + userId
                        + " since it has FINISHED.");

        if (ack) { // sends back an ack if required
            super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, false, gridletId,
                    userId);
        }

        gridletFinish(rgl, Gridlet.SUCCESS);
    } else { // otherwise moves this Gridlet to a different GridResource
        rgl.finalizeGridlet();

        // Set PE on which Gridlet finished to FREE
        super.resource_.setStatusPE(PE.FREE, rgl.getMachineID(),
                rgl.getPEID());
        super.gridletMigrate(rgl.getGridlet(), destId, ack);
        //  allocateQueueGridlet();
    }
}

public void gridletPause(int gridletId, int userId, boolean ack) {
}

public void gridletResume(int gridletId, int userId, boolean ack) {
}

public int gridletStatus(int gridletId, int userId) {
    ResGridlet rgl;

    // Find in EXEC List first
    int found = super.findGridlet(gridletInExecList_, gridletId, userId);
    if (found >= 0) {
        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletInExecList_.get(found);
        return rgl.getGridletStatus();
    }
```

```
    // Find in Queue List
    found = super.findGridlet(gridletQueueList_, gridletId, userId);
    if (found >= 0) {
        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletQueueList_.get(found);
        return rgl.getGridletStatus();
    }

    // if not found in all 3 lists then no found
    return -1;
}

public void gridletSubmit(Gridlet gl, boolean ack) {
    StatisticalAnalysis.Communications++;
    // update the current Gridlets in exec list up to this point in time
    updateGridletProcessing();

    // reset number of PE since at the moment, it is not supported
    if (gl.getNumPE() > 1) {
        String userName = GridSim.getEntityName(gl.getUserID());
        System.out.println();
        System.out.println(super.get_name() + ".gridletSubmit(): "
                + " Gridlet #" + gl.getGridletID() + " from " + userName
                + " user requires " + gl.getNumPE() + " PEs.");
        System.out.println("--> Process this Gridlet to 1 PE only.");
        System.out.println();

        // also adjusted the length because the number of PEs are reduced
        int numPE = gl.getNumPE();
        double len = gl.getGridletLength();
        gl.setGridletLength(len * numPE);
        gl.setNumPE(1);
    }

    ResGridlet rgl = new ResGridlet(gl);
    boolean success = false;

    // if there is an available PE slot, then allocate immediately
    if (gridletInExecList_.size() < super.totalPE_) {
        success = allocatePEtoGridlet(rgl);
    }

    // if no available PE then put the ResGridlet into a Queue list
    if (!success) {
        rgl.setGridletStatus(Gridlet.QUEUED);
        gridletQueueList_.add(rgl);
    }

    // sends back an ack if required
    if (ack) {
        super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, true, gl
                .getGridletID(), gl.getUserID());
    }
}
```

```
public int getresId() {
    return resId_;
}

public double calculateTotalLoad() {
    return calculateTotalLoad(gridletQueueList_.size() +
            gridletInExecList_.size());
}
/////////////////////// Private Methods //////////////////////////////

protected double calculateTotalLoad(int size) {
    int totalRating = 0;
    PEList peList = (resource_.getMachineList().getMachine(0)).getPEList();
    for (int i = 0; i < peList.size(); i++) {
        totalRating += ((PE) peList.get(i)).getMIPSRating();
    }

    totalRating = totalRating / 10;
    // Devide by the lowest PE rate in the Grid.
    // Here we have 10 and 50 so we divide by 10
    double val = (size + 1.0) / totalRating;
    int numGridletPerPE = (int) Math.ceil(val);

    // load is between [0.0, 1.0] where 1.0 is busy and 0.0 is not busy
    double localLoad = resCalendar_.getCurrentLoad();
    double load = 1.0 - ((1 - localLoad) / numGridletPerPE);
    if (load < 0.0) {
        load = 0.0;
    }

    return load;
}

private void updateGridletProcessing() {
    // Identify MI share for the duration (from last event time)
    double time = GridSim.clock();
    double timeSpan = time - lastUpdateTime_;

    // if current time is the same or less than the last update time,
    // then ignore
    if (timeSpan <= 0.0) {
        return;
    }

    // Update Current Time as Last Update
    lastUpdateTime_ = time;

    // update the GridResource load
    //TODO : What ?!!!!
    int size = gridletInExecList_.size();
    double load = super.calculateTotalLoad(size);
    super.addTotalLoad(load);
```

```
        // if no Gridlets in execution then ignore the rest
        if (size == 0) {
            return;
        }

        ResGridlet obj;

        // a loop that allocates MI share for each Gridlet accordingly
        Iterator iter = gridletInExecList_.iterator();
        while (iter.hasNext()) {
            obj = (ResGridlet) iter.next();

            // Updates the Gridlet length that is currently being executed
            load = getMIShare(timeSpan, obj.getMachineID());
            obj.updateGridletFinishedSoFar(load);
        }
    }

    private void checkGridletCompletion() {
        ResGridlet obj;
        int i = 0;

        // NOTE: This one should stay as it is since gridletFinish()
        // will modify the content of this list if a Gridlet has finished.
        // Can't use iterator since it will cause an exception
        while (i < gridletInExecList_.size()) {
            obj = (ResGridlet) gridletInExecList_.get(i);

            if (obj.getRemainingGridletLength() == 0.0) {
                gridletInExecList_.remove(obj);
                gridletFinish(obj, Gridlet.SUCCESS);
                continue;
            }
            i++;
        }

        // if there are still Gridlets left in the execution
        // then send this into itself for an hourly interrupt
        // NOTE: Setting the internal event time too low will make the
        // simulation more realistic, BUT will take longer time to
        // run this simulation. Also, size of sim_trace will be HUGE!
        if (gridletInExecList_.size() > 0) {
            super.sendInternalEvent(60.0 * 60.0);
        }
    }

    private ResGridlet cancel(int gridletId, int userId) {
        ResGridlet rgl = null;

        // Find in QUEUE list
        int found = super.findGridlet(gridletQueueList_, gridletId, userId);
        if (found >= 0) {
            rgl = (ResGridlet) gridletQueueList_.remove(found);
            rgl.setGridletStatus(Gridlet.CANCELED);
```

```
    }
    return rgl;
}

/**
 * Allocates the first Gridlet in the Queue list (if any) to execution list
 *
 * @pre $none
 * @post $none
 */
private void allocateQueueGridlet() {
    // if there are many Gridlets in the QUEUE, then allocate a
    // PE to the first Gridlet in the list since it follows FCFS
    // (First Come First Serve) approach. Then removes the Gridlet from
    // the Queue list
    if (gridletQueueList_.size() > 0
            && gridletInExecList_.size() < super.totalPE_) {
        ResGridlet obj = (ResGridlet) gridletQueueList_.get(0);

        // allocate the Gridlet into an empty PE slot and remove it from
        // the queue list
        boolean success = allocatePEtoGridlet(obj);
        if (success) {
            gridletQueueList_.remove(obj);
        }
    }
}

private boolean allocatePEtoGridlet(ResGridlet rgl) {
    // IDENTIFY MACHINE which has a free PE and add this Gridlet to it.
    Machine myMachine = resource_.getMachineWithFreePE();

    // If a Machine is empty then ignore the rest
    if (myMachine == null) {
        return false;
    }

    // gets the list of PEs and find one empty PE
    PEList MyPEList = myMachine.getPEList();
    int freePE = MyPEList.getFreePEID();

    // ALLOCATE IMMEDIATELY
    rgl.setGridletStatus(Gridlet.INEXEC); // change Gridlet status
    rgl.setMachineAndPEID(myMachine.getMachineID(), freePE);

    // add this Gridlet into execution list
    gridletInExecList_.add(rgl);

    // Set allocated PE to BUSY status
    super.resource_.setStatusPE(PE.BUSY, rgl.getMachineID(), freePE);

    // Identify Completion Time and Set Interrupt
    int rating = machineRating_[rgl.getMachineID()];
    double time = forecastFinishTime(rating, rgl
```

```
                .getRemainingGridletLength());

        int roundUpTime = (int) (time + 1); // rounding up
        rgl.setFinishTime(roundUpTime);

        // then send this into itself
        super.sendInternalEvent(roundUpTime);
        return true;
    }

    private double forecastFinishTime(double availableRating, double length) {
        double finishTime = (length / availableRating);

        // This is as a safeguard since the finish time can be extremely
        // small close to 0.0, such as 4.5474735088646414E-14. Hence causing
        // some Gridlets never to be finished and consequently hang the program
        if (finishTime < 1.0) {
            finishTime = 1.0;
        }
        return finishTime;
    }

    private void gridletFinish(ResGridlet rgl, int status) {
        // Set PE on which Gridlet finished to FREE
        super.resource_.setStatusPE(PE.FREE, rgl.getMachineID(), rgl.getPEID());

        // the order is important! Set the status first then finalize
        // due to timing issues in ResGridlet class
        rgl.setGridletStatus(status);
        rgl.finalizeGridlet();
        super.sendFinishGridlet(rgl.getGridlet());
        allocateQueueGridlet(); // move Queued Gridlet into exec list
    }

    private double getMIShare(double timeSpan, int machineId) {
        // 1 - localLoad_ = available MI share percentage
        double localLoad = super.resCalendar_.getCurrentLoad();

        // each Machine might have different PE Rating compare to another
        // so much look at which Machine this PE belongs to
        double totalMI = machineRating_[machineId] * timeSpan * (1 - localLoad);
        return totalMI;
    }

}

package allocpolicy.statebroadcast;

/**
 * Created by IntelliJ IDEA.
 * User: azm537
 * Date: Nov 19, 2008
 * Time: 11:15:37 PM
 * To change this template use File | Settings | File Templates.
```

```
 */
public class IdLoadBean {
    private int id;
    private double load;


    public IdLoadBean(int id, double load) {
        this.id = id;
        this.load = load;

    }

    public int getId() {
        return id;
    }

    public void setId(int ip) {
        this.id = ip;
    }

    public double getLoad() {
        return load;
    }

    public void setLoad(double load) {
        this.load = load;
    }
}

package allocpolicy.statebroadcast;

/*
 * Title:        GridSim Toolkit
 * Description:  GridSim (Grid Simulation) Toolkit for Modeling and Simulation
 *               of Parallel and Distributed Systems such as Clusters and Grids
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
 *
 * $Id: TimeShared.java,v 1.39 2006/03/09 05:56:32 anthony Exp $
 */


import java.util.Iterator;
import java.util.HashMap;
import java.util.Random;
import java.util.Date;
import java.io.IOException;

import gridsim.*;
import eduni.simjava.*;
import grid.StatisticalAnalysis;
import sun.util.calendar.Gregorian;


public class SBA extends AllocPolicy {
```

```
private ResGridletList gridletInExecList_;  // storing exec Gridlets
private double lastUpdateTime_;   // a timer to denote the last update time
private MIShares share_;   // a temp variable
private HashMap ssv; // IP -> load
private boolean log = false;
private Sim_port output;


public SBA(String resourceName, String entityName, Sim_port output)
        throws Exception {
    super(resourceName, entityName);

    // initialises local data structure
    this.gridletInExecList_ = new ResGridletList();
    ssv = new HashMap();
    this.share_ = new MIShares();
    this.lastUpdateTime_ = 0.0;
    this.output = output;
}

public void body() {

    double time1, time2 = GridSim.clock();

    // a loop that is looking for internal events only
    sim_schedule(resId_, GridSimTags.SCHEDULE_NOW,
            GridSimTags.BROADCAST_STATE);
    Sim_event ev = new Sim_event();
    while (Sim_system.running()) {
        time1 = time2;
        time2 = GridSim.clock();

        if ((log) && (time2 - time1 > 200))
            try {
                StatisticalAnalysis.get_sba_log().writeChars(
                        String.valueOf(calculateTotalLoad(gridletInExecList_
                                .size())) + "\n");
            } catch (IOException e) {
                e.printStackTrace();
            }

        sim_schedule(resId_, GridSimTags.SCHEDULE_NOW,
                GridSimTags.BROADCAST_STATE);
        super.sim_get_next(ev);

        // if the simulation finishes then exit the loop
        if (ev.get_tag() == GridSimTags.END_OF_SIMULATION
                || super.isEndSimulation() == true) {
            break;
        }

        // Internal Event if the event source is this entity
        if (ev.get_src() == super.myId_) {
```

```
                internalEvent();
            }


        }

        // CHECK for ANY INTERNAL EVENTS WAITING TO BE PROCESSED
        while (super.sim_waiting() > 0) {
            // wait for event and ignore since it is likely to be related to
            // internal event scheduled to update Gridlets processing
            super.sim_get_next(ev);
            System.out.println(super.resName_ +
                    ".TimeShared.body(): ignoring internal events");
        }
    }

    /**
     * Schedules a new Gridlet that has been received by the GridResource
     * entity.
     *
     * @param gl  a Gridlet object that is going to be executed
     * @param ack an acknowledgement, i.e. <tt>true</tt> if wanted to know
     *            whether this operation is success or not, <tt>false</tt>
     *            otherwise (don't care)
     * @pre gl != null
     * @post $none
     */
    public void gridletSubmit(Gridlet gl, boolean ack) {


        StatisticalAnalysis.Communications++;

        // update Gridlets in execution up to this point in time
        updateGridletProcessing();

        // reset number of PE since at the moment, it is not supported
        if (gl.getNumPE() > 1) {
            String userName = GridSim.getEntityName(gl.getUserID());
            System.out.println();
            System.out.println(super.get_name() + ".gridletSubmit(): " +
                    " Gridlet #" + gl.getGridletID() + " from " + userName +
                    " user requires " + gl.getNumPE() + " PEs.");
            System.out.println("--> Process this Gridlet to 1 PE only.");
            System.out.println();

            // also adjusted the length because the number of PEs are reduced
            int numPE = gl.getNumPE();
            double len = gl.getGridletLength();
            gl.setGridletLength(len * numPE);
            gl.setNumPE(1);
        }

        Random random = new Random();
        if (!gl.isExecute()) {
```

```
            gl.setExecute(true);
            double minLoad = calculateTotalLoad(gridletInExecList_.size());
            int minNode = resId_;
            Iterator iterator = ssv.keySet().iterator();
//            System.out.println("SIZE: " + ssv.size());
            while (iterator.hasNext()) {
                int id = (Integer) iterator.next();
                if (((Double) ssv.get(id) == minLoad)) {
                    if (random.nextDouble() > 0.5)
                        minNode = id;
                } else if ((Double) ssv.get(id) < minLoad) {
                    minLoad = (Double) ssv.get(id);
                    minNode = id;
                }
            }
//            System.out.println("Send Job From " + resId_ + " with load:
// " + calculateTotalLoad(gridletInExecList_.size()) + " to " +
// minNode + " with load: " + minLoad);
            sim_schedule(output, 0, GridSimTags.GRIDLET_SUBMIT_ACK,
                    new IO_data(gl, gl.getGridletFileSize(),
                            minNode, 0));
        } else if (gl.isExecute()) {
            // adds a Gridlet to the in execution list
            ResGridlet rgl = new ResGridlet(gl);
            rgl.setGridletStatus(Gridlet.INEXEC); // set the Gridlet status to exec
            gridletInExecList_.add(rgl);   // add into the execution list

            // sends back an ack if required
            if (ack) {
                super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, true,
                        gl.getGridletID(), gl.getUserID());
            }
            sim_schedule(resId_, GridSimTags.SCHEDULE_NOW,
                    GridSimTags.BROADCAST_STATE);
        }
        // forecast all Gridlets in the execution list
        forecastGridlet();

    }

    /**
     * Finds the status of a specified Gridlet ID.
     *
     * @param gridletId a Gridlet ID
     * @param userId    the user or owner's ID of this Gridlet
     * @return the Gridlet status or <tt>-1</tt> if not found
     * @pre gridletId > 0
     * @pre userId > 0
     * @post $none
     * @see gridsim.Gridlet
     */
    public int gridletStatus(int gridletId, int userId) {
        ResGridlet rgl;
```

```
        // Find in EXEC List first
        int found = super.findGridlet(gridletInExecList_, gridletId, userId);
        if (found >= 0) {
            // Get the Gridlet from the execution list
            rgl = (ResGridlet) gridletInExecList_.get(found);
            return rgl.getGridletStatus();
        }

        // if not found in all lists
        return -1;
    }

    /**
     * Cancels a Gridlet running in this entity.
     * This method will search the execution and paused list. The User ID is
     * important as many users might have the same Gridlet ID in the lists.
     * <b>NOTE:</b>
     * <ul>
     * <li> Before canceling a Gridlet, this method updates all the
     * Gridlets in the execution list. If the Gridlet has no more MIs
     * to be executed, then it is considered to be <tt>finished</tt>.
     * Hence, the Gridlet can't be canceled.
     * <p/>
     * <li> Once a Gridlet has been canceled, it can't be resumed to
     * execute again since this method will pass the Gridlet back to
     * sender, i.e. the <tt>userId</tt>.
     * <p/>
     * <li> If a Gridlet can't be found in both execution and paused list,
     * then a <tt>null</tt> Gridlet will be send back to sender,
     * i.e. the <tt>userId</tt>.
     * </ul>
     *
     * @param gridletId a Gridlet ID
     * @param userId    the user or owner's ID of this Gridlet
     * @pre gridletId > 0
     * @pre userId > 0
     * @post $none
     */
    public void gridletCancel(int gridletId, int userId) {
        // Finds the gridlet in execution and paused list
        ResGridlet rgl = cancel(gridletId, userId);

        // If not found in both lists then report an error and sends back
        // an empty Gridlet
        if (rgl == null) {
            System.out.println(super.resName_ +
                    ".TimeShared.gridletCancel(): Cannot find " +
                    "Gridlet #" + gridletId + " for User #" + userId);

            super.sendCancelGridlet(GridSimTags.GRIDLET_CANCEL, null,
                    gridletId, userId);
            return;
        }
```

```
    // if a Gridlet is found
    rgl.finalizeGridlet();     // finalise Gridlet

    // if a Gridlet has finished execution before canceling, the reports
    // an error msg
    if (rgl.getGridletStatus() == Gridlet.SUCCESS) {
        System.out.println(super.resName_
                + ".TimeShared.gridletCancel(): Cannot cancel"
                + " Gridlet #" + gridletId + " for User #" + userId
                + " since it has FINISHED.");
    }

    // sends the Gridlet back to sender
    super.sendCancelGridlet(GridSimTags.GRIDLET_CANCEL, rgl.getGridlet(),
            gridletId, userId);
}

public void gridletPause(int gridletId, int userId, boolean ack) {

}

public void gridletResume(int gridletId, int userId, boolean ack) {

}

/**
 * Moves a Gridlet from this GridResource entity to a different one.
 * This method will search in both the execution and paused list.
 * The User ID is important as many Users might have the same Gridlet ID
 * in the lists.
 * <p/>
 * If a Gridlet has finished beforehand, then this method will send back
 * the Gridlet to sender, i.e. the <tt>userId</tt> and sets the
 * acknowledgment to false (if required).
 *
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @param destId    a new destination GridResource ID for this Gridlet
 * @param ack       an acknowledgement, i.e. <tt>true</tt> if wanted to know
 *                  whether this operation is success or not, <tt>false</tt>
 *                  otherwise (don't care)
 * @pre gridletId > 0
 * @pre userId > 0
 * @pre destId > 0
 * @post $none
 */
public void gridletMove(int gridletId, int userId, int destId, boolean ack) {
    // cancel the Gridlet first
    ResGridlet rgl = cancel(gridletId, userId);

    // If no found then print an error msg
    if (rgl == null) {
        System.out.println(super.resName_ +
                ".TimeShared.gridletMove(): Cannot find " +
```

```
                    "Gridlet #" + gridletId + " for User #" + userId);

            if (ack) {   // sends ack that this operation fails
                super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, false,
                        gridletId, userId);
            }
            return;
        }

        // if found
        rgl.finalizeGridlet();   // finalise Gridlet
        Gridlet gl = rgl.getGridlet();

        // if a Gridlet has finished execution
        if (gl.getGridletStatus() == Gridlet.SUCCESS) {
            System.out.println(super.resName_
                    + ".TimeShared.gridletMove(): Cannot move"
                    + " Gridlet #" + gridletId + " for User #" + userId
                    + " since it has FINISHED.");

            if (ack) {
                super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, false, gridletId,
                        userId);
            }

            super.sendFinishGridlet(gl);   // sends the Gridlet back to sender
        }
        // moves this Gridlet to another GridResource entity
        else {
            super.gridletMigrate(gl, destId, ack);
        }
    }

    public double getLoad() {
        return calculateTotalLoad(gridletInExecList_.size());
    }

    public void setSSV(IdLoadBean bean) {
        ssv.put(bean.getId(), bean.getLoad());
    }

//////////////////////////////// PRIVATE METHODS //////////////////////////////

    protected double calculateTotalLoad(int size) {
        int totalRating = 0;
        PEList peList = (resource_.getMachineList().getMachine(0)).getPEList();
        for (int i = 0; i < peList.size(); i++) {
            totalRating += ((PE) peList.get(i)).getMIPSRating();
        }

        totalRating = totalRating / 10; // Devide by the lowest PE rate in the Grid.
        //  Here we have 10 and 50 so we divide by 10
        double val = (size + 1.0) / totalRating;
        int numGridletPerPE = (int) Math.ceil(val);
```

```
    // load is between [0.0, 1.0] where 1.0 is busy and 0.0 is not busy
    double localLoad = resCalendar_.getCurrentLoad();
    double load = 1.0 - ((1 - localLoad) / numGridletPerPE);
    if (load < 0.0) {
        load = 0.0;
    }

    return load;
}

/**
 * Updates the execution of all Gridlets for a period of time.
 * The time period is determined from the last update time up to the
 * current time. Once this operation is successfull, then the last update
 * time refers to the current time.
 *
 * @pre $none
 * @post $none
 */
private void updateGridletProcessing() {
    // Identify MI share for the duration (from last event time)
    double time = GridSim.clock();
    double timeSpan = time - lastUpdateTime_;

    // if current time is the same or less than the last update time,
    // then ignore
    if (timeSpan <= 0.0) {
        return;
    }

    // Update Current Time as the Last Update
    lastUpdateTime_ = time;

    // update the GridResource load
    int size = gridletInExecList_.size();
    double load = super.calculateTotalLoad(size);
    super.addTotalLoad(load);       // add the current resource load

    // if no Gridlets in execution then ignore the rest
    if (size == 0) {
        return;
    }

    // gets MI Share for all Gridlets
    MIShares shares = getMIShare(timeSpan, size);
    ResGridlet obj = null;

    // a loop that allocates MI share for each Gridlet accordingly
    // In this algorithm, Gridlets at the front of the list
    // (range = 0 until MIShares.maxCount-1) will be given max MI value
    // For example, 2 PEs and 3 Gridlets. PE #0 processes Gridlet #0
    // PE #1 processes Gridlet #1 and Gridlet #2
    int i = 0;  // a counter
```

```
        Iterator iter = gridletInExecList_.iterator();
        while (iter.hasNext()) {
            obj = (ResGridlet) iter.next();

            // Updates the Gridlet length that is currently being executed
            if (i < shares.maxCount) {
                obj.updateGridletFinishedSoFar(shares.max);
            } else {
                obj.updateGridletFinishedSoFar(shares.min);
            }

            i++;   // increments i
        }
    }

    /**
     * Identifies MI share (max and min) for all Gridlets in
     * a given time duration
     *
     * @param timeSpan duration
     * @param size     total number of Gridlets in the execution list
     * @return the total MI share that a Gridlet gets for a given
     *         <tt>timeSpan</tt>
     */
    private MIShares getMIShare(double timeSpan, int size) {
        // 1 - localLoad_ = available MI share percentage
        double localLoad = super.resCalendar_.getCurrentLoad();
        double TotalMIperPE = super.resource_.getMIPSRatingOfOnePE() * timeSpan
                * (1 - localLoad);

        // This TimeShared is not Round Robin where each PE for 1 Gridlet only.
        // a PE can have more than one Gridlet executing.
        // minimum number of Gridlets that each PE runs.
        int glDIVpe = size / super.totalPE_;

        // number of PEs that run one extra Gridlet
        int glMODpe = size % super.totalPE_;

        // If num Gridlets in execution > total PEs in a GridResource,
        // then divide MIShare by the following constraint:
        // - obj.max = MIShare of a PE executing n Gridlets
        // - obj.min = MIShare of a PE executing n+1 Gridlets
        // - obj.maxCount = a threshold number of Gridlets will be assigned to
        //                  max MI value.
        //
        // In this algorithm, Gridlets at the front of the list
        // (range = 0 until maxCount-1) will be given max MI value
        if (glDIVpe > 0) {
            // this is for PEs that run one extra Gridlet
            share_.min = TotalMIperPE / (glDIVpe + 1);
            share_.max = TotalMIperPE / glDIVpe;
            share_.maxCount = (super.totalPE_ - glMODpe) * glDIVpe;
        }
```

```
        // num Gridlet in Exec < total PEs, meaning it is a
        // full PE share: i.e a PE is dedicated to execute a single Gridlet
        else {
            share_.max = TotalMIperPE;
            share_.min = TotalMIperPE;
            share_.maxCount = size;   // number of Gridlet
        }

        return share_;
    }

    /**
     * Determines the smallest completion time of all Gridlets in the execution
     * list. The smallest time is used as an internal event to
     * update Gridlets processing in the future.
     * <p/>
     * The algorithm for this method:
     * <ul>
     * <li> identify the finish time for each Gridlet in the execution list
     * given the share MIPS rating for all and the remaining Gridlet's
     * length
     * <li> find the smallest finish time in the list
     * <li> send the last Gridlet in the list with
     * <tt>delay =  smallest finish time - current time</tt>
     * </ul>
     *
     * @pre $none
     * @post $none
     */
    private void forecastGridlet() {
        // if no Gridlets available in exec list, then exit this method
        if (gridletInExecList_.size() == 0) {
            return;
        }

        // checks whether Gridlets have finished or not. If yes, then remove
        // them since they will effect the MIShare calculation.
        checkGridletCompletion();

        // Identify MIPS share for all Gridlets for 1 second, considering
        // current Gridlets + No of PEs.
        MIShares share = getMIShare(1.0, gridletInExecList_.size());

        ResGridlet rgl;
        int i = 0;
        double time;
        double rating;
        double smallestTime = 0.0;

        // For each Gridlet, determines their finish time
        Iterator iter = gridletInExecList_.iterator();
        while (iter.hasNext()) {
            rgl = (ResGridlet) iter.next();
```

```
            // If a Gridlet locates before the max count then it will be given
            // the max. MIPS rating
            if (i < share.maxCount) {
                rating = share.max;
            } else {   // otherwise, it will be given the min. MIPS Rating
                rating = share.min;
            }

            time = forecastFinishTime(rating, rgl.getRemainingGridletLength());

            int roundUpTime = (int) (time + 1);   // rounding up
            rgl.setFinishTime(roundUpTime);

            // get the smallest time of all Gridlets
            if (i == 0 || smallestTime > time) {
                smallestTime = time;
            }

            i++;
        }

        // sends to itself as an internal event
        super.sendInternalEvent(smallestTime);
    }

    /**
     * Checks all Gridlets in the execution list whether they are finished or
     * not.
     *
     * @pre $none
     * @post $none
     */
    private void checkGridletCompletion() {
        ResGridlet rgl;

        // a loop that determine the smallest finish time of a Gridlet
        // Don't use an iterator since it causes an exception because if
        // a Gridlet is finished, gridletFinish() will remove it from the list.
        int i = 0;
        while (i < gridletInExecList_.size()) {
            rgl = (ResGridlet) gridletInExecList_.get(i);

            // if a Gridlet has finished, then remove it from the list
            if (rgl.getRemainingGridletLength() <= 0.0) {
                gridletFinish(rgl, Gridlet.SUCCESS);
                continue;  // not increment i coz the list size also decreases
            }

            i++;
        }
    }

    /**
     * Forecast finish time of a Gridlet.
```

```
     * <tt>Finish time = length / available rating</tt>
     *
     * @param availableRating the shared MIPS rating for all Gridlets
     * @param length          remaining Gridlet length
     * @return Gridlet's finish time.
     */
    private double forecastFinishTime(double availableRating, double length) {
        double finishTime = length / availableRating;

        // This is as a safeguard since the finish time can be extremely
        // small close to 0.0, such as 4.5474735088646414E-14. Hence causing
        // some Gridlets never to be finished and consequently hang the program
        if (finishTime < 1.0) {
            finishTime = 1.0;
        }

        return finishTime;
    }

    /**
     * Updates the Gridlet's properties, such as status once a
     * Gridlet is considered finished.
     *
     * @param rgl    a ResGridlet object
     * @param status the status of this ResGridlet object
     * @pre rgl != null
     * @post $none
     */
    private void gridletFinish(ResGridlet rgl, int status) {
        // NOTE: the order is important! Set the status first then finalize
        // due to timing issues in ResGridlet class.
        rgl.setGridletStatus(status);
        rgl.finalizeGridlet();

        // sends back the Gridlet with no delay
        Gridlet gl = rgl.getGridlet();
        super.sendFinishGridlet(gl);

        // remove this Gridlet in the execution
        gridletInExecList_.remove(rgl);
        sim_schedule(resId_, GridSimTags.SCHEDULE_NOW, GridSimTags.BROADCAST_STATE);
    }

    /**
     * Handles internal event
     *
     * @pre $none
     * @post $none
     */
    private void internalEvent() {
        // this is a constraint that prevents an infinite loop
        // Compare between 2 floating point numbers. This might be incorrect
        // for some hardware platform.
        if (lastUpdateTime_ == GridSim.clock()) {
```

```
            return;
        }
        // update Gridlets in execution up to this point in time
        updateGridletProcessing();

        // schedule next event
        forecastGridlet();
    }

    private ResGridlet cancel(int gridletId, int userId) {
        ResGridlet rgl = null;

        // Check whether the Gridlet is in execution list or not
        int found = super.findGridlet(gridletInExecList_, gridletId, userId);

        // if a Gridlet is in execution list
        if (found >= 0) {
            // update the gridlets in execution list up to this point in time
            updateGridletProcessing();

            // Get the Gridlet from the execution list
            rgl = (ResGridlet) gridletInExecList_.remove(found);

            // if a Gridlet is finished upon cancelling, then set it to success
            if (rgl.getRemainingGridletLength() == 0.0) {
                rgl.setGridletStatus(Gridlet.SUCCESS);
            } else {
                rgl.setGridletStatus(Gridlet.CANCELED);
            }

            // then forecast the next Gridlet to complete
            forecastGridlet();
        }
        return rgl;
    }

////////////////////////////////// INTERNAL CLASS ///////////////////////////////

    /**
     * Gridlets MI share in Time Shared Mode
     */
    private class MIShares {
        /**
         * maximum amount of MI share Gridlets can get
         */
        public double max;

        /**
         * minimum amount of MI share Gridlets can get when
         * it is executed on a PE that runs one extra Gridlet
         */
        public double min;

        /**
```

```
         * Total number of Gridlets that get Max share
         */
        public int maxCount;

        /**
         * Default constructor that initializes all attributes to 0
         *
         * @pre $none
         * @post $none
         */
        public MIShares() {
            max = 0.0;
            min = 0.0;
            maxCount = 0;
        }
    } // end of internal class

}

/*
 * Title:        GridSim Toolkit
 * Description:  GridSim (Grid Simulation) Toolkit for Modeling and Simulation
 *               of Parallel and Distributed Systems such as Clusters and Grids
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
 *
 * $Id: SBA.java,v 1.28 2006/03/09 05:56:31 anthony Exp $
 */

package allocpolicy;

import java.util.Iterator;
import java.io.IOException;

import eduni.simjava.Sim_event;
import eduni.simjava.Sim_system;
import gridsim.*;
import grid.StatisticalAnalysis;
import grid.SimulationWithoutFailure;


/**
 * SBA class is an allocation policy for GridResource that behaves
 * exactly like First Come First Serve (FCFS). This is a basic and simple
 * scheduler that runs each Gridlet to one Processing Element (PE).
 * If a Gridlet requires more than one PE, then this scheduler only assign
 * this Gridlet to one PE.
 *
 * @author Manzur Murshed and Rajkumar Buyya
 * @author Anthony Sulistio (re-written this class)
 * @invariant $none
 * @see gridsim.GridSim
 * @see gridsim.ResourceCharacteristics
 * @since GridSim Toolkit 2.2
 */
```

```
public class SpaceShared extends AllocPolicy {
    private ResGridletList gridletQueueList_;     // Queue list
    private ResGridletList gridletInExecList_;    // Execution list
    private ResGridletList gridletPausedList_;    // Pause list
    private double lastUpdateTime_;    // the last time Gridlets updated
    private int[] machineRating_;      // list of machine ratings available

    private double time1 = 0;
    private double time2 = 0;
    private boolean log = false;


    /**
     * Allocates a new SBA object
     *
     * @param resourceName the GridResource entity name that will contain
     *                     this allocation policy
     * @param entityName   this object entity name
     * @throws Exception This happens when one of the following scenarios occur:
     *                   <ul>
     *                   <li> creating this entity before initializing GridSim
     *                          package
     *                   <li> this entity name is <tt>null</tt> or empty
     *                   <li> this entity has <tt>zero</tt> number of PEs
     *                          (Processing
     *                   Elements). <br>
     *                   No PEs mean the Gridlets can't be processed.
     *                   A GridResource must contain one or more Machines.
     *                   A Machine must contain one or more PEs.
     *                   </ul>
     * @pre resourceName != null
     * @pre entityName != null
     * @post $none
     * @see gridsim.GridSim#init(int,java.util.Calendar,boolean,String[],String[],
     *String)
     */
    public SpaceShared(String resourceName, String entityName) throws Exception {
        super(resourceName, entityName);

        // initialises local data structure
        this.gridletInExecList_ = new ResGridletList();
        this.gridletPausedList_ = new ResGridletList();
        this.gridletQueueList_ = new ResGridletList();
        this.lastUpdateTime_ = 0.0;
        this.machineRating_ = null;
    }

    /**
     * Handles internal events that are coming to this entity.
     *
     * @pre $none
     * @post $none
     */
    public void body() {
```

```
        // Gets the PE's rating for each Machine in the list.
        // Assumed one Machine has same PE rating.
        MachineList list = super.resource_.getMachineList();
        int size = list.size();
        machineRating_ = new int[size];
        for (int i = 0; i < size; i++) {
            machineRating_[i] = super.resource_.getMIPSRatingOfOnePE(i, 0);
        }

        // a loop that is looking for internal events only
        Sim_event ev = new Sim_event();
        while (Sim_system.running()) {
            time1 = time2;
            time2 = GridSim.clock();

            if ((log) && (time2 - time1 > 200))
                try {
                    StatisticalAnalysis.getRandom_SpaceShared_log().
                            writeChars(String.valueOf(calculateTotalLoad(
                                    gridletInExecList_.size() + gridletQueueList_.
                                            size())) + "\n");
                } catch (IOException e) {
                    e.printStackTrace();
                }
            super.sim_get_next(ev);

            // if the simulation finishes then exit the loop
            if (ev.get_tag() == GridSimTags.END_OF_SIMULATION ||
                    super.isEndSimulation() == true) {
                break;
            }

            // Internal Event if the event source is this entity
            if (ev.get_src() == super.myId_ && gridletInExecList_.size() > 0) {
                updateGridletProcessing();   // update Gridlets
                checkGridletCompletion();    // check for finished Gridlets
            }
        }

        // CHECK for ANY INTERNAL EVENTS WAITING TO BE PROCESSED
        while (super.sim_waiting() > 0) {
            // wait for event and ignore since it is likely to be related to
            // internal event scheduled to update Gridlets processing
            super.sim_get_next(ev);
            System.out.println(super.resName_ +
                    ".SBA.body(): ignore internal events");
        }
    }

    /**
     * Schedules a new Gridlet that has been received by the GridResource
     * entity.
     *
     * @param gl  a Gridlet object that is going to be executed
```

```
 * @param ack an acknowledgement, i.e. <tt>true</tt> if wanted to know
 *            whether this operation is success or not, <tt>false</tt>
 *            otherwise (don't care)
 * @pre gl != null
 * @post $none
 */
public void gridletSubmit(Gridlet gl, boolean ack) {

    StatisticalAnalysis.Communications++;
    // update the current Gridlets in exec list up to this point in time
    updateGridletProcessing();

    // reset number of PE since at the moment, it is not supported
    if (gl.getNumPE() > 1) {
        String userName = GridSim.getEntityName(gl.getUserID());
        System.out.println();
        System.out.println(super.get_name() + ".gridletSubmit(): " +
                " Gridlet #" + gl.getGridletID() + " from " + userName +
                " user requires " + gl.getNumPE() + " PEs.");
        System.out.println("--> Process this Gridlet to 1 PE only.");
        System.out.println();

        // also adjusted the length because the number of PEs are reduced
        int numPE = gl.getNumPE();
        double len = gl.getGridletLength();
        gl.setGridletLength(len * numPE);
        gl.setNumPE(1);
    }

    ResGridlet rgl = new ResGridlet(gl);
    boolean success = false;

    // if there is an available PE slot, then allocate immediately
    if (gridletInExecList_.size() < super.totalPE_) {
        success = allocatePEtoGridlet(rgl);
    }

    // if no available PE then put the ResGridlet into a Queue list
    if (success == false) {
        rgl.setGridletStatus(Gridlet.QUEUED);
        gridletQueueList_.add(rgl);
    }

    // sends back an ack if required
    if (ack == true) {
        super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, true,
                gl.getGridletID(), gl.getUserID()
        );
    }
}

/**
 * Finds the status of a specified Gridlet ID.
 *
```

```
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @return the Gridlet status or <tt>-1</tt> if not found
 * @pre gridletId > 0
 * @pre userId > 0
 * @post $none
 * @see gridsim.Gridlet
 */
public int gridletStatus(int gridletId, int userId) {
    ResGridlet rgl = null;

    // Find in EXEC List first
    int found = super.findGridlet(gridletInExecList_, gridletId, userId);
    if (found >= 0) {
        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletInExecList_.get(found);
        return rgl.getGridletStatus();
    }

    // Find in Paused List
    found = super.findGridlet(gridletPausedList_, gridletId, userId);
    if (found >= 0) {
        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletPausedList_.get(found);
        return rgl.getGridletStatus();
    }

    // Find in Queue List
    found = super.findGridlet(gridletQueueList_, gridletId, userId);
    if (found >= 0) {
        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletQueueList_.get(found);
        return rgl.getGridletStatus();
    }

    // if not found in all 3 lists then no found
    return -1;
}

/**
 * Cancels a Gridlet running in this entity.
 * This method will search the execution, queued and paused list.
 * The User ID is
 * important as many users might have the same Gridlet ID in the lists.
 * <b>NOTE:</b>
 * <ul>
 * <li> Before canceling a Gridlet, this method updates all the
 * Gridlets in the execution list. If the Gridlet has no more MIs
 * to be executed, then it is considered to be <tt>finished</tt>.
 * Hence, the Gridlet can't be canceled.
 * <p/>
 * <li> Once a Gridlet has been canceled, it can't be resumed to
 * execute again since this method will pass the Gridlet back to
 * sender, i.e. the <tt>userId</tt>.
```

```
 * <p/>
 * <li> If a Gridlet can't be found in both execution and paused list,
 * then a <tt>null</tt> Gridlet will be send back to sender,
 * i.e. the <tt>userId</tt>.
 * </ul>
 *
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @pre gridletId > 0
 * @pre userId > 0
 * @post $none
 */
public void gridletCancel(int gridletId, int userId) {
    // cancels a Gridlet
    ResGridlet rgl = cancel(gridletId, userId);

    // if the Gridlet is not found
    if (rgl == null) {
        System.out.println(super.resName_ +
                ".SBA.gridletCancel(): Cannot find " +
                "Gridlet #" + gridletId + " for User #" + userId);

        super.sendCancelGridlet(GridSimTags.GRIDLET_CANCEL, null,
                gridletId, userId);
        return;
    }

    // if the Gridlet has finished beforehand then prints an error msg
    if (rgl.getGridletStatus() == Gridlet.SUCCESS) {
        System.out.println(super.resName_
                + ".SBA.gridletCancel(): Cannot cancel"
                + " Gridlet #" + gridletId + " for User #" + userId
                + " since it has FINISHED.");
    }

    // sends the Gridlet back to sender
    rgl.finalizeGridlet();
    super.sendCancelGridlet(GridSimTags.GRIDLET_CANCEL, rgl.getGridlet(),
            gridletId, userId);
}

/**
 * Pauses a Gridlet only if it is currently executing.
 * This method will search in the execution list. The User ID is
 * important as many users might have the same Gridlet ID in the lists.
 *
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @param ack       an acknowledgement, i.e. <tt>true</tt> if wanted to know
 *                  whether this operation is success or not, <tt>false</tt>
 *                  otherwise (don't care)
 * @pre gridletId > 0
 * @pre userId > 0
 * @post $none
```

```
 */
public void gridletPause(int gridletId, int userId, boolean ack) {
    boolean status = false;

    // Find in EXEC List first
    int found = super.findGridlet(gridletInExecList_, gridletId, userId);
    if (found >= 0) {
        // updates all the Gridlets first before pausing
        updateGridletProcessing();

        // Removes the Gridlet from the execution list
        ResGridlet rgl = (ResGridlet) gridletInExecList_.remove(found);

        // if a Gridlet is finished upon cancelling, then set it to success
        // instead.
        if (rgl.getRemainingGridletLength() == 0.0) {
            found = -1;  // meaning not found in Queue List
            gridletFinish(rgl, Gridlet.SUCCESS);
            System.out.println(super.resName_
                    + ".SBA.gridletPause(): Cannot pause"
                    + " Gridlet #" + gridletId + " for User #" + userId
                    + " since it has FINISHED.");
        } else {
            status = true;
            rgl.setGridletStatus(Gridlet.PAUSED);  // change the status
            gridletPausedList_.add(rgl);   // add into the paused list

            // Set the PE on which Gridlet finished to FREE
            super.resource_.setStatusPE(PE.FREE, rgl.getMachineID(),
                    rgl.getPEID());

            // empty slot is available, hence process a new Gridlet
            allocateQueueGridlet();
        }
    } else {      // Find in QUEUE list
        found = super.findGridlet(gridletQueueList_, gridletId, userId);
    }

    // if found in the Queue List
    if (status == false && found >= 0) {
        status = true;

        // removes the Gridlet from the Queue list
        ResGridlet rgl = (ResGridlet) gridletQueueList_.remove(found);
        rgl.setGridletStatus(Gridlet.PAUSED);   // change the status
        gridletPausedList_.add(rgl);            // add into the paused list
    }
    // if not found anywhere in both exec and paused lists
    else if (found == -1) {
        System.out.println(super.resName_ +
                ".SBA.gridletPause(): Error - cannot " +
                "find Gridlet #" + gridletId + " for User #" + userId);
    }
```

```
        // sends back an ack if required
        if (ack == true) {
            super.sendAck(GridSimTags.GRIDLET_PAUSE_ACK, status,
                    gridletId, userId);
        }
    }

    /**
     * Moves a Gridlet from this GridResource entity to a different one.
     * This method will search in both the execution and paused list.
     * The User ID is important as many Users might have the same Gridlet ID
     * in the lists.
     * <p/>
     * If a Gridlet has finished beforehand, then this method will send back
     * the Gridlet to sender, i.e. the <tt>userId</tt> and sets the
     * acknowledgment to false (if required).
     *
     * @param gridletId a Gridlet ID
     * @param userId    the user or owner's ID of this Gridlet
     * @param destId    a new destination GridResource ID for this Gridlet
     * @param ack       an acknowledgement, i.e. <tt>true</tt> if wanted to know
     *                  whether this operation is success or not, <tt>false</tt>
     *                  otherwise (don't care)
     * @pre gridletId > 0
     * @pre userId > 0
     * @pre destId > 0
     * @post $none
     */
    public void gridletMove(int gridletId, int userId, int destId, boolean ack) {
        // cancels the Gridlet
        ResGridlet rgl = cancel(gridletId, userId);

        // if the Gridlet is not found
        if (rgl == null) {
            System.out.println(super.resName_ +
                    ".SBA.gridletMove(): Cannot find " +
                    "Gridlet #" + gridletId + " for User #" + userId);

            if (ack == true)   // sends back an ack if required
            {
                super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, false,
                        gridletId, userId);
            }

            return;
        }

        // if the Gridlet has finished beforehand
        if (rgl.getGridletStatus() == Gridlet.SUCCESS) {
            System.out.println(super.resName_
                    + ".SBA.gridletMove(): Cannot move Gridlet #"
                    + gridletId + " for User #" + userId
                    + " since it has FINISHED.");
```

```
        if (ack == true) // sends back an ack if required
        {
            super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, false,
                    gridletId, userId);
        }

        gridletFinish(rgl, Gridlet.SUCCESS);
    } else   // otherwise moves this Gridlet to a different GridResource
    {
        rgl.finalizeGridlet();

        // Set PE on which Gridlet finished to FREE
        super.resource_.setStatusPE(PE.FREE, rgl.getMachineID(),
                rgl.getPEID());

        super.gridletMigrate(rgl.getGridlet(), destId, ack);
        allocateQueueGridlet();
    }
}

/**
 * Resumes a Gridlet only in the paused list.
 * The User ID is important as many Users might have the same Gridlet ID
 * in the lists.
 *
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @param ack       an acknowledgement, i.e. <tt>true</tt> if wanted to know
 *                  whether this operation is success or not, <tt>false</tt>
 *                  otherwise (don't care)
 * @pre gridletId > 0
 * @pre userId > 0
 * @post $none
 */
public void gridletResume(int gridletId, int userId, boolean ack) {
    boolean status = false;

    // finds the Gridlet in the execution list first
    int found = super.findGridlet(gridletPausedList_, gridletId, userId);
    if (found >= 0) {
        // removes the Gridlet
        ResGridlet rgl = (ResGridlet) gridletPausedList_.remove(found);
        rgl.setGridletStatus(Gridlet.RESUMED);

        // update the Gridlets up to this point in time
        updateGridletProcessing();
        status = true;

        // if there is an available PE slot, then allocate immediately
        boolean success = false;
        if (gridletInExecList_.size() < super.totalPE_) {
            success = allocatePEtoGridlet(rgl);
        }
```

```
            // otherwise put into Queue list
            if (success == false) {
                rgl.setGridletStatus(Gridlet.QUEUED);
                gridletQueueList_.add(rgl);
            }

            System.out.println(super.resName_ + "TimeShared.gridletResume():" +
                    " Gridlet #" + gridletId + " with User ID #" +
                    userId + " has been sucessfully RESUMED.");
        } else {
            System.out.println(super.resName_ +
                    "TimeShared.gridletResume(): Cannot find " +
                    "Gridlet #" + gridletId + " for User #" + userId);
        }

        // sends back an ack if required
        if (ack == true) {
            super.sendAck(GridSimTags.GRIDLET_RESUME_ACK, status,
                    gridletId, userId);
        }
    }

    ///////////////////////////// PRIVATE METHODS /////////////////////
    protected double calculateTotalLoad(int size) {
        int totalRating = 0;
        PEList peList = (resource_.getMachineList().getMachine(0)).getPEList();
        for (int i = 0; i < peList.size(); i++) {
            totalRating += ((PE) peList.get(i)).getMIPSRating();
        }

        totalRating = totalRating / 10;
        // Devide by the lowest PE rate in the Grid.
        // Here we have 10 and 50 so we divide by 10
        double val = (size + 1.0) / totalRating;
        int numGridletPerPE = (int) Math.ceil(val);

        // load is between [0.0, 1.0] where 1.0 is busy and 0.0 is not busy
        double localLoad = resCalendar_.getCurrentLoad();
        double load = 1.0 - ((1 - localLoad) / numGridletPerPE);
        if (load < 0.0) {
            load = 0.0;
        }

        return load;
    }

    /**
     * Allocates the first Gridlet in the Queue list (if any) to execution list
     *
     * @pre $none
     * @post $none
     */
    private void allocateQueueGridlet() {
        // if there are many Gridlets in the QUEUE, then allocate a
```

```
        // PE to the first Gridlet in the list since it follows FCFS
        // (First Come First Serve) approach. Then removes the Gridlet from
        // the Queue list
        if (gridletQueueList_.size() > 0 &&
                gridletInExecList_.size() < super.totalPE_) {
            ResGridlet obj = (ResGridlet) gridletQueueList_.get(0);

            // allocate the Gridlet into an empty PE slot and remove it from
            // the queue list
            boolean success = allocatePEtoGridlet(obj);
            if (success == true) {
                gridletQueueList_.remove(obj);
            }
        }
    }

    /**
     * Updates the execution of all Gridlets for a period of time.
     * The time period is determined from the last update time up to the
     * current time. Once this operation is successfull, then the last update
     * time refers to the current time.
     *
     * @pre $none
     * @post $none
     */
    private void updateGridletProcessing() {
        // Identify MI share for the duration (from last event time)
        double time = GridSim.clock();
        double timeSpan = time - lastUpdateTime_;

        // if current time is the same or less than the last update time,
        // then ignore
        if (timeSpan <= 0.0) {
            return;
        }

        // Update Current Time as Last Update
        lastUpdateTime_ = time;

        // update the GridResource load
        int size = gridletInExecList_.size();
        double load = super.calculateTotalLoad(size);
        super.addTotalLoad(load);

        // if no Gridlets in execution then ignore the rest
        if (size == 0) {
            return;
        }

        ResGridlet obj = null;

        // a loop that allocates MI share for each Gridlet accordingly
        Iterator iter = gridletInExecList_.iterator();
        while (iter.hasNext()) {
```

```
            obj = (ResGridlet) iter.next();

            // Updates the Gridlet length that is currently being executed
            load = getMIShare(timeSpan, obj.getMachineID());
            obj.updateGridletFinishedSoFar(load);
        }
    }

    /**
     * Identifies MI share (max and min) each Gridlet gets for
     * a given timeSpan
     *
     * @param timeSpan  duration
     * @param machineId machine ID that executes this Gridlet
     * @return the total MI share that a Gridlet gets for a given
     *         <tt>timeSpan</tt>
     * @pre timeSpan >= 0.0
     * @pre machineId > 0
     * @post $result >= 0.0
     */
    private double getMIShare(double timeSpan, int machineId) {
        // 1 - localLoad_ = available MI share percentage
        double localLoad = super.resCalendar_.getCurrentLoad();

        // each Machine might have different PE Rating compare to another
        // so much look at which Machine this PE belongs to
        double totalMI = machineRating_[machineId] * timeSpan * (1 - localLoad);
        return totalMI;
    }

    /**
     * Allocates a Gridlet into a free PE and sets the Gridlet status into
     * INEXEC and PE status into busy afterwards
     *
     * @param rgl a ResGridlet object
     * @return <tt>true</tt> if there is an empty PE to process this Gridlet,
     *         <tt>false</tt> otherwise
     * @pre rgl != null
     * @post $none
     */
    private boolean allocatePEtoGridlet(ResGridlet rgl) {
        // IDENTIFY MACHINE which has a free PE and add this Gridlet to it.
        Machine myMachine = resource_.getMachineWithFreePE();

        // If a Machine is empty then ignore the rest
        if (myMachine == null) {
            return false;
        }

        // gets the list of PEs and find one empty PE
        PEList MyPEList = myMachine.getPEList();
        int freePE = MyPEList.getFreePEID();

        // ALLOCATE IMMEDIATELY
```

```
        rgl.setGridletStatus(Gridlet.INEXEC);   // change Gridlet status
        rgl.setMachineAndPEID(myMachine.getMachineID(), freePE);

        // add this Gridlet into execution list
        gridletInExecList_.add(rgl);

        // Set allocated PE to BUSY status
        super.resource_.setStatusPE(PE.BUSY, rgl.getMachineID(), freePE);

        // Identify Completion Time and Set Interrupt
        int rating = machineRating_[rgl.getMachineID()];
        double time = forecastFinishTime(rating,
                rgl.getRemainingGridletLength());

        int roundUpTime = (int) (time + 1);   // rounding up
        rgl.setFinishTime(roundUpTime);

        // then send this into itself
        super.sendInternalEvent(roundUpTime);
        return true;
    }

    /**
     * Forecast finish time of a Gridlet.
     * <tt>Finish time = length / available rating</tt>
     *
     * @param availableRating the shared MIPS rating for all Gridlets
     * @param length          remaining Gridlet length
     * @return Gridlet's finish time.
     * @pre availableRating >= 0.0
     * @pre length >= 0.0
     * @post $none
     */
    private double forecastFinishTime(double availableRating, double length) {
        double finishTime = (length / availableRating);

        // This is as a safeguard since the finish time can be extremely
        // small close to 0.0, such as 4.5474735088646414E-14. Hence causing
        // some Gridlets never to be finished and consequently hang the program
        if (finishTime < 1.0) {
            finishTime = 1.0;
        }

        return finishTime;
    }

    /**
     * Checks all Gridlets in the execution list whether they are finished or
     * not.
     *
     * @pre $none
     * @post $none
     */
    private void checkGridletCompletion() {
```

```
    ResGridlet obj = null;
    int i = 0;

    // NOTE: This one should stay as it is since gridletFinish()
    // will modify the content of this list if a Gridlet has finished.
    // Can't use iterator since it will cause an exception
    while (i < gridletInExecList_.size()) {
        obj = (ResGridlet) gridletInExecList_.get(i);

        if (obj.getRemainingGridletLength() == 0.0) {
            gridletInExecList_.remove(obj);
            gridletFinish(obj, Gridlet.SUCCESS);
            continue;
        }

        i++;
    }

    // if there are still Gridlets left in the execution
    // then send this into itself for an hourly interrupt
    // NOTE: Setting the internal event time too low will make the
    //       simulation more realistic, BUT will take longer time to
    //       run this simulation. Also, size of sim_trace will be HUGE!
    if (gridletInExecList_.size() > 0) {
        super.sendInternalEvent(60.0 * 60.0);
    }
}

/**
 * Updates the Gridlet's properties, such as status once a
 * Gridlet is considered finished.
 *
 * @param rgl    a ResGridlet object
 * @param status the Gridlet status
 * @pre rgl != null
 * @pre status >= 0
 * @post $none
 */
private void gridletFinish(ResGridlet rgl, int status) {
    // Set PE on which Gridlet finished to FREE
    super.resource_.setStatusPE(PE.FREE, rgl.getMachineID(), rgl.getPEID());

    // the order is important! Set the status first then finalize
    // due to timing issues in ResGridlet class
    rgl.setGridletStatus(status);
    rgl.finalizeGridlet();
    super.sendFinishGridlet(rgl.getGridlet());

    allocateQueueGridlet();   // move Queued Gridlet into exec list
}


private ResGridlet cancel(int gridletId, int userId) {
    ResGridlet rgl = null;
```

```
        // Find in EXEC List first
        int found = super.findGridlet(gridletInExecList_, gridletId, userId);
        if (found >= 0) {
            // update the gridlets in execution list up to this point in time
            updateGridletProcessing();

            // Get the Gridlet from the execution list
            rgl = (ResGridlet) gridletInExecList_.remove(found);

            // if a Gridlet is finished upon cancelling, then set it to success
            // instead.
            if (rgl.getRemainingGridletLength() == 0.0) {
                rgl.setGridletStatus(Gridlet.SUCCESS);
            } else {
                rgl.setGridletStatus(Gridlet.CANCELED);
            }

            // Set PE on which Gridlet finished to FREE
            super.resource_.setStatusPE(PE.FREE, rgl.getMachineID(),
                    rgl.getPEID());
            allocateQueueGridlet();
            return rgl;
        }

        // Find in QUEUE list
        found = super.findGridlet(gridletQueueList_, gridletId, userId);
        if (found >= 0) {
            rgl = (ResGridlet) gridletQueueList_.remove(found);
            rgl.setGridletStatus(Gridlet.CANCELED);
        }

        // if not, then find in the Paused list
        else {
            found = super.findGridlet(gridletPausedList_, gridletId, userId);

            // if found in Paused list
            if (found >= 0) {
                rgl = (ResGridlet) gridletPausedList_.remove(found);
                rgl.setGridletStatus(Gridlet.CANCELED);
            }

        }
        return rgl;
    }
    /*protected double calculateTotalLoad(int size) {
        return size / (totalPE_  - (totalPE_ * resCalendar_.getCurrentLoad()));
    }*/

} // end class

/*
 * Title:        GridSim Toolkit
 * Description:  GridSim (Grid Simulation) Toolkit for Modeling and Simulation
```

```
 *               of Parallel and Distributed Systems such as Clusters and Grids
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
 *
 * $Id: TimeShared.java,v 1.39 2006/03/09 05:56:32 anthony Exp $
 */

package allocpolicy;

import java.util.Iterator;
import java.io.IOException;

import gridsim.*;
import eduni.simjava.*;
import grid.StatisticalAnalysis;


/**
 * TimeShared class is an allocation policy for GridResource that behaves
 * similar to a round robin algorithm, except that all Gridlets are
 * executed at the same time.
 * This is a basic and simple
 * scheduler that runs each Gridlet to one Processing Element (PE).
 * If a Gridlet requires more than one PE, then this scheduler only assign
 * this Gridlet to one PE.
 *
 * @author Manzur Murshed and Rajkumar Buyya
 * @author Anthony Sulistio (re-written this class)
 * @invariant $none
 * @see gridsim.GridSim
 * @see gridsim.ResourceCharacteristics
 * @since GridSim Toolkit 2.2
 */
public class TimeShared extends AllocPolicy {
    private ResGridletList gridletInExecList_;  // storing exec Gridlets
    private ResGridletList gridletPausedList_;  // storing Paused Gridlets
    private double lastUpdateTime_;   // a timer to denote the last update time
    private MIShares share_;   // a temp variable

    private double time1 = 0;
    private double time2 = 0;
    private boolean log = false;


    /**
     * Allocates a new TimeShared object
     *
     * @param resourceName the GridResource entity name that will contain
     *                     this allocation policy
     * @param entityName   this object entity name
     * @throws Exception This happens when one of the following scenarios occur:
     *                   <ul>
     *                   <li> creating this entity before initializing
     *                         GridSim package
     *                   <li> this entity name is <tt>null</tt> or empty
```

```
 *                   <li> this entity has <tt>zero</tt> number of PEs
 *                          (Processing
 *                   Elements). <br>
 *                   No PEs mean the Gridlets can't be processed.
 *                   A GridResource must contain one or more Machines.
 *                   A Machine must contain one or more PEs.
 *                   </ul>
 * @pre resourceName != null
 * @pre entityName != null
 * @post $none
 * @see gridsim.GridSim#init(int,java.util.Calendar,boolean,String[],String[],
 *String)
 */
public TimeShared(String resourceName, String entityName) throws Exception {
    super(resourceName, entityName);

    // initialises local data structure
    this.gridletInExecList_ = new ResGridletList();
    this.gridletPausedList_ = new ResGridletList();
    this.share_ = new MIShares();
    this.lastUpdateTime_ = 0.0;
}

////////////////////// INTERNAL CLASS /////////////////////////////////

/**
 * Gridlets MI share in Time Shared Mode
 */
private class MIShares {
    /**
     * maximum amount of MI share Gridlets can get
     */
    public double max;

    /**
     * minimum amount of MI share Gridlets can get when
     * it is executed on a PE that runs one extra Gridlet
     */
    public double min;

    /**
     * Total number of Gridlets that get Max share
     */
    public int maxCount;

    /**
     * Default constructor that initializes all attributes to 0
     *
     * @pre $none
     * @post $none
     */
    public MIShares() {
        max = 0.0;
        min = 0.0;
```

```
            maxCount = 0;
        }
    } // end of internal class

    ////////////////////// End of Internal Class ///////////////////////

    /**
     * Handles internal events that are coming to this entity.
     *
     * @pre $none
     * @post $none
     */
    public void body() {
        // a loop that is looking for internal events only
        Sim_event ev = new Sim_event();
        while (Sim_system.running()) {

            time1 = time2;
            time2 = GridSim.clock();

            if ((log) && (time2 - time1 > 200))
                try {
                    StatisticalAnalysis.getRandom_TimeShared_log().
                            writeChars(String.valueOf(calculateTotalLoad
                                    (gridletInExecList_.size())) + "\n");
                } catch (IOException e) {
                    e.printStackTrace();
                }
            super.sim_get_next(ev);

            // if the simulation finishes then exit the loop
            if (ev.get_tag() == GridSimTags.END_OF_SIMULATION ||
                    super.isEndSimulation() == true) {
                break;
            }

            // Internal Event if the event source is this entity
            if (ev.get_src() == super.myId_) {
                internalEvent();
            }
        }

        // CHECK for ANY INTERNAL EVENTS WAITING TO BE PROCESSED
        while (super.sim_waiting() > 0) {
            // wait for event and ignore since it is likely to be related to
            // internal event scheduled to update Gridlets processing
            super.sim_get_next(ev);
            System.out.println(super.resName_ +
                    ".TimeShared.body(): ignoring internal events");
        }
    }

    /**
     * Schedules a new Gridlet that has been received by the GridResource
```

```
 * entity.
 *
 * @param gl  a Gridlet object that is going to be executed
 * @param ack an acknowledgement, i.e. <tt>true</tt> if wanted to know
 *            whether this operation is success or not, <tt>false</tt>
 *            otherwise (don't care)
 * @pre gl != null
 * @post $none
 */
public void gridletSubmit(Gridlet gl, boolean ack) {
    StatisticalAnalysis.Communications++;
    // update Gridlets in execution up to this point in time
    updateGridletProcessing();

    // reset number of PE since at the moment, it is not supported
    if (gl.getNumPE() > 1) {
        String userName = GridSim.getEntityName(gl.getUserID());
        System.out.println();
        System.out.println(super.get_name() + ".gridletSubmit(): " +
                " Gridlet #" + gl.getGridletID() + " from " + userName +
                " user requires " + gl.getNumPE() + " PEs.");
        System.out.println("--> Process this Gridlet to 1 PE only.");
        System.out.println();

        // also adjusted the length because the number of PEs are reduced
        int numPE = gl.getNumPE();
        double len = gl.getGridletLength();
        gl.setGridletLength(len * numPE);
        gl.setNumPE(1);
    }

    // adds a Gridlet to the in execution list
    ResGridlet rgl = new ResGridlet(gl);
    rgl.setGridletStatus(Gridlet.INEXEC); // set the Gridlet status to exec
    gridletInExecList_.add(rgl);   // add into the execution list

    // sends back an ack if required
    if (ack == true) {
        super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, true,
                gl.getGridletID(), gl.getUserID()
        );
    }

    // forecast all Gridlets in the execution list
    forecastGridlet();
}

/**
 * Finds the status of a specified Gridlet ID.
 *
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @return the Gridlet status or <tt>-1</tt> if not found
 * @pre gridletId > 0
```

```
 * @pre userId > 0
 * @post $none
 * @see gridsim.Gridlet
 */
public int gridletStatus(int gridletId, int userId) {
    ResGridlet rgl = null;

    // Find in EXEC List first
    int found = super.findGridlet(gridletInExecList_, gridletId, userId);
    if (found >= 0) {
        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletInExecList_.get(found);
        return rgl.getGridletStatus();
    }

    // if not found then find again in Paused List
    found = super.findGridlet(gridletPausedList_, gridletId, userId);
    if (found >= 0) {
        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletPausedList_.get(found);
        return rgl.getGridletStatus();
    }

    // if not found in all lists
    return -1;
}

/**
 * Cancels a Gridlet running in this entity.
 * This method will search the execution and paused list. The User ID is
 * important as many users might have the same Gridlet ID in the lists.
 * <b>NOTE:</b>
 * <ul>
 * <li> Before canceling a Gridlet, this method updates all the
 * Gridlets in the execution list. If the Gridlet has no more MIs
 * to be executed, then it is considered to be <tt>finished</tt>.
 * Hence, the Gridlet can't be canceled.
 * <p/>
 * <li> Once a Gridlet has been canceled, it can't be resumed to
 * execute again since this method will pass the Gridlet back to
 * sender, i.e. the <tt>userId</tt>.
 * <p/>
 * <li> If a Gridlet can't be found in both execution and paused list,
 * then a <tt>null</tt> Gridlet will be send back to sender,
 * i.e. the <tt>userId</tt>.
 * </ul>
 *
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @pre gridletId > 0
 * @pre userId > 0
 * @post $none
 */
public void gridletCancel(int gridletId, int userId) {
```

```
    // Finds the gridlet in execution and paused list
    ResGridlet rgl = cancel(gridletId, userId);

    // If not found in both lists then report an error and sends back
    // an empty Gridlet
    if (rgl == null) {
        System.out.println(super.resName_ +
                ".TimeShared.gridletCancel(): Cannot find " +
                "Gridlet #" + gridletId + " for User #" + userId);

        super.sendCancelGridlet(GridSimTags.GRIDLET_CANCEL, null,
                gridletId, userId);
        return;
    }

    // if a Gridlet is found
    rgl.finalizeGridlet();     // finalise Gridlet

    // if a Gridlet has finished execution before canceling, the reports
    // an error msg
    if (rgl.getGridletStatus() == Gridlet.SUCCESS) {
        System.out.println(super.resName_
                + ".TimeShared.gridletCancel(): Cannot cancel"
                + " Gridlet #" + gridletId + " for User #" + userId
                + " since it has FINISHED.");
    }

    // sends the Gridlet back to sender
    super.sendCancelGridlet(GridSimTags.GRIDLET_CANCEL, rgl.getGridlet(),
            gridletId, userId);
}

/**
 * Pauses a Gridlet only if it is currently executing.
 * This method will search in the execution list. The User ID is
 * important as many users might have the same Gridlet ID in the lists.
 *
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @param ack       an acknowledgement, i.e. <tt>true</tt> if wanted to know
 *                  whether this operation is success or not, <tt>false</tt>
 *                  otherwise (don't care)
 * @pre gridletId > 0
 * @pre userId > 0
 * @post $none
 */
public void gridletPause(int gridletId, int userId, boolean ack) {
    boolean status = false;

    // find this Gridlet in the execution list
    int found = super.findGridlet(gridletInExecList_, gridletId, userId);
    if (found >= 0) {
        // update Gridlets in execution list up to this point in time
        updateGridletProcessing();
```

```
            // get a Gridlet from execution list
            ResGridlet rgl = (ResGridlet) gridletInExecList_.remove(found);

            // if a Gridlet is finished upon pausing, then set it to success
            // instead.
            if (rgl.getRemainingGridletLength() == 0.0) {
                System.out.println(super.resName_
                        + ".TimeShared.gridletPause(): Cannot pause"
                        + " Gridlet #" + gridletId + " for User #" + userId
                        + " since it is FINISHED.");

                gridletFinish(rgl, Gridlet.SUCCESS);
            } else {
                status = true;
                rgl.setGridletStatus(Gridlet.PAUSED);

                // add the Gridlet into the paused list
                gridletPausedList_.add(rgl);
                System.out.println(super.resName_ +
                        ".TimeShared.gridletPause(): Gridlet #" + gridletId +
                        " with User #" + userId + " has been sucessfully PAUSED.");
            }

            // forecast all Gridlets in the execution list
            forecastGridlet();
        } else   // if not found in the execution list
        {
            System.out.println(super.resName_ +
                    ".TimeShared.gridletPause(): Cannot find " +
                    "Gridlet #" + gridletId + " for User #" + userId);
        }

        // sends back an ack
        if (ack == true) {
            super.sendAck(GridSimTags.GRIDLET_PAUSE_ACK, status,
                    gridletId, userId);
        }
    }

    /**
     * Moves a Gridlet from this GridResource entity to a different one.
     * This method will search in both the execution and paused list.
     * The User ID is important as many Users might have the same Gridlet ID
     * in the lists.
     * <p/>
     * If a Gridlet has finished beforehand, then this method will send back
     * the Gridlet to sender, i.e. the <tt>userId</tt> and sets the
     * acknowledgment to false (if required).
     *
     * @param gridletId a Gridlet ID
     * @param userId    the user or owner's ID of this Gridlet
     * @param destId    a new destination GridResource ID for this Gridlet
     * @param ack       an acknowledgement, i.e. <tt>true</tt> if wanted to know
```

```
     *                  whether this operation is success or not, <tt>false</tt>
     *                  otherwise (don't care)
     * @pre gridletId > 0
     * @pre userId > 0
     * @pre destId > 0
     * @post $none
     */
    public void gridletMove(int gridletId, int userId, int destId, boolean ack) {
        // cancel the Gridlet first
        ResGridlet rgl = cancel(gridletId, userId);

        // If no found then print an error msg
        if (rgl == null) {
            System.out.println(super.resName_ +
                    ".TimeShared.gridletMove(): Cannot find " +
                    "Gridlet #" + gridletId + " for User #" + userId);

            if (ack == true)   // sends ack that this operation fails
            {
                super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, false,
                        gridletId, userId);
            }
            return;
        }

        // if found
        rgl.finalizeGridlet();   // finalise Gridlet
        Gridlet gl = rgl.getGridlet();

        // if a Gridlet has finished execution
        if (gl.getGridletStatus() == Gridlet.SUCCESS) {
            System.out.println(super.resName_
                    + ".TimeShared.gridletMove(): Cannot move"
                    + " Gridlet #" + gridletId + " for User #" + userId
                    + " since it has FINISHED.");

            if (ack == true) {
                super.sendAck(GridSimTags.GRIDLET_SUBMIT_ACK, false, gridletId,
                        userId);
            }

            super.sendFinishGridlet(gl);   // sends the Gridlet back to sender
        }
        // moves this Gridlet to another GridResource entity
        else {
            super.gridletMigrate(gl, destId, ack);
        }
    }

    /**
     * Resumes a Gridlet only in the paused list.
     * The User ID is important as many Users might have the same Gridlet ID
     * in the lists.
     *
```

```
 * @param gridletId a Gridlet ID
 * @param userId    the user or owner's ID of this Gridlet
 * @param ack       an acknowledgement, i.e. <tt>true</tt> if wanted to know
 *                  whether this operation is success or not, <tt>false</tt>
 *                  otherwise (don't care)
 * @pre gridletId > 0
 * @pre userId > 0
 * @post $none
 */
public void gridletResume(int gridletId, int userId, boolean ack) {
    boolean success = false;

    // finds in the execution list first
    int found = super.findGridlet(gridletPausedList_, gridletId, userId);
    if (found >= 0) {
        // need to update Gridlets in execution up to this point in time
        updateGridletProcessing();

        // remove a Gridlet from paused list and change the status
        ResGridlet rgl = (ResGridlet) gridletPausedList_.remove(found);
        rgl.setGridletStatus(Gridlet.RESUMED);

        // add the Gridlet back to in execution list
        gridletInExecList_.add(rgl);

        // then forecast Gridlets in execution list
        forecastGridlet();

        success = true;
        System.out.println(super.resName_ +
                ".TimeShared.gridletResume(): Gridlet #" + gridletId +
                " with User #" + userId + " has been sucessfully RESUMED.");
    } else  // if no found then prints an error msg
    {
        System.out.println(super.resName_ +
                ".TimeShared.gridletResume(): Cannot find Gridlet #" +
                gridletId + " for User #" + userId);
    }

    // sends back an ack to sender
    if (ack == true) {
        super.sendAck(GridSimTags.GRIDLET_RESUME_ACK, success,
                gridletId, userId);
    }
}

////////////////////// PRIVATE METHODS //////////////////////////////

protected double calculateTotalLoad(int size) {
    int totalRating = 0;
    PEList peList = (resource_.getMachineList().getMachine(0)).getPEList();
    for (int i = 0; i < peList.size(); i++) {
        totalRating += ((PE) peList.get(i)).getMIPSRating();
    }
```

```
        totalRating = totalRating / 10;
        // Devide by the lowest PE rate in the Grid.
        // Here we have 10 and 50 so we divide by 10
        double val = (size + 1.0) / totalRating;
        int numGridletPerPE = (int) Math.ceil(val);

        // load is between [0.0, 1.0] where 1.0 is busy and 0.0 is not busy
        double localLoad = resCalendar_.getCurrentLoad();
        double load = 1.0 - ((1 - localLoad) / numGridletPerPE);
        if (load < 0.0) {
            load = 0.0;
        }

        return load;
    }

    /**
     * Updates the execution of all Gridlets for a period of time.
     * The time period is determined from the last update time up to the
     * current time. Once this operation is successfull, then the last update
     * time refers to the current time.
     *
     * @pre $none
     * @post $none
     */
    private void updateGridletProcessing() {
        // Identify MI share for the duration (from last event time)
        double time = GridSim.clock();
        double timeSpan = time - lastUpdateTime_;

        // if current time is the same or less than the last update time,
        // then ignore
        if (timeSpan <= 0.0) {
            return;
        }

        // Update Current Time as the Last Update
        lastUpdateTime_ = time;

        // update the GridResource load
        int size = gridletInExecList_.size();
        double load = super.calculateTotalLoad(size);
        super.addTotalLoad(load);       // add the current resource load

        // if no Gridlets in execution then ignore the rest
        if (size == 0) {
            return;
        }

        // gets MI Share for all Gridlets
        MIShares shares = getMIShare(timeSpan, size);
        ResGridlet obj = null;
```

```
        // a loop that allocates MI share for each Gridlet accordingly
        // In this algorithm, Gridlets at the front of the list
        // (range = 0 until MIShares.maxCount-1) will be given max MI value
        // For example, 2 PEs and 3 Gridlets. PE #0 processes Gridlet #0
        // PE #1 processes Gridlet #1 and Gridlet #2
        int i = 0;  // a counter
        Iterator iter = gridletInExecList_.iterator();
        while (iter.hasNext()) {
            obj = (ResGridlet) iter.next();

            // Updates the Gridlet length that is currently being executed
            if (i < shares.maxCount) {
                obj.updateGridletFinishedSoFar(shares.max);
            } else {
                obj.updateGridletFinishedSoFar(shares.min);
            }

            i++;   // increments i
        }
    }

    /**
     * Identifies MI share (max and min) for all Gridlets in
     * a given time duration
     *
     * @param timeSpan duration
     * @param size     total number of Gridlets in the execution list
     * @return the total MI share that a Gridlet gets for a given
     *         <tt>timeSpan</tt>
     */
    private MIShares getMIShare(double timeSpan, int size) {
        // 1 - localLoad_ = available MI share percentage
        double localLoad = super.resCalendar_.getCurrentLoad();
        double TotalMIperPE = super.resource_.getMIPSRatingOfOnePE() * timeSpan
                * (1 - localLoad);

        // This TimeShared is not Round Robin where each PE for 1 Gridlet only.
        // a PE can have more than one Gridlet executing.
        // minimum number of Gridlets that each PE runs.
        int glDIVpe = size / super.totalPE_;

        // number of PEs that run one extra Gridlet
        int glMODpe = size % super.totalPE_;

        // If num Gridlets in execution > total PEs in a GridResource,
        // then divide MIShare by the following constraint:
        // - obj.max = MIShare of a PE executing n Gridlets
        // - obj.min = MIShare of a PE executing n+1 Gridlets
        // - obj.maxCount = a threshold number of Gridlets will be assigned to
        //                  max MI value.
        //
        // In this algorithm, Gridlets at the front of the list
        // (range = 0 until maxCount-1) will be given max MI value
        if (glDIVpe > 0) {
```

```
            // this is for PEs that run one extra Gridlet
            share_.min = TotalMIperPE / (glDIVpe + 1);
            share_.max = TotalMIperPE / glDIVpe;
            share_.maxCount = (super.totalPE_ - glMODpe) * glDIVpe;
        }

        // num Gridlet in Exec < total PEs, meaning it is a
        // full PE share: i.e a PE is dedicated to execute a single Gridlet
        else {
            share_.max = TotalMIperPE;
            share_.min = TotalMIperPE;
            share_.maxCount = size;   // number of Gridlet
        }

        return share_;
    }

    /**
     * Determines the smallest completion time of all Gridlets in the execution
     * list. The smallest time is used as an internal event to
     * update Gridlets processing in the future.
     * <p/>
     * The algorithm for this method:
     * <ul>
     * <li> identify the finish time for each Gridlet in the execution list
     * given the share MIPS rating for all and the remaining Gridlet's
     * length
     * <li> find the smallest finish time in the list
     * <li> send the last Gridlet in the list with
     * <tt>delay =  smallest finish time - current time</tt>
     * </ul>
     *
     * @pre $none
     * @post $none
     */
    private void forecastGridlet() {
        // if no Gridlets available in exec list, then exit this method
        if (gridletInExecList_.size() == 0) {
            return;
        }

        // checks whether Gridlets have finished or not. If yes, then remove
        // them since they will effect the MIShare calculation.
        checkGridletCompletion();

        // Identify MIPS share for all Gridlets for 1 second, considering
        // current Gridlets + No of PEs.
        MIShares share = getMIShare(1.0, gridletInExecList_.size());

        ResGridlet rgl = null;
        int i = 0;
        double time = 0.0;
        double rating = 0.0;
        double smallestTime = 0.0;
```

```
        // For each Gridlet, determines their finish time
        Iterator iter = gridletInExecList_.iterator();
        while (iter.hasNext()) {
            rgl = (ResGridlet) iter.next();

            // If a Gridlet locates before the max count then it will be given
            // the max. MIPS rating
            if (i < share.maxCount) {
                rating = share.max;
            } else {   // otherwise, it will be given the min. MIPS Rating
                rating = share.min;
            }

            time = forecastFinishTime(rating, rgl.getRemainingGridletLength());

            int roundUpTime = (int) (time + 1);   // rounding up
            rgl.setFinishTime(roundUpTime);

            // get the smallest time of all Gridlets
            if (i == 0 || smallestTime > time) {
                smallestTime = time;
            }

            i++;
        }

        // sends to itself as an internal event
        super.sendInternalEvent(smallestTime);
    }

    /**
     * Checks all Gridlets in the execution list whether they are finished or
     * not.
     *
     * @pre $none
     * @post $none
     */
    private void checkGridletCompletion() {
        ResGridlet rgl = null;

        // a loop that determine the smallest finish time of a Gridlet
        // Don't use an iterator since it causes an exception because if
        // a Gridlet is finished, gridletFinish() will remove it from the list.
        int i = 0;
        while (i < gridletInExecList_.size()) {
            rgl = (ResGridlet) gridletInExecList_.get(i);

            // if a Gridlet has finished, then remove it from the list
            if (rgl.getRemainingGridletLength() <= 0.0) {
                gridletFinish(rgl, Gridlet.SUCCESS);
                continue;  // not increment i coz the list size also decreases
            }
```

```
            i++;
        }
    }

    /**
     * Forecast finish time of a Gridlet.
     * <tt>Finish time = length / available rating</tt>
     *
     * @param availableRating the shared MIPS rating for all Gridlets
     * @param length          remaining Gridlet length
     * @return Gridlet's finish time.
     */
    private double forecastFinishTime(double availableRating, double length) {
        double finishTime = length / availableRating;

        // This is as a safeguard since the finish time can be extremely
        // small close to 0.0, such as 4.5474735088646414E-14. Hence causing
        // some Gridlets never to be finished and consequently hang the program
        if (finishTime < 1.0) {
            finishTime = 1.0;
        }

        return finishTime;
    }

    /**
     * Updates the Gridlet's properties, such as status once a
     * Gridlet is considered finished.
     *
     * @param rgl    a ResGridlet object
     * @param status the status of this ResGridlet object
     * @pre rgl != null
     * @post $none
     */
    private void gridletFinish(ResGridlet rgl, int status) {
        // NOTE: the order is important! Set the status first then finalize
        // due to timing issues in ResGridlet class.
        rgl.setGridletStatus(status);
        rgl.finalizeGridlet();

        // sends back the Gridlet with no delay
        Gridlet gl = rgl.getGridlet();
        super.sendFinishGridlet(gl);

        // remove this Gridlet in the execution
        gridletInExecList_.remove(rgl);
    }

    /**
     * Handles internal event
     *
     * @pre $none
     * @post $none
     */
```

```
private void internalEvent() {
    // this is a constraint that prevents an infinite loop
    // Compare between 2 floating point numbers. This might be incorrect
    // for some hardware platform.
    if (lastUpdateTime_ == GridSim.clock()) {
        return;
    }

    // update Gridlets in execution up to this point in time
    updateGridletProcessing();

    // schedule next event
    forecastGridlet();
}


private ResGridlet cancel(int gridletId, int userId) {
    ResGridlet rgl = null;

    // Check whether the Gridlet is in execution list or not
    int found = super.findGridlet(gridletInExecList_, gridletId, userId);

    // if a Gridlet is in execution list
    if (found >= 0) {
        // update the gridlets in execution list up to this point in time
        updateGridletProcessing();

        // Get the Gridlet from the execution list
        rgl = (ResGridlet) gridletInExecList_.remove(found);

        // if a Gridlet is finished upon cancelling, then set it to success
        if (rgl.getRemainingGridletLength() == 0.0) {
            rgl.setGridletStatus(Gridlet.SUCCESS);
        } else {
            rgl.setGridletStatus(Gridlet.CANCELED);
        }

        // then forecast the next Gridlet to complete
        forecastGridlet();
    }

    // if a Gridlet is not in exec list, then find it in the paused list
    else {
        found = super.findGridlet(gridletPausedList_, gridletId, userId);

        // if a Gridlet is found in the paused list then remove it
        if (found >= 0) {
            rgl = (ResGridlet) gridletPausedList_.remove(found);
            rgl.setGridletStatus(Gridlet.CANCELED);
        }
    }

    return rgl;
}
```

```
    /* protected double calculateTotalLoad(int size) {
        return size / (totalPE_  - (totalPE_ * resCalendar_.getCurrentLoad()));
    }*/
} // end class

package grid;

import java.util.*;
import java.io.FileWriter;
import java.sql.Time;

import eduni.simjava.Sim_system;
import eduni.simjava.Sim_event;

import gridsim.*;
import gridsim.net.Link;
import gridsim.index.AbstractGIS;
import allocpolicy.*;
import allocpolicy.statebroadcast.SBA;
import allocpolicy.antZ.AntColonyAllocPolicy;
import allocpolicy.particleZ.ParticleSwarmAllocPolicy;
import allocpolicy.particleZ.AllocPolicyList;


public class SimulationWithoutFailure extends GridSim {


    private static int num_resource;
    private static int num_gridlet;
    private static int GRIDLENGTH_COEF;           //500 * 100
    public static int MAX_NUMBER_PE = 4;
    private boolean trace_flag = false;
    private static String loadbalancing; // SBA |  AntColony  |
    // TimeShared | SpaceShared | ParticleSwarm

    private Integer ID;
    private double pollingTime_;
    private GridletList GridletReceiveList_;
    private ArrayList GridletSubmittedList_;   // list of submitted Gridlets
    private static final int BASE = 440000;
    public static final int SUBMIT_GRIDLET = BASE + 1;
    // we keep here the time when each gridlet is submitted
    private double gridletSubmissionTime[];
    private double gridletLatencyTime[];

    private double startTime = Double.MAX_VALUE;
    private double finishTime = 0;
    Topology topology;
    private static long Simulation_Time = 0;


    SimulationWithoutFailure(String name, double baud_rate, double pollTime)
            throws Exception {
```

```
    super(name, baud_rate);

    this.GridletSubmittedList_ = new ArrayList();
    this.GridletReceiveList_ = new GridletList();
    gridletSubmissionTime = new double[num_gridlet];
    gridletLatencyTime = new double[num_gridlet];
    pollingTime_ = pollTime;
    ID = getEntityId(name);
}

public void body() {

    initializeResultsFile();
    createGridlet(super.get_id(), num_gridlet);

    LinkedList<Integer> resList;
    while (true) {
        super.gridSimHold(1.0);
        resList = super.getGridResourceList();
        if (resList.size() == num_resource)
            break;
        //else
        // System.out.println("Waiting to get list of resources ...");
    }


    try {
        topology = Topology.getInstance(num_resource, getResList());
    } catch (Exception e) {
        e.printStackTrace();
    }

    int PAUSE = 10 * 60;  // 10 mins
    Random random = new Random();
    int init_time = PAUSE + random.nextInt(5 * 60);

    // sends a reminder to itself
    super.send(super.get_id(), init_time, SUBMIT_GRIDLET);
    System.out.println(super.get_name() +
            ": initial SUBMIT_GRIDLET event will be at clock: " +
            init_time + ". Current clock: " + GridSim.clock());

    /*    if (loadbalancing.equals("SBA"))
   for (int i = 0; i < resList.size(); i++){
      send(resList.get(i), GridSimTags.SCHEDULE_NOW,
      GridSimTags.BROADCAST_STATE);
  }  */

    ////////////////////////////////////////////////////////////
    // Now, we have the framework of the entity:
    Simulation_Time = System.currentTimeMillis();
    while (Sim_system.running()) {
        Sim_event ev = new Sim_event();
        super.sim_get_next(ev); // get the next event in the queue
```

```
        //System.out.println("STILL RUNNING" + GridSim.clock());

        switch (ev.get_tag()) {
            // submit a gridlet
            case SUBMIT_GRIDLET: {
                if (GridSim.clock() < startTime) {
                    startTime = GridSim.clock();
                }

                processGridletSubmission(ev); // process the received event
                break;
            }

            // Receive a gridlet back
            case GridSimTags.GRIDLET_RETURN: {
                if (GridSim.clock() > finishTime) {
                    finishTime = GridSim.clock();
                }
                processGridletReturn(ev);

                break;
            }

            case GridSimTags.END_OF_SIMULATION:
                System.out.println("\n============== " + super.get_name() +
                        ". Ending simulation...");
                break;

            default:
                if (trace_flag)
                    System.out.println(super.get_name() + ": " +
                            "Received an event: " + ev.get_tag());
                break;

        } // switch

    } //  while

    // wait for few seconds before printing the output
    super.sim_pause(super.get_id() * 2);
    super.terminateIOEntities();
    Simulation_Time = System.currentTimeMillis() - Simulation_Time;
    if (trace_flag)
        printGridletList(GridletReceiveList_, super.get_name(), false,
                gridletLatencyTime);
    printGridletList(GridletReceiveList_);
    printUsefulInfo();

}

// TODO: What the gridlets should be like ?!

private Gridlet createGridlet(int id) {
    long seed = 11L * 13 * 17 * 19 * 23 + 1;
```

```
        // sets the PE MIPS Rating
        GridSimStandardPE.setRating(100);
        double length;
        Random random;
        long file_size, output_size;

        random = new Random(seed);
        length = GridSimStandardPE.toMIs(random.nextDouble() * GRIDLENGTH_COEF);

        // determines the Gridlet file size that varies within the
        // range
        // 100 + (10% to 40%)
        file_size = (long) GridSimRandom.real(100, 0.10, 0.40, random
                .nextDouble());

        // determines the Gridlet output size that varies within the
        // range
        // 250 + (10% to 50%)
        output_size = (long) GridSimRandom.real(250, 0.10, 0.50, random
                .nextDouble());

        // creates a new Gridlet object
        // System.out.println("A Gridlet is created : --- " + id);
        Gridlet gridlet = new Gridlet(id, length, file_size, output_size);

        gridlet.setUserID(ID);
        return gridlet;

    }

    private void createGridlet(int userID, int numGridlet) {
        for (int i = 0; i < numGridlet; i++) {
            // Creates a Gridlet
            Gridlet gl = createGridlet(i);
            gl.setUserID(userID);

            // Originally, gridlets are created to be submitted
            // as soon as possible (the 0.0 param)
            GridletSubmission gst = new GridletSubmission(gl, false);

            // add this gridlet into a list
            this.GridletSubmittedList_.add(gst);
        }
    }

    private void printUsefulInfo() {

        System.out.println("Comunications : " + StatisticalAnalysis.Communications);
        System.out.println("LINK NO: " + Topology.LINKNO);
    }


    private void initializeResultsFile() {
        if (!trace_flag) {
```

```
            return;
        }

        // Initialize the results file
        FileWriter fwriter = null;
        try {
            fwriter = new FileWriter(super.get_name(), true);
        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println(
                    "Unwanted errors while opening file " + super.get_name() +
                            " or " + super.get_name() + "_Fin");
        }

        try {
            fwriter.write("Event \t GridletID \t Resource \t GridletStatus " +
                    "\t\t Clock\n");
        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println(
                    "Unwanted errors while writing on file " + super.get_name() +
                            " or " + super.get_name() + "_Fin");
        }

        try {
            fwriter.close();
        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println(
                    "Unwanted errors while closing file " + super.get_name() +
                            " or " + super.get_name() + "_Fin");
        }
    }

    private void processGridletSubmission(Sim_event ev) {
        if (trace_flag) {
            System.out.println(super.get_name() +
                    ": received an SUBMIT_GRIDLET event. Clock: " +
                    GridSim.clock());
        }

        /***********
         We have to submit:
         - the gridlet whose id comes with the event
         - all the gridlets with the "gridletSub.getSubmitted() == false"

         So, set the gridletSub.getSubmitted() to false for the gridlet whose
         id comes with the event
         ***********/

        int i = 0;
        GridletSubmission gridletSub;
        int resourceID[];
        Random random = new Random(5);   // a random generator with a random seed
```

```
int index;
Gridlet gl;
Integer obj;
int glID;

// This is because the initial GRIDLET_SUBMIT event, at the beginning
// of sims, does not have any gridlet id. We have to submit
// all the gridlets.
if (ev.get_data() instanceof Integer) {
    obj = (Integer) ev.get_data();
    glID = obj; // get the gridlet id.
} else {
    glID = 99; // a value at random, not used at all in this case
}

while (i < GridletSubmittedList_.size()) {
    gridletSub = (GridletSubmission) GridletSubmittedList_.get(i);

    if ((gridletSub.getGridlet()).getGridletID() == glID) {
        // set this gridlet whose id comes with the event as not submitted,
        // so that it is submitted as soon as possible.
        ((GridletSubmission) GridletSubmittedList_.get(i)).
                setSubmitted(false);
    }

    // Submit the gridlets with the "gridletSub.getSubmitted() == false"
    if ((gridletSub.getSubmitted() == false)) {
        // we have to resubmit this gridlet
        gl = ((GridletSubmission) GridletSubmittedList_.get(i)).
                getGridlet();
        resourceID = getResList(); // Get list of resources from GIS

        // If we have resources in the list
        if ((resourceID != null) && (resourceID.length != 0)) {
            index = random.nextInt(resourceID.length);

            // make sure the gridlet will be executed from the begining
            resetGridlet(gl);

            // submits this gridlet to a resource
            super.gridletSubmit(gl, resourceID[index]);

            //Submit to one resource
            //super.gridletSubmit(gl, resourceID[0]);
            gridletSubmissionTime[gl.getGridletID()] = GridSim.clock();

            // set this gridlet as submitted
            ((GridletSubmission) GridletSubmittedList_.
                    get(i)).setSubmitted(true);

            if (trace_flag) {
                System.out.println(super.get_name() +
                        ": Sending Gridlet #" + i + " to " +
                        GridSim.getEntityName(resourceID[index]) +
```

```
                        " at clock: " + GridSim.clock());

                    // Write into a results file
                    write(super.get_name(), "Sending", gl.getGridletID(),
                            GridSim.getEntityName(resourceID[index]),
                            gl.getGridletStatusString(), GridSim.clock());
                }

            }
            // No resources available at this moment, so schedule an event
            // in the future. The event wil be in 15 min (900 sec), as
            // resource failures may last several hours.
            // This event includes the gridletID, so that the user will
            // try to submit only this gridlet
            else {
                super.send(super.get_id(), GridSimTags.SCHEDULE_NOW + 900,
                        SUBMIT_GRIDLET, new Integer(gl.getGridletID()));
            }

        }// if (gridletSub.getSubmitted() == false)

        i++;
    } // while (i < GridletSubmittedList_.size())

} // processGridletSubmission

private void processGridletReturn(Sim_event ev) {
    if (trace_flag) {
        System.out.println(super.get_name() +
              ": received an GRIDLET_RETURN event. Clock: " + GridSim.clock());
    }

    Object obj = ev.get_data();
    Gridlet gl;
    Random random = new Random(5);   // a random generator with a random seed

    if (obj instanceof Gridlet) {
        gl = (Gridlet) obj;
        gridletLatencyTime[gl.getGridletID()] = GridSim.clock();

        // Write into a results file
        if (trace_flag) {
            write(super.get_name(), "Receiving", gl.getGridletID(),
                    GridSim.getEntityName(gl.getResourceID()),
                    gl.getGridletStatusString(), GridSim.clock());
        }

        //////////////////////// Gridlet Success
        if (gl.getGridletStatusString().compareTo("Success") == 0) {
            if (trace_flag)
                System.out.println(super.get_name() + ": Receiving Gridlet #" +
                        gl.getGridletID() + " with status Success at time = " +
                        GridSim.clock() + " from resource " +
                        GridSim.getEntityName(gl.getResourceID()));
```

```
        this.GridletReceiveList_.add(gl); // add into the received list
        gridletLatencyTime[gl.getGridletID()] =
                gridletLatencyTime[gl.getGridletID()] -
                        gridletSubmissionTime[gl.getGridletID()];

        // We have received all the gridlets. So, finish the simulation.
        if (GridletReceiveList_.size() == GridletSubmittedList_.size()) {
            super.shutdownUserEntity();
            super.terminateIOEntities();
        }

    } // if (gl.getGridletStatusString() == "Success")

    //////////////////////// Gridlet Failed
    else if (gl.getGridletStatusString().compareTo("Failed") == 0) {
        if (trace_flag)
            System.out.println(super.get_name() + ": Receiving Gridlet #" +
                    gl.getGridletID() + " with status Failed at time = " +
                    GridSim.clock() + " from resource " +
                    GridSim.getEntityName(gl.getResourceID()));

        // Send the gridlet as soon as we have resources available.
        // This gridlet will be resend as soon as possible,
        // in the first loop.
        int pos = findGridletInGridletSubmittedList(gl);
        if (pos == -1) {
            System.out.println(super.get_name() +
                    ". Gridlet not found in GridletSubmittedList.");
        } else {
            // set this gridlet as submitted, because otherwise
            // this gridlet may be submitted several times.
            // A gridlet will only be submitted when the event carrying
            // its id reaches the user
            ((GridletSubmission) GridletSubmittedList_.get(pos)).
                    setSubmitted(true);

            // Now, schedule an event to itself to submit the gridlet
            // The gridlet will be sent as soon as possible
            Integer glID_Int = new Integer(gl.getGridletID());

            // This event includes the gridletID, so that the user
            // will try to submit only this gridlet
            super.send(super.get_id(), GridSimTags.SCHEDULE_NOW,
                    SUBMIT_GRIDLET, glID_Int);
        }
    } // if (gl.getGridletStatusString() == "Failed")

    ////////////////////////////// Gridlet Failed_resource
    else if (gl.getGridletStatusString().compareTo(
            "Failed_resource_unavailable") == 0) {
        int pos = findGridletInGridletSubmittedList(gl);
        if (pos == -1) {
            System.out.println(super.get_name() +
```

```
                        ". Gridlet not found in GridletSubmittedList.");
            } else {
                // Now, set its submission time for a random time between
                // 1 and the polling time
                double resubmissionTime = random.nextDouble() * pollingTime_;

                // this is to prevent the gridlet from being submitted
                // before its resubmission time.
                // This is different from the FAILED case, because
                // in that case, the gridlet should be resubmited as soon
                // as possible. As oppossed to that, this gridlet should
                // not be resubmited before its resubmission time.
                ((GridletSubmission) GridletSubmittedList_.get(pos)).
                        setSubmitted(true);


                System.out.println(super.get_name() + ": Receiving Gridlet #" +
                        gl.getGridletID() +
                        " with status Failed_resource_unavailable at time = " +
                        GridSim.clock() + " from resource " +
                        GridSim.getEntityName(gl.getResourceID()) +
                        "(resID: " + gl.getResourceID() +
                        "). Resubmission time will be: " +
                        resubmissionTime + GridSim.clock());

                // Now, we have to inform the GIS  about this failure, so it
                // can keep the list of resources up-to-date.
                informGIS(gl.getResourceID());

                // Now, schedule an event to itself to submit the gridlet
                Integer glID_Int = new Integer(gl.getGridletID());

                // This event includes the gridletID, so that the user
                // will try to submit only this gridlet
                super.send(super.get_id(), resubmissionTime, SUBMIT_GRIDLET,
                        glID_Int);
            }
        } // else if
        else {
            System.out.println(super.get_name() + ": Receiving Gridlet #" +
                    gl.getGridletID() + " with status " +
                    gl.getGridletStatusString() + " at time = " +
                    GridSim.clock() + " from resource " +
                    GridSim.getEntityName(gl.getResourceID()) +
                    " resID: " + gl.getResourceID());
        }

    } // if (obj instanceof Gridlet)
}

private void informGIS(int resID) {
    Integer resID_Int = new Integer(resID);

    super.send(super.output, 0.0, AbstractGIS.NOTIFY_GIS_RESOURCE_FAILURE,
```

```
            new IO_data(resID_Int, Link.DEFAULT_MTU, this.ID));
}

private int findGridletInGridletSubmittedList(Gridlet gl) {
    Gridlet g;
    GridletSubmission gst;
    for (int i = 0; i < GridletSubmittedList_.size(); i++) {
        gst = (GridletSubmission) GridletSubmittedList_.get(i);
        g = gst.getGridlet();

        if (g.getGridletID() == gl.getGridletID())
            return i;
    }

    return -1;
}

private void write(String user, String event, int glID, String resName,
                   String status, double clock) {
    if (trace_flag == false) {
        return;
    }

    // Write into a results file
    FileWriter fwriter = null;
    try {
        fwriter = new FileWriter(super.get_name(), true);
    } catch (Exception ex) {
        ex.printStackTrace();
        System.out.println(
                "Unwanted errors while opening file " + super.get_name());
    }

    try {
        fwriter.write(event + "\t\t" + glID + "\t" + resName + "\t" + status +
                "\t\t" + clock + "\n");
    } catch (Exception ex) {
        ex.printStackTrace();
        System.out.println(
                "Unwanted errors while writing on file " + super.get_name());
    }


    try {
        fwriter.close();
    } catch (Exception ex) {
        ex.printStackTrace();
        System.out.println(
                "Unwanted errors while closing file " + super.get_name());
    }

}
```

```
private int[] getResList() {

    LinkedList resList = super.getGridResourceList();
    int[] resourceID = null;

    // if we have any resource
    if ((resList != null) && (resList.size() != 0)) {
        resourceID = new int[resList.size()];
        for (int x = 0; x < resList.size(); x++) {
            // Resource list contains list of resource IDs
            resourceID[x] = ((Integer) resList.get(x)).intValue();
            if (trace_flag == true) {
                System.out.println(super.get_name() +
                        ": resource[" + x + "] = " + resourceID[x]);
            }
        }
    }
    return resourceID;
}

private void resetGridlet(Gridlet gl) {
    gl.setGridletLength(gl.getGridletLength());
    gl.setGridletFinishedSoFar(0);

}

private void printGridletList(GridletList list, String name,
                              boolean detail, double gridletLatencyTime[]) {
    int size = list.size();
    Gridlet gridlet = null;

    String indent = "    ";
    StringBuffer buffer = new StringBuffer(1000);
    buffer.append("\n\n============== OUTPUT for " + name + " ===========");
    buffer.append("\nGridlet ID" + indent + "STATUS" + indent +
            "Resource ID" + indent + indent + "Cost" + indent +
            indent + "CPU Time" + indent + indent + "Latency");

    // a loop to print the overall result
    int i = 0;
    boolean header = true;

    for (i = 0; i < size; i++) {
        gridlet = (Gridlet) list.get(i);

        buffer.append("\n");
        buffer.append(indent + gridlet.getGridletID() + indent + indent);
        buffer.append(gridlet.getGridletStatusString());
        buffer.append(indent + indent + gridlet.getResourceID() +
                indent + gridlet.getProcessingCost() +
                indent + gridlet.getActualCPUTime() +
                indent + gridletLatencyTime[gridlet.getGridletID()]);

        if (i != 0) {
```

```
                header = false;
            }

            writeFin(name, gridlet.getGridletID(),
                    GridSim.getEntityName(gridlet.getResourceID()),
                    gridlet.getProcessingCost(), gridlet.getActualCPUTime(),
                    GridSim.clock(), header);
        }

        if (detail == true) {
            // a loop to print each Gridlet's history
            for (i = 0; i < size; i++) {
                gridlet = (Gridlet) list.get(i);

                buffer.append(gridlet.getGridletHistory());
                buffer.append("Gridlet #" + gridlet.getGridletID());
                buffer.append(", length = " + gridlet.getGridletLength()
                        + ", finished so far = " +
                        gridlet.getGridletFinishedSoFar());
                buffer.append("===========================================");
            }
        }

        buffer.append("\n====================================================");
        System.out.println(buffer.toString());

    }

    private void writeFin(String user, int glID, String resName,
                          double cost, double cpu, double clock,
                          boolean header) {
        if (trace_flag == false) {
            return;
        }

        // Write into a results file
        FileWriter fwriter = null;
        try {
            fwriter = new FileWriter(user, true);
        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println(
                    "Unwanted errors while opening file " + user);
        }

        try {
            if (header == true) {
                fwriter.write(
                        "\n\nGridletID \t Resource \t Cost \t CPU time " +
                                "\t Latency\n");
            }

            fwriter.write(glID + "\t" + resName + "\t" + cost + "\t" + cpu +
                    "\t" + +clock + "\n");
```

```
    } catch (Exception ex) {
        ex.printStackTrace();
        System.out.println(
                "Unwanted errors while writing on file " + user);
    }

    try {
        fwriter.close();
    } catch (Exception ex) {
        ex.printStackTrace();
        System.out.println(
                "Unwanted errors while closing file " + user);
    }

}

// ///////////////////////// STATIC METHODS //////////////////////

private static void initialize() {

    int num_user = 1; // number of grid users
    Calendar calendar = Calendar.getInstance();
    boolean trace_flag = false; // mean don't trace GridSim events

    // Initialize the GridSim package
    System.out.println("Initializing GridSim package");
    GridSim.init(num_user, calendar, trace_flag);
}

private static void createResource(String name, int totalMachine, int totalPE,
                                   double[] baudRate, int[] peRating,
                                   double[] price) {
    double bandwidth;
    double cost;
    Random random = new Random();

    if (random.nextBoolean() == true) {
        bandwidth = baudRate[0];
        cost = price[0];
    } else {
        bandwidth = baudRate[1];
        cost = price[1];
    }

    // creates a GridResource
    createGridResource(name, totalMachine, totalPE, bandwidth, peRating,
            cost);
}

private static void createGridResource(String name, int totalMachine,
                                       int totalPE, double bandwidth,
                                       int[] peRating, double cost) {
    // Here are the steps needed to create a Grid resource:
    // 1. We need to create an object of MachineList to store one or more
```

```
// Machines
Random random = new Random();
MachineList mList = new MachineList();

int rating;
for (int i = 0; i < totalMachine; i++) {
    // 2. A Machine contains one or more PEs or CPUs. Therefore, should
    // create an object of PEList to store these PEs before creating
    // a Machine.
    PEList peList = new PEList();

    // even Machines have different PE rating compare to odd ones
    if (random.nextBoolean() == true) {
        rating = peRating[0];
    } else {
        rating = peRating[1];
    }

    // 3. Create PEs and add these into an object of PEList.
    for (int k = 0; k < totalPE; k++) {
        // need to store PE id and MIPS Rating
        peList.add(new PE(k, rating));
    }

    // 4. Create one Machine with its id and list of PEs or CPUs
    mList.add(new Machine(i, peList));
}

// 5. Create a ResourceCharacteristics object that stores the
// properties of a Grid resource: architecture, OS, list of
// Machines, allocation policy: time- or space-shared, time zone
// and its price (G$/PE time unit).
String arch = "Sun Ultra"; // system architecture
String os = "Solaris"; // operating system
double time_zone = 0.0; // time zone this resource located

ResourceCharacteristics resConfig = new ResourceCharacteristics(arch,
        os, mList, ResourceCharacteristics.OTHER_POLICY_DIFFERENT_RATING,
        time_zone, cost);

// 6. Finally, we need to create a GridResource object.
long seed = 11L * 13 * 17 * 19 * 23 + 1;
double peakLoad = 0.0; // the resource load during peak hour
double offPeakLoad = 0.0; // the resource load during off-peak hr
double holidayLoad = 0.0; // the resource load during holiday

// incorporates weekends so the grid resource is on 7 days a week
LinkedList<Integer> Weekends = new LinkedList<Integer>();
Weekends.add(new Integer(Calendar.SATURDAY));
Weekends.add(new Integer(Calendar.SUNDAY));

// incorporates holidays. However, no holidays are set in this example
LinkedList Holidays = new LinkedList();
```

```
        ResourceCalendar resourceCalendar = new ResourceCalendar(time_zone,
                peakLoad, offPeakLoad, holidayLoad, Weekends, Holidays, seed);

        try {
            AllocPolicy allocPolicy = getAllocPolicy(name);
            if (allocPolicy == null)
                throw new Exception("Define Alloc Plicy");
            new GridResource(name, bandwidth, resConfig, resourceCalendar,
                    allocPolicy);
            if (loadbalancing.equals("ParticleSwarm")) {
                AllocPolicyList.getInstance().addallocPolicy(
                        (ParticleSwarmAllocPolicy) allocPolicy);
            }
        } catch (Exception e) {
            e.printStackTrace(); // To change body of catch statement use
            // File | Settings | File Templates.
        }
        // m System.out.println("Creates one Grid resource with name = " +
        // name);
    }

    private static AllocPolicy getAllocPolicy(String name) throws Exception {
        AllocPolicy allocPolicy = null;

        if (loadbalancing.equals("AntColony")) {
            allocPolicy = new AntColonyAllocPolicy(name, name + "AllocPolicy",
                    output);
        } else if (loadbalancing.equals("TimeShared")) {
            allocPolicy = new TimeShared(name, name + "AllocPolicy");
        } else if (loadbalancing.equals("SpaceShared")) {
            allocPolicy = new SpaceShared(name, name + "AllocPolicy");
        } else if (loadbalancing.equals("ParticleSwarm")) {
            allocPolicy = new ParticleSwarmAllocPolicy(name, name + "AllocPolicy");
        } else if (loadbalancing.equals("SBA")) {
            allocPolicy = new SBA(name, name + "AllocPolicy", output);
        }
        return allocPolicy;
    }

    public static void main(String[] args) {
        Random random = new Random();

        loadbalancing = args[0];
        num_resource = Integer.valueOf(args[1]);
        num_gridlet = Integer.valueOf(args[2]);
        GRIDLENGTH_COEF = Integer.valueOf(args[3]);


        try {
            initialize();

            // Second step: Creates one or more GridResource objects
            double pollTime = 100;   // time between polls
            double baudRate[] = {1000, 5000}; // bandwidth for even, odd
```

```
        int peRating[] = {10, 50}; // PE Rating for even, odd
        double price[] = {3.0, 5.0}; // resource for even, odd

        for (int i = 0; i < num_resource; i++) {
            createResource("GridResource_" + i, 1,
                    random.nextInt(MAX_NUMBER_PE) + 1,
                    baudRate, peRating, price);
        }

        // Third step: Creates the grid.MyGridSimulator object
        SimulationWithoutFailure obj = new SimulationWithoutFailure(
                "grid.MyGridSimulator", 560.00, pollTime);

        // Fourth step: Starts the simulation
        GridSim.startGridSimulation();

        // Final step: Prints the Gridlets when simulation is over
        //GridletList newList = obj.getGridletList();
        // printGridletList(newList);
        // printUsefulInfo(newList);

    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("Unwanted errors happen");
    }
}

private static void printGridletList(GridletList list) {
    double time1 = Double.MAX_VALUE;
    double time2 = Double.MIN_VALUE;
    double averageTurnAroundTime = 0;
    int size = list.size();
    Gridlet gridlet;

    String indent = "    ";
    System.out.println();
    System.out.println("========== OUTPUT ==========");
    //System.out.println("Gridlet ID" + indent + "STATUS" + indent
    //        + "Resource ID" + indent + "Cost" + indent + "START TIME"
    //       + indent + "END TIME");

    for (int i = 0; i < size; i++) {
        gridlet = (Gridlet) list.get(i);
        if (gridlet.getSubmissionTime() < time1)
            time1 = gridlet.getSubmissionTime();
        if (gridlet.getFinishTime() > time2)
            time2 = gridlet.getFinishTime();
        averageTurnAroundTime += gridlet.getFinishTime()
                - gridlet.getSubmissionTime();
        //System.out.print(indent + gridlet.getGridletID() + indent + indent);

        //if (gridlet.getGridletStatus() == Gridlet.SUCCESS)
        //    System.out.print("SUCCESS");
```

```
            //System.out.println(indent + indent + gridlet.getResourceID()
            //        + indent + indent + gridlet.getProcessingCost() + indent
            //       + indent + gridlet.getExecStartTime() + indent + indent
            //      + gridlet.getFinishTime());

        }
        System.out.println("time1 = " + time1);
        System.out.println("time2 = " + time2);
        System.out.println("Difference : " + (time2 - time1));
        System.out.println("Simulation_Time = " + Simulation_Time);
        System.out.println("averageTurnAroundTime = " + averageTurnAroundTime /
                num_gridlet);
    }


}

package grid;

import java.io.RandomAccessFile;
import java.io.FileNotFoundException;
import java.io.IOException;

/**
 * Created by IntelliJ IDEA.
 * User: azm537
 * Date: Jul 18, 2008
 * Time: 6:46:41 PM
 * To change this template use File | Settings | File Templates.
 */
public class StatisticalAnalysis {
    public static int Communications = 0;
    public static int Number_resource_failure = 0;
    public static double MAX_MAKESPAN = Double.MIN_VALUE;

    public static double heaviestNode = 0;
    public static double lightestNode = 0;

    private static RandomAccessFile random_Timeshared_log = null;
    private static RandomAccessFile antColony_log = null;
    private static RandomAccessFile random_SpaceShared_log = null;
    private static RandomAccessFile particle_log = null;
    private static RandomAccessFile SBA_log = null;


    public static RandomAccessFile getRandom_TimeShared_log() {
        if (random_Timeshared_log == null) {
            try {
                random_Timeshared_log = new RandomAccessFile(
                        "random_timeshared_log.txt", "rw");
                random_Timeshared_log.setLength(0);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
```

```
            e.printStackTrace();
        }

    }
    return random_Timeshared_log;
}
public static RandomAccessFile getAntColony_log() {
    if (antColony_log == null) {
        try {
            antColony_log = new RandomAccessFile("antColony_log.txt", "rw");
            antColony_log.setLength(0);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
    return antColony_log;
}
public static RandomAccessFile getRandom_SpaceShared_log(){
      if (random_SpaceShared_log == null) {
        try {
            random_SpaceShared_log = new RandomAccessFile(
                    "random_spaceshared.txt", "rw");
            random_SpaceShared_log.setLength(0);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
    return random_SpaceShared_log;

}
public static RandomAccessFile get_particle_log(){
      if (particle_log == null) {
        try {
            particle_log = new RandomAccessFile("particle_log.txt", "rw");
            particle_log.setLength(0);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
    return particle_log;

}
public static RandomAccessFile get_sba_log(){
     if (SBA_log == null) {
        try {
```

```
                SBA_log = new RandomAccessFile("sba_log.txt", "rw");
                SBA_log.setLength(0);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }

        }
        return SBA_log;

    }
}

package grid;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;

public class Topology {

    private static HashMap<Integer, ArrayList> connectedNodes;
    private static HashMap<Integer, Boolean> alive;
    private static Topology topology;
    public static int LINKNO = 0;

    public static Topology getInstance(int num_resource, int[] resourceID)
            throws Exception {
        if (topology == null)
            topology = new Topology(num_resource, resourceID);
        return topology;
    }

    private Topology(int num_resource, int[] resourceID) throws Exception {

        boolean connected[] = new boolean[num_resource];
        alive = new HashMap<Integer, Boolean>();
        connectedNodes = new HashMap<Integer, ArrayList>();
        Random random = new Random();
        int connectedNodeIp, notConnectedNodeIp;


        for (int i = 0; i < resourceID.length; i++)
            alive.put(resourceID[i], true);

        connected[0] = true;

        for (int i = 0; i < num_resource - 1; i++) {
            do {
                connectedNodeIp = random.nextInt(num_resource);
            } while (!connected[connectedNodeIp]);
            do {
                notConnectedNodeIp = random.nextInt(num_resource);
```

```
            } while (connected[notConnectedNodeIp]);

            connected[notConnectedNodeIp] = true;

            ArrayList<Integer> list = connectedNodes.get(
                    resourceID[connectedNodeIp]);

            if (list == null)
                list = new ArrayList<Integer>();
            list.add(resourceID[notConnectedNodeIp]);
            connectedNodes.put(resourceID[connectedNodeIp],
                    list);

            ArrayList<Integer> list2 = connectedNodes.get(
                    resourceID[notConnectedNodeIp]);
            if (list2 == null)
                list2 = new ArrayList<Integer>();
            list2.add(resourceID[connectedNodeIp]);
            connectedNodes.put(resourceID[notConnectedNodeIp],
                    list2);
            LINKNO++;
        }
        int node1, node2;

        for (int i = 0; i < 50; i++) {

            while (true) {
                node1 = random.nextInt(num_resource);
                do {
                    node2 = random.nextInt(num_resource);
                } while (node1 == node2);

                ArrayList arrayList = connectedNodes.get(resourceID[node1]);
                for (int j = 0; j < arrayList.size(); j++)
                    if (((Integer) arrayList.get(j)).intValue() ==
                            resourceID[node2])
                        continue;
                break;
            }


            connectedNodes.get(resourceID[node1]).add(resourceID[node2]);
        //    if(connectedNodes.get(resourceID[node2]) == null)
        //        System.out.println(connectedNodes.get(resourceID[node2]));
            connectedNodes.get(resourceID[node2]).add(resourceID[node1]);
            LINKNO++;
        }

/* Iterator<Integer> iterator = connectedNodes.keySet().iterator();
        while(iterator.hasNext()){
            Integer key = iterator.next();
            ArrayList<Integer> neighbours = connectedNodes.get(key);
            for(int i=0 ;  i < neighbours.size() ; i++)
                System.out.println(neighbours.get(i));
```

```
            System.out.println("-------------");
        }
        System.out.println("Topology created !");
*/
    }

    public static ArrayList getConnectedResources(int ID) {
        ArrayList result = connectedNodes.get(ID);
        for (int i = 0; i < result.size();) {
            if (!isAlive((Integer) result.get(i)))
                result.remove(i);
            else
                i++;

        }
        return result;

    }

    public static int gerRandomNodeId() {
        Random random = new Random();
        Object[] nodes = connectedNodes.keySet().toArray();
        return (Integer) (nodes[random.nextInt(nodes.length)]);

    }

    public static void failNode(int id) {
        alive.put(id, false);
    }

    public static void liveNode(int id) {
        alive.put(id, true);
    }

    private static boolean isAlive(int id) {
        return alive.get(id);
    }
}
```